Your First Application

Contents

Guestbook Application

This tutorial will guide you through building a simple guestbook application using Luminus. The guestbook allows users to leave a message and to view a list of messages left by others. The application will demonstrate the basics of HTML templating, database access, and project architecture.

If you don't have a preferred Clojure editor already, then it's recommended that you use Light Table to follow along with this tutorial.

Using the Docker Image

If you're using Docker, then you can follow these steps to get up and running:

1. `docker pull danboykis/luminus-guestbook`
2. `docker run -p 3000:3000 -p 7000:7000 -it danboykis/luminus-guestbook`

If you prefer to build your own docker image follow the directions here

Installing JDK

Clojure runs on the JVM and requires a copy of JDK to be installed. IF you don't have JDK already on your system then OpenJDK is recommended and can be downloaded here. Note that Luminus requires JDK 8 to work with the default settings.

Installing a Build Tool

Luminus supports the two major build tools, Leiningen or Boot. Either may be installed and this documentation supports both. You can choose which version of the documentation to use by selecting the dropdown here.

In general Leiningen does more for you and therefore is easier to use but more rigid. Boot allows more customization and is more flexible but isn't quite as slick.

If you are unsure which to choose, stick with Leiningen as it is the most popular, and continue reading.

Note: Most of the documentation is equally accurate for Boot, but as of now some pages are not updated to have the boot commands instead of lein ones.

Installing Leiningen is accomplished by followings the step below.

1. Download the script.
2. Set it to be executable. (eg: chmod +x lein)
3. Place it on your $PATH. (eg: ~/bin)
4. Run `lein` and wait for the self-installation to complete.

```
wget https://raw.github.com/technomancy/leiningen/stable/bin/lein
chmod +x lein
mv lein ~/bin
lein
```

Creating a new application

Once you have Leiningen installed you can run the following commands in your terminal to initialize your application:

```
lein new luminus guestbook +h2
cd guestbook
```

The above will create a new template project with the support for H2 embedded database engine.

Anatomy of a Luminus application

The newly created application has the following structure:

```
guestbook
├── Capstanfile
├── Dockerfile
├── Procfile
├── README.md
├── env
│   ├── dev
│   │   ├── clj
│   │   │   ├── guestbook
│   │   │   │   ├── dev_middleware.clj
│   │   │   │   └── env.clj
│   │   │   └── user.cl
│   │   └── resources
│   │       ├── config.edn
│   │       └── logback.xml
│   ├── prod
│   │   ├── clj
│   │   │   └── guestbook
│   │   │       └── env.clj
│   │   └── resources
│   │       ├── config.edn
│   │       └── logback.xml
│   └── test
│       └── resources
│           └── config.edn
├── dev-config.edn
├── test-config.edn
├── project.clj
├── resources
│   ├── docs
│   │   └── docs.md
│   ├── migrations
│   │   ├── 20160811175305-add-users-table.down.sql
│   │   └── 20160811175305-add-users-table.up.sql
│   ├── public
│   │   ├── css
│   │   │   └── screen.css
│   │   ├── favicon.ico
│   │   ├── img
│   │   └── js
│   ├── sql
│   │   └── queries.sql
│   └── html
│       ├── about.html
│       ├── base.html
│       ├── error.html
```

```
│       └── home.html
├── src
│   └── clj
│       └── guestbook
│           ├── config.clj
│           ├── core.clj
│           ├── db
│           │   └── core.clj
│           ├── handler.clj
│           ├── layout.clj
│           ├── middleware.clj
│           └── routes
│               └── home.clj
└── test
    └── clj
        └── guestbook
            └── test
                ├── db
                │   └── core.clj
                └── handler.clj
```

Let's take a look at what the files in the root folder of the application do:

- `project.clj` - used to manage the project configuration and dependencies by Leiningen
- `Capstanfile` - used to facilitate OSv deployments
- `Dockerfile` - used to facilitate Docker container deployments
- `Procfile` - used to facilitate Heroku deployments
- `README.md` - where documentation for the application is conventionally put
- `dev-config.edn` - used for local development configuration that should not be checked into the code repository
- `test-config.edn` - used for test development configuration that should not be checked into the code repository
- `.gitignore` - a list of assets, such as build generated files, to exclude from Git

The Source Directory

All our code lives under the `src/clj` folder. Since our application is called guestbook, this is the root namespace for the project. Let's take a look at all the namespaces that have been created for us.

