The Bitumen Framework Handbook

Shantanu Kumar

The Bitumen Framework Handbook

Shantanu Kumar

Publication date 02 March, 2011

[1] URL: http://creativecommons.org/licenses/by-nd/3.0/

Table of Contents

Chapter 1. Introduction

Premise

The Bitumen Framework Handbook serves as a developer's guide to using Bitumen Framework for developing applications in the Clojure programming language, giving you the knowhow to use it in various situations.

What is it?

Bitumen Framework[1] is a collection of libraries for developing Clojure[2] applications on the Java™ Virtual Machine. This handbook describes how to use those libraries for various use-cases.

Clojure is a functional Lisp with strong focus on immutability and concurrency. Clojure also has excellent Java interoperability, something that Bitumen Framework libraries use heavily.

Intended Audience

An ideal reader would know enough Clojure to be able to write short programs on her own and try out things on the REPL. She would know general programming and logic concepts. Prior experience in application programming using another language and knowing about Java/JVM to some extent may be helpful.

Bitumen Framework Libraries

This handbook contains several chapters, each dedicated to one specific library from the Bitumen Framework. They broadly cover the following libraries:

- Clj-MiscUtil
- Clj-DBSpec
- OSS-JDBC
- Clj-DBCP
- Clj-Liquibase
- Lein-LB
- Fountain-JDBC
- SQLRat
- Clj-StringTemplate

Description of Libraries

The libraries listed above serve different purposes toward developing applications. Their description is below.

[1] URL: http://code.google.com/p/bitumenframework/ [http://code.google.com/p/bitumenframework/]
[2] http://clojure.org [http://clojure.org]

Clj-MiscUtil

Description: Clj-MiscUtil is an assortment of Clojure functions/macros to carry out miscellaneous common activities (see *Table of Contents, Chapter 2*.)

Project URL: https://bitbucket.org/kumarshantanu/clj-miscutil/src [https://bitbucket.org/kumarshantanu/clj-miscutil/src]

Clj-DBSpec

Description: Clj-DBSpec is a common configuration spec for dealing with relational databases e.g. data source, connection, conversion of schema/tables/columns/indices names between the database and Clojure.

Project URL: https://bitbucket.org/kumarshantanu/clj-dbspec/src [https://bitbucket.org/kumarshantanu/clj-dbspec/src]

OSS-JDBC

Description: OSS-JDBC is a regularly updated collection of Open Source JDBC drivers for various databases. The OSS-JDBC Maven artefact pulls in all JDBC drivers for supported databases.

Project URL: https://bitbucket.org/kumarshantanu/oss-jdbc/src [https://bitbucket.org/kumarshantanu/oss-jdbc/src]

Clj-DBCP

Description: Clj-DBCP is a simple Java-6/Clojure wrapper around the Apache DBCP library for creating database connection pools and for embedding databases in applications.

Project URL: https://bitbucket.org/kumarshantanu/clj-dbcp/src [https://bitbucket.org/kumarshantanu/clj-dbcp/src]

Clj-Liquibase

Description: Clj-Liquibase is a simple Clojure DSL/wrapper around the Liquibase library http://www.liquibase.org/ for carrying out relational database change management and migrations.

Project URL: https://bitbucket.org/kumarshantanu/clj-liquibase/src [https://bitbucket.org/kumarshantanu/clj-liquibase/src]

Lein-LB

Description: Leiningen plugin for Liquibase,[3] a database change management software.

Project URL: https://bitbucket.org/kumarshantanu/lein-lb/src [https://bitbucket.org/kumarshantanu/lein-lb/src]

Fountain-JDBC

Description: Fountain-JDBC is a Clojure wrapper for Spring-JDBC[4].

Project URL: https://bitbucket.org/kumarshantanu/fountain-jdbc/src [https://bitbucket.org/kumarshantanu/fountain-jdbc/src]

[3] URL: http://www.liquibase.org/ [http://www.liquibase.org/]
[4] URL: http://www.springsource.org/about [http://www.springsource.org/about]

SQLRat

Description: SQLRat is a Clojure (v1.2 or later) library to access relational databases using entity objects and to navigate entity relations in a stateless manner. Easy to use and flexible - you can also pass in native SQL for accessing the database.

