2DV609

Design Document

WinnerDrinks

Version 2

2021-05-30

Delfi Šehidić
Joel Martelleur
Lucas Sjölander
Susanna Persson
Caesar Lennartsson

Contents

1 Introduction

1.1 Purpose

The main purpose of this document is to describe and document the architecture of the application WinnerDrinks. It is mostly written for the software developers who work with developing and maintaining the application but can also be read by other stakeholders to get a deeper understanding of the application's architecture.

The document first describes the main goals and philosophies of the application's architecture, see section *2 Architectural goals and philosophy*, the assumptions and dependencies that have been made in the architectural decisions, see section *3 Assumptions and dependencies,* as well as the requirements that has had the greatest impact on the architecture, see section *4 Architecturally significant requirements*.

In section *5 Decisions, constraints, and justifications* you will find a descriptive list of the architectural decisions and constraints that have been made and why those decisions and constraints have been made.

In section *6 Architectural Mechanisms* you will find descriptions of architectural mechanisms and tactics that have been decided to use in the architecture to solve identified standard problems.

The last three sections focus on describing the architecture. In section *7 Key abstractions* you can read about the key abstractions in the architecture, in section *8 Layers or architectural framework* you can read about the architectural patterns used in the architecture and in the last section *9 Architectural views* you find different views describing the architecture.

1.2 Terminology and abbreviations

MERN: Stack of development technologies for building a full stack web application, containing technologies:

1. MongoDB: General purpose, document-based, distributed database.
2. Express.js: Framework for building a web server.
3. React.js: A JavaScript library for building user interfaces.
4. Node.js: Javascript server platform.

Progressive web application (PWA): A type of web applications built and enhanced with modern APIs to deliver enhanced capabilities, reliability, and installability while reaching anyone, anywhere, on any device with a single codebase. The goal for PWA is to have the power as a native app and be as portable as a web application.

Single Page Application (SPA): A type of web application built on only one html page that the user interacts with. Instead of making new http requests and loading entire new html pages, the user

interactions will dynamically rewrite the current web page. The goal with this is to make the web application feel and perform more like a native application.

FAT client: A client in a client-server architecture that implements a big part of the application logic independent of the server. A “fat” client can also be called a “heavy”, “rich” or “thick” client.

REST API: A type of application interface that conforms to the constraints of REST architectural style. REST uses http as stateless communication to get, create, update, delete resources such as html, json, plain text etc.

2 Architectural goals and philosophy

The philosophy behind the architecture is to create an architecture that can balance the users' needs and the client quality attribute demands but at the same time is maintainable, deployable and extendible. The architecture should also be a support and guide for the current developer in the project but also for future developers that may be working on maintenance, improvements and extensions of the application.

User needs:

1. An application that fulfills the performance requirements, see requirement WD.NF.5 in the requirement specification document.
2. An application that is secure to use.
3. An application that is fully functional and playable offline and online.

Client quality attribute demands:

1. An application that implements the requirements, domain rules and use cases that is specified in the Requirement Specification document.
2. An application that is consistent with the user interface described in section 3.1 User Interface in the Requirement Specification document.

Goals of the architecture

1. A well structured modular architecture that focuses on extendability, maintainability and deployability.
2. A well documented architecture that supports the implementation of the application for current and future developers of the application.
3. An architecture that supports offline and online user needs and client demands.
4. An architecture that balances user needs, client demands and developer demands.

3 Assumptions and dependencies

The assumptions that we've made have driven and affected our decisions. They have been made due to our skills, earlier experiences, and knowledge within the different techniques and technologies, such as our decision of programming language and frameworks. The management of the project have made some assumptions concerning the developers due to the developers, from the beginning, said that they had a lack of knowledge within some of the technologies that will be used.

3.1 Team member knowledge and availability

The client and the management makes the assumption that the developers have the knowledge within the different technologies that they are using for the best fitting approach. The team of developers per se assume that each developer in the team has knowledge in the technologies, such as the decided development stack, MERN, as well as the application type, a Progressive Web Application (PWA) as a Single Page Application (SPA), so they all can contribute to the development. The team, when developing the software, does also make the assumption that each developer will be available during the development-phase and notify the others if they are not available to contribute or get stuck during the development.

3.3 Microservices availability

The project-management assumes that the microservices within the software will have high availability and high reliability to deliver a well-valued software to the client. The management assumes that the software and microservices will be able to scale, auto-start, re-start when needed to offer good availability of the software which as an effect will increase the reliability of the project. This may affect how the developers decide to implement and design the software to make sure that it is possible to scale the application.

3.4 Trivia API availability

The developers assume that they can use the external Trivia API https://opentdb.com/api_config.php to create the game events for the game module Trivia described in section 2.5 in the requirement specification document.

3.5 Mainly users have smart-phones (for application download)

The client of the project assumes that their customers and the users of the application are using a newer phone, specifically a smartphone. The reason for this is that they want to offer the ability to download the application. The developers will make this assumption as well and design and develop the software accordingly to this.

3.6 Application license

The architecture is based and dependent on technologies such as React which is under the open source license MIT, this means that this application will also be under the MIT license. It is assumed that the developers and client are aware of this choice and that the application will support this license.

4 Architecturally significant requirements

4.1 Game modules and Game events reliability and usability

Requirements: WD.NF.1, WD.NF.3, WD.NF.4

Rationale: The game modules and the game events that are a part of the game modules are a large part of the application and the fulfillment of the user needs of the application. The architecture must support and focus on these requirements.

4.2 Game modules extensibility

Requirement: WD.NF.2

Rationale: The architecture must support development and maintenance of the game modules independently of each other. Also this requirement implies that the application should be easy to deploy to have a fast return of investment of newly updated or implemented game modules.

