NipponTech: a dynamic Web shopping cart written in Kotlin

SEPTEMBER 17, 2019

令和１年９月１７日

Created by Francisco Cáneva for Diverta Inc.

Diverta

Table of contents

Project description

Task definition

Create a shopping cart page using the programming language which you have more experience and feel more comfortable, so you can really show off your skills.

Estimated work time: ~5hr

Deadline: September 18th, 2019

Requirements:

- Server side:
  - API for cart (list, add, remove)
- Client side:
  - Product list page (a simple static page)
    - Ability to add items to the cart
  - Cart view page
    - Ability to remove items or proceed to payment

Bonus points:

- Good error handling
- Good user experience
- Well-organized, well-formatted code
- Anything creative

"NipponTech", a project background

The fictional company "NipponTech" was created in this project to allow me to give the project a better background, which itself dedicates to sell computer appliances. For a better introduction, here is a short introductory text in its story:

> *We are NipponTech, your partner in computer technologies for your personal and professional life. From a simple flash thumb drive to a full-fledged server and all its workstations, we're the chosen by Japanese companies and enterprises, as well as for public in general. From our offices located at Akihabara, in Tōkyō, we distribute 20,000+ products every month to all the country through our distribution centers located at Sapporo (Hokkaidō), Ōsaka and Fukuoka; so, wherever you are located, we can reach you within a week. When searching for the best prices, the best product quality and the fastest shipping in the country, regardless of how many products you need, the answer is very clear: NipponTech.*

Project characteristics

Features

- Highly portable, requires only a Java Virtual Machine and a MySQL/MariaDB server for operating.
- Works both on Windows and Linux (using Oracle JVM).
- Simple and concise configuration file, lets you configure the database to use and the port to serve the dynamically generated pages.
- Fast serving time (approx. 5 to 50ms) thanks to the ahead-of-time compilation of the HTML pages.

Limitations

- As the application is highly oriented to Japan, it may be not simple to convert for other countries. This happens, mainly, because of the data types used for the currency handling (`BIGINT` instead of `DECIMAL` in the SQL server and its corresponding `Long` type in Kotlin). Other tables are highly influenced by Japan, such as the `Prefecture` table.
- No account is required to buy: although some web cart systems allow to do so, in theory nothing can prevent you from buying the same order twice if for some reason the connection fails. This can be prevented by handling the orders within a database instead of using the server storage location, but will require some ID related to an account to make sure the carts' states stay valid.
- Although the sessions are stored in disk, they're not resistant to server restarts. This behavior is similar to PHP but, as well as with the previous point, using a database to make it persistent will solve the problem. The con of it is that whenever the compatibility of previous versions is broken, this information must be deleted.

How is it made?

The project is divided in two parts, both of them bundled in the same application. The first part, the server side, is fully programmed in Kotlin for Java Virtual Machine; and the second part, the client side, comprises the HTML outputted from Kotlin plus CSS and JS libraries for making the page dynamic and responsive. As Kotlin is transpiled into Java for operating inside the JVM, it'll transparently access all Java's classes as well as other libraries without any additional glue code.

Server-side software libraries

Following up is a list of the libraries used to build the server side of the application:

- Jetbrains Kotlin: The core and main part of the application, this package ensures all Kotlin generated code run properly inside the Java Virtual Machine. It is composed of two elements: the Kotlin transpiler (only used during development) and the Kotlin runtime itself. Can be found at https://kotlinlang.org.
- Javalin: Built on top of Eclipse's Jetty, Javalin provides an easier to configure HTTP and WebSockets servlet for use in your applications. Extremely flexible, lets you configure everything using Java objects and lambda functions. Its web page is at https://javalin.io.
- Kotlin Exposed: An Object-Relational Mapper (ORM) designed for Kotlin which allows you to work with SQL databases by using Kotlin objects instead of the standard SQL sentences. While removes a bit of flexibility like any other ORM, it adds more correctness control because, since the sentences are generated through the use of Kotlin objects, they're checked when the application is built. Its Github repository is located at https://github.com/JetBrains/Exposed.
- KotlinX HTML: This library is in charge of creating HTML pages. By using the lambda function system in Kotlin and using a tag/object consumer system, it lets you create your pages by using Kotlin code directly, allowing to use it as a template system without the need to learn a new template engine syntax. As well as the ORM, it benefits from the correctness checking at compile time, ensuring your DOM tree is well-formed by checking if the tags are hierarchically correctly placed (eg: `body` inside `html`, and not `html` inside `body`) as well as

the tags' attributes names and values. Also distributed through Github; its repo is at https://github.com/Kotlin/kotlinx.html.

- MySQL Connector Java: This library allows you to connect to a MySQL compatible database and send and receive data from it. It is indirectly used by Exposed for connecting to the database the application requires. Can be obtained from the MySQL official page at https://dev.mysql.com/downloads/connector/j/, or through Maven repositories.
- org.json: Easily lets you create JSON objects for exporting and importing data to the web frontend. Can be downloaded through Maven.
- SLF4J: Simple Logging Façade for Java is a highly configurable logging system for tracing the events that occur in the server. It is used main and internally by Javalin, but we also take the advantage of it by logging the start and shutdown of our server and the exceptions fired. Can be obtained from Maven as well.

