# CodeClan Final Project Backend Script

The backend consists of the Java based Framework, Spring and the SQL based database, H2. The choice of database was mainly because we've used it before and had no issues with the driver used to connect to it. Given more time we'd have opted for a NoSQL database such as MongoDB due to the flexibility it offers. We're also using Java 8 rather than more advanced versions of the language since we're more familiar with 8.

The backend and frontend technology aren't heavily reliant on one another so you could in theory replace Spring with Express and H2 with MySQl and have few issues. The backend's prime role is just to provide JSON taken from a database for the frontend to manipulate and store the information sent to it. The backend has full CRUD functionality, as for the U part we went with PUT over PATCH just because it was simpler to implement. The Create, Read and Delete aspects were tested both in JUnit and Insomnia whilst the Update aspect was just tested in Insomnia. The GETs vary from GET ALL to GET one entry with a specific id and various other GET filters mostly returning JSON files. A rare few actually return Longs such as a count of how many reservations a specific customer has made.

It is also possible to delete via a specific id or delete everything in a specific table if you need to clear out its content without resorting to a full drop and recreation of every single database table.

We're using the these dependencies for Spring Boot 2.3.1,  "Spring Boot DevTools" to assist us with development and fast application restarts, "Spring Web" to help us build our controllers, "Spring Data JPA" to help us interact with the database and as previously mentioned "H2 Database". IntelliJ IDEA was used to write the Java at a rapid pace. It's a superb IDE capable of rapidly generating the getters, setters and constructors we need in our models which are indispensable for the Object Relational Mapping with our database.

We have 4 database tables, Venues, Venue Tables, Customers and Reservations. All of these tables obviously have ids in the form of Longs to use as primary keys.

Venues just have names, currently we are just working with one venue called "The Capybara Cafe" but if the future more could be used.

Venue Tables holds a cover (the amount of people you can seat at the table) and an id relating to which Venue it belongs to. We have to call them venue tables so they aren't confused with regular database tables.

For Customers we have an email, first name, last name and phone. This is the customer that calls in to make the reservations & can be contacted if there's a change of plans rather than the entire party that shows up.

The Reservations table has a start and end date using local date-time, the size of the party attending and each reservation must contain a customer and venue table. The front end's form prevents you from being able to make a reservation with a party size that is greater than the amount of people you can seat at a table. An optional value that can be entered is called "reservationNotes" where you can add information related to the party such as allergic to "nuts", or "sing happy birthday".

[Show class diagrams]

The Java packages consist of "components" that holds our data loader, "models" and "controllers" whose names are self explanatory in an MVC context and "repositories" whose files are interfaces that utilise the JPA Repository so we can interact with the database.

The current app's front end is currently fetching all the results from 3 tables, customers, venue-tables and reservations. The front end also allows Posts to be made to the customers and reservations tables.

[Show image of get requests]

One of the things I disliked about apis used during some of our tutorials was the limited amount of filtering GET requests you could make which inspired me to experiment a great deal with the database querying that JPA offers. There is a great deal of future functionality in the back end with lots of specific filters which are provided with example urls. This is in case one day we decide to port the app to another front end framework that behaves different from React or just retrieve JSON files via vanilla JavaScript. So for example in the backend there are GET urls that allow you to retrieve customers by their first name and last name, the start of their last name, some letters the last name contains for those that find spelling tricky or view all of them in descending order of their last name whilst ignoring case. There are date filters for reservations using crude "greater than" and "less than" which work fine when Unit tested but I've failed to be able to carry them out in the browser without resorting to mapping errors.

So far there are 58 unit tests from making sure the setters in the models are working to checking that entries get deleted and the correct ones are getting retrieved from the database.