GUESTBOOK

`core.clj` - this is the entry point for the application that contains the logic for starting and stopping the server

`handler.clj` - defines the base routes for the application, this is the entry point into the application

`layout.clj` - a namespace for the layout helpers used to render the content for our pages

`middleware.clj` - a namespace that contains custom middleware for the application

GUESTBOOK.DB

The `db` namespace is used to define the model for the application and handle the persistence layer.

`core.clj` - used to house the functions for interacting with the database

GUESTBOOK.ROUTES

The `routes` namespace is where the routes and controllers for our home and about pages are located. When you add more routes, such as authentication, or specific workflows you should create namespaces for them here.

`home.clj` - a namespace that defines the home and about pages of the application

The Env Directory

Environment specific code and resources are located under the `env/dev`, `env/test`, and the `env/prod` paths. The `dev` configuration will be used during development, `test` during testing, while the `prod` configuration will be used when the application is packaged for production.

dev/clj

`user.clj` - a utility namespace for any code you wish to run during REPL development

`guestbook/env.clj` - contains the development configuration defaults

`guestbook/dev_middleware.clj` - contains middleware used for development that should not be compiled in production

dev/resources

`config.edn` - default environment variables for the development

`logback.xml` file used to configure the development logging profile

test/resources

`config.edn` - default environment variables for testing

prod/clj

`guestbook/env.clj` namespace with the production configuration

prod/resources

`config.edn` - default environment variables that will be packaged with the application

`logback.xml` - default production logging configuration

The Test Directory

Here is where we put tests for our application, a couple of sample tests have already been defined for us.

The Resources Directory

This is where we put all the static resources for our application. Content in the `public` directory under `resources` will be served to the clients by the server. We can see that some CSS resources have already been created for us.

HTML TEMPLATES

The `resources/html` directory is reserved for the Selmer templates that represent the application pages.

`about.html` - about page

`base.html` - base layout for the site

`home.html` - home page

`error.html` - error page template

SQL QUERIES

The SQL queries are found in the `resources/sql` folder.

`queries.sql` - defines the SQL queries and their associated function names

THE MIGRATIONS DIRECTORY

Luminus uses Migratus for migrations. Migrations are managed using up and down SQL files. The files are conventionally versioned using the date and will be applied in order of their creation.

`20150718103127-add-users-table.up.sql` - migrations file to create the tables

`20150718103127-add-users-table.down.sql` - migrations file to drop the tables

The Project File

As was noted above, all the dependencies are managed via updating the `project.clj` file. The project file of the application we've created is found in its root folder and should look as follows:

```
(defproject guestbook "0.1.0-SNAPSHOT"
  :description "FIXME: write description"
  :url "http://example.com/FIXME"
  :dependencies [[cheshire "5.8.1"]
                 [clojure.java-time "0.3.2"]
                 [com.h2database/h2 "1.4.197"]
                 [compojure "1.6.1"]
                 [conman "0.8.3"]
                 [cprop "0.1.13"]
                 [funcool/struct "1.3.0"]
                 [luminus-immutant "0.2.4"]
                 [luminus-migrations "0.6.3"]
                 [luminus-transit "0.1.1"]
                 [luminus/ring-ttl-session "0.3.2"]
                 [markdown-clj "1.0.5"]
                 [metosin/muuntaja "0.6.3"]
                 [metosin/ring-http-response "0.9.1"]
                 [mount "0.1.15"]
                 [nrepl "0.5.3"]
                 [org.clojure/clojure "1.10.0"]
                 [org.clojure/tools.cli "0.4.1"]
                 [org.clojure/tools.logging "0.4.1"]
                 [org.webjars.bower/tether "1.4.4"]
                 [org.webjars/bootstrap "4.1.3"]
                 [org.webjars/font-awesome "5.6.1"]
                 [org.webjars/jquery "3.3.1-1"]
                 [org.webjars/webjars-locator "0.34"]
                 [ring-webjars "0.2.0"]
                 [ring/ring-core "1.7.1"]
                 [ring/ring-defaults "0.3.2"]
                 [selmer "1.12.5"]]

  :min-lein-version "2.0.0"

  :source-paths ["src/clj"]
  :test-paths ["test/clj"]
  :resource-paths ["resources"]
  :target-path "target/%s/"
  :main ^:skip-aot guestbook.core

  :plugins [[lein-immutant "2.1.0"]]

  :profiles
  {:uberjar {:omit-source true
             :aot :all
             :uberjar-name "guestbook.jar"
             :source-paths ["env/prod/clj"]
             :resource-paths ["env/prod/resources"]}

   :dev           [:project/dev :profiles/dev]
   :test          [:project/dev :project/test :profiles/test]

   :project/dev  {:jvm-opts ["-Dconf=dev-config.edn"]
                  :dependencies [[expound "0.7.2"]
                                 [pjstadig/humane-test-output "0.9.0"]
                                 [prone "1.6.1"]
                                 [ring/ring-devel "1.7.1"]
                                 [ring/ring-mock "0.3.2"]]
                  :plugins      [[com.jakemccrary/lein-test-refresh "0

                  :source-paths ["env/dev/clj"]
                  :resource-paths ["env/dev/resources"]
                  :repl-options {:init-ns user}
                  :injections [(require 'pjstadig.humane-test-output)
```

```
                                (pjstadig.humane-test-output/activate!)
    :project/test {:jvm-opts ["-Dconf=test-config.edn"]
                   :resource-paths ["env/test/resources"]}
    :profiles/dev {}
    :profiles/test {}})
```

