

BATTLESHIP

INHALT

1. Allgemein	2
2. Architektur	3
3. Backend	4
3.1 DB	4
Dateistruktur	4
Pod Erstellung	4
3.2 API	5
Dateistruktur	5
Spiellogik	5
Datenbankanbindung	5
Pod Erstellung	6
4. Frontend	7
4.1 Load Balancer	7
Pod Erstellung	8
4.2 Web	9
Webseite	9
Dateistruktur	10
Pod Erstellung	10
5. Sockets	11

1. ALLGEMEIN

Als Challenge-Task für das Modul Distributed Systems haben wir das bekannte Spiel Battleship implementiert. Ziel der Challenge-Task war es, ein verteiltes System zu entwerfen und zu implementieren mit mindestens einem Service, dass über einen Websocket kommuniziert. Dabei kann ein Service ausfallen, ohne dass das ganze System nicht mehr funktioniert.

Insgesamt galten folgende Voraussetzungen:

1. Load Balancing mit skalierenden Services
2. Failover einer Service-Instanz
3. Ein Websocket muss Daten vom Service an das Frontend senden
4. Dockerized
5. Persistenter Speicher
6. Die Verwendungen von aktuellen Libraries und Frameworks

Battleship (bekannt als «Schiffe versenken») ist ein Strategie-Ratespiel für zwei Spieler. Es wird auf Gittern gespielt, auf denen verschiedene Kriegsschiffe von jedem Spieler platziert werden, die dem Gegenüber verborgen bleiben. Die Spieler wechseln sich ab und rufen "Schüsse" auf die Schiffe des anderen Spielers, und das Ziel des Spiels ist es, die Flotte des gegnerischen Spielers zu zerstören. In dieser Implementation spielt man gegen das System. Die Kriegsschiffe wurden nach den Regeln von Hasbro aus dem Jahre 2002 definiert.

Nr.	Klasse des Kriegsschiffs	Grösse (Felder)
1	Carrier	5
2	Battleship	4
3	Destroyer	3
4	Submarine	3
5	Patrol Boat	2

2. ARCHITEKTUR

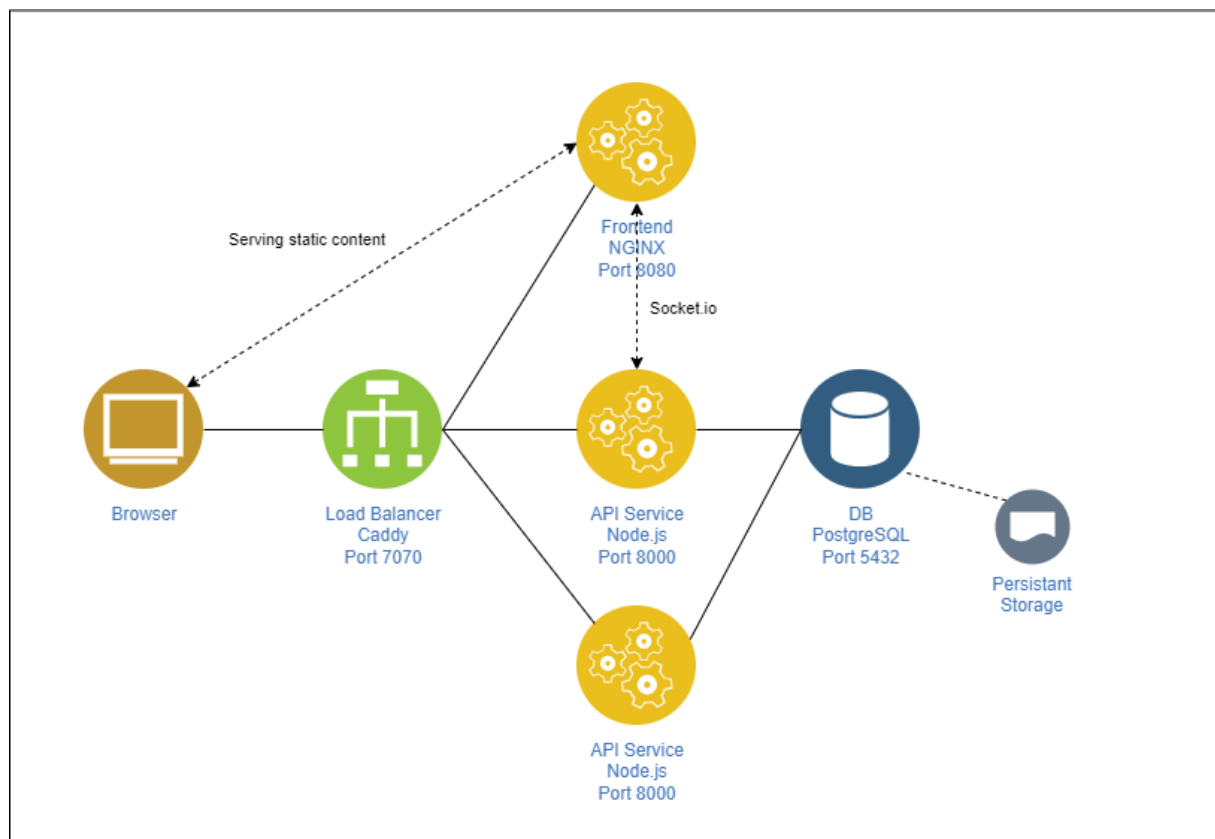
Die Infrastruktur von Battleship ist in (mindestens) fünf Docker Container unterteilt.

