Shrimp

A Rather Practical Example Of Application Development With RESTinio and SObjectizer

stiffstream

A few words about us

We are a small software development company: https://stiffstream.com

We are mostly known as maintainers of several OpenSource projects, including, but not limited to:

- SObjectizer. One of a few live and evolving implementations of Actor Model for C++.
- RESTinio. Easy to use embeddable asynchronous HTTP/WebSocker server.

A starting point

What is Shrimp and why it was created?

What is Shrimp? From 30000 feets

Shrimp is a small demo-project. OpenSource of course.

It shows how *RESTinio* and *SObjectizer* can be used.

It can be seen as almost real-world application built on top of *RESTinio* and *SObjectizer*.

Why Shrimp was created?

There are several reasons:

- we wanted to show how to use *RESTinio* and *SObjectizer* in real-world cases;
- we wanted to feel ourselves as users of our products to take a different look to *RESTinio* and *SObjectizer* in a new project;
- we wanted to create a testing site for checking new features of our products.

But there is another reason

Shrimp is a good example of a case where embeddable C++ HTTP server is needed to reuse existing C/C++ code base.

In our case we use *ImageMagick++* and several other C/C++ libraries like *libwebp* and *libheif* for the main operations.

And wrap them by new C++ code to provide a HTTP-based interface.

We need to go deeper...

What is Shrimp?

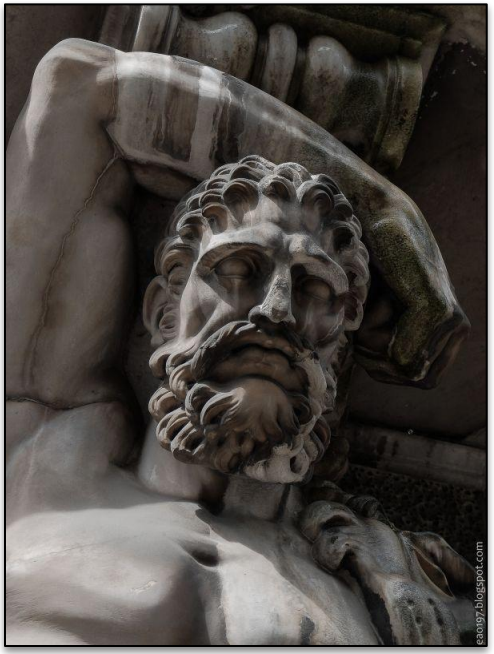
What is Shrimp? Actually

Shrimp is a small web-server application that hosts user's images and returns them in response to HTTP GET requests.

During serving HTTP GET requests Shrimp can:

- resize an image to new size and/or
- convert an image to a different format (for example from .jpg to .webp).

Shrimp in a simple example



Original: DSCF6555.jpg, 1086KiB, 1440x1920px.

Shrimp in a simple example

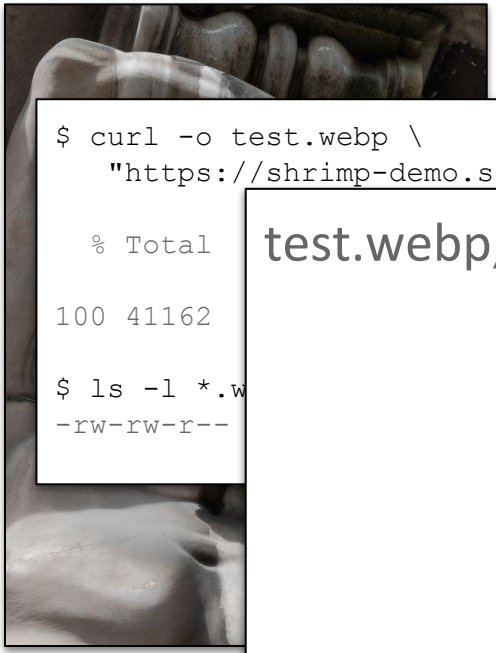
Original: DSCF6555.jpg, 1086KiB, 1440x1920px.

```
$ curl -o test.webp \
   "https://shrimp-demo.stiffstream.com/DSCF6555.jpg?op=resize&width=300&target-format=webp"

  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 41162  100 41162    0     0  41162      0  0:00:01 --:--:--  0:00:01  151k

$ ls -l *.webp
-rw-rw-r-- 1 eao197 eao197 41162 сен  4 08:28 test.webp
```

Shrimp in a simple example



Original: DSCF6555.jpg, 1086KiB, 1440x1920px.

```
$ curl -o test.webp \
   "https://shrimp-demo.stiffstream.com/DSCF6555.jpg?op=resize&width=300&target-format=webp"

  % Total

100 41162

$ ls -l *.w
-rw-rw-r--
```

test.webp, 40KiB, 300x400px

More formal description

Shrimp handles HTTP GET request in the format:

/<file-name>[?params]

where params are:

- *op=resize*. Performs resize operation. Requires *max, height* or *width* param;
- *max=N* or *height=N* or *width=N*. New size of transformed image.
- *target-format={jpg, png, gif, webp, heic}*. New format of image.

