Distributed Systems Project Documentation

Distributed Systems Project

Semester: Spring 2023

Version: 02.00
Date: 2023-05-21 21:29:33+02:00

Project Team: Dario Glasl
Jasmin Fitz
Samuel Maissen
Mirio Eggmann
Nick Wallner

Project Advisor: Thomas Bocek

School of Computer Science
OST Eastern Switzerland University of Applied Sciences

Contents

Part I

Product Documentation

Chapter 1

Introduction

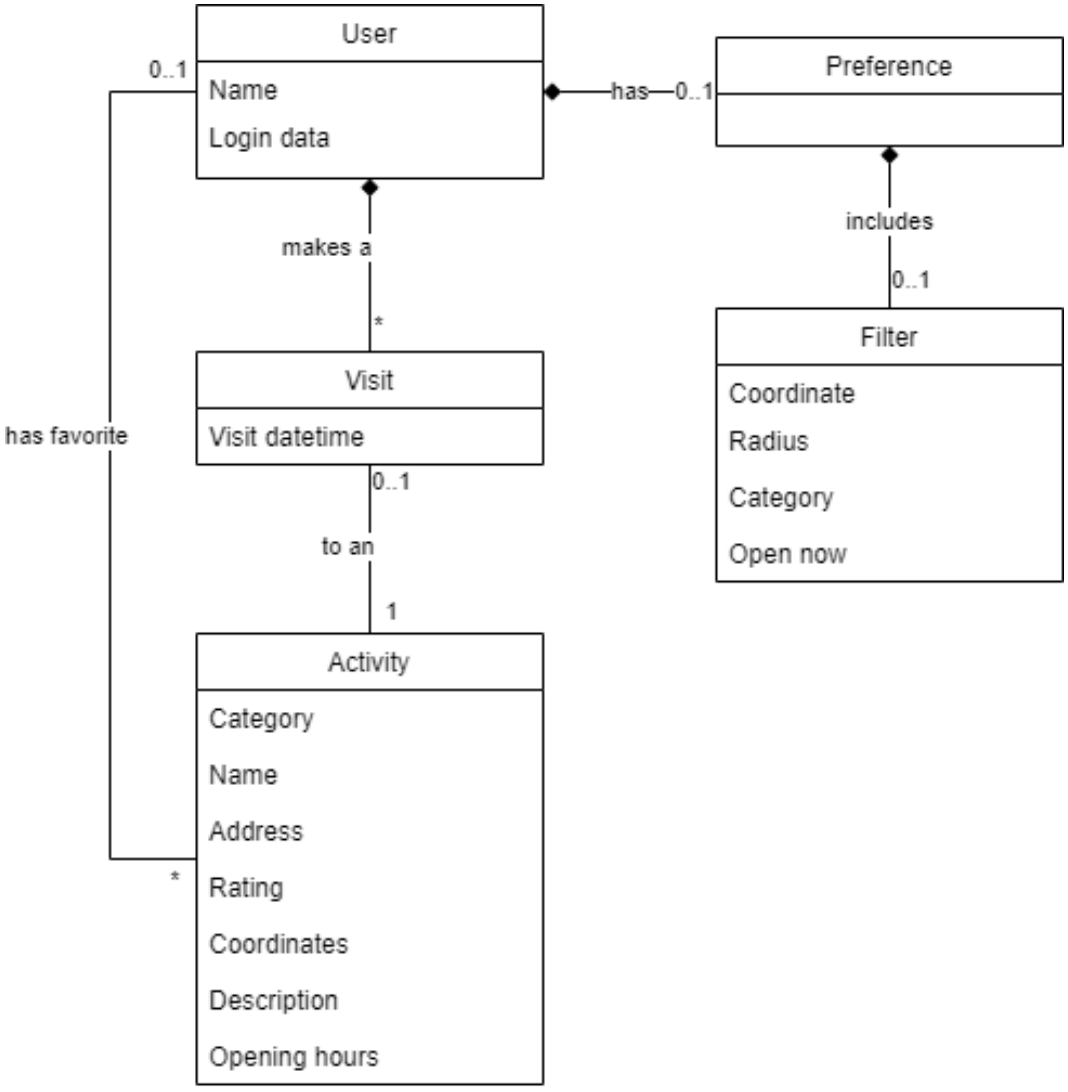
1.1 Vision

The app is called "Spontaneous" and it helps users find fun things to do and nice restaurants on the fly. With a simple interface some basic criterias can be choosen, in order to get a random propose for an activity or a place to eat. The app should make it easy to try something new with the condition the user aswell using it as considered.

Chapter 2

Domain Analysis

2.1 Domain model



2.1.1 User

A user can mark several activities as favorites.

2.1.2 Visit

A visit is made by a user, at a specific time. This represents the execution of an activity.

2.1.3 Activity

An activity is always generated anew and is the result when using the Spontaneous app. The reason for this is that activities are sourced from an external entity so that the data is always up to date. Because of that (even if this activity has already been generated once), there can only be one visit to an activity.

It does not have to be marked as a favorite. Furthermore, an activity can also be suggested without a user being logged in. This results in the fact that no visit can be made either.

2.1.4 Preference

A user's preferences include information that is necessary to provide a specific user with permanent, individual setting options, such as preset filter options for generating activities.

2.1.5 Filter

Filters are used to find activities better. They can be saved per user in the preferences.

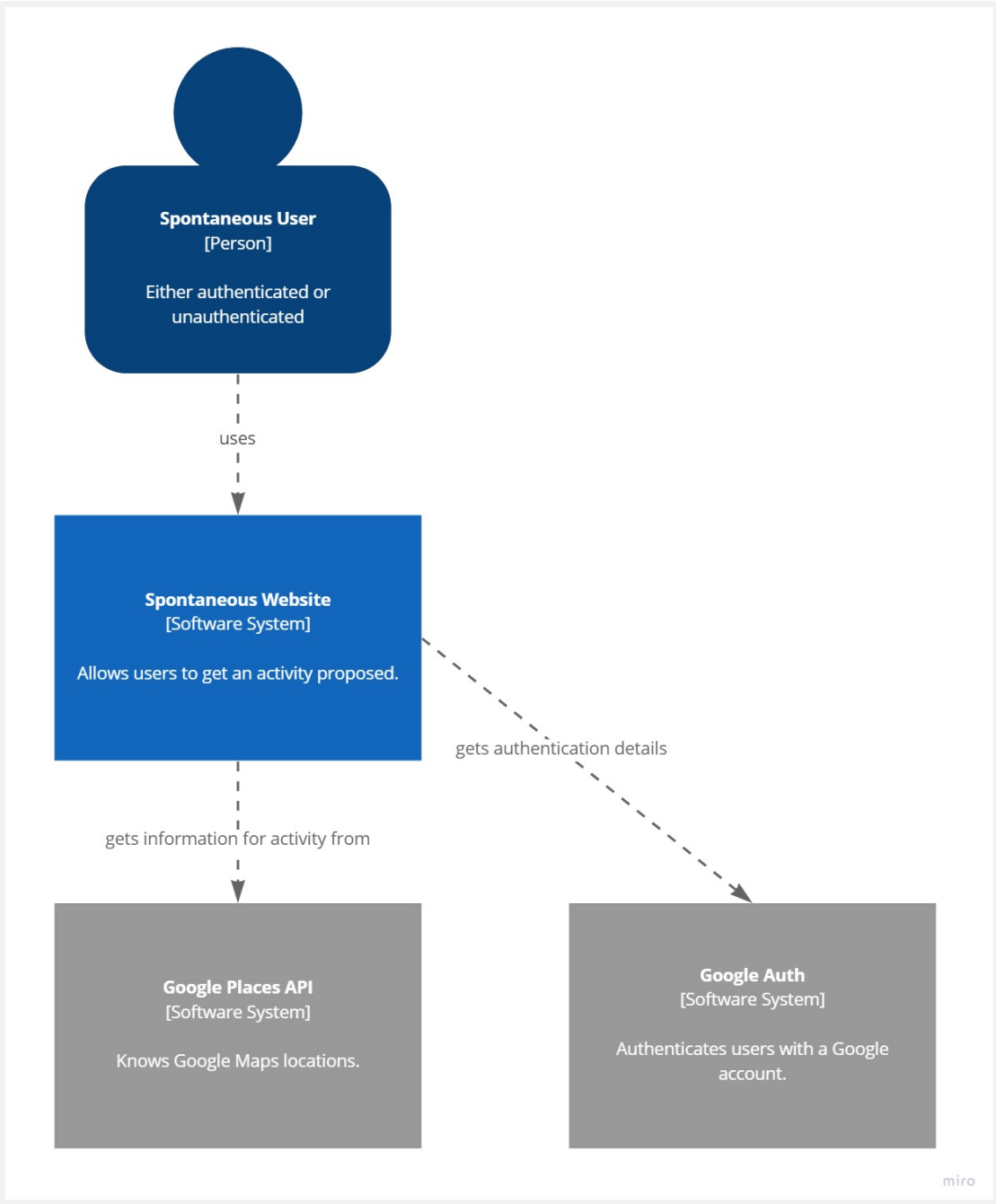
Chapter 3

Architecture

3.1 Architecture Diagrams C4

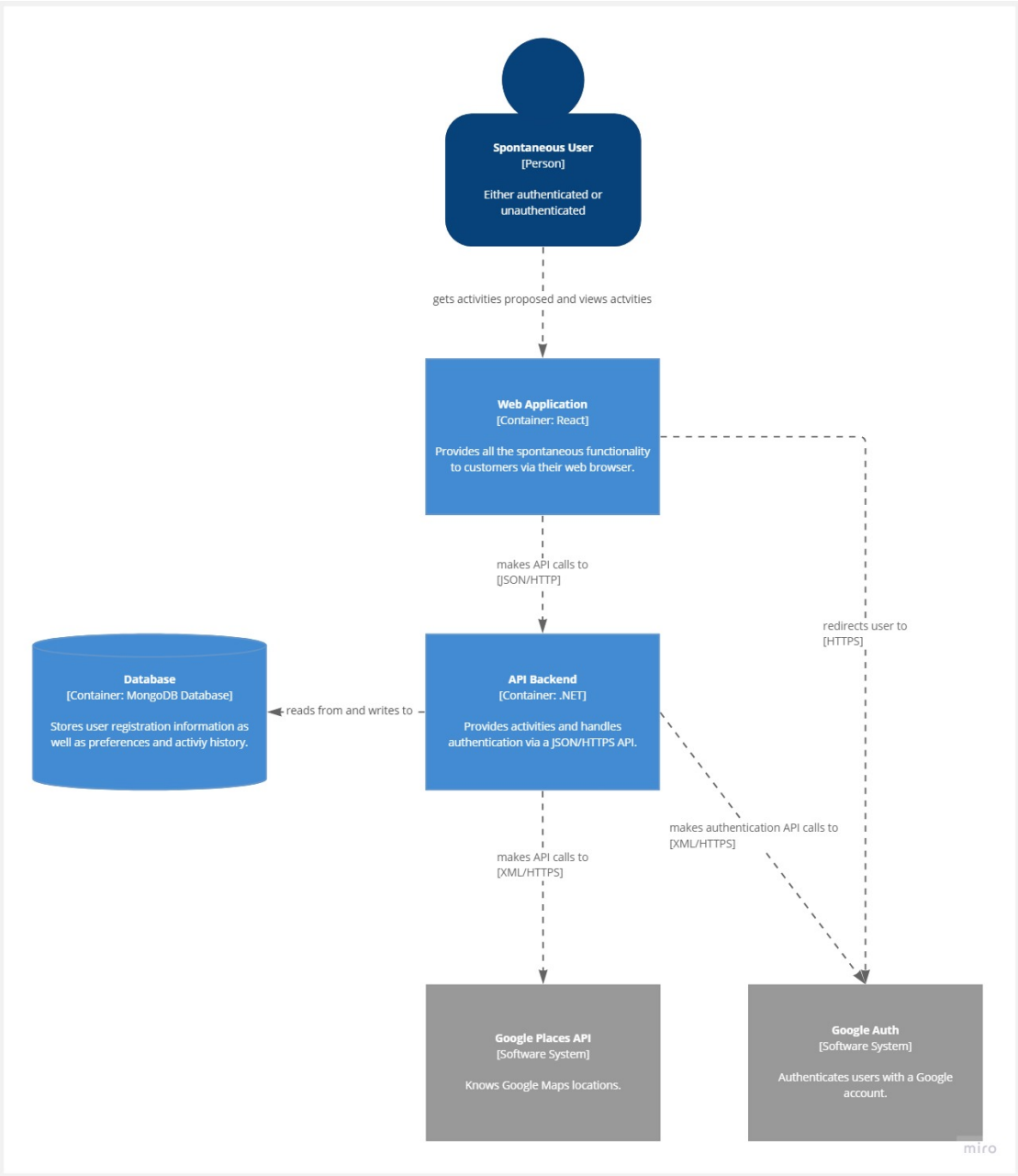
We defined the architecture in the C4 standard for context and container only because of the components and code beeing to volatile and better described by the architecture pattern above.

