JEDD HOPKINS  
CAPGEMINI  
  
  
2. Tracking a Request from UI to Database

**\*\*Please note, many method, class and variable names have been changed to generic terms such as ‘thing’ to maintain anonymity\*\***

**Contents**

Page 2 – Acceptance Criteria/User Story

Page 4 – Creating the Project

Page 5 – Creating the RESTful API

Page 10 – Building the User Interface (UI)

Page 13 – Conclusion

**Please refer to submission:**

1. **Introduction & Selenium Testing**

**For an introduction to the company, how we operate and how I pick up tickets**

**Acceptance Criteria/User Story**

The client needed an application to maintain a number of items and associated business processes. This would include a database, a RESTful API layer, and a user interface (UI).

In this evidence submission I will be going over part of the production of the application landing page and associated RESTful API.

As part of the full stack nature of our work, it is required to build a UI for testing purposes, simple enough so that it would take up too much of our time but extensive enough to perform all the operations that the final product will. This UI would test the connection and requests between the front-end and back-end. Any errors that appear here when testing can be solved very quickly giving the client confidence that what we build will be fully functional when developed.

For this ticket I contributed to the writing of the acceptance criteria in Figure 1.1:

**ACCEPTANCE CRITERIA FOR RESTFUL API**

AC1: Add an item to an empty list  
Given: A create item button And: There are no items in the database When: I send a 'CREATE' request  
Then: I can see a list of items containing my new item

AC2: Add an item to an already populated list Given: A create item button And: There is one item in the database When: I send a 'CREATE' request. Then: I can see a list of items in alphabetical order, containing my new item

**ACCEPTANCE CRITERIA FOR UI**

AC1: Landing page with one item  
Given: The user has accessed the system And: There are no items in the database When: I send a 'CREATE' request  
Then: I can see a list of items containing my new item

AC2: Landing page with multiple items displayed Given: The user has accessed the system And: There is one item in the database When: I send a 'CREATE' request. Then: I can see a list of items in alphabetical order, containing my new item

Figure 1.1 – Acceptance criteria

I felt that these acceptance criteria fully covered everything the ticket/scenario was requiring and any edge cases that could have caught us out. For instance, rather than just adding an item to an empty list and being happy that it works, it is necessary to then continue to build upon that and add multiple items so that I can ensure the whole process not only works with one single item but where multiple items are grouped together. This is much more evident when building a UI as some components’ behaviour may change based on the number of items that may be contained. Therefore, I believe the acceptance criteria in Figure 1.1 fully covers the ticket requirements.

For some context, the requests been made here will be made through the UI. However, the items associated, and the database that they are housed in, are managed by our ‘Information Service’. It is therefore vital to ensure an efficient connection between the two so that requests can be made as quickly as possible.

A picture containing game

Description automatically generatedFigure 1.2 shows a control flow diagram I created which demonstrates how the UI interacts with the information service and then the database. The information service and the database or in line as they are directly linked and currently, only the information service has access to this database. It is just for the storing of items. The blue arrows show the flow of requests from the user all the way to the database, and the red arrows show the flow of the data back to the UI.

Figure 1.2 – The flow of data

**Creating the Project**

Now that I had written the user stories, my first task was to set up a new project. This process is a fairly standard one but is one that is set out by my company to ensure a high and consistent standard between all projects.

Using the Gitlab UI, I created a new project called ‘thing-information-service’ which was initialised with a README.md. I then clone down this project locally using the commands shown in Figure 2.1:

**

Figure 2.1 – Git commands

As this project is going to use Spring, I then headed to <https://start.spring.io> to use the Spring Initializr. I needed to add in web dependencies and then click generate which provided me with a simple Spring Boot application with dependencies already pre-populated. This project code then needed to be added into my thing-information-service under a ‘service’ directory, so I opened up Intellij IDE (which is where I made changes to my code) and added it all in. I finally added a .gitignore file to prevent unwanted files being committed to Gitlab. The project structure is shown in Figure 2.2 (picture taken from the IDE):

A screenshot of a cell phone

Description automatically generated

Figure 2.2 – Project structure

Happy that the project skeleton was complete I pushed my changes to my branch and created a merge request ready for a member of my team to review and approve before merging into the master branch. If, at any point, my colleague had comments on the merge request, I would make the suggested changes and resubmit for approval, however this time my branch was merged without needing any amendments.

**Creating the RESTful API**

Spring framework applications are perfect for performing CRUD (create, read, update, delete) operations on a database or repository and it is the create part that I am interested in for my user story.

I didn’t have much knowledge on which database to use in a Spring application, so I initially did some research. I came across lots of StackOverflow posts with people commenting that Java Persistence API (JPA) and Spring Data JPA were a good combination. Reading further, I found that JPA provides the user with a simple way to perform CRUD operations on a database by mapping Java objects to the fields in the database tables. This layer of abstraction means I can keep my code clean and simple without the need to write long SQL statements for CRUD operations, as the JPARepository class will handle this.

Now I needed to create a database to store these items. Its structure needed to be simple, the item itself just needed to have a unique id (ideally starting at 1), then columns would link directly with the information that would be passed through in the Java objects. I turned to my team lead for support on this issue as I wanted to choose a database that the project was happy to use, one that had been set in our delivery plan. I was informed we would be using a Postgres instance of database using JDBC (Java Database Connectivity) Interface. I contacted our platform team and asked them to set up a Postgres instance that I could use.

I now had to configure my project to use the database. In Spring applications, configurations are set in the application.yaml. The application.yaml is where the project looks to determine which profile to use, whether that be ‘test’ or ‘development’ to name two examples. It also then determines what extra frameworks each profile will use. In our case, we are configuring Spring JPA to work with a Postgres dialect. The datasource points to a database with at a specific url, username and password. All these values are stored as environment variables for security reasons.

The application.yaml file can be seen in Figure 3.1.



Figure 3.1 – Application.yaml

I wanted to encapsulate my code effectively by splitting out the business logic into three classes; ItemController, Item and ItemRepository. This would allow my code to follow the single responsibility principle.

However, while developing this RESTful API I wanted to follow a test-driven development (TDD) approach. TDD reduces the number of bugs in production and improves code quality. In other words, it makes code easier to maintain and understand. This is because the user will write a failing test first describing the ideal behaviour, before implementing the feature code. As the controller was the only part of the code I needed to test, I started with that.

Figure 3.2 shows the initial test I wrote.



Figure 3.2 – Test case

If I run my test for the first time it immediately fails with an error saying that it cannot find the ItemController class. So, to fix this error message I go and create the ItemController class as seen in Figure 3.3.

Figure 3.3 – ItemController class (1)

You will notice that the class is empty. I know that it is not ideal to have an empty class but that is not the objective with TDD. I want to do the minimum possible to make my test pass to keep my code clean.

So, I run the test again. This time it gets past the ItemController class error and is now telling me that it cannot find the Item class. So, to fix this I go and create the Item class as seen in Figure 3.4.

Figure 3.4 – Item class

The public fields within the Item class are the three pieces of information that we want to store within the database. The @Getter and @Setter annotations above the class name are Spring annotations that belong to Lombok. Lombok is a tool that creates getter and setter methods on all your class fields. This make the code much, much cleaner as it removes all these methods (two for each field) in the body of your class and hides them behind the annotation. The @AllArgsConstructor & @NoArgsConstructor annotations also save us writing lots of extra code, in this case it would be the constructors. Here I am stating that the Item class can be instantiated with either no arguments, or all arguments. The no arguments annotation will especially come in handy when testing as we don’t have to build out full Items.

I run my test once again. Now it is telling me that it cannot find the ItemRepository class. Again, to fix this error I will need to go and create this class. It can be seen In Figure 3.5.

Figure 3.5 – ItemRepository class

As mentioned before, I wanted to use JPA and the way that I do this is to have my class extend JpaRepository. The extend keyword indicates that my ItemRepository class will inherit from JpaRepository along with all the default methods it implements. This allows me to keep this class clean while still maintaining functionality. The ItemRepository class is empty at the moment. I do not need to create and extra methods as all the functionality I need it already baked in behind the scenes. I only need to use the save() method housed in the JpaRepository class and using this ItemRepository class as an interface will allow me to access that.

The @Repository annotation marks the specific class as a Data Access Object, thus clarifying its role within the Spring project.

In the ItemControllerTest class above I am using the JpaRepository’s in built save() method on my itemRepository object to save my item to the database that I specified in the application.yaml. You can see that the use of the JpaRepository means that my code can be really clean, concise and easily readable to any other developer that may want to come along and alter this code in the future.

Because of Spring and JpaRepository, the SQL commands are hidden from the work I do, therefore negating the need for me to write them. However, it is still useful for me to know which SQL command is running behind the scenes as this gives me greater visibility and understanding if it comes to manipulating this data in the future. That command would look something like Figure 3.6.

A picture containing table

Description automatically generated

Figure 3.6 – Example sql statement

The command in Figure 3.6 adds something to the table “table\_name”. The values that it is given represent each of the columns within the database, which correlate to the columns seen in the UI.

