Final Project
Development of Large Systems

Contents

Project description

The goal of this project is to develop a system with an architecture suitable for large IT systems (distributed architecture). We are aiming for characteristics like:

- Scalability:
  - databases: amount of data, number of requests, query optimization
  - application: microservices - example: using Kubernetes to orchestrate running of the containerized microservices.
- Portability: using containerized microservices
- Security: Authentication and authorization, VPC, …
- Interoperability: designing proper APIs

The whole system should be deployed in the cloud. Each service - microservice or database server - can run on a different cloud. We are aiming for characteristics like:

- elasticity
- high availability
- low latency

NOTE: Due to the limitations of what can be provisioned for free, the deployment solution can be simplified. Then it is enough to describe the optimal deployment strategy in your presentation.

There should be a minimum of 4 microservices – this means backend applications (each can be programmed in a different language). It is also preferred that there is some frontend (for example React, Svelte, Vue, Angular, …). Frontend application does not count as a microservice.

For this project you are free to choose any development stack – programming languages, frameworks, cloud providers, database servers, etc.

You must write the specifications for your project including the functional and non-functional requirements.

From the very beginning of the project, you should set up a proper project management strategy (for the team collaboration) and use a version control system like git/GitHub and set up a CI/CD pipelines (for example using GitHub Actions) with automated tests.

Recommendations

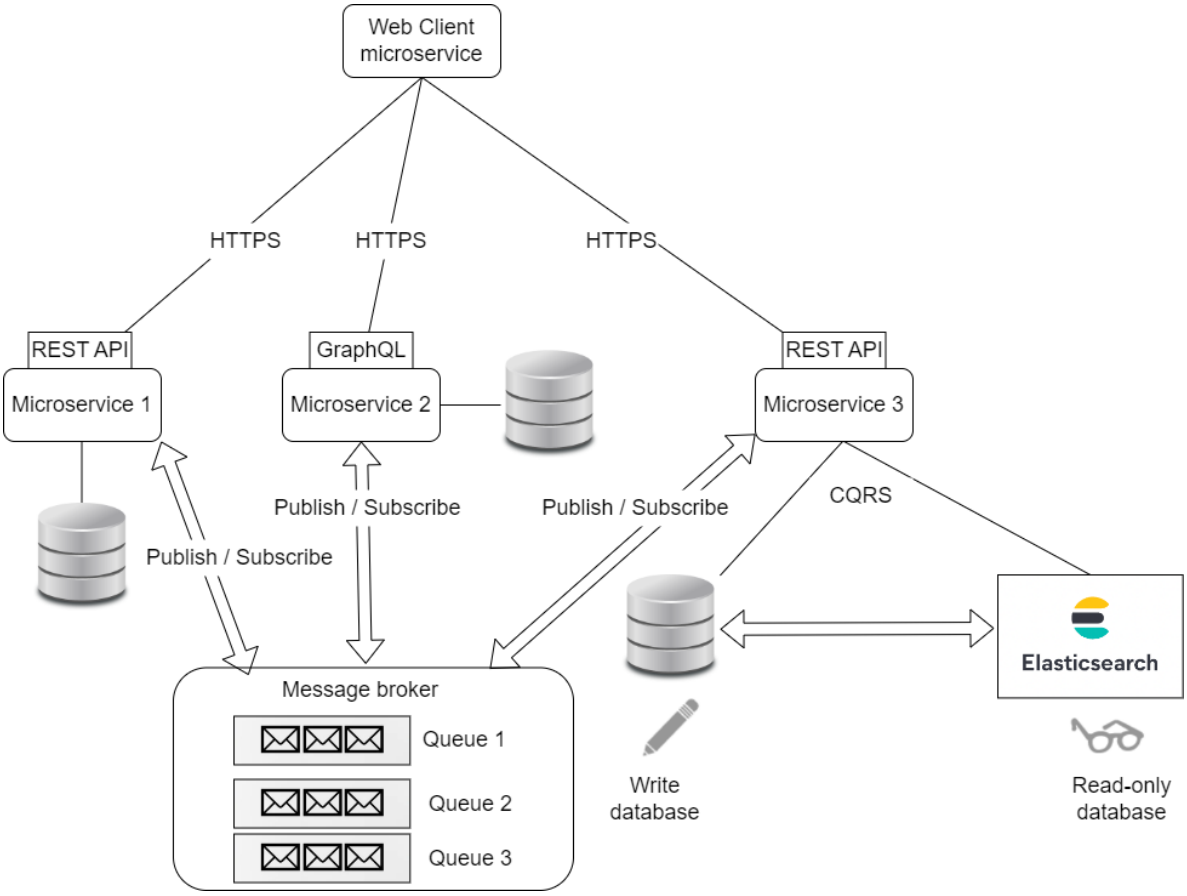
- Microservices should mainly communicate between each other using message queues (asynchronous communication).
- The backend should communicate with the frontend using REST API or GraphQL. You must implement both ways in your project - for example one microservice can use REST API and another one GraphQL, or you can implement both REST API and GraphQL for a single microservice.
- You can use serverless functions for some of the tasks (for example image processing after uploading an image).
- Logging and monitoring system.
- Authentication implementation + 3rd party integration (for example gmail, facebook)
- Email service (for example for email verification or just for a notification).
- Admin service - there could be some extra GUI and backend logic for admin role.

