Frankfurt University of Applied Sciences
Fachbereich 2: Informatik und Ingenieurwissenschaften
Allgemeine Informatik (M.Sc.) / High Integrity Systems (M.Sc.)

Cloud Computing Project

Milestone 3 - Implementation of Cloud Transformation Scenario

Edward Späth - 1386244
Dennis Mark - 1384589
Vinay Duhan - 1475786
Jatender Singh Jossan - 1346747
Dominique Conceicao Rosario — 1399464

Lecturer: Henry-Norbert Cocos
Course: Cloud Computing
Winter Semester 24/25

Contents

List of Figures

1 Introduction and Goals

Describes the relevant requirements and the driving forces that software architects and development team must consider. These include

- underlying business goals,
- essential features,
- essential functional requirements,
- quality goals for the architecture and
- relevant stakeholders and their expectations

Requirements Overview

Contents

Short description of the functional requirements, driving forces, extract (or abstract) of requirements. Link to (hopefully existing) requirements documents (with version number and information where to find it).

Motivation

From the point of view of the end users a system is created or modified to improve support of a business activity and/or improve the quality.

Form

Short textual description, probably in tabular use-case format. If requirements documents exist this overview should refer to these documents.

Keep these excerpts as short as possible. Balance readability of this document with potential redundancy w.r.t to requirements documents.

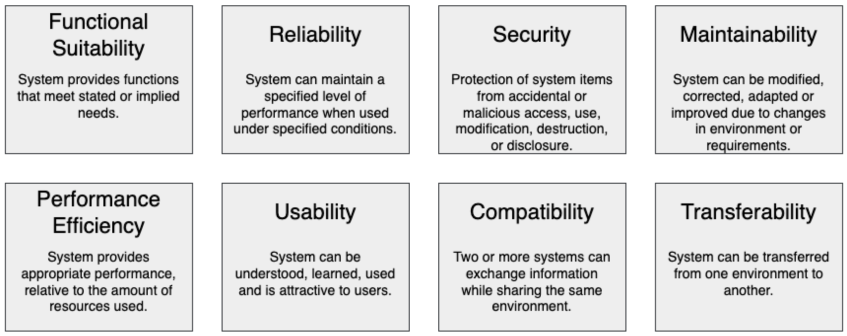
See Introduction and Goals in the arc42 documentation.

Quality Goals

Contents

The top three (max five) quality goals for the architecture whose fulfillment is of highest importance to the major stakeholders. We really mean quality goals for the architecture. Don't confuse them with project goals. They are not necessarily identical.

Consider this overview of potential topics (based upon the ISO 25010 standard):



ISO 25010 Quality Characteristics

Motivation

You should know the quality goals of your most important stakeholders, since they will influence fundamental architectural decisions. Make sure to be very concrete about these qualities, avoid buzzwords. If you as an architect do not know how the quality of your work will be judged…

Form

A table with quality goals and concrete scenarios, ordered by priorities

Stakeholders

Contents

Explicit overview of stakeholders of the system, i.e. all person, roles or organizations that

- should know the architecture
- have to be convinced of the architecture
- have to work with the architecture or with code
- need the documentation of the architecture for their work
- have to come up with decisions about the system or its development

Motivation

You should know all parties involved in development of the system or affected by the system. Otherwise, you may get nasty surprises later in the development process. These stakeholders determine the extent and the level of detail of your work and its results.

Form

Table with role names, person names, and their expectations with respect to the architecture and its documentation.

Role/Name	Contact	Expectations
<Role-1>	*<Contact-1>*	*<Expectation-1>*
<Role-2>	*<Contact-2>*	*<Expectation-2>*

2 Architecture Constraints

Contents

Any requirement that constraints software architects in their freedom of design and implementation decisions or decision about the development process. These constraints sometimes go beyond individual systems and are valid for whole organizations and companies.

Motivation

Architects should know exactly where they are free in their design decisions and where they must adhere to constraints. Constraints must always be dealt with; they may be negotiable, though.

Form

Simple tables of constraints with explanations. If needed you can subdivide them into technical constraints, organizational and political constraints and conventions (e.g. programming or versioning guidelines, documentation or naming conventions)

See Architecture Constraints in the arc42 documentation.

3 System Scope and Context

Contents

This section defines the scope and context of the Webshop system, a platform designed to display products to users, process payments, and confirm orders. The system retrieves product data from an Azure-hosted database, integrates with Stripe for payment processing, and uses an Azure email service to send order confirmations. It outlines the external entities—users, the Azure database, Stripe, and the Azure email service—and specifies the business and technical interfaces connecting them to the Webshop.

Motivation

Understanding the Webshop's and its external entities' interfaces is crucial for stakeholders to make informed architectural decisions. Clear boundaries ensure alignment on what the system handles (e.g., product display and payment) versus what it relies on externally (e.g., payment processing via Stripe), guiding both development and deployment decisions.

Form

The business context will be presented with a context diagram showing the Webshop as a black box linked to its external partners, alongside a table listing communication partners, inputs, and outputs. The technical context will use a UML deployment diagram to illustrate the system's technical connections, supplemented by a mapping table tying domain inputs/outputs to specific channels.

Business Context

Contents

The Webshop system interacts with several external entities: (1) Users, who browse products, submit orders, provide payment information, and receive order confirmations; (2) Azure-hosted Database, which provides product data; (3) Stripe, which processes payment transactions; and (4) Azure Email Service, which delivers order confirmation emails. This subsection specifies the domain-specific inputs and outputs exchanged between the Webshop and these partners.

Motivation

Defining these interactions ensures stakeholders understand the Webshop's core business functions—displaying products, processing orders, and confirming purchases—and its dependencies on external systems for data, payments, and notifications.

Form

The context diagram below depicts the Webshop system and its external interactions. The table details the inputs and outputs for each communication partner.

Communication Partner	Inputs	Outputs
Users	User Interface	Product requests, order details, payment info
Azure Database	Product data, order storage requests	Product info
Stripe	Payment info	Payment success/failure response
Azure Email Service	Order details	email delivered to users

Table 2: Inputs and Outputs for Webshop Communication Partners

Technical Context

Contents

This subsection details the technical interfaces connecting the Webshop system to its environment, including the channels, protocols, and hardware used. The Webshop operates as a web application hosted on Azure, communicating with users via HTTP/HTTPS over the internet, accessing the Azure-hosted Database through REST API calls, integrating with Stripe via HTTPS for payment processing, and utilizing Azure Email Service for SMTP-based email delivery. It

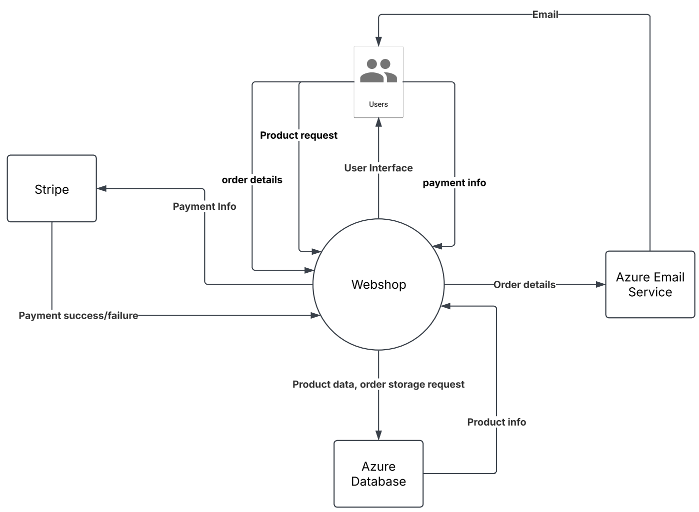


Figure 1: Context Diagram of the Webshop System and Its External Entities

maps these technical connections to the business inputs and outputs described in the Business Context.