As you can see the `project.clj` file is simply a Clojure list containing key/value pairs describing different aspects of the application.

The most common task is adding new libraries to the project. These libraries are specified using the `:dependencies` vector. In order to use a new library in our project we simply have to add its dependency here.

The items in the `:plugins` vector can be used to provide additional functionality.

The `:profiles` contain a map of different project configurations that are used to initialize it for either development or production builds.

Note that the project sets up composite profiles for `:dev` and `:test`. These profiles contain the variables from `:project/dev` and `:project/test` profiles, as well as from `:profiles/dev` and `:profiles/test` found in the `profiles.clj`. The latter can be used for additional local configuration that is not meant to be checked into the shared code repository.

Please refer to the official Leiningen documentation for further details on structuring the `project.clj` build file.

Creating the Database

First, we will create a model for our application, to do that we'll open up the `<date>-add-users-table.up.sql` file located under the `migrations` folder. The file has the following contents:

```
CREATE TABLE users
(id VARCHAR(20) PRIMARY KEY,
 first_name VARCHAR(30),
 last_name VARCHAR(30),
 email VARCHAR(30),
 admin BOOLEAN,
 last_login TIME,
 is_active BOOLEAN,
 pass VARCHAR(300));
```

We'll replace the `users` table with one that's more appropriate for our application:

```
CREATE TABLE guestbook
(id INTEGER PRIMARY KEY AUTO_INCREMENT,
 name VARCHAR(30),
 message VARCHAR(200),
 timestamp TIMESTAMP);
```

The guestbook table will store all the fields describing the message, such as the name of the commenter, the content of the message and a timestamp. Next, let's replace the contents of the `<date>-add-users-table.down.sql` file accordingly:

```
DROP TABLE guestbook;
```

We can now run the migrations using the following command from the root of our project:

```
lein run migrate
```

If everything went well we should now have our database initialized.

Accessing The Database

Next, we'll take a look at the `src/clj/guestbook/db/core.clj` file. Here, we can see that we already have the definition for our database connection.

```
(ns guestbook.db.core
  (:require
    [conman.core :as conman]
    [mount.core :refer [defstate]]
    [guestbook.config :refer [env]]))

(defstate ^:dynamic *db*
           :start (conman/connect! {:jdbc-url (env :database-url)})
           :stop (conman/disconnect! *db*))

(conman/bind-connection *db* "sql/queries.sql")
```

The database connection is read from the environment map at runtime. By default, the `:database-url` key points to a string with the connection URL for the database. This variable is populated from the `dev-config.edn` file during development and has to be set as an environment variable for production, e.g:

```
export DATABASE_URL="jdbc:h2:./guestbook.db"
```

Since we're using the embedded H2 database, the data is stored in a file specified in the URL that's found in the path relative to where the project is run.

The functions that map to database queries are generated when `bind-connection` is called. As we can see it references the `sql/queries.sql` file. This location is found under the `resources` folder. Let's open up this file and take a look inside.

```
-- :name create-user! :! :n
-- :doc creates a new user record
INSERT INTO users
```

```
(id, first_name, last_name, email, pass)
VALUES (:id, :first_name, :last_name, :email, :pass)

-- :name update-user! :! :n
-- :doc update an existing user record
UPDATE users
SET first_name = :first_name, last_name = :last_name, email = :email
WHERE id = :id

-- :name get-user :? :1
-- :doc retrieve a user given the id.
SELECT * FROM users
WHERE id = :id
```

As we can see each function is defined using the comment that starts with `-- :name` followed by the name of the function. The next comment provides the doc string for the function and finally we have the body that's plain SQL. The parameters are denoted using `:` notation. Let's replace the existing queries with some of our own:

```
-- :name save-message! :! :n
-- :doc creates a new message
INSERT INTO guestbook
(name, message, timestamp)
VALUES (:name, :message, :timestamp)

-- :name get-messages :? :*
-- :doc selects all available messages
SELECT * FROM guestbook
```

Now that our model is all setup, let's start up the application.

