Designing a gaming site

Sam Philipsen | 12-1-2022

# Versioning table

|  |  |  |
| --- | --- | --- |
| Date | Version | Changes |
| 05-11-2021 | V0.1 | Created design document with C4 diagrams, test plan and design decisions merged into one file. |
| 26-11-2021 | V0.2 | 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 | V0.3 | 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. |
| 12-1-2022 | V1.0 | Changed the C4 diagrams to reflect the design changes of the application and added a description with it. Also changed the title and moved the test plan to its own document. |

# 

Table of contents

[Introduction 4](#_Toc88853942)

[C4 diagrams 5](#_Toc88853943)

[C1 5](#_Toc88853944)

[C2 6](#_Toc88853945)

[C3 7](#_Toc88853946)

[Design decisions 9](#_Toc88853947)

[CI setup diagram 11](#_Toc88853949)

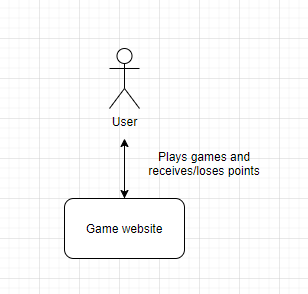
# Introduction

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.

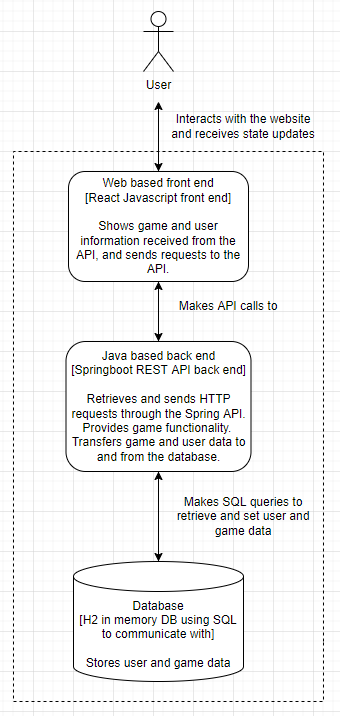
# C4 diagrams

## C1



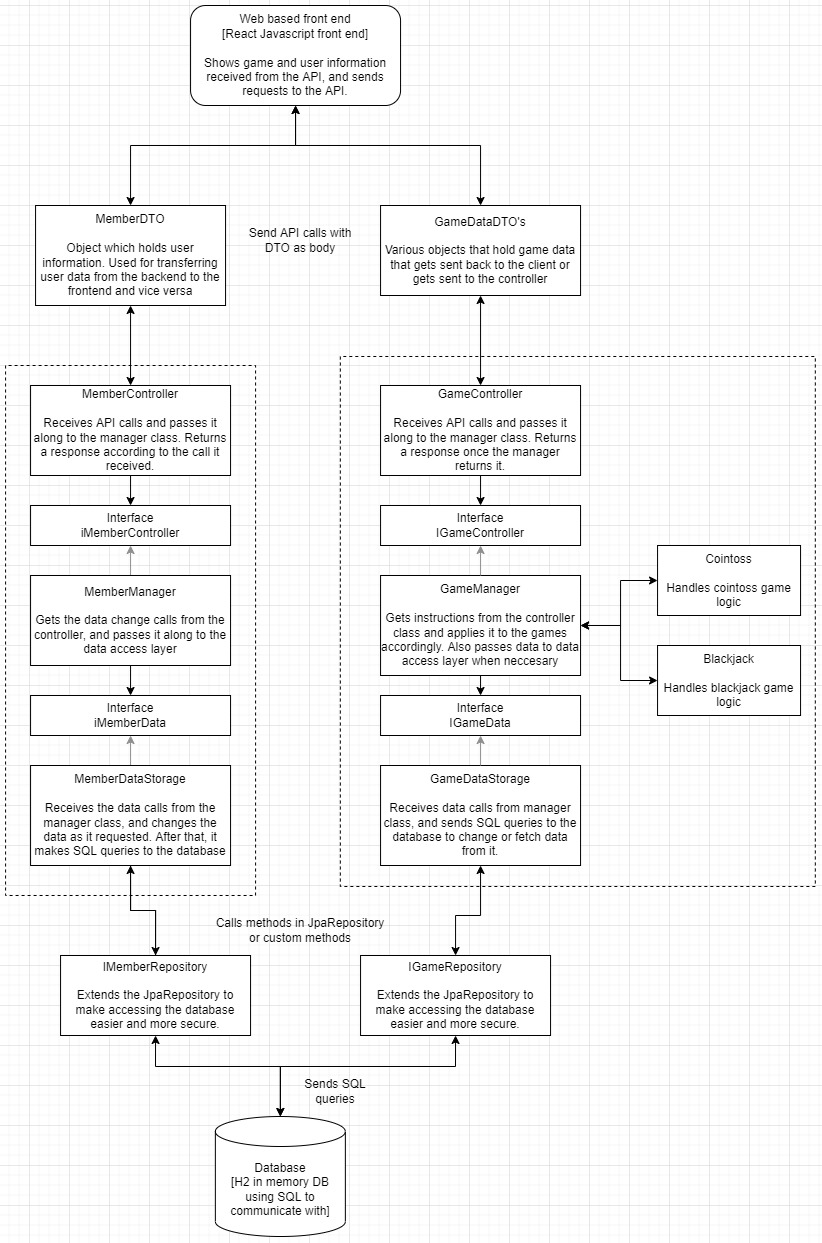
The user that wants to use the site logs in. They are logged in with their account and can play games. When they win or lose these games, their account gains or loses points respectively.

