

## Dokumentation Semesterprojekt JEA

### Java Jakarta – Online Shop



# Inhaltsverzeichnis

Java Jakarta – Online Shop .....	1
1. Einleitung.....	5
1.1. Hintergründe dieser Arbeit .....	5
1.2. Ziele dieser Arbeit .....	5
2. Glossar .....	5
3. Theoretische Hintergründe.....	5
3.1. Definition von E-Commerce .....	6
4. Anforderungsanalyse .....	6
4.1. Stakeholder .....	6
4.2. Anforderungen.....	8
4.2.1. Funktionale Anforderungen.....	8
4.2.2. Nicht-Funktionale Anforderungen .....	9
4.2.3. Rechtliche Anforderungen.....	11
5. Abgrenzungen .....	12
6. GitLab .....	12
7. Vorgehensweise .....	13
8. Technologien .....	13
9. Architektur .....	15
9.1. Designentscheidungen.....	15
9.2. Backend .....	16
9.2.1. Projektstruktur .....	17
9.2.2. Endpoints .....	21
9.2.3. Datasource .....	22
9.2.4. Entitäten .....	23
9.2.5. Repository Interface .....	24
9.2.6. Controller.....	25
9.2.7. Service .....	26
9.3. Datenbank .....	26
9.3.1. Docker Container .....	26
9.3.2. Docker Compose File .....	27
9.3.3. Integration mit IDE.....	27
9.3.4. Data Model .....	28
9.3.5. Data Transfer Object (DTO) .....	30
9.4. Frontend Vue.js .....	30
9.4.1. Vue.js .....	30
9.4.1.1. Projektstruktur.....	31
9.4.1.2. Quasar .....	32

9.4.1.3.	Pinia .....	32
9.4.1.4.	Axios .....	34
9.5.	Bestellprozess .....	35
9.5.1.	Backend .....	35
9.5.1.1.	JPA-Modellierung .....	38
9.5.1.2.	Service .....	40
9.5.1.3.	Controller .....	41
9.5.1.4.	Interne Datenbank .....	42
9.5.2.	Frontend .....	43
9.5.2.1.	Vue-Stripe .....	43
9.5.2.2.	Einsatz im Projekt .....	43
10.	Deployment View .....	45
11.	Vorarbeiten .....	45
12.	Installationsanleitung .....	46
13.	Fazit .....	48
14.	Verzeichnis .....	50
14.1.	Literaturverzeichnis .....	50
14.2.	Abbildungsverzeichnis .....	50
14.3.	Tabellenverzeichnis .....	51
14.4.	Glossar .....	51

## Abstract

Während den vergangenen Monaten wurde im Unterricht des Faches «Java Enterprise Application» verschiedene Kernelemente bezüglich des Erstellens von Web-Applikationen behandelt. Dabei wurde die Aufsetzung eines neuen Projektes von Grund auf durchgenommen. Zudem wurden die verschiedenen Architektur Prinzipien vertieft, indem die häufigsten Architektur-Fehler angeschaut wurde und wie man diese vermeiden kann.

Das Ziel dieses Projekt ist es, das neu erlernte Wissen nicht nur in der Theorie zu beherrschen, sondern auch in die Praxis umzusetzen. Dafür eignet sich nichts besser als das neu Erlernte in einem Projekt selbst umzusetzen und die erarbeiteten Schritte in einer Arbeit festzuhalten.

# 1. Einleitung

Im ersten Kapitel die Motivation zur Auswahl des Themas genauer beschrieben. Dabei wird auf die Hintergründe sowie die Ziele der Arbeit eingegangen.

## 1.1. Hintergründe dieser Arbeit

In der heutigen digitalen Ära, in der E-Commerce ein wesentlicher Bestandteil des modernen Handels geworden ist, stehen Unternehmen vor der Herausforderung, ihre Präsenz online zu etablieren und effektive Online-Shops bereit zu stellen. Die Erstellung eines erfolgreichen Online-Shops erfordert jedoch eine sorgfältige Auswahl der richtigen Tools und Plattformen, die den individuellen Anforderungen des Unternehmens entsprechen. Mit der Vielzahl von Tools zur Erstellung von Online-Shops, die auf dem Markt verfügbar sind, ist es von entscheidender Bedeutung, Architekturpläne zu erstellen und mit Hilfe gemeinsamer Forschungen und Absprachen, die optimalen Lösungen zu finden.

## 1.2. Ziele dieser Arbeit

Im Rahmen der Modularbeit wird eine Web-Applikation spezifiziert und entwickelt. Diese Modularbeit widerspiegelt die verschiedenen Architekturstile und -konzepte, die während des Moduls behandelt werden. Die Lösung basiert auf Jakarta EE, wird um die Konzepte von Spring erweitert und mit in Quarkus erstellte REST Services ergänzt. Die verschiedenen Services oder Komponenten werden letztlich als eigenständige Microservices als Container in Docker und Kubernetes verteilt.<sup>1</sup>

# 2. Glossar

Das Glossar stellt alle wichtigen Begriffe und ihre Definitionen bereit. Es dient dazu, die verwendeten Begriffe zu beschreiben, zu definieren und dem Leser verständlich zu machen. Durch das Glossar soll jegliche Mehrdeutigkeit der Begriffe vermieden werden. Das Glossar ist im Anhang 14.4 am Ende des Dokuments zu finden.

# 3. Theoretische Hintergründe

In diesem Kapitel wird eine theoretische Grundlage für das Thema E-Commerce im Kontext der Erstellung von Online-Shops geschaffen. Es werden grundlegende Konzepte und Begriffe erläutert, um ein besseres Verständnis für die Modularbeit zu ermöglichen.<sup>2</sup>

---

<sup>1</sup> (Moodle, 2023)

<sup>2</sup> (Bloomenthal, 2023)

### 3.1. Definition von E-Commerce

Zu Beginn wird der Begriff "E-Commerce" definiert und erläutert. E-Commerce bezieht sich auf den elektronischen Handel, bei dem der Kauf und Verkauf von Produkten oder Dienstleistungen über das Internet erfolgt. Der E-Commerce umfasst den gesamten Prozess des Online-Kaufs, einschliesslich der Produktsuche, Auswahl, Bestellung, Zahlungstransaktionen, Lieferung und des Kundenservices. Durch den Einsatz digitaler Technologien ermöglicht der E-Commerce Unternehmen und Verbrauchern den Austausch von Waren und Dienstleistungen in einem elektronischen Umfeld. Diese Form des Handels hat sich in den letzten Jahren stark weiterentwickelt und bietet zahlreiche Vorteile wie globale Reichweite, 24/7-Verfügbarkeit und bequemen Zugang zu einer breiten Produktpalette. Der E-Commerce hat die Art und Weise, wie Unternehmen Geschäfte betreiben, und wie Verbraucher einkaufen, nachhaltig verändert.



Abbildung 1: E-Commerce

## 4. Anforderungsanalyse

In diesem Kapitel wird als erstes auf die Stakeholder eingegangen, um ein besseres Verständnis für die Interessensgruppen aufzubauen. Im weiteren Verlauf folgen die Anforderungen an den Online-Shop, diese wurden fiktiv mit den Stakeholdern betrachtet und bei Konflikten Kompromisse gefunden.

### 4.1. Stakeholder

Um das Potenzial zukünftiger Zielkonflikte zwischen den einzelnen Interessensgruppen rund um das Projekt zu vermeiden, wird bereits in der Anfangsphase des Projektes die Stakeholder identifiziert.

Dabei wird das Ziel verfolgt, von Beginn an die unterschiedlichen Interessensgruppen zu identifizieren, damit direkte oder indirekte Auswirkungen auf den Erfolg des Projektes berücksichtigt werden können. Anhand der erhaltenen Identifikation können Bedürfnisse, Erwartungen und Einflüsse der Stakeholder verstanden werden und in der Projektumsetzung begünstigt werden. Aufgrund dessen können auch Massnahmen erhoben werden, die mögliche Risiken minimieren oder sogar ausschliessen.

## Identifizieren der Stakeholder

Im folgenden Abschnitt wurde eine Liste der relevanten Stakeholder erstellt, welche mit dem Online-Shop in Verbindung stehen können. Anhand der fiktiv durchgeführten Interviews mit den beteiligten Personen am Projekt, wurden die Stakeholder identifiziert und deren Erwartungen und Bedürfnisse niedergeschrieben.<sup>3</sup>

Tabelle 1: Stakeholder

Stakeholder	Beschreibung
<b>Kunden</b>	Die Kunden sind die Hauptzielgruppe des Online-Shops und stehen mit ihren Bedürfnissen an vorderster Stelle. Natürlich wollen sie faire Preisen und neuwertige Pokémon-Karten kaufen
<b>Eigentümer / Mitarbeiter</b>	Die Verfasser dieser Arbeit sind auch gleich die Eigentümer des fiktiven Online-Shops. Da der Online-Shop noch in seiner Startup-Phase ist, werden erst zu einem späteren Zeitpunkt Mitarbeiter eingestellt.
<b>Lieferanten und Partner</b>	Externe Unternehmen oder Dienstleister, die Produkte und Dienstleistungen für den Online-Shop bereitstellen. Je nach Ergebnis der Evaluation, müssen noch weitere externe Dienste in das Projekt miteinbezogen werden, wie zum Beispiel ein Unternehmen für die Lieferung der Pokémon-Karten oder ein Zahlungsanbieter, welcher die Transaktions- und Zahlungsdienstleistungen für den Online-Shop übernehmen.
<b>Zahlungsanbieter</b>	Das Unternehmen, welches die Transaktions- und Zahlungsdienstleistungen für den Online-Shop anbietet.
<b>Konkurrenten</b>	Da es sich um einen fiktiven Online-Shop handelt, werden andere Online-Shops oder Unternehmen, welche in derselben Branche tätig sind, nicht beachtet oder berücksichtigt.
<b>Technologieanbieter</b>	Unternehmen, das bei der Evaluation am besten abschneidet und die Technologie (Hosting, Infrastruktur usw.) für den Online-Shop bereitstellen wird.

<sup>3</sup> (bwl-lexikon.de, 2023)

## 4.2. Anforderungen

Ein Softwaresystem dient einem bestimmten Zweck und dieser besteht darin, eigenständige Aufgaben wahrzunehmen oder ihre Nutzer bei der Wahrnehmung ihrer Aufgabe zu unterstützen. Bei der Erfassung der Anforderungen wird dann festgelegt, durch welche Funktionen das zu erstellende System diese Aufgaben wahrnehmen soll.

Bei den durchgeführten Interviews mit den fiktiven Stakeholdern wurden die funktionalen, nicht-funktionalen und rechtliche Anforderungen gemeinsam erarbeitet. Dabei wurde jeweils zwischen Muss- und Kann-Kriterium unterschieden. Ein Muss-Kriterium ist notwendigerweise erforderlich und sofern dieses Kriterium nicht erfüllt wird, kann der Online-Shop nicht eingesetzt werden. Auf ein Kann-Kriterium ("Nice-to-have") kann man verzichten, verbessert jedoch den Webbesuch für den Kunden. <sup>4, 5</sup>

### 4.2.1. Funktionale Anforderungen

Funktionale Anforderungen beschreiben die spezifischen Funktionen und Aufgaben, die ein System oder eine Software erfüllen muss, damit das entwickelte System die Bedürfnisse und Erwartungen der Benutzer gerecht wird.

Diese Anforderungen legen fest, welche Funktionalitäten das System bieten muss, um den gestellten Anforderungen gerecht zu werden.

Tabelle 2: Funktionale Anforderungen

ID	Bezeichnung	Beschreibung	Kriterium
FA-1	Warenkorb	Der Online-Shop sollte über einen Warenkorb verfügen, der es den Kunden ermöglicht, Produkte auszuwählen und diese dann sicher und einfach zu bestellen.  Zudem sollten die Produkte im Warenkorb für den Kunden mindestens zwei Stunden reserviert werden, sofern die Bestellung nicht gleich getätigt wird.	Muss
FA-2	Bestellabwicklung	Die Bestellabwicklung sollte Zahlungsoptionen, Lieferadressen, Rechnungsadresse, Versandmethoden und Bestellbestätigungen umfassen. Sowie alle Kosten auflisten, sofern Liefergebühren dazu kommen.	Muss
FA-3	Benutzerkonten	Kunden haben die Möglichkeit haben, Benutzerkonten zu erstellen, in denen sie ihre	Muss

<sup>4</sup> (Informationstechnologie im KMU, 2023)

<sup>5</sup> (Broy & Kuhrmann, 2020)



		persönlichen Informationen speichern, Bestellverlauf einsehen und Bestellungen verfolgen können.	
<b>FA-4</b>	Such- und Filterfunktion	Ein Online-Shop sollte über eine effektive Suchfunktion verfügen, mit der Benutzer nach bestimmten Produkten suchen und Filter anwenden können, um das gewünschte Produkt leichter zu finden.	Kann
<b>FA-5</b>	Kategorisierung	Produkte sollten in logischen Kategorien organisiert sein, um die Navigation für Benutzer zu erleichtern.	Kann
<b>FA-6</b>	Navigation	Um das Erlebnis des Besuchs des Online-Shops besser zu gestalten, sollte eine übersichtliche und leicht bedienbare Navigation vorhanden sein. Eine klare Menüstruktur und Filteroptionen können, das Stöbern und die Produktfindung verbessern.	Muss
<b>FA-7</b>	Kundenbewertungen und Feedback	Kunden sollten die Möglichkeit haben, Bewertungen und Feedback zu Produkten abzugeben, um anderen potenziellen Käufern bei ihrer Kaufentscheidung zu helfen.	Kann
<b>FA-8</b>	Bestandsverwaltung	Der Online-Shop sollte über ein effizientes Bestandsmanagementsystem verfügen, um sicherzustellen, dass Bestandsinformationen stets aktuell sind und keine Produkte überverkauft werden.	Muss

#### 4.2.2. Nicht-Funktionale Anforderungen

Nicht-Funktionale Anforderungen beschreiben die Eigenschaften und Qualitäten, die die Software aufweisen sollte, jedoch nicht direkt mit den spezifischen Funktionen zusammenhängen.

In der folgenden Tabelle werden die wichtigsten Nicht-Funktionalen Anforderungen aufgelistet und näher beschrieben:

*Tabelle 3: Nicht-Funktionale Anforderungen*

ID	Bezeichnung	Beschreibung	Kriterium
<b>NFA-1</b>	Leistungsanforderungen	Leistungsanforderungen legen fest, wie schnell und effizient der Online-Shop bestimmte Aufgaben erfüllen muss. Dazu gehören die Antwortzeit, Skalierbarkeit,	Muss

		Datenvolumen, Systemverfügbarkeit und viele weitere Merkmale.	
<b>NFA-2</b>	Produktdarstellung	Der Online-Shop sollte eine ansprechende Benutzeroberfläche bieten, auf der Produkte übersichtlich präsentiert werden. Dazu gehören Produktbilder, Beschreibungen, Preis, Verfügbarkeit usw.	Muss
<b>NFA-3</b>	Benutzerfreundlichkeit	Dies bezieht sich auf das Nutzererlebnis der Webseite, um den Besuchern eine einfache und angenehme Nutzung zu ermöglichen. Die Interaktion zwischen Benutzer und Online-Shop sollte intuitiv und fehlerfrei gestaltet werden. (Navigation, Responsive Design usw.)	Kann
<b>NFA-4</b>	Zuverlässigkeit	Der Online-Shop sollte stabil sein und seine Funktionen und Dienste zuverlässig bereitstellen, ohne unerwartete Ausfälle oder Unterbrechungen.	Muss
<b>NFA-5</b>	Sicherheit	Ein Online-Shop sollte Sicherheitsmassnahmen implementieren, um die sensiblen Informationen der Kunden zu schützen, einschliesslich sicherer Zahlungsabwicklung und Datenschutzrichtlinien. (Datenschutz und Zugriffskontrollen)	Muss
<b>NFA-6</b>	Wartbarkeit	Die technische Infrastruktur des Systems sollte so gestaltet sein, dass Wartungsarbeiten und Aktualisierungen effizient und ohne grössere Ausfallzeiten durchgeführt werden können, um eine reibungslose Betriebskontinuität zu gewährleisten.	Kann
<b>NFA-7</b>	Analytics und Reporting	Es ist wichtig, dass der Online-Shop über Analysefunktionen verfügt, um Einblick in das Kundenverhalten, Verkaufsdaten und andere Kennzahlen zu erhalten. Dies ermöglicht es dem Unternehmen, den Shop kontinuierlich zu verbessern.	Kann
<b>NFA-8</b>	Mobiles Erlebnis	Der Online-Shop sollte responsive gestaltet sein, um optimale Benutzererfahrungen auf verschiedenen Geräten wie Smartphones und Tablets zu gewährleisten.	Kann

### 4.2.3. Rechtliche Anforderungen

Ein Online-Shop muss verschiedene rechtliche Anforderungen beachten, um gesetzeskonform zu sein und um rechtliche Probleme zu vermeiden. Infolgedessen sind die wichtigsten rechtlichen Anforderungen in der Tabelle unterhalb aufgelistet. In dieser Tabelle wurde auf die Spalte der Muss/Kann-Kriteriums verzichtet, da es sich um gesetzliche Anforderungen handelt, welche für das Unternehmen essentiell sind.

*Tabelle 4: Rechtliche Anforderungen*

ID	Bezeichnung	Beschreibung
RA-1	Impressum	Ein Online-Shop muss ein Impressum bereitstellen, das leicht zugänglich ist und bestimmte Informationen enthält, wie den Namen und die Kontaktdaten des Shop-Betreibers, Handelsregisternummern und eine E-Mail-Adresse für die Kontaktaufnahme.
RA-2	Allgemeine Geschäftsbedingungen (AGB)	Ein Online-Shop sollte klare und verständliche Allgemeine Geschäftsbedingungen haben. Diese sollten Informationen über Vertragsbedingungen, Zahlungsbedingungen, Widerrufsrechte, Rückgabe- und Umtauschregelungen, Lieferbedingungen und Datenschutzbedingungen enthalten.
RA-3	Widerrufsrechte	Ein Online-Shop muss über die gesetzlichen Widerrufsrechte des Landes informiert zu sein, um den Kunden die Möglichkeit zu geben, innerhalb einer bestimmten Frist Waren zurückzugeben und den Kaufvertrag zu widerrufen.
RA-4	Datenschutz	Ein Online-Shop muss die Datenschutzbestimmungen einhalten und den Schutz der personenbezogenen Daten sicherstellen. Dies beinhaltet die Erfassung, Verarbeitung und Speicherung von Kundendaten in Übereinstimmung mit den geltenden Datenschutzgesetzen.
RA-5	Verbraucherrechte	Ein Online-Shop muss die Rechte der Verbraucher schützen und sicherstellen, dass Kunden korrekte Informationen über Produkte, Preise, Versandkosten, Lieferzeiten und Garantien erhalten.

## 5. Abgrenzungen

Aufgrund des begrenzten Zeitrahmens der Arbeit konnten nicht alle Aspekte des E-Commerce umfassend berücksichtigt werden. Daher wurden in diesem Projekt die folgenden Bereiche nicht behandelt.

- Administration des Lagerbestandes
- Anbietung einer Verstand-Dienstleistung
- Anbindung eines Mail-Servers (Zahlungsbestätigungen in Form von Mail)
- Bewertung der Produkte inkl. Kommentarsektor
- Adresse des Kunden («cities» Tabelle) wird nicht gebraucht, da kein Lieferdienst genutzt wird

## 6. GitLab

In diesem Projekt wurde mit dem Branching-Modell «Git-Flow» gearbeitet. Dies definiert einen Satz von Regeln und Konventionen für das Branching, Zusammenführen und Veröffentlichen von Code. Der Git-Flow erleichtert die Zusammenarbeit in Teams und ermöglicht eine klare Strukturierung des Entwicklungsprozesses. In der Abbildung 2 grafisch dargestellt.

Die wichtigsten Branches und ihre Verwendung:

- **Master-Branch:** Enthält den Code der Hauptentwicklungslinie. Er sollte immer funktionsfähig und bereit sein.
- **Develop-Branch:** Der Develop-Branch ist der Ausgangspunkt für die Entwicklung neuer Features. Er enthält den aