Progetto di Tecnologie e Applicazioni Web

Battleship on the MEAN stack

Ardi Suka
879439@stud.unive.it
Ca' Foscari University
Italy

Filippo Zane
880119@stud.unive.it
Ca' Foscari University
Italy

Mirco De Zorzi
891275@stud.unive.it
Ca' Foscari University
Italy

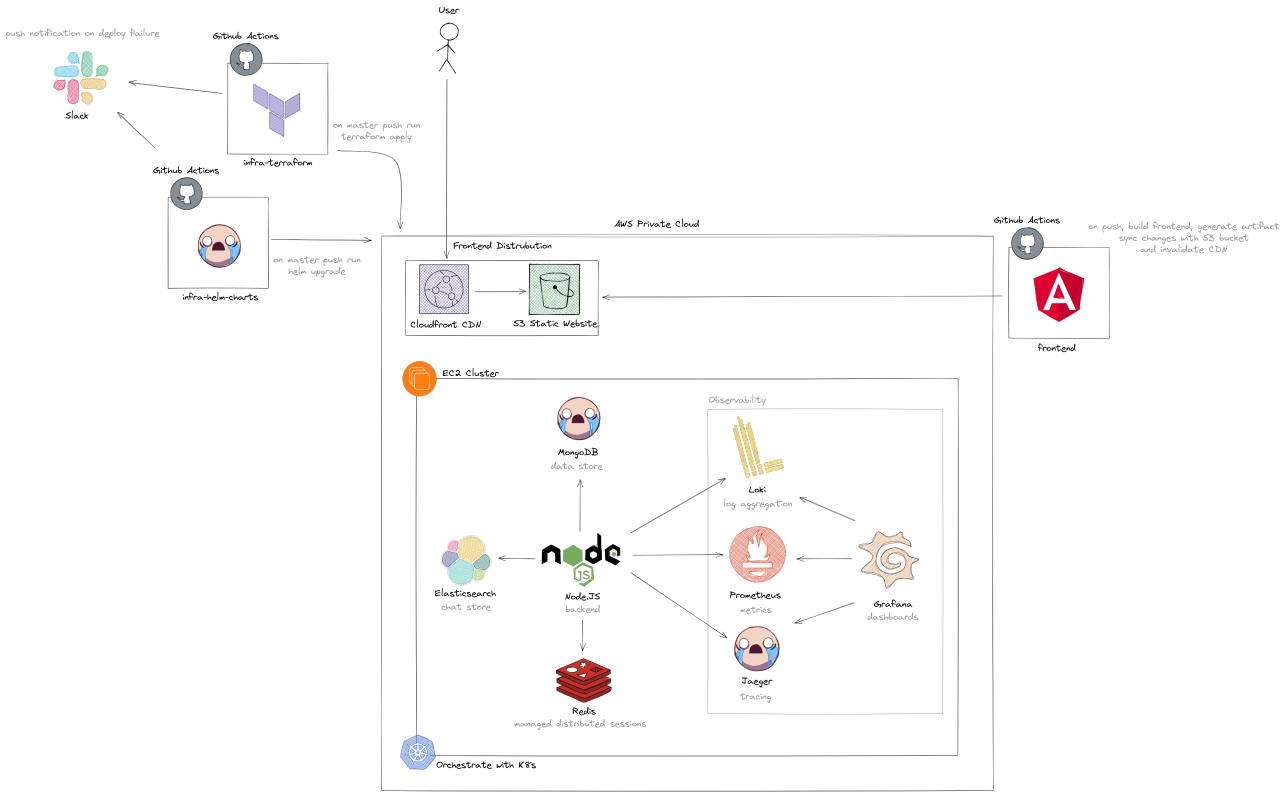

Figure 1: Original infrastructure deployed through Kubernetes running on an EC2 Cluster.

ABSTRACT

The goal is to develop a full-stack web application, comprising a REST-style API backend and a SPA Angular frontend to let the users play the game of "Battleship".

To run the project execute the following command:

```
$ docker compose up --build
```

1 TECHNOLOGIES STACK

During the design phase of the project, we have tried to create a scalable and future-proof infrastructure, leveraging all modern technologies that would allow us to handle a huge amount of traffic, while keeping track of everything that goes on in the system.

Thus we theorized an infrastructure that would levarage the following stack:

(1) **A**WS: Infrastructure hosting service.
(2) **A**ngular: Frontend framework.
(3) **E**lasticSearch: Search engine to index user chats.
(4) **E**xpress: Backend library for routing.
(5) **H**elm: Templating tool for Kubernetes.
(6) **J**aeger: End-to-end distributed tracing service.
(7) **K**ubernetes: Container orchestration tool.
(8) **L**oki: Log aggregation service.
(9) **M**ongo: No-SQL data-store for user related data.
(10) **N**ode: Backend language.
(11) **P**rometheus: Timeseries database for monitoring.
(12) **R**edis: Key-value data store for distributed sessions.
(13) **T**erraform: IaC to deploy and provision AWS.

Due to time constraints we have opted to only implement the essential and more conventional **MEAN** stack.

