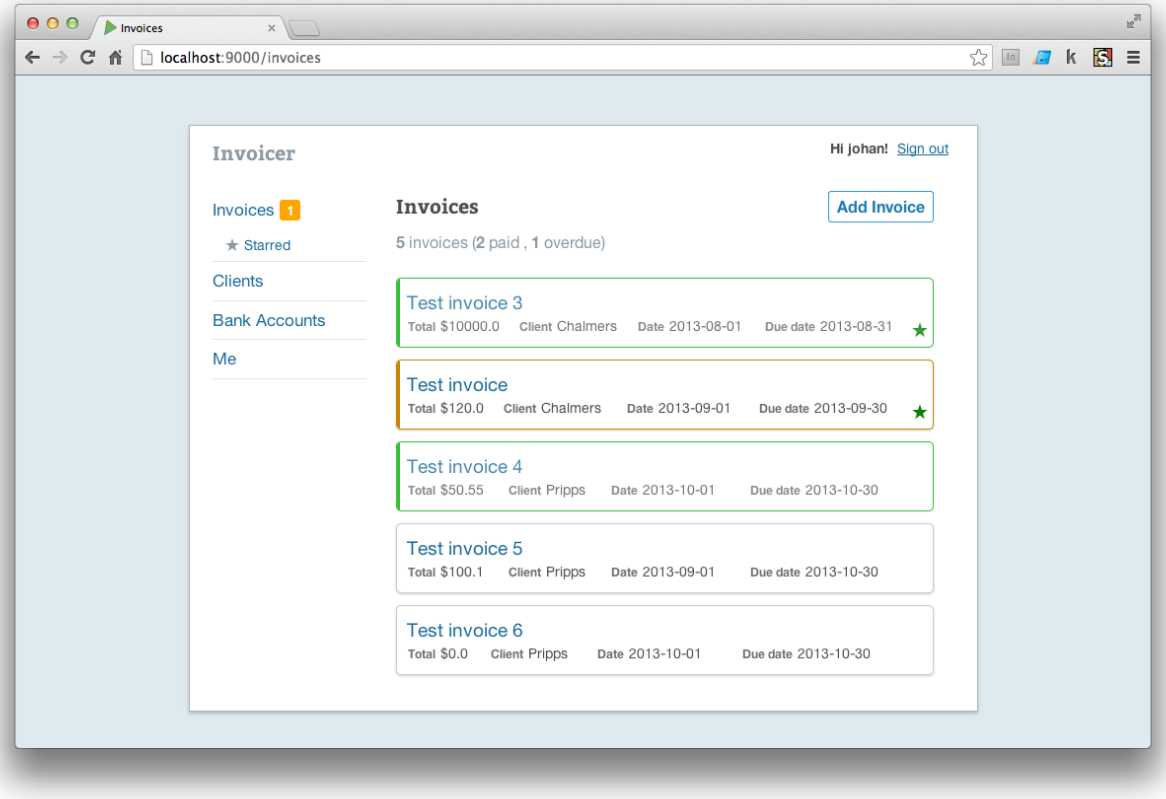
Project Document

Invoicer

Web Applications course (DAT076)
at Chalmers University of Technology



Group Members (group 8)
Johan Brook (900720-0216) – johan.jvb@gmail.com
Henrik Andersson (881024-1631) – henan@student.chalmers.se
Robin Andersson, (900122-0574) – k.robin.andersson@gmail.com
Andreas Rolén (930208-4570) – andreasrolen93@gmail.com

Project overview

For the Web Applications course we have created an application, “Invoicer”, for managing invoices and their clients. Users can add, edit, and remove invoices and clients, as well as connecting different bank accounts to an invoice. The user can get notified in the UI about paid and overdue invoices, as well as sending out invoices and reminders to the clients with e-mail.

A separate component based JSF project which acts as payment app for clients has been created, which communicates with the main application over the built-in REST API using JSON. Notifications about paid invoices are instantly appearing in real-time in the main application using Javascript and Server Sent Events (SSE).

