Design document

Sam Philipsen

Inhoud

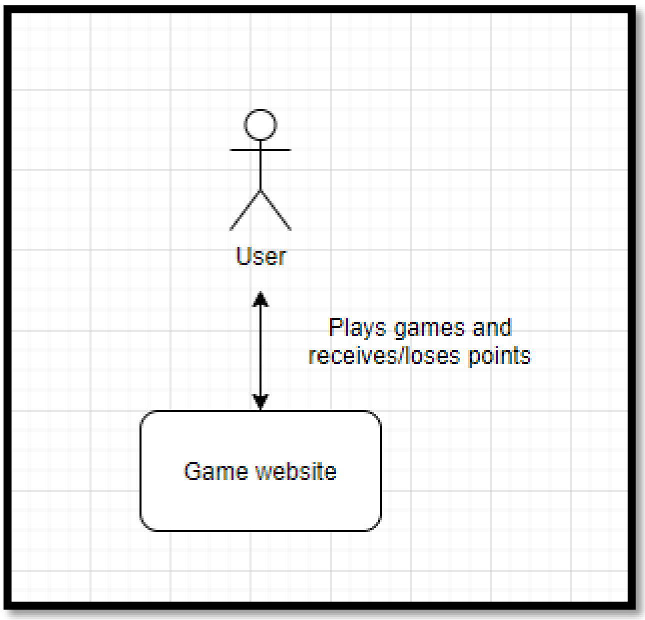
Introduction

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.

