

Часть I

ОСНОВЫ

Глава 1

Введение

С тех пор, как появилось программирование для веб, люди старались сделать процесс разработки более приятным. Мы постоянно применяли новые техники для борьбы со сложностью — угрозами безопасности, отсутствием текущего состояния при использовании HTTP, необходимостью использовать множество языков программирования (HTML, CSS, Javascript) при создании мощных веб-приложений и тп.

Yesod пытается упростить процесс веб-разработки, играя на сильных сторонах языка программирования Haskell. Строгие проверки на этапе компиляции в языке Haskell затрагивают не только типы, а прозрачность по ссылкам гарантирует, что мы не имеем непреднамеренных побочных эффектов. Сопоставление с образцом алгебраических типов данных позволяет гарантировать, что мы учли все возможные случаи. Программируя на Haskell, мы избавляемся от целых классов ошибок.

К сожалению, одного использования Haskell недостаточно. Всемирная паутина по своей природе не является типобезопасной. У нас даже нет элементарной возможности отличить строку от числа — все данные в вебе передаются в виде простых последовательностей байт, сводя на нет наши усилия относительно безопасности типов. Задача проверки входных данных полностью возложена на разработчика приложения. Я называю это пограничной проблемой — поскольку ваше приложение является безопасным в отношении типов, любая граница с окружающим миром нуждается в «дезинфекции».

1.1. Безопасность типов

Здесь в дело вступает Yesod. Используя высокоуровневые декларативные приёмы, вы можете точно указать ожидаемые типы входных параметров. Да и сама обработка данных работает не так, как обычно — используя типобезопасные URL, вы можете быть уверены в том, что выходные данные также сформированы правильно.

С пограничной проблемой имеет дело не только клиент, она также существует при сохранении и при загрузке данных. И вновь Yesod спасает нас от пограничных проблем, производя сериализацию данных. Вы можете работать с высокоуровневым определением сущностей, оставаясь в блаженном неведении относительно деталей реализации.

1.2. Выразительность

Не секрет, что веб-приложения содержат массу шаблонного кода. Где это возможно, Yesod старается использовать особенности языка Haskell, чтобы уберечь ваши пальцы от лишней работы:

- Библиотека для работы с формами в большинстве случаев уменьшает количество кода путём использования класса типов `Applicative`.
- Объявления маршрутов очень немногословны, но не в убыток безопасности типов.
- Код для сериализации и десериализации ваших данных генерируется автоматически.

В Yesod есть два вида кодогенерации. Для начала нового проекта предоставляется утилита, предназначенная для создания структуры файлов и каталогов. Тем не менее, большая часть кода генерируется на этапе компиляции благодаря механизму метапрограммирования. Таким образом, сгенерированный код никогда не устаревает. Простое обновление библиотеки автоматически обновит весь сгенерированный код.

Но для тех, кто предпочитает сохранять контроль и точно знать, что делает написанный код, предусмотрена возможность оставаться ближе к компилятору и писать весь код самостоятельно.

1.3. Производительность

GHC, наиболее популярный компилятор языка Haskell, обладает потрясающими характеристиками производительности, и они постоянно улучшаются. Один только выбор языка программирования дал Yesod огромное преимущество в плане производительности по сравнению с альтернативами. Но одного языка не достаточно — нам нужна архитектура, ориентированная на производительность.

Для примера рассмотрим подход к работе с шаблонами. Разрешив анализ HTML, CSS и JavaScript кода на этапе компиляции, Yesod не только избежал дорогостоящих операций ввода/вывода с диска, но и получил возможность оптимизировать рендеринг этого кода. Однако на этом архитектурные решения не заканчиваются. Благодаря тому, что лежащие в основе Yesod библиотеки используют такие продвинутые приёмы, как каналы (`conduits`) и строители (`builders`), мы можем быть уверены в том, что наш код потребляет постоянное количество оперативной памяти, не приводя к исчерпанию файловых дескрипторов и других дорогостоящих ресурсов. Используя высокоуровневые абстракции, вы можете получить сильно сжатый и должным образом кэшируемый код на CSS и JavaScript.

Warp, флагманский веб-сервер Yesod, является быстрейшим веб-сервером на Haskell. Совместно используя эти две технологии, мы получаем одно из самых высокопроизводительных решений для создания веб-приложений.

1.4. Модульность

Yesod породил множество пакетов, большинство из которых могут быть использованы отдельно от самого Yesod. Одна из целей проекта состоит в том, чтобы вернуть сообществу как можно больше. Таким образом, даже если вы не собираетесь использовать Yesod в своём следующем проекте, существенная часть этой книги может оказаться для вас полезной.

Разумеется, эти библиотеки были разработаны с расчётом на возможность лёгкой интеграции их друг с другом. Использование фреймворка Yesod должно дать вам ощущение сильной согласованности различных API.

1.5. Прочная кодовая база

Помню, как-то раз мне попался фреймворк на языке PHP, среди преимуществ которого отмечалась поддержка UTF-8. Я был поражён: то есть, вы хотите сказать, что поддержка UTF-8 не является автоматической? В мире Haskell такие вещи, как возможность работы с различными кодировками, полностью поддерживаются уже давно. На самом деле, мы обычно сталкиваемся с полностью противоположной проблемой, когда существует более одного пакета, предоставляющего мощное и хорошо спроектированное решение проблемы. Сообщество Haskell-программистов находится в непрерывном поиске более чистого и более эффективного решения для каждой проблемы.

Обратная сторона медали заключается в сложности выбора. Используя Yesod, вы получаете готовый набор инструментов, выбранный за вас, и вы можете быть уверены в том, что эти инструменты будут работать совместно. Разумеется, за вами всегда остаётся возможность выбрать решение на свой вкус.

Например, Yesod и Hamlet (язык для создания шаблонов, используемый по умолчанию) используют библиотеку blaze-builder для генерации текстового контента. Такой выбор был сделан по той причине, что blaze предоставляет наиболее быстрый интерфейс для генерации данных в кодировке UTF-8. Любой, кто желает использовать одну из других замечательных библиотек, например, text, без труда сможет сделать это.

1.6. Введение в Haskell

Haskell является мощным, производительным, типобезопасным функциональным языком программирования. Эта книга написана в предположении, что вы уже знакомы с основами языка Haskell. Существует две прекрасные книги для изучения Haskell, при этом они обе доступны для чтения онлайн¹:

- Learn You a Haskell for Great Good!²
- Real World Haskell³

¹В качестве русскоязычного аналога можно порекомендовать онлайн-книгу Антона Холoméва «Учебник по Haskell», ознакомиться с которой можно по адресу <http://goo.gl/xIggc>. Обратите также внимание, что книга «Learn You a Haskell for Great Good!» недавно была выпущена в печатном виде на русском языке.

²<http://leamyouahaskell.com/>

³<http://book.realworldhaskell.org/>

Yesod опирается на некоторые возможности языка Haskell, которые не освещены в учебниках для начинающих. Хотя от вас не часто будет требоваться понимание того, как именно они работают, всегда лучше понимать, что делают используемые вами инструменты. Упомянутые возможности рассматриваются в следующей главе.

Глава 2

Haskell

Для использования Yesod вам необходимо знать по крайней мере основы языка Haskell. Кроме того, в Yesod используются некоторые особенности Haskell, которые не описаны в большинстве руководств начального уровня. Хотя от читателя предполагается базовое владение Haskell, эта глава призвана заполнить возможные пробелы.

Если вы уже хорошо владеете Haskell, можете пропустить эту главу. Также, если вы предпочитаете начать непосредственно с погружения в Yesod, то вы всегда сможете потом к этой главе вернуться.

Если вам необходимо более полное введение в Haskell, я бы порекомендовал книги «Real World Haskell» или «Learn You a Haskell».

Возможно, стоит вставить ссылки на книги (в оригинале ссылок нет).

2.1. Терминология

Даже у людей, хорошо знакомых с языком Haskell, порой может возникать путаница с терминологией. Сформулируем некоторые основные термины, которые мы будем использовать на протяжении всей книги.

2.1.1. Тип данных

Это один из основных строительных блоков языков со строгой типизацией, к которым относится и Haskell. Некоторые типы данных, такие как **Int**, можно считать элементарными, а остальные типы строятся на их основе для создания более сложных значений. Например, вы могли бы представить человека следующим типом:

```
data Person = Person Text Int
```

Здесь Text содержит имя человека, а **Int** — его возраст. Благодаря простоте этого примера мы будем обращаться к нему на протяжении всей книги.

Существует три способа, с помощью которых вы можете создать новый тип данных:

- Определение **type**, такое как `type GearCount = Int`, просто создаёт синоним существующего типа. Система типов не станет препятствовать вам использовать **Int** везде, где требуется `GearCount`. Использование такого типа может сделать ваш код более самодокументируемым.
- Определение **newtype**, например, `newtype Make = Make Text`. В этом случае вы не можете случайно использовать `Text` вместо `Make`, компилятор не даст вам этого сделать. Обёртка, создаваемая посредством **newtype**, всегда удаляется во время компиляции и не создаёт накладных расходов.
- Определение **data**, как в приведённом выше примере с `Person`. Вы можете также создавать алгебраические типы данных (АТД), такие как `data Vehicle = V Bicycle GearCount | Car Make Model`.

2.1.2. Конструктор данных

В приведённых выше примерах `Person`, `Make`, `Bicycle` и `Car` — конструкторы данных.

2.1.3. Конструктор типа

В свою очередь `Person`, `Make` и `Vehicle` — это конструкторы типов.

2.1.4. Переменные типов

Рассмотрим тип данных `data Maybe a = Just a | Nothing`. Здесь `a` — переменная типа.

2.2. Инструментарий

Для разработки на Haskell вам понадобятся два основных инструмента. Glasgow Haskell Compiler (GHC) — стандартный компилятор Haskell, единственный официально поддерживаемый Yesod. Также вам понадобится Cabal — стандартное средство сборки для Haskell. Cabal используется не только для сборки локального кода, он также может автоматически скачивать и устанавливать зависимости из Hackage, репозитория пакетов Haskell.

Тем, кто работает в среде Windows или Mac, настоятельно рекомендуется скачать Haskell Platform¹. Многие дистрибутивы Linux содержат Haskell Platform в своих репозиториях. Например, в дистрибутивах, основанных на Debian, стоит начать с выполнения команды `apt-get install haskell-platform`. Если в вашем дистрибутиве Haskell Platform недоступна, вы можете установить её вручную, следуя инструкциям на официальном сайте.

Ещё один важный инструмент, который вам нужно будет обновить — это alex. Haskell Platform включает версию 2, но используемой Yesod программе минификации

¹<http://hackage.haskell.org/platform/>

JavaScript, `hjsmin`², требуется версия 3. Не забудьте выполнить `cabal install alex` после установки Haskell Platform, иначе получите сообщения об ошибках в отношении пакета `language-javascript`³.

Некоторым нравится жить на острие прогресса и устанавливать последнюю версию GHC до того, как она станет доступна в Haskell Platform. Мы стараемся поддерживать совместимость Yesod со всеми современными версиями GHC, но официально мы поддерживаем только Haskell Platform. Если вы всё-таки решили пойти по пути ручной установки GHC, то имейте в виду:

- Вам нужно будет установить некоторые дополнительные инструменты, в частности, `alex` и `happy`.
- Убедитесь, что установлены все требуемые C-библиотеки^a. В дистрибутивах, основанных на Debian, это можно сделать, выполнив команду:
`apt-get install libedit-dev libbsd-dev libgmp3-dev zlib1g-dev freeglut3-dev.`

^a<http://www.vex.net/~trebla/haskell/haskell-platform.xhtml>

Независимо от способа установки инструментов вам нужно добавить путь к каталогу `bin Cabal` в переменную окружения `PATH`. В Mac и Linux это будет `$HOME/.cabal/bin`, а в Windows — `%APPDATA%\cabal\bin`.

У программы `cabal` есть множество доступных опций, но для начала ознакомьтесь с двумя командами:

- `cabal update` скачивает актуальный список пакетов с Hackage.
- `cabal install yesod-platform` устанавливает Yesod и его зависимости.

Многие члены сообщества предпочитают выполнять изолированную сборку пакетов Haskell, что предотвращает порчу существующих пакетов вследствие установки Yesod или порчу Yesod из-за установки каких-то пакетов в будущем. В этой книге я не буду углубляться в детали того, как это делается, но два наиболее часто используемых для этих целей инструмента — это `cabal-dev`^a и `hsenv`^b.

^a<http://hackage.haskell.org/package/cabal-dev>

^b<http://hackage.haskell.org/package/hsenv>

2.3. Указания компилятору

По умолчанию GHC работает в режиме, очень близком к Haskell98. Однако с ним также поставляется большое число языковых расширений, предоставляющих более мощные классы типов, изменения в синтаксисе и прочее. Есть несколько способов заставить GHC включить эти расширения. В большинстве фрагментов кода, приведённых в этой книге, вы увидите указания компилятору, которые выглядят следующим образом:

²<http://hackage.haskell.org/package/hjsmin>

³<http://hackage.haskell.org/package/language-javascript>


```
{-# LANGUAGE MyLanguageExtension #-}
```

Эти указания должны всегда находиться в самом начале исходного файла. Помимо этого, есть ещё два распространённых способа:

- при вызове GHC передать ему аргумент `-XMyLanguageExtension`;
- добавить блок `extensions` в файл `cabal`.

Лично я никогда не использую подход с аргументами командной строки. Это личное предпочтение, но мне нравится, когда мои настройки явным образом указаны в файле. В общем случае также рекомендуется избегать помещения расширений в файл `cabal`. Впрочем, в сгенерированном Yesod шаблоне сайта мы намеренно применяем этот подход, чтобы избежать необходимости задавать одни и те же указания компилятору в каждом исходном файле.

В конечном счёте мы будем использовать в этой книге немало языковых расширений (начальный каркас сайта использует 11). Мы не будем рассматривать назначение каждого из них. За этим, пожалуйста, обращайтесь к документации по GHC⁴.

Нужно добавить ссылку на главу «Scaffolding and the Site Template»

2.4. Перегруженные строки

Какой тип имеет литерал `"hello"`? Обычно это `String`, определённый как `type String = [Char]`. К сожалению, у этого подхода есть ряд ограничений:

- Это очень неэффективный способ хранения текстовых данных. Мы должны выделить память под каждую конс-ячейку, плюс каждый символ занимает целое машинное слово.
- Иногда у нас есть данные, напоминающие строку, но не являющиеся текстом, такие как `ByteString` и `HTML`.

Для преодоления этих ограничений в GHC есть языковое расширение, называемое `OverloadedStrings`. Когда оно включено, строковые литералы будут иметь не мономорфный тип `String`, а тип `IsString a => a`, где `IsString` определён следующим образом:

```
class IsString a where
  fromString :: String -> a
```

Существуют экземпляры `IsString` для ряда типов Haskell, таких как `Text` (намного более эффективный тип для упакованного хранения строк), `ByteString` и `HTML`. Практически каждый пример в этой книге предполагает, что это языковое расширение включено.

К сожалению, у этого расширения есть один недостаток: иногда оно может сбить с толку проверку типов GHC. Представьте, что у нас есть следующий код:

⁴http://www.haskell.org/ghc/docs/latest/html/users_guide/ghc-language-features.html

```

{-# LANGUAGE OverloadedStrings, TypeSynonymInstances,
FlexibleInstances #-}

import Data.Text (Text)

class DoSomething a where
  something :: a -> IO ()

instance DoSomething String where
  something _ = putStrLn "String"

instance DoSomething Text where
  something _ = putStrLn "Text"

myFunc :: IO ()
myFunc = something "hello"

```

Что выведет эта программа: «String» или «Text»? Неясно. Поэтому вам придётся предоставить явную аннотацию типа, чтобы указать, должен ли литерал "hello" трактоваться как **String** или как **Text**.

2.5. Семейства типов

Основная идея семейства типов — устанавливать некоторую ассоциацию между двумя различными типами. Предположим, мы хотим написать функцию, которая безопасным образом получает первый элемент списка. Но мы не хотим, чтобы она работала только на списках; нам бы хотелось, чтобы она трактовала `ByteString` как список, состоящий из элементов типа `Word8`. Чтобы этого добиться, нам нужно ввести некоторый ассоциированный тип, чтобы указать, каково содержимое определённого типа.

```

{-# LANGUAGE TypeFamilies, OverloadedStrings #-}

import Data.Word (Word8)
import qualified Data.ByteString as S
import Data.ByteString.Char8 () -- получить обособленно определённый
экземпляр IsString

class SafeHead a where
  type Content a
  safeHead :: a -> Maybe (Content a)

instance SafeHead [a] where
  type Content [a] = a
  safeHead [] = Nothing
  safeHead (x:_) = Just x

```

```
instance SafeHead S.ByteString where
  type Content S.ByteString = Word8
  safeHead bs
    | S.null bs = Nothing
    | otherwise = Just $ S.head bs

main :: IO ()
main = do
  print $ safeHead (" " :: String)
  print $ safeHead ("hello" :: String)

  print $ safeHead (" " :: S.ByteString)
  print $ safeHead ("hello" :: S.ByteString)
```

02/type-families.hs

Новый синтаксис выражается в возможности разместить тип внутри класса и экземпляра. Вместо этого мы также можем использовать **data**, что создаст новый тип данных вместо ссылки на уже существующий.

Есть и другие способы использования ассоциированных типов вне контекста класса типов. Однако в Yesod все наши ассоциированные типы являются, фактически, частью класса типов. Более подробную информацию о семействах типов можно найти на соответствующей странице Haskell Wiki^a.

^ahttp://www.haskell.org/haskellwiki/GHC/Type_families

2.6. Template Haskell

Template Haskell (ТН) — это способ *генерации кода*. Мы используем его в ряде мест в Yesod для уменьшения объёма вспомогательного кода и для того, чтобы быть уверенными в корректности сгенерированного кода. Код на Template Haskell — это, в сущности, код на Haskell, который генерирует абстрактное синтаксическое дерево (АСД) кода на Haskell.

На самом деле, возможности ТН этим не ограничиваются, так как он позволяет фактически выполнять интроспекцию кода. Однако эти возможности мы в Yesod не используем.

Написание кода на ТН может быть непростым делом, и, к сожалению, безопасность типов при этом помогает мало. Несложно написать на ТН программу, генерирующую код, который не скомпилируется. Это является проблемой только для разработчиков Yesod, но не для его пользователей. В ходе разработки мы используем большой набор модульных тестов, чтобы убедиться в корректности сгенерированного кода. Как пользователю, всё, что вам нужно делать — вызывать уже существующие функции. Например, чтобы включить внешний шаблон Hamlet, вы можете написать:

```
$(hamletFile "myfile.hamlet")
```

Вставить ссылку на главу «Shakespearean Templates»

(Hamlet обсуждается в главе о Shakespeare.) Символ доллара, за которым следует открывающая скобка, сообщает GHC, что то, что следует далее, является функцией Template Haskell. Код внутри скобок затем запускается компилятором и генерирует АСД программы на Haskell, которое затем компилируется. И да, метапрограммирование можно использовать даже здесь⁵.

Приятная особенность — код на TH может выполнять произвольные действия ввода-вывода, и, следовательно, мы можем разместить входные данные во внешних файлах, которые будут разбираться во время компиляции. Одним из примеров использования этой возможности является получение шаблонов HTML, CSS и JavaScript, проверяемых во время компиляции.

Если код на Template Haskell используется для генерации объявлений и располагается на верхнем уровне наших файлов, то мы можем избавиться от знаков доллара и скобок. Иными словами,

```
{-# LANGUAGE TemplateHaskell #-}

-- Обычное определение функции, ничего особенного
myFunction = ...

-- Включить код на TH
$(myThCode)

-- Или, то же самое
myThCode
```

Может быть полезно посмотреть, какой код генерирует для вас Template Haskell. Для этого вам нужно использовать опцию GHC `-ddump-splices`.

Есть ещё много других особенностей Template Haskell, которые здесь не рассматриваются. Более подробную информацию можете посмотреть на странице Haskell Wiki^a.

^ahttp://www.haskell.org/haskellwiki/Template_Haskell

2.6.1. QuasiQuotes

QuasiQuotes (QQ) — это небольшое расширение Template Haskell, которое позволяет нам включать произвольные данные в файлы с исходным кодом на Haskell. Например, выше упоминалась функция Template Haskell `hamletFile`, которая считывает содержимое шаблона из внешнего файла. У нас также есть обработчик квазичитирования, называемый `hamlet`, который считывает содержимое, хранящееся непосредственно в исходном коде:

```
{-# LANGUAGE QuasiQuotes #-}

[hamlet|<p>Это квазичитированный Hamlet.|]
```

⁵<http://www.yesodweb.com/blog/2010/09/yo-dawg-template-haskell>

Синтаксис выделяется квадратными скобками и символами вертикальной черты. Имя обработчика квазичитирования размещается между открывающей скобкой и первой вертикальной чертой, а содержимое приводится между двумя символами вертикальной черты.

На протяжении книги мы во многих случаях будем предпочитать внешним файлам QQ-подход, поскольку такой код проще скопировать из книги. Однако в реальных приложениях внешние файлы являются предпочтительными во всех случаях, кроме совсем коротких данных, так как это даёт нам хорошее разделение кода на Haskell и кода с иным синтаксисом.

2.7. Выводы

Вам не нужно быть экспертом в Haskell, чтобы использовать Yesod, достаточно базового знакомства. Надеюсь, эта глава дала вам достаточно информации, чтобы чувствовать себя более комфортно при изучении оставшейся части книги.

Глава 3

ОСНОВЫ

Первый шаг при изучении любой новой технологии — сделать так, чтобы она заработала. Цель данной главы — познакомить вас с простым приложением Yesod и охватить некоторые основные концепции и терминологию.

3.1. Здравствуй, Мир

Давайте начнём эту книгу, как полагается: сделаем простую веб-страницу, на которой выводится «Здравствуй, Мир!»:

```
{-# LANGUAGE TypeFamilies, QuasiQuotes, MultiParamTypeClasses,
    TemplateHaskell, OverloadedStrings #-}
import Yesod

data HelloWorld = HelloWorld

mkYesod "HelloWorld" [parseRoutes|
/ HomeR GET
|]

instance Yesod HelloWorld

getHomeR :: Handler RepHtml
getHomeR = defaultLayout [whamlet|Здравствуй, Мир!|]

main :: IO ()
main = warpDebug 3000 HelloWorld
```

03/hello-world.hs

Если вы сохраните этот код в файл `helloworld.hs` и запустите его командой `runhaskell helloworld.hs`, то получите веб-сервер, запущенный на порту 3000. Ес-

ли вы откроете браузером страницу `http://localhost:3000`, то получите следующий HTML-код:

```
<!DOCTYPE html>
<html><head><title></title></head><body>Здравствуй, Мир!</body></html>
```

Далее на протяжении главы мы ещё будем возвращаться к этому примеру.

3.2. Маршрутизация

Как и большинство современных веб-фреймворков, Yesod следует шаблону «единая точка входа». Это означает, что каждый запрос к приложению Yesod поступает через общую точку и оттуда уже маршрутизируется. В отличие от этого подхода, в системах наподобие PHP и ASP вы обычно создаёте множество различных файлов, и веб-сервер автоматически направляет запросы к соответствующему файлу.

Кроме того, Yesod использует декларативный стиль задания маршрутов. В приведённом выше примере это выглядело так:

```
mkYesod "HelloWorld" [parseRoutes]
/ HomeR GET
[]
```

`mkYesod` — это функция Template Haskell, а `parseRoutes` — обработчик квазицитирования.

По-русски это означает: создать в приложении `HelloWorld` один путь по имени `HomeR`. Он должен слушать запросы к `/` (корню приложения) и отвечать на GET-запросы. `HomeR` мы называем ресурсом, отсюда и суффикс «R» в названии.

Суффикс «R» в именах ресурсов — это просто соглашение, но его придерживаются практически повсеместно. Оно позволяет немного упростить чтение и понимание кода.

Функция TH `mkYesod` генерирует довольно много кода: тип данных маршрута, функцию диспетчеризации и функцию рендеринга. Мы рассмотрим их более подробно в главе «Маршрутизация URL и обработчики». Но используя опцию `GHC -ddump-splices`, мы можем посмотреть непосредственно на сгенерированный код. Вот сильно подчищенная его версия:

```
instance RenderRoute HelloWorld where
  data Route HelloWorld = HomeR
  deriving (Show, Eq, Read)
  renderRoute HomeR = ([], [])

instance YesodDispatch HelloWorld HelloWorld where
  yesodDispatch logger master sub toMaster app404 app405 method []
  pieces =
    case dispatch pieces of
      Just f -> f
```

подчищенная звучит глупо, но более хорошего немногословного варианта не вижу

```

        logger
        master
        sub
        toMaster
        app404
        app405
        method
    Nothing -> app404
where
    dispatch = Yesod.Routes.Dispatch.toDispatch
              [ Yesod.Routes.Dispatch.Route [] False onHome
              ]
    onHome [] = Just $ \logger master sub toMaster _app404 []
app405 method ->
    case method of
        "GET" -> yesodRunner
            logger
            (fmap chooseRep getHomeR)
            master
            sub
            (Just HomeR)
            toMaster
        _ -> app405 HomeR

```

Что-то из этого вам, вероятно, пока непонятно. В частности, реализация `yesodDispatch` выглядит страшновато из-за того, что она должна быть приспособлена к разным способам диспетчеризации и соответствовать модели, необходимой для наших структур высокопроизводительной маршрутизации. Тем не менее, реализация `RenderRoute` вместе с ассоциированным типом данных должна дать вам хорошее представление о том, что происходит внутри.

3.3. Функция-обработчик

Итак, у нас есть маршрут по имени `HomeR`, и он отвечает на GET-запросы. Как вы определяете наш ответ? Вы пишете функцию-обработчик. `Yesod` следует стандартной схеме именования таких функций: имя метода в нижнем регистре (т. е., GET становится `get`), после которого идёт имя маршрута. В нашем случае эта функция будет называться `getHomeR`.

Большая часть вашего кода в `Yesod` будет находиться в функциях-обработчиках. Именно здесь вы обрабатываете пользовательский ввод, выполняете запросы к базе данных и создаёте ответы. В нашем простом примере мы создаём ответ с помощью функции `defaultLayout`. Эта функция оборачивает переданное ей содержимое в шаблон вашего сайта. По умолчанию она создаёт файл HTML с `doctype`, тегами `html`, `head` и `body`. Как мы увидим в главе «Класс типов `Yesod`», эта функция может быть переопределена, чтобы делать гораздо больше.

В нашем примере в `defaultLayout` мы передаём `[whamlet|Hello world!]`. `whamlet` — это ещё один обработчик квазицитирования. В данном случае он преобразует синтаксис `Hamlet` в `Widget`. `Hamlet` — это движок HTML-шаблонов, используемый в `Yesod` по умолчанию. Вместе со своими «родственниками» `Cassius`, `Lucius` и `Julius` он позволяет создавать HTML, CSS и JavaScript типобезопасным образом с проверкой во время компиляции. Гораздо больше информации по этому поводу мы увидим в главе о `Shakespeare`.

Виджеты — ещё один краеугольный камень `Yesod`. Они позволяют вам создавать модульные компоненты сайта, состоящие из HTML, CSS и JavaScript, и повторно использовать их в любом месте вашего сайта. Более детально мы будем их рассматривать в главе «Виджеты».

3.4. Тип-основание

Слово «`HelloWord`» появляется в нашем примере несколько раз. Каждое приложение `Yesod` имеет тип-основание. Этот тип должен быть экземпляром класса типов `Yesod`, который предоставляет общее место для определения различных настроек, определяющих выполнение нашего приложения.

В нашем случае этот тип данных довольно скучный: он не содержит никакой информации. Тем не менее, тип-основание имеет существенное влияние на то, как выполняется наш пример: он связывает воедино маршруты с определением экземпляра и даёт им всем возможность выполняться. На протяжении книги мы увидим, как тип-основание всплывает в разных местах.

Но типы-основания не обязательно должны быть скучными: они могут использоваться для хранения множества полезной информации, обычно той, которая должна быть инициализирована при запуске и использоваться повсюду. Вот некоторые наиболее распространённые примеры:

- Пул соединений с базой данных;
- Настройки, загружаемые из конфигурационного файла;
- Менеджер соединений HTTP.

Кстати, слово `yesod` (ייסוד) означает «основание» на иврите.

3.5. Запуск

Ещё раз мы упоминаем `HelloWorld` в нашей основной функции. Тип-основание содержит всю необходимую информацию для маршрутизации и ответов на запросы в нашем приложении; теперь нам нужно просто преобразовать его в нечто, что можно запустить. Для этого в `Yesod` есть полезная функция `war rDebug`, которая запускает веб-сервер `Warp` с отладочным выводом в указанный порт (в нашем случае это 3000).

Одной из особенностей `Yesod` является то, что вы не привязаны к единственной стратегии развёртывания. `Yesod` построен поверх `Web Application Interface (WAI)`, позволяющего запускать приложение через `FastCGI`, `SCGI`, `Warp` или даже как настольное

приложение, используя библиотеку WebKit. Мы рассмотрим некоторые из этих возможностей в главе о развёртывании. И в конце данной главы мы рассмотрим сервер разработки.

Warp является для Yesod основным способом развёртывания. Это легковесный, высокоэффективный веб-сервер, разработанный специально для хостинга приложений Yesod. Он используется и вне Yesod для других разработок на Haskell (как для приложений, основанных на фреймворках, так и для не зависящих от них), а также как стандартный файл-сервер в различных боевых окружениях.

3.6. Ресурсы и типобезопасные URL

В нашем «здравствуй-мире» мы определили только один ресурс (homeR). Веб-приложение обычно намного более интересно, когда в нём больше одной страницы. Давайте посмотрим:

```
{-# LANGUAGE TypeFamilies, QuasiQuotes, MultiParamTypeClasses,
      TemplateHaskell, OverloadedStrings #-}
import Yesod

data Links = Links

mkYesod "Links" [parseRoutes|
 / homeR GET
 /page1 Page1R GET
 /page2 Page2R GET
 |]

instance Yesod Links

getHomeR = defaultLayout [whamlet|<a href=@{Page1R}>Перейти на
  страницу 1!|]
getPage1R = defaultLayout [whamlet|<a href=@{Page2R}>Перейти на
  страницу 2!|]
getPage2R = defaultLayout [whamlet|<a href=@{HomeR}>Вернуться к
  началу!|]

main = warpDebug 3000 Links
```

03/routes.hs

В целом это очень похоже на «Здравствуй, Мир». Тип-основание теперь Links вместо HelloWorld, и помимо ресурса homeR мы добавили Page1R и Page2R. В связи с этим мы также добавили ещё две функции-обработчика: getPage1R и getPage2R.

Единственная действительно новая особенность находится внутри квазичитирования whamlet. Мы углубимся в изучение этого синтаксиса в главе о Shakespeare, но мы можем увидеть, что

```
<a href=@{Page1R}>Перейти на страницу 1!
```

создаёт ссылку на ресурс Page1R. Здесь важно отметить, что Page1R — это конструктор данных. Делая каждый ресурс конструктором данных, мы получаем возможность, называемую типобезопасными URL. Вместо соединения строк для получения URL мы создаём старое доброе значение Haskell. Используя @-интерполяцию (@{...}), Yesod автоматически отображает эти значения в текстовые ссылки перед отправкой их пользователю. Мы можем увидеть, как это реализовано, ещё раз посмотрев на вывод `--dump-splices`:

```
instance RenderRoute Links where
  data Route Links = HomeR | Page1R | Page2R
  deriving (Show, Eq, Read)

  renderRoute HomeR = ([], [])
  renderRoute Page1R = (["page1"], [])
  renderRoute Page2R = (["page2"], [])
```

В ассоциированном с Links типе Route у нас есть дополнительные конструкторы для Page1R и Page2R. У нас также есть более хорошее представление о возвращаемых значениях returnRoute. Первый элемент кортежа содержит компоненты пути для данного маршрута. Второй элемент содержит параметры строки запроса; практически во всех вариантах использования это будет пустой список.

Трудно переоценить значение типобезопасных URL. Они позволяют вам достичь большой гибкости и надёжности при разработке приложения. Мы можете произвольным образом менять URL ресурсов, ни разу не повредив ссылки на них. В главе «Маршрутизация URL и обработчики» мы увидим, что маршруты могут принимать параметры, как, например, URL записи в блоге, принимающий идентификатор поста.

Предположим, вы хотите перейти от маршрутизации по числовому идентификатору поста к маршрутизации по шаблону «год/месяц/название». В традиционном веб-фреймворке вам придётся исправить каждую ссылку на пост в блоге, и если вы пропустите хотя бы одну, то будете получать ошибки 404 во время выполнения. В Yesod всё, что вам придётся сделать, — это обновить маршрут и перекомпилировать приложение: GHC сам найдёт все ссылки, которые должны быть откорректированы.

Есть подозрение, что вместо returnRoute должно быть renderRoute

3.7. Каркас сайта

При установке Yesod вы получаете как библиотеку Yesod, так и исполняемый файл yesod. Этот исполняемый файл принимает несколько команд, но первая команда, с которой вы захотите познакомиться, — это `yesod init`. Она задаст вам несколько вопросов, а затем сгенерирует каталог, содержащий каркас сайта по умолчанию. В этом каталоге вы можете выполнить `cabal install --only-dependencies`, чтобы установить дополнительные зависимости (такие как бэкенды баз данных), и затем `yesod devel`, чтобы запустить сайт.

Каркас сайта даёт вам «из коробки» множество установившихся практик, располагая файлы и настраивая зависимости проверенным временем образом, применяемым в большинстве боевых сайтов на Yesod. Однако все эти удобства могут встать на пути изучения Yesod. Поэтому на протяжении большей части этой книги инструмент построения каркаса сайта не используется, и работа идёт напрямую с Yesod как с библиотекой.

Мы рассмотрим структуру каркаса сайта во всех подробностях позже.

3.8. Сервер разработки

Одним из преимуществ интерпретируемых языков перед компилируемыми является быстрое прототипирование: вы просто сохраняете изменения в файл и обновляете страницу в браузере. Если мы хотим внести изменения в приведённые выше приложения Yesod, нам придётся вызывать `runhaskell`, что может быть немного утомительным.

К счастью, существует решение этой проблемы: `yesod devel` автоматически пересобирает код и перезагружает ваш код. Это может быть отличным способом разработки ваших Yesod-проектов, и когда вы будете готовы перейти на боевой сервер, вы всё равно получите в результате компиляции невероятно эффективный код. Построение каркаса Yesod настраивает всё для вас автоматически. Это позволяет использовать лучшее из двух миров: быстрое прототипирование и быстрый боевой код.

Подготовка вашего кода для использования с `yesod devel` немного более сложна, поэтому наши примеры будут просто использовать `warptest`. Но когда вы будете готовы создавать реальные приложения, `yesod devel` будет ждать вас.

3.9. Выводы

Каждое Yesod-приложение строится вокруг типа-основания. Мы ассоциируем некоторые ресурсы с этим типом данных и определяем несколько функций-обработчиков, а Yesod занимается всей маршрутизацией. Эти ресурсы также являются конструкторами данных, которые позволяют нам использовать типобезопасные URL.

В силу того, что приложения Yesod построены поверх WAI, они могут работать с различными бэкэндами. `warptest` — это простой путь начать работу, так как он поставляется с Yesod. Для быстрой разработки выбором является `yesod devel`. А когда вы будете готовы перейти к использованию в боевом окружении, у вас будет высокопроизводительный сервер Warp.

При разработке приложений для Yesod у вас будет выбор из множества стилей кодирования: квазичитирование или внешние файлы, `warptest` или `yesod devel`, и так далее. Примеры в этой книге используют стили, наиболее облегчающие копирование кода. Но когда вы начнёте создавать реальные приложения с помощью Yesod, вам будут доступны более мощные средства.

Глава 4

Шекспировские шаблоны

Yesod использует Шекспировское семейство шаблонных языков как стандартный подход к созданию HTML, CSS и JavaScript. Это семейство языков имеет похожий синтаксис и общие принципы:

- Как можно меньшее вмешательство в языки, на базе которых создаются Шекспировские шаблоны, но в то же время использование преимуществ этих языков.
- Гарантии корректности контента обеспечиваются компилятором.
- Безопасность статической типизации, которая также предотвращает XSS (cross-site scripting) атаки.
- Автоматическая проверка валидности URL-ов, где это возможно, с помощью типобезопасных URL-ов.

По сути, ничего не связывает Yesod с этими языками. Или, другими словами, и языки, и Yesod можно использовать по-отдельности. Данная глава будет рассматривать эти шаблонные языки сами по себе, в то время как оставшаяся часть книги будет их использовать для разработки приложений для Yesod.

4.1. Краткий обзор

Всего в игре 4 основных языка: Hamlet - это шаблонный язык HTML, Julius - для JavaScript, Cassius и Lucius - оба для CSS. Hamlet и Cassius - два языка, чувствительные к форматированию, использующие отступы для обозначения вложенных блоков. Lucius же, являясь расширением CSS, использует фигурные скобки для обозначения вложенных блоков. Julius - это простой однопроходный язык, который служит для генерирования JavaScript; единственная добавленная функциональность - это интерполяция переменных.

4.1.1. Hamlet (HTML)

```
$doctype 5
<html>
  <head>
    <title>#{pageTitle} - My Site
    <link rel=stylesheet href=@{Stylesheet}>
  <body>
    <h1 .page-title>#{pageTitle}
    <p>Вот список ваших друзей:
    $if null friends
      <p>Прошу прощения, я соврал, у вас нет друзей.
    $else
      <ul>
        $forall Friend name age <- friends
          <li>#{name} (#{age} лет)
      </ul>
    <footer>^{\copyright}
```

4.1.2. Cassius (CSS)

```
#myid
  color: #{red}
  font-size: #{bodyFontSize}
foo bar baz
  background-image: url(@{MyBackgroundR})
```

4.1.3. Lucius (CSS)

```
section.blog {
  padding: 1em;
  border: 1px solid #000;
  h1 {
    color: #{headingColor};
  }
}
```

4.1.4. Julius (Javascript)

```
$(function(){
  $("section.#{sectionClass}").hide();
});
```

```
$("#mybutton").click(function(){document.location =
"@{SomeRouter}";});
^ {addBling}
});
```

4.2. Types

Прежде чем мы перейдем к синтаксису давайте взглянем на различные используемые типы данных. Мы уже обсуждали 1 во вступлении, что типы помогают нам защищаться от XSS атак. К примеру, скажем, у нас есть HTML шаблон, который должен отображать чье-то имя. Он может выглядеть как-то так:

```
<p>Привет, меня зовут #{name}</p>
```

`#{ . . . }` - это способ интерполяции переменных в Шекспировских шаблонах.

Что должно произойти с именем, и какого оно должно быть типа данных? Наивное решение - это использовать `Text` переменную, и вставлять явно её значение в код. Но это вызовет проблемы для случая, когда `name = "<script src='http://nefarious.com/evil.js'></script>"`. Что хотелось бы получить в данном случае - это имя с закодированными символами, так что `<` становится `<`;

Таким же наивным решением было бы просто кодировать каждый кусок вставляемого текста. Что получится, если у вас есть заранее сгенерированный другим процессом HTML? Например, на сайте Yesod, все Haskell-куски кода пропущены через раскрашивающую функцию, которая оборачивает каждое слово в соответствующий `span`-тэг. Если мы закодируем все символы такого текста, все куски раскрашенного кода станут просто нечитабельными.

Вместо этого мы имеем специальный тип данных - `Html`. Для того, чтобы сгенерировать `Html` значение, у нас есть две возможности в API: `toHtml` класс типов предоставляет способ конвертировать `String` и `Text` значения в `Html` через `toHtml` функцию, которая автоматически конвертирует все символы. Этот подход подходит для решения проблемы с `name`, описанный выше. Для примеров с кусками кода мы бы использовали `preEscaped` семейство функций.

Когда вы используете интерполяцию переменной в `Hamlet` (HTML Шекспировский язык), она автоматически применяет `toHtml` функцию на значение этой переменной. Т.е. если вы интерполируете `String`, все ее символы будут закодированы. Но если вы возьмете переменную типа `Html`, ее значение будет появляться неизменённым. В примере с кусками кода, мы могли бы интерполировать как-нибудь так: `#{preEscapedText myHaskellHtml}`.

Тип данных `Html`, как и все описанные выше функции, объявлены в пакете `blaze-html`. Это позволяет `Hamlet` взаимодействовать со всеми остальными `blaze-html` пакетами, а также предоставляет более общее решение для создания `blaze-html` значений. Еще одним преимуществом пакета `blaze-html` является его высокая производительность.

Аналогично, у нас есть `Css/ToCss` и `Javascript/ToJavascript`. Это дает возможность во время компиляции проводить некоторую проверку работоспособности программы: что мы случайно не добавили какой-то HTML код в CSS.

Еще одно достоинство, уже со стороны CSS, - это вспомогательные типы данных для цветов и единиц измерений. Например:

```
.red { color: #{colorRed} }
```

Подробности ищите в документации по Haddock.

4.3. Типобезопасные URL

Наверное, самая уникальная особенность Yesod - это типобезопасные URL, и возможность удобного их использования предоставлена напрямую системой Шекспировских шаблонов. Использование типобезопасных URL почти идентично интерполяции переменных, мы просто используем символ "собаки" (@) вместо "решетки" (#). Мы чуть позже остановимся подробнее на синтаксисе, сперва, давайте создадим общее понимание концепции.

Предположим, у нас есть приложение с двумя маршрутами `http://example.com/profile/home` - это домашняя страница, и `http://example.com/display/time` - показывает текущее время. И, скажем, мы хотим сделать ссылку с домашней страницы на страницу со временем. Я вижу три различных способа конструирования URL:

1. Как относительную ссылку: `../display/time`
2. Как абсолютную ссылку без домена: `/display/time`
3. Как абсолютную ссылку с доменом: `http://example.com/display/time`

Каждый из этих способов имеет недостатки: первый перестанет работать, если любой из URL изменится. К тому же, он подходит не для всех случаев: RSS и Atom, к примеру, требуют абсолютный URL. Второй способ более защищен от изменений URL, чем первый, но все еще неприменим для RSS и Atom. И, хотя, третий способ хорошо работает для всех случаев, вам придется каждый раз обновлять все URL, когда поменяется ваш домен. Вы думаете, это происходит не так часто? Просто подождите, пока вы переместитесь с сервера разработки на пробный сервер, а потом и на "боевой" сервер.

Но что более важно, есть ещё одна очень большая проблема во всех этих подходах: если вы изменяете ваши маршруты, компилятор не предупредит вас о сломанных ссылках. Не говоря уже об опечатках, которые также могут причинить большой ущерб.

Цель введения типобезопасных URL - это позволить компилятору проверять за нами как можно больше вещей. Чтобы сделать этому, нашим первым шагом должен быть уход от обычного текста, который компилятор не умеет понимать, к каким-то хорошо определенным типам данных. Для нашего простого приложения, давайте будем моделировать наши маршруты следующим типом данных:

```
data MyRoute = Home | Time
```


Вместо того, чтобы добавлять в наш шаблон ссылку вида `/display/time`, мы можем использовать конструктор `Time`. Но в итоге HTML - это текст, а не типы данных, так что мы должны иметь какой-то способ конвертировать эти значения в текст. Мы называем это - функцией рендеринга URL в строку, одна из которых:

```
renderMyRoute :: MyRoute -> Text
renderMyRoute Home = "http://example.com/profile/home"
renderMyRoute Time = "http://example.com/display/time"
```

Функции рендеринга URL обычно чуть более сложные, чем описанный выше пример. Они должны уметь работать с параметрами строки запросов (`query string parameters`), с аргументами конструкторов и более правильно управляться с доменными именами. На практике, вам не придется беспокоиться об этом, т.к. `Yesod` автоматически создает вам функции рендеринга. Единственная вещь, которую стоит здесь отметить, - это то, что сигнатура типа, на самом деле, чуть усложняется для работы с параметрами строки запросов.

```
type Query = [(Text, Text)]
type Render url = url -> Query -> Text
renderMyRoute :: Render MyRoute
renderMyRoute Home _ = ...
renderMyRoute Time _ = ...
```

Что ж, хорошо, у нас теперь есть наша функция рендеринга в строку, и у нас есть типобезопасные URL, встроенные в шаблоны. Как всё это работает вместе? Вместо генерации HTML (или CSS или Javascript) значений напрямую, Шекспировские шаблоны фактически создают функции, которые, используя данную функцию рендеринга, создает сам HTML. Чтобы лучше понять данную концепцию, давайте взглянем на то, как `Hamlet` обработает следующий шаблон:

```
<a href=@{Time}>The time
```

Эта строчка будет представлена примерно таким Haskell кодом

```
\render -> mconcat ["<a href='", render Time, "'>The_time</a>"]
```

4.4. Синтаксис

Все Шекспировские языки используют один и тот же интерполяционный синтаксис, и все они могут использовать типобезопасные URL. Они отличаются синтаксисом, специфичном для их целевых языков (HTML, CSS, Javascript).

4.5. Синтаксис языка Hamlet

Hamlet - это самый сложный из языков семейства. Он предоставляет не только синтаксис для генерации HTML, но также и базовые структуры для контроля: условия, циклы, maybes.

4.5.1. Тэги

Очевидно, тэги играют важную роль в любом шаблонном языке HTML. В Hamlet мы пытаемся оставаться как можно ближе к текущему синтаксису HTML, чтобы сделать язык более удобным. Однако, вместо использования закрывающих тэгов для обозначения вложенности, мы используем отступы. Например, следующий код в HTML:

```
<body>
<p>Какой-то абзац.</p>
<ul>
<li>пункт 1</li>
<li>пункт 2</li>
</ul>
</body>
```

будет выглядеть следующим образом:

```
<body>
  <p>Какой-то абзац.
  <ul>
    <li>Пункт 1
    <li>Пункт 2
```

В целом, мы считаем, что этот способ более прост в использовании, чем оригинальный HTML, как только вы привыкнете к нему. Единственная сложная часть - это работа с пробелами до и после тэгов. К примеру, скажем, вы хотите создать следующий HTML:

```
<p>Абзац <i>курсив</i> конец.</p>
```

Мы хотим быть уверенными, что пробелы останутся после слова "Paragraph" и до слова "end". Чтобы добиться этого мы используем два простых escape-символа:

```
<p>
  Абзац #
  <i>курсив
  \ конец.
```

Правила для использования escape-пробелов очень простые:

1. Если первый непробельный символ в строке - это обратная косая черта, он игнорируется

2. Если последний символ в строке - это решетка, он игнорируется

И ещё одна важная особенность. Hamlet не экранирует содержимое своего контента. Это сделано намеренно, чтобы позволять копировать уже существующий HTML-код. То есть пример выше может быть записан как:

```
<r>Абзац <i>курсив</i> конец.
```

Заметим, что Hamlet автоматически закроет первый тэг, в то время как внутренний тэг "i" останется как есть. Вы вольны использовать любой из предложенных подходов, они равноценны. Будьте, однако, осторожны, Hamlet закрывает только подобные встроены тэги; нормальные тэги не будут закрыты.

4.5.2. Интерполяция

Всё, что мы видели до текущего момента - это приятный, упрощенный HTML, но он не предоставляет никакой возможности взаимодействовать с нашим кодом на Haskell. Как мы будем встраивать в него переменные? Очень просто: с помощью интерполяции:

```
<head>
  <title>#{title}
```

Символ решетки #, за которым следуют фигурные скобки - определяет интерполяцию переменной. В примере выше будет использована переменная title из той области видимости, где был использован шаблон. Мне хочется подчеркнуть ещё раз: Hamlet имеет доступ к переменным той области видимости, из которой был вызван. При этом не нужно особым образом указывать список использованных в шаблоне переменных.

Вы также можете применять функции внутри интерполяции. Вы можете использовать строковые и численные литералы в интерполяции. Вы можете использовать квалификационные имена модулей (`import qualified ... as ...`). Внутри интерполяции можно использовать как скобки, так и знак доллара \$ для группировки выражений. И ещё, в самом конце, функция `toHtml` применится к результату вычисления, что значит, что все экземпляры класса `ToHtml` могут быть сынтерполированы. Возьмите, к примеру, следующий код.

```
-- Пока что игнорируйте QuasiQuotes и функцию shamlet, мы всё это []
  объясним позже
{-# LANGUAGE QuasiQuotes #-}

import Text.Hamlet (shamlet)
import Text.Blaze.Html.Renderer.String (renderHtml)
import Data.Char (toLower)
import Data.List (sort)

data Person = Person
  { name :: String
```

```

    , age :: Int
    }

main :: IO ()
main = putStrLn $ renderHtml [shamlet|
<p>Привет, меня зовут #{name person} и мне #{show $ age person}.
<p>
    Давайте сделаем с моим именем разные смешные штуки: #
    <b>#{sort $ map toLower (name person)}
<p>Ох, а через 5 лет мне будет #{show ((+) 5 (age person))}.
|]
where
    person = Person "Michael" 26

```

04/var-interpolation.hs

Что же с нашими расхваленными типобезопасными URL? Они почти что идентичны интерполяции переменных по всем показателям, за исключением того, что они начинаются с символа @. Кроме того, существует также встраивание с помощью знака вставки ^, которое позволяет вам вставить другой шаблон того же типа. Код ниже демонстрирует обе описанные концепции.

```

{-# LANGUAGE QuasiQuotes #-}
{-# LANGUAGE OverloadedStrings #-}

import Text.Hamlet (HtmlUrl, hamlet)
import Text.Blaze.Html.Renderer.String (renderHtml)
import Data.Text (Text)

data MyRoute = Home

render :: MyRoute -> [(Text, Text)] -> Text
render Home _ = "/home"

footer :: HtmlUrl MyRoute
footer = [hamlet|
<footer>
    Назад на #
    <a href=@{Home}>Домашнюю страницу
    .
|]

main :: IO ()
main = putStrLn $ renderHtml $ [hamlet|
<body>
    <p>Это моя страница.
    ^{footer}

```

```
|] render
```

```
04/url-interpolation.hs
```

Кроме того, есть вариант интерполяции URL, который даёт вам возможность встраивать параметры строки запроса. Это может быть полезно, например, для создания постраничных ответов. Вместо `@{...}` используйте вариант со знаком вопроса (`@?{...}`) для обозначения присутствия параметров строки запроса. Передаваемое значение должно быть двухэлементным кортежем, первое значение в котором — это типобезопасный URL, а второе — список пар параметров строки запроса. Для примера смотрите фрагмент кода ниже.

```
<p>
  Вы сейчас на странице #{currPage}.
  <a href=@?{(SomePage, [("page", pack $ show $ currPage - 1])}>Предыдущая
  <a href=@?{(SomePage, [("page", pack $ show $ currPage + 1])}>Следующая
```

Листинг 4.1: Интерполяция URL с параметрами в строке запроса

В итоге создаётся вполне ожидаемый HTML:

```
<p>Вы сейчас на странице 2.
  <a href="/home?page=1">Предыдущая</a>
  <a href="/home?page=3">Следующая</a>
</p>
```

4.5.3. Атрибуты

В последнем примере, мы добавили `href` атрибут в тэг `a`. Давайте уточним синтаксис:

- Вы можете иметь интерполяции внутри значения атрибута.
- Знак равенства и значения атрибута - опциональны, точно так же, как в HTML. Так что `<input type=checkbox checked>` - абсолютно валидный код.
- Есть два удобных атрибута: для `id`, вы можете использовать знак решетка `#`, а для класса - точку `..`. Например, `<p #paragraphid .class1 .class2>`.
- Кавычки вокруг значения атрибута опциональны, но они обязательны, если вы хотите использовать пробел в составе этого значения.
- Вы можете использовать атрибут опционально с помощью использования двоеточия. Чтобы сделать `checkbox` активированным только если переменная `isChecked` установлена в `True`, вы можете написать `<input type=checkbox :isChecked:checked>`. Чтобы сделать параграф опционально красным, вы можете использовать `<p :isRed:style="color:red">`.

4.5.4. Условные выражения

В конечном счёте вы захотите добавить некоторую логику на вашу страницу. Цель языка Hamlet - это сделать эту логику как можно более минималистичной, возлагая всю сложную работу на Haskell. Например, логические выражения - очень просты... Настолько просты, что это базовый набор `if`, `elsif` и `else`.

```
$if isAdmin
  <p>Добро пожаловать в админскую секцию.
$elseif isLoggedIn
  <p>Вы не администратор.
$else
  <p>Я не знаю, кто Вы. Пожалуйста, войдите в систему, чтобы я мог
  понять, есть ли у вас доступ.
```

Все те же самые правила обычной интерполяции применяются и к содержимому условных выражений.

4.5.5. Maybe

Подобным же образом, мы имеем специальную конструкцию, чтобы работать со значениями **Maybe**. Технически, это можно делать с помощью `if`, `isJust` и `fromJust`, но наш подход более удобный и позволяет избегать частично вычисляемых функций.

```
$maybe name <- maybeName
  <p>Вас зовут #{name}
$nothing
  <p>Я не знаю Вашего имени.
```

Помимо простых идентификаторов вы можете использовать слева от стрелки некоторые другие, более сложные значения, например, конструкторы и кортежи.

```
$maybe Person firstName lastName <- maybePerson
  <p>Ваше имя - #{firstName} #{lastName}
```

Выражение справа от стрелки подчиняется тем же правилам интерполяции, позволяет использовать переменные, вызывать функции и т.д.

4.5.6. Forall

А что с пробеганием по элементам списка? Мы покрываем и этот шаблон:

```
$if null people
  <p>Нет людей.
$else
  <ul>
    $forall person <- people
      <li>#{person}
```

4.5.7. Case

Сопоставление шаблонов - одна из сильных сторон Haskell. Алгебраические типы данных позволяют вам удобно моделировать любые типы данных из реального мира, а case-выражения позволяют вам безопасно их сопоставлять, возлагая на компилятор обязанность предупреждать, если вы пропустили одно из значений. Hamlet предоставляет вам те же самые преимущества.

```
$case foo
  $of Left bar
    <p>Он был слева: #{bar}
  $of Right baz
    <p>Он был справа: #{baz}
```

4.5.8. With

Мы заканчиваем обзор выражений языка, рассматривая with. Это, по сути, просто удобный способ объявления синонима для длинного выражения. К примеру, в данном примере мы определяем один раз переменную с помощью with, а затем используем ее дважды.

```
$with foo <- some very (long ugly) expression that $ should only $
  happen once
  <p>Я использую эту переменную #{foo} несколько раз. #{foo}
```

4.5.9. Doctype

Последняя ложка синтаксического сахара: выражение doctype. Мы поддерживаем несколько разных версий doctype, хотя мы рекомендуем \$doctype 5 для современных интернет-приложений, который генерирует <!DOCTYPE html>

```
$doctype 5
<html>
  <head>
    <title>Hamlet просто чудесен
  <body>
    <p>Все готово.
```

Мы всё ещё поддерживаем и старый синтаксис: три восклицательных знака !!! . Вы всё ещё можете увидеть его в коде. Мы планируем и дальше поддерживать этот синтаксис, но в целом, находим \$doctype подход более легким для чтения.

4.6. Синтаксис языка Cassius

Cassius - это оригинальный шаблонный язык CSS. Он использует простые правила с пробелами для разделения блоков, делая фигурные скобки и точки с запятой необязательными. Он поддерживает интерполяцию переменных и URL, но не поддерживает встраивание. Его синтаксис очень простой:

```
#banner
  border: 1px solid #{bannerColor}
  background-image: url(@{BannerImageR})
```

4.7. Синтаксис языка Lucius

В то время, как Cassius использует модифицированный, чувствительный к пробелам синтаксис для CSS, подход Lucius полностью оригинальный. Вы можете взять любой CSS файл и он будет валидным Lucius файлом. Однако, есть некоторые дополнения:

- Как в Cassius, мы позволяем интерполяцию как переменных, так и URL.
- Возможность вложенных CSS блоков.
- Вы можете определять переменные в ваших шаблонах.

Что касается второго пункта: скажем, вы хотите иметь специфичный стиль для какого-то тэга в вашей статье. В плоском CSS вы должны были бы написать

```
article code { background-color: grey; }
article p { text-indent: 2em; }
article a { text-decoration: none; }
```

В этом случае получилось не так уж и много клозов, но необходимость печатать `article` каждый раз - слегка надоедает. А представьте, что таких выражений, скажем, около десятка - согласитесь, раздражает. Lucius поможет в этом случае:

```
article {
  code { background-color: grey; }
  p { text-indent: 2em; }
  a { text-decoration: none; }
  > h1 { color: green; }
}
```

Само наличие переменных в Lucius позволяет вам избегать повторов. Простым примером будет определить общий используемый цвет как переменную:

```
@textcolor: #ccc; /* просто потому, что мы ненавидим наших □
пользователей */
```



```
body { color: #{textcolor} }  
a:link, a:visited { color: #{textcolor} }
```

В остальном, Lucius идентичен CSS.

4.8. Синтаксис языка Julius

Julius - наиболее простой среди всех обсуждаемых здесь языков. На самом деле, многие могут сказать, что это на самом деле всего лишь Javascript. Julius позволяет использовать три формы интерполяции, которые мы уже упоминали. Либо, вообще не применяет никаких преобразований к вашему контенту.

Если вы используете Julius со сгенерированным каркасом сайта Yesod, вы возможно заметили, что ваш Javascript автоматически минифицирован. Это не следствие использования Julius, это результат использования пакета hjsmin в Yesod, который служит для компактизации выдачи Julius.

4.9. Как заставить Шекспировские шаблоны работать

Конечно, в какой-то момент у вас появится этот вопрос: "Как же мне заставить это всё работать?". Существует три разных способа вызывать Шекспировские языки из вашего Haskell кода:

4.9.1. Квази-цитирование

Квази-цитирование (quasiquotes) позволяют вам встроить произвольное содержимое в ваш Haskell код, и конвертировать это содержимое в Haskell во время компиляции.

4.9.2. Внешние файлы

В этом случае, код шаблонов находится в отдельных файлах, на который мы ссылаемся с помощью шаблонов Haskell.

4.9.3. Режим перезагрузки

Оба использованных выше подхода требуют полной перекомпиляции для того, чтобы увидеть какие-то изменения. В режиме перезагрузки ваши шаблоны хранятся в отдельных файлах, и ссылаются из шаблонов Haskell. Но при этом, в рантайме, эти внешние файлы каждый раз пересканируются на лету

Режим перезагрузки нельзя использовать для Hamlet, только для Cassius, Lucius и Julius. В Hamlet слишком много сложной логики, которая полагается напрямую на Haskell компилятор, что не позволяет использовать этот режим для Hamlet.

Первые два подхода следует использовать в "боевых" условиях на серверах. Они оба встраивают целиком шаблоны в финальный исполняемый файл, упрощая развертывание кода и увеличивая производительность. Достоинство квази-цитирования - это простота: всё остается в одном файле. Для коротких шаблонов, это может быть очень удобно. Однако, в целом, мы рекомендуем использовать внешние файлы для шаблонов, поскольку:

- Это следование традиции разделения логики и представления данных.
- Вы можете легко переключиться между внешними файлами и отладочным режимом работы с помощью простых макро-определений CPP, в том смысле, что вы можете совмещать быструю и удобную разработку и всё ещё получать высокую производительность на "боевых" серверах.

Так как это специальные функции Квази-цитирования и шаблонного Haskell, вы должны удостовериться, что вы используете соответствующие расширения языка и корректный синтаксис. Вы можете посмотреть простейшие примеры каждого подхода на следующих примерах

```
{-# LANGUAGE OverloadedStrings #-} -- мы используем Text чуть ниже
{-# LANGUAGE QuasiQuotes #-}

import Text.Hamlet (HtmlUrl, hamlet)
import Data.Text (Text)
import Text.Blaze.Html.Renderer.String (renderHtml)

data MyRoute = Home | Time | Stylesheet

render :: MyRoute -> [(Text, Text)] -> Text
render Home _ = "/home"
render Time _ = "/time"
render Stylesheet _ = "/style.css"

template :: Text -> HtmlUrl MyRoute
template title = [hamlet|
$doctype 5
<html>
  <head>
    <title>#{title}
    <link rel=stylesheet href=@{Stylesheet}>
  <body>
    <h1>#{title}
|]

main :: IO ()
main = putStrLn $ renderHtml $ template "Мой_заголовок" render
```

Листинг 4.2: Квазицитирование

```

{-# LANGUAGE OverloadedStrings #-} -- мы используем Text чуть ниже
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE CPP #-} -- чтобы контролировать "боевой" режим против
    отладочного

import Text.Lucius (CssUrl, luciusFile, luciusFileDebug, renderCss)
import Data.Text (Text)
import qualified Data.Text.Lazy.IO as TLIO

data MyRoute = Home | Time | Stylesheet

render :: MyRoute -> [(Text, Text)] -> Text
render Home _ = "/home"
render Time _ = "/time"
render Stylesheet _ = "/style.css"

template :: CssUrl MyRoute
#if PRODUCTION
template = $(luciusFile "template.lucius")
#else
template = $(luciusFileDebug "template.lucius")
#endif

main :: IO ()
main = TLIO.putStrLn $ renderCss $ template render
-- @template.lucius
foo { bar: baz }

```

Листинг 4.3: Внешние файлы

Способ	именования	функций	очень	консистентная.
Language	Quasiquoter	External file	Reload	
Hamlet	hamlet	hamletFile	N/A	
Cassius	cassius	cassiusFile	cassiusFileReload	
Lucius	lucius	luciusFile	luciusFileReload	
Julius	julius	juliusFile	juliusFileReload	

4.10. Добавление функциональности в типы Hamlet

До сих пор мы видели только как генерировать `HtmlUrl` значение из `Hamlet`, что является HTML со встроенными типо-безопасными URL. Используя `Hamlet`, мы можем генерировать еще три других результата: простой HTML, HTML с URL и интернациональными текстами, и виджеты. Последние будут отдельно описаны в главе "Виджеты" 5.

Чтобы сгенерировать простой HTML без встроенных URL, мы используем "Упрощенный `Hamlet`" с некоторыми изменениями:

- Мы используем другой набор функций, которые начинаются с дополнительного префикса "s". Так что квази-цитирование - это `shamlet` и функция использования внешнего файла - это `shamletFile`. Вопрос о том, как следует произносить названия этих функций, всё ещё находится в обсуждении.
- Не позволяется использовать интерполяцию URL. Если попытаться это сделать, мы получим ошибку во время компиляции.
- Встраивание (интерполяция-вставка) более не позволяет использовать произвольные `HtmlUrl` значения. Правило здесь таково, что встраиваемое значение должно иметь точно такой же тип, что и сам шаблон, что в данном случае - `Html`. Это означает, что для `shamlet`, встраивание может быть полностью заменено интерполяцией переменных (с использованием решетки #).

Работа с интернационализацией (`i18n`) в `Hamlet` происходит чуть более сложно. `Hamlet` поддерживает `i18n` с помощью специального типа данных для сообщений - решение концептуально и по реализации похоже на типобезопасные URL. Для примера, скажем, мы хотим написать программу, которая приветствует нас и говорит, как много яблок мы съели. Мы могли бы представить эти сообщения в виде типа данных

```
data Msg = Hello | Apples Int
```

Далее, мы бы хотели иметь возможность представлять это в каком-то более удобном для чтения формате, поэтому мы определяем некоторую функцию рендеринга:

```
renderEnglish :: Msg -> Text
renderEnglish Hello = "Hello"
renderEnglish (Apples 0) = "You did not buy any apples."
renderEnglish (Apples 1) = "You bought 1 apple."
renderEnglish (Apples i) = T.concat ["You bought T.pack $ show i, "
  apples."]
```

Теперь мы хотим интерполировать эти значения типа `Msg` напрямую в шаблоне. Для этого мы используем интерполяцию с подчеркиванием.

```
$doctype 5
<html>
  <head>
    <title>i18n
  <body>
    <h1>_{Hello}
    <p>_{Apples count}
```

Этот шаблон, теперь надо как-то представить в HTML. Для этого, как и для типобезопасных URL, мы определяем специальную функцию рендеринга в строку. Для удобства, определим синоним типа:

```

type Render url = url -> [(Text, Text)] -> Text
type Translate msg = msg -> Html
type HtmlUrlI18n msg url = Translate msg -> Render url -> Html

```

В данной точке, вы можете передать `renderEnglish`, `renderSpanish` или `renderKlingon` в этот шаблон. И он сгенерирует хорошо переведенный контент (качество которого, конечно, зависит от ваших переводчиков). Программа целиком будет выглядеть так:

```

{-# LANGUAGE QuasiQuotes #-}
{-# LANGUAGE OverloadedStrings #-}

import Data.Text (Text)
import qualified Data.Text as T
import Text.Hamlet (HtmlUrlI18n, ihamlet)
import Text.Blaze.Html (toHtml)
import Text.Blaze.Html.Renderer.String (renderHtml)

data MyRoute = Home | Time | Stylesheet

renderUrl :: MyRoute -> [(Text, Text)] -> Text
renderUrl Home _ = "/home"
renderUrl Time _ = "/time"
renderUrl Stylesheet _ = "/style.css"

data Msg = Hello | Apples Int

renderRussian :: Msg -> Text
renderRussian Hello = "Привет"
renderRussian (Apples 0) = "Вы не купили яблок."
renderRussian (Apples 1) = "Вы купили одно яблоко."
renderRussian (Apples i) = T.concat ["Вы купили", T.pack $ show i,
  if i < 5 then " яблока." else " яблок."]

template :: Int -> HtmlUrlI18n Msg MyRoute
template count = [ihamlet|
  $doctype 5
  <html>
    <head>
      <title>i18n
      <body>
        <h1>_{Hello}
        <p>_{Apples count}
    |]

```

```
main :: IO ()
main = putStrLn $ renderHtml
      $ (template 5) (toHtml . renderRussian) renderUrl
```

04/i18n.hs

4.11. Другие Шекспировские возможности

В дополнение к языкам-помощникам для HTML, CSS и Javascript, есть также и более общий пакет утилит. `shakespeare-text` предоставляет простой способ создания интерполированных строк, который очень нравится тем, кто привык к скриптовым языкам вроде Ruby или Python. Конечно, утилиты этого пакета можно использовать не только для Yesod.

```
{-# LANGUAGE QuasiQuotes, OverloadedStrings #-}

import Text.Shakespeare.Text
import qualified Data.Text.Lazy.IO as TLIO
import Data.Text (Text)
import Control.Monad (forM_)

data Item = Item
  { itemName :: Text
  , itemQty  :: Int
  }

items :: [Item]
items =
  [ Item "яблоки" 5
  , Item "бананы" 10
  ]

main :: IO ()
main = forM_ items $ \item -> TLIO.putStrLn
      [lt|У вас есть #{show $ itemQty item} #{itemName item}.|]
```

04/shakespeare-text.hs

Несколько быстрых подсказок по этому простому примеру:

- Мы используем три разных текстовых типа данных в примере (**String**, строгий и ленивый `Text`). Они замечательно работают вместе.
- Мы используем оператор квазицитирования `lt`, который генерирует ленивую версию текста. Также есть его строгий аналог `st`.
- Есть и более длинные имена для этих операторов (`ltext` и `stext`).

4.12. Общие рекомендации

Ниже приводятся общие рекомендации комьюнити Yes od по использованию Шекспировского семейства языков.

- Для сайтов используйте внешние файлы. Для библиотек это вполне нормально - использовать квази-квотирование, при условии, что оно не слишком большое.
- Патрик Брисбин (Patrick Brisbin) сделал правила раскраски для Vim, которые сильно помогают при разработке.
- Вам следует почти всегда начинать тэги `hamlet` на новых строках вместо того, чтобы писать их после существующих тэгов. Единственное исключение этому правилу - это использование `<i>` и `` тэгов в большом блоке текста.

Глава 5

Виджеты

Одна из проблем в веб-разработке — нам приходится координировать три различные клиентские технологии: HTML, CSS и Javascript. Хуже того, мы должны размещать эти компоненты в различных местах на странице: CSS в теге `style` в заголовке, Javascript в теге `script` в заголовке, а HTML в теле. И это не говоря уже о случае, когда вы хотите разместить свои CSS и Javascript в отдельных файлах!

На деле, это работает довольно хорошо при создании одиночной страницы, потому что мы можем разделить нашу структуру (HTML), стили (CSS) и логику (Javascript). Но когда мы хотим строить код из модулей, которые можно будет легко компоновать, раздельная координация всех трёх частей может стать головной болью. Виджеты — решение Yesod для этой задачи. Они также помогают с проблемой однократного подключения библиотек наподобие jQuery.

Наши четыре языка шаблонов — Hamlet, Cassius, Lucius и Julius — предоставляют исходные инструменты для конструирования вашего вывода. Виджеты обеспечивают клей, который позволяет работать им вместе как единое целое.

5.1. Краткий обзор

```
getRootR = defaultLayout $ do
  setTitle "Название_моей_страницы"
  toWidget [lucius| h1 { color: green; } |]
  addScriptRemote []
  "https://ajax.googleapis.com/ajax/libs/jquery/1.6.2/jquery.min.js"
  toWidget [julius|
$(function() {
  $("h1").click(function(){ alert("Вы_кликнули_на_заголовок!"); });
});
|]
  toWidgetHead [hamlet| <meta name=keywords content="некоторые_
ключевые_слова_примера">|]
  toWidget [hamlet| <h1> Это один способ вставки контента |]
```



```
[whamlet| <h2> А это другой |]
toWidgetBody [julius| alert("Это вставлено в само тело"); |]
```

Этот код порождает следующий HTML (отступы добавлены):

```
<!DOCTYPE html>
<html>
  <head>
    <title>Название моей страницы</title>
    <style>h1 { color : green }</style>
    <script [
      src="https://ajax.googleapis.com/ajax/libs/jquery/1.6.2/jquery.min.js"></script>
    <script>
$(function() {
  $("h1").click(function(){ alert("Вы кликнули на заголовок!"); });
});
</script>
    <meta name="keywords" content="некоторые ключевые слова [
      примера">
    </head>
    <body>
      <h1> Это один способ вставки контента </h1>
      <h2> А это другой </h2>
      <script> alert("Это вставлено в само тело"); </script>
    </body>
  </html>
```

5.2. Что в Виджете?