4.3 Winner Drinks performance

Requirement: WD.NF.5

Rationale: The response time for user interactions is a key to make the application enjoyable for the user and the architecture must support the performance goals that have been agreed upon in every interaction between the application and the user interacting with the application.

4.4 Winner Drinks portability

Requirements: WD.NF6

Rationale: Since the application must work on different mobile devices, different reading tablets and on different web browsers and in different conditions such as online, online without client-side storage, and offline. The architecture must support these different environments and conditions.

5 Decisions, constraints, and justifications

The following are the decisions, justifications and constraints that have been made. Together they form the proposition of what the architecture needs to support. For a more concrete and visual oversight chapter 9 contains figures that represent the proposed architecture.

5.1 Decisions

5.1.1 Progressive web application

Since the client demands that the application be fully functional and playable on targeted lists of mobile phones, reading tablets and web browsers, it has been decided to create a Progressive Web Application (PWA) (for details on the exact mobile and reading tablet operating systems and browsers, see the document *Requirement Specifications section 2.2.14: WinnerDrinks portability*).

Rationale: The choice was between building separate native applications for different platforms and a web application that can be used in different environments, or creating a PWA that can be used as a cross platform native application and a regular web application. Since time is short and the application does not require features like camera, contact list, accelerometer, etc that only native applications can use on mobile phones and reading tablets, the choice has been to build a progressive web application. This choice will also support developers' quality attribute demands that the application must be maintainable and deployable, because building a PWA requires a smaller code base and a single location to deploy the application. The finished deliverable is a coherent unit which makes it have a high maintainability and deployability due to being a single application instead of several, such as would have been needed in the case of separate native applications.

5.1.2 “Fat client” and client/server architecture

The main architecture shall be based on a client/server architecture with a “fat client”. A “fat client” approach means that the client in a client/server architecture provides a rich functionality that may be mostly or almost entirely independent of the server. This allows for the development of a rich interaction experience for the end user of the application.

Rationale: Since it has been decided to build a progressive web application, the architecture of the application will be based on a client/server architecture with a fat client where the responsibility of application logic will be put on the client and the server only has the responsibility to serve the client static files and the data needed by the client. This also conforms to the requirement that the application should be designed to allow use even if the end user’s device is offline, by serving an amount of data used by the application later immediately as the application is started, and the subsequent handling of that data is done client-side, until more data is requested by the client (see the document *Requirement Specifications,* section *2.3.5: Load game data*).
A more visually palpable description of the minimal responsibility of the server compared to the client is shown in figure 1, for a more holistic perspective of the whole application see figure 5 in chapter 9.2.
The architecture that will be used on the client side is the three tier architecture. This will provide a foundation for how the application is going to be structured. An example of this is provided in figure 2.

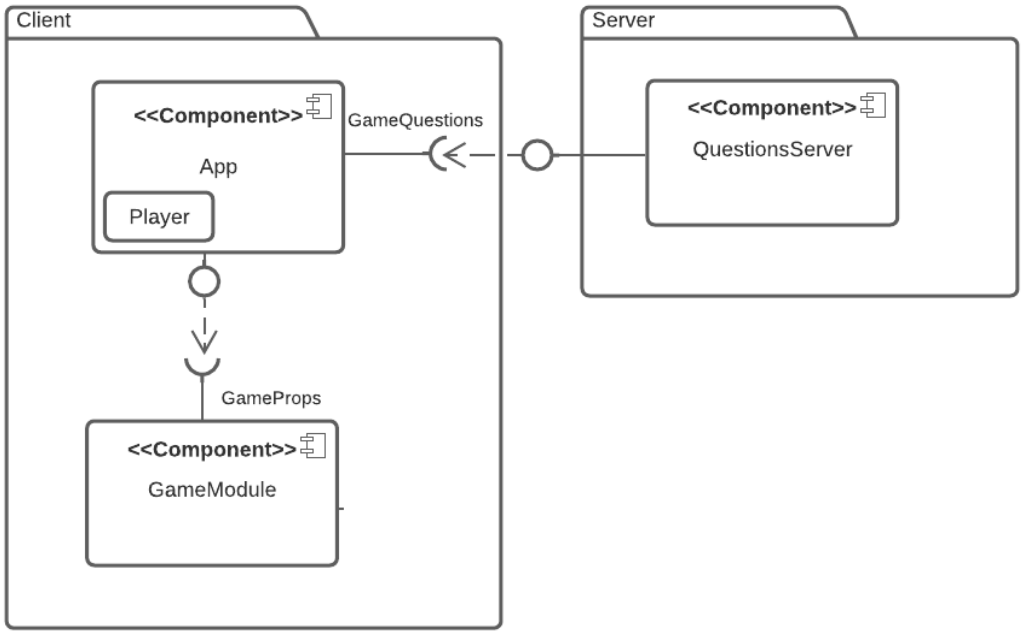


Figure 1: Component diagram of the client and server where the server's only responsibility is providing the questions.