Examples

/DSCF6555.jpg

returns original image (no resize, no conversion)

/DSCF6555.jpg?target-format=heic

returns original image (no resize) converted to .heic format

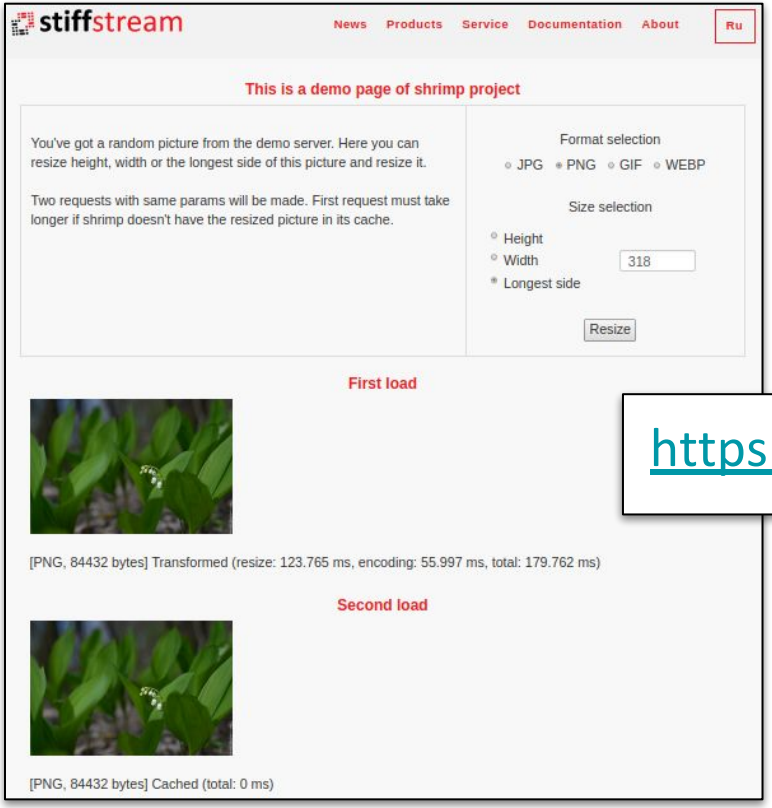
/DSCF6555.jpg?op=resize&max=1024

returns resized image (1024px on longest size) in the original format

/DSCF6555.jpg?op=resize&max=1024&target-format=heic

returns resized image converted to .heic format

Live demo



https://stiffstream.com/en/shrimp-demo.html

Project's sources

Can be found on BitBucket:

https://bitbucket.org/sobjectizerteam/shrimp-demo

There is also a mirror on GitHub:

https://github.com/Stiffstream/shrimp-demo

Disclaimer

Shrimp is a quick-and-dirty prototype.

Don't expect a production ready quality from Shrimp's source.

No benchmarks

We don't provide any performance related data.

Because the real CPU consumer in Shrimp is *ImageMagick*, not *RESTinio* nor *SObjectizer*.

Anyway the Shrimp's repository contains `Dockerfile` and one can build and benchmark Shrimp by him/herself.

Technical details start here...

How does Shrimp work?

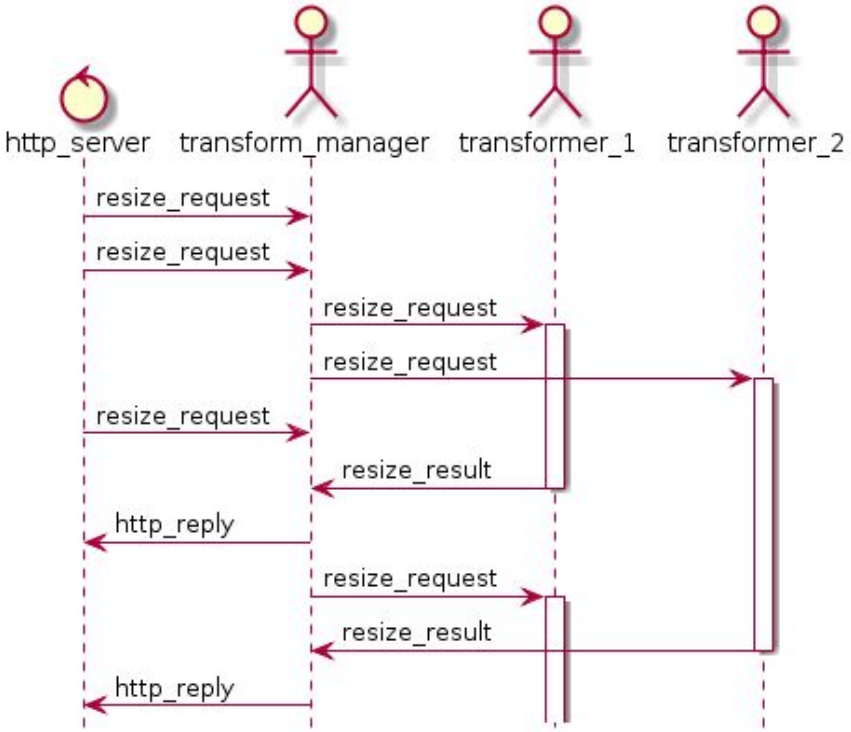
Shrimp's internals in a few words

Shrimp is a multithreaded application.