Technical overview

Code repository

GitHub has been used as the main repository for the project, residing at this URL:
https://github.com/johanbrook/dat076-webapps
The secondary JSF project used as a client app is available in a separate repo:
https://github.com/henke101/invoicer_client

For the main project, the `master` branch is considered stable, and `develop` is considered unstable.

Framework

The Java Play framework has been used for building the application. The group chose the framework thanks to its modern features, strong MVC approach, clean approach to Java web development, convention over configuration-style, and sense of “developer happiness” and productivity.

Highlighted features of Play:

- RESTful and MVC
- Compile on browser refresh
- EBean ORM support
- Automated SQL generated from model classes (“evolutions”)
- Good testing support (unit, integration and functional testing)
- Real-time support (WebSockets, SSE, Comet)
- Console based (run, clean, test, compile from a command prompt)
- Intuitive action routing
- Managing dependencies with *sbt*

- Actively developed

Summarized, Play follows common standards for building modern web applications, and has a strong community. The group felt we wanted to use this tool instead of the Java EE technologies taught in the course due to personal taste and an strong desire to try out the framework.

System overview

All application specific code reside in the `app` directory by convention. Further, the modules are further divided in Java packages:

- `models`
  The model classes: entities which should be persisted to the database. Includes data fields with associations, validations, and some helper methods.
- `controllers`
  Each model has a corresponding controller class which handles common CRUD operations in a RESTful way.
- `views`
  Organized per controller, and includes HTML and JS (see more below) templates for most actions in the corresponding controller. The template language is plain Scala.
- `service`
  Contains loosely coupled services, such as sending e-mails, which can be injected into controllers.
- `util`
  General utilities package.

The `conf` directory includes configuration files, such as error messages, database seed data, database evolution scripts, and HTTP routes.

Models

The model classes inherit a super class for convenience, and solving as much data validation as they can with annotations. The ORM used in Play, Ebean, provides a powerful set of methods for fetching records without writing any SQL (worth noting is that JPA, Hibernate, or any other persistence framework can be used instead of EBean – the Play framework is agnostic).

Controllers

The controllers works as a “glue” between the routes, models and the views. They receive the incoming requests, perhaps querying their model for data, and render the results to a view (may be HTML, JSON or Javascript). The controllers also deal with authentication, sending e-mails, handling form errors – really all logic in the application. Their actions (often) represent a RESTful action: the index, show, new, create, update, destroy actions are present in almost all controllers in the application.

Views

The view layer consists mainly of HTML templates where data from the controllers are rendered. They are organized according to their controller and action, i.e. the Invoices controller has a view package named `views.invoices`, and the template for listing all invoices is called `index.scala` – the same as the controller action. The views contains

These naming conventions make it easy to navigate the app structure and keeping things in order.

HTTP Routing

How to route HTTP methods and URLs to specific controller actions is pretty intuitive. The `routes` file in the `conf` directory contains a listing of HTTP methods, URLs and controller actions following the pattern: `METHOD /url controllers.Controller.action().`

Example:

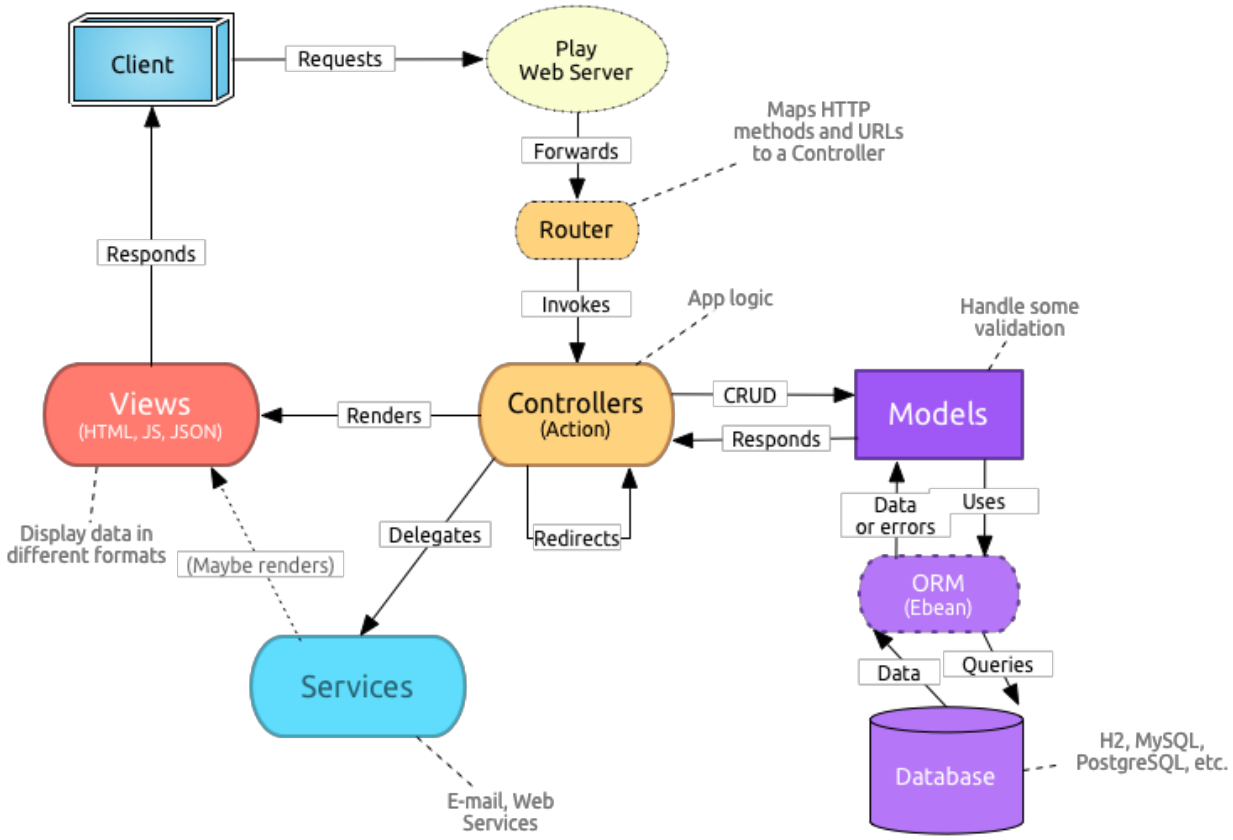
```
GET   /invoices              controllers.Invoices.index()
GET   /invoices/new          controllers.Invoices.newInvoice()
POST  /invoices              controllers.Invoices.create()
GET   /invoices/:id          controllers.Invoices.show(id: Long)
```

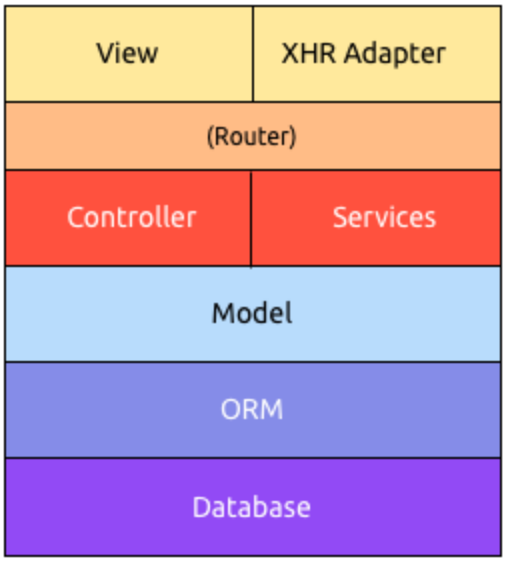
We found this way of managing routes very unobtrusive, centralized and managable.

