Kotlin Web Server

COMP40009 – Computing Practical 1

13th – 17th November 2023

Aims

- To provide experience writing programs in Kotlin.
- To define and make use of functions and simple data types in Kotlin, adapting knowledge already gained from functional programming in Haskell.
- To explore the basic mechanics of how a web framework might be implemented.

Introduction

Whenever we use a web application and browse to a new page, our browser sends an *HTTP request* to a server. The server processes the request and returns an *HTTP response* back to the browser containing the data it needs to render the page. We can consider that at a basic level, the server is a *function* from HTTP request to HTTP response. We say that the server *handles* the request, and so such a function is an *HTTP handler*.

A request can contain many different pieces of information, but one of the most important is the "Uniform Resource Locator", or URL. The URL specifies which page we are requesting. One server may serve many different pages, with different code for each one, and so the server uses the URL to identify which HTTP handler to invoke, passes the request to it for processing, and then returns the response back to the client. The response has a body containing the main content of the page, and also a *status code*, for example 200 (OK) or 404 (not found)[1].

The request may also include various parameters. For example, if you are performing a Google search then the request will include the text that you typed in the search box as a parameter, *q*. These parameters are also passed to the handler function so that it can use them in processing the request, e.g. to find the relevant search results for your query.

In this lab exercise you will build a Kotlin implementation of a simple web framework to process requests following this pattern.

Although what we will do here is a lot simpler than a real web application, the ideas are inspired by the `http4k` web framework[2], and the paper "Your Server as a Function"[3] by Marius Eriksen from Twitter. Have a look at those if you are interested.

[1] `https://developer.mozilla.org/en-US/docs/Web/HTTP/Status`
[2] `https://www.http4k.org/`
[3] `https://monkey.org/~marius/funsrv.pdf`

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_autumn/kotlinwebserver_username.git`

Work on the code using the IntelliJ IDEA integrated development environment (IDE) which you can download from the JetBrains website. As Kotlin is a JVM language, you will also need a Java Development Kit (JDK). See the links in the course material for how to set up IntelliJ with a JDK, how to open the provided files, and how to work with the code and run tests.

The repository also includes a script `build.sh` which you can run on your machine to compile your code, run your tests, and check your code style, before you submit it to LabTS.

What to do: Part 1

Dealing with URLs

To start off, we'll look in more detail at URLs, and write some functions to extract different parts of a URL. A URL (or more generally a *URI*[4]) has a number of distinct component parts. For example, consider the URL `http://www.google.com/search?q=kotlin&safe=active`.

In this case:

- The *scheme* is given by the part before the `://` - in this case that is `http`.
- The *host* identifies the network address of the webserver - in this case `www.google.com`.
- The *path* identifies a particular resource or web page on that web server. In the example above the path is `/search`.
- The *query string* allows parameters to be passed as name/value pairs. The query string is separated from the path by a question mark (`?`). Multiple parameters are separated by ampersands. In our example, the query string is `q=kotlin&safe=active`.

For more detail on how a URL is made up, see `https://en.wikipedia.org/wiki/URL`. For this exercise, assume that you can represent a URL as a Kotlin `String`.

In the file `WebServer.kt` define the following functions to extract different parts of a given URL. You will find some `TODO()`s for you to replace to complete the functions. You may find it useful to investigate the various methods that you can call on a String in order to break it down into parts, e.g. `substring()`, `substringBefore()`, `substringAfter()` and `split()`.

- Define a function `scheme()` that takes a URL as a `String`, and returns a `String` giving just the scheme (e.g. "http" or "https", not including the colon or the slashes).
- Define a function `host()` that takes a URL and returns just the host (e.g. "www.google.com").
- Define a function `path()` that extracts the path component.
- Define a function `queryParams()` that extracts the query string, and returns it as a list of key-value pairs, so if the input URL is the string `q=kotlin&safe=active` then the output would be a `List` containing two `Pair`s, ("q", "kotlin") and ("safe", "active").
  If the URL contains no query parameters then the function should return an empty list.

In the file `WebServerTest.kt` you will find some tests for these functions. Make sure these tests pass and add in any more tests you feel are needed by adding more functions in the same style.

[4] `https://en.wikipedia.org/wiki/Uniform_Resource_Identifier`

Requests and Responses

In the file `Http.kt` in the provided code you will find types `Request`, `Response` and `Status`. You can create a simple request by calling the constructor passing a URL as a parameter, e.g.

```
val request = Request("http://www.imperial.ac.uk/computing")
```

Similarly you can create a `Response` by specifying a status code and a body.

```
val response = Response(Status.OK, "<html>...</html>")
```

You can access the properties of these objects using dot notation, e.g. `request.url`, `response.status` or `response.body`.

Write a function `helloHandler()` that takes a `Request` as an argument and always returns a `Response` with status OK and the body "Hello, World!". This is your first HTTP handler. Extend your handler function to extract a parameter from the request and include it in the response, e.g. if you handle a request with the URL `http://www.helloworld.com/say-hello?name=fred` then a response should be generated with the body "Hello, Fred!".

Again, you'll find some tests for these functions in `WebServerTest.kt`. Initially most of the tests will be commented out. One by one, uncomment the code for each test in the test suite and implement the functions you need until you have everything passing.