There are several types of work threads:

- network IO threads on which embedded HTTP-server works;
- transformation manager's thread;
- transformations threads. Actual resizing and format conversions are performed here.

A picture is worth a thousand words



What is used inside Shrimp?

HTTP server is implemented using *RESTinio*.

Request processing is implemented using *SObjectizer*.

Image transformations are performed by *ImageMagick++*.

Let's speak about

RESTinio in Shrimp

RESTinio in Shrimp

Several important *RESTinio's* features are used in Shrimp:

- running of *RESTinio* on external *Asio* context;
- using logging via *spdlog* framework;
- using ExpressJS-like routing for incoming requests;
- sendfile-functionality is used for serving files when no transformations is required.

External Asio context

There are two groups of work threads in Shrimp:

- the first one is controlled by *SObjectizer*;
- the second is controlled by *RESTinio*.

RESTinio's threads finish before *SObjectizer's* threads.

But there are references to *RESTinio* related objects inside *SObjectizer's* threads.

These objects should have valid references to *Asio* context used for *RESTinio*.

External Asio context

We should create *Asio's* `io_context` object before launching *SObjectizer*.

And only when *SObjectizer* is started we can run *RESTinio*.

So we need to pass a reference to existing `io_context` object to *RESTinio's* `run()` function...

External Asio context

```
// ASIO io_context must outlive sobjectizer.
asio::io_context asio_io_ctx;

// Launch SObjectizer and wait while balancer will be started.
...
so_5::wrapped_env_t sobj{
   [&]( so_5::environment_t & env ) {...},
   [&]( so_5::environment_params_t & params ) {...} };

// Now we can launch HTTP-server.
restinio::run(
      asio_io_ctx,
      shrimp::make_http_server_settings(...) );
```

External Asio context

```
// ASIO io_context must outlive sobjectizer.
asio::io_context asio_io_ctx;

// Launch SObjectizer and
...
so_5::wrapped_env_t sobj{
   [&]( so_5::environment_t &
   [&]( so_5::environment_par

// Now we can launch HTTP-server.
restinio::run(
      asio_io_ctx,
      shrimp::make_http_server_settings(...) );
```

Asio's `io_context` object is created.

External Asio context

```
// ASIO io_context must outlive sobjectizer.
asio::io_context asio_io_ctx;

// Launch SObjectizer and wait while balancer will be started.
...
so_5::wrapped_env_t sobj{
  [&]( so_5::environment_t & env ) {...},
  [&]( so_5::environment_params_t & params ) {...} };

// Now we can launch HTTP-server.
restinio::run(
      asio_io_ctx,
      shrimp::make_http_server_settings(...
```

SObjectizer is started here.

SObjectizer will be stopped and destroyed before destruction of `io_context`.

External Asio context

```
// ASIO io_context must outlive sobjectizer.
asio::io_context asio_io_ctx;

// Launch SObjectizer and wait while balanc
...
so_5::wrapped_env_t sobj{
   [&]( so_5::environment_t & env ) {...},
   [&]( so_5::environment_params_t & p

// Now we can launch HTTP-serv
restinio::run(
      asio_io_ctx,
      shrimp::make_http_server_settings(...) );
```

RESTinio is started here.

`restinio::run()` returns only after shutdown of HTTP-server.

Logging via spdlog

RESTinio has a customizable logging facility.

An user can adapt its own logger to be used by *RESTinio*.

We created such adaptation for *spdlog's* logger in Shrimp.

Wrapper around spdlog's logger

```
class http_server_logger_t
{
  public:
    http_server_logger_t( std::shared_ptr<spdlog::logger> logger ) : m_logger{std::move(logger)} {}

    template<typename Builder> void trace( Builder && msg_builder ) {
      log_if_enabled( spdlog::level::trace, std::forward<Builder>(msg_builder) );
    }
    template<typename Builder> void info( Builder && msg_builder ) {
      log_if_enabled( spdlog::level::info, std::forward<Builder>(msg_builder) );
    }
    ...
  private:
    template<typename Builder> void log_if_enabled(
        spdlog::level::level_enum lv, Builder && msg_builder ) {
      if( m_logger->should_log(lv) ) { m_logger->log( lv, msg_builder() ); }
    }

    std::shared_ptr<spdlog::logger> m_logger;
};
```

Tie our wrapper type with RESTinio's traits

```
struct http_server_traits_t
   :  public restinio::default_traits_t
{
   using logger_t = http_server_logger_t;
   using request_handler_t = http_req_router_t;
};
```

Tie our wrapper type with RESTinio's traits

```
struct http_server_traits_t
   :  public restinio::default_traits_t
{
   using logger_t = http_server_logger_t;
   using request_handler_t = http_req_router_t;
};
```

Now *RESTinio* server with this traits will use instance of *http_server_logger_t* for logging.