Motivation

Understanding these technical interfaces is critical for infrastructure designers and developers to ensure reliable connectivity, secure data transmission, and scalable deployment of the Webshop. It informs decisions about hosting, network configuration, and integration with external services like Stripe and Azure.

Form

The UML deployment diagram below illustrates the Webshop's technical architecture and its connections to external entities. The table maps the domain-specific inputs and outputs to their technical channels and protocols.

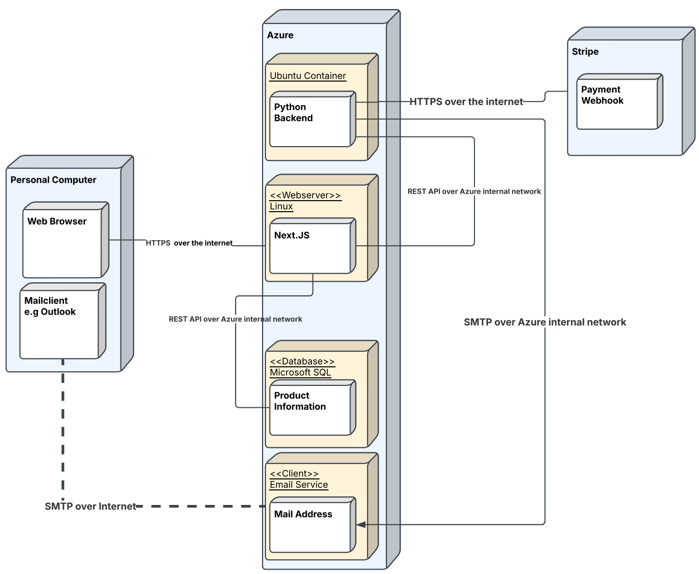


Figure 2: UML Deployment Diagram of the Webshop Technical Architecture

Input/Output	Channel/Protocol	Description (Optional)
Product requests, order details, payment info (Users → Webshop)	HTTPS over the Internet	User interactions via web browser
Product listings, order confirmation (Webshop → Users)	HTTPS over the Internet	Web app responses
Product data, order storage requests (Webshop → Azure Database)	REST API over Azure internal network	Database queries and updates
Product info (Azure Database → Webshop)	REST API over Azure internal network	Data retrieval for product display
Payment info (Backend → Stripe)	HTTPS over the Internet	Payment processing requests
Payment webhook (Stripe → Backend)	HTTPS over the Internet	Payment status updates via webhook
Order details (Webshop → Azure Email Service)	SMTP over Azure internal network	Email notification setup
Email delivered to users (Azure Email Service → Users)	SMTP over Internet	Email delivery to users

Table 3: Mapping of Webshop Inputs/Outputs to Technical Channels

4 Solution Strategy

Contents

A short summary and explanation of the fundamental decisions and solution strategies, that shape system architecture. It includes

- technology decisions
- decisions about the top-level decomposition of the system, e.g. usage of an architectural pattern or design pattern
- decisions on how to achieve key quality goals
- relevant organizational decisions, e.g. selecting a development process or delegating certain tasks to third parties.

Motivation

These decisions form the cornerstones for your architecture. They are the foundation for many other detailed decisions or implementation rules.

Form

Keep the explanations of such key decisions short.

Motivate what was decided and why it was decided that way, based upon problem statement, quality goals and key constraints. Refer to details in the following sections.

See Solution Strategy in the arc42 documentation.

5 Building Block View

This section shows the static decomposition of the system into building blocks (e.g. modules, components, subsystems, classes, interfaces, packages, libraries, frameworks, layers, partitions, tiers, functions, operations, . . .) including their relationships and associations. It helps to maintain an overview of your source code by making its structure understandable through abstraction.

Frontend

Component-Based Frontend Architecture

The Vivendo webshop frontend follows a **component-based modular frontend design**. The different modules are interconnected to provide a seamless shopping experience. The primary technologies used include **Next.js, Tailwind CSS**, API integration and Context API for state management. This approach ensures:

- Clear separation of concerns through distinct modules.
- Reusability of components across different sections.
- Better maintainability and scalability.
- Efficient state and API management.

The architectural overview is depicted in the figure below:

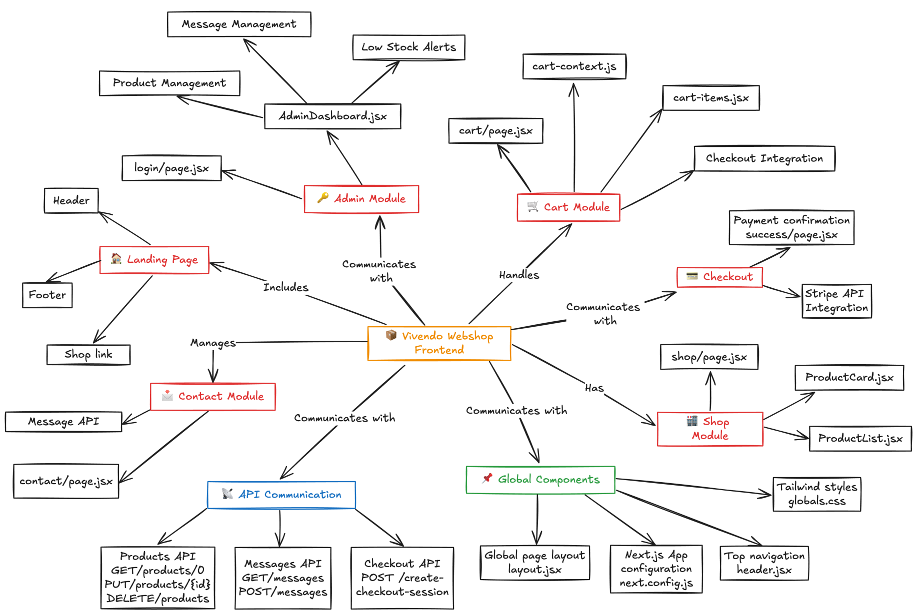


Figure 3: Component-Based Frontend Architecture

Core Frontend Modules and Interactions

The system consists of several modules:

- **Landing Page Module**: Includes key UI components such as the header, footer and navigation links.
- **Admin Module**: Manages the admin dashboard, including product management, low stock alerts and message management. The admin

module is responsible for managing products, messages, and stock alerts. It is connected to the **Admin Dashboard** component, which communicates with the API layer.

- **Cart Module**: Handles cart functionality, including cart context, cart items and checkout integration. The cart module manages cart-related functionalities and state using `cart-context.js`. It also connects with the checkout module to handle Stripe payments.
- **Shop Module**: Displays product listings and product cards with interactivity.
- **Checkout Module**: Integrates with Stripe API for handling payments and success confirmations.
- **Contact Module**: Manages user interactions via the contact page and handles message submissions. It connects with the API layer to store and retrieve messages.
- **API Communication Layer**: Manages requests to the backend services, including product APIs, messages API and checkout API. This layer serves as the middleware between the frontend and backend, making requests via REST APIs. It handles product data, message submissions, and checkout sessions.
- **Global Components**: Includes shared layout elements, styles and configurations.

Backend

6 Runtime View

Contents

The runtime view describes concrete behavior and interactions of the system's building blocks in form of scenarios from the following areas:

- important use cases or features: how do building blocks execute them?
- interactions at critical external interfaces: how do building blocks cooperate with users and neighboring systems?
- operation and administration: launch, start-up, stop
- error and exception scenarios

Remark: The main criterion for the choice of possible scenarios (sequences, workflows) is their **architectural relevance**. It is **not** important to describe a large number of scenarios. You should rather document a representative selection.

Motivation

You should understand how (instances of) building blocks of your system perform their job and communicate at runtime. You will mainly capture scenarios in your documentation to communicate your architecture to stakeholders that are less willing or able to read and understand the static models (building block view, deployment view).

Form

There are many notations for describing scenarios, e.g.

- numbered list of steps (in natural language)
- activity diagrams or flow charts
- sequence diagrams
- BPMN or EPCs (event process chains)
- state machines
- ...

See Runtime View in the arc42 documentation.

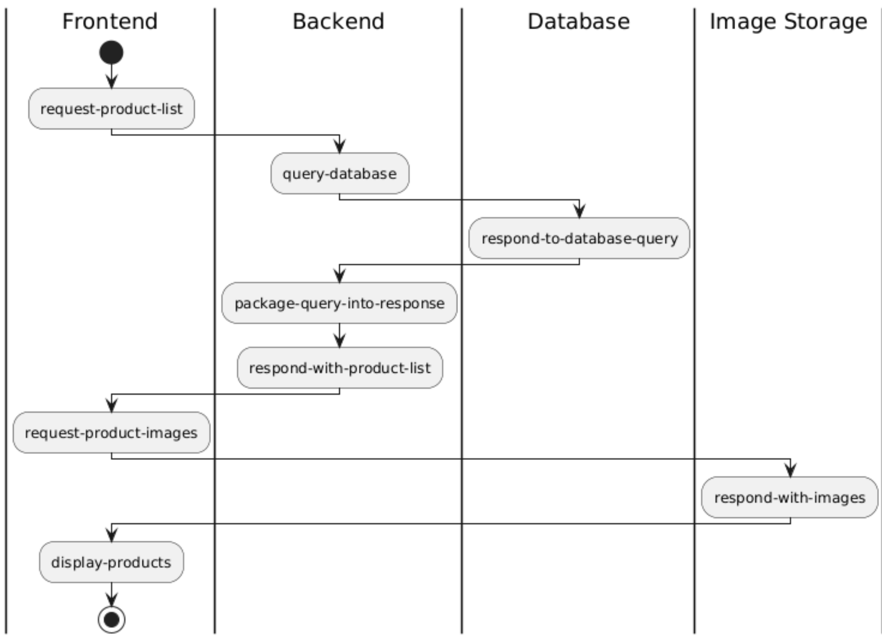
<Runtime Scenario 1>

- *<insert runtime diagram or textual description of the scenario>*
- *<insert description of the notable aspects of the interactions between the building block instances depicted in this diagram.>*

<Runtime Scenario 2>

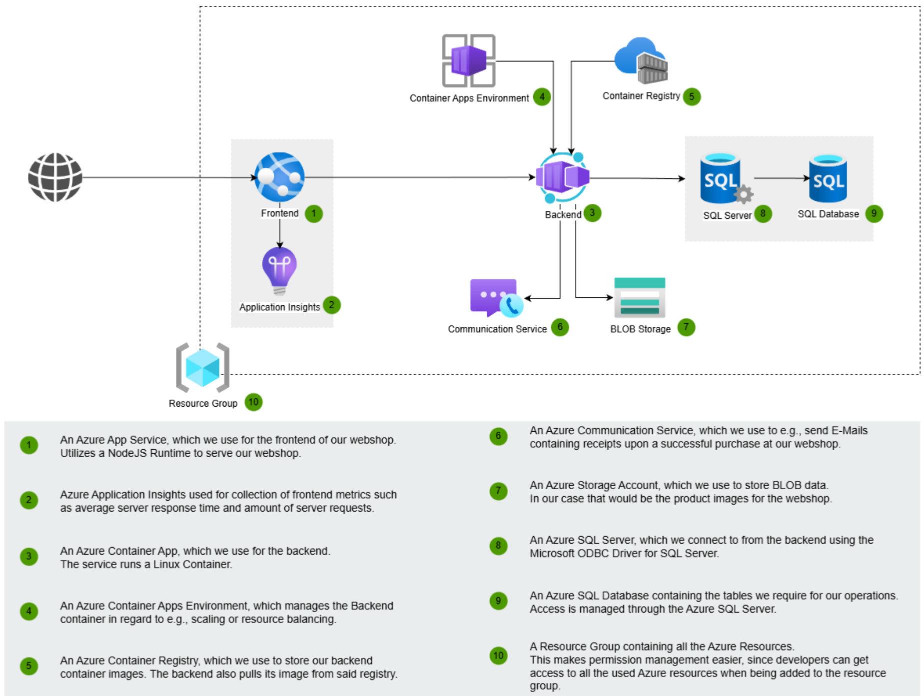
. . .

<Runtime Scenario n>



7 Deployment View

Microsoft Azure Deployment



Motivation

Instead of hosting the application locally on-premise, we instead opted for hosting it on the cloud. This has multiple advantages such as not having to manage our own hardware as well as allowing more efficient resource utilization by using a pay-as-you-go method of billing, rather than investing into our own hardware outright.

We chose to go forward with Microsoft Azure as our Cloud Service Provider, because they offer advanced features, which we can make good use of, being reliable and easy to deploy with.

Now follow rationales for select services, where multiple Azure offerings could have been considered instead.

Component	Rationale
Frontend	The frontend is served using NodeJS, therefore we deployed it using an Azure App Service, since that is the most convenient way of deploying NodeJS applications on Azure.
Backend	The python backend was designed as a Docker container in order to have e.g., rather simple horizontal scaling. Therefore, it is suitable to use an Azure Container App to deploy the container into the cloud.
Database	As our database, we are utilizing the SaaS offering of Azure SQL Database, since it is generally more reliable and more comfortable to work with, compared to hosting your own database with a PaaS offering. Furthermore, we have personal preference toward managing relational databases, rather than non-relational databases. Otherwise, we could have opted for non-relational database SaaS offering such as Azure Cosmos DB.

Quality and/or Performance Features

Component	Quality and/or Performance Feature(s)
Frontend	The Azure App Service features 99.95% guaranteed uptime as per the service-level agreement (SLA), which is crucial for fulfilling our requirement of having high availability.
Backend	The Azure Container App features the ability of creating replicas, allowing for horizontal scaling, which can even be scaled automatically based on e.g., amount of concurrent incoming requests or CPU usage, allowing for greater performance.

Component	Quality and/or Performance Feature(s)
Database	The Azure SQL Database SaaS offering provides 99.99% availability as per the service-level agreement. This option allows for seamless upgrades to more premium service tiers, allowing for even availability guarantees exceeding 99.99%. Furthermore, more premium service tiers allow for using features, such as geo-redundant backup storage, which replicates additional backups to a physical location hundreds of miles away from the primary region, making it even more unlikely to lose significant amounts of data, even in catastrophic cases.
BLOB Storage	With Azure Storage Accounts, which we use for the blob storage, we have a 99.9% availability as per the service-level agreement. More premium options allow for going as high as 99.99% availability.
Azure Communication Services	Azure Communication Services have an availability of 99.9% guaranteed availability as per the service-level agreement, making it so that our mailing service is highly available.

Mapping

(Mapping from components in deployment view to components in building block view!)