- Ein Caddy Load Balancer zur Lastenverteilung und als Proxy.
- Ein NGINX Container zur Auslieferung der statischen Frontend-Daten.
- Zwei Node.js API Container, welche die Spielzüge entgegennehmen, verarbeiten und an die Datenbank zur Speicherung weiterleiten.
- Ein PostgreSQL Container zur persistenten Speicherung der Spielsituation.

Die ganze Infrastruktur ist dockerized und wird mit docker-compose gestartet.

Der Befehl dazu lautet:

`docker-compose up`



3. BACKEND

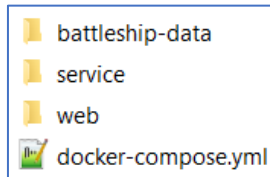
3.1 DB

Bei der Datenbank handelt es sich um ein normales Postgres Image, in das wir mit Umgebungsvariablen den Benutzer und den Datenbanknamen einfügen.

Zusätzlich erstellen wir die Tabelle für unser Spiel mittels SQL-Datei, die beim Erstellen des Pods ausgeführt wird.

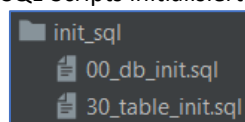
Den Ablageort der Datenbank geben wir in docker-compose mit, und lassen die Daten ins Dateisystem vom Host schreiben. Damit überlebt es auch das Löschen und Neuerstellen des Datenbank Pods.

Die Datenbank wird auf dem Docker Host gespeichert. Unser Battleship-Ordner beinhaltet entsprechend den Ordner für die Datenbank, das Repository für den API (Service) Server, das Repository für die statischen Frontend-Daten und das docker-compose File.



DATEISTRUKTUR

Die Datenbank wird initial mit zwei simplen SQL-Scripts initialisiert.



POD ERSTELLUNG

Das Image von PostgreSQL wird direkt vom Docker Hub gezogen.

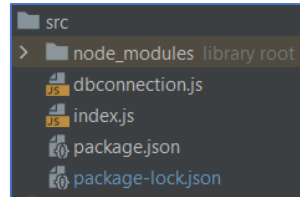
Auszug aus docker-compose.yml

```
db:
  image: postgres
  ports:
    - "5432:5432"
  restart: on-failure:3
  environment:
    POSTGRES_DB: battleship
    POSTGRES_USER: battleuser
    POSTGRES_PASSWORD: jkwhg8okertg8o73
  volumes:
    - ./service/init_sql:/docker-entrypoint-initdb.d
    - ./battleship-data:/var/lib/postgresql/data
```

3.2 API

DATEISTRUKTUR

Der API-Server besteht aus einer Datei für die Spiellogik und einer für die Kommunikation mit der Datenbank.



