Synoptic Project:

Apprentice: Nimalan Kirubakaran

Overview:

This piece of work provides documentation for my apprenticeship synoptic project. I was asked to implement a REST API which would satisfy the business requirements. The user guide for this is part of the README.MD on the GitHub repository.

I did this project in Java using Spring Boot.

Design:

The requirements for the REST API are as follows:

1. If card isn’t registered then the card owner will be required to provide employee information
2. 4 digit pin chosen by the employee should be used for further security
3. Data should be stored in the database
4. If the user’s card isn’t registered then the system will respond by requesting that it needs to be registered
5. Tapping again 2x informs the user “Goodbye”
6. System should timeout after a number of minutes of inactivity

I managed to implement all of the requirements fully but with a partial implementation of (2). I did not manage to implement ensuring that the card\_id consists of a unique sequence of 16 alphanumeric characters.

Assumptions:

I made the following assumptions when interpreting these requirements:

1. Front-end app responsible for making API calls
   * I assumed some validation would be of the data before being sent to the API
2. Card number stored in the physical card itself
   * I assumed that the touch screen would have some way of communicating this to the front-end app
3. Physical cards contain 16 random alphanumeric characters
   * I assumed that the cards would have these imprinted physically on them (much like MAC addresses for network devices) and so there is no code to create these cards
4. Amount in pence
   * I assumed that the amount would be in pence to ensure easy processing through the code and storage in the database
5. No need to enter PIN once card has been tapped
   * I assumed that the PIN number was for “further security” – and that there would be no need to enter once the card had been tapped in

Limitations:

Due to the time constraints there are several shortcomings of my API.

1. PIN numbers stored in plain-text
   * This is rather glaring problem for security reasons and given more time I would remedy this first
2. Unsatisfactory use of PIN number
   * At the moment the PIN number isn’t used for anything as the requirements weren’t clear as to what it should do
   * Given more time I would like to have the user enter their PIN once they tap their card
3. No validation
   * There is no validation of the data that is being sent
   * Similarly there is nothing to prevent a user entering in an already existing user’s details (e.g. no duplicate first and last name). This is the edge case that I would handle first
4. Less than optimal use of cookies
   * The timeout feature is universal. That is to say a user could repeatedly top-up and then be timed out. A better implementation would be to have the time-out feature reset after each API call
5. Administrative functions
   * There is no way to issue new cards apart from manually going into the database and entering the details.
   * The API could benefit from additional administrative endpoints such as displaying list of cards, tracking the amounts, top-ups etc…

Data model:

The crucial ‘entities’ here are the Card object representing the physical card as well as the Employee object representing the Employee. Below is an entity-relationship diagram of these two:

A screenshot of a cell phone

Description automatically generated

Since it is impossible to have user details without having a card the CardId attribute is a foreign key in the Employee table. For the purposes of the MVP I decided against creating a database table called PaymentEvents which would track each top up and payment. Below is the class model I used for these entities:

A screenshot of a cell phone

Description automatically generated

A sample flow for getting the card details is displayed below in the following sequence diagram:

A screenshot of a cell phone

Description automatically generated

Tests:

Due to time constraints I could only write integration tests for the implemented features. My test setup used JUnit 4 and a PostgreSQL docker container which would be used to house the test database. I configured the container so the database schema would be setup by Liquibase which would automatically run the database migrations (see the following screen shot).

A screenshot of a cell phone

Description automatically generated

A JUnit **@Before** method was used to set up some initial test data in the database. Finally I used Spring’s **mockMvc** to make REST calls and verify whether the JSON response matched as expected and the correct cookies were present (see the following screen shot).

A screenshot of a cell phone

Description automatically generated

The following tests are contained within **CardResourceTest**:

* **tapInSuccessfully**
  + This test checks whether a user could tap in successfully, if a cookie containing the card\_id has been set and whether the resulting JSON response returns the card details and a welcome message
  + This test was successful
* **doesNotRecognizeCard**
  + This test checks whether the correct response is returned when a card is not recognized on the database and whether the resulting JSON response has a message indicating this
  + This test was successful
* **cardRequiresRegistration**
  + This test checks whether the correct response is returned and appropriate cookies are set when a card without a user is present
  + The JSON response should return a message indicating this and return the card details as well as set a cookie with the card\_id
  + This test was successful
* **topUpCard**
  + This test checks whether the correct response is returned when a user sends an amount to top up by
  + The JSON response should return a message indicating this as well as return the card details
  + This test was successful
* **paySuccessfully**
  + This test checks whether the correct response is returned when a user attempts to pay
  + This test was successful
* **payUnsuccessfully**
  + This test checks whether the correct response is returned by the exception handler when a user attempts to pay but has insufficient funds on their card
  + This test was successful
* **tapOutSuccessfully**
  + This test checks when a user taps out a message is displayed in the JSON response indicating they are logged out as well as removing the cookies associated with the card
  + This test was successful

The following tests are contained within **EmployeeResourceTest**:

* **timedOutWhileRegistering**
  + This test checks whether the correct response is returned by the exception handler when a user attempts to register but their session has been timed out
  + Since some of the endpoints in the **Employee** class would also timeout a user if their session had expired there was no need to test this functionality again
  + This test was successful
* **registerCard**
  + This test checks whether the correct response is returned to the user when they submit their details to register their card
  + The response returns a message indicating success as well as returning their details back
  + This test was successful
* **updateEmployee**
  + This test checks whether a user’s details are updated when their details are submitted
  + The response returns a message indicating successful as well as returning their details
  + This test was successful

For the integration tests in **EmployeeResourceTest** the feature under test required that a user’s card details be sent along so to replicate this I manually set a cookie as part of the REST call.

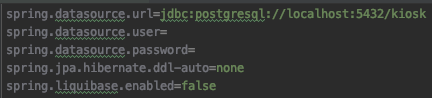
Construction:

I began constructing the project by first setting up the database. I used the open-source database platform Postgres as well as the free GUI tool Tableplus to see the contents of the tables. I used Liquibase to perform the database migrations rather than having to manually update the database using SQL (see screenshot below):

A screenshot of a computer

Description automatically generated

The database and application properties were stored in the application.properties file used by Spring Boot for configuration (see screenshot below):



I set spring.liquibase.enabled to false so that I could run the migrations manually using the Maven plugin. The advantage of using Liquibase was that when I set up my test database using the TestContainers package I could simply have Spring Boot automatically run my migrations.

For the database mapping I used the Java Persistence API (JPA) annotations as follows (see screenshot below):

A screenshot of a cell phone

Description automatically generated

Because the **Employee** class contained the PIN number and I didn’t want this being displayed to the user in the front-end app I added the **@JsonIgnore** and **@JsonProperty** annotations so that the PIN would be deserialized but not serialized on return (see screenshot below):

A screenshot of a cell phone

Description automatically generated

Because I added Spring Data JPA it provides the Spring application with CRUD operations out of the box (see screenshot below):

A screenshot of a cell phone

Description automatically generated

Once again I used the JavaEE architecture in designing the API. Below is the **pay** method in the CardService responsible for deducting the balance from the card (see screenshot below):

A screenshot of a cell phone

Description automatically generated

Finally the controller for the **Card** object is as follows:

A screenshot of a cell phone

Description automatically generated

The biggest challenge involved finding the card details upon receiving the card\_id. My original plan was to use Spring Security to provide HTTP Basic Authentication and send the card\_id as the username and the PIN as the password. However because my database design had the card\_id and PIN stored in separate entities I had to implement a custom authentication provider. I succeeded in doing so however I decided against this approach as to meet the other requirements it required directly modifying the servlet configuration which I neither had the time, patience or knowledge for. Instead I decided to do the authentication manually using just the card\_id by storing it in a cookie and upon receiving this cookie would find the associated user.

Rather than complicate the service layer and to promote reusuability of code I implemented exception handlers so that for certain events (e.g. insufficient funds or time out) a RuntimeException would be thrown allowing me to customize the JSON response (see screenshots below):

A close up of a sign

Description automatically generated

The Exception handler is below:

A screenshot of a cell phone

Description automatically generated

Brief conclusion:

The synoptic project has given me a better understanding of the Spring framework and how to implement custom authentication. Upon reflection if I had the chance to redo it I would modify my database design so that I would have the PIN as part of the card entity and allow nullable values to account for the case of a card not being associated with the user. This would allow me to cleanly implement HTTP basic authentication using Spring Security.