Project URL: https://bitbucket.org/kumarshantanu/sqlrat/src [https://bitbucket.org/kumarshantanu/sqlrat/src]

Clj-StringTemplate

Description: Clj-StringTemplate is a simple Clojure wrapper around the StringTemplate library.

Project URL: https://bitbucket.org/kumarshantanu/clj-stringtemplate/src [https://bitbucket.org/kumarshantanu/clj-stringtemplate/src]

Version matrix

The libraries described above have the following inter-dependncies:

Library==>	**Version**	**Depends on==>**	**Version**
Clj-MiscUtil	0.2	-	-
Clj-DBSpec	0.1	-	-
OSS-JDBC	0.4	-	-
Clj-DBCP	0.4	-	-
Clj-Liquibase	0.1	Clj-DBSpec	0.1
Lein-LB	0.1	- (will use Clj-Liquibase in 0.2)	-
		Clj-MiscUtil	0.2
Fountain-JDBC	0.1	Clj-DBSpec	0.1
		Clj-MiscUtil	0.2
SQLRat	0.2	Clj-ArgUtil (deprecated)	0.1
Clj-StringTemplate	0.2	-	-

Resources

Resources information related to the project are below.

Project location

URL: http://code.google.com/p/bitumenframework/ [http://code.google.com/p/bitumenframework/]

Documentation

You can access the source of this handbook: https://bitbucket.org/kumarshantanu/bituf-handbook/src [https://bitbucket.org/kumarshantanu/bituf-handbook/src]. Apart from this handbook every project may have its own tutorial/documentation that you can access at the respective project page.

License

Unless mentioned otherwise all libraries in Bitumen Framework are released under Apache License 2.0[5] However, please note each library may have its own respective license (owing to the dependencies).

Project Blog

On BlogSpot: http://bitumenframework.blogspot.com/ [http://bitumenframework.blogspot.com/]

Twitter

Author: http://twitter.com/kumarshantanu [http://twitter.com/kumarshantanu]

Project: http://twitter.com/bituf [http://twitter.com/bituf]

Reporting Bugs

You can report bugs for every library to its respective project issue tracker on BitBucket.[6]

Discussion Group

On Google groups: http://groups.google.com/group/bitumenframework [http://groups.google.com/group/bitumenframework]

Contributing

You are encouraged to participate in the development of Bitumen Framework. There is plenty of work ahead to be accomplished and your contribution in terms of code, ideas, bug reports, feedback, graphic art or whichever way you can, are most welcome.

Contact

Feel free to get in touch with the project author and members on the project discussion group. You can also contact by writing an email to kumar(dot)shantanu(at)gmail(dot)com or via Twitter.

[5]URL: http://www.apache.org/licenses/LICENSE-2.0 [http://www.apache.org/licenses/LICENSE-2.0]
[6]URL: https://bitbucket.org/kumarshantanu/ [https://bitbucket.org/kumarshantanu/]

Chapter 2. Miscellaneous Activities

This chapter describes the usage of *Clj-MiscUtil* library for miscellaneous tasks. The tasks are categorized below into sub-sections. Clj-MiscUtil is under the namespace `org.bituf.clj-miscutil` that you can include as follows:

```
(use 'org.bituf.clj-miscutil)
```

or

```
(ns example.app
  (:require
    [org.bituf.clj-miscutil :as mu]))
```

The examples below assume the first statement.

Random values

Random numeric and string values may be required for various purposes while programming. The `random-number` and `random-string` functions cater for these.

