Class ISS202 – Team 1:

Hamza el Haouti

Haris Voloder

Hidde Balestra

Technical Documentation

Project Enterprise Web Application (EWA)

Table of Contents

1 Introduction

The purpose of this document is to inform the deployment and/or future software engineers. So they know how the project is build. This useful for when this project needs to extended or modified for future development.

Therefore, we will share:

1) Our vision behind the product;
2) A summary of most important epic user stories;
3) Class and deployment diagrams;
4) The technical challenges;
5) An analytical reflection.

2 Product vision

For this project we delivered a monopoly game as a product. The product has a couple of key features.

One of them is the ability to create invites so other players can join your created monopoly game. Because of this feature a player can seek out other players to play a game with. The invites can be created with certain criteria for a game, such as max number of players for the game, max time per turn and maximum game time.

The most important feature that we have built in is of course the ability to play a game of monopoly. This big and important feature is split into a lot of smaller features to make the game work.

One of those important smaller features is the ability to roll a dice and based on the roll you can move along the gameboard. The ability to interact with the `places` or cards on the gameboard is also an important feature. The player can click on a card and see who owns it. When a player has clicked on a card, he or she is able to view how much the properties will cost to own, such houses and hotels. One other important feature is the ability to buy properties. A player can only buy property when the player is standing on the card or property at that moment.

For this product we also have a target user community in mind. We are trying to target a user community that likes to play physical gameboard but wants to play them at a quicker pace and with more/other people. The pace of the games will be quicker because the players do not need to set up a game of monopoly for themselves. Everything will already be automated so the pace of the game will be quicker. The admin does have to specify some game settings and the game will be ready to go once players have joined the game.

A player of our target community also has the interest to play the game with a lot of different people so he or she does not have a problem to start a game of monopoly. Our product provides the accomplishment of this interest because every user in our application can join a game once they have created an account and no other requirements are needed.

Our product does have a lot of competitors. We have online competitors for the gameboard such as, Uno, Scrabble, Chess, Battleship, Risk, Yahtzee, etc. There are also other versions of the online monopoly gameboard available, so these versions of the game can also be counted as our competitors. Once we see all our competitors, we can say that our position in the market will not be the best, as our version does not have much differentiation and has a lot of online game versions that provide the same features but with a bigger userbase.

There are a couple of directions in which we see potential for future growth of our product. The most interesting one would be to make the game available on more platforms. One of the abilities here would be to make the game available in the app store and android play store. So, users also could play the game on an app on their phone if they prefer this over joining a game via their browser. This would also make our product grow because the product would be more accessible

to potential users. Other ideas would be to add the game as a feature in social media apps such as Facebook, which already has a lot of games for users to play and a lot of users.

Another future we would like to see, is the ability for a user to design their own gameboard with unique locations and (chance and chest) cards. These gameboards could then be shared by players with their friends. The official Monopoly is available in allot varieties, but giving people the ability to create a version of a lesser known, but just as beloved, city/village, would set our game apart.

3 Summary of most important epic user stories

During this project we have completed a couple of important epic user stories. In this chapter we will summaries these important user stories and their most important acceptance criteria.

3.1 As a user I want to be able to register and identify myself on the platform, so that I can profile and protect my identity.

For use to complete this user story we had to complete a couple of important acceptance criteria. First, we had to create a registration page. In this page we included the following required data for a user to fill in to create an account: a username, password, geographical location, and email address. An optional choice for a user that wants to create an account would be to create a profile picture, but this is not necessary for the completion of an account.

Besides this acceptance criteria for completion, we also create a login page in which a user must fill in his or her username and password to login.

There is also a security criterion that we added for this epic user story to be completed, which is the following: routes/Rest-endpoints on both the front-end and back-end are unreachable for unauthenticated and/or unauthorized users.

3.2 As a user I want to be able to start new instances of the game.

This epic user story focuses more on the creation of a game and inviting other players to join the created game.

One acceptance criteria was of course the creation of a page where the user can create a game and set certain settings for how the game should be played. Those settings are max players for the game, maximum time per turn, max game time and minutes before the game starts. Once the admin has chosen the settings, a button had to be created with which the admin of a game could send invites to all players and wait for them to join the game. After these criteria have been complete a lobby had to be made in which the admin and other users can wait for the completion of the max players amount, so the game can be started.

