1. HA-Chat

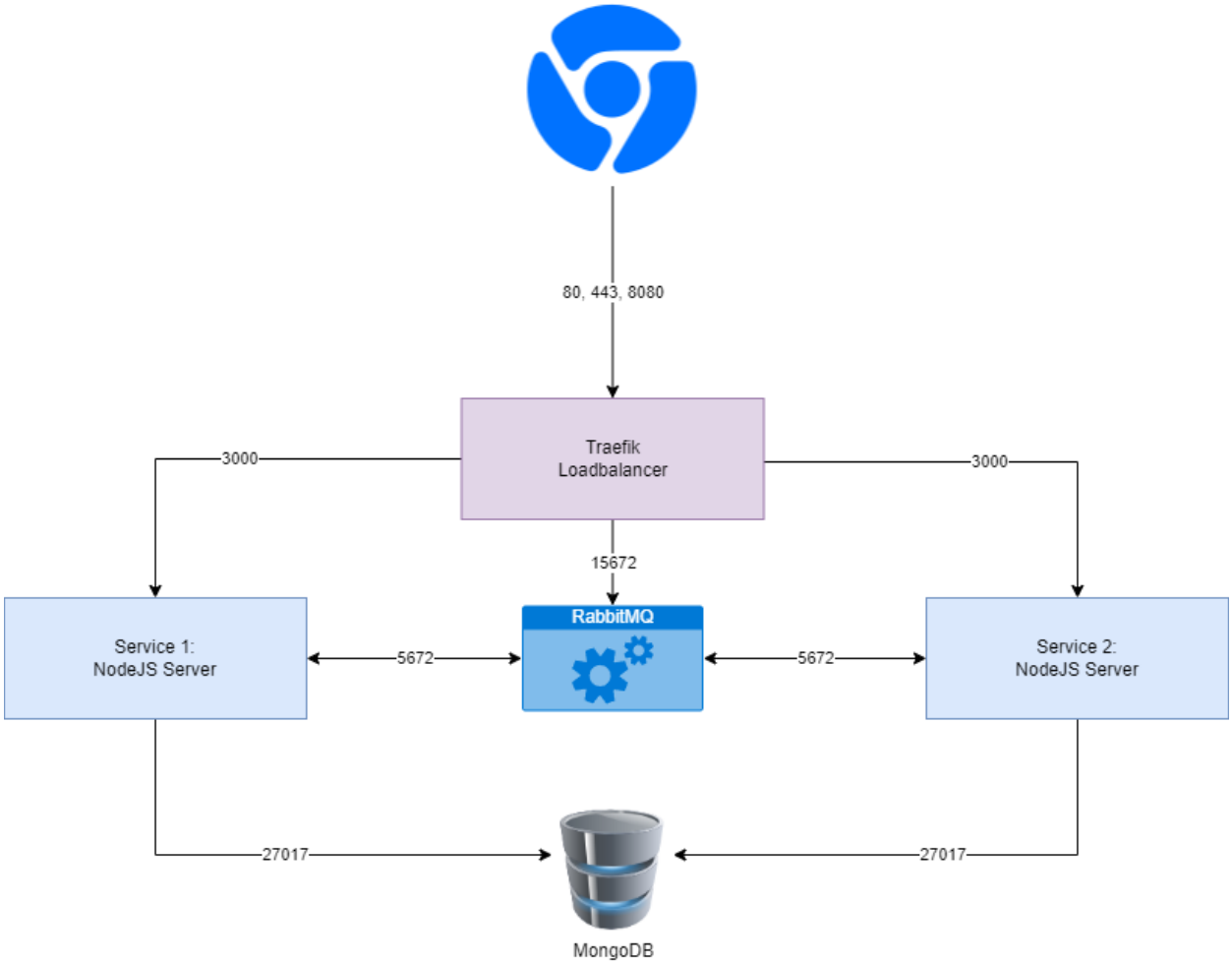
1.1. Requirements

All requirements below must be met in order to pass this lecture.

1. Load balancing with scalable service

2. Failover of a service instance
3. Dockerized and Frontend connects via HTTP/3
4. Persistant storage (storage does not need to be scalable, but you can build it scalable if you want)
5. Use latest stable releases of chosen libraries and frameworks

1.2. Topology



1.3. Application

We decided to work with Docker Compose, because it's more or less simple to setup in a local environment and it has multiple advantages:

- Simplified Orchestration: Docker Compose allows you to define and manage multi-container applications using a single configuration file. It simplifies the process of deploying and orchestrating complex setups by specifying the relationships and dependencies between different containers.