If we were building a real web application, or a fully-featured web framework, we would probably use a templating engine like *Thymeleaf*, *Freemarker* or *Mustache* (or many others) to generate the body content, to help to separate the layout concern from the logic of the application, but we'll just keep things very simple in this exercise.

What to do: Part 2

Routing

A modern web application will typically be able to handle requests for many different pages (or API endpoints[5]) by specifying a separate handler function for each one, and using the *path* specified in the URL to select the correct handler to handle the request. The process of mapping a URL to a handler is called *routing*.

For example we could have the following mapping:

```
/ => homepage handler
/search => search page handler
/browse/books/programming => programming books page handler
```

Setting up routing in this way allows us to write each of our handlers as self-contained pieces of code, rather than having one huge function that serves all of our requests. This makes our application code much more modular, which means it should be much easier to understand, to test, and to maintain over time.

Write a function `route()` that takes a `Request`, uses the path component of the URL to look up an appropriate handler, and returns the `Response` yielded by passing the request to that handler. The code might well resemble the mapping above. If no handler can be found that matches the given path, your router should return an empty response with a Not Found (404) status code.

Guided by the tests provided, build up a set of mappings and handlers to fill out a simple web application. Add any further tests that you think would be useful.

[5] `https://smartbear.com/learn/performance-monitoring/api-endpoints/`

Extensions

These parts are optional. If you have got the first two parts working and want to explore more, try some of these extensions.

More Flexible Routing

In our implementation so far, the routing logic and the configuration of which path maps to which handler was all done together in one piece of code. It would be cleaner if we could separate these. We can try to specify a set of mappings from path to handler, and create a function that uses this to build us a router.

First of all, let's make a specific type to represent HTTP handlers. We have said that an HTTP handler is just a function from `Request` to `Response`, so let's codify that by using a Kotlin *type alias* and also a *function type*.

```
typealias HttpHandler = (Request) -> Response
```

Each of our individual path handlers should match this type signature already. What we want now (thinking back to Haskell...) is a function that takes a set of mappings, and *returns us a function* that will perform the right routing for us.

We can define our routing configuration as a list of pairs where the first element of each pair is the path, and the second is the handler to route to. The type of this list would be:

```
List<Pair<String, HttpHandler>>
```

Create a list of this type containing your current routing configuration. To refer to a function, you can use the double-colon notation like this:

```
fun a(x: Int) = 2 * x
fun b(x: Int) = 5 * x

val funcs = listOf(::a, ::b)
```

Now create a new function `configureRoutes()` that instead of being an `HttpHandler` itself (taking a `Request` and returning a `Response`) instead takes your list of route mappings and returns a *function* that is an `HttpHandler`. The returned function should be able to process a request, route it to the relevant handler according to the mappings you specified, and return the response. So for example you might write:

```
val app: HttpHandler = configureRoutes(...)
val request = Request(...)
val response = app(request)
```

If you aren't sure how to do this, have a look at the language reference for defining lambdas and anonymous functions[6].

Once you have this working, a nice piece of syntactic sugar is to use the infix function `to` to create a `Pair` rather than the `Pair()` constructor. So instead of writing `Pair('a', 1)` to can just write `'a' to 1`. Try converting your route mappings list to use this format and see how it looks.

Uncomment the relevant test and complete the code inside the test method to configure a route mapping and make it pass.

Filters

Sometimes we want to apply a cross-cutting concern, like logging, or security, to several parts of our application, without coding it explictly in every one of those parts. The concept of a *filter*

[6] `https://kotlinlang.org/docs/reference/lambdas.html#lambda-expressions-and-anonymous-functions`

allows us to specify this sort of processing separately. We can then put the filter into the request pipeline, before it gets to the relevant handler, to add some additional processing. So if we want a filter to check for authorization, we can put that filter in front of all of our top-secret pages, and reject an unauthorized request without even invoking the relevant handler. Let's see how we can implement this.

We have already implemented `HttpHandler`s to process particular requests, and it is interesting to note now that a filter can also be implemented with the same type. If we do this then it enables a very flexible composition of filters in different combinations with handlers. From the point of view of the client, whether a request is handled directly by a handler, or by a filter and then a handler, the type looks exactly the same. There's no need to make a special case in the code.

Try creating a function `requireToken()` which controls access to certain pages in our webapp. This function should take another `HttpHandler` as a parameter (which is the handler that we want to protect) as well as the required authorization token (this isn't a particularly secure authentication scheme, but we won't worry about that here...). Your function signature might be:

```
requireToken(token: String, wrapped: HttpHandler): HttpHandler
```

`requireToken()` should return a function (`HttpHandler`) that when passed a request, will check whether the request contains the correct authorization token, and if it does, passes the request through to the wrapped handler to be processed. If the correct token is not present, then we just return a Forbidden (403) status code and do not invoke the wrapped handler. Our `Request` class allows you to specify an authorization token as an optional second parameter to the constructor, after the URL. If it isn't specified then it defaults to an empty string.

Once you have your filter, you can use it together with your route mapping, for example:

```
"/homepage" to ::homepageHandler,
"/secretpage" to requireToken("password1", ::restrictedPageHandler)
```

You could try building some other types of filters (for example logging when page accesses happen) and combining them into filter chains.

Note: your repository containing the skeleton for this lab can be found at *https://gitlab.doc.ic.ac.uk/lab2324_autumn/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.
```