3.3 As a user, I want to be able to sign up for invitations for a new game.

This epic story is a follow-up to the previously mentioned epic story and completes the process in our application for the sign up for a game and creating a new game. In this user story we had to create a page where all available invites were visible for users that want to join a game. Each invite had to have a couple of specifics so they could be clearer to a potential player. Those specifics are the name of the initiator, max players for the game, maximum time per turn, max game time and minutes before the game starts.

Each invite had to have its own button so the user can be redirected to the lobby specific for that game and wait for the game to be started.

3.4 As a player, I want to follow (a consistent subset of) variants of rules from the original boardgame so I can play the game.

This was one of the most important epic user stories and one of the biggest. For use to achieve this epic story, we had to split it up in multiple user stories.

The first acceptance criterion was to create a page on which the player can see all the rules related to the game, so he or she can have a better understanding of how the game should be played. Other criteria were the creation of dice. The player had to be able to roll dice and based on the rolled number, move a certain number of steps along the gameboard. When we created this ability another important aspect of the game had to be created. That aspect was that a player should not be able to always roll some dice, and that the action should be blocked when it is not the player's turn.

Furthermore, the creation of the entire gameboard had to be established and the ability to draw community and chest cards so the user can play the game as close to the original gameboard as possible. When a player drew a card a certain action that was displayed on the card had also to be made possible, such as paying or receiving money.

Those actions that change the state of the player, such as the amount of money that a player owns, should also be displayed for other players. This was also a major acceptance criterion so each player can have a clear view of how the game is unfolding.

At the end of a game the application should decide which player has won and redirect all players to a page where the username of the winner is shown, and the game action history of the current game.

3.5 As a user, I want to use an intuitive, dynamic user interface for the execution of my turn.

In this epic user story, we had to work further on some already created aspects of our game. One of them was the dice function. For this user story to be completed we had to make the dice more dynamic and more appealing to the player. A clearer way of showing which players turn it was to roll the dice and the amount of time that a player had to complete his or her turn. Were also created acceptance criteria for us to complete this epic story.

One other important acceptance criterion was the creation of side-views so players could track the current state of the game better. We split this criterion into multiple user story which consisted of the creation of a leaderboard (with the amount of money a player has), game settings tab and a board with the overview of all taken game actions.

3.6 As an administrator, I can deploy your platform on an online cloud service, so that it is accessible for the public internet.

This is one the crucial user stories that shaped our application and workflow.

