Food-Bridge: Synergistische Vernetzung zur Förderung nachhaltiger Lebensmittelallokation

Verfasser: Dominic Sieber, Philip Tobler

Referent: Daniel Tobler

Co-Referent: Stefan Kapferer

Architekturdokumentation Food-Bridge

Quellennachweis Titelblatt

Foodwaste [Foto]. (ohne Datum). Gefunden am 01. September 2025 unter https://www.zebrabox.ch/sites/default/files/2023-07/Zebrabox_gegen_foodwaste_teaser.jpeg

Management Summary

Ausgangslage

Mit Food-Bridge wird eine Webplattform entwickelt, die gemeinnützige Organisationen dabei unterstützt, Lebensmittelverschwendung zu reduzieren. Durch die digitale Vernetzung von Produzenten, Hilfsorganisationen und Lieferanten können überschüssige, jedoch vollwertige Lebensmittel effizient und nachhaltig ausgetauscht werden. Food-Bridge soll einen echten Mehrwert für unsere Gesellschaft und unseren Auftraggeber Schweizer Tafel schaffen. Dazu soll Food-Bridge eine moderne, skalierbare und praxisnahe Lösung werden.

Ziele

Das Hauptziel ist eine Plattform mit allen Kernfunktionen produktiv zu bringen: Registrierung und Rollenverwaltung, Inserieren von überschüssigen Lebensmitteln, Echtzeit-Bestellung durch Hilfsorganisationen sowie Planung und Verwaltung der Abholung durch Lieferanten. Ergänzend werden CI/CD-Pipelines in Azure DevOps, ein durchdachtes Datenbankmodell, Sicherheitsmechanismen und Monitoring umgesetzt. Neben der technischen Umsetzung verfolgt die Arbeit das Ziel, Erfahrungen in modernen Technologien wie Angular, Spring Boot und PostgreSQL zu vertiefen und diese praxisnah anzuwenden.

Vorgehen

Das Projekt wird agil nach Scrum durchgeführt. In kurzen Iterationen werden Anforderungen umgesetzt, getestet und durch Stakeholder evaluiert. Technisch basiert die Lösung auf einem Angular-Frontend, einem Java/Spring-Boot-Backend und einer relationalen SQL-Datenbank. Eine CI/CD-Pipeline in Azure DevOps stellt Build, Tests und Deployment sicher. Sicherheitsaspekte wie JSON Web Token und rollenbasierte Zugriffskontrolle werden integriert. Die Entwicklung folgt einem testgetriebenen Ansatz mit Unit- und Integrationstests, um Qualität und Stabilität zu gewährleisten.

Erkenntnisse

Die Arbeit zeigt, dass durch eine gezielte digitale Plattform ein wesentlicher Beitrag zur Reduktion von Food Waste geleistet werden kann. Gleichzeitig verdeutlicht sie die Komplexität der Vernetzung von Produzenten, Hilfsorganisationen und Lieferanten in einem einzigen System. Einfach gehaltene Kernprozesse zum Inserieren und Bestellen von Lebensmitteln sowie eine lose gekoppelte, hexagonale Architektur erweisen sich als Schlüsselfaktoren für die Praxistauglichkeit und Wartbarkeit. Zudem bestätigt das Projekt, dass moderne Frameworks wie Angular und Spring Boot in Kombination mit Cloud-Infrastruktur (Amazon Web Services) eine robuste Grundlage für nachhaltige und skalierbare Anwendungen bieten.

Empfehlungen

Das Projektteam empfiehlt die Plattform produktiv einzusetzen. Dabei sollen weitere hilfreiche Funktionalitäten schrittweise integriert werden, um den Prozess besser abzurunden und die Nutzerfreundlichkeit zu steigern. Ebenso ist eine enge Zusammenarbeit mit der Schweizer Tafel und einem Produzenten als Beta-Tester zentral, um die Praxistauglichkeit der Software zu gewährleisten. Auf technologischer Ebene sollten die Skalierbarkeit und Security-Aspekte im Multiuser-Betrieb beobachtet werden. Mit diesen Massnahmen ist ein offizieller Release im Jahr 2026 äusserst realistisch.

Inhaltsverzeichnis

1 Einleitung und Ziele

Im Rahmen dieser Masterarbeit wird eine Webplattform zur Förderung von nachhaltiger Lebensmittelverteilung zwischen Hilfsorganisationen und Produzenten für die Schweizer Tafel entwickelt. Die Plattform mit dem Namen «Food-Bridge» leistet damit einen Beitrag zur Reduktion von Lebensmittelverschwendung und zur Unterstützung gemeinnütziger Organisationen. Mithilfe einer Anforderungsanalyse und Kundenfeedback werden die verschiedenen Funktionalitäten mit den Frameworks Angular und Java/Spring-Boot umgesetzt. Ergänzt wurde die Entwicklung durch den Einsatz einer relationalen SQL-Datenbank, einer CI/CD-Pipeline in Azure DevOps sowie dem Hosting mittels Amazon Web Services, um einen stabilen Betrieb sicherzustellen.

1.1 Qualitätsziele

Die Kernanforderungen werden in der nachfolgenden Tabelle aufgelistet und erläutert. In unserem Wiki haben wir die Anforderungen weiter ausgeführt.

Anforderung	**Beschreibung**
Benutzerregistrierung und -verwaltung	Benutzer können ein eigenes Konto auf Food-Bridge erstellen. Zudem bietet die Software eine Rollenverwaltung, um bspw. Admin-Rechte zu vergeben.
Inserieren von überschüssigen Lebensmitteln	Benutzer einer Organisation können überschüssige Lebensmittel inserieren. Dem Warenangebot können mehrere Artikel hinzugefügt werden.
Bestellung durch Hilfsorganisationen	Die inserierten Lebensmittel können von Hilfsorganisationen bestellt werden.
Planung und Verwaltung der Abholung durch Lieferanten	Der Transport der Bestellung kann über Transportaufträge abgefertigt werden. Zusätzlich besteht die Möglichkeit, auf einem Transportauftrag Hinweise für den Lieferanten zu hinterlegen.

Die Webapp muss zusätzlich zu den funktionalen Anforderungen ebenfalls nicht funktionale Anforderungen erfüllen.

Die Software muss:

- über 99% der Zeit erreichbar sein.
- skalieren, um einen Betrieb von bis zu zehn parallelen Usern zu ermöglichen.
- die Bestimmungen des Datenschutzgesetztes der Schweiz erfüllen.
- die Daten müssen in der Schweiz gehostet werden.
- eine wartbare und erweiterbare Architektur bieten.

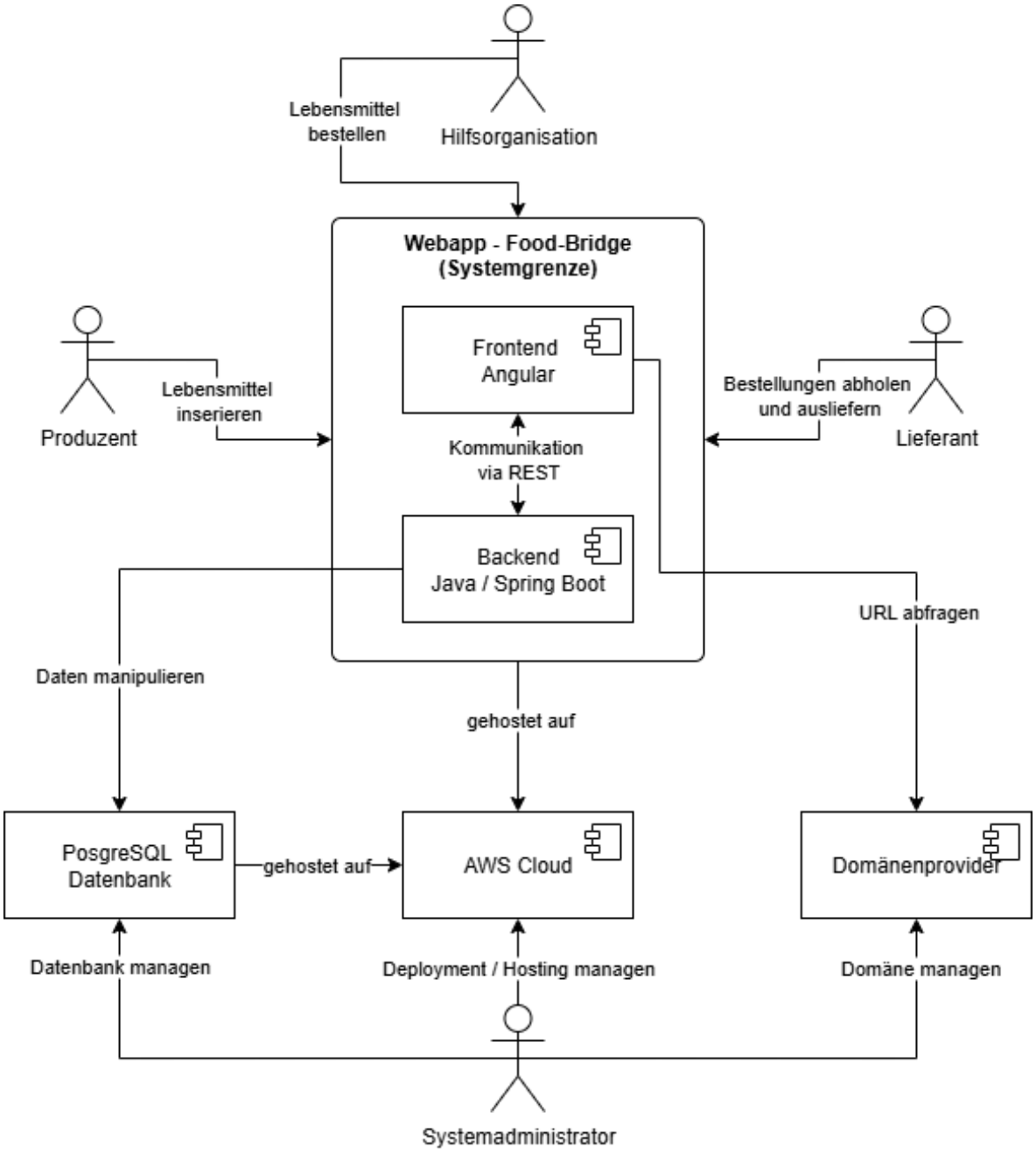
1.2 Stakeholder

Nachfolgend sind unsere vier Hauptinteressensgruppen aufgelistet mit einer kurzen Beschreibung deren Erwartungen.

Stakeholder	Erwartung
Schweizer Tafel	Einfache Administration innerhalb der Software. Zudem soll die Lösung kostengünstig in der Wartung sein.
Produzent	Intuitive Lösung, um überschüssige Lebensmittel zu inserieren, damit die Lagerplätze für andere Produkte genutzt werden können.
Hilfsorganisation	Intuitive Lösung, um Lebensmittel zu bestellen.
Lieferant	Einfache Lösung, um Informationen zum Transport einzusehen.

2 Systemkontext-Diagramm

In der folgenden Grafik ist unser Systemkontext-Diagramm ersichtlich.



3 Lösungsstrategie

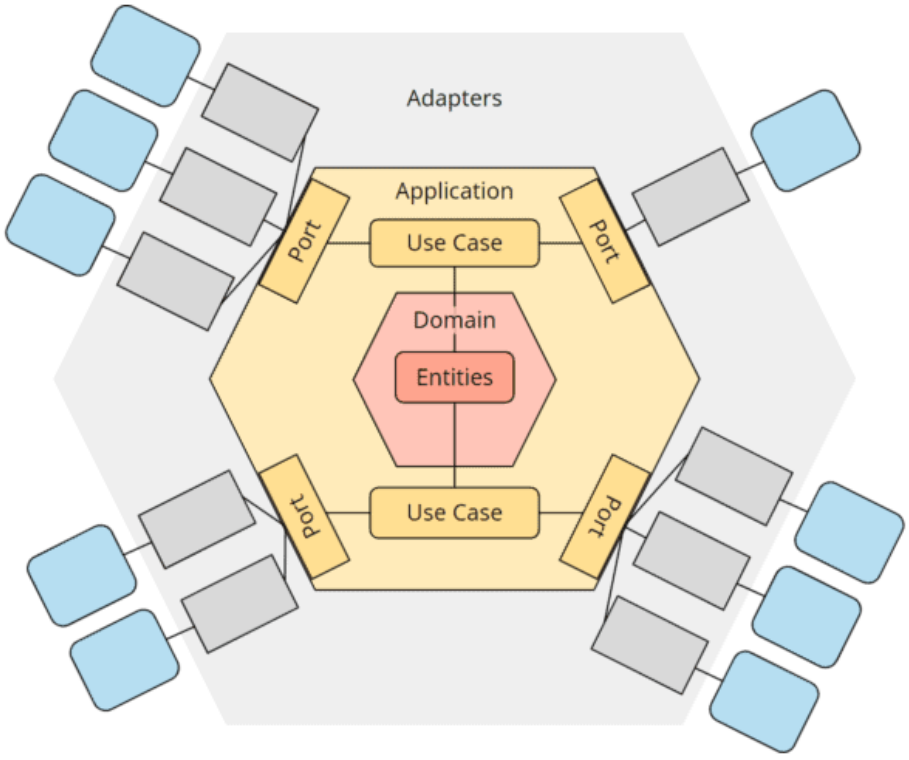
Nachstehend wird unsere Lösungsstrategie genauer erläutert.

3.1 Leitziele

- Nachhaltige Lieferketten für Überbestände effizient vermitteln
- Einfach bedienbar für Produzenten, Hilfsorganisationen, Lieferanten
- Robust und erweiterbar (modular, testbar, Cloud-ready)

3.2 Architekturgrundsätze

- Trennung von Frontend und Backend
- Hexagonale Architektur im Backend: Domain unabhängig von Frameworks sowie Integration von Ports und Adapter, um die Kommunikation mit dem Frontend und der Datenbank zu separieren.
    - Damit die Architektur zwingend eingehalten wird, wurde mit Maven-Submodulen gearbeitet. Somit ist der Zugriff auf ein unerlaubtes Modul klar geregelt.
    - Maven-Submodule: bootstrap (App-Starter und Security-Konfigurationen), adapters (REST-Controller und Datenbank-Adapter), application (Business-Logik), domain (Business-Objekte)



3.3 Technologieentscheidungen

- Frontend: Angular und TypeScript mit ESLint
- Backend: Java mit Spring Boot und Data JPA
- Datenbank: PostgreSQL
- Mail: Spring Mail (SMTP Provider).
- Container: Docker
- Infrastruktur: Amazon Web Services
    - EC2: Virtuelle Ubuntu Umgebung
    - ECR: Repository für Docker-Images
    - S3: Kleiner Speicher für Docker-Compose- und Variablen-File (env)
    - RDS: Datenbankservice für PostgreSQL
- CI/CD: Azure DevOps Pipelines für Tests und Deployment
- AuthGuard, RoleGuard und TokenGuard: Trennung der Sicherheitslogik direkt im Router
- HTTP-Interceptor: Zentrale Stelle für JWT, Error-Handling und Response-Manipulation
- Notification Service: Einheitliche Benutzerkommunikation für Fehler, Warnungen und Informationen
- NgRx Store mit App State: Zentralisierte State-Verwaltung für Stammdaten
- Language Service (ngx-translate): Einfache Erweiterung um neue Sprachen

3.4 Qualitätsziele und Taktiken

- Sicherheit: JWT, Rollen, Input-Validierung, Argon2 für Passwortverschlüsselung
- Performance: Caching der Stammdaten im Frontend und Datenbank-Indizes
- Wartbarkeit: saubere Ports und Adapter, modulare Services, Code-Style-Checks, Unit- sowie Integrations-Tests
- Skalierbarkeit: horizontale Skalierung aufgrund von stateless Backend möglich

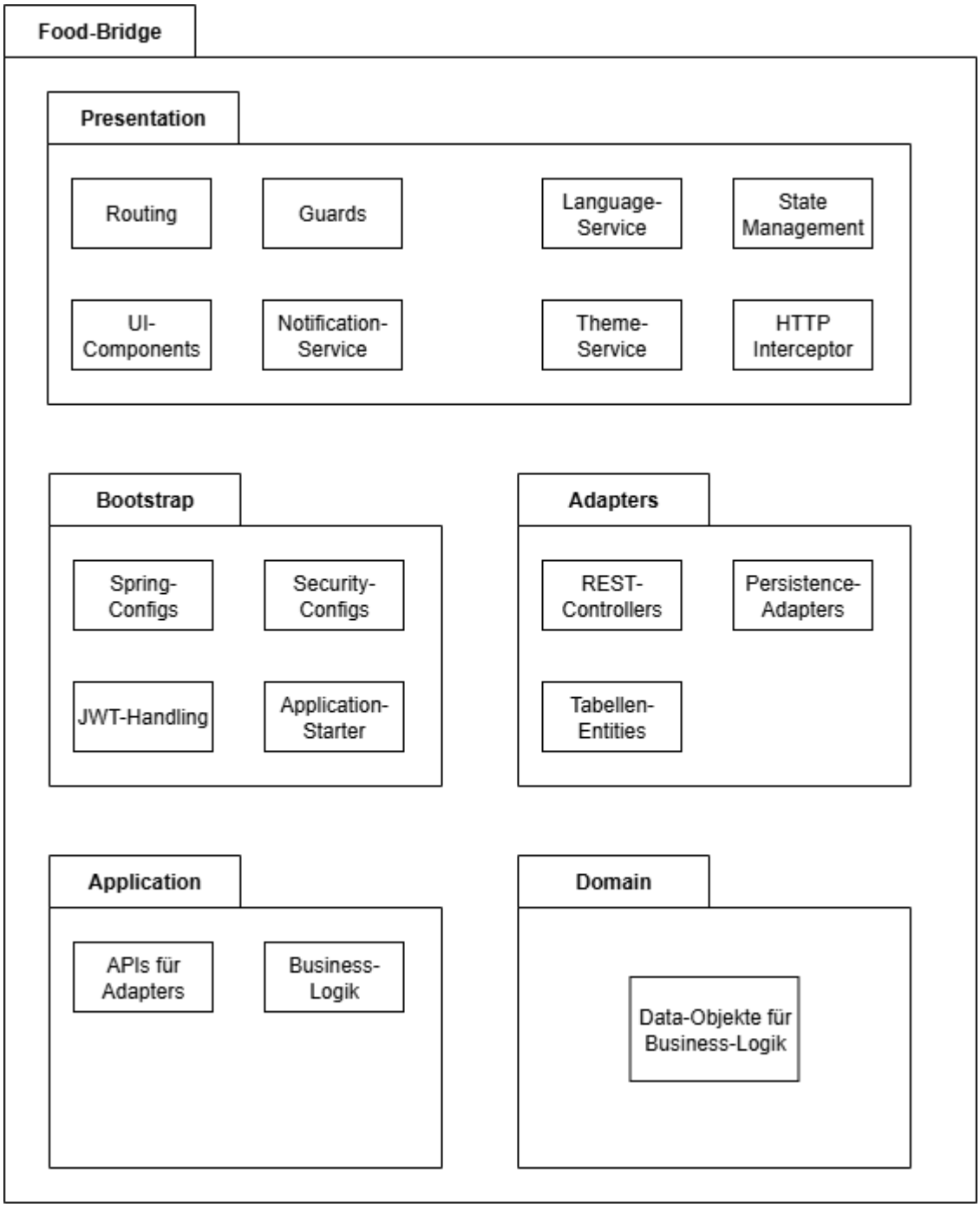
3.5 Deployment-Strategie

Auf das Deployment wird im Kapitel 6 vertieft eingegangen. Nachfolgend ein kurzer Überblick.

- Prod-Pipeline: Für Frontend und Backend Docker-Image erstellen → Die neusten Images mittels Docker-Compose auf der EC2-Umgebung deployen
- Rollback: Aufgrund Repository mit versionierten Docker-Images möglich

4 Bausteinschicht

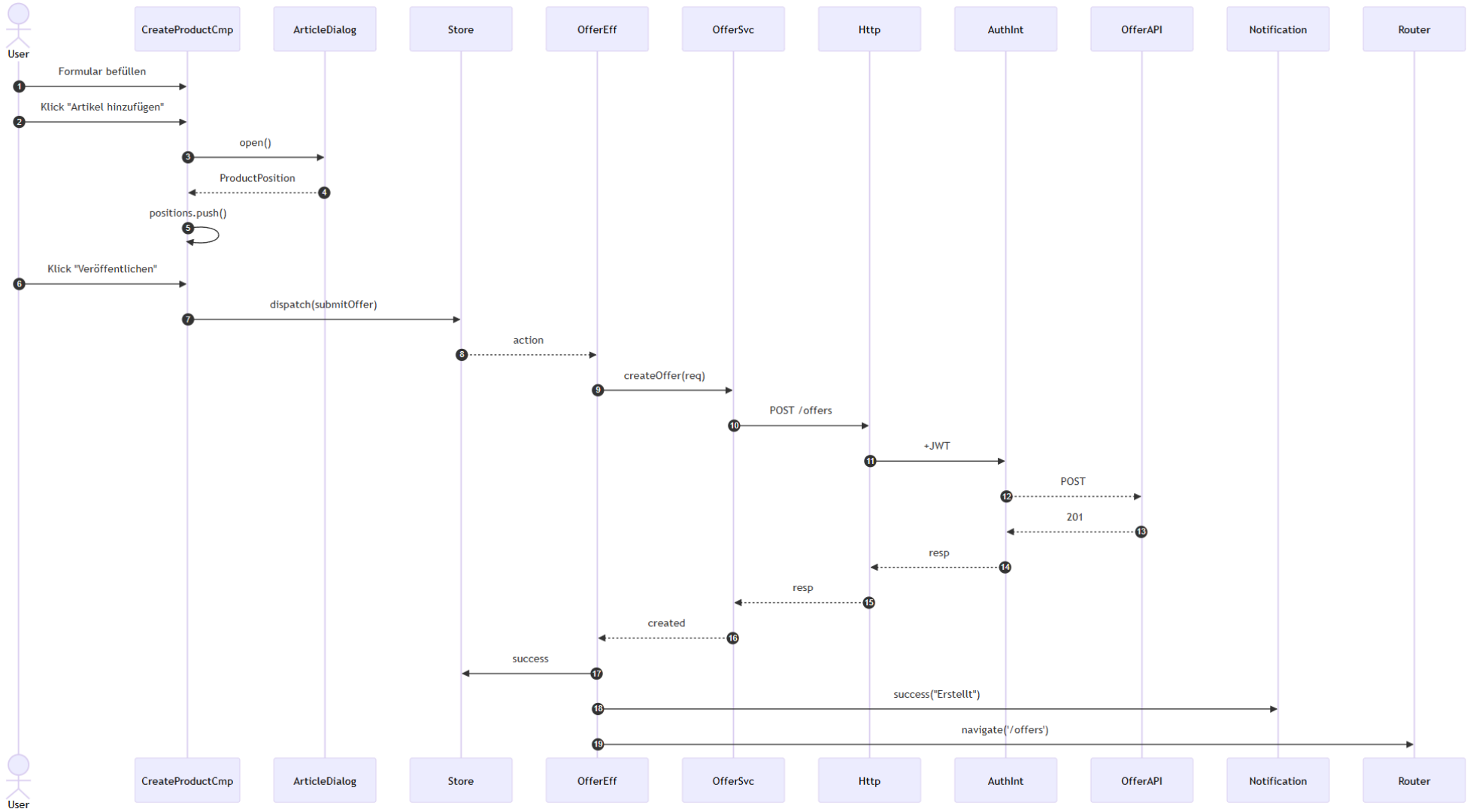
Die nachfolgende Grafik widerspiegelt grob die wichtigsten Komponenten des Frontend- und Backend-Projekts. Zum Frontend-Projekt gehört das Presentation-Package und zum Backend-Projekt gehören die Packages Bootstrap, Adapters, Application und Domain.

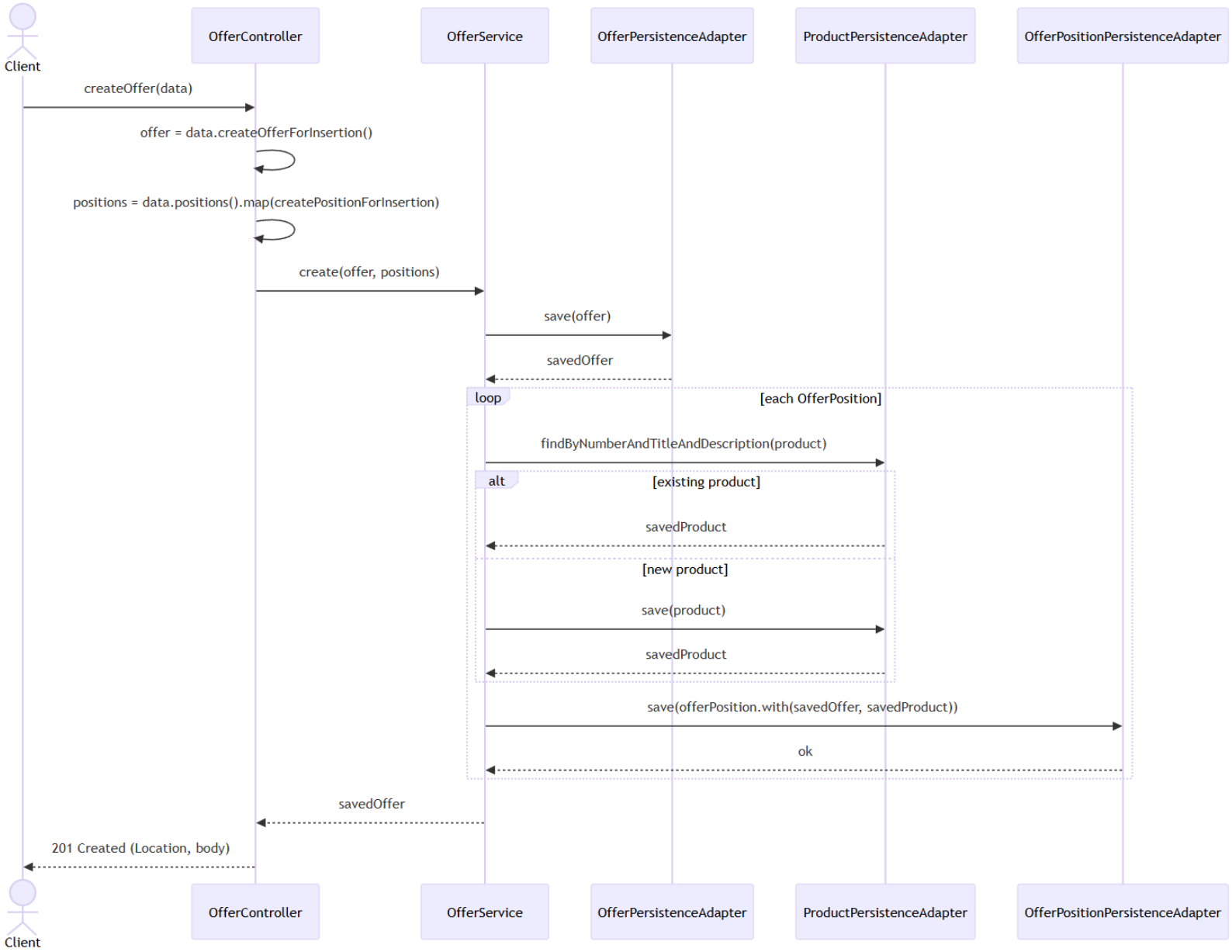


5 Laufzeitschicht

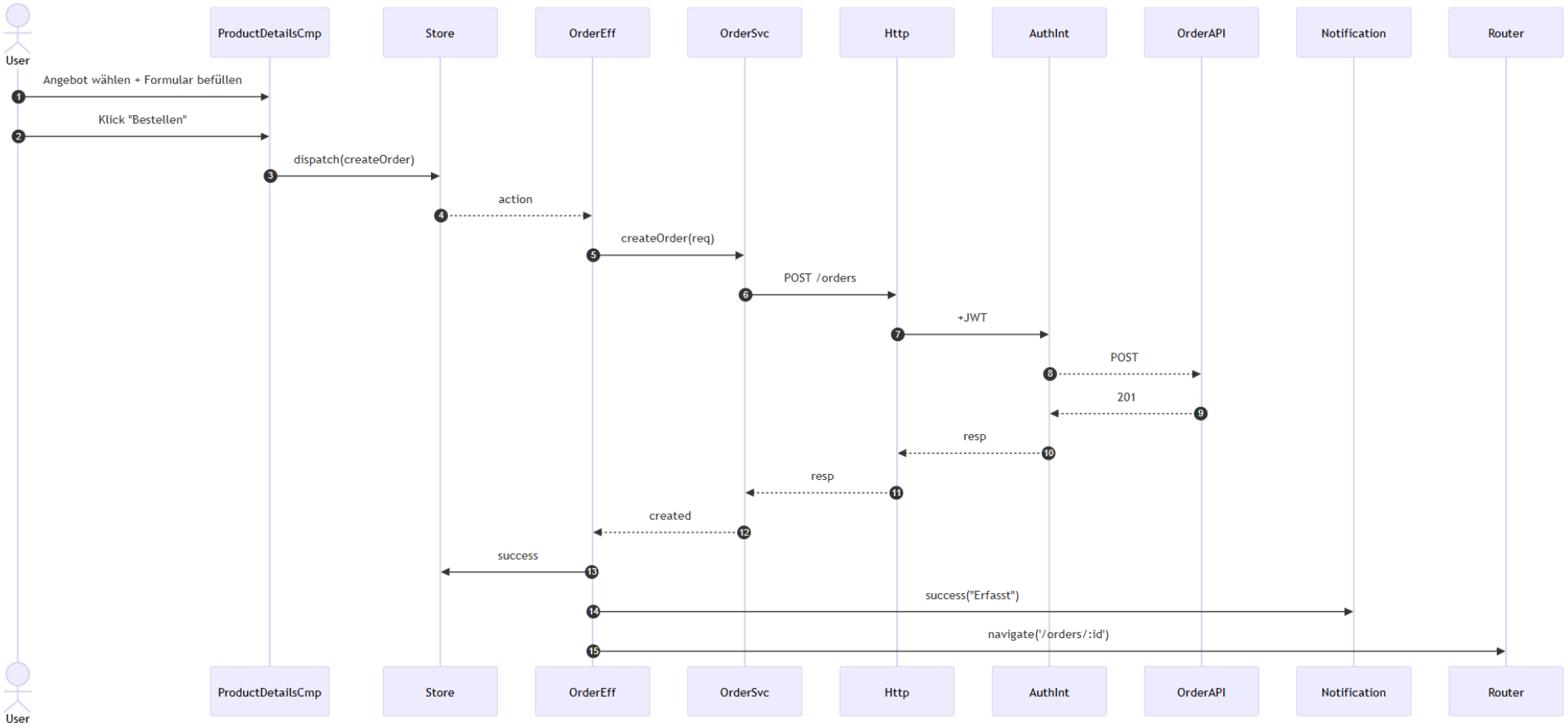
In diesem Kapitel wird das Zusammenspiel der Komponenten zur Laufzeit aufgezeigt. Nachfolgend sind die wichtigsten Use Cases als Sequenzdiagramm dargestellt. Für jeden Use Case gibt es einmal das Frontend- und einmal das Backend-Diagramm.

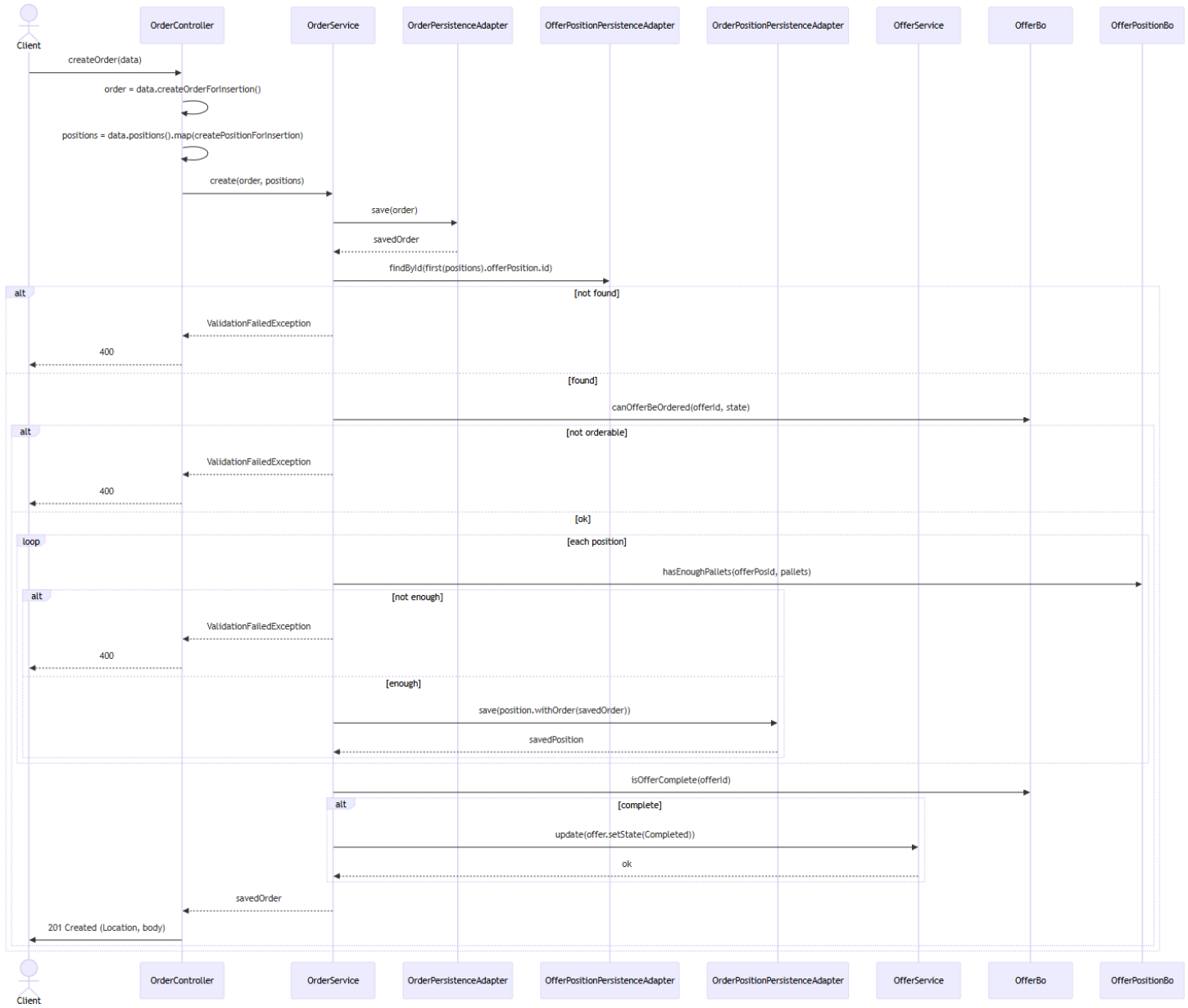
5.1 Warenangebot erstellen



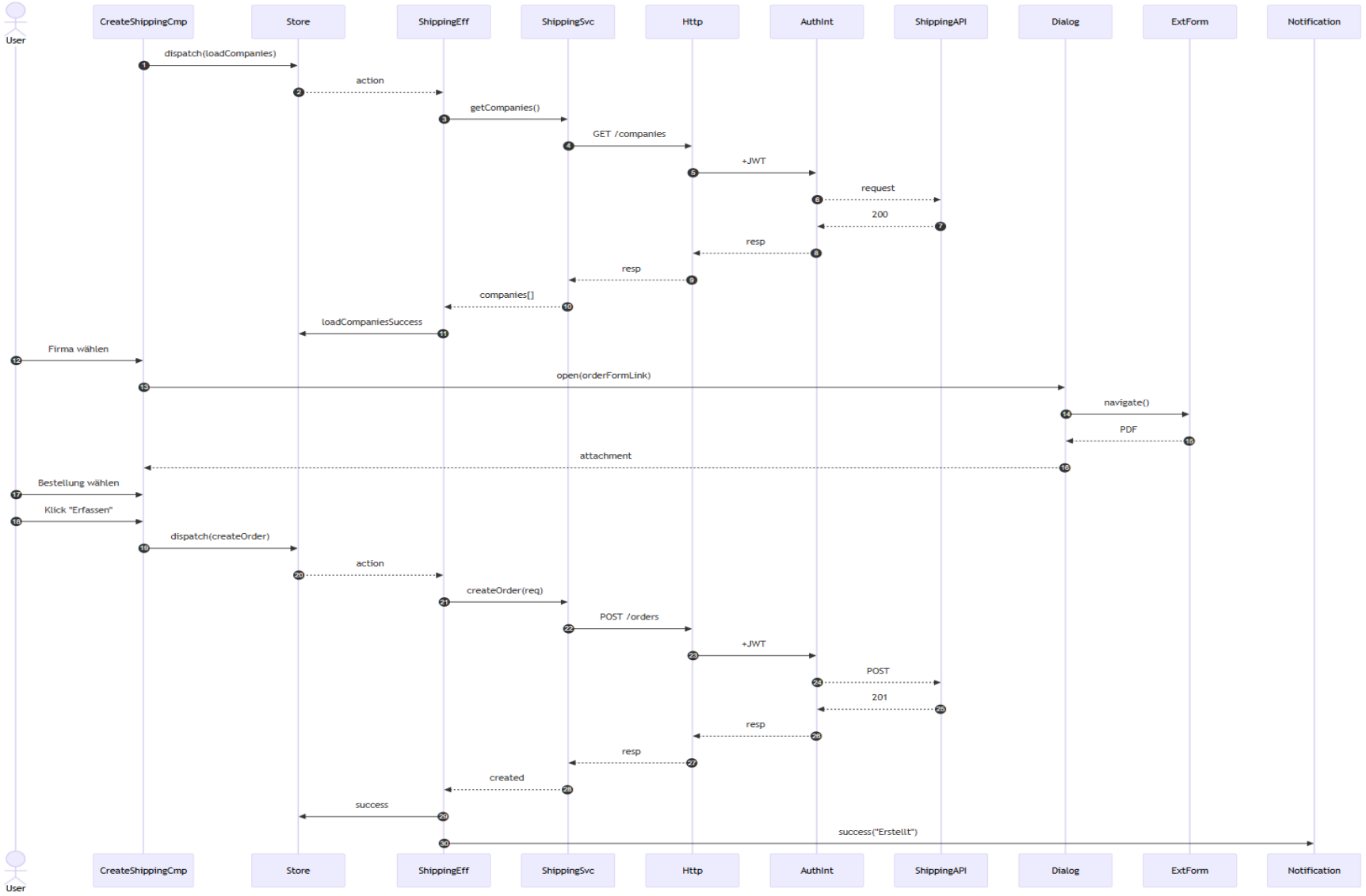

Client
OfferController
OfferService
OfferPersistenceAdapter
ProductPersistenceAdapter
OfferPositionPersistenceAdapter
createOffer(data)
offer = data.createOfferForInsertion()
positions = data.positions().map(createPositionForInsertion)
create(offer, positions)
save(offer)
savedOffer
loop
[each OfferPosition]
findByNumberAndTitleAndDescription(product)
alt
[existing product]
savedProduct
[new product]
save(product)
savedProduct
save(offerPosition.with(savedOffer, savedProduct))
ok
savedOffer
201 Created (Location, body)
Client
OfferController
OfferService
OfferPersistenceAdapter
ProductPersistenceAdapter
OfferPositionPersistenceAdapter

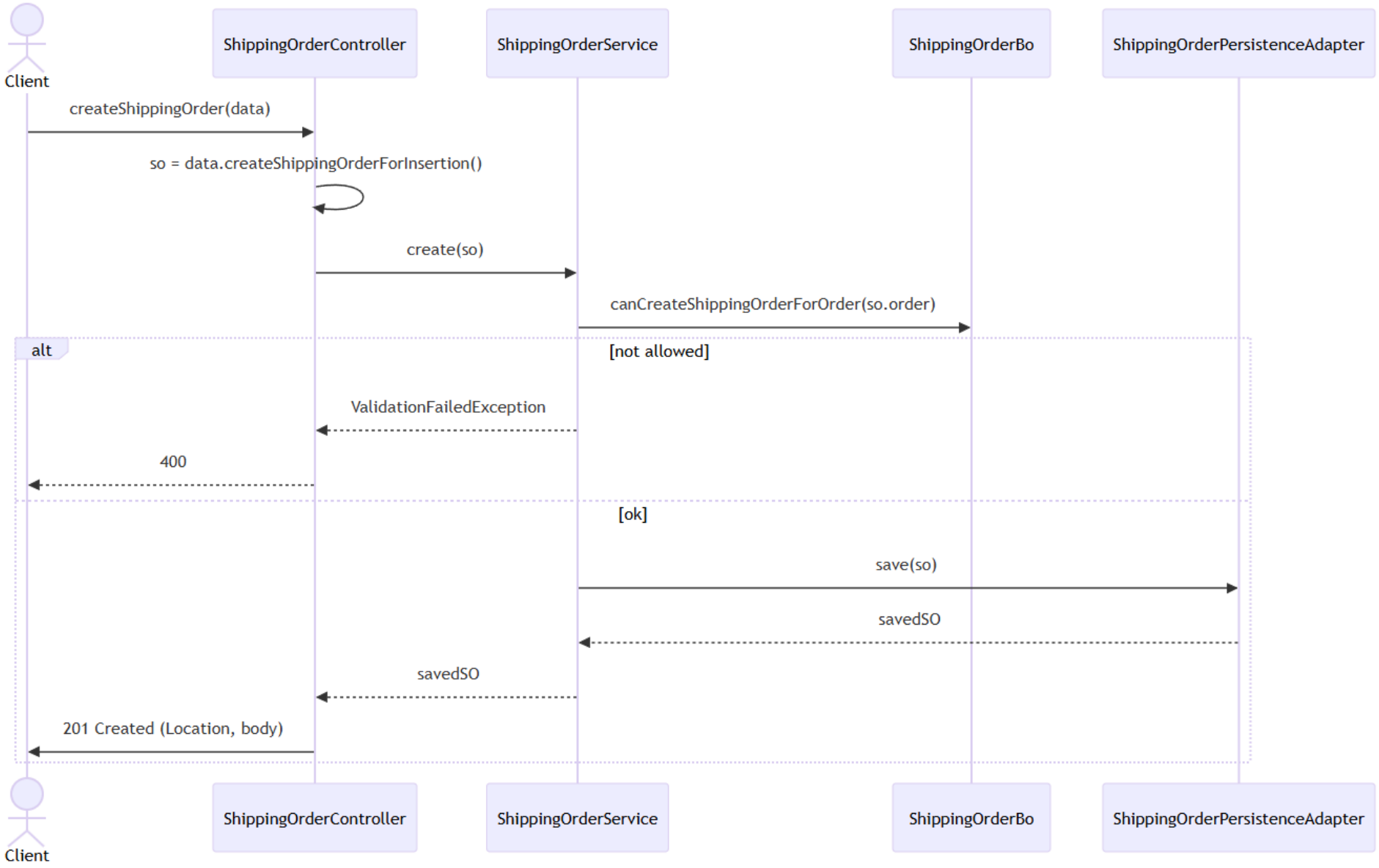
5.2 Warenangebot bestellen




Client
OrderController
OrderService
OrderPersistenceAdapter
OfferPositionPersistenceAdapter
OrderPositionPersistenceAdapter
OfferService
OfferBo
OfferPositionBo
createOrder(data)
order = data.createOrderForInsertion()
positions = data.positions().map(createPositionForInsertion)
create(order, positions)
save(order)
savedOrder
findById(first(positions).offerPosition.id)
alt
[not found]
ValidationFailedException
400
[found]
canOfferBeOrdered(offerId, state)
alt
[not orderable]
ValidationFailedException
400
[ok]
loop
[each position]
hasEnoughPallets(offerPosId, pallets)
alt
[not enough]
ValidationFailedException
400
[enough]
save(position.withOrder(savedOrder))
savedPosition
isOfferComplete(offerId)
alt
[complete]
update(offer.setState(Completed))
ok
savedOrder
201 Created (Location, body)
Client
OrderController
OrderService
OrderPersistenceAdapter
OfferPositionPersistenceAdapter
OrderPositionPersistenceAdapter
OfferService
OfferBo
OfferPositionBo

5.3 Transportauftrag erstellen




Client
ShippingOrderController
ShippingOrderService
ShippingOrderBo
ShippingOrderPersistenceAdapter
createShippingOrder(data)
so = data.createShippingOrderForInsertion()
create(so)
canCreateShippingOrderForOrder(so.order)
alt
[not allowed]
ValidationFailedException
400
[ok]
save(so)
savedSO
savedSO
201 Created (Location, body)
Client
ShippingOrderController
ShippingOrderService
ShippingOrderBo
ShippingOrderPersistenceAdapter

6 Verteilungssicht

In diesem Kapitel gehen wir näher auf den Aufbau des Deployments ein.

6.1 Ausführungsumgebung

- Cloud: Amazon Web Services (AWS)
- Compute: EC2 (Ubuntu, Docker + Compose)
- Edge/Proxy: Caddy auf EC2 (TLS, Routing → Frontend, /api → Backend)
- Docker-Container:
    - Frontend: Angular + Caddy (statische Auslieferung)
    - Backend: Java / Spring Boot
- Datenbank: RDS PostgreSQL
- Storage/Assets: S3
- Registry: ECR (Container-Images)
- Konfig/Secrets: .env- und Docker-Compose-Dateien auf S3

6.2 Deployment via Pipelines

Auf Azure DevOps haben wir eine Frontend- und eine Backend-Pipeline. Diese sorgen dafür, dass der Code von den Main-Branches in einem Docker-Image bereitgestellt werden, um anschliessend auf der EC2-Instanz deployt zu werden. Die PostgreSQL-Datenbank wurde über einen Relational Database Service (RDS) an die EC2-Instanz angebunden.

Stages der Frontend-Pipeline: Docker-Image anhand von Code erstellen (Im Docker ist ebenfalls der Reverse-Proxy Caddy integriert) → erstelltes Image auf ECR hochladen → Deployment des aktuellen Images auf der EC2-Instanz

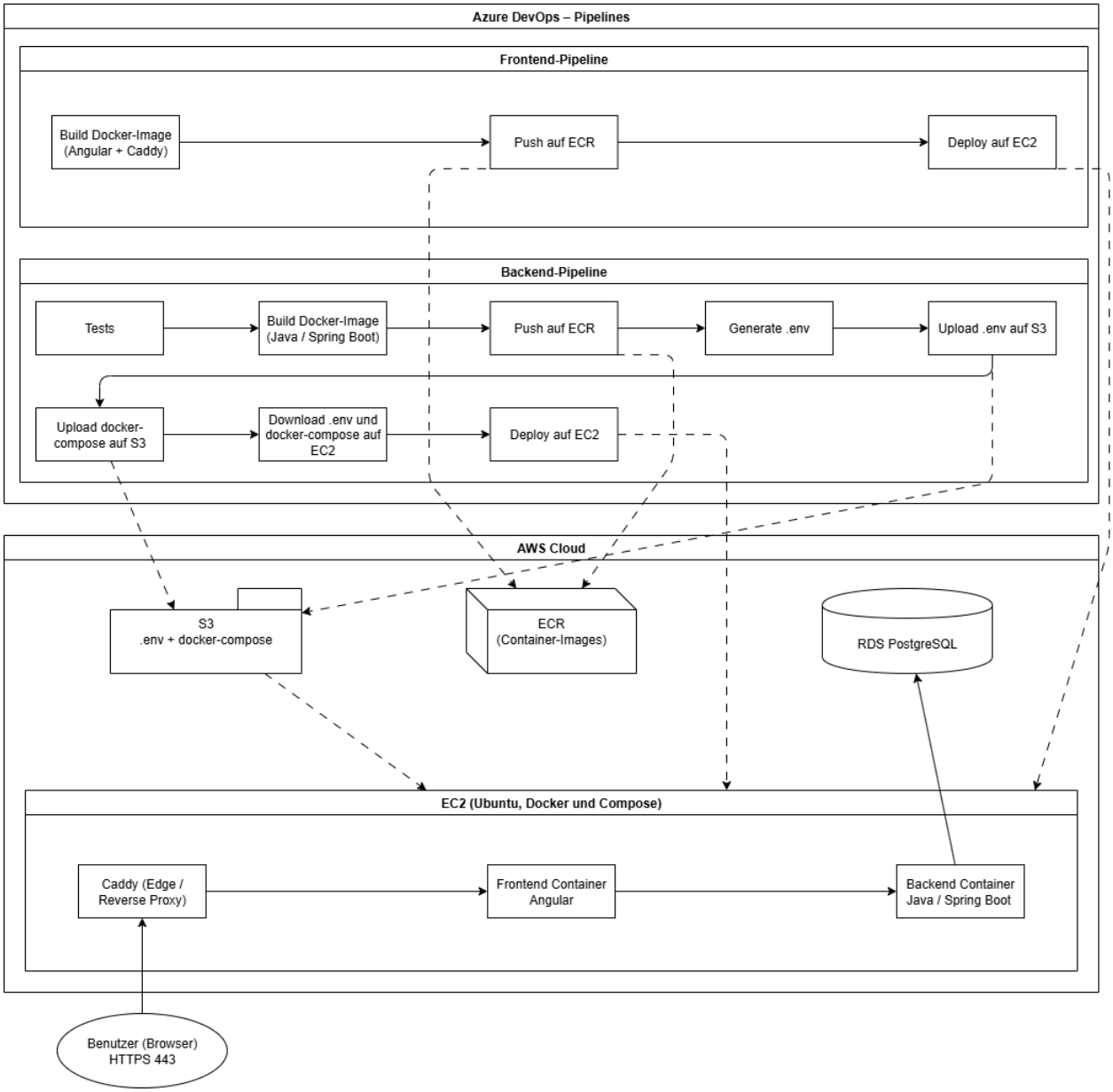
Stages der Backend-Pipeline: Tests ausführen → Docker-Image anhand von Code erstellen → erstelltes Image auf ECR hochladen → .env-File anhand von Pipeline-Variablen generieren und auf S3 hochladen → Docker-Compose-File aus Projekt auf S3 hochladen → .env- und Docker-Compose-Dateien auf EC2-Instanz herunterladen → Deployment des aktuellen Images auf der EC2-Instanz

6.3 Betrieb und Wartung

- CI/CD: Damit Änderungen deployt werden können, müssen die produktiven Pipelines für Frontend und Backend gestartet werden.
- Backups: Die Datenbank wird über automatisierte Snapshots von RDS gesichert.

- Rollback: Anhand der versionierten Images auf ECR kann ein Rollback über das Docker-Compose-File veranlasst werden.

6.4 Übersicht



7 Querschnittliche Konzepte

In diesem Kapitel gehen wir auf generelle Prinzipien und Lösungsansätze ein, die in vielen Teilen der Architektur benutzt wurden. Der Inhalt bezieht sich vor allem aufs Frontend, da in den vorherigen Kapiteln das Backend mit deren Architektur bereits beleuchtet wurde.

7.1 State-Management (NgRx)

- Ziele: Vorhersehbare Zustände, Trennung von UI/Side-Effects, Debugbarkeit, Time-Travel.
- Store-Struktur: auth, organisation, position, product-offer, order, shipping, user, dashboard, contact sowie app.
- Selectors: einzige Datenquelle für Container-Komponenten (async pipe).
- Effects: kapseln Side-Effects (HTTP, Routing, Notifications).
- InitialLoad: InitialService dispatcht AppActions.initialDataLoad().
    - Lädt Benutzer (me), Organisationen, ConditionTypes, Settings.
    - UI wird erst nach initialDataLoaded gerendert.
- Entities: @ngrx/entity für performante Listen (IDs, Adapter, Memoized Selectors).
- Optimistic Updates: bei idempotenten Mutationen mit Rollback.

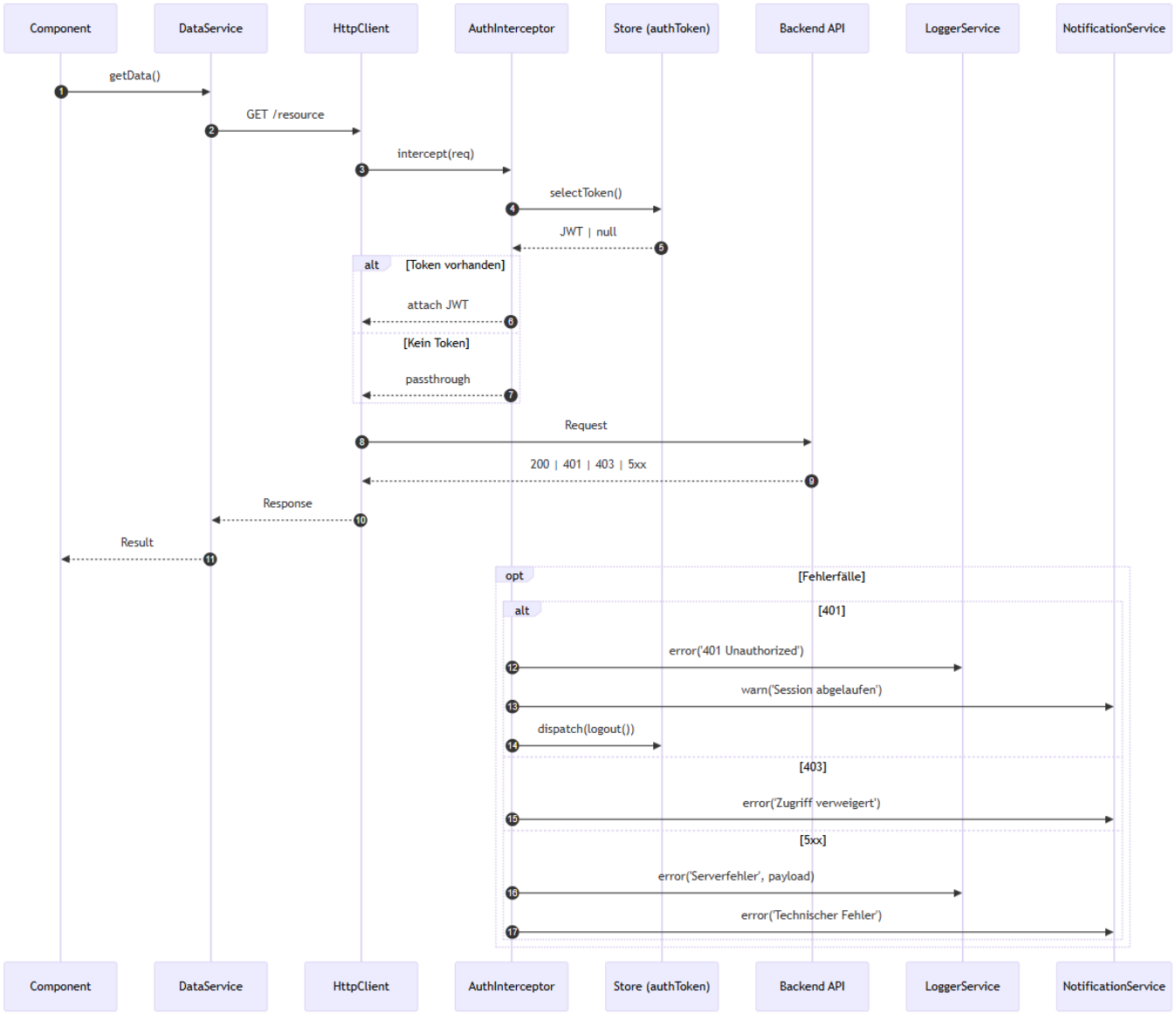
7.2 Globales Fehler- und Logging-Konzept

- Global Error Handler (ErrorHandler): Fängt ungefangene Fehler, schreibt Log, zeigt User-Feedback.
- HTTP-Fehler: zentral über Interceptor.
    - 401: Logout/Redirect → SignIn
    - 403: AccessDenied
    - 5xx: Notification + Logging
- LoggerService:
    - Logs: userId, tenantId, severity, message, stack, payload.
    - Channels: Console (Dev), Backend (Prod).
    - PII-Filter: keine sensiblen Daten in Logs.

7.3 Security (UI)

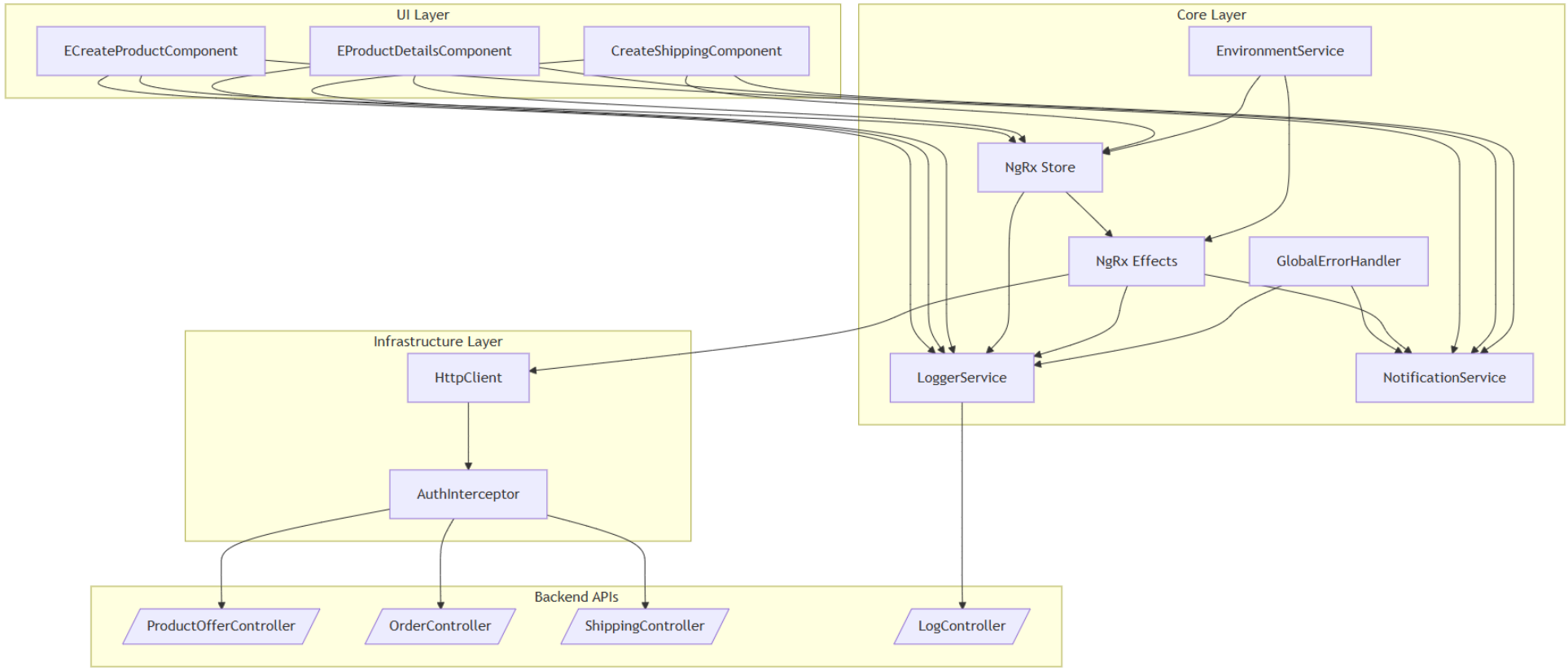
- Guards:
    - AuthGuard: nur eingeloggte User
    - RoleGuard: prüft Berechtigungen
    - TokenGuard: prüft Ablauf von JWT

- AuthInterceptor hängt JWT nur an API-Requests (Whitelist).
- Externe Formulare (z. B. Login und Registrierung) → kein Token.



7.4 UI-Architektur & Wiederverwendung

- Container/Presenter (Smart/Dumb).
- Shared: Pipes (ID-Format), Utils (Paginator, Error-Utils, JWT), UI-Controls.
- Feature-Module pro Domäne (Product, Order, Shipping).


UI Layer
ECreateProductComponent
EProductDetailsComponent
CreateShippingComponent
Core Layer
EnvironmentService
NgRx Store
NgRx Effects
GlobalErrorHandler
LoggerService
NotificationService
Infrastructure Layer
HttpClient
AuthInterceptor
Backend APIs
ProductOfferController
OrderController
ShippingController
LogController

8 Entschiedungslog

In der untenstehenden Tabelle sind unsere Entscheidungen festgehalten, die den Projektablauf massgebend beeinflusst haben.

Datum - Entscheidung	Beschreibung	Begründung Änderungsdatum
04.05.2025 – Mandantenfähigkeit	Die Datenbank wird mandantenfähig gestaltet, auch wenn aktuell nur ein Mandant (Schweizer Tafel) betrieben wird.	**Pro**: Zukunftssicherheit, da Erweiterungen (z. B. Betrieb für andere Organisationen oder Länder) ohne tiefgreifende Umstrukturierung möglich sind. **Kontra**: Initialer Mehraufwand für mandantenfähige Datenmodellierung.
04.05.2025 - Eigene Benutzerverwaltung	Eigene Benutzerverwaltung statt Azure Active Directory (AzureAD).	**Pro**: Unabhängigkeit von Microsoft-Diensten, besser steuerbar für Non-Profit-Use-Cases, flexibler für zukünftige Erweiterungen (z. B. externe Partner, OAuth). **Kontra**: Mehraufwand für das Handling der eigenen Benutzerverwaltung inkl. Datenspeicherung.
28.04.2025 - Rollenmodell	Benutzer haben Rollen (z. B. Admin, User, Superuser), die ihre Zugriffsrechte auf Ressourcen steuern.	**Pro**: Klare Trennung von Rechten und Verantwortlichkeiten (z. B. wer darf Organisationen erfassen oder Benutzer bearbeiten?). Einfach erweiterbar für neue Nutzergruppen. **Kontra**: Erfordert sorgfältiges Rechtekonzept und Testaufwand.
05.08.2025 - Hosting	Die gesamte Plattform wird über Amazon Web Services betrieben.	**Pro**: Einheitliches Setup für Frontend, Backend und Datenbank. Nutzung von AWS-Diensten für eine virtuelle Umgebung und Datenbank. Dieses Setup garantiert Skalierbarkeit, Stabilität und geringen Wartungsaufwand. **Kontra**: Einarbeitungszeit für das Kennenlernen von AWS.

9 Risikoanalyse

Die folgende Tabelle veranschaulicht die Risiken beurteilt nach Eintrittswahrscheinlichkeit und Auswirkung.

Wahrscheinlichkeit → Auswirkung ↓	**1 - unwahrscheinlich**	**2 - neutral**	**3 - wahrscheinlich**
1 - gering	**1.1.1** Kosten von Azure sind nicht mehr tragbar		
2 - mittel	**2.1.1** Fehlende Tools **2.1.2** Repos werden gelöscht	**2.2.1** Ausfall von Teammitgliedern	**2.3.1** Fehlende Ressourcen/Zeit um den gesamten Projektumfang abzudecken **2.3.2** Datenverlust durch Programmierfehler **2.3.3** Inkonsistenz der UI-Elemente und Darstellungsprobleme auf Endgeräten
3 - hoch	**3.1.1** Unklare Anforderungen oder Ziele **3.1.2** Kundenfeedback zu spät einholen	**3.2.1** Datenverlust durch Einsatz von unbekannten Technologien (bspw. Springboot) **3.2.2** Fehlerhaftes Statemanagement (NGRX)	**3.3.1** Sicherheitslücken beim Passwort-Handling

9.1 Massnahmen

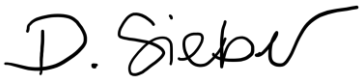
Die nachfolgenden Massnahmen sollen dafür sorgen, dass die Risiken mit hoher Auswirkung und wahrscheinlichem Eintritt verhindert werden können:

- **3.3.1**: Bestehendes Passwort-Framework benutzen
- **3.2.1, 3.2.2, 2.3.2, 2.3.3**: Testabdeckung hochhalten und zeitnah die Testumgebung hochziehen für End-to-End Testing / Testumgebung auch frühzeitig dem Kunden bereitstellen
- **2.3.1**: Zu Beginn des Projektes bereits viel Zeit investieren, damit genügend Puffer für herausfordernde/unerwartete Probleme vorhanden ist
- **2.2.1**: Regelmässige Auszeiten planen

Selbstständigkeitserklärung

Hiermit erklären wir, dass wir die vorliegende Masterarbeit im MAS Software Engineering mit dem Titel «Food-Bridge: Synergistische Vernetzung zur Förderung nachhaltiger Lebensmittelallokation» selbstständig und ohne unerlaubte fremde Hilfe angefertigt, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet und die den verwendeten Quellen und Hilfsmitteln wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht haben. Weiterhin erklären wir, dass wir keine durch Copyright geschützten Materialien (z.B. Bilder) in dieser Arbeit in unerlaubter Weise verwendet haben und in dieser Arbeit keine Adressen, Telefonnummern und andere persönliche Daten von Personen, die nicht zum Kernteam gehören, publizieren.

9000 St.Gallen, 11.09.2025:



Dominic Sieber

Philip Tobler