Running the Application

We can run our application in development mode as follows:

```
>lein run
[2016-02-28 15:05:34,970][DEBUG][org.jboss.logging] Logging Provider:
[2016-02-28 15:05:36,067][INFO][com.zaxxer.hikari.HikariDataSource] Hi
[2016-02-28 15:05:36,252][INFO][luminus.http-server] starting HTTP ser
[2016-02-28 15:05:36,294][INFO][org.xnio] XNIO version 3.4.0.Beta1
[2016-02-28 15:05:36,344][INFO][org.xnio.nio] XNIO NIO Implementation
[2016-02-28 15:05:36,406][INFO][org.projectodd.wunderboss.web.Web] Reg
[2016-02-28 15:05:36,407][INFO][luminus.repl-server] starting nREPL se
[2016-02-28 15:05:36,422][INFO][guestbook.core] #'guestbook.config/env
[2016-02-28 15:05:36,422][INFO][guestbook.core] #'guestbook.db.core/*d
[2016-02-28 15:05:36,422][INFO][guestbook.core] #'guestbook.core/http-
[2016-02-28 15:05:36,423][INFO][guestbook.core] #'guestbook.core/repl-
[2016-02-28 15:05:36,423][INFO][guestbook.env]
-=[guestbook started successfully using the development profile]=-
```

Once server starts, you should be able to navigate to http://localhost:3000 and see the app running. The server can be started on an alternate port by either passing it as a parameter as seen below, or setting the `PORT` environment variable.

```
lein run -p 8000
```

Alternatively, you can start the application from the REPL using `start` function defined in the `user` namespace, e.g:

```
lein repl

2018-01-30 15:48:31,147 [main] DEBUG org.jboss.logging - Logging Provi
nREPL server started on port 51655 on host 127.0.0.1 - nrepl://127.0.0
REPL-y 0.3.7, nREPL 0.2.12
Clojure 1.9.0
Java HotSpot(TM) 64-Bit Server VM 1.8.0_45-b14
    Docs: (doc function-name-here)
          (find-doc "part-of-name-here")
  Source: (source function-name-here)
 Javadoc: (javadoc java-object-or-class-here)
    Exit: Control+D or (exit) or (quit)
 Results: Stored in vars *1, *2, *3, an exception in *e

user=>(start)
018-01-30 15:48:58,211 [nREPL-worker-0] INFO  guestbook.env -
-=[guestbook started successfully using the development profile]=-
2018-01-30 15:48:58,505 [nREPL-worker-0] INFO  luminus.http-server - s
2018-01-30 15:48:58,547 [nREPL-worker-0] DEBUG io.undertow - starting
2018-01-30 15:48:58,593 [nREPL-worker-0] INFO  org.xnio - XNIO version
2018-01-30 15:48:58,707 [nREPL-worker-0] DEBUG io.undertow - Configuri
2018-01-30 15:48:58,745 [nREPL-worker-0] INFO  org.projectodd.wunderbo
{:started ["#'guestbook.config/env" "#'guestbook.handler/init-app" "#'
```

Note that the page is prompting us to run the migrations in order to initialize the database. However, we've already done that earlier, so we won't need to do that again.

Creating Pages and Handling Form Input

Our routes are defined in the `guestbook.routes.home` namespace. Let's open it up and add the logic for rendering the messages from the database. We'll first need to add a reference to our `db` namespace along with references for Bouncer validators and ring.util.response

```
(ns guestbook.routes.home
  (:require
    ...
    [ring.util.http-response :as response]
    [clojure.java.io :as io]
    [struct.core :as st]))
```

Next, we'll create a schema that defines the form parameters and add a function to validate them:

```
(def message-schema
  [[:name
    st/required
    st/string]
```

```
    [:message
     st/required
     st/string
     {:message "message must contain at least 10 characters"
      :validate #(> (count %) 9)}]])

(defn validate-message [params]
  (first (st/validate params message-schema)))
```

The function uses the `validate` function from the Struct library to check that the `:name` and the `:message` keys conform to the rules we specified. Specifically, the name is required and the message must contain at least 10 characters. Struct uses a vector to specify the fields being validated where each field is itself a vector starting with the keyword pointing to the value being validated followed by one or more validators. Custom validators can be specified using a map as seen with with the validator for the character count in the message.