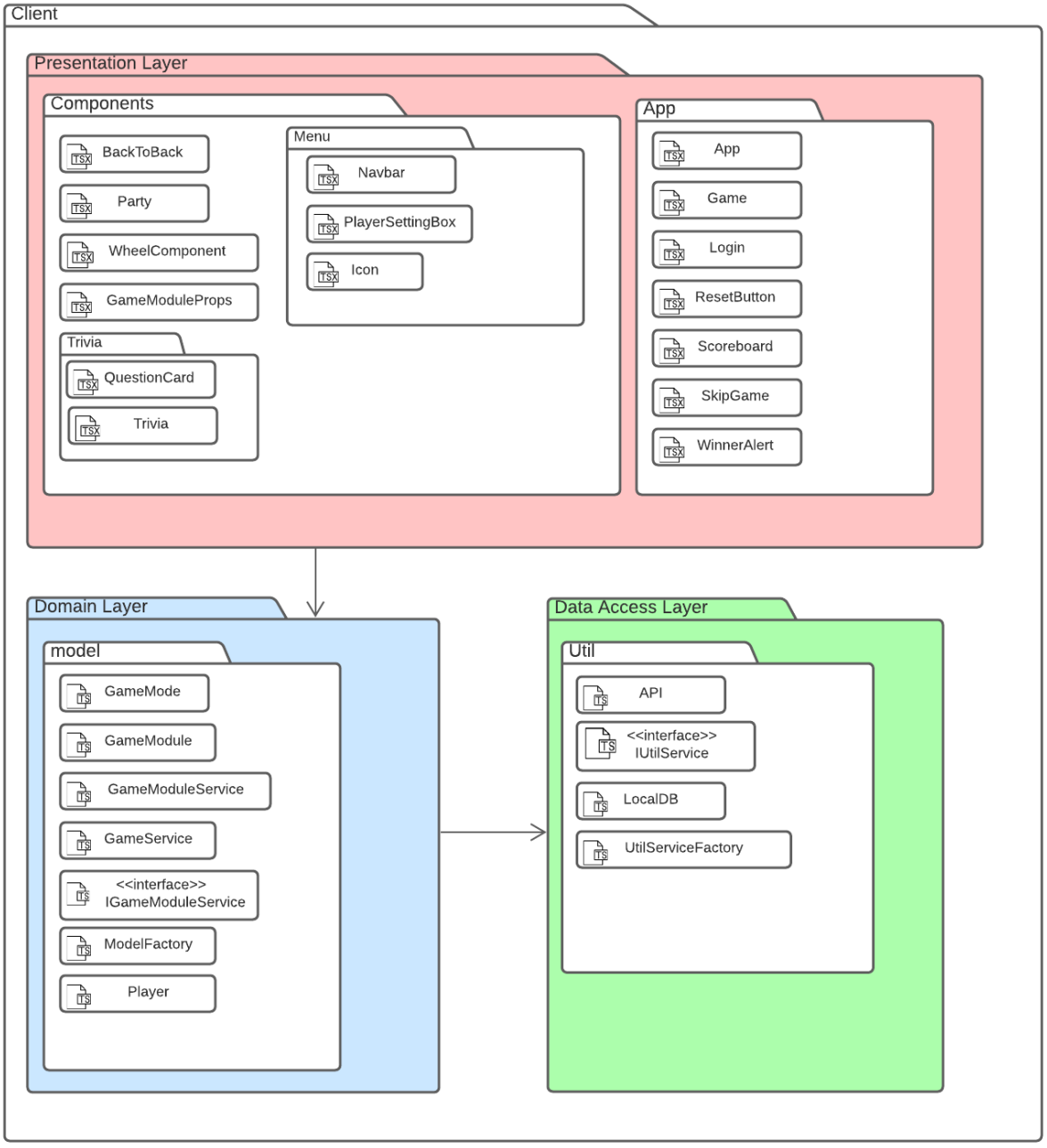


Figure 2: Package diagram of the client that follows the 3-tier architecture

5.1.3 Single page application

Since the client demands that good performance is an important issue (see section *4.3: WinnerDrinks performance*), it has been decided to create a single page application (SPA), that makes one HTTP GET-request to the server and responds with the content. After that, each interaction will dynamically

re-render the page and replace the content with the content desired rather than making new HTTP requests for each interaction.

Rationale: The choice was between a single page or a multi-page progressive web application architecture. Because single page web applications usually have better performance when it comes to speed, due to not needing to render new pages that may need new data while the application is in use by the user, it has been decided to create a single page web architecture. Another factor is the caching capabilities. Single-page applications have an advantage over multi-page due to how the entire application is loaded in the browser at the start, and thus no or very little new data (except for communication of data from the server) needs to be accessed while the application is in use. The disadvantage is a possibly slightly longer load time at startup due to a larger volume of Javascript. However, as this feature of single page applications also conforms to the demand that the application be usable offline or in bad connection situations (due to one single data transfer instead of several), the single page architecture is a superior design for this project.

5.1.4 Development stack

MERN is the chosen development stack used in the development of the application. MERN includes MongoDB, Express, React, and Node.

Rationale: Developing a web application requires the use of several compatible technologies. When choosing these techniques, it is wise to follow a stack of technologies that have been tested in production and that have proven to be a good solution. When choosing the stack, we have discussed software technologies knowledge and previous experience with all the developers involved in the team that will work with implementing the application and also looked for a stack that is a good solution for building a single page progressive web application. The choice of stack eventually landed in the MERN stack, a stack that uses the technologies MongoDB as database, Express.js as web server framework, React.js as a client side framework and Node.js as Javascript platform on the server.

5.1.5 Typescript

Typescript is the chosen programming language in the application.

Rationale: Some of the advantages that made us use Typescript over Javascript are that Typescript displays the compilation error when developing which makes the software more robust and decreases the number of errors that could occur at runtime. Typescript also supports better Object Oriented Programming (OOP), with support for interfaces, which can increase the reusability and flexibility of the software.

5.1.6 Server Database

To store persistent data on the server it has been decided to use MongoDB.

Rationale: The choice was between two different types of databases, relational (SQL) and non-relational (noSQL). We chose the noSQL-database system because of its simplicity and because it fits the modularity and extensibility of the software architecture; noSQL is often easier to update and change. Because of the development team's knowledge and previous experience of MongoDB, this specific noSQL Database Management System (DBMS) is the chosen DBMS for this project. This also affected the choice of development stack (see above).

5.1.7 Client side storage and offline compatibility

In order for the application to be fully functional and playable in offline mode, we have decided to use three APIs that most modern web browsers implement.

1. IndexedDB API as client side storage
2. Service Worker API acting as a type of proxy server.
3. Cache interface to store the network's requests.

Rationale: Because offline mode and performance are two important requirements in the application the architecture will include the use of service workers to enhance the performance of network requests in the application and to store persistent data in the web browser. Because local storage is not compatible with service workers, IndexedDB will be used as the persistent storage in the web browser.

5.2 General development constraints

Constraints that apply when developing all parts of the project.

Programming language: Typescript
The software must be developed using Typescript. Typescript should benefit the development process in multiple ways. E.g. by enabling the use of interfaces which can be used to define a template in how the different components should be developed to work together.

Code quality: Eslint
The development process shall include ESLint for code analysis and to define a code standard.
The default ESLint configurations for each framework should be used, with further configuration if necessary. The use of ESLint in combination with Typescript should provide basic rules for how the code is supposed to be written. Allowing for more similar code between developers developing separate components.

Version Control System: git
Git will be used in conjunction with Github to support version control.

Text Editor: Visual Studio Code
Visual Studio Code will be used as the IDE.

Package manager: Npm

Frameworks and libraries:
The following frameworks will be used.

- React.js - Front-end framework
- Express.js - Back-end framework
- ESLint - Linting tool.
- Typescript - Programming language
- Jest - Testing framework
- Bulma - CSS framework

5.2.1 Client development constraints

Constraints that apply when developing the client side of the application.

Client-side framework: The application frontend shall be developed using the framework React.js, specifically with functional components (hooks).

Client-side runtime environment: During the development of the client, node.js version 14 will be used as the runtime environment.

Todo list
Based on what is considered the best practices regarding React development the following is recommended.

- ➔ ***DO*** *make use of functional components while developing the application.*
- ➔ ***DON'T*** *make use of class components while developing the application.*

Rationale: There are multiple arguments regarding why to use functional components instead of classes, primarily the reasons to use functional components is that the code is easier to read and understand while also ending up with less code. A functional component is considered to be easier to test and debug than its counterpart while also being more reusable.

5.2.2 Server development constraints

Constraints that apply when developing the server side of the application.

Server-side framework: The application backend shall be developed using the framework Express.js.

Server runtime environment: Node.js version 14 will be used to execute the application during the development process and also to host the web-server serving the final application.

Todo list

- ➔ ***DO*** *check the current version of Node.js and update if necessary while doing development work on the application.*
- ➔ ***DON'T*** *disregard checking of Node.js version while doing development work on the application.*

Rationale: It is essential for the development process and the stability of the application that a unified environment is used by all developers.

5.3 Application constraints

The application should be developed as a Single Page Application (SPA) and a progressive web application (PWA).

GUI: The application shall be usable by a list of mobile devices as specified in *Requirement Specifications section 2.2.14: WinnerDrinks portability.*

Todo list

- ➔ ***DO*** *adapt the application's UI according to the mobile-first mindset.*
- ➔ ***DON'T*** *adapt the applications' UI for desktop circumstances mainly or only.*

Rationale: The requirements call for a mobile-adapted application and the decision to make a PWA (Progressive Web Application) allows the developer to test the application in a desktop environment while doing development work. It is therefore imperative that developers consider the different constraints (screen size, etc) that apply to this application compared to their development environment.

5.4 Design constraints

UX prototyping: Quant-UX will be used to design prototypes for the user interface.

UML diagrams: Lucidchart.com, draw.io and websequencediagrams.com will be used for UML diagrams.

5.6 Deployment constraints

The deployment of the application will be tested on the private cloud CSCloud that is built on the cloud platform OpenStack.

Available resources:

- OS image: Ubuntu 20.04.
- Number of virtual machines: 3
- Number of CPU:s: 10
- RAM: 8 GB
- Volumes: 10 units with a total capacity of 1020 GB
- Public IP addresses: 3
- Number of OpenStack Security Groups (IP filter rules): 10

6 Architectural Mechanisms

6.1 Portability tactic

State: Analysis

Description: To allow for the application to be used across multiple devices, offline and online the application should be a PWA (Progressive web app), using the required technologies and APIs.

6.2 Reliability/Usability tactic

State: Analysis

Description: The different game modules should have similarities.
Two different game modules' order of execution can be abstracted down to something similar.
First the selection of players, second the game event which represents the “game part” of the active game module, and lastly a result of the game event.
This abstraction or a further abstraction, is something that could be part of all the game modules to allow for different game modules to resemble each other better.

Attributes:

- Load times: Compare the load times between different steps of different game modules.
- Reliability: Common error handling for the different steps of different game modules.

6.3 Performance tactic

State: Analysis

Description: The architecture should support achieving the required performance, for example if the latency is too high to make a request to the server it can potentially use the offline server for that request.

Attributes:

- Latency: The duration of the network request to complete.

6.4 Deployment tactic

State: Analysis

Description: To make the application easier to deploy and easier to scale, the various components in the system will be containerized. This includes the application and the MongoDB database.

7 Key abstractions

The following list contains and describes the key abstractions of the system. They are each generically described as self-standing, not dependent on each other but only on itself, with a few exceptions.

7.1 Server

The system consists of a server accepting HTTP-requests. The server will however only accept and respond with a valid success status code when a correct request method to a correct server endpoint is made. The server is per se self standing and does not depend on any other part of the system except for the database, it is for itself an own service that is used by the web server (client). The server will run on its own virtual machine in the cloud and will not be affected by any other machine within the network (except the load balancer).

7.2 User Interface

A User Interface in the system will contain each component that the user interacts with. The user interface will display output to the users and take input when given by the user. The user interface will be adjusted to work both on big screens such as desktops, as well as smaller screens such as mobile screens.