- Service Isolation: Docker Compose enables you to isolate individual services within their own containers, ensuring that each component of your application runs independently. This isolation helps in avoiding conflicts between dependencies, simplifying troubleshooting, and facilitating scalability.
- Quick Setup and Tear-down: Docker Compose simplifies the process of starting and stopping your application stack. With a single command, you can spin up all the necessary containers and their dependencies. Similarly, tearing down the environment is straightforward, reducing the cleanup effort and ensuring a clean state for each deployment.
- Version Control: By defining your application's infrastructure and dependencies in a Docker Compose file, you can store it alongside your source code. This approach allows for version control of your entire application stack, ensuring reproducibility and enabling collaboration with other developers.
- Environment Standardization: Docker Compose promotes consistency across different environments. Developers, testers, and operators can use the same configuration to run the application, ensuring that everyone is working with the same software versions, dependencies, and environment settings.
- Scalability: Docker Compose supports scaling individual services by specifying the desired number of instances. This capability allows you to scale specific components of your application stack based on demand, optimizing resource utilization and improving performance.

As frontend we are using simple dockerized chat application for this challenge.

Keyfeatures of the application:

- The ability to store messages and users perssistently in a database
- Session management
- WebSocket support
- Advanced setup for loadbalancer

1.3.1. Components

Database -> MongoDB

- It uses the official MongoDB image with an additional startup script which sets up users in order to have a securized database (using MONGO_DB_APP_PASSWORD, MONGO_DB_APP_USERNAME, MONGO_DB_APP_DATABASE enviroment variables).

```
db:
    build: ./mongo
    image: ageapps/docker-chat:mongo
    volumes:
      - ./database:/data # volume in host -> $(pwd)/database
    environment:
      - MONGO_DB_APP_PASSWORD=node
      - MONGO_DB_APP_USERNAME=node
      - MONGO_DB_APP_DATABASE=nodedb
    ports:
      - "27017:27017"
    healthcheck:
      test: ["CMD", "echo", "show dbs", "|", "mongo"]
      interval: 30s
      timeout: 10s
      retries: 3
    command: mongod --smallfiles
```

Webserver -> NodeJS

- NodeJS server containin all business logic and that features mentioned above. It uses the official NodeJS image as base image.
- Session Management: whenever the name is introduced, a session is created and managed using NodeJS module Express Session. Aditionally, in order to have persistent sessions while scalling our services, sessions are stored in the database using MongoStore.

```
app:
    build: ./app
    image: ageapps/docker-chat:app
    volumes:
      - ./app:/app # volume in host -> $(pwd)/database
      - /app/node_modules # volume node_modules from container
    depends_on:
      db:
        condition: service_healthy

    environment:
      - DEBUG=docker-chat:*
      - MONGO_USERNAME=node
      - MONGO_PASSWORD=node
      - MONGO_DATABASE=nodedb
      - SCALABLE=true
      - RABBIT_HOST=rabbit
      # - PARSE_MSG=true
    command: [nodemon, ./bin/www]
```

WebSocket -> Socket.io

- Socket.IO is a library that enables low-latency, bidirectional and event-based communication between a client and a server.
- It is built on top of the WebSocket protocol and provides additional guarantees like fallback to HTTP long-polling or automatic reconnection.