Here are some examples of distributed systems - just to get an idea:

- Web shop
- Banking system
- Car rental system
- Stock brokerage
- Hospital system
- …

The project should be complex enough to cover the curriculum of the whole course.

Simplified example of the system architecture:

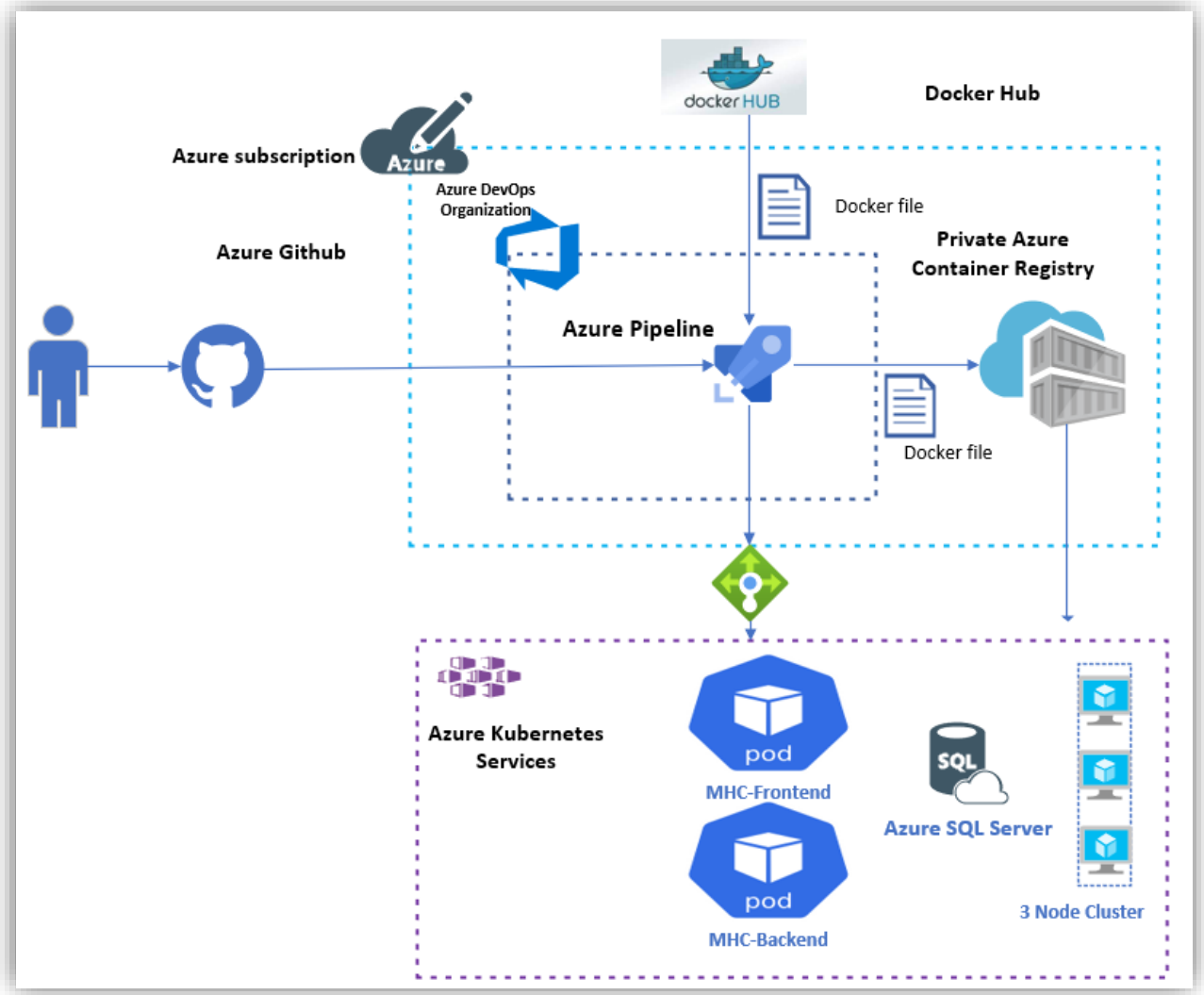


Other useful things that could be implemented:

- CQRS pattern
- Immutable data (Tombstone pattern and Snapshot pattern)
- Idempotence - if a web client retries a request, the actual execution of this request only happens once.
- Commutative message handlers - it should not matter in which order the messages are handled. The result should be the same even if they come in the wrong order.
- Saga pattern - implementing compensating transactions for the cases where multiple database servers need to be part of a global transaction.
- Versioning:
  - Source code: git / GitHub
  - Database: database migrations
  - APIs: API versioning for the REST APIs (GraphQL is by nature not versioned)
- Documentation:
  - GitHub readme, wiki
  - swagger (Open API standard) for the REST APIs
  - GraphQL endpoint documentation
  - system documentation

Simple example of cloud deployment architecture and workflow

Orchestrating containerized microservices with Kubernetes. We also need to provision all the database servers and other services from cloud providers.



In our case we will have multiple backend microservices instead of just one.

Use cloud storage like AWS S3 to store resources like images, videos, files.
Use CDN (Content distribution network) to deliver stored resources (like AWS CloudFront or Cloudflare)