Instantiation of our wrapper

```
[[nodiscard]] inline auto
make_http_server_settings(
   unsigned int thread_pool_size,
   const app_params_t & params,
   std::shared_ptr<spdlog::logger> logger,
   so_5::mbox_t req_handler_mbox )
{
   ...
   return restinio::on_thread_pool< http_server_traits_t >( thread_pool_size )
         .port( http_srv_params.m_port )
         .protocol( ip_protocol(http_srv_params.m_ip_version) )
         .address( http_srv_params.m_address )
         .handle_request_timeout( std::chrono::seconds(60) )
         .write_http_response_timelimit( std::chrono::seconds(60) )
         .logger( std::move(logger) )
         .request_handler( make_router( params, req_handler_mbox ) );
}
```

Instantiation of our wrapper

```
[[nodiscard]] inline auto
make_http_server_settings(
   unsigned int thread_pool_size,
   const app_params_t & params,
   std::shared_ptr<spdlog::logger> logger,
   so_5::mbox_t req_handler_mbox )
{
   ...
   return restinio::on_thread_pool< http_server_traits_t >( thread_pool_size )
         .port( http_srv_params.m_port )
         .protocol( ip_protocol(http_srv_params.m_ip_version) )
         .address( http_srv_params.m_address )
         .handle_request_timeout( std::chrono::seconds(60) )
         .write_http_response_timelimit( std::chrono::seconds(60) )
         .logger( std::move(logger) )
         .request_handler( make_router( params, req_handler_mbox ) );
}
```

Instantiation of our wrapper

```
[[nodiscard]] inline auto
make_http_server_settings(
   unsigned int thread_pool_size,
   const app_params_t & params,
   std::shared_ptr<spdlog::logger> logger,
   so_5::mbox_t req_handler_mbox )
{
   ...
   return restinio::on_thread_pool< http_se
         .port( http_srv_params.m_port )
         .protocol( ip_protocol(http_srv_par
         .address( http_srv_params.m_add
         .handle_request_timeout( std::c ono::
         .write_http_response_timelimit( std::chrono::seconds(60) )
         .logger( std::move(logger) )
         .request_handler( make_router( params, req_handler_mbox ) );
}
```

Object of type *http_server_traits_t::logger_t* (aka *http_server_logger_t*) will be created here.

Spdlog's logger will be passed to the constructor of that object.

Log example (with --restinio-tracing -l trace)

```
eao197@z930: ~/sandboxes/shrimp-demo/dev
[2018-09-05 11:12:32.117] [run_app] [info] shrimp threads count: io_threads=1, worker_threads=3
[2018-09-05 11:12:32.119] [restinio] [trace] starting server on 127.0.0.1:8080
[2018-09-05 11:12:32.120] [restinio] [info] init accept #0
[2018-09-05 11:12:32.120] [restinio] [info] server started on 127.0.0.1:8080
[2018-09-05 11:13:52.335] [restinio] [trace] accept connection from 127.0.0.1:55214 on socket #0
[2018-09-05 11:13:52.335] [restinio] [trace] [connection:1] start connection with 127.0.0.1:55214
[2018-09-05 11:13:52.335] [restinio] [trace] [connection:1] start waiting for request
[2018-09-05 11:13:52.335] [restinio] [trace] [connection:1] continue reading request
[2018-09-05 11:13:52.335] [restinio] [trace] [connection:1] received 129 bytes
[2018-09-05 11:13:52.335] [restinio] [trace] [connection:1] request received (#0): GET /DSCF6555.jpg?op=resize&width=300&target-format=webp
[2018-09-05 11:13:52.335] [manager] [trace] request received; request_key={{path /DSCF6555.jpg} {format: webp} {params: {w 300}}}, connection_
id=1
[2018-09-05 11:13:52.335] [manager] [debug] store request to pending requests queue; request_key={{path /DSCF6555.jpg} {format: webp} {params:
 {w 300}}}
[2018-09-05 11:13:52.335] [manager] [trace] initiate processing of a request; request_key={{path /DSCF6555.jpg} {format: webp} {params: {w 300
}}}, worker_mbox=7
[2018-09-05 11:13:52.335] [worker_2] [trace] transformation started; request_key={{path /DSCF6555.jpg} {format: webp} {params: {w 300}}}
[2018-09-05 11:13:52.830] [worker_2] [debug] resize finished; request_key={{path /DSCF6555.jpg} {format: webp} {params: {w 300}}}, time=331ms
[2018-09-05 11:13:52.859] [worker_2] [debug] serialization finished; request_key={{path /DSCF6555.jpg} {format: webp} {params: {w 300}}}, time
=27ms
[2018-09-05 11:13:52.859] [manager] [trace] resize_result received; request_key={{path /DSCF6555.jpg} {format: webp} {params: {w 300}}}, worke
r_mbox=7
[2018-09-05 11:13:52.859] [manager] [debug] successul resize result; request_key={{path /DSCF6555.jpg} {format: webp} {params: {w 300}}}, blob
_size=49704
[2018-09-05 11:13:52.859] [manager] [trace] sending positive response back; request_key={{path /DSCF6555.jpg} {format: webp} {params: {w 300}}
}, connection_id=1
[2018-09-05 11:13:52.859] [restinio] [trace] [connection:1] append response (#0), flags: { final_parts, connection_keepalive }, bufs count: 2
[2018-09-05 11:13:52.859] [restinio] [trace] [connection:1] sending resp data, buf count: 2
[2018-09-05 11:13:52.859] [restinio] [trace] [connection:1] outgoing data was sent: 50176 bytes
[2018-09-05 11:13:52.859] [restinio] [trace] [connection:1] should keep alive
[2018-09-05 11:13:52.859] [restinio] [trace] [connection:1] start waiting for request
[2018-09-05 11:13:52.859] [restinio] [trace] [connection:1] continue reading request
[2018-09-05 11:13:52.860] [restinio] [trace] [connection:1] EOF and no request, close connection
[2018-09-05 11:13:52.860] [restinio] [trace] [connection:1] close
[2018-09-05 11:13:52.860] [restinio] [trace] [connection:1] destructor called
```