2 SYSTEM ARCHITECTURE

The application is a simple CRUD single page application developed in Angular and Node. MongoDB is used as a data store.

3 COMPONENTS

3.1 Docker & Docker Compose

The following commands can be used to build and run the individual backend or frontend components:

```
# backend/
$ docker buildx build -t taw/backend .
$ docker run --rm -p 6969 taw/backend

# frontend/
$ docker buildx build -t taw/frontend .
$ docker run --rm -p 80 taw/frontend
```

To automatically build and run all required components use:

```
$ docker compose up --build
```

This command will spin up 3 containers:

- `frontned`: nginx container serving the frontend;
- `backend`: container running the node backend;
- `mongodb`: container running MongoDB;

The container's IPs are respectively: `172.22.0.2`, `172.22.0.3`, and `172.22.0.4`, but a domain mapping the container's name and it's IP will be set up in the docker network, so the backend will be able to connect to the mongo container by connecting to `mongodb:27017`.

3.2 Backend

3.2.1 Authentication - JWT. Authentication by the backend is handled throught Javascript Web Tokens.
The following snippets is an example of how to create a middleware that will verify if the `Authorization` header contains a valid JWT.

```
const mustAuth = (
    req: Request,
    res: Response,
    next: NextFunction
) => {
    const header = req.headers.authorization;

    if (header) {
        const [_, token] = header.split(" ");
        jwt.verify(token, secret, (err, user) => {
            // ... error handling ...
            req.body.username = user.username;
            next();
        });
    } else {
        // ... error handling, return 400 ...
    }
};
```

3.2.2 Data Model. In this project MongoDB has been used as a data store for user and match data, statistics and user-to-user messages.
The following is a snippet of code levaraging *mongoose* to abstarct away all CRUD boilerplate.

```
const UserSchema = new Schema({
  name:         { type: String },
  username:     { type: String },
  email:        { type: String },
  password:     { type: String },
  wins:         { type: Number },
  looses:       { type: Number },
  matches:      { type: Number },
  isModerator:  { type: Boolean },
  isFirstLogin: { type: Boolean },
  friends:      { type: [Types.ObjectID] }
});
```

Other models are `Match` and `Message`, of which the complete definition can be found respectively in `backend/model/match.js` and `backend/model/message.js`.

3.2.3 REST API.

- `/signup`
  - Method: `POST`
  - Params: `name`, `username`, `email`, `password`
  - Description: create a new user.
- `/signin`
  - Method: `POST`
  - Params: `username`, `password`
  - Description: log the user in.
- `/logout`
  - Method: `GET`
  - Description: log the user out.
- `/user`
  - Method: `GET`
  - Params: `role`
  - Description: return users with the specified role.
- `/user/:username`
  - Method: `POST`
  - Params: `moderator`, `username`
  - Description: delete a user profile.
- `/friend`
  - Method: `GET`
  - Description: return the current user's friends.
- `/friend/:username`
  - Method: `PUT`
  - Params: `username`
  - Description: add a friend.
- `/friend/:username`
  - Method: `DELETE`
  - Params: `username`
  - Description: remove friend.
- `/match/:id`
  - Method: `GET`
  - Params: `id`
  - Description: return information about the match with the given `id`.
- `/firstLogin`
  - Method: `POST`

  - Params: `username`, `password`, `name`, `email`
  - Description: by default, the moderator's credentials are `admin:admin`. When logging in the first time the moderator is required to change the default credentials. In the future this should be refactored and merged into a single endpoint with `/signin`.
- `/moderator`
  - Method: `PUT`
  - Params: `username`, `password`
  - Description: create a moderator account with the given credentials. In the future we should be able to change a user's role without having to create an extra account.
- `/matches`
  - Method: `GET`
  - Description: return all maches being currently played.
- `/history/:username`
  - Method: `GET`
  - Description: return all matches a user has played.
- `/chat/:username`
  - Method: `PUT`
  - Params: `username`, `message`
  - Description: send a message to the recipient (`username`). recipient must but a friend.
- `/chat/:username`
  - Method: `POST`
  - Params: `username`
  - Description: return all messages between two current user and (`username`).
- `/chat/:username/read`
  - Method: `GET`
  - Description: mark messages as read.

Currently the chat data is contained in MongoDB, but was originally stored in Elasticsearch.

3.2.4 ~~Chat - Elasticsearch.~~ ~~There exists a wide range of possible implementations for a user-to-user chat service. The main features of such are the following:~~

- ~~*persistance* (or lack there of) which indicates wether the messagges are persisted server-side and can be fetched more than once (e.g. Telegram) or if it's up to the client to store them locally (e.g. IRC);~~
- ~~*push/pull* which indicates whether the client must ask for updates (*pull*), or is notified by the server (*push*). *pull* systems usually involve a call to an API that returns any new data, while *push* systems take advantage of websockets.~~

~~For out implementation, mainly due to time constraints we have opted for a persistent pull-based system. All messages are stored in an ElasticSearch single node cluster with the following schema.~~

```
[
    {
      "from": <USER_ID>,
      "to": <USER_ID>,
      "message": <MESSAGE>,
    },
    ...
]
```