3.1.1 C4 Context Diagram



The user doesn't have a direct line to the google auth software system, because it isn't necessary for our context.

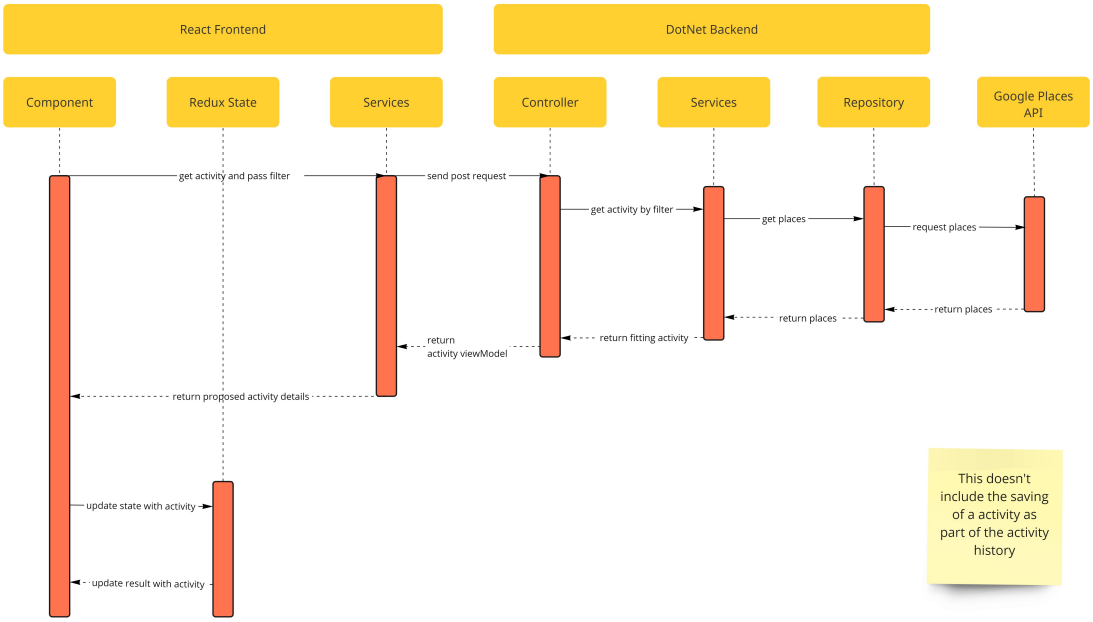
3.1.2 C4 Container Diagram



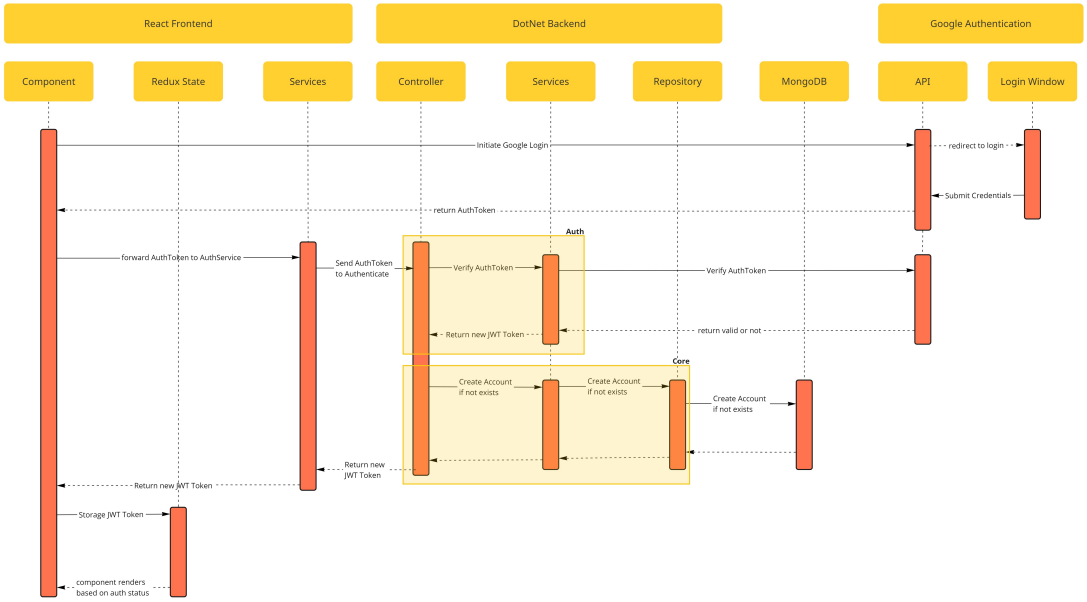
3.2 Sequence Diagrams

The following diagrams show a dynamic view to the most important features.