Generating a random number

To generate a random number (long), you can use the `random-number` function.

```
(random-number)
```

Random numbers may be useful to create unique filenames or database table names with a common prefix.

```
(str "invoices-2011-02-20-" (random-number))
```

To obtain a random number (double precision) from a certain min/max range, you can specify the range:

```
;; return a random number between 10 (included) and 20 (excluded)
(random-number 10 20)
```

For an example, to randomly pick a value from a vector, you would use something like this:

```
(let [v [1 2 3 4 5 6 7 8 9]]
  (v (int (random-number 0 (count v)))))
```

Generating a random string

Random alphanumeric string can be generated using the `random-string` function that returns string of length 11-13 characters by default. You can optionally specify a length of the random string.

```
(random-string)     ; string of random 11-13 characters

(random-string 20)  ; string of 20 random characters
```

Another way of generating alphanumeric random characters is to use the `random-charseq` function. By default this function returns a lazy-sequence of infinite alphanumeric characters but you can optionally specify a length.

```
(random-charseq)     ; lazy seq of infinite random characters
```

```
(random-charseq 10)  ; lazy seq of 10 random characters
```

Type checking

Clojure core has some built in functions to determine the types of values, such as `number?`, `string?`, `map?` etc. Additional such functions not available in Clojure core are below:

```
(boolean? false)       ; returns true
(boolean? "hello")     ; returns false

(not-boolean? "hello") ; returns true
(not-boolean? true)    ; returns false

(date? (java.util.Date.))      ; returns true
(date? "hello")                ; returns false

(not-date? (java.util.Date.))  ; returns false
(not-date? "hello")            ; returns true
```

Pretty printing

These are some pretty printing functions that you can use:

`with-stringwriter`

This macro is used to assign a StringWriter to specified symbol in a let binding and execute body of code in that context. Returns the string from StringWriter . Example:

```
(with-stringwriter s
  (.append s "Hello")
  (.append s "World"))  ; returns "HelloWorld"
```

`with-err-str`

This macro is like `with-out-str` but for `*err*` instead of `*out*`.

`pprint-str`

This function (accepts one argument) prints anything that you pass to it using `clojure.pprint/pprint` and returns the result as string.

`comma-sep-str`

This function accepts a collection as argument and returns a comma separated string representation.

`echo`

This function is a simple diagnostic tool that pretty-prints anything (single argument) you pass to it and returns the same argument.

Number sign detection

Detecting sign of numbers may be tricky. `(pos? 34)` returns `true` as expected, but `(pos? "Hello")` throws `ClassCastException`. The alternatives below return false in event of exceptions.

```
(zeronum? 0)        ; return true
```

```
(zeronum? 36)       ; return false
(zeronum? "hello")  ; returns false

(posnum? 378)       ; returns true
(posnum? -32)       ; returns false
(posnum? :pqrs)     ; returns false

(negnum? 378)       ; returns false
(negnum? -32)       ; returns true
(negnum? :pqrs)     ; returns false
```

Print tables

Printing tables of data is useful for many scenarios. The data to be printed as a table is generally a collection of rows, optionally with a header row. You can use the `print-table` function to print a table of data to `*out*`.

```
(print-table [{:a 10 :b 20 :c 30}
              {:a 40 :b 50 :c 60}
              {:a 70 :b 80 :c 90}])  ; with titles "a", "b" and "c"

(print-table [[10 20 30]
              [40 50 60]
              [70 80 90]])          ; without any titles

(print-table  [:a :b :c]
             [[10 20 30]
              [40 50 60]
              [70 80 90]])          ; with titles "a", "b" and "c"
```

The examples above use the defaults to print the tables. You can override the defaults to alter the way tables are printed. For example, every column width is computed by default, if you want to specify width of columns you can use something like this:

```
(binding [*pt-cols-width* [5 -1 7]]
  (print-table [{:id 1001 :name "Harry"    :gender :male}
                {:id 2997 :name "Samantha" :gender :female}
                {:id 8328 :name "Christie" :gender :female}]))
```

Other options you can override are as follows:

```
*pt-column-delim*   -- column delimiter string
*pt-min-cols-width* -- collection of minimum width for each column
*pt-max-cols-width* -- collection of maximum width for each column
*pt-cols-width*     -- collection of numeric width for each column
```

Note: One notable feature of `*pt-cols-width*` is that a non-positive number implies that the width would be automatically computed.

Var metadata

Details about vars can be very useful during debugging, diagnostics or error reporting. You can find out the name (string) of a var using the macro `var-name`, fn-body of the var using `var-body` function and type/value of a value using the `val-dump` function.

```
(var-name map?)  ; returns "map?"

(var-body map?)  ; returns source code for map? function

(val-dump #"[a-z0-9]")  ; returns type and value as string
```

Throwing exceptions

Throwing exceptions with sufficient diagnostic context in them is very important for meaningful error reporting. The functions shown below let you throw exceptions with relevant context:

Function	Which exception	When to use
`illegal-arg`	IllegalArgumentException	You want to specify the reason as one or more string values
`illegal-arg-wrap`	IllegalArgumentException	You want to wrap another exception
`illegal-argval`	IllegalArgumentException	Actual argument is different from expected input
`illegal-state`	IllegalStateException	You want to specify the reason as one or more string values
`illegal-state-wrap`	IllegalStateException	You want to wrap another exception
`unsupported-op`	UnsupportedOperationException	You want to specify the reason as one or more string values
`unsupported-op-wrap`	UnsupportedOperationException	You want to wrap another exception

Examples of these functions are as follows:

```
(illegal-arg "name should not have more than 3 vowels")

(try (get-fname empname)
  (catch Exception e
    (illegal-arg-wrap e (str "bad empname: " empname))))

(illegal-argval "empname" "string having 3 vowels or less" empname)

(illegal-state "Value of x cannot be > " max-x)

;; assuming e is an exception
(illegal-state-wrap e "Fahrenheit cannot be more than 98.4 degrees")

(unsupported-op "Not yet implemented")

;; assuming e is an exception
(unsupported-op e "Attempt to carry out activity failed")
```

Non-breaking error handling

This has been discussed [1] [2] on the Bitumen Framework blog.

When executing code that might throw an exception we generally wrap it in a try/catch block as we want to deal with the breakage in execution flow. Dealing with execution breakage in-place makes the code imperative and often brittle.

The `maybe` macro family

The `maybe` macro executes body of code and returns a vector of two elements - the first element being the return value, and the second being the exception.

```
(maybe (pos? 648)) ; returns [true nil]
(maybe (pos? nil)) ; returns [nil <NullPointerException instance>]
```

Since `maybe` is a macro you can pass arbitrary body of well-formed code to it and it will consistently return a 2-element vector every time. An example usage of `maybe` is as follows:

```
(doseq [[ret ex] (map #(maybe (process-order %)) orders)]
  (or ret (log/success ret))
  (or ex  (do (log/error ex)
              (trigger-alert ex))))
```

There are two close cousins of the `maybe` macro, called `maybe-ret` (gets the return value, or `nil` when an exception is thrown) and `maybe-ex` (gets the exception, or `nil` when no exception is thrown).

```
(maybe-ret (Integer/parseInt "45"))    ; returns 45
(maybe-ret (Integer/parseInt "hello")) ; returns nil

(maybe-ex (Integer/parseInt "45"))     ; returns nil
(maybe-ex (Integer/parseInt "hello"))  ; returns NumberFormatException
```

Selectively uphold or ignore exceptions

At times we may need to ignore or uphold exceptions based on the context. The macros `filter-exception` (takes a predicate function) and `with-exceptions` (takes list of exceptions to uphold and ignore) let us do exactly that. Both macros return `nil` when an exception is ignored.

When you need arbitrary control over how/when to filter an exception you can use `filter-exception`.

```
(filter-exception #(instance? ClassCastException %)
  (pos? "hello"))  ; returns nil
```

Another situation is when you know beforehand which exceptions to uphold and which ones to ignore.

```
;; throws exception
(with-exceptions [IllegalArgumentException IllegalStateException]
                 [RuntimeException]
  "foo" ; non-effective return value
```

[1] Part 1: http://bitumenframework.blogspot.com/2010/11/non-breaking-error-handling-in-clojure.html
[2] Part 2: http://bitumenframework.blogspot.com/2011/01/non-breaking-error-handling-in-clojure.html

```
  (throw (IllegalArgumentException. "dummy")))

;; swallows exception
(with-exceptions [IllegalArgumentException IllegalStateException]
                 [RuntimeException]
  "foo" ; non-effective return value
  (throw (NullPointerException. "dummy")))
```

Type conversion

Type conversion is one of the most frequent needs during data processing. The table below describes which function converts to which type. All functions in this section accept input in various formats and try to coerce the input into desired type.

Function	Converts to	Remarks
`as-string`	string	converts anything to string (i.e. `:key` becomes `"key"`)
`java-filepath`	string	Replaces path separators in supplied filepath with Java-compatible platform independent separator
`split-filepath`	vector of 2 string elements	Splits filepath as filedir (with platform-independent path separator) and filename and returns a vector containing both
`pick-filedir`	string	Picks filedir from a given filepath
`pick-filename`	string	Picks filename from a given filepath
`as-vstr`	string	Verbose string (i.e. `nil` becomes `"<nil>"`)
`as-keys`	collection	Gets keys of a map, or the entire collection if not a map
`as-vals`	collection	Gets vals of map, or the entire collection of not a map
`as-vector`	vector	Turns anything into a vector
`as-set`	set	Turns anything into a set
`as-map`	map	Turns anything into a map
`coerce`	depends on the predicate function	Coerces value using a predicate function
`as-boolean`	boolean	Parses anything as boolean
`as-short`	short integer	Parses anything as short
`as-integer`	integer	Parses anything as integer
`as-long`	long integer	Parses anything as long
`as-float`	float	Parses anything as float
`as-double`	double	Parses anything as double

`not-` associated functions

Quite often we use a `(not ..)` version of a boolean function, e.g. `(not (map? foo))` while checking for conditions. The functions listed below are shorthand of using with `not`:

Function==>	Counterpart in clojure.core	Function==>	Counterpart in clojure.core
any?	not-any?	not-associative?	associative?
not-bound?	bound?	not-char?	char?
not-chunked-seq?	chunked-seq?	not-class?	class?
not-coll?	coll?	not-contains?	contains?
not-counted?	counted?	not-decimal?	decimal?
not-delay?	delay?	not-distinct?	distinct?
not-empty?	empty?	not-even?	even?
not-extends?	extends?	not-false?	false?
not-float?	float?	not-fn?	fn?
not-future-cancelled?	future-cancelled?	not-future-done?	future-done?
not-future?	future?	not-identical?	identical?
not-ifn?	ifn?	not-instance?	instance?
not-integer?	integer?	not-isa?	isa?
not-keyword?	keyword?	not-list?	list?
not-map?	map?	not-neg?	neg?
not-nil?	nil?	not-number?	number?
not-odd?	odd?	not-pos?	pos?
not-ratio?	ratio?	not-rational?	rational?
not-reversible?	reversible?	not-satisfies?	satisfies?
not-seq?	seq?	not-sequential?	sequential?
not-set?	set?	not-sorted?	sorted?
not-special-symbol?	special-symbol?	not-string?	string?
not-symbol?	symbol?	not-thread-bound?	thread-bound?
not-true?	true?	not-var?	var?
not-vector?	vector?	not-zero?	zero?

Map transformation

Transforming collections can be easily done using `map` or `for`. However, transforming maps always involves destructuring the key and value and then apply any transformation. The functions `map-keys` and `map-vals` let you simply transform either the keys or the values of a map. When using `map-keys` you must ensure that the transformed set of keys are unique.

```
(map-keys inc (array-map 1 2 3 4 5 6)) ; returns {2 2 4 4 6 6}

(map-vals dec (array-map 1 2 3 4 5 6)) ; returns {1 1 3 3 5 5}
```

Array types

Dealing with arrays may become unavoidable when working with Java libraries. The following functions may help:

array-type

`array-type` returns the common type (class) of elements that can be contained in the array.

`array?`

`array?` returns true if the argument is an array, false otherwise.

`not-array?`

`not-array?` is same as `(not (array? foo))`.

`contains-val?` : `contains?` for value

The `contains?` function in clojure.core looks for a key in a collection. For a vector the keys are the indices, for sets they are the elements and for maps they are keys. The `contains-val?` function looks for values instead of keys.

```
(contains?     [:a :b :c] :b) ; returns false
(contains-val? [:a :b :c] :b) ; returns true

(contains?     {:a 10 :b 20} 20) ; returns false
(contains-val? {:a 10 :b 20} 20) ; returns true
```

Stack trace and Exceptions

This concept has been discussed[3] on the Bitumen Framework Blog.

Exception stack trace for Clojure code usually includes quite some unwanted entries, which are not very useful while debugging and rather clutter the view. The function `print-exception-stacktrace` can be used to print an exception stack trace with reduced clutter. It falls back to the following as stack trace elements (in that order):

1. Application code and Dependencies (without Clojure core/contrib or Java code)
2. Clojure core/contrib and application code (without Java code)
3. All Java and Clojure code (everything)

When trying this on the REPL with Clojure core/contrib libraries, you may not encounter #1 in the stack trace.

There are two convenience macros - `!` and `!!` that accept a body of code and print friendly stack trace if there is any exception. The difference between the two is that `!` prints only required columns of the stack trace and `!!` prints an additional IDE Reference column to generate filenames clickable within the IDE (tested on Eclipse and IDEA). Example is below:

```
(! (foo arg))   ; prints normal stack-trace columns
```

[3]URL:http://bitumenframework.blogspot.com/2010/10/stack-traces-for-clojure-app.html

```
(!! (foo arg))  ; prints extra IDE reference column
```

Assertion helpers

verify-arg

Throws IllegalArgumentException if body of code does not return true.

```
(verify-arg (map? arg)) ; verifies arg is a map
```

verify-type

Throws IllegalArgumentException if argument does not match expected type.

```
(verify-type java.util.Date join-date)
```

verify-cond

This macro is same as `verify-arg` but throws IllegalStateException instead of IllegalArgumentException.

verify-opt

This function ensures that only permitted optional arguments are passed as optional arguments to a function or macro.

```
(defn foo
  [arg & {:keys [a b] :as opt}] {:pre [(verify-opt [:a :b] opt)]}
  ...)
```

Type annotation

This topic has been discussed[4] on the Bitumen Framework blog.

Type annotation is a way to inject type metadata into regular objects without altering their content. The type metadata can be read back later to act upon them in different ways.

Annotating with types

The following functions help you annotate objects with type metadata:

typed

This function annotates an object with specified type(s), e.g.

```
(typed [:argentina :spain]
  :speaks-spanish)                ; tag the object with one type

(typed {:name "Henry"
        :age   23
        :place "Connecticut"}
```

[4]URL:http://bitumenframework.blogspot.com/2010/10/typed-abstractions-in-clojure.html

```
  :person-data :has-age :has-name) ; tag object with multiple types

(typed 65 :average-weight)         ; throws exception - 65 is not object
```

ftyped

For non-objects such as numbers, string, date etc. we need to use `ftyped` so that they can be coerced as objects before they are type-annotated, e.g.

```
(ftyped 65 :average-weight)        ; this works fine
```

*Note:*Objects created using `ftyped` are no-arg functions that must be executed to return the wrapped value, e.g.

```
(let [d (ftyped 60 :retirement-age)] (d))
```

obj?

not-obj?

These functions tell whether a value is an object (i.e. whether it implements the IObject protocol) or not. Only such objects can be annotated with type metadata.