Application flow

Below is a high-level diagram for a typical request lifecycle in the application. Note the separate layers

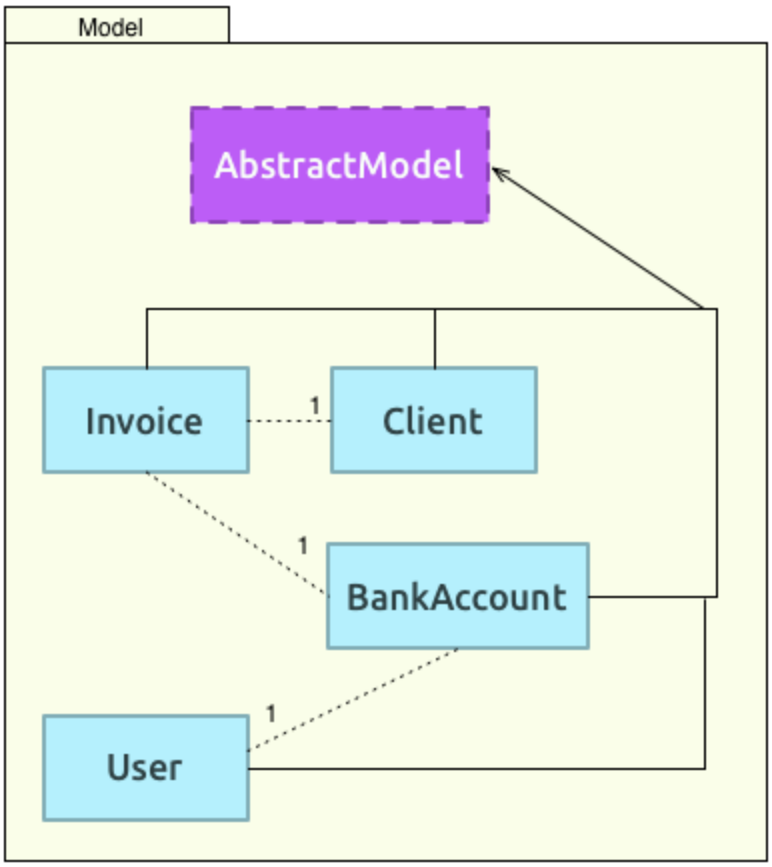
and clear MVC structure. Far down is a layered view of the system.





Domain model

There are four concrete model classes in the application: `Invoice, User, BankAccount` and `Client`. Below is a entity diagram showing all associations:



Content negotiation

In Play 2.x, there is no built-in content negotiation support, JAX-RS style. Instead, a “responder” pattern was used in order to deliver the correct response for a certain request. By using an interface with public methods specifying response types (Javascript, JSON, HTML) and letting the controller actions respond with an anonymous instance of the interface, it was possible to check the “Accepts” header for a content type to match.

Unobtrusive XHR adapter and Javascript responses

We chose not to implement a Javascript based middle-layer between the UI and the REST backend. The reason was that it would just become a redundant middle-layer, not doing anything special other than sending a asynchronous request to an API endpoint, and parsing the response as JSON (or any other format), and at last updating the UI. It creates an obtrusive blockade with ad-hoc logic to maintain.

Instead a concept was used where as little obtrusive Javascript code as possible was written, and still

made use of the service based approach and our REST API. To its core, the adapter hooks into every link or form element specifying a custom attribute (and optionally HTTP method and content type), and is then doing a asynchronous request to the URL specified in the link or form's `href` or `action` attribute, respectively. This means the adapter can be disabled or removed, and the functionality will still be there, just being synchronous (request based approach). The content type defaults to `application/script`, which means the server may respond as Javascript (see below).

Javascript responses

A core idea of MVC is that one should be able to exchange the view for another, and things will still work. The view is just another representation of the data. The controller responses may be seen in exactly the same way: we don't have to respond as HTML all the time. This has been shown when switching between rendering JSON or XML for the same API endpoint. In the application, a third option is used: responding as Javascript.

By using Javascript Scala templates (named `action.scala.js` in the `views` package) regular server code is injected into the Javascript, making it possible to render and re-use templates and data directly into the Javascript, which can update the UI and other things.

Example:

For marking invoices as starred, a link element like this is present (using the unobtrusive XHR adapter described above):

```
<a href="@routes.Invoices.toggleStarred(invoice.id)"
       data-remote="true"
       data-method="put"
       class="star @if(invoice.starred) {starred}">★</a>
```

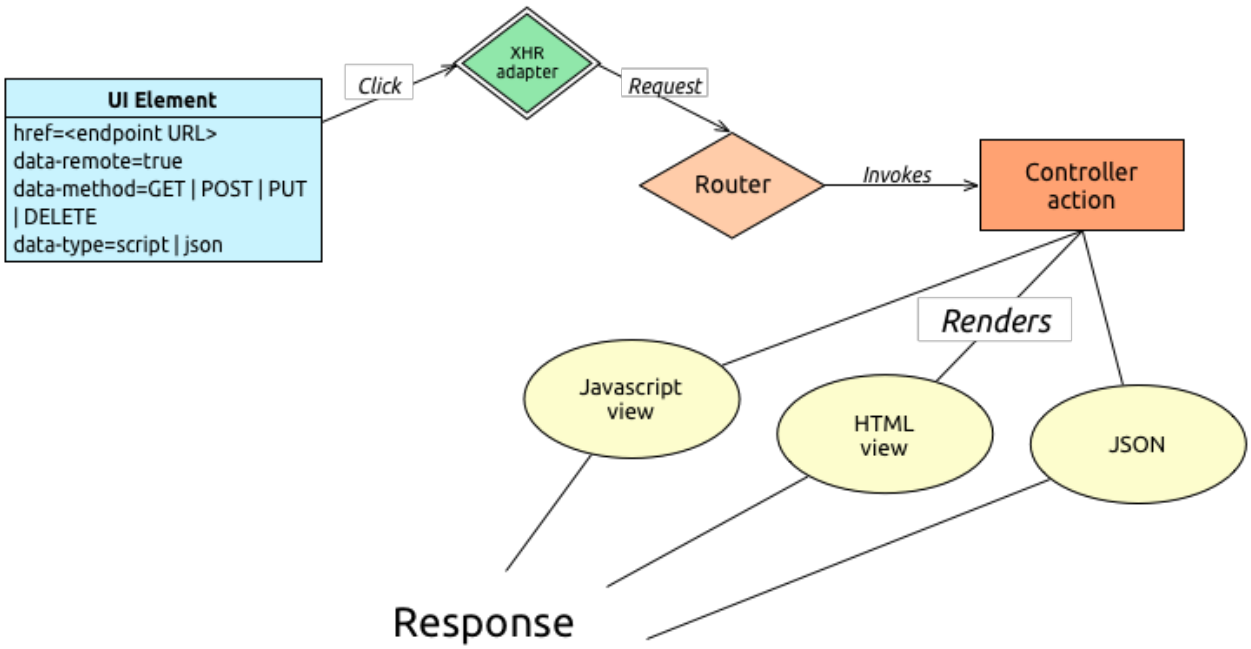
The HTTP method is specified as `PUT` (the invoice is being *updated*) and will route to the `toggleStarred` action in the `Invoices` controller. The action will do its work and respond with Javascript, if possible. It looks like this in the end of the action:

```
@Override
public Result script() {
       return ok(views.js.invoices.starred.render(invoice));
}
```

The `starred` Javascripte template is rendered with an invoice:

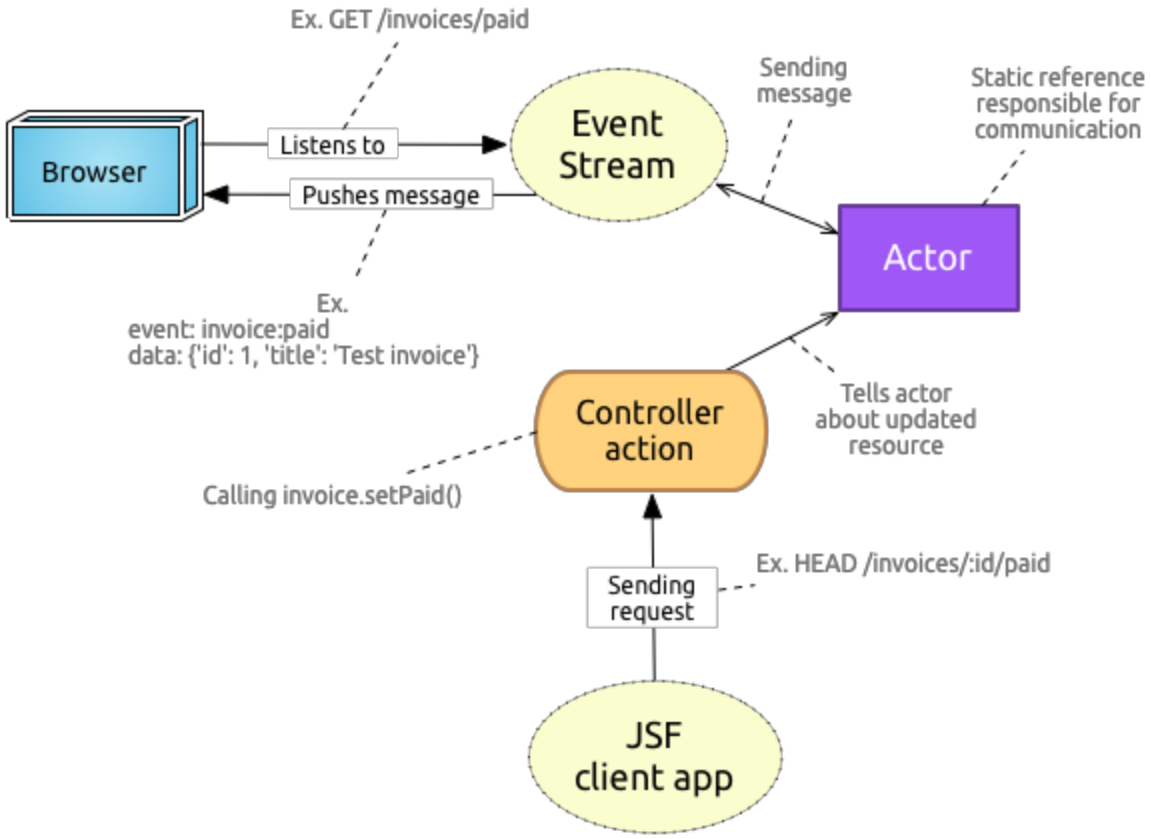
```
@(invoice : Invoice)

$("#invoice-@invoice.id").find(".star").toggleClass("starred");
```

The template works as any ordinary Play template: the invoice's containing element is looked up in the UI and the previously mentioned link element has its class toggled. This approach is simple, consistent, and removes the use of a client side middle-layer sitting between our API and the client.



Server-Sent Events and the JSF client interface application

Real-time pushes from the server is implemented in the application with use of Server-Sent Events (SSE). The front-end is listening to a server endpoint using the Javascript EventSource API, and when a change on the server happens it is pushed to listening browsers.



A realistic use case was formed: a separate JSF based application where clients can pay their invoices. It follows the component pattern where the client's invoices is fetched as JSON from the main application API, and is shown in the interface. The client can "pay" their invoices, and a request to the main application is made, marking the invoice as paid and pushing a JSON payload with SSE down to listening browsers in real time, where they show a notification. An *Actor* from the real-time framework Akka was used for communication and message passing between the relevant controller actions.

Mailing and Dependency Injection

The application is capable of sending e-mails to clients for various events, such as whole invoices or reminders. A general mailer interface is used for abstraction, and may be implemented by different mailing implementations working with perhaps different mail servers. The application is currently using a mailer which is sending traffic with Google's Gmail SMTP servers.

The mailer instance itself is injected in relevant controllers with dependency injection (utilizing Google's Guice DI framework). The body parts of the e-mails are standalone HTML templates which functions as every other view in the app.

Import clients, bank accounts and invoices with JSON file uploads

The application supports file uploads of three models (Invoice, BankAccount and Client) written as correctly formated JSON files. The application uses the standard way of accomplishing the upload, that is by using a form encoded with the special multipart/form-data to allow file attachments. The file's content is read and then parsed by Play using the Java based JSON library Jackson. The parser provides a powerful data binder between JSON and POJO with full data binding for any Java bean class, among other things.

Hashing for safe storing of passwords

All new user's passwords are hashed using jBCrypt before storing them in the database. Not storing passwords in clear text adds a layer of security as malicious users still cannot (at least easily) use the passwords even if they get hold of them. A downside to this on the other hand is that we cannot send the password back to the user if needed. This is because hashed passwords cannot be changed back to the original form.

Authenticated actions

The application is using action composition to protect actions with authentication. Action composition is the ability to chain actions together. Each action can manipulate the result, pass it on to another action or generate the result instead of passing it along. Play has a built in authenticator action which we extend with our own authentication logic (checking for attribute in session). In practice each action that needs a user to be logged in is annotated with @Security.Authenticated. This causes each request to these actions to pass through our authentication and redirects them to the login screen if needed.

Implemented use cases

User can add/create invoice

Prerequisites:

1. User is logged in.
2. One client with one bankaccount exists.

Success scenario:

1. User navigates to the invoices view.
2. User click on “Add Invoice”
3. User fill out the form correctly
4. User press “Add Invoice”
5. A new invoice is created and showed in the invoice view.

User can edit an existing invoice

Prerequisites:

1. User is logged in.
2. Atleast one invoice exists.

Success scenario:

1. User navigates to the invoices view.
2. User press an existing invoice.
3. User press “Edit”.
4. User changes something in the form.
5. User press “Update”
6. A green animation shows up and says “Your invoice was updated!”
7. User press “Back to invoice”
8. The invoice is updated with the new info.

User can view an Invoice

Prerequisites:

1. User is logged in.
2. Atleast one invoice exists.

Success scenario:

1. User navigates to the invoices view.
2. User press on an exsisting invoice.
3. The selected invoice and it’s info is showed in a new view.

User can delete invoice

Prerequisites:

1. User is logged in.
2. Atleast one invoice exists.

Success scenario:

1. User navigates to the invoices view.
2. User press the cross on any invoice.
3. User press “Ok” on the pop-up window
4. The invoice disappear.
5. A yellow animation that says that the invoice is removed appear.

User can add client

Prerequisites:

1. User is logged in.

Success scenario:

1. User navigates to the clients view.

2. User press “Add client”
3. User fill out the form with correct data and a unique organization number.
4. User press “Add client”
5. The client appear in the list below the form.

User can remove client

Prerequisites:

1. User is logged in.
2. Atleast one client exists

Success scenario:

1. User navigates to the clients view.
2. User hovers over one existing client with the mouse and press the cross that appears.
3. User press “Ok” on the pop-up window that appers.
4. The client is removed from the view.
5. A yellow animation that says that the client was removed appears.

User can view a client

Prerequisites:

1. User is logged in.
2. Atleast one client exists.

Success scenario:

1. User navigates to the clients view.
2. User press one of the existing clients name.
3. A new view with info about the selected client is showed.

User can send all clients invoices through e-mail to the client

Prerequisites:

1. User is logged in.
2. Atleast one client with a correct emailaddress exists.
3. Atleast one invoice is connected to the client.

Success scenario:

1. User navigates to the clients view.
2. User press one of the existing clients name.
3. User press “Send invoices”.
4. A green animation that says how many invoices that were sent appears.

