Kotlin Web Application

COMP40009 – Computing Practical 1

15th – 19th January 2024

Aims

- To provide experience writing a web application in Kotlin.
- To build a larger application making use of third party libraries.
- To use a database from Kotlin.

Introduction

Earlier in this course web built a minimal web framework, handling HTTP requests and returning HTTP responses. The code that we wrote in that lab was inspired by the `http4k` web framework written by Ivan Sanchez and David Denton (`https://www.http4k.org/`). This is a widely used framework for building web services and applications in Kotlin, which takes a functional approach to web development. In this lab we will try building something using the full `http4k` library, rather than our cut down version.

Compared to some of the other labs that you have done so far, this lab is more open-ended. We'll start off by taking a tour of some of the elements of the web application and see how they fit together, then you can explore further, try out different things, and build on the skeleton to extend the application and make something interesting.

Getting started

As per the previous exercises, get the skeleton files from GitLab. You can clone your repository with the following command (remember to replace the *`username`* with your own username).

`git clone https://gitlab.doc.ic.ac.uk/lab2324_spring/kotlinwebapp_`*`username`*`.git`

Behind the scenes, we have been using a tool called Gradle to configure and build our Kotlin projects. The `build.gradle` file in the root of the project lists (amongst other things) all the libraries ("dependencies") that we want to use in our project. We have added several `http4k` components to this list. Gradle is integrated with IntelliJ, so all the libraries listed will be automatically downloaded and made available to your code when you open the project.

If you want to add any further libraries to the project later on, you can do that by adding them into this file and re-triggering the Gradle import in IntelliJ.

Part 1: A Tour of the Web Application

The first place to look is in the `WebApp.kt` file. Here you will find the configuration for the routes (the mapping from URLs to HTTP handlers, think back to our first Kotlin lab) as well as the `main()` function which we can use to start the application.

Run the `main()` function. This should start a webserver, and you should be able to open a browser window, navigate to `http://localhost:9000/index`, and see a (very basic) webpage. You may see a few warnings when you run `main()`, for example about "SLF4J providers". That's nothing to worry about, the server should start ok anyway.

Although there is not a lot of code in the `WebApp.kt` file, it is worth taking a moment to understand some of the concepts that are being used here, and how they fit together to produce the page that you see in your browser.

The `routes()` function is where we define the mapping from URLs to handlers. One thing that we did not support in our first lab was different HTTP *methods*. HTTP methods are used to indicate the type of action that the client wants to perform on a particular URL. The most common method is `GET`, which is used to request a resource from the server. Other methods include `POST`, which is used to send data to the server, `PUT` and `DELETE`.

`GET` requests to the `"/index"` route are mapped to a simple handler that uses a library called *FreeMarker* to render a page of HTML based on a template. Which template file to use, and the data to populate the template, are defined in what we call a *view model*.

Here we are using a data class called `IndexPage` as the view model We define[1] a method `template()` which returns the name of the relevant FreeMarker template file. It also contains the data that we want to pass to the template - in this case just the property `name`.

Templates are loaded and processed by a *renderer*. We are using the FreeMarker renderer, which is configured to look for templates in the directory `src/main/resources`. If you find the `IndexPage.ftl` template file you will see that it contains some basic HTML. The `${name}` syntax is used to insert a value from the view model into the template.

Try making some changes to the template file, and see how they are reflected when you refresh the browser. You can add extra properties to the `IndexPage` class and use them in the template. Try adding a property that returns a list of Strings, and use it to render an HTML list in the template. See the FreeMarker documentation online[2] to find out how to use special tags to iterate over a list and control output in other ways.

Static Assets

One other thing you may have noticed is that the template file contains a reference to a "stylesheet". This is a *CSS* file that is used to style the page. You can find `style.css` if you look in the `src/main/resources/assets` directory.

We won't go into too much detail about CSS here, but you can try making some changes to the CSS file to style the page in different ways, changing the colours, fonts, layout, etc. There are a lot of detailed CSS tutorials online if you are interested to learn more.

One thing to note is that the CSS file is not processed by the renderer. Instead, it is served directly by the webserver. This is because it is a *static asset* - a file that is served directly to the client without any processing by the server. The browser loads the HTML and the CSS separately and then it's the browser that applies the styling from the CSS to the HTML.

The `"assets"` route in `WebApp.kt` is where we configure the webserver to serve static assets. The `static()` function is a pre-defined function from `http4k` which takes a path to a directory containing static assets, and returns a handler. This handler is then mapped to the `"assets"` route, so that any requests for files under that route will be handled by the static asset handler.

Edit the CSS file to change the styling and refresh the browser to see your changes.

[1] Eagle-eyed readers may notice that this is actually an *override* function - `IndexPage` *extends* a class called `ViewModel` and overrides some of its behaviour. More on this concept next term.

[2] `https://freemarker.apache.org/docs/ref_directive_list.html`

Part 2: Accepting User Input

Most web applications accept user input in some form, and process it dynamically to provide a customised response. In this section we'll see how to add a form to our web page for the user to enter some data, and then process that data in Kotlin on the server.

Let's start by adding some input fields to our user interface. We do this by editing the HTML in the template file.

In HTML we group related input fields together using a `<form>` element. A form contains a number of `<input>` elements. Here is an example:

```
<form action="/submit" method="post">
    <label for="namefield">Name:</label>
    <input type="text" id="namefield" name="name">
    <label for="agefield">Age:</label>
    <input type="number" id="agefield" name="age">
    <input type="submit" value="Submit">
</form>
```

Each input element has a `name` attribute, which is used to identify the input field when the form is submitted. A `<label>` element can be used to provide a text label for the input field, which is displayed next to the input box. There is a special `<input>` element for submitting the form, which has a `type` of `submit` – this will appear as a button on the rendered webpage.

Two other key parts of the form are the `action` and `method` attributes on the `<form>` element. The `action` attribute specifies the URL that the form data should be submitted to. Clicking the submit button will trigger a new HTTP request to this URL. The `method` attribute specifies the HTTP method that should be used to submit the form. We normally use `POST` for forms, as this signals that this HTTP request will update the state of the server, rather than just reading data.

In our server-side code we need to implement a new handler to process the form data. We might add something like the following to the list of routes:

```
"/submit" bind POST to { request ->

  val name = request.form("name")
  val age = request.form("age")

  // process the form data in some way...

  Response(OK)
},
```

Add a form to the webpage and then implement a handler to process the form data.

Update the web app so that:

- The user sees a form where they can enter up to five words in separate boxes
- When they submit the form, the app calculates the Scrabble score for each word
- The user is returned a page which shows the words they entered, listed together with their Scrabble[3] scores, with the highest scoring word first

[3] `https://en.wikipedia.org/wiki/Scrabble`

Part 3: Connecting a Database

It is very common for web applications to store data in a database. In this section we'll see how to connect our web application to a SQLite database, and use it to store and retrieve data. We will execute SQL queries from within our Kotlin code to retrieve data from the database.

The skeleton contains a file called `PostsDatabase.kt`, with example code showing how to connect to and interact with a SQLite database using some classes from the standard library.

SQLite is a very basic database that stores all the data in a file on your local disk. If you run the `main()` function in `PostsDatabase.kt`, it will create a new SQLite database file called `posts.db` inside your Kotlin project and insert a few records. You would be unlikely to use SQLite in a production web application, but it is useful for testing and experimentation.

If you run the `main()` function again, it will fail, as it will try to create a database table when it already exists. If you comment out the `createTable()` and `insertSomeData()` lines, it will run successfully, as it will just query and print the previously stored data.

The `PostsDatabase` class contains a number of functions that you can use to interact with the database. You can see how we include SQL inside our Kotlin code, and call an API to send it to the database. Here we do this by just adding in String literals containing the SQL, but there are better – more typesafe – ways to do this, which you could investigate later on.

One thing to note is that we have a Kotlin data class called `Post` which represents a simple social media post. Databases tend to work with primitive types like `Int` and `String`, rather than objects. We use the `Post` class to model a post in our code, and then convert it to and from a database row when we need to store or retrieve it. The function asListOfPosts() shows how we can convert a generic `ResultSet` from the database into a typed list of `Post` objects.

- Add a new page to your web application which displays all the posts in the database.
- Add a form to this page which allows the user to add a new post, adds it to the database, and then displays the updated list of posts.

Part 4: Preventing Duplicate Submissions

One problem commonly associated with web forms is what happens if the user refreshes the page after submitting the form. This may cause the form to be submitted again, which may well not be what the user intended – think about when you make a payment online, you don't want to pay twice if you refresh the page. To fix this, it is common to use the *redirect after post* pattern. After processing the form data, instead of rendering a template, we issue a *redirect*. This is a special type of HTTP response with HTTP status code 302, which is also known as *found*[4]. Receiving a redirect will cause the browser to make a new `GET` request to a different URL. The URL that the browser should redirect to is specified in the `Location` header of the response. This is a good example of how the HTTP protocol can be used to implement application logic.

In `http4k` we can create a redirect response using code similar to the following:

```
"/submit" bind POST to { request ->
    // ... process the form data in some way ...

    Response(FOUND).header("Location", "/index")
},
```

- Update your Posts page so that it uses the above pattern, and doesn't add the same post again if you refresh the page.

[4] `https://httpstatus.in/302/`

Extensions

Once you've done the basics, you could try some of the following extensions:

- Add a new option which allows the user to click to delete a post from the database.
- Download a sample SQLite database from the internet, and update the database code, domain types, and view template code in your web app to your webapp to work with it and do something interesting with the data.
- Embedding SQL inside Kotlin as raw Strings is a bit inelegant (especially when it comes to quoting strings), and does not take advantage of the Kotlin type system to prevent errors. Use some of Kotlin's language features to build a *domain specific language*, allowing you to write your SQL queries in a more typesafe way, and then programmatically generate the SQL string to send to the database.

Note: your repository containing the skeleton for this lab can be found at *https://gitlab.doc.ic.ac.uk/lab2324_spring/kotlinwebserver_username.* ***As always, you should use LabTS to test and submit your code.***

Assessment

In general, the assessment for laboratory exercises uses the following scheme:

```
F - E: Very little to no attempt made.
       Submissions that fail to compile cannot score above an E.

D - C: Implementations of most functions attempted;
       solutions may not be correct, or may not have a good style.

B:     Implementations of all functions attempted, and solutions
       are mostly correct. Code style is generally good.

A:     There are no obvious deficiencies in the solution or
       the student's coding style. In addition, there is
       evidence of productive testing.

A*:    As for an A -- plus the student has done additional work
       beyond the basic spec, e.g. by considering (and clearly
       commenting) interesting variations or extensions to the
       given functions; e.g. based on their own research.
```