We'll now add a function to validate and save messages:

```
(defn save-message! [{:keys [params]}]
  (if-let [errors (validate-message params)]
    (-> (response/found "/")
        (assoc :flash (assoc params :errors errors)))
    (do
      (db/save-message!
       (assoc params :timestamp (java.util.Date.)))
      (response/found "/"))))
```

The function will grab the `:params` key from the request that contains the form parameters. When the `validate-message` functions returns errors we'll redirect back to `/`, we'll associate a `:flash` key with the response where we'll put the supplied parameters along with the errors. Otherwise, we'll save the message in our database and redirect.

We can now change the `home-page` handler function to look as follows:

```
(defn home-page [{:keys [flash]}]
  (layout/render
   "home.html"
   (merge {:messages (db/get-messages)}
          (select-keys flash [:name :message :errors]))))
```

The function renders the home page template and passes it the currently stored messages along with any parameters from the `:flash` session, such as validation errors.

Recall that the database accessor functions were automatically generated for us by the `(conman/bind-connection *db* "sql/queries.sql")` statement ran in the `guestbook.db.core` namespace. The names of these functions are inferred from the `-- :name` comments in the SQL templates found in the `resources/sq/queries.sql` file.

Our routes will now have to pass the request to both the `home-page` and the `save-message!` handlers:

```
(defroutes home-routes
  (GET "/" request (home-page request))
  (POST "/" request (save-message! request))
  (GET "/about" [] (about-page)))
```

Don't forget to refer `POST` from `compojure.core`

```
(ns guestbook.routes.home
  (:require ...
            [compojure.core :refer [defroutes GET POST]]
            ...))
```

Now that we have our controllers setup, let's open the `home.html` template located under the `resources/html` directory. Currently, it simply renders the contents of the `content` variable inside the content block:

```
{% extends "base.html" %}
{% block content %}
  <div class="jumbotron">
    <h1>Welcome to guestbook</h1>
    <p>Time to start building your site!</p>
    <p><a class="btn btn-primary btn-lg" href="http://luminusweb.net">
  </div>

  <div class="row">
    <div class="span12">
    {{docs|markdown}}
    </div>
  </div>
{% endblock %}
```

We'll update our `content` block to iterate over the messages and print each one in a list:

```
{% extends "base.html" %}
{% block content %}
  <div class="jumbotron">
    <h1>Welcome to guestbook</h1>
    <p>Time to start building your site!</p>
    <p><a class="btn btn-primary btn-lg" href="http://luminusweb.net">
  </div>

  <div class="row">
    <div class="span12">
        <ul class="messages">
            {% for item in messages %}
            <li>
                <time>{{item.timestamp|date:"yyyy-MM-dd HH:mm"}}</time
                <p>{{item.message}}</p>
                <p> - {{item.name}}</p>
            </li>
            {% endfor %}
        </ul>
    </div>
</div>
{% endblock %}
```

As you can see above, we use a `for` iterator to walk the messages. Since each message is a map with the message, name, and timestamp keys, we can access them by name. Also, notice the use of the `date` filter to format the timestamps into a human readable form.

Finally, we'll create a form to allow users to submit their messages. We'll populate the name and message values if they're supplied and render any errors associated with them. Note that the forms also uses the `csrf-field` tag that's required for cross-site request forgery protection.

```
<div class="row">
    <div class="span12">
        <form method="POST" action="/">
                {% csrf-field %}
                <p>
                    Name:
                    <input class="form-control"
                           type="text"
                           name="name"
                           value="{{name}}" />
                </p>
                {% if errors.name %}
                <div class="alert alert-danger">{{errors.name|join}}</
                {% endif %}
                <p>
                    Message:
                <textarea class="form-control"
                          rows="4"
                          cols="50"
                          name="message">{{message}}</textarea>
                </p>
                {% if errors.message %}
                <div class="alert alert-danger">{{errors.message|join}
                {% endif %}
                <input type="submit" class="btn btn-primary" value="co
        </form>
    </div>
</div>
```