7.3 Game Modules

The system contains game modules, specifically three of them. Each game module has its own responsibility and features but works pretty much the same with input and output and some manipulation of data. They are each important and the software wouldn't exist without them due to they play a major role offering the features that the software is built upon.

7.4 Game Events

Each game module is responsible for the presentation of a game event. A game event contains necessary data passed to the game modules.

7.5 Database

The system does also consist of a database responsible for storing the data. The database does, just as the server, run on its own machine in the cloud. The rest of the system does not know anything about the database or how the data is structured, but just has to query the data desired.

7.6 Player

The system will consist of players with attributes such as their names as well as the number of points that they score during the game. Each player will have the same attributes as well as the methods needed for the class used within the system. The players will be stored and used by other parts of the system as well as other parts that are dependent on the players.

8 Layers or architectural framework

The entire system will be built with a client-server architecture. It will consist of several subsystems, each implemented using an appropriate architectural framework fit for the specific task. A web server will serve the client side code on the internet over HTTPS. The server side application and the client application constitute the model-view-controller pattern (MVC).

The client application is served from the main web server. It is a fat client that contains most of the business logic. It is thus slightly modified from a traditional MVC, since it does not only take responsibility for displaying data. The client queries the web server for all data it requires. The client will also have responsibility for caching some data, to enable offline usage. The client is built using functional components, according to best practices in React development. A component, which can be made up of other components, is responsible for its own state. State shared between two components is lifted up to a higher component.

The server side architecture constitutes the model and controller. It contains a web server and a database. The web server communicates with the database and API as RESTful services. When the client queries the web server for data, the web server queries the database data and sends it back to the client.

9 Architectural views

An architectural view is a way of describing the entire system from the perspective of a certain stakeholder. A view can be explained with e.g. relevant UML diagrams. The views that will be utilized are outlined and described below.

9.1 Logical view

The logical view contains artifacts relevant to the software developers in the team. This includes UML class diagrams. Each box represents either a React functional component or a class.

Table: Explanation of the client side and figure 3

Components	**Description**
App	The main component of the application.
Game Module	The interface for game modules (minigames).
GameEvent	A game module's event (question, trivia, etc)
Settings	The settings menu.
StartPage	Landing page for adding players.
Player	A class for each player with name, score and status.
Game	Game play component, responsible for keeping track of players, current module, etc.

Figure 3: Logical view of the client

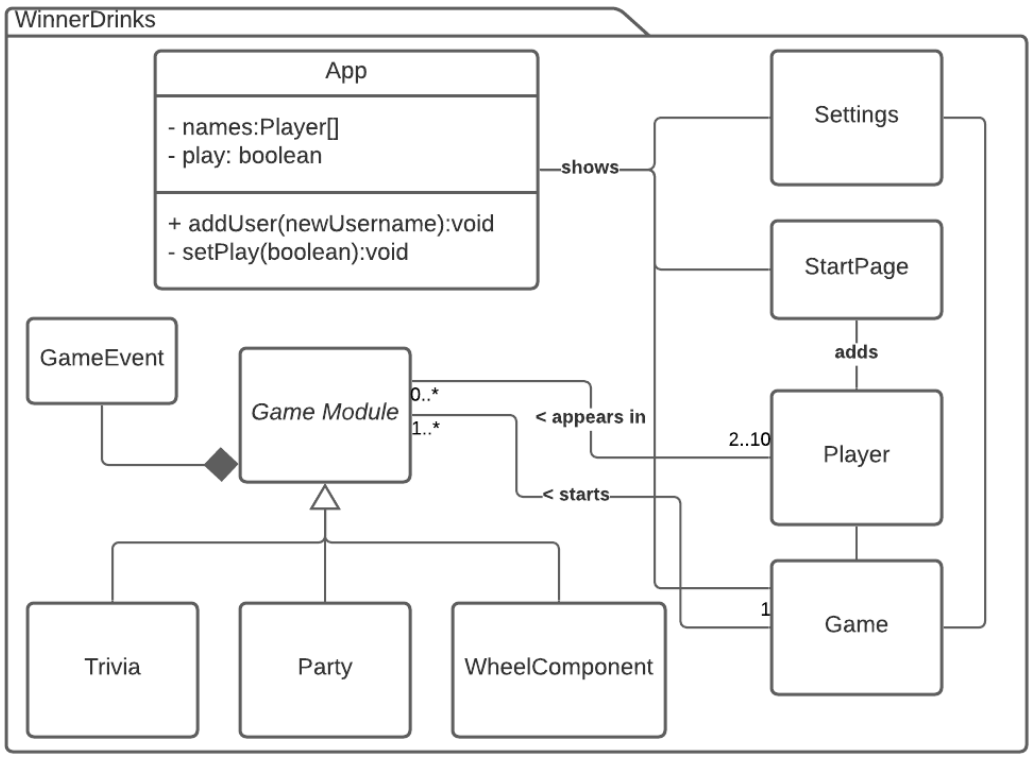
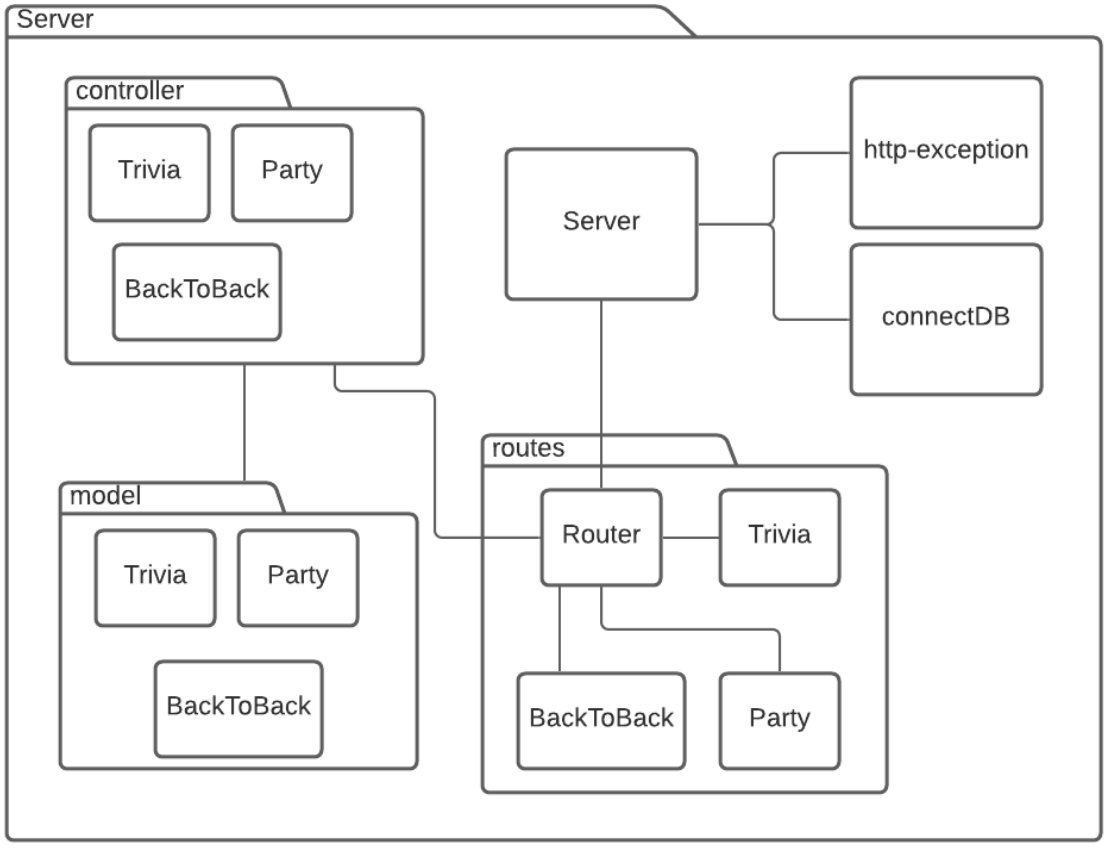