Client-side libraries

Now, it is time to reveal which libraries run the web pages we see in the browser:

- Accounting.js: A small library which provides formatting for everything related to numbers and/or currency. Can be obtained at http://openexchangerates.github.io/accounting.js/.
- Bootstrap + Popper + Tooltip: Provides the look and feel, as well as the organization for the elements in the page. To install it, grab it at https://getbootstrap.com and follow the instructions to install the auxiliary libraries.
- Bootstrap-Select: Allows `selects` that contain multiple sections to be properly displayed under Bootstrap. You can fetch it from https://developer.snapappointments.com/bootstrap-select/.
- Bootswatch: Provides several theming variants for sites based on Bootstrap; for this project, the Darkly theme was chosen. You can see all themes at https://bootswatch.com/.
- JQuery: A all-in-one library for interacting with your website, simplifying tasks and operations. Grab it from https://jquery.com/.
- Toastr: Shows toast notifications in the upper-right corner of the screen. The Github repository is located at https://github.com/CodeSeven/toastr.
- WaitMe: A library for showing up wait boxes in objects and the whole page. You can see it in action in its Github page, at http://vadimsva.github.io/waitMe/.

Helper programs

The following programs were used in the course of the develop of the application:

- Jetbrains IntelliJ IDEA: The IDE for excellence in everything related to Kotlin. This program was built in the Community edition. Its main page is located at https://www.jetbrains.com/idea/.
- Git: A tool for tracing the development of the application through its source code. You can install it from https://git-scm.com/.
- Mozilla Firefox & Google Chrome: Web browsers, for testing the generated pages.
- Postman: Used in first term for testing the AJAX/REST API. You can get it from https://www.getpostman.com/.
- Gradle: The build system used for building the application into its distribution form. It is bundled alongside JetBrains IntelliJ IDEA IDE.
- Prettier: A nodeJS-based application to pretty print Javascript files. You must have NodeJS installed to run this application, which can be obtained using NPM.

Source files in detail

Inside this Github repository, you will have all the files that builds the project, as well as the configuration to build it through the Gradle build system. The following is the list of the files that can be found with its corresponding description:

/	Root of the file system hierarchy
/.git/**	GIT control folder
/.idea/**	IntelliJ IDEA control folder
/gradle/**	Gradle's control folder
/src	Root of the sources
/src/main	Main project package
/src/main/java/**	Root of Java sources (not used in this project)
/src/main/kotlin/**	Root of Kotlin sources (see below for more details)
/src/main/resources	Root of resources folder
/src/main/resources/static_www	Root of the static files for the web server (static CSS+JS libraries)
/src/main/resources/log4j.properties	Configuration file for SLF4J.
/src/test/**	Test project package (not used in this project)
/var_www/**	Root of the static files for the web server (products' images)
/build.gradle + /settings.gradle	Gradle's build configuration
/config.json	Application's configuration file (will be copied on built)
/diverta-cart.sh	Linux bootstrap script
/gradle + /gradlew + /gradlew.bat	Gradle's bootstrap scripts
/products.sql	Sample products to register in the database

The Kotlin files stored inside `/src/main/kotlin/` folder are all inside the `io.github.sonicarg.diverta_cart` package. Their contents will be described in the following table:

/handlers	Root folder for web handlers
/handlers/rest	AJAX/REST handlers
…/CartHandler.kt	Handler which manages the cart contents
…/CheckoutHandler.kt	Manages the credit card payment
…/MethodNotAllowedErrorHandler.kt	Simple handler when requesting data using invalid methods
…/ShippingHandler.kt	List the prefectures and shipping fares
/handlers/www	Web page generator handlers
…/CartPageHandler.kt	Renders the cart contents and shows the prices including shipping fares
…/CheckoutPageHandler.kt	Shows the forms to send the purchase and the card to charge from
…/ErrorPageHandler.kt	Template for standard pages
…/HTMLFragments.kt	Basic HTML fragments common to all pages
…/MainPageHandler.kt	Shows the products' page (also known as the main page)
/CartServer.kt	Class which models the server for the application
/Database.kt	Contains definitions for the ORM system, both the tables and the objects instances
/Main.kt	Main entry point of the application
/ResourceLoader.kt	Tools for loading files inside JARs

Installation

Prerequisites

As previously mentioned in the features' section, the application requires a Java Virtual Machine installation and a working MySQL server. MariaDB can be used as it is fully compatible with MySQL. In the following section, we'll dig into the details of installation of the software.