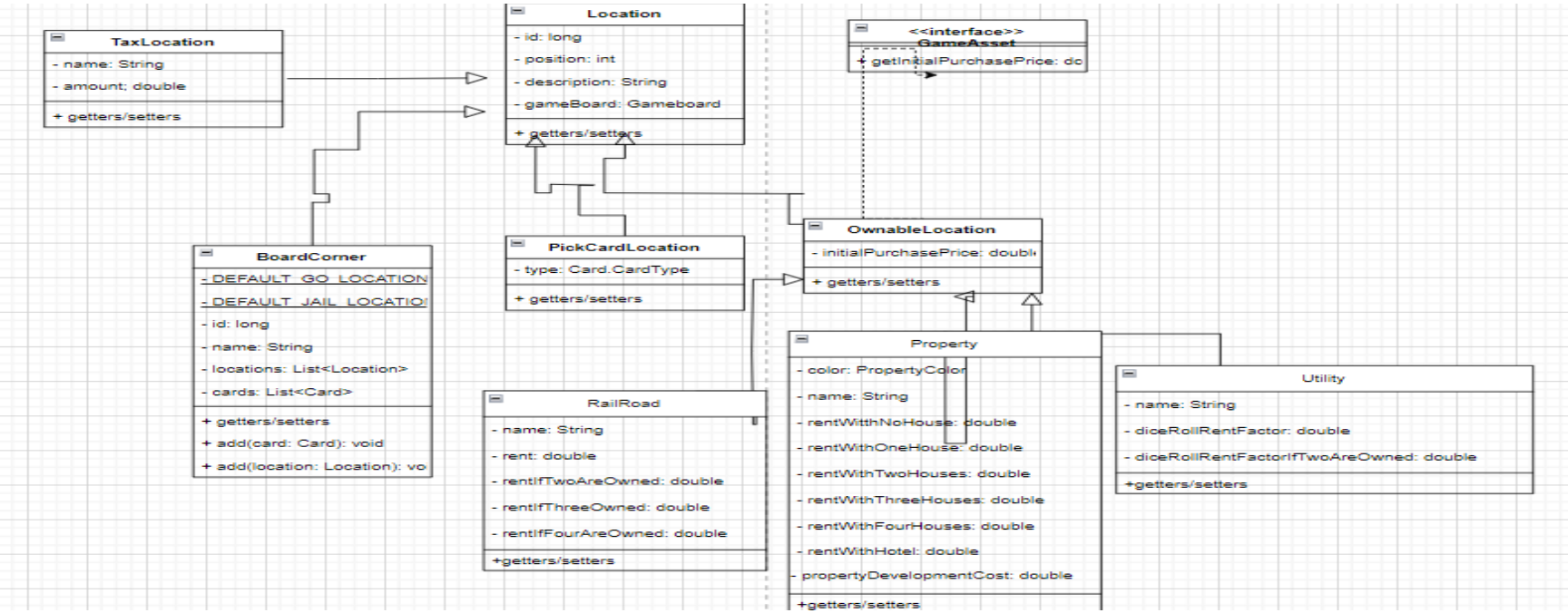
Firstly, a cloud environment needs to be setup, has a near 100% up-time, is cost competitive (free preferably), and has enough capacity for multiple concurrent users.

Secondly, the application needs to facilitate development and production environments. That only alter the state of the database in the production environment. And provide test accounts to aid development.

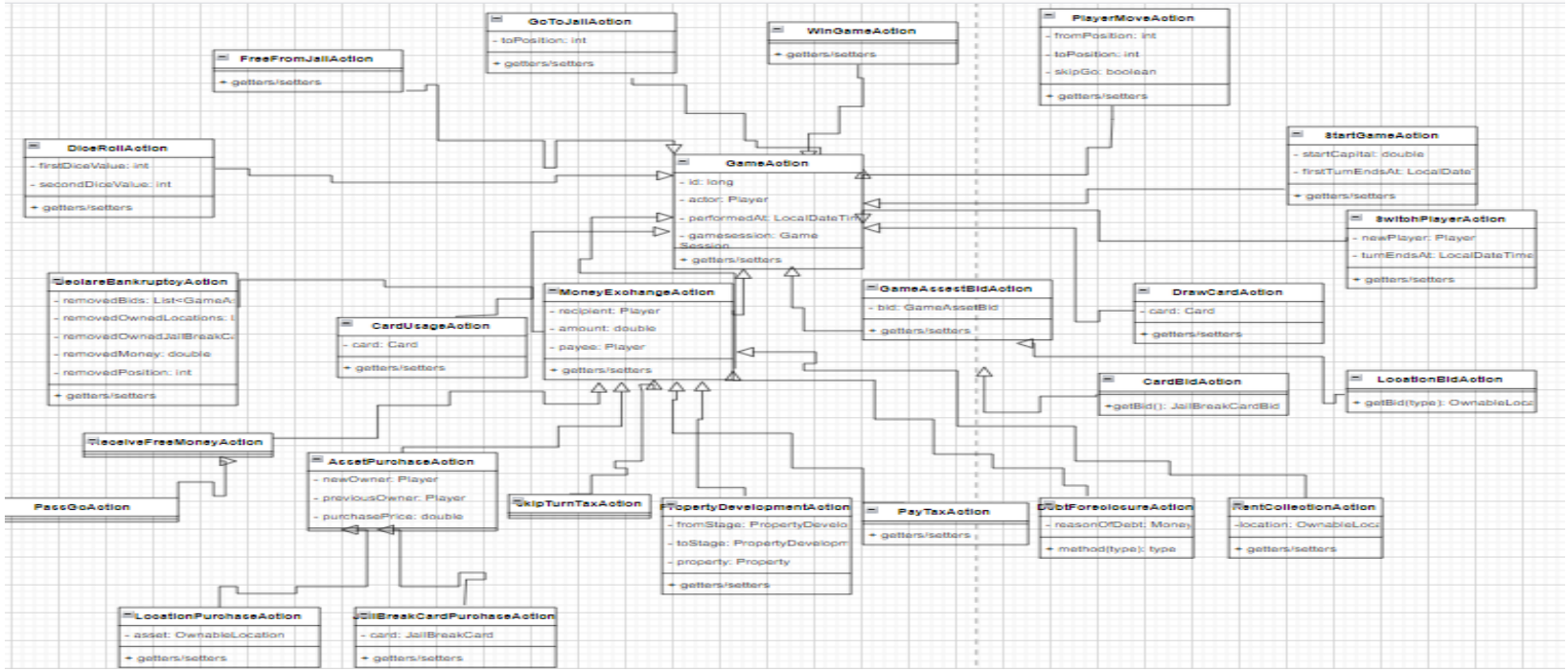
Lastly, the application needs to be automatically deployed to cloud, when a new version is pushed to GitLab version control, using its CI pipeline, without harming existing data.