На самом поверхностном уровне, HTML документ — это просто набор вложенных тегов. Такой подход выбирают большинство генерирующих HTML инструментов: вы просто определяете иерархию тегов и всё. Но давайте представим, что я хочу написать компонент страницы для отображения навигационной панели. Я хочу, чтобы он был «plug and play»: я просто вызываю функцию в нужное время, и панель вставляется в соответствующее место в иерархии.

Тут наш поверхностный подход к генерации HTML даёт сбой. Наша навигационная панель вероятно содержит какой-нибудь CSS и Javascript в дополнение к HTML. К тому времени, когда мы вызываем функцию навигационной панели, мы уже сформировали тег <head>, так что уже поздно добавлять новый тег <style> для наших CSS объявлений. В рамках обычных стратегий, нам нужно было бы разбить нашу функцию навигационной панели на три части: HTML, CSS и JavaScript, и убедиться, что мы всегда вызываем все три части.

Виджеты используют другой подход. Вместо рассмотрения HTML документа как монолитного дерева тегов, виджеты видят на странице набор различных компонентов. В частности:

- Название страницы
- Внешние стили
- Внешний Javascript
- Объявления CSS
- Код Javascript
- Произвольный контент в `<head>`
- Произвольный контент в `<body>`

Различные компоненты имеют различный смысл. К примеру, может быть только одно название, но может быть множество внешних скриптов и стилей. Однако, эти внешние скрипты и стили должны быть добавлены только раз. Произвольный контент в заголовке и теле, с другой стороны, не имеет ограничений (кто-то, возможно, захочет пять блоков `lorem ipsum`, в конце концов).

Работа виджета — держаться за эти несопоставимые компоненты и применять правильную логику для комбинирования виджетов вместе. Логика эта включает такие действия, как фиксация последнего установленного названия и игнорирование остальных, фильтрация дубликатов из списка внешних скриптов и стилей, объединение содержимого заголовка и тела.

5.3. Конструирование Виджетов

Для использования виджетов, вам, очевидно, необходима возможность их создания. Наиболее распространённый способ сделать это — через класс типов `Towidget` и его метод `toWidget`. Метод предоставляет вам возможность конвертировать ваши Shakespearean шаблоны прямо в `Widget`: код Hamlet появится в теле, скрипты Julius — в теге `<script>` в заголовке, а Cassius и Lucius — в теге `<style>` заголовка.

На самом деле, вы можете переопределить стандартное поведение и получить скрипт и код стилей в отдельных файлах. Сгенерированный начальный каркас сайта делает это для вас автоматически. Кроме того, в главе «Класс типов `Yesod`» мы увидим, как включить асинхронную загрузку скриптов, которая поместит содержимое ваших скриптов в конец тела страницы.

Но что, если вы захотите добавить какие-нибудь теги `<meta>`, которые должны появиться в заголовке? Или если вы захотите, чтобы некоторый Javascript появился в теле, а не в заголовке? Для этих целей, `Yesod` предоставляет два дополнительных класса типов: `TowidgetHead` и `TowidgetBody`. Они работают именно так, как вы от них ожидаете.

Кроме того, есть ряд других функций для создания специфических видов виджетов:

setTitle

Преобразует некоторый HTML в название страницы.

addCassiusMedia, addLuciusMedia

Работают также как `toWidget`, но принимают дополнительный параметр для обозначения, к какому типу медиа контента применяется. Полезно для создания стилей печати, например.

addStylesheet

Добавляет ссылку, используя тег `<link>`, на внешнюю таблицу стилей. Принимает типобезопасный URL.

addStylesheetRemote

Аналогична `addStylesheet`, но принимает обычный URL. Полезна для ссылок на файлы, хранящиеся на CDN, подобно таблицам стилей CSS для jQuery UI, предоставляемым Google.

addScript

Добавляет ссылку, используя тег `<script>`, на внешний скрипт. Принимает типобезопасный URL.

addScriptRemote

Аналогична `addScript`, но принимает обычный URL. Полезна для ссылок на файлы, хранящиеся на CDN, подобно jQuery от Google.

5.4. Комбинирование Виджетов

Вся идея виджетов — это повышение компонуемости. Вы можете взять отдельные части HTML, CSS и Javascript, скомбинировать их вместе в нечто более сложное, и затем скомпоновать эти большие сущности в законченную страницу. Это всё работает, естественно, через экземпляр класса **Monad** для `Widget`, и значит вы можете использовать `do`-нотацию для комбинирования частей вместе.

```
myWidget1 = do
  toWidget [hamlet|<h1>My Title|]
  toWidget [lucius|h1 { color: green } |]

myWidget2 = do
  setTitle "My Page Title"
  addScriptRemote "http://www.example.com/script.js"

myWidget = do
  myWidget1
  myWidget2

-- или, если захотите
myWidget' = myWidget1 >> myWidget2
```

Листинг 5.1: Комбинирование Виджетов

Если для вас это принципиально, для `Widget` есть также экземпляр класса `Monoid`, поэтому вы можете использовать `mconcat` или монаду `Writer` для построения сущностей. По моему опыту, самое лёгкое и самое естественное просто использовать `do`-нотацию.

5.5. Генерирование идентификаторов

Если мы на самом деле выбираем настоящее повторное использование кода, мы со временем получим конфликт имён. Скажем, есть две вспомогательные библиотеки, и обе используют класс с именем «foo» для изменения стилового оформления. Мы хотим избежать такой возможности. И для этого у нас есть функция `newIdent`. Эта функция автоматически генерирует слово, которое уникально для текущего обработчика.

```
getRootR = defaultLayout $ do
  headerClass <- lift newIdent
  toWidget [hamlet|<h1 .#{headerClass}>My Header|]
  toWidget [lucius| .#{headerClass} { color: green; } |]
```

Листинг 5.2: Использование `newIdent`

Вам, наверное, интересно: что означает `lift`? `Widget` — это трансформатор монад, базирующийся на `Handler`. `newIdent` — это функция `Handler`, поэтому нам необходимо «втянуть» (`lift`) её с уровня `Handler` на уровень `Widget`, чтобы использовать. На самом деле, мы можем использовать точно такой же подход для выполнения сложных действий, наподобие запросов к базе данных, из виджета. Мы подробно рассмотрим этот вопрос при обсуждении монад в `Yesod` в главе 13.

5.6. `whamlet`

Предположим, у вас есть довольно стандартный шаблон `Hamlet`, который встраивает другой шаблон `Hamlet` для отображения подвала страницы (`footer`):

```
page = [hamlet|
<p>Это моя страница. Я надеюсь, вы наслаждаетесь ею.
^{footer}
|]

footer = [hamlet|
<footer>
  <p>Это всё, люди!
|]
```

Это работает хорошо, если в `footer` обычный старый HTML, но что если мы хотим добавить какое-нибудь стиловое оформление? Хорошо, мы можем легко приправить `footer`, преобразовав его в виджет:

```

footer = do
  toWidget [lucius| footer { font-weight: bold; text-align: center |
  } |]
  toWidget [hamlet|
<footer>
  <p>Это всё, люди!
|]

```

Но теперь у нас проблема: шаблон Hamlet может встраивать только другой шаблон Hamlet; он ничего не знает о widget. И вот тут на сцену выходит whamlet. Он принимает точно такой же синтаксис, как обычный Hamlet, и не меняет интерполяцию переменных (`{...}`) и URL (`@{...}`). Но встраивание (`^{...}`) принимает widget, и финальный результат тоже widget. Для его использования, мы можем просто сделать так:

```

page = [whamlet|
<p>Это моя страница. Я надеюсь, вы наслаждаетесь ею.
^{footer}
|]

```

Есть также `whamletFile`, если вы предпочитаете держать ваш шаблон в отдельном файле.

В каркасном сайте есть ещё более удобная функция, `widgetFile`, которая также автоматически включает ваши Lucius, Cassius и Julius файлы. Мы рассмотрим это в главе Создание каркаса сайта.

5.6.1. Типы

Возможно, вы уже заметили, что я избегал сигнатуры типов до сих пор. Это потому, что в них заключена некоторая сложность. На самом простом уровне, всё, что вам нужно знать, — это что есть синоним типов с именем `Widget`, который вы почти всегда и будете использовать. Дальше идут технические подробности, но не беспокойтесь, если они окажутся немного туманными.

На самом деле, нет типа `Widget`, определённого в библиотеках Yesod, так как его точный смысл меняется от сайта к сайту. Взамен, у нас есть более общий тип `GWidget sub master a`. Первые два параметра — это основные типы подсайта и основного сайта, соответственно. Последний параметр — это контейнерное значение, как в любом экземпляре **Monad**.

Так в чём дело с этими `sub/master` штуковинами? Когда вы пишете некий повторно используемый код, такой как CRUD приложение, вы можете сделать его как подсайт, который может быть встроено в любое другое приложение Yesod. В таком случае, нам нужно отслеживать информацию для обоих сайтов: подсайта и основного. Самый простой пример — типобезопасные URL: Yesod должен знать, как получить путь для CRUD подсайта и преобразовать его в путь для основного сайта, чтобы корректно его отобразить.

Однако, эти различия подсайт/основной сайт имеют значение только тогда, когда вы имеете дело с подсайтами. Когда же вы пишете код вашего стандартного ответа, вы работаете только со своим приложением, и в этом случае подсайт и основной сайт — это одно и то же. Поскольку это самый распространённый случай, каркасный сайт объявляет синоним типа, чтобы помочь вам. Скажем, ваш основной тип это `MyCoolApp`, тогда будет определён `type widget = Gwidget MyCoolApp MyCoolApp ()`. И как следствие, мы можем получить очень даже дружелюбные к пользователю сигнатуры типа для наших виджетов:

```
footer :: Widget
footer = do
  towidget [lucius| footer { font-weight: bold; text-align: center |
  } |]
  towidget [hamlet|
<footer>
  <p>Это всё, люди!
|]

page :: Widget
page = [whamlet|
<p>Это моя страница. Я надеюсь, вы наслаждаетесь ею.
^{footer}
|]
```

Если вы пристально следили за изложением, вы, должно быть, сбиты с толку. Выше, в примере с генерированием идентификаторов, мы использовали `lift` для `widget`, но `Gwidget` на самом деле не является трансформатором монад. Что же здесь происходит? Да, в ранних версиях `Yesod`, он и был трансформатором вокруг типа `Handler`. К несчастью, это привело к сложным для анализа сообщениям об ошибках. Как результат, `Gwidget` теперь обёртка **`newtype`**, которая прячет свою сущность трансформатора монад. Но мы по-прежнему хотим иметь возможность втягивать (`lift`) функции из внутренней монады `Handler`.

В качестве решения `Yesod` предоставляет альтернативную, более общую функцию `lift`, которая работает для обоих стандартных экземпляров класса `MonadTrans`, и специальные обёртки **`newtype`** наподобие `Gwidget`. В результате, вы можете считать, что `Gwidget` — это стандартный трансформатор, в то же время сохраняя ваше понятное сообщение об ошибке.

Одно последнее замечание: как у нас есть разделение между `Widget` и `Gwidget`, у нас есть точно такое же разделение между `Handler` и `GHandler`.

5.7. Использование виджетов

Это, конечно, замечательно, что у нас есть эти красивые типы данных для виджетов, но как именно нам превратить их во что-то, с чем пользователь может взаимодействовать? Самая часто используемая функция — это `defaultLayout`, которая по

существо имеет сигнатуру типа `Widget -> Handler RepHtml`. (Я говорю «по существу» из-за целой проблемы с `GHandler`.) `RepHtml` — это тип данных, содержащий некоторый сырой HTML вывод, готовый к отправке в сеть.

На самом деле, `defaultLayout` — метод класса типов, который может быть переопределён для каждого приложения. Так для приложений `Yesod` задаются темы¹. Но мы всё ещё остаёмся с вопросом: когда мы внутри `defaultLayout`, как нам развернуть `Widget`? Ответ — `widgetToPageContent`. Давайте посмотрим на некоторые (упрощённые) типы:

```
widgetToPageContent :: Widget -> Handler (PageContent url)
data PageContent url = PageContent
  { pageTitle :: Html
  , pageHead  :: HtmlUrl url
  , pageBody  :: HtmlUrl url
  }
```

Это уже ближе к тому, что нам нужно. Теперь у нас есть прямой доступ к HTML, формирующему заголовок и тело страницы, а так же к заголовку. В этот момент, мы можем использовать `Hamlet` для комбинирования их всех вместе в единый документ, наряду с нашим макетом сайта, и мы используем `hamletToRepHtml` для отображения результата `Hamlet` в настоящий HTML, готовый к показу пользователю. Следующий листинг демонстрирует этот процесс.

```
myLayout :: GWidget s MyApp () -> GHandler s MyApp RepHtml
myLayout widget = do
  pc <- widgetToPageContent widget
  hamletToRepHtml [hamlet|
$doctype 5
<html>
  <head>
    <title>#{pageTitle pc}
    <meta charset=utf-8>
    <style>body { font-family: verdana }
    ^{pageHead pc}
  <body>
    <article>
      ^{pageBody pc}
|]

instance Yesod MyApp where
  defaultLayout = myLayout
```

Листинг 5.3: Использование `widgetToPageContent`

¹Имеются в виду темы для визуального оформления. — Прим. перев.

Вы могли заметить, что мы использовали `GWidget` и `GHandler` взамен `Widget` и `Handler`. Это потому, что метод `defaultLayout` может быть вызван подсайтами для гарантии, что они получают такие же стили, как и основной сайт. Поэтому здесь нам необходимо поддерживать гибкость наших типов.

Это всё очень хорошо, но есть один момент, который беспокоит меня: `teг style`. Есть несколько проблем с ним:

- В отличие от `Lucius` или `Cassius`, он не проверяется на корректность во время компиляции.
- Конечно, рассматриваемый пример очень простой, но в чём-то более сложном мы можем получить проблемы с экранированием символов.
- У нас теперь 2 `тега style` взамен одного: один из `myLayout`, а другой генерируется `pageHead`, основанный на стилях, установленных в виджете.

Для решения этой проблемы у нас есть ещё один трюк в запасе: мы в последний момент перед вызовом `widgetToPageContent` скорректируем сам виджет. На самом деле, это очень легко сделать: мы просто опять используем `do`-нотацию, как в листинге 5.4.

```
myLayout :: GWidget s MyApp () -> GHandler s MyApp RepHtml
myLayout widget = do
  pc <- widgetToPageContent $ do
    widget
    toWidget [lucius| body { font-family: verdana } |]
  hamletToRepHtml [hamlet|
$doctype 5
<html>
  <head>
    <title>#{pageTitle pc}
    <meta charset=utf-8>
    ^{pageHead pc}
  <body>
    <article>
      ^{pageBody pc}
|]
```

Листинг 5.4: Корректировка виджета «в последний момент»

5.8. Выводы

Базовый строительный блок каждой страницы — это виджет. Отдельные фрагменты HTML, CSS, и Javascript могут быть преобразованы в виджет с помощью полиморфной функции `toWidget`. Используя `do`-нотацию, вы можете комбинировать эти отдельные виджеты в виджеты большего размера, в конечном счёте содержащие всё наполнение вашей страницы.

Разворачивание полученных виджетов обычно выполняется функцией `defaultLayout`, которая может быть использована для применения единого стиля оформления для всех ваших страниц.

Глава 6

Класс типов Yesod

Каждое ваше Yesod-приложение требует наличия экземпляра класса типов Yesod. До этого мы видели только defaultLayout. В этой главе мы рассмотрим предназначение многих методов класса типов Yesod.

Класс типов Yesod является основным местом для определения настроек приложения. Для всех методов этого класса типов имеются определения по умолчанию, которые обычно делают то, что требуется. Но чтобы создать мощное, соответствующее требованиям заказчика приложение, вам, скорее всего, потребуется переопределить по меньшей мере некоторые из этих методов.

6.1. Рендеринг и разбор URL

Мы уже отмечали, что в Yesod есть возможность автоматического рендеринга типобезопасных URL в текстовое представление, которое может быть вставлено в HTML-страницу. Допустим, у нас есть определение маршрута, которое выглядит следующим образом:

```
mkYesod "MyApp" [parseRoutes|
/some/path SomePathR GET
|
```

Если мы используем SomePath в hamlet-шаблоне, то во что Yesod его отобразит? Yesod всегда пытается сформировать *абсолютные* URL. Это особенно полезно, когда мы начинаем создавать XML-карты сайта и Atom-ленты или рассылать электронную почту. Но чтобы сформировать абсолютный URL, нам необходимо знать доменное имя приложения.

Возможно, вы считаете, что можно получить эту информацию из запроса пользователя, но нам также надо учитывать номера портов. И даже если бы мы брали номер порта из запроса, то как узнать, используется ли HTTP или HTTPS? Даже если бы и *это* нам было известно, то это бы означало, что в зависимости от того, как пользователь делает запрос, получались бы различные URL. К примеру, формировались бы различные URL в зависимости от того, обращается ли пользователь к «example.com»

или к «www.example.com». Для поисковой оптимизации нам бы хотелось иметь возможность объединить эти варианты в одном каноническом URL.

И, наконец, Yesod не делает никаких предположений о том, где вы будете разворачивать ваше приложение. К примеру, у нас может быть по большей части статический сайт (<http://static.example.com/>), но нам также хотелось бы разместить вики, построенную на основе Yesod, по адресу `/wiki/`. Нет надёжного способа получить из приложения адрес, по которому оно развёрнуто. Таким образом, вместо того, чтобы опираться на догадки, вам надо указать Yesod адрес, который является корнем приложения.

В примере с вики вам нужно определить экземпляр Yesod следующим образом:

```
instance Yesod MyWiki where
  approot = ApprootStatic "http://static.example.com/wiki"
```

Заметьте, что на конце нет косой черты. Далее, когда Yesod будет конструировать URL для `SomePathR`, он определит, что относительный путь для `SomePathR` выглядит как `/some/path`, добавит эту строку к `approot` и получит в итоге <http://static.example.com/wiki/some/path>.

Но, если вы помните, большинство наших примеров не использовало метод `approot`. В таких случаях наше приложение получает пустой `approot` или пустую строку. Это подходит для многих случаев, так как многие приложения работают с корня домена. В таком случае маршрут, аналогичный `SomePathR`, будет рендериться как `/some/path`, что является корректной относительной ссылкой в любом месте вашего сайта. Однако для некоторых вариантов использования (ленты Atom/RSS или XML-карты сайтов) требуются абсолютные пути. В таком случае вам следует указать `approot` явно.

Кроме конструктора `ApprootStatic`, уже показанного выше, можно использовать конструкторы `AppMaster` и `AppRequest`. Первый позволяет определить `approot` из значения основы, что позволяет вам, к примеру, загрузить `approot` из файла конфигурации. Второй же даёт возможность использовать ещё и данные запроса для определения `approot`, и с помощью этого вы сможете, например, задавать различные доменные имена в зависимости того, как пользователь обращается к сайту.

Сгенерированный шаблон сайта использует по умолчанию `ApprootMaster` и получает `approot` из конфигурационного файла при запуске. Помимо этого, он загружает различные настройки для сборок разработки, тестирования, подготовки к развёртыванию и боевой сборки. Таким образом, вы с лёгкостью можете проводить тестирование на одном домене, к примеру, на `localhost`, а запускать приложение на другом домене.

6.2. joinPath

Чтобы конвертировать типобезопасный URL в текстовое значение, Yesod использует две вспомогательные функции. Первая — это метод `renderRoute` класса типов `RenderRoute`. Каждый типобезопасный URL является экземпляром этого класса типов. `renderRoute` преобразовывает значение в список компонентов пути. Так `SomePathR` из примера выше будет преобразован в `["some", "path"]`.

На самом деле, `renderRoute` создаёт и компоненты пути, и список параметров строки запроса. Определение `renderRoute` по умолчанию всегда выдаёт пустой список параметров строки запроса. Но это поведение можно переопределить. Одним из вариантов использования этого является статический подсайт, который добавляет хэш содержимого файла в строку параметров для управления кэшированием.

Другой функцией является метод `joinPath` класса типов `Yesod`. Эта функция принимает четыре аргумента: значение основы, корень приложения, список компонентов пути и список параметров строки запроса, а возвращает URL в виде текста. Реализация по умолчанию «делает всё правильно»: отделяет участки пути при помощи косой черты, добавляет корень приложения в начало и дописывает в конец строку запроса.

Если вас устраивает рендеринг URL в текст по умолчанию, то вам не надо изменять эту функцию. Но если же вам требуется модифицировать рендеринг так, чтобы, например, добавлять косую черту в конце URL, то этот метод будет подходящим местом для того, чтобы сделать это.

6.3. `cleanPath`

Противоположной функцией по отношению к `joinPath` является `cleanPath`. Давайте посмотрим, как она используется в процессе диспетчеризации:

1. Путь, запрашиваемый пользователем, разбивается на последовательность компонентов пути.
2. Мы передаём компоненты пути в функцию `cleanPath`.
3. Если `cleanPath` предписывает выполнить перенаправление (возвращая **Left**), то пользователю возвращается ответ с кодом 301. Такое поведение применяется для принудительного использования канонических URL (например, убирая лишние косые черты).
4. В противном случае мы пытаемся выполнить диспетчеризацию, используя результат `cleanPath` (в данном случае **Right**). Если получается, то возвращаем получившийся ответ, иначе возвращаем статус 404.

Такая комбинация даёт подсайтам полный контроль над тем, как будут выглядеть их URL, в то же время позволяя основному сайту иметь изменённые URL. В качестве простого примера давайте посмотрим, как мы можем указать `Yesod`, что нужно всегда добавлять косую черту в конце URL:

```
{-# LANGUAGE TypeFamilies, QuasiQuotes, MultiParamTypeClasses,
    TemplateHaskell, OverloadedStrings #-}

import Yesod
import Network.HTTP.Types (encodePath)
import Blaze.ByteString.Builder.Char.Utf8 (fromText)
import qualified Data.Text as T
```

```

import qualified Data.Text.Encoding as TE
import Control.Arrow ((***))
import Data.Monoid (mappend)

data Slash = Slash

mkYesod "Slash" [parseRoutes|
/ RootR GET
/foo FooR GET
|]

instance Yesod Slash where
  joinPath _ ar pieces' qs' =
    fromText ar 'mappend' encodePath pieces qs
  where
    qs = map (TE.encodeUtf8 *** go) qs'
    go "" = Nothing
    go x = Just $ TE.encodeUtf8 x
    pieces = pieces' ++ ["" ]

    -- Мы хотим сохранять канонические URL. Поэтому, если на конце
    -- URL отсутствует
    -- косая черта, делаем перенаправление. Но пустой набор
    -- компонентов пути остаётся
    -- как есть.
    cleanPath _ [] = Right []
    cleanPath _ s
      | dropWhile (not . T.null) s == ["" ] = -- единственно
        -- возможная пустая строка - последняя
          Right $ init s
      -- Т. к. joinPath добавит недостающую косую черту, мы просто
      -- удаляем пустые компоненты пути.
      | otherwise = Left $ filter (not . T.null) s

getRootR = defaultLayout [whamlet|
<p>
  <a href=@{RootR}>RootR
<p>
  <a href=@{FooR}>FooR
|]

getFooR = getRootR

main = warpDebug 3000 Slash

```

Во-первых, давайте посмотрим на нашу реализацию `joinPath`. Она почти полностью скопирована из реализации по умолчанию в Yesod с одним изменением: мы добавляем дополнительную пустую строку в конец. При обработке компонентов пути пустая строка приведёт к добавлению дополнительной косой черты. Таким образом дополнительная пустая строка приведёт к обязательному наличию косой черты на конце.

`cleanPath` немного сложнее. Сначала мы, как и раньше, проверяем путь на пустоту, и если он пуст, то оставляем как есть. Мы используем `Right`, чтобы указать, что нет необходимости в перенаправлении. Следующий клоз в действительности проверяет 2 возможных случая с URL:

- В пути имеется две косых черты подряд, которые будут выглядеть как пустая строка в середине списка компонентов пути.
- В конце пути нет косой черты, в результате чего последний компонент пути не будет равен пустой строке.

Предполагая, что ни одно из этих условий не выполняется, получаем, что только последний компонент является пустой строкой, и нам нужно выполнить диспетчеризацию по списку всех компонентов, за исключением последнего. С другой стороны, если это не так, то нам необходимо выполнить перенаправление на канонический URL. В этом случае мы удаляем все пустые компоненты и не утруждаем себя добавлением косой черты в конец, т. к. `joinPath` сделает это за нас.

6.4. defaultLayout

Большинство веб-сайтов используют некий общий шаблон для всех своих страниц. Для реализации этого `defaultLayout` является рекомендуемым подходом. Хотя вы и можете так же легко определить свою собственную функцию и вызывать её вместо этого метода, но если вы переопределяете `defaultLayout`, то все страницы, генерируемые Yesod (страницы сообщений об ошибках, страницы аутентификации) автоматически получат заданный стиль.

Переопределение делается довольно просто: мы используем `widgetToPageContent`, чтобы преобразовать `Widget` в заглавие, теги заголовка и тела страницы, а затем используем `hamletToRepHtml` для преобразования шаблона `Hamlet` в `RepHtml`. В `defaultLayout` мы можем даже добавить дополнительные компоненты виджетов, такие как шаблоны `Lucius`. Пример должен сделать это понятным:

```
defaultLayout contents = do
  PageContent title headTags bodyTags <- widgetToPageContent $ \
  do
    toWidget [cassius]
#body
  font-family: sans-serif
#wrapper
  width: 760px
```

```

    margin: 0 auto
  ]
    contents
      hamletToRepHtml [hamlet|
$doctype 5

<html>
  <head>
    <title>#{title}
    ^{headTags}
  <body>
    <div id="wrapper">
      ^{bodyTags}
  ]

```

6.5. getMessage

Хотя мы ещё не рассматривали сессии, мне бы хотелось упомянуть здесь `getMessage`. Обычным приёмом в веб-программировании является установка сообщения в одном обработчике и отображение его в другом. Например, если пользователь отправляет форму с помощью метода POST, вы можете захотеть перенаправить его на другую страницу с сообщением «Форма отправлена».

Такой сценарий известен как Post/Redirect/Get⁰.

Чтобы облегчить реализацию этого подхода, в Yesod есть пара функций: `setMessage`, устанавливающая сообщение в пользовательской сессии, и `getMessage`, достающая это сообщение (и удаляющая его, чтобы оно не появилось второй раз). Рекомендуется размещать результат `getMessage` в `defaultLayout`. Например:

```

defaultLayout contents = do
  PageContent title headTags bodyTags <- widgetToPageContent []
  contents
    mmsg <- getMessage
    hamletToRepHtml [hamlet|
$doctype 5

<html>
  <head>
    <title>#{title}
    ^{headTags}
  <body>
    $maybe msg <- mmsg
    <div #message>#{msg}
    ^{bodyTags}

```

⁰<http://en.wikipedia.org/wiki/Post/Redirect/Get>

```
|]
```

Мы рассмотрим `getMessage/setMessage` более детально, когда будем обсуждать сессии.

6.6. Нестандартные страницы сообщений об ошибках

Одной из отличительных черт профессионального веб-сайта являются должным образом спроектированные страницы сообщений об ошибках. Yesod помогает вам в этом, автоматически используя ваш `defaultLayout` для отображения страниц ошибок. Но иногда вам потребуется пойти чуть дальше. Для этого вам нужно переопределить метод `errorHandler`:

```
errorHandler NotFound = fmap chooseRep $ defaultLayout $ do
    setTitle "Запрашиваемая страница не найдена"
    toWidget [hamlet|
<h1>Страница не найдена
<p>Просим прощения за неудобства, но запрашиваемая страница не может
    быть найдена.
|]
errorHandler other = defaultErrorHandler other
```

Здесь мы указываем нестандартную страницу для ошибки 404. Также мы можем использовать `defaultErrorHandler`, если не хотим писать индивидуальный обработчик для каждого вида ошибки. Из-за ограничений типов нам приходится начинать определение методов с `fmap chooseRep`, но в остальных отношениях вы можете писать обычную функцию-обработчик.

Фактически, вы даже можете возвращать специальные типы ответов, такие как перенаправления:

```
errorHandler NotFound = redirect RootR
errorHandler other = defaultErrorHandler other
```

Несмотря на то, что вы можете сделать это, я не рекомендую пользоваться таким приёмом. Ошибка 404 должна быть ошибкой 404.

6.7. Внешние CSS и Javascript

Функциональность, описываемая в этом разделе, автоматически включается в сгенерированный шаблон сайта. Таким образом, вам самим не нужно беспокоиться о её реализации.

Одним из наиболее мощных и устрашающих методов класса типов Yesod является `addStaticContent`. Вспомним, что `Widget` состоит из нескольких компонентов, включая CSS и Javascript. Каким образом эти CSS/JS попадают в браузер пользователя? По

умолчанию они выдаются в теге `<head>` страницы: в тегах `<style>` и `<script>`, соответственно.

Такое решение является простым, но оно далеко от эффективного. Каждая загрузка страницы будет требовать загрузки всех CSS/JS «с нуля», даже в том случае, когда ничего не изменилось! Что нам на самом деле хотелось бы, так это сохранить это содержимое во внешнем файле, а затем сослаться на него из HTML.

И в этом нам поможет `addStaticContent`. Этот метод принимает три аргумента: расширение файла содержимого (`css` или `js`), MIME-тип содержимого (`text/css` или `text/javascript`) и само содержимое. А возвращает он один из трёх вариантов результата:

Nothing

Сохранение статического файла не выполнено; содержимое встраивается непосредственно в HTML. Это поведение по умолчанию.

Just (Left Text)

Содержимое было сохранено в виде отдельного файла, и используется заданная текстовая ссылка для указания на этот файл.

Just (Right (Route a, Query))

Аналогично, но используется типобезопасный URL с указанием параметров строки запроса.

Вариант с **Left** полезен, если вы хотите сохранять статические файлы на внешнем сервере, например на CDN или сервере с большим объёмом памяти. Чаще используется вариант **Right**, который хорошо связывается со статическим подсайтом. Это рекомендуемый подход для большинства приложений, и он используется в сгенерированном шаблонном сайте по умолчанию.

речь, судя по всему про `yesod-static`, возможно, нужно применение

Возможно, вы спросите: если это рекомендуемый подход, то почему он не является вариантом по умолчанию? Проблема заключается в том, что он делает несколько предположений, которые не всегда соблюдаются: наличие в вашем приложении статического сайта, а также местоположение ваших статических файлов.

Включённый в сгенерированный шаблон сайта `addStaticContent` облегчает вам жизнь, выполняя ряд разумных действий:

- Он автоматически сжимает ваш Javascript используя пакет `hjsmin`¹
- Он именуем выходные файлы на основе хэша содержимого файлов. Это значит, что вы можете устанавливать заголовки кэширования на дату в будущем, не боясь получить устаревшее содержимое.
- Также, так как имена файлов основаны на хэшах, вы можете быть уверены, что файл не потребуется перезаписывать поверх существующего. Сгенерированный шаблонный код автоматически проверяет существование такого файла и исключает затратные операции работы с диском, если операция записи не является необходимой.

¹<http://hackage.haskell.org/package/hjsmin>

6.8. Оптимизация использования статических файлов

Google рекомендует использовать одну важную оптимизацию: отдавать статические файлы с отдельного домена². Преимущество такого подхода состоит в том, что куки, установленные для вашего домена, не будут отправляться при запросе статических файлов, позволяя сэкономить немного пропускной способности.

Для осуществления этого у нас есть метод `urlRenderOverride`. Этот метод перехватывает обычный рендеринг URL и выставляет специальное значение для определённых маршрутов. Например, в сгенерированном шаблонном коде этот метод реализуется так:

```
urlRenderOverride y (StaticR s) =
  Just $ uncurry (joinPath y (Settings.staticRoot $ settings []
    y)) $ renderRoute s
urlRenderOverride _ _ = Nothing
```

Это означает, что статические маршруты обслуживаются со специального «статического корня», в качестве значения которого вы можете указать другой домен при конфигурировании. Это является прекрасным примером мощности и гибкости типобезопасных URL: при помощи одной строки кода вы можете изменить рендеринг статических маршрутов во всех ваших обработчиках.

6.9. Аутентификация/авторизация

Для простых приложений проверка прав доступа в каждой функции-обработчике может быть простым и удобным решением. Однако такой подход не очень хорошо масштабируется. В конце концов вам захочется использовать более декларативный подход. Многие системы определяют свои списки контроля доступа, специальные файлы конфигурации и другие «фокусы». В Yesod это делается при помощи старого доброго Haskell. Для этого используются следующие три метода:

`isWriteRequest`

Определяет, является ли текущий запрос операцией чтения или записи. По умолчанию Yesod следует RESTful-принципам и предполагает, что запросы GET, HEAD, OPTIONS, и TRACE предназначены только для чтения, тогда как остальные могут выполнять запись.

`isAuthorized`

Этот метод принимает в качестве параметров маршрут (то есть, типобезопасный URL) и булево значение, указывающее на то, является ли запрос запросом записи. А возвращает он `AuthResult`, который может быть одним из трёх значений:

- `Authorized`

²<http://code.google.com/speed/page-speed/docs/request.html#ServeFromCookielessDomain>

- `AuthenticationRequired`
- `Unauthorized`

По умолчанию он возвращает `Authorized` для всех запросов.

`authRoute`

Если `isAuthorized` возвращает `AuthenticationRequired`, то метод выполняет перенаправление по указанному маршруту. Если маршрут не указан (поведение по умолчанию), возвращается ошибка 403 «`Permission Denied`».

Эти методы прекрасно взаимодействуют с пакетом `yesod-auth`³, который используется сгенерированным шаблонным сайтом, предоставляя целый набор вариантов аутентификации, таких как `OpenID`, `BrowserID`, электронная почта, имя пользователя и `Twitter`. Мы рассмотрим более конкретные примеры в главе об аутентификации.

6.10. Некоторые простые настройки

Не всё в классе типов `Yesod` является сложным. Некоторые методы представляют собой простые функции. Давайте пройдемся по их списку.

6.10.1. `encryptKey`

`Yesod` использует сессии на стороне клиента, которые хранятся в зашифрованных, криптографически хэшированных куках. Но только если вы указали ключ шифрования. Если функция возвращает **Nothing**, то сессии отключаются. Это может быть полезной оптимизацией для сайтов, которым не требуются сессии, так как это отключает пару операций шифрования/расшифрования для каждого запроса.

Комбинация шифрования и хэширования гарантирует два свойства: данные сессии защищены от несанкционированного доступа и непрозрачны. Шифрование без хэширования позволило бы пользователю произвольно менять содержимое куки, после чего они по-прежнему продолжили бы приниматься сервером, а хэширование без шифрования дало бы возможность просматривать данные.

6.10.2. `clientSessionDuration`

Задаёт время, которое будет длиться сессия. По умолчанию — 2 часа.

6.10.3. `sessionIpAddress`

По умолчанию сессии привязаны к IP-адресу. Если ваши пользователи находятся за прокси-сервером, то это может привести к проблемам, если их IP вдруг изменится. Данная настройка позволяет отключить эту меру безопасности.

³<http://hackage.haskell.org/package/yesod-auth>

6.10.4. cookiePath

Задаёт путь в текущем домене, для которого устанавливаются куки. По умолчанию — «/», и это в большинстве случаев будет правильным вариантом. Исключением может быть сценарий, когда приложение работает на подпути в домене (как в примере с вики, приведённом ранее).

6.10.5. maximumContentLength

Чтобы предотвратить DoS-атаки⁴, Yesod ограничивает максимальный размер тела запроса. Иногда вам потребуется поднять этот максимум для некоторых маршрутов (к примеру для страниц загрузки файлов). Данный метод позволяет вам сделать это.

6.10.6. уернореJs

Здесь мы можете указать местоположение JavaScript-библиотеки уерноре⁵. Если это местоположение задано, то уерноре будет асинхронно загружать JavaScript для ваших страниц.

6.11. Выводы

Класс типов Yesod включает в себя набор переопределяемых методов, которые позволяют конфигурировать ваше приложение. Все они не обязательны и имеют разумные реализации по умолчанию. Используя встроенные в Yesod конструкции, такие как `defaultLayout` и `getMessage`, вы получите единообразный внешний вид для всего вашего сайта, включая страницы, автоматически сгенерированные Yesod, такие как страницы сообщений об ошибках и страницы аутентификации.

Мы не рассмотрели полностью все методы класса типов Yesod в этой главе. Для просмотра полного списка доступных методов обратитесь к Haddock-документации.

⁴Denial of Service — отказ в обслуживании

⁵<http://yeepnorejs.com/>

Глава 7

Маршрутизация URL и обработчики

Если рассматривать Yesod как веб-фреймворк, построенный по шаблону «Модель–Вид–Контроллер» (MVC, «Model–View–Controller»), то маршрутизация URL и обработчики запросов занимают в этом шаблоне место «Контроллера». Для сравнения опишем два других подхода к маршрутизации, используемые в других средах веб-разработки:

- Используется диспетчеризация обработчиков, основанная на имени файла. Так, например, работают PHP и ASP.
- Используется централизованная функция, которая обрабатывает маршруты, опираясь на регулярные выражения. Такому подходу следуют Django и Rails.

Yesod в основном следует второму подходу, однако имеются и значительные отличия. Вместо использования регулярных выражений Yesod сопоставляет компоненты маршрута. Вместо однонаправленного отображения маршрут–обработчик Yesod использует промежуточный тип данных (называемый тип данных маршрута, или типобезопасный URL) и создаёт функции для двустороннего преобразования.

Ручная реализация этой более изощрённой системы утомительна и подвержена ошибкам. Поэтому Yesod определяет предметно-ориентированный язык (Domain Specific Language, DSL) для описания маршрутов и предоставляет функции Template Haskell для преобразования кода на этом языке в код на Haskell. В настоящей главе разъясняется синтаксис объявлений маршрута, даётся некоторое представление о том, какой код генерируется за вас, и описывается взаимодействие между маршрутизацией и функциями обработки запросов.

7.1. Синтаксис записи маршрута

Вместо того, чтобы пытаться втиснуть объявления маршрутов в рамки существующего синтаксиса, Yesod использует упрощённый синтаксис, разработанный специально для маршрутов. Преимущество такого подхода — код получается не только проще в

написании, но и достаточно простым для того, чтобы люди, не имеющие опыта работы с Yesod, могли его прочитать и понять схему сайта вашего приложения.

Простой пример синтаксиса:

```

/           RootR     GET
/blog      BlogR     GET POST
/blog/#BlogId BlogPostR GET POST

/static    StaticR   Static getStatic

```

В следующих нескольких разделах будет подробно описано, что происходит при объявлении маршрута.

7.1.1. Компоненты пути URL

Одно из первых действий, совершаемых Yesod при получении запроса, — разбиение запрошенного пути на компоненты. Компоненты разделяются символами косой черты (/). Например:

```

toPieces "/" = []
toPieces "/foo/bar/baz/" = ["foo", "bar", "baz", ""]

```

Вы можете заметить, что некоторые забавные вещи происходят с завершающим символом косой черты, с двойным символом косой черты («/foo//bar//») и ещё с некоторыми другими комбинациями. Yesod рассчитывает на получение канонических адресов URL; если кто-то запросит URL с завершающим символом косой черты или с двойным символом косой черты, он будет автоматически переадресован на каноническую версию адреса. Это гарантирует, что у вас одному URL соответствует один ресурс, и может помочь с рейтингами в поисковых системах.

Для вас это означает, что нет необходимости беспокоиться о точной структуре URL: вы можете спокойно думать о компонентах маршрута, а Yesod автоматически выполнит вставку символов косой черты и экранирование проблемных символов.

Если, кстати, вы захотите более тонко настраивать, как маршруты разбиваются на компоненты и собираются обратно, вам стоит взглянуть на методы `cleanPath` и `joinPath` в главе 6 о классе типов `Yesod`.

Типы компонентов

Когда вы объявляете маршруты, у вас в распоряжении имеются три типа компонентов:

Статический

Простая строка, с которой в точности должен совпадать компонент URL.

Динамический одинарный

Одиночный компонент (т.е. между двумя символами косой черты), который может представлять собой значение, задаваемое пользователем. Это основной способ получения дополнительных данных от пользователя при запросе страницы.

Определение таких компонентов начинаются с символа решётки (#), за которым следует тип данных. Тип данных должен быть экземпляром `PathPiece`.

Динамический множественный

Аналогичен предыдущему, но может соответствовать множеству компонентов URL. Это всегда должен быть последний компонент в шаблоне ресурса. Задаётся символом звёздочки (*), за которым следует тип данных, который должен быть экземпляром `PathMultiPiece`. Множественные компоненты не столь распространены, как предыдущие два, хотя они очень важны для реализации таких возможностей, как статические деревья, описывающие структуру файла или вики с произвольными иерархиями.

Давайте взглянем на некоторые стандартные типы шаблонов ресурса, которые вы могли бы захотеть написать. Начнём с простого: корень приложения будет просто `/`. Также вы, вероятно, захотите разместить страницу с часто задаваемыми вопросами (FAQ) по адресу `/page/faq`.

Теперь давайте представим, что вы собираетесь сделать сайт для вычисления чисел Фибоначчи. Ваши URL могут выглядеть как `/fib/#Int`. Но тут есть небольшая проблема: мы не хотим, чтобы в наше приложение можно было передавать отрицательные числа или ноль. К счастью, система типов может нас защитить:

```
newtype Natural = Natural Int
instance PathPiece Natural where
  toPathPiece (Natural i) = T.pack $ show i
  fromPathPiece s =
    case reads $ T.unpack s of
      (i, ""):_
        | i < 1 -> Nothing
        | otherwise -> Just $ Natural i
      [] -> Nothing
```

В строке 1 мы определяем для `Int` изоморфный тип, который защитит нас от некорректного ввода. Мы видим, что `PathPiece` — это класс типов с двумя методами. `toPathPiece` просто конвертирует в `Text` и ничего больше. `fromPathPiece` пытается преобразовать `Text` в наш тип данных, возвращая `Nothing`, если преобразование невозможно. Используя этот тип данных, мы можем быть уверены, что нашей функции-обработчику будут передаваться только натуральные числа, позволяя нам в очередной раз использовать систему типов для преодоления трудностей, связанных с граничными условиями.

В реальном приложении мы бы ещё предпочли удостовериться, что никогда случайно не создадим некорректное значение `Natural` внутри нашего приложения. Для этого мы могли бы использовать подход наподобие умных конструкторов (smart constructors¹). Но для нашего примера мы решили оставить код простым.

¹http://www.haskell.org/haskellwiki/Smart_constructors

Определение `PathMultiPiece` столь же простое. Допустим, что мы хотим сделать вики с иерархией не менее, чем два уровня; тогда мы могли бы определить тип данных как:

```
data Page = Page Text Text [Text] -- 2 или больше
instance PathMultiPiece Page where
  toPathMultiPiece (Page x y z) = x : y : z
  fromPathMultiPiece (x:y:z) = Just $ Page x y z
  fromPathMultiPiece _ = Nothing
```

7.1.2. Имя ресурса

С каждым шаблоном ресурса связано имя. Оно станет именем конструктора типа данных для типобезопасного URL, связанного с вашим приложением, поэтому оно должно начинаться с прописной буквы. Также имена ресурсов принято завершать прописной буквой «R». Это не обязательное требование, просто общепринятая практика.

Конкретное определение нашего конструктора зависит от шаблона ресурса, к которому оно относится. Все типы данных, которые включены в одинарные и множественные компоненты шаблона, становятся аргументами типа данных. Это даёт нам однозначное соответствие между нашим типом для типобезопасных URL и корректными для нашего приложения URL.

Это не обязательно означает, что каждое значение типа соответствует работающей странице, лишь означает, что значение типа является потенциально корректным URL. Например, значение `PersonR "Michael"` может не разрешиться в корректную страницу, если в базе данных нет записи для "Michael".

Давайте теперь обратимся к реальным примерам. Если бы у вас были такие шаблоны ресурсов: `/person/#Text` с именем `PersonR`; `/year/#Int` с именем `YearR` и `/page/faq` с именем `FaqR`, то ваш тип данных для маршрутов выглядел бы приблизительно так:

```
data MyRoute = PersonR Text
              | YearR Int
              | FaqR
```

Если пользователь запросит относительный URL `/year/2009`, Yesod преобразует его в значение `YearR 2009`. `/person/Michael` станет `PersonR "Michael"` и `/page/faq` станет `FaqR`. С другой стороны, `/year/two-thousand-nine`, `/person/michael/snooman` и `/page/FAQ` привели бы к получению 404 ошибок даже без обращения к вашему коду.

7.2. Спецификация обработчика

Последний кусочек головоломки объявления ваших ресурсов — описание того, как они будут обрабатываться. Yesod предлагает три варианта:

- у заданного маршрута единственный обработчик для всех методов запроса;
- у заданного маршрута для каждого метода отдельный обработчик. Для неуказанных методов будет генерироваться ответ с кодом 405 «Метод не поддерживается» (405 Method Not Allowed);
- передача управления на подсайт.

В первых двух случаях спецификация обработчика очень проста. Для единственного обработчика это будет просто строка с шаблоном ресурса и его именем, например `/page/faq FaqR`. В таком случае функция-обработчик должна иметь имя `handleFaqR`.

В случае отдельного обработчика для каждого метода спецификация аналогична, только добавляется список доступных методов запросов. Методы запросов должны быть записаны прописными буквами. Например, `/person/#String PersonR GET POST DELETE`. В данном случае вам потребуется определить три функции-обработчика: `getPersonR`, `postPersonR` и `deletePersonR`.

Подсайты — очень полезная, но сложная тема в Yesod. Мы рассмотрим создание подсайтов позже, в главе «Создание дочернего сайта», но использовать их не так уж и сложно. Самый часто используемый подсайт — статический сайт, который раздаёт статические файлы для вашего приложения. Для того, чтобы раздавать статические файлы из `/static`, вам нужно описать ресурс вида:

```
/static StaticR Static getStatic
```

В этой строке, `/static` просто говорит о том, откуда в вашей структуре URL отдавать статические файлы. Нет никакой магии в слове «static» (статический), вы легко можете заменить его на `/my/non-dynamic/files`.

Следующее слово, `StaticR`, присваивает ресурсу имя. Следующие два слова указывают на то, что мы используем подсайт. `Static` — это имя типа-основания подсайта, а `getStatic` — функция, которая получает значение типа `Static` из значения типа-основания вашего основного сайта.

Давайте пока не будем углубляться в подробности, касающиеся подсайтов. Мы рассмотрим подробнее статический сайт в главе 15 о каркасе сайта.

7.3. Диспетчеризация

Как только вы определили свои маршруты, Yesod позаботится за вас обо всех утомительных деталях диспетчеризации URL. Вам нужно только удостовериться, что вы предоставили соответствующие функции обработки. Для маршрутов на подсайты вам не требуется писать какие-либо функции обработки, но для двух других типов обработчиков это сделать необходимо. Выше мы уже упомянули правило именования: `MyHandlerR GET` становится `getMyHandlerR`, `MyOtherHandlerR` становится `handleMyOtherHandlerR`. Теперь нам нужно определиться с сигнатурой типа.

7.3.1. Возвращаемый тип

Давайте посмотрим на простую функцию обработки:

```
mkYesod "Simple" [parseRoutes|
/ HomeR GET
|]

getHomeR :: Handler RepHtml
getHomeR = defaultLayout [whamlet|<h1>This is simple
|]
```

Посмотрите на сигнатуру типа для `getHomeR`. Первый компонент — это `Handler`. `Handler` — это специальная монада, в которой живут все функции обработки. Она предоставляет доступ к информации о запросе, позволяет вам осуществлять переадресацию, и ещё много других вещей, которые мы рассмотрим вскоре.

Затем идёт `RepHtml`. Когда мы будем рассматривать представления в главе «REST-содержимое», мы исследуем подробнее вопрос «*почему это так*»; сейчас нас просто интересует вопрос «*как это сделать*».

Как вы, возможно, догадались, `RepHtml` — это тип данных для ответов в формате HTML. И также вы можете предположить, что веб-сайтам приходится возвращать ответы не только в HTML. CSS, Javascript, изображения, XML тоже являются необходимыми для веб-сайта. Поэтому возвращаемый тип функции обработки может быть любым экземпляром `HasReps`.

`HasReps` — это мощная концепция, которая позволяет Yesod автоматически выбирать правильное представление ваших данных, основываясь на запросе клиента. Пока мы сосредоточимся на простых экземплярах, наподобие `RepHtml`, которые предоставляют только один формат представления.

7.3.2. Аргументы

Не все маршруты столь же просты, как `HomeR`, который мы определили чуть выше. Возьмём для примера наш маршрут `PersonR` из предыдущих разделов. Имя человека необходимо передавать в функцию обработки. Переход очень прост и, надеюсь, интуитивно понятен. Пример:

```
mkYesod "Args" [parseRoutes|
/person/#Text PersonR GET
/year/#Integer/month/#Text/day/#Int DateR
/wiki/*Texts WikiR GET
|]

getPersonR :: Text -> Handler RepHtml
getPersonR name = defaultLayout [whamlet|<h1>Hello #{name}!|]

handleDateR :: Integer -> Text -> Int -> Handler RepPlain -- []
    text/plain
handleDateR year month day =
    return $ RepPlain $ toContent $
```

```

    T.concat [month, " ", T.pack $ show day, ", ", T.pack $ show
year]

getWikiR :: [Text] -> Handler RepPlain
getWikiR = return . RepPlain . toContent . T.unwords

```

Аргументы имеют типы динамических компонент каждого маршрута, в порядке указания. Также обратите внимание, как мы можем использовать и `RepHtml`, и `RepPlain`.

7.4. Монада Handler

подавляющая часть кода, который вы пишете в Yesod, находится внутри монады `Handler`. Если вы подходите к этому с точки зрения шаблона «Модель–Вид–Контроллер», ваш код здесь `Handler` — это «Контроллер». Перечислим несколько важных фактов о `Handler`, которые следует знать:

- Это экземпляр `MonadIO`, поэтому вы можете использовать любое действие **IO** в ваших обработчиках с помощью `liftIO`. Кстати, для вашего удобства `liftIO` экспортируется модулем `Yesod`.
- Как и `Widget`, `Handler` — это псевдотрансформатор монад (`fake-monad-transformer`). Он обёртывает монаду `ResourceT IO`. Мы подробно обсудим это в приложении о `conduit`, а сейчас только скажем, что это позволяет вам безопасно выделять ресурсы.
- Под «псевдо», я имею в виду, что вы не можете использовать стандартную функцию `lift`, предоставляемую пакетом `transformers`², вы должны использовать замещающую функцию из `Yesod` (как и в случае с виджетами).
- `Handler` — это просто синоним типа для `GHandler`. `GHandler` позволяет вам точно указать, какой подсайт и основной сайт вы используете. Синоним `Handler` сообщает, что подсайт и основной сайт принадлежат типу вашего приложения.
- `Handler` предоставляет большое число возможностей, среди которых:
 - предоставление информации о запросе;
 - хранение списка дополнительных полей заголовка ответа, которые вы добавили;
 - изменения пользовательской сессии;
 - сокращённая обработка запросов для переадресации, отправки статических файлов или сообщения об ошибках.

В оставшейся части главы приведено краткое введение в некоторые самые распространённые функции, находящиеся в монаде `Handler`. Я специально не описываю функции для работы с сессиями; они будут рассмотрены в отдельной главе.

²<http://hackage.haskell.org/package/transformers>

7.4.1. Информация о приложении

Есть набор функций, которые возвращают информацию о вашем приложении в целом и не дают никакой информации об отдельных запросах. Среди них:

getYesod

Возвращает значение основы (foundation value) вашего приложения. Если вы храните значения настроек в основе, то вы, вероятно, будете часто использовать эту функцию.

getYesodSub

Возвращает значение основы для подсайта. Возвращает то же значение, что и `getYesod`, если только вы не работаете в подсайте.

getUrlRender

Возвращает функцию рендеринга URL, которая преобразует типобезопасный URL в `Text`. В большинстве случаев, как и при использовании `Hamlet`, `Yesod` сам вызывает эту функцию за вас, но иногда может потребоваться вызвать эту функцию напрямую.

getUrlRenderParams

Вариант `getUrlRender`, который преобразует и типобезопасный URL, и список параметров из строки запроса. Это функция выполняет необходимое кодирование символов в URL (percent-encoding).

7.4.2. Информация о запросе

Информация о текущем запросе, которую вы обычно желаете получить, — это запрошенный путь, параметры из строки запроса и данные форм, отправленные на сервер методом `POST`. С первым пунктом имеет дело маршрутизация, как описано выше. С оставшимися двумя лучше всего работать, используя модуль работы с формами.

Тем не менее, вам иногда требуется получить данные в менее обработанном виде. Для этой цели `Yesod` предоставляет тип данных `Request` вместе с функцией `getRequest` для его получения. Это даёт вам доступ к полному набору параметров `GET`-запроса, куки и списку предпочитаемых языков. Есть несколько удобных функций для упрощения поиска, такие как `lookupGetParam`, `lookupCookie` и `languages`. Для прямого доступа к данным `POST`-запроса следует использовать `runRequest`.

Если вам требуются ещё более сырые данные, такие как заголовки запросов, вы можете использовать `waiRequest` для доступа к значению запроса Интерфейса Веб-приложения (Web Application Interface, WAI). За деталями обращайтесь к приложению о WAI.

7.4.3. Сокращённая обработка запроса

Описанные ниже функции немедленно прекращают выполнение обработчика и возвращают результат пользователю.

redirect

Отправляет пользователю ответ, предписывающий выполнить переадресацию (код 303). Если вы хотите использовать другой код ответа (например, 301 «Постоянная переадресация»), вы можете воспользоваться `redirectWith`.

Yesod использует ответ 303 для клиентов, использующих протокол HTTP/1.1, и ответ 302 для клиентов, использующих HTTP/1.0. Вы можете прочитать об этой ужасной истории в спецификации протокола HTTP.

notFound

Возвращает ответ 404. Это может быть полезно, если пользователь запросил из базы данных несуществующее значение.

permissionDenied

Возвращает ответ 403 с указанным сообщением об ошибке.

invalidArgs

Ответ 400 со списком некорректных аргументов.

sendFile

Отправляет файл из файловой системы с указанным типом контента. Это предпочтительный способ отправки статических файлов, так как лежащий в основе обработчик WAI может оказаться в состоянии оптимизировать отправку до системного вызова `sendfile`. Использование `readFile` для отправки статических файлов не является обязательным.

sendResponse

Отправить обычный `HasReps` ответ с кодом 200. На самом деле, определён для удобства, чтобы вы могли прервать глубоко вложенный код немедленным ответом.

sendWaiResponse

Иногда вам нужно спуститься на нижний уровень и отправить сырой ответ WAI. Это может быть особенно полезным для создания потоковых ответов или техник наподобие событий, отправляемых сервером.

7.4.4. Заголовки ответа**setCookie**

Установить куки клиенту. Вместо того, чтобы принимать дату истечения срока жизни, эта функция принимает продолжительность жизни куки в минутах. Помните, вы не сможете увидеть это куки, используя `lookupCookie`, до *следующего* запроса.

deleteCookie

Предписывает клиенту удалить куки. Опять же, `lookupCookie` не увидит изменений до следующего запроса.

setHeader

Устанавливает указанный заголовок запроса.

setLanguage

Устанавливает предпочитаемый пользователем язык, который будет показываться в ответе функции `languages`.

cacheSeconds

Устанавливает заголовок `Cache-Control` равным количеству секунд, на которое ответ может быть кэширован. Это может быть в частности полезно, если вы используете `varnish`³ на своём сервере.

neverExpires

Устанавливает заголовок `Expires` равным 2037-ому году. Вы можете использовать это для контента, время жизни которого неограничено, например, когда маршрут запроса имеет связанное с ним значение хэша.

alreadyExpired

Устанавливает заголовок `Expires` равным некоторому моменту в прошлом.

expiresAt

Устанавливает заголовок `Expires` равным заданному значению даты/времени.

7.5. Выводы

Маршрутизация и диспетчеризация — это, пожалуй, основная часть `Yesod`: здесь определяются наши типобезопасные `URL`, и большая часть нашего кода пишется внутри монады `Handler`. В этой главе описаны некоторые самые важные и центральные концепции `Yesod`, поэтому важно, чтобы вы как следует усвоили её.

Также в главе намечено несколько более сложных тем `Yesod`, которые будут охвачены далее. Но вы уже должны быть в состоянии реализовать очень сложные веб-приложения, используя только те знания, которые вы уже получили.

³<http://www.varnish-cache.org/>

Глава 8

ФОРМЫ

Я уже упоминал граничное условие: если данные приходят в приложение или покидают его, мы должны их проверять. Вероятно, наиболее сложная проверка происходит в формах. Программировать формы не так просто; в идеальных условиях мы хотели бы иметь решение, которое может следующее:

- гарантировать, что данные корректны;
- преобразовывать строковые данные формы в типы данных Haskell;
- генерировать код HTML для отображения формы;
- генерировать Javascript, выполняющий валидацию на стороне клиента и предоставляющий более дружелюбные пользователю виджеты, например, для выбора даты;
- строить более сложные формы, объединяя вместе более простые;
- автоматически присваивать полям имена, которые гарантированно будут уникальными.

Пакет `yesod-form`¹ предоставляет все эти возможности с помощью простого, декларативного API. Он построен на базе виджетов `Yesod` для упрощения стиливого оформления форм и применения Javascript надлежащим образом. И, как и везде в `Yesod`, для обеспечения корректной работы используется система типов Haskell.

8.1. Краткий обзор

```
{-# LANGUAGE QuasiQuotes, TemplateHaskell, MultiParamTypeClasses,
    OverloadedStrings, TypeFamilies #-}
import Yesod
```

¹<http://hackage.haskell.org/package/yesod-form>

```

import Yesod.Form.Jquery
import Data.Time (Day)
import Data.Text (Text)
import Control.Applicative ((<$>), (<*>))

data Synopsis = Synopsis

mkYesod "Synopsis" [parseRoutes]
/ RootR GET
/person PersonR POST
[]

instance Yesod Synopsis

-- Указывает приложению использовать стандартные английские сообщения
-- Если вам нужна интернационализация, вы можете задать функцию []
  перевода
instance RenderMessage Synopsis FormMessage where
  renderMessage _ _ = defaultMessage

-- Укажите также, где найти библиотеки jQuery. Мы будем использовать []
  значение
-- по умолчанию, указывающее на Google CDN
instance YesodJquery Synopsis

-- Тип данных, который мы хотим получить из формы
data Person = Person
  { personName :: Text
  , personBirthday :: Day
  , personFavoriteColor :: Maybe Text
  , personEmail :: Text
  , personWebsite :: Maybe Text
  }
deriving Show

-- Объявление формы. Сигнатура типа несколько пугающая, но вот её []
  обзор:
--
-- * Параметр Html используется для кодирования некоторой []
  дополнительной
-- информации. См. обсуждение runFormGet и runFormPost ниже для
-- дополнительного объяснения
--
-- * Как обычно, у нас есть типы подсайта и главного сайта
--
-- * FormResult может находиться в трёх состояниях: FormMissing (нет

```



```

-- доступных данных), FormFailure (некорректные данные) и FormSuccess
--
-- * Widget --- отображаемая форма для вставки на страницу
--
-- Обратите внимание, что каркас сайта предоставляет удобный синоним []
  типа
-- Form, так что наша сигнатура может быть переписана как:
-- > personForm :: Form Person
--
-- Для целей обучения лучше видеть полную версию
personForm :: Html -> MForm Synopsis Synopsis (FormResult Person, []
  Widget)
personForm = renderDivs $ Person
  <$> areq textField "Имя" Nothing
  <*> areq (jqueryDayField def
    { jdsChangeYear = True -- выпадающий список
      , jdsYearRange = "1900:-5" -- от 1900 года до пятилетней []
        давности
      }) "Дата_рождения" Nothing
  <*> aopt textField "Любимый_цвет" Nothing
  <*> areq emailField "Адрес_Email" Nothing
  <*> aopt urlField "Сайт" Nothing

-- Обработчик GET-запроса отображает форму
getRootR :: Handler RepHtml
getRootR = do
  -- генерируем форму, которую будем отображать
  (widget, enctype) <- generateFormPost personForm
  defaultLayout [whamlet|
<p>Сгенерированный виджет включает только содержимое формы, без тега []
  form.
Поэтому...
<form method=post action=@{PersonR} enctype=#{enctype}>
  ^{widget}
  <p>В виджете также нет кнопки отправки формы.
  <input type=submit>
|]

-- Обработчик POST-запроса обрабатывает форму. Если обработка успешно
-- завершилась, он отображает данные переданного человека. В ином []
  случае -- снова
-- форму с сообщениями об ошибке.
postPersonR :: Handler RepHtml
postPersonR = do
  ((result, widget), enctype) <- runFormPost personForm
  case result of

```

```

    FormSuccess person -> defaultLayout [whamlet|<p>#{show person}|]
    _ -> defaultLayout [whamlet|
<p>Некорректный ввод, попробуйте ещё раз.
<form method=post action=@{PersonR} enctype=#{enctype}>
  ^{widget}
  <input type=submit>
|]

main :: IO ()
main = warpDebug 3000 Synopsis

```

08/synopsis.hs

8.2. Виды форм

Погружение в непосредственное рассмотрение типов нам следует начать с обзора различных видов форм. Их три категории:

Апplikативные

Наиболее широко используемые (см. пример выше). Апplikативные формы позволяют нам объединять сообщения об ошибках друг с другом, сохраняя при этом декларативный высокоуровневый подход. Более детальную информацию об апplikативном подходе можно почерпнуть в Haskell-вики².

Монадические

Более мощная альтернатива апplikативным. Гибкость достигается за счёт несколько большей многословности. Бывают полезны, если необходимо создать форму, которая не укладывается в стандартный двухстолбцовый формат.

Формы для ввода

Используются только для получения ввода. Не генерируют никакого HTML для получения ввода пользователя. Полезны для взаимодействия с уже существующими формами.

Кроме того существует ряд различных переменных для каждого вида формы и поля, которые вам захочется установить:

- Обязательное поле или нет?
- Данные должны передаваться методом GET или POST?
- Есть у поля значение по умолчанию или нет?

²http://www.haskell.org/haskellwiki/Applicative_functor

Главная цель — минимизировать число определений полей и позволить им работать в максимально возможном количестве контекстов. Одним из результатов этого явилось то, что мы пришли к добавлению нескольких дополнительных параметров для каждого поля. В обзоре вы, наверное, заметили слово `areq` или дополнительный параметр **Nothing**. В рамках данной главы мы обсудим зачем они нужны, а пока примите себе, что сделав эти параметры явными, мы получили возможность повторно использовать отдельные поля (например, `intField`³) большим количеством различных способов.

Краткое замечание по поводу именования. Каждая форма имеет однобуквенный префикс (A, M или I), который используется в нескольких местах, как например в `MForm`. Мы также будем использовать `req` и `opt` для обозначения обязательных (`required`) и необязательных (`optional`) полей соответственно. В итоге, обязательное аппликативное поле мы создаём с помощью `areq`, а необязательное поле для ввода данных — с помощью `iort`.

8.3. Типы

Модуль `Yesod.Form.Types`⁴ определяет несколько типов. Давайте начнём со вспомогательных:

Enctype

Тип кодировки: либо `UrlEncoded`, либо `Multipart`. Этот тип данных объявляет экземпляр для `toHtml`, так что вы можете его использовать непосредственно в шаблонах `hamlet`.

Env

Отображает имя параметра в список значений.

FileEnv

Отображает имя параметра в соответствующий загруженный файл.

Ints

Как было сказано во введении, `yesod-form` автоматически присваивает уникальное имя каждому полю. `Ints` используется для отслеживания следующего номера для присваивания.

FormResult

Имеет одно из трёх возможных состояний: `FormMissing` — если данные не были переданы, `FormFailure` — если произошла ошибка при разборе формы (например, не заполнено обязательное поле или содержимое поля некорректно) или `FormSuccess`, когда всё прошло гладко.

Три следующих типа данных используются для определения отдельных полей.

Поле — это отдельная порция информации, например, число, строка или адрес электронной почты. Поля собирают вместе для построения форм.

³<http://hackage.haskell.org/packages/archive/yesod-form/latest/doc/html/Yesod-Form-Fields.html#v:intField>

⁴<http://hackage.haskell.org/packages/archive/yesod-form/latest/doc/html/Yesod-Form.Types.html>

Field

Определяют два вида функциональности: как преобразовать текстовый ввод пользователя в значение языка Haskell и как создавать виджет, который будет показан пользователю. `yesod-form` определяет ряд отдельных полей в модуле `Yesod.Form.Fields`⁵.

FieldSettings

Основная информация о том, как поле следует отображать: отображаемое имя, необязательная подсказка, и, возможно, явно заданные атрибуты `id` и `name`. (Если ничего не указано, то они генерируются автоматически.)

`FieldSettings` предоставляет экземпляр `IsString`, так что, когда вам понадобится указать значение типа `FieldSettings`, вы можете обойтись просто строковым литералом. Как мы и сделали во введении.

FieldView

Промежуточный формат, содержащий большой объём информации об отображении поля. Едва ли будет использоваться напрямую. Детали мы рассмотрим ниже.

И, наконец, мы добрались до важного: до самих форм. Есть три типа данных для форм: `MForm` для монадических, `AForm` для аппликативных и `IForm` (определённая в `IForm`⁶) для форм ввода данных.

`MForm` на самом деле является синонимом типа для стека монад, который предоставляет следующие возможности:

- Монада `Reader` предоставляет нам параметры (`Env` и `FileEnv`), аргумент основного сайта (`master site`) и список языков, которые поддерживаются пользователем. Последние два используются для интернационализации (подробнее об этом ниже).
- Монада `Writer` отслеживает `Enctype`. Для формы значение всегда будет `UrlEncoded`, за исключением случая, когда присутствует поле для загрузки файла. Тогда будет использовано значение `Multipart`.
- Монада `State` хранит `Ints` для отслеживания следующего генерируемого уникального имени.

Форма `AForm` устроена похожим образом. Однако, есть несколько важных различий:

- Она генерирует список `FieldViews`. Это позволяет нам оперировать абстракцией отображения форм, выбирая соответствующую функцию для отрисовки её на странице в самом конце. Во введении мы использовали `renderDivs`, которая создаёт связку тегов `div`. Другим вариантом может быть `renderTable`.

⁵<http://hackage.haskell.org/packages/archive/yesod-form/latest/doc/html/Yesod.Form.Fields.html>

⁶<http://hackage.haskell.org/packages/archive/yesod-form/latest/doc/html/Yesod-Form-Input.html#t:IForm>

- У неё нет экземпляра **Monad**. Задача `Applicative` заключается в том, чтобы позволить форме выполниться целиком, собрать как можно больше информации, и выдать конечный результат. Это не работает в контексте **Monad**.

`IForm` ещё проще: она возвращает либо список ошибок, либо результат.

8.4. Преобразование

«Минуточку», — скажете вы. «Вы говорите, что во введении использовались аппликативные формы, но я уверен, что сигнатура типа указывает `MForm`. Не были ли они монадическими?». Да, верно, итоговая форма, которую мы создали, была монадической. Но на самом деле произошло вот что: мы преобразовали аппликативную форму в монадическую.

Повторюсь, наша цель заключается в том, чтобы повторно использовать как можно больше кода и минимизировать число функций в API. А монадические формы являются более мощными, чем аппликативные, поэтому, грубо говоря, всё, что может быть выражено аппликативными формами, может быть выражено и монадическими. Есть две основные функции, которые нас здесь выручают: `aFormToForm` преобразует любую аппликативную форму в монадическую, а `formToAForm` преобразует определённые виды монадических форм в их аппликативные варианты.

«Ещё минутку», — настаиваете вы. «Я не видел никаких `aFormToForm`!». Это тоже верно. Функция `renderDivs` позаботилась об этом для нас.

8.5. Создание аппликативных форм

Теперь, когда я (надеюсь) убедил вас, что во введении мы действительно имели дело с аппликативными формами, давайте попытаемся понять, как они создаются. Начнём с простого примера:

```
data Car = Car
  { carModel :: Text
  , carYear  :: Int
  }
deriving Show

carAForm :: AForm Synopsis Synopsis Car
carAForm = Car
  <$> areq textField "Модель" Nothing
  <*> areq intField "Год_выпуска" Nothing

carForm :: Html -> MForm Synopsis Synopsis (FormResult Car, Widget)
carForm = renderTable carAForm
```

Здесь мы явным образом разделили аппликативные и монадические формы. В `carAForm` мы использовали операторы `<$>` и `<*>`. Это не должно удивлять; они часто используются в коде, использующем аппликативный стиль. Также у нас по одной

строчке для каждой записи нашего типа данных `Car`. Возможно не удивляет, что мы использовали `textField` для записи с типом `Text` и `intField` для записи с типом `Int`.

Давайте взглянем поближе на функцию `areq`. Её (упрощённая) сигнатура типа — `Field a -> FieldSettings -> Maybe a -> AForm a`. Первый аргумент определяет тип данных поля, как его разбирать и как отображать. Следующий аргумент, `FieldSettings`, даёт нам метку, подсказку, имя и идентификатор поля. В данном случае, мы используем выше упомянутый экземпляр `IsString` для `FieldSettings`.

