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

Introduction and Goals **5**
Overview 5
Project Scope and Requirements 5
Quality Goals 5
Stakeholders 6

Constraints **7**
Technical Constraints 7
Organizational Constraints 7

3 System Scope and Context **8**
Business Context 8
Technical Context 9

4 Solution Strategy **11**
Flask for the backend 11
Next.js for the frontend 11

5 Building Block View **12**
System White Box 12
Motivation 13
Frontend 13
Component-Based Frontend Architecture 13
Core Frontend Modules and Interactions 14
Backend 15
E-Mail Service 16
Database 17
Image Storage 17
Important Interfaces 18
/product API Endpoint 18
/messages API Endpoint 18
/create-checkout-session API Endpoint 19
/webhook API Endpoint 20

6 Runtime View **22**
Display Product List 22
Cart Checkout Process 23
Creating/Updating/Deleting Products in Admin Panel 24

7 Deployment View **25**
Microsoft Azure Deployment 25
Motivation 25
Quality and/or Performance Features 26
Mapping 27
Quality and/or Performance Features 27

8 Crosscutting Concepts and Architectural Decisions **29**
Frontend 29
Domain Model & Data Structures 29
UI & UX Conventions 30
Routing & Folder Structure (Next.js) 30
Shared State Management 30
Security & Authentication 30
Error Handling & Logging 31
Consolidated Architectural Decisions 31
Backend 31
Domain Model & Data Structures 31
User Experience Concept 32
Implementation rules 32

9 Quality Requirements **34**
Quality Tree 34
Quality Scenarios 34

9 Risks and Technical Debts **36**

10 Glossary **38**

12 Information about Work **40**
Distribution of Report Work 40
Repository 40

List of Figures

1 Context Diagram of the Webshop System and Its External Entities 9
2 UML Deployment Diagram of the Webshop Technical Architecture 10
3 Whie Box View of the System 12
4 Component-Based Frontend Architecture 14
5 Backend Building Block View 16
6 Database Entity-Relationship Diagram 17
7 UML Swimlane Diagram for Displaying Product List 22
8 UML Swimlane Diagram for the Checkout Process 23
9 UML Swimlane Diagram for Creating, Updating, and Deleting Products via the Admin Panel 24
10 Azure Infrastructure Diagram 25
11 Backend Domain Model 31
12 Example of an database entity 32
13 Quality Tree for the Webshop System 35

Introduction and Goals

Overview

This project focuses on developing a **Webshop system** for **LowTech GmbH** using **cloud computing**. The objective is to create a **fully functional e-commerce platform** that is **scalable, highly available, and responsive**. The system follows a **Three-Tier Architecture** and is deployed on **Microsoft Azure**, leveraging cloud services for hosting, storage, and computing power.

Project Scope and Requirements

The Webshop system is designed to meet the following key requirements:

- **High Availability:** The system should have a **99.99% uptime**, ensuring minimal downtime.
- **Scalability:** It should dynamically scale to handle increased user traffic.
- **User-Friendly Interface:** The frontend should be **fast, responsive, and accessible on multiple devices**.
- **Separation of Concerns:**
  - **Frontend (User Interface):** Built using **Next.js** and **Tailwind CSS**.
  - **Backend (Logic and APIs):** Implemented using **Python Flask**, providing business logic and API endpoints.
  - **Database (Data Storage):** Uses **Microsoft Azure SQL Database** for structured storage of product, order, and inventory data.
- **Cloud-Based Deployment:** The system is hosted on **Microsoft Azure**, using its cloud services for computation and storage.
- **Payment System Integration:** The Webshop integrates with **Stripe API** for payment processing.
- **Modern Development Practices:** The project follows **best practices**, including **GitHub version control** and **CI/CD pipelines**.

Quality Goals

The system prioritizes the following **quality attributes**:

1. **Availability and Reliability:** Ensuring **99.99% uptime** with **Azure Load Balancer and failover mechanisms**.
2. **Scalability and Performance:** Supporting **horizontal scaling** using **Azure Container Apps**.

3. **Security and Compliance:** Protecting data using **Azure's encryption and identity management**.
4. **Maintainability and Extensibility:** Using **modular architecture** to facilitate future improvements.
5. **User Experience and Accessibility:** Delivering a **smooth and optimized shopping experience** across different devices.

Stakeholders

The key stakeholders in the project are:

Role	**Who They Are**	**What They Expect**
CEO	LowTech GmbH CEO	A modern, reliable cloud-based Webshop.
Development Team	Project Team	Implementing the Webshop with cloud best practices.
End Users	Customers	A fast and smooth shopping experience.
Cloud Provider	Microsoft Azure	Hosting, scaling, and security management.

Table 1: Stakeholders and their expectations

Constraints

Technical Constraints

The Webshop system is built with the following fixed technologies:

Category	Chosen Technology
Frontend	Next.js (React Framework) with Tailwind CSS
Backend	Python Flask (REST API implementation)
Database	Microsoft Azure SQL Database
Storage	Azure Blob Storage for product images
Hosting	Azure App Services and Azure Container Apps
Authentication	No authentication required (demo version)
Version Control	GitHub repository with structured commit history
Scalability	Azure auto-scaling and load balancing

Table 2: Technical Constraints for Webshop Development

Organizational Constraints

- **Cloud Provider:** The project is hosted on **Microsoft Azure**.

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, 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.

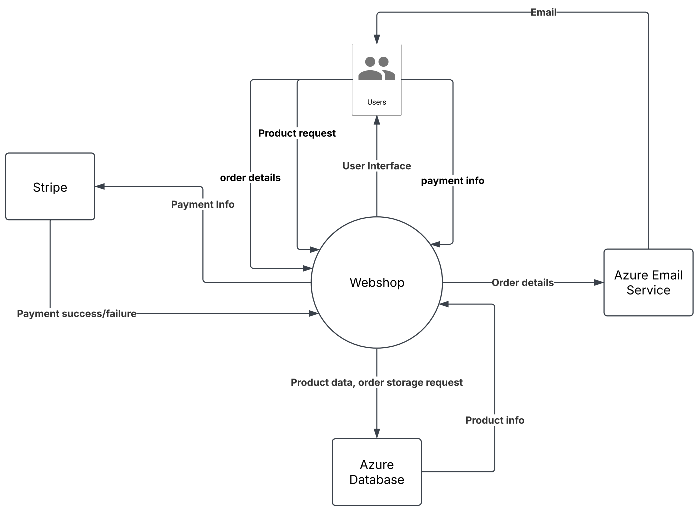


Figure 1: Context Diagram of the Webshop System and Its External Entities

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 3: 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 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.

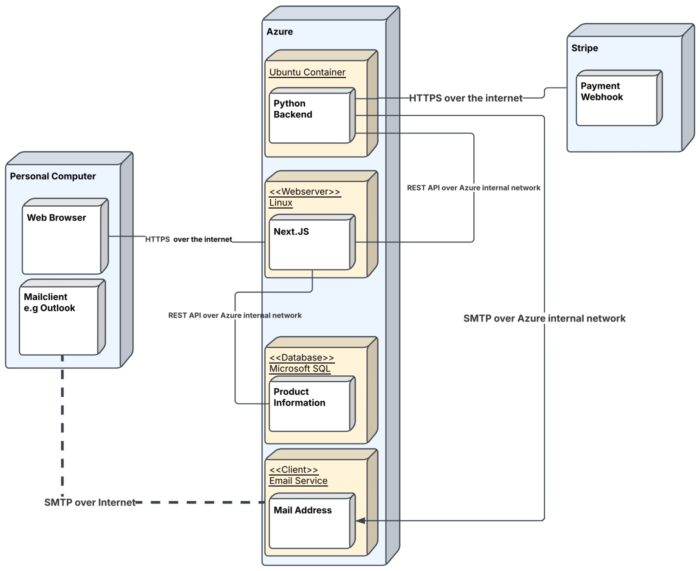


Figure 2: UML Deployment Diagram of the Webshop Technical Architecture

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 4: Mapping of Webshop Inputs/Outputs to Technical Channels

4 Solution Strategy

Content

This section provides a brief overview of the technologies used in the development of the project, along with the design patterns applied.

Flask for the backend

We chose Flask for the backend because it is a lightweight and flexible micro web framework, perfect for handling routing and database interactions with minimal overhead. Flask's simplicity allows us to build and maintain our web application efficiently, which is ideal given that our project is a small-scale webshop with limited complexity. Additionally, Flask's modular design makes it easy to extend with necessary tools or libraries without being weighed down by unnecessary components, making it a practical choice for our specific use case.

There was no need to implement a specific design pattern for the backend due to the simplicity and small scale of the application. The straightforward nature of the project allowed us to keep the backend lightweight and functional without the overhead of a complex architecture.

Next.js for the frontend

The main advantage of using Next.js over React is its built-in support for API routes, enabling seamless integration of backend functionality within the same framework. This allows us to easily perform CRUD operations (POST, GET, PUT, DELETE) without needing a separate server setup. Additionally, like React, Next.js empowers us to create dynamic websites with smooth user experiences, while offering features like server-side rendering and static site generation for better performance and SEO optimization.

In Next.js we decided to use the App Router Structure for the frontend due to its simplicity in establishing a clear and organized file-based routing structure. Next.js automatically handles the routing by serving the requested files based on the folder structure, which reduces the need for manual routing configuration. This makes development more efficient and ensures that our project remains maintainable as it scales.

5 Building Block View

In this chapter, we present the building block view of our application, illustrating how each component is structured and integrated within the system. The aim is to provide a comprehensive overview of the major software building blocks, focusing on their responsibilities and interactions.

System White Box

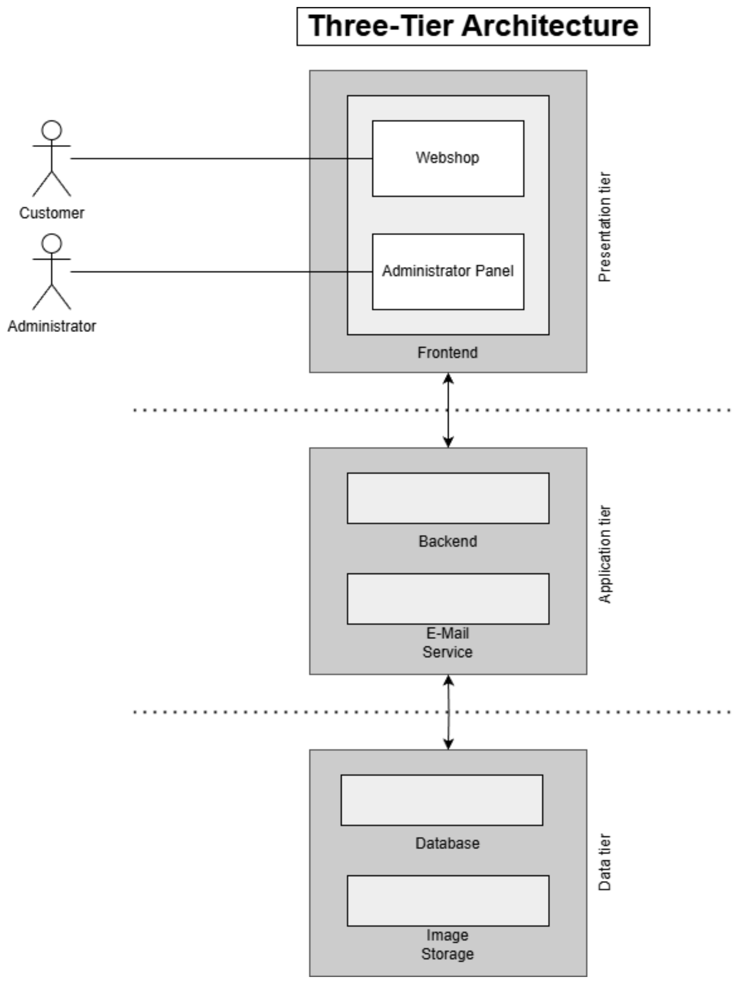


Figure 3: Whie Box View of the System

Motivation

We utilize a three-tier architecture for our system. This allows us to split the system into three tiers with different responsibility. The presentation tier is used as the interface between the users and the system. Meanwhile, the application tier is used to process requests from the presentation tier. Lastly, the data tier stores the actual data of the system.

The advantage of such a distribution is greater separation of concerns, where tasks of each layer are clearly separated. Furthermore, having a middle layer between the presentation tier and the data tier, enhances reliability, since modifying operations have to go through the application tier, which can make sure to validate requests before any data is impacted.

Now follows an explanation of the individual building blocks of the system.

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 4 below.