## C2



The user plays games on the website to earn points. The website receives user and game information from the REST API that runs on the backend. The user can send http requests to the API (through the website) to play games. For example, the user picks a coin side, puts in a bet and presses the play button. The website then makes a request to the API where the chosen coin side and the bet is taken, the game logic runs (the coin is flipped) and it is determined if the user won or not. The user either loses or wins points depending on the outcome of the game. Then the back end sends an SQL query to the H2 database which changes the points of the user to the new value that was determined. Spring then returns a response back to the website where it displays the outcome of the game, and the new points of the user are displayed.

## C3



**User logging in (left side)**

The user on the front end interacts with the website to generate HTTP requests. Once the user is logged in it needs to get its information from the database. The website sends a request to the API where it is picked up. 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 calls a method from the IMemberRepository which extends the JpaRepository interface. The interface sends an SQL query to the database that retrieves all the information the user needs.

This data then takes the ladder down back to the controller class. There, the data is put in the MemberDTO object which is sent back to the user where it is unpacked and the information is displayed on the webpage.

**User playing a game (right side)**

The user wants to play a game of cointoss. He puts in a bet and the coin side he wants to use and presses the play button. A request is sent to the GameController class with a GameDataDTO as the body. In the body is the user’s id, the user’s bet and the chosen coin side. The controller finds the user using the ID that has been sent, and the rest of the information can be used to play the game.

The controller sends this information to the GameManager class through the IGameController interface. There, it creates a new Cointoss class. It fills in the information in the chosen coin side and picks a random side of a coin. If the side of the coin that the user chose is the same as the side of the coin that the class chose, the user wins. The data of the Cointoss game object is checked in the manager class to see who the winner is. It then either increases or reduces the points of the user and stores the new information in the database using the MemberManager class.

The result of the game and the new points of the user are then sent back to the controller class where it is sent to the website through another GameDataDTO. There the information is displayed, and the user gets to know if he won or not.

Alternatively, if the user wants to play the blackjack game this follows a similar path at first. However instead of immediately deciding the result of the game, the game’s information is stored in the database. The state of the game is sent back to the user, and they can decide what to do with the game further. They can choose to hit or stand. If they choose hit the website sends a request to the GameController class again with the game’s ID and the action they want to perform in a DTO. In the controller class this data is sent to the GameManager class through the IGameController interface. There, it fetches the game from the database using it’s ID in the same way a user is fetched in the MemberManager. Once it has the game, it draws a new card from the deck in the game as per the user’s request.

If the user lost after this request, the points are reduced from its account like explained in the Cointoss example. Despite what happened, the game’s state is returned back to the website through the GameController using a DTO where it displays the game’s state. Here, the user can see the game’s state and if he has not lost yet, choose the following action he wants to perform.

# 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 IMemberManager 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.

# CI setup diagram