Now, running the test once more, its fails in the @BeforeEach method. This is where I set up some common test objects so that the same code isn’t repeated throughout each test. This time the failure is stating that the constructor for the ItemController is expecting no arguments, but it was given one. To fix this I need to create and ItemRepository object within my ItemController class. This can be seen in Figure 3.6.

Figure 3.6 – ItemController class (2)

Running the test once more, it finally fails in the actual test method. It is failing on the itemController.create(flag) method as the error message is saying that it cannot find the method create() within the ItemController class. I implement that method with all its functionality as shown in Figure 3.7

Figure 3.7 – ItemController class (3)

Running the test one final time, it passes. This may seem quite long-winded reading it, but following TDD the way I did, not only helped me understand the relationships between all the classes within my project but enabled me to keep the code as simple as can be. It is also good to try and catch any bugs as soon as they happen and isolate them to particular lines or functions. One line is a lot easier to fix than a whole test class that could have multiple bugs within.

**Building the User Interface (UI)**

**NOTES**

* By the time I came around to this part of the ticket, one of my colleagues had built the basic UI project using React and implemented a get function which currently displays all the Items within the database. I would go on to use this but also build upon it for my ticket.
* I continued to use TDD but to save myself repeating the whole process I will skip over that and just talk about how I implemented the create() method on the UI side.

The first thing I had to do was create a button that would allow me to add an item to my database. This button would automatically use the uk.gov theme that we were using for our UI, the only major decision I had was where to put the button on the page itself. I googled how to create a button in React just to confirm to myself that I remembered from the react I did at Makers Academy. The code in Figure 4.1 is what I wrote to create the button.

Figure 4.1 – React button

In Figure 4.2 you can see how the button looks and how it is placed on the webpage.

A picture containing bird

Description automatically generated

Figure 4.2 – Initial UI

Here the table sits below the redacted header and then the ‘Create Item’ button sits just below that. I thought this would be a very logical and practical place to place the button and it stands out well compared to the rest of the page. The table is currently empty as there are not Items within the database.

As stated before, the UI needed to be simple and throwaway as there would not be and end user as such (no one besides our dev team will use this UI). It just needed to be clear and concise so that our real end users (our developers) could easily read the code, use the functions, test and debug where necessary. We do not have a test team within our project, so we become the testers. This simple design ultimately has the aim of making our lives as easy as possible when it comes to editing code and find bugs that may arise.

I gave this button an easily recognisable className so it could be found when performing operations. The onClick field defines what happens to the app when the button is clicked.

Figure 4.3 – Refactor of create function

The method in Figure 4.3 was purely created for ease of reading and to tidy up the part of the class that rendered the page as it is bad practice to clutter that. The final part between the <Button> tags is what text I want the button to have. Here I went for the self-explanatory ‘Create Item’.

The ‘/create’ endpoint had been created by a different team member of mine. In its current state it just has three fields of: ItemName, ProcessName & Description. This is the information that will be redirected to the InformationService in the form of an Item object, and ultimately is what I store in the database.

Finally, I wanted to see it all working properly by seeing the populated table on the home page. As you can see from Figures 4.4 and 4.5, I was able to add one, and multiple items to our database, which are then populated in the home page table (thanks to a piece of work a colleague produced).

A screenshot of a cell phone

Description automatically generatedOne Item:

Figure 4.4 – UI one item

A screenshot of a cell phone

Description automatically generatedMultiple Items:

Figure 4.5 – UI two items

**Conclusion**

In conclusion I think the work I have done here provides an efficient way to connect to our postgres database and fulfils the acceptance criteria.

I am also happy in the fact that even though the UI is throwaway, and just used to prove out our methods and concepts, it still is appealing to read and easy to navigate thanks to the simple nature of the table and buttons. This means that when another developer needs build on this project, they can easily follow the work that I have done, hopefully resulting in a shorter time needed in the transfer of knowledge and a faster completion of the ticket that they are working on.

The control flow diagram I created, while simple, helped me see how all the pieces fit together in this ticket, also ensuring that what I am building represents what is seen there.

It was really interesting to get stuck into CRUD operations, seeing how we will utilise them further in future tickets. It was also good to be given some responsibility in creating the button, which we will now use as a standard for any button we need on our UI (as it saves us making any more and it already fits well with the UI). The business impact of this work means that we can move forward with building out our system, improving the quality of our work and making sure it can be refashioned in the future to link in with the UI of the finished product. Building our own throwaway UI means we can be more efficient in testing and ultimately this means that as a project, we hit our sprint targets and in turn the client is happy with the work we are doing.