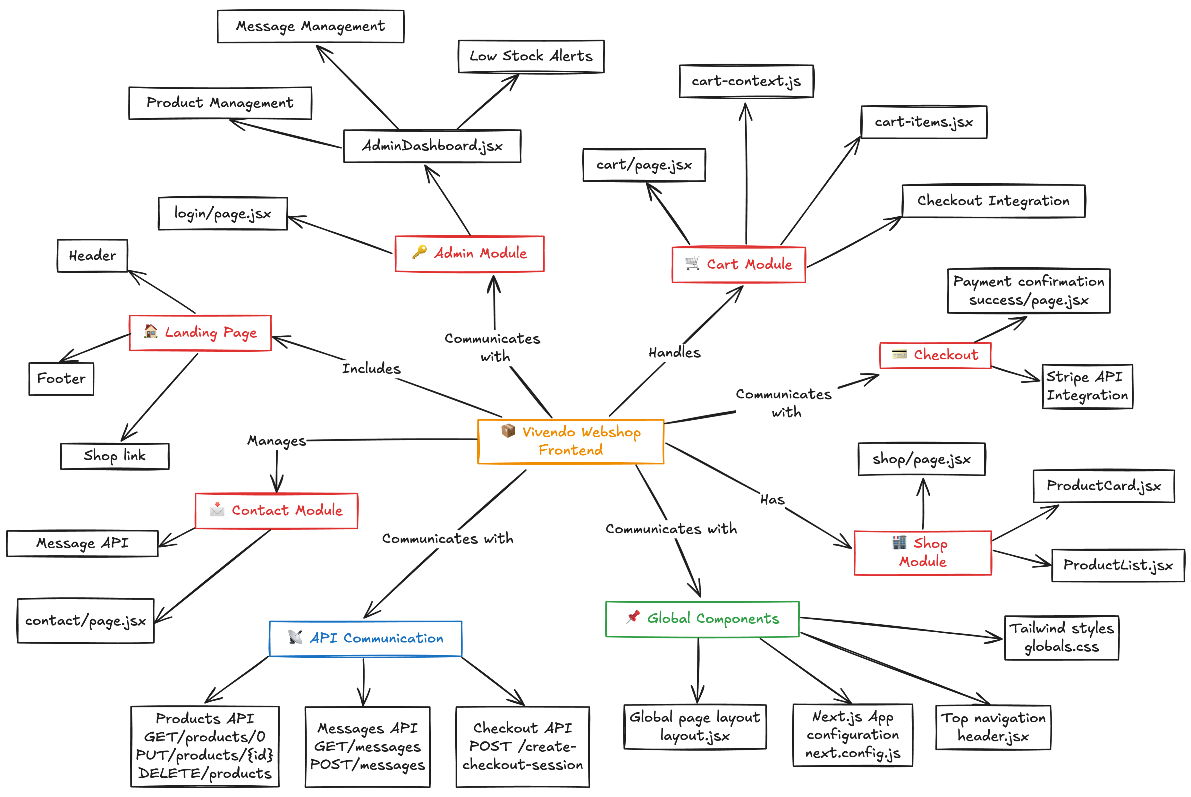


Figure 4: Component-Based Frontend Architecture

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

The backend, whose building block view can be seen in fig. 5, consists of the following elements:

- **Docker Container**: Runs the backend web app.
- **Container Registry**: Hosts the images used by the Docker container.
- **Container Apps Environment**: Manages the Docker container in regard to e.g., replicas and load balancing.

The Docker container consists of the following elements:

- **Python Web App**: Contains all the functions of the backend.
- **Gunicorn**: Serves the python web app, allowing for multiple “workers”, i.e., threads, making use of the multiple CPU cores available from the container app.
- **Microsoft ODBC 18 Driver for SQL Server**: The database driver is required in order to be able to connect to the SQL database on Azure.

The python web app consists of the following elements:

- **Flask Web Framework**: Flask is a web framework for creating web applications in python. We use it to simplify the development of the web app in python. The only part the developer has to take care of, is the configuration and definition of HTTP routes. The other parts such as routing, parsing of request data, and listening for connections is handled by Flask.
- **SQLAlchemy Python Library**: The SQLAlchemy python library provides an SQL toolkit for e.g., connecting to databases and configuring connection properties. For instance, we enable a connection pooling, which

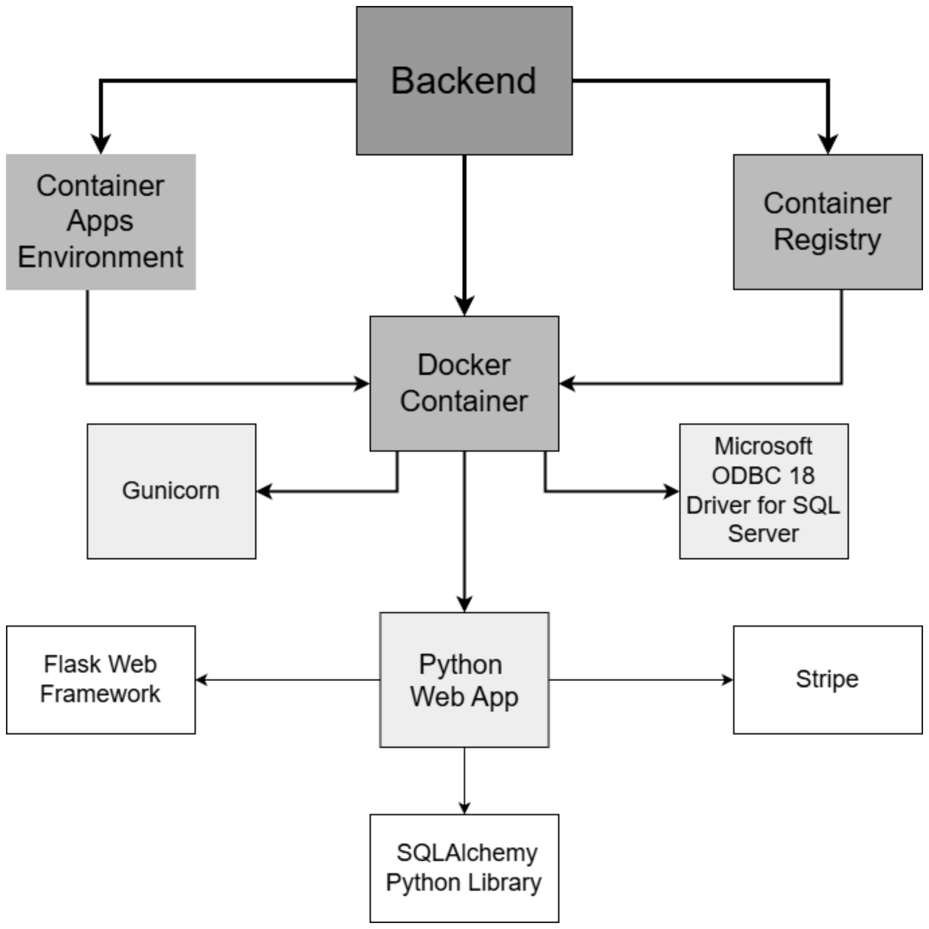

Backend
Container Apps Environment
Container Registry
Docker Container
Gunicorn
Microsoft ODBC 18 Driver for SQL Server
Flask Web Framework
Python Web App
Stripe
SQLAlchemy Python Library

Figure 5: Backend Building Block View

makes the backend maintain multiple connections, which are periodically recycled. This prevents issues with stale connections.

- **Stripe**: We integrate the Stripe payment processing platform, so that customers can use said platform to pay for products, rather than us having to deal with e.g., accepting credit payments or other payment methods.

E-Mail Service

The Email Service is a standalone microservice designed to handle email notifications for customers in the system. Its primary task is to send confirmation emails containing the receipt in PDF format. Implemented in Python, the Email Service uses the Azure Communication Service for reliable email delivery.

The workflow starts when a customer places an order through the front-end application. The front-end submits the order details, including items, quantities, prices, and customer email, to the backend via a secure API call. The backend initiates a payment request using Stripe's Checkout API. Upon successful payment, Stripe notifies the backend via a webhook event (payment_intent.succeeded), confirming the transaction.

After receiving and validating the Stripe webhook, the backend triggers the email service by sending a request containing the order ID and customer email. The Email Service retrieves the order details from the backend, generates a PDF receipt encapsulating the order information, and constructs an email with the PDF receipt attached. This email is then sent to the customer using the Azure Communication Service.

Database

We use an Azure SQL database for storing our data. The database schema can be described with the entity-relationship diagram seen in fig. 6.

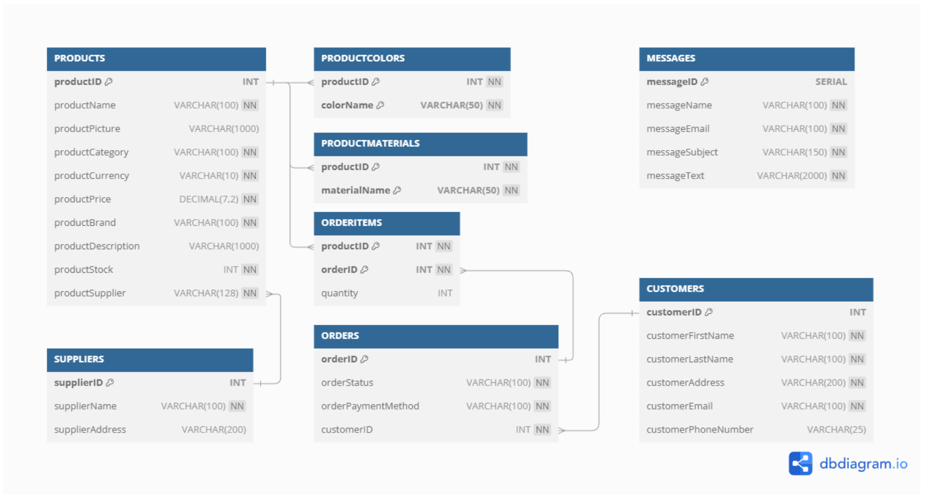

PRODUCTS
productID INT
productName VARCHAR(100) NN
productPicture VARCHAR(1000)
productCategory VARCHAR(100) NN
productCurrency VARCHAR(10) NN
productPrice DECIMAL(7,2) NN
productBrand VARCHAR(100) NN
productDescription VARCHAR(1000)
productStock INT NN
productSupplier VARCHAR(128) NN
PRODUCTCOLORS
productID INT NN
colorName VARCHAR(50) NN
PRODUCTMATERIALS
productID INT NN
materialName VARCHAR(50) NN
ORDERITEMS
productID INT NN
orderID INT NN
quantity INT
MESSAGES
messageID SERIAL
messageName VARCHAR(100) NN
messageEmail VARCHAR(100) NN
messageSubject VARCHAR(150) NN
messageText VARCHAR(2000) NN
SUPPLIERS
supplierID INT
supplierName VARCHAR(100) NN
supplierAddress VARCHAR(200)
ORDERS
orderID INT
orderStatus VARCHAR(100) NN
orderPaymentMethod VARCHAR(100) NN
customerID INT NN
CUSTOMERS
customerID INT
customerFirstName VARCHAR(100) NN
customerLastName VARCHAR(100) NN
customerAddress VARCHAR(200) NN
customerEmail VARCHAR(100) NN
customerPhoneNumber VARCHAR(25)
dbdiagram.io

Figure 6: Database Entity-Relationship Diagram

Interaction with the database go through the SQLAlchemy ORM of the backend.

Image Storage

We use an Azure Storage Account as a BLOB storage for our product images. Anonymous read requests are allowed for the product images, allowing the frontend to fetch a product image from the BLOB storage if it has the correct file name, which is included in the backend response for any given product.

Important Interfaces

The frontend can communicate with the backend using a RESTful API. The following routes are used by the frontend:

/product API Endpoint

Route	Methods
• /products • /products/<ID>	• GET • POST • PUT • DELETE

This route concerns itself with products. The admin panel has access to POST, PUT, and DELETE operations, while the webshop only has access to GET. GET returns a JSON object containing information about the product with the given ID. The object contains the following fields:

- *id*: Identifier for the product
- *name*: Name of the product. Example: "Modern Sofa".
- *category*: Category of the product. Example: "Sofas".
- *price*: Unit price of the product. Example: "999.99".
- *currency*: Currency symbol. Example: "€".
- *description*: Detailed description of the product. Example: "A sleek and luxurious sofa that . . . ".
- *brand*: Brand producing the product. Example: "Furniture LLC".
- *materials*: A list of materials the product is made out of. Example: "[Fabric]".
- *colors*: A list of colors of the product. Example: "[gray]".
- *pictureUrl*: The file name under which our BLOB storage stores the product image. Example: "modern-sofa.webp".

There is no product with an ID of 0. Instead, inputting ID 0 retrieves all the products in the shop catalog as a list of JSON objects.

/messages API Endpoint

Route	Methods
• /messages • /messages/<ID>	• GET • POST • DELETE

This endpoint is responsible for handling messages that customers can leave if they have questions or any other issues. Customers can submit their messages through a POST request on the contact page of the website. Once submitted, the admin can view all the messages via a GET request. Both the GET and POST requests follow a similar JSON structure, which includes the following fields:

- *id*: Identifier for the message (only accessible for GET request. POST requests add a new message to the database, hence the id is created at that point).
- *name*: Name of the person who wants to get in contact with the vivendo staff.
- *email*: E-Mail address of that person, so that the admin can answer the request of the person.
- *subject*: A title for the request.
- *message*: The request the person has. This field can have up to 1000 characters.

Unlike the product endpoint, the message endpoint does not require an ID when retrieving all messages via a GET request. This is because the admin can simply view all messages without the need for any specific identifier. However, an ID is required when the admin wants to delete a particular message using a DELETE request. The ID ensures that the correct message is deleted from the database, allowing for accurate and targeted operations.

/create-checkout-session API Endpoint

The `/create-checkout-session` API endpoint is the backend's entry point for initiating the payment process, facilitating the transition from a customer's cart submission to Stripe's hosted checkout experience. It handles the creation of a Stripe Checkout session and ensures order details are persisted for downstream processing.

Route	Methods
• `/create-checkout-session`	• `POST`

The endpoint accepts POST requests from the frontend, which submits a JSON payload containing the cart details—such as item names, quantities, prices, and the customer's email. Upon receiving the request, the backend performs the following steps:

1. **Validation:** Ensures required fields (e.g., items and email) are present and valid. If not, it returns a `400 Bad Request` response with an error message.
2. **Stripe Session Creation:** Constructs a Stripe Checkout session using the Stripe API. Cart items are transformed into line items, with prices converted to cents and the currency set to EUR. Success and cancel URLs are dynamically generated from the request's origin (e.g., `http://frontend-domain/success`), and metadata including the `order_id` is attached. The customer's email is also embedded for receipt purposes.
3. **Response:** Returns a `200 OK` response with the Stripe session ID, which the frontend uses to redirect the customer to Stripe's checkout page. If an error occurs (e.g., Stripe API failure), it rolls back the database transaction and returns a `500 Internal Server Error`.

Once the session ID is received, the frontend leverages the `Stripe.js` library to redirect the customer to Stripe's hosted checkout page, where payment is completed. The endpoint's role concludes here, with payment confirmation handled asynchronously via the `/webhook` endpoint.