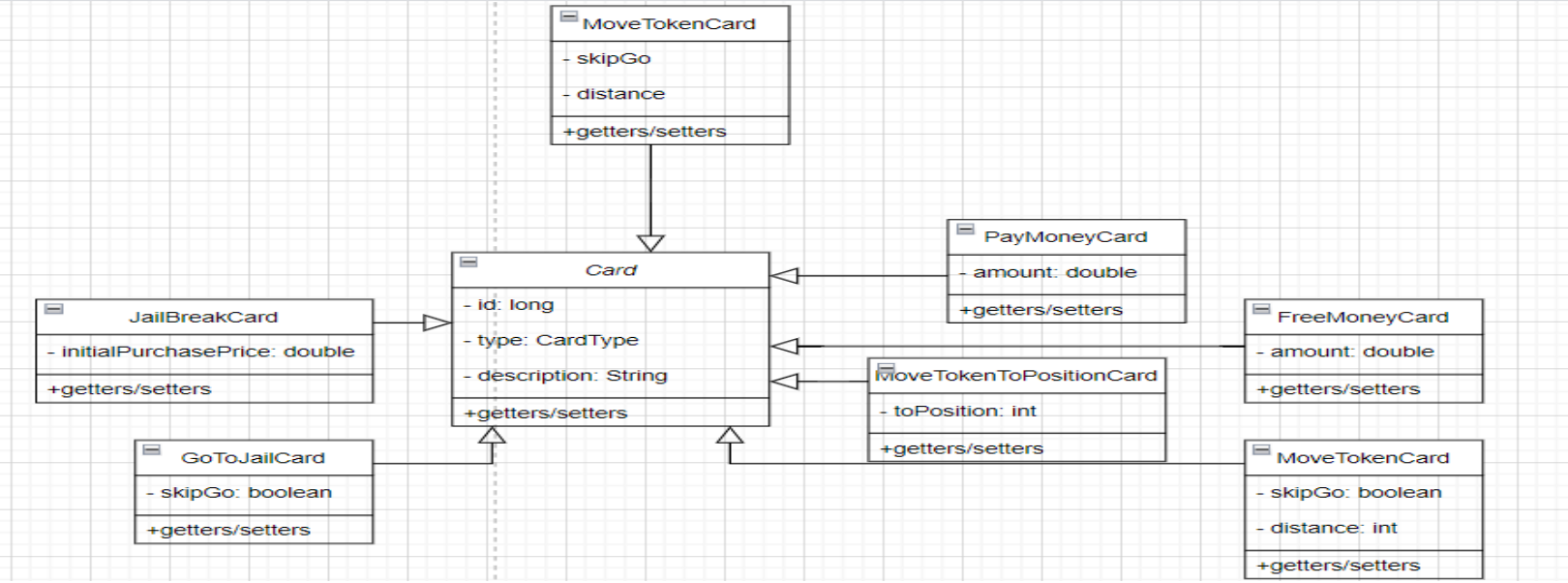
4 Navigable Class Diagram

In this part of our UML, we show the relation between the abstract Location class and all its extended classes. The Location class has 5 classes that extend the abstract Location class: BoardCorner, PickCardLocation, TaxLocation and OwnableLocation classes. The OwnableLocation class implements the GameAsset interface, so that it can use the getInitailPurchasePrice method. The OwnableLocation class is a class that is extended in the Property and Utility classes.