Wir verwenden die, zum Zeitpunkt der Dokumentation, aktuellen Module.

```
"dependencies": {  
  "express": "^4.18.0",  
  "http": "^0.0.1-security",  
  "pg": "^8.7.3",  
  "socket.io": "^4.5.0"  
},
```

SPIELLOGIK

Das Spielfeld wird als eindimensionaler String Array implementiert, auf dem wir die vertikalen Schiffspositionen entsprechend umrechnen. Im Array wird der Status der Felder gespeichert, damit wir wissen, ob es frei, besetzt, beschossen oder gar getroffen ist.

Die Spielzüge werden mit Sockets übertragen, deren Ausführung ab Seite 11 zu finden ist.

DATENBANKANBINDUNG

Sobald ein Spieler seinen Angriff abschickt, lädt der API-Server das aktuelle Spiel aus der Datenbank, um bei einem allfälligen API-Failover die fehlerlose Weiterführung des Spiels zu gewährleisten. Im Anschluss wird der Angriff ausgewertet, der Spielzug des Computergegners vollzogen und den neuen Spielstand in die Datenbank geschrieben, bevor diese Aktionen dem Spieler zurückgesendet werden. Wir stellen damit sicher, dass ein Spiel immer alle dem Spieler gemeldeten Aktionen gespeichert hat, auch wenn es zu einem Absturz kommt. Der API-Server ist stateless und speichert somit keine Daten im eigenen Container.

```
export async function getGame(playerName) {  
  let dbGame = {};  
  await pool.query('SELECT * FROM game WHERE player_name = $1', [playerName])  
    .then(res => {  
      if(res.rows.length > 0){  
        dbGame = {  
          // mixed method  
          playerGrid:      cleanUploadedArrayString(res.rows[0].player_grid),  
          opponentGrid:    cleanUploadedArrayString(res.rows[0].opponent_grid),  
          unmarkedPlayerSquares: cleanUploadedArrayString(res.rows[0].unmarked_player_squares),  
          playerShips:      JSON.parse(res.rows[0].player_ships),  
          opponentShips:    JSON.parse(res.rows[0].opponent_ships),  
        };  
      } else {  
        dbGame = {  
          empty: true,  
          playerGrid: Array(100).fill( value: '' ),  
          playerShips: null,  
          opponentGrid: Array(100).fill( value: '' ),  
          opponentShips: null,  
          unmarkedPlayerSquares: Array.from(Array(100).keys()),  
        };  
      }  
    })  
    .catch(err => {  
      console.log(err.stack);  
    });  
  return dbGame;  
}
```

POD ERSTELLUNG

Dockerfile

```
FROM node:alpine
WORKDIR /service
COPY src/ .
RUN npm install
CMD ["npm", "start"]
EXPOSE 8000
```

Auszug aus docker-compose.yml

```
api:
  build: ./service/
  ports:
    - "8001-8002:8000"
  deploy:
    replicas: 2
  restart: on-failure:3
  environment:
    - CONNECTION_STRING=postgres://battleuser:jkwhg8okertg8o73@db/battleship
  depends_on:
    - db
```

4. FRONTEND

4.1 LOAD BALANCER

Der Reverse-Proxy von Caddy 2 wird als Load Balancer für den Traffic verwendet. Dieser gewährleistet zwei wichtige Aspekte:

- Das Load Balancing zwischen mehreren skalierenden Services
- Das Failover einer Service-Instanz

Sowohl der Webserver als auch die API-Services befinden sich hinter dem Load Balancer. Dieser hat den Port 7070 und leitet anhand von der URL den Request zur richtigen Instanz. Requests auf das Root-Verzeichnis werden zum Webserver weitergeleitet. Requests auf den Pfad /api werden zu den verschiedenen API-Services weitergeleitet. Caddy verteilt die Sessions zufällig an die API-Services, basierend auf Cookies..