/webhook API Endpoint

The `/webhook` API endpoint enables the backend to receive asynchronous event notifications from Stripe, facilitating real-time processing of payment outcomes. It serves as the critical link between Stripe's payment system and the backend's notification workflows.

Route	Methods
• `/webhook`	• `POST`

The endpoint processes POST requests from Stripe, which delivers event data in JSON format whenever a payment-related action occurs, such as the `payment_intent.succeeded` event indicating a successful transaction. The backend handles these requests as follows:

1. **Validation:** Verifies the request's authenticity by checking the `Stripe-Signature` header against the `STRIPE_WEBHOOK_SECRET` environment variable. If invalid, it returns a `400 Bad Request` response.
2. **Event Processing:** Extracts essential data from the payload, including the payment intent ID, amount received (in cents), customer email, and metadata (e.g., `order_id`).
3. **Notification Trigger:** Initiates the Email Service by dispatching a request with the `order_id` and customer email, then returns a `200 OK` response to Stripe.

This interface ensures secure and reliable communication with Stripe, leveraging the webhook's event-driven nature to synchronize payment status with backend operations.

6 Runtime View

Display Product List

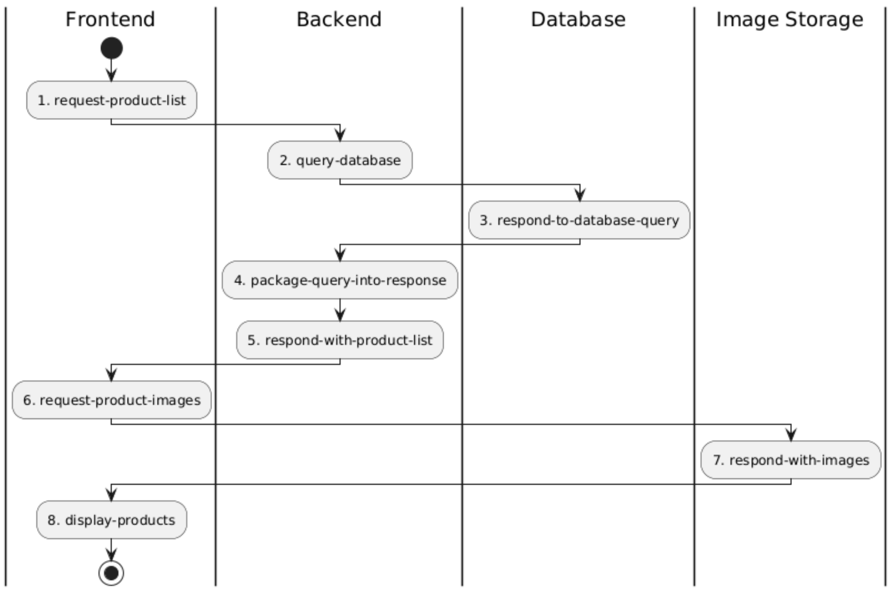


Figure 7: UML Swimlane Diagram for Displaying Product List

Explanation:

1. Frontend request product list from backend.
2. The backend queries the database to retrieve all the products.
3. The database responds to the query with the products.
4. The backend packages the result of the query into a JSON response.
5. The backend sends the response to the frontend.
6. Since the backend response contains URLs to images rather than the images themselves, the frontend needs to fetch it from the BLOB Storage.
7. Read operations can be carried out anonymously on the images, therefore the BLOB storage responds with the image data.
8. Now the product list can be displayed in the frontend.

Cart Checkout Process

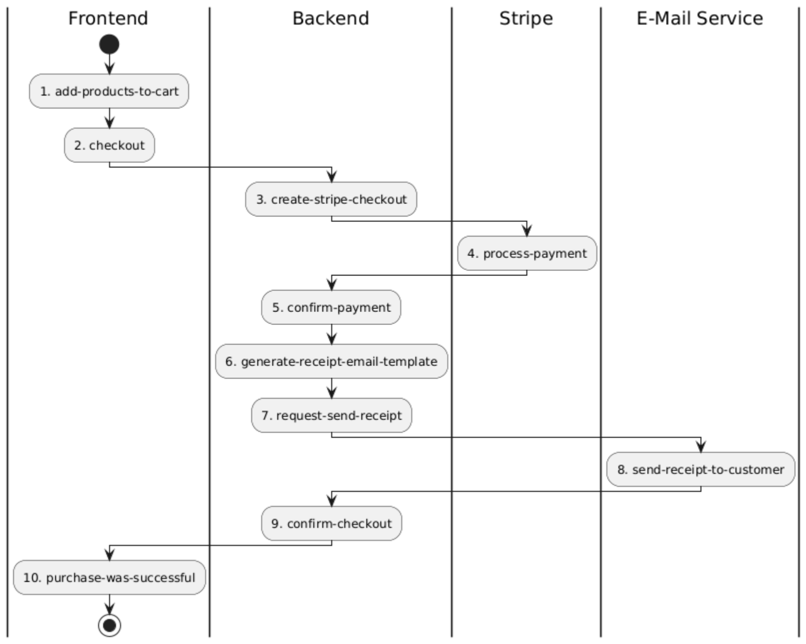


Figure 8: UML Swimlane Diagram for the Checkout Process

Explanation:

1. The user uses the frontend user interface to add items to the cart.
2. The user wants to start the checkout process.
3. The backend creates a checkout on the Stripe platform.
4. The Stripe platform handles the payment process such as accepting credit card information.
5. The backend confirms whether the payment with Stripe was successful.
6. The backend generates a template E-Mail containing an order confirmation with a receipt as a PDF attachment.
7. The backend forwards this E-Mail to the E-Mail Service, so that it can be sent to the user.
8. The E-Mail Service sends the order confirmation and receipt to the user's E-Mail address.
9. The backend confirms that the checkout was successful.
10. The transaction is shown as successful in the frontend user interface.