А что с `Maybe a`? Этот аргумент даёт нам необязательное значение по умолчанию. Например, если мы хотим, чтобы наша форма подставляла для года машины значение по умолчанию «2007», мы будем использовать `areq intField "Year" (Just 2007)`. Мы даже можем подняться на уровень выше, и получить форму, которая принимает необязательный аргумент, задающий значения по умолчанию.

```
carAForm :: Maybe Car -> AForm Synopsis Synopsis Car
carAForm mcar = Car
    <$> areq textField "Модель" (carModel <$> mcar)
    <*> areq intField "Год_выпуска" (carYear <$> mcar)
```

Листинг 8.1: Формы со значениями по умолчанию

8.5.1. Необязательные поля

Предположим, что нам нужно необязательное поле (например, цвет машины). Всё что нужно — это воспользоваться функцией `aopt`.

```
data Car = Car
    { carModel :: Text
    , carYear  :: Int
    , carColor :: Maybe Text
    }
deriving Show

carAForm :: AForm Synopsis Synopsis Car
carAForm = Car
    <$> areq textField "Модель" Nothing
    <*> areq intField "Год_выпуска" Nothing
    <*> aopt textField "Цвет" Nothing
```

Листинг 8.2: Необязательные поля

Как и в случае обязательных полей, последний аргумент — необязательное значение по умолчанию. В итоге, тут получается двойное оборачивание в `Maybe`. Это может показаться чрезмерным (что так и есть), но существенно упрощает написание кода, который принимает в качестве необязательного параметра форму со значениями по умолчанию, как в следующем примере.

```

data Car = Car
  { carModel :: Text
  , carYear  :: Int
  , carColor :: Maybe Text
  }
deriving Show

carAForm :: Maybe Car -> AForm Synopsis Synopsis Car
carAForm mcar = Car
  <$> areq textField "Модель" (carModel <$> mcar)
  <*> areq intField "Год_выпуска" (carYear <$> mcar)
  <*> aopt textField "Цвет" (carColor <$> mcar)

carForm :: Html -> MForm Synopsis Synopsis (FormResult Car, Widget)
carForm = renderTable $ carAForm $ Just $ Car "Forte" 2010 $ Just []
  "серый"

```

Листинг 8.3: Необязательные поля со значениями по умолчанию

8.6. Валидация

Как мы можем сделать, чтобы наша форма принимала только машины, созданные после 1990 года? Если вы помните, мы говорили выше, что Field сам по себе содержит информацию о том, что представляет собой корректное значение. То есть, всё что нам надо сделать — это написать новый Field, верно? Это будет несколько утомительно. Вместо этого, давайте изменим уже существующее поле.

```

carAForm :: Maybe Car -> AForm Synopsis Synopsis Car
carAForm mcar = Car
  <$> areq textField "Модель" (carModel <$> mcar)
  <*> areq carYearField "Год_выпуска" (carYear <$> mcar)
  <*> aopt textField "Цвет" (carColor <$> mcar)
where
  errorMessage :: Text
  errorMessage = "Ваша_машина_чересчур_стара,_купите_новую!"

  carYearField = check validateYear intField

  validateYear y
    | y < 1990 = Left errorMessage
    | otherwise = Right y

```

Хитрость в функции check. Она принимает функцию (validateYear), которая возвращает либо сообщение об ошибке, либо модифицированное значение поля. В этом

примере, мы вообще не изменяли значение. Это самый частый случай использования. И так как этот вид проверки довольно распространён, у нас есть сокращение:

```
carYearField = checkBool (>= 1990) errorMessage intField
```

Функция `checkBool` принимает два параметра: условие, которое должно быть выполнено, и сообщение об ошибке, которое следует отобразить, если условие не верно.

Вы могли заметить явное указание `Text` в сигнатуре типа функции `errorMessage`. При использовании расширения `OverloadedStrings` это необходимо. Чтобы поддерживать интернационализацию, сообщения могут иметь множество различных типов, и у GHC нет возможности определить, какой экземпляр `IsString` вам нужен.

Это замечательно — гарантировать, что наша машина не очень старая. А если мы хотим убедиться в том, что указанный год не из будущего? Чтобы получить текущий год, нам надо будет выполнить немного **IO**. Для таких случаев, нам потребуется функция `checkM`:

```
carYearField = checkM inPast $ checkBool (>= 1990) errorMessage []
  intField

inPast y = do
  thisYear <- liftIO getCurrentYear
  return $ if y <= thisYear
    then Right y
    else Left ("Да, у вас машина времени!" :: Text)

getCurrentYear :: IO Int
getCurrentYear = do
  now <- getCurrentTime
  let today = utctDay now
  let (year, _, _) = toGregorian today
  return $ fromInteger year
```

Функция `inPast` вернёт результат типа **Either**. Однако, она использует монаду `Handler`. Мы используем `liftIO getCurrentYear`, чтобы получить текущий год, и затем сравниваем его с годом, указанным пользователем. Также обратите внимание, как мы можем связывать вместе несколько валидаторов.

Так как валидатор `checkM` работает в монаде `Handler`, он имеет доступ ко всем операциям, которые вы можете обычно выполнять в `Yesod`. Это особенно полезно для выполнения операций с базой данных, которые мы рассмотрим в главе `Persistent`.

8.7. Поля посложнее

Наше поле для ввода цвета хорошее, но не совсем дружественное пользователю. Что мы хотим на самом деле — это выпадающий список.


```

data Car = Car
  { carModel :: Text
  , carYear  :: Int
  , carColor :: Maybe Color
  }
deriving Show

data Color = Red | Blue | Gray | Black
  deriving (Show, Eq, Enum, Bounded)

carAForm :: Maybe Car -> AForm Synopsis Synopsis Car
carAForm mcar = Car
  <$> areq textField "Модель" (carModel <$> mcar)
  <*> areq carYearField "Год_выпуска" (carYear <$> mcar)
  <*> aopt (selectFieldList colors) "Цвет" (carColor <$> mcar)
where
  colors :: [(Text, Color)]
  colors = [("Красный", Red), ("Синий", Blue), ("Серый", Gray),
    ("чёрный", Black)]

```

Листинг 8.4: Выпадающие списки

Функция `selectFieldList` принимает список пар. Первый элемент пары — это текст, который отобразится пользователю в выпадающем списке, а вторым элементом является соответствующее значение Haskell. Конечно, код выше выглядит повторяющимся; мы можем получить тот же результат воспользовавшись экземплярами `Enum` и `Bounded`, которые GHC автоматически выводит для нас.

```

data Car = Car
  { carModel :: Text
  , carYear  :: Int
  , carColor :: Maybe Color
  }
deriving Show

data Color = Red | Blue | Gray | Black
  deriving (Show, Eq, Enum, Bounded)

carAForm :: Maybe Car -> AForm Synopsis Synopsis Car
carAForm mcar = Car
  <$> areq textField "Модель" (carModel <$> mcar)
  <*> areq carYearField "Год_выпуска" (carYear <$> mcar)
  <*> aopt (selectFieldList colors) "Цвет" (carColor <$> mcar)
where
  colors = map (pack . show &&& id) $ [minBound..maxBound]

```

Листинг 8.5: Используя Enum и Bounded

Интервал `[minBound..maxBound]` даёт нам список всех отличных друг от друга значений типа `Color`. Мы затем применяем `map` и `&&&` (так называемый “оператор разветвления”⁷), превращая его в список пар.

Некоторые люди предпочитают выпадающим спискам переключатели. К счастью, это изменение в одно слово. Пример в листинге 8.6.

```
data Car = Car
  { carModel :: Text
  , carYear  :: Int
  , carColor :: Maybe Color
  }
deriving Show

data Color = Red | Blue | Gray | Black
  deriving (Show, Eq, Enum, Bounded)

carAForm :: Maybe Car -> AForm Synopsis Synopsis Car
carAForm mcar = Car
  <$> areq textField "Модель" (carModel <$> mcar)
  <*> areq carYearField "Год_выпуска" (carYear <$> mcar)
  <*> aopt (radioFieldList colors) "Цвет" (carColor <$> mcar)
  where
    colors = map (pack . show &&& id) $ [minBound..maxBound]
```

Листинг 8.6: Переключатели

8.8. Выполнение форм

В какой-то момент у нас возникнет необходимость воспользоваться нашими прекрасными формами и получить некоторые результаты. Для этого в нашем распоряжении есть целый ряд различных функций, каждая из которых имеет своё назначение. Я пройду по всем, начиная с самых распространённых.

runFormPost

Функция выполнит вашу форму с любыми предоставленными POST параметрами. Если запрос не является POST-запросом, вернёт `FormMissing`. Функция автоматически вставляет маркер безопасности⁸ в виде скрытого поля для предупреждения межсайтовых (CSRF⁹) атак.

⁷ориг. fan-out operator

⁸ориг. security token

⁹http://en.wikipedia.org/wiki/Cross-site_request_forgery

runFormGet

Аналог `runFormPost` для GET-параметров. Чтобы отличить обычную загрузку страницы через GET от отправки GET формы, включает в форму дополнительное скрытое поле `_hasdata`. И, в отличие от `runFormPost`, не содержит защиты от межсайтовых атак.

runFormPostNoNonce

То же что и `runFormPost`, но не включает (или не требует) маркера безопасности CSRF.

generateFormPost

Вместо привязки к существующим POST параметрам, действует так, будто их нет. Может быть полезна, когда вам нужно сгенерировать новую форму после того, как предыдущая была отправлена, например, в мастере форм.

generateFormGet

То же что и `generateFormPost`, но для GET.

Тип возвращаемого значения первых трёх функций — `((FormResult a, Widget), Enctype)`. `Widget` уже будет включать любые ошибки валидации и ранее отправленные значения.

8.9. Интернационализация

В этой главе уже было несколько ссылок на интернационализацию. Эта тема подробнее освещена в своей собственной главе, но, так как она имеет глубокое влияние на `yesod-form`, я хотел дать краткий обзор. Идея поддержки интернационализации в `Yesod` состоит в том, чтобы иметь типы данных для представления сообщений. Каждый сайт может иметь экземпляр `RenderMessage` для конкретного типа данных, который будет переводить сообщение в соответствии со списком языков, принимаемых клиентом. Как результат, есть несколько моментов, о которых вам следует знать:

- Для каждого сайта автоматически создаётся экземпляр `RenderMessage` для типа `Text`, так что вы можете просто использовать простые строки, если не заботитесь о поддержке интернационализации. Однако, вам изредка может потребоваться использовать явные сигнатуры типов.
- Все сообщения в `yesod-form` выражаются в терминах типа данных `FormMessage`. Поэтому, для использования `yesod-form` вам нужен соответствующий экземпляр `RenderMessage`. Простая реализация, которая использует английский вариант перевода по умолчанию, будет выглядеть так:

```
instance RenderMessage MyApp FormMessage where
  renderMessage _ _ = defaultFormMessage
```

Такой экземпляр автоматически предоставляется сгенерированным каркасом сайта.

8.10. Монадические формы

Зачастую, простая вёрстка форм достаточна, и тут аппликативные формы превосходят. Иногда, однако, вы захотите получить более настраиваемый вид вашей формы.

Привет, меня зовут и мне лет.

Рис. 8.1: Нестандартная вёрстка формы

В таких случаях следует использовать монадические формы. Они несколько более многословны, чем их аппликативные родственники, но эта многословность позволяет вам получить полный контроль над тем, как форма будет выглядеть. Чтобы получить такую же форму, как приведённая выше, мы могли бы написать что-то подобное следующему:

```
{-# LANGUAGE OverloadedStrings, TypeFamilies, QuasiQuotes,
      TemplateHaskell, MultiParamTypeClasses #-}

import Yesod
import Control.Applicative
import Data.Text (Text)

data MFormExample = MFormExample

mkYesod "MFormExample" [parseRoutes|
/ RootR GET
|]

instance Yesod MFormExample

instance RenderMessage MFormExample FormMessage where
  renderMessage _ _ = defaultFormMessage

data Person = Person { personName :: Text, personAge :: Int }
  deriving Show

personForm :: Html -> MForm MFormExample MFormExample (FormResult ()
  Person, Widget)
personForm extra = do
  (nameRes, nameView) <- mreq textField "Это не используется"
  Nothing
  (ageRes, ageView) <- mreq intField "И это тоже" Nothing
  let personRes = Person <$> nameRes <*> ageRes
      let widget = do
```

```

        toWidget [lucius|
##{fvId ageView} {
    width: 3em;
}
|]
        [whamlet|
#{extra}
<p>
    Привет, меня зовут #
    ^{fvInput nameView}
    \ и мне #
    ^{fvInput ageView}
    \ лет. #
    <input type=submit value="Представиться">
|]
    return (personRes, widget)

getRootR :: Handler RepHtml
getRootR = do
    ((res, widget), enctype) <- runFormGet personForm
    defaultLayout [whamlet|
<p>Результат: #{show res}
<form enctype=#{enctype}>
    ^{widget}
|]

main :: IO ()
main = warpDebug 3000 MFormExample

```

08/monadic-forms.hs

Подобно аппликативной функции `areq` мы используем `mgreq` для монадических форм. (И, конечно, есть `port` для необязательных полей). Но здесь есть существенное различие: `mgreq` возвращает нам пару значений. Вместо скрытого значения типа `FieldView`¹⁰ и автоматической его вставки в виджет, мы получаем управление вставкой по своему усмотрению.

Тип данных `FieldView` содержит ряд записей с информацией о поле формы. Самая важная из них — `fvInput`, которая и есть фактическое поле формы. В этом примере мы также использовали `fvId`, которая возвращает нам HTML атрибут `id` тега `input`. В нашем примере мы её использовали, чтобы задать ширину поля.

Вас, наверное, интересует, что это за значения «Это не используется» и «И это тоже». Функция `mgreq` принимает вторым аргументом значение типа `FieldSettings`. Так как для `FieldSettings` существует экземпляр `IsString`, такие строки по сути раскрываются компилятором в:

¹⁰<http://hackage.haskell.org/packages/archive/yesod-form/latest/doc/html/Yesod-Form-Types.html#t:FieldView>

```
fromString "Это_не_используется" == FieldSettings
  { fsLabel = "Это_не_используется"
  , fsTooltip = Nothing
  , fsId = Nothing
  , fsName = Nothing
  , fsClass = []
  }
```

В случае аппликативных форм, значения `fsLabel` и `fsTooltip` использовались при построении HTML. В случае же монадических форм, Yesod не генерирует для нас никаких HTML-обёрток, и поэтому эти значения игнорируются. Однако, мы храним параметр типа `FieldSettings`, чтобы позволить вам при желании переопределять атрибуты `id` и `name` ваших полей.

Ещё один интересный момент — значение `extra`. Формы типа GET добавляют поле для обозначения отправки, а POST формы добавляют признак безопасности для предупреждения межсайтовых атак. Если вы не включите это дополнительное скрытое поле, то Yesod не примет вашу форму.

Всё остальное довольно-таки тривиально. Мы создаём наше значение `personRes` из значений `nameRes` и `ageRes`, а затем возвращаем кортеж, включающий полученное значение типа `Person` и виджет для формы. А в функции `getRootR` всё также, как для аппликативной формы. В действительности, вы могли бы заменить монадическую форму на аппликативную, и код всё равно работал бы.

8.11. Формы ввода данных

Аппликативные и монадические формы выполняют как генерацию HTML, так и разбор данных. Иногда вам нужно только последнее, например, когда где-то уже создана форма, или если вы хотите генерировать форму динамически, используя JavaScript. В этих случаях вам пригодятся формы ввода данных.

Они аналогичны аппликативным и монадическим с некоторыми различиями:

- Вы используете `runInputPost` и `runInputGet`.
- Вы используете `ireq` и `iort`. Эти функции теперь принимают только 2 аргумента: тип рассматриваемого поля и его имя (т.е. значение HTML атрибута `name`).
- После выполнения формы, она возвращает значение. Она не возвращает виджет или тип кодировки.
- В случае ошибок проверки данных, возвращается страница с сообщением об ошибке «некорректные аргументы (`invalid arguments`)».

Вы можете переписать предыдущий пример с помощью форм ввода данных. Однако, стоит заметить, что эта версия менее дружелюбна пользователю. Если вы допустите ошибку при заполнении аппликативной или монадической формы, то вы вернётесь на ту же страницу с ранее введёнными данными, а сообщение об ошибке укажет, что

надо исправить. При использовании форм ввода данных пользователь получит просто сообщение об ошибке.

```
{-# LANGUAGE OverloadedStrings, TypeFamilies, QuasiQuotes,
      TemplateHaskell, MultiParamTypeClasses #-}

import Yesod
import Control.Applicative
import Data.Text (Text)

data Input = Input

mkYesod "Input" [parseRoutes]
/ RootR GET
/input InputR GET
]

instance Yesod Input

instance RenderMessage Input FormMessage where
  renderMessage _ _ = defaultFormMessage

data Person = Person { personName :: Text, personAge :: Int }
  deriving Show

getRootR :: Handler RepHtml
getRootR = defaultLayout [whamlet|
<form action=@{InputR}>
  <p>
    Меня зовут #
    <input type=text name=name>
    \ и мне #
    <input type=text name=age>
    \ лет. #
    <input type=submit value="Представиться">
  ]

getInputR :: Handler RepHtml
getInputR = do
  person <- runInputGet $ Person
    <$> ireq textField "name"
    <*> ireq intField "age"
  defaultLayout [whamlet|<p>#{show person}|]
```

```
main :: IO ()
main = warpDebug 3000 Input
```

08/input-forms.hs

8.12. Пользовательские поля

Поля, которые встроены в Yesod, скорее всего покроют большую часть ваших потребностей для работы с формами. Но иногда вам может понадобиться что-то более специализированное. К счастью, вы можете создавать новые поля в Yesod самостоятельно. Тип данных `Field` имеет две записи: `fieldParse` принимает список значений, отправленных пользователем, и возвращает один из трёх результатов:

- Сообщение об ошибке в случае, если проверка данных закончилась неудачей.
- Разобранное значение.
- **Nothing**, являющееся указанием на то, что данные не были предоставлены.

Последний случай может несколько удивить: разве Yesod не может автоматически понять, что информации не было предоставлено, если входной список пуст? Не совсем. Флажки, например, сообщают о сброшенном состоянии, посылая пустой список.

Почему список? Не лучше ли **Maybe**? Не всегда. В случае группы флажков или списков с возможностью множественного выбора у вас будет множество виджетов с одним именем. Мы используем этот трюк в нашем примере ниже.

Вторая запись — `fieldView` — формирует виджет для отображения пользователю. У этой функции четыре аргумента: атрибут `id`, атрибут `name`, результат и значение типа **Bool** для обозначения, является ли поле обязательным. Что мы подразумеваем под результатом? В действительности, это значение типа **Either**, возвращающее либо неразобраный ввод (когда разбор не удался), либо успешно разобранное значение. Великолепный пример этому — поле `intField`. Если вы введёте 42, то в результате будет **Right** 42. Но если вы введёте слово «черепаша», результатом будет **Left** "черепаша". Это позволит вам заполнить атрибут `value` вашего тега `input` для организации единообразного пользовательского интерфейса.

В качестве маленького примера мы создадим новый тип поля — поле для подтверждения пароля. В нём будет два элемента для ввода текста (оба с одним и тем же значением атрибута `name`), и оно будет возвращать сообщение об ошибке, если пароли не совпадают. Заметьте, что в отличие от большинства полей, новое поле *не* предоставляет атрибуты `value` для тегов `input`, потому что вы **никогда** не захотите отправлять обратно введённый пользователем пароль в своём HTML.

```
passwordConfirmField :: Field sub master Text
passwordConfirmField = Field
  { fieldParse = \rawVals _ ->
    case rawVals of
      [a, b]
        | a == b -> return $ Right $ Just a
        | otherwise -> return $ Left "Пароли не совпадают"
```



```

        [] -> return $ Right Nothing
        _ -> return $ Left "Вы_должны_ввести_два_значения"
        , fieldView = \idAttr nameAttr _ eResult isReq -> [whamlet|
<input id=#{idAttr} name=#{nameAttr} type=password>
<div>Confirm:
<input id=#{idAttr}-confirm name=#{nameAttr} type=password>
|]
        , fieldEncType = UrlEncoded
        }

getRootR :: Handler RepHtml
getRootR = do
    ((res, widget), encType) <- runFormGet $ renderDivs
        $ areq passwordConfirmField "Пароль" Nothing
        defaultLayout [whamlet|
<p>Result: #{show res}
<form encType=#{encType}>
    ^{widget}
    <input type=submit value="Сменить_пароль">
|]

```

8.13. Выводы

Формы в Yesod делятся на три вида. Аппликативные используются чаще всего, так как предоставляют красивый интерфейс и простой для использования API. Моноадические формы дают больше возможностей, но их сложнее использовать. Формы ввода данных полезны, когда вам надо просто принять данные пользователя, не генерируя сложных виджетов.

Из коробки Yesod предоставляет несколько различных типов полей для форм. Для их использования вам необходимо указать вид формы и является ли поле обязательным или нет. Для этого есть шесть вспомогательных функций: `areq`, `aopt`, `mreq`, `mopt`, `ireq` и `iopt`.

Формы обладают большой мощностью. Они могут автоматически вставлять код на JavaScript, чтобы облегчить вам получение более привлекательных элементов управления, как, например, диалог выбора даты из библиотеки jQuery UI. Формы также полностью готовы для включения интернационализации, так что вы можете поддерживать глобальное сообщество пользователей. А в случае более специфических потребностей, вы можете привязать функции валидации данных для существующих полей, или реализовать своё поле «с чистого листа».

Глава 9

Сессии

HTTP является протоколом без состояния. И хотя некоторые рассматривают это как недостаток, сторонники RESTful веб-разработки восхваляют это как плюс. Ведь когда состояние отсутствует, тогда приложение легче масштабировать, доступно автоматическое кэширование и многие другие приятные побочные эффекты. В общем, вы можете провести много параллелей с неизменяемой природой Haskell.

Насколько это возможно, RESTful приложениям следует избегать сохранения состояния с информацией о взаимодействии с клиентом. Тем не менее, иногда это неизбежно. Классическим примером является такая функциональность, как корзина покупок, но и другие, более обычные взаимодействия, наподобие должной обработки входа в систему, могут быть значительно улучшены путём правильного использования сессий.

В этой главе описывается, как Yesod сохраняет данные сессии, как вы можете получить доступ к этим данным, а также некоторые специальные функции, которые помогут вам использовать сессии наилучшим образом.

9.1. Clientsession

Одним из первых пакетов, которые отделились от Yesod, был `clientsession`¹. Этот пакет использует шифрование и подписи для хранения данных в куки-файлах на стороне клиента. Шифрование препятствует изучению данных пользователем, а подпись гарантирует, что сессия не была ни захвачена, ни изменена.

Возможно, с точки зрения эффективности, хранение данных в куки звучит как плохая идея: ведь это означает, что данные должны быть отправлены на каждый запрос. Однако на практике, `clientsession` может значительно улучшить производительность.

- Для обработки запроса не требуется обращаться к базе данных на стороне сервера.
- Легко масштабировать по горизонтали: каждый запрос содержит всю информацию, необходимую для ответа.

¹<http://hackage.haskell.org/package/clientsession>

- Для того чтобы уменьшить нагрузку на канал, сайты могут предоставлять статический контент с отдельного доменного имени, чтобы избежать накладных расходов на передачу куки для каждого запроса.

Сохранение мегабайт информации в сессии будет плохой идеей. И поэтому большинство рекомендаций по реализации сессий не рекомендуют такую практику. Если вам действительно нужно сберечь много данных для пользователя, то лучше хранить ключ поиска в сессии, а фактические данные в базе данных.

Все взаимодействия с `clientsession` обрабатывается внутри `Yesod`, но есть несколько мест, где вы можете немного настроить его поведение.

9.2. Управление сессией

В классе типов `Yesod` существуют три функции, которые управляют работой сессии. Функция `encryptKey`² возвращает текущий ключ шифрования. По умолчанию она берёт его из локального файла, чтобы сессии могли сохраняться между перезагрузками базы данных. Если этот файл не существует, то он будет автоматически создан и заполнен случайными данными. А если переопределить эту функцию так, чтобы она возвращала **Nothing**, то сессии будут отключены.

Зачем отключать сессии? Они вносят накладные расходы на **производительность**. При нормальных обстоятельствах, эти расходы минимальны, особенно по сравнению с доступом к базе данных. Однако когда имеешь дело с очень простыми задачами, эти накладные расходы могут стать заметными. Будьте осторожны с отключением сессий: ведь это также отключит такую функциональность, как защиту от подделки межсайтовых запросов (Cross-Site Request Forgery, сокращённо CSRF).

Следующая функция `clientSessionDuration`³. Эта функция возвращает количество минут, в течении которых сессия должна быть активной. Значение по умолчанию составляет 120 (2 часа).

Это значение в конечном итоге влияет на куки сессии двумя способами: во-первых, оно определяет срок действия куки. Однако что более важно, время окончания сессии закодировано внутри подписи сессии. Когда `Yesod` расшифровывает подпись, он проверяет, в прошлом ли дата, и если да, то значения сессии игнорируются.

Каждый раз, когда `Yesod` посылает ответ клиенту, он отправляет куки с обновлённым сроком годности. Таким образом, даже если вы не изменяете сами значения сессии, срок сессии не истечёт, если пользователь просто продолжает просматривать ваш сайт.

И это подводит к последней функции: `sessionIpAddress`⁴. По умолчанию, чтобы предотвратить перехват сессии, `Yesod` также кодирует в куки IP-адрес клиента. В

²<http://hackage.haskell.org/packages/archive/yesod-core/latest/doc/html/Yesod-Core.html#v:encryptKey>

³<http://hackage.haskell.org/packages/archive/yesod-core/latest/doc/html/Yesod-Core.html#v:clientSessionDuration>

⁴<http://hackage.haskell.org/packages/archive/yesod-core/latest/doc/html/Yesod-Core.html#v:sessionIpAddress>

общем, это хорошая вещь. Тем не менее, некоторые провайдеры держат своих пользователей за прокси серверами, которые переписывают их IP-адреса, иногда меняя исходный адрес в середине сессии. Если это произойдёт, и `sessionIpAddress` включён, то сессия пользователя будет сброшена. Установка же этого параметра в **False** позволит сессии продолжаться в таких условиях ценой подержания пользователя возможному захвату сессии.

9.3. Работа с сессией

Как и в большинстве фреймворков, сессия в Yesod представляет собой хранилище пар ключ-значение. Базовое API сессии сводится к четырём функциям: `lookupSession` получает значение для ключа (если имеется), `getSession` возвращает все пары ключ/значение, `setSession` задаёт значение для ключа, а `deleteSession` очищает значение для ключа.

```
{-# LANGUAGE TypeFamilies, QuasiQuotes, TemplateHaskell,
      MultiParamTypeClasses, OverloadedStrings #-}
import Yesod
import Control.Applicative ((<$>), (<*>))
import qualified Web.ClientSession as CS

data SessionExample = SessionExample

mkYesod "SessionExample" [parseRoutes]
/ Root GET POST
[]

getRoot :: Handler RepHtml
getRoot = do
    sess <- getSession
    hamletToRepHtml [hamlet|
<form method=post>
  <input type=text name=key>
  <input type=text name=val>
  <input type=submit>
<h1>#{show sess}
|]

postRoot :: Handler ()
postRoot = do
    (key, mval) <- runInputPost $ (,) <$> ireq textField "key" <*>
    iopt textField "val"
    case mval of
        Nothing -> deleteSession key
        Just val -> setSession key val
    liftIO $ print (key, mval)
```

```

    redirect Root

instance Yesod SessionExample where
    -- Устанавливаем таймаут сессии в 1 минуту, чтобы облегчить
    -- тестирование
    makeSessionBackend _ = do
        key <- CS.getKey CS.defaultKeyFile
        (cacher, _) <- clientSessionDateCacher 60
        return $ Just $ clientSessionBackend2 key cacher

instance RenderMessage SessionExample FormMessage where
    renderMessage _ _ = defaultFormMessage

main :: IO ()
main = warpDebug 3000 SessionExample

```

09/session-example.hs

9.4. Сообщения

Одно из использований сессий — это сообщения. Они используются, чтобы решить одну из обычных задач в веб-разработке: пользователь выполняет POST запрос, веб-приложение делает изменения, а затем должно *одновременно* перенаправить пользователя на новую страницу и отправить ему сообщение об успехе. (Это известно как Post/Redirect/Get).

Yesod предоставляет пару функций, чтобы сделать эту задачу очень простой: `setMessage` сохраняет значение в сессии, а `getMessage` одновременно считывает последнее значение в сессии и очищает старое значение, так что оно не будет случайно отображено два раза.

Рекомендуется вызывать `getMessage` в `defaultLayout`, так любое доступное сообщение покажется пользователю немедленно, без необходимости помнить о добавлении вызова `getMessage` в каждом обработчике.

```

{-# LANGUAGE OverloadedStrings, TypeFamilies, TemplateHaskell,
      QuasiQuotes, MultiParamTypeClasses #-}

import Yesod

data Messages = Messages

mkYesod "Messages" [parseRoutes]
/ RootR GET
/set-message SetMessageR POST
[]

```

```

instance Yesod Messages where
  defaultLayout widget = do
    pc <- widgetToPageContent widget
    mmsg <- getMessage
    hamletToRepHtml [hamlet|
$doctype 5
<html>
  <head>
    <title>#{pageTitle pc}
    ^{pageHead pc}
  <body>
    $maybe msg <- mmsg
    <p>Your message was: #{msg}
    ^{pageBody pc}
|]

instance RenderMessage Messages FormMessage where
  renderMessage _ _ = defaultFormMessage

getRootR :: Handler RepHtml
getRootR = defaultLayout [whamlet|
<form method=post action=@{SetMessageR}>
  My message is: #
  <input type=text name=message>
  <input type=submit>
|]

postSetMessageR :: Handler ()
postSetMessageR = do
  msg <- runInputPost $ ireq textField "message"
  setMessage $ toHtml msg
  redirect RootR

main :: IO ()
main = warpDebug 3000 Messages

```

09/messages.hs

9.5. Пункт назначения

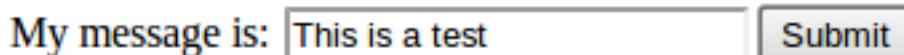
Не путайте с фильмом ужасов, пункт назначения — это понятие которое используется внутри `yesod-auth`⁵. Предположим, пользователь запрашивает страницу, которая

⁵<http://hackage.haskell.org/package/yesod-auth>



My message is:

Рис. 9.1: Первая загрузка страницы, нет сообщения



My message is:

Рис. 9.2: Новое сообщение введено в текстовое поле

требует аутентификации. Если пользователь ещё не вошёл в систему, вам нужно отправить его на страницу входа. Хорошо продуманное веб-приложение затем *отправит его обратно на первоначально запрошенную страницу*. Это то, что мы называем пунктом назначения.

Функция `redirectUltDest` отправляет пользователя в пункт назначения, установленный в его сессии, очищая это значение в сессии. Так же она берёт значение по умолчанию, в случае, если значение пункта назначения не установлено. Для установки сессии, есть три варианта:

- `setUltDest` устанавливает пункт назначения в данный URL;
- `setUltDestCurrent` устанавливает пункт назначения в текущий запрошенный URL;
- `setUltDestReferer` устанавливает пункт назначения на основе заголовка; Referer страница, которая привела пользователя на текущую страницу;

Давайте рассмотрим небольшой пример приложения. Оно позволит пользователю задать его имя в сессии, а затем скажет ему его имя с другого маршрута. Если имя ещё

Your message was: This is a test



My message is:

Рис. 9.3: После отправки формы, сообщение появляется в верхней части страницы

My message is:

Рис. 9.4: После обновления, сообщение очищено

не было установлено, пользователь будет перенаправлен на страницу ввода имени, с пунктом назначения, установленным в возврат к текущей странице.

```
{-# LANGUAGE OverloadedStrings, TypeFamilies, TemplateHaskell,
      QuasiQuotes, MultiParamTypeClasses #-}

import Yesod

data UltDest = UltDest

mkYesod "UltDest" [parseRoutes|
/ RootR GET
/setname SetNameR GET POST
/sayhello SayHelloR GET
|]

instance Yesod UltDest

instance RenderMessage UltDest FormMessage where
  renderMessage _ _ = defaultFormMessage

getRootR = defaultLayout [whamlet|
<p>
  <a href=@{SetNameR}>Set your name
<p>
  <a href=@{SayHelloR}>Say hello
|]

-- Отобразить форму для ввода имени
getSetNameR = defaultLayout [whamlet|
<form method=post>
  My name is #
  <input type=text name=name>
  . #
  <input type=submit value="Set_ name">
|]

-- Достать указанное пользователем имя
postSetNameR :: Handler ()
```



```

postSetNameR = do
  -- Получить указанное имя и установить его для сессии
  name <- runInputPost $ ireq textField "name"
  setSession "name" name

  -- После того как мы получили имя, перенаправить в пункт назначения.
  -- Если пункт назначения не задан, по умолчанию направляем на домашнюю страницу
  redirectUltDest RootR

getSayHelloR = do
  -- Ищем значение имени установленное в сессии
  mname <- lookupSession "name"
  case mname of
    Nothing -> do
      -- Имя не задано, устанавливаем текущую станицу как пункт назначения и перенаправляем на станицу задания имени - SetName
      setUltDestCurrent
      setMessage "Please_tell_me_your_name"
      redirect SetNameR
    Just name -> defaultLayout [whamlet|
<p>Welcome #{name}
|]

main :: IO ()
main = warpDebug 3000 UltDest

```

09/ultimate-destination.hs

9.6. Выводы

Сессии — это способ номер один для преодоления отсутствия состояния в HTTP. Но мы не должны использовать этот чёрный ход всякий раз, когда хотим: отсутствие состояний в веб-приложениях — это хорошее свойство, и мы должны уважать его, когда это возможно. Тем не менее, есть определённые случаи, когда важно сохранять некоторое состояние.

В Yesod очень простой API для работы с сессиями. Он предоставляет хранилище пар ключ-значение, и несколько удобных функций для работы с ними для типичных сценариев использования. При правильном использовании, с небольшой нагрузкой, сессии должны быть скромной частью вашей веб-разработки.

Глава 10

Persistent

Формы представляют собой границу между пользователем и приложением. Другая граница, с которой нам приходится иметь дело, находится между приложением и хранилищем. Является ли это хранилище SQL базой данных, YAML- или бинарным файлом, вам с большой вероятностью придется попотеть, чтобы хранилище принимало типы данных вашего приложения. Persistent представляет собой ответ Yesod'a на проблему хранения данных. Это универсальный типобезопасный интерфейс к хранилищу данных для Haskell.

Haskell предлагает множество различных биндингов к базам данных. Однако большинство из них имеют малое представление о схеме базы данных и потому не обеспечивают полезных статических проверок. Кроме того, они вынуждают программиста использовать API и типы данных, зависящие от конкретной базы данных. Чтобы избавиться от этих проблем, программистами на Haskell была предпринята попытка пойти более революционным путем и создать хранилище данных, специфичное для Haskell, тем самым получив возможность с легкостью хранить любой тип данных Haskell. Эта возможность действительно прекрасна в некоторых случаях, но она делает программиста зависимым от техники хранения данных и используемой библиотеки, плохо взаимодействует с другими языками, а также для обеспечения гибкости может требовать от программиста написания кучи кода, запрашивающего данные. В отличие от Persistent, который предоставляет выбор среди множества баз данных, каждая из которых оптимизирована для различных случаев, позволяет взаимодействовать с другими языками, а также использовать безопасный и производительный интерфейс запросов.

Persistent следует принципам безопасности типов и краткого, декларативного синтаксиса. Среди других возможностей следует отметить:

- Независимость от базы данных. Имеется первоклассная поддержка PostgreSQL, SQLite и MongoDB, а также экспериментальная поддержка CouchDB и находящаяся в разработке поддержка MySQL.
- Будучи нереляционным по своей природе, Persistent позволяет одновременно поддерживать множество слоев хранения данных и не обременен проблемами производительности, связанными с использованием JOIN'ов.

- Основным источником разочарования при использовании SQL баз данных является попытка изменения схемы базы данных. Persistent позволяет автоматически выполнять обновление схемы базы данных.

10.1. Краткое содержание

```
{-# LANGUAGE QuasiQuotes, TemplateHaskell, TypeFamilies,   
    OverloadedStrings #-}  
{-# LANGUAGE GADTs, FlexibleContexts #-}  
  
import Database.Persist  
import Database.Persist.Sqlite  
import Database.Persist.TH  
import Control.Monad.IO.Class (liftIO)  
  
share [mkPersist sqlSettings, mkMigrate "migrateAll"]   
    [persistUpperCase]  
Person  
    name String  
    age Int Maybe  
    deriving Show  
BlogPost  
    title String  
    authorId PersonId  
    deriving Show  
]  
  
main :: IO ()  
main = runSqlite ":memory:" $ do  
    runMigration migrateAll  
  
    johnId <- insert $ Person "John_Doe" $ Just 35  
    janeId <- insert $ Person "Jane_Doe" Nothing  
  
    insert $ BlogPost "My_first_post" johnId  
    insert $ BlogPost "One_more_for_good_measure" johnId  
  
    oneJohnPost <- selectList [BlogPostAuthorId ==. johnId] [LimitTo   
    1]  
    liftIO $ print (oneJohnPost :: [Entity BlogPost])  
  
    john <- get johnId  
    liftIO $ print (john :: Maybe Person)  
  
    delete janeId
```

```
deleteWhere [BlogPostAuthorId ==. johnId]
```

10/synopsis.hs

10.2. Решение пограничной проблемы

Допустим, вы храните информацию о людях в SQL базе данных. Соответствующая таблица может выглядеть как-то так:

```
CREATE TABLE Person(id SERIAL PRIMARY KEY, name VARCHAR NOT NULL,
age INTEGER)
```

И если вы используете такую СУБД, как PostgreSQL, вы можете быть уверены, что СУБД никогда не сохранит какой-нибудь дополнительный текст в поле age. (Нельзя сказать то же самое в отношении SQLite, однако пока что забудем об этом.) Для отображения этой таблицы вы можете захотеть создать примерно такой тип данных:

```
data Person = Person
  { personName :: Text
  , personAge  :: Int
  }
```

Все выглядит вполне типобезопасно — схема базы данных соответствует типу данных в Haskell, СУБД гарантирует, что некорректные данные никогда не будут сохранены в таблице, и все в целом выглядит прекрасно. До поры до времени.

Вы хотите получить данные из СУБД, которая в свою очередь предоставляет их в нетипизированном формате.

- Вы хотите найти все людей, старше 32-х лет, но по ошибке пишете «тридцать два» в SQL-запросе. И значете что? Все прекрасно скомпилируется и вы не узнаете о проблеме до тех пор, пока не запустите программу.
- Вы решили найти первых десяти человек в алфавитном порядке. Нет проблем... до тех пор, пока вы не сделаете опечатку в SQL-запросе. И снова, вы не узнаете об этом до тех пор, пока не запустите программу.
- В языках с динамической типизацией ответом на эти проблемы является модульное тестирование. Проверьте, что для всего, что может пойти не так, вы не забыли написать тест. Но как, я полагаю, вы уже знаете, это не очень согласуется с подходом, принятом в Yesod. Мы предпочитаем использовать преимущества статической типизации языка Haskell для нашей собственной защиты, насколько это возможно, и хранение данных не является исключением.

Итак, вопрос остается открытым: как мы можем использовать систему типов языка Haskell, чтобы исправить положение?

10.2.1. Типы

Как и в случае с маршрутами, нет ничего невероятно сложного в типобезопасном доступе к данным. Он всего лишь требует написания монотонного, подверженного ошибкам избыточного шаблонного кода. Как обычно, это означает, что мы можем использовать систему типов для того, чтобы избежать лишних ошибок. А чтобы не заниматься нудной работой, мы вооружимся Template Haskell.

В ранних версиях Persistent очень активно использовался Template Haskell. Начиная с версии 0.6 используется новая архитектура, позаимствованная из пакета `groundhog`. Благодаря новому подходу существенная часть нагрузки была переложена на фантомные типы.

`PersistValue` является основным строительным блоком в Persistent. Этот тип представляет данные, посылаемые базе данных или получаемые от нее. Вот его определение:

```
data PersistValue = PersistText Text
                  | PersistByteString ByteString
                  | PersistInt64 Int64
                  | PersistDouble Double
                  | PersistBool Bool
                  | PersistDay Day
                  | PersistTimeOfDay TimeOfDay
                  | PersistUTCTime UTCTime
                  | PersistNull
                  | PersistList [PersistValue]
                  | PersistMap [(T.Text, PersistValue)]
                  | PersistForeignKey ByteString -- ^ предназначен
прежде всего для MongoDB
```

Каждый из бэкэндов Persistent должен знать, как переводить соответствующие значения во что-то, понятное СУБД. Однако было бы неудобно выражать все данные через эти базовые типы. Следующим слоем является класс типов `PersistField`, определяющий, как произвольный тип может быть преобразован в тип `PersistValue` или обратно. `PersistField` соответствует столбцам в SQL базах данных. В приведенном ранее примере с людьми `name` и `age` будут нашими `PersistField`'ами.

Чтобы связать пользовательский код, нам понадобится последний класс типов — `PersistEntity`. Экземпляр класса типов `PersistEntity` соответствует таблице в SQL базе данных. Этот класс типов определяет несколько функций и связанные с ними типы. Таким образом, имеет место следующее соответствие между Persistent и SQL:

SQL	Persistent
Тип (VARCHAR, INTEGER и тд)	PersistValue
Столбец	PersistField
Таблица	PersistEntity

10.2.2. Генерация кода

Дабы убедиться, что экземпляры класса `PersistEntity` корректно соответствуют нашим типам данных, `Persistent` берет на себя ответственность и за тех, и за других. Это хорошо и с точки зрения принципа DRY (Не повторяйтесь, Don't Repeat Yourself): от вас требуется объявить сущности только один раз. Рассмотрим следующий пример:

```
{-# LANGUAGE QuasiQuotes, TypeFamilies, GeneralizedNewtypeDeriving,
  TemplateHaskell, OverloadedStrings, GADTs #-}

import Database.Persist
import Database.Persist.TH
import Database.Persist.Sqlite
import Control.Monad.IO.Class (liftIO)

mkPersist sqlSettings [persist|
Person
  name String
  age Int
  deriving Show
|]

```

Здесь мы используем комбинацию из `Template Haskell` и квазицитирования (как при определении маршрутов): `persist` является обработчиком квазицитирования, который преобразует чувствительный к пробелам синтаксис в список определений сущностей. (Также вы можете вынести определение сущностей в отдельный файл и воспользоваться `persistFileWith`.) `mkPersist` принимает список этих сущностей и определяет:

- По одному типу данных языка `Haskell` на сущность;
- Экземпляр класса `PersistEntity` для каждого определенного типа данных;

Приведенный выше пример генерирует код, который выглядит примерно следующим образом:

```
{-# LANGUAGE TypeFamilies, GeneralizedNewtypeDeriving,
  OverloadedStrings, GADTs #-}

import Database.Persist
import Database.Persist.Store
import Database.Persist.Sqlite
import Database.Persist.EntityDef
import Control.Monad.IO.Class (liftIO)
import Control.Applicative

data Person = Person
  { personName :: String
  , personAge  :: Int
  }

```

```

    }
    deriving (Show, Read, Eq)

type PersonId = Key SqlPersist Person

instance PersistEntity Person where
    -- Обобщенный алгебраический тип данных.
    -- Это дает нам типобезопасный подход к сопоставлению
    -- полей с их типами данных
    data EntityField Person typ where
        PersonId    :: EntityField Person PersonId
        PersonName  :: EntityField Person String
        PersonAge   :: EntityField Person Int

type PersistEntityBackend Person = SqlPersist

toPersistFields (Person name age) =
    [ SomePersistField name
    , SomePersistField age
    ]

fromPersistValues [nameValue, ageValue] = Person
    <$> fromPersistValue nameValue
    <*> fromPersistValue ageValue
fromPersistValues _ = Left "Invalid input"

-- Информация о каждом поле для внутреннего использования при
-- генерации SQL выражений
persistFieldDef PersonId = FieldDef
    (HaskellName "Id")
    (DBName "id")
    (FTTypeCon Nothing "PersonId")
    []
persistFieldDef PersonName = FieldDef
    (HaskellName "name")
    (DBName "name")
    (FTTypeCon Nothing "String")
    []
persistFieldDef PersonAge = FieldDef
    (HaskellName "age")
    (DBName "age")
    (FTTypeCon Nothing "Int")
    []
data Unique Person typ = IgnoreThis
entityDef = undefined
halfDefined = undefined

```

```

persistUniqueToFieldNames = undefined
persistUniqueToValues = undefined
persistUniqueKeys = undefined
persistIdField = undefined

```

Как и следовало ожидать, определение типа данных `Person` очень близко к определению, данному в оригинальной версии кода, где использовался `Template Haskell`. Мы также имеем обобщенный алгебраический тип данных (ОАТД), предоставляющий отдельный конструктор для каждого поля. Этот ОАТД кодирует как тип сущности, так и тип поля. Мы используем его конструкторы через модуль `Persistent`, например, чтобы убедиться, что когда мы применяем фильтр, типы фильтруемого значения и поля совпадают.

Мы можем использовать сгенерированный тип `Person` как и любой другой тип языка `Haskell`, а затем передать его в одну из функций модуля `Persistent`.

```

main = runSqlite ":memory:" $ do
  michaelId <- insert $ Person "Michael" 26
  michael <- get michaelId
  liftIO $ print michael

```

Мы начнем со стандартного кода, работающего с базой данных. В данном случае мы использовали функции для работы с одним соединением. Модуль `Persistent` также предоставляет функции для работы с пулом соединений, которые использовать в боевом окружении обычно предпочтительнее.

В приведенном примере мы видим две функции. Функция `insert` создает новую запись в базе данных и возвращает ее ID. Как и все остальное в модуле `Persistent`, ID являются типобезопасными. Более подробно о том, как работают эти ID, мы узнаем позже. Итак, код `insert $ Person "Michael" 25`, возвращает значение типа `PersonId`.

Следующая функция, которую мы видим — это `get`. Она пытается загрузить из базы данных значение, используя заданный ID. При использовании `Persistent` вам никогда не придется беспокоиться, что вы, возможно, используете ключ не от той таблицы. Код, который пытается получить другую сущность (например, `House`), используя `PersonId`, никогда не будет скомпилирован.

10.2.3. PersistStore

Последний момент, который остался без объяснения в предыдущем примере: что делает функция `runSqlite`? И что это за монада, в которой выполняются все наши действия с базой данных?

Все действия с базой данных должны выполняться в экземпляре `PersistStore`. Как следует из его названия, каждое хранилище (`PostgreSQL`, `SQLite`, `MongoDB`) имеет свой экземпляр `PersistStore`. Именно с его помощью происходят генерация SQL-запросов, преобразования из `PersistValue` в значения, специфичные для конкретной СУБД и так далее.

Как вы, вероятно, догадываетесь, несмотря на то, что `PersistStore` предоставляет безопасный, хорошо типизированный интерфейс, во время взаимодействия с базой данных многое может пойти не так. Однако тестируя код автоматически и тщательно в каждом отдельном месте, мы можем централизовать склонный к ошибкам код и убедиться, что он настолько свободен от ошибок, насколько это вообще возможно.

Функция `runSqlite` создает отдельное соединение с базой данных, используя предоставленную строку. Для тестов мы воспользуемся строкой «:memoгy:», которая означает использовать базу данных, расположенную в памяти. `SQLite` и `PostgreSQL` используют один и тот же экземпляр `PersistStore: SqlPersist`.

В действительности существует еще несколько классов типов — это `PersistUpdate` и `PersistQuery`. Различные классы типов предоставляют различную функциональность, что позволяет нам писать бэкенды, использующие более простые хранилища (например, `Redis`) несмотря на то, что они не обладают всей высокоуровневой функциональностью, предоставляемой `Persistent`.

Важный момент, который следует отметить, заключается в том, что каждый отдельный вызов `runSqlConnection` выполняется в отдельной транзакции. Отсюда два следствия:

- Для многих СУБД выполнение коммита может быть дорогой операцией. Помещая множество запросов в одну транзакцию, вы можете существенно ускорить выполнение кода.
- Если где-либо внутри вызова `runSqlConnection` бросается исключение, все выполненные действия будут откаты (конечно, если используемый бэкенд поддерживает транзакции).

10.3. Миграции

Мне очень жаль, но все это время я вам лгал: пример из предыдущего раздела на самом деле не работает. Если вы попытаетесь запутить его, то получите ошибку о несуществующей таблице.

При работе с реляционными СУБД, изменение схемы базы данных обычно является большой проблемой. Вместо того, чтобы возлагать эту проблему на плечи пользователя, `Persistent` делает шаг вперед и протягивает руку помощи. Только нужно его об этом попросить. Вот как примерно это выглядит:

```
{-# LANGUAGE QuasiQuotes, TypeFamilies, GeneralizedNewtypeDeriving,
    TemplateHaskell, OverloadedStrings, GADTs, FlexibleContexts #-}

import Database.Persist
import Database.Persist.TH
import Database.Persist.Sqlite
import Control.Monad.IO.Class (liftIO)

share [mkPersist sqlSettings, mkSave "entityDefs"] [persistUpperCase|
Person
```

```

    name String
    age Int
    deriving Show
]

main = runSqlite ":memory:" $ do
    runMigration $ migrate entityDefs (undefined :: Person) -- []
    добавлена эта строчка, и только!
    michaelId <- insert $ Person "Michael" 26
    michael <- get michaelId
    liftIO $ print michael

```

10/migration.hs

Благодаря этому небольшому изменению, Persistent будет автоматически создавать для вас таблицу Person. Разбиение между функциями runMigration и migrate позволяет производить миграции множества таблиц одновременно.

Это хорошо работает, когда речь идет о нескольких сущностях, но становится несколько утомительным при работе с десятками. Вместо того, чтобы повторяться, Persistent предлагает функцию mkMigrate:

```

{-# LANGUAGE QuasiQuotes, TypeFamilies, GeneralizedNewtypeDeriving, []
    TemplateHaskell, OverloadedStrings, GADTs, FlexibleContexts #-}

import Database.Persist
import Database.Persist.Sqlite
import Database.Persist.TH

share [mkPersist sqlSettings, mkMigrate "migrateAll"] []
    [persistUpperCase|
Person
    name String
    age Int
    deriving Show
Car
    color String
    make String
    model String
    deriving Show
]

main = runSqlite ":memory:" $ do
    runMigration migrateAll

```

10/mkmmigrate.hs

`mkMigrate` — это функция `Template Haskell`, которая создает новую функцию, что будет вызывать `migrate` для всех сущностей, объявленных в блоке `persist`. Функция `share` является небольшим хелпером, который передает информацию из блока `persist` каждой из функций `Template Haskell`, а затем объединяет результаты их выполнения.

В `Persistent` используются очень консервативные правила относительно того, что следует делать во время миграции. Сначала он загружает из базы данных всю информацию о таблицах, вместе со всеми объявленными типами данных `SQL`. Эту информацию он сравнивает с определениями сущностей, приведенными в коде. В следующих случаях схема базы данных будет изменена автоматически:

- Изменился тип данных поля. Но СУБД может возражать против такого изменения, если данные не могут быть преобразованы.
- Было добавлено новое поле. Однако если поле не может быть пустым (`NULL`), не было предоставлено значение по умолчанию (как это сделать, мы обсудим позже), и в таблице уже есть какие-то данные, СУБД не позволит добавить поле.
- Некоторое поле отныне может быть пустым. В обратном случае `Persistent` попытается выполнить преобразование, если СУБД позволит это сделать.
- Была добавлена совершенно новая сущность.

Однако есть и случаи, которые `Persistent` не в состоянии обработать:

- Переименование сущностей или полей. У `Persistent` нет никакой возможности узнать, что поле `«name»` было переименовано в `«fullName»`. Все, что он видит — это старое поле с именем `«name»` и новое поле с именем `«fullName»`.
- Удаление полей. Поскольку это может привести к потере данных, по умолчанию `Persistent` отказывается выполнять такие преобразования. Вы можете настоять на этом, воспользовавшись функцией `runMigrationUnsafe` вместо `runMigration`, но это не рекомендуется.

Функция `runMigration` выводит выполняемые миграции в `STDERR` (если вам не нравится такое поведение, воспользуйтесь функцией `runMigrationSilent`). По возможности она использует запросы `ALTER TABLE`. Однако в `SQLite` `ALTER TABLE` имеет очень малые возможности, поэтому `Persistent` приходится прибегнуть к копированию данных из одной таблицы в другую.

Наконец, если вы хотите, чтобы вместо выполнения миграций `Persistent` дал вам подсказку по самостоятельному выполнению этих миграций, воспользуйтесь функцией `printMigration`. Эта функция выводит действия, которые были бы выполнены функцией `runMigration`. Это может быть полезно в случае выполнения миграций, на который `Persistent` не способен, выполнения дополнительных `SQL`-запросов во время миграций, или же просто для логирования происходящих миграций.

10.4. Уникальность

Помимо объявления полей у сущности мы также можем объявлять ограничение уникальности. Типичный пример — это требование уникальности имени пользователя:

```
User
  username Text
  UniqueUsername username
```

В то время, как имя каждого поля должно начинаться с маленькой буквы, ограничение уникальности должно начинаться с большой буквы.

```
{-# LANGUAGE QuasiQuotes, TypeFamilies, GeneralizedNewtypeDeriving,
      TemplateHaskell, OverloadedStrings, GADTs, FlexibleContexts #-}
import Database.Persist
import Database.Persist.Sqlite
import Database.Persist.TH
import Data.Time
import Control.Monad.IO.Class (liftIO)
import Control.Monad.Trans.Resource (runResourceT)

share [mkPersist sqlSettings, mkMigrate "migrateAll"] []
      [persistUpperCase]
Person
  firstName String
  lastName String
  age Int
  UniqueName firstName lastName
  deriving Show
]

main = runSqlite ":memory:" $ do
  runMigration migrateAll
  insert $ Person "Michael" "Snoyman" 26
  michael <- getBy $ UniqueName "Michael" "Snoyman"
  liftIO $ print michael
```

10/uniqueness.hs

Чтобы сообщить об уникальности комбинации нескольких полей, добавим одну дополнительную строку в наше определение. Persistent знает, что таким образом определяется уникальный конструктор, потому что строка начинается с заглавной буквы. Каждое последующее слово должно быть именем поля в сущности.

Главное ограничение, связанное с уникальностью, состоит в том, что она может использоваться только для непустых (non-null) полей. Причина заключается в том,

что стандарт SQL неоднозначен относительно уникальности пустых полей (например, `NULL=NULL` является истиной или ложью?). К тому же, в большинстве СУБД реализованы правила, которые не соответствуют правилам для соответствующих типов данных в Haskell (например, в PostgreSQL `NULL=NULL` — это ложь, а в Haskell `Nothing=Nothing` есть True).

В дополнение к предоставлению гарантий на уровне СУБД относительно согласованности данных, ограничение уникальности также может быть использовано для выполнения некоторых специфических запросов из кода на Haskell, как, например, в случае с `getBy`, продемонстрированном выше. Здесь используется ассоциативный тип `Uniqe`. В конце приведенного выше примера используется следующий конструктор:

```
UniqueName :: String -> String -> Unique Person
```

В случае использования MongoDB ограничение уникальности не может быть использовано — вы должны создать уникальный индекс по полю.

10.5. Запросы

В зависимости от вашей цели, могут быть использованы различные запросы к базе данных. В некоторых запросах используется численный ID, когда в других происходит фильтрация по значению поля. Запросы также различаются по количеству значений, которые они возвращают. Одни должны возвращать не более одного результата, другие же могут возвращать множество результатов.

В связи с этим Persistent предоставляет множество различных функций для выполнения запросов. Как обычно, мы стараемся закодировать с помощью типов столько инвариантов, сколько возможно. Например, если запрос может возвращать либо 0, либо 1 результат, используется обертка `Maybe`. Если же запрос может вернуть много результатов, возвращается список.

10.5.1. Выборка по ID

Простейший запрос, который может быть выполнен в Persistent — это выборка по ID. Поскольку в этом случае значение может существовать или не существовать, возвращаемое значение оборачивается в `Maybe`.

Использование функции `get`:

```
personId <- insert $ Person "Michael" "Snoyman" 26
maybePerson <- get personId
case maybePerson of
  Nothing -> liftIO $ putStrLn "Ничего нет"
  Just person -> liftIO $ print person
```

Это может быть очень удобно на сайтах, предоставляющих URL типа `/person/5`. Однако в таких случаях мы обычно не беспокоимся о `Maybe`, а просто хотим получить значение или вернуть код 404, если оно не найдено. К счастью, есть функция `get404`,

которая поможет нам в этом. Мы разберемся с этим вопросом более детально, когда дойдем до интеграции с Yesod.

10.5.2. Выборка по уникальному ключу

Функция `getBy` почти идентична `get`, только вместо ID она принимает значение `Unique`.

Использование функции `getBy`:

```
personId <- insert $ Person "Michael" "Snoyman" 26
maybePerson <- getBy $ UniqueName "Michael" "Snoyman"
case maybePerson of
  Nothing -> liftIO $ putStrLn "Ничего нет"
  Just person -> liftIO $ print person
```

Аналогично `get404`, также существует функция `getBy404`.

10.5.3. Другие функции выборки

Скорее всего, вам хотелось бы выполнять более сложные запросы, например, найти всех людей определенного возраста, все свободные машины синего цвета, всех пользователей без указанного email и тд. Для этого вам понадобится одна из следующих функций выборки.

Все эти функции имеют похожий интерфейс и немного различающиеся возвращаемые значения:

- **selectSource.** Возвращает источник (`source`), содержащий все ID и значения из базы данных. Это позволяет писать поточный код.

Мы более подробно рассмотрим источники в приложении, посвященном кондуитам (`conduits`). Кроме того, есть и другая функция под названием `selectSourceConn`, которая предоставляет больше контроля над выделением соединений. Мы рассмотрим ее в главе, посвященной работе со Sphinx.

- **selectList.** Возвращает список, содержащий все ID и значения из базы данных. Все записи будут помещены в память.
- **selectFirst.** Просто возвращает первый ID и первое значение из базы данных, если они есть.
- **selectKeys.** Возвращает только ключи, без значений, в качестве источника.

Чаще всего используется функция `selectList`, так что мы рассмотрим ее отдельно. После этого понять остальные функции будет проще простого.

Функция `selectList` принимает два аргумента: список `Filter`'ов и список `SelectOpt`'ов. Первый из них определяет ограничения, накладываемые на свойства сущностей, и позволяет использовать предикаты «равно», «меньше чем», «принадлежит множеству» и тп. `SelectOpt`'ы предоставляют три различных возможности —

сортировку, ограничение количества возвращаемых строк и смещение возвращаемого значения на заданное количество строк.

Комбинация из ограничения и смещения очень важна, она позволяет реализовать эффективное разбиение на страницы в вашем веб-приложении.

Сразу перейдем к примеру с фильтрацией, а затем проанализируем его:

```
people <- selectList [PersonAge >. 25, PersonAge <=. 30] []
liftIO $ print people
```

Несмотря на простоту примера, необходимо отметить три момента:

1. PersonAge является конструктором ассоциативного фантомного типа. Звучит ужасающе, однако действительно важно лишь то, что он однозначно определяет столбец «age» таблицы «person», а также знает, что возраст на самом деле является Int'ом. (В этом и состоит его фантомность.)
2. Мы имеем дело с группой фильтрующих операторов пакета Persistent. Они довольно прямолинейны и делают в точности то, что вы от них ожидаете. Однако тут есть три тонких момента, которые я объясню ниже.
3. Список фильтров объединяется логическим И, то есть, ограничение следует читать, как «возраст больше 25-и И возраст меньше или равен 30-и». Использование логического ИЛИ мы рассмотрим ниже.

Также имеется оператор с удивительным названием «не равно». Мы используем обозначение !=., поскольку /=. используется при UPDATE-запросах (ради «разделяй-и-устанавливай», о котором я расскажу позже). Не беспокойтесь, если вы воспользуетесь неверным оператором, компилятор предупредит вас. Еще два удивительных оператора — это «принадлежит множеству» и «не принадлежит множеству». Они обозначаются, соответственно, <-. и /<-. (оба с точкой на конце).

Что же касается логического ИЛИ, для него есть оператор ||.. Например:

```
people <- selectList
  (
    [PersonAge >. 25, PersonAge <=. 30]
    ||. [PersonFirstName /<-. ["Adam", "Bonny"]]
    ||. ([PersonAge ==. 50] ||. [PersonAge ==. 60])
  )
  []
liftIO $ print people
```

Этот (совершенно нелепый) пример означает «найти людей, чей возраст составляет от 26-и до 30-и лет включительно ИЛИ чье имя не Адам и не Бонни ИЛИ чей возраст — 50 или 60 лет».

SelectOpt

Все наши вызовы selectList имели пустой список в качестве второго аргумента. Это не задает никаких параметров и означает «сортируй на усмотрение СУБД, возвращай

все результаты, не пропускай никаких результатов». `SelectOpt` имеет четыре конструктора, которые могут быть использованы для изменения этого поведения:

- **Asc**. Сортировать по заданному столбцу в неубывающем порядке. Тут используется такой же фантомный тип, как и при фильтрации, например `PersonAge`.
- **Desc**. Аналогично `Asc`, только в невозрастающем порядке.
- **LimitTo**. Принимает аргумент типа `Int`. Вернуть не более указанного количества результатов.
- **OffsetBy**. Также принимает аргумент типа `Int`. Пропустить указанное количество результатов.

В следующем отрывке кода определяется функция, которая разбивает результат на страницы. Она возвращает всех людей старше 18-и лет, отсортированных по возрасту (более старшие идут первыми). Люди с одинаковым возрастом сортируются по фамилиям, а затем по именам.

```
resultsForPage pageNumber = do
  let resultsPerPage = 10
  selectList
    [ PersonAge >= . 18
    ]
    [ Desc PersonAge
    , Asc PersonLastName
    , Asc PersonFirstName
    , LimitTo resultsPerPage
    , OffsetBy $ (pageNumber - 1) * resultsPerPage
    ]
```

10.6. Другие действия с данными

Извлечение данных — это только полдела. Нам также необходимо иметь возможность добавлять данные и модифицировать данные, находящиеся в базе.

10.6.1. Вставка

Иметь возможность работать с данными из базы — это здорово и замечательно, но как эти данные туда попадут? Для этого есть функция `insert`. Вы просто передаете ей значение, а она возвращает ID.

В связи с этим имеет смысл немного пояснить философию `Persistent`. Во многих ORM типы, используемые для работы с данными, непрозрачны. Вам приходится продирааться через определяемый ими интерфейс, чтобы получить, а затем изменить данные. Однако в `Persistent` все иначе — для всего используются старые добрые алгебраические типы данных. Таким образом, вы по-прежнему можете иметь огромный выигрыш от использования сопоставления с образцом, каррирования и всего остального, к чему вы привыкли.

Однако есть вещи, которые мы не можем делать. Например, нет способа автоматически обновлять данные в базе данных при каждом их изменении в Haskell. Конечно, учитывая позицию языка Haskell в отношении чистоты и неизменяемости, от этого все равно было бы мало проку, так что не будем лить слезы.

Тем не менее, есть момент, который часто беспокоит новичков. Почему ID и значения совершенно разделены? Казалось бы, куда логичнее было бы включить ID в само значение. Другими словами, вместо:

```
data Person = Person { name :: String }
```

... мы имели бы:

```
data Person = Person { personId :: PersonId, name :: String }
```

Одна из проблем сразу бросается в глаза. Как прикажете производить вставку? Если Person требуется ID, а ID возвращается функцией insert, которой в свою очередь требуется Person, мы получаем проблему курицы и яйца. Мы могли бы решить эту проблему, используя неопределенный ID, однако это верный способ нарваться на неприятности.

Вы скажете, хорошо, давайте попробуем что-то более безопасное:

```
data Person = Person { personId :: Maybe PersonId, name :: String }
```

Намного предпочтительнее писать `insert $ Person Nothing "Michael"` вместо `insert $ Person undefined "Michael"`. И наши типы стали намного проще, не так ли? Например, `selectList` теперь может возвращать просто `[Person]` вместо уродливого `[Entity SqlPersist Person]`.

Entity представляет собой тип данных, который связывает ID и значение сущности воедино. Поскольку ID могут быть разными в зависимости от бэкенда, необходимо также предоставить используемый бэкенд пакета Persistent. Тип данных `Entity SqlPersist Person` следует читать как «ID и значение некого человека, хранящиеся в SQL базе данных».

Проблема заключается в том, что «уродство» оказывается невероятно полезным. Запись `Entity SqlPersist Person` делает очевидным тот факт, что мы работаем со значением, которое существует в базе данных. Допустим, мы хотим создать URL, в котором присутствует `PersonId` (не такой уж редкий случай, как мы вскоре выясним). `Entity SqlPersist Person` недвусмысленно предоставляет доступ к требуемой информации. Тем временем, использование обертки `Maybe` приводит к потребности в дополнительных проверках во время выполнения, вместо того, чтобы убедиться в корректности программы еще на этапе компиляции.

Наконец, в случае присоединения ID к значению имеет место семантическое несоответствие. `Person` — это значение. Два человека являются идентичными (с точки зрения базы данных), если все их поля одинаковы. Присоединяя ID к значению, мы начинаем говорить не о человеке, а о строке в базе данных. Равенство перестает быть равенством, оно превращается в идентичность: «это тот же самый человек» вместо «это такой же человек».

Другими словами, есть нечто раздражающее в отделении ID от значения, но в конце концов, это правильный подход, который в великой схеме вещей ведет к более хорошему, менее дырявому коду.

10.6.2. Обновление

Теперь подумаем об обновлении в контексте нашего обсуждения. Вот простейший способ сделать обновление:

```
let michael = Person "Michael" 26
    michaelAfterBirthday = michael { personAge = 27 }
```

Однако в действительности этот код ничего не обновляет. Он просто создает новое значение типа Person, основанное на старом значении. Когда мы говорим об обновлении, мы имеем ввиду не модификацию значений в Haskell. (И вправду, не стоило бы этого делать, поскольку данные в Haskell неизменяемы.)

На самом деле, мы ищем способ изменить строки в таблице. И проще всего сделать это с помощью функции update:

```
personId <- insert $ Person "Michael" "Snoyman" 26
update personId [PersonAge =. 27]
```

Функция update принимает два аргумента: ID и список Update'ов. Простейший способ обновления поля заключается в присвоении ему нового значения, однако этот способ не лучший. Что, если вы хотите увеличить чей-то возраст на единицу, но текущий возраст вам не известен? В Persistent предусмотрено и это:

```
haveBirthday personId = update personId [PersonAge +=. 1]
```

Как и следовало ожидать, в нашем распоряжении есть все базовые математические операторы: += ., -= ., *= . и /= . . Они не только удобны для обновления единичной записи, но и необходимы для соблюдения гарантий ACID. Представьте, что бы мы делали без этих операторов. Нам приходилось бы извлекать из базы Person, увеличивать возраст, а затем обновлять значение в базе данных на новое. Как только у вас появляется два процесса или потока, одновременно работающих с базой данных, вы попадаете в мир боли (подсказка: состояние гонки).

Иногда мы хотим обновить несколько строк одновременно (например, повысить зарплату на 5% всем сотрудникам). Функция updateWhere принимает два аргумента: список фильтров и список обновлений, которые следует применить.

```
updateWhere [PersonFirstName ==. "Michael"] [PersonAge *=. 2] -- []
это был длинный день
```

Иногда хочется просто заменить одно значение в базе данных на другое. Для этого (сюрприз!) есть функция replace:

```
personId <- insert $ Person "Michael" "Snoyman" 26
```

```

replace personId $ Person "John" "Doe" 20
update personId [PersonAge =. 27]
haveBirthday personId
updateWhere [PersonFirstName ==. "Michael"] [PersonAge *.=. 2] -- []
это был длинный день
resultsForPage 1 >>= liftIO . print

```

10.6.3. Удаление

Как ни печально, иногда мы вынуждены расстаться с нашими данными. Для этого у нас есть аж три функции:

<code>delete</code>	Удалить по ID
<code>deleteBy</code>	Удалить по уникальному ключу
<code>deleteWhere</code>	Удалить по множеству фильтров

```

personId <- insert $ Person "Michael" "Snoyman" 26
delete personId
deleteBy $ UniqueName "Michael" "Snoyman"
deleteWhere [PersonFirstName ==. "Michael"]

```

С помощью `deleteWhere` мы можем удалить вообще все данные из таблицы. Нужно только подсказать GHC, какая таблица нас интересует:

```
deleteWhere ([] :: [Filter Person])
```

10.7. Атрибуты

До сих пор мы видели базовый синтаксис для наших `persist`-блоков — строка с именем сущности, за которой для каждого поля идет по одной строке с отступами, состоящей из двух слов, имени поля и типа данных поля. `Persistent` поддерживает не только это. После двух слов в строке вы можете указать произвольный список атрибутов.

Допустим, вы хотите, чтобы сущность `Person` имела необязательный возраст, а также время добавления его или ее в систему. Для сущностей, уже находящихся в базе данных, в качестве этого времени мы хотим использовать текущее время.

```

{-# LANGUAGE QuasiQuotes, TypeFamilies, GeneralizedNewtypeDeriving, []
    TemplateHaskell, OverloadedStrings, GADTs, FlexibleContexts #-}

import Database.Persist
import Database.Persist.SQLite
import Database.Persist.TH
import Data.Time
import Control.Monad.IO.Class

```

```

share [mkPersist sqlSettings, mkMigrate "migrateAll"] []
      [persistUpperCase]
Person
  name String
  age Int Maybe
  created UTCTime default=CURRENT_TIME
  deriving Show
]

main = runSqlite ":memory:" $ do
  time <- liftIO getCurrentTime
  runMigration migrateAll
  insert $ Person "Michael" (Just 26) time
  insert $ Person "Greg" Nothing time

```

10/attributes.hs

Maybe является встроенным атрибутом из одного слова. Он делает поле необязательным. Это означает, что в Haskell данное поле будет обернуто в Maybe, а в SQL оно может иметь значение NULL.