ExpressJS-like routing

RESTinio allows to use ExpressJS-like routing for HTTP requests.

See https://stiffstream.com/en/docs/restinio/0.4/expressrouter.html for more details.

This routing mechanism is used in Shrimp.

ExpressJS-like routing

First of all we should define a type for router:

```
using http_req_router_t =
   restinio::router::express_router_t<
      restinio::router::pcre_regex_engine_t<
         restinio::router::pcre_traits_t<
            // Max capture groups for regex.
            5 > > >;
```

ExpressJS-like routing

First of all we should define a type for router:

```
using http_req_router_t =
   restinio::router::express_router_t<
      restinio::router::pcre_regex_engine_t<
         restinio::router::pcre_traits_t<
            // Max capture groups for reg
            5 > > >;
```

Express router can use different engines.

There are engines based on *std::regex*, *Boost.Regex* and *PCRE/PCRE2*.

A PCRE-based engine is used here.

ExpressJS-like routing

Then we should specify the router type in server traits:

```
struct http_server_traits_t
   :  public restinio::default_traits_t
{
   using logger_t = http_server_logger_t;
   using request_handler_t = http_req_router_t;
};
```

ExpressJS-like routing

And now we can define routes for our server:

```
void add_transform_op_handler(
   const app_params_t & app_params,
   http_req_router_t & router,
   so_5::mbox_t req_handler_mbox )
{
   router.http_get(
      R"(/:path(.*)\.:ext(.{3,4}))",
      restinio::path2regex::options_t{}.strict( true ),
      [req_handler_mbox, &app_params]( auto req, auto params ) {...} );
}
```

ExpressJS-like routing

And now we can define routes for our server:

```
void add_transform_op_hanc
   const app_params_t & app
   http_req_router_t & router,
   so_5::mbox_t req_handler_
{
   router.http_get(
      R"(/:path(.*)\.:ext(.{3,4}))",
      restinio::path2regex::options_t{}.strict( true ),
      [req_handler_mbox, &app_params]( auto req, auto params ) {...} );
}
```

A route with two arguments ('path' and 'ext') is defined here.

sendfile for serving original files

RESTinio supports effective *sendfile* functionality.

It is implemented via `sendfile()` on POSIX and `TransmitFile()` on Windows.

See https://stiffstream.com/en/docs/restinio/0.4/sendfile.html for more details.

Shrimp uses this functionality for serving original files (e.g. files without resizing and conversion).

sendfile for serving original files

```
[[nodiscard]] restinio::request_handling_status_t
serve_as_regular_file(const std::string & root_dir, restinio::request_handle_t req, image_format_t image_format)
{
  const auto full_path = make_full_path( root_dir, req->header().path() );
  try {
    auto sf = restinio::sendfile( full_path );
    const auto last_modified = sf.meta().last_modified_at();

    auto resp = req->create_response();

    return set_common_header_fields_for_image_resp( last_modified, resp )
        .append_header( restinio::http_field::content_type, image_content_type_from_img_format(image_format) )
        .append_header( restinio::http_header_field_t{
            http_header::shrimp_image_src_hf(),
            image_src_to_str( http_header::image_src_t::sendfile ) } )
        .set_body( std::move(sf) )
        .done();
  }
  catch(...) { }

  return do_404_response( std::move( req ) );
}
```

sendfile for serving original files

```
[[nodiscard]] restinio::request_handling_status_t
serve_as_regular_file(const std::string & root_dir, restinio::request_handle_t req, image_format_t image_format)
{
  const auto full_path = make_full_path( root_dir, req->header().path() );
  try {
    auto sf = restinio::sendfile( full_path );
    const auto last_modified = sf.meta().last_modified_at();

    auto resp = req->create_response();

    return set_common_header_fields_fo
        .append_header( restinio::http_fi
        .append_header( restinio::http_h
            http_header::shrimp_image_
            image_src_to_str( http_head
        .set_body( std::move(sf) )
        .done();
  }
  catch(...) { }

  return do_404_response( std::move( req ) );
}
```

sendfile operation is defined here.

sendfile for serving original files

```
[[nodiscard]] restinio::request_handling_status_t
serve_as_regular_file(const std::string & root_dir, restinio::request_handle_t req, image_format_t image_format)
{
  const auto full_path = make_full_path( root_dir, req->header().path() );
  try {
    auto sf = restinio::sendfile( full_path );
    const auto last_modified = sf.meta

    auto resp = req->create_response(

    return set_common_header_fields_
        .append_header( restinio::http_
        .append_header( restinio::http_
            http_header::shrimp_image_src_hr(),
            image_src_to_str( http_header                image_src_t::sendfile ) } )
        .set_body( std::move(sf) )
        .done();
  }
  catch(...) { }

  return do_404_response( std::move( req ) );
}
```

sendfile operation is used here for sending file's content as response's body.