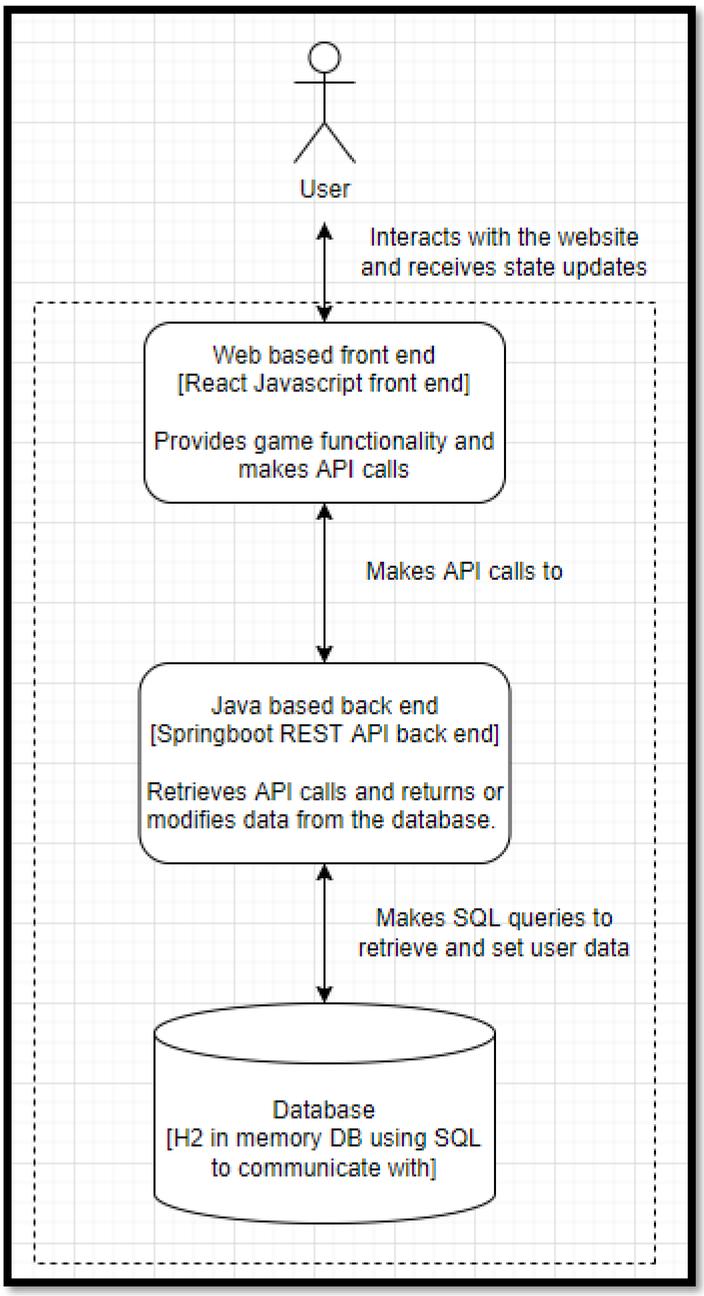
C4 diagrams

C1



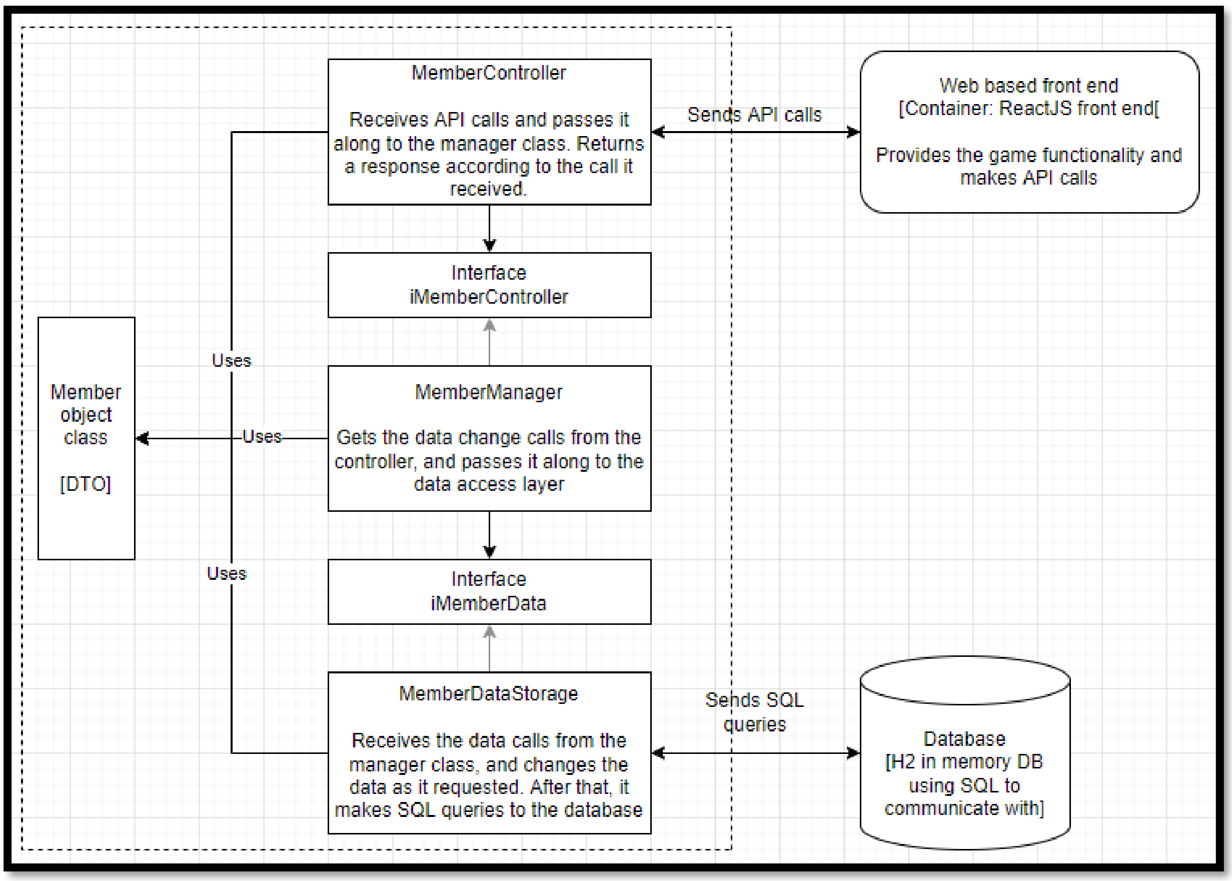
The user that wants to use the site logs in. They are logged in with their account and can play games. When they win or lose these games, their account gains or loses points respectively.

C2


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]
Retrieves API calls and returns or modifies data from the database.
Makes SQL queries to retrieve and set user data
Database [H2 in memory DB using SQL to communicate with]

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


MemberController
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
iMemberController
Uses
Member object class
[DTO]
Uses
MemberManager
Gets the data change calls from the controller, and passes it along to the data access layer
Uses
Interface
iMemberData
MemberDataStorage
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
[H2 in memory DB using SQL to communicate with]

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 front end?
The front end (ReactJS framework) handles all the game logic. This is mostly because this means the program does not have to constantly make HTTP requests to my API, and then have the webpage wait for a response to display the information. This is much slower than just putting the game logic in the front end. This is especially noticeable if you have a slow internet connection. It also puts much more strain on the back end if a lot of users are trying to play a game and are constantly making HTTP requests to Spring.

It is also much easier to code the game logic with React itself because you can instantly see the changes you made and the result. Compare this to handling the logic in Spring, where you would have to constantly re-build the application and wait for it to launch and connect for every change you made, small or big.

Test plan

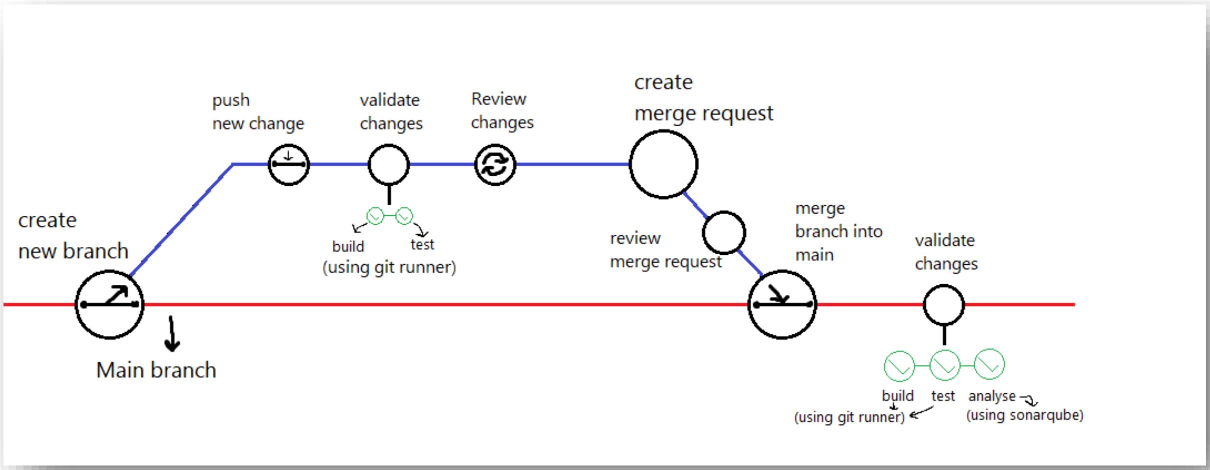
For the testing of my classes I will be inserting unit tests for every major class, like the controller, manager, and repository classes. This is to ensure they all work according to what I designed, and it does not fail.

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


push
new change
validate
changes
Review
changes
create
merge request
create
new branch
build
test
(using git runner)
review
merge request
merge
branch into
main
validate
changes
Main branch
build
test
analyse
(using git runner)
(using sonarqube)