Reading the types back

type-meta

This function returns the type metadata of an object

```
(type-meta (typed {:order-id 34}
             :pending))              ; returns :pending

(type-meta (typed [:france :germany]
             :european :countries))  ; returns [:european :countries]

(type-meta (typed (typed [10 20]
                    :numbers)
             :sample))               ; returns [:numbers :sample]
```

Removing type information

untyped

This function can be used to remove type information from an object.

```
(type-meta (untyped (typed {:a 10 :b 20} :abc))) ; returns nil
```

Type hierarchies and implied types

> *Note:* The type and hierarchy system described here works in conjunction with `type` and `isa?` functions and integrates with Clojure multi-methods [5] [6].

[5] URL:http://clojure.org/multimethods
[6] URL:http://dosync.posterous.com/beyond-javascript-prototype-chains

Clojure has a built-in feature of type annotation and hierarchy independent of the objects themselves. It means objects are not hard-bound to the types (as in classes, e.g. Java) and type hierarchies can be applied at runtime. This feature enables multiple-inheritance of types in Clojure. See this example:

```
(derive ::employee ::salaried) ; employee is salaried
(derive ::salaried ::person)   ; salaried is a person

;; returns true because ::employee is both ::salaried and ::person
(every? #(typed? (typed {:id 3964 :name "Joe"}
                   ::employee) %)
  [::salaried ::person])

;; returns true because ::employee is at least ::person
(some   #(typed? (typed {:id 9604 :name "Cher"}
                   ::employee) %)
  [::freelancer ::person])
```

typed?

You would notice that the code snippet makes use of the function `typed?`, which returns true if the object is of specified type. It internally makes use of the `isa?` function that knows about the specified and global hierarchies.

Keyword/String conversion

The keyword/string conversion functions would be best shown with examples:

```
(k-to-camelstr :to-do) ; returns "toDo"
(camelstr-to-k "toDo") ; returns :to-do

(k-to-methodname :to-do ["add"]) ; returns "addToDo" (now see k-to-camelstr)
(k-to-setter :price)       ; returns "setPrice"
(k-to-setter :set-price)   ; returns "setPrice" (detects "set", no repeat)
(k-to-getter :price)       ; returns "getPrice"
(k-to-getter :get-price)   ; returns "getPrice"  (detects "get", no repeat)
(k-to-getter :is-in-stock) ; returns "isInStock" (detects "is" too)

(coll-as-string [:a :b :c])    ; returns ["a" "b" "c"]
(coll-as-keys   ["a" "b" "c"]) ; returns [:a :b :c]
(keys-to-str {:a 10 :b 20})    ; returns {"a" 10 "b" 20}
(str-to-keys {"a" 10 "b" 20})  ; returns {:a 10 :b 20}
```

Reflection (not for performance-critical code)

The examples below show how to use the API:

Call Java methods

```
(method "Hello" :char-at 0) ; .charAt(0) - returns \H

(method (call-specs "Hello"
          [:char-at 0]     ; .charAt(0)       - returns \H
```

```
          [:substring 3 4] ; .substring(3, 4) - returns "l"
          [:to-string]     ; .toString()      - returns "Hello"
          ))

(map #((apply pojo-fn "Hello" %))
  [[:char-at 0]     ; returns primitive char
   [:substring 3 4] ; returns string
   [:to-string]     ; no-arg method
   ]) ; returns lazy (\H "l" "Hello")
```

Call Java setter methods

```
(setter (StringBuilder.)
  :length 0) ; .setLength(0) - returns void, 'setter' returns nil

(setter (call-specs sb
          [:length 4]     ; .setLength(4)      - returns void
          [:char-at 0 \C] ; .setCharAt(0, 'C') - returns void
          )) ; 'setter' returns [nil nil]

(map #((apply setter-fn sb %))
  [[:length 4]     ; .setLength(4)      - returns void
   [:char-at 0 \C] ; .setCharAt(0, 'C') - returns void
   ]) ; returns lazy (nil nil)
```