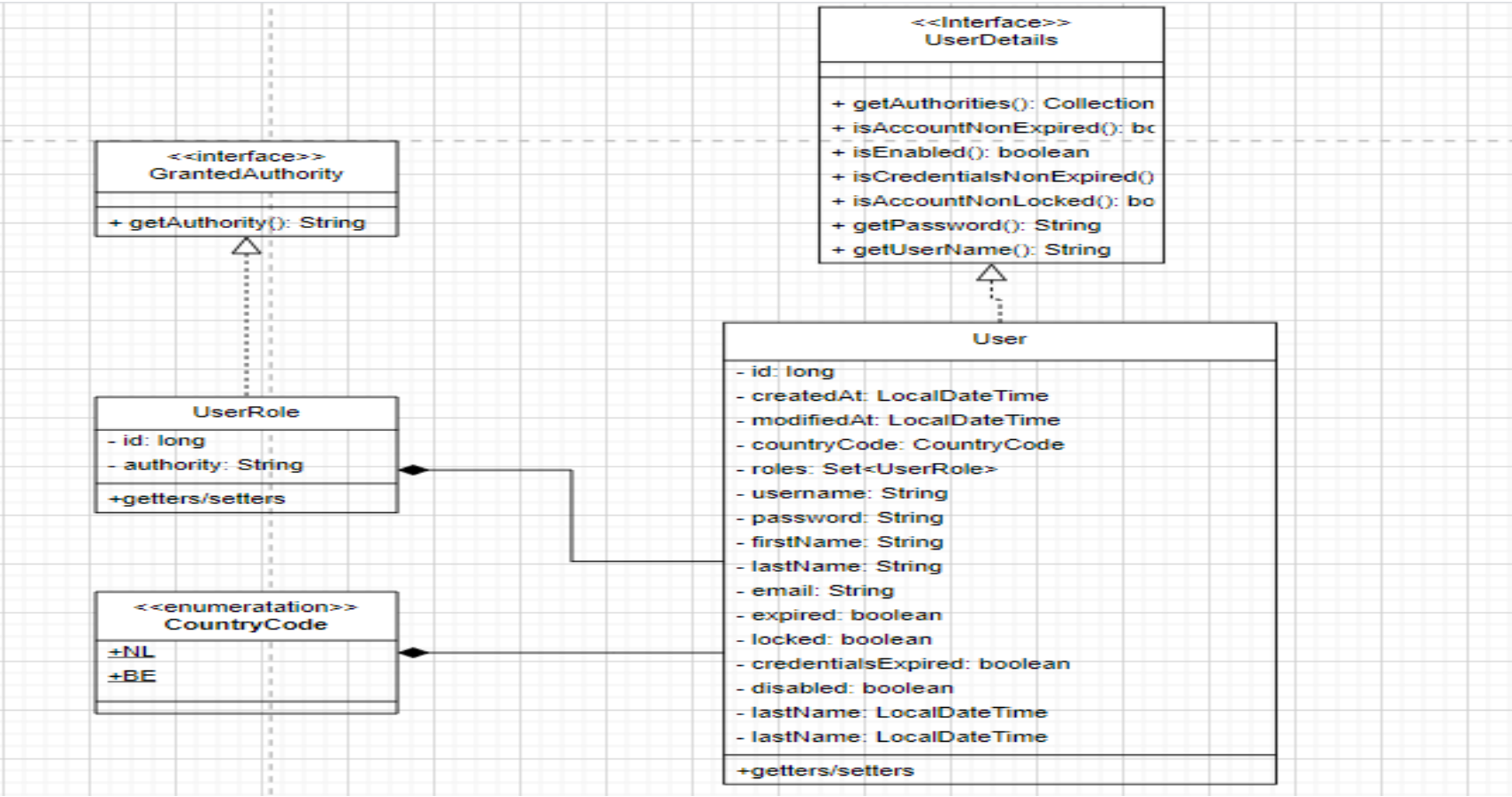


Here we see how we have structured the game actions in our application. The abstract GameAction class is the class that gives each class that extends it an id, actor, performed at and game session that is related to the game action. After that we have a couple of more specific actions, such as MoneyExchangeAction, GameAssetBidAction, DrawCardAction, SwitchPlayerAction, etc. After these classes have been implemented, we start going into more specific actions, such as diceRollAction, GotToJailAction and PlayerMoveAction, which extend the above-mentioned classes.