3.2.1 Activity Proposal Sequence Diagram



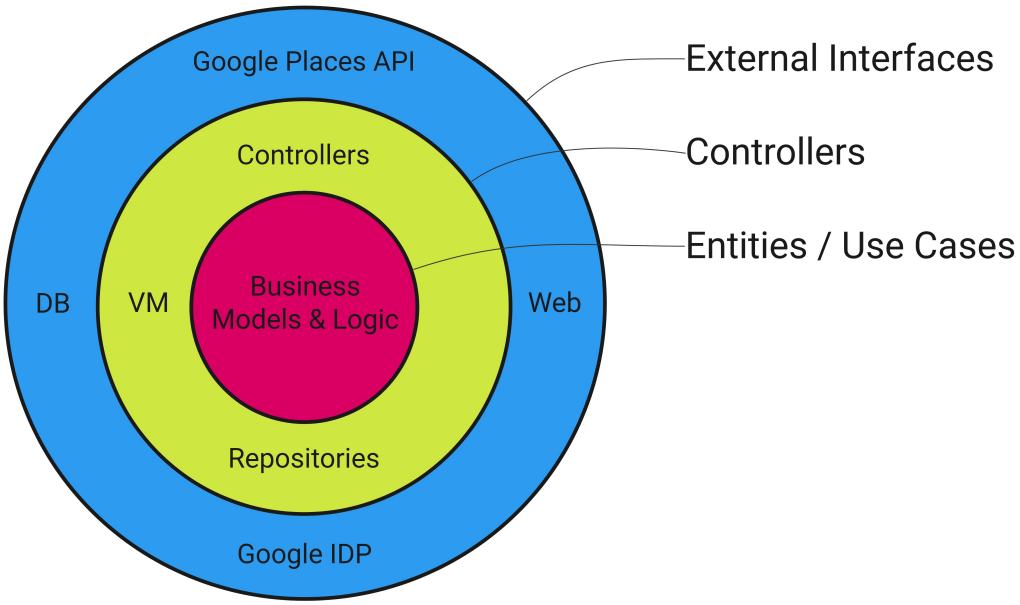
3.2.2 JWT Authentication Sequence Diagram



3.3 Backend Architecture

3.3.1 Architectural Pattern

Based on the Clean Architecture diagram, we built our architecture. In contrast to the original structure of Robert C. Martin, we have merged the Entities and Use Cases layers, since the size of our project does not require such a separation of Enterprise and Application.

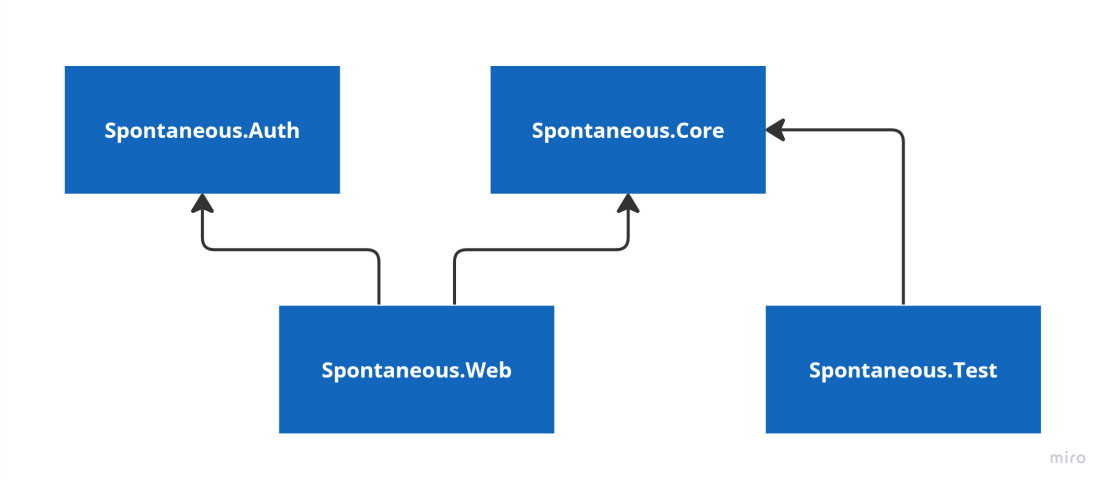


(a) The Clean Architecture Approach

The abbreviation VM stands for View Models.

3.3.2 Architectural structure

The backend is developed with the programming language C# in the version .NET 7. It can be found in the "Spontaneous.sln" solution in the "Backend" folder. The solution is divided into different projects. This makes the individual components easier to change and replace with newer technologies. In this way, the dependencies are clearly defined.



Core

The core project will contain most of the code. This includes the essential logic to make the application work, this will be handled in so-called "services", as well as database access and requests to the Google Places API. These are managed in so-called "repositories". Repositories differ from services in that they are only responsible for fetching data in its original form. The processing then takes place in the services. Business objects are modeled here.

We deliberately decided against another project with only database accesses, as this would have added more overhead than it would have benefited in the context of our SE project. That's why database accesses are integrated in the core project in the form of repositories.

Auth

The Auth project includes all the authorization and authentication mechanisms.

Web

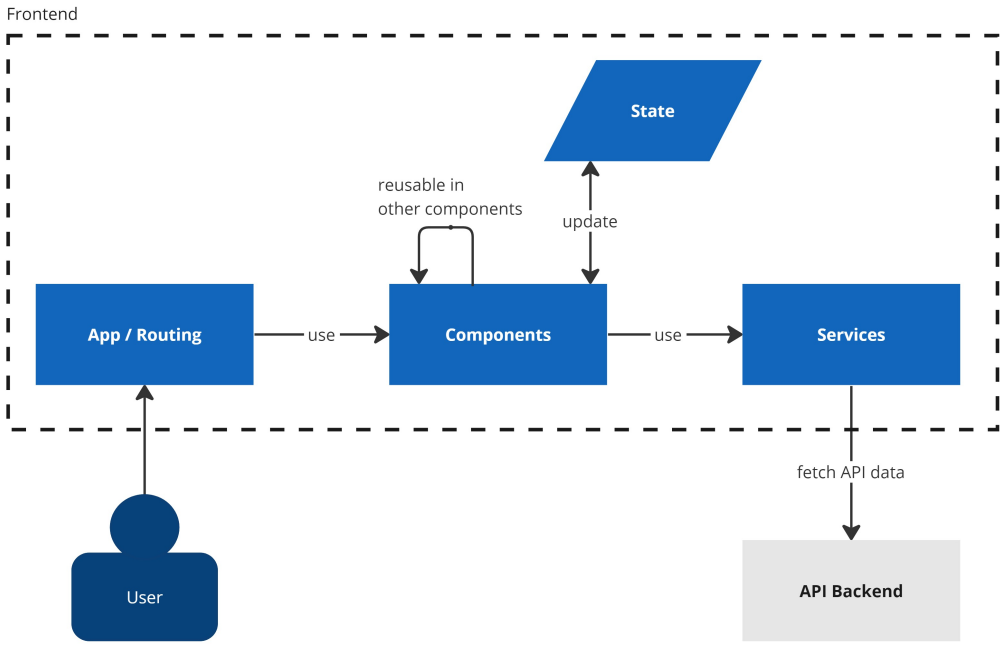
The API which communicates with the frontend is in the project "Web". This contains only little code which is responsible for the mentioned communication. Business objects are converted into view models here.

Test

In the "Test" project, XUnit is used to cover important elements of the "Core" project using unit tests.

3.4 Frontend Architecture

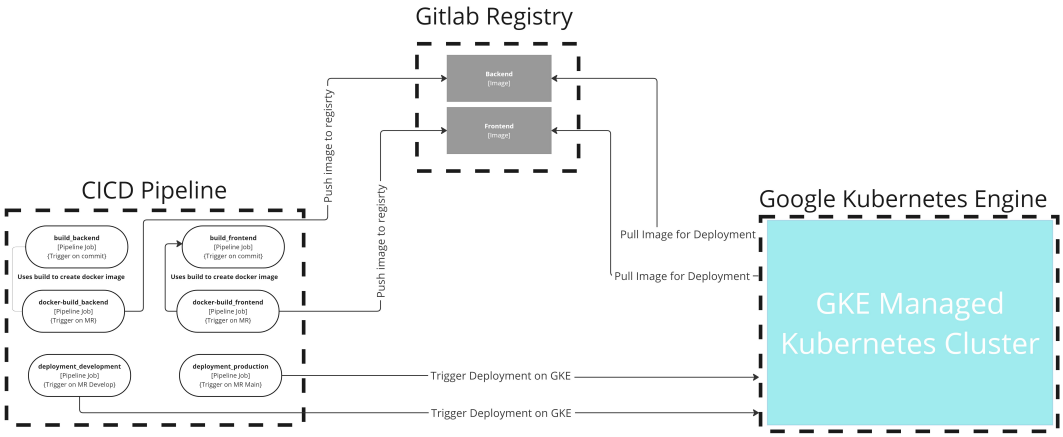
In the frontend, we use React.js. This diagram show the general structure of the frontend. We simplified the state because Redux manages the state, see this Redux Architecture. By state we mean the data that a component uses and updates through hooks.



We don't have a clean architecture diagram for the frontend because we are bound to the technology architecture.

3.5 Pipeline / Deployment Architecture

This diagram only shows the jobs required for the deployment. Other jobs for example testing and linting are not concluded in this diagram. The pipeline concludes 6 jobs: 2 build jobs (frontend, backend), 2 docker build jobs which create the Docker images via Dockerfile (frontend, backend) and 2 deploy jobs that deploy the containerized workload to Kubernetes (development, production).



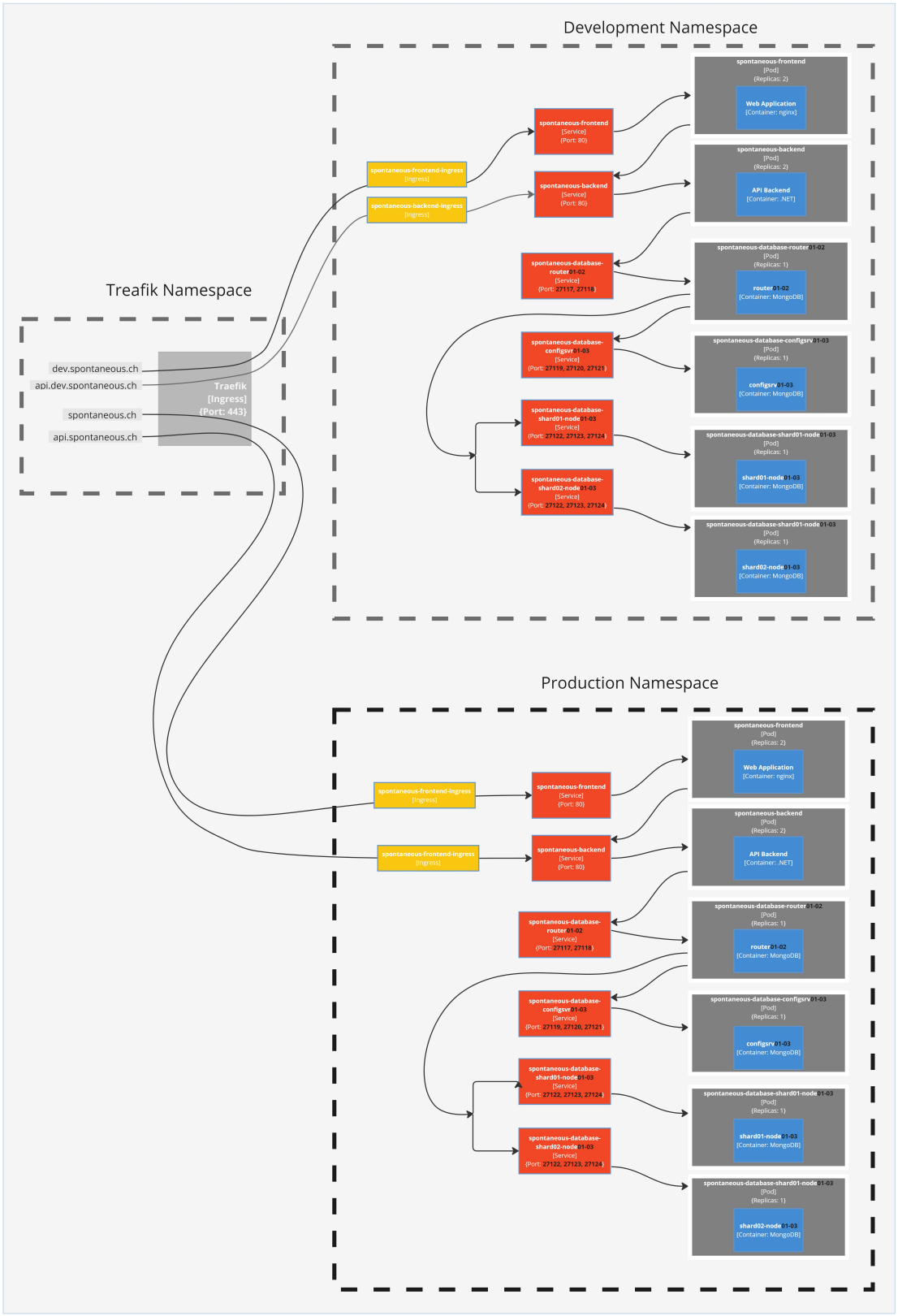
3.6 Infrastructure Architecture

Following the Kubernetes deployment is visualized. It concludes all important components which are necessary for our application. The Kubernetes deployment runs on the Google Kubernetes Engine. Since it's a managed cluster, the architecture of the cluster itself is not relevant and as well not visualized further. The important and relevant is the deployment.

3.6.1 Database

The Database deployment is quite complex, since it's a sharded database which is scalable and high available. Since drawing all the instances on the diagram would overfill it, we decided to pull together the same instance types and name it with the count-suffix "01-03". The whole sharded database consits of following instances:

- 2x Database Router: Responsible for receiving requests and redirecting it to right shard.
- 3x Config Servers: Store the information about which shard is stored on which nodes.
- 3x Nodes for shard01: store the actual data for the specific shard01
- 3x Nodes for shard02: store the actual data for the specific shard02



GKE Managed Kubernetes Cluster

3.7 Technology Decisions

Most technologies were chosen because we had the most experience with them and they were the most popular and therefore best documented.

- We use a lot of different technologies, so we don't have to reinvent the wheel and can focus on the core features of our application.

3.7.1 Frontend

- **Typescript**, we simply wanted types and use JavaScript in a more structured and clean way.
- **React**, best known by us and most popular framework.

  Alternatives: Angular, VueJS, .Net MAUI, Blazor
- **Sass**, we wanted to use a CSS preprocessor for better code reuse and better readability.
- **Material UI**, we wanted to use a UI framework for faster development and better design and responsiveness.

  Alternative: Bootstrap
- **Webpack**, used for bundling and minifying the frontend files.
- **NPM**, used for managing frontend dependencies.
- **Axios**, used for making HTTP requests to the backend.
- **Redux**, used to store state used in different components.
- **Analytics**, if we would go live we would use Piwik or Google analytics for analytics and improving usability.

3.7.2 Backend

- **.Net 7**, ASP .NET Core Web API, we wanted to use a framework which is well documented and has a lot of support.

  Alternatives: Java, NodeJS
- **Swagger**, is really useful for testing and documenting the API.
- **MongoDB lib** (mongodb.bson), nuget package for accessing MongoDB.
- **Google API lib**, nuget package for Google API models.

3.7.3 Infrastucture

- **Traefik**, we use Traefik for load balancing and proxying. We didn't use Nginx because we wanted to use HTTP/3 which isn't supported by Nginx yet.

  Alternatives: Nginx, Caddy, HAProxy, Envoy
- **Nginx**, used as a web server for hosting the static frontend files.
- **Kubernetes**, used for container orchestration and scaling.
- **Helm**, used in Kubernetes for managing applications.

- **Cert Manager**, used for managing certificates for HTTPS in our cluster.
- **Let's Encrypt**, used for getting free certificates for HTTPS.
- **Google Cloud**, was the most convinient and cheapest option for hosting our services. Our infrastructure guys knew it the best.

  Alternatives would have been: (AWS, Azure).
- **Cloudflare**, used as our DNS provider.
- **Hostpoint**, is where we bought our domain.
- **Docker**, used for containerization and for local development.

Possible technologies for future use:

- **Terraform**, we might use Terraform for infrastructure deployments if we need to change the platform often.
- **Redis**, could be used later for better scaling and performance by using caching.
- **Kafka**, if we would do microservices or use Postgres we would use Kafka for communication between services and writing to the sharded Database.

3.7.4 Pipeline

- **Gitlab CodeQuality**, used for checking Backend code quality (running in the pipeline). Not used for the frontend because it doesn't work with our EsLint config.

  Alternative: SonarQube, but it is not free and a big effort to integrate with a private repository, but we can use SonarLint locally in the IDE.
- **EsLint**, used for checking Frontend code quality in the pipeline and locally. We had the most experience with it compared to others.

  Alternatives: StyleCop, Prettier

3.7.5 Database

- **MongoDB**, easiest to use on multiple nodes and offers everything we need.

  Alternatives: CosmosDB, Postgres, Redis. More complicated on multiple nodes.

3.7.6 API's

- **Google Places API**, Google has the best data and we fit in their free tier.

  Alternative: OpenStreetMap
- **Google Identity**, Google Auth is secure, needs less coding and is easier for the user.

  Alternative: Basic Auth with JWT (we would need to implement it ourselves)

3.7.7 Testing

- **Cypress**, easiest to use on multiple nodes and offers everything we need.

  Alternative: Selenium.

- **XUnit tests**, used for testing the backend.
- **Mock**, used for mocking in the backend.

3.7.8 Tools

- **Jira**, we choose Jira because of it's popularity and because more of us know it rather than YouTrack.

  Alternatives: YouTrack, Trello, GitLab (was too function limited)

- **Confluence**, because we use Jira we can reference Jira tickets in Confluence and vice versa.
- **Miro**, free and perfect for creating diagrams together.
- **Figma**, de facto standard for UI design.
- **GitHub Copilot**, optionaly used for quicker coding.
- **Overleaf**, optionaly used for writing LaTeX quicker.
- Various **source control tools** like: Git Bash, GitKraken, SourceTree

3.8 Tech Stack

This is a visualisation of our used technologies.

Chapter 4

System Configuration

4.1 Setup Traefik

Since our task was to Setup Traefik with HTTP3 it required some extra steps in order to run Traefik with HTTP3. It took us quite some time to figure out all extra changes since it's barely documented somewhere, but we managed to do so with different blog posts and some Github comments.

4.1.1 Deploying the helm chart

First we applied the official Helm chart from Traefik with the following additional arguments in the values file:

```
helm upgrade --install traefik traefik/traefik --namespace traefik
    --create-namespace
```

Listing 4.1: Traefik Helm Chart

```
additionalArguments:
- "--experimental.http3=true"
- "--entrypoints.websecure.http3"
- "--entrypoints.websecure.http3.advertisedport=443"
```

Listing 4.2: Values file for Traefik Chart

We thought that is it, but soon we recognized the client (browser) was only communicating through normal HTTP2. Well after this part we all knew why the HTTP3 feature is still marked as experimental. So we started troubleshooting the problem.

4.1.2 The problem

A couple hours later after doing some troubleshooting and research we could identify the problem: The problem lays within, that there are issues in Kubernetes when trying to listen to the same port with different protocols (TCP + UDP). This first hint, which actually was quite hard to find, can be found at Traefik's official helm-chart repository [1], see snipped below:

```
## Enable HTTP/3 on the entrypoint
   ## Enabling it will also enable http3 experimental feature
   ## https://doc.traefik.io/traefik/routing/entrypoints/#http3
   ## There are known limitations when trying to listen on same ports for
   ## TCP & UDP (Http3). There is a workaround in this chart using dual
       Service.
   ##
       https://github.com/kubernetes/kubernetes/issues/47249#issuecomment-587960741
```

[1] Traefik Helm Chart Value File: https://github.com/traefik/traefik-helm-chart/blob/master/traefik/values.yaml

```
    http3:
      enabled: false
    # advertisedPort: 4443
```

Listing 4.3: Traefik Deployment

When troubleshooting the deployment and services in kubernetes, it was visible that only the TCP port 8443 was exposed on the deployment and services, no UDP Port 8433 was exposed.

4.1.3 Adjusting the deployment

Knowing the actual problem, a possible hint for the workaround could be found. [2].

So finally we ended up deleting and reapplying the traefik deployment with the additonal defined UDP port using kubectl apply:

```
apiVersion: apps/v1
kind: Deployment
metadata:
...
  spec:
      containers:
...
        ports:
...
        - containerPort: 8443
          name: websecure-udp
          protocol: UDP
...
```

Listing 4.4: Traefik Deployment

From this moment on, it could be seen that the container in the Traefik pod was exposing 8843 as well on the UDP protocol.

4.1.4 Adjusting the Service

Furthermore the Traefik service definition had to be adjusted and reapplied, in order to make the external Loadbalancers (Google LB) listen on the UDP Port. Very useful was the blog from Getambassador [3] , which explained that two sperate loadbalancers have to be requested, since Google doesn't allow external loadbalancers in mixed mode (TCP + UDP) at the moment. Normally when requesting a loadbalancer, an external IP gets assigned randomly. Since we need the same IP address for both UDP and TCP loadbalancer, a reservation for the IP address had to be done first. This IP address was then defined in the service definition with the property "loadBalancerIP".

```
apiVersion: v1
kind: Service
...
metadata:
spec:
  ports:
  - name: web
    port: 80
    protocol: TCP
    targetPort: web
  - name: websecure
    port: 443
    protocol: TCP
    targetPort: websecure
...
  type: LoadBalancer
```

[2] abc https://github.com/k3s-io/k3s/issues/6757

[3] Getambassador GKE LB Setup mixed mode https://www.getambassador.io/docs/edge-stack/latest/howtos/http3-gke

```
  loadBalancerIP: 34.27.119.20
---
apiVersion: v1
kind: Service
...
spec:
...
  ports:
  - name: websecure-udp
    port: 443
    protocol: UDP
    targetPort: websecure-udp
...
  type: LoadBalancer
  loadBalancerIP: 34.27.119.20
...
```

Listing 4.5: Traefik Service

4.1.5 Verify Communication

After applying the described changes the website was successfully using HTTP3:

Status	Method	Domain	File	Protocol
304	GET	dev.spontaneous.ch	/	HTTP/3
200	GET	dev.spontaneous.ch	app.js	HTTP/3

4.2 Setup shared Mongo DB

4.2.1 MongoDb Enterprise

There are different ways to achieve the sharded MongoDB setup. Probably the simplest and best for an productive environment, is to use the Mongodb's enterprise products (Ops Manager and MongoDB Enterprise Kubernetes Operator). Using the enterprise tools the deployment is very easy, basically just apply following deployment for the custom resource MongoDB. As well montoring and maintenance of the cluster is way easier. [4]

```
---
apiVersion: mongodb.com/v1
kind: MongoDB
metadata:
  name: <my-sharded-cluster>
spec:
  shardCount: 2
  mongodsPerShardCount: 3
  mongosCount: 2
  configServerCount: 3
  version: "4.2.2-ent"
  opsManager:
    configMapRef:
      name: <configMap.metadata.name>
            # Must match metadata.name in ConfigMap file
  credentials: <mycredentials>
  type: ShardedCluster
  persistent: true
```

Listing 4.6: Traefik Deployment

4.2.2 Kubernetes Deployment Sharded Cluster

Since we have no enterprise licence and therefore were not able to use this tools, we created our own deployment for the sharded cluster. The sharded cluster consists of following instances, as described and visualised in the arichitecture part of the documentation:

- 2x Database Router: Responsible for receiving requests and redirecting it to right shard.
- 3x Config Servers: Store the information about which shard is stored on which nodes.
- 3x Nodes for shard01: store the actual data for the specific shard01
- 3x Nodes for shard02: store the actual data for the specific shard02

For each of the 11 instances a corresponding service and PVC (Persistent volume claim) had to be defined. The whole deployment can be viewed in our repository. Following we are going into the crucial argument parameters defined in the deployment:

[4] MongoDb Enterprise deployment https://www.mongodb.com/docs/kubernetes-operator/stable/tutorial/deploy-sharded-cluster/

Argument parameters for routers

To tell the routers to which configdatabase to use, the configuration servers need to be defined:

```
...
    args:
    - mongos
    - --port
    - "27017"
    - --configdb
    - rs-config-server/spontaneous-database-configsvr01:27119,
      spontaneous-database-configsvr02:27120,spontaneous-database-configsvr03:27121
    - --bind_ip_all
...
```

Listing 4.7: deployment arguments routers

Argument parameters for configservers

In the configuration servers deployment, the arguments need to be defined that they actual are configservers and to which replicationset of configservers they belong.

```
...
    args:
    - mongod
    - --port
    - "27017"
    - --configsvr
    - --replSet
    - rs-config-server

...
```

Listing 4.8: deployment arguments routers

Argument parameters for shard nodes

In the shard nodes deployment, the arguments need to be defined that they actual are shard nodes and to which set of shard they belong.

```
...
    args:
        - mongod
        - --port
        - "27017"
        - --shardsvr
        - --replSet
        - rs-shard-01
...
```

Listing 4.9: deployment arguments routers

4.2.3 Configuration Sharded cluster

After deployment of the instances the actual configuration can be done. The configuration cannot be included in the deployment itself, since all the instances need to be up in order to be able to configure the sharded cluster.

Setup Configuration Servers

Following command can be executed on 1 of the 3 config servers available. It defines the replicaset with the 3 instaces as config servers.

```
rs.initiate(
{_id: 'rs-config-server', configsvr: true, version: 1, members: [
    { _id: 0, host :  'spontaneous-database-configsvr01:27119 ' },
    { _id: 1, host :  'spontaneous-database-configsvr02:27120 ' },
    { _id: 2, host :  'spontaneous-database-configsvr03:27121 ' },
] })
```

Listing 4.10: configure configuration servers

Setup Shards

In order to create a shard, the shard name as well as the members hosting the shard need to be defined. Following command can be executed on 1 of the 3 nodes deployed for shard01 to complete the shard creation:

```
rs.initiate({_id: 'rs-shard-01', version: 1, members: [
    { _id: 0, host : 'spontaneous-database-shard01-node01:27122' },
    { _id: 1, host : 'spontaneous-database-shard01-node02:27123' },
    { _id: 2, host : 'spontaneous-database-shard01-node03:27124' },
] })
```

Listing 4.11: configure shard01

The same can be done for shard02

```
rs.initiate({_id: 'rs-shard-02', version: 1, members: [
    { _id: 0, host : 'spontaneous-database-shard02-node01:27125' },
    { _id: 1, host : 'spontaneous-database-shard02-node02:27126' },
    { _id: 2, host : 'spontaneous-database-shard02-node03:27127' },
] })
```

Listing 4.12: configure shard02

Setup router

Finally we need to tell the router which shards are available and where they are available.

```
sh.addShard('rs-shard-01/spontaneous-database-shard01-node01:27122')
sh.addShard('rs-shard-01/spontaneous-database-shard01-node02:27123')
sh.addShard('rs-shard-01/spontaneous-database-shard01-node03:27124')
sh.addShard('rs-shard-02/spontaneous-database-shard02-node01:27125')
sh.addShard('rs-shard-02/spontaneous-database-shard02-node02:27126')
sh.addShard('rs-shard-02/spontaneous-database-shard02-node03:27127')
```

Listing 4.13: add shards to router

Create sharded database

As last step the database can be defined to be sharded. In order to complete for each collection of the database a shardingkey needs to be defined.

```
use Spontaneous
sh.enableSharding('Spontaneous')
db.adminCommand( { shardCollection: 'Spontaneous.History', key: { _id: 1 } } )
db.History.insertMany([{ 'PlayerName': 'Hans Muster', 'Activity': 'Skating',
    'ActivityCategory': 'Outdoor' }, { 'PlayerName': 'Mirio', 'Activity':
    'CSGO',
'ActivityCategory': 'Gaming' }])
```

Listing 4.14: enabling sharding for database

4.3 Setup Spontaneous webapp

Our task was to create a simple frontend (e.g., HTML, Vue, React, Svelte). Since we combined our Distributed Systems Project with our SE-Project, we decided to use the same webapp for both projects.

4.3.1 Deployment on Kubernetes

Kustomize

As you can see in the following section we use kustomize to create multiple overlays for our webapp. Currently we have one for the development environment and one for the production environment.

```
        README.md
        base
                deployment.yaml
                kustomization.yaml
                middleware.yaml
                pvc.yaml
                service.yaml
        overlay
            development
                clusterissuer.yaml
                deployment_patch.yaml
                ingress.yaml
                kustomization.yaml
            production
                clusterissuer.yaml
                ingress.yaml
                kustomization.yaml
```

Listing 4.15: Kustomize

Cert manager

In order to use https we use Cert Manager and Let's Encrypt. For each environment we have a cluster issuer which is used to create the certificates.

```
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
name: letsencrypt-production
spec:
acme:
server: https://acme-v02.api.letsencrypt.org/directory
email: info@spontaneous.ch
privateKeySecretRef:
name: letsencrypt-production
solvers:
- http01:
ingress:
class: traefik
```

Listing 4.16: Cert manager cluster issuer

In the ingress yaml we then defined the following annotations:

```
annotations:
    kubernetes.io/ingress.class: traefik
    cert-manager.io/cluster-issuer: letsencrypt-production
...
  tls:
    - hosts:
        - spontaneous.ch
      secretName: spontaneous-ch-tls-secret
```

Listing 4.17: Cert manager annotations