~~The following is an example implementation of the `/chat` endpoint, which returns all messages belonging to a chat between the logged-in user and the one specified via request parameters.~~

```
app.get('/chat', mustAuth, async (req, res) => {
    await new ChatClient(req.user.id)
        .get(req.from)
        .then(items => res.json(items))
});
```

~~**Further discussion**~~:

- ~~indexing strategies for text-based search;~~
- ~~group chat implementation;~~

3.3 Frontend

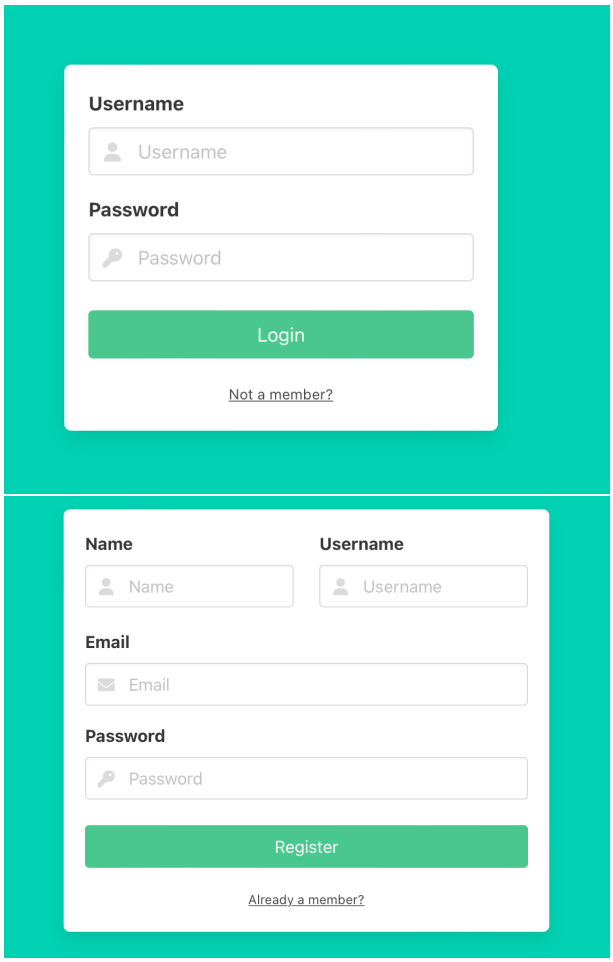

Figure 2: When connecting to the game, a login page is shown. From here the user can either enter their credentials and logi in, or sign up if they don't have an account already. In case the input data is incorrect, a toaster is show.

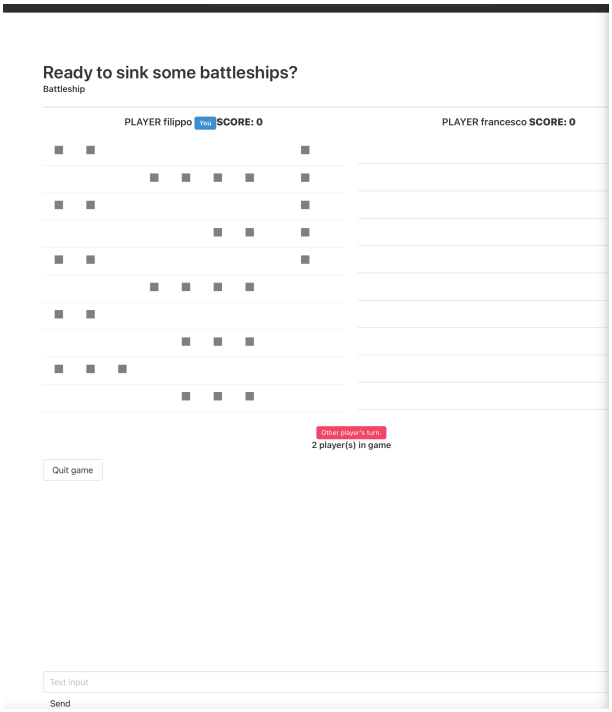

Figure 5: When starting a match, first the player is able to place it's ships. Once both players have played their ships and have confirmed, the game starts.

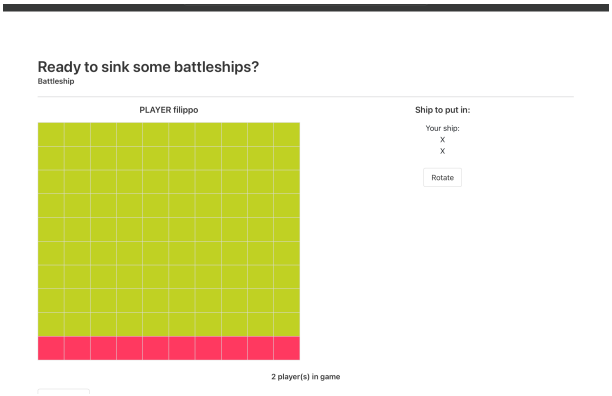

Figure 6: While the game is being played, the opponent field is shown, with green squares indicating unknown tiles, and red ones indicating empty ones.