If you pick a card in the game this is how we structured the gamecard in our application. The abstract Card class gives each class that extends it an id, type and description that is related to the card. We have a couple more variants of cards for example PayMoneyCard and FreeMoneyCard. These classes are needed for specific card actions, because every action is deferent.



Here you will see classes related to User. User has one interface UserDetails where al the options located such as getPassword() or isEnabled(). Every user also has a UserRole to get track if the user is admin or not.

5 Challenges

We had numerous challenges during the development of the monopoly game. Namely: data transmission, security, maintenance of state. Which we will cover later on in greater depth.

5.1 Real-time data transmission

The biggest hurdle during development was real-time data transmission. Within our application this done in two manners, the first being with the HTTPS protocol (REST-fully) and lasty with WebSocket protocol.

The advantages and disadvantages will of course depend on the specific use case, but we'll try to point out some differences between WebSocket and HTTP.

WebSocket is more complex than HTTP. It has bigger handshake requirements (which include the use of the SHA1 hash function), you would then be unable to properly mask and frame the data to be sent, and the server would close the connection.

WebSocket connections are also intended to be more persistent than HTTP connections. They are useful for when you want to receive updates every second, because establishing an HTTPS connection takes a lot of time.

To establish an HTTPS connection, you first have to establish a TCP connection, then send a GET request with a pretty big header, then finally receive the server's response (along with another big header).

With an open WebSocket you simply receive the response (no request needed), and it comes with a much smaller header.

During the development we weighed the two costs of keeping a connection open vs establishing a new connection.

5.1.1 HTTPS

For the interaction with most resources between the front- and back-end application HTTPS is used, with REST constraints. In most places this was the logical choice, because the only practical benefit WebSocket offered was a full duplex connection.

Due to a lack of skill and a looming deadline, we chose to also implement the retrieval of GameAction objects from the back-end with HTTPS, with the front-end performing a request for new objects on a fixed interval of 10 seconds. This fixed interval consumes allot computational resources, because the back-end has to be queried for new information, instead of it providing it dynamically. In the future we would thus like to implement it over the WebSocket protocol.

5.1.2 WebSocket

Later on in sprint 4 we found enough time, and implemented a group chat using the WebSocket protocol for the retrieval of new chat history.

This is the right protocol, because it allows it has a much lower overhead, due to not continuously perform a new request.

5.2 Application security

Data transmission is the art of getting information to move between the server and client. This process also needs to be secure to ensure that sensitive data does not get in to the wrong hands.

In the back-end, we therefore chose to make use of the Spring Security Framework in combination with JSON Web Token (JWT) tokens and hashed user password.

In the front-end, we did not much as this only the presentation layer, apart from the obvious.

5.2.1 JWT

The next paragraph is according to auth0: JWT is an open standard that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. Our JWTs are signed using a HS512 algorithm with a private key to ensure the authenticity of the tokens.

These tokens provide a number of benefits namely:

- They are self-contained. The JWT can contain the user's details. So you don't need to query a database / auth server for that information on each request.
- They offer strong security guarantees. JWTs are digitally signed which safeguards them from being modified by the client or an attacker.
- JWTs are stored only on the client. You generate JWTs on the server and send them to the client. The client then submits the JWT with every request. This saves database space.
- They are efficient and quick to verify. This is because JWTs don't require a database lookup.

The lack of a database can also become a major problem, because you will not be able to revoke any generated tokens, or a malevolent person might steal the private key so he could generate as many tokens as he wants.

