WHAT DO YOU MEME?

Boschi Francesco, Rossi Mattia, Tumedei Gianni

Table of contents

Abstract

Like many board games before it, it's time for "What Do You Meme" to have an online counterpart. The rules are simple: in each round, the player who is established as a judge has to pull a picture from the deck and place it on the board so that everyone can see it. Then each competitor chooses from the 7 caption cards in his hand the one that best describes the picture on the board. All the caption cards chosen by the players are collected and shuffled to faces covered by the judge. In this way the judge can decide the winner of that round with total impartiality after reading all the entries. In a clockwise direction each player becomes the judge of the next rounds. The winner is the one who first reaches the chosen maximum score.
The project aims to implement an online and distributed version of the game, with a lobby system that allows groups of friends to play together.

Through the implementation of an architecture based on microservices, each of which manages a specific aspect of the system, we want to provide players with the ability to create new games, join existing ones and complete game sessions following the rules available in the official game manual.

Goal and Requirements

The project's goal is to create an online version of the "What Do You Meme" card game. The system will allow each player to set his username and then either join a lobby through a code or create a new one of which he will assume control. When inside of a match, the rules are the ones aforementioned, just like in the card game. The system will be based on a client-server architecture, with the server consisting of a set of microservices and the simple web client served through one of them.

Functional Requirements

The project's functional requirements have been defined from a user's point of view and are examined in detail in the Functional Requirements Analysis section.

- Define the username
- Create a new lobby and define settings related to it
- Join an existing lobby or game by using the associated code
- Leave a lobby
- Start games from a lobby
- Play a game
  - Play a caption card
  - Vote for a caption card

Scenarios

Since the client does not require an on-device installation, players can use it through any web browser connected to the Internet, making the system completely cross-platform. It must be noted that, since it's not the focus of the project, the frontend UI will not be made responsive so there are no guarantees that it will be playable on small-screen devices.

As for a more practical definition of the usage scenarios, when connected to the website a user can either create a new lobby or join an existing one using a code provided by another player. Once the game has started, each player can view his cards and those on the game table and his choices will be constrained by the game logic. If a disconnection occurs, the game will continue without the player, who can reconnect at any time, keeping his previous score and cards. Finally, at the end of the game, players will be able to see the final ranking.

Self-assessment policy

The core part of the project will be developed using Java and the Spring framework. This choice allows to produce clear and flexible code, in which components are decoupled from each other and can be easily mocked for unit testing.

The effectiveness and correctness of the project will be verified through unit tests, during the whole lifecycle, by analyzing the defined requirements and through concrete play testing in the final stages.

The frontend, being just a proof of concept of what can be done in a client of the game, won't have any unit tests, but will be deeply play-tested.

Requirements Analysis

Functional Requirements Analysis

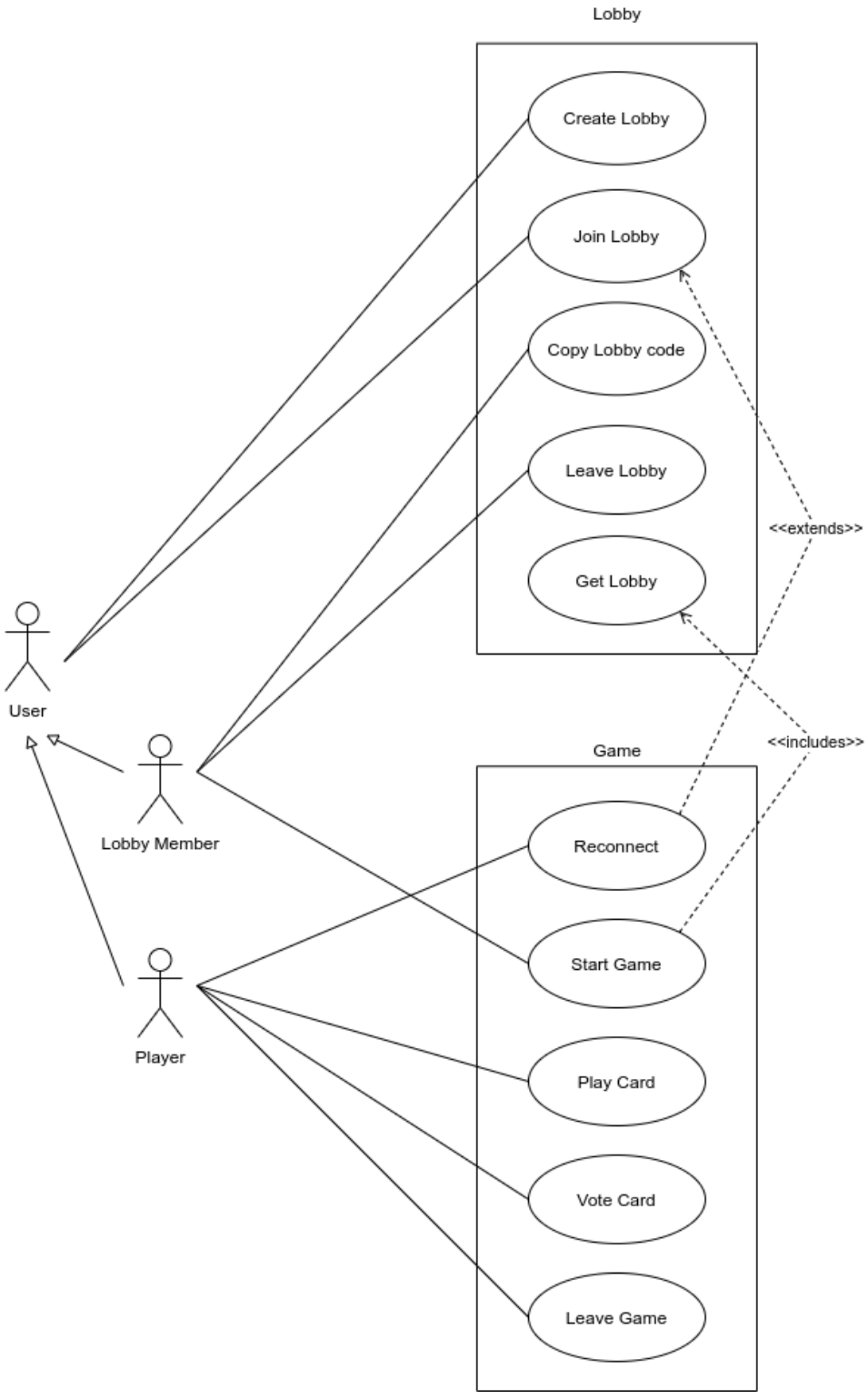
The functional requirements have been defined through user stories, then translated into a use case diagram, emerged after trying the board game and analyzing other online applications with a similar structure.