Table: Explanation of the server side and figure 4

Components and packages	**Description**
Server	The starting point of the application.
controller	Controllers between routes and models.
model	Data access and definitions for mongoDB collections.
routes	Server endpoints for specific game modules.
http-exception	Various error handling utilities.
connectDB	Functions for connecting the server to the database.

Figure 4: Logical view of the server



9.2 Physical view

The physical view is concerned with the deployment of the application and thus includes an UML deployment diagram. As we can see in figure *Figure 3*, the deployment of the application consists of three machines, one for the load balancer/reverse proxy, one for the web server and one for the database. There is also an external server that is not managed by us. Each service within our system is communicating through rest within our own network. Each machine does only have the minimum required port open as a best practice and best security.

Table: Explanation of the deployment diagram, figure 5

Nodes / devices	**Description**
Web Browser	Requests the application from the server and runs it.
Cache	For installation of the application with IndexDB.
Load Balancer	NGINX used as a reverse proxy/load balancer. Forwards requests.
Web Server	The main application.
Database	A database running MongoDB that contains application data.

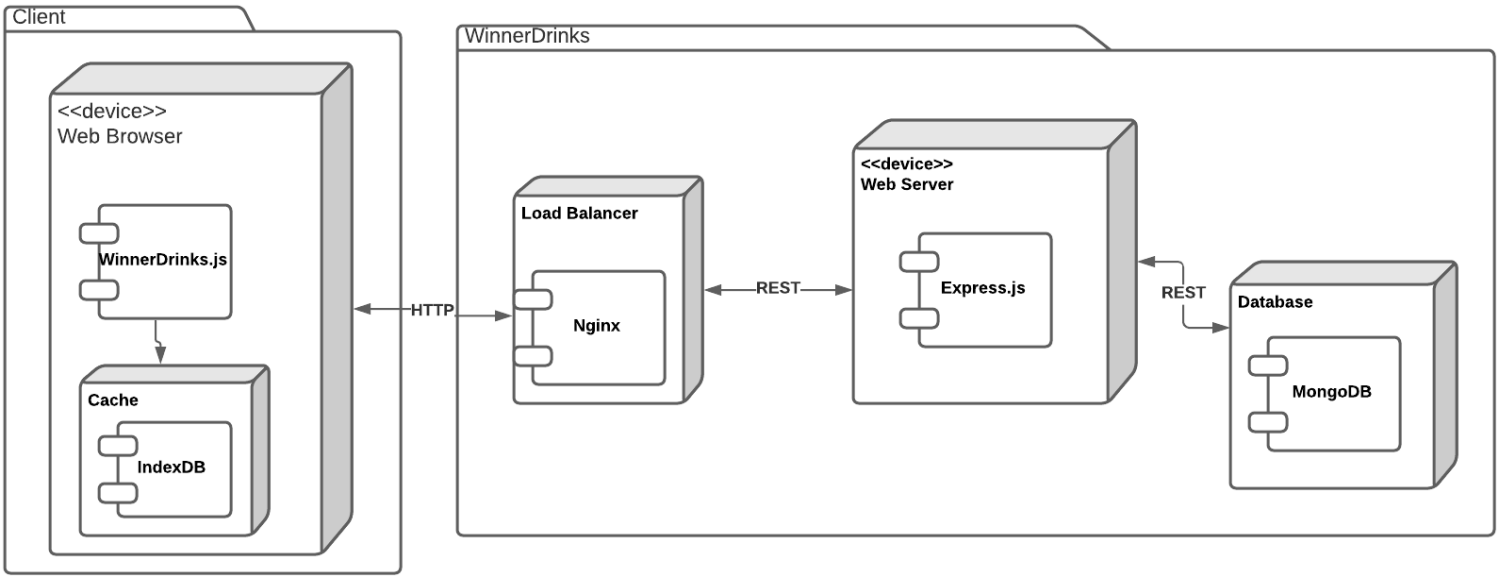


Figure 5: Physical view of WinnerDrinks

9.3 Use case view

The use case views are based on the use cases that contain architecturally significant requirements, they are constructed according to a common template.