Атрибут default зависит от используемого бэкенда и может использовать любой синтаксис, лишь бы он был понятен СУБД. В приведенном примере используется встроенная функция СУБД CURRENT_TIME. Допустим, теперь мы хотим добавить в сущность Person поле с любимым языком программирования:

```

{-# LANGUAGE QuasiQuotes, TypeFamilies, GeneralizedNewtypeDeriving, []
      TemplateHaskell, OverloadedStrings, GADTs, FlexibleContexts #-}

import Database.Persist
import Database.Persist.Sqlite
import Database.Persist.TH
import Data.Time

share [mkPersist sqlSettings, mkMigrate "migrateAll"] []
      [persistUpperCase]
Person
  name String
  age Int Maybe
  created UTCTime default=CURRENT_TIME
  language String default='Haskell'
  deriving Show
]

main = runSqlite ":memory:" $ do
  runMigration migrateAll

```

10/attribute-default.hs

Атрибут `default` абсолютно никак не затрагивает код на Haskell, то есть, вам по-прежнему придется заполнять все значения. Атрибут влияет только на схему базы данных и автоматические миграции.

Мы должны окружить строку в одинарные кавычки, чтобы СУБД могла правильно интерпретировать ее. Также `Persistent` позволяет использовать двойные кавычки для строк, содержащих пробелы. Например, если мы хотим сделать страной по умолчанию Российскую Федерацию, то должны написать:

```
{-# LANGUAGE QuasiQuotes, TypeFamilies, GeneralizedNewtypeDeriving,
  TemplateHaskell, OverloadedStrings, GADTs, FlexibleContexts #-}

import Database.Persist
import Database.Persist.Sqlite
import Database.Persist.TH
import Data.Time

share [mkPersist sqlSettings, mkMigrate "migrateAll"] []
      [persistUpperCase]
Person
  name String
  age Int Maybe
  created UTCTime default=CURRENT_TIME
  language String default='Haskell'
  country String "default='Российская_Федерация'"
  deriving Show
]

main = runSqlite ":memory:" $ do
  runMigration migrateAll
```

10/attribute-whitespace.hs

Последний трюк, который вы можете проделать с атрибутами — это указать имена таблиц и столбцов, используемые в SQL. Это может пригодиться при взаимодействии с существующими базами данных.

```
share [mkPersist sqlSettings, mkMigrate "migrateAll"] [persist|
Person sql=the-person-table
  firstName String sql=first_name
  lastName String sql=fldLastName
  age Int "sql=The_Age_of_the_Person"
  UniqueName firstName lastName
  deriving Show
```

[]

10.8. Отношения

Persistent поддерживает ссылки между типами данных, таким образом, что они остаются согласованными в поддерживаемых NoSQL базах данных. Ссылка создается путем добавления ID в связанную сущность. Вот как выглядит пример для человека с большим количеством машин:

```
{-# LANGUAGE QuasiQuotes, TypeFamilies, GeneralizedNewtypeDeriving, []
      TemplateHaskell, OverloadedStrings, GADTs, FlexibleContexts #-}

import Database.Persist
import Database.Persist.Sqlite
import Database.Persist.TH
import Control.Monad.IO.Class (liftIO)
import Data.Time

share [mkPersist sqlSettings, mkMigrate "migrateAll"] []
      [persistUpperCase]
Person
  name String
  deriving Show
Car
  ownerId PersonId Eq
  name String
  deriving Show
]

main = runSqlite ":memory:" $ do
  runMigration migrateAll
  bruce <- insert $ Person "Bruce_Wayne"
  insert $ Car bruce "Bat_Mobile"
  insert $ Car bruce "Porsche"
  -- это может занять много времени
  cars <- selectList [CarOwnerId ==. bruce] []
  liftIO $ print cars
```

10/relations.hs

С помощью этой техники вы можете определять отношения один-ко-многим. Чтобы определить отношение многие-ко-многим, нам понадобится связующая сущность, которая имеет отношения один-ко-многим с каждой из оригинальных таблиц. В данном случае будет хорошей идеей воспользоваться ограничением уникальности. Допустим,

мы хотим смоделировать ситуацию, в которой мы отслеживаем, какие люди в каких магазинах делают покупки:

```
{-# LANGUAGE QuasiQuotes, TypeFamilies, GeneralizedNewtypeDeriving, []
   TemplateHaskell, OverloadedStrings, GADTs, FlexibleContexts #-}

import Database.Persist
import Database.Persist.Sqlite
import Database.Persist.TH
import Data.Time

share [mkPersist sqlSettings, mkMigrate "migrateAll"] []
[persistUpperCase]
Person
  name String
Store
  name String
PersonStore
  personId PersonId
  storeId StoreId
  UniquePersonStore personId storeId
[]

main = runSqlite ":memory:" $ do
  runMigration migrateAll

  bruce <- insert $ Person "Bruce_Wayne"
  michael <- insert $ Person "Michael"

  target <- insert $ Store "Target"
  gucci <- insert $ Store "Gucci"
  sevenEleven <- insert $ Store "7-11"

  insert $ PersonStore bruce gucci
  insert $ PersonStore bruce sevenEleven

  insert $ PersonStore michael target
  insert $ PersonStore michael sevenEleven
```

Поскольку суффикс `Id` в имени типа используется в `Persistent` для обозначения связи по внешнему ключу, в настоящий момент не представляется возможным использовать неключевые типы, чье имя заканчивается на `Id`. Простое решение этой проблемы заключается в определении синонима типа с другим суффиксом, например:

```
data MyExistingTypeEndingInId = ...
type IdIsNotTheSuffix = MyExistingTypeEndingInId
[persist|
Person
    someField IdIsNotTheSuffix
```

10.9. Подробнее о типах

До сих пор мы говорили о `Person` и `PersonId` без особого объяснения, чем они на самом деле являются. В простейшем случае, если мы говорим только о реляционных базах данных, `PersonId` мог бы быть просто `type PersonId = Int64`. Но в этом случае на уровне типов ничто не связывало бы `PersonId` и сущность `Person`. В результате вы могли бы по ошибке воспользоваться `PersonId` для получения `Car`. Для моделирования таких отношений мы используем фантомные типы. Итак, наш следующий наивный шаг был бы следующим:

```
newtype Key entity = Key Int64
type PersonId = Key Person
```

До `Persistent 0.6` мы использовали ассоциативные типы вместо фантомных типов. Таким способом также можно решить проблему, но фантомы справляются с ней лучше.

И это действительно отлично работает до тех пор, пока вы не столкнетесь с бэкендом, который не использует `Int64` в качестве ID. И это далеко не теоретический вопрос, поскольку `MongoDB` для этих целей использует тип `ByteString`. Итак, нам нужно значение ключа, которое может содержать либо `Int64`, либо `ByteString`. Кажется, настало отличное время для применения суммарных типов:

```
data Key entity = KeyInt Int64 | KeyByteString ByteString
```

Но на самом деле мы только ищем неприятности. В следующий раз нам попадется бэкенд, который использует в качестве ключа временные метки и нам придется ввести дополнительный конструктор для `Key`. Так может продолжаться какое-то время. К счастью, у нас уже есть суммарный тип, предназначенный для представления произвольных данных, `PersistValue`:

```
newtype Key entity = Key PersistValue
```


Но тут есть другая проблема. Скажем, у нас есть веб-приложение, которое принимает ID в качестве параметра от пользователя. Этому приложению придется принимать параметр, как Text, а затем пытаться преобразовать его в Key. Нет проблем, давайте напишем функцию, которая преобразовывает Text в PersistValue, а затем передадим возвращаемое ее значение в конструктор Key. Правильно?

Нет, неправильно. Мы пробовали, и это оказалось большим геморроем. Все закончилось тем, что нам пришлось принимать ключи, которых не может быть. Например, если мы имеем дело с SQL, ключ должен быть целым числом. Но при подходе, описанном выше, в качестве ключа мы вынуждены принимать произвольные текстовые данные. В результате мы получали 500-ые ошибки, потому что СУБД была в шоке от попыток сравнивать целочисленные поля с текстом.

Что нам действительно нужно, это способ преобразования текста в Key, но с учетом правил используемого бэкенда. И как только вопрос становится сформулирован таким образом, мы тут же получаем ответ — добавить еще одного фантома. В действительности, определение Key в Persistent следующее:

```
newtype Key backend entity = Key { unKey :: PersistValue }
```

И это прекрасно работает. Теперь мы можем получить функцию Text -> Key MongoDB entity и функцию Text -> Key SqlPersist entity, после чего все работает, как по маслу. Но теперь есть другая проблема — отношения. Скажем, мы хотим представлять блоги и посты в блогах. Будем использовать следующее определение сущностей:

```
Blog
  title Text
Post
  title Text
  blogId BlogId
```

Но как это будет выглядеть с точки зрения типа данных Key?

```
data Blog = Blog { blogTitle :: Text }
data Post = Post { postTitle :: Text, postBlogId :: Key <что должно быть здесь?> Blog }
```

Мы должны указать какой-то бэкенд. В теории, мы можем захардкодить SqlPersist или Mongo, но тогда наши типы данных будут работать только с одним бэкендом. Для одного приложения это может быть приемлемым, но как насчет библиотек с определениями типов данных, которые могут использоваться в различных приложениях с различными бэкендами?

Так что, все становится чуть сложнее. На самом деле, наши типы такие:

```
data BlogGeneric backend = Blog { blogTitle :: Text }
data PostGeneric backend = Post { postTitle :: Text, postBlogId :: Key backend (BlogGeneric backend) }
```

Обратите внимание, что мы все еще используем короткие имена для конструкторов и записей. Наконец, чтобы предоставить простой интерфейс для нормального кода, мы определяем кое-какие синонимы типов:

```
type Blog = BlogGeneric SqlPersist
type BlogId = Key SqlPersist Blog
type Post = PostGeneric SqlPersist
type PostId = Key SqlPersist Post
```

И нет, `SqlPersist` не захардкожен где бы то ни было в `Persistent`. Это параметр `sqlSettings`, что вы передали в `mkPersist`, говорит нам использовать `SqlPersist`. В коде, использующем `MongoDB`, вместо него будет использован параметр `mongoSettings`.

Описанное выше может показаться несколько сложным, но при написании пользовательского кода все это едва ли когда-нибудь вам понадобится. Просмотрите еще раз эту главу — нам ни разу не приходилось работать с `Key` или `Generic` напрямую. Единственное место, где они могут всплыть, это сообщения компилятора об ошибках. Так что в целом информация из этого раздела вам пригодятся, но вряд ли ею придется пользоваться каждый день.

10.10. Поля произвольного типа

У вас может возникнуть желание использовать в вашем хранилище поля произвольного типа. Наиболее типичный случай — это перечисление, например состояние найма сотрудников. Для этого `Persistent` предоставляет специальную функцию `Template Haskell`:

```
{-# LANGUAGE QuasiQuotes, TypeFamilies, GeneralizedNewtypeDeriving, []
      TemplateHaskell, OverloadedStrings, GADTs, FlexibleContexts #-}

import Database.Persist
import Database.Persist.SQLite
import Database.Persist.TH

data Employment = Employed | Unemployed | Retired
      deriving (Show, Read, Eq)
derivePersistField "Employment"

share [mkPersist sqlSettings, mkMigrate "migrateAll"] []
      [persistUpperCase]
Person
      name String
      employment Employment
[]

main = runSqlite ":memory:" $ do
      runMigration migrateAll
```

```
insert $ Person "Bruce_Wayne" Retired
insert $ Person "Peter_Parker" Unemployed
insert $ Person "Michael" Employed
```

10/custom-fields.hs

`derivePersistField` позволяет хранить данные в базе, используя строковые поля, а также реализует сериализацию с помощью экземпляров классов `Read` и `Show` вашего типа данных. Это может быть не так эффективно, как использование целых чисел, зато удобнее вот в каком плане. Даже если в будущем вы добавите новые конструкторы, данные в базе останутся валидными.

10.11. Persistent: сырой SQL

Пакет `Persistent` предоставляет типобезопасный интерфейс к хранилищу данных. Он старается не зависеть от используемого бэкенда, например, не полагаясь на реляционные возможности SQL. По моему опыту в 95% случаев вы с легкостью решите стоящую перед вами задачу, используя высокоуровневый интерфейс. (В действительности, большинство моих веб-приложений используют только высокоуровневый интерфейс.)

Но иногда возникает желание использовать возможности, специфичные для конкретного бэкенда. Одной из таких возможностей, которую мне приходилось использовать, был полнотекстовый поиск. В этом случае в SQL-запросе требуется использовать `LIKE`, который не моделируется в `Persistent`. Давайте попробуем найти всех людей с фамилией «Споупан» и вывести найденные записи.

На самом деле, вы можете воспользоваться оператором `LIKE` с помощью нормального синтаксиса, поскольку в `Persistent 0.6` была добавлена возможность, позволяющая использовать операторы, специфичные для бэкенда. Но это все равно хороший пример, так что давайте рассмотрим его.

```
{-# LANGUAGE OverloadedStrings, TemplateHaskell, QuasiQuotes, []
     TypeFamilies #-}
{-# LANGUAGE GeneralizedNewtypeDeriving, GADTs, FlexibleContexts #-}

import Database.Persist.Sqlite (runSqlite)
import Database.Persist.TH (mkPersist, persistUpperCase, share, []
                             mkMigrate, sqlSettings)
import Database.Persist.GenericSql (runSqlConn, runMigration, []
                                     SqlPersist)
import Database.Persist.GenericSql.Raw (withStmt)
import Data.Text (Text)
import Database.Persist
import Database.Persist.Store (PersistValue)
import Control.Monad.IO.Class (liftIO)
import qualified Data.Conduit as C
```

```

import qualified Data.Conduit.List as CL

share [mkPersist sqlSettings, mkMigrate "migrateAll"] []
      [persistUpperCase]
Person
  name Text
[]

main :: IO ()
main = runSqlite ":memory:" $ do
  runMigration migrateAll
  insert $ Person "Michael_Snoyman"
  insert $ Person "Miriam_Snoyman"
  insert $ Person "Eliezer_Snoyman"
  insert $ Person "Gavriella_Snoyman"
  insert $ Person "Greg_Weber"
  insert $ Person "Rick_Richardson"

  -- Persistent не предоставляет ключевого слова LIKE, но нам []
  -- хотелось бы
  -- получить всю семью Snoyman'ов...
  let sql = "SELECT name FROM Person WHERE name LIKE '%Snoyman'"
  C.runResourceT $ withStmt sql []
      C.$$ CL.mapM_ $ liftIO . print

```

10/raw-sql.hs

Также существует высокоуровневая поддержка сериализации. Подробности вы можете найти в Haddock-документации по API.

10.12. Интеграция с Yesod

Итак, вы прониклись всей мощью Persistent. Как теперь интегрировать его с Yesod-приложением? Если вы использовали автоматическую генерацию шаблона приложения, большая часть работы уже была проделана за вас. Но, по традиции, сейчас мы проделаем все вручную, чтобы лучше понять, как все устроено.

Пакет `yesod-persistent` представляет собой «клей» между Persistent и Yesod. Он предоставляет класс типов `YesodPersist`, который стандартизует доступ к базе данных с помощью `runDB`. Вот как это выглядит в действии.

```

{-# LANGUAGE QuasiQuotes, TypeFamilies, GeneralizedNewtypeDeriving, []
      FlexibleContexts #-}
{-# LANGUAGE TemplateHaskell, OverloadedStrings, GADTs, []
      MultiParamTypeClasses #-}

import Yesod

```

```

import Database.Persist.Sqlite
import Control.Monad.Trans.Resource (runResourceT)
import Control.Monad.Logger (runNoLoggingT)

-- Определяем наши сущности, как обычно
share [mkPersist sqlSettings, mkMigrate "migrateAll"] []
  [persistUpperCase|
Person
  firstName String
  lastName String
  age Int
  deriving Show
  ]

-- Мы являемся держателями пула соединений. Когда программа []
  инициализируется, мы
-- создаем начальный пул, и каждый раз, когда нам нужно произвести []
  действие,
-- мы выделяем соединение из пула
data PersistTest = PersistTest ConnectionPool

-- Мы создаем один-единственный маршрут для доступа к человеку. Это []
  довольно распространенная
-- практика, когда в маршрутах используется Id.
mkYesod "PersistTest" [parseRoutes]
/person/#PersonId PersonR GET
  ]

-- Тут ничего особенного
instance Yesod PersistTest

-- Теперь нам нужно определить экземпляр класса Yesod Persist, []
  который будет следить
-- за тем, какой бэкенд мы используем и как следует выполнять действия
instance YesodPersist PersistTest where
  type YesodPersistBackend PersistTest = SqlPersist

  runDB action = do
    PersistTest pool <- getYesod
    runSqlPool action pool

-- Мы просто возвращаем строковое представление человека
-- или ошибку 404, если такой Person не существует
getPersonR :: PersonId -> Handler RepPlain
getPersonR personId = do
  person <- runDB $ get404 personId

```

```

    return $ RepPlain $ toContent $ show person

openConnectionCount :: Int
openConnectionCount = 10

main :: IO ()
main = runNoLoggingT $ runResourceT $ withSqlitePool "test.db3" \
    openConnectionCount $ \pool -> do
    runSqlPool (runMigration migrateAll) pool
    runSqlPool (insert $ Person "Michael" "Snoyman" 26) pool
    liftIO $ warpDebug 3000 $ PersistTest pool

```

10/yesod.hs

Тут есть два важных момента. Для выполнения действий над базой данных в обработчике используется `runDB`. Внутри `runDB` вы можете использовать все те функции, о которых шла речь выше, например, `insert` и `selectList`.

```

runDB имеет тип runDB :: YesodDB sub master a -> GHandler sub \
master a. YesodDB определен, как:

```

```

type YesodDB sub master = YesodPersistBackend master \
(GHandler sub master)

```

Поскольку он построен на ассоциативном типе `YesodPersistBackend`, используется подходящий для текущего сайта бэкенд.

Другая новая фишка — это `get404`. Эта функция работает в точности, как `get`, только вместо того, чтобы возвращать `Nothing`, когда результат не может быть найден, она возвращает страницу с сообщением об ошибке 404. В функции `getPersonR` использован очень распространенный в реальных приложениях на Yesod подход — значение получается через функцию `get404`, а затем в зависимости от результата возвращается ответ.

10.13. Выводы

Persistent приносит строгую типизацию языка Haskell в ваш слой доступа к данным. Вместо того, чтобы писать склонный к возникновению ошибок, нетипизированный доступ к данным, или вручную писать шаблонный код сериализации, вы можете автоматизировать работу, используя Persistent.

Цель состоит в том, чтобы большую часть времени предоставлять все необходимое. В тех же случаях, когда вам нужно более мощное средство, Persistent предоставляет прямой доступ к нижележащему хранилищу данных, так что вы можете написать любое пятистороннее объединение таблиц, какой пожелаете.

Persistent напрямую интегрируется в общий рабочий процесс с Yesod. И речь тут идет не только о небольших пакетах вроде `yesod-persistent` — пакеты вроде `yesod-form` и `yesod-auth` также прекрасно взаимодействуют с Persistent.

Глава 11

Развёртывание вашего веб-приложения

Не могу говорить за всех, но лично я предпочитаю программирование системному администрированию. Но, на самом деле, рано или поздно вам придётся как-нибудь разворачивать ваше приложение, и скорее всего вам придётся быть тем самым человеком, который будет всё это настраивать.

В Haskell сообществе существуют многообещающие инициативы, нацеленные на то, чтобы облегчить процесс развёртывания. В будущем, возможно, у нас даже будет сервис, который позволит развернуть ваше веб-приложение при помощи одной команды.

Однако пока что это не так, и, даже если бы это было так, подобное решение никогда бы не подошло абсолютно всем. Данная глава описывает различные существующие варианты развёртывания и даёт некоторые общие рекомендации касательно того, какой из них следует выбрать в том или ином случае.

11.1. Компиляция

Начнём с начала: как вы собираете своё приложение? Если вы используете сгенерированный сайт-каркас — это просто вызов команды `cabal build`. Я бы также рекомендовал очистку рабочих каталогов перед сборкой, чтобы гарантировать отсутствие закешированной информации. Таким образом, простейшая последовательность команд для сборки вашего приложения будет выглядеть так:

```
cabal clean && cabal configure && cabal build
```

11.2. Warp

Как было отмечено ранее, Yesod построен на *Интерфейсе веб-приложения (Web Application Interface, WAI)*, что позволяет ему запускаться на любом WAI-совместимом

бэкенд сервере. На момент написания этого текста доступны следующие варианты:

- Warp
- FastCGI
- SCGI
- CGI
- Webkit
- Сервер разработки (development server)

Последние два не предназначены для боевого использования. Из оставшихся четырёх, теоретически, для этих целей может быть использован любой. На практике же CGI будет ужасно неэффективен, так как для каждого соединения будет создаваться новый процесс. SCGI, в свою очередь, не настолько хорошо поддерживается фронтенд веб-серверами как Warp (в режиме обратного прокси) или FastCGI.

Таким образом, из двух оставшихся вариантов Warp является рекомендуемым, потому что:

- Он значительно быстрее;
- Как и FastCGI может быть запущен за фронтенд сервером, таким как Nginx в режиме обратного прокси;
- Кроме того, он является полностью функциональным сервером, и потому может быть использован сам по себе, без какого-либо фронтенда.

Итого, открытым остался лишь один вопрос: следует ли использовать Warp самостоятельно или через обратный прокси за другим фронтенд сервером? В большинстве случаев я рекомендую последнее, потому что:

- Несмотря на то, что Warp быстр, он оптимизирован для использования в качестве сервера приложений, а не сервера статических файлов;
- Используя Nginx, вы можете настроить виртуальный хостинг и раздавать статические файлы с отдельного домена. (Это возможно и с Warp, хоть и несколько сложнее);
- Вы можете использовать Nginx как балансировщик нагрузки или SSL прокси. (Хотя, используя warp-tls, вполне можно запускать https сайт на одном только Warp).

Таким образом, моя итоговая рекомендация следующая: используйте Nginx как обратный прокси для Warp.

Некоторые участники сообщества Yesod не согласны со мной в этом вопросе и полагают, что увеличенная производительность и уменьшенная сложность решения с одним только Warp делают его лучшим выбором. Так что, поступайте как считаете правильным, оба подхода абсолютно корректны.

11.3. Конфигурация

В общем случае, Nginx будет ожидать соединений на 80 порту, а ваше Yesod/Warp приложение будет слушать какой-нибудь другой непривилегированный порт (скажем, 4321). В таком случае ваш файл `nginx.conf` будет выглядеть примерно так:

```
daemon off; # НЕ запускайте nginx в фоновом режиме. Это очень удобно []
            для мониторинга приложений
events {
    worker_connections 4096;
}

http {
    server {
        listen 80; # Порт для входящих соединений Nginx
        server_name www.myserver.com;
        location / {
            proxy_pass http://127.0.0.1:4321; # Порт обратного []
            прокси для вашего Yesod приложения
        }
    }
}
```

Вы можете добавлять сколько угодно блоков `server`. Общая практика — добавлять блок, который гарантирует, что ваши пользователи всегда будут получать страницы с префиксом `www` в имени домена, соблюдая таким образом RESTful принцип канонических URLов. (Вы можете с тем же успехом делать обратное и всегда удалять префикс `www`. Главное, не забудьте отразить это и в файле конфигурации `nginx` и в `approot` вашего сайта).

```
server {
    listen 80;
    server_name myserver.com;
    rewrite ^/(.*) http://www.myserver.com/$1 permanent;
}
```

Крайне рекомендуется в качестве оптимизации раздавать статические файлы с отдельного домена, избегая таким образом ненужной передачи `cookie`. Предполагая, что наши статические файлы хранятся в подкаталоге `static` каталога нашего сайта, который имеет путь `/home/michael/sites/mysite`, это выглядело бы следующим образом:

```
server {
    listen 80;
    server_name static.myserver.com;
    root /home/michael/sites/mysite/static;
    # Так как yesod-static добавляет хэш содержимого к строке запроса,
```

```

# мы можем свободно выставлять дату истечения срока действия на
# далёкое будущее
# без опасений получить устаревшее содержимое.
expires max;
}

```

Для того, чтобы это заработало, ваш сайт должен корректно изменять статические URLы так, чтобы они указывали на этот альтернативный домен. Сайт-каркас настроен таким образом, чтобы упростить эту задачу используя функцию `Settings.staticRoot` и определение `urlRenderOverride`. Однако если вам надо всего лишь использовать преимущества Nginx при раздаче статических файлов без использования отдельных доменных имён, вы можете просто модифицировать исходный блок `server` следующим образом:

```

server {
    listen 80; # Порт для входящих соединений Nginx
    server_name www.myserver.com;
    location / {
        proxy_pass http://127.0.0.1:4321; # Порт обратного прокси
        # для вашего Yesod приложения
    }
    location /static {
        root /home/michael/sites/mysite; # Обратите внимание, что мы
        *НЕ* включаем /static
        expires max;
    }
}

```

11.4. Серверный процесс

Многие знакомы с конфигурациями типа Apache/mod_php или Lighttpd/FastCGI, в которых веб-сервер автоматически запускает веб-приложение. При использовании nginx, как в режиме обратного прокси, так и в режиме FastCGI, это не так: вы сами ответственны за запуск вашего процесса. Я настоятельно рекомендую использовать какой-либо инструмент мониторинга, который будет автоматически перезапускать ваше приложение в случае падений. Существует множество отличных вариантов для этого, таких как `angel` или `daemontools`.

В качестве конкретного примера ниже приведён файл конфигурации Upstart. Он должен располагаться в `/etc/init/mysite.conf`:

```

description "My_awesome_Yesod_application"
start on runlevel [2345];
stop on runlevel [!2345];
respawn
chdir /home/michael/sites/mysite

```

```
exec /home/michael/sites/mysite/dist/build/mysite/mysite
```

Как только этот файл на месте, запуск вашего приложения потребует всего лишь одной команды `sudo start mysite`.

11.5. FastCGI

Некоторые предпочитают использовать FastCGI для развёртывания своих приложений. В этом случае вам потребуется ещё один инструмент. FastCGI работает, принимая новые соединения из файлового дескриптора. Стандартная библиотека C предполагает, что этот файловый дескриптор будет равен 0 (стандартный поток ввода), а значит, надо использовать специальную программу `spawn-fcgi`, которая свяжет стандартный поток ввода вашего приложения с правильным сокетом.

Может оказаться очень удобным использовать для этих целей именованные сокеты Unix вместо привязки к порту, особенно если вы собираетесь запускать несколько приложений на одном физическом сервере. Возможный сценарий для запуска вашего приложения может выглядеть следующим образом:

```
spawn-fcgi \
  -d /home/michael/sites/mysite \
  -s /tmp/mysite.socket \
  -n \
  -M 511 \
  -u michael \
  -- \
  /home/michael/sites/mysite/dist/build/mysite-fastcgi/mysite-fastcgi
```

Вам также потребуется сконфигурировать ваш фронтенд сервер для общения с вашим приложением через FastCGI. В случае с Nginx это относительно просто:

```
server {
    listen 80;
    server_name www.myserver.com;
    location / {
        fastcgi_pass unix:/tmp/mysite.socket;
    }
}
```

Этот пример должен выглядеть знакомым, учитывая описанные ранее варианты. Последний трюк состоит в том, что с Nginx вам необходимо вручную задать все переменные FastCGI. Рекомендуется хранить эти переменные в отдельном файле (скажем, `fastcgi.conf`), и добавлять `include fastcgi.conf`; в конец блока `http`. Содержимое этого файла, настроенного для работы через WAI, должно быть следующим:

```
fastcgi_param QUERY_STRING      $query_string;
fastcgi_param REQUEST_METHOD    $request_method;
```

```

fastcgi_param  CONTENT_TYPE      $content_type;
fastcgi_param  CONTENT_LENGTH    $content_length;
fastcgi_param  PATH_INFO         $fastcgi_script_name;
fastcgi_param  SERVER_PROTOCOL   $server_protocol;
fastcgi_param  GATEWAY_INTERFACE CGI/1.1;
fastcgi_param  SERVER_SOFTWARE   nginx/$nginx_version;
fastcgi_param  REMOTE_ADDR       $remote_addr;
fastcgi_param  SERVER_ADDR       $server_addr;
fastcgi_param  SERVER_PORT       $server_port;
fastcgi_param  SERVER_NAME       $server_name;

```

11.6. Настольное приложение

Другой замечательный способ запускать ваше приложение — `wai-handler-webkit`¹. Он объединяет `Warp` и `QtWebkit` для создания исполняемого файла, который может быть запущен при помощи простого двойного щелчка мышью. Это может быть удобным способом сделать оффлайн версию вашего приложения.

Одно из удобств `Yesod` в данном случае состоит в том, что все ваши шаблоны страниц компилируются в исполняемый файл, избавляя вас от необходимости распространять их отдельно. В случае со статическими файлами это, однако, не так.

На самом деле, поддержка включения статических файлов прямо в приложение существует, обращайтесь к документации к `yesod-static`² за подробностями.

Аналогичный подход `wai-handler-launch`³, который не требует библиотеки `QtWebkit`, запускает `Warp` сервер и затем открывает пользовательский браузер по умолчанию. Следует отметить небольшой трюк: для того, чтобы понять, что пользователь всё ещё пользуется сайтом, `wai-handler-launch` вставляет небольшой отрывок `Javascript` сценария на каждую `HTML` страницу, которую раздаёт. Этот сценарий периодически посылает “пинг”(запрос, не несущий полезной нагрузки и требующий ответа) на сервер. Если `wai-handler-launch` не получает ответа на такой запрос в течение двух минут, то он останавливается.

11.7. CGI и Apache

`CGI` и `FastCGI` работают с `Apache` почти одинаково, так что подправить конфигурацию должно быть довольно просто. Вам всего лишь надо добиться выполнения двух пунктов:

1. Заставить веб-сервер раздавать ваши файлы как (Fast)CGI;
2. Сделать так, чтобы все запросы к вашему сайту шли через (Fast)CGI исполняемый файл.

¹<http://hackage.haskell.org/package/wai-handler-webkit>

²<http://hackage.haskell.org/package/yesod-static>

³<http://hackage.haskell.org/package/wai-handler-launch>

Вот пример файла конфигурации для приложения-блога, исполняемый файл которого называется “bloggy.cgi”, расположенного в подкаталоге “blog”. Этот пример бы взят из приложения, находящегося в каталоге /f5/snoyman/public/blog.

```
Options +ExecCGI
AddHandler cgi-script .cgi
Options +FollowSymlinks

RewriteEngine On
RewriteRule ^/f5/snoyman/public/blog$ /blog/ [R=301,S=1]
RewriteCond $1 !^bloggy.cgi
RewriteCond $1 !^static/
RewriteRule ^(.*) bloggy.cgi/$1 [L]
```

Первая запись RewriteRule отвечает за правильную обработку подкаталогов. В частности, она перенаправляет запросы к /blog на /blog/. Первая запись RewriteCond предотвращает запрос самого исполняемого файла, вторая — разрешает Apache раздавать статические файлы. Последняя строка, собственно, и осуществляет перенаправление запросов.

11.8. FastCGI и lighttpd

Для этого примера я опустил некоторые из основных настроек FastCGI, такие как mime-типы. В реальном приложении у меня используется более сложный файл, который добавляет префикс “www.” в начало URL, если он отсутствует, а также раздаёт статические файлы с отдельного домена. Однако следующего примера должно быть достаточно, чтобы понять основы.

Здесь “/home/michael/fastcgi” — это fastcgi приложение. Идея состоит в том, чтобы перенаправлять все запросы на “/app”, после чего раздавать всё, начинающееся с “/app” через исполняемый файл FastCGI.

```
server.port = 3000
server.document-root = "/home/michael"
server.modules = ("mod_fastcgi", "mod_rewrite")

url.rewrite-once = (
    "(.*)" => "/app/$1"
)

fastcgi.server = (
    "/app" => (
        "socket" => "/tmp/test.fastcgi.socket",
        "check-local" => "disable",
        "bin-path" => "/home/michael/fastcgi", # полный путь к
        исполняемому файлу
        "min-procs" => 1,
```

```
        "max-procs" => 30,  
        "idle-timeout" => 30  
    ))  
)
```

11.9. CGI и lighttpd

Ниже представлен конфигурационный файл, почти идентичный версии для FastCGI, однако он говорит lighttpd запускать файл, заканчивающийся на ".cgi" как CGI исполняемый файл. В данном случае этот файл располагается в "/home/michael/myapp.cgi".

```
server.port = 3000  
server.document-root = "/home/michael"  
server.modules = ("mod_cgi", "mod_rewrite")  
  
url.rewrite-once = (  
    "(.*)" => "/myapp.cgi/$1"  
)  
  
cgi.assign = (".cgi" => "")
```

Часть II

Продвинутые техники

Глава 12

REST-содержимое

Существуют истории о том, как на заре Интернета поисковые системы уничтожали целые веб-сайты. Когда динамическое содержимое было ещё новой концепцией, разработчики не принимали во внимание различий между GET и POST запросами. В результате, они создавали страницы с доступом через GET-метод, которые могли удалить содержимое. Когда поисковики сканировали такие сайты, они могли всё удалить.

Если бы эти разработчики точнее следовали спецификации HTTP, этого бы не случилось. GET-запрос подразумевает отсутствие побочных эффектов (вроде стирания сайта, к примеру). Не так давно, в веб-разработке произошло движение к правильному пониманию «передачи состояния представления» (REpresentational State Transfer), известного также как REST. Эта глава описывает особенности Yesod, связанные с поддержкой REST, и как они могут быть использованы для создания более надёжных веб-приложений.

12.1. Методы HTTP-запросов

Во многих веб-фреймворках вы пишете для ресурса одну функцию-обработчик. В Yesod по умолчанию предполагается отдельная функция-обработчик для каждого метода запроса. Два самых частых метода, с которыми вы будете иметь дело при создании веб-сайтов — это GET и POST. Эти методы лучше всего поддерживаются в формате HTML, поскольку только они используются в веб-формах. Однако при создании RESTful API, другие методы также очень полезны.

Технически, вы можете создать какие угодно методы, но настоятельно рекомендуется придерживаться описанных в спецификации HTTP. Наиболее часто встречаются методы:

- GET. Запрос на чтение. Предполагается, что при вызове GET не произойдёт никаких изменений на сервере. При многократном вызове GET-метода, сервер должен выдавать тот же ответ, если исключить такие вещи как «текущее время» или результаты, получаемые случайно.
- POST. Обычный запрос на изменение. Он никогда не должен посылаться поль-

зователем дважды. Обычно, в качестве примера POST-запроса приводят перевод денежных средств с одного банковского счёта на другой.

- PUT. Создание нового ресурса на сервере или изменение существующего. Этот метод может быть выполнен многократно с одинаковым эффектом.
- DELETE. В точности соответствует названию. Удаляет ресурс с сервера. Повторные вызовы не должны приводить к проблемам.

В определённой степени, это соответствует философии Haskell: GET-запрос подобен чистой функции, без побочных эффектов. На практике ваши GET-функции вероятно будут производить IO-операции — чтение информации из базы данных, логирование пользовательских действий и так далее.

В главе Маршрутизация URL и обработчики можно найти более полную информацию по синтаксису определения функций-обработчиков для каждого метода HTTP-запроса.

12.2. Представления

Пусть мы определили тип данных и значение:

```
data Person = Person { name :: String, age :: Int }
michael = Person "Michael" 25
```

Мы можем представить эти данные в HTML:

```
<table>
  <tr>
    <th>Name</th>
    <td>Michael</td>
  </tr>
  <tr>
    <th>Age</th>
    <td>25</td>
  </tr>
</table>
```

или в JSON:

```
{"name": "Michael", "age": 25}
```

или в XML:

```
<person>
  <name>Michael</name>
  <age>25</age>
</person>
```

Зачастую веб-приложения используют различные URL для получения этих представлений: возможно `/person/michael.html`, `/person/michael.json`, и т.п. Yesod, следуя принципам RESTful, использует один URL для каждого ресурса. Поэтому в Yesod все представления будут доступны через `/person/michael`.

Возникает вопрос, как мы определим, какое представление требуется. Ответ можно найти в Ассерт-заголовке запроса: он получает приоритезированный список типов содержимого, ожидаемого клиентом. Основываясь на этом заголовке, Yesod будет автоматически определять, какое представление требуется.

Немного конкретизируем последнее предложение следующим кодом:

```
type ChooseRep = [ContentType] -> IO (ContentType, Content)
class HasReps a where
    chooseRep :: a -> ChooseRep
```

Функция `chooseRep` получает два аргумента: значение, для которого мы получаем представление, и список типов содержимого, которые клиент будет получать. Список определяется чтением из заголовка Ассерт-запроса. `chooseRep` возвращает пару, включающую тип содержимого ответа и непосредственно содержимое.

Этот класс типов — основа RESTful-подхода, принятого в Yesod для представлений. Каждая функция-обработчик должна возвращать экземпляр `HasReps`. Когда Yesod генерирует функцию-диспетчер, он автоматически применяет `chooseRep` к каждому обработчику, по существу давая всем функциям тип `Handler ChooseRep`. После запуска `Handler` и получения результата `ChooseRep`, он применяет список типов содержимого, полученный из Ассерт-заголовка.

В Yesod определены многочисленные экземпляры класса `HasReps`. Например, когда мы используем `defaultLayout`, возвращается тип `RepHtml`, который выглядит так:

```
newtype RepHtml = RepHtml Content
instance HasReps RepHtml where
    chooseRep (RepHtml content) _ = return ("text/html", content)
```

Заметьте, что мы полностью игнорируем список ожидаемых типов содержимого. Многочисленные встроенные представления (`RepHtml`, `RepPlain`, `RepJson`, `RepXml`), в действительности, поддерживают только единственное представление и, следовательно, то, что содержится в Ассерт-заголовке клиентского запроса, является несущественным.

12.3. RepHtmlJson

Противоположный пример — `RepHtmlJson`, который создаёт либо HTML, либо JSON представление. Этот экземпляр класса особенно полезен в программировании AJAX приложений. Вот пример, возвращающий либо HTML, либо JSON данные, в зависимости от того, что хочет клиент.

```
{-# LANGUAGE QuasiQuotes, TypeFamilies, OverloadedStrings #-}
{-# LANGUAGE MultiParamTypeClasses, TemplateHaskell #-}
```

```

import Yesod
data R = R
mkYesod "R" [parseRoutes|
/ RootR GET
/#String NameR GET
|]
instance Yesod R

getRootR = defaultLayout $ do
  setTitle "Homepage"
  addScriptRemote []
  "http://ajax.googleapis.com/ajax/libs/jquery/1.4/jquery.min.js"
  toWidget [julius|
$(function(){
  $("#ajax_a").click(function(){
    jQuery.getJSON($(this).attr("href"), function(o){
      $("#div").text(o.name);
    });
    return false;
  });
});
|]
  let names = words "Larry_Moe_Curly"
  [whamlet|
<h2>AJAX Version
<div #results>
  AJAX results will be placed here when you click #
  the names below.
<ul #ajax>
  $forall name <- names
    <li>
      <a href=@{NameR name}>#{name}

<h2>HTML Version
<p>
  Clicking the names below will redirect the page #
  to an HTML version.
<ul #html>
  $forall name <- names
    <li>
      <a href=@{NameR name}>#{name}

|]

getNameR name = do
  let widget = do

```

```

        setTitle $ toHtml name
        [whamlet|Looks like you have Javascript off. Name: |
        #{name}||]
        let json = object ["name" .= name]
        defaultLayoutJson widget json

main = warpDebug 4000 R

```

12/RepHtmlJson.hs

Наш обработчик `getRootR` создаёт страницу с тремя ссылками и Javascript-кодом, который перехватывает нажатия на ссылки и производит асинхронные запросы. Если у пользователя разрешён Javascript, то нажатие на ссылку приводит к запросу, отсылаемому с Асепт-заголовком `application/json`. В этом случае, `getNameR` будет возвращать JSON-представление.

Если пользователь запрещает Javascript, то нажатие на ссылку будет отсылать пользователя к соответствующему URL. Браузер укажет приоритет HTML-представления данных и, следовательно, будет возвращена страница, определённая виджетом.

Мы, конечно, можем расширить это поведение для работы с XML, Atom-лентами, или даже двоичным представлением данных. Для развлечения можно было бы написать веб-приложение, которое готовит данные, просто используя для типов данных стандартные экземпляры класса **Show**. И затем написать веб-клиента, который разбирает результаты, используя стандартные экземпляры класса **Read**.

Возможно, вас беспокоит эффективность этого подхода. Разве это не означает, что мы должны генерировать и HTML и JSON ответы для каждого запроса? Благодаря ленивости этого не происходит. В функции `getNameR` ни `widget`, ни `json` не будет вычислен, пока не будет выбран соответствующий тип ответа, и, следовательно, всегда будет выполняться только один из них.

12.4. Новостные ленты

Отличный практический пример множественных представлений даёт пакет `yesod-newsfeed`. В вебе существуют два главных формата новостных лент: RSS и Atom. Они содержат почти в точности схожую информацию, но по-разному упакованную.

В пакете `yesod-newsfeed` определён тип данных `Feed`, который содержит такую информацию, как название, описание и время последнего изменения. Далее определено два набора функций для отображения данных: одна для RSS, другая для Atom. Для каждого набора определён собственный тип данных представления:

```

newtype RepAtom = RepAtom Content
instance HasReps RepAtom where
    chooseRep (RepAtom c) _ = return (typeAtom, c)
newtype RepRss = RepRss Content
instance HasReps RepRss where
    chooseRep (RepRss c) _ = return (typeRss, c)

```

В третьем модуле определён другой тип данных:

```
data RepAtomRss = RepAtomRss RepAtom RepRss
instance HasReps RepAtomRss where
  chooseRep (RepAtomRss (RepAtom a) (RepRss r)) = chooseRep
    [ (typeAtom, a)
      , (typeRss, r)
    ]
```

Этот тип данных автоматически определяет представление, предпочтительное для клиента (по умолчанию выбирается Atom). Если клиентские соединения понимают только RSS, полагая, что они обеспечивают правильные HTTP-заголовки, Yesod выдаст RSS представление.

12.5. Другие заголовки запроса

Существует большое количество других заголовков запроса. Некоторые из них влияют только на передачу данных между сервером и клиентом, и не должны влиять на приложение в целом. Например, Accept-Encoding сообщает серверу о том, какие схемы сжатия понимает клиент, а Host информирует сервер, какие виртуальные хосты обслуживаются.

Другие заголовки влияют на приложение и автоматически считываются в Yesod. Например, заголовок Accept-Language определяет, какой язык предпочитает клиент (Английский, Испанский, Немецкий и т.п.). В части про i18n детально описывается, как используется этот заголовок.

12.6. Отсутствие состояния

Я оставил этот раздел напоследок, не от того, что он менее важен, а скорее потому что здесь нет особенностей Yesod, обеспечивающих соответствующую поддержку.

Протокол HTTP не поддерживает состояний (т.е. является stateless-протоколом): каждый запрос рассматривается как начало общения. Это означает, к примеру, что серверу не важно то, что вы уже запрашивали пять страниц — он будет обрабатывать ваш шестой запрос так, как если бы он был самым первым.

С другой стороны, некоторая функциональность сайта не будет работать без хранения состояния. Например, как реализовать корзину покупок, если не хранить информацию о покупках между запросами?

Решением являются куки-файлы (cookies) и построенные на их основе сессии. У нас есть целый раздел, посвящённый сессиям в Yesod. Однако я не могу не подчеркнуть, что их следует использовать с осторожностью.

Позвольте привести пример. Существует популярная система отслеживания ошибок, с которой я работал ежедневно и которая слишком много использовала сессии. Там есть маленький выпадающий список на каждой странице для выбора текущего проекта. Ничто не предвещает проблем, да? Всё, что этот список делает — устанавливает текущий проект для вашей сессии.

В результате, щелчок на ссылке «просмотреть ошибки» внутренне зависит от того, какой проект вы выбирали последним. Из-за этого нет способа сделать закладку на ваши открытые ошибки для Yesod и отдельную ссылку на ваши ошибки для Hamlet.

Правильный RESTful подход состоит в том, чтобы иметь один ресурс для всех Yesod-ошибок, а другой для всех Hamlet. В Yesod это легко делается примерно таким определением маршрута:

```
/ ProjectsR GET
/projects/#ProjectID ProjectIssuesR GET
/issues/#IssueID IssueR GET
```

Будьте внимательны к своим пользователям: правильная архитектура без использования состояний означает, что такие базовые вещи, как закладки, хранимые ссылки и кнопки «Вперёд», «Назад» будут всегда работать.

12.7. Выводы

Yesod придерживается следующих принципов REST:

- использовать правильные методы запроса;
- каждый ресурс должен иметь в точности один URL;
- разрешать множественные представления данных для одного URL;
- проверять заголовки запросов для определения дополнительной информации о том, что ждёт клиент.

Это упрощает использование Yesod не только для построения веб-сайтов, но также и для построения программных интерфейсов (API). В действительности, используя подобную `RepHtmlJson` технику, с одного URL вы можете обслужить одновременно «дружелюбную к пользователю» HTML-страницу и «дружелюбную к машине» JSON-страницу.

Глава 13

Монады в Yesod

Как вы уже увидели, в этой книге появлялось несколько монад: `Handler`, `Widget` и `YesodDB` (для `Persistent`). Как и всякая монада, каждая из них предоставляет некоторую специфическую функциональность: `Handler` предоставляет доступ к запросу и позволяет отправлять ответы, `Widget` содержит HTML, CSS и Javascript, а `YesodDB` позволяет делать запросы к базе данных. В терминах Model-View-Controller (MVC), мы могли бы рассматривать `YesodDB` как модель, `Widget` — как представление, а `Handler` — как контроллер.

До сих пор у нас были представлены очень простые способы использования этих монад: основной обработчик работает в монаде `Handler`, используя `runDB` для выполнения запроса `YesodDB` и `defaultLayout` для возврата `Widget`, который, в свою очередь, был создан вызовом `toWidget`.

Тем не менее, если у нас будет глубокое понимание этих типов, мы сможем достичь более интересных результатов.

13.1. Трансформаторы монад

Монады, они как луковицы. Монады *не* как пирожные.

Вроде бы Шрек

Прежде чем мы углубимся в монады `Yesod`, мы должны немного понимать трансформаторы монад. (Если вы уже знаете о трансформаторах монад всё, вы, скорее всего, можете пропустить этот раздел.) Различные монады предоставляют различную функциональность: `Reader` предоставляет доступ только для чтения к некоторым данным по всему вычислению, `Error` позволяет обрывать вычисления, и так далее.

Часто, однако, хотелось бы иметь возможность комбинировать функциональность некоторых из этих монад. В конце концов, почему бы не иметь вычисление с доступом только для чтения к некоторым параметрам настройки, которое в любой момент могло бы прерваться с ошибкой? Одним из подходов к решению могло бы стать написание новой монады, например `ReaderError`, но это имеет очевидный недостаток

экспоненциальной сложности: потребуется писать новую монаду для каждой возможной комбинации.

Вместо этого мы используем трансформаторы монад. Вместе с `Reader`, у нас есть `ReaderT`, который добавляет функциональность `Reader` к любой другой монаде. Таким образом, мы могли бы представить `ReaderError` так (концептуально):

```
type ReaderError = ReaderT Error
```

Чтобы получить доступ к параметру настройки, мы можем использовать функцию `ask`. А что насчёт обрывания вычисления? Мы бы хотели вызвать `throwError`, но это не вполне будет работать. Вместо этого мы должны использовать `lift`, чтобы втянуть наш вызов в монаду на уровень выше. Другими словами:

```
throwError :: errValue -> Error
lift . throwError :: errValue -> ReaderT Error
```

Сейчас вы должны уловить несколько идей:

- Трансформатор может быть использован для добавления функциональных возможностей к существующим монадам.
- Трансформатор должен всегда обёртываться вокруг существующей монады.
- Функциональные возможности получившейся монады будут зависеть не только от трансформатора монады, но и от монады, обёрнутой внутри.

Отличный пример последнего утверждения — это монада `IO`. Независимо от того, сколько слоёв трансформаторов у вас есть вокруг `IO`, там всё ещё есть ядро `IO`, то есть вы можете выполнять ввод/вывод в любом из этих стеков трансформаторов монад. Вы будете часто видеть код, который выглядит как `liftIO $ putStrLn "Привет_вам!"`.

13.2. Три трансформатора

Мы уже обсуждали два из наших трансформаторов ранее: `Handler` и `Widget`. Напомним два особых факта про них:

- Для упрощения сообщений об ошибках они не являются фактическими трансформаторами. Вместо этого, они объявлены как `newtype` с жёстко заданной внутренней монадой. Именно поэтому `Yesod` предоставляет специальную функцию `lift`, которая работает для `Handler` и `Widget`.
- На самом деле у них есть дополнительные параметры для типов подсайта и основного сайта. В результате, библиотеки `Yesod` предоставляют `GHandler sub master a` и `GWidget sub master a`, а каждый сайт получает пару синонимов типов `type Handler = GHandler MyApp MyApp` и `type Widget = GWidget MyApp MyApp` (`.`).

В пакете `persistent`¹, есть класс типов `PersistStore`. Этот класс типов определяет все примитивные операции, которые можно выполнять с базой данных, например, `get`. Этот класс типов по существу выглядит как `class (Monad b m) => PersistStore b m`. Где `b` определяет сервер базы данных, а на самом деле является трансформатором монады, а `m` является внутренней монадой, которую `b` оборачивает. И `SQL`, и `MongoDB` имеют свои собственные экземпляры; в случае `SQL`, он выглядит так:

```
instance MonadBaseControl IO m => PersistBackend SqlPersist m
```

Это означает, что вы можете работать с базой данных `SQL` при любой базовой монаде, до тех пор, пока эта монада поддерживает `MonadBaseControl IO`, позволяющую правильно обрабатывать исключения на стеке монад. В общем, это означает любой стек трансформаторов, построенный вокруг `IO` (кроме исключительных случаев, как `ContT`). К счастью для нас, это включает в себя как `Handler` так и `Widget`. Следовательно, мы можем наслаивать наш `Persistent` трансформатор на `Handler` или `Widget`.

Это не всегда было так. Перед `Yesod 0.10`, `Yesod` был построен на базе пакета `enumerators`, который не поддерживает `MonadBaseControl`. В `Yesod 0.10`, мы перешли к пакету `conduit`², который значительно упрощает всё, что мы здесь обсуждаем.

Для того, чтобы упростить обращение к соответствующим `Persistent` трансформаторам, пакет `yesod-persistent`³ определяет ассоциированный тип `YesodPersistBackend`. Например, если у нас есть сайт, который называется `MyApp` и использует `SQL`, мы можем определить что-то вроде `type instance YesodPersistBackend MyApp = SqlPersist`.

Когда мы захотим выполнить наши действия с базой данных, у нас будет `SqlPersist`, обернутый вокруг `Handler` или `Widget`. Так мы сможем использовать стандартные функции развёртки `Persistent` (например, `runSqlPool`) для исполнения действия и возврата в нормальный `Handler/Widget`. Для автоматизации этого, предоставляется функция `runDB`. Суммируя всё это, мы теперь можем исполнять действия с базой данных внутри наших обработчиков и виджетов.

Большую часть времени в коде `Yesod`, и особенно на протяжении предыдущих глав, виджеты рассматривались как контейнеры, которые просто объединяют вместе `HTML`, `CSS` и `Javascript` и не содержат действий. Но если вы посмотрите на последний абзац ещё раз, вы поймёте, что это не обязательное требование. Поскольку виджет — это трансформатор над обработчиком, всё, что вы делаете в обработчике, может быть сделано в виджете, в том числе и операции с базой данных. Всё, что вам нужно сделать, — это протянуть вызов, используя `lift`.

¹<http://hackage.haskell.org/package/persistent>

²<http://hackage.haskell.org/package/conduit>

³<http://hackage.haskell.org/package/yesod-persistent>

13.3. Пример: навигационная панель на основе базы данных

Давайте применим некоторые из новых знаний на практике. Мы хотим сделать виджет, который генерирует свой вывод на основе содержимого базы данных. Ранее мы бы загрузили данные в обработчике, а затем передали эти данные в виджет. Теперь же загрузим данные в самом виджете. Это хорошо для модульности, так как этот виджет может использоваться в любом обработчике по нашему желанию, без необходимости передавать содержимое базы данных.

```
{-# LANGUAGE OverloadedStrings, TypeFamilies, TemplateHaskell, []
      FlexibleContexts,
      QuasiQuotes, MultiParamTypeClasses, GADTs #-}
import Yesod
import Database.Persist.Sqlite
import Data.Text (Text)
import Data.Time
import Control.Monad.Logger (runStderrLoggingT)

share [mkPersist sqlSettings, mkMigrate "migrateAll"] []
      [persistUpperCase]
Link
  title Text
  url Text
  added UTCTime
[]

data LinksExample = LinksExample ConnectionPool

mkYesod "LinksExample" [parseRoutes]
/ RootR GET
/add-link AddLinkR POST
[]

instance Yesod LinksExample

instance RenderMessage LinksExample FormMessage where
  renderMessage _ _ = defaultFormMessage

instance YesodPersist LinksExample where
  type YesodPersistBackend LinksExample = SqlPersist
  runDB db = do
    LinksExample pool <- getYesod
    runSqlPool db pool

getRootR :: Handler RepHtml
```

```

getRootR = defaultLayout [whamlet|
<form method=post action=@{AddLinkR}>
  <p>
    Добавить ссылку на #
    <input type=url name=url value=http://>
    \ с названием #
    <input type=text name=title>
    \ #
    <input type=submit value="Добавить">
  <h2>Добавленные ссылки:
  ^{existingLinks}
  |]

existingLinks :: Widget
existingLinks = do
  links <- lift $ runDB $ selectList [] [LimitTo 5, Desc LinkAdded]
  [whamlet|
<ul>
  $forall Entity _ link <- links
    <li>
      <a href=#{linkUrl link}>#{linkTitle link}
  |]

postAddLinkR :: Handler ()
postAddLinkR = do
  url <- runInputPost $ ireq urlField "url"
  title <- runInputPost $ ireq textField "title"
  now <- liftIO getCurrentTime
  runDB $ insert $ Link title url now
  setMessage "Ссылка добавлена"
  redirect RootR

main :: IO ()
main = withSqlitePool "links.db3" 10 $ \pool -> do
  runStderrLoggingT $ runSqlPool (runMigration migrateAll) pool
  warpDebug 3000 $ LinksExample pool

```

13/navbar.hs

В частности, обратите внимание на функцию `existingLinks`. Заметьте, всё, что нужно сделать, — это применить `lift` к обычному действию с базой данных. А в `getRootR`, мы трактуем `existingLinks` как обычный виджет без каких-либо специальных параметров вообще. На рисунке представлен вывод этого приложения.

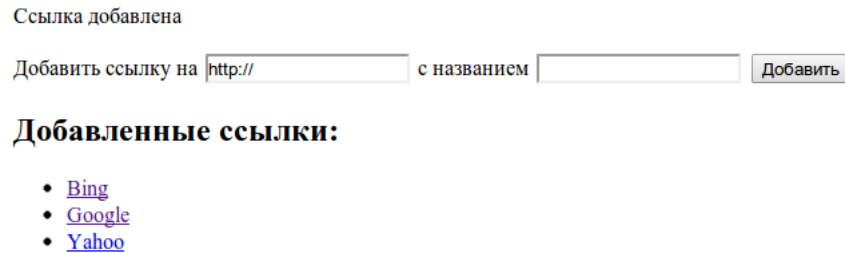


Рис. 13.1: Скриншот панели навигации

13.4. Пример: информация из запроса

Таким же образом вы можете получить информацию из запроса внутри виджета. В примере ниже мы определяем порядок сортировки списка на основе значения параметра GET-запроса.

```
{-# LANGUAGE OverloadedStrings, TypeFamilies, TemplateHaskell,
      QuasiQuotes, MultiParamTypeClasses, GADTs #-}
import Yesod
import Data.Text (Text)
import Data.List (sortBy)
import Data.Ord (comparing)

data Person = Person
  { personName :: Text
  , personAge  :: Int
  }

people :: [Person]
people =
  [ Person "Miriam" 25
  , Person "Eliezer" 3
  , Person "Michael" 26
  , Person "Gavriella" 1
  ]

data People = People

mkYesod "People" [parseRoutes|
/ RootR GET
|]

instance Yesod People
```

```

instance RenderMessage People FormMessage where
  renderMessage _ _ = defaultMessage

getRootR :: Handler RepHtml
getRootR = defaultLayout [whamlet|
<p>
  <a href="?sort=name">Сортировать по имени
  \ | #
  <a href="?sort=age">Сортировать по возрасту
  \ | #
  <a href="">Без сортировки
  ^{showPeople}
  |]

showPeople :: Widget
showPeople = do
  msort <- lift $ runInputGet $ iopt textField "sort"
  let people' =
      case msort of
        Just "name" -> sortBy (comparing personName) people
        Just "age"  -> sortBy (comparing personAge)  people
        _           -> people

  [whamlet|
<dl>
  $forall person <- people'
    <dt>#{personName person}
    <dd>#{show $ personAge person}
  |]

main :: IO ()
main = warpDebug 3000 People

```

13/request-information.hs

Повторим, всё, что нужно сделать, — это протянуть функцию стандартную для обработчика (в данном случае, `runInputGet`), чтобы выполнить её в нашем виджете.

13.5. Выводы

Если вы полностью пропустили эту главу, вы всё равно сможете с большой пользой использовать `Yesod`. Преимущество понимания как взаимодействуют монады `Yesod` — возможность производить более чистый и более модульный код. Способность выполнять произвольные действия в `Widget` может быть мощным инструментом, и пони-

вание того, как взаимодействуют Persistent и ваш Handler код, может помочь вам сделать более обоснованные проектные решения в вашем приложении.

Глава 14

Аутентификация и Авторизация

Аутентификация и авторизация — две тесно связанные, но в то же время различные концепции. Тогда как первая имеет дело с идентификацией пользователя, вторая определяет, что пользователю позволено делать. К сожалению, поскольку для обоих терминов часто используется сокращение «auth», эти концепции часто объединяются.

Yesod имеет встроенную поддержку для нескольких сторонних систем аутентификации, таких как OpenID, BrowserID и OAuth. Это внешние системы, которым ваше приложение доверяет удостоверение личности пользователя. Также есть поддержка более привычных способов аутентификации, таких как имя пользователя и пароль или адрес почты и пароль. Первый способ гарантирует простоту как для пользователей (нет необходимости запоминать новые пароли), так и для разработчиков (нет нужды иметь дело со всей архитектурой безопасности), в то время как второй даёт разработчику больший контроль.

Для авторизации мы можем воспользоваться преимуществами REST и типобезопасными URL, чтобы создать простую и декларативную систему. К тому же, поскольку весь код авторизации написан на Haskell, в вашем распоряжении будет вся гибкость языка.

В этой главе мы рассмотрим, как настроить «auth» в Yesod, и обсудим некоторые плюсы и минусы различных вариантов аутентификации.

14.1. Обзор

Пакет `yesod-auth`¹ предоставляет унифицированный интерфейс для различных плагинов аутентификации. Единственное, что требуется от этих плагинов, так это чтобы они идентифицировали пользователя по какой-нибудь уникальной строке. В OpenID, к примеру, это может быть фактическое значение OpenID. В BrowserID это адрес электронной почты. Для HashDB (которая использует базу данных хэшированных паролей) это имя пользователя.

Каждый плагин аутентификации предоставляет свою собственную систему для входа на сайт, будь то передача токенов с внешнего сайта или же форма входа с вво-

¹<http://hackage.haskell.org/package/yesod-auth>

дом адреса почты и пароля. После успешного входа плагин устанавливает значение в пользовательской сессии, определяющее его `AuthId`. Значением этого `AuthId` обычно является `Persistent ID` из таблицы, используемой для отслеживания пользователей.

Есть несколько функций, позволяющих получить пользовательский `AuthId`, наиболее используемыми из которых являются `maybeAuthId`, `requireAuthId`, `maybeAuth` и `requireAuth`. Версии с `require` перенаправляют на страницу входа, если пользователь ещё не вошёл, а те функции, что не оканчиваются на `Id`, возвращают два значения: `ID` в таблице и значение сущности.

Поскольку хранение `AuthId` построено на сессиях, то применимы все их правила. В частности, данные сохраняются в зашифрованом, подписаном `HMAC` куки, которое автоматически устаревает после определённого, задаваемого в конфигурации, периода неактивности. Кроме того, поскольку не существует серверной составляющей сессий, то при выходе из системы просто удаляются данные из куки этой сессии, и если пользователь повторно использует старое значение куки, то сессия будет всё ещё считаться действительной.

С другой стороны, авторизация контролируется несколькими методами внутри класса типов `Yesod`. Эти методы запускаются для каждого запроса, чтобы определить — разрешить пользователю доступ, запретить его, или же потребовать от пользователя войти на сайт. По умолчанию эти методы разрешают доступ для каждого запроса. В качестве альтернативы вы можете реализовать авторизацию способом, подходящим для конкретного случая, вызывая `requireAuth` и подобные ей функции в индивидуальных функциях-обработчиках, хотя это и отрицательно отразится на преимуществах декларативной системы авторизации.

14.2. Аутентифицируй меня

Давайте сразу рассмотрим пример с аутентификацией.

```
{-# LANGUAGE OverloadedStrings, TemplateHaskell, TypeFamilies,
      MultiParamTypeClasses, QuasiQuotes #-}
import Yesod
import Yesod.Auth
import Yesod.Auth.BrowserId
import Yesod.Auth.GoogleEmail
import Data.Text (Text)
import Network.HTTP.Conduit (Manager, newManager, def)

data MyAuthSite = MyAuthSite
  { httpManager :: Manager
  }

mkYesod "MyAuthSite" [parseRoutes]
/ RootR GET
/auth AuthR Auth getAuth
|]
```



```

instance Yesod MyAuthSite where
  -- Внимание! Что бы работал вход с BrowserID, вы должны здесь
  -- корректно установить адрес вашего хоста.
  approot = ApprootStatic "http://localhost:3000"

instance YesodAuth MyAuthSite where
  type AuthId MyAuthSite = Text
  getAuthId = return . Just . credsIdent

  loginDest _ = RootR
  logoutDest _ = RootR

  authPlugins _ =
    [ authBrowserId
      , authGoogleEmail
    ]

  authHttpManager = httpManager

instance RenderMessage MyAuthSite FormMessage where
  renderMessage _ _ = defaultFormMessage

getRootR :: Handler RepHtml
getRootR = do
  maid <- maybeAuthId
  defaultLayout [whamlet|
<p>Your current auth ID: #{show maid}
$maybe _ <- maid
  <p>
    <a href=@{AuthR LogoutR}>Logout
$nothing
  <p>
    <a href=@{AuthR LoginR}>Go to the login page
  |]

main :: IO ()
main = do
  man <- newManager def
  warpDebug 3000 $ MyAuthSite man

```

14/authentication.hs

Начнём с объявлений маршрутов. Сперва объявляем наш стандартный маршрут RootR, а затем определяем подсайт для аутентификации. Помните, что ему нужны четыре параметра: путь к подсайту, имя маршрута, имя подсайта и функция для получе-

ния значения подсайта. Другими словами, судя по строке

```
/auth AuthR Auth getAuth
```

нам нужно иметь функцию `getAuth :: MyAuthSite -> Auth`. Пока мы не напишем эту функцию сами, `yesod-auth`² предоставляет её автоматически. Для других подсайтов (к примеру, со статическими файлами) мы указываем настройки конфигурации в значении подсайта, и поэтому нам нужно определять функцию `get`. Для `auth`-подсайта мы указываем эти настройки в отдельном классе типов `YesodAuth`.

Почему бы не использовать значение подсайта? Существует достаточно много настроек, которые мы бы хотели передать `auth`-подсайту, и делать это из типа записи (`record type`) было бы неудобно. Также, поскольку мы хотим иметь ассоциируемый тип `AuthId`, использование класса типов будет более естественным.

С другой стороны, почему бы не использовать класс типов для всех подсайтов? У такого подхода есть минус: вы сможете иметь только один экземпляр класса на сайт, не позволяя раздавать различные группы статичных файлов с различных маршрутов. Также, если мы хотим загружать данные во время инициализации приложения, значение подсайта подойдёт лучше.

Так что же именно входит в экземпляр класса `YesodAuth`? Шесть объявлений являются обязательными:

- Ассоциированный тип `AuthId`. Это то значение, которое `yesod-auth` будет возвращать вам, когда вы будете запрашивать, вошёл ли пользователь (через `maybeAuthId` или `requireAuthId`). В нашем случае мы просто используем `Text` для сохранения необработанного идентификатора. Как вы скоро увидите, в нашем случае это адрес электронной почты.
- `getAuthId` получает текущий `AuthId` из типа данных `Creds` (credentials — учётные данные). Этот тип содержит три фрагмента информации: используемый способ аутентификации (в нашем случае `browserid` или `googleemail`), текущий идентификатор и ассоциированный список с различной дополнительной информацией.
- `loginDest` возвращает маршрут, на который перенаправляется пользователь после успешного входа.
- Аналогичным образом `logoutDest` возвращает маршрут, на который происходит перенаправление после выхода.
- `authPlugins` — это список используемых способов аутентификации. В нашем примере мы используем `BrowserID`, который входит через систему `Mozilla BrowserID`, и `Google Email`, который аутентифицирует почтовый адрес пользователя, используя его учётную запись `Google`. Что хорошо в этих двух способах, так это то, что:

²<http://hackage.haskell.org/package/yesod-auth>

- Они не требуют установки, в отличие от Facebook или OAuth, которые требуют задания учётных данных.
 - Они используют в качестве идентификатора привычный людям адрес электронной почты, в отличие от OpenID, который использует URL.
- `authHttpRequestManager` получает менеджер HTTP-соединений из основного типа. Это позволяет системам аутентификации, которые используют HTTP-соединения (то есть почти всем сторонним системам входа), использовать эти соединения повторно, избегая затрат на установку нового TCP-соединения для каждого запроса.

Наш обработчик `RootR` содержит ссылки на страницы входа и выхода, которые отображаются в зависимости от того, вошёл пользователь на сайт или нет. Обратите внимание, как мы конструируем эти ссылки на подсайт: первым идёт имя маршрута подсайта (`AuthR`), следом маршрут в подсайте (`LoginR` и `LogoutR`).

Иллюстрации ниже показывают процесс входа со стороны пользователя.

Your current auth ID: Nothing

[Go to the login page](#)

Рис. 14.1: Начальная загрузка страницы

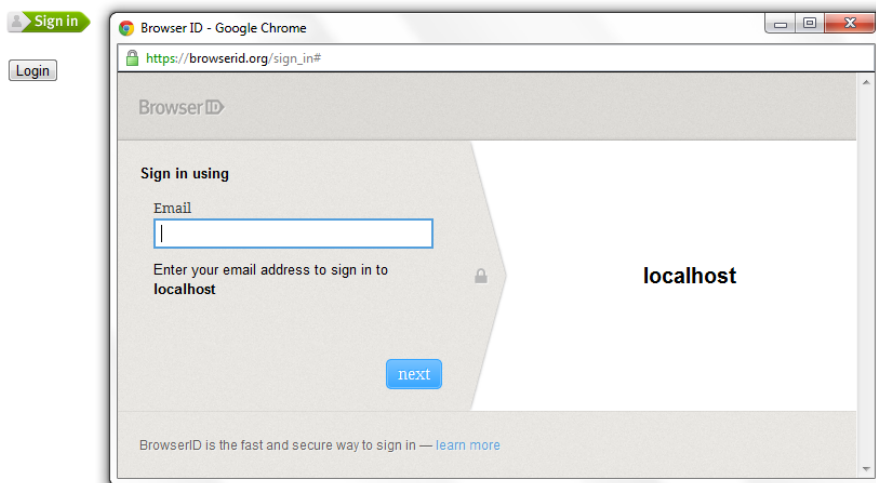


Рис. 14.2: Экран входа с помощью BrowserID

You are now logged in

Your current auth ID: Just "michael@snoyman.com"

[Logout](#)

Рис. 14.3: Домашняя страница после входа

14.3. Электронная почта

Для большинства случаев будет достаточно аутентификации по адресу электронной почты, осуществляемой сторонним сервисом. Но иногда вам может понадобиться, чтобы пользователи использовали для вашего сайта пароль. Сгенерированный сайт не включает эту функциональность, так как:

- Чтобы безопасно использовать пароли, надо использовать SSL. Многие пользователи не предоставляют доступ к сайту по SSL.
- Несмотря на то, что система аутентификации по адресу электронной почты должным образом хранит пароли (с использованием хэширования и “соли”), скомпрометированная база данных всё равно может быть проблемой. Помните: мы не делаем никаких предположений о том, что пользователи Yesod используют безопасные методы развёртывания.
- Вам нужна работающая система для отправки электронной почты. Многие веб-серверы в наши дни не имеют достаточных средств для защиты от спама по сравнению с теми, что используют почтовые сервера.