8 Crosscutting Concepts and Architectural Decisions

Content

This section describes overall, principal regulations and solution ideas that are relevant in multiple parts (= crosscutting) of our system and its architecture. Such concepts and decisions are often related to multiple building blocks and can include many different topics, such as

- models, especially domain models
- architecture or design patterns
- rules for using specific technology
- principal, often technical decisions of an overarching nature
- implementation rules

Domain Model & Data Structures

The Vivendo Webshop frontend relies on several key data models that keep information consistent across different parts of the application:

- **Product Model**: This structure includes fields such as the product ID, name, description, price, and stock level. Any part of the webshop that displays or updates product information—including the Shop Module and the Admin Dashboard—uses this same model.
- **Cart Item Model**: This extends the Product Model by incorporating cart-related details, such as quantity or item-level notes. The Cart Context employs this structure to ensure that updates to product attributes, like pricing or availability, remain in sync with a user's cart.
- **Message Model**: This standardized format describes user inquiries with fields such as subject, sender information, and message text. Both the Contact Module and the Admin Dashboard interact with this single model to create, display, and manage messages consistently.

We maintain these data models as shared sources of truth (SSOT). Splitting them into different product models for the shop and the admin area would risk creating mismatches if one model would be updated without the other. It would also complicate maintenance, since any field changes would need to be replicated in multiple places. By using these core structures throughout the system, we reduce the likelihood of data inconsistencies and simplify ongoing feature development.

UI & UX Conventions

All pages in the Vivendo Webshop share a unified layout and design language. A global layout component (`layout.jsx`) includes site-wide elements such as the header, footer and navigation links. This approach guarantees that the user experience remains consistent across all modules. Tailwind CSS is used to provide a utility-first styling approach so that developers can quickly apply spacing, color, and typography classes to maintain uniform visuals.

We selected Tailwind because of its flexibility and the minimal overhead it adds, Although other frameworks like Bootstrap might provide predefined components, those often require extensive overrides to achieve a specific kind of identity.

Routing & Folder Structure (Next.js)

Our Next.js setup uses file-based routing to map files within the `pages/` directory to distinct routes. This arrangement makes the URL structure predictable for both developers and users. Public pages - including the shop, cart, and contact form - reside under straightforward displayed paths, while administrative features such as login or the Admin Dashboard are kept separate to detach sensitive functionality.

We chose Next.js because it offers server-side rendering and built-in image optimization. These features are valuable for an online shop (e.g. Vivendo) where both SEO and performance are important. Besides alternative solutions, Next.js fits best with the platform's requirement for dynamic content, simple configuration and a strong user experience across devices.

Shared State Management

React Context API is used to share and manage global data. For example, the Cart Context holds a user's cart items and provides methods to add, remove, or update them. By centralizing cart logic, all components in the webshop - from product pages to checkout flows - operate on the same state and can render consistent information.

Security & Authentication

The webshop uses an authentication-based route for the dashboard including all administrative features. This ensures that unauthorized individuals cannot edit products or read messages. The current approach checks whether the username and password match the required values and if valid, stores them in a local storage to simulate a logged-in state. This design is sufficient for a simple prototype but would need to be replaced with a more secure solution in a production environment.

For payment processing the system depends on Stripe to handle sensitive credit card data. This reliance on a secure external provider reduces compliance overhead for our team and limits the exposure of critical payment details in our infrastructure. We selected Stripe based on its popularity, strong security posture and clear documentation for the integration with Next.js.

Error Handling & Logging

React error boundaries catch unexpected exceptions in key components, preventing the entire application from failing if one feature encounters a problem.

Consolidated Architectural Decisions

- **Next.js** is our framework for delivering server-rendered pages and handling routing automatically.
- **Stripe** is used for payment processing, removing the need to manage credit card data in our own systems.
- **React Context** manages global state, such as the shopping cart, in order to reduce boilerplate and maintain clarity.
- **Single Source of Truth** for data models avoids the duplication of field definitions across modules.

These decisions ensure that the our frontend remains flexible and performant through maintaining an architecture that is adaptable as the platform evolves.

9 Quality Requirements

Content

This section contains all quality requirements as quality tree with scenarios. The most important ones have already been described in section 1.2. (quality goals)

Here you can also capture quality requirements with lesser priority, which will not create high risks when they are not fully achieved.