Installation procedure

1. As previously mentioned in the prerequisites, install Java in your system. Make sure you use version 8 or greater, since Kotlin runtime depends extensively on the use of lambda functions. On Linux-based systems, I recommend using the Oracle's runtime since it runs much better than the OpenJVM alternative.
2. As previously mentioned in the prerequisites, install either MySQL or MariaDB in your system.
3. Refer to the file `config.json` located at the root folder of the application. In there, you'll find the required configuration to start the application, such as the database user, password and name of the database to connect.
    a. You'll need to create an empty database and a user with a password (you can use these default values provided in the file or you can set to the ones you want) and modify the file accordingly to the values you chose (in the `database` section, modify the values `database`, `user` and `password`).
    b. Set up the port used by the server in the `server` section, in the `port` parameter. If you use the standard web port (80), remember to run the server with elevated privileges (`Run as administrator` in Windows and `sudo` in Linux).
    c. *Optional*: You can choose whether to let the application open the default web client on the first run by setting to `true` the options `autoBrowserLaunch` in the `ui` section. Also, you can set the VAT value for use in the application by changing the `vat` value (`0.1` = 10% of VAT).
4. Launch the application.
    a. In Windows: double-click the application `diverta-cart.exe` and a console window will open.
    b. In Linux: open a terminal, navigate through the file system until reaching the folder which contains `diverta-cart.sh`. Launch it by writing `./diverta-cart.sh`. The application will start its logging.
    c. In both cases, do not close the console window until you decide you'll no longer work with the application. When you're done, push the `CTRL` + `C` key combination to trigger the application shutdown.
5. When the application starts for the first time, it creates several system and user tables inside the database the application will use. These are differentiable because all system tables are filled with default data, but the user ones remain empty. In this case, the table `Products` remains empty as there can be no products to sell from the very first moment. I provided some example products with their images: the images are already placed in the folder `var_www`, while the definitions are inside the file `products.sql`. Load this SQL file to the database to register them (you not need to close the web server during the installation).

How it works

Since Javalin is a programmable server, we can easily tie the controllers to the addresses and methods we want. In this steps, we show what happens inside Javalin since we receive a request until it is sent back to the client. The numbers in `[square brackets]` refer to the lines in the source file `CartServer.kt`:

1. Everything begins when the listening port receives a HTTP request from any capable client. All the request is received at once and caught by the Javalin library.
2. After request is fully received, Javalin then reads the whole of it and creates a `Context` object to be further processed by our implementation. Mainly, it processes the headers, the form data and the query string and places everything into variables contained in the `Context`.
3. The next step is start the process pipeline: these are a series of lambda functions that take our `Context` and take the decisions they need. Whenever a pipeline step sets either the status code (which, by default is `200 OK`) or sets any content to return, the `Context` object abandons the pipeline.
4. First step in the pipeline is the pre-router (namely `before [69-83]`): a piece of software that runs before the routing of the context to the actual handler. This can be used to check or set some common properties all handlers should check or write; in our case, we check the cart and the VAT session variables are correctly set for the handlers to work with them. At the end of this step, the `Context` should not be returned to the client.
5. Next step is the context routing and authorization: the server checks its routing configuration to determine which handler should take care of the `Context`. After the method and the route matches the programmed configuration, an authorization phase takes place before the chosen handler runs. This authorization phase checks whether the client is allowed or not to visit the address with the method in question (useful to check cookies or any other data in the `Context` to prove the identity of the client). It is expected that the authorization phase either runs the handler or returns either `401 Unauthorized` or

`403 Forbidden`. By default, as is this case, we allow access to all routes; so the effective routing takes place immediately.

6. After authorization was granted, the actual handler is run `[85-131]`. Handlers are composed of: a method (`GET`, `POST`, `PUT`, etc.), a path (e.g. `/checkout`) and a handler which can be either an anonymous lambda function or a function handler. Typically, this functions should work with the context and return nothing (e.g.: `void` functions) because all the feedback to the client is managed inside the context. At the end of the handler, the HTTP response should had been fully managed and ready to return to the client.

7. After the routing, the post-routing takes place (namely `after`). Like its before counterpart, this is a common step run after the routing successfully took place and the handler has just run. Is this after this moment, that the HTTP response is sent back to the client.

8. If the routing fails for our custom handlers given in the steps 5 and 6, then Javalin tries to load a static file from its configured folders `[51-52]`. A static folder can be either loaded from either the Classpath or from an external folder. In our case, all the files corresponding to CSS and JS libraries are stored in `static_www` inside the Classpath, which is a read only zone. The images of the products are stored in `var_www` in an external folder, which allows to add or remove images easily. If the static search also fails, a `404 Not found` error is returned.

9. If, by any reason, an exception is fired inside the steps 4 to 7, an exception handling mechanism is activated. Exception handlers work in a similar fashion as the `try-catch` mechanism Java has: it tries to match the corresponding exception type to the one the handler expects. If it does, the handler is run. If it doesn't, it continues searching until it matches one or runs a default exception handler which returns `500 Internal server error`.

10. As well as exceptions, HTTP errors can be caught to show custom error pages; the handlers run after exceptions have been processed `[137-147]`. The handlers run in the same fashion as the page handlers or the exception handlers: a function receives a `Context` and returns `void`, rendering the page into the `Context` object.