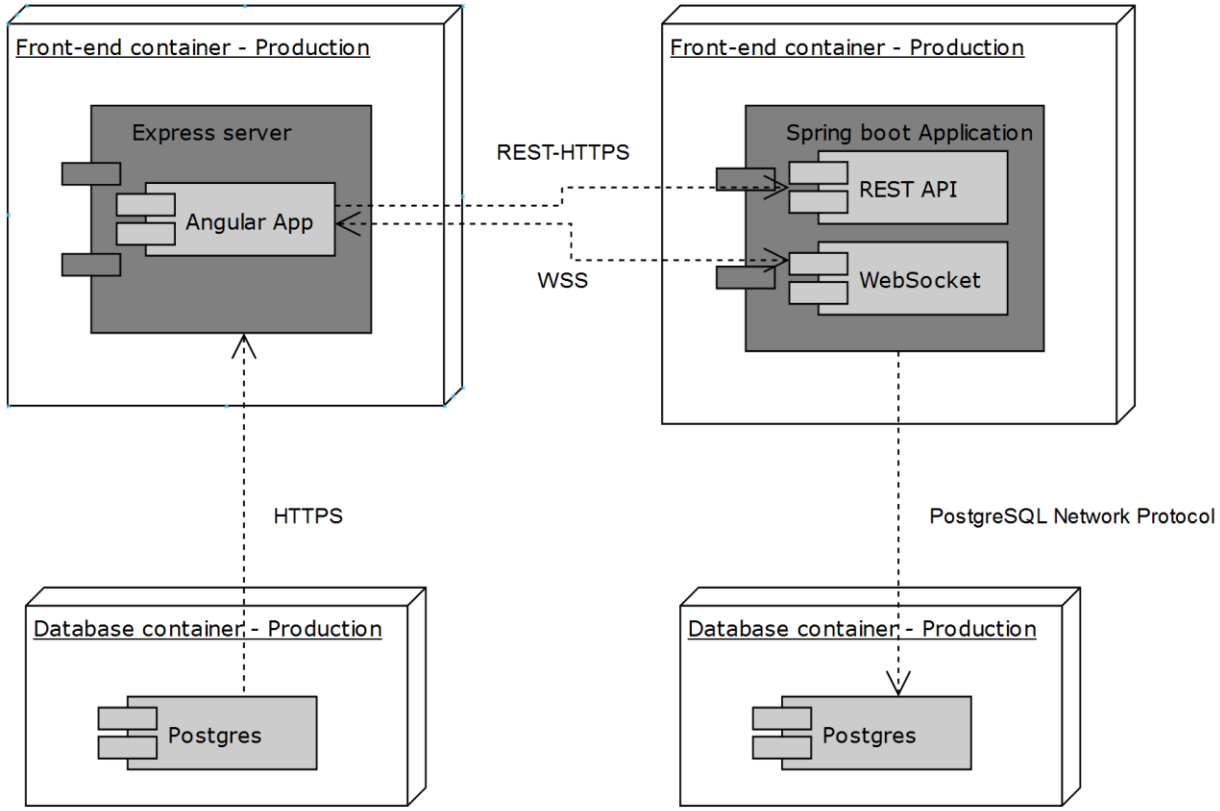
In the design of our application we mitigate these challenges by, limiting the duration of a token and storing the private key safely as a environment variable in our cloud environment, only accessible with two factor authentication.

5.2.2 Using JWT

JWTs are quite easy to understand, but implementing them is quite complex.

In the front-end we get a token by performing a get request and then accompany each request with one. Once the request reaches the back-end, the request will be funneled through a filter where the token will be validated, to see whether it is authentic and still valid. Depending on the result the request will be allowed to pass or denied with a clear error message.

6 Deployment Diagram



6.1 Considerations & rational

There a numerous hosting providers, with each providing their twist on the same concepts. For the deployment of our application, we chose Dyno's by Heroku. Which hosts applications, with no need to worry about server configuration.

There are many providers of similar services, notably: Microsoft, Google and Amazon. It is not really possible to choose the wrong one for a non-demanding simple application, that only needs to serve as a minimum viable product (MVP).