Call Java getter methods

```
(let [lst (java.util.LinkedList.)
      _   (.add lst 1)
      _   (.add lst 2)]
  (getter lst :first) ; .getFirst() - returns 1
  (getter (call-specs lst :first ; .getFirst() - returns 1
                          :last  ; .getLast()  - returns 2
                          )) ; returns [1 2]
  (map (getter-fn lst) [:first ; .getFirst() - returns 1
                        :last  ; .getLast()  - returns 2
                        ])) ; returns lazy (1 2)
```

java.util.Properties handling

Assuming that the file sample.properties has the following content:

```
a=10
b=20
c=true
```

Properties can be transformed into maps:

```
(let [ps (read-properties "src/test/conf/sample.properties")]
  (property-map ps)     ; returns {"a" "10" "b" "20" "c" "true"}
  (strkey-to-keyword
    (property-map ps))) ; returns {:a "10" :b "20" :c "true"}
```

```
(is-true? "true") ; returns true (useful to test bool values in properties)
```

JNDI functions

These JNDI tree-printing functions can be used to debug the JNDI configuration:

```
(print-jndi-tree) ; print the JNDI tree referring initial context

(find-jndi-subcontext (javax.naming.InitialContext.)
         "java:comp") ; returns javax.naming.Context (if configured)

(jndi-lookup
  "java:comp/env/myDataSource") ; returns javax.sql.DataSource (if configured)
```

Chapter 3. RDBMS Essentials (TODO)

TODO

Configuration

Obtaining JDBC Drivers

Connection Pooling

Simple SQL

Parameterized SQL

CRUD - Create, Retrieve, Update, Delete

Batch Operations

Stored Procedures/Functions

Parameters to Stored Procedures

Error handling and Recovery

Exceptions

Vendor specific Error Codes

Chapter 4. RDBMS Transactions (TODO)

Programmatic Transactions

Isolation Levels

Declarative Transactions

Chapter 5. Managing RDBMS Changes

Databases often undergo changes due to application upgrades and changes in requirements during development. We need to apply changes to databases to propogate sets of changes to various environments (Dev, QA, Staging, Production etc.) To consistently manage such changes we need a canonical format to express the database changes that can be also used to track the database state and apply changes.

Clj-Liquibase wraps over *Liquibase* to provide database change management functionality. While Liquibase lets you define the changes in an XML format, the Liquibase extensions integrate at API and invocation level to provide a seamless experience. When using Clj-Liquibase you define the database changes in pure Clojure code and invoke the changes using Clojure API.

Quickstart

To use Clj-Liquibase you need to include the required namespace in your application and define a changelog.

```
;; filename: fooapp/src/fooapp/dbchange.clj
(ns fooapp.dbchange
  (:require
    [org.bituf.clj-liquibase        :as lb]
    [org.bituf.clj-liquibase.change :as ch]))

(def ct-change1 (ch/create-table :sample-table1
                  [[:id     :int          :null false :pk true :autoinc true]
                   [:name   [:varchar 40] :null false]
                   [:gender [:char 1]     :null false]]))

(def changeset-1 ["id=1" "author=shantanu" [ct-change1]])

(lb/defchangelog changelog [changeset-1])
```

After defining the changelog, you need to apply the changes **(see below)**.

```
;; filename: fooapp/src/fooapp/dbmigrate.clj
(ns fooapp.dbmigrate.clj
  (:require [fooapp.dbchange         :as dbch]
            [org.bituf.clj-dbcp      :as dbcp]
            [org.bituf.clj-dbspec    :as spec]
            [org.bituf.clj-liquibase :as lb]))

;; define datasource for supported database using Clj-DBCP
(def ds (dbcp/mysql-datasource "localhost" "dbname" "user" "pass"))

(defn do-lb-action "Wrap f using DBSpec middleware and execute it"
  [f]
  (let [g (spec/wrap-dbspec (spec/make-dbspec ds)
            (lb/wrap-lb-init f))]
    (g)))

(defn do-update "Invoke this function to update the database"
  []
  (do-lb-action #(lb/update dbch/changelog-1)))
```

How it works

Changes

Actions

Leiningen Integration

Chapter 6. Using RDBMS - Entities (TODO)

Chapter 7. Text Templates (TODO)