Um zu überprüfen, welche Instanzen des API-Service verfügbar sind, werden durch Caddy aktive Healthchecks ausgeführt über die URL /healthcheck. Dies wird verwendet, um bei einem Absturz eines Service das Failover einzuleiten, damit die restlichen Services dessen Traffic übernehmen können.

Das Caddyfile ist wie folgt aufgebaut:

- Der Port des Load Balancers lautet 7070. Mit **reverse_proxy** wird ein Proxy für die angegebene Direktive erstellt. In diesem Fall eine für /api und eine für *, welche alle üblichen Requests zum Webserver weiterleitet.
- Mit **rewrite** wird ein interner Redirect von /api auf /api/ ausgeführt.
- Mit **handle /api/*** wird die Gruppe der API-Direktiven eigenständig evaluiert und von anderen Handle-Blocks ausgeschlossen
- **uri strip_prefix** – entfernt innerhalb vom Proxy den /api Prefix, da die Services es nicht kennen und nur für den Ingress von Caddy benötigt wird.
- **header_up** definiert, wie die Header der Requests behandelt werden sollen. Mit Host {host} und X-Real-IP {remote} sollen Source und Destination unverändert bleiben und kein Forward Header vorhanden sein.
- Mit **to** werden alle Service-Instanzen angegeben, auf welche die Last verteilt werden soll.
- **lb_policy cookie** regelt die Load Balancing Policies über Cookies. Beim Request eines Clients wird zufällig ein Service ausgewählt, auf dem die Session persistieren soll (Sticky Session). Socket.io ist stateful und hat Probleme, wenn ständig der Request auf einem neuen Service landet. Dabei wird durch den angegebenen Namen *mycookie* ein Hash als Identifikation erstellt, das durch den *set-cookie* Header vom Client bei jedem Request mitgesendet wird. So weiss der Load Balancer, an welchem Service der Request weitergeleitet werden muss. Bei einem Absturz wird ein neuer Service für die Session gewählt.
- **health_uri** definiert den Pfad an für die aktiven Healthchecks.
- **health_interval** definiert wie oft aktive Healthchecks ausgeführt werden.
- **health_timeout** definiert wie lange gewartet werden muss, bis die Instanz als «down» markiert wird.
- **health_status** definiert den HTTP-Status, welcher der Instanz zurückgegeben werden muss, um als «up» markiert zu werden.

```
#Caddyfile
:7070
rewrite /api /api/
handle /api/* {
  uri strip_prefix /api
  reverse_proxy * {
    header_up Host {host}
    header_up X-Real-IP {remote}

    to http://dsy_api_1:8000
    to http://dsy_api_2:8000
    to http://dsy_api_3:8000

    lb_policy cookie mycookie
```

```
health_uri /healthcheck
health_interval 2s
health_timeout 5s
health_status 200
}
}

reverse_proxy * {
  to http://dsy_web_1
}
```

POD ERSTELLUNG

Auszug aus docker-compose.yml

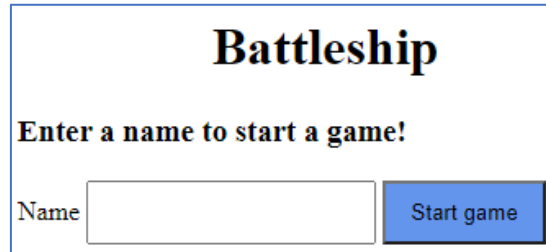
```
lb:
  image: caddy
  ports:
    - "7070:7070"
  volumes:
    - ./web/Caddyfile:/etc/caddy/Caddyfile
```