Пример ниже использует системную программу для отправки писем — `sendmail`. Если вы хотите избежать хлопот, работая с серверами электронной почты самостоятельно, вы можете использовать Amazon SES. Есть пакет `mime-mail-ses`^a, который предоставляет альтернативу использованию `sendmail`. Этот подход мы используем на сайте `Haskellers.com`.

^a<http://hackage.haskell.org/package/mime-mail-ses>

Но если предположить, что вы можете удовлетворить эти требования и хотите иметь вход по паролю именно для вашего сайта, то Yesod может предложить встроенную систему аутентификации. Для её использования от вас потребуется немного больше кода, ведь будет необходимо безопасно сохранять пароли в базе данных и отправлять пользователю почтовые сообщения (верификация учётной записи, восстановление пароля, и т.д.).

Давайте посмотрим на сайт, предоставляющий аутентификацию по адресу электронной почты и паролю и хранящий пароли в Persistent базе данных SQLite.

```
{-# LANGUAGE OverloadedStrings, TypeFamilies, QuasiQuotes, GADTs,
      TemplateHaskell, MultiParamTypeClasses,
      FlexibleContexts #-}
```

```

import Yesod
import Yesod.Auth
import Yesod.Auth.Email
import Database.Persist.Sqlite
import Database.Persist.TH
import Data.Text (Text)
import Network.Mail.Mime
import qualified Data.Text.Lazy.Encoding
import Text.Shakespeare.Text (stext)
import Text.Blaze.Html.Renderer.Utf8 (renderHtml)
import Text.Hamlet (shamlet)
import Data.Maybe (isJust)
import Control.Monad (join)
import Control.Monad.Logger (runStderrLoggingT)

share [mkPersist sqlSettings, mkMigrate "migrateAll"] []
[persistUpperCase]
User
  email Text
  password Text Maybe -- Пароль может быть ещё не задан
  verkey Text Maybe -- Используется для сброса пароля
  verified Bool
  UniqueUser email
[]

data MyEmailApp = MyEmailApp Connection

mkYesod "MyEmailApp" [parseRoutes]
/ RootR GET
/auth AuthR Auth getAuth
[]

instance Yesod MyEmailApp where
  -- Электронные письма будут содержать ссылки, так что убедитесь, []
  -- что включили approot,
  -- чтобы ссылки были правильными!
  approot = ApprootStatic "http://localhost:3000"

instance RenderMessage MyEmailApp FormMessage where
  renderMessage _ _ = defaultFormMessage

-- Установка Persistent
instance YesodPersist MyEmailApp where
  type YesodPersistBackend MyEmailApp = SqlPersist
  runDB f = do
    MyEmailApp conn <- getYesod

```

```

        runSqlConn f conn

instance YesodAuth MyEmailApp where
    type AuthId MyEmailApp = UserId

    loginDest _ = RootR
    logoutDest _ = RootR
    authPlugins _ = [authEmail]

    -- Необходимо найти UserId по заданному адресу электронной почты.
    getAuthId creds = runDB $ do
        x <- insertBy $ User (credsIdent creds) Nothing Nothing False
        return $ Just $
            case x of
                Left (Entity userid _) -> userid -- свежедобавленный
                пользователь
                Right userid -> userid -- существующий пользователь

    authHttpManager = error "Email doesn't need an HTTP manager"

    -- Здесь весь код работающий с электронной почтой
instance YesodAuthEmail MyEmailApp where
    type AuthEmailId MyEmailApp = UserId

    addUnverified email verkey =
        runDB $ insert $ User email Nothing (Just verkey) False

    sendVerifyEmail email _ verurl =
        liftIO $ renderSendMail (emptyMail $ Address Nothing
            "noreply")
            { mailTo = [Address Nothing email]
            , mailHeaders =
                [ ("Subject", "Verify your email address")
                ]
            , mailParts = [[textPart, htmlPart]]
            }
        where
            textPart = Part
                { partType = "text/plain; charset=utf-8"
                , partEncoding = None
                , partFilename = Nothing
                , partContent = Data.Text.Lazy.Encoding.encodeUtf8 [stext|
Please confirm your email address by clicking on the link below.

\#{verurl}|

```

```

Thank you
[]
    , partHeaders = []
    }
htmlPart = Part
  { partType = "text/html; charset=utf-8"
  , partEncoding = None
  , partFilename = Nothing
  , partContent = renderHtml [shamlet|
<p>Please confirm your email address by clicking on the link below.
<p>
  <a href=#{verurl}>#{verurl}
<p>Thank you
[]
    , partHeaders = []
    }
getVerifyKey = runDB . fmap (join . fmap userVerkey) . get
setVerifyKey uid key = runDB $ update uid [UserVerkey =. Just key]
verifyAccount uid = runDB $ do
  mu <- get uid
  case mu of
    Nothing -> return Nothing
    Just _ -> do
      update uid [UserVerified =. True]
      return $ Just uid
getPassword = runDB . fmap (join . fmap userPassword) . get
setPassword uid pass = runDB $ update uid [UserPassword =. Just
pass]
getEmailCreds email = runDB $ do
  mu <- getBy $ UniqueUser email
  case mu of
    Nothing -> return Nothing
    Just (Entity uid u) -> return $ Just EmailCreds
      { emailCredsId = uid
      , emailCredsAuthId = Just uid
      , emailCredsStatus = isJust $ userPassword u
      , emailCredsVerkey = userVerkey u
      }
getEmail = runDB . fmap (fmap userEmail) . get

getRootR :: Handler RepHtml
getRootR = do
  maid <- maybeAuthId
  defaultLayout [whamlet|
<p>Your current auth ID: #{show maid}
$maybe _ <- maid

```

```

    <p>
      <a href=@{AuthR LogoutR}>Logout
$nothing
    <p>
      <a href=@{AuthR LoginR}>Go to the login page
  ]

main :: IO ()
main = withSqliteConn "email.db3" $ \conn -> do
  runStderrLoggingT $ runSqlConn (runMigration migrateAll) conn
  warpDebug 3000 $ MyEmailApp conn

```

14/email-authentication.hs

14.4. Авторизация

Как только вы получили возможность аутентифицировать пользователей, вы можете использовать их учётные данные для *авторизации* дальнейших запросов. Авторизация в Yesod проста и декларативна: в большинстве случаев необходимо всего лишь добавить методы `authRoute` и `isAuthorized` в ваш экземпляр класса типов Yesod. Давайте рассмотрим пример:

```

{-# LANGUAGE OverloadedStrings, TemplateHaskell, TypeFamilies,
      MultiParamTypeClasses, QuasiQuotes #-}
import Yesod
import Yesod.Auth
import Yesod.Auth.Dummy -- только для тестирования, не используйте в []
                        -- реальной жизни!!!
import Data.Text (Text)
import Network.HTTP.Conduit (Manager, newManager, def)

data MyAuthSite = MyAuthSite
  { httpManager :: Manager
  }

mkYesod "MyAuthSite" [parseRoutes]
/ RootR GET POST
/admin AdminR GET
/auth AuthR Auth getAuth
]

instance Yesod MyAuthSite where
  authRoute _ = Just $ AuthR LoginR

```



```

-- имя маршрута и булево значение определяющее, является ли []
текущий запрос запросом на запись
isAuthorized RootR True = isAdmin
isAuthorized AdminR _ = isAdmin

-- любой может получить доступ к другим страницам
isAuthorized _ _ = return Authorized

isAdmin = do
  mu <- maybeAuthId
  return $ case mu of
    Nothing -> AuthenticationRequired
    Just "admin" -> Authorized
    Just _ -> Unauthorized "You must be an admin"

instance YesodAuth MyAuthSite where
  type AuthId MyAuthSite = Text
  getAuthId = return . Just . credsIdent

  loginDest _ = RootR
  logoutDest _ = RootR

  authPlugins _ = [authDummy]

  authHttpManager = httpManager

instance RenderMessage MyAuthSite FormMessage where
  renderMessage _ _ = defaultFormMessage

getRootR :: Handler RepHtml
getRootR = do
  maid <- maybeAuthId
  defaultLayout [whamlet|
<p>Note: Log in as "admin" to be an administrator.
<p>Your current auth ID: #{show maid}
$maybe _ <- maid
  <p>
    <a href=@{AuthR LogoutR}>Logout
  <p>
    <a href=@{AdminR}>Go to admin page
  <form method=post>
    Make a change (admins only)
    \ #
    <input type=submit>
|]

```

```

postRootR :: Handler ()
postRootR = do
    setMessage "You made some change to the page"
    redirect RootR

getAdminR :: Handler RepHtml
getAdminR = defaultLayout [whamlet|
<p>I guess you're an admin!
<p>
<a href=@{RootR}>Return to homepage
|]

main :: IO ()
main = do
    manager <- newManager def
    warpDebug 3000 $ MyAuthSite manager

```

14/authorization.hs

authRoute должен быть вашей страницей входа, почти всегда это будет AuthR и LoginR. Функция isAuthorized имеет два параметра: запрашиваемый маршрут, и является или нет запросом на запись. На самом деле вы можете переопределить, что именно является запросом на запись, используя метод isWriteRequest, но реализация «из коробки» следует принципам RESTful: все запросы, кроме GET, HEAD, OPTIONS или TRACE — это запросы на запись.

В теле isAuthorized вы можете исполнять любой Handler-код, какой только захотите, что очень удобно. Это означает, что вы можете:

- обращаться к файловой системе (обычный ввод/вывод);
- делать запросы к базе данных;
- получать любые параметры сессии или запроса.

Используя эти техники, вы можете разработать настолько сложную систему авторизации, насколько захотите, или даже привязаться к уже существующей системе, используемой в вашей организации.

14.5. Выводы

Эта глава рассматривает основы настройки аутентификации пользователей, а также то, как встроенные функции авторизации предоставляют пользователям простой и декларативный механизм. Хотя это и сложные концепции со многими подходами, Yesod предоставляет все необходимые строительные блоки для построения системы аутентификации и авторизации, соответствующей вашим требованиям.

Глава 15

Создание каркаса сайта

Вы уже устали запускать маленькие примеры и готовы написать настоящий сайт? Тогда вы читаете нужную главу. Даже при наличии библиотеки Yesod остаётся ещё много шагов, которые нужно пройти, чтобы получить сайт промышленного качества:

- разбор файлов конфигурации
- поддержка сигналов (для *nix)
- более эффективная поддержка статических файлов
- удобное расположение файлов

Каркас сайта является комбинацией множества лучших техник, практикуемых пользователями Yesod и собранных вместе в готовый к использованию «скелет» сайта. Он настоятельно рекомендуется для всех сайтов. Эта глава описывает структуру каркаса, как его использовать, а также некоторые не совсем очевидные его возможности.

Глава практически не содержит примеров кода. Рекомендуется просматривать актуальный каркас по мере чтения главы.

По самой природе каркаса сайта, он является наиболее изменяемым компонентом Yesod и может изменяться от версии к версии. Поэтому информация в этой главе может оказаться немного устаревшей.

15.1. Как создавать каркас

Пакет `yesod`¹ включает в себя и библиотеку, и исполняемый файл (для удобства также названный `yesod`). Этот файл предоставляет несколько команд (запустите `yesod`, чтобы увидеть весь список). Команда `yesod init` создаёт каркас сайта. Она задаёт ряд вопросов, в ответах на которые вам потребуется предоставить необходимую базовую информацию (ваше имя, название проекта и т.д.). После ответа на вопросы в подкаталоге с названием проекта создаётся шаблон проекта.

¹<http://hackage.haskell.org/package/yesod>

Самый важный из задаваемых вопросов — используемый бэкэнд базы данных. У вас будет четыре варианта для выбора: SQLite, PostgreSQL, MongoDB и tiny. Tiny — это не бэкэнд базы данных, а указание, что вы не хотите использовать какую-либо базу данных. Этот выбор так же отключит некоторые дополнительные зависимости, и сайт получится более компактным. Далее в этой главе мы предполагаем, что выбран один из бэкэндов баз данных. Для варианта с бэкэндом tiny будут небольшие отличия.

После создания файлов утилита выводит сообщение с командами для запуска сайта. Существует два варианта команд: один для использования cabal, другой для использования cabal-dev. cabal-dev — это обёртка вокруг cabal, собирающая все зависимости в «песочнице». Его использование — хороший способ избежать поломки вашего сайта в результате установки других пакетов. И поэтому настоятельно рекомендуется. Если cabal-dev не установлен, установить его можно, выполнив команду cabal install cabal-dev.

Обратите внимание, что на самом деле вам нужно использовать команду cabal install --only-dependencies (или cabal-dev install --only-dependencies). Скорее всего, у вас не будут установлены все зависимости необходимые для сайта. Например, ни бэкэнды баз данных, ни минификатор Javascript (hjsmin²) не устанавливаются при установке пакета yesod.

И, наконец, чтобы запустить сайт в режиме разработки, вам нужно использовать yesod devel (или yesod --dev devel). В этом режиме сайт будет автоматически пересобирается и перезагружаться при каждом изменении кода.

15.2. Файловая структура

Каркас сайта создаётся как полноценный пакет Haskell, использующий cabal для сборки. В дополнение к исходным файлам также создаются конфигурационные файлы, шаблоны и статические файлы.

15.2.1. Файл Cabal

Используете ли вы напрямую cabal или косвенно через yesod devel, сборка вашего кода всегда проходит через cabal-файл. Если вы откроете этот файл, то увидите, что там есть и блок для сборки библиотеки, и блок для сборки исполняемого файла. При сборке используется только один, в зависимости от значения флага library-only. Если флаг library-only установлен, то собирается библиотека. И именно так yesod devel запускает приложение. В противоположном случае собирается исполняемый файл.

Флаг library-only должен использоваться только yesod devel; вы никогда не должны явно передавать его cabal. Есть ещё дополнительный флаг dev, который указывает cabal собрать исполняемый файл, но с включением части тех же особенностей, что и сборка с флагом library-only: без оптимизации и с использованием версий с перезагрузкой функций, работающих с Шекспировскими шаблонами.

В общем, собирайте приложение так:

²<http://hackage.haskell.org/package/hjsmin>

- В процессе разработки используйте только `yesod devel`.
- Для сборки боевой версии запускайте `cabal clean && cabal configure && cabal build`. В результате будет создан оптимизированный исполняемый файл в каталоге `dist`.

Раньше использовался флаг `-fproduction`. Поэтому, если вы создали каркас сайта с ранней версией `yesod`, вам, возможно, потребуется использовать этот флаг для сборки боевой версии.

Вы также заметите, что мы указали все расширения языка в `cabal`-файле. Расширения указаны *дважды*: один раз для исполняемого файла, и второй раз для библиотеки. Если вы будете добавлять расширения, то добавляйте их в оба блока.

Вы, возможно, удивитесь, увидев расширение `NoImplicitPrelude`. Мы его включили, поскольку сайт включает свой собственный модуль `Import` с некоторыми изменениями в `Prelude`, которые позволяют сделать работу с `Yesod` более удобной.

И, напоследок, следует отметить список экспортируемых модулей. Если вы добавляете какой-либо модуль в ваше приложение, то вы **должны** обновить этот список для корректной работы `yesod devel`. К сожалению, ни `Cabal`, ни `GHC` не выдадут предупреждение о том, что вы забыли сделать такое обновление, и, вместо предупреждения, вы получите жутко выглядящую ошибку от `yesod devel`.

Одним из планируемых улучшений `yesod devel` является проверка наличия неуказанных модулей.

15.2.2. Маршруты и сущности

Неоднократно на протяжении книги вы встречали подобные комментарии: «Мы определяем пути/сущности квазичитированием для простоты. Для рабочего сайта вам следует использовать внешний файл». Каркас сайта использует такие внешние файлы.

Маршруты определяются в файле `config/routes`, а сущности — в `config/models`. Эти файлы имеют в точности такой же синтаксис, что и квазичитирование, которое повсеместно используется в книге, а `yesod devel` автоматически пересобирает соответствующие модули при изменении этих файлов.

Файл сущностей обрабатывается в `Model.hs`. В этом файле вы вольны объявлять что угодно, но есть несколько рекомендаций:

- Все типы данных, использующиеся в сущностях, **должны** быть импортированы/определены в `Model.hs` выше вызова `persistFile`.
- Вспомогательные утилиты следует определять или в `Import.hs`, или, если они относятся только к сущностям, в файле каталога `Model`, и импортировать в `Import.hs`.

15.2.3. Модули `Foundation` и `Application`

Функция `mkYesod`, которую мы использовали на протяжении книги, объявляет следующее:

- Тип маршрута;

- Функцию отображения маршрута;
- Функцию диспетчеризации.

Функция диспетчеризации ссылается на все остальные функции-обработчики, поэтому все они должны быть или определены в том же файле, что и функция диспетчеризации, или импортированы модулем, в котором определена функция диспетчеризации.

Между тем, функции-обработчики практически обязательно будут ссылаться на тип маршрута. Поэтому и *они* должны или находиться в том же файле, где определён тип маршрута, или импортировать этот файл. Если следовать логике, то получается, что всё приложение, фактически, должно находиться в одном файле!

Очевидно, это не то, что мы хотим. Поэтому вместо использования `mkYesod` каркас сайта использует «расщеплённую» версию этой функции. Модуль `Foundation` вызывает функцию `mkYesodData`, которая определяет тип маршрута и функцию отображения. Так как модуль не определяет функцию диспетчеризации, функции-обработчики не должны быть в той же области видимости. `Import.hs` импортирует `Foundation.hs`, а все модули обработчиков импортируют `Import.hs`.

В `Application.hs` мы вызываем `mkYesodDispatch`, которая создаёт функцию диспетчеризации. Чтобы это заработало, все функции-обработчики должны быть в той же области видимости, так что не забывайте добавлять импорт всех вновь создаваемых модулей обработчиков.

В остальном модуль `Application.hs` достаточно прост. Он предоставляет две функции: `withDevelAppPort`, используемую `yesod devel` для запуска приложения, и `getApplication`, используемую исполняемым файлом для запуска.

Модуль `Foundation.hs` гораздо более интересен. Он:

- Определяет тип-основание;
- Определяет экземпляры ряда классов типов таких как `Yesod`, `YesodAuth` и `YesodPersist`;
- Импортирует файлы сообщений. Если вы поищете строку, начинающуюся с `mkMessage`, то увидите, что она определяет каталог, в котором находятся сообщения (`messages`), и язык по умолчанию (`en`, для английского).

В этот же файл следует добавлять дополнительные экземпляры классов типов для типа-основания, такие как `YesodAuthEmail` или `YesodBreadcrumbs`.

Мы ещё вернёмся к этому файлу ниже, когда будем обсуждать ряд особых реализаций методов класса типов `Yesod`.

15.2.4. Import

Модуль `Import` появился из нескольких часто повторяемых приёмов:

- Я хочу определять несколько вспомогательных функций (возможно, оператор `<> = mappend`), чтобы использовать их во всех обработчиках.
- Я всегда добавляю одни и те же пять инструкций импорта (`Data.Text`, `Control.Applicative` и др.) в каждый модуль-обработчик.

- Я хочу быть уверен, что никогда не использую некоторые «плохие» (evil) функции (**head**, **readFile**, ...) из модуля **Prelude**.

Да, «плохие» — это преувеличение. Если вам интересно, почему я причислил эти функции к плохим: **head** является частичной функцией и выбрасывает исключение для пустого списка, а **readFile** использует ленивый ввод-вывод, который недостаточно быстро высвобождает дескрипторы файлов. Кроме того, **readFile** использует **String** вместо **Text**.

Решение следующее: использовать расширение `NoImplicitPrelude`, реэкспортировать необходимые нам части **Prelude**, добавить всё остальное, нам необходимое, определить наши собственные функции и затем импортировать полученный файл во всех обработчиках.

15.2.5. Модули-обработчики

Модули-обработчики следует помещать внутрь каталога `handler`. Шаблон сайта включает один модуль: `handler/home.hs`. Вам решать, как разделять функции обработчиков на модули, но вот хорошее проверенное правило:

- различные методы, относящиеся к одному маршруту, следует собирать в одном файле, например: `getBlogR` и `postBlogR`;
- связанные маршруты также обычно можно поместить в одном файле, например: `getPeopleR` и `getPersonR`.

Естественно, решение зависит только от вас. Когда вы добавляете новый файл обработчика, убедитесь, что вы сделали следующее:

1. Добавили файл в систему контроля версий (вы ведь *используете* систему контроля версий?).
2. Добавили модуль в файл `cabal`.
3. Добавили его в файл `Application.hs`.
4. Разместили объявление модуля в начале файла и `import Import` на следующей строке.

Одно из планируемых улучшений `Yesod` — автоматизация этих четырёх шагов.

15.3. widgetFile

Достаточно часто требуется добавить на страницу специфичный код `CSS` и `JavaScript`. Вы не хотите помнить о необходимости ручного подключения файлов `Lucius` и `Julius` каждый раз, когда ссылаетесь на файл `Hamlet`. Для этого шаблон сайта предоставляет функцию `widgetFile`.

Если у вас есть функция-обработчик:

```
getHomeR = defaultLayout $(widgetFile "homepage")
```

Yesod будет искать следующие файлы:

```
templates/homepage.hamlet
templates/homepage.lucius
templates/homepage.cassius
templates/homepage.julius
```

Если какой-либо из них существует, то он будет автоматически включён в вывод.

В связи с особенностями работы описанного механизма, если вы запустите приложение через `yesod devel`, и затем создадите новый файл (например, `templates/homepage.julius`), его содержимое *не* будет подключено, пока файл, вызывающий `widgetFile`, не будет перекомпилирован. В подобном случае, вам, возможно, потребуется принудительно сохранить вызывающий файл, чтобы `yesod devel` его перекомпилировал.

15.4. defaultLayout

Одно из первых, что вы, наверное, захотите настроить, — это внешний вид вашего сайта. Разметка, фактически, разделена на два файла:

- `templates/default-layout-wrapper.hamlet` содержит базовый скелет страницы. Этот файл интерпретируется как простой файл Hamlet, а не как виджет, и поэтому не может обращаться к другим виджетам, встраивать интернационализированные строки или добавлять дополнительный код CSS/JS.
- `templates/default-layout.hamlet` — это файл, где вы будете размещать основную часть вашей страницы. Вы **должны** помнить о включении значения `widget` на страницу, т.к. оно представляет собой непосредственное содержание каждой страницы. Этот файл интерпретируется как виджет.

Также, поскольку файл `default-layout` включён с помощью функции `widgetFile`, то любой файл `Lucius`, `Cassius` или `Julius` с именем вида `default-layout.*` будет также автоматически включён.

15.5. Статические файлы

Шаблон сайта автоматически включает подсайт для статических файлов, оптимизированный для обслуживания файлов, которые не изменяются за время жизни текущей сборки приложения. Это означает следующее:

- В процессе генерации идентификаторов статических файлов (например, `static/mylogo.png` становится `mylogo.png`) к ним добавляется параметр строки запроса с хэшем содержимого файла. Всё это происходит во время компиляции.

- Когда `yesod-static` отдаёт ваши статические файлы, он устанавливает заголовки истечения срока действия (`expiration headers`) на даты в далёком будущем и добавляет заголовок `etag` на основе хэша содержимого файла.
- Всякий раз, когда вы вставляете ссылку на `myLogo.png`, при рендеринге маршрута к ней добавляется параметр строки запроса. Если вы меняете логотип и запустите после перекомпиляции ваше новое приложение, строка запроса изменится, что вынудит пользователей скачать новую версию логотипа, игнорируя кэшированную копию.

Кроме того, вы можете задать специфический корневой каталог для статических файлов в файле `Settings.hs`, чтобы отдавать файлы с другого доменного имени. Преимущество такого подхода — отсутствие необходимости передачи файлов куки для запросов статических файлов, а также возможность переложить нагрузку по отдаче статических файлов на CDN³ или сервис наподобие Amazon S3. Смотрите комментарии в файле для более подробной информации.

Другая оптимизация заключается в том, что CSS и Javascript, включённые в ваши виджеты, не будут вставлены в код HTML. Вместо этого их содержимое будет записано во внешний файл и дана ссылка на него. Имя этого файла так же основано на хэше содержимого, что означает следующее:

1. Кэширование будет работать корректно.
2. Yesod может избежать дорогостоящей записи на диск CSS/Javascript файла, если уже существует файл с таким же хэшем.

И, наконец, весь код Javascript автоматически минимизируется, используя `hjsmin`⁴.

15.6. Выводы

Целью этой главы было не объяснить каждую строчку кода в каркасе сайта, а дать общее представление о том, как он работает. Самый лучший способ узнать подробности — это взять и начать писать свой сайт на Yesod, используя каркас.

³Content Delivery Network - сеть доставки (и дистрибуции) контента

⁴<http://hackage.haskell.org/package/hjsmin>

Глава 16

Интернационализация

Пользователи ожидают от наших программ, что те будут разговаривать с ними на одном языке. Также, к сожалению, от нас скорее всего потребуется поддержка более, чем одного языка. В то время, как простая замена строк не представляет собой большой проблемы, корректное формирование фраз в соответствии со всеми правилами грамматики может оказаться нетривиальной задачей. В конце концов, кому из нас приятно видеть в выводе программы «Список 1 файл(ов)»?

Для решения проблемы интернационализации¹ требуется не только обеспечить возможность формировать корректный вывод, необходимо сделать этот процесс простым как для программиста, так и для переводчика, а также достаточно безошибочным. Решение, реализованное в Yesod, позволяет вам:

- Определять язык пользователя, основываясь на информации, переданной в HTTP запросе, с возможностью перезаписи.
- Простой синтаксис для формирования переводов, не требующий знания Haskell. (В конце концов, не всякий переводчик является еще и программистом.)
- Возможность при необходимости использовать всю мощь языка Haskell для нетривиальных грамматических проблем, вместе с набором вспомогательных функций по умолчанию, что покрывает большинство ваших потребностей.
- Полное отсутствие проблем с порядком слов в предложении.

16.1. Краткое содержание

```
Hello: Hello
EnterItemCount: I would like to buy:
Purchase: Purchase
ItemCount count@Int: You have purchased #{showInt count} #{plural count
  count "item" "items" "items"}.
```

¹118N, от англ. internationalization — прим. пер.

```
SwitchLanguage: Switch language to:
Switch: Switch
```

Листинг 16.1: messages/en.msg

```
Hello: Привет
EnterItemCount: Я хотел бы купить:
Purchase: Покупка
ItemCount count: Вы приобрели #{showInt count} #{plural count}
    "единицу" "единицы" "единиц"} товара.
SwitchLanguage: Переключить язык на:
Switch: Переключить
```

Листинг 16.2: messages/ru.msg

```
{-# LANGUAGE OverloadedStrings, QuasiQuotes, TemplateHaskell,
    TypeFamilies,
    MultiParamTypeClasses #-}
import Yesod

data I18N = I18N

mkMessage "I18N" "messages" "en"

plural :: Int -> String -> String -> String -> String
plural 1 x _ _ = x
plural n _ y z = if n < 5 then y else z

showInt :: Int -> String
showInt = show

instance Yesod I18N

instance RenderMessage I18N FormMessage where
    renderMessage _ _ = defaultFormMessage

mkYesod "I18N" [parseRoutes|
/ RootR GET
/buy BuyR GET
/lang LangR POST
|]

getRootR :: Handler RepHtml
getRootR = defaultLayout [whamlet|
<h1>_{MsgHello}
```

```

<form action=@{BuyR}>
  _{MsgEnterItemCount}
  <input type=text name=count>
  <input type=submit value=_{MsgPurchase}>
</form>
<form action=@{LangR} method=post>
  _{MsgSwitchLanguage}
  <select name=lang>
    <option value=en>English
    <option value=ru>Russian
  </select>
  <input type=submit value=_{MsgSwitch}>
</form>

getBuyR :: Handler RepHtml
getBuyR = do
  count <- runInputGet $ ireq intField "count"
  defaultLayout [whamlet|
<p>_{MsgItemCount count}
|]

postLangR :: Handler ()
postLangR = do
  lang <- runInputPost $ ireq textField "lang"
  setLanguage lang
  redirect RootR

main :: IO ()
main = warpDebug 3000 I18N

```

16/i18n-synopsis.hs

16.2. Обзор

Большинство существующих решений, таких как `gettext` или пакеты сообщений Java, работают на принципе подстановки строк. Для подстановки переменных в строки обычно используются функции, похожие на `printf`. Вместо этого в Yesod, как вы могли догадаться, делается ставка на типы. Это даёт нам все обычные преимущества, такие как обнаружение ошибок на этапе компиляции.

Теперь рассмотрим конкретный пример. Допустим, мы хотим от нашего приложения две вещи — приветствовать пользователя и показать количество посетителей, использующих систему в текущий момент. Это можно смоделировать с помощью следующего типа:

```
data MyMessage = MsgHello | MsgUsersLoggedIn Int
```

Мы можем легко написать функцию, преобразующую данный тип в строку на английском языке:

```
toEnglish :: MyMessage -> String
toEnglish MsgHello = "Hello there!"
toEnglish (MsgUsersLoggedIn 1) = "There is 1 user logged in."
toEnglish (MsgUsersLoggedIn i) = "There are" ++ show i ++ " users
logged in."
```

Также мы можем написать подобные функции для других языков. Преимущество такого подхода заключается в возможности использовать всю мощь языка Haskell для решения нетривиальных грамматических задач, особенно образования формы множественного числа.

Вам может показаться, что образование множественного числа не такая уж и сложная задача — всего-то требуется один вариант для одного элемента и другой для любого другого количества. Это может быть справедливо для английского языка, но для произвольного языка это в общем неверно. В русском языке, к примеру, существует отдельная форма множественного числа для количества от 2 до 4, а также шесть различных падежей и вам придется использовать отдельный модуль с логикой, определяющей, какую форму следует использовать.

Обратная сторона медали заключается в необходимости писать все это на Haskell, что едва ли обрадует переводчика. Чтобы решить эту проблему, в Yesod используются файлы сообщений. Чуть ниже мы поговорим о них более подробно.

Допустим, у нас уже есть полный набор функций перевода, но как их использовать? Что нам нужно, так это новая функция, оборачивающая их и вызывающая подходящую функцию перевода в зависимости от языка, выбранного пользователем. Имея её, Yesod сможет автоматически выбирать наиболее подходящую функцию и вызывать её со значениями, которые вы передадите.

Чтобы немного упростить нам жизнь, в Hamlet предусмотрен специальный синтаксис интерполяции, `_ { . . }`, который отвечает за все вызовы функций перевода. Чтобы связать функцию перевода с вашим приложением, используйте класс типов `YesodMessage`.

16.3. Файлы сообщений

Самый простой подход к созданию переводов заключается в использовании файлов сообщений. Идея простая — имеется каталог, содержащий все файлы перевода, по одному файлу на каждый язык. Каждый файл называется в соответствии с кодом языка, например `en.msg`. При этом каждая строка файла содержит одну фразу, которая соответствует одному конструктору вашего типа данных для сообщений.

Сгенерированный сайт, содержит полностью настроенный каталог сообщений.

Для начала поговорим о кодах для обозначения языков. На самом деле их существует два вида — двухбуквенные коды языков и коды вида «язык-СТРАНА». Например, когда я загружаю страницу в веб-браузере, он передает два кода: «en-US» и «en». Это

означает «я бы предпочел американский английский, но если вы его не поддерживаете, сойдет и английский».

Так какой же формат следует использовать в своем приложении? Скорее всего это двухбуквенные коды, если только вы на самом деле не создаете отдельные переводы для разных стран. Это гарантирует, что если кто-то запросит канадский английский, он все равно получит английский. Также Yesod добавляет двухбуквенные коды там, где это уместно. Допустим, пользователь передал следующий список языков:

```
pt-BR, es, he
```

Это означает «я предпочел бы бразильский португальский, затем испанский, а затем иврит». Допустим, ваше приложение поддерживает языки «pt» (общий португальский) и английский, при этом английский используется по умолчанию. Если строго следовать списку, предоставленному пользователем, то будет выбран английский язык. Однако, Yesod преобразует этот список в такой:

```
pt-BR, es, he, pt
```

Другими словами, если только вы не предоставляете отдельные переводы на диалекте языка, используемом в конкретной стране, используйте двухбуквенные коды.

А что на счет файлов сообщений? После работы с Hamlet и Persistent их синтаксис должен показаться вам очень знакомым. Строка начинается с имени сообщения. Поскольку это конструктор данных, оно должно начинаться с заглавной буквы. Затем могут быть параметры. Их следует набирать в нижнем регистре. Они будут использованы в качестве аргументов конструктора данных.

Список аргументов завершается двоеточием, за которым следует переведённая строка, в которой можно использовать традиционный синтаксис интерполяции `#{myVar}`. Ссылаясь на параметры, заданные перед двоеточием, и используя вспомогательные функции, к примеру для образования множественного числа, вы можете получить любые переводы, какие пожелаете.

16.4. Определяем типы

Поскольку мы собираемся создавать тип данных на основе спецификации сообщений, для каждого параметра конструктора данных должен быть указан тип. Для этого используется символ `@`. Например, для создания типа данных `data MyMessage = [] MsgHello | MsgSayAge Int` следует написать:

```
Hello: Привет!
SayAge age@Int: Ваш возраст: #{show age}
```

Но тут имеют место две проблемы:

1. Определять типы параметров в каждом файле не очень-то соответствует принципу DRY («don't repeat yourself», или «не повторяйтесь»).
2. Переводчикам будет затруднительно определить правильный тип.

По этим причинам определение типов данных требуется только в главном языковом файле. Он задается третьим аргументом функции mkMessage. Этот файл так же определяет язык по умолчанию, который используется в случае, если приложение не поддерживает ни один из языков, указанных в списке пользователя.

16.5. Класс типов RenderMessage

Вызов функции mkMessage создает экземпляр класса типов RenderMessage, который представляет собой ядро интернационализации в Yesod. Вот его определение:

```
class RenderMessage master message where
  renderMessage :: master
                 -> [Text] -- ^ языки
                 -> message
                 -> Text
```

Обратите внимание, что у класса RenderMessage есть два параметра типа — тип главного сайта (master) и тип сообщений (message). Теоритически, мы могли бы опустить тип главного сайта, но это означало бы, что каждый сайт должен иметь одинаковый набор переводов для каждого типа сообщения. Когда дело доходит, до совместно используемых библиотек, например для работы с формами, это решение становится нерабочим.

Функция renderMessage принимает аргументы для каждого из параметров типа класа — master и message, и дополнительный параметр со списком языков, понимаемых пользователем, в порядке убывания приоритета. Функция возвращает Text, который должен быть показан пользователю.

Простейший экземпляр класса RenderMessage может в действительности не реализовывать никакого перевода. Вместо этого он может просто отображать одни и те же значения для любого языка. Например:

```
data MyMessage = Hello | Greet Text
instance RenderMessage MyApp MyMessage where
  renderMessage _ _ Hello = "Привет"
  renderMessage _ _ (Greet name) = "Добро_пожаловать,_" <> name <> "!"
```

Обратите внимание, что мы проигнорировали два первых аргумента renderMessage. А теперь изменим этот экземпляр так, чтобы он поддерживал несколько языков:

```
renderEn Hello = "Hello"
renderEn (Greet name) = "Welcome,_" <> name <> "!"
renderRu Hello = "Привет"
renderRu (Greet name) = "Добро_пожаловать,_" <> name <> "!"
instance RenderMessage MyApp MyMessage where
  renderMessage _ ("en":_) = renderEn
```

```
renderMessage _ ("ru":_) = renderRu
renderMessage master (_:langs) = renderMessage master langs
renderMessage _ [] = renderEn
```

Идея довольно проста. Сначала мы объявляем вспомогательные функции для поддержки каждого языка. Затем для каждого языка мы добавляем соответствующее условие в определение функции `renderMessage`. И два последних варианта означают, что если текущий язык не совпал ни с одним из поддерживаемых, взять из списка следующий по приоритету язык, а если в списке больше не осталось языков, то использовать язык по умолчанию (в приведенном примере — английский).

Однако есть вероятность, что вам никогда не придется писать такое самостоятельно, поскольку интерфейс файла сообщений сделает все это за вас. Тем не менее, всегда полезно знать, что происходит под капотом.

16.6. Интерполяция

Одним из способов использования вашего экземпляра класса `RenderMessage` является вызов функции `renderMessage` напрямую. Это будет работать, хотя и является несколько утомительным, поскольку вам придется передавать основное значение и список языков самостоятельно. Вместо этого `Hamlet` предоставляет интерполяцию специализированную для интернационализации, записываемую, как `_ { . . . }`.

Почему подчеркивание? Этот символ стал общепринятым для интернационализации, поскольку он используется в библиотеке `gettext`.

`Hamlet` автоматически преобразует это в вызов `renderMessage`. Получив результат типа `Text`, `Hamlet` использует функцию `toHtml` для получения значения `Html`. Таким образом, специальные символы (`<`, `&`, `>`) автоматически экранируются.

16.7. Фразы, а не слова

В заключение мне хотелось бы дать вам совет по интернационализации. Допустим, у вас есть интернет-магазин, продающий черепахи. Вы собираетесь использовать слово «черепаха» во многих предложениях, например «Вы добавили 4 черепахи в вашу корзину» и «Поздравляем, вы заказали 4 черепахи». Как программист, вы наверняка обратили внимание на дублирование кода — фраза «4 черепахи» встречается дважды. В связи с этим вы можете составить следующий файл сообщений:

```
AddStart: Вы добавили
AddEnd: в вашу корзину.
PurchaseStart: Поздравляем, вы заказали
PurchaseEnd: .
Turtles count@Int: #{show count} #{plural count "черепаху"
  "черепахи" "черепах" }
```

Стойте! Это все, конечно, очень хорошо с точки зрения программиста, но переводчики — не программисты. Даже при таком подходе могут возникнуть сложности:

- В некоторых языках «в вашу корзину» может идти перед «Вы добавили».
- Возможно, «добавили» будет преобразовываться в зависимости от того, добавили вы одну черепаху или несколько.
- Также может возникнуть множество других проблем.

Итак, совет такой: старайтесь переводить целые предложения, а не отдельные слова.

Глава 17

Создание дочернего сайта

Сколько сайтов требуют систему аутентификации? Или функции управления данными (CRUD)? Или блог? Или вики?

Идея в том, что многие веб-сайты включают общие компоненты, которые можно использовать для нескольких сайтов. Однако часто бывает довольно трудно получить модульный код, который действительно был бы `plug-and-play`: такая компонента, вероятно, потребует включения в систему маршрутизации нескольких маршрутов, а также от нее потребуются соответствовать стилю основного сайта.

Решением в Yesod являются подсайты. Подсайт представляет собой набор маршрутов и их обработчиков, которые могут быть легко включены в основной сайт. Используя классы типов, легко убедиться, что основной сайт предоставляет определенные возможности. Также с их помощью несложно получить доступ к стандартной разметке сайта. В свою очередь типобезопасные URL позволяют с легкостью ссылаться с основного сайта на подсайты.

17.1. Привет, мир!

Создание подсайтов достаточно мудрёный процесс, вовлекающий ряд различных типов. Давайте начнем с простого приложения «Привет, мир!»:

```
{-# LANGUAGE QuasiQuotes, TypeFamilies, MultiParamTypeClasses #-}
{-# LANGUAGE TemplateHaskell, FlexibleInstances, OverloadedStrings #-}
import Yesod

-- У подсайтов, также как и у основного сайта, есть основной тип []
-- данных.
data HelloSub = HelloSub

-- Тут аналог знакомого нам mkYesod, с одним дополнительным []
-- параметром.
-- Мы обсудим это позже.
mkYesodSub "HelloSub" [] [parseRoutes]
```

```

/ SubRootR GET
[]

-- Опишем сигнатуру типа для обработчика.
getSubRootR :: Yesod master => GHandler HelloSub master RepHtml
getSubRootR = defaultLayout [whamlet|Welcome to the subsite!|]

-- Давайте создадим основной сайт, который будет его вызывать.
data Master = Master
  { getHelloSub :: HelloSub
  }

mkYesod "Master" [parseRoutes|
/ RootR GET
/subsite SubsiteR HelloSub getHelloSub
|]

instance Yesod Master

-- Опять опишем сигнатуру типа.
getRootR :: GHandler sub Master RepHtml -- так же могли бы заменить
sub на Master
getRootR = defaultLayout [whamlet|
<h1>Добро пожаловать на главную страницу
<p>
  Обратите внимание, что вы также можете посетить #
  <a href=@{SubsiteR SubRootR}>подсайт
  \ .
|]

main = warpDebug 3000 $ Master HelloSub

```

17/hellosub.hs

Этот очень простой пример на самом деле демонстрирует большинство сложностей, связанных с созданием подсайта. Как и в обычном приложении Yesod, в подсайте всё сосредоточено вокруг основного типа данных, в нашем случае HelloSub. Затем мы используем mkYesodSub, во многом так же, как мы используем mkYesod, чтобы создать тип данных маршрута и функции деспетчеризации/рендеринга. (Мы еще вернемся к дополнительному параметру через секунду).

Что интересно, так это сигнатура типа getSubRootR. До сих пор мы старались не обращать внимания на тип данных GHandler, или, в случае необходимости, делать вид, что первые два аргумента типа всегда одинаковы. Теперь мы должны, наконец, узнать правду об этом странном типе данных.

У функции обработчика всегда есть два основных типа, связанных с ней: подсайт и основнойсайт. Когда вы пишете обычное приложение, эти два типа данных одинако-

вы. Однако, если вы работаете с подсайтом, они обязательно будут отличаться. Таким образом, сигнатура типа для `getSubRootR` использует `HelloSub` для первого аргумента и `master` для второго.

Функция `DefaultLayout` принадлежит классу типов `Yesod`. Таким образом, для того, чтобы её вызывать, аргумент типа `master` должен быть экземпляром `Yesod`. Преимуществом такого подхода заключается в том, что любые изменения в методе `defaultLayout` основного сайта будут автоматически отражены в подсайтах.

Когда мы включаем подсайт в определение маршрутов нашего основного сайта, мы должны определить четыре вещи: маршрут, используемый подсайтом в качестве базового (в данном случае `/subsite`), конструктор для маршрутов подсайта (`SubsiteR`), основной тип данных (`HelloSub`) для подсайта и функцию, которая принимает основное значение основного сайта и возвращает основное значение подсайта (`getHelloSub`).

В определении `getRootR`, мы можем видеть, как используется конструктор маршрута. В некотором смысле, `SubsiteR` переводит любой маршрут подсайта в маршрут основного сайта, что позволяет безопасно ссылаться на него из любого шаблона основного сайта.

Часть III

Примеры

Глава 18

Блог: локализация, аутентификация, авторизация и база данных

Это приложение — простой блог. Оно позволяет администратору добавлять записи в блог с помощью текстового редактора (`nicedit`), зарегистрированным пользователям — оставлять комментарии, а также имеет полную поддержку локализации. Это также хороший пример использования базы данных `Persistent`, применения системы авторизации `Yesod` и шаблонов.

Хотя в целом мы рекомендуем размещать шаблоны, определения сущностей `Persist` и маршрутизацию в отдельных файлах, здесь, для удобства, мы будем держать всё это в одном файле. Единственным исключением, как вы увидите ниже, будут локализованные сообщения.

Начнём с расширений языка. В коде сгенерированного шаблона сайта расширения языка указаны в файле `cabal`, так что вам не нужно будет указывать их в ваших файлах `Haskell`.

```
{-# LANGUAGE OverloadedStrings, TypeFamilies, QuasiQuotes,  
           TemplateHaskell, GADTs, FlexibleContexts,  
           MultiParamTypeClasses #-}
```

Теперь импорт.

```
import Yesod  
import Yesod.Auth  
import Yesod.Form.Nic (YesodNic, nicHtmlField)  
import Yesod.Auth.BrowserId (authBrowserId)  
import Data.Text (Text)  
import Network.HTTP.Conduit (Manager, newManager, def)  
import Database.Persist.Sqlite  
  ( ConnectionPool, SqlPersist, runSqlPool, runMigration
```

```

    , createSqlitePool
  )
import Data.Time (UTCTime, getCurrentTime)
import Control.Applicative ((<$>), (<*>), pure)
import Control.Monad.Logger (runNoLoggingT)

```

Сначала мы настроим наши сущности Persistent. Мы создадим наши типы данных (через `mkPersist`) и функцию миграции, которая будет автоматически создавать и обновлять нашу SQL-схему. Если вы используете сервер MongoDB, миграция будет не нужна.

```

share [mkPersist sqlSettings, mkMigrate "migrateAll"] []
      [persistLowerCase]

```

Данные пользователей. В более сложных приложениях мы бы также хранили дату регистрации, отображаемое имя и т.д.

```

User
  email Text
  UniqueUser email

```

Запись в блоге.

```

Entry
  title Text
  posted UTCTime
  content Html

```

Мы должны добавить **deriving**, поскольку `Html` не определяет экземпляры для **Read**, **Show** или **Eq**. Если вы получаете сообщение об ошибке «cannot derive» в своём собственном коде, попробуйте добавить **deriving**.

```

deriving

```

И комментарий к записи.

```

Comment
  entry EntryId
  posted UTCTime
  user UserId
  name Text
  text Textarea
]

```

Каждый сайт имеет тип-основание. Это значение инициализируется перед запуском приложения и доступно в течение всего времени работы. В нашем мы будем хранить пул соединений с базой данных и менеджер HTTP-соединения. Как они будут инициализированы — смотрите в конце этого файла.

```
data Blog = Blog
  { connPool :: ConnectionPool
  , httpManager :: Manager
  }
```

Чтобы упростить локализацию и сделать её дружественной к переводчику, у нас есть специальный формат файлов для перевода сообщений. Для каждого языка существует отдельный файл, который именуется на основе кода языка (например, «en», «es», «de-DE») и размещается в одноимённом каталоге. Также мы указываем основной языковой файл (в данном случае, «en») в качестве языка по умолчанию.

```
mkMessage "Blog" "../messages-blog" "en"
```

Содержимое нашего файла «en»:

```
NotAnAdmin: You must be an administrator to access this page.

WelcomeHomepage: Welcome to the homepage
SeeArchive: See the archive

NoEntries: There are no entries in the blog
LoginToPost: Admins can login to post
NewEntry: Post to blog
NewEntryTitle: Title
NewEntryContent: Content

PleaseCorrectEntry: Your submitted entry had some errors, please
  correct and try again.
EntryCreated title@Text: Your new blog post, #{title}, has been
  created

EntryTitle title@Text: Blog post: #{title}
CommentsHeading: Comments
NoComments: There are no comments
AddCommentHeading: Add a Comment
LoginToComment: You must be logged in to comment
AddCommentButton: Add comment

CommentName: Your display name
CommentText: Comment
CommentAdded: Your comment has been added
PleaseCorrectComment: Your submitted comment had some errors, please
  correct and try again.

HomepageTitle: Yesod Blog Demo
BlogArchiveTitle: Blog Archive
```


А теперь настроим таблицу маршрутизации. У нас будет четыре записи: домашняя страница, страница со списком записей (BlogR), страница отдельной записи (EntryR) и подсайт аутентификации. Обратите внимание, что BlogR и EntryR принимают методы GET и POST. POST используется для добавления в блог, соответственно, новой записи и нового комментария.

```
mkYesod "Blog" [parseRoutes|
/ RootR GET
/blog BlogR GET POST
/blog/#EntryId EntryR GET POST
/auth AuthR Auth getAuth
|]
```

Каждый тип-основание должен быть экземпляром класса типов Yesod. Именно здесь мы настраиваем различные параметры.

```
instance Yesod Blog where
```

Корень нашего приложения. Обратите внимание, что для того, чтобы BrowserID работал правильно, это должен быть корректный URL.

```
approot = ApprootStatic "http://localhost:3000"
```

Наша схема авторизации. Мы хотим, чтобы соблюдались следующие правила:

- Только администраторы могут добавлять новую запись.
- Только зарегистрированные пользователи могут добавлять комментарии.
- Все остальные страницы доступны всем.

Мы сконфигурировали наши маршруты в RESTful-стиле: действия, которые могут вносить изменения, всегда используют метод POST. В результате мы можем легко проверить, является ли запрос запросом на запись: если вторым параметром передано **True**, значит это так.

Во-первых, мы будем авторизовать запросы на добавление новой записи.

```
isAuthorized BlogR True = do
  mauth <- maybeAuth
  case mauth of
    Nothing -> return AuthenticationRequired
    Just (Entity _ user)
      | isAdmin user -> return Authorized
      | otherwise   -> unauthorizedI MsgNotAnAdmin
```

Также мы будем авторизовать запросы на добавление нового комментария.

```

isAuthorized (EntryR _) True = do
  mauth <- maybeAuth
  case mauth of
    Nothing -> return AuthenticationRequired
    Just _   -> return Authorized

```

Все остальные запросы авторизуем всегда.

```
isAuthorized _ _ = return Authorized
```

Указываем, куда должен быть перенаправлен пользователь, если он получает `AuthenticationRequired`.

```
authRoute _ = Just (AuthR LoginR)
```

Здесь мы определяем внешний вид нашего сайта. Функция получает содержимое отдельной страницы и оборачивает его в стандартный шаблон.

```
defaultLayout inside = do
```

Yesod поощряет подход «get-following-post», когда после POST-запроса пользователь перенаправляется на другую страницу. Чтобы позволить POST-странице дать пользователю какой-то отклик, у нас есть функции `getMessage` и `setMessage`. Хорошей идеей будет всегда проверять на наличие ожидающих сообщений в вашей функции `defaultLayout`.

```
mmsg <- getMessage
```

Чтобы объединить вместе HTML, CSS и JavaScript, мы используем виджеты. В конце концов мы должны развернуть всё это в обычный HTML. Для этого есть функция `widgetToPageContent`. Мы передадим ей виджет, состоящий из содержимого, которое мы получили от отдельной страницы (`inside`), и стандартного CSS для всех страниц. Для создания последнего мы используем язык шаблонов Lucius.

```

pc <- widgetToPageContent $ do
  toWidget [lucius]
body {
  width: 760px;
  margin: 1em auto;
  font-family: sans-serif;
}
textarea {
  width: 400px;
  height: 200px;
}
#message {
  color: #900;

```

```
}
|]
    inside
```

В заключение мы используем новый шаблон Hamlet, чтобы обернуть отдельные части (заголовок, содержимое тегов `<head>` и `<body>`) в конечный результат.

```
    hamletToRepHtml [hamlet|
$doctype 5
<html>
  <head>
    <title>#{pageTitle pc}
    ^{pageHead pc}
  <body>
    $maybe msg <- mmsg
      <div #message>#{msg}
    ^{pageBody pc}
|]
```

Это простая функция для проверки, является ли пользователь администратором. В реальных приложениях мы, скорее всего, будем хранить признак администратора в базе данных или проверять через какую-то внешнюю систему. А сейчас я просто жёстко задам адрес своей электронной почты.

```
isAdmin :: User -> Bool
isAdmin user = userEmail user == "michael@snoyman.com"
```

Чтобы получить доступ к базе данных, нам нужно создать экземпляр `YesodPersist`, который указывает, какой сервер мы используем и как выполнять действия.

```
instance YesodPersist Blog where
  type YesodPersistBackend Blog = SqlPersist
  runDB f = do
    master <- getYesod
    let pool = connPool master
    runSqlPool f pool
```

Это синоним типа для удобства. Он определяется автоматически при генерации шаблона сайта.

```
type Form x = Html -> MForm Blog Blog (FormResult x, Widget)
```

Для использования `yesod-form` и `yesod-auth` нам нужен экземпляр `RenderMessage` для `FormMessage`. Это позволит контролировать локализацию отдельных сообщений форм.

```
instance RenderMessage Blog FormMessage where
  renderMessage _ _ = defaultFormMessage
```

Этот экземпляр нужен для того, чтобы использовать встроенный HTML-редактор Nic. Используем значения по умолчанию для CDN-hosted версии Nic.

```
instance YesodNic Blog
```

Чтобы использовать yesod-auth, нам нужен экземпляр YesodAuth.

```
instance YesodAuth Blog where
  type AuthId Blog = UserId
  loginDest _ = RootR
  logoutDest _ = RootR
  authHttpManager = httpManager
```

Мы будем использовать систему BrowserID¹, использующую адрес электронной почты в качестве вашего идентификатора. Это позволит в будущем легко перейти на другие системы для локально аутентифицируемых адресов электронной почты (также входит в yesod-auth).

```
authPlugins _ = [authBrowserId]
```

Эта функция принимает регистрационные данные пользователя (то есть, адрес электронной почты) и возвращает UserId.

```
getAuthId creds = do
  let email = credsIdent creds
      user = User email
  res <- runDB $ insertBy user
  return $ Just $ either entityKey id res
```

Обработчик домашней страницы. Одна важная деталь здесь — это использование 'setTitleI', что позволяет использовать локализованные сообщения в заголовке страницы. Мы также используем это сообщение с '{_Msg...}'-интерполяцией в Hamlet.

```
getRootR :: Handler RepHtml
getRootR = defaultLayout $ do
  setTitleI MsgHomepageTitle
  [whamlet|
<p>_{MsgWelcomeHomepage}
<p>
  <a href=@{BlogR}>_{MsgSeeArchive}
|]
```

¹<https://browserid.org/>

Определяем форму для добавления новых записей. Мы хотим, чтобы пользователь заполнил заголовок и содержание, а дату создания записи заполним автоматически с помощью 'getCurrentTime'.

```
entryForm :: Form Entry
entryForm = renderDivs $ Entry
  <$> areq textField (fieldSettingsLabel MsgNewEntryTitle) Nothing
  <*> aformM (liftIO getCurrentTime)
  <*> areq nicHtmlField (fieldSettingsLabel MsgNewEntryContent) []
Nothing
```

Получаем список всех записей и для администратора отображаем форму создания новой записи.

```
getBlogR :: Handler RepHtml
getBlogR = do
  muser <- maybeAuth
  entries <- runDB $ selectList [] [Desc EntryPosted]
  (entryWidget, enctype) <- generateFormPost entryForm
  defaultLayout $ do
    setTitleI MsgBlogArchiveTitle
    [whamlet|
$if null entries
  <p>_{MsgNoEntries}
$else
  <ul>
    $forall Entity entryId entry <- entries
      <li>
        <a href=@{EntryR entryId}>#{entryTitle entry}
```

У нас есть три варианта: пользователь-администратор вошёл в систему, обычный пользователь вошёл в систему и пользователь не вошёл в систему. В первом случае мы должны отобразить форму создания записи. Во втором мы ничего не делаем. В третьем мы отображаем ссылку для входа.

```
$maybe Entity _ user <- muser
  $if isAdmin user
    <form method=post enctype=#{enctype}>
      ^{entryWidget}
      <div>
        <input type=submit value=_{MsgNewEntry}>
$nothing
  <p>
    <a href=@{AuthR LoginR}>_{MsgLoginToPost}
[]
```

Обрабатываем добавление новой записи. Мы не делаем никакой проверки авторизации, так как `isAuthorized` делает это за нас. Если в форме заданы корректные значения, мы добавляем запись в базу данных и перенаправляем пользователя на эту запись. В противном случае мы просим пользователя попробовать ещё раз.

```
postBlogR :: Handler RepHtml
postBlogR = do
  ((res, entryWidget), enctype) <- runFormPost entryForm
  case res of
    FormSuccess entry -> do
      entryId <- runDB $ insert entry
      setMessageI $ MsgEntryCreated $ entryTitle entry
      redirect $ EntryR entryId
    _ -> defaultLayout $ do
      setTitleI MsgPleaseCorrectEntry
      [whamlet|
<form method=post enctype=#{enctype}>
  ^{entryWidget}
  <div>
    <input type=submit value=_{MsgNewEntry}>
|]

```

Форма для комментариев, очень похожая на `entryForm` выше. Она принимает `EntryID` записи, к которой относится комментарий. Используя `pure`, мы вкладываем это значение в результирующее значение `Comment`, не позволяя ему появиться в сгенерированном HTML.

```
commentForm :: EntryId -> Form Comment
commentForm entryId = renderDivs $ Comment
  <$> pure entryId
  <*> aformM (liftIO getCurrentTime)
  <*> aformM requireAuthId
  <*> areq textField (fieldSettingsLabel MsgCommentName) Nothing
  <*> areq textAreaField (fieldSettingsLabel MsgCommentText) Nothing

```

Показываем отдельную запись, комментарии и, для зарегистрированных пользователей, форму добавления комментария.

```
getEntryR :: EntryId -> Handler RepHtml
getEntryR entryId = do
  (entry, comments) <- runDB $ do
    entry <- get404 entryId
    comments <- selectList [CommentEntry ==. entryId] [Asc CommentPosted]
  return (entry, map entityVal comments)
muser <- maybeAuth
(commentWidget, enctype) <-

```

```

        generateFormPost (commentForm entryId)
    defaultLayout $ do
        setTitleI $ MsgEntryTitle $ entryTitle entry
        [whamlet|
<h1>#{entryTitle entry}
<article>#{entryContent entry}
    <section .comments>
        <h1>_{MsgCommentsHeading}
        $if null comments
            <p>_{MsgNoComments}
        $else
            $forall Comment _entry posted _user name text <- comments
                <div .comment>
                    <span .by>#{name}
                    <span .at>#{show posted}
                    <div .content>#{text}
            <section>
                <h1>_{MsgAddCommentHeading}
                $maybe _ <- muser
                <form method=post enctype=#{enctype}>
                    ^{commentWidget}
                    <div>
                        <input type=submit |
    value=_{MsgAddCommentButton}>
                $nothing
                <p>
                    <a href=@{AuthR LoginR}>_{MsgLoginToComment}
|]

```

Получаем добавляемый комментарий.

```

postEntryR :: EntryId -> Handler RepHtml
postEntryR entryId = do
    ((res, commentWidget), enctype) <-
        runFormPost (commentForm entryId)
    case res of
        FormSuccess comment -> do
            _ <- runDB $ insert comment
            setMessageI MsgCommentAdded
            redirect $ EntryR entryId
        _ -> defaultLayout $ do
            setTitleI MsgPleaseCorrectComment
            [whamlet|
<form method=post enctype=#{enctype}>
    ^{commentWidget}
    <div>

```

```
<input type=submit value=_{MsgAddCommentButton}>
[]
```

Наконец, наша главная функция.

```
main :: IO ()
main = do
  -- создаём новый пул
  pool <- createSqlitePool "blog.db3" 10
  -- выполняем необходимую миграцию
  runNoLoggingT $ runSqlPool (runMigration migrateAll) pool
  -- создаём новый менеджер HTTP
  manager <- newManager def
  -- запускаем наш сервер
  warpDebug 3000 $ Vlog pool manager
```


Глава 19

Wiki: разметка, подсайт чата, источник событий

Этот пример свяжет воедино несколько различных идей. Мы начнём с подсайта чата, позволяющего внедрить виджет чата в любую страницу. Для организации отправки событий клиенту с сервера будем использовать API источника событий (event source) HTML 5.

```
-- @Chat.hs
{-# LANGUAGE OverloadedStrings, TypeFamilies, QuasiQuotes,
      TemplateHaskell, FlexibleInstances,
      MultiParamTypeClasses,
      FlexibleContexts
 #-}
-- | Этот модуль определяет подсайт, который позволяет вам вставлять
-- элемент управления с чатом в любую страницу вашего сайта. Он
-- использует
-- eventsource для отправки сообщений браузеру с сервера.
module Chat where

import Yesod
import Control.Concurrent.Chan (Chan, dupChan, writeChan)
import Data.Text (Text)
import Network.Wai.EventSource (ServerEvent (..), eventSourceAppChan)
import Language.Haskell.TH.Syntax (Type (VarT), Pred (ClassP), mkName)
import Blaze.ByteString.Builder.Char.Utf8 (fromText)
import Data.Monoid (mappend)
import Data.Aeson (toJSON)

-- | Тип-основание нашего подсайта. Мы поддерживаем канал событий,
-- который будет совместно использоваться всеми подключениями.
data Chat = Chat (Chan ServerEvent)
```

```

-- | Нам нужно иметь возможность проверять, что пользователь
--   выполнил вход,
--   и получать его имя (для вывода сообщений).
class (Yesod master, RenderMessage master FormMessage)
  => YesodChat master where
  getUserName :: GHandler sub master Text
  isLoggedIn :: GHandler sub master Bool

-- Теперь создадим наш подсайт. Первый аргумент - это подсайт, что
--   очень напоминает то, как мы раньше использовали mkYesod. Второй
--   аргумент
--   характерен для создания подсайта. В данном случае он означает
--   "основной сайт
--   должен быть экземпляром YesodChat".
--
-- Мы определяем два маршрута: маршрут отправки сообщений от клиента
--   серверу
--   и маршрут открытия потока событий для приёма сообщений от сервера.
mkYesodSub "Chat"
  [ ClassP ''YesodChat [VarT $ mkName "master"]
  ] [parseRoutes]
/send SendR POST
/recv ReceiverR GET
[]

-- | Получить сообщение от пользователя и отправить его всем
--   слушателям.
postSendR :: YesodChat master => GHandler Chat master ()
postSendR = do
  from <- getUserName

  -- Обратите внимание, что мы используем параметры GET для
  --   упрощения
  --   кода Ajax. Можно легко переключиться на POST. Тем не менее,
  --   общий
  --   наш подход всё же является RESTful, так как доступ к этому
  --   маршруту
  --   можно получить только через POST-запрос.
  body <- runInputGet $ ireq textField "message"

  -- Получить канал
  Chat chan <- getYesodSub

  -- Отправить событие с именем пользователя и сообщением всем
  --   слушателям.

```

```

liftIO $ writeChan chan $ ServerEvent Nothing Nothing $ return $
  fromText from 'mappend' fromText ":\u" 'mappend' fromText body

-- | Отправить eventstream-ответ, в который включены все сообщения.
getReceiver :: GHandler Chat master ()
getReceiver = do
  -- Сначала получаем главный канал
  Chat chan0 <- getYesodSub

  -- Мы сделали копию канала, что позволяет нам создавать []
  -- широковещательные каналы.
  chan <- liftIO $ dupChan chan0

  -- Теперь воспользуемся API источника событий. eventSourceAppChan
  -- принимает два параметра: канал событий, из которого выполнять []
  -- чтение,
  -- и запрос WAI. Он возвращает отклик WAI, который мы можем []
  -- вернуть
  -- с помощью sendWaiResponse.
  req <- waiRequest
  res <- lift $ eventSourceAppChan chan req
  sendWaiResponse res

-- | Предоставим виджет, который основной сайт сможет внедрить в []
-- любую страницу.
chatWidget :: YesodChat master
           => (Route Chat -> Route master)
           -> GWidget sub master ()
-- Аргумент toMaster говорит нам как преобразовывать Route Chat в []
-- маршрут
-- основного сайта. Вы можете счесть эту информация избыточной, но []
-- использование
-- такого подхода даёт нам возможность иметь несколько подсайтов с []
-- чатом в пределах
-- одного сайта.
chatWidget toMaster = do
  -- Получим некоторые уникальные идентификаторы, помогающие в []
  -- создании
  -- наших HTML/CSS. Вспомним, что у нас нет представления о том, []
  -- как
  -- выглядит HTML основного сайта. Поэтому мы не можем []
  -- предполагать, что способны
  -- придумать идентификаторы, которые не будут использоваться []
  -- повторно.
  -- Кроме того, несколько экземпляров chatWidget могут быть []
  -- внедрены в одну и ту же страницу.

```

```

chat <- lift newIdent -- обрамляющий div
output <- lift newIdent -- элемент, где отображаются сообщения
input <- lift newIdent -- поле пользовательского ввода

ili <- lift isLoggedIn -- проверить, выполнили ли мы вход
if ili
  then do
    -- Вход выполнен: показать виджет
    [whamlet|
<div ##{chat}>
  <h2>Chat
  <div ##{output}>
    <input ##{input} type=text placeholder="Введите сообщение">
  ]
    -- Немного CSS
    toWidget [lucius|
##{chat} {
  position: absolute;
  top: 2em;
  right: 2em;
}
##{output} {
  width: 200px;
  height: 300px;
  border: 1px solid #999;
  overflow: auto;
}
  ]
    -- И теперь вот такой JavaScript
    toWidgetBody [julius|
// Настроим принимающую сторону
var output = document.getElementById("#{toJSON output}");
var src = new EventSource("@{toMaster_Receiver}");
src.onmessage = function(msg) {
  // Эта функция будет вызвана для каждого нового сообщения
  var p = document.createElement("p");
  p.appendChild(document.createTextNode(msg.data));
  output.appendChild(p);

  // А теперь прокрутим вниз в обрамляющем div так, чтобы было видно
  // последнее сообщение.
  output.scrollTop = output.scrollHeight;
};

// Настроим отправляющую сторону: отправлять сообщение через Ajax □
всякий раз,

```

```

// когда пользователь нажимает Enter.
var input = document.getElementById("#{toJSON input});
input.onkeyup = function(event) {
  var keycode = (event.keyCode ? event.keyCode : event.which);
  if (keycode == '13') {
    var xhr = new XMLHttpRequest();
    var val = input.value;
    input.value = "";
    var params = "?message=" + encodeURIComponent(val);
    xhr.open("POST", "@{toMaster_SendR}" + params);
    xhr.send(null);
  }
}
]

else do
  -- Пользователь не выполнил вход, выдать об этом []
  сообщение.
  master <- lift getYesod
    [whamlet|
<p>
  Вы должны #
  $maybe ar <- authRoute master
    <a href=@{ar}>выполнить вход
  $nothing
    выполнить вход
  \ чтобы воспользоваться чатом.
]

```

19/Chat.hs

Этот модуль является независимым и может использоваться в любом приложении. Ниже мы создадим такое приложение-драйвер — wiki. Наша wiki получит жёстко заданную домашнюю страницу, а затем и wiki-раздел сайта. Мы будем использовать множество динамических фрагментов, позволяющих строить произвольную иерархию страниц в рамках Wiki.

В качестве способа хранения мы просто используем изменяемую ссылку на Мар. Для реального приложения нужно заменить её на подходящую базу данных. Содержимое будет храниться и предоставляться как Markdown. Плагин-пустышка `yesod-auth` предоставит нам способ выполнять (фальшивую) аутентификацию.

```

{-# LANGUAGE OverloadedStrings, TypeFamilies, QuasiQuotes,
      TemplateHaskell, FlexibleInstances, []
      MultiParamTypeClasses,
      FlexibleContexts
  #-}
import Yesod
import Yesod.Auth

```

```

import Yesod.Auth.Dummy (authDummy)
import Chat
import Control.Concurrent.Chan (Chan, newChan)
import Network.Wai.Handler.Warp (run)
import Data.Text (Text)
import qualified Data.Text.Lazy as TL
import qualified Data.IORef as I
import qualified Data.Map as Map
import Text.Markdown (markdown, def) -- предоставляется пакетом markdown

-- | Наш тип-основание включает и подсайт чата, и изменяемую ссылку IORef
--   на словарь
-- содержимого wiki. Обратите внимание, что ключом в этом словаре IORef
--   является список
-- из Text, так как wiki может иметь произвольную иерархию.
--
-- В реальном приложении мы бы предпочли хранить эту информацию в IORef
--   какой-нибудь
-- базе данных.
data Wiki = Wiki
  { getChat :: Chat
  , wikiContent :: I.IORef (Map.Map [Text] Text)
  }

-- Настраиваем наши маршруты, как обычно.
mkYesod "wiki" [parseRoutes]
/ RootR GET -- домашняя страница
/wiki/*Texts WikiR GET POST -- обратите внимание на IORef
-- многокомпонентный путь,
-- необходимый для иерархии wiki
/chat ChatR Chat getChat -- подсайт чата
/auth AuthR Auth getAuth -- подсайт аутентификации
|]

instance Yesod Wiki where
  -- получим работающую ссылку на вход
  authRoute _ = Just $ AuthR LoginR

  -- Наш особый defaultLayout добавит виджет чата на каждую IORef
  --   страницу.
  -- Он также добавит ссылки на вход и на выход вверху страниц.
  defaultLayout widget = do
    pc <- widgetToPageContent $ widget >> chatWidget ChatR
    mmsg <- getMessage
    hamletToRepHtml [hamlet|

```

```

$doctype 5
<html>
  <head>
    <title>#{pageTitle pc}
    ^{pageHead pc}
  <body>
    $maybe msg <- mmsg
      <div .message>#{msg}
    <nav>
      <a href=@{AuthR LoginR}>Войти
      \ | #
      <a href=@{AuthR LogoutR}>Выйти
    ^{pageBody pc}
  ]

-- Довольно стандартный экземпляр YesodAuth. Мы будем использовать []
-- плагин-пустышку,
-- чтобы вы могли создать любое имя, какое пожелаете, и хранить имя []
-- вошедшего
-- пользователя как AuthId.
instance YesodAuth Wiki where
  type AuthId Wiki = Text
  authPlugins _ = [authDummy]
  loginDest _ = RootR
  logoutDest _ = RootR
  getAuthId = return . Just . credsIdent
  authHttpManager = error "authHttpManager" -- authDummy это не []
  использует

-- Просто реализуем аутентификацию на основе нашего yesod-auth.
instance YesodChat Wiki where
  getUsername = requireAuthId
  isLoggedIn = do
    ma <- maybeAuthId
    return $ maybe False (const True) ma

instance RenderMessage Wiki FormMessage where
  renderMessage _ _ = defaultFormMessage

-- Здесь ничего особенного, просто даём ссылку на корень wiki.
getRootR :: Handler RepHtml
getRootR = defaultLayout [whamlet|
<p>Добро пожаловать в Wiki!
<p>
  <a href=@{wikiRoot}>Корень wiki
  ]

```

```

where
  wikiRoot = WikiR []

  -- Форма для получения wiki-содержимого
  wikiForm mtext = renderDivs $ areq textAreaField "Содержимое_
  страницы" mtext

  -- Отобразить страницу wiki и форму для редактирования
  getWikiR :: [Text] -> Handler RepHtml
  getWikiR page = do
    -- Получим ссылку на словарь содержимого
    icontent <- fmap wikiContent getYesod

    -- И прочтём словарь из ссылки
    content <- liftIO $ I.readIORef icontent

    -- Получим содержимое текущей страницы, если оно доступно
    let mtext = Map.lookup page content

    -- Сгенерируем форму с текущим содержимым в качестве значения по
    умолчанию.
    -- Обратите внимание, что мы используем обёртку Textarea для
    получения <textarea>.
    (form, _) <- generateFormPost $ wikiForm $ fmap Textarea mtext
    defaultLayout $ do
      case mtext of
        -- Трактует ввод как markdown. Пакет markdown
        автоматически
        -- выполняет для нас защиту от XSS.
        Just text -> toWidget $ markdown def $ TL.fromStrict text
        Nothing -> [whamlet|<p>Страница не существует|]
      [whamlet|
<h2>Редактировать страницу
<form method=post>
  ^{form}
  <div>
    <input type=submit>
|]

  -- Получим отправленную wiki-страницу и обновим содержимое.
  postWikiR :: [Text] -> Handler RepHtml
  postWikiR page = do
    icontent <- fmap wikiContent getYesod
    content <- liftIO $ I.readIORef icontent
    let mtext = Map.lookup page content
    ((res, form), _) <- runFormPost $ wikiForm $ fmap Textarea mtext

```



```
    case res of
      FormSuccess (Textarea t) -> do
        liftIO $ I.atomicModifyIORef icontent $
          \m -> (Map.insert page t m, ())
        setMessage "Страница_обновлена"
        redirect $ WikiR page
      _ -> defaultLayout [whamlet|
<form method=post>
  ^{form}
  <div>
    <input type=submit>
  ]
main :: IO ()
main = do
  -- Создадим наш канал серверных событий
  chan <- newChan

  -- Изначально у нас пустая база данных wiki-страниц
  icontent <- I.newIORef Map.empty

  -- Запустим наше приложение
  warpDebug 3000 $ Wiki (Chat chan) icontent
```

Глава 20

JSON веб-сервис

Давайте создадим простой веб-сервис: он будет принимать запрос в формате JSON и в нём же отдавать ответ. Мы напишем сервер на WAI/Warp, а клиент — с помощью http-conduit. Мы будем использовать aeson¹ для разбора JSON и его отображения. Мы также могли бы написать сервер и на Yesod, но для такого простого примера большие возможности Yesod немного нам дают.