Message Broker -> RabbitMQ

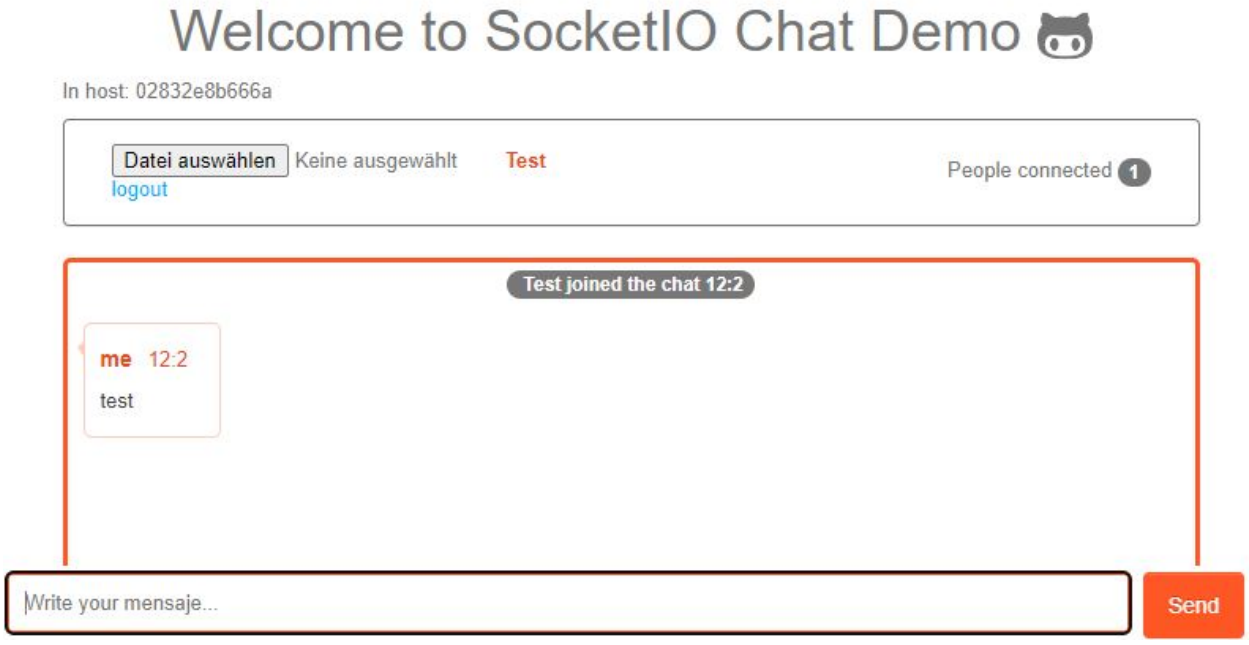
- This service is needed in order to scale WebSockets.
- RabbitMQ is a widely used open-source message broker that implements the Advanced Message Queuing Protocol (AMQP). It serves as an intermediary for exchanging messages between different systems, applications, and components within a distributed architecture.
- RabbitMQ has extensive language support with client libraries available for multiple programming languages, making it easy to integrate into different applications and systems. It is widely used in various domains, such as microservices architectures, event-driven systems, and distributed systems, to enable reliable and scalable message-based communication.

```
rabbit:
   image: rabbitmq:3-management
   ports:
     - "15672:15672"
```

1.3.2. Preview

Login:

Chat:



1.4. Traefik Loadbalancer

For the loadbalancing for our application we choose traefik. Traefik is a popular open-source reverse proxy and load balancer designed for modern microservices architectures. It acts as a gateway between your applications and the outside world, routing incoming traffic to the appropriate services based on their configurations. Traefik supports dynamic service discovery, automatically adapting to changes in your infrastructure without requiring manual configuration updates. It also integrates well with container orchestrators like Kubernetes and Docker. So in our case since we are using docker-compose for our application it is the best fit for us. Additionaly we also had a short interduction to traefik during our lectures, and how to get http/3 enabled in traefik.

1.4.1. Compose service definition

```yaml
services:
  proxy:
    image: traefik:v2.10.1
    ports:
      - "80:80"
      - "443:443"
      - "443:443/udp"
    volumes:
      - ./traefik.yaml:/etc/traefik/traefik.yaml
      - ./traefik_dynamic.yml:/etc/traefik/traefik_dynamic.yml
      - ./certs/key.pem:/etc/traefik/key.pem
      - ./certs/cert.pem:/etc/traefik/cert.pem
      - ./certs/cert.pem:/etc/ssl/certs/cert.pem
```

- `proxy` : This is the name of our service.
- `traefik:v2.10.1` : Specifies the Docker image to be used for the Traefik container. In this case, the image named "traefik" is used. We use a specific version instead of latest due security concerns.
- `ports` : Defines the port mappings for the container. The following ports are exposed:
  - Port 80 of the host is mapped to port 80 of the container for HTTP traffic.
  - Port 443 of the host is mapped to port 443 of the container for HTTPS and HTTP/3 traffic.
  - Port 443 of the host is also mapped to port 443/udp of the container for HTTP/3 traffic.
- `volumes` : Specifies the volumes to be mounted in the container. The following files and directories are mounted:

- `./traefik.yaml:/etc/traefik/traefik.yaml` : Mounts the local file `traefik.yaml` into the container at `/etc/traefik/traefik.yaml` . This file contains the static configuration for Traefik.
- `./traefik_dynamic.yml:/etc/traefik/traefik_dynamic.yml` : Mounts the local file `traefik_dynamic.yml` into the container at `/etc/traefik/traefik_dynamic.yml` . This file may contain dynamic configuration for Traefik, such as routing rules.
- `./certs/key.pem:/etc/traefik/key.pem` : Mounts the local file `key.pem` into the container at `/etc/traefik/key.pem` . This file contains the private key for TLS encryption.
- `./certs/cert.pem:/etc/traefik/cert.pem` : Mounts the local file `cert.pem` into the container at `/etc/traefik/cert.pem` . This file contains the SSL/TLS certificate.
- `./certs/cert.pem:/etc/ssl/certs/cert.pem` : Mounts the local file `cert.pem` into the container at `/etc/ssl/certs/cert.pem` . This file may be used as the SSL/TLS certificate in another location within the container.

traefik.yml

Experimental Configuration:

```
experimental:
  http3: true
```

Enables the expermental HTTP/3 support in Traefik.

Entry Points: This section defines the entry points where Traefik will listen for incoming traffic.

```
entryPoints:
  web:
    address: ":80"
    http:
      redirections:
        entryPoint:
          to: websecure
          scheme: https
```

The "web" entry point listens on port 80 for incoming HTTP traffic and redirects it to the "websecure" entry point using HTTPS.

```
  websecure:
    address: ":443"
    http3:
      advertisedPort: "443"
```

The "websecure" entry point listens on port 443 for incoming HTTPS traffic and supports HTTP/3. The advertised port is set to 443 for HTTP/3.

API Configuration:

```
api:
  dashboard: true
```

This enables the Traefik dashboard API, providing a web-based dashboard to monitor and manage Traefik's configuration and metrics.

Providers: This section configures the providers that Traefik uses to obtain dynamic configuration.

```
providers:
  file:
    filename: "/etc/traefik/traefik_dynamic.yml"
```

The "file" provider is used to read dynamic configuration from a file located at "/etc/traefik/traefik_dynamic.yml". Traefik will apply the configuration defined in this file.

Logging Configuration

```
log:
  level: TRACE
```

This sets the log level to "TRACE", enabling detailed logging for Traefik with extensive information about its internal processes and actions.

traefik_dynamic.yml

HTTP Configuration

```
http:
  routers:
    dashboard:
      rule: "PathPrefix(`/api`) || PathPrefix(`/dashboard`)"
      entrypoints:
        - "websecure"
      service: "api@internal"
    docker-chat:
      rule: "Host(`app.localhost`)"
      service: docker-chat
      entrypoints:
        - "websecure"
      tls:
        domains:
          - main: "app.localhost"
  services:
    docker-chat:
      loadBalancer:
        healthcheck:
          path: "/"
          interval: "10s"
          timeout: "1s"
        servers:
          - url: "http://app:3000"
```

This configuration defines HTTP routers, services, and TLS settings.

- `dashboard` router:
  - Rule: This router has a rule that matches requests with a path prefix of /api or /dashboard.
  - Entrypoints: It uses the "websecure" entrypoint to handle incoming traffic.
  - Service: Requests matching this rule are forwarded to the internal Traefik API service (api@internal).
- `docker-chat` router:
  - Rule: This router matches requests with the host `app.localhost`.
  - Service: Requests matching this rule are forwarded to a service named docker-chat.
  - Entrypoints: It uses the "websecure" entrypoint.
  - TLS: It specifies TLS settings for this router with the `app.localhost` domain.

- `docker-chat` service:
  - Load Balancer: It configures a load balancer for this service.
  - Healthcheck: It defines a health check for the service using the specified path, interval, and timeout.
  - Servers: The service is associated with a server located at http://app:3000.

TLS Configuration

```
tls:
  certificates:
    - certFile: "/etc/traefik/cert.pem"
      keyFile: "/etc/traefik/key.pem"
```

This section specifies TLS certificates to be used by Traefik.

- Certificates: It defines the location of the certificate file and the key file to be used for TLS encryption.