The next big topic

SObjectizer in Shrimp

Actors in Shrimp

Actors are used in Shrimp for actual request handling.

There are two types of actors in Shrimp:

- *transform_manager*

  Receives requests from HTTP-server, delegates requests processing to transformers.
  Owns cache of transformed images and queue of pending requests;
- *transformer*

  Performs actual resizing and/or conversion of images.

Actors in Shrimp

There is just one *transform_manager* actor.

There are several *transformer* actors.

Every *transformer* actor works on its own work thread.

Actors in Shrimp

There is just one *transform_manager* actor.

There are several *transformer* actors.

Every *tran*

A different term is used for **actor** in SObjectizer.

It is **agent**.

When we speak about ***actors*** we mean ***agents*** and vice versa.

transformer agent

transformer is the simplest agent. It handles just one message:

```
class a_transformer_t final : public so_5::agent_t
{
public:
  struct resize_request_t final : public so_5::message_t {...};

  a_transformer_t(context_t ctx, std::shared_ptr<spdlog::logger> logger, storage_params_t cfg);

  void so_define_agent() override {
    so_subscribe_self().event( &a_transformer_t::on_resize_request );
  }

private:
  ...
  on_resize_request(mutable_mhood_t<resize_request_t> cmd);
  ...
};
```

transformer agent

transformer is the simplest agent. It handles just one message:

```
class a_transformer_t final : public so_5::agent_t
{
public:
   struct resize_request_t final : public so_5::message_t {...};

   a_transformer_t(context_t ctx, std::shared_ptr<spdlog::logger> logger, storage_params_t cfg);

   void so_define_agent() override {
      so_subscribe_self().event( &a_transformer_t::
   }

private:
   ...
   on_resize_request(mutable_mhood_t<resize_request_t> cmd);
   ...
};
```

Message to be handled.

transformer agent

transformer is the simplest a

```
class a_transformer_t final : public so_5:
{
public:
  struct resize_request_t final : public so

  a_transformer_t(context_t ctx, std::share

  void so_define_agent() override {
    so_subscribe_self().event( &a_transformer_t::on_resize_request );
  }

private:
  ...
  on_resize_request(mutable_mhood_t<resize_request_t> cmd);
  ...
};
```

Subscription to the message.

When message *resize_request_t* arrives the *on_resize_request()* method will be called.

Type of message to be subscribed is deduced from handler's prototype.

transform_manager agent

transform_manager is more complex than *transformer*.

It handles several message types.

But *transform_manager* is not too complex. It has just one state.

Really complex agents have several states and handle their messages differently in different states.

So *transform_manager* is very simple in that sense.

transform_manager agent (1)

```
class a_transform_manager_t final : public so_5::agent_t
{
public:
   struct resize_request_t final : public so_5::message_t {...};
   struct resize_result_t final : public so_5::message_t {...};
   struct delete_cache_request_t final : public so_5::message_t {...};

   a_transform_manager_t(context_t ctx, std::shared_ptr<spdlog::logger> logger);

   void so_define_agent() override;
   void so_evt_start() override;
   ...
```

transform_manager agent (1)

```
class a_transform_manager_t final : public so_5::agent_t
{
public:
   struct resize_request_t final : public so_5::message_t {...};
   struct resize_result_t final : public so_5::message_t {...};
   struct delete_cache_request_t final : public so_5::message_t {...};

   a_transform_manag                    text_t ctx, std::shared_ptr<spdlog::logger> logger);

   void so_defi
   void so_evt
   ...
```

Those are "public" messages. They are sent to *transform_manager* by external entities:

- *resize_request_t* and *delete_cache_request_t* are sent by HTTP-server;
- *resize_result_t* is sent by *transformer* agent.

transform_manager agent (2)

```
private :
   struct negative_delete_cache_response_t : public so_5::message_t {...};
   struct clear_cache_t final : public so_5::signal_t {};
   struct check_pending_requests_t final : public so_5::signal_t {};
   ...
   void on_resize_request(mutable_mhood_t<resize_request_t> cmd);
   void on_resize_result(mutable_mhood_t<resize_result_t> cmd);
   void on_delete_cache_request(mutable_mhood_t<delete_cache_request_t> cmd);
   void on_negative_delete_cache_response(
      mutable_mhood_t<negative_delete_cache_response_t> cmd);
   void on_clear_cache(mhood_t<clear_cache_t>);
   void on_check_pending_requests(mhood_t<check_pending_requests_t>);
   ...
};
```

transform_manager agent (2)

```
private :
   struct negative_delete_cache_response_t : public so_5::message_t {...};
   struct clear_cache_t final : public so_5::signal_t {};
   struct check_pending_requests_t final : public so_5::signal_t {};
   ...
   void on_resize_reque                    mbood_t<resize_request_t> cmd);
   void on_re
   void on_de
   void on_ne
      mutable_
   void on_cl
   void on_check_pending_requests(mhood_t<check_pending_requests_t>);
   ...
};
```

Those are "private" message and signals. *transform_manager* sends them to itself.

transform_manager agent (3)

```
void a_transform_manager_t::so_define_agent()
{
  so_subscribe_self()
      .event( &a_transform_manager_t::on_resize_request )
      .event( &a_transform_manager_t::on_resize_result )
      .event( &a_transform_manager_t::on_delete_cache_request )
      .event( &a_transform_manager_t::on_negative_delete_cache_response )
      .event( &a_transform_manager_t::on_clear_cache )
      .event( &a_transform_manager_t::on_check_pending_requests );
}
```

Basic working scheme

1. HTTP-server receives HTTP GET request and sends `a_transform_manager_t::resize_request_t` message to *transform_manager*.

Basic working scheme

1. HT

a_ti

trans

```
void handle_resize_op_request(
  const so_5::mbox_t & req_handler_mbox,
  image_format_t image_format,
  const restinio::query_string_params_t & qp,
  restinio::request_handle_t req )
{
  try_to_handle_request( [&]{
      auto op_params = transform::resize_params_t::make(
        restinio::opt_value< std::uint32_t >( qp, "width" ),
        restinio::opt_value< std::uint32_t >( qp, "height" ),
        restinio::opt_value< std::uint32_t >( qp, "max" ) );

      transform::resize_params_constraints_t{}.check( op_params );

      std::string image_path{ req->header().path() };
      so_5::send< so_5::mutable_msg<a_transform_manager_t::resize_request_t> >(
            req_handler_mbox,
            std::move(req), std::move(image_path), image_format, op_params );
    },
    req );
}
```

Basic working scheme

2. *transform_manager* stores new request to pending queue and then delegates request processing to *transformer* by sending `a_transformer_t::resize_request_t` message.

Basic working scheme

2. *tr*

deleg

a_t

```
void a_transform_manager_t::try_initiate_pending_requests_processing()
{
  while( !m_free_workers.empty() && !m_pending_requests.empty() ) {
    auto atoken = m_pending_requests.oldest().value();
    const auto key = atoken.key();

    m_pending_requests.extract_values_for_key( std::move(atoken),
        [&]( auto && value ) {
            m_inprogress_requests.insert( transform::resize_request_key_t{key}, std::move(value) );
         } );

    auto worker = std::move(m_free_workers.top());
    m_free_workers.pop();

    m_logger->trace("initiate processing of a request; request_key={}, worker_mbox={}",
       key, worker->id());

    so_5::send< so_5::mutable_msg<a_transformer_t::resize_request_t> >(worker, key, so_direct_mbox());
  }
}
```

Basic working scheme

3. *transformer* perform all necessary transformations and replies by `a_transform_manager_t::resize_result_t`.

Basic working scheme

3. *tra*

a_ti

```
void a_transformer_t::on_resize_request(
  mutable_mhood_t<resize_request_t> cmd)
{
  auto result = handle_resize_request( cmd->m_key );

  so_5::send< so_5::mutable_msg<a_transform_manager_t::resize_result_t> >(
        cmd->m_reply_to,
        so_direct_mbox(),
        std::move(cmd->m_key),
        std::move(result) );
}
```

Basic working scheme

4. *transform_manager* writes HTTP response. That response will be served by HTTP-server.

Basic working scheme

4. *tro*

by HT

```
void a_transform_manager_t::on_resize_result(
  mutable_mhood_t<resize_result_t> cmd )
{
  m_logger->trace( "resize_result received; request_key={}, worker_mbox={}",
        cmd->m_key,
        cmd->m_worker->id() );

  m_free_workers.push( std::move(cmd->m_worker) );
  try_initiate_pending_requests_processing();

  auto key = std::move(cmd->m_key);
  auto requests = extract_inprogress_requests(
        std::move(m_inprogress_requests.find_first_for_key( key ).value()) );

  std::visit( variant_visitor{
        [&]( successful_resize_t & result ) {
          on_successful_resize( std::move(key), result, std::move(requests) );
        },
        [&]( failed_resize_t & result ) {
          on_failed_resize( std::move(key), result, std::move(requests) );
        } },
        cmd->m_result );
}
```

Basic working scheme

4. *tra*

by HT

```
void a_transform_manager_t::on_resize_result(
   mutable_mhood_t<resize_result_t> cmd )
{
   m_l
   m_f
   try_
   aut
   aut
   std:
         cmd->m_result );
}
```

```
void a_transform_manager_t::on_failed_resize(
   transform::resize_request_key_t key,
   failed_resize_t & result,
   original_request_container_t requests )
{
   m_logger->warn( "failed resize; request_key={}, reason={}", key, result.m_reason );

   for( auto & rq : requests )
   {
      m_logger->trace( "sending negative response back; request_key={}, connection_id={}",
            key, rq->m_http_req->connection_id() );

      do_404_response( std::move(rq->m_http_req) );
   }
}
```

Providing work threads to agents

SObjectizer's dispatchers mechanism is used to provide separate work threads to Shrimp's agents.