20.1. Сервер

WAI использует пакет conduit² для того, чтобы обрабатывать тела потоковых запросов, и эффективно формирует ответы при помощи blaze-builder³. aeson⁴ использует attoparsec⁵ для разбора текста; используя attoparsec-conduit⁶ мы получаем простое взаимодействие с WAI. В итоге код выглядит следующим образом:

```
{-# LANGUAGE OverloadedStrings #-}
import Network.Wai (Response, responseLBS, Application, requestBody)
import Network.HTTP.Types (status200, status400)
import Network.Wai.Handler.Warp (run)
import Data.Aeson.Parser (json)
import Data.Conduit.Attoparsec (sinkParser)
import Control.Monad.IO.Class (liftIO)
import Data.Aeson (Value, encode, object, (.=))
import Control.Exception (SomeException)
import Data.ByteString (ByteString)
import Data.Conduit (ResourceT, ($$))
import Control.Exception.Lifted (handle)
```

¹<http://hackage.haskell.org/package/aeson>

²<http://hackage.haskell.org/package/conduit>

³<http://hackage.haskell.org/package/blaze-builder>

⁴<http://hackage.haskell.org/package/aeson>

⁵<http://hackage.haskell.org/package/attoparsec>

⁶<http://hackage.haskell.org/package/attoparsec-conduit>

```

main :: IO ()
main = run 3000 app

app :: Application
app req = handle invalidJson $ do
  value <- requestBody req $$ sinkParser json
  newValue <- liftIO $ modValue value
  return $ responseLBS
    status200
    [("Content-Type", "application/json")]
    $ encode newValue

invalidJson :: SomeException -> ResourceT IO Response
invalidJson ex = return $ responseLBS
  status400
  [("Content-Type", "application/json")]
  $ encode $ object
    [ ("message" .= show ex)
    ]

-- Бизнес-логика будет находиться в этой функции
modValue :: Value -> IO Value
modValue = return

```

20/server.hs

20.2. Клиент

`http-conduit`⁷ был написан как сопутствующий компонент к WAI. Он также использует `conduit` и `blaze-builder` повсеместно, что означает, что мы также получаем простое взаимодействие с `aeson`. Несколько комментариев для тех, кто ещё не знаком с `http-conduit`:

- `Manager` управляет открытыми соединениями таким образом, что повторные запросы к тому же серверу используют то же самое соединение. Чаще всего вам потребуется использовать функцию `withManager` для создания и очистки `Manager`, потому как она безопасна по отношению к исключениям.
- Нам необходимо знать размер тела запроса, который мы не можем определить напрямую из `Builder`. Вместо этого мы преобразуем `Builder` к ленивому `ByteString` и берём размер из него.

⁷<http://hackage.haskell.org/package/http-conduit>

- Есть несколько различных функций для инициализации запроса. Мы используем `http`, которая позволяет нам получать прямой доступ к потоку данных. Есть и другие, более высокоуровневые функции (такие как `httpLbs`), которые позволяют игнорировать вопросы с источниками и дают возможность получить непосредственно тело целиком.

```

{-# LANGUAGE OverloadedStrings #-}
import Network.HTTP.Conduit
  ( http, parseUrl, withManager, RequestBody (RequestBodyLBS)
  , requestBody, method, Response (..)
  )
import Data.Aeson (Value (Object, String))
import Data.Aeson.Parser (json)
import Data.Conduit (($$+-))
import Data.Conduit.Attoparsec (sinkParser)
import Control.Monad.IO.Class (liftIO)
import Data.Aeson (encode, (.=), object)

main :: IO ()
main = withManager $ \manager -> do
  value <- liftIO makeValue
  -- Нам надо знать размер тела запроса, поэтому преобразуем его к
  -- ByteString
  let valueBS = encode value
      req' <- liftIO $ parseUrl "http://localhost:3000/"
      let req = req' { method = "POST", requestBody = RequestBodyLBS $
          valueBS }
      res <- http req manager
      resValue <- responseBody res $$+- sinkParser json
      liftIO $ handleResponse resValue

  -- Функция бизнес-логики приложения, создающая значение для запроса
makeValue :: IO Value
makeValue = return $ object
  [ ("foo" .= ("bar" :: String))
  ]

  -- Функция бизнес-логики приложения, обрабатывающая ответ от сервера
handleResponse :: Value -> IO ()
handleResponse = print

```

Глава 21

Полнотекстовый поиск с использованием Sphinx

Sphinx¹ — это сервер полнотекстового поиска, благодаря которому реализован поиск на многих сайтах, включая сайт самого Yesod. Код, необходимый для интеграции Sphinx и Yesod довольно краток, но, в то же время, он затрагивает несколько непростых моментов, а потому является прекрасной иллюстрацией того, как использовать некоторые детали внутреннего устройства Yesod.

По существу нам предстоит реализовать три вещи:

- Сохранение данных, по которым мы хотели бы вести поиск. Для этого используется довольно незатейливый код Persistent, и мы не будем подробно останавливаться на этом моменте.
- Доступ из Yesod к результатам поиска Sphinx. На самом деле, благодаря пакету sphinx², это довольно просто.
- Предоставление содержимого документов серверу Sphinx. Вот где происходят интересные вещи! Вы узнаете, как напрямую передавать контент из базы данных напрямую в XML, который затем передаётся клиенту.

21.1. Установка Sphinx

В отличие от большинства других примеров, здесь для начала нам потребуется настроить и запустить сервер Sphinx. Я не собираюсь затрагивать все особенности Sphinx, отчасти, потому что они не относятся к делу, но в основном — потому что я далеко не эксперт в Sphinx.

Sphinx предоставляет три основных программы. Демон searchd непосредственно принимает запросы от клиента (в данном случае — нашего веб-приложения) и возвращает результат поиска. Программа indexer обрабатывает документы и создаёт индекс

¹<http://sphinxsearch.com/>

²<http://hackage.haskell.org/package/sphinx>

поиска. Утилита `search` предназначена для отладки, она посылает простые поисковые запросы серверу Sphinx.

Настройки Sphinx содержат два важных параметра — источник (`source`) и индекс (`index`). Параметр `source` указывает Sphinx, откуда следует считывать информацию о документах. Поддерживается как прямое чтение из MySQL и PostgreSQL, так и использование XML-формата, известного как `xmlpipe2`. Им мы и воспользуемся. Это не только даст гибкость в плане выбора Persistent-бэкендов, но и продемонстрирует некоторые мощные концепции Yesod.

Второй важный параметр — это `index`. Sphinx может поддерживать несколько индексов одновременно, что позволяет организовать поиск для нескольких служб с помощью одного сервера. Каждому параметру `index` соответствует параметр `source`, определяющий, откуда брать данные.

Наше приложение будет предоставлять специальный URL (`/search/xmlpipe`) для генерации XML-файла для Sphinx, который и будет передаваться в `indexer`. В конфигурационный файл Sphinx следует прописать следующее:

```
source searcher_src
{
    type = xmlpipe2
    xmlpipe_command = curl http://localhost:3000/search/xmlpipe
}

index searcher
{
    source = searcher_src
    path = /var/data/searcher
    docinfo = extern
    charset_type = utf-8
}
```

Чтобы построить индекс, необходимо запустить `indexer searcher`. Очевидно, это не будет работать, пока мы не запустим наше приложение. В рабочей версии сайта, эту команду следует запускать периодически с помощью `crontab`, чтобы индекс регулярно обновлялся.

21.2. Базовая настройка Yesod

Создадим новое Yesod-приложение. Нам понадобится одна таблица в базе данных для хранения документов, которые будут состоять из заголовка и текста. Эту таблицу мы будем хранить в SQLite. Создадим маршруты для поиска, добавления и просмотра документов, а также для генерации `xmlpipe`-файла для Sphinx.

```
share [mkPersist sqlSettings, mkMigrate "migrateAll"] [persist|
Doc
    title Text
    content Textarea
```

```

[]

data Searcher = Searcher ConnectionPool

mkYesod "Searcher" [parseRoutes|
/ RootR GET
/doc/#DocId DocR GET
/add-doc AddDocR POST
/search SearchR GET
/search/xmlpipe XmlpipeR GET
|]

instance Yesod Searcher

instance YesodPersist Searcher where
  type YesodPersistBackend Searcher = SqlPersist

  runDB action = do
    Searcher pool <- getYesod
    runSqlPool action pool

instance RenderMessage Searcher FormMessage where
  renderMessage _ _ = defaultMessage

```

Надеюсь, всё это должно быть уже знакомым на данный момент. Далее мы определим две формы — одну для создания документа и одну для поиска:

```

addDocForm :: Html -> MForm Searcher Searcher (FormResult Doc, Widget)
addDocForm = renderTable $ Doc
  <$> areq textField "Title" Nothing
  <*> areq textareaField "Contents" Nothing

searchForm :: Html -> MForm Searcher Searcher (FormResult Text, []
  Widget)
searchForm = renderDivs $ areq (searchField True) "Query" Nothing

```

Передача параметра **True** в функцию `searchField` обеспечивает автоматическую установку фокуса на поле ввода при загрузке страницы. Наконец, объявляем обработчики для главной страницы (на ней отображаются формы добавления и поиска документов), а также для отображения и добавления документа:

```

getRootR :: Handler RepHtml
getRootR = do
  docCount <- runDB $ count ([] :: [Filter Doc])
  ((_, docWidget), _) <- runFormPost addDocForm
  ((_, searchWidget), _) <- runFormGet searchForm

```

```

let docs = if docCount == 1
  then "There is currently 1 document."
  else "There are currently " ++ show docCount ++
    " documents."
defaultLayout [whamlet|
<p>Welcome to the search application. #{docs}
<form method=post action=@{AddDocR}>
  <table>
    ^{docWidget}
    <tr>
      <td colspan=3>
        <input type=submit value="Add document">
<form method=get action=@{SearchR}>
  ^{searchWidget}
  <input type=submit value=Search>
|]

postAddDocR :: Handler RepHtml
postAddDocR = do
  ((res, docWidget), _) <- runFormPost addDocForm
  case res of
    FormSuccess doc -> do
      docid <- runDB $ insert doc
      setMessage "Document added"
      redirect $ DocR docid
    _ -> defaultLayout [whamlet|
<form method=post action=@{AddDocR}>
  <table>
    ^{docWidget}
    <tr>
      <td colspan=3>
        <input type=submit value="Add document">
|]

getDocR :: DocId -> Handler RepHtml
getDocR docid = do
  doc <- runDB $ get404 docid
  defaultLayout $
    [whamlet|
<h1>#{docTitle doc}
<div .content>#{docContent doc}
|]

```


21.3. Поиск

Теперь, когда мы разобрались со всей рутинной, займёмся непосредственно поиском. Нам понадобятся три вещи для отображения результатов поиска: идентификатор документа, к которому относится конкретная строка, заголовок этого документа, а также выдержка. Выдержки представляют собой выделенные участки документа, содержащие поисковый запрос (рис. 21.1).

Query ✕

Results

[United States Constitution](#)

... Departments. The President shall have **Power** to fill up all Vacancies ... people. Amendment 11 The Judicial **power** of the United States shall ... jurisdiction. 2. Congress shall have **power** to enforce this article by ... 5. The Congress shall have **power** to enforce, by appropriate legislation ...

[Declaration of Independence](#)

... of and superior to civil **power**. He has combined with others ... , and declaring themselves invested with **power** to legislate for us in ... independent states, they have full **power** to levy war, conclude peace ...

Рис. 21.1: Результат поиска

Давайте начнём с определения типа данных Result:

```
data Result = Result
  { resultId :: DocId
  , resultTitle :: Text
  , resultExcerpt :: Html
  }
```

Теперь взглянем на обработчик запроса поиска:

```
getSearchR :: Handler RepHtml
getSearchR = do
  ((formRes, searchWidget), _) <- runFormGet searchForm
  searchResults <-
    case formRes of
      FormSuccess qstring -> getResults qstring
      _ -> return []
  defaultLayout $ do
    toWidget [lucius|
.excerpt {
```

```

        color: green; font-style: italic
    }
    .match {
        background-color: yellow;
    }
    []

    [whamlet|
<form method=get action=@{SearchR}>
    ^{searchWidget}
    <input type=submit value=Search>
$if not $ null searchResults
    <h1>Results
    $forall result <- searchResults
        <div .result>
            <a href=@{DocR $ resultId result}>#{resultTitle result}
            <div .excerpt>#{resultExcerpt result}
    ]
    ]

```

Ничего волшебного здесь не происходит: мы просто полагаемся на `searchForm`, объявленную выше, и ещё необъявленную `getResults`. Эта функция просто принимает строку с поисковым запросом и возвращает список результатов. Здесь мы впервые взаимодействуем с API сервера Sphinx. Мы будем использовать две функции: `query` будет возвращать список совпадений, а `buildExcerpts` — выделенные отрывки. Сначала разберём функцию `getResults`:

```

getResults :: Text -> Handler [Result]
getResults qstring = do
    sphinxRes' <- liftIO $ S.query config "searcher" (unpack qstring)
    case sphinxRes' of
        ST.Ok sphinxRes -> do
            let docids = map (Key . PersistInt64 . ST.documentId) $
                    ST.matches sphinxRes
                fmap catMaybes $ runDB $ forM docids $ \docid -> do
                    mdoc <- get docid
                    case mdoc of
                        Nothing -> return Nothing
                        Just doc -> liftIO $ Just <$>
                            getResult docid doc qstring
            _ -> error $ show sphinxRes'
    where
        config = S.defaultConfig
            { S.port = 9312
            , S.mode = ST.Any
            }

```

Функция `query` принимает три аргумента: параметры конфигурации, строку с именем индекса, по которому будет производиться поиск, и поисковый запрос. Функция возвращает список идентификаторов документов, содержащих искомую строку. Здесь есть небольшая хитрость, связанная с тем, что возвращаемые идентификаторы имеют тип `Int64`, в то время как нам нужен `DocId`. Благодаря тому, что SQL-бэкенды `Persistent` используют конструктор `PersistInt64` для идентификаторов, мы сможем получить требуемые значения, просто обернув их соответствующим образом.

Если вы работаете с бэкендом, использующим нечисловые идентификаторы, например `MongoDB`, вам придётся разработать какой-нибудь более изощрённый способ.

Затем мы проходим в цикле по этим идентификаторам, получая список `[Maybe Result]`, и используем функцию `catMaybes` для преобразования этого списка в `[Result]`. В `where`-клизе мы определяем локальные настройки, которые замещают номер порта, используемый по умолчанию, а также задают режим поиска, в котором возвращаются все документы, содержащие хотя бы одно слово из поискового запроса.

Наконец, взглянем на функцию `getResult`:

```
getResult :: DocId -> Doc -> Text -> IO Result
getResult docid doc qstring = do
  excerpt' <- S.buildExcerpts
    excerptConfig
    [T.unpack $ escape $ docContent doc]
    "searcher"
    (unpack qstring)
  let excerpt =
        case excerpt' of
          ST.Ok bss -> preEscapedToMarkup $ decodeUtf8With \
            ignore $ L.concat bss
          _ -> ""
      return Result
        { resultId = docid
        , resultTitle = docTitle doc
        , resultExcerpt = excerpt
        }
  where
    excerptConfig = E.altConfig { E.port = 9312 }

escape :: Textarea -> Text
escape =
  T.concatMap escapeChar . unTextarea
  where
    escapeChar '<' = "&lt;"
    escapeChar '>' = "&gt;"
    escapeChar '&' = "&amp;"
    escapeChar c = T.singleton c
```

Функция `buildExcerpts` принимает четыре аргумента: параметры конфигурации, содержимое документа, строку с именем индекса и поисковый запрос. Обратите внимание на экранирование специальных символов в документе. Sphinx его не производит, поэтому нам приходится заниматься этим самим.

Результат поиска, который мы получаем от Sphinx, представляет собой список ленивых `ByteString`. Но мы, само собой разумеется, предпочли бы получить `Html`. Поэтому мы конкатенируем элементы списка в одну ленивую `ByteString`, декодируем её в ленивый текст (игнорируя некорректные последовательности символов UTF-8), а затем используем функцию `preEscapedToMarkup` для того, чтобы тэги, используемые для выделения найденных совпадений, не экранировались. Вот пример полученного в итоге HTML-кода:

```
&#8230; Departments. The President shall have  to fill up all Vacancies
&#8230; people. Amendment 11 The Judicial  of the United States shall
&#8230; jurisdiction. 2. Congress shall have  to enforce this article by
&#8230; 5. The Congress shall have  to enforce, by appropriate legislation
&#8230;
```

21.4. Генерируем `xmlpipe`

Самое интересное мы приберегли напоследок. Для большинства обработчиков в Yesod рекомендуемый подход заключается в загрузке данных из базы в оперативную память и генерации вывода, основанного на этих данных. Так проще, но намного важнее то, что такой подход более устойчив к ошибкам. Если возникнет проблема во время загрузки данных из базы, пользователь получит должный код ответа 500.

Что я имею в виду, говоря «должный код ответа 500»? Если вы начнёте передавать данные клиенту и в середине процесса наткнётесь на исключение, у вас не будет возможности изменить код ответа. Пользователь получит ответ с кодом 200, просто передача данных прервётся посередине. Помимо того, что частично сгенерированная страница сама по себе сбивает с толку, такое поведение ещё и не соответствует спецификации протокола HTTP.

Генерация `xmlpipe`-вывода является прекрасным примером, где следует использовать альтернативный подход. У нас может быть огромное количество документов (код сайта `yesodweb.com` оперирует десятками тысяч документов), а документы могут иметь размер порядка нескольких сотен килобайт. Если мы не воспользуемся точным подходом, это может привести к большому времени ответа и использованию большого объёма оперативной памяти.

Так каким именно образом мы можем генерировать ответ в виде потока? Как мы узнаем из главы, посвящённой WAI, у нас есть конструктор `ResponseSource`, который использует поток элементов типа `Builder` из библиотеки `blaze-builder`. На стороне

Yesod, вместо того, чтобы генерировать ответ, как обычно, мы можем послать WAI-ответ напрямую с помощью функции `sendWaiResponse`. Таким образом имеется по крайней мере две части этой головоломки.

Теперь нам известно, что из неких XML-данных требуется создать поток `Builder`'ов. К счастью, пакет `xml-conduit`³ предоставляет интерфейс непосредственно для этого. Вообще, пакет `xml-conduit` предоставляет высокоуровневый интерфейс для работы с документами целиком, однако в нашем случае понадобится низкоуровневый интерфейс `Event`, дабы обеспечить минимальное использование памяти. Вот функция, которая нам нужна:

```
renderBuilder :: Resource m => RenderSettings -> Conduit Event m
              Builder b
```

В переводе на русский язык это означает, что `renderBuilder` принимает некие настройки (мы воспользуемся настройками по умолчанию), и преобразует поток `Event`'ов в поток `Builder`'ов. Выглядит неплохо. Всё, что нам теперь нужно — это поток `Event`'ов.

Кстати говоря, а как должен выглядеть наш XML-документ? Он довольно прост, имеется родительский элемент `sphinx:docset`, элемент `sphinx:schema`, содержащий одиночный элемент `sphinx:field` (который определяет элемент с содержимым документа), а затем по одному элементу `sphinx:document` для каждого документа в базе данных. Эти элементы будут содержать атрибут `id` и дочерний элемент `content`.

```
<sphinx:docset xmlns:sphinx="http://sphinxsearch.com/">
  <sphinx:schema>
    <sphinx:field name="content"/>
  </sphinx:schema>
  <sphinx:document id="1">
    <content>bar</content>
  </sphinx:document>
  <sphinx:document id="2">
    <content>foo bar baz</content>
  </sphinx:document>
</sphinx:docset>
```

Листинг 21.1: Пример xmlpipe-документа

Каждый XML-документ будет начинаться с одинаковых событий (начать `docset`, начать `schema` и т.д.) и заканчиваться одним и тем же событием (закончить `docset`). Вот соответствующий код:

```
toName :: Text -> X.Name
toName x = X.Name x (Just "http://sphinxsearch.com/") (Just "sphinx")

docset, schema, field, document, content :: X.Name
docset = toName "docset"
```

³<http://hackage.haskell.org/package/xml-conduit>

```

schema = toName "schema"
field = toName "field"
document = toName "document"
content = "content" -- no prefix

startEvents, endEvents :: [X.Event]
startEvents =
  [ X.EventBeginDocument
  , X.EventBeginElement docset []
  , X.EventBeginElement schema []
  , X.EventBeginElement field [("name", [X.ContentText "content"])]
  , X.EventEndElement field
  , X.EventEndElement schema
  ]

endEvents =
  [ X.EventEndElement docset
  ]

```

Теперь, когда у нас есть оболочка нашего документа, нам нужно получить Event для каждого конкретного документа. Это можно сделать при помощи довольно простой функции:

```

entityToEvents :: (Entity Doc) -> [X.Event]
entityToEvents (Entity docid doc) =
  [ X.EventBeginElement document [("id", [X.ContentText $
    toPathPiece docid])]
  , X.EventBeginElement content []
  , X.EventContent $ X.ContentText $ unTextarea $ docContent doc
  , X.EventEndElement content
  , X.EventEndElement document
  ]

```

Мы открываем элемент `document` с атрибутом `id`, затем открываем элемент `content`, вставляем содержимое документа, после чего закрываем оба элемента. Для преобразования `DocId` в значение типа `Text` мы используем функцию `toPathPiece`. Теперь нам нужно как-то преобразовать поток сущностей в поток событий. Для этого воспользуемся функцией `concatMap` из модуля `Data.Conduit.List`: `CL.concatMap entityToEvents`.

Но чего мы *действительно* хотим, так это создать поток этих событий непосредственно из базы данных. На протяжении большей части книги, мы использовали функцию `selectList`, однако `Persistent` также предоставляет (более мощную) функцию `selectSourceConn`. С её помощью мы получаем в итоге следующую функцию:

```

docSource :: Connection -> C.Source (C.ResourceT IO) X.Event
docSource conn = selectSourceConn conn [] [] C.$= CL.concatMap
  entityToEvents

```

Оператор `$=` соединяет источник с каналом, возвращая новый источник. Теперь, когда у нас есть источник `Event`'ов, всё, что нам нужно — это окружить его событиями начала и конца документа. Благодаря тому, что `Source` является экземпляром класса `Monad`, это проще простого:

```
fullDocSource :: Connection -> C.Source (C.ResourceT IO) X.Event
fullDocSource conn = mconcat
  [ CL.sourceList startEvents
  , docSource conn
  , CL.sourceList endEvents
  ]
```

Мы почти закончили, осталось только применить всё это в функции `getXmlpipeR`. Нам необходимо установить соединение с базой данных. Обычно соединения с БД берутся и автоматически возвращаются с помощью функции `runDB`. В нашем случае требуется получить соединение и иметь к нему доступ до тех пор, пока тело ответа не будет передано полностью. Для этого воспользуемся функцией `takeResource`, которая регистрирует действие по очистке в монаде `ResourceT`.

Все WAI-приложения живут в трансформаторе монады `ResourceT`. Вы можете получить больше информации о `ResourceT` в приложении, посвящённом потокам.

По умолчанию ресурс не возвращается в пул. Это связано с правильной обработкой ошибок, что, впрочем, не относится к нашему случаю. Поэтому мы должны принудительно вернуть соединение в пул:

```
getXmlpipeR :: Handler RepXml
getXmlpipeR = do
  Searcher pool <- getYesod
  let headers = [("Content-Type", "text/xml")]
      managedConn <- lift $ takeResource pool
      conn = mrValue managedConn
      lift $ mrReuse managedConn True
      let source = fullDocSource conn C.$= renderBuilder def
          sendWaiResponse $ ResponseSource status200 headers source
```

Пул соединений получается из базовой переменной, после чего отправляется WAI-ответ. Мы используем конструктор `ResponseSource`, передавая ему код ответа, а также заголовки и тело ответа.

21.5. Весь код

```
{-# LANGUAGE OverloadedStrings, TypeFamilies, TemplateHaskell,
      QuasiQuotes, MultiParamTypeClasses, GADTs, FlexibleContexts
```

```

#-}
import Yesod
import Control.Monad.Logger (runStderrLoggingT)
import Data.Text (Text)
import Control.Applicative ((<$>), (<*>))
import Database.Persist.Sqlite
import Database.Persist.Query.GenericSql (selectSourceConn)
import Database.Persist.Store (PersistValue (PersistInt64))
import qualified Text.Search.Sphinx as S
import qualified Text.Search.Sphinx.Types as ST
import qualified Text.Search.Sphinx.ExcerptConfiguration as E
import qualified Data.ByteString.Lazy as L
import Data.Text.Lazy.Encoding (decodeUtf8With)
import Data.Text.Encoding.Error (ignore)
import Data.Maybe (catMaybes)
import Control.Monad (forM)
import qualified Data.Text as T
import Text.Blaze (preEscapedToMarkup)
import qualified Data.Conduit as C
import qualified Data.Conduit.List as CL
import qualified Data.XML.Types as X
import Network.Wai (Response (ResponseSource))
import Network.HTTP.Types (status200)
import Text.XML.Stream.Render (renderBuilder, def)
import Data.Monoid (mconcat)
import Data.Conduit.Pool (takeResource, mrValue, mrReuse)

share [mkPersist sqlSettings, mkMigrate "migrateAll"] []
  [persistUpperCase]
Doc
  title Text
  content Textarea
[]

data Searcher = Searcher ConnectionPool

mkYesod "Searcher" [parseRoutes]
/ RootR GET
/doc/#DocId DocR GET
/add-doc AddDocR POST
/search SearchR GET
/search/xmlpipe XmlpipeR GET
[]

instance Yesod Searcher

```



```

instance YesodPersist Searcher where
  type YesodPersistBackend Searcher = SqlPersist

  runDB action = do
    Searcher pool <- getYesod
    runSqlPool action pool

instance RenderMessage Searcher FormMessage where
  renderMessage _ _ = defaultMessage

addDocForm :: Html -> MForm Searcher Searcher (FormResult Doc, Widget)
addDocForm = renderTable $ Doc
  <$> areq textField "Title" Nothing
  <*> areq textareaField "Contents" Nothing

searchForm :: Html -> MForm Searcher Searcher (FormResult Text, Widget)
searchForm = renderDivs $ areq (searchField True) "Query" Nothing

getRootR :: Handler RepHtml
getRootR = do
  docCount <- runDB $ count ([] :: [Filter Doc])
  ((_, docWidget), _) <- runFormPost addDocForm
  ((_, searchWidget), _) <- runFormGet searchForm
  let docs = if docCount == 1
    then "There is currently 1 document."
    else "There are currently " ++ show docCount ++ " documents."
  defaultLayout [whamlet|
<p>Welcome to the search application. #{docs}
<form method=post action=@{AddDocR}>
  <table>
    ^{docWidget}
    <tr>
      <td colspan=3>
        <input type=submit value="Add document">
<form method=get action=@{SearchR}>
  ^{searchWidget}
  <input type=submit value=Search>
|]

postAddDocR :: Handler RepHtml
postAddDocR = do
  ((res, docWidget), _) <- runFormPost addDocForm
  case res of
    FormSuccess doc -> do

```

```

        docid <- runDB $ insert doc
        setMessage "Document added"
        redirect $ DocR docid
    _ -> defaultLayout [whamlet|
<form method=post action=@{AddDocR}>
  <table>
    ^{docWidget}
    <tr>
      <td colspan=3>
        <input type=submit value="Add document">
    ]
]

getDocR :: DocId -> Handler RepHtml
getDocR docid = do
  doc <- runDB $ get404 docid
  defaultLayout $
    [whamlet|
<h1>#{docTitle doc}
<div .content>#{docContent doc}
|]

data Result = Result
  { resultId :: DocId
  , resultTitle :: Text
  , resultExcerpt :: Html
  }

getResult :: DocId -> Doc -> Text -> IO Result
getResult docid doc qstring = do
  excerpt' <- S.buildExcerpts
    excerptConfig
    [escape $ docContent doc]
    "searcher"
    qstring
  let excerpt =
        case excerpt' of
          ST.Ok t -> preEscapedToMarkup $ T.concat t
            _ -> ""
  return Result
    { resultId = docid
    , resultTitle = docTitle doc
    , resultExcerpt = excerpt
    }
where
  excerptConfig = E.altConfig { E.port = 9312 }

```

```

escape :: Textarea -> Text
escape =
  T.concatMap escapeChar . unTextarea
  where
    escapeChar '<' = "&lt;"
    escapeChar '>' = "&gt;"
    escapeChar '&' = "&amp;"
    escapeChar c   = T.singleton c

getResults :: Text -> Handler [Result]
getResults qstring = do
  sphinxRes' <- liftIO $ S.query config "searcher" qstring
  case sphinxRes' of
    ST.Ok sphinxRes -> do
      let docids = map (Key . PersistInt64 . ST.documentId) $
          ST.matches sphinxRes
          fmap catMaybes $ runDB $ forM docids $ \docid -> do
              mdoc <- get docid
              case mdoc of
                Nothing -> return Nothing
                Just doc -> liftIO $ Just <$> getResult docid []
          doc qstring
      _ -> error $ show sphinxRes'
    _ -> error $ show sphinxRes'
  where
    config = S.defaultConfig
      { S.port = 9312
      , S.mode = ST.Any
      }

getSearchR :: Handler RepHtml
getSearchR = do
  ((formRes, searchWidget), _) <- runFormGet searchForm
  searchResults <-
    case formRes of
      FormSuccess qstring -> getResults qstring
      _ -> return []
  defaultLayout $ do
    toWidget [lucius|
.excerpt {
  color: green; font-style: italic
}
.match {
  background-color: yellow;
}
|]
  [whamlet|

```

```

<form method=get action=@{SearchR}>
  ^{searchWidget}
  <input type=submit value=Search>
$if not $ null searchResults
  <h1>Results
  $forall result <- searchResults
    <div .result>
      <a href=@{DocR $ resultId result}>#{resultTitle result}
      <div .excerpt>#{resultExcerpt result}
  ]
]

getXmlpipeR :: Handler RepXml
getXmlpipeR = do
  Searcher pool <- getYesod
  let headers = [("Content-Type", "text/xml")]
  managedConn <- lift $ takeResource pool
  let conn = mrValue managedConn
  lift $ mrReuse managedConn True
  let source = fullDocSource conn C.$= renderBuilder def
      flushSource = C.mapOutput C.Chunk source
  sendWaiResponse $ ResponseSource status200 headers flushSource

entityToEvents :: (Entity Doc) -> [X.Event]
entityToEvents (Entity docid doc) =
  [ X.EventBeginElement document [("id", [X.ContentText $
    toPathPiece docid])]
  , X.EventBeginElement content []
  , X.EventContent $ X.ContentText $ unTextarea $ docContent doc
  , X.EventEndElement content
  , X.EventEndElement document
  ]

fullDocSource :: Connection -> C.Source (C.ResourceT IO) X.Event
fullDocSource conn = mconcat
  [ CL.sourceList startEvents
  , docSource conn
  , CL.sourceList endEvents
  ]

docSource :: Connection -> C.Source (C.ResourceT IO) X.Event
docSource conn = C.transPipe runStderrLoggingT $ selectSourceConn []
  conn [] [] C.$= CL.concatMap entityToEvents

toName :: Text -> X.Name
toName x = X.Name x (Just "http://sphinxsearch.com/") (Just "sphinx")

```

```
docset, schema, field, document, content :: X.Name
docset = toName "docset"
schema = toName "schema"
field = toName "field"
document = toName "document"
content = "content" -- no prefix

startEvents, endEvents :: [X.Event]
startEvents =
  [ X.EventBeginDocument
  , X.EventBeginElement docset []
  , X.EventBeginElement schema []
  , X.EventBeginElement field [("name", [X.ContentText "content"])]
  , X.EventEndElement field
  , X.EventEndElement schema
  ]

endEvents =
  [ X.EventEndElement docset
  ]

main :: IO ()
main = withSqlitePool "searcher.db3" 10 $ \pool -> do
  runStderrLoggingT $ runSqlPool (runMigration migrateAll) pool
  warpDebug 3000 $ Searcher pool
```

Приложения

Приложение А

monad-control

Пакет `monad-control`¹ используется в нескольких местах внутри `Yesod`, в первую очередь для обеспечения надлежащей обработки исключений в `Persistent`. Это универсальный пакет, расширяющий стандартную функциональность трансформаторов монад.

А.1. Обзор

Одной из мощных, но иногда сбивающих с толку, возможностей `Haskell` являются трансформаторы монад. Они позволяют брать различные части функциональности, такие как изменяемое состояние, обработка ошибок, или протоколирование, и легко объединять их вместе. Хотя я и поклялся, что никогда не буду писать учебник по монадам, я воспользуюсь слезоточивой аналогией: монады как луковицы. (Монады не как пирожные.) Под этим я имею в виду вот что — *слои*.

У нас есть центральная монада, также известная как внутренняя или основная монады. И вокруг этого центра, мы добавляем слои, каждый из которых добавляет новые возможности, действие которых распространяется наружу/вверх. В качестве объясняющего примера, давайте рассмотрим монаду `Error`, обёрнутую вокруг монады `IO`:

```
newtype ErrorT e m a = ErrorT { runErrorT :: m (Either e a) }  
type MyStack = ErrorT MyError IO
```

Обратите пристальное внимание: `ErrorT` — это простой `newtype` вокруг типа `Either`, завернутого в монаду. Избавляясь от `newtype`, мы получим:

```
type ErrorTUnwrapped e m a = m (Either e a)
```

В какой-то момент нам будет нужно выполнить некоторый ввод/вывод внутри нашего `MyStack`. Если бы мы использовали развёрнутый подход, это было бы тривиально, так как у нас на пути не стоял бы конструктор `ErrorT`. Тем не менее, нам нужна обёртка `newtype` по целому ряду причин, связанных с типами, в которые я не хочу здесь

¹<http://hackage.haskell.org/package/monad-control>

вдаваться (в конце концов, это ведь не учебник по трансформаторам монад). Таким образом, решением является класс типов `MonadTrans`:

```
class MonadTrans t where
  lift :: Monad m => m a -> t m a
```

Должен признать, что когда я увидел эту сигнатуру типа в первый раз, моей реакцией было замешательство и сомнение, что она хоть что-то значит. Но изучение экземпляра класса немного помогло:

```
instance (Error e) => MonadTrans (ErrorT e) where
  lift m = ErrorT $ do
    a <- m
    return (Right a)
```

Всё что мы здесь делаем — это оборачиваем содержимое **IO** в значение **Right**, а затем применяем нашу обёртку **newtype**. Это позволяет нам взять действие, которое живёт в **IO**, и «протянуть» (`lift`) его во внешнюю/верхнюю монаду.

А сейчас к делу. Для простых функций это работает очень хорошо. Например:

```
sayHi :: IO ()
sayHi = putStrLn "Привет"

sayHiError :: ErrorT MyError IO ()
sayHiError = lift $ putStrLn "Привет"
```

Но давайте возьмём что-то посложнее, к примеру функцию обратного вызова (`callback`):

```
withMyFile :: (Handle -> IO a) -> IO a
withMyFile = withFile "test.txt" WriteMode

sayHi :: Handle -> IO ()
sayHi handle = hPutStrLn handle "Привет!"

useMyFile :: IO ()
useMyFile = withMyFile sayHi
```

Пока всё хорошо, не так ли? Теперь предположим, что нам нужна версия `sayHi`, которая имеет доступ к монаде `Error`:

```
sayHiError :: Handle -> ErrorT MyError IO ()
sayHiError handle = do
  lift $ hPutStrLn handle "Привет, ошибка!"
  throwError MyError
```

И мы бы хотели написать функцию, которая объединяет `withMyFile` и `sayHiError`. К сожалению, GHC это не очень нравится:


```
useMyFileErrorBad :: ErrorT MyError IO ()
useMyFileErrorBad = withMyFile sayHiError
```

```
Couldn't match expected type 'ErrorT MyError IO ()'
  with actual type 'IO ()'
```

Почему это происходит, и как мы можем это обойти?

А.2. Интуитивный подход

Давайте попробуем интуитивно понять, что здесь происходит. Трансформатор монады `ErrorT` добавляет дополнительную функциональность к монаде `IO`. Мы описали способ, как «присоединять» эту дополнительную функциональность к обычным действиям ввода/вывода: мы добавляем конструктор `Right` и заворачиваем всё в `ErrorT`. Заворачивание в `Right` — наш способ сообщить, что всё прошло успешно, если не было никаких ошибок при выполнении действия.

Интуитивно это имеет смысл: поскольку монада `IO` не имеет концепции возврата `MyError`, когда что-то идёт не так, она всегда будет успешна в вызове `lift`. (Примечание: Это не имеет **ничего** общего с исключениями времени исполнения, даже не думайте о них). То что у нас есть, это гарантированное однонаправленное преобразование вверх по стеку монад.

Давайте возьмём другой пример: монада `Reader`. `Reader` имеет доступ к некоторым дополнительным данным. То, что исполняется во внутренней монаде, ничего не знает об этой дополнительной информации. Так как вы реализуете `lift`? Вы просто проигнорируете эту информацию. А монада `Writer`? Ничего не пишете. Монада `State`? Ничего не меняйте. Я вижу здесь закономерность.

А теперь давайте попробуем пойти в обратном направлении: допустим у меня есть что-то в монаде `Reader`, и я хотел бы выполнить это в основной монаде (например, `IO`). Ну... это не будет работать, не так ли? Ведь мне нужна эта дополнительная информация, я полагаюсь на неё, а она не доступна. В общем, нет способа пройти в обратном направлении, не предоставляя этого дополнительного значения.

Или есть? Если вы помните, ранее мы специально обратили внимание на то, что `ErrorT` всего лишь обёртка вокруг внутренней монады. Другими словами, если у меня есть `errorValue :: ErrorT MyError IO MyValue`, я могу применить `runErrorT` и получить значение типа `IO (Either MyError MyValue)`. Выглядит похоже на двунаправленное преобразование, не так ли?

Ну, не совсем. Изначально у нас была монада `ErrorT MyError IO`, со значением типа `MyValue`. А сейчас у нас монада `IO` со значением типа `Either MyError MyValue`. Таким образом, этот процесс на самом деле изменил значение, в то время как протягивание (lifting process) оставляет его тем же.

Но, тем не менее, немного причудливых плясок и мы можем развернуть `ErrorT`, обработать данные и обернуть его снова.

```
useMyFileError1 :: ErrorT MyError IO ()
useMyFileError1 =
```

```

let unwrapped :: Handle -> IO (Either MyError ())
    unwrapped handle = runErrorT $ sayHiError handle
    applied :: IO (Either MyError ())
    applied = withMyFile unwrapped
    rewrapped :: ErrorT MyError IO ()
    rewrapped = ErrorT applied
in rewrapped

```

Это ключевой момент всей статьи, так что смотрите внимательно. Сначала мы разворачиваем нашу монаду. Это означает, что для внешнего мира, это теперь старое доброе значение **IO**. Внутри мы сохранили всю информацию про наш трансформатор `ErrorT`. Теперь, когда у нас есть просто **IO**, мы можем легко передать его `withMyFile`. `withMyFile` принимает внутреннее состояние и возвращает его обратно без изменений. В конце концов, мы оборачиваем всё обратно в наш исходный `ErrorT`.

В этом весь принцип `monad-control`: мы встраиваем дополнительные возможности нашего трансформатора монады внутрь значения. Как только они попадают в значение, система типов игнорирует его и сосредотачивается на внутренней монаде. Когда мы закончили работать с внутренней монадой, мы можем вытащить наше состояние обратно и восстановить наш исходный стек монад.

А.3. Типы

Я нарочно начал с трансформатора `ErrorT`, так как он один из самых простых для этого механизма обращения. К сожалению, другие немного сложнее. Возьмём, к примеру, `ReaderT`. Он определяется как `newtype ReaderT r m a = ReaderT { \runReaderT :: r -> m a }`. Если мы к нему применим `runReaderT`, мы получим функцию, которая возвращает монадическое значение. Так что нам понадобятся дополнительные механизмы, чтобы с этим справиться. И здесь мы попадаем в Волшебную страну.

Есть несколько подходов к решению этих проблем. В прошлом я реализовал решение, используя семейство типов в пакете `neither`². Андерс Касеорг (Anders Kaseorg) реализовал намного более простое решение в пакете `monad-peel`³. А для эффективности, в `monad-control`⁴, Бас ван Дийк (Bas van Dijk) использует стиль передачи продолжения (Continuation Passing Style, сокращённо CPS) и экзистенциальные типы.

Код, взятый из `monad-control`, на самом деле относится к версии 0.2. В версии 0.3 некоторые вещи немного поменялись, состояние сделали явным с ассоциированным типом, а `MonadControlIO` обобщили до `MonadBaseControl`, но концепция всё ещё та же.

Первым типом, который мы рассмотрим, будет:

²<http://hackage.haskell.org/package/neither>

³<http://hackage.haskell.org/package/monad-peel>

⁴<http://hackage.haskell.org/package/monad-control>

```
type Run t = forall n o b. (Monad n, Monad o, Monad (t o)) => t n b -> n (t o b)
```

Это определение невероятно компактно, давайте его разберём. Единственным «входным» параметром типа является `t`, трансформатор монады. `Run` — это функция, которая будет работать для любой комбинации типов `n`, `o` и `b` (вот что означает `forall`). И тип `n`, и тип `o` являются монадами, а `b` — это просто значение, которое содержится в них.

Левая часть функции `Run` — `t n b` — это наш трансформатор монады, обернутый вокруг монады `n` и содержащий значение `b`. Например, это может быть `MyTrans FirstMonad MyValue`. А возвращает она значение, в котором трансформатор «засунут» внутрь, с совершенно новой центральной монадой. Другими словами, `FirstMonad (MyTrans NewMonad MyValue)`.

По началу, это может выглядеть устрашающе, но, на самом деле, всё это нам уже знакомо: по сути, то же самое мы делали с `ErrorT`. Мы начали с `ErrorT` снаружи, обернутым вокруг `IO`, а закончили с `IO`, самым содержащим `Either`. Знаете, что другой способ представить `Either` — это `ErrorT MyError Identity`? Так что, фактически, мы вытянули `IO` наружу и поместили на её место `Identity`. То же самое мы делаем в `Run`: вытягиваем монаду `FirstMonad` наружу, замещая её монадой `NewMonad`.

Сейчас хорошее время для глотка пива.

Хорошо, это уже кое-что. Если бы у нас был доступ к одной из этих функций `Run`, мы могли бы использовать её для снятия `ErrorT` с результата нашей функции `sayHiError`, и передачи его в `withMyFile`. С помощью магии `undefined` мы можем сыграть в эту игру:

```
errorRun :: Run (ErrorT MyError)
errorRun = undefined

useMyFileError2 :: IO (ErrorT MyError Identity ())
useMyFileError2 =
  let afterRun :: Handle -> IO (ErrorT MyError Identity ())
      afterRun handle = errorRun $ sayHiError handle
      applied :: IO (ErrorT MyError Identity ())
      applied = withMyFile afterRun
  in applied
```

Это выглядит очень похоже на предыдущий пример. Фактически, `errorRun` ведёт себя почти так же, как и `runErrorT`. Однако, у нас всё ещё есть две проблемы: мы не знаем, где взять значение `errorRun`, и нам всё ещё надо восстанавливать исходный `ErrorT` после выполнения действия.

A.3.1. MonadTransControl

Очевидно, что в данном конкретном случае мы могли бы использовать наши знания про трансформатор `ErrorT`, чтобы положить систему типов на лопатки и написать

функцию `Run` вручную. Но что мы хотим *на самом деле* — это общее решение для многих трансформаторов. В таком случае, как вы знаете, нам нужен класс типов.

Итак, давайте повторим, что нам нужно: доступ к функции `Run` и некий способ восстановления нашего исходного трансформатора. Так появился `MonadTransControl` с единственным методом `liftControl`:

```
class MonadTrans t => MonadTransControl t where
  liftControl :: Monad m => (Run t -> m a) -> t m a
```

Давайте посмотрим внимательнее. Метод `liftControl` принимает функцию (которую нам следует написать). Эта функция получает на вход функцию `Run` и должна возвращать значение в некоторой монаде (`m`). Затем `liftControl` возьмёт результат этой функции и восстановит исходный трансформатор поверх него.

```
useMyFileError3 :: Monad m => ErrorT MyError IO (ErrorT MyError m ())
useMyFileError3 =
  liftControl inside
where
  inside :: Monad m => Run (ErrorT MyError) -> IO (ErrorT MyError m ())
  inside run = withMyFile $ helper run
  helper :: Monad m
             => Run (ErrorT MyError) -> Handle -> IO (ErrorT MyError m ())
  helper run handle = run (sayHiError handle :: ErrorT MyError IO ())
```

Близко, но не совсем то, что я имел в виду. Почему двойные монады? Хорошо, давайте начнём с конца: `sayHiError handle` возвращает значение типа `ErrorT m MyError IO ()`. Это мы уже знаем, никаких сюрпризов. Что может быть немного удивительно (по крайней мере, было для меня) — это следующие два шага.

Сначала мы применяем `run` к этому значению. Как мы уже обсуждали ранее, в результате этого внутренняя монада `IO` извлекается наружу для замены некоторой произвольной монадой (представленной здесь как `m`). Так мы в конечном итоге приходим к `IO (ErrorT MyError m ())`. Хорошо... Собственно, мы получаем тот же результат после применения `withMyFile`. Ничего удивительного.

У меня заняло много времени, чтобы правильно понять последний шаг. Помните, как мы говорили про восстановление исходного трансформатора? Ну, так мы это и делаем: плюхаем его прямо на всё остальное. Таким образом, наш конечный результат будет: предыдущий тип — `IO (ErrorT MyError m ())` — с приклеенным в начале `ErrorT MyError`.

Всё это кажется чрезвычайно бесполезным, не так ли? Ну, почти. Не стоит забывать, что «`m`» может быть любой монадой, в том числе `IO`. Если рассматривать это таким образом, то мы получаем `ErrorT MyError IO (ErrorT MyError IO ())`. Выглядит очень похоже на `m (m a)`, а мы хотим просто `m a`. К счастью, сейчас нам повезло:

```
useMyFileError4 :: ErrorT MyError IO ()
```

```
useMyFileError4 = join useMyFileError3
```

И оказывается, что такое использование настолько распространено, что Бас сжался над нами и определил вспомогательную функцию:

```
control :: (Monad m, Monad (t m), MonadTransControl t)
         => (Run t -> m (t m a)) -> t m a
control = join . liftControl
```

Так что всё, что нам надо написать, это:

```
useMyFileError5 :: ErrorT MyError IO ()
useMyFileError5 =
  control inside
  where
    inside :: Monad m => Run (ErrorT MyError) -> IO (ErrorT MyError m ())
    inside run = withMyFile $ helper run
    helper :: Monad m
              => Run (ErrorT MyError) -> Handle -> IO (ErrorT MyError m ())
    helper run handle = run (sayHiError handle :: ErrorT MyError IO ())
```

Или же немного короче:

```
useMyFileError6 :: ErrorT MyError IO ()
useMyFileError6 = control $ \run -> withMyFile $ run . sayHiError
```

А.3.2. MonadControlIO

Класс `MonadTrans` предоставляет метод `lift`, который позволяет протягивать действие на один уровень по стеку монад. Также существует класс `MonadIO`, предоставляющий метод `liftIO`, который протягивает действие `IO` так далеко по стеку, как хочется. У нас такое же разделение в `monad-control`. Но сперва, нам нужно функцию подобную `Run`:

```
type RunInBase m base = forall b. m b -> base (m b)
```

Вместо того что бы иметь дело с трансформатором, мы работаем с двумя монадами. Монада `base` — это основная монада, а `m` — это стек, построенный на ней. `RunInBase` — это функция, которая принимает значение всего стека, вытягивает `base` и размещает её снаружи. В отличие от типа `Run`, мы не замещали её произвольной монадой, а использовали исходную. Пример с конкретными типами:

```
RunInBase (ErrorT MyError IO) IO = forall b. ErrorT MyError IO b -> IO
  IO (ErrorT MyError IO b)
```

Это очень похоже на то, что мы рассматривали до сих пор, с той лишь разницей, что тут мы хотим иметь дело с конкретной внутренней монадой. На самом деле, класс `MonadControlIO` — это просто расширение `MonadControlTrans` с помощью `RunInBase`.

```
class MonadIO m => MonadControlIO m where
  liftControlIO :: (RunInBase m IO -> IO a) -> m a
```

Проще говоря, `liftControlIO` принимает функцию, которая ожидает на вход функцию `RunInBase`. Эта `RunInBase` может быть использована для разбора нашей монады до простой `IO`, а затем `liftControlIO` выстраивает всё назад. Так же, как и `MonadControlTrans`, он поставляется со вспомогательной функцией

```
controlIO :: MonadControlIO m => (RunInBase m IO -> IO (m a)) -> m a
controlIO = join . liftControlIO
```

Используя её, мы легко можем переписать наш предыдущий пример:

```
useMyFileError7 :: ErrorT MyError IO ()
useMyFileError7 = controlIO $ \run -> withMyFile $ run . sayHiError
```

И, как преимущество, она легко масштабируется на несколько трансформаторов:

```
sayHiCrazy :: Handle -> ReaderT Int (StateT Double (ErrorT MyError IO)) ()
sayHiCrazy handle = liftIO $ hPutStrLn handle "Madness!"

useMyFileCrazy :: ReaderT Int (StateT Double (ErrorT MyError IO)) ()
useMyFileCrazy = controlIO $ \run -> withMyFile $ run . sayHiCrazy
```

А.4. Примеры из реальной жизни

Давайте, используя этот код, решим несколько задач из реальной жизни. Наверное, самый интересный вариант использования — обработка исключений в стеке трансформатора. Предположим, для примера, что мы хотим автоматически исполнять некоторый код очистки при возникновении исключения. Если бы это был обычный код ввода/вывода, мы бы использовали:

```
onException :: IO a -> IO b -> IO a
```

Но, находясь в монаде `ErrorT`, мы не можем передать ни действие, ни код очистки. На помощь приходит `controlIO`:

```
onExceptionError :: ErrorT MyError IO a
                  -> ErrorT MyError IO b
                  -> ErrorT MyError IO a
```

```
onExceptionError action after = controlIO $ \run ->
  run action 'onException' run after
```

Теперь допустим, что нам надо выделить некоторый объём памяти для хранения значения типа **Double**. В монаде **IO** мы просто использовали бы функцию `alloca`. И опять наше решение очень простое:

```
allocaError :: (Ptr Double -> ErrorT MyError IO b)
             -> ErrorT MyError IO b
allocaError f = controlIO $ \run -> alloca $ run . f
```

A.5. Потерянное состояние

Давайте, отмотаем немного назад — к нашему определению функции `onExceptionError`. Она использует `onException`, которая имеет сигнатуру типа: **IO** a -> **IO** b -> **IO** a. Позвольте мне спросить вас: что случилось с аргументом b на выходе? Он был полностью проигнорирован. Но это, кажется, создаёт нам небольшую проблему. Ведь, в конце концов, мы сохраняем информацию о состоянии нашего трансформатора в значении внутренней монады. И если мы его игнорируем, то мы, по существу, также игнорируем и монадические побочные эффекты!

Да, так и есть. Такое случается при использовании `monad-control`. Определённые функции теряют некоторые из монадических побочных эффектов. Лучше всего это описал Бас в комментариях к соответствующим функциям:

Обратите внимание, что любые монадические побочные эффекты в m кода "очистки" будут отброшены; он выполняется только ради его побочных эффектов в **IO**.

На практике, `monad-control` обычно будет делать то, что вам надо, но имейте в виду, что некоторые побочные эффекты могут исчезнуть.

A.6. Случаи посложнее

До сих пор, чтобы заставить наши уловки работать, нам требовались функции, дающие полный доступ к своим значениями. Иногда это не представляется возможным. Возьмём, к примеру:

```
addMVarFinalizer :: MVar a -> IO () -> IO ()
```

В этом случае требуется, чтобы функция завершения не имела никакого значения. Интуитивно, первое, что нам следует отметить, — нет никакого способа захватить наши монадические побочные эффекты. Так как же мы сможем что-то вроде такого скомпилировать? Очень просто, нам необходимо явно описать отказ от всей сохраняющей состояние информации:

```

addMVarFinalizerError :: MVar a -> ErrorT MyError IO () -> ErrorT []
    MyError IO ()
addMVarFinalizerError mvar f = controlIO $ \run ->
    return $ liftIO $ addMVarFinalizer mvar (run f >> return ())

```

Другой случай из того же модуля:

```

modifyMVar :: MVar a -> (a -> IO (a, b)) -> IO b

```

Теперь у нас есть ограничения на тип возвращаемого значения во втором аргументе: это должен быть кортеж из значения, переданного в функцию, и итогового возвращаемого значения. К сожалению, я не вижу способа написать небольшую обёртку вокруг `modifyMVar`, чтобы заставить её работать с `ErrorT`. Вместо этого, в данном случае, я скопировал определение `modifyMVar` и изменил его:

```

modifyMVar :: MVar a
    -> (a -> ErrorT MyError IO (a, b))
    -> ErrorT MyError IO b
modifyMVar m io =
    Control.Exception.Control.mask $ \restore -> do
        a <- liftIO $ takeMVar m
        (a',b) <- restore (io a) 'onExceptionError' liftIO (putMVar m a)
        liftIO $ putMVar m a'
        return b

```


Приложение В

Кондуиты

Кондуиты используются для обработки потоков данных. Обычно ленивые вычисления позволяют обрабатывать большие объемы данных, не загружая их в память целиком. Однако использование такого подхода для ввода-вывода влечёт требование ленивости последнего. Основной недостаток ленивого ввода-вывода — его недетерминированность: у нас нет никаких гарантий того, когда будут выполнены финализаторы наших ресурсов. Для небольшого приложения это допустимо, но для высоко нагруженного веб-сервера мы можем очень быстро исчерпать ограниченные системные ресурсы, например, дескрипторы файлов.

Кондуиты позволяют оперировать большими потоками данных при детерминированном управлении ресурсами. Они предоставляют унифицированный интерфейс для потоков данных вне зависимости от того, откуда они поступают: из файлов, сокетов или памяти. В сочетании с `ResourceT` мы можем безопасно работать с ресурсами, зная, что они будут гарантированно освобождены даже в случае возникновения исключений.

В этом приложении рассматривается пакет `conduit`¹ версии 0.2.

В.1. Кондуиты в двух словах

Хотя понимание низкоуровневой механики кондуитов рекомендуется, вы можете далеко продвинуться и без неё. Давайте начнём с нескольких высокоуровневых примеров. Не беспокойтесь, если некоторые детали будут вам непонятны сразу, мы разберём всё в этом приложении. Начнём с терминологии, после чего перейдём к примерам кода.

- *Источник* (*source*) генерирует данные. Они могут быть прочитаны из файла, прийти из сокета или находиться в памяти в виде списка. Мы будем обращаться к этим данным, забирая их из источника.
- *Стоки* (*sinks*) потребляют данные. Простейшими примерами могут служить: функция суммирования (сложение чисел из потока), файловый сток (который записывает все полученные байты в файл) или сокет. По окончании обработки данных (мы объясним это чуть позже) сток возвращает некоторое значение.

¹<http://hackage.haskell.org/package/conduit>

- *Кондуиты* преобразуют данные. Простейший пример — функция `map`, хотя существует много других. Мы добавляем данные в конduit так же как и в сток, однако вместо возврата одного значения конduit может вернуть несколько результатов каждый раз как в него добавляются данные.
- *Комбинирование (fuse)* (Спасибо Давиду Мазьересу за термин.) Конduit можно скомбинировать с источником данных (с помощью оператора `=>`) и получить новый источник. Например, мы можем взять источник, читающий байты из файла, и конduit, преобразующий байты в текст. Скомбинировав их мы получим источник, читающий текст из файла. Аналогично, конduit и сток можно скомбинировать в новый сток (оператор `=>`), а два кондуита — в новый конduit (оператор `=>`).
- *Соединение*. Мы можем присоединять источник к стоку, используя оператор `$$`. Это приведет к тому, что данные будут передаваться из источника в сток до тех пор, пока один из них не сообщит, что он «закончил».

Рассмотрим несколько примеров кода.

```
{-# LANGUAGE OverloadedStrings #-}
import Data.Conduit -- основная библиотека
import qualified Data.Conduit.List as CL -- несколько функций для []
  работы со списками
import qualified Data.Conduit.Binary as CB -- байты
import qualified Data.Conduit.Text as CT

import Data.ByteString (ByteString)
import Data.Text (Text)
import qualified Data.Text as T
import Control.Monad.ST (runST)
```

Для начала соединим источник со стоком. Будем использовать встроенные функции по работе с файлами для эффективного, использующего постоянный объём памяти ресурсо-безопасного копирования файлов.

Обратите внимание: вначале мы используем `$$` для соединения источника и стока, а затем используем `runResourceT`.

```
copyFile :: FilePath -> FilePath -> IO ()
copyFile src dest = runResourceT $ CB.sourceFile src $$ CB.sinkFile []
  dest
```

Модуль `Data.Conduit.List` предоставляет несколько вспомогательных функций для создания источников, стоков и кондуитов. Вот так выглядит типичная свёртка: суммирование чисел.

```
sumSink :: Resource m => Sink Int m Int
sumSink = CL.fold (+) 0
```

Мы можем реализовать то же самое на более низком уровне, используя функцию `sinkState`. Она принимает три параметра: начальное состояние, функцию приёма дополнительных данных и функцию закрытия.

```
sumSink2 :: Resource m => Sink Int m Int
sumSink2 = sinkState
  0 -- начальное значение
  -- обновим состояния согласно полученным данным
  -- и сообщим, что необходимы дополнительные данные
  (\accum i -> return $ StateProcessing (accum + i))
  (\accum -> return accum) -- вернуть текущее значение при закрытии
```

Другая полезная функция — `sourceList`. Скомбинировав её с нашей функцией `sumSink`, мы получим встроенную реализацию функции `sum`.

```
sum' :: [Int] -> Int
sum' input = runST $ runResourceT $ CL.sourceList input $$ sumSink
```

Поскольку это Haskell, давайте создадим источник, который генерирует все числа Фибоначчи. Для этого воспользуемся `sourceState`. Состояние будет содержать следующие два числа в последовательности. Также нам понадобится функция, которая вернёт следующее число и обновит состояние.

```
fibs :: Resource m => Source m Int
fibs = sourceState
  (0, 1) -- начальное состояние
  (\(x, y) -> return $ StateOpen (y, x + y) x)
```

Посчитаем сумму первых 10 чисел Фибоначчи. Мы можем использовать кондуит `isolate`, чтобы быть уверенными, что сток суммирования обработает только 10 значений.

```
sumTenFibs :: Int
sumTenFibs =
  runST -- прекрасно работает в чистом коде
  $ runResourceT
  $ fibs
  $= CL.isolate 10 -- комбинирует источник и кондуит в источник
  $$ sumSink
```

Мы также можем скомбинировать кондуит и сток, поменяв местами некоторые операторы.

```
sumTenFibs2 :: Int
sumTenFibs2 =
  runST
  $ runResourceT
```

```

$ fibs
$$ CL.isolate 10
=$ sumSink

```

Отлично, а теперь сделаем несколько кондуитов. Давайте преобразовывать числа в текст. Звучит, как задача для функции `map`...

```

intToText :: Int -> Text -- вспомогательная функция
intToText = T.pack . show

textify :: Resource m => Conduit Int m Text
textify = CL.map intToText

```

Как и раньше, воспользуемся функцией `conduitState`. Здесь нам не нужно состояние, поэтому подставим фиктивное значение.

```

textify2 :: Resource m => Conduit Int m Text
textify2 = conduitState
    ()
    (\() input -> return $ StateProducing () [intToText input])
    (\() -> return [])

```

Сделаем кондуит `unlines`, который будет добавлять перевод строки в конце каждого блока входных данных. Воспользуемся функцией `CL.map`. Для тренировки можете попробовать реализовать то же самое при помощи `conduitState`.

```

unlines' :: Resource m => Conduit Text m Text
unlines' = CL.map $ \t -> t 'T.append' "\n"

```

А теперь напишем функцию, которая печатает первые `N` чисел Фибоначчи. Используем кодировку UTF8.

```

writeFibs :: Int -> FilePath -> IO ()
writeFibs count dest =
    runResourceT
        $ fibs
        $= CL.isolate count
        $= textify
        $= unlines'
        $= CT.encode CT.utf8
        $$ CB.sinkFile dest

```

Мы использовали оператор `$=`, чтобы комбинировать кондуиты с источниками, получая новые источники. Можно делать и обратное: комбинировать кондуиты и стоки. Можно даже скомбинировать два кондуита.

```

writeFibs2 :: Int -> FilePath -> IO ()

```

```
writeFibs2 count dest =
    runResourceT
    $ fibs
    $= CL.isolate count
    $= textify
    $$ unlines'
    =$ CT.encode CT.utf8
    =$ CB.sinkFile dest
```

Или мы можем скомбинировать все написанные кондуиты в один...

```
someIntLines :: ResourceThrow m -- изменение кодировки может []
              -- выбросить исключение
              => Int
              -> Conduit Int m ByteString
someIntLines count =
    CL.isolate count
    =$= textify
    =$= unlines'
    =$= CT.encode CT.utf8
```

.. и затем использовать этот кондуит:

```
writeFibs3 :: Int -> FilePath -> IO ()
writeFibs3 count dest =
    runResourceT
    $ fibs
    $= someIntLines count
    $$ CB.sinkFile dest

main :: IO ()
main = do
    putStrLn $ "First ten fibs: " ++ show sumTenFibs
    writeFibs 20 "fibs.txt"
    copyFile "fibs.txt" "fibs2.txt"
```

В.2. Содержание этой главы

Далее в этой главе освещаются следующие темы:

- ResourceT — техника, которая позволяет гарантировать освобождение ресурсов;
- Источники — наши генераторы данных;
- Стоки — потребители данных;
- Кондуиты — преобразователи данных;

- Буферизация — техника борьбы с инверсией управления (Inversion of Control).

В.3. Трансформер монады Resource

Трансформер монады Resource (ResourceT) играет существенную роль в управлении ресурсами в проектах, использующих кондуиты. Он поставляется вместе с библиотекой conduit. Мы будем рассматривать ResourceT как отдельную сущность. Несмотря на то, что некоторые решения в его дизайне ориентированы на использование с кондуитами, ResourceT и сам по себе остаётся полезным инструментом.

В.3.1. Назначение

Что не так с этим кодом?

```
import System.IO

main = do
  output <- openFile "output.txt" WriteMode
  input  <- openFile "input.txt"  ReadMode
  hGetContents input <<= hPutStr output
  hClose input
  hClose output
```

Если файл input.txt отсутствует, бросится исключение при попытке его открыть. В результате hClose output никогда не будет вызвано и мы получим утечку важного ресурса — дескриптора файла. В небольшой программе это не критично, но очевидно, что мы не можем себе этого позволить в долго работающем, высоко нагруженном серверном процессе.

К счастью, проблема решается довольно просто:

```
import System.IO

main =
  withFile "output.txt" WriteMode $ \output ->
  withFile "input.txt"  ReadMode  $ \input  ->
  hGetContents input <<= hPutStr output
```

Использование withFile гарантирует, что дескриптор будет закрыт даже в случае исключений. Оно также поддерживает асинхронные исключения. Вообще говоря, это прекрасный подход для случаев, когда его использование допустимо. Зачастую использование withFile довольно просто, но иногда может повлечь переписывание всей программы, а это может быть очень скучно или сильно неэффективно.

Возьмём, например, еnumераторы (пакет enumerator). Если вы заглянете в документацию, то найдёте функцию enumFile для чтения содержимого файла, но не найдёте функции iterFile для записи содержимого в файл. Так сделано потому, что поток

управления итераторов не позволяет правильно управлять дескрипторами. Поэтому, чтобы записать в файл вам надо создавать дескриптор до запуска `Iteratee`, например:

```
import System.IO
import Data.Enumerator
import Data.Enumerator.Binary

main =
  withFile "output.txt" WriteMode $ \output ->
    run_ $ enumFile "input.txt" $$ iterHandle output
```

Этот код работает хорошо, но представьте, что вместо простой передачи данных в файл, нам надо провести длительное вычисление перед использованием дескриптора. Мы захватим дескриптор задолго до того, как он нам понадобится, занимая важный ресурс нашего приложения. Кроме того, часто мы не можем открыть файл заранее, так как мы поймем какой файл открывать только после прочтения данных из входного файла.

Одна из заявленных целей кондуитов — решить эту проблему, и они делают это с помощью `ResourceT`. Программа выше может быть переписана следующим образом:

```
{-# LANGUAGE OverloadedStrings #-}
import Data.Conduit
import Data.Conduit.Binary

main = runResourceT $ sourceFile "input.txt" $$ sinkFile "output.txt"
```

В.3.2. Как это работает

Всего основных функций по работе с `ResourceT` — три, остальные сделаны поверх этих трёх исключительно для удобства. Вот первая из этой тройцы:

```
register :: IO () -> ResourceT IO ReleaseKey
```

На самом деле, эта и другие функции ниже более полиморфны, чем показано здесь. Они могут работать с другими монадами кроме `IO`. В действительности, почти любой трансформер над `IO`, также как над любым `ST` стэком, будет работать. Мы разьясим детали позднее.

Эта функция регистрирует кусок кода, который будет выполнен. Она возвращает значение `ReleaseKey`, которое используется в следующей функции:

```
release :: ReleaseKey -> ResourceT IO ()
```

Вызов функции `release` от `ReleaseKey` немедленно выполняет действие, которое было зарегистрировано ранее. Мы можем вызывать `release` с одним и тем же значением

`ReleaseKey` столько раз, сколько пожелаем. При первом вызове производится регистрация действия. Это значит, что вы можете безопасно зарегистрировать действие освобождения памяти, не заботясь о том, что оно будет выполнено дважды.

В один прекрасный момент, нам захочется покинуть `ResourceT`. Чтобы сделать это, воспользуемся:

```
runResourceT :: ResourceT IO a -> IO a
```

В этой внешне невинной функции происходит вся магия. Она проходит по всем зарегистрированным действиям освобождения ресурсов и выполняет их. Эта функция безопасна с точки зрения исключений, в том смысле, что освобождение ресурсов будет выполнено в случае и синхронных и асинхронных исключений. И, как было упомянуто выше, вызов функции `release` отменит регистрацию действия. Таким образом нам не стоит беспокоиться о повторном освобождении ресурсов.

Наконец, для удобства, мы приведем ещё одну функцию для выделения ресурса и регистрации действия по его освобождению:

```
with :: IO a -- выделить
      -> (a -> IO ()) -- освободить
      -> ResourceT IO (ReleaseKey, a)
```

Теперь перепишем первый некачественный пример с использованием `ResourceT`:

```
import System.IO
import Control.Monad.Trans.Resource
import Control.Monad.Trans.Class (lift)

main = runResourceT $ do
  (release0, output) <- with (openFile "output.txt" WriteMode) hClose
  (releaseI, input) <- with (openFile "input.txt" ReadMode) hClose
  lift $ hGetContents input <=< hPutStr output
  release releaseI
  release release0
```

Теперь мы можем не беспокоиться об исключениях, предотвращающих освобождение ресурсов. В таких маленьких программах как эта можно опустить вызовы `release`, это ни на что не повлияет. Однако в больших приложениях, где мы продолжим обработку дальше, этот код гарантирует, что дескрипторы ресурсов освободятся как только это станет возможным, снижая потребление ресурсов до минимума.

В.3.3. Несколько слов о типах

Как было упомянуто выше, `ResourceT` это нечто большее, чем код, исполняемый в `IO`. Давайте обсудим некоторые требования, предъявляемые к используемой монаде.

- Изменяемые ссылки для хранения зарегистрированных действий по освобождению ресурсов. Вам может казаться, что мы могли бы использовать трансформер StateT, но тогда наше состояние не сможет корректно пережить исключительные ситуации.
- Мы хотим регистрировать действия только в базовой монаде. Например, если мы имеем стек ResourceT (WriterT [Int] IO), желательно зарегистрировать только IO действия. Это позволит легко втягивать (lift) наш стек (т.е. добавлять новый трансформер в середину стека) и избежать сбивающих с толку проблем с “нанизыванием” сайд-эффектов других монад.
- Какой-то способ гарантировать, что действие будет выполнено даже в случае исключений. Отсюда следует необходимость функции a-ля bracket.

Для первого пункта определим класс типов для монад, которые имеют изменяемые ссылки:

```
class Monad m => HasRef m where
  type Ref m :: * -> *
  newRef' :: a -> m (Ref m a)
  readRef' :: Ref m a -> m a
  writeRef' :: Ref m a -> a -> m ()
  modifyRef' :: Ref m a -> (a -> (a, b)) -> m b
  mask :: ((forall a. m a -> m a) -> m b) -> m b
  mask_ :: m a -> m a
  try :: m a -> m (Either SomeException a)
```

Здесь у нас ассоциированный тип для указания типа ссылок. (Фанатам функциональных зависимостей я покажу в следующем разделе, что ассоциированный тип необходим.) Затем мы определяем несколько основных операций над ссылками. В конце несколько функций для работы с исключениями, которые необходимы для безопасной реализации функций, описанных в последнем разделе. Реализация экземпляра для IO довольно прямолинейна:

```
instance HasRef IO where
  type Ref IO = I.IORef
  newRef' = I.newIORef
  modifyRef' = I.atomicModifyIORef
  readRef' = I.readIORef
  writeRef' = I.writeIORef
  mask = E.mask
  mask_ = E.mask_
  try = E.try
```

Однако, при реализации экземпляра для монады ST мы сталкиваемся с проблемой: обработка исключений в монаде ST невозможна. В результате, функции mask, mask_ и try имеют реализацию по умолчанию без проверки исключений. Из-за этого мы должны сформулировать первое предупреждение:

Операции в монаде ST не безопасны относительно исключений. Вы не должны выделять редкие ресурсы в монаде ST когда используете ResourceT.

Вы, возможно, спрашиваете себя, зачем тогда вообще использовать ResourceT в комбинации с ST. Причина в том, что с кондуитами можно делать много чего и без использования редких ресурсов, и ST является прекрасным способом делать это чисто. Однако об этом позднее.

Пункт номер 2: нам надо как-то работать с концепцией базовой монады. Опять же, мы можем использовать ассоциированные типы (и снова это будет объяснено в следующем разделе). Наше решение будет выглядеть примерно так:

```
class (HasRef (Base m), Monad m) => Resource m where
  type Base m :: * -> *

  resourceLiftBase :: Base m a -> m a
```

Мы забыли о пункте 3: bracket-подобной функции. Нам понадобится ещё один метод в классе типов:

```
resourceBracket_ :: Base m a -> Base m b -> m c -> m c
```

Причиной, по которой первые два аргумента функции resourceBracket_ «живут» в базовой монаде, является то, что в ResourceT все выделения и освобождения ресурсов, происходят в базовой монаде.

Итак, над нашим экземпляром HasRef для IO нам также нужен экземпляр Resource. Реализация довольно очевидна:

```
instance Resource IO where
  type Base IO = IO
  resourceLiftBase = id
  resourceBracket_ = E.bracket_
```

Теперь у нас есть несколько сходных экземпляров ST, в которых функция resourceBracket_ небезопасна относительно исключений. Решающий шаг заключается в поддержке трансформеров. Нам не нужен экземпляр HasRef, но нужен экземпляр Resource. Сложность состоит в корректной реализации функции resourceBracket_, для чего воспользуемся некоторыми функциями из пакета monad-control (monad-control²):

```
instance (MonadTransControl t, Resource m, Monad (t m))
  => Resource (t m) where
  type Base (t m) = Base m

  resourceLiftBase = lift . resourceLiftBase
  resourceBracket_ a b c =
    control' $ \run -> resourceBracket_ a b (run c)
```

²<http://hackage.haskell.org/package/monad-control>

```

where
  control' f = liftWith f >>= restoreT . return

```

Для всякого трансформера, его база является базой внутренней монады. Аналогично, мы втягиваем значение в базовую монаду посредством втягивания сначала во внутреннюю монаду, а затем в базовую. Сложная часть — реализация `resourceBracket_`. Я не буду вдаваться в подробные объяснения, чтобы случайно не поставить себя в глупое положение.

В.3.4. Определение ResourceT

Теперь у нас достаточно информации, чтобы осмыслить определение типа `ResourceT`:

```

newtype ReleaseKey = ReleaseKey Int

type RefCount = Int
type NextKey = Int

data ReleaseMap base =
  ReleaseMap !NextKey !RefCount !(IntMap (base ()))

newtype ResourceT m a =
  ResourceT (Ref (Base m) (ReleaseMap (Base m)) -> m a)

```

Мы видим, что `ReleaseKey` это просто `Int`. Если вы заглянете на несколько строк вниз, то это определение обретёт смысл, так как мы используем `IntMap` для хранения зарегистрированных действий. Мы также определяем два типа-синонима: `RefCount` и `NextKey`. `NextKey` хранит последнее присвоенное значение ключа и увеличивается на единицу каждый раз, когда вызывается функция `register`. `RefCount` мы коснёмся немного позже.

`ReleaseMap` хранит три вида информации: следующий ключ, счётчик ссылок и `map` всех зарегистрированных действий (`actions`). Заметьте, что `ReleaseMap` принимает типовую переменную `base`, которая определяет, в какой монаде должны жить действия по освобождению ресурсов.

Наконец, `ResourceT` является по сути `ReaderT`, который хранит изменяемую ссылку на `ReleaseMap`. Тип ссылки определяется типом базовой монады точно так же как и «очищающая» монада. Именно поэтому мы и использовали ассоциированные типы.

Большая часть оставшегося кода в модуле `Control.Monad.Trans.Resource` — определение экземпляров для типа `ResourceT`.

В.3.5. Другие классы типов

Модуль предоставляет ещё три класса типов:

ResourceUnsafeIO

Любая монада, которая может втянуть (lift) IO действия в себя, но это может быть небезопасно. Первый пример этого — ST. Осторожность необходима только при втягивании действий, которые не используют ресурсы и которые не «запускают ракеты». Другими словами, применимы все обычные предупреждения насчёт unsafeIOtoST.

ResourceThrow

Для действий, которые могут бросать исключения. Автоматически применимо ко всем монадам на базе IO. Для ST-монад можно воспользоваться трансформером ExceptionT, чтобы предоставить возможность пробрасывания исключений. Это потребуется для некоторых функций кондуитов, например, для декодирования текста.

ResourceIO

Включает в себя несколько классов типов, в том числе два упомянутых выше. Он создан исключительно для удобства, мы можем добиться того же результата и без него, разве что потребуется написать больше кода.

В.3.6. Ветвления (Forking)

Может показаться, что ответвление потока по определению небезопасно при использовании ResourceT, так как родительский поток может вызвать runResourceT в то время как дочерний использует ресурсы. Это, конечно же, так, если вы используете обычную функцию forkIO.

Вообще, вы не можете использовать стандартный forkIO, так как он использует монаду IO, однако вполне можно воспользоваться функцией fork из пакета lifted-base. Именно по этой причине пакет regions не предоставляет экземпляр MonadBaseControl для своего трансформера (который очень похож на ResourceT). Однако, назначение ResourceT заключается не в запрещении программистам стрелять себе в ногу, а только в облегчении написания правильного кода. Поэтому, мы всё же предоставляем этот экземпляр, даже несмотря на то, что он может быть использован неправильно.

Чтобы решить эту проблему ResourceT включает в себя счётчик ссылок. Когда вы создаёте новый поток с помощью resourceForkIO, значение RefCount в ReleaseMap увеличивается на единицу. Всякий раз, когда вызывается runResourceT, это значение уменьшается на 1. Только когда оно достигнет нуля, выполняются действия по освобождению ресурсов.

В.3.7. Вспомогательные функции

В дополнение к тому, о чём уже было упомянуто, мы расскажем про несколько функций, созданных (в основном) для удобства.

Функции newRef, writeRef и readRef оборачивают функции с HasRef и позволяют им работать с любым ResourceT. Функция withIO по сути является ограниченной по типу функцией with, избегающей некоторых сложностей с типами, с которыми

иначе пришлось бы иметь дело. В общем случае, вам стоит использовать `withIO` для написания **IO** кода.

Функция `transResourceT` позволит вам модифицировать в какой монаде исполняется ваш `ResourceT`, учитывая, что база остаётся прежней.

```
transResourceT :: (Base m ~ Base n)
               => (m a -> n a)
               -> ResourceT m a
               -> ResourceT n a
transResourceT f (ResourceT mx) = ResourceT (\r -> f (mx r))
```

В.4. Источники

Мне кажется, проще всего понять что это такое, посмотрев на типы:

```
data SourceResult m a = Open (Source m a) a | Closed
data Source m a = Source
  { sourcePull :: ResourceT m (SourceResult m a)
  , sourceClose :: ResourceT m ()
  }
```

Источник поддерживает две операции: вы можете запросить ещё данных и вы можете закрыть его (как, например, закрыть дескриптор файла). При запросе новых данных вы или получаете немного данных и новое значение типа `Source` (источник остается открытым), или ничего (источник закрывается). Давайте посмотрим на простейшие примеры.

```
import Prelude hiding (repeat)
import Data.Conduit

-- | Никогда не выдает данные
eof :: Monad m => Source m a
eof = Source
  { sourcePull = return Closed
  , sourceClose = return ()
  }

-- | Всегда выдает одно и то же
repeat :: Monad m => a -> Source m a
repeat a = Source
  { sourcePull = return $ Open (repeat a) a
  , sourceClose = return ()
  }
```

Эти источники довольно тривиальны, так как они всегда возвращают одно и то же. К тому же их функции закрытия ничего не делают. Вам может показаться, что это ошибка:

не должны ли мы в функции `sourcePull` возвращать `Closed` после того, как источник был закрыт? В этом нет необходимости, так как согласно одному из правил источников они никогда не могут быть переиспользованы. Другими словами:

Если источник вернул `Open`, то он предоставил вам новый источник, с которым и стоит работать вместо оригинального. Если источник вернул `Closed`, тогда вы больше не можете выполнять операции над ним.

Не стоит очень беспокоиться по поводу сохранения этого инварианта. На практике, вам почти не придётся вызывать `sourcePull` или `sourceClose` самостоятельно. Более того, вряд ли вам придётся их самостоятельно реализовывать (для этого существуют `sourceState` и `sourceIO`). Идея заключается в том, что мы можем сделать некоторые предположения, когда реализовываем источники.

В.4.1. Состояние

В двух примерах источников выше есть кое-что общее: они никогда не меняются и всегда возвращают одно и то же значение. Другими словами, у них нет состояния. Для более-менее серьёзных источников нам потребуется какое-то состояние.

Состояние может запросто быть определено вне нашей программы. Например, если мы реализуем источник, который читает данные из дескриптора (**Handle**), нам не нужно вручную указывать никакого состояния, потому что дескриптор имеет состояние сам по себе.

Будем хранить состояние в источнике путём обновления возвращаемого значения типа `Source` в конструкторе `Open`. Лучше всего это видно на примере.

```
import Data.Conduit
import Control.Monad.Trans.Resource

-- Выдает данные из списка по одному
sourceList :: Resource m => [a] -> Source m a
sourceList list = Source
  { sourcePull =
    case list of
      [] -> return Closed -- больше нет данных
      -- Здесь будем хранить состояние, возвращая новый
      -- источник с остатком списка
      x:xs -> return $ Open (sourceList xs) x
    , sourceClose = return ()
  }
```

Всякий раз, когда мы забираем данные из источника, он проверяет список. Если тот пуст, возвращаем `Closed`, что разумно. Если не пуст, возвращаем `Open` со следующим значением из списка и новым значением источника, содержащим хвост списка.

В.4.2. Функции `sourceState` и `sourceIO`

Кроме возможности создавать источники, у нас также имеется несколько вспомогательных функций, позволяющих создавать источники на более высоком уровне. Функция `sourceState` позволяет писать код как будто вы используете монаду `State`. Вы предоставляете начальное состояние и функцию получения данных от текущего состояния, возвращающую новое состояние и новое значение. Перепишем `sourceList` с её помощью:

```
import Data.Conduit
import Control.Monad.Trans.Resource

-- Выдаёт данные из списка по одному
sourceList :: Resource m => [a] -> Source m a
sourceList state0 = sourceState
    state0
    pull
  where
    pull [] = return StateClosed
    pull (x:xs) = return $ StateOpen xs x
```

Обратите внимание на использование конструкторов `StateClosed` и `StateOpen`. Они очень похожи на `Closed` и `Open`, за исключением того, что вместо указания следующего источника вы указываете следующее состояние (в данном случае остаток списка).

Другое распространённое применение — выделение ресурсов ввода-вывода (например, открытие файла), регистрация функции освобождения ресурса (заккрытие файла) и предоставление функции получения данных из ресурса. В пакете `conduit` присутствует встроенная функция `sourceFile`, которая выдаёт поток значений типа `ByteString`. Давайте напишем ужасно неэффективную альтернативу, которая будет возвращать поток символов.

```
import Data.Conduit
import Control.Monad.Trans.Resource
import System.IO
import Control.Monad.IO.Class (liftIO)

sourceFile :: ResourceIO m => FilePath -> Source m Char
sourceFile fp = sourceIO
    (openFile fp ReadMode)
    hClose
    (\h -> liftIO $ do
        eof <- hIsEOF h
        if eof
            then return IOClosed
            else fmap IOOpen $ hGetChar h)
```

Как и `sourceState` она использует конструкторы `Open` и `Closed`. Функция `sourceIO` выполняет для нас несколько действий:

- Регистрирует функцию освобождения ресурса с трансформером `ResourceT`, гарантируя, что она будет вызвана даже в случае исключений;
- Инициализирует поле `sourceClose` чтобы освободить ресурс немедленно;
- Как только вернётся `IOClosed`, ресурс будет освобождён.

В.5. Стоки (Sinks)

Сток поглощает поток данных и генерирует результат. Он должен всегда выдавать результат и всегда только один. Это отражено в типах.

Экземпляр `Monad` для стоков упрощает композицию нескольких стоков, в больший сток. Вы также можете использовать встроенные функции для большинства своих нужд. Как и с источниками, вам редко понадобится погружаться в тонкости реализации.

Начнём с примера: получение строк из потока символов (для простоты, будем предполагать наличие переводов строк в стиле Unix).

```
import Data.Conduit
import qualified Data.Conduit.List as CL

-- Получить одну строку из потока
sinkLine :: Resource m => Sink Char m String
sinkLine = sinkState
    id -- Начальное состояние. В начале строки ничего нет
    push
    close
  where
    -- В конце строки возвращаем содержимое до текущего места
    push front '\n' =
        return $ StateDone Nothing $ front []

    -- Следующий символ - добавим его в начало и продолжим
    push front char =
        return $ StateProcessing $ front . (char:)

    -- Если встретили конец файла до перевода строки, то выдадим то,
    что имеем
    close front = return $ front []

-- Получить все строки из потока, до тех пор, пока не встретим
-- пустой строки или конца файла
sinkLines :: Resource m => Sink Char m [String]
sinkLines = do
```



```

line <- sinkLine
if null line
  then return []
  else do
    lines <- sinkLines
    return $ line : lines

content :: String
content = unlines
  [ "This is the first line."
  , "Here's the second."
  , ""
  , "After the blank."
  ]

main :: IO ()
main = do
  lines <- runResourceT $ CL.sourceList content $$ sinkLines
  mapM_ putStrLn lines

```

Запустив этот пример мы получим следующий результат:

```

This is the first line.
Here's the second.

```

Функция `sinkLine` демонстрирует использование `sinkState`, которая очень похожа на только что виденную функцию `sourceState`. Она принимает три параметра: начальное состояние, функцию добавления `push` (принимает текущее состояние и входные данные и возвращает новое состояние и результат) и функцию закрытия (принимает текущее состояние и возвращает вывод). В отличие от `sourceState`, которая не нуждается в функции закрытия, сток должен всегда возвращать результат.

В нашей функции `push` два клоза. Когда она получает символ перевода строки, процесс заканчивается, о чём говорит `StateDone`. **Nothing** означает, что входных данных не осталось (мы обсудим это позднее). Она также возвращает все полученные символы. Второй клоз просто добавляет символ к текущему состоянию и сообщает с помощью `StateProcessing`, что мы продолжаем работу. Функция `close` возвращает все символы.

Функция `sinkLines` демонстрирует как мы можем использовать монадический интерфейс для создания новых стоков. Если вы замените `sinkLine` на `getLine`, это будет выглядеть как обычный код, который читает строки из стандартного потока ввода. Этот стандартный интерфейс должен помочь вам легко освоиться и быстро начать писать код.

B.5.1. Типы

Типы для стоков несколько более хитроумны, чем для источников. Давайте взглянем на них:

```

type SinkPush input m output = input -> ResourceT m (SinkResult ⊞
    input m output)
type SinkClose m output = ResourceT m output

data SinkResult input m output =
    Processing (SinkPush input m output) (SinkClose m output)
  | Done (Maybe input) output

data Sink input m output =
    SinkNoData output
  | SinkData
    { sinkPush :: SinkPush input m output
    , sinkClose :: SinkClose m output
    }
  | SinkLift (ResourceT m (Sink input m output))

```

Всякий раз, когда в сток добавляются данные, он может сказать, что ему нужно больше данных (`Processing`), или, что всё готово. Если обработка ещё идет, то он должен предоставить обновлённые функции `push` и `close`. Если всё готово, то он возвращает результат и необработанные данные. Довольно тривиально.

Первая настоящая неочевидность заключается в наличии трёх конструкторов у типа `Sink`. Зачем нам конструктор `SinkNoData`: разве стоки созданы не для потребления данных? Дело в том, что он нужен для эффективной реализации экземпляра класса `Monad`. Когда мы вызываем `return`, мы вывозаем значение, которое не требует данных для вычисления. Мы могли бы эмулировать подобное поведение с помощью конструктора `SinkData` примерно так:

```

myReturn a = SinkData (\input -> return (Done (Just input) a)) ⊞
    (return a)

```

Но это бы вызывало чтение лишней порции входных данных, которая нам сейчас не нужна и, вероятно, вообще не понадобится. (Взгляните ещё раз на пример с `sinkLines`) Именно поэтому мы и имеем дополнительный конструктор, обозначающий, что никаких входных данных не требуется. Подобным образом, `SinkLift` предоставляется для более эффективной реализации экземпляра `MonadTrans`.

В.5.2. Стоки: без вспомогательных функций

Попробуем реализовать несколько стоков «в лоб», без использования вспомогательных функций.

```

import Data.Conduit
import System.IO
import Control.Monad.Trans.Resource
import Control.Monad.IO.Class (liftIO)

```

```

-- Прочитать данные целиком и выкинуть их
sinkNull :: Resource m => Sink a m ()
sinkNull =
    SinkData push close
  where
    push _ignored = return $ Processing push close
    close = return ()

-- Запишем поток символов в файл. Нам понадобится что-то вроде []
-- инициализации.
-- Инициализируем функцию push, а затем вернём другую функцию push []
-- для
-- вложенного вызова. Используя withIO мы уверены в том, что []
-- дескриптор будет
-- закрыт даже в случае исключений
sinkFile :: ResourceIO m => FilePath -> Sink Char m ()
sinkFile fp =
    SinkData pushInit closeInit
  where
    pushInit char = do
        (releaseKey, handle) <- withIO (openFile fp WriteMode) hClose
        push releaseKey handle char
    closeInit = do
        -- Если файл не открывали, то нечего и закрывать
        return ()

    push releaseKey handle char = do
        liftIO $ hPutChar handle char
        return $ Processing (push releaseKey handle) (close []
        releaseKey handle)

    close releaseKey _ = do
        -- Закрывать дескриптор как можно скорее
        return ()

-- Теперь посчитаем сколько значений было в потоке.
count :: Resource m => Sink a m Int
count =
    SinkData (push 0) (close 0)
  where
    push count _ignored =
        return $ Processing (push count') (close count')
      where
        count' = count + 1

    close count = return count

```

Здесь нет ничего сложного. Разве что, обратите внимание на типичный шаблон использования: описываем свои функции `push` и `close` в клوزه **where**, а затем используем их дважды: один раз для инициализации `SinkData`, а другой — в конструкторе `Processing`. Это может быть несколько утомительно, именно поэтому и существуют вспомогательные функции.

В.5.3. Стоки: реализация со вспомогательными функциями

Давайте перепишем `sinkFile`, используя преимущества вспомогательных функций `sinkIO` и `sinkState`.

```
import Data.Conduit
import System.IO
import Control.Monad.IO.Class (liftIO)

-- Нам не нужно менять release key напрямую, sinkIO автоматически
-- освобождает ресурс, как только мы возвращаем IODone из функции
-- push,
-- или когда вызывается sinkClose.
sinkFile :: ResourceIO m => FilePath -> Sink Char m ()
sinkFile fp = sinkIO
    (openFile fp WriteMode)
    hClose
    -- push: заметьте, что сюда передаются и дескриптор и входные
    -- данные
    (\handle char -> do
        liftIO $ hPutChar handle char
        return IOProcessing)
    -- close: у нас также есть дескриптор, но мы его не используем
    (\_handle -> return ())

-- А теперь посчитаем сколько значений было в потоке
count :: Resource m => Sink a m Int
count = sinkState
    0
    -- Функция push получает на вход текущее состояние и следующие
    -- входные данные...
    (\state _ignored ->
        -- и возвращает новое состояние
        return $ StateProcessing $ state + 1)
    -- Функция close получает финальное состояние и возвращает
    -- результат.
    (\state -> return state)
```

Ничего особенного, лишь слегка меньше кода, который к тому же менее подвержен ошибкам. Использование этих двух вспомогательных функций крайне рекомендуется, так как оно гарантирует правильное управление ресурсами и обновление состояния.

В.5.4. Функции работы со списками

Даже несмотря на то, что реализовывать свои стоки довольно просто, вы скорее всего захотите воспользоваться встроенными стоками, предоставляемыми модулем `Data.Conduit.List`³. Там есть аналоги для типичных функций работы со списками, например, свертки. (Этот модуль так же содержит некоторые кондуиты, например, `map`⁴.)

Если вы желаете поупражняться в кондуитах, то реализовать функции из модуля `List` (с использованием вспомогательных функций и без) будет неплохим началом.

Давайте посмотрим на простые вещи, которые можно получить используя встроенные стоки.

```
import Data.Conduit
import qualified Data.Conduit.List as CL
import Control.Monad.IO.Class (liftIO)

-- Суммирование
sum' :: Resource m => Sink Int m Int
sum' = CL.fold (+) 0

-- Вывести все входные значения в стандартный поток вывода
printer :: (Show a, ResourceIO m) => Sink a m ()
printer = CL.mapM_ (liftIO . print)

-- Просуммировать все числа в потоке кроме первых пяти
sumSkipFive :: Resource m => Sink Int m Int
sumSkipFive = do
  CL.drop 5
  CL.fold (+) 0

-- Суммировать числа и печатать накапливаемую сумму
printSum :: ResourceIO m => Sink Int m Int
printSum = do
  total <- CL.foldM go 0
  liftIO $ putStrLn $ "Sum:␣" ++ show total
  return total
where
  go accum int = do
    liftIO $ putStrLn $ "New␣input:␣" ++ show int
    return $ accum + int
```

³<http://hackage.haskell.org/packages/archive/conduit/latest/doc/html/Data.Conduit.List.html>

⁴<http://hackage.haskell.org/packages/archive/conduit/latest/doc/html/Data-Conduit-List.html#v:map>

В.5.5. Соединение

Наконец, мы хотим как-то использовать наши стоки. Несмотря на то, что мы умеем вручную вызывать `sinkPush` и `sinkClose`, это утомительно. Например:

```
main :: IO ()
main = runResourceT $ do
  res <-
    case printSum of
      SinkData push close -> loop [1..10] push close
      SinkNoData res -> return res
  liftIO $ putStrLn $ "Got a result:␣" ++ show res
  where
    loop [] _push close = close
    loop (x:xs) push close = do
      mres <- push x
      case mres of
        Done _leftover res -> return res
        Processing push' close' -> loop xs push' close'
```

Вместо этого, рекомендуется соединять сток с источником. Это не только проще, но и менее подвержено ошибкам, кроме того такой подход обеспечивает большую гибкость в том, откуда приходят ваши данные. Перепишем пример выше:

```
main :: IO ()
main = runResourceT $ do
  res <- CL.sourceList [1..10] $$ printSum
  liftIO $ putStrLn $ "Got a result:␣" ++ show res
```

Подобное соединение заботится о проверке конструктора стока (`SinkData` vs. `SinkNoData` vs. `SinkLift`), запросе данных из источника, их передаче в сток и его закрытии.

Однако, есть один момент, который я хотел бы особенно отметить в длинном примере. В предпоследней строке мы игнорируем значение, возвращаемое с `Done`. Это приводит к потере данных. Это важная тема, над которой мы долго ломали голову. К сожалению, пока мы не можем полностью её осветить, так как ещё не обсудили главного виновника драмы: `Conduit` (тип, не пакет).

Если вкратце, то оставшееся значение входа из конструктора `Done` не всегда игнорируется. Экземпляр класса `Monad`, например, использует его для передачи данных из одного стока в другой по цепочке. В действительности, настоящий оператор соединения не всегда отбрасывает остатки. Когда мы обсудим возобновляемые источники, мы увидим, что оставшееся значение складывается обратно в буфер, чтобы позволить последующим стокам переиспользовать существующий источник для получения данных.

В.6. Кондуиты

В этой части мы рассмотрим главный тип данных в нашем пакете — кондуиты. В то время как источники генерируют данные, а стоки их поглощают, кондуиты преобразуют поток данных.

В.6.1. Типы

Так же как и ранее, начнём с изучения используемых типов.

```
data ConduitResult input m output =
  Producing (Conduit input m output) [output]
  | Finished (Maybe input) [output]

data Conduit input m output = Conduit
  { conduitPush :: input -> ResourceT m (ConduitResult input m [output])
  , conduitClose :: ResourceT m [output]
  }
```

Это очень похоже на то, что мы видели со стоками. В конduit можно положить данные и в этом случае он вернет результат. Этот результат демонстрирует либо, что данные ещё генерируются, либо, что работа закончена. При закрытии конduit возвращает ещё немного данных.

Обратим внимание на отличительные особенности. Как и в случае со стоками, мы можем добавлять только один фрагмент данных за раз, соответственно, остаток состоит из 0 или 1 фрагмента. Однако, есть некоторые отличия:

- При генерации (аналог обработки данных в стоке) мы можем возвращать выходные данные. Это связано с тем, что конduit создаёт новый поток выходных данных вместо того, чтобы вернуть единичное значение по окончании обработки.
- Сток всегда выдает одно значение на выходе, в то время как кондуиты — ноль или несколько (список). Чтобы понять почему, можно рассматривать кондуиты как функцию **concatMap** (она генерирует несколько результатов для единичного входа) или как функцию **filter** (возвращает 0 или 1 фрагмент выходных данных для каждого фрагмента входных).
- У нас нет особенных конструкторов подобных `SinkNoData`. Это так, потому что мы не предоставляем экземпляр **Monad** для кондуитов. Позже мы увидим как использовать знакомый монадический подход для их создания.

В целом, кондуиты очень похожи на всё то, что мы рассмотрели до сих пор.

В.6.2. Простые кондуиты

Начнём с определения простейших кондуитов без состояния.

```

import Prelude hiding (map, concatMap)
import Data.Conduit

-- Простой кондуит передает данные как есть
passThrough :: Monad m => Conduit input m input
passThrough = Conduit
  { conduitPush = \input -> return $ Producing passThrough [input]
  , conduitClose = return []
  }

-- применяет функцию к значениям из потока
map :: Monad m => (input -> output) -> Conduit input m output
map f = Conduit
  { conduitPush = \input -> return $ Producing (map f) [f input]
  , conduitClose = return []
  }

-- а здесь ещё и конкатенирует
concatMap :: Monad m => (input -> [output]) -> Conduit input m output
concatMap f = Conduit
  { conduitPush = \input -> return $ Producing (concatMap f) $ f []
  , conduitClose = return []
  }

```

В.6.3. Кондуиты с состоянием

Конечно же, не все кодуиты могут быть объявлены без состояния. Реализация кондуитов с состоянием в лоб не очень сложна.

```

import Prelude hiding (reverse)
import qualified Data.List
import Data.Conduit
import Control.Monad.Trans.Resource

-- Расставить элементы потока в обратном порядке. Здесь та же самая
-- проблема, что и при использовании стандартной функции reverse:
-- необходимо предварительно прочитать весь поток в память.
reverse :: Resource m => Conduit input m input
reverse =
  mkConduit []
  where
    mkConduit state = Conduit (push state) (close state)
    push state input = return $ Producing (mkConduit $ input : []
    state) []

```



```

    close state = return state

-- То же самое с сортировкой
sort :: (Ord input, Resource m) => Conduit input m input
sort =
    mkConduit []
  where
    mkConduit state = Conduit (push state) (close state)
    push state input = return $ Producing (mkConduit $ input : []
      state) []
    close state = return $ Data.List.sort state

```

Но мы можем сделать лучше. Как и в случаях с `sourceState` и `sinkState`, мы с помощью `conduitState` кое-что упростим.

```

import Prelude hiding (reverse)
import qualified Data.List
import Data.Conduit

reverse :: Resource m => Conduit input m input
reverse =
    conduitState [] push close
  where
    push state input = return $ StateProducing (input : state) []
    close state = return state

sort :: (Ord input, Resource m) => Conduit input m input
sort =
    conduitState [] push close
  where
    push state input = return $ StateProducing (input : state) []
    close state = return $ Data.List.sort state

```

В.6.4. Использование кондуитов

Кондуиты работают с другими сущностями этого пакета с помощью комбинирования (`fusing`). Кондуит может быть скомбинирован с источником, создавая новый источник; со стоком для получения нового стока; или с другим кондуитом для получения нового кондуита. Лучше всего просто взглянуть на операторы комбинирования.

```

-- Комбинирование слева: источник + кондуит = источник
($=) :: (Resource m, IsSource src) => src m a -> Conduit a m b -> []
  Source m b

-- Комбинирование справа: кондуит + сток = сток
(=$) :: Resource m => Conduit a m b -> Sink b m c -> Sink a m c

```

```
-- Комбинирование по середине: кондуит + кондуит = кондуит
(=$=) :: Resource m => Conduit a m b -> Conduit b m c -> Conduit a m c
```

Использование этих операторов довольно прямолинейно.

```
useConduits = do
  runResourceT
    $ CL.sourceList [1..10]
    $= reverse
    $= CL.map show
    $$ CL.consume

-- эквивалентно
runResourceT
  $ CL.sourceList [1..10]
  $$ reverse
  =$ CL.map show
  =$ CL.consume

-- и так же эквивалентно
runResourceT
  $ CL.sourceList [1..10]
  $$ (reverse =$= CL.map show)
  =$ CL.consume
```

Существует ещё один способ выразить то же самое. Его поиск оставлен читателю в качестве упражнения.

Вам может показаться, что такое количество различных способов комбинирования чрезмерно. Хотя в некоторых ситуациях вы можете выбирать какой способ вам больше нравится, чаще всего подойдёт только один из них. Например:

- Если у вас поток чисел и вы хотите применить кондуит (например, `map show`) только к некоторой части потока, которая будет передана определенному стоку, то нужно использовать оператор комбинирования справа.
- Если вы читаете файл и хотите распарсить его целиком как текстовые данные, стоит использовать оператор комбинирования слева, чтобы преобразовать весь поток.
- Если вам нужны переиспользуемые кондуиты, которые будут объединяться в большие кондуиты, используйте срединное комбинирование.

В.6.5. Потери данных

Забудем о кондуитах на минутку. Предположим, что мы хотим написать программу (используя только списки), которая примет список чисел, применит к ним некоторое преобразование, возьмёт первые 5 преобразованных элементов и сделает с ними

что-то, а затем возьмёт оставшиеся непреобразованные данные и сделает с ними что-нибудь ещё. Как-то так, например:

```
main = do
  let list = [1..10]
      transformed = map show list
      (begin, end) = splitAt 5 transformed
      untransformed = map read end
  mapM_ putStrLn begin
  print $ sum untransformed
```

Ясно, что это не очень хорошее общее решение, ведь мы не хотим преобразовывать все элементы списка сначала в одну сторону, а потом обратно. К тому же, мы не всегда будем иметь функцию обратного преобразования. Другая причина — неэффективность. В данном случае мы можем написать более эффективное решение:

```
main = do
  let list = [1..10]
      (begin, end) = splitAt 5 list
      transformed = map show begin
  mapM_ putStrLn transformed
  print $ sum end
```

Обратите внимание: мы разбиваем список до применения нашего преобразования. Этот подход работает, потому что при использовании `map` у нас есть взаимно-однозначное соответствие между элементами. Поэтому выделение первых пяти элементов до или после преобразования суть одно и то же. Но что будет, если мы заменим `map show` чем-то более сложным?

```
deviousTransform =
  concatMap go
  where
    go 1 = [show 1]
    go 2 = [show 2, "two"]
    go 3 = replicate 5 "three"
    go x = [show x]
```

Теперь взаимно-однозначное соответствие отсутствует, а следовательно, мы не можем использовать второй способ. На самом деле всё гораздо хуже: мы также не можем использовать и первый способ, потому что у нас нет обратного преобразования к функции `deviousTransform`.

Я знаю только одно решение данной проблемы: преобразовывать значения по одному. Конечный вариант будет выглядеть примерно так:

```
deviousTransform 1 = [show 1]
deviousTransform 2 = [show 2, "two"]
deviousTransform 3 = replicate 5 "three"
```

```

deviousTransform x = [show x]

transform5 :: [Int] -> ([String], [Int])
transform5 list =
  go [] list
  where
    go output (x:xs)
      | newLen <= 5 = (take 5 output', xs)
      | otherwise = go output' xs
      where
        output' = output ++ deviousTransform x
        newLen = length output'

-- Вырожденный случай: недостаточно данных для получения 5
результатов
go output [] = (output, [])

main = do
  let list = [1..10]
      (begin, end) = transform5 list
  mapM_ putStrLn begin
  print $ sum end

```

Результат работы программы будет таким:

```

1
2
two
three
three
49

```

Стоит обратить внимание, что число 3 преобразуется в пять копий слова "three", однако только две из них попадают в результат, остальные же отбрасываются при вызове `take 5`.

Этот пример наглядно демонстрирует проблему потери данных при использовании кондуитов. Заставляя кодуиты принимать только один фрагмент данных за раз мы избегаем ненужного преобразования чрезмерного количества данных. Однако это не значит, что мы не теряем данные: если кондуит, генерирует так много данных, что стек не может их все принять, то некоторая их часть потеряется.

Другими словами, кондуиты не разбивают данные на куски во избежание потерь. Эта проблема встречается не только при использовании кондуитов. Если вы взглянете на реализацию `concatMapM` для эnumераторов⁵, вы увидите, что там элементы обрабатываются по одному. В кондуитах мы предпочли форсировать эту проблему на уровне типов.

⁵<http://hackage.haskell.org/package/enumerator>

В.6.6. SequencedSink

Предположим, что нам необходимо скомбинировать существующие кондуиты и стоки, чтобы получить новый, более сложный кондуит. Например, мы хотим написать кондуит, который принимает поток чисел и суммирует их по пять. Другими словами, для входа [1..50] он должен вернуть [15,40,65,90,115,140,165,190,215,240]. Мы определенно можем это сделать с помощью низкоуровневого интерфейса кондуитов.

```
sum5Raw :: Resource m => Conduit Int m Int
sum5Raw =
  conduitState (0, 0) push close
  where
    push (total, count) input
      | newCount == 5 = return $ StateProducing (0, 0) [newTotal]
      | otherwise     = return $ StateProducing (newTotal, 0
newCount) []
      where
        newTotal = total + input
        newCount = count + 1
    close (total, count)
      | count == 0 = return []
      | otherwise  = return [total]
```

Но это неудобно, так как мы уже имеем всё, что нужно, чтобы реализовать то же самое на более высоком уровне. У нас есть сток свёртки, чтобы складывать числа, и изолирующий кондуит, который позволит передать в сток только определённое количество данных. Может быть мы сможем их скомбинировать?

Нам нужен SequencedSink, который является обычным стоком, за исключением того, что он возвращает специальный SequencedSinkResponse. Это значение может вернуть новый результат, остановить обработку данных или передать управление в новый кондуит. (Дополнительную информацию вы можете почерпнуть в Haddock's.) Потом мы сможем преобразовать его в кондуит с помощью функции sequenceSink. Она также принимает состояние, которое будет передано напрямую в сток.

Теперь мы можем переписать sum5Raw на значительно более высоком уровне.

```
sum5 :: Resource m => Conduit Int m Int
sum5 = sequenceSink () $ \() -> do
  nextSum <- CL.isolate 5 =$ CL.fold (+) 0
  return $ Emit () [nextSum]
```

Здесь все () — это неиспользуемое состояние, которое передаётся между функциями, его можно игнорировать. Кроме того, мы делаем ровно то, что хотели. Мы скомбинировали isolate с fold, чтобы получить сумму следующих пяти элементов из потока. После чего мы возвращаем это значение и начинаем всё сначала.

Предположим, что мы хотим слегка изменить условие. Мы хотим получить первые 8 сумм, а затем вернуть оставшиеся значения, увеличенные вдвое. Будем хранить

количество значений, которые мы вернули, в состоянии, а затем воспользуемся конструктором `StartConduit`, чтобы передать управление кондуиту, умножающему на 2.

```
sum5Pass :: Resource m => Conduit Int m Int
sum5Pass = sequenceSink 0 $ \count -> do
  if count == 8
  then return $ StartConduit $ CL.map (* 2)
  else do
    nextSum <- CL.isolate 5 =$ CL.fold (+) 0
    return $ Emit (count + 1) [nextSum]
```

Очевидно, что примеры выше несколько искусственны, но я надеюсь, что они помогли прояснить мощь и простоту этого подхода.

В.7. Буферизация

Буферизация — это одно из уникальных свойств кондуитов. С её использованием вам больше не придётся заботиться о потоке управления в вашей программе. Иногда, она поможет упростить код.

В.7.1. Инверсия управления

Буферизация была одной из главных причин для создания кондуитов. Чтобы понять её важность, рассмотрим подход, который мы видели не так давно, назовём его инверсией управления.

Инверсия управления в разных контекстах может означать разное. Если вы возражаете против использования этого термина здесь, то можете называть это “тёплой пушистой штуковиной”, я не буду возражать.

Предположим, что вы хотите посчитать, как много переводов строки в файле. При стандартном императивном подходе вы бы действовали примерно так:

- открываем файл;
- читаем данные в буфер;
- обходим данные в буфере, инкрементируя счётчик каждый раз, как встречаем перевод строки;
- возвращаемся на шаг два;
- закрываем файл.

Обратите внимание, что ваш код явно вызывает другой код, и этот другой код передает управление обратно. Вы полностью контролируете поток управления вашей программой. В кондуитах, как мы видели, так сделать не получится. Вместо этого мы будем:

- писать в сток, который считает переводы строки и прибавляет полученное значение к аккумулятору;

- соединять сток с источником.

Я не сомневаюсь в том, что этот подход проще. Нам не придётся заботиться об открытии и закрытии файла, ровно как и о чтении данных из него. Вместо этого необходимые данные просто предоставляются вам. Это преимущество инверсии управления: вы можете сосредоточиться именно на своей части кода.

Мы используем подобный подход всюду в Хаскеле: например, вместо `readMVar` и `putMVar`, мы используем `withMVar`. Не используем `openFile` и `closeFile`, а сразу пишем `withFile` и передаем функцию, которая использует дескриптор. Даже в C используется нечто подобное: зачем вызывать `malloc` и `free` вместо `alloca`?

Вообще-то, последний пункт не совсем корректен. Конечно же, мы не можем использовать `alloca` везде. Функция `alloca` выделяет память только локально на стеке, а не в куче. Выделенную таким образом память никак нельзя вернуть наружу из данной функции.

На самом деле, подобное ограничение применимо ко всему семейству `with`-функций: мы никогда не сможем вернуть выделенный ресурс наружу из данного «блока» кода. Обычно это и не нужно, просто мы должны признать, что это изменение повлияет на структуру наших программ. Зачастую, в простых случаях, изменение минимально, однако в более крупных задачах с ним становится непросто справиться, а порой и попросту не возможно.

В.7.2. Веб-сервер

Предположим, мы собираемся написать веб-сервер. Будем использовать следующие низкоуровневые операции:

```
data Socket
recv    :: Socket -> Int -> IO ByteString -- вернёт пустую строку []
        если сокет закрыт
sendAll :: Socket -> ByteString -> IO ()
```

Теперь нам надо реализовать функцию `handleConn`, которая будет обрабатывать одиночное соединение. Она будет выглядеть примерно так:

```
data Request -- заголовки запроса, версия HTTP протокола и т.д.
data Response -- код ответа, заголовки, тело.
type Application = Request -> IO Response
handleConn :: Application -> Socket -> IO ()
```

Что должна делать `handleConn`? Если кратко, то:

- разобрать тело запроса;
- разобрать заголовки запроса;
- создать значение типа `Request`;
- передать его приложению и получить обратно ответ типа `Response`;

- передать ответ обратно в сокет.

Начнём с реализации пунктов 1 и 2 вручную, без использования кондуитов. Упростим задачу, предположив, что запрос состоит из трёх строк, разделённых пробелами. В конечном итоге у нас выйдет что-то похожее на:

```
data RequestLine = RequestLine ByteString ByteString ByteString

parseRequestLine :: Socket -> IO RequestLine
parseRequestLine socket = do
  bs <- recv socket 4096
  let (method:path:version:ignored) = S8.words bs
  return $ RequestLine method path version
```

У этого подхода есть две проблемы: никак не обрабатывается случай, когда данные состоят из менее чем трёх строк, кроме того, лишние данные отбрасываются. Мы определённо можем решить обе проблемы вручную, но это будет утомительно. Гораздо проще переписать всё в терминах кондуитов.

```
import Data.ByteString (ByteString)
import qualified Data.ByteString as S
import Data.Conduit
import qualified Data.Conduit.Binary as CB
import qualified Data.Conduit.List as CL

data RequestLine = RequestLine ByteString ByteString ByteString

parseRequestLine :: Sink ByteString IO RequestLine
parseRequestLine = do
  let space = toEnum $ fromEnum ' '
  let getWord = do
    CB.dropWhile (== space)
    bss <- CB.takeWhile (/= space) =$ CL.consume
    return $ S.concat bss

  method <- getWord
  path <- getWord
  version <- getWord
  return $ RequestLine method path version
```

Это значит, что нашему коду будут предоставлены данные как только они поступят, а дополнительные данные будут автоматически буферизованы в источнике, готовые к последующему использованию. Теперь мы можем легко собрать нашу программу из составных частей, пользуясь мощностью подхода на кондуитах:

```
import Data.ByteString (ByteString)
import Data.Conduit
```



```

import Data.Conduit.Network (sourceSocket)
import Control.Monad.IO.Class (liftIO)
import Network.Socket (Socket)

data RequestLine = RequestLine ByteString ByteString ByteString
type Headers = [(ByteString, ByteString)]
data Request = Request RequestLine Headers
data Response = Response
type Application = Request -> IO Response

parseRequestHeaders :: Sink ByteString IO Headers
parseRequestHeaders = undefined

parseRequestLine :: Sink ByteString IO RequestLine
parseRequestLine = undefined

sendResponse :: Socket -> Response -> IO ()
sendResponse = undefined

handleConn :: Application -> Socket -> IO ()
handleConn app socket = do
  req <- runResourceT $ sourceSocket socket $$ do
    requestLine <- parseRequestLine
    headers <- parseRequestHeaders
    return $ Request requestLine headers
  res <- liftIO $ app req
  liftIO $ sendResponse socket res

```

В.7.3. Где же тело запроса?

Всё выглядит замечательно до тех пор, пока мы не осознаём, что не можем прочитать тело запроса. Сейчас `Application` просто получает значение типа `Request`, и живёт в монаде `IO`, не имея абсолютно никакого доступа ко входному потоку данных.

Это можно легко исправить, завернув `Application` в монаду `Sink`. Это тот же самый трюк, который мы использовали в `WAI 0.4`, основанном на пакете `enumerator`. Однако, есть две проблемы:

- Люди находят это сбивающим с толку. Они ожидают, что значение типа `Request` будет иметь поле `requestBody` типа `Source`.
- В некоторых случаях использование такого подхода становится невероятно трудным. Например, написать `HTTP`-прокси комбинируя `WAI` и `http-enumerator` оказалось почти невозможным.

Это обратная сторона инверсии управления. Наш код желает контролировать ситуацию. Он желает, чтобы ему дали нечто, откуда можно брать данные, нечто куда их можно сложить, и работать с этим. Нам нужно какое-то решение этой проблемы.

Если вы считаете, что описанная проблема с прокси не так ужасна, это только потому, что я не стал вдаваться в подробности. Нам надо также учитывать, что тело запроса это тоже поток, работа с которым происходит и на клиентской и на серверной сторонах.

Простейшим решением было бы создание нового источника и передача его в `Application`. К сожалению, это влечёт проблемы с буферизацией. Когда мы соединяем наш источник со стоками `parseRequestLine` и `parseRequestHeaders`, происходит вызов функции `recv`. Если прочитанных данных будет недостаточно, чтобы разобрать все заголовки, то произойдёт ещё один вызов. Когда данных будет достаточно, процесс остановится. Однако, есть вероятность, что он остановится не строго в конце заголовков, и некоторая часть тела запроса также будет прочитана.

Если мы просто создадим новый источник и передадим его запросу, там будет отсутствовать начало тела запроса. Нам нужен какой-то способ передать буферизированные данные.

В.7.4. Буферизованные источники (`BufferedSource`)

Наконец, мы можем представить последний тип данных в кондуитах: `BufferedSource`. Это абстрактный тип данных, и всё что он в действительности делает, это содержит изменяемую ссылку на буфер и соответствующий ему источник. Чтобы создать значение такого типа, вы можете воспользоваться функцией `bufferSource`.

```
bufferSource :: Resource m => Source m a -> ResourceT m (BufferedSource m a)
```

Это небольшое изменение позволит нам легко решить нашу дилемму связанную с веб-сервером. Вместо соединения источника со стоком, осуществляющими разбор, мы будем использовать `BufferedSource`. При закрытии каждого соединения, все оставшиеся данные будут складываться обратно в буфер. Для нашего веб-сервера мы создадим `BufferedSource`, воспользуемся им для чтения строки запроса и заголовков, а затем передадим этот же `BufferedSource` в `Application` для чтения тела запроса.

В.7.5. Классы типов

Мы хотим иметь возможность соединять буферизованный источник со стоком, как будто это обычный источник. Мы также хотим уметь комбинировать его с кондуитами. Для обеспечения этой функциональности, в кондуитах есть класс типов `IsSource`. Его экземпляры реализованы как для `Source`, так и для `BufferedSource`. Операторы соединения (`$$`) и комбинирования слева (`$=`) используют этот класс типов.

В экземпляре этого класса типов для `BufferedSource` есть одна тонкость. Предположим, что мы хотим написать функцию копирования файлов без буферизации. Это довольно стандартный случай использования кондуитов:

```
sourceFile input $$ sinkFile output
```

Когда этот код исполняется, открываются оба файла: и файл ввода и файл вывода; данные копируются, после чего оба файла закрываются. Слегка изменим код, чтобы использовать буферизацию:

```
bsrc <- bufferSource $ sourceFile input
bsrc $$ isolate 50 =$ sinkFile output1
bsrc $$ sinkFile output2
```

А здесь, когда открывается и закрывается входной файл? Открытие происходит в первой строке, а закрытие, если следовать обычным правилам для источников, после исполнения второй. Однако, если это так, то мы не сможем использовать файл в третьей строке!

Вместо этого, оператор \$\$ не закрывает файл. Следовательно, вы можете передавать буферизованный источник для выполнения тех действий, которые вам нужны, не беспокоясь, что дескриптор файла будет закрыт без вашего ведома.

Как мы упоминали ранее, инвариант гласит, что из источника невозможно забрать данные, если он уже вернул `Closed`. Чтобы облегчить работу с буферизованными источниками, этот инвариант здесь не выполняется. Ответственность за закрытие соответствующего источника, а также за то, что он больше не будет использоваться, лежит на реализации `BufferedSource`.

Будьте осторожны: когда вы закончили работу с буферизованным источником, вам следует вручную вызвать `bsourceClose`. Однако, это обычно является оптимизацией, так как источник будет автоматически закрыт по окончании работы функции `runResourceT`.

В.7.6. Возвращаясь к веб-серверу

Как же именно будет выглядеть теперь наш веб-сервер?

```
import Data.ByteString (ByteString)
import Data.Conduit
import Data.Conduit.Network (sourceSocket)
import Control.Monad.IO.Class (liftIO)
import Network.Socket (Socket)

data RequestLine = RequestLine ByteString ByteString ByteString
type Headers = [(ByteString, ByteString)]
data Request = Request RequestLine Headers (BufferedSource IO ByteString)
data Response = Response
type Application = Request -> ResourceT IO Response

parseRequestHeaders :: Sink ByteString IO Headers
parseRequestHeaders = undefined

parseRequestLine :: Sink ByteString IO RequestLine
parseRequestLine = undefined

sendResponse :: Socket -> Response -> IO ()
```

```
sendResponse = undefined

handleConn :: Application -> Socket -> IO ()
handleConn app socket = runResourceT $ do
  bsrc <- bufferSource $ sourceSocket socket
  requestLine <- bsrc $$ parseRequestLine
  headers <- bsrc $$ parseRequestHeaders
  let req = Request requestLine headers bsrc
  res <- app req
  liftIO $ sendResponse socket res
```

Мы произвели несколько небольших изменений. Во-первых, наше приложение теперь завернуто в `ResourceT IO` монаду. Это не необходимо, но очень удобно: `application` теперь может регистрировать функции освобождения ресурсов, которые выполняются только тогда, когда ответ будет полностью послан клиенту.

Главные же изменения произошли в функции `handleConn`. Сейчас мы начинаем работу с буферизации источника, который затем используется дважды в нашей функции и передается в приложение.

Приложение С

Интерфейс веб-приложений

Практически всякий язык, используемый для веб-разработки, сталкивается с проблемой низкоуровневого интерфейса между веб-сервером и приложением. Самым ранним примером решения этой проблемы является почтенный и потёртый в боях CGI (Common Gateway Interface), который предоставляет не зависящий от языка интерфейс на основе стандартных потоков ввода/вывода и переменных среды окружения.

Но уже в те времена, когда Perl только становился языком веб-программирования де-факто, стал очевиден основной недостаток CGI — для каждого запроса должен быть запущен новый процесс. При работе с интерпретируемым языком и приложениями, требующими подключения к базе данных, эти издержки стали чрезмерны. FastCGI (и позднее SCGI) возник как преемник CGI, но, похоже, основная часть мира программирования направилась в другом направлении.

Каждый язык начал создавать свой собственный стандарт для взаимодействия с сервером. `mod_perl`, `mod_python`, `mod_php`, `mod_ruby`. Для одного и того же языка появлялись различные интерфейсы. В некоторых случаях получались даже интерфейсы поверх интерфейсов. Всё это привело к ещё большему дублированию усилий: приложение на Python, созданное для работы с FastCGI, не работало с `mod_python`, а `mod_python` существовал только для определённых веб-серверов. И подобные модули расширения веб-сервера должны были быть реализованы для каждого языка.

У Haskell своя история. Сначала у нас был пакет `cgi`¹, который предоставлял монадный интерфейс. Позднее появился пакет `fastcgi`² с таким же интерфейсом. Тем временем, казалось, большая часть веб-разработки на Haskell сфокусировалась на написании отдельных серверов. Проблема в том, что у каждого сервера свой собственный интерфейс, а это означает, что вы вынуждены при разработке ориентироваться на конкретный сервер. Что, в свою очередь, означает невозможность использовать общий функционал наподобие GZIP кодирования, сервера разработки или фреймворка для тестирования.

WAI пробует решить эту проблему, обеспечивая обобщённый и эффективный интерфейс между веб-серверами и приложениями. Любой *обработчик*, поддерживающий этот интерфейс может обслуживать любое приложение WAI, в то же время любое

¹<http://hackage.haskell.org/package/cgi>

²<http://hackage.haskell.org/package/fastcgi>

приложение, использующее этот интерфейс, может быть запущено любым обработчиком.

На момент написания этой книги существует несколько веб-серверов, реализующих WAI, такие как Warp, FastCGI и сервер разработки. Существуют и другие реализации, известные лишь посвященным, например, `wai-handler-webkit`³, используемый для создания настольных приложений. Пакет `wai-extra`⁴ предоставляет много обычных компонент промежуточного уровня, к примеру, реализующих поддержку GZIP, JSON-P и виртуального хостинга. Библиотека `wai-test`⁵ облегчает написание модульных тестов, а `wai-handler-devel`⁶ позволяет разрабатывать приложения, не отвлекаясь на остановку сервера для компиляции. Yesod уже ориентирован на использование WAI, Haproxy также находится в процессе перехода на него. Этот интерфейс также используется некоторыми приложениями, которые вообще не используют фреймворки, включая новую версию Hoogle.

Yesod предоставляет альтернативный подход для сервера разработки, известный как `yesod-devel`. Отличие от `wai-handler-devel` состоит в том, что `yesod-devel` компилирует ваш код каждый раз, учитывая все настройки в вашем `cabal`-файле. Этот подход рекомендуется при разработке на Yesod.

С.1. Интерфейс

Интерфейс сам по себе довольно незатейлив: приложение принимает запрос и возвращает ответ. Ответом является статус HTTP, список заголовков и тело ответа. Запрос содержит различную информацию: запрашиваемый путь, строку запроса, тело запроса, версию HTTP и т.д.

С.1.1. Тело ответа

В Haskell есть тип данных известный как ленивая строка байтов. Используя ленивость, вы можете создавать большие значения, не переполняя память. Например, используя ленивый ввод/вывод, вы можете иметь значение, которое представляет всё содержимое файла и в то же время занимает небольшой участок памяти. В теории, ленивая строка байтов является единственным необходимым представлением тела ответа.

На практике же, в то время как ленивые строки байтов замечательно подходят для «чистых» значений, ленивый ввод/вывод, необходимый для чтения файлов, вносит некоторый недетерминизм в нашу программу. При обработке тысяч небольших файлов в секунду, ограничивающим фактором является не память, а число дескрипторов файлов. При использовании ленивого ввода/вывода файловые дескрипторы не могут быть освобождены мгновенно, что приводит к исчерпанию ресурсов. Для решения этой проблемы WAI использует кондуиты⁷.

³<http://hackage.haskell.org/package/wai-handler-webkit>

⁴<http://hackage.haskell.org/package/wai-extra>

⁵<http://hackage.haskell.org/package/wai-test>

⁶<http://hackage.haskell.org/package/wai-handler-devel>

⁷<http://hackage.haskell.org/package/conduit>

Версии WAI до 1.0 использовали еnumераторы^a вместо кондуитов. В то время как, и кондуиты, и еnumераторы решают одну и ту же самую базовую задачу, опыт показал, что еnumераторы слишком ограничивают возможности своим подходом — инверсией управления, что усложняет структурирование сложных систем, таких как потоковый прокси-сервер (streaming proxy server). Кондуиты были спроектированы с единственной целью — сделать WAI лучше.

^a<http://hackage.haskell.org/package/enumerator>

Уместным для нас типом данных при работе с кондуитами является тип `Source` — источник. Источник производит поток данных, отдавая одну порцию за раз. В случае WAI, тело запроса будет источником, передаваемым приложению, а тело ответа будет источником, возвращаемым из приложения.

Есть ещё две оптимизации: многие системы предоставляют системный вызов `sendfile`, который отправляет файл напрямую в сокет, минуя неизбежные копирования в памяти, которые присущи более общим системным вызовам ввода/вывода. Кроме того, в Haskell есть тип данных `Builder` — строитель, который позволяет более эффективно копировать байты в буферы.

Как следствие, для тела ответа WAI есть три конструктора: один для чистых строителей (`ResponseBuilder`), один для источника, производящего строителей, (`ResponseSource`) и один для файлов (`ResponseFile`).

C.1.2. Тело запроса

Чтобы избежать загрузки всего тела запроса в оперативную память, здесь мы также используем источники. Так как это значение предназначено для чтения (а не для записи), вместо `Builder` мы используем `ByteString`. Тип данных `Request` содержит поле с именем `requestBody`, имеющее тип `BufferedSource IO ByteString`. Мы можем использовать все стандартные функции кондуитов для работы с этим источником.

Теоретически тело запроса может содержать любой тип данных, но наиболее частыми являются данные, закодированные URL-кодировкой, и составные данные форм. Пакет `wai-extra`⁸ включает встроенную поддержку для разбора данных форматов эффективно использующим память способом.

C.2. Hello world

Чтобы продемонстрировать простоту WAI, давайте взглянем на простейший пример. В этом примере мы используем расширение языка `OverloadedStrings`, чтобы избежать явной упаковки строковых значений в строки байт.

```
{-# LANGUAGE OverloadedStrings #-}
import Network.Wai
import Network.HTTP.Types (status200)
import Network.Wai.Handler.Warp (run)
```

⁸<http://hackage.haskell.org/package/wai-extra>

```

application _ = return $
  responseLBS status200 [("Content-Type", "text/plain")] "Hello_World"

main = run 3000 application

```

24/hello-world.hs

В строках со 2-й по 4-ю выполняется импорт необходимых модулей. Warp предоставляется пакетом `warp`⁹ и является исторически первым веб-сервером, реализующим WAI. Также WAI использует пакет `http-types`¹⁰, который предоставляет некоторые типы данных и вспомогательные значения, включая `status200`.

Сперва мы определяем наше приложение. Так как нам безразличны конкретные параметры запроса, то мы игнорируем аргумент нашей функции. Для любого запроса мы возвращаем ответ с кодом статуса 200 («ОК»), типом содержимого `text/plain` и телом, содержащим слова «Hello World». Довольно просто.

С.3. Middleware — компоненты промежуточного уровня

В дополнение к возможности запуска наших приложений без изменения кода на различных серверах реализующих WAI, WAI имеет ещё и другое преимущество — возможность создавать и использовать компоненты промежуточного уровня. Они являются по сути преобразователями приложения, получая на вход одно приложение и возвращая другое.

Компоненты промежуточного уровня могут использоваться для различных нужд: исправления URL, аутентификации, кэширования, обработки запросов JSON-P. Но, возможно, наиболее полезным и наиболее интуитивно понятным является компонент реализующий `gzip`-сжатие. Этот компонент работает довольно просто: он разбирает заголовки запроса для определения, поддерживает ли клиент сжатие, и, если клиент поддерживает, то выполняет сжатие тела ответа, добавляя соответствующий заголовок ответа.

Самое замечательное в компонентах промежуточного уровня заключается в том, что они ненавязчивы. Давайте посмотрим, как применить `gzip` к нашему примеру.

```

{-# LANGUAGE OverloadedStrings #-}
import Network.Wai
import Network.Wai.Handler.Warp (run)
import Network.Wai.Middleware.Gzip (gzip, def)
import Network.HTTP.Types (status200)

application _ = return $ responseLBS status200 [("Content-Type", "
  "text/plain")] "Hello_World"

main = run 3000 $ gzip def application

```

⁹<http://hackage.haskell.org/package/warp>

¹⁰<http://hackage.haskell.org/package/http-types>

24/hello-world-gzip.hs

Мы добавили строку импорта, чтобы получить доступ к компоненте, и затем просто применили `gzip` к нашему приложению. Вы также можете *выстраивать в цепочку* несколько компонент: к примеру, строка `gzip False $ jsonp $ \ othermiddleware $ myapplication` вполне корректна. Но следует заметить, что порядок применения компонент может быть важен. К примеру, для `jsonp` нужны несжатые данные. Если вы его примените после применения `gzip`, то получите проблемы.

Приложение D

Типы для настроек

Представим, что вы пишете веб-сервер. Вы хотите, чтобы вашему серверу передавали порт, который он будет слушать, и приложение для запуска. И вы создаёте следующую функцию:

```
run :: Int -> Application -> IO ()
```

Но внезапно вы понимаете, что некоторые люди захотят настроить продолжительность ожидания таймаута. И тогда вы меняете свой API:

```
run :: Int -> Int -> Application -> IO ()
```

Какой **Int** в итоге для времени ожидания, а какой для номера порта? Да, вы могли бы создать какие-нибудь синонимы для типов или прокомментировать ваш код. Но здесь есть ещё одна проблема, пробирающаяся в ваш код: такая функция `run` становится неподдерживаемой. Вскоре потребуется получить дополнительный параметр для обозначения, как следует обрабатывать исключения, а затем ещё один для управления, к какому хосту привязываться, и так далее.

Поэтому более масштабируемое решение — ввести тип данных для настроек:

```
data Settings = Settings
  { settingsPort :: Int
  , settingsHost :: String
  , settingsTimeout :: Int
  }
```

И это делает вызывающий код почти самодокументированным:

```
run Settings
  { settingsPort = 8080
  , settingsHost = "127.0.0.1"
  , settingsTimeout = 30
  } myApp
```

Великолепно, яснее быть не может, не так ли? Да, так, но что случится, когда у вас будет 50 настроек для вашего веб-сервера? Вы действительно хотите задавать их все каждый раз? Конечно, нет. Так что вместо этого веб-сервер должен предоставлять набор значений по умолчанию:

```
defaultSettings = Settings 3000 "127.0.0.1" 30
```

И теперь, вместо необходимости писать длинный кусок кода как выше, мы можем отделаться лишь этим:

```
run defaultSettings { settingsPort = 8080 } myApp -- (1)
```

Это замечательно, за исключением одного маленького препятствия. Допустим, мы решаем добавить дополнительную запись в `Settings`. Любой код во внешнем мире, выглядящий как

```
run (Settings 8080 "127.0.0.1" 30) myApp -- (2)
```

станет нерабочим, так как теперь конструктор `Settings` принимает четыре аргумента. Правильной вещью стало бы увеличение номера основной версии (`major version`), чтобы не нарушить работоспособность зависимых пакетов. Но менять номер основной версии для каждого минимального изменения настроек — это неудобство. Решение? Не экспортировать конструктор `Settings`:

```
module MyServer
  ( Settings
  , settingsPort
  , settingsHost
  , settingsTimeout
  , run
  , defaultSettings
  ) where
```

С таким подходом никто не сможет написать код наподобие (2), поэтому вы можете свободно добавлять новые настройки, не боясь поломать код.

Единственный недостаток такого подхода — не сразу очевидно из документации `Haddock`, что вы фактически можете менять настройки, используя синтаксис записей. Основная идея этой главы: прояснить, что происходит в библиотеках, использующих такую технику.

Я сам использую эту технику в нескольких местах, не стесняйтесь заглянуть в `Haddock`s, чтобы понять, что я имею в виду.

- `Warp: Settings`¹

¹<http://hackage.haskell.org/packages/archive/warp/latest/doc/html/Network-Wai-Handler-Warp.html#t:Settings>

- `http-conduit`: `Request`² и `ManagerSettings`³
- `xml-conduit`:
 - Разбор: `ParseSettings`⁴
 - Представление: `RenderSettings`⁵

Кстати, `http-conduit` и `xml-conduit` фактически создают экземпляры класса типов `Default`⁶ из пакета `Data.Default` вместо объявления своего нового идентификатора. Это означает, что вы можете просто печатать `def` вместо `defaultParserSettings`.

²<http://hackage.haskell.org/packages/archive/http-conduit/latest/doc/html/Network-HTTP-Conduit.html#t:Request>

³<http://hackage.haskell.org/packages/archive/http-conduit/latest/doc/html/Network-HTTP-Conduit.html#t:ManagerSettings>

⁴<http://hackage.haskell.org/packages/archive/xml-conduit/latest/doc/html/Text-XML-Stream-Parse.html#t:ParseSettings>

⁵<http://hackage.haskell.org/packages/archive/xml-conduit/latest/doc/html/Text-XML-Stream-Render.html#t:RenderSettings>

⁶<http://hackage.haskell.org/packages/archive/data-default/latest/doc/html/Data-Default.html#t:Default>

Приложение E

http-conduit

Большая часть Yesod относится к предоставлению контента по протоколу HTTP. Но это лишь часть картины: кто-то должен получать этот контент. И даже если вы пишете веб-приложение, иногда этим «кто-то» можете быть вы сами. Если вы хотите получать контент от других сервисов или взаимодействовать с RESTful API, вам потребуется писать клиентский код, и рекомендуемый подход для этого — `http-conduit`¹.

Эта глава напрямую с Yesod не связана, и будет, вообще говоря, полезной для всех, кто хочет выполнять HTTP запросы.

E.1. Конспект

```
{-# LANGUAGE OverloadedStrings #-}
import Network.HTTP.Conduit -- основной модуль

-- Поточковый интерфейс использует кондуиты
import Data.Conduit
import Data.Conduit.Binary (sinkFile)

import qualified Data.ByteString.Lazy as L
import Control.Monad.IO.Class (liftIO)

main :: IO ()
main = do
    -- Простейший запрос: просто загружаем информацию с указанного URL
    -- как ленивую ByteString.
    simpleHttp "http://www.example.com/foo.txt" >>= L.writeFile "foo.txt"

    -- Теперь используем потоковый интерфейс. Нам необходимо запускать
```

¹<http://hackage.haskell.org/package/http-conduit>

```

-- всё внутри ResourceT, чтобы быть уверенными, что все наши
-- соединения корректно закрываются в случае возникновения []
исключений
runResourceT $ do
  -- Нам нужен Manager, который отслеживает открытые соединения.
  -- simpleHttp создаёт нового менеджера соединений при каждом
  -- запуске (т.е. соединения никогда не используются повторно)
  manager <- liftIO $ newManager def

  -- Более эффективная версия запроса с simpleHttp выше.
  -- Сначала разбираем URL в запрос
  req <- liftIO $ parseUrl "http://www.example.com/foo.txt"

  -- Теперь получаем ответ
  res <- http req manager

  -- И, наконец, записываем результат в файл
  responseBody res $$+- sinkFile "foo.txt"

  -- Сделаем это POST запросом, не следуем за перенаправлениями
  -- и принимаем любой код статуса ответа
  let req2 = req
      { method = "POST"
      , redirectCount = 0
      , checkStatus = \_ _ _ -> Nothing
      }
  res2 <- http req2 manager
  responseBody res2 $$+- sinkFile "post-foo.txt"

```

26/synopsis.hs

E.2. Основные положения

Самый простой способ выполнения запроса в http-conduit — использовать функцию `simpleHttp`. Эта функция принимает строку (типа `String`), представляющую URL, и возвращает значение типа `ByteString` с содержимым этого URL. Её реализация состоит из нескольких шагов:

- Создаётся новый менеджер соединения `Manager`;
- URL преобразуется в `Request`. Если URL некорректный, то выбрасывается исключение;
- Выполняется HTTP запрос, следующий по всем перенаправлениям с сервера;
- Если код статуса ответа находится вне диапазона 200-х значений, выбрасывается исключение;

- Тело запроса считывается в память и возвращается;
- Вызывается `runResourceT`, которая освобождает все ресурсы (например, открытый сокет к серверу).

Если вы хотите больше контроля над тем, что происходит, вы можете настроить любой из описанных шагов (плюс ещё несколько), явно создавая значение `Request`, вручную размещая `Manager`, и используя функции `http` и `httpLbs`.

E.3. Request

Самый лёгкий путь создания `Request` — с помощью функции `parseUrl`. Эта функция вернёт значение в любой монаде, являющейся экземпляром класса `Failure`, например, `Maybe` или `IO`. Последняя из указанных используется чаще всего и приводит к исключению во время выполнения, если функции передан некорректный URL. Однако, вы можете использовать другую монаду, если, например, хотите проверять пользовательский ввод.

```
import Network.HTTP.Conduit
import System.Environment (getArgs)
import qualified Data.ByteString.Lazy as L
import Control.Monad.IO.Class (liftIO)

main :: IO ()
main = do
  args <- getArgs
  case args of
    [urlString] ->
      case parseUrl urlString of
        Nothing -> putStrLn "Извините, некорректный URL"
        Just req -> withManager $ \manager -> do
          res <- httpLbs req manager
          liftIO $ L.putStr $ responseBody res
        _ -> putStrLn "Извините, передавайте, пожалуйста, только один URL"
```

26/response-body.hs

Тип `Request` абстрактный, так что `http-conduit` может добавлять новые настройки в будущем, не нарушая API (см. приложение Типы для настроек для более подробной информации). Чтобы изменить значения отдельных полей, используйте `record-нотацию`². К примеру, модификация нашей программы, которая отправляет HEAD запросы и печатает заголовки ответа, могла бы выглядеть так:

```
{-# LANGUAGE OverloadedStrings #-}
```

²Запись вида `req { method = "HEAD" }`

```

import Network.HTTP.Conduit
import System.Environment (getArgs)
import qualified Data.ByteString.Lazy as L
import Control.Monad.IO.Class (liftIO)

main :: IO ()
main = do
  args <- getArgs
  case args of
    [urlString] ->
      case parseUrl urlString of
        Nothing -> putStrLn "Извините, некорректный URL"
        Just req -> withManager $ \manager -> do
          let reqHead = req { method = "HEAD" }
              res <- http reqHead manager
              liftIO $ do
                print $ responseStatus res
                mapM_ print $ responseHeaders res
          _ -> putStrLn "Извините, передавайте, пожалуйста, только
один URL"

```

26/response-head.hs

API предоставляет целый ряд различных настроек. Вот те, на которые стоит обратить внимание:

proxy

Позволяет отправлять запросы через указанный прокси-сервер;

redirectCount

Указывает количество переадресаций, по которым следовать. По умолчанию — 10;

checkStatus

Проверка кода статуса ответа. По умолчанию, бросает исключение для любого кода вне диапазона 200-х значений;

requestBody

Тело запроса для отправки. Удостоверьтесь, что так же обновили значение `method`. Для общего случая кодирования данных в URL можно использовать функцию `urlEncodedBody`.