Our final `home.html` template should look as follows:

```
{% extends "base.html" %}
{% block content %}
<div class="row">
    <div class="span12">
        <ul class="messages">
            {% for item in messages %}
            <li>
                <time>{{item.timestamp|date:"yyyy-MM-dd HH:mm"}}</time
                <p>{{item.message}}</p>
                <p> - {{item.name}}</p>
            </li>
            {% endfor %}
        </ul>
    </div>
</div>
<div class="row">
    <div class="span12">
        <form method="POST" action="/">
```

```
                {% csrf-field %}
                <p>
                    Name:
                    <input class="form-control"
                           type="text"
                           name="name"
                           value="{{name}}" />
                </p>
                {% if errors.name %}
                <div class="alert alert-danger">{{errors.name|join}}</
                {% endif %}
                <p>
                    Message:
                <textarea class="form-control"
                          rows="4"
                          cols="50"
                          name="message">{{message}}</textarea>
                </p>
                {% if errors.message %}
                <div class="alert alert-danger">{{errors.message|join}
                {% endif %}
                <input type="submit" class="btn btn-primary" value="co
        </form>
    </div>
</div>
{% endblock %}
```

Finally, we can update the `screen.css` file located in the `resources/public/css` folder to format our form nicer:

```
html,
body {
        font-family: 'Helvetica Neue', Helvetica, Arial, sans-serif;
  height: 100%;
  line-height: 1.4em;
        background: #eaeaea;
        width: 520px;
        margin: 0 auto;
}
.navbar {
  margin-bottom: 10px;
}
.navbar-brand {
  float: none;
}
.navbar-nav .nav-item {
  float: none;
}
.navbar-divider,
.navbar-nav .nav-item+.nav-item,
.navbar-nav .nav-link + .nav-link {
  margin-left: 0;
}
@media (min-width: 34em) {
  .navbar-brand {
    float: left;
  }
  .navbar-nav .nav-item {
    float: left;
  }
  .navbar-divider,
  .navbar-nav .nav-item+.nav-item,
```

```
  .navbar-nav .nav-link + .nav-link {
    margin-left: 1rem;
  }
}

.messages {
  background: white;
  width: 520px;
}
ul {
        list-style: none;
}

ul.messages li {
        position: relative;
        font-size: 16px;
        padding: 5px;
        border-bottom: 1px dotted #ccc;
}

li:last-child {
        border-bottom: none;
}

li time {
        font-size: 12px;
        padding-bottom: 20px;
}

form, .error {
        padding: 30px;
        margin-bottom: 50px;
        position: relative;
  background: white;
}
```

When we reload the page in the browser we should be greeted by the guestbook page. We can test that everything is working as expected by adding a comment in our comment form.

Adding some tests

Now that we have our application working we can add some tests for it. Let's open up the `test/clj/guestbook/test/db/core.clj` namespace and update it as follows:

```
(ns guestbook.test.db.core
  (:require [guestbook.db.core :refer [*db*] :as db]
            [luminus-migrations.core :as migrations]
            [clojure.test :refer :all]
            [clojure.java.jdbc :as jdbc]
            [guestbook.config :refer [env]]
            [mount.core :as mount]))

(use-fixtures
  :once
  (fn [f]
    (mount/start
      #'guestbook.config/env
      #'guestbook.db.core/*db*)
    (migrations/migrate ["migrate"] (select-keys env [:database-url]))
    (f)))
```

```
(deftest test-message
  (jdbc/with-db-transaction [t-conn *db*]
    (jdbc/db-set-rollback-only! t-conn)
    (let [timestamp (org.joda.time.DateTime. org.joda.time.DateTimeZon
      (is (= 1 (db/save-message!
                t-conn
                {:name "Bob"
                 :message "Hello, World"
                 :timestamp timestamp}
                {:connection t-conn})))
      (is (=
           {:name "Bob"
            :message "Hello, World"
            :timestamp timestamp}
           (-> (db/get-messages t-conn {})
               (first)
               (select-keys [:name :message :timestamp])))))))
```

We can now run

`lein test` in the terminal to see that our database interaction works as expected.

Luminus comes with lein-test-refresh enabled by default. This plugin allows running tests continuously whenever a change in a namespace is detected. We can start a test runner in a new terminal using the `lein test-refresh` command.

Packaging the application

The application can be packaged for standalone deployment by running the following command:

```
lein uberjar
```

This will create a runnable jar that can be run as seen below:

```
export DATABASE_URL="jdbc:h2:./guestbook_dev.db"
java -jar target/uberjar/guestbook.jar
```

Note that we have to supply the `DATABASE_URL` environment variable when running as a jar, as it's not packaged with the application.

Complete source listing for the tutorial is available here.