Providing work threads to agents (1)

```
[[nodiscard]] so_5::mbox_t
create_agents(
   spdlog::sink_ptr logger_sink,
   const shrimp::app_params_t & app_params,
   so_5::environment_t & env,
   unsigned int worker_threads_count )
{
   using namespace shrimp;
   using namespace so_5::disp::one_thread; // For create_private_disp.

   so_5::mbox_t manager_mbox;

   env.introduce_coop([&]( so_5::coop_t & coop ) {
      auto manager = coop.make_agent_with_binder< a_transform_manager_t >(
            create_private_disp( env, "manager" )->binder(),
            make_logger( "manager", logger_sink ) );
      manager_mbox = manager->so_direct_mbox();
```

Providing work threads to agents (1)

```
[[nodiscard]] so_5::mbox_t
create_agents(
   spdlog::sink_ptr logger_sink,
   const shrimp::app_params_t & app_params,
   so_5::environment_t & env,
   unsigned int worker_threads
{
   using namespace shrimp;
   using namespace so_5::disp

   so_5::mbox_t manager_mbox

   env.introduce_coop([&]( so_5::coop_t & coop ) {
      auto manager = coop.make_agent_with_binder< a_transform_manager_t >(
            create_private_disp( env, "manager" )->binder(),
            make_logger( "manager", logger_sink ) );
      manager_mbox = manager->so_direct_mbox();
```

transform_manager agent is created and is bound to separate one_thread dispatcher. It means that *transform_manager* will have its own work thread.

Providing work threads to agents (2)

```
    // Every worker will work on its own private dispatcher.
    for( unsigned int worker{}; worker < worker_threads_count; ++worker )
    {
       const auto worker_name = fmt::format( "worker_{}", worker );
       auto transformer = coop.make_agent_with_binder< a_transformer_t >(
             create_private_disp( env, worker_name )->binder(),
             make_logger( worker_name, logger_sink ),
             app_params.m_storage );
       manager->add_worker( transformer->so_direct_mbox() );
    }
  } );

  return manager_mbox;
}
```

Providing work threads to agents (2)

```
    // Every worker will work on its own private dispatcher.
    for( unsigned int worker{}; worker < worker_threads_count; ++worker )
    {
       const auto worker_name = fmt::format( "worker_{}", worker );
       auto transformer = coop.make_agent_with_binder< a_transformer_t >(
             create_private_disp( env, worker_name )->binder(),
             make_logger( worker_name, logger_sink ),
             app_params.m_storage );
       manager->add_worker( transformer->so_direct_mbox() );
    }
  } );

  return manager_mbox;
}
```

The same is for every *transformer* agent.

And now the end is near...

Some final words

The results of the experiment

Shrimp was a very interesting and useful experiment for us.

At least two important points can be emphasised...

It's not a 10-lines HelloWorld

Now we have a rather complex example that shows how real-world code on top of *RESTinio* and *SObjectizer* looks like.

That is impossible to show in small HelloWorld-like examples which can be found in almost every web-server- or actor-frameworks.

It's not a 10-lines HelloWorld

For example, it's a classical *RESTinio's* HelloWorld:

```
#include <restinio/all.hpp>

int main()
{
   restinio::run(
      restinio::on_this_thread<>()
        .port(8080)
        .address("localhost")
        .request_handler([](auto req) {
            return req->create_response().set_body("Hello, World!").done();
        }));

   return 0;
}
```

It's not a 10-lines HelloWorld

For example, it's a classical *RESTinio's* HelloWorld:

```
#includ

int mai
{
   restin
      res
        .p
        .a
        .r

        }

   return 0;
}
```

But it is very far from code we have to write for production. Where we have to cope with configuration, logging, error handling and so on..

It's not a 10-lines HelloWorld

For example, it's a classical *RESTinio's* HelloWorld:

```
#includ

int mai
{
   restin
      res
         .p
         .a
         .r

         });

   return 0;
}
```

But it is very far from code we have to write for production.
Where we

Shrimp gives you more objective picture.

New ideas for implementation

We discovered several new ideas for *RESTinio* and *SObjectizer*.

Some of them are already implemented in *RESTinio*.

The bright example is *notificators* from *RESTinio* v.0.4.8.

Some of new ideas are pending for next versions of our OpenSource products.

We are open for new ideas...

We have implemented almost all we wanted in Shrimp.

But if you have an interesting idea for Shrimp or for *RESTinio* or *SObjectizer* feel free to tell us.

References at one place

Shrimp:

https://bitbucket.org/sobjectizerteam/shrimp-demo

https://github.com/Stiffstream/shrimp-demo

RESTinio:

https://stiffstream.com/en/products/restinio.html

SObjectizer:

https://stiffstream.com/en/products/sobjectizer.html

That's all

Thank you!

https://stiffstream.com

info @ stiffstream.com