Suggest geo replication and partitioning / sharding for your databases. (This can be expensive, so you are not expected to use it in production).

Final Project Delivery to WISEflow

Final Project Artifacts (things to deliver)

- Report
- Links to all GitHub repositories
    - Documentation:
        - GitHub readme, wiki
        - swagger for REST APIs
- Link to the deployed application
- A brief installation procedure that specifies how to organize the code and import the databases in a development environment with full operational capabilities (for example using docker images and docker compose).

Final Project Report

The final report will have a big influence on the exam grade. It is the only information that is accessible to the external censor.

Report size is defined in the course catalog: https://katalog.kea.dk/course/9942251/2024-2025.

The report should have this structure:

Cover page, including:

- Title
- Full names of all students in the group
- Date of delivery

List of figures
List of appendices
Table of contents (paginated index)

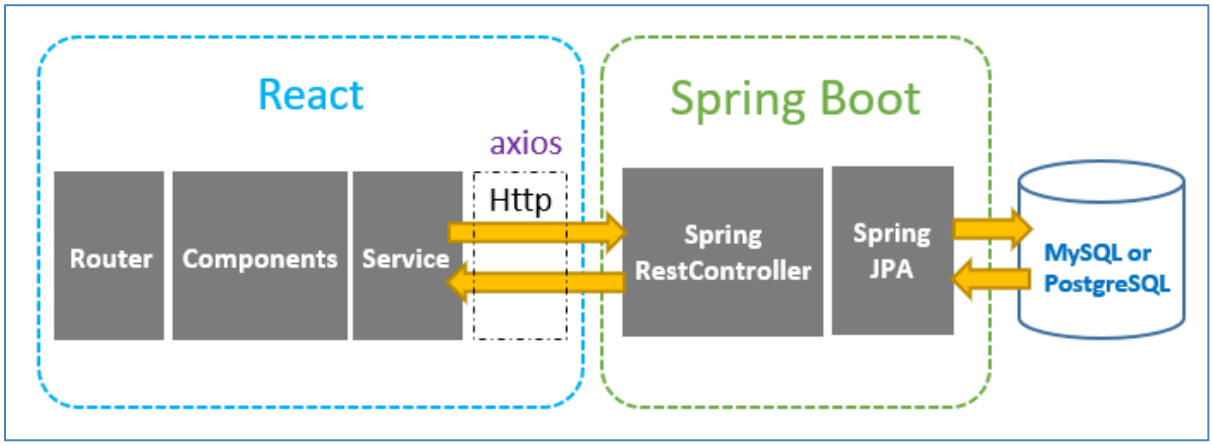
1. **Introduction**
    1.1. Problem description
    1.2. Functional and non-functional requirements
    1.3. Explanation of choices for the technology stack (databases, programming languages, frameworks, etc.).
2. **System architecture**
    2.1. Introduction to the microservices architecture + schema of the whole system
    2.2. Microservice 1 description
    2.3. Microservice 2 description
    2.4. Microservice 3 description
    2.5. …
    2.6. Communication between microservices
    2.7. Description of the patterns and techniques used in the project.
        2.7.1. CQRS

2.7.2. Immutable data (Tombstone pattern and Snapshot pattern)
2.7.3. Idempotence
2.7.4. Commutative message handlers
2.7.5. Saga pattern
2.7.6. Caching
2.7.7. …

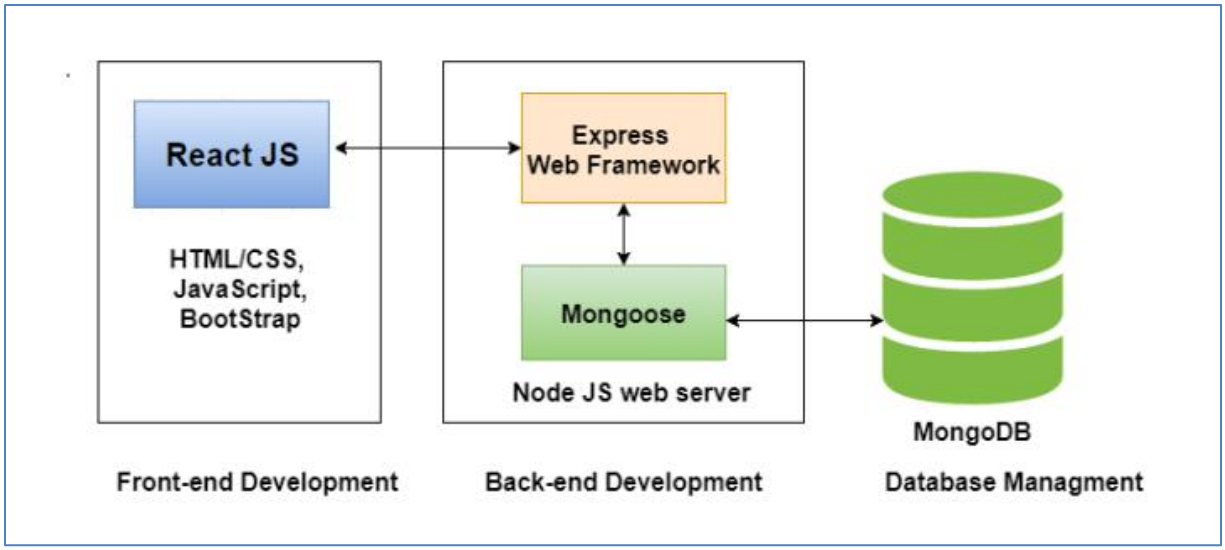
3. **Deployment**
   3.1. Introduction to the cloud deployment
   3.2. Description of used technologies
   3.3. CI/CD pipeline description
   3.4. Monitoring and logging of the deployed system
4. **Project management and team collaboration**
   4.1. Introduction to the project management and team collaboration
   4.2. Description of the methods used during the project.
   4.3. Versioning strategies for the source code, databases, and APIs
   4.4. Documentation strategy
5. **Conclusion**
   5.1. Advantages and challenges of the distributed systems (microservices architecture)
   5.2. Pros and cons of used patterns like CQRS etc.
   5.3. Scalability
   5.4. Possible improvements
6. **References**
7. **Appendix**

