Template

January 2023

Solution Strategy

Motivation

Currently our system is designed as a monolith, which tightly couples our services. In the future, we want to have a loosely coupled microservice architecture, in order to be more resilient to errors and to increase our developing flexibility.

Technological decisions: We will use Docker and Kubernetes for containerization and service orchestration, making our services highly scalable and portable.

Top-level system decomposition: We will use the Strangler application pattern to continuously decompose our monolith into multiple microservices.

Achieving key quality goals: Asynchronous communication and horizontal scaling will ensure resilience, scalability and flexibility. We will use Kafka for asynchronous communication.

Organizational decisions: Our teams will work as agile SCRUM-teams to achieve incremental migration over multiple cycles.

Goal	Architectural Approach
Loose Coupling	We will strip each core functionality into an own microservice step-by-step to dissolve our monolithic architecture into a microservice architecture
Scalability	Each of our services will have their own database and server, allowing them to scale easily
Resilience	Loose coupling will make our services more resilient to downtimes and errors
Performance	We can increase performance by independently scaling our microservices, to improve their response times and possible workloads. Service-caching using Redis decreases latencies.
Flexibility	Having multiple microservices allows us to change our system independent of other system components, thus increasing our flexibility.

Building Block View

Whitebox Overall System

Top-level decomposition

Motivation
The following diagram shows the top-level view of our system. The target system should be composed of multiple loosely-coupled microservices.

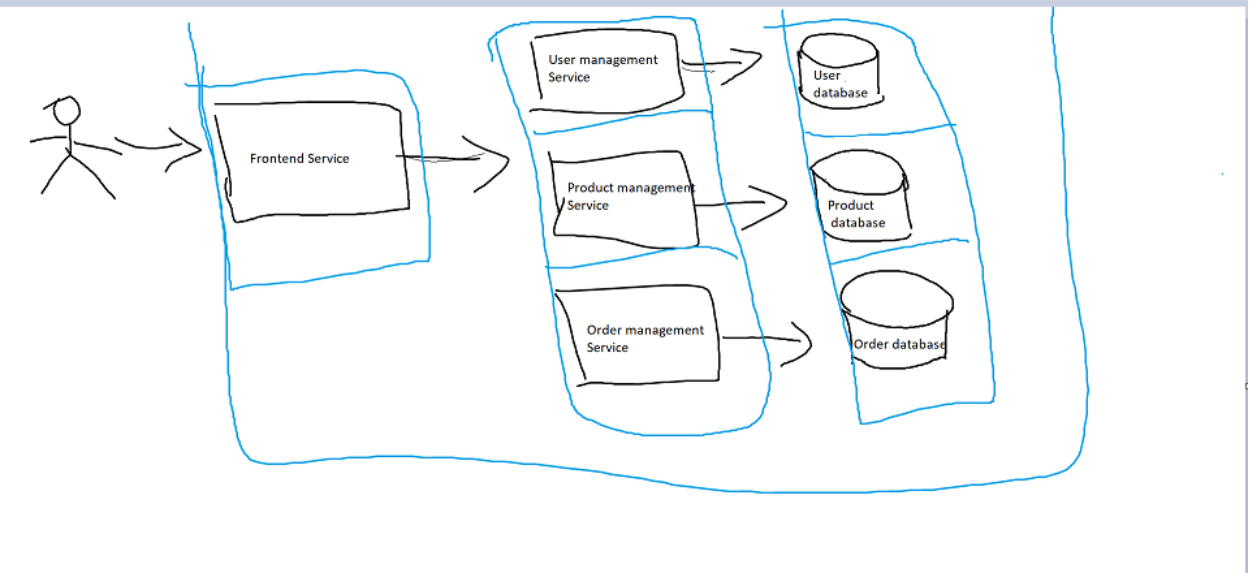
Contained Building Blocks
Frontend Service: Responsible for displaying our web shop to users via a browser.

User Management Service: Responsible for user creation and maintenance, keeping users' data secure and up-to-date.

Product Management Service: Responsible for keeping track of our warehouse status and product details.

Order Management Service: Responsible for order creation and completion.

Important Interfaces
REST API: Responsible for connecting our services, which need to communicate with each other to provide all intended functionalities.



Level 2

White Box *<building block 1>*

<white box template>

White Box *<building block 2>*

<white box template>

...

White Box *<building block m>*

<white box template>

Architecture Decisions

The following architectural decisions have been made.

ADR 1: Strangler Application Pattern

Context: Our monolith is difficult to scale and the tight coupling makes our system prone to downtimes if one part of our system fails. Using the Strangler application pattern we can gradually migrate to independent services, without risking availability issues with our existing monolith.

Consequences: Reducing risks by gradually moving towards a more independent architecture.

ADR 2: Docker and Kubernetes

Context: In order to achieve the migration, we decided to use Docker and Kubernetes, as they provide a stable and industry standard environment which increases our scalability and portability of our services.

Consequences: Rapid deployment, easy scalability. A drawback is that it increases our system complexity.

ADR 3: Asynchronous communication

Context: To increase resilience and to minimize bottlenecks, we require asynchronous communication, for which Kafka is the best fit. Kafka's message queueing solution will provide the required functionality.

Consequence: Asynchronous communication decouples our services and improves our system's resilience. A potential drawback is increased maintenance efforts, which come with adding another tool to our stack.

ADR 4: Service-Caching

Context: Frequently required services such as our product management service could increase latency when using our web shop. Caching can improve our system's performance, which is why we decided to use Redis to rapidly provide frequently used data.

Consequences: Redis will allow us to decrease system latency and thus improving the overall system performance. One potential issue is data consistency, as cached data might not be entirely up-to-date.