Motivation

Since quality requirements will have a lot of influence on architectural decisions you should know for every stakeholder what is really important to them, concrete and measurable.

See Quality Requirements in the arc42 documentation.

Quality Tree

Content

The quality tree (as defined in ATAM – Architecture Tradeoff Analysis Method) with quality/evaluation scenarios as leafs.

Motivation

The tree structure with priorities provides an overview for a sometimes large number of quality requirements.

Form

The quality tree is a high-level overview of the quality goals and requirements:

- tree-like refinement of the term "quality". Use "quality" or "usefulness" as a root
- a mind map with quality categories as main branches

In any case the tree should include links to the scenarios of the following section.

Quality Scenarios

Contents

Concretization of (sometimes vague or implicit) quality requirements using (quality) scenarios.

These scenarios describe what should happen when a stimulus arrives at the system.

For architects, two kinds of scenarios are important:

- Usage scenarios (also called application scenarios or use case scenarios) describe the system's runtime reaction to a certain stimulus. This also includes scenarios that describe the system's efficiency or performance. Example: The system reacts to a user's request within one second.
- Change scenarios describe a modification of the system or of its immediate environment. Example: Additional functionality is implemented or requirements for a quality attribute change.

Motivation

Scenarios make quality requirements concrete and allow to more easily measure or decide whether they are fulfilled.

Especially when you want to assess your architecture using methods like ATAM you need to describe your quality goals (from section 1.2) more precisely down to a level of scenarios that can be discussed and evaluated.

Form

Tabular or free form text.

10 Risks and Technical Debts

Contents

A list of identified technical risks or technical debts, ordered by priority

Motivation

"Risk management is project management for grown-ups" (Tim Lister, Atlantic Systems Guild.)

This should be your motto for systematic detection and evaluation of risks and technical debts in the architecture, which will be needed by management stakeholders (e.g. project managers, product owners) as part of the overall risk analysis and measurement planning.

Form

List of risks and/or technical debts, probably including suggested measures to minimize, mitigate or avoid risks or reduce technical debts.

See Risks and Technical Debt in the arc42 documentation.

11 Glossary

Contents

The most important domain and technical terms that your stakeholders use when discussing the system.

You can also see the glossary as source for translations if you work in multi-language teams.

Motivation

You should clearly define your terms, so that all stakeholders

- have an identical understanding of these terms
- do not use synonyms and homonyms

A table with columns <Term> and <Definition>.

Potentially more columns in case you need translations.

See Glossary in the arc42 documentation.

Term	Definition
Cloud Service Provider (CSP)	*A Cloud Service Provider (CSP) is a third-party company, which offers (paid) services in regard to cloud capabilities, be it compute, storage, management, and/or analytics. In our case, we are solely referring to public Cloud Service Providers, which provide these services to the public, opposed to offering services exclusive to one or multiple companies.*
Microsoft Azure ("Azure")	*Microsoft Azure is a public cloud service provider belonging to Microsoft.*
Service-Level Agreement (SLA)	*Service-level agreements are made between a (cloud) service provider and a customer. For our purposes, the main point of interest, is availability, where the (cloud) service provider guarantees a certain availability for a given service, allowing us to fulfill our own availability requirements when relying on said service.*
Horizontal Scaling	*Horizontal Scaling refers to improving performance by spinning up multiple instances, so that requests can be distributed among them, allowing for greater parallel processing, rather than just increasing hardware performance directly (Vertical Scaling).*

Term	Definition
Replica	*A replica in our case refers to additional copies of either Docker containers or database copies. The purpose is to provide increased performance (horizontal scaling), such as having multiple read-only replicas of a database, allowing for higher throughput of read operations, especially if additional replicas are spread geographically to decrease latency. Replicas also increases availability, especially if they are spread geographically, making it so that an instance of a service is running even if there were to be a data center failure at N-1 locations.*