4.2 WEB

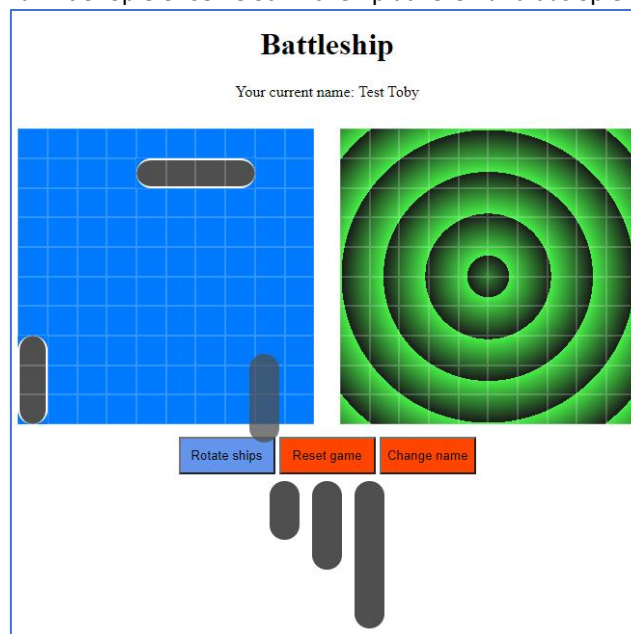
Das Frontend besteht aus statischen Dateien, die mit NGINX ausgeliefert werden. Die Kommunikation mit dem Backend findet ausschliesslich über Sockets statt, welche später beschrieben werden.

WEBSEITE

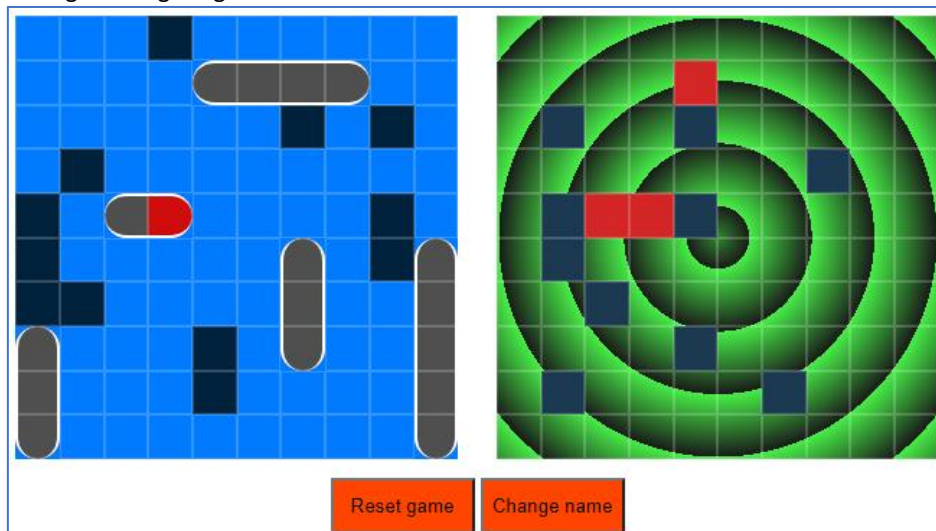
Ein Benutzer kommt dabei als erstes ein Eingabefeld für seinen Spielernamen zu sehen. Unter diesem Namen wird sein Spiel anschliessend in der Datenbank gespeichert.



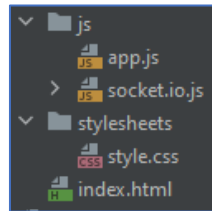
Nach der Namenseingabe kann der Spieler seine Schiffchen platzieren und das Spiel beginnen.



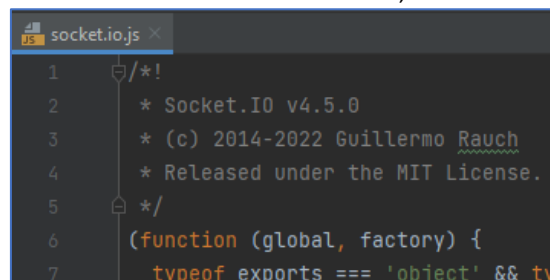
Während dem Spiel kann der Spieler im rechten Spielfeld seine Angriffe ausführen, während ihm im linken Feld die Treffer des Gegners angezeigt werden.