User can send invoice to client through e-mail

Prerequisites:

1. User is logged in.
2. Atleast one client with a correct emailaddress exists.
3. Atleast one invoice is connected to the client.

Success scenario:

1. User navigates to the invoices view.
2. User press one of the existing invoices.
3. User press “Send invoice”.
4. A green animation that says that the invoice has been sent appears.

User can send invoice reminder to client through e-mail

Prerequisites:

1. User is logged in.
2. Atleast one client with a correct emailaddress exists.
3. Atleast one invoice that is overdue exists and is connected to the client.

Success scenario:

1. User navigates to the invoices view.
2. User press one of the existing invoices that is overdue.
3. User press “Send reminder”.
4. A green animation that says that a mail was sent to the client appears.

User can view own invoices

Prerequisites:

1. User is logged in
2. Atleast one invoice with user as owner exist

Success scenario:

1. User navigates to invoices view
2. Only user’s own invoices is shown

User can star an Invoice

Prerequisites:

1. User is logged in.
2. Atleast one unstarred invoice exists

Success scenario:

1. User navigates to the invoices view.
2. User hoovers over one of the invoices with the mouse and press the star in the lower right corner.

3. The star turn green and the invoice appears in the “Starred” view.

User can upload JSON invoice

Prerequisites:

1. User is logged in

Success scenario:

1. User navigates to new invoice view
2. User uploads an invoice as a correctly formatted JSON file
3. The invoice is shown in invoice view

User can upload JSON client

Prerequisites:

1. User is logged in

Success scenario:

1. User navigates to new client view
2. User uploads a client as a correctly formatted JSON file
3. The client is shown in client view

User can upload JSON bank account

Prerequisites:

1. User is logged in

Success scenario:

1. User navigates to new bank account view
2. User uploads a bank account as a correctly formatted JSON file
3. The bank account is shown in bank account view

User can login

Prerequisites:

1. User has an account with username and password

Success scenario:

1. User navigates to login screen
2. User enters username and password
3. User is logged in

User can logout

Prerequisites:

1. User is logged in

Success scenario:

1. User presses the logout link
2. User is logged out
3. Login screen is shown

User cannot view sensitive views without being logged in

Prerequisites:

1. User is logged in

Success scenario:

1. User tries to navigate to invoices view
2. User is redirected to login view

User can create user

Prerequisites:

None

Success scenario:

1. User navigates to create user view
2. User enters at least
    - username
    - password
    - repeated password
    - and accepts the terms and conditions
3. User is created with entered information
4. User is logged in

User can view user information

Prerequisites:

1. User is logged in

Success scenario:

1. User navigates to user view
2. User sees

User can edit user

Prerequisites:

1. User is logged in

Success scenario:

1. User navigates to edit user view

2. User makes changes to user and saves
3. User's changes is shown

User can add a bank account to a user

Prerequisites:

1. User is logged in.

Success scenario:

1. User navigates to the bank accounts view.
2. User press "Add account".
3. User fill out the form correctly with atleast:
    a. Account type
    b. Account number
4. User press "Add account".
5. The account appear below the form.

User can view a bank account

Prerequisites:

1. User is logged in.
2. Atleast one bank account exists.

Success scenario:

1. User navigates to the bank account view.
2. User press the account number on one of the accounts.
3. A new view with info about the selected account appears.

References

Anatomy of a Play application
http://www.playframework.com/documentation/2.2.x/Anatomy

Akka
http://akka.io/

Ebean ORM
http://www.avaje.org/

Google Guice
https://code.google.com/p/google-guice/

Jackson
http://jackson.codehaus.org/
https://github.com/FasterXML/jackson

jBCrypt
http://mvnrepository.com/artifact/org.mindrot/jbcrypt/0.3m
http://www.mindrot.org/projects/jBCrypt/

Play framework
http://www.playframework.com/

sbt (Scala Build Tool)
http://www.scala-sbt.org/