Use case view template

1. Title
2. Description of the view, what use case it is based on, and why it is important.
3. Table of the actors.
4. Table of the included architecturally significant requirements.
5. System sequence diagram based on the use case.

9.3.1 Start the game

This use case view is based on the use case *Start the game* and describes a scenario when the user starts the application and the game. It is important for the performance of the application and contains domain rules for game participants.

Table 1: Actors, use case view "Start the game"

Actors	**Actors description**
User	A person that interacts with the system.
WinnerDrinks	The system
MongoDB	Remote database used when the system is online
IndexedDB	Local database used when the system is offline

Table 2: Architecturally significant requirements, use case view "Start the Game"

Requirement ID	**Requirement Name**
WD.UI.1	Enter game participants
WD.UI.2	User error message- Name input error message
WD.NF.3	WinnerDrinks performance
WD.NF.6	WinnerDrinks portability

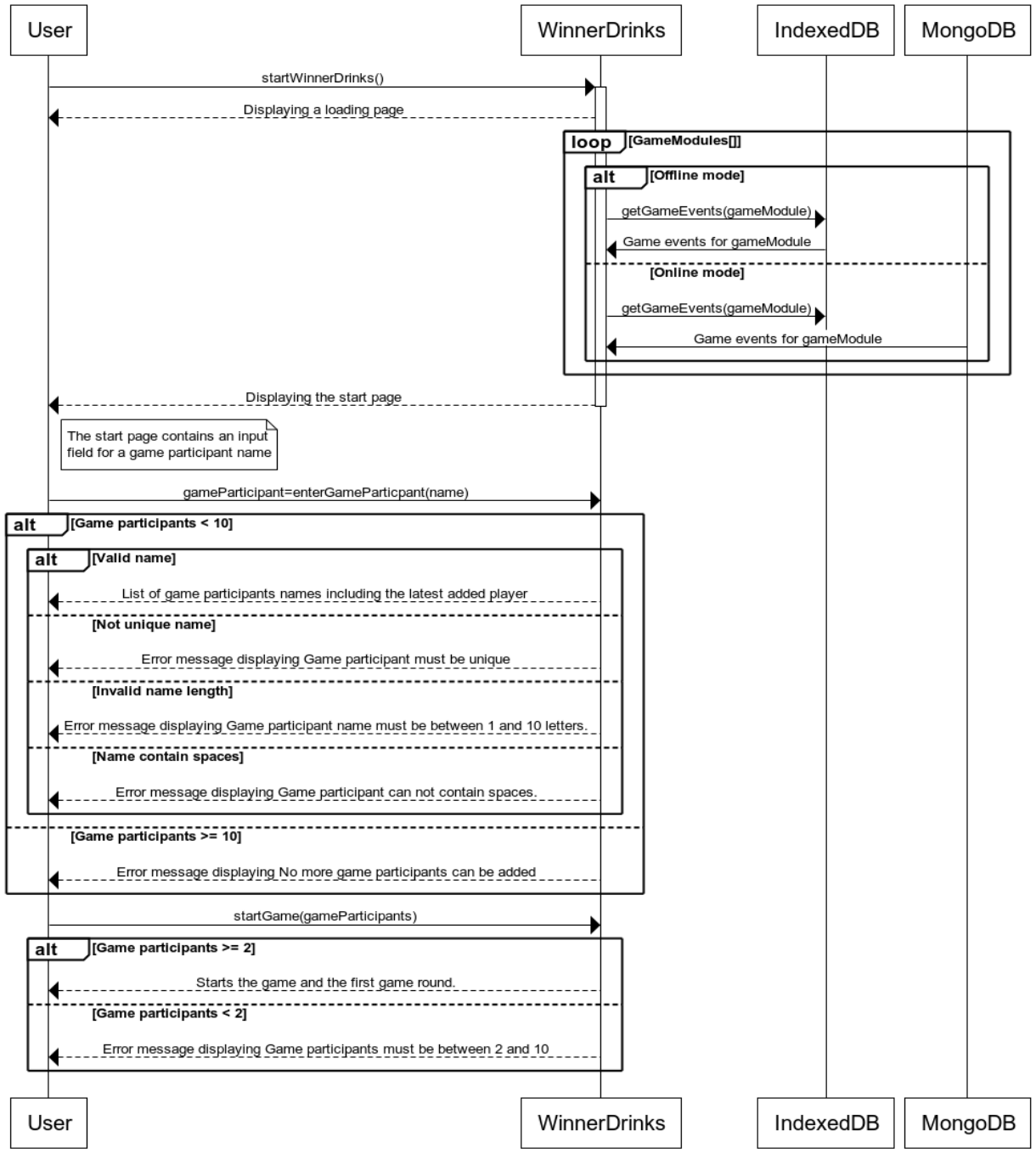


Figure 6: System sequence diagram of scenario "Start the game"

.

9.3.2 Play the game

This use case view is based on the use case *Play the game* and describes the scenario when a user plays the game. It is important for the overall user experience in the application.