Examples of the technology stack

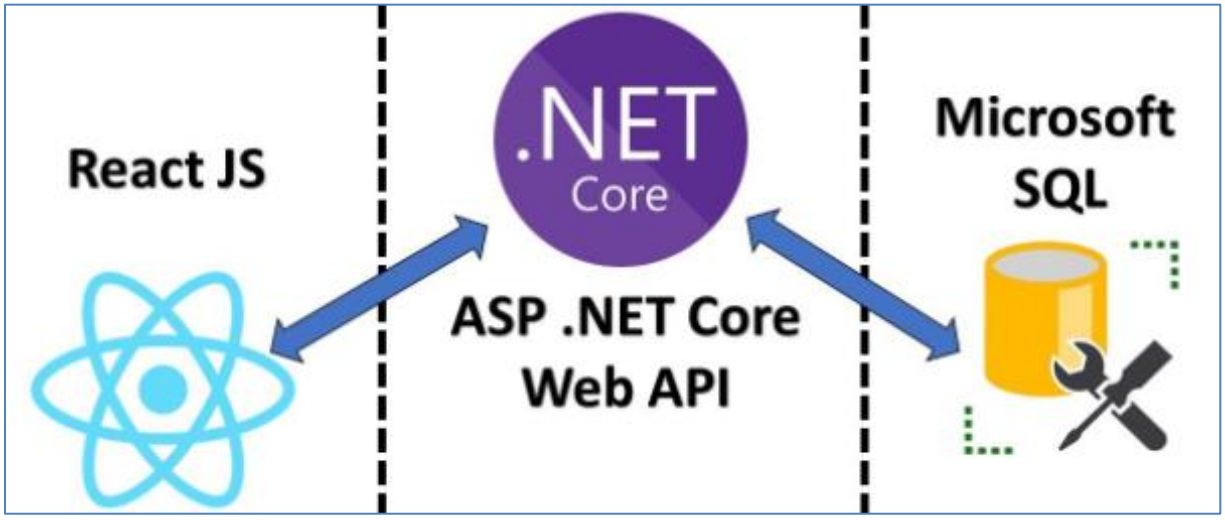
1. **React.js + Spring Boot + MySQL**



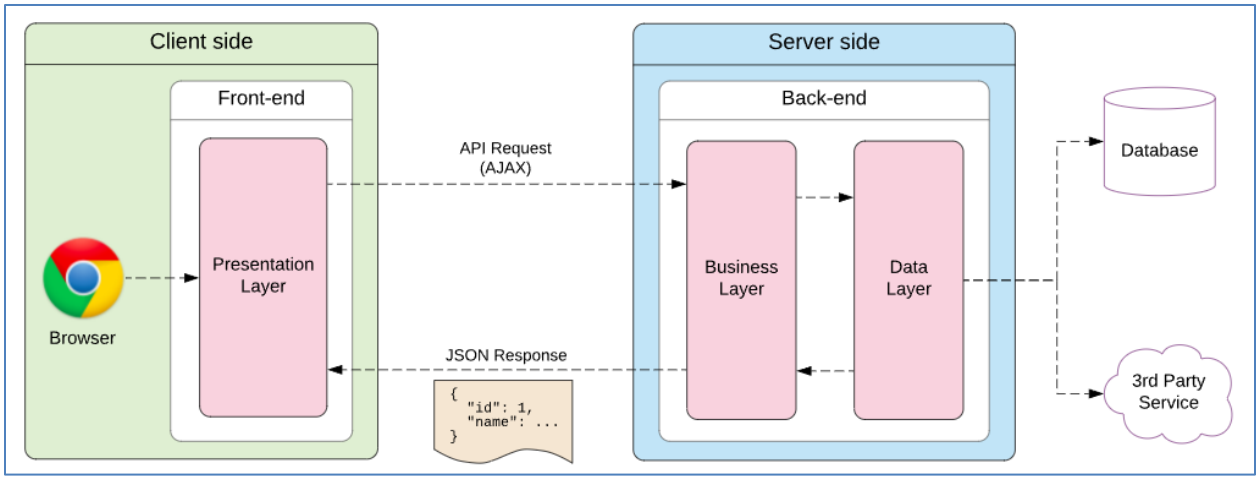
2. **React.js + Node.js + MongoDB (MERN stack):**



