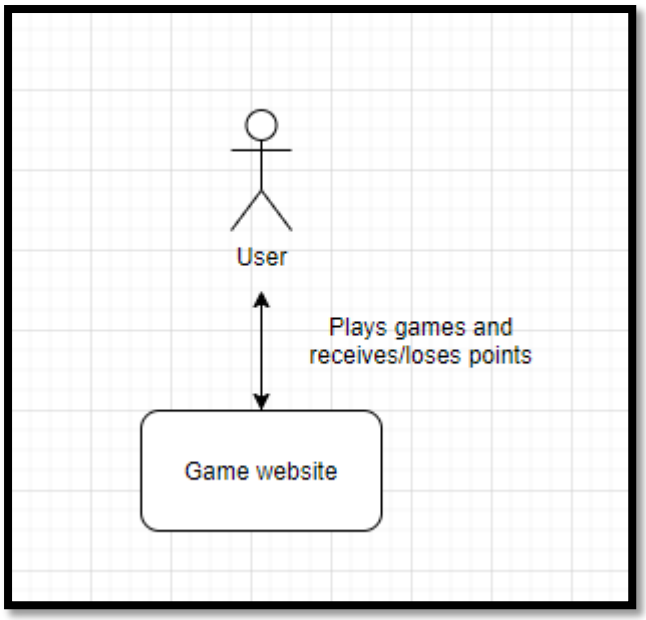
Design document

Sam Philipsen

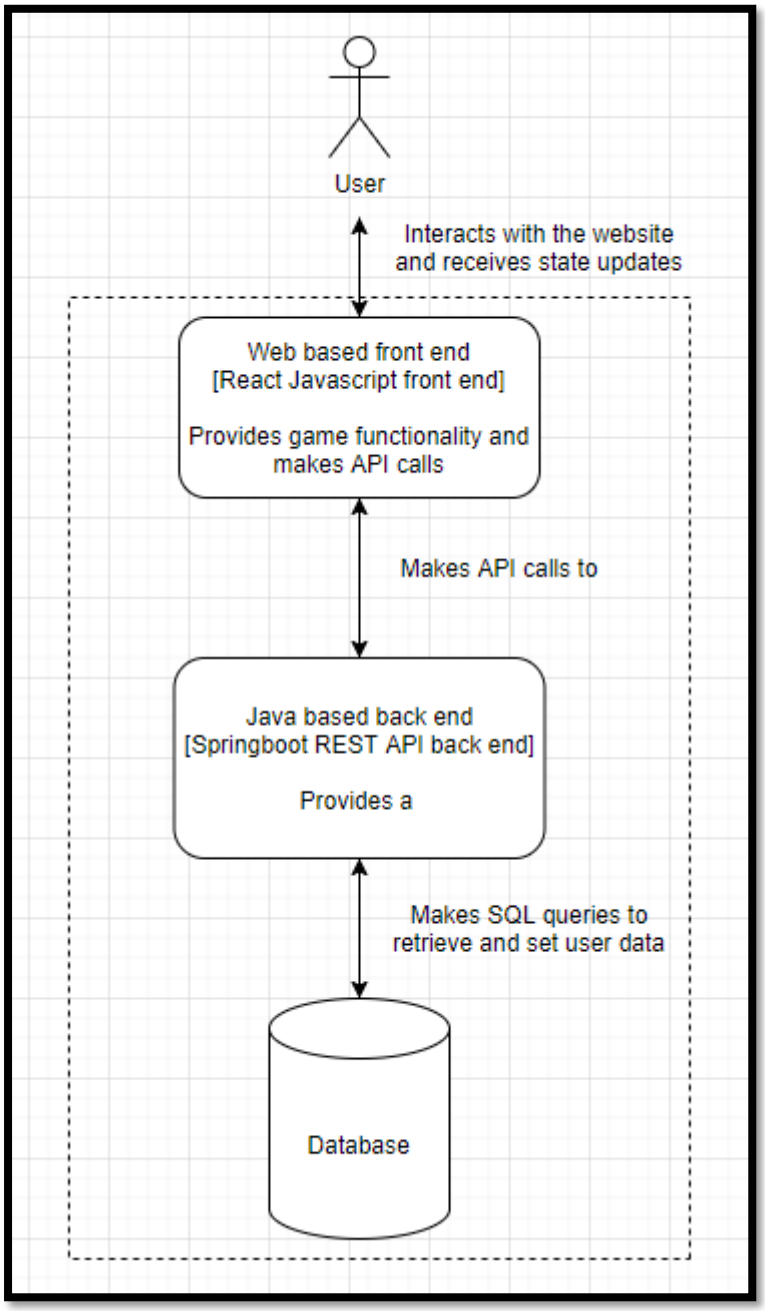
Inhoud

C4 diagrams

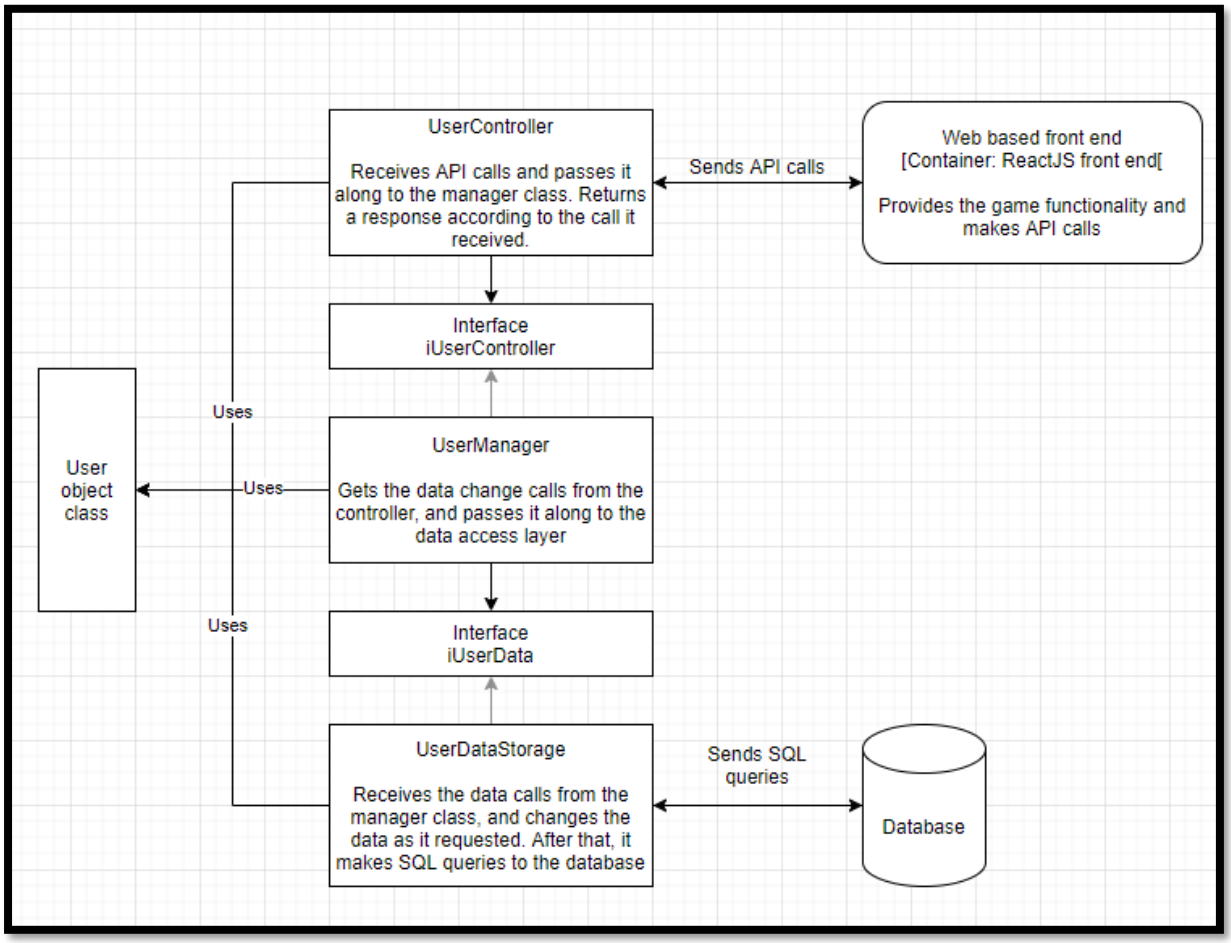
C1



C2


User
Interacts with the website
and receives state updates
Web based front end
[React Javascript front end]
Provides game functionality and
makes API calls
Makes API calls to
Java based back end
[Springboot REST API back end]
Provides a
Makes SQL queries to
retrieve and set user data
Database

C3


UserController
Receives API calls and passes it along to the manager class. Returns a response according to the call it received.
Sends API calls
Web based front end
[Container: ReactJS front end[
Provides the game functionality and makes API calls
Interface
iUserController
Uses
User object class
Uses
UserManager
Gets the data change calls from the controller, and passes it along to the data access layer
Uses
Interface
iUserData
UserDataStorage
Receives the data calls from the manager class, and changes the data as it requested. After that, it makes SQL queries to the database
Sends SQL queries
Database

Design decisions

As seen in the C4 diagrams above, the system is currently built up like the following:

The user can interact with the web based front end service. These interactions are planned to be logging in and playing games, though more might be added in the future. Once the user tries to log in to the system, a HTTP request with the filled in user details is created. This request is sent to the Spring boot API. The request is received by the controller class, which then tries to get the required information from the manager class. It does this through the IUserManager interface class, as to avoid every program in the system from depending on each other.
The manager class then tries to get the requested data from the data access layer, also going through an interface in the process. The data access class gets the requested data (the user's information) and it all gets passed back to the API controller class, which sends the data to the web service. Once the web service has this information, it can now display it and store it for further HTTP requests.

Why dependency inversion?
The reason the manager and database access layer have interfaces is because of dependency inversion. If I'd want to test a single method in one of the classes, it would need to load every class just for testing a single method. To prevent this, these classes are connected to an interface. This means that I can put a fake data class instead of a real one so it does not have to load the real one each time.

Why spring boot?
Spring boot is used because it is an easy to use, but still very functional way to make the web API work. The required setup and learning curve is minimal compared to other popular web frameworks. It is also very easy to connect to the rest of my application.

Test plan

For the testing of my classes I will be inserting unit tests for every major class, like the controller, manager, and repository classes. This is to ensure they all work according to what I designed, and it does not fail.

Integration testing is not necessary because the unit tests already provide enough coverage of my classes' methods. I chose the unit tests above integration testing because I can check each method individually if they work or need changing.

In the future I might try TTD (Test driven development) if there need to be more classes, but for now this is it.

For a full list of the tests, I plan to complete at the end, see the project planning document.