ESIR 2nd year
Software Architecture
Final Lab Exercise

Your µ-service-based Web application

Objectives

— Learn to use containers through Docker
— Learn to decouple your services into µ-services using containers
— Manage multi-container applications with `docker-compose`
— Increase decoupling through the use of Message-Oriented-Middlewares (RabbitMQ or Apache Kafka)
— Learn to deploy minimal Java-based applications with Quarkus and GraalVM
— Understand when and how to use different architectural approaches and what properties they provide (decoupling, latency, throughput, memory or CPU usage)

The architecture of your µ-service Web application

Your Web application should ideally have :

1. An HTTP front-end Web server, for example, `Nginx` Apache's `HTTPD`, that exposes the HTTP standard ports, (port 80 for HTTP traffic, port 443 if you provide an SSL key for encrypted HTTPS traffic).
2. One or more µ-services for your API backend.
3. One µ-service for your application's frontend code (as a side-note, your HTTP server can also store all of the front-end code that is served to client devices).
4. A notification µ-service written using Quarkus that, at a minimal, should send emails for announced events.
5. A message-oriented-middleware (MOM) to strongly decouple the notification service from the rest of your application.
6. As many volume containers as necessary to properly persist your data (e.g., database, MOM).

Development vs Production containers

You may provide a development mode which opens many ports so you can use tooling (for example, your database client or RabbitMQs management service), and that allows for *hot-reloading* of your code (for example, through the use of docker bind-mounts). This is extremely useful to avoid having to redeploy your entire application after each small iteration. This is often done through the creation of an additional `docker-compose` development file, which can be used instead of the production configuration when you are programming and testing your application.

Regarding the production configuration, we expect the minimal amount of ports to be open, docker volumes to be properly attached to the containers to persist your data, all external traffic should pass through the HTTP server and not directly on internal ports (e.g., port 3000 should not be available outside of the docker network).

Ideally, your docker production images contain all the necessary files for deployment. One example, Nginx should ideally copy it's configuration file into the container, allowing it to be deployed to production without any additional dependencies [1].

1. See section on *Managing Content and Configuration files* to better understand your options
`https://www.nginx.com/blog/deploying-nginx-nginx-plus-docker/`

Notification service (Quarkus, MOM, email and more)

You must develop a notification service for your application. In it's most simplest form, an association should be able to send a notification to all of their members by email. A more interesting form, would allow configuring multiple external services, such as social media accounts (e.g., Twitter or Facebook), and automatically posting the event to their accounts. The Notification Service should be written in Quarkus and use GraalVM. It must be entirely decoupled from your existing backend by a Message-Oriented-Middleware, we recommend either RabbitMQ or Apache Kafka. Your backend thus produces messages and your notification service consumes and dispatches them.

A minimal notification service requires :

1. Adding the notification functionality to your NestJS backend (new API, front-end code, etc.).
2. Configuring and starting your MOM service (in it's own container of course, either RabbitMQ or Kafka are recommended).
3. Your MOM, in it's simplest form, is a simple message queue, but you may explore other communication alternatives (PubSub, fanout, etc.).
4. Writing the message *producer* code in your NestJS backend to produce messages on the MOM.
5. Creating a notification service in Quarkus that consumes messages from the queue.
6. Configuring and starting an SMTP server (MailDev [2] is a simple service you may use for testing emails).
7. Your Quarkus service should thus be able to consume messages from the MOM and send emails.
8. Each email, if it's an event at a specific time, should attach an ical file with the corresponding event information.

You may of course enhance the notification service with any number of functionalities, such as adding additional connectors (e.g., to social media, to PDF files to be printed), allowing the NestJS to configure the sending time, filtering users, etc. You may also configure your Quarkus notification service to only run at specific hours, thus creating "batch" loads to send all notifications at specific times. If your Quarkus service does not run all the time, ensure that your MOM is properly configured to persist the messages so they are not lost when the service is down.

Load testing

Software load testing is an important part of any software project. Load testing is a performance-test that automatically interacts with your application's services and tests how well it reacts to different loads. For example, you may call your backend services thousands of times to simulate the service being used in production by clients.

There are many load and performance testers for applications that use HTTP and REST. Of them, you are free to choose any. Please justify this choice. Of the many choices available, here's a list of some of the most widely used load testers today :

— Wrk `https://github.com/wg/wrk`
— K6 `https://github.com/grafana/k6`
— Vegeta `https://github.com/tsenart/vegeta`
— Locust`https://github.com/locustio/locust`
— Apache JMeter `https://github.com/apache/jmeter`
— Hey `https://github.com/rakyll/hey`

Each tool has a different set of options and tradeoffs. At a minimal, we expect you to learn to use one of these tools to be able to call your backend APIs thousands of times and extract performance information such as :

— Number of failed requests
— Average latency per request
— Highest latency
— Lowest latency

Application monitoring, telemetry or tracing

Monitoring is an essential part of maintaining any software running in production. The idea is to extract data points from your application to ensure that it is running correctly, and to quickly identify any issues.

2. `https://maildev.github.io/maildev/`

There are many framework to perform application monitoring, and again, you are free to choose any one you want. Of them, we can recommended using Prometheus [3], which, in particular when coupled to Grafana, can provide dashboards with important information about your application. We expect you to implement a monitoring framework that provides real-time information regarding your different services, with, at a minimal, the following information per-service :

— Requests per second or minute
— Request duration

Deliverable ("rendu")

You must provide a **technical** written report for your project where you describe your application's architecture. At a minimal, we expect :

— A schema of your µ-services architecture that shows all services and their connections.
— An explanation of your architecture including different architectures that were possible, the trade-offs they would have implied, and clear explanations for the choices you made.
— A list of the services that are implemented
— The status of each service (e.g., proposed, design, implemented, load tested, validated)
— The configuration options of your application
— How to use your application, including the existence of development and production modes
— What you have done to ensure the minimal number of ports are exposed in production, that your services are protected, etc.
— Any additional functionalities that you have provided.
— Any feedback or improvements you'd like to propose to the *Architecture Logicielle* course and the project.

The report should be available in your git repository under the name `Architecture.md` or `Architecture.pdf`. Please do not forget to add Paul Temple and Walter Rudametkin to the members of your project (ensure they have access to your code !)

3. `https://prometheus.io/`