E.4. Manager

Менеджер соединений позволяет повторно использовать сетевые соединения. При выполнении многочисленных запросов к единственному серверу (например, обращаясь к Amazon S3), это может быть критично для создания эффективного кода. Мене-

джер будет отслеживать множественные соединения к указанному серверу (в том числе учитывая номер порта и SSL), автоматически собирая неиспользуемые соединения при необходимости. Когда вы делаете запрос, `http-conduit` сначала пытается проверить наличие существующего соединения. Когда работа с соединением завершена (в случае если сервер позволяет поддерживать открытое соединение (`keep-alive`)), оно возвращается менеджеру. Если что-нибудь пошло не так, соединение закрывается.

Чтобы предохранить наш код от исключений, мы используем преобразователь монад `ResourceT`. Для вас это означает, что ваш код должен быть завёрнут внутрь вызова `runResourceT`, явно или неявно, и код внутри этого блока должен использовать `liftIO` для выполнения обычных действий ввода/вывода (IO).

Есть два способа получить значение типа `Manager`. `newManager` возвращает нового менеджера, который не будет закрыт автоматически (вы можете использовать `closeManager` для ручного закрытия), тогда как `withManager` откроет новый блок `ResourceT`, даст возможность использовать менеджера и затем автоматически закроет `ResourceT`, когда вы закончите. Если вы хотите использовать `ResourceT` для всего приложения, и нет необходимости его закрывать, вам, вероятнее всего, следует использовать `newManager`.

Обратите внимание: очевидно, что не следует создавать нового менеджера соединений для каждого запроса; это полностью противоречило бы его назначению. Вместо этого `Manager` следует создавать как можно раньше и затем переиспользовать.

E.5. Response

Тип данных `Response` состоит из трёх информационных блоков: кода статуса, заголовка ответа и тела ответа. Первые два понятны; давайте обсудим тело.

У типа данных `Response` есть типовая переменная, чтобы тело запроса могло иметь любой тип. Если вы хотите использовать потоковый интерфейс `http-conduit`, вам нужен тип `Source`. Для простого интерфейса это будет ленивая `ByteString`. Обращаю внимание, что, хотя мы и используем ленивую `ByteString`, *весь ответ хранится в памяти*. Другими словами, мы не используем ленивый ввод/вывод в этом пакете.

Пакет <code>conduit</code> предоставляет ленивую реализацию модуля, которая позволит вам прочитать это значение лениво, но, как и любой ленивый ввод/вывод, это не совсем безопасно и, определённо, недетерминировано. Однако если надо, вы можете его использовать.
--

E.6. http и httpLbs

Давайте соберём всё вместе. Функция `http` даёт вам доступ к потоковому интерфейсу (т.е., она возвращает `Response`, использующий `BufferedSource`), тогда как `httpLbs` возвращает ленивую `ByteString`. В обоих случаях возвращаемые значения находятся в преобразователе `ResourceT`, так что они получают доступ к `Manager` и корректную работу с соединениями в случае возникновения исключений.

Если вы хотите проигнорировать остаток большого тела запроса, этого можно добиться соединившись со стоком `sinkNull`. Используемое сетевое соединение будет автоматически закрыто, что позволит избежать чтения большого тела запроса по сети.

Приложение F

Пакет `xml-conduit`

Многих разработчиков трясёт от одной только мысли о работе с XML файлами. XML имеет репутацию формата, имеющего излишне усложнённую модель данных, запутанные библиотеки и целые слои сложности, находящиеся между разработчиком и его целью. Но я смею заверить, что многое из этого относится скорее к проблемам языков программирования и библиотек, чем к самому формату XML.

Как я уже отмечал, система типов языка Haskell позволяет нам с лёгкостью свести любую проблему к её самой базовой форме. Пакет `xml-types`¹ аккуратно преобразует модель данных XML (поддерживается работа как с целыми документами, так и с потоком данных) в простые алгебраические типы данных. Стандартные неизменяемые структуры данных языка Haskell упрощают преобразование документов, а простой набор функций делает разбор и отображение лёгкими и непринуждёнными.

Рассмотрим пакет `xml-conduit`². В нём используется множество подходов, которые Yesod в целом применяет для получения высокой производительности: пакеты `blaze-builder`³, `text`⁴, `conduit`⁵ и `attoparsec`⁶. С точки зрения пользователя он предоставляет всё, начиная с простейших API (`readFile/writeFile`) и заканчивая полным контролем над потоками событий XML.

В дополнение к `xml-conduit`, есть ещё несколько связанных пакетов, например, `xml-hamlet`⁷ и `xml2html`⁸. Мы рассмотрим и как их использовать, и когда это следует делать.

F.1. Краткое содержание

```
<document title="My_Title">
```

¹<http://hackage.haskell.org/package/xml-types>

²<http://hackage.haskell.org/package/xml-conduit>

³<http://hackage.haskell.org/package/blaze-builder>

⁴<http://hackage.haskell.org/package/text>

⁵<http://hackage.haskell.org/package/conduit>

⁶<http://hackage.haskell.org/package/attoparsec>

⁷<http://hackage.haskell.org/package/xml-hamlet>

⁸<http://hackage.haskell.org/package/xml2html>

```

<para>This is a paragraph. It has <em>emphasized</em> and
<strong>strong</strong> words.</para>
<image href="myimage.png"/>
</document>

```

Листинг F.1: Входной XML файл

```

{-# LANGUAGE QuasiQuotes #-}
{-# LANGUAGE OverloadedStrings #-}
import Prelude hiding (readFile, writeFile)
import Text.XML
import Text.Hamlet.XML
import qualified Data.Map as M

main :: IO ()
main = do
  -- readFile выбросит исключение времени исполнения для любой
  -- ошибки разбора
  -- def использует настройки по умолчанию
  Document prologue root epilogue <- readFile def "input.xml"

  -- root - это корневой элемент документа, давайте модифицируем его
  let root' = transform root

  -- А теперь запишем в файл. Давайте выведем результат с отступами
  writeFile def
    { rsPretty = True
    } "output.html" $ Document prologue root' epilogue

  -- Мы преобразуем <document> в документ XHTML
transform :: Element -> Element
transform (Element _name attrs children) = Element "html" M.empty
  [xml|
<head>
  <title>
    $maybe title <- M.lookup "title" attrs
    \#{title}
    $nothing
    Untitled Document
<body>
  $forall child <- children
    ^{goNode child}
|]

goNode :: Node -> [Node]
goNode (NodeElement e) = [NodeElement $ goElem e]

```

```

goNode (NodeContent t) = [NodeContent t]
goNode (NodeComment _) = [] -- скроем комментарии
goNode (NodeInstruction _) = [] -- и инструкции обработки тоже

-- преобразуем каждый исходный элемент в эквивалент на XHTML
goElem :: Element -> Element
goElem (Element "para" attrs children) =
  Element "p" attrs $ concatMap goNode children
goElem (Element "em" attrs children) =
  Element "i" attrs $ concatMap goNode children
goElem (Element "strong" attrs children) =
  Element "b" attrs $ concatMap goNode children
goElem (Element "image" attrs _children) =
  Element "img" (fixAttr attrs) [] -- у тэга img не может быть
  дочерних элементов
  where
    fixAttr mattrs
      | "href" `M.member` mattrs = M.delete "href" $ M.insert "
      "src" (mattrs M.! "href") mattrs
      | otherwise                = mattrs
goElem (Element name attrs children) =
  -- не знаем что делать, поэтому просто передаём как есть...
  Element name attrs $ concatMap goNode children

```

Листинг F.2: Код на Haskell

```

<?xml version="1.0" encoding="UTF-8"?>
<html>
  <head>
    <title>
      My Title
    </title>
  </head>
  <body>
    <p>
      This is a paragraph. It has
      <i>
        emphasized
      </i>
      and
      <b>
        strong
      </b>
      words.
    </p>
    

```

```
</body>
</html>
```

Листинг F.3: Выходные данные в формате XHTML

F.2. Типы

Воспользуемся восходящим подходом для анализа типов. Помимо прочего, этот раздел будет служить введением в саму модель данных XML, так что не беспокойтесь, если вы не вполне знакомы с ней.

Я думаю, первое место, где Haskell по-настоящему показывает свою силу, это тип данных `Name`⁹. Многие языки программирования (например, Java) испытывают трудности с выразительным представлением имён в XML. Проблема заключается в том, что на самом деле эти имена состоят из трёх частей: локального имени, пространства имён (необязательно) и префикса (также необязательно). Для наглядности рассмотрим следующий кусок XML-документа:

```
<no-namespace/>
<no-prefix xmlns="first-namespace" first-attr="value1"/>
<foo:with-prefix xmlns:foo="second-namespace"
  foo:second-attr="value2"/>
```

Первый тег имеет локальное имя `no-namespace`, не имеет префикса и не принадлежит какому-либо пространству имён. Второй тег (с локальным именем `no-prefix`) также не имеет префикса, но он принадлежит пространству имён `first-namespace`. Однако, атрибут `first-attr` не наследует это пространство имён: пространства имён атрибутов всегда должны точно задаваться с помощью префикса.

Пространства имён почти всегда представляют собой своего рода URI, хотя ничего подобного не требуется ни в одной спецификации.

Третий тег имеет локальное имя `with-prefix`, префикс `foo` и принадлежит пространству имён `second-namespace`. Его атрибут имеет локальное имя `second-attr`, имеет тот же префикс и принадлежит тому же пространству имён. Атрибуты `xmlns` и `xmlns:foo` являются частью спецификации пространства имён и не рассматриваются в качестве атрибутов соответствующих элементов.

Ещё раз: из чего состоит имя? Из локального имени, а также необязательных префикса и пространства имён. Похоже на подходящий случай для применения записей:

```
data Name = Name
  { nameLocalName :: Text
  , nameNamespace :: Maybe Text
  , namePrefix    :: Maybe Text
  }
```

⁹<http://hackage.haskell.org/packages/archive/xml-types/latest/doc/html/Data-XML-Types.html#t:Name>

Согласно стандарту пространств имён в XML, два имени считаются эквивалентными, если они имеют одинаковое локальное имя и принадлежат одному пространству имён. Другими словами, префикс неважен. Потому в пакете `xml-types` определены экземпляры классов `Eq` и `Ord`, игнорирующие префиксы.

Последний экземпляр класса типов, который стоит упомянуть, — это `IsString`. Было бы очень утомительно печатать `Name "p" Nothing Nothing` каждый раз, когда нам нужен новый параграф. Если вы включите расширение `OverloadedStrings`, `"p"` будет преобразовываться во всё это хозяйство самостоятельно! Кроме того, экземпляр `IsString` распознаёт так называемую нотацию Кларка, которая позволяет использовать в именах префикс с пространством имён в фигурных скобках. Другими словами:

```
"{namespace}element" == Name "element" (Just "namespace") Nothing
"element" == Name "element" Nothing Nothing
```

F.2.1. Четыре типа узлов

XML-документы представляют собой дерево вложенных узлов. По факту, существует четыре различных типа узлов — элементы, содержимое (то есть, текст), комментарии, а также инструкции по обработке (`processing instructions`).

Вероятно, вы не знакомы с последними, поскольку они используются довольно редко. Инструкции по обработке обозначаются следующим образом:

```
<?target data?>
```

Есть два удивительных факта об инструкциях по обработке:

- Инструкции по обработке не имеют атрибутов. Несмотря на то, что вам могут попасться инструкции, имеющие атрибуты, на самом деле не существует никаких правил относительно этих данных в инструкциях.
- Строка `<?xml ...?>` в начале документа не является инструкцией по обработке. Это просто начало документа (также известное, как объявление XML), и, так получилось, что оно выглядит поразительно похожим на инструкции по обработке. Разница заключается в том, что строка `<?xml ...?>` не появится в разобранных данных.

Учитывая, что каждая инструкция имеет два фрагмента текста, связанных с ней (`target` и `data`), получается очень простой тип данных:

```
data Instruction = Instruction
  { instructionTarget :: Text
  , instructionData  :: Text
  }
```

Для комментариев нет специального типа, потому что они представляют собой обычный текст. Зато содержимое куда интереснее — оно может состоять из простого текста или неразрешённых сущностей (например, `©right-statement;`). Пакет

xml-types¹⁰ оставляет эти сущности неразрешёнными во всех типах данных, чтобы полностью соответствовать спецификации. Однако, на практике может быть очень трудно работать с такими типами данных. И, в большинстве случаев, неразрешённая сущность в конечном итоге приведёт к возникновению ошибки.

По этой причине модуль Text.XML¹¹ определяет собственный набор типов данных для узлов, элементов и документов, в которых удаляются все неразрешённые сущности. Если вам нужно работать с неразрешёнными сущностями, используйте модуль Text.XML.Unresolved¹². Начиная с этого момента, мы сосредоточимся на типах данных модуля Text.XML, которые, впрочем, почти идентичны версиям из пакета xml-types.

В общем, заканчивая лирическое отступление: содержимое представляет собой обычный текст, и потому у него также нет специального типа данных. Последним типом узлов является элемент, который состоит из имени, списка атрибутов и списка дочерних узлов. Каждый атрибут состоит из двух частей — имени и значения. (В пакете xml-types значение атрибута также может содержать неразрешённые сущности.) Итак, тип Element определён следующим образом:

```
data Element = Element
  { elementName :: Name
  , elementAttributes :: Map Name Text
  , elementNodes :: [Node]
  }
```

Возникает закономерный вопрос — а как должен выглядеть тип данных Node? Вот где Haskell по-настоящему показывает себя с лучшей стороны: его типы-суммы идеально представляют модель данных XML.

```
data Node
  = NodeElement Element
  | NodeInstruction Instruction
  | NodeContent Text
  | NodeComment Text
```

F.2.2. Документы

Итак, у нас есть элементы и узлы, но как насчёт целых документов? Рассмотрим следующие типы данных:

```
data Document = Document
  { documentPrologue :: Prologue
  , documentRoot :: Element
  , documentEpilogue :: [Miscellaneous]
```

¹⁰<http://hackage.haskell.org/package/xml-types>

¹¹<http://hackage.haskell.org/packages/archive/xml-conduit/latest/doc/html/Text-XML.html>

¹²<http://hackage.haskell.org/packages/archive/xml-conduit/latest/doc/html/Text-XML-Unresolved.html>


```

    }

data Prologue = Prologue
  { prologueBefore :: [Miscellaneous]
  , prologueDoctype :: Maybe Doctype
  , prologueAfter :: [Miscellaneous]
  }

data Miscellaneous
  = MiscInstruction Instruction
  | MiscComment Text

data Doctype = Doctype
  { doctypeName :: Text
  , doctypeID :: Maybe ExternalID
  }

data ExternalID
  = SystemID Text
  | PublicID Text Text

```

В спецификации XML сказано, что документ может иметь только один корневой элемент (`documentRoot`). Он также может содержать необязательное объявление типа документа (`doctype`). И тип документа, и корневой элемент разрешается окружать комментариями и инструкциями по обработке. (Также разрешены пробелы, но они игнорируются во время разбора.)

Так что там насчёт типа документа? Он определяет корневой элемент документа, а затем, необязательно, публичный (`public`) и системный (`system`) идентификаторы. Эти идентификаторы используются для ссылок на DTD-файлы, которые предоставляют больше информации о документе (например, правила валидации, атрибуты по умолчанию, разрешения сущностей). Рассмотрим несколько примеров:

```

<!DOCTYPE root> <!-- no external identifier -->
<!DOCTYPE root SYSTEM "root.dtd"> <!-- a system identifier -->
<!DOCTYPE root PUBLIC "My_Root_Public_Identifier" "root.dtd"> <!-- 
  public identifiers have a system ID as well -->

```

Это, друзья мои, и есть вся модель данных XML. На практике, в большинстве случаев вы можете просто игнорировать тип данных `Document` и переходить сразу к `documentRoot`.

F.2.3. События

В дополнение к API для работы с документами пакет `xml-types` также определяет тип данных `Event`¹³. Он может быть использован для конструирования поточных инструментов, которые могут потреблять намного меньше оперативной памяти при решении определённых задач обработки (например, добавления нового атрибута всем элементам). Сейчас мы не будем рассматривать соответствующий API, однако, если вы посмотрите на него после прочтения этой главы, он наверняка покажется вам очень знакомым.

Вы можете найти пример использования «поточного» API в главе 21, посвящённой работе со Sphinx.

F.3. Модуль Text.XML

Рекомендуемой точкой входа в пакет `xml-conduit` является модуль `Text.XML`¹⁴. Этот модуль экспортирует все типы данных, которые могут вам потребоваться для манипулирования XML в DOM-стиле (Document Object Model, объектная модель документа), а также предоставляет несколько подходов к разбору и рендеренгу XML-документов. Начнём с простого:

```
readFile :: ParseSettings -> FilePath -> IO Document
writeFile :: RenderSettings -> FilePath -> Document -> IO ()
```

Здесь вы видите типы данных `ParseSettings` и `RenderSettings`. Вы можете использовать их для изменения поведения парсера или рендерера, например, добавления сущностей или включения форматированного (то есть, с отступами) вывода. Оба типа являются экземплярами класса типов `Default`¹⁵, так что вы можете просто использовать функцию `def` в тех случаях, когда требуется предоставить значение одного из них. Собственно, так мы и собираемся поступать в оставшейся части этой главы. Дополнительную информацию вы можете найти в документации на `Hackage`.

Следует также отметить, что в дополнение к API для работы с файлами также имеются API для работы с текстом и байтовыми строками (`bytestring`). В функциях для работы с байтовыми строками реализовано интеллектуальное определение кодировки. Поддерживаются кодировки UTF-8, UTF-16 и UTF-32, с прямым (`little endian`) и обратным (`big endian`) порядком байт, как с BOM (`Byte-Order Marker`), так и без него. Весь вывод генерируется в UTF-8.

Для сложных выборок данных по XML-документам мы рекомендуем использовать API более высокого уровня для работы с курсорами. Стандартный API модуля `Text.XML` не только формирует базис для этого более высокого уровня. Он также предоставляет отличный API для простого преобразования и генерации XML. Пример его использования вы можете найти в разделе «Краткое содержание» этой главы.

¹³<http://hackage.haskell.org/packages/archive/xml-types/latest/doc/html/Data-XML-Types.html#t:Event>

¹⁴<http://hackage.haskell.org/packages/archive/xml-conduit/latest/doc/html/Text-XML.html>

¹⁵<http://hackage.haskell.org/packages/archive/data-default/latest/doc/html/Data-Default.html#t:Default>

F.3.1. Замечание относительно путей к файлам

В приведённых выше сигнатурах функций вы видели тип `FilePath`. Однако *это не Prelude.FilePath*. В модуле `Prelude` определяется синоним `type FilePath = [Char]`. К сожалению, данный подход имеет множество ограничений, включая неопределённость кодировки имени файла, а также возможность использования различных символов в качестве разделителей в пути.

Вместо этого в `xml-conduit` используется пакет `system-filepath`¹⁶, в котором определяется абстрактный тип `FilePath`. Я лично нахожу такой подход более удобным. Пакет очень прост в использовании, так что здесь я не буду останавливаться на деталях. Вместо этого я приведу лишь краткие пояснения относительно его использования:

- Поскольку `FilePath` является экземпляром класса `IsString`, вы можете вводить обычные строки и они будут интерпретированы правильно, если активировано расширение `OverloadedStrings`. (Я настоятельно рекомендую использовать его в любом случае, поскольку это делает работу со значениями типа `Text` намного приятнее.)
- Если вам требуется явное преобразование в или из `FilePath` модуля `Prelude`, вы должны использовать функции `encodeString`¹⁷ или `decodeString`¹⁸, соответственно. Эти функции корректно обрабатывают кодировку пути к файлу.
- Вместо того, чтобы вручную соединять имена директорий, имена файлов и их расширения, используйте операторы из модуля `Filesystem.Path.CurrentOS`, например `myfolder </> filename <.> extension`.

F.4. Курсоры

Допустим, вы хотите получить заголовок (`title`) из XHTML-документа. Вы можете сделать это с помощью интерфейса модуля `Text.XML`, который мы только что изучили, используя стандартное сопоставление с образцом потомков элементов. Но работа над программой с использованием этого подхода очень быстро станет утомительной. Вероятно, золотым стандартом для такого типа поиска является `XPath`, который позволяет вам обращаться к элементам, используя пути типа `/html/head/title`. Именно `XPath` вдохновил дизайн комбинаторов из модуля `Text.XML.Cursor`¹⁹.

Курсор представляет собой узел, который помнит своё положение в дереве; он может перемещаться вверх, в сторону или вниз. (Это реализовано с помощью приёма "завязывание узлов"²⁰.) Есть две функции, позволяющие преобразовывать типы модуля `Text.XML` в курсоры — `fromDocument` и `fromNode`.

¹⁶<http://hackage.haskell.org/package/system-filepath>

¹⁷<http://hackage.haskell.org/packages/archive/system-filepath/latest/doc/html/Filesystem-Path-CurrentOS.html#v:encodeString>

¹⁸<http://hackage.haskell.org/packages/archive/system-filepath/latest/doc/html/Filesystem-Path-CurrentOS.html#v:decodeString>

¹⁹<http://hackage.haskell.org/packages/archive/xml-conduit/latest/doc/html/Text-XML-Cursor.html>

²⁰http://www.haskell.org/haskellwiki/Tying_the_Knot

Также имеется концепция оси (axis), определённой в виде `type Axis = Cursor [] -> [Cursor]`. Проще всего понять эту концепцию на примере. Функция `child` возвращает список из нуля или более курсоров, которые являются потомками текущего курсора. Функция `parent` возвращает единственный родительский курсор входного курсора или пустой список для корневого элемента. И так далее.

Некоторые оси используют предикаты. Функция `element` обычно используется для фильтрации элементов по имени. Например, `element "title"` вернёт входной элемент только в том случае, если он имеет имя "title", иначе будет возвращён пустой список.

Ещё одна функция, которая не вполне является осью, это `content :: Cursor [] -> [Text]`. Для всех узлов с неким содержимым она возвращает содержимый текст, иначе возвращается пустой список.

Благодаря тому, что списки являются монадами, не составляет труда объединить всё вышесказанное воедино. Например, следующая программа предназначена для поиска заголовка XHTML-документа:

```
{-# LANGUAGE OverloadedStrings #-}
import Prelude hiding (readFile)
import Text.XML
import Text.XML.Cursor
import qualified Data.Text as T

main :: IO ()
main = do
  doc <- readFile def "test.xml"
  let cursor = fromDocument doc
      print $ T.concat $
          child cursor >>= element "head" >>= child
              >>= element "title" >>= descendant >>= []
      content
```

27/cursor.hs

В переводе на русский это значит:

- Найти все дочерние узлы корневого элемента.
- Отфильтровать элементы, оставив лишь элементы с именем «head».
- Найти всех потомков (child) элементов, полученных на предыдущем шаге.
- Отфильтровать элементы, оставив лишь элементы с именем «title».
- Найти всех наследников (descendant) полученных элементов. (Наследник — это потомок или наследник потомка. Да, это рекурсивное определение.)
- Оставить только текстовые узлы.

Таким образом для входного документа:

```
<html>
  <head>
    <title>My <b>Title</b></title>
  </head>
  <body>
    <p>Foo bar baz</p>
  </body>
</html>
```

Мы получим в результате «My Title». Это всё, конечно, здорово и замечательно, но вообще-то тут намного больше кода, чем в случае использования XPath. Для борьбы с этой многословностью Aristid Breitkreuz добавил в модуль Cursor набор операторов для обработки большинства случаев. С их помощью мы можем переписать наш пример следующим образом:

```
{-# LANGUAGE OverloadedStrings #-}
import Prelude hiding (readFile)
import Text.XML
import Text.XML.Cursor
import qualified Data.Text as T

main :: IO ()
main = do
  doc <- readFile def "test.xml"
  let cursor = fromDocument doc
      print $ T.concat $
          cursor $/ element "head" &/ element "title" &// content
```

27/cursor-operator.hs

Оператор `$/` применяет ось справа к потомкам (children) курсора слева. Оператор `&/` практически идентичен, только используется он для комбинирования двух осей. Это общее правило в модуле `Text.XML.Cursor`: операторы, начинающиеся со знака `$`, напрямую применяют ось, а начинающиеся со знака `&` объединяют две оси. Оператор `&//` используется для применения оси ко всем наследникам (descendants).

Рассмотрим более сложный (или более надуманный?) пример. Имеется следующий документ:

```
<html>
  <head>
    <title>Headings</title>
  </head>
  <body>
    <hgroup>
      <h1>Heading 1 foo</h1>
      <h2 class="foo">Heading 2 foo</h2>
```

```

    </hgroup>
    <hgroup>
      <h1>Heading 1 bar</h1>
      <h2 class="bar">Heading 2 bar</h2>
    </hgroup>
  </body>
</html>

```

Мы хотим получить содержимое всех тегов `h1`, которые предшествуют тегу `h2` с атрибутом `class`, имеющим значение «bar». Для выполнения этого запутанного поиска мы можем написать:

```

{-# LANGUAGE OverloadedStrings #-}
import Prelude hiding (readFile)
import Text.XML
import Text.XML.Cursor
import qualified Data.Text as T

main :: IO ()
main = do
  doc <- readFile def "test2.xml"
  let cursor = fromDocument doc
      print $ T.concat $
          cursor $// element "h2"
              >=> attributeIs "class" "bar"
              >=> precedingSibling
              >=> element "h1"
              &// content

```

27/cursor-h1.hs

Давайте попробуем разобраться, что здесь происходит. Сначала мы получаем все элементы `h2` в документе. (Оператор `$//` получает всех наследников корневого элемента.) Из них мы оставляем только те, что имеют атрибут `class` со значением «bar». Оператор `>=>` на самом деле является стандартным оператором из модуля `Control.Monad`²¹; вот ещё одно преимущество того, что списки являются монадами. Функция `precedingSibling` находит все узлы, что идут перед заданным *u* и имеют с ним общего родителя. (Имеется также ось `preceding`, которая принимает все предыдущие элементы в дереве.) Затем мы просто берём все элементы `h1` и получаем их содержимое.

Эквивалентный XPath, для сравнения, будет `//h2[@class = 'bar']/preceding-sibling::h1//text()`.

Хоть API курсоров и уступает XPath в краткости, зато он является стандартным кодом на Haskell и обеспечивает безопасность типов.

²¹<http://hackage.haskell.org/packages/archive/base/latest/doc/html/Control-Monad.html>

F.5. Пакет xml-hamlet

Благодаря простоте системы типов языка Haskell, создание XML-документов с помощью API модуля `Text.XML` является крайне простым, хотя и несколько многословным. Следующий код:

```
{-# LANGUAGE OverloadedStrings #-}
import Text.XML
import Prelude hiding (writeFile)
import Data.Map (empty)

main :: IO ()
main =
  writeFile def "test3.xml" $ Document (Prologue [] Nothing []) []
  root []
  where
    root = Element "html" empty
      [ NodeElement $ Element "head" empty
        [ NodeElement $ Element "title" empty
          [ NodeContent "My"
            , NodeElement $ Element "b" empty
              [ NodeContent "Title"
                ]
            ]
          ]
        ]
      , NodeElement $ Element "body" empty
        [ NodeElement $ Element "p" empty
          [ NodeContent "foo_bar_baz"
            ]
          ]
        ]
      ]
```

27/xml.hs

... генерирует:

```
<?xml version="1.0" encoding="UTF-8"?>
<html><head><title>My <b>Title</b></title></head><body><p>foo bar
  baz</p></body></html>
```

Это во много раз проще, чем использовать императивный API с изменяемыми состояниями (как, кхм, в Java), но всё же далеко от идеала, и к тому же делает неясными наши настоящие намерения. Для исправления ситуации у нас есть пакет `xml-hamlet`²², который использует квазицитирование и позволяет вводить XML, используя естественный синтаксис. Например, предыдущий пример может быть переписан так:

²²<http://hackage.haskell.org/package/xml-hamlet>

```

{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE QuasiQuotes #-}
import Text.XML
import Text.Hamlet.XML
import Prelude hiding (writeFile)
import Data.Map (empty)

main :: IO ()
main =
    writeFile def "test3.xml" $ Document (Prologue [] Nothing []) []
    root []
  where
    root = Element "html" empty [xml]
<head>
  <title>
    My #
    <b>Title
<body>
  <p>foo bar baz
|]

```

27/xml-hamlet.hs

Тут нужно обратить внимание на следующее:

- Синтаксис практически идентичен Hamlet, если не считать отсутствия интерполяции URL (@...). Таким образом:
 - Закрывающие теги не нужны.
 - Имеет место чувствительность к пробелам.
 - Если вам нужны пробелы в конце строки, используйте на конце #. В начале строки используйте обратную косую черту.
- XML-интерполяция возвращает список узлов. Поэтому вам потребуется обращивать результат в обычную конструкцию из Document и корневого Element.
- Нет поддержки специальных атрибутов **.class** и **#id**.

Как и в обычном Hamlet, вы можете использовать интерполяцию переменных и управляющие структуры. Рассмотрим это на чуть более сложном примере:

```

{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE QuasiQuotes #-}
import Text.XML
import Text.Hamlet.XML
import Prelude hiding (writeFile)
import Data.Text (Text, pack)

```



```

import Data.Map (empty)

data Person = Person
  { personName :: Text
  , personAge  :: Int
  }

people :: [Person]
people =
  [ Person "Michael" 26
  , Person "Miriam"  25
  , Person "Eliezer"  3
  , Person "Gavriella" 1
  ]

main :: IO ()
main =
  writeFile def "people.xml" $ Document (Prologue [] Nothing []) []
  root []
  where
    root = Element "html" empty [xml|
<head>
  <title>Some People
<body>
  <h1>Some People
  $if null people
    <p>There are no people.
  $else
    <dl>
      $forall person <- people
        ^{personNodes person}
|]

personNodes :: Person -> [Node]
personNodes person = [xml|
<dt>#{personName person}
<dd>#{pack $ show $ personAge person}
|]

```

27/xml-hamlet-vars.hs

Ещё пара моментов:

- Saret-интерполяция ($\wedge\{\dots\}$) принимает список узлов и, следовательно, может с лёгкостью включать другие xml-цитаты.
- В отличие от Hamlet, hash-интерполяция ($\#\{\dots\}$) не является полиморфной и

может принимать *только* значения типа `Text`.

F.6. Пакет `xml2html`

До сих пор в этой главе наши примеры вращались вокруг XHTML. Я делал это по той простой причине, что XHTML, скорее всего, окажется наиболее знакомой формой XML для большинства читателей. Но в этом есть и отрицательный момент, который следует признать: не всякий XHTML является корректным HTML. Существуют следующие расхождения:

- Некоторые «пустые» HTML-теги (например, `img`, `br`) не обязаны иметь парные закрывающие теги, и вообще-то не имеют права их иметь.
- HTML не понимает самозакрывающиеся теги, поэтому `<script></script>` и `<script/>` означают разные вещи.
- Объединяя два предыдущих пункта: «пустые» теги могут быть самозакрывающимися, однако это ничего не значит для браузера.
- Во избежание недоразумений, HTML-документы следует начинать с инструкции `DOCTYPE`.
- XML-объявление `<?xml ... ?>` не нужно в HTML-страницах.
- В HTML нам не нужны пространства имён, в то время, как XHTML полон ими.
- Содержимое тегов `<style>` и `<script>` не должно экранироваться.

И вот здесь в игру вступает пакет `xml2html`. Он предоставляет экземпляры класса `toHtml` для типов `Node`, `Document` и `Element`. Для их использования просто импортируйте модуль `Text.XML.Xml2Html`.

```
{-# LANGUAGE OverloadedStrings, QuasiQuotes #-}
import Text.Blaze.Html (toHtml)
import Text.Blaze.Html.Renderer.String (renderHtml)
import Text.XML
import Text.Hamlet.XML
import Text.XML.Xml2Html ()
import Data.Map (empty)

main :: IO ()
main = putStr $ renderHtml $ toHtml $ Document (Prologue [] Nothing []
  []) root []

root :: Element
root = Element "html" empty [xml|
<head>
  <title>Test
```

```
<script>if (5 < 6 || 8 > 9) alert("Hello World!");  
<style>body > h1 { color: red }  
<body>  
  <h1>Hello world!  
|]
```

27/xml2html.hs

Выводит (пробелы добавлены вручную):

```
<!DOCTYPE HTML>  
<html>  
  <head>  
    <title>Test</title>  
    <script>if (5 < 6 || 8 > 9) alert("Hello World!");</script>  
    <style>body > h1 { color: red }</style>  
  </head>  
  <body>  
    <h1>Hello world!</h1>  
  </body>  
</html>
```

Приложение G

Пакет Enumerator

В связи с тем, что Yesod перешел на использование пакета conduit, эта глава больше не соответствует текущему состоянию Yesod. Она приведена здесь в качестве справки для тех, кто до сих пор используют enumerator.

Enumerator — это одна из новых концепций в Haskell предназначенная для решения проблем производства, изменения и потребления потоков данных. Enumerator считается немного пугающей концепцией. Возможно, потому что:

- Существует несколько ее реализаций, все с несколько различными подходами;
- Некоторые из этих реализаций (на мой взгляд) используют сбивающую с толку систему именования;
- Как правило, существующие руководства не нацелены непосредственно на конкретную реализацию, а больше работают над развитием интуитивного понимания, и не дают инструкций по использованию конкретной библиотеки.

Yesod использует enumerator во многих местах: в WAI/Warp для работы с телами запроса и ответа, в Persistent для получения результатов запроса к базе данных, а также в различных сторонних пакетах, таких как http-enumerator и xml-enumerator. Мы постоянно используем пакет enumerator. И этот пакет является темой данной главы.

Эта глава состоит из трех основных разделов, каждый из которых рассматривает одно из основных понятий:

- Начнем с перечисляемого (iteratee), которое является потребителем. Ему скармливаются данные с которыми он что-то делает.
- Далее мы рассмотрим эnumератор (enumerator), который является производителем. Он скармливает данные перечисляемому.
- В конце мы рассмотрим повторяемое (enumeratee). Это конвейер, который получает данные от эnumератора и в свою очередь передает их перечисляемому.

Вы можете быть удивлены, почему мы должны возиться с этим, ведь большинство из данных проблем решаются ленивым вводом/выводом. Но основная проблема с ленивым вводом/выводом — это его недетерминизм. Для получения дополнительной информации лучше обратиться к оригинальному источнику по этой теме — Олегу Киселеву^a.

^a<http://okmij.org/ftp/Streams.html#iteratee>

G.1. Iteratee

Перечисляемое является потребителем данных. Это центральный тип пакета `enumerator`, на котором основаны другие типы.

G.2. Интуитивное понимание

Предположим, что мы хотим написать функцию, которая суммирует числа в списке. Пренебрегши неинтересными деталями, такими как требованиями к памяти, вполне хорошей реализацией будет:

```
sum1 :: [Int] -> Int
sum1 [] = 0
sum1 (x:xs) = x + sum1 xs
```

Но допустим, что у нас нет списка чисел. Вместо этого, пользователь может вводить числа в командной строке, и «q» для окончания ввода. Иными словами, у нас есть такая функция:

```
getNumber :: IO (Maybe Int)
getNumber = do
  x <- readLine
  if x == "q"
    then return Nothing
    else return $ Just $ read x
```

Мы можем написать новую функцию суммирования так:

```
sum2 :: IO Int
sum2 = do
  maybeNum <- getNumber
  case maybeNum of
    Nothing -> return 0
    Just num -> do
      rest <- sum2
      return $ num + rest
```

Довольно неудобно писать две разные функции суммирования только потому, что изменился наш источник данных. В идеале, мы бы хотели немного обобщить сущности. Начнем с того, что выделим сходство между этими двумя функциями: они обе возвращают значения только тогда, когда чисел больше нет. В случае `sum1` мы проверяем, пустой ли список, в `sum2` проверяем на `Nothing`.

G.3. Тип данных Stream

Первый тип данных, определенный в пакете `enumerator`, это:

```
data Stream a = Chunks [a] | EOF
```

Конструктор `EOF` означает, что больше нет доступных данных. Конструктор `Chunks` просто позволяет нам объединять несколько частей данных вместе для повышения эффективности. Теперь мы можем переписать `sum2`, используя тип данных `Stream`:

```
getNumber2 :: IO (Stream Int)
getNumber2 = do
  maybeNum <- getNumber -- используем оригинальную функцию getNumber
  case maybeNum of
    Nothing -> return EOF
    Just num -> return $ Chunks [num]

sum3 :: IO Int
sum3 = do
  stream <- getNumber2
  case stream of
    EOF -> return 0
    Chunks nums -> do
      let nums' = sum nums
      rest <- sum3
      return $ nums' + rest
```

Нельзя сказать, что этот способ гораздо лучше, чем `sum2`, но по крайней мере он демонстрирует использование типа данных `Stream`. Здесь проблема в том, что мы по-прежнему явно ссылаемся на функцию `getNumber2`, жестко задавая источник данных.

Одно из возможных решений — сделать источник данных аргументом функции суммирования, а именно:

```
sum4 :: IO (Stream Int) -> IO Int
sum4 getNum = do
  stream <- getNum
  case stream of
    EOF -> return 0
    Chunks nums -> do
      let nums' = sum nums
```

```
rest <- sum4 getNum
return $ nums' + rest
```

Это все хорошо, но давайте представим, что мы хотим иметь два источника данных для суммирования: значения, которые пользователь вводит в командной строке, и числа, которые мы читаем через соединение HTTP. Здесь перед нами встает задача контроля: `sum4` управляет процессом вызова `getNum`. Это модель вытягивания данных (pull data model). В эnumераторах используется модель инверсии управления, или же модель проталкивания (inversion of control/push model), назначая ответственным эnumератор. Это позволяет делать классные вещи, наподобии потребления данных из нескольких источников. Также становится проще писать эnumераторы, которые правильно работают с выделением ресурсов.

G.4. Тип данных Step

Итак, нам нужен новый тип данных, которые будут представлять состояние нашей операции суммирования. Мы позволим операции находиться в одном из трех состояний:

- В ожидании новых данных.
- С уже подсчитанным результатом.
- Для удобства, мы также введем состояние ошибки. Это не обязательно (к примеру, это состояние может быть смоделировано выбором монады вроде `EitherT`), но так проще.

Как вы могли догадаться, эти состояния будут соответствовать трем конструкторам типа данных `Step`. Состояние ошибки моделируется с помощью `Error SomeException`, построенном на расширяемой системе исключений `Haskell`. Вот уже рассчитанный конструктор:

```
Yield b (Stream a)
```

Здесь, `a` это вход для нашего перечисляемого, `a b` — это выход. Этот конструктор позволяет одновременно произвести результат и сохранить «остатки» входных данных для другого перечисляемого, который может работать после нас. (Это не происходит в случае функции суммы, которая всегда потребляет все входные данные, но мы увидим и другие примеры, которые не потребляют все данные.)

Теперь вопрос заключается в том, как представлять состояние перечисляемого, которое ждет дополнительных данных. Вы можете захотеть объявить некоторый тип данных, чтобы представлять внутреннее состояние и передавать его каким-то образом. Но это не так, как оно сделано: на самом деле, мы просто используем функцию (весьма в стиле `Haskell`, не так ли?):

```
Continue (Stream a -> Iteratee a m b)
```

Эврика! Мы наконец увидели тип данных Iteratee! В действительности, Iteratee очень скучный тип, который присутствует только для того, чтобы мы могли объявить крутые экземпляры (например, Monad) для наших функций. Iteratee определяется как:

```
newtype Iteratee a m b = Iteratee (m (Step a m b))
```

А целиком тип данных Step выглядит так:

```
data Step a m b = Error SomeException
                | Yield b (Stream a)
                | Continue (Stream a -> Iteratee a m b)
```

Это важно: Iteratee является всего лишь newtype-оберткой вокруг типа Step, завернутого в монаду. Просто имейте это в виду, когда будете смотреть на определения в пакете enumerator. Таким образом, мы можем представлять себе конструктор Continue как:

```
Continue (Stream a -> m (Step a m b))
```

К этому определению гораздо легче подойти: функция берет некоторое количество входных данных и возвращает новое состояние перечисляемого. Давайте посмотрим, как будет выглядеть наша функция суммирования, использующая тип данных Step:

```
sum5 :: Monad m => Step Int m Int -- входные данные типа Int, любая
-- монада, результат типа Int
sum5 =
  Continue $ go 0 -- обычный подход, всегда начинайте с Continue
where
  go :: Monad m => Int -> Stream Int -> Iteratee Int m Int
  -- Складываем новые входные данные с текущей суммой и создаем
  -- новый Continue
  go runningSum (Chunks nums) = do
    let runningSum' = runningSum + sum nums
    -- Следующая строчка *страшная*, но хорошо, что есть
    -- вспомогательные функции, что бы сделать это яснее.
    -- Подробнее об этом ниже.
    Iteratee $ return $ Continue $ go runningSum'
  -- Произвести конечный результат
  go runningSum EOF = Iteratee $ return $ Yield runningSum EOF
```

Для того, чтобы запустить этот код, вы можете использовать `run_` `$ enumList 8 [1 .. 10] sum5`. Но здесь используется черная магия Enumerator, которую мы рассмотрим позже.

Первая «настоящая» строка (`Continue $ go 0`) приводит наше перечисляемое в начальное состояние. Как и во всех других функциях суммирования, мы должны явно заявить, что начинаем с нуля. Настоящая рабочая лошадка это функция `go`. Обратите

внимание, как мы передаем состояние перечисляемого в качестве первого аргумента `go`: это тоже очень распространенный подход в программировании перечисляемого.

Мы должны обрабатывать два разных случая: когда получаем EOF, функция `go` должна произвести значение `Yield`. (Она также может произвести `Yield`, но определенно не `Continue`.) В этом случае, мы просто возвращаем текущую сумму и говорим, что данных не осталось. Когда же мы получаем некоторые входные данные через `Chunks`, мы просто складываем их и создаем новое `Continue`, основанное на той же функции `go`.

Давайте теперь поработаем над тем, чтобы сделать эту функцию немного красивее с помощью встроенных вспомогательных функций. Конструкция `Iteratee . return` достаточно обычна, чтобы оправдать вспомогательную функцию:

```
returnI :: Monad m => Step a m b -> Iteratee a m b
returnI = Iteratee . return
```

К примеру:

```
go runningSum EOF = Iteratee $ return $ Yield runningSum EOF
```

становится:

```
go runningSum EOF = returnI $ Yield runningSum EOF
```

Но даже это достаточно обычно, что бы ввести еще одну вспомогательную функцию:

```
yield :: Monad m => b -> Stream a -> Iteratee a m b
yield x chunk = returnI $ Yield x chunk
```

после чего наша строка становится:

```
go runningSum EOF = yield runningSum EOF
```

Таким же образом:

```
Iteratee $ return $ Continue $ go runningSum'
```

становится:

```
continue $ go runningSum'
```

G.5. Экземпляр Monad для Iteratee

Это все очень хорошо: теперь у нас есть перечисляемое, которому могут скормиться числа из любой монады, и он сможет их складывать. Он может даже брать входные данные из различных источников, и складывать их вместе. (Кстати, я на самом деле еще не показал вам, как скормить ему эти числа: об этом во второй части,

про эnumераторы) Но давайте будем честными: `sum5` является уродливой функцией. Разве нельзя написать как-нибудь попроще?

На самом деле, можно. Помните, я уже говорил, что `Iteratee` существуют просто чтобы облегчить определение экземпляров классов типов? Это включает в себя и экземпляр монады. Вы можете посмотреть код, чтобы увидеть, как этот экземпляр определен, а здесь мы рассмотрим, как его можно использовать:

```
sum6 :: Monad m => Iteratee Int m Int
sum6 = do
  maybeNum <- head -- это не функция head из Prelude!
  case maybeNum of
    Nothing -> return 0
    Just i -> do
      rest <- sum6
      return $ i + rest
```

Функция `head` не из `Prelude`, она из модуля `Data.Enumerator`. Вот ее сигнатура типа:

```
head :: Monad m => Iteratee a m (Maybe a)
```

Она означает «дай мне следующую часть входных данных, если они есть». Чуть позже мы рассмотрим эту функцию более подробно.

Сравните код `sum6` и `sum2`: они удивительно похожи. Часто можно строить более сложные перечисляемые с помощью простых перечисляемых и экземпляра `Monad` для `Iteratee`.

G.6. Чередующийся ввод/вывод

Отлично, давайте теперь рассмотрим совершенно другую задачу. Допустим, мы хотим получить на вход некоторое количество строк и вывести их по одной на экран. Одним из подходов будет использование ленивого ввода/вывода:

```
lazyIO :: IO ()
lazyIO = do
  s <- lines `fmap` getContents
  mapM_ putStrLn s
```

Но это решение имеет два недостатка:

- Оно привязано к одному источнику данных — стандартному вводу. Это ограничение можно обойти аргументом, указывающим источник данных.
- Но, допустим, в качестве источника данных используется некоторый ограниченный ресурс (представьте себе дескрипторы файлов на очень загруженном веб-сервере). С ленивым вводом/выводом у нас нет никаких гарантий относительно того, когда эти дескрипторы файлов будут освобождены.

Давайте рассмотрим, как написать то же самое, используя новый высокоуровневый монадический подход с использованием перечисляемого:

```
interleaved :: MonadIO m => Iteratee String m ()
interleaved = do
  maybeLine <- head
  case maybeLine of
    Nothing -> return ()
    Just line -> do
      liftIO $ putStrLn line
      interleaved
```

Функция `liftIO` — из пакета `transformers`, она просто продвигает действие в монаде IO в любое произвольное действие в `MonadIO`. Обратите внимание, что мы не отслеживаем состояние в этом перечисляемом: мы не заботимся о результате, только о побочных эффектах.

G.7. Реализация head

В качестве последнего примера, давайте попробуем реализовать функцию `head`.

```
head' :: Monad m => Iteratee a m (Maybe a)
head' =
  continue go
  where
    go (Chunks []) = continue go
    go (Chunks (x:xs)) = yield (Just x) (Chunks xs)
    go EOF = yield Nothing EOF
```

Как и наша функция `sum6`, она тоже оборачивает внутреннюю функцию `go`, используя `continue`. Однако теперь у нас есть три варианта для функции `go`. Первый обрабатывает случай пустых блоков (`Chunks []`). Цитируя документацию `enumerator`:

(`Chunks []`) используется для обозначения того, что поток все еще активный, но в настоящее время нет доступных данных. `Iteratee` должен игнорировать пустые блоки.

Второй вариант обрабатывает случай, когда нам даны некоторые данные. В этом случае мы возвращаем первый элемент списка, и остатки. Третий пункт обрабатывает конец ввода, возвращая `Nothing`.

G.8. Enumerator

В то время как `Iteratee` является потребителем, `Enumerator` является производителем. На высоком уровне, `Enumerator` является в действительности трансформатором `Iteratee`, переводя `Iteratee` из одного состояния в другое. Давайте посмотрим, как это работает.

G.9. Извлечение значений

Мы уже написали несколько перечисляемых, но мы до сих пор не знаем, как из них извлекать значения. Для начала вспомним, что `Iteratee` это просто `newtype`-обертка вокруг `Step`:

```
newtype Iteratee a m b = Iteratee { runIteratee :: m (Step a m b) }
```

Давайте развернем `Iteratee`, чтобы иметь дело со значением `Step` внутри. Также вспомним, что для типа `Step` есть три конструктора: `Continue`, `Yield` и `Error`. Мы будем обрабатывать конструктор `Error`, возвращая наш результат в `Either`. `Yield` уже содержит данные, которые мы ищем.

Немного сложнее с `Continue`: в этом случае у нас есть перечисляемое, которое ожидает больше данных. Здесь на помощь приходит конструктор `EOF`: это наш простой способ сказать перечисляемому закончить то, что оно делает, и продолжить работу. Если вы помните, выше я сказал, что правильно реализованное перечисляемое никогда не вернет `Continue` после получения `EOF`. Сейчас мы увидим, почему:

```
extract :: Monad m => Iteratee a m b -> m (Either SomeException b)
extract (Iteratee mstep) = do
  step <- mstep
  case step of
    Continue k -> do
      let Iteratee mstep' = k EOF
          step' <- mstep'
          case step' of
            Continue _ -> error "Misbehaving_iteeree"
            Yield b _ -> return $ Right b
            Error e -> return $ Left e
      Yield b _ -> return $ Right b
    Error e -> return $ Left e
```

К счастью, вам не нужно переопределять все это самостоятельно: `enumerator` уже содержит функции `run` и `run_`. Давайте пойдем дальше и воспользуемся этим в нашей `sum6` функции:

```
main = run_ sum6 >>= print
```

Если вы запустите этот код, результатом будет `0`. Это подчеркивает один важный момент: перечисляемое отвечает не только за обработку входных данных, но и за состояние обработки. В этом случае, мы не сделали ничего, чтобы изменить начальное состояние `sum6`, в связи с чем начальное значение не изменилось.

Для аналогии представьте перечисляемое как автомат. Когда вы вводите в него данные, вы изменяете его внутреннее состояние, но вы не видите всех этих изменений снаружи. Когда же вы закончили ввод, вы нажимаете кнопку, и он выдает результат. Если вы не вводили никаких данных, ваш результат будет в начальном состоянии.

G.10. Добавление данных

Давайте предположим, что сейчас нам действительно нужно найти сумму нескольких чисел. Например, от 1 до 10. Мы должны каким-то образом скормить их нашему перечисляемому `sum6`. Для того, чтобы как-то подойти к этому, нам опять нужно развернуть наш `Iteratee` и работать со значением `Step` напрямую.

В данном случае мы точно знаем, что конструктор `Step`, который мы использовали — это `Continue`, так что можно безопасно записать нашу функцию как:

```
sum7 :: Monad m => Iteratee Int m Int
sum7 = Iteratee $ do
  Continue k <- runIteratee sum6
  runIteratee $ k $ Chunks [1..10]
```

Но в общем случае мы не будем знать, какой конструктор мы встретим, и должны как положено обрабатывать `Continue`, `Yield` и `Error`. Мы уже знаем, что делать с `Continue`: скормить ему данные. С `Yield` и `Error`, в общем случае, правильным действием будет ничего не делать, поскольку мы уже прибыли в наше конечное состояние (это или успешный `Yield` или `Error`). Таким образом, «правильный» способ записи функции:

```
sum8 :: Monad m => Iteratee Int m Int
sum8 = Iteratee $ do
  step <- runIteratee sum6
  case step of
    Continue k -> runIteratee $ k $ Chunks [1..10]
    _ -> return step
```

G.11. Синоним типа `Enumerator`

В функциях `sum7` и `sum8` мы выполняли преобразование `Iteratee`. Но делали это в очень ограниченном виде: мы явно вписали использование оригинальной функции `Iteratee` (`sum6`). А могли бы сделать ее аргументом функции:

```
sum9 :: Monad m => Iteratee Int m Int -> Iteratee Int m Int
sum9 orig = Iteratee $ do
  step <- runIteratee orig
  case step of
    Continue k -> runIteratee $ k $ Chunks [1..10]
    _ -> return step
```

Так как мы всегда хотим развернуть значение `Iteratee`, более естественно сделать аргумент типа `Step`, а именно:

```
sum10 :: Monad m => Step Int m Int -> Iteratee Int m Int
sum10 (Continue k) = k $ Chunks [1..10]
sum10 step = returnI step
```

Эта сигнатура типа (взять Step, вернуть Iteratee) оказывается очень распространенной:

```
type Enumerator a m b = Step a m b -> Iteratee a m b
```

Значит, сигнатура типа функции sum10 является следующей:

```
sum10 :: Monad m => Enumerator Int m Int
```

Конечно, нам понадобится вспомогательная функция для связи Enumerator и Iteratee:

```
applyEnum :: Monad m => Enumerator a m b -> Iteratee a m b -> IO
  Iteratee a m b
applyEnum enum iter = Iteratee $ do
  step <- runIteratee iter
  runIteratee $ enum step
```

Позвольте мне повторить, на интуитивном уровне, Enumerator трансформирует Iteratee из его исходного состояния в новое состояние, скармливая ему данные. Для того, чтобы использовать эту функцию, мы можем написать:

```
run_ (applyEnum sum10 sum6) >>= print
```

Здесь результатом будет 55, в точности, как мы ожидали. Но теперь мы видим одно из преимуществ эnumераторов: мы можем использовать несколько источников данных. Допустим, у нас есть еще один эnumератор:

```
sum11 :: Monad m => Enumerator Int m Int
sum11 (Continue k) = k $ Chunks [11..20]
sum11 step = returnI step
```

Тогда мы можем просто применить оба эnumератора:

```
run_ (applyEnum sum11 $ applyEnum sum10 sum6) >>= print
```

И мы получим результат 210. (Да, $(1 + 20) * 10 = 210$), но не волнуйтесь, вам не нужно писать функцию applyEnum самим: enumerator предоставляет оператор \$\$, который делает то же самое. Его сигнатура типа немного страшнее, так как это обобщение applyEnum, но он работает так же, и даже делает код более читабельным:

```
run_ (sum11 $$ sum10 $$ sum6) >>= print
```

\$\$ является синонимом для ==<<, который представляет собой просто **flip** >>==. Лично я нахожу \$\$ наиболее читаемым, но для вас это может быть по-другому.

G.12. Некоторые встроенные эnumераторы

Конечно, писать целую функцию только для того, чтобы передать несколько чисел функции суммирования, кажется несколько утомительным. Мы могли бы легко сделать список аргументом функции:

```
sum12 :: Monad m => [Int] -> Enumerator Int m Int
sum12 nums (Continue k) = k $ Chunks nums
sum12 _ step = returnI step
```

Но в этой функции нет ничего, что привязывало бы нас к типу `Int`, и мы могли бы легко обобщить ее до:

```
genericSum12 :: Monad m => [a] -> Enumerator a m b
genericSum12 nums (Continue k) = k $ Chunks nums
genericSum12 _ step = returnI step
```

Оказывается, в пакете `enumerator` есть встроенная функция `enumList`, которая делает все это. Функция `enumList` также принимает аргумент типа `Integer`, который указывает максимальное количество элементов в блоке. Например, мы можем написать:

```
run_ (enumList 5 [1..30] $$ sum6) >>= print
```

(Совершенно очевидно, что результатом будет 465.) Первый аргумент `enumList` никогда не должен влиять на результат, хотя он может иметь некоторое влияние на производительность.

Модуль `Data.Enumerator` включает в себя два других эnumератора: `enumEOF` просто передает EOF в перечисляемое. `concatEnums` немного более интересный, он объединяет несколько эnumераторов воедино. Например:

```
run_ (concatEnums
      [ enumList 1 [1..10]
      , enumList 1 [11..20]
      , enumList 1 [21..30]
      ] $$ sum6) >>= print
```

Здесь мы также получаем результат 465.

G.13. Не чистые входные данные

Использование эnumераторов гораздо интереснее, если иметь дело не только с чистыми значениями. В первой части этой главы мы привели пример ввода чисел пользователем в командной строке:

```
getNumber :: IO (Maybe Int)
getNumber = do
  x <- getLine
```

```

    if x == "q"
      then return Nothing
      else return $ Just $ read x

sum2 :: IO Int
sum2 = do
  maybeNum <- getNumber
  case maybeNum of
    Nothing -> return 0
    Just num -> do
      rest <- sum2
      return $ num + rest

```

Мы говорили об этом как о модели вытягивания данных: `sum2` вытаскивал каждое значение из `getNumber`. Давайте посмотрим, сможем ли мы переписать `getNumber` так, что бы она была в роли толкателя (`pusher`), а не пассивного источника (`pullee`).

```

getNumberEnum :: MonadIO m => Enumerator Int m b
getNumberEnum (Continue k) = do
  x <- liftIO getLine
  if x == "q"
    then continue k
    else k (Chunks [read x]) >>= getNumberEnum

getNumberEnum step = returnI step

```

Во-первых, обратите внимание, что мы проверяем, какой нам передали конструктор, и только в случае `Continue` осуществляем какие-либо действия. Если это был `Continue`, мы получим строку ввода от пользователя. Если получили «q» (наш сигнал окончания ввода), то ничего не делаем. Вы, наверное, подумали, что мы должны вернуть EOF. Но если бы мы это сделали, мы бы предотвратили передачу других данных этому перечисляемому. Вместо этого, мы просто возвращаем исходное значение `Step`.

Если строка была не «q», мы превращаем ее в `Int` через `read`, создаем значение `Stream` с типом данных `Chunks`, и передаем его `k`. (Если бы мы хотели все делать правильно, мы бы проверяли, на самом ли деле `x` является целым числом и использовали бы конструктор `Either`; оставляю это вам в качестве упражнения.) А сейчас давайте посмотрим на сигнатуру типа:

```

k (Chunks [read x]) :: Iteratee Int m b

```

Если бы мы просто отбросили остаток строки, наша программа прошла бы проверку соответствия типов. Однако она читала бы только одно значение из командной строки; `==>>` `getNumberEnum` заставляет наш энумератор зацикливаться.

И последнее, что следует отметить о нашей функции: обратите внимание на `b` в сигнатуре типа.

```

getNumberEnum :: MonadIO m => Enumerator Int m b

```


Это означает, что наш `Enumerator` может скармливать `Int` любому `Iteratee`, принимающему `Int`, и не имеет значения, какой тип будет у окончательного вывода. Это обычная манера работы эnumераторов, что позволяет создавать совершенно разные перечисляемые, которые работают с одними и теми же эnumератором:

```
intsToStrings :: (Show a, Monad m) => Iteratee a m String
intsToStrings = (unlines . map show) `fmap` consume
```

И после этого обе следующие строки работают:

```
run_ (getNumberEnum $$ sum6) >>= print
run_ (getNumberEnum $$ intsToStrings) >>= print
```

G.14. Enumeratee

Заключительной частью головоломки является `Enumeratee`. Это сочетание `Iteratee` (он потребляет некоторые данные) и `Enumerator` (передает свои данные). Его можно рассматривать как трансформатор потока, или как конвейер, соединяющий `Enumerator` с `Iteratee`.

G.15. Обобщение `getNumberEnum`

Ранее мы создали функцию `getNumberEnum` с сигнатурой типа:

```
getNumberEnum :: MonadIO m => Enumerator Int m b
```

Если вы подзабыли, это означает, что `getNumberEnum` производит поток `Int`'ов. В частности, наша функция `getNumberEnum` читает строки из стандартного ввода, преобразовывает их в `Int` и скармливает их в перечисляемое. Она перестет читать строки, когда встречает «q».

Но кажется, что эта функциональность может быть полезна и вне контекста `Int`. Например, мы можем захотеть работать с оригинальными `String`, или `Bool`, или множеством других типов. Так мы можем легко определить более обобщенную функцию, которая просто не делает преобразования `String` в `Int`:

```
lineEnum :: MonadIO m => Enumerator String m b
lineEnum (Continue k) = do
  x <- liftIO getLine
  if x == "q"
  then continue k
  else k (Chunks [x]) >>= lineEnum
lineEnum step = returnI step
```

Круто, давайте используем это с нашей функции `sumIter` (я переименовал функции `sumb` из двух предыдущих частей):

```
lineEnum $$ sumIter
```

Но оказывается, что это не проходит проверку типов: `lineEnum` производит `String`, а `sumIter` хочет `Int`. Нам нужно как-то изменить одну из них.

```
sumIterString :: Monad m => Iteratee String m Int
sumIterString = Iteratee $ do
  innerStep <- runIteratee sumIter
  return $ go innerStep
where
  go :: Monad m => Step Int m Int -> Step String m Int
  go (Yield res _) = Yield res EOF
  go (Error err) = Error err
  go (Continue k) = Continue $ \strings -> Iteratee $ do
    let ints = fmap read strings :: Stream Int
        step <- runIteratee $ k ints
    return $ go step
```

То, что мы тут сделали — это обернули оригинальное перечисляемое. Как обычно, нам в первую очередь необходимо развернуть конструктор `Iteratee` и монаду, чтобы добраться до сердца значения `Step`. Как только мы получим значение `innerStep`, мы передаем его функции `go`, которая просто преобразует значения в значении `Stream`, из `String` в `Int`.

G.16. Еще большее обобщение

Конечно, было бы неплохо, если бы мы могли применить это преобразование вообще к любому перечисляемому. Для начала, давайте просто передадим внутреннее перечисляемое и отображающую функцию в качестве параметров.

```
mapIter :: Monad m => (aOut -> aIn) -> Iteratee aIn m b -> Iteratee ()
  aOut m b
mapIter f innerIter = Iteratee $ do
  innerStep <- runIteratee innerIter
  return $ go innerStep
where
  go (Yield res _) = Yield res EOF
  go (Error err) = Error err
  go (Continue k) = Continue $ \strings -> Iteratee $ do
    let ints = fmap f strings
        step <- runIteratee $ k ints
    return $ go step
```

Это можно использовать так:

```
run_ (lineEnum $$ mapIter read sumIter) >>= print
```

Здесь нет ничего особенного, это в общем-то идентично предыдущей версии. Что интересно, в пакете `enumerator` есть функция `map`, которая делает как раз это, но у нее значительно отличается сигнатура типа:

```
map :: Monad m => (a0 -> ai) -> Enumeratee a0 ai m b
```

так как:

```
type Enumeratee aOut aIn m b = Step aIn m b -> Iteratee aOut m (Step aIn m b)
```

это эквивалентно:

```
map :: Monad m => (aOut -> aIn) -> Step aIn m b -> Iteratee aOut m (Step aIn m b)
```

К чему все это дополнительное усложнение сигнатуры типа? Это не является необходимым для самой `map`, но это необходимо для целой кучи других подобных функций. Но давайте на секунду сосредоточимся на `map`, чтобы мы не заблудились: первый аргумент — это та же старая отображающая функция, что была раньше. Вторым аргументом является значение `Step`. Это не так уж удивительно: в нашей `mapIter` мы взяли `Iteratee` с теми же параметрами, и внутренне просто развернул его в `Step`.

Но что происходит с возвращаемым значением? Помня о значении всех этих типов данных, это `Iteratee` которому будет подаваться поток `aOut` и который будет возвращать `Step` (aka, новое перечисляемое, правда?). Этот вроде как имеет интуитивный смысл: мы ввели посредника, который принимает ввод от одного источника и преобразует `Step` в новое состояние.

А теперь, возможно, самая сложная часть всего этого: как мы можем использовать функцию `map`? Оказывается, что `Enumeratee` достаточно близок по сигнатуре типа к `Enumerator`, чтобы мы могли сделать:

```
map read $$ sumIter
```

Но сигнатура типа оказывается немного странной:

```
Iteratee String m (Step Int m Int)
```

Помня, что `Iteratee` — это просто завернутый `Step`, мы имеем здесь перечисляемое, которое принимает `String` и возвращает `Iteratee`, который, в свою очередь принимает `Int` и производит `Int`. Этот причудливый результат позволяет сделать один из наших великих трюков с перечисляемым: подключить данные из нескольких источников. Например, мы могли бы подключить несколько `String` к этой уродливой вещи, запустить ее, получить новое перечисляемое, которое получает `Int`, скармливает эти `Int` и получает результат `Int`.

(Если это все прошло мимо ваших ушей, не волнуйтесь. Я больше не буду говорить о таких вещах.)

Но зачастую, нам не нужна все эта мощь. Мы просто хотим прилепить энумератор к нашему перечисляемому и получить новое перечисляемое. В нашем случае, мы хотим присоединить `map` к `sumIter` для получения нового перечисляемого, которая принимает `String` и возвращает `Int`. Для того, чтобы это сделать, нам нужна функция подобная этой:

```

unnest :: Monad m => Iteratee String m (Step Int m Int) -> Iteratee ()
  String m Int
unnest outer = do -- используем экземпляр Monad для Iteratee
  inner <- outer -- inner :: Step Int m Int
  go inner
where
  go (Error e) = throwError e
  go (Yield x _) = yield x EOF
  go (Continue k) = k EOF >>= go

```

Мы можем запустить этот полнейший беспорядок следующим образом:

```
run_ (lineEnum $$ unnest $ map read $$ sumIter) >>= print
```

На самом деле функция `unnest` доступна в `Data.Enumerator`, и там она называется `joinI`. Таким образом, в действительности мы должны писать:

```
run_ (lineEnum $$ joinI $ map read $$ sumIter) >>= print
```

G.17. Пропуск значений

Давайте напишем чуть более интересное повторяемое, которое пропускает входные значения через одно.

```

skip :: Monad m => Enumeratee a a m b
skip (Continue k) = do
  x <- head
  _ <- head -- пропускаем одно значение
  case x of
    Nothing -> return $ Continue k
    Just y -> do
      newStep <- lift $ runIteratee $ k $ Chunks [y]
      skip newStep
skip step = return step

```

Что интересно в этом подходе, это то, как он похож на `Enumerator`. Мы делаем много тех же вещей: проверяем, или значение `Step` это `Continue`; если это не так, то просто возвращаем его. Затем мы пользуемся тем, что `Iteratee` является экземпляром `Monad`, используя функцию `head` извлекаем два значения из потока. Если данных больше нет,

мы возвращаем исходное значение Continue: как и с Enumerator, мы не возвращаем EOF, чтобы мы позже смогли скормить еще данные в перечисляемое. Если есть данные, мы передаем их в перечисляемое, получаем наше новое значение Step, и затем зацикливаемся.

Что круто в повторяемом, это то, что мы можем сцеплять все вместе:

```
run_ (lineEnum $$ joinI $ skip $$ joinI $ map read $$ sumIter) >>= \
  print
```

Здесь мы читаем строки, пропускаем каждую вторую, преобразуем String в Int и суммируем.

G.18. Примеры из жизни: пакет http-enumerator

Я начал работать на этой главой, во время работы над пакетом http-enumerator. Я думаю, что использование повторяемых дает отличное объяснение преимуществ, которые они могут предложить в реальной жизни. Существуют три различных способа разбить тело ответа:

- Блочное кодирование. В этом случае веб-сервер возвращает шестнадцатеричную строку, определяющую длину следующего блока, а затем этот блок. В конце концов, он отправляет 0, чтобы указать конец этого ответа.
- Длина содержимого. Здесь, веб-сервер отправляет перед телом ответа заголовок, указывающий полную длину тела.
- Ничего не делать. В этом случае, тело ответа длится до конца файла.

Кроме того, тело может быть и не быть запаковано GZIP. В конечном итоге мы остаемся со следующими повторяемыми, каждое с сигнатурой типа Enumeratee `ByteString ByteString m b: chunkedEncoding, contentLength` и `ungzip`. Затем мы должны сделать что-то похожее на:

```
let parseBody x =
  if ("transfer-encoding", "chunked") 'elem' responseHeaders
  then joinI $ chunkedEncoding $$ x
  else case mlen of
    Just len -> joinI $ contentLength len $$ x
    Nothing -> x -- вообще не применяем ни одно \
  повторяемое
let decompress x =
  if ("content-encoding", "gzip") 'elem' responseHeaders
  then joinI $ ungzip $$ x
  else x
run_ $ socketEnumerator $$ parseBody $ decompress $ bodyIteratee
```

Мы создаем цепочку: данные с сервера поступает в функцию `parseBody`. В случае блочного кодирования, данные обрабатываются соответствующим образом, и затем заголовки отфильтровываются. Если мы имеем дело с длинной содержимого, то считываем только указанное количество байт. В противном случае `parseBody` является пустой операцией.

В любом случае, тело ответа затем скармливается `decompress`. Если тело закодировано GZIP, то `unzip` распаковывает его, в противном случае `decompress` является пустой операцией. Наконец, разобранные и распакованные данные скармливаются в поставленную пользователем функцию `bodyIteratee`. Пользователь остается в блаженном неведении о шагах, которые потребовались, чтобы доставить ему данные.

G.19. Упражнения

1. Написать повторяемое, которое принимает шестнадцатеричные символы (например, «DEADBEEF»), и записывает их в `Word8`. Его сигнатура типа должна быть `Enumerator Char Word8 m b`.
2. Написать противоположное повторяемое, например `Enumerator Word8 Char m b`.
3. Создать тест `quickcheck`, который проверяет, что эти две функции работают правильно.

G.20. Выводы

- `Enumerator` является конвейером, связывающим эnumератор с перечисляемым.
- Странная сигнатура типа `Enumerator` скрывает множество мощных возможностей. Особенно обратите внимание, как схожа его сигнатура типа с `Enumerator`.
- Вы можете слить `Enumerator` в `Iteratee`, используя `joinI $ enumerator $$ iteratee`.
- Не забывайте, что вы можете использовать экземпляр `Monad` для `Iteratee` при создании собственных повторяемых.
- Вы всегда можете объединить несколько повторяемых вместе, как это сделано в `http-enumerator`.