

Design document

Sam Philipsen

Versioning table

DATE	VERSION	CHANGES
05-11-2021	V1.0	Created design document with C4 diagrams, test plan and design decisions merged into one file.
26-11-2021	V2.0	Added CI setup diagram and reasoning for why the game logic is in the front end. Also added reasoning behind the C4 diagrams.
17-12-2021	V3.0	Removed explanation of why the game logic is in the front end, and added why the game logic is in the back end. Also added security header in design decisions tab. Finally, added versioning table.

Table of contents

Introduction	4
C4 diagrams.....	5
C1	5
C2	6
C3	7
Design decisions.....	8
Test plan.....	10
CI setup diagram	11

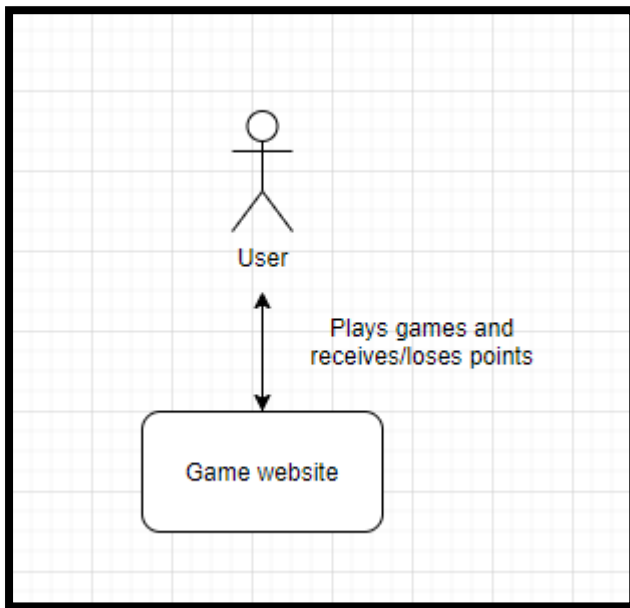
Introduction

Designing a full stack web application that uses your own API which is connected to a database means there are a LOT of decisions you have to make. Each of these decisions are important to the final design of the application. It can determine the framework to use for the API or the website, or it can simply determine how it's going to handle the game logic. These decisions should be thought of carefully and have a good reason behind their implementation.

In this document I will be explaining how I designed my application, and why I made the decisions that I made.