The identified user stories are the following:

- As a user, I want to define my username
- As a user, I want to create a new lobby
- As a user, I want to share the code of my lobby with others so that they can join
- As a user, I want to join an existing lobby
- As a lobby member, I want to leave the lobby I'm in
- As a lobby owner, I want to define the maximum game score for my lobby
- As a lobby owner, I want to start a game
- As a player, I want to leave the game I'm in
- As a player, I want to be able to re-join a game after a disconnection
- As a player, I want to play a card
- As a player, I want to vote for a played card

In the transition from user stories to use case diagram, it was decided to merge the actions *Create a new lobby* and *Define the maximum score for my lobby* into a single *Create lobby* use case. Basically, the maximum score will be set in the lobby creation process.
In addition, the *Define my username* action will be contextual to the *Create lobby* and *Join lobby* actions, and as such is not present in the diagram.

The following image shows the use case diagram derived from the user stories above.


Lobby
Create Lobby
Join Lobby
Copy Lobby code
Leave Lobby
Get Lobby
<<extends>>
User
Lobby Member
Player
Game
<<includes>>
Reconnect
Start Game
Play Card
Vote Card
Leave Game

At this stage, it emerged that the system should be split up in two main components: Lobby and Game. As the names suggest, the first is going to handle the lobby system and the second is going to implement the game logic. The system architecture is detailed in the corresponding section (Architecture).

Non-functional Requirements

The non-functional requirements identified in the analysis phase and listed below are mostly related to the system architecture and code organization.

- Scalability: the system must be able to handle a variable number of simultaneous games without suffering from performance issues
- Resilience: a failure of one entity in the system should not halt its operation
- Availability: the system must have a high availability degree
- Testability: the system must be easy to test, in its individual components and as a whole
- Ease of deployment: the system and its components must be easy to deploy
- Maintainability: maintenance operations on the system must be easy to perform
- Code Quality and Readability: produced code must be understandable and high quality
- Decoupling: each part of the system should interact with the others through standardized means of communication, and the re-implementation of a component in a different programming language and/or framework should have minimal to zero impact on other components

Used technologies

Based on the functional and non functional requirements emerged in the above sections, the following technologies have been chosen to carry out the project.

Languages

- Java: chosen as the development language for the backend, as it has been previously studied and is widely used, supported and documented. In addition, by adopting the OO paradigm, it allows to produce high quality code that more easily meets the non-functional requirements listed above.
- TypeScript: it's the go-to choice for type safe frontend code.

Frameworks

- Spring Boot: having selected Java as the programming language, Spring Boot is the obvious choice for the framework. It is typically used in the development of web applications, since it integrates a web server and has an ecosystem of libraries that cover a wide range of use cases.
- Vue: there are a lot of frontend frameworks to chose from nowadays. Vue has been selected for its ease of use, great TypeScript support (from version 3) and familiarity from previous projects.

Tools

- Gradle: chosen over Maven as the build system for the Java microservices thanks to its flexibility, DSL capabilities and due to the fact that it has been used in previous projects.
- Docker: since What Do You Meme is a system based on microservices, the design of the project is particularly well suited for the use of Docker, useful both to orchestrate the various containers and to simplify the deployment phases.
- NGINX: chosen as the frontend web server, thanks to its reverse proxy capabilities it will also be used to expose all the microservices on a single port.

Other technologies

- MongoDB: given that the system will deal with data that's organized in independent lobbies, the choice of a non relational DMBS makes sense because it obviates the problem of impedance mismatch. Furthermore, MongoDB is suitable for the distributed nature of the project since it's easily replicable and scalable.
- REST: mean of communication for microservice to microservice interaction
- WebSocket: chosen in favor of HTTP + REST for frontend-backend communication since it enables bilateral push messaging

Design

Architecture

This section describes how the system is split into separate microservices. Each one of them should ideally serve a single purpose and be loosely coupled from the others. According to this logic, the following microservices were identified during the design phase.

Database Service

For simplicity's sake, it was decided to have a centralized database instead of one for each microservice that needs data persistence. This design choice is easily reversible because each microservice has its own collections, independent from the ones of other services.

Database Import Service

This service was added in the later stages of the project to check for the presence of essential data inside Database Service on system boot (namely, the card collections), and perform data entry if needed. This allowed Database Service to be completely independent from the data structures inside it.
This is the only microservice that, after performing its operations, immediately terminates and is no longer executed until the system is shut down and started back up.

Lobby Service

Manages the lobby system that allows players to meet and start games.

Game Service

It was decided to split the game logic from the lobby management to have a nice separation of concerns. Since scaling can be performed on a microservice basis, this has the benefit of allowing the assignment of additional resources to Game Service, which is probably going to have a higher workload compared to Lobby Service.

Middleware Service

As the name suggests, it acts as the middleware between frontend and backend, sorting out requests from the client to the microservice responsible for their fulfillment and forwarding the latter's response back to users.

Static Service

Has the main purpose of serving any static file needed by the system (frontend code, card images). Furthermore, it performs reverse proxying on Middleware Service, making it possible to expose a single port for both the frontend and backend and redirecting requests to the correct service based on the URL structure.