3. **React.js + ASP .NET + SQL Server:**

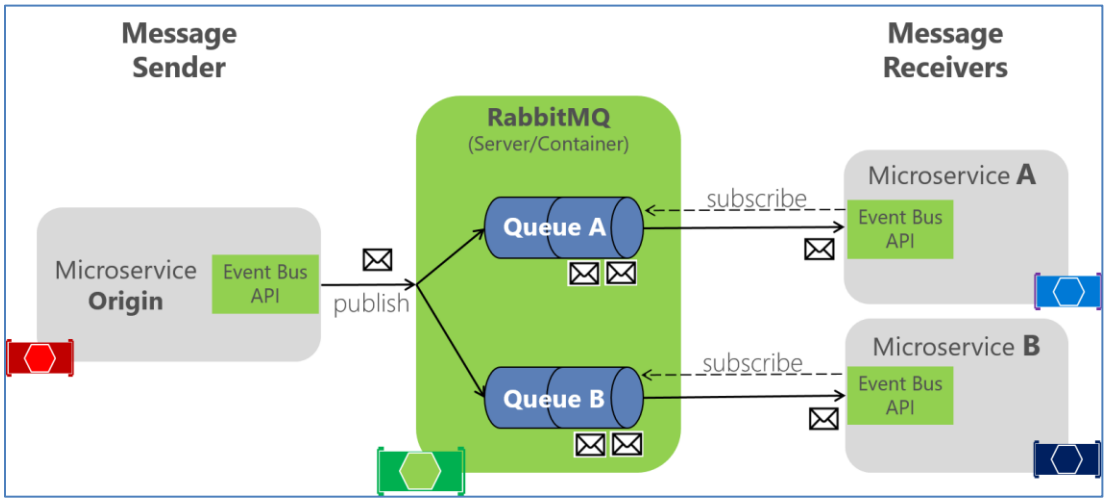


Frontent app

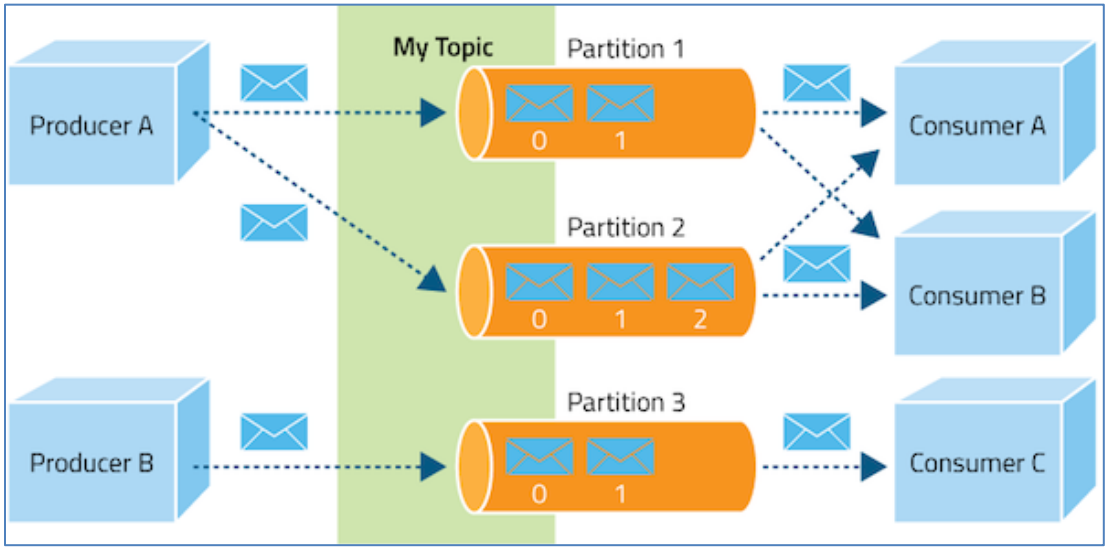


Message brokers

RabbitMQ



Kafka



Databases

- Redis
- MongoDB
- MySQL
- SQL Server
- Postgres
- Elastic Search
- Neo4j
- …