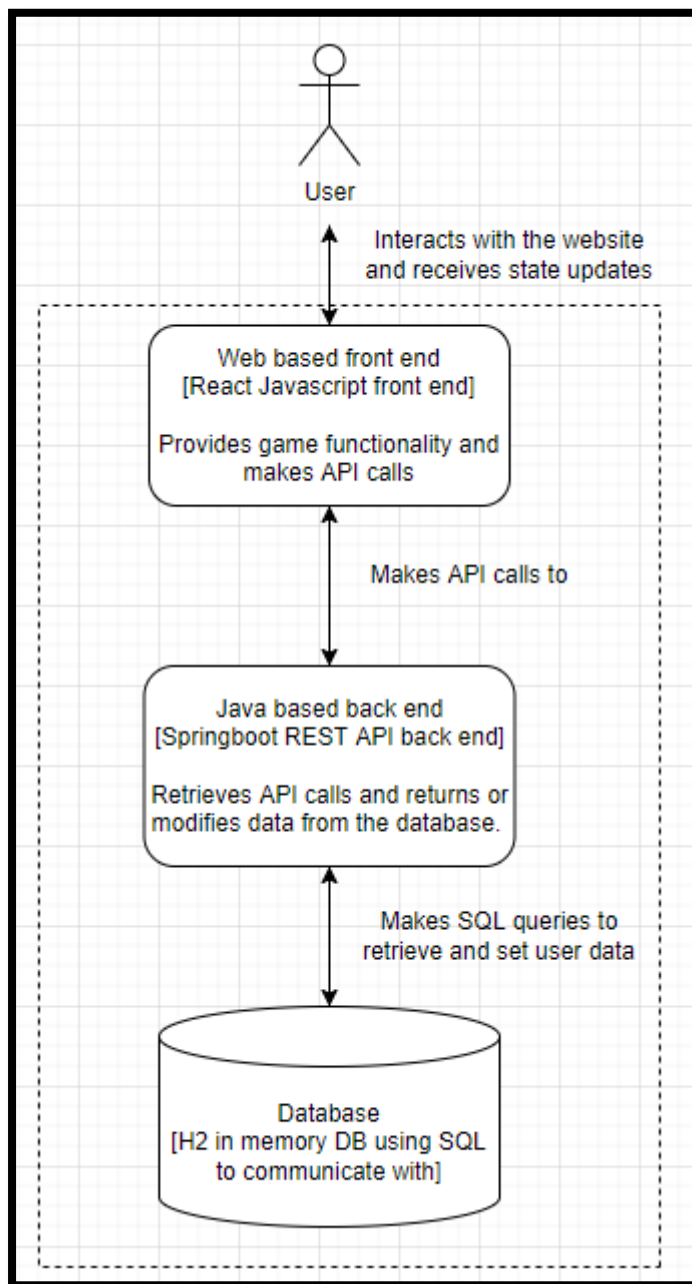
C4 diagrams

C1



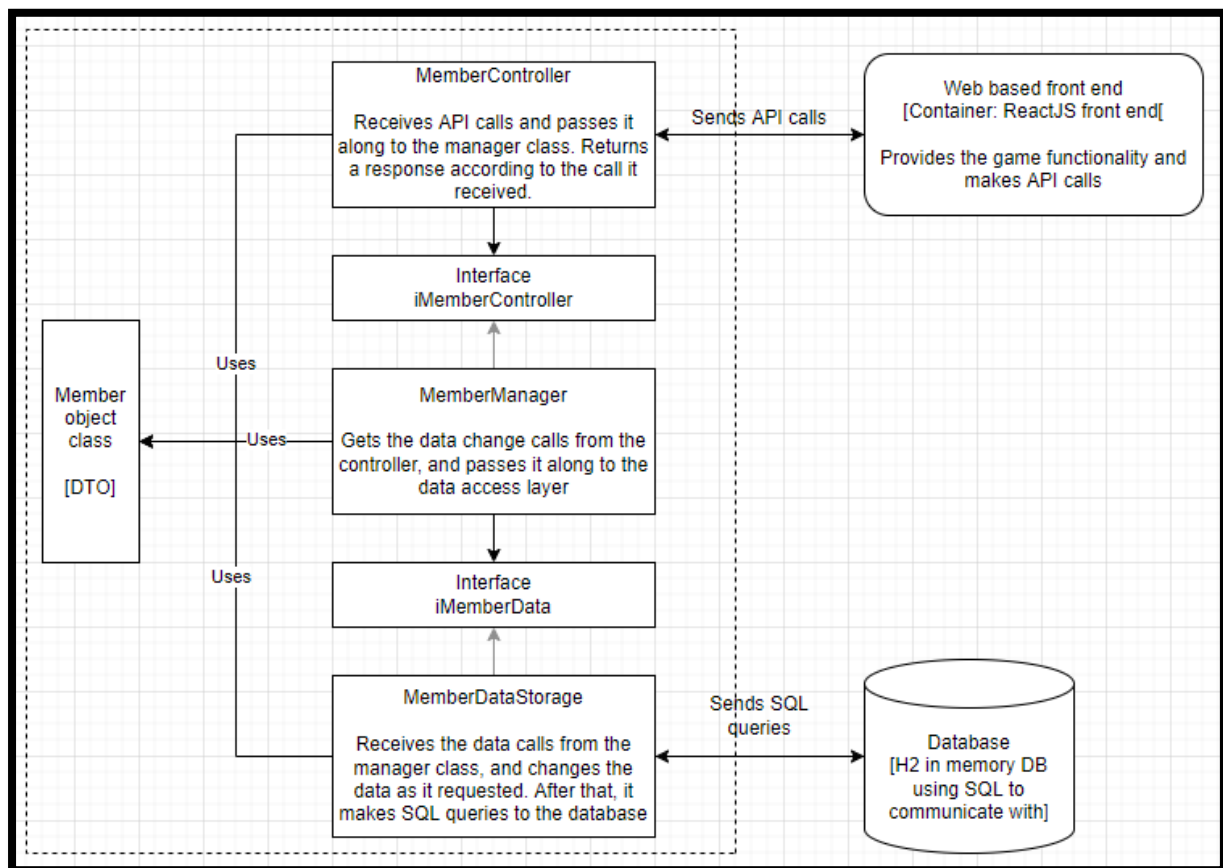
The user that wants to use the site logs in. They are logged in with their account and can play games. When they win or lose these games, their account gains or loses points respectively.

C2



The user plays games on the site to earn points. Once the game determines they won/lost points, it makes an API call to Spring boot. The REST API looks at the request and decides what it has to do with the data. It then send an SQL query to the H2 database in the memory, either changing or retrieving data on it depending on what the API call was. Spring then returns a response back to the front end website, where it can display the changes.

C3



Once a user has won a game, the front end sends a HTTP request to the backend. This request is picked up by the controller class where it determines what the request wants. Once the controller knows what the request is about, it calls a function from the MemberManager through the IMemberManager interface. The manager class then calls a function from the Database management class (MemberDataStorage) through the interface, to retrieve or edit data that is inside the database. The Database management class sends SQL queries to the database. The database then either edits data inside or returns requested data. The requested data or simply an ok/not ok then steps down the entire ladder it just took to get to the database, and is returned to the front end.

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.

Why is game logic in the back end?

The game logic has been placed in the back end instead of the front end. Before, it was in the front end. This was to avoid constant HTTP requests being sent to the API which could be interfered with by things like the user's internet connection.

While this is still true, it is much more secure to store the game logic on the back end. This is because if it is stored on the front end, the user directly has control over the code. This means they can change the code so they always win, or directly just get millions of points.

This is avoided by having the game logic on the back end and having the front end have access to it through specific endpoints which should root out the security flaws that are currently in place.

Security

The application currently is secured with the use of JWT tokens. The user has to log in first, where the password is sent to the backend to be checked. If it's correct the user will get an authorization token that allows it to make further API calls. This stops unauthorized users from making API calls and having access to everything.

The application also encrypts password using the BCrypt library so the passwords are not stored directly on the database. In the case of the database being hacked, the passwords should still be safe as long as they are not unencrypted.

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.

CI setup diagram