The reason we chose Heroku is quite simple, because it:

1) has support from Salesforce, giving us excellent documentation, community support and configuration;
2) is also recommended and supported by the teachers team of this course;
3) is a self-managed service which removes the need for any configuration;
4) scales well, due to a offering of many different server configurations;
5) the free tier covers a lot, and paid offerings are relatively affordable;

6.2 Actual deployment

We do not recommend deploying this application to any server environment. The reason for this is that our current application does not scale well, horizontally. To add capacity you will need to continuously replace your server with one, which has more ram, faster processing, more storage and higher throughput. This means that this application can only serve a small target audience, before becoming very costly, or not scalable at all leaving you with a high latency. For this reason we recommend the addition of load balancing technology and document based databases to the server stack.

Apart from these horizontal scaling issues, the application functions, and is separated into multiple containers, to ensure separation of concerns.

For the hosting of these separate containers, you cannot go wrong with any of the earlier mentioned solution, with Heroku being the easiest to setup, as it already configured and extensively tested by us.

7 Analytical reflection

The design of our application is split into two parts, a back-end module, and a front-end module.

7.1 Back-end

The back-end part of our design consists of a couple of packages, such as:

- API
- Config
- Domain
- Repository
- Services
- Utils

The API-package consists of our controllers which are split into two folders. A folder named `open` and a role-player folder. We decided on these two folders so that we can have a distinction between controllers that are meant for in game activities and controllers that are meant for activities such as authentication. Each controller has its own class in these folders.

The config package has a couple of two packages itself. It has an audit package, which is meant to keep track of the current authenticated user and lets JPA use the @LastModifiedD ate annotation. The security package inside the config package takes care of the security side of our application. It creates JWT-tokens and checks for unauthorized requests and sets permission endpoints. The last class in this package is the GlobalExceptionHandler. This class acts as a blueprint for the exceptions we call upon in our back-end services.

The domain package holds all our models and DTO'S. The model package starts abstract by specifying two packages, game, and user. The game package is divided into three parts which a game consists of. The parts are action, board, and card. All game actions are then further divided into actions that are possible in a game, such as: card, gameplay, and money actions. The further we go into these packages the more specific each element gets. After we open one of the previously named packages, we only get classes that are specific to their package and have clear implementation.

One of the most important classes in this package is the GameSession class. This class is an aggregate of a lot of critical and important classes we already mentioned. This class represents the game the user is going to create, which consists of the players that are joining, the actions for this game session, when it is created, what the state of the game is, etc. Because of the Jackson configuration this class has, we can send data via JSON using one class, instead of having to create multiple end-points for each game-action.

The classes in the DTO package help us to send clearer and concise data to the back end. An example of this is the UpdateGameRequest class. This class helps us by updating a game session by not needing to provide a full game session, which consists of a lot of attributes. Instead of that,

the UpdateGameRequest only lets the user send two parameters which a user can update instead of the whole game session.

The repository package helps us connect with the JPA-repository and lets us make changes to entities. This package consists of a lot of interfaces which hold custom made queries that are useful to certain methodes in our back-end module.

Each interface belongs to a service class, so we do not have to call upon the entity manager directly in our back-end service classes.

The service package consists of packages that manipulate the data that is received from the repository and save it accordingly. The service package is split into the same way as the repository package so that there can be clear communication between the service and the JPA repository.

The Utils package is created to create dummy data we could use on startup and JWTUtil class. This class is used to perform operations related to JWT-tokens.

7.2 Front-end

The design for the front-end relies for a large part on the data that is received via the gameActionService. This service sends and receives game actions to the back-end module of our application. The crucial part here is that the gameActionService looks for new actions every ten seconds and gives the ability to update the front-end based on the new data that is received.

The pages of the front-end are found in the components package. Each page has its own component and receives data from the inject service that is relatable to the component.

7.3 Areas of further improvement

One area of further improvement could be found in improving the ability to send and receive data from and to our back-end module at a more acceptable time.

Our application now runs on rest end points which have more overhead, and lacks instantaneous response from the server. The data is updated every ten seconds with the help of the GameAction service. Because of this the user does not immediately see the necessary changes that are made. If we implemented web sockets for at least some of the actions in our game such as: fine payment and rent payment, the game would be much more responsive and clearer for the user. The user would immediately see for example that an amount of his current balance is taken and does not have to wait up to ten seconds to see this made change in his or her data.