Table 3: Actors, use case view "Play the game"

Actors	**Actors description**
User	A person that interacts with the system.
WinnerDrinks	The system

Table 4: Requirements, use case view "Play the game"

Requirement ID	**Requirement Name**
WD.NF.1	Game modules reliability
WD.NF.3	Game events reliability
WD.NF.5	Game events usability

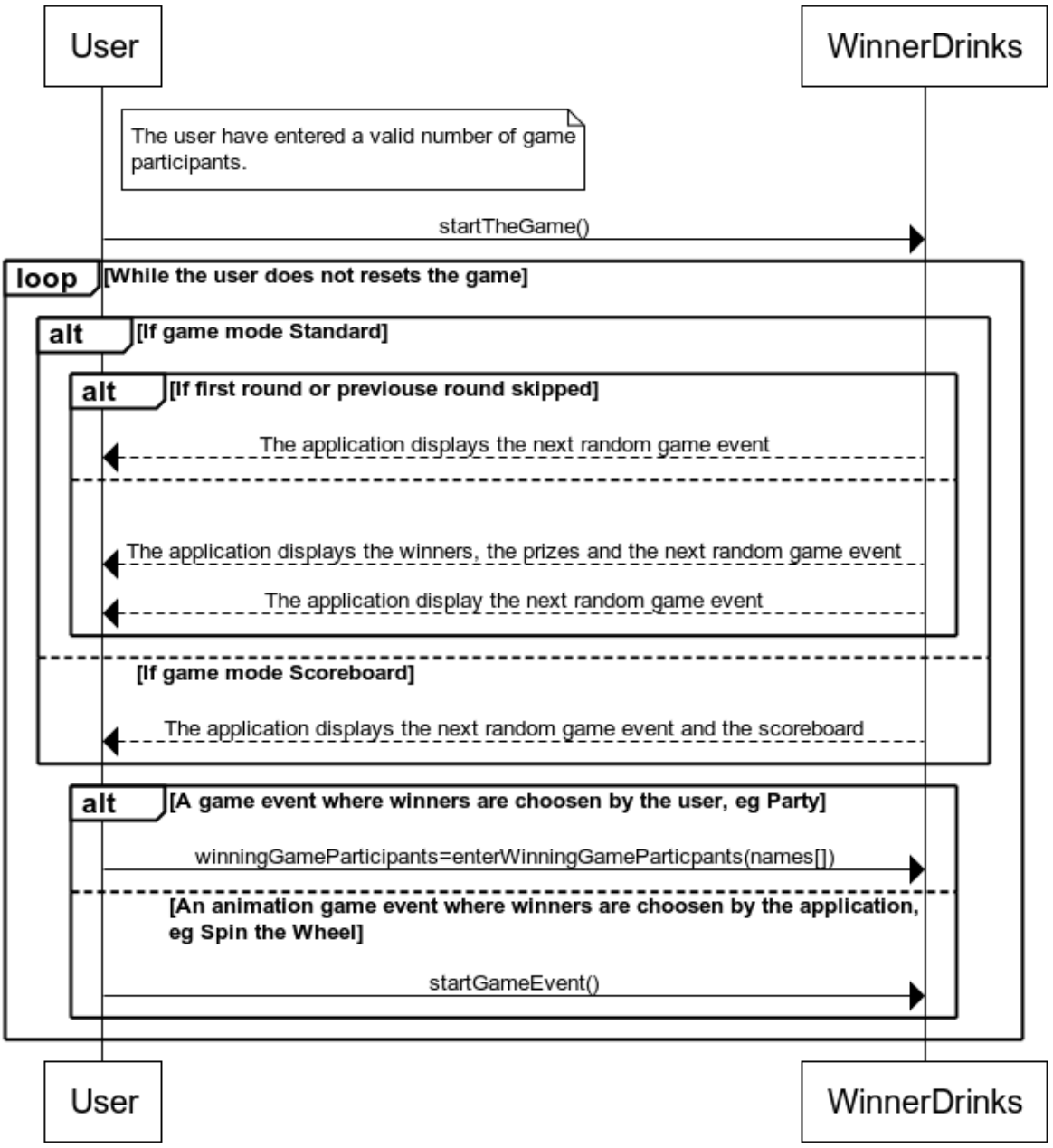


Figure 7: System sequence diagram of scenario "Play the game"

10 Design Document changelog

This section describes major changes that have been made to this document's requirements between different versions of this document. Since we work with an agile approach in this project to be able to get feedback from the project's stakeholders (primarily the client and the end users) and other parts of the project, such as the application design, the application implementation, alpha testing, and beta testing of the application, the feedback will be used continuously to improve this project and changes will be a part of the success of this project. Therefore, we will not describe every change between the different versions but only the major changes that have been made between the different versions as well as changes that have a significant impact on the application and the project.

10.1 Major changes between version 1 and 2

10.1.1 Architectural goals and philosophy

Some of the goals have been rewritten in order to make the goals more specific, instead of ambiguous statements like "good performance".

10.1.2 Assumptions and dependencies

The section "Scheduling accuracy" has been removed and the section "Trivia API availability" considerably shortened so that it only concerns the availability of the Trivia API.

10.1.3 Decisions, constraints, and justifications

Explanatory diagrams have been added as well as explanations accompanying these. In the section "General development constraints" we have added the rationales behind the respective DO and DON'T sections.

10.1.4 Architectural mechanisms

The "Extensibility tactic" and the "Persistence tactic" have been entirely removed as this did not explain a relevant tactic in the project as was intended, according to the feedback.

10.1.4 Key abstractions

The section "API" has been removed.

10.1.5 Architectural views

The diagrams have been updated to reflect the changes made to the application's architecture. We have also added tables that explain the concepts in the diagrams, in order to make it easier for a reader to understand the diagrams and more quickly get a good overview of the codebase. Section "Process view" has been renamed to "Use case view"and now also contains explanatory tables that accompany the sequence diagrams.