Creating/Updating/Deleting Products in Admin Panel

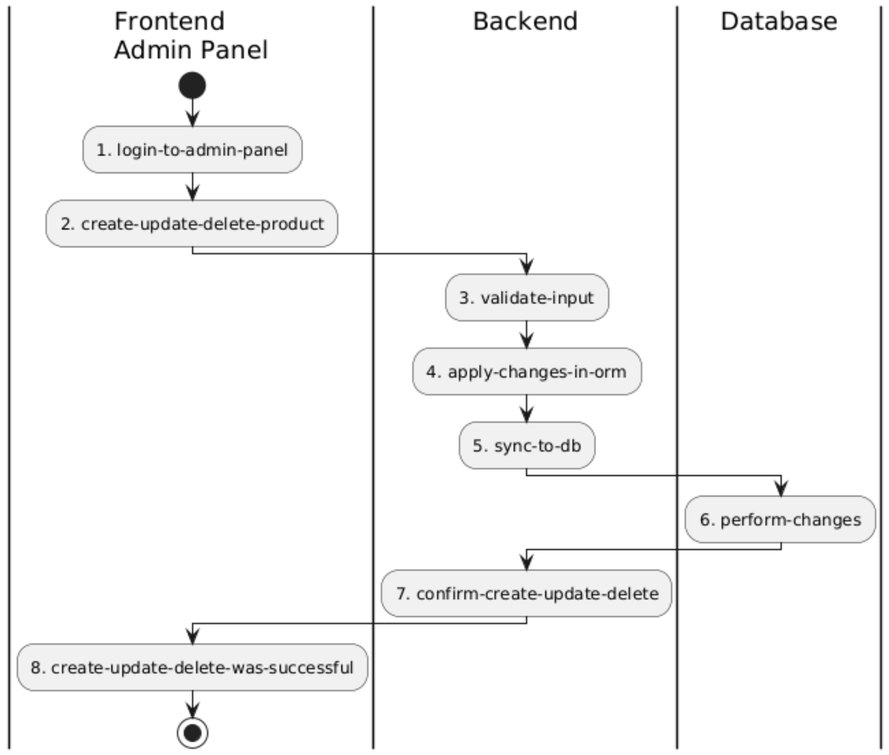


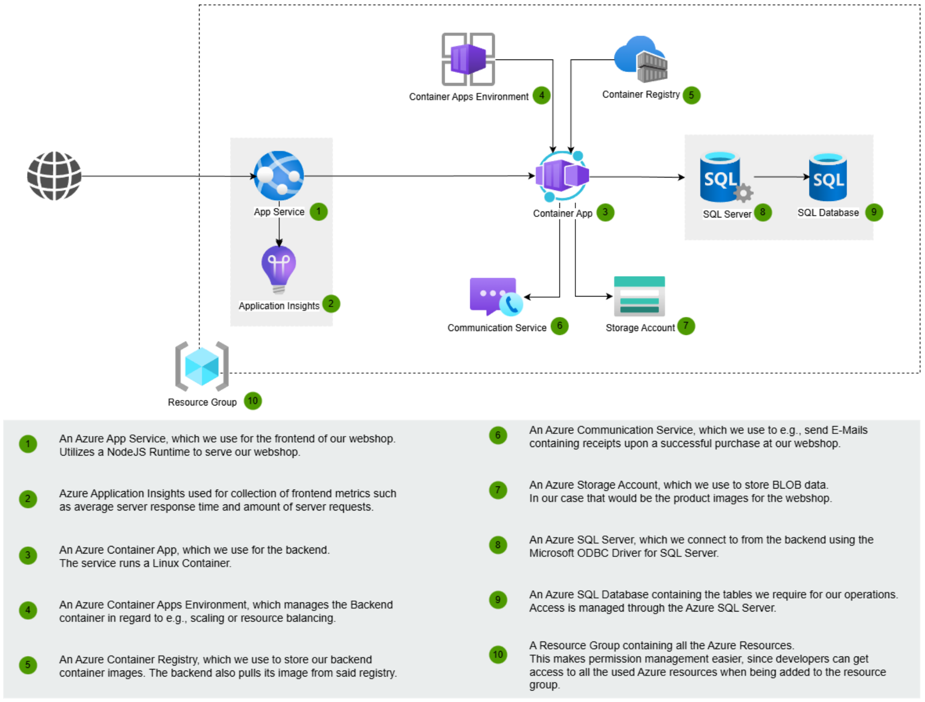
Figure 9: UML Swimlane Diagram for Creating, Updating, and Deleting Products via the Admin Panel

Explanation:

1. An administrator logs into the admin panel with admin credentials.
2. The administrator creates, updates and/or deletes product(s).
3. The backend performs input validation to e.g., prevent negative prices or invalid input formats.
4. The operation is carried out using the object relational mapping of the backend.
5. The object relational mapping syncs up with the database.
6. The database performs the required write operations.
7. The backend confirms, that the operation was successful.
8. The administrator now receives feedback, that the operation was successful.

7 Deployment View

Microsoft Azure Deployment



1 An Azure App Service, which we use for the frontend of our webshop. Utilizes a NodeJS Runtime to serve our webshop.

2 Azure Application Insights used for collection of frontend metrics such as average server response time and amount of server requests.

3 An Azure Container App, which we use for the backend. The service runs a Linux Container.

4 An Azure Container Apps Environment, which manages the Backend container in regard to e.g., scaling or resource balancing.

5 An Azure Container Registry, which we use to store our backend container images. The backend also pulls its image from said registry.

6 An Azure Communication Service, which we use to e.g., send E-Mails containing receipts upon a successful purchase at our webshop.

7 An Azure Storage Account, which we use to store BLOB data. In our case that would be the product images for the webshop.

8 An Azure SQL Server, which we connect to from the backend using the Microsoft ODBC Driver for SQL Server.

9 An Azure SQL Database containing the tables we require for our operations. Access is managed through the Azure SQL Server.

10 A Resource Group containing all the Azure Resources. This makes permission management easier, since developers can get access to all the used Azure resources when being added to the resource group.

Figure 10: Azure Infrastructure Diagram

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

Now follows a mapping of components in the building block view to components in the deployment view:

Quality and/or Performance Features

Component in Building Block View	Component in Deployment View
Frontend	• App Service • Application Insights
Backend	• Container App • Container Apps Environment • Container Registry
E-Mail Service	• Communication Service

Component in Building Block View	Component in Deployment View
Database	• SQL Server • SQL Database
Image Storage	• Storage Account

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

Frontend

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.

Backend

Domain Model & Data Structures

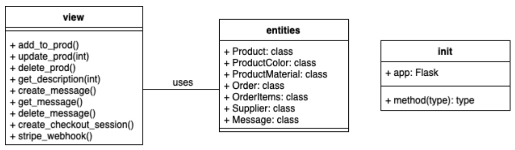
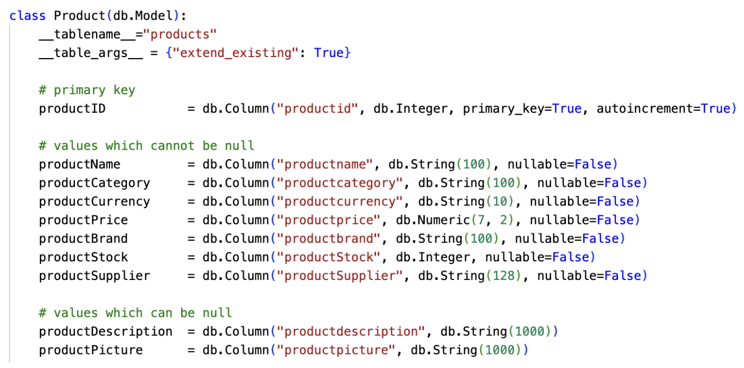


Figure 11: Backend Domain Model

Figure 11 illustrates the main structure of the backend. As discussed in section , the backend is designed to be lightweight, with no specific design pattern applied. There is an association between the view and entity layers, as the view script interacts with the entity classes when saving data to the database.

The init script, as the name suggests, initializes the app variable, which is required throughout other scripts in the backend.

The entities script defines classes that represent the database tables. These data structures are then utilized within the views script to perform CRUD operations.



```python
class Product(db.Model):
    __tablename__="products"
    __table_args__ = {"extend_existing": True}

    # primary key
    productID           = db.Column("productid", db.Integer, primary_key=True, autoincrement=True)

    # values which cannot be null
    productName         = db.Column("productname", db.String(100), nullable=False)
    productCategory     = db.Column("productcategory", db.String(100), nullable=False)
    productCurrency     = db.Column("productcurrency", db.String(10), nullable=False)
    productPrice        = db.Column("productprice", db.Numeric(7, 2), nullable=False)
    productBrand        = db.Column("productbrand", db.String(100), nullable=False)
    productStock        = db.Column("productStock", db.Integer, nullable=False)
    productSupplier     = db.Column("productSupplier", db.String(128), nullable=False)

    # values which can be null
    productDescription  = db.Column("productdescription", db.String(1000))
    productPicture      = db.Column("productpicture", db.String(1000))
```

Figure 12: Example of an database entity

Figure 12 displays the entity object representing the Product table. This data structure allows us to store and manage the various products offered by Vivendo. Similar entity objects are used for all tables in the database to ensure consistent handling of data.

User Experience Concept

While the backend does not directly influence the look and feel of the user interface, it plays an important role in enhancing the overall user experience by providing clear and informative push notifications. These messages notify users when an action is successful or unsuccessful, helping to keep them informed throughout their interaction with the system.

Additionally, the frontend interacts with the backend through RESTful API endpoints, allowing users to retrieve data in JSON format. This seamless integration between frontend and backend ensures that data is delivered efficiently and consistently to improve the user experience.

Implementation rules

JSON Validation Concept

In the views script, each endpoint begins by checking whether the incoming request object is in JSON format. This serves two main purposes: first, it ensures that the data follows the agreed-upon format; second, it simplifies data handling for the backend developer. Once the data format is verified, the appropriate actions are taken based on the type of operation.

Single Responsibility Concept

Another key principle followed in the backend was ensuring that each function performs only one operation at a time. Although Flask allows handling multiple CRUD operations within a single function by specifying them in the route decorator and filtering requests using if-statements, this approach was avoided.

Instead, each CRUD operation was assigned its own function to maintain a clear and organized backend code structure.

Response Messaging Concept
Thirdly, if any exception occurs during the processing of the request data, either an HTTP 400 error (for bad requests) or an HTTP 500 error (for internal server issues) is returned to the frontend, along with a detailed message. This message is used to inform the user that an issue has occurred and to provide more context, ensuring they are aware that something went wrong. In cases where the request is successfully processed, an HTTP 200 status code is returned, accompanied by a message or some data from the database confirming that the operation was completed successfully, tailored to the specific action performed.

9 Quality Requirements

Contents

This section details the quality requirements for the Webshop system, focusing on performance, security, usability, maintainability, and reliability. It includes a quality tree to prioritize these attributes and quality scenarios to make them concrete and measurable, covering both critical goals (e.g., security) and lower-priority ones (e.g., reliability).

Motivation

Quality requirements are essential for the Webshop to meet user expectations, ensure reliable operation, and support future growth. They guide architectural decisions about cloud infrastructure, payment integration, and user interface design, ensuring the system is secure, performant, and user-friendly for users.

Quality Tree

Contents The quality tree organizes the Webshop's quality attributes, starting with 'Quality' as the root and branching into categories like Performance, Security, Usability, Maintainability, and Reliability, with sub-attributes and priorities linked to quality scenarios. It reflects both normal operating conditions and peak demand scenarios to ensure comprehensive coverage.

Motivation The quality tree provides a structured overview and prioritization of the Webshop's quality requirements, helping Developers focus on critical aspects like security and performance while addressing secondary goals like usability and system reliability. It enables architects to evaluate and design the system effectively, linking to detailed scenarios for validation.

Form The quality tree is presented as a hierarchical diagram, with 'Quality' at the top-left, branching horizontally into categories and sub-attributes, each with priorities (High, Medium, Low) and references to corresponding quality scenarios in the Quality Scenarios section.

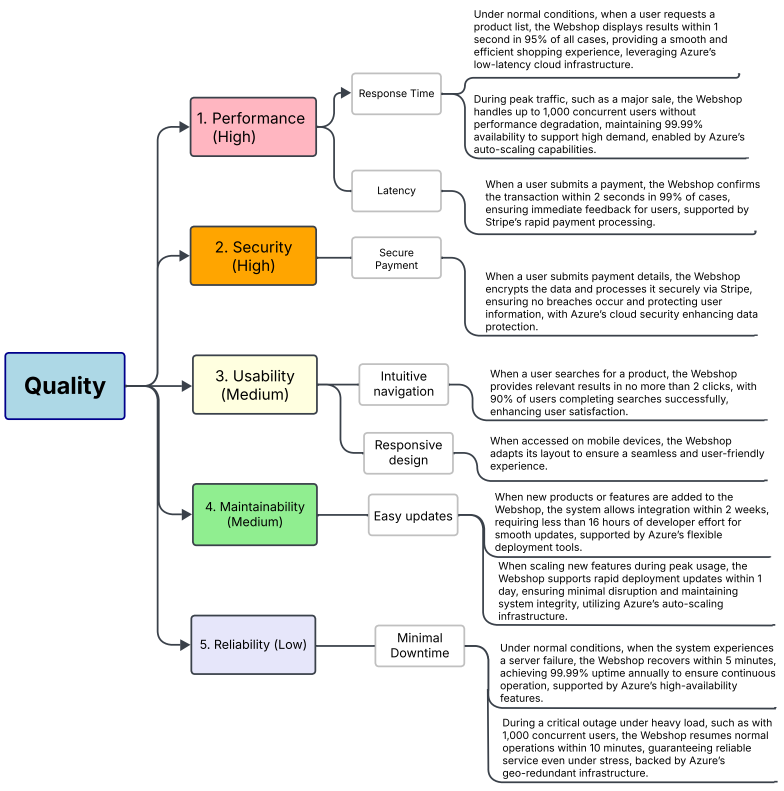
Quality Scenarios

Contents

This subsection provides concrete quality scenarios for the Webshop, detailing runtime responses to stimuli (usage scenarios) and system modifications (change scenarios), with measures and priorities linked to the quality tree. It includes scenarios for both normal conditions and peak demand to ensure robust system performance.

Motivation

Quality scenarios make the Webshop's quality requirements tangible, allowing architects to design and test for performance, security, usability, and reliability.



Under normal conditions, when a user requests a product list, the Webshop displays results within 1 second in 95% of all cases, providing a smooth and efficient shopping experience, leveraging Azure’s low-latency cloud infrastructure.

During peak traffic, such as a major sale, the Webshop handles up to 1,000 concurrent users without performance degradation, maintaining 99.99% availability to support high demand, enabled by Azure’s auto-scaling capabilities.

When a user submits a payment, the Webshop confirms the transaction within 2 seconds in 99% of cases, ensuring immediate feedback for users, supported by Stripe’s rapid payment processing.

When a user submits payment details, the Webshop encrypts the data and processes it securely via Stripe, ensuring no breaches occur and protecting user information, with Azure’s cloud security enhancing data protection.

When a user searches for a product, the Webshop provides relevant results in no more than 2 clicks, with 90% of users completing searches successfully, enhancing user satisfaction.

When accessed on mobile devices, the Webshop adapts its layout to ensure a seamless and user-friendly experience.

When new products or features are added to the Webshop, the system allows integration within 2 weeks, requiring less than 16 hours of developer effort for smooth updates, supported by Azure’s flexible deployment tools.

When scaling new features during peak usage, the Webshop supports rapid deployment updates within 1 day, ensuring minimal disruption and maintaining system integrity, utilizing Azure’s auto-scaling infrastructure.

Under normal conditions, when the system experiences a server failure, the Webshop recovers within 5 minutes, achieving 99.99% uptime annually to ensure continuous operation, supported by Azure’s high-availability features.

During a critical outage under heavy load, such as with 1,000 concurrent users, the Webshop resumes normal operations within 10 minutes, guaranteeing reliable service even under stress, backed by Azure’s geo-redundant infrastructure.

Figure 13: Quality Tree for the Webshop System

They ensure the system meets user expectations, supports cloud-based operations and payment integration, and can be evaluated for trade-offs, providing clarity for Developers in development and evaluation.

Form

Quality scenarios are presented in a tabular format, with columns for scenario type, stimulus, response, measure, and priority, corresponding to leaves in the quality tree.

Scenario Type	Stimulus	Response	Measure	Priority
Usage (Performance)	User requests a product list	System displays products within 1 second	95% of requests complete in <1 second	High
Change (Performance)	Traffic spikes during a sale	System handles 1,000 concurrent users	99% availability during peak load	High
Usage (Performance)	User submits payment	System confirms transaction within 2 seconds	99% of transactions complete in <2 seconds	High
Usage (Security)	User submits payment details	System encrypts data and processes via Stripe	Data encrypted with TLS 1.2, no breaches	High
Usage (Usability)	User searches for a product	System shows relevant results in 2 clicks	90% of users complete searches successfully	Medium
Usage (Usability)	User accesses Webshop on mobile	System adapts layout for mobile devices	95% of mobile users report satisfaction	Medium
Change (Maintainability)	New products or features are added	System integrates within 2 weeks	Integration completed with <16 hours effort	Medium
Change (Maintainability)	Scaling new features during peak usage	System integrates updates within 1 day	Integration completed with <24 hours effort	Medium
Usage (Reliability)	System experiences server failure	System recovers within 5 minutes	99.99% uptime annually	Low
Usage (Reliability)	System outage under heavy load	System resumes normal operations within 10 minutes	99.99% uptime during peak load	Low

Table 12: Quality Scenarios for the Webshop System

9 Risks and Technical Debts

Contents

This section lists the identified technical risks and technical debts for the Webshop system, ordered by priority, to ensure proactive management and system reliability. The risks and debts focus on scalability, security, reliability, and maintainability, reflecting the system's cloud-based architecture and payment integration.

Motivation

"Risk management is project management for grown-ups" (Tim Lister, Atlantic Systems Guild.) This motto underscores the importance of systematically detecting and evaluating risks and technical debts in the Webshop's architecture. These insights are crucial for management stakeholders, such as project managers and product owners, to plan mitigation strategies and ensure the system's success in a cloud-based, e-commerce context.

Form

The technical risks and debts are presented in a bulleted list, ordered by priority, with descriptions, priorities, and suggested measures to minimize, mitigate, or avoid risks or reduce technical debts.

- **Scalability Limits Under Peak Load (High Priority)**: The Webshop may struggle to handle sudden spikes in traffic (e.g., 1,000 concurrent users during a sale) if Azure's auto-scaling or Stripe's transaction processing capacity is underestimated, potentially leading to performance degradation or downtime. *Measures*: Conduct load testing with Azure's monitoring tools (e.g., Azure Monitor) to validate scalability, optimize Azure auto-scaling policies, and verify Stripe's plan supports high transaction volumes.
- **Security Vulnerabilities in Payment Integration (Medium Priority)**: Improper configuration of Stripe's API or Azure's security settings could expose payment data or user information to breaches, violating privacy regulations or user trust. *Measures*: Implement regular security audits, use Azure's security features (e.g., Key Vault), and follow Stripe's security best practices, including PCI DSS compliance.
- **Downtime Due to Azure Service Outages (Medium Priority)**: Reliance on Azure's infrastructure poses a risk of downtime if Azure experiences regional outages or if geo-redundancy isn't configured properly, affecting the 99.9% uptime goal. *Measures*: Configure Azure's geo-redundant backups and disaster recovery options, monitor Azure SLAs (e.g., 99.95%–99.99% uptime), and implement failover strategies.
- **Limited Frontend Maintainability Due to Rapid Prototyping (Low Priority)**: The Next.js frontend, developed quickly with React Context and Tailwind CSS, may have tightly coupled components or inconsistent code patterns, making future updates time-consuming. *Measures*: Refactor critical components to improve modularity, document code patterns, and adopt a style guide for Tailwind CSS and React.
- **Basic Authentication in Admin Module (Low Priority)**: The current authentication for the admin dashboard (e.g., username/password in local storage) lacks robust security (e.g., OAuth, Azure AD), creating a debt needing production upgrades. *Measures*: Plan to integrate Azure Active Directory or OAuth for secure authentication, replacing the basic implementation in future iterations.

10 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.*
Webshop	*A web-based e-commerce platform designed to display products, process orders, handle payments, and send confirmations, enabling users to browse and purchase items online, as implemented in this project.*
Stripe	*A payment processing service integrated into the Webshop to securely handle online transactions, ensuring fast, reliable, and PCI DSS-compliant payment processing for users.*
Auto-scaling	*A cloud computing feature, used in the Webshop's infrastructure, that automatically adjusts resources (e.g., server instances) based on demand, improving performance and availability during peak traffic.*
Uptime	*The percentage of time the Webshop is operational and available to users, critical for ensuring reliable service, typically measured against service-level agreements (SLAs) in cloud environments like Azure.*

12 Information about Work

Distribution of Report Work

Member	Sections
Vinay Duhan	• 1- Introduction and Goals • 2- Constraints
Jatender Singh Jossan	• 4- Solution Strategy • 5- Building Block View (Frontend, Backend side) • 8- Crosscutting Concepts
Dennis Mark	• 4- Solution Strategy • 5- Building Block View (Frontend side) • 8- Crosscutting Concepts
Dominique Conceicao Rosario	• 3- Context and Scope • 5- Building Block View (Backend side) • 6- Runtime View • 7- Deployment View • 10- Quality Requirements • 11- Risks and Technical Debt
Edward Späth	• 5- Building Block View (Backend and Database) • 6- Runtime View • 7- Deployment View

Repository

The public GitHub Repository is online and can be reached at https://github.com/EdwardSpaeth/Cloud-Project-Group-11. The GitHub usernames can be mapped to each project member in the following way:

Member	GitHub Username
Vinay Duhan	duhanvinay
Jatender Singh Jossan	jatenderjossan
Dennis Mark	solipskierr

Member	**GitHub Username**
Dominique Conceicao Rosario	DomiRosario
Edward Späth	EdwardSpaeth