Structure

In this section we discuss the structure of the microservices based on Java and Spring, which make up the core of the system.

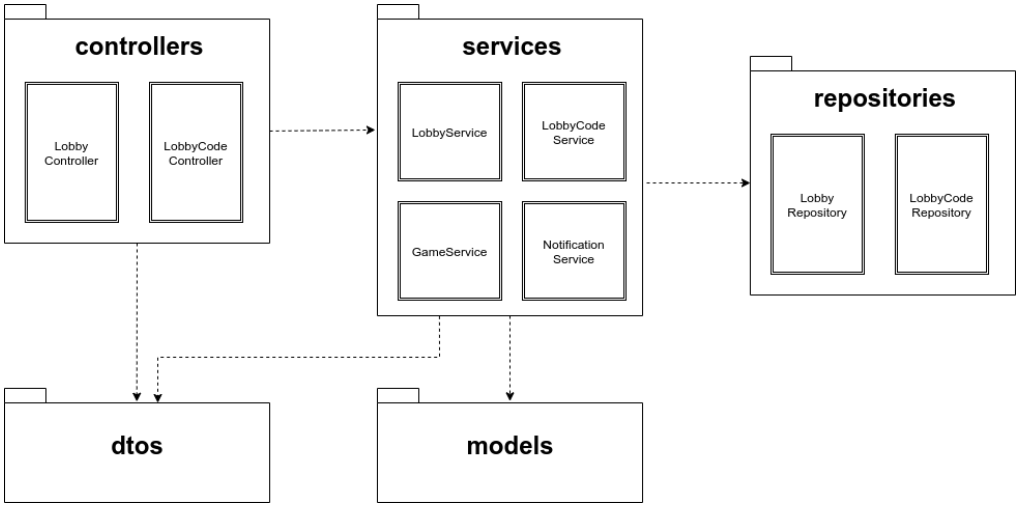
A typical Spring Boot project is organized by concept into *modules*, where each module contains all the code required for a certain aspect of the system (such as *Authentication*).
What Do You Meme is a system composed by microservices, where each project has a single purpose and, by consequence, would have a single module if the aforementioned directory structure was used. Therefore, it was decided to organize files in a simplified manner, grouping them by *type*.

The following list shows the package structure of a single microservice. The same organization is applied to every microservice.

- `controllers`: classes belonging to this package define the interface offered to the outside by the microservice. Each controller contains methods that intercept requests (HTTP or WebSocket), call a service to get the response and forward it to the caller
- `dtos`: contains the Data Transfer Objects, which encapsulate any data sent to the microservice from the outside
- `exceptions`: contains all exceptions that can occur when a method of the services package is called
- `models`: contains the definition of all entities returned to players by the microservice
- `repositories`: contains the implementation of different MongoRepositories, that abstract and take care of any database operation

- `services`: contains all the services needed to manage the business logic of the system and satisfy requests forwarded by controllers or other services

The following image show the package diagram for the aforementioned project structure.



UML class diagrams for the Spring microservices can be found at the following links:

- middlewareservice class diagram
- lobbyservice class diagram
- gameservice class diagram

Behavior

This section goes into detail on how the Java microservices behave internally.

Game Service and Lobby Service

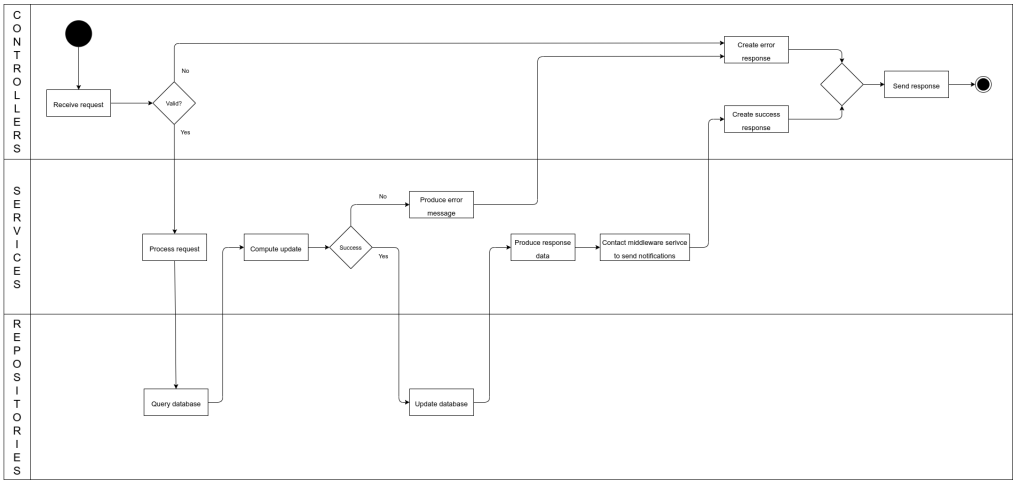
The internal behavior of these services is organized in three layers (also defined as packages, see Structure). Each layer can only interact with the adjacent ones.

- The controller layer defines external REST interfaces to interact with the microservice (methods, data transfer objects, response status codes, reply data). It operates by receiving a request, validating it, forwarding it to the appropriate service and sending back the response to the requester.
- The service layer implements the microservice's business logic, abstracting away from the upper layer. When a method in a service is called, it can either

produce a successful response or throw a meaningful exception, informing the invoker of what went wrong.

- The repository layer encapsulates the logic required to access data sources. In particular, the following MongoDB collections are defined:
  - Lobby Service: defines a single Lobby collection to store lobby data
  - Game Service: defines three collections:
    - ImageCard: stores the full list of image cards
    - CaptionCard: stores the full list of caption cards
    - Game: stores game data

Below is the UML activity diagram for Lobby Service and Game Service.

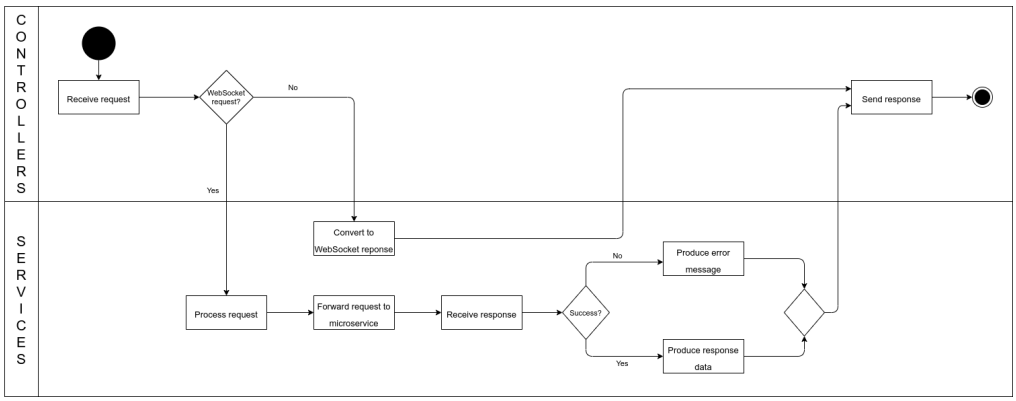


Middleware Service

This service's internal organization is very similar to the one of Game Service and Lobby Service, with two main differences:

- The controller layer mainly defines WebSocket endpoints instead of REST ones, since they are the primary mean of communication used by this service
- There is no repository layer since this service only acts as a medium between the frontend and other services and as such it has no need to persist any data

Below is the UML activity diagram for Middleware Service.



Interaction

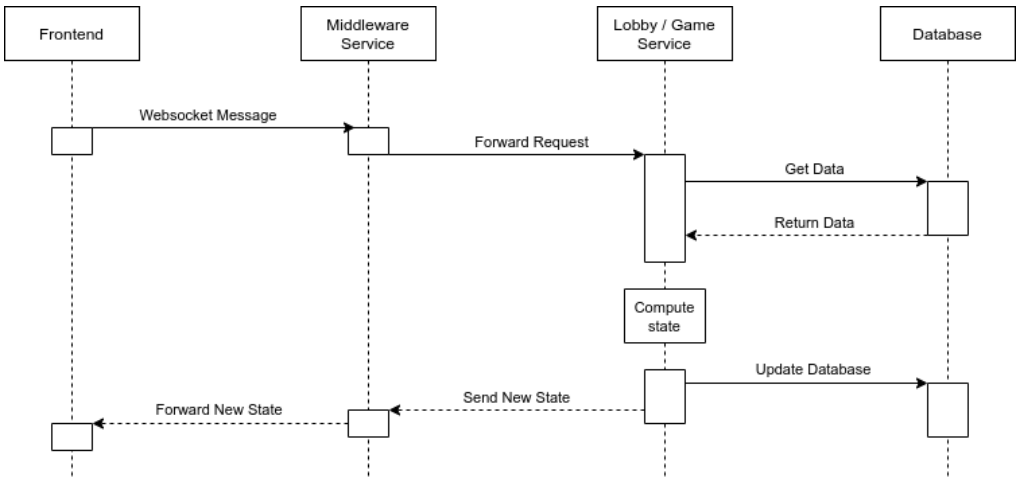
This section explains how the various components of the system interact with each other.

The state of a game consists of two parts: one is public among all the players (game board) and the other is private to each player (cards in his hand).
A critical point of the interaction with the system is that one player's action can trigger an update on the game board and each player's hand at the same time. It is therefore necessary to send a private message to each player containing both public and private data. To achieve this, microservices can send push messages to any player by contacting an appropriate endpoint of Middleware Service.

A typical interaction with the system, during a game, is composed by the following steps:

- The player, interacting with the frontend, triggers a WebSocket message to the backend
- Middleware Service is in charge of forwarding the message as a HTTP request to the appropriate microservice
- The microservice performs business logic operations, interacting with the Database Service in the process to get/set data. During this process, it computes the updated game state
- Lastly, the game state is forwarded back to each player's frontend through Middleware Service

Below is the UML sequence diagram for the system that represents the interaction between the backend and a single instance of the frontend.


Frontend
Middleware Service
Lobby / Game Service
Database
Websocket Message
Forward Request
Get Data
Return Data
Compute state
Update Database
Send New State
Forward New State

Implementation Details

Spring microservices

Most of the implementation details on the Spring microservices are found in the Javadoc for Lobby Service, Game Service and Middleware Service. Some additional topics that don't figure in the docs are reported below.

application.properties files

Each microservice has two properties files: `application.properties` and `application.docker.properties`. The first is loaded when the project is ran through `./gradlew bootRun`, while the second is used in the entrypoint of the Dockerfile.
This configuration allows to specify different parameters based on the environment and is mainly needed because the microservices' URLs are different when running locally (i.e. `http://localhost:8090`) vs. on Docker (i.e. `http://wdym-lobbyservice:8090`).

Turn timeout during games

The various phases of the game do not last indefinitely, but are marked by two types of timeouts, one for the playing phase and one for the voting phase. The exact values are defined inside gameservice's `application.properties` files, and are respectively 120 and 90 seconds.
To make it possibile to trigger timeouts, `SchedulingService` was created. This class is basically a wrapper for a `ScheduledExecutorService`, with a single method to schedule a `Runnable` for execution after a given time.
Inside `GameService`, a `scheduleNextPhase` method was created. Its purpose is to schedule a new timeout at the beginning of each phase. Once the runnable is executed, it checks if its phase is still running and, if that's the case, it moves the game to the next phase.

This solution is effective even if there are multiple instances of `gameservice`, since it guarantees that only one of them schedules a runnable for each phase timeout.

MongoDB documents expiration

To ensure that no stale data pollutes the database, a MongoDB TTL index is set on each document (beside cards) to make it expire after 24 hours. This should only be necessary if a major outage occurs, closing the players' connections without marking them as disconnected from their lobby.

Frontend

The frontend client implemented in the project is just a proof of concept on how it's possible to interact with the system, and as such it has a very simple structure. The WebSocket communication is managed by a single `useWebSocket` utility, that:

- Opens the connection to the backend at startup
- Parses any incoming message
- Has a list of callbacks that can be attached from the outside and get executed once for each message (to update the UI)
- Has a `send` method to dispatch a new message to the backend

Besides the Vue framework, the main libraries used by the frontend are:

- Stomp.js for the STOMP over WebSocket communication
- Tailwind CSS and DaisyUI to style the user interface

Docker

This section gives some implementation details on the Docker configuration for each microservice, then it explains how they are setup when deployed through Docker Compose.

dbservice and dbservice-import

`dbservice` is a very simple container based on the mongo official image. The only thing added on top is the initialization of a single host replica set. This enables the use of transactions from the Spring microservices and makes it easy to setup bigger replica sets if the need arises.

`dbservice-import` is based on the mongo image too. Thanks to this, it has access to the `mongoimport` utility, and uses it in a script to seed the `dbservice` database with the cards collections. This container is executed at startup and immediately terminates after the import is performed.
To make sure that the database is reachable when the script runs, a utility called `docker-compose-wait` (GitHub link) was used. It's a simple script configured through environment variables to run until a given host is available (in this case, it waits for `wdym-dbservice:27017`).

lobbyservice, gameservice and middlewareservice

The Spring microservices' Dockerfiles perform a two step build.
The first part happens on a Gradle image. Here, the source code is copied from the host to the container and, if the tests are passing, is built to a production jar called `app.jar`.
The second part is on a Zulu OpenJDK Alpine image. Here, the `app.jar` file is copied from the previous image and then executed.
This process makes it possible to have a very slim image at runtime (about 60MB), since the only thing needed to run a jar file is the JRE.

staticservice

Like the Spring microservices, `staticservice` uses a two step build, with the first part consisting in building the frontend through the official Node image, and the second part copying over the built website and starting an NGINX container.

One peculiarity of this container's build process is the following code:

```
COPY package*.json .
RUN npm ci
COPY . .
RUN npm run build
```

Thanks to the layer system that Docker images use, copying `package.json` and `package-lock.json` and running `npm ci` to install the dependencies *before* transferring the source code, allows to cache the dependencies and only re-install them if the list (in the `package.json`) has changed.

As for the NGINX configuration, it's very similar to the default one provided by the Docker image. The main difference the reverse proxy that redirects any request starting with `/wsapp` to Middleware Service, as shown below.

```
location = /wsapp {
    proxy_pass http://wdym-middlewareservice:8080;
    proxy_http_version 1.1;
    proxy_set_header Upgrade 'websocket';
    proxy_set_header Connection 'Upgrade';
}
```

Docker Compose

The Docker Compose configuration is split up in three files:

- `docker-compose.yml`: for each container, defines the name, image, build path, dependencies and environment variables.
- `docker-compose.dev.yml`: maps to the host machine all of the ports exposed by the containers, so that they are reachable for debugging during development.
- `docker-compose.prod.yml`: exposes all of the ports to the internal Docker network, but only maps port 8081 of `staticservice` to port 80 of the host, so that, when going to production, it's the only container reachable from the outside. In addition, this file configures a restart policy of `on-failure:3` on all containers, so that they automatically try to restart up to three times in case of error

This setup makes it possible to start the system in two modes, one for development and one for production, with the following commands:

- Development:
  ```
  docker-compose -f docker-compose.yml -f docker-compose.dev.yml up --build
  ```
- Production:
  ```
  docker-compose -f docker-compose.yml -f docker-compose.prod.yml up --build
  ```

Additional notes

Most of the containers use the `alpine` variant of their base image to keep the size down. Alpine Linux is a lightweight, secure and resource efficient distribution and its docker image only weighs 5MB, so any image based on it basically takes up just the space needed for its dependencies.

Self-assessment

It was decided to evaluate the system in two main ways:

- through a TDD approach, during the whole lifecycle, by developing code that has to pass previously defined unit tests, thus ensuring that the product always meets the project requirements;
- through concrete play testing in the final stage of the project, to ensure the fulfillment of every user story and consequently of functional requirements.

Testing

Each microservice has its own suites of unit tests, independent from the rest of the system. Each suite performs tests on a single component of the microservice, while mocking any other component that interacts with it through Mockito.

The categories of tested components are listed below, with some considerations:

- Controllers: controller tests aim to check the correctness of the API interface provided to the outside, so they must perform input validation and verify the structure of the output for each endpoint.

  - For REST controllers this means testing against URL parameters, request body and response status code.
  - For WebSocket controllers, the structure of input and output objects is tested.

  Each controller generates the response though a service call. To test controllers, it is therefore necessary to mock services below them.

- Services: service tests aim to verify the business logic by calling service methods and checking their return values or thrown exceptions.
  Services use repositories to interact with the data layer and HTTP calls to communicate with other microservices. There is no database or network during unit testing, so in this case it's necessary to mock repositories and REST templates (the Spring class to make HTTP requests).

- Models: although most of the models are already tested through controller and service tests, some specific features such as secondary constructors, getters and setters are tested in this suite.

Integration testing

Performing integration testing on a component of a microservice system requires a considerable setup. For example, an integration test on lobbyservice would require starting (or mocking) gameservice and dbservice because, depending on the system state, lobbyservice could interact with the database and/or gameservice to process a request. As a result, integration tests cannot be integrated into the deployment pipeline like unit tests, but would require a standalone CI pipeline (or on-demand local execution, which would take quite a long time). Given the complexity of the problem, not having a large number of pipeline minutes available on GitLab, where the repository is hosted, and considering the good coverage obtained from unit tests, it was decided not to carry out integration tests and instead focus more on detailed play testing.

Coverage

Test coverage was verified using JaCoCo. At the end of the project, coverage data is the following:

- lobbyservice has a coverage of 94,9%
- middlewareservice has a coverage of 71,1%
- gameservice has a coverage of 91,5%

The overall coverage percentage obtained is 91% (weighted average). This value is negatively influenced by some features that cannot be tested individually, as the behavior of some components cannot be simulated. For example, full unit test coverage on middlewareservice is impossible because it serves the purpose of forwarding requests between frontend and backend.

Merged Sessions

What Do You Meme Total Coverage

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
gameservice		91%		59%	127	468	128	1.263	40	335	0	44
middlewareservice		71%		28%	29	55	41	129	15	41	0	11
lobbyservice		94%		76%	36	231	37	645	22	195	0	33
Total	816 of 9.387	91%	145 of 366	60%	192	754	206	2.037	77	571	0	88

Created with JaCoCo 0.8.7.202105040129

Continuous Integration

Using GitLab CI/CD as a Continuous Integration environment allows to automate some procedures related to the project build, such as compilation, testing and quality assurance. The pipeline entry point is the `.gitlab-ci.yml` file, which contains instructions on how to build the project.
Since the project is deployed through Docker Compose, the CI pipeline is based on the Docker image with the Docker-in-Docker service, thus allowing the simulation of a full deployment procedure. After checking the correct installation of Docker and Docker-Compose, the two steps necessary to complete the pipeline are carried out:

- The first, called `compose-build`, verifies the correct syntax and executes the DockerFiles related to each container, which in turn build and test the corresponding microservice
- The second, called `compose-up`, boots the system to check the presence of any startup errors

Documentation

To automate the process of deploying the documentation, we created a dedicated *docs* branch, containing the compiled Javadoc HTML of the three main microservices. This branch has a pipeline file (`.gitlab-ci.yml`), that gets triggered on each push and uses GitLab Pages to deploy a static website containing the documentation.
Since the project's repository is part of a group, documents published online are only visible to authorized users.

Satisfaction of Non-Functional Requirements

This section explains how each non-functional requirement listed in the corresponding chapter has been fulfilled.

Scalability and Availability

These requirements are guaranteed by the fact that the microservices with the highest workload (lobbyservice and gameservice) are stateless and infinitely replicable. Furthermore, since lobbies are independent from each other, the whole system can be replicated and hosted somewhere else, configuring DNSs so that they redirect requests to the appropriate system based on the geographical area of origin.

Resilience

The following list explains, for each microservice, the outcome of an eventual outage.

- dbservice: if a MongoDB instance goes offline, there should be no impact on the system functionality, as long as a replica is configured to take its place
- lobbyservice: this microservice is fully stateless, so there is no impact if one of its instances stops working
- gameservice: if an instance of gameservice goes offline, the game phases timeouts scheduled by it will never run. This has minimal impact on the game, since every turn will end anyway, as long as each player performs his move
- middlewareservice: a crash on middlewareservice can make existing lobbies permanently unreachable from the outside. The Docker restart policy should quickly bring the service back up so that users can re-join. Stale lobbies will be automatically deleted after 24 hours (see the *MongoDB documents expiration* section of Spring microservices)
- staticservice: this is the entry point to reach other microservices, and as such a crash will put the whole system offline. However, thanks to being stateless and using Docker auto restart capabilities, it should have very little downtime in case of an outage.

Testability

Ensured by the organization of unit tests (see Testing)

Ease of deployment

By adopting Docker to package and deploy the system, deployments only require one dependency and can be performed with a single command (see Deployment Instructions).

Maintainability, Code Quality and Readability

Development has followed the OOP paradigm and the components subdivision proposed by Spring, which has allowed the realization of a complex code base that still retains good quality, readability an, consequently, maintainability. These requirements are further satisfied through the reuse of shared code, the application of coding patterns known from the literature, and the enclosed Javadoc.

Decoupling

Thanks to the microservices architecture, modifications to the internal logic of a service should have no impact on other components. With the usage of HTTP and WebSocket for communication, it's possible to re-implement any part of the system with a different language or framework, as long as it follows the defined communication interfaces.

Deployment Instructions

The following section is taken directly from the repo's README.

Requirements

- Docker and Docker Compose on the machine where the system is going to be installed
- A web browser to access the game client

System setup

- Clone the repo and cd into it

```
git clone https://gitlab.com/pika-lab/courses/ds/projects/ds-project-boschi-r
cd ds-project-boschi-rossi-tumedei-ay1920
```

- Download the archive with the images of the cards from here and extract it into `whatdoyoumeme.frontend/public/`
- Start the system with `docker-compose`

```
docker-compose -f docker-compose.yml -f docker-compose.prod.yml up --build
```

- Access the frontend from `http://localhost` (or any external IP/domain configured on the machine)

Usage Examples

Having started the system by following the Deployment Instructions, the frontend and the backend are reachable from http://localhost and ws://localhost/api respectively. If the system was started in development mode, the internal microservices ports are exposed as well, so it's possible to contact each backend service individually by using the following addresses:

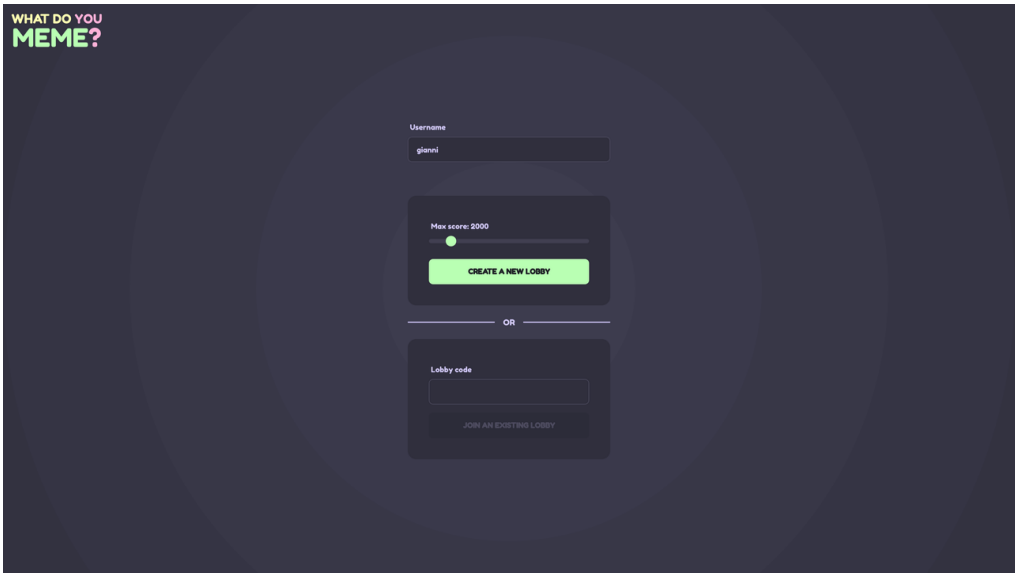
- dbservice: mongodb://localhost:27017
- lobbyservice: http://localhost:8090
- gameservice: http://localhost:8091
- middlewareservice: ws://localhost:8080

Frontend usage

This section explains how each user story defined in the corresponding chapter can be completed by interacting with the frontend client.

Create or join a lobby

From the home page, a user can decide whether to create a new lobby or join an existing one. To create a lobby it is required to define the maximum score of the games that will take place within it, while to join one the user has to insert the unique code that identifies it. Both procedures then require to enter the username that the user wants to have in the lobby.



Share the lobby code

Each lobby is identified by a 6 characters code. From the lobby and game pages (`/lobby` and `/game`), users can copy this code to share it with others so that they can join too.

Start a game

When there are at least three players inside of a lobby, its owner can decide to start the game by clicking on the dedicated button at the bottom.

Leave a game or lobby

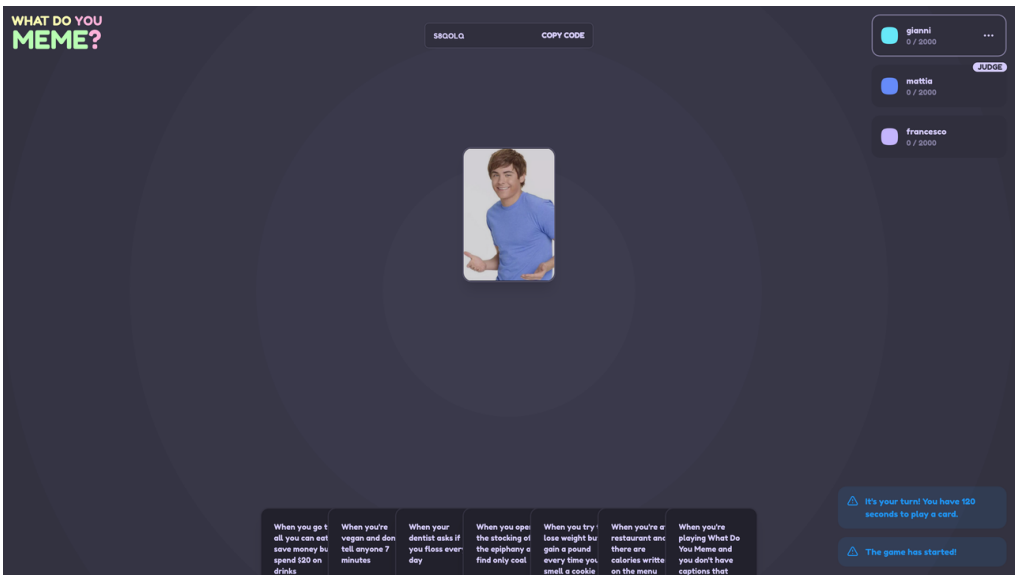
To leave a lobby, the user has to refresh the page or close the browser tab, and that will implicitly remove him from the lobby.
To leave a game, the user has to explicitly click on the three dots in the top right and choose the *Leave game* option. This will permanently remove him from the game, that will continue on without him.

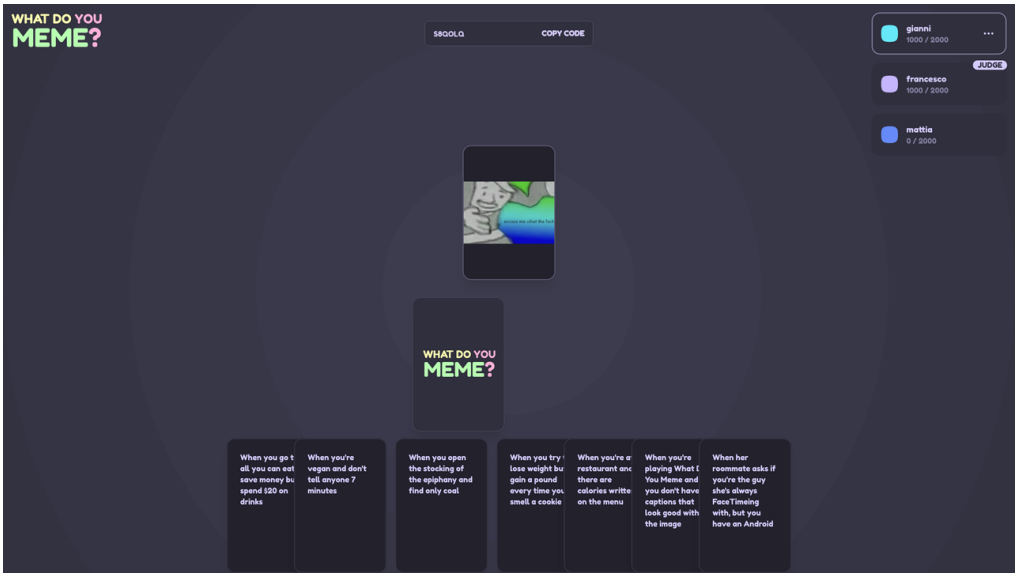
Re-join a game or lobby

The procedure is identical to joining a lobby for the first time. If a game is running within the lobby, the user can only join if he was already part of it and got disconnected, for example due to a network issue.

Play

Playing the game is very simple and consists of just clicking on cards. The user can see who is the judge and check players' score from the top right section. Every time something important happens in the game, for example if the judge votes for a card, a notification appears at the bottom right.


WHAT DO YOU MEME?
S8QOLQ
COPY CODE
gianni
0 / 2000
JUDGE
mattia
0 / 2000
francesco
0 / 2000
When you go t all you can eat save money bu spend $20 on drinks
When you're vegan and don tell anyone 7 minutes
When your dentist asks if you floss ever day
When you ope the stocking of the epiphany a find only coal
When you try lose weight bu gain a pound every time you smell a cookie
When you're a restaurant and there are calories writte on the menu
When you're playing What Do You Meme and you don't have captions that
It's your turn! You have 120 seconds to play a card.
The game has started!


WHAT DO YOU MEME?
S8QOLQ
COPY CODE
gianni
1000 / 2000
JUDGE
francesco
1000 / 2000
mattia
0 / 2000
WHAT DO YOU MEME?
When you go t all you can eat save money bu spend $20 on drinks
When you're vegan and don't tell anyone 7 minutes
When you open the stocking of the epiphany and find only coal
When you try lose weight bu gain a pound every time you smell a cookie
When you're a restaurant and there are calories writte on the menu
When you're playing What D You Meme and you don't have captions that look good with the image
When her roommate asks if you're the guy she's always FaceTimeing with, but you have an Android

Conclusion

The project was carried out to achieve the goals set during the analysis phase. Coming to the conclusion, we are satisfied with the work that has been done as we have managed to fulfill all the requirements.

The realization of the project has seen us engaged mainly in the following activities:

- Study and documentation of both problem and solution: first we had to analyze the problem and find the most suitable technologies. Then we had to define in detail the design of the solution in order to realize a distributed system that meets the needs.
- Further study of the used technologies and frameworks: some of the used technologies and frameworks were unknown or only superficially known to us. In order to be able to complete the project, a necessary process of in-depth studying was started in the initial phases and carried out for the entire life cycle.
- Implementation: the most challenging part of the project was the implementation which, although it followed the guidelines defined during the analysis phase, was not free of difficulties.

Criticalities

The main difficulties encountered during development are listed below.

Debugging

The debugging phase was not always simple. Having numerous microservices interacting with each other, finding bugs required more steps than just looking at the console of a microservice or setting up a breakpoint, and often involved monitoring multiple services at the same time.

NGINX setup

Setting up NGINX to perform reverse proxying on middlewareservice was not easy, mostly because none of the group members had previous experience on editing the `nginx.conf` file. In the end, the main issue was that creating a `server` block in the configuration file overrode the default NGINX one, preventing the frontend from being served correctly. The simple solution was to include a `location = /` block with instructions on how to do it.

Seeding data into the database

Testing with Docker implies having to destroy and re-create the containers multiple times. This can be a problem because some data needs to be present in the database for the system to work. Docker volumes can help, but only to to some extent: if the database models change, it's required to clean up the documents. The solution was to create dbservice-import to automatically import any needed data at system startup.

Creating the MongoDB replica set

Since multiple microservices instances interact with the database in a concurrent manner, transactions were required to make everything consistent. MongoDB supports them starting from version 4 through the creation of a replica set, so it should have been no problem. However, when we implemented them by following the official MongoDB and Spring documentation, they did not work, preventing our Spring microservices from being able to connect to the database. We later found out that this was due to a bug in the MongoDB version we were using. The issue was quickly fixed by switching to 5.0.5.

Future works

The work done so far has allowed us to implement a transposition of What Do You Meme that, feature-wise, is substantially identical to the board version of the game. Therefore, in a future perspective, most of the aspects on which the project can evolve (listed below) are related to the infrastructure or rely on the level of customization given to the user, rather than on the introduction of new game rules.

- In the later stages of the project, we noticed that middlewareservice was starting to behave more and more like a message broker. A future upgrade for the system would be to replace it with something like RabbitMQ or Apache Kafka, to allow for even better scalability

- The project is well suited for the implementation of an authentication system for the users to sign up. This would open up a lot of new possibilities, such as:
  - setting user profile images;
  - interacting with other users more easily, for example by inviting them into a lobby without having to share the code through an external platform;
  - giving the lobby owners the ability to customize cards, both image and captions, by uploading their own
  - implementing a leaderboard system
- The online publication of the game is certainly one of the future aspects that represent its natural evolution, but it presents some bureaucratic criticalities. Since the board version of What Do You Meme is protected by copyright, it would be necessary to reach an agreement with the creators of the game to decide under which constraints it can be published
- The nature of What Do You Meme and the system architecture fit very well with an expansion on multiple platforms. The first step would be to make the web frontend responsive, and then move on to creating a mobile app and a desktop client.

Acquired knowledge

This project has greatly enriched our knowledge in a variety of areas.

We learned how to develop web applications using Spring, which is an important knowledge in the professional field since it's the one of the leading frameworks in this department, used by the likes of Netflix.

In this project, the use of Docker turned out to be a key part and was exploited in a much more complex way compared to past experiences we had. Thanks to this, it was possible to understand in more detail many features that will be of use in future projects.

Analyzing the project from an architectural point of view, it has allowed us to extend our knowledge about microservices, their strengths, criticalities and needs. This experience will surely be useful for future work since this model is rising in popularity among developers and becoming one of the most used for distributed systems development.