DATEISTRUKTUR



Die socket.io.js Datei, ist die Einzige, die wir nicht selbst geschrieben, sondern vom socket.io Projekt übernommen haben. Es handelt sich dabei um die aktuelle Version, wie der Kommentar in der Datei zeigt.

A screenshot of a code editor showing the content of 'socket.io.js'. The file starts with a multi-line comment block containing the Socket.IO version (v4.5.0), copyright information (2014-2022 Guillermo Rauch), and the MIT license. Below the comment, a function is defined: `(function (global, factory) {`. The next line shows a type check: `typeof exports === 'object' && t`.

POD ERSTELLUNG

Dockerfile

```
FROM nginx:latest
COPY ./src /usr/share/nginx/html
EXPOSE 8080
```

Auszug aus docker-compose.yml

```
web:
  build: ./web/
  ports:
    - "8080:80"
  deploy:
    replicas: 1
  restart: on-failure:3
```

5. SOCKETS

Für die Kommunikation zwischen Frontend und Backend nutzen wir Socket.IO. Wir haben dabei 8 Events erstellt, auf die die Sockets hören:

1. **login:**
Wird vom Frontend ausgelöst, sobald sich der Spieler mit seinem Benutzernamen angemeldet hat. Der Server lädt daraufhin den Spielstand aus der Datenbank (falls dieser vorhanden ist) und löst loadGame aus.
2. **loadGame:**
Wird vom Backend verwendet, um den aus der Datenbank geladenen Spielstand an das Frontend zu senden. Das Frontend entscheidet basierend auf dem erhaltenen Objekt, ob ein neues Spiel oder ein altes geladen werden soll.
3. **restart:**
Wird vom Frontend ausgelöst, sobald der Spieler auf den Reset Game Knopf drückt. Das Backend setzt alle Informationen über den Spielstand zurück, so dass ein frisches Spiel gestartet werden kann.
4. **placeShips**
Wird vom Frontend ausgelöst, sobald der Spieler auf den Start Game Knopf drückt. Dieser kann erst betätigt werden, wenn alle Schiffe gesetzt sind. Das Frontend übermittelt die Platzierung der Schiffe an das Backend, welches diese dann im Spielstand auf der Datenbank speichert.
5. **playerAttack:**
Wird vom Frontend verwendet, um ans Backend zu übermitteln, auf welches Feld der Spieler geklickt hat. Das Backend berechnet dann, ob es sich um einen Treffer handelt oder ob der Spieler verfehlt hat. Zusätzlich wählt das Backend ein zufälliges, nicht-getroffenes Feld auf der Spielerseite um einen Gegenangriff zu tätigen.
6. **playerAttackResult:**
Wird vom Backend verwendet, um auf playerAttack zu antworten. Es werden folgende Informationen gesendet:
 - a. Das Feld, auf das der Spieler gezielt hat.
 - b. Ob der Angriff des Spielers erfolgreich war.
 - c. Das Feld, auf das der Gegner gezielt hat.
 - d. Ob der Angriff des Gegners erfolgreich war.Daraufhin nimmt das Frontend diese Informationen entgegen und markiert die entsprechenden Felder als Treffer bzw. Fehlschuss.
7. **requestPlayerName:**
Wird vom Backend benutzt, um bei Spielstart den Spielernamen vom Spieler anzufordern.
8. **playerName:**
Wird vom Frontend benutzt, um den Spielernamen an das Backend zu senden.

Da Socket.IO stateful ist, nutzen wir auf dem Loadbalancer die bereits erwähnten Cookies, damit wir die Sockets nicht über alle API Server synchronisieren müssen.