D

DSy-FiTrack
Project ID: 6490 Leave project

Finish documentation
Pascal Lehmann authored 1 hour ago

Name	Last commit	Last update
db	Locally persist Cassandra with volume m...	1 week ago
img	Finish documentation	1 hour ago
proxy	Finish documentation	1 hour ago
web	Finish documentation	1 hour ago
Makefile	Rename dirs	3 days ago
README.md	Finish documentation	1 hour ago
docker-compose.yml	Finish documentation	1 hour ago

README.md

DSy-FiTrack

Authors: Nicolas Gattlen, Pascal Lehmann

A simple body tracker for submission in the Distributed Systems (DSy) Challenge Task FS 2023.

screenshot_web_interface

Table of Contents

Requirements

- https://docs.docker.com/get-docker/
- https://github.com/FiloSottile/mkcert#installation
- https://www.jetbrains.com/idea/download/
- `sudo apt install openjdk-17-jdk`
- `sudo apt install make`

Quick Start

1. Initialize the directory with `make init`.
2. Generate a new certificate with mkcert named `cert.pem` and place it in the `proxy/certs` folder.

3. Build the application with `make build`.
4. Start the application with `make start`.

> Alternatively, you can start the dev environment (without the webserver containers) with `make start-dev`. That way you can run the web application within your IDE.

Whenever you're done you can stop all containers with `make stop`.

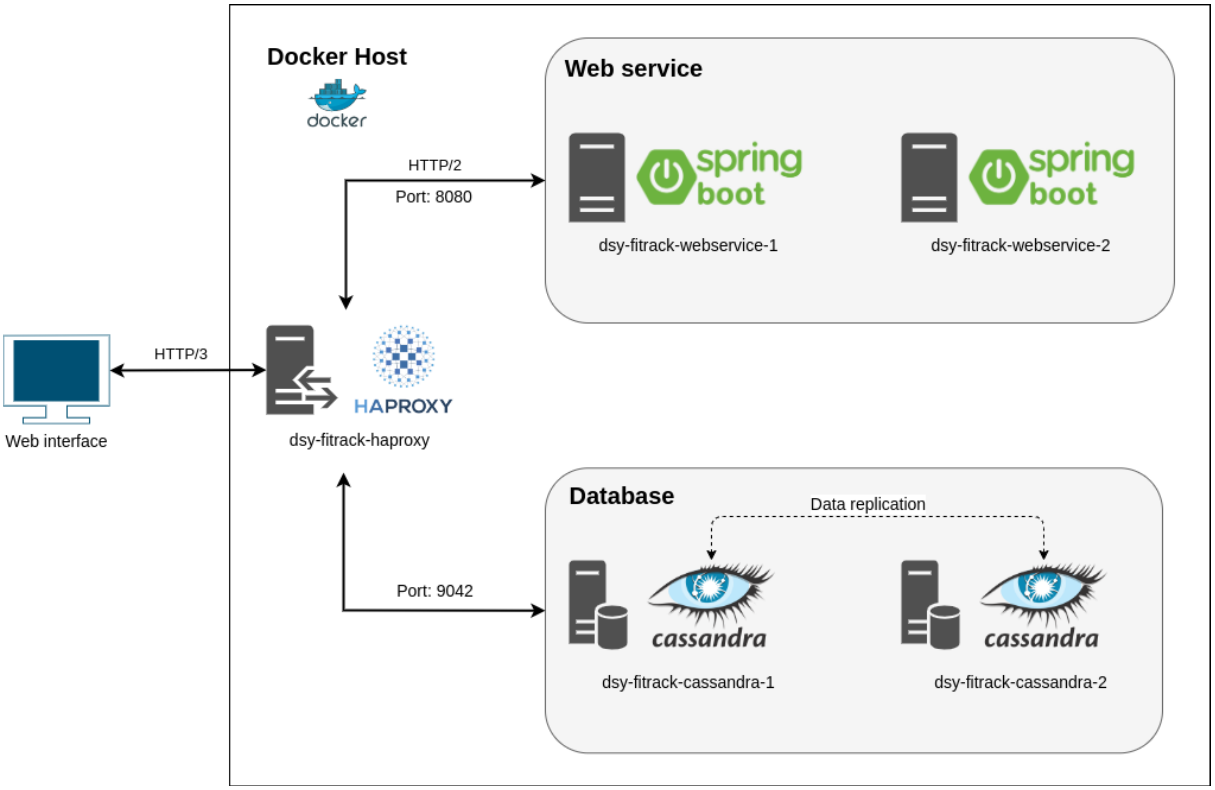
Architecture

The application consists of 3 components, the web service, the database and the load balancer / reverse proxy. All components are deployed in their own Docker container and are orchestrated with Docker-Compose. The web service and the database deployment donsists of two containers each to achieve redundancy.

The load balancer / reverse proxy is an HAProxy instance that handles the communication from the client (Port 80 and 443) and between the different components. Requests between HAProxy and the client are sent over HTTP/3 with HTTPS (Port 8080) and are load-balanced between the two web service instances using round-robin. HTTPS connections are terminated when arriving at the proxy and get forwarded to the web client unencrypted with HTTP/2. All communication between the web service and the database (Port 9042) is also routed and load-balanced via HAProxy.

The web service consists of two containers running the same Spring Boot application. It provides a simple web interface with server-side rendering and persists its data in the database. A requirements of the challenge task is to be able to continue serving the website even when one of the containers crash. Hence, we designed the web application to be stateless and are deploying it to run in two containers simultaneously.

The database consists of two containers running Apache Cassandra instances that automatically replicate their between each other. The communication for the replication happens directly between the two databases and not via HAProxy. It is the only communication between components that happens directly.



Design Decisions

HAProxy

The challenge task requirements dictate that the communication between the client and the load balancer has to be via HTTP/3. Therefore Nginx was unfortunately out of the race. We wanted to use a popular load balancer which we might come across later in our career. So HAProxy was a good fit with its big community and relatively comprehensive documentation.

Kotlin

We decided to use Kotlin as our programming language because we have frequently used Java during our time studying but didn't like it's verbosity and lack of modern syntax features like optional chaining, a nullish coalescing operator or operator unpacking. We had heard a lot

of great things about Kotlin and seen a few impressive comparisons of code snippets between Java and Kotlin, which got us excited to try it out.

Spring Boot

Spring / Spring Boot is by far the most popular web framework in the Java ecosystem, so it was the natural choice to use together with Kotlin. Also, we figured that there would be a good chance that we might come across Spring Boot in our future career and would be able to use the experience gained throughout this project.

We used Thymeleaf our HTML templating enginge because it is easy to integrate in Spring Boot and is known to be secure and robust.

Apache Cassandra

We decided to use Apache Cassandra as our database because it's a popular, open source, distributed database. Since we had to provide redundancy for the web service we wanted to achieve the same for the database. Unlike many other distributed databases, Cassandra is completely free and was therefore a good choice for us. Additionally, Cassandra can be conveniently integrated into Spring Boot with a custom provider that can connect to the instances, create the database and generate queries from code.

We opted for the third-party Docker image `bitnami/cassandra` because the original Cassandra Image doesn't support the `/docker-entrypoint-initdb.d/` convention yet. Scripts placed in this directory are automatically executed by docker after the container was started. We use it to create the keyspace for our application which needs to exists for the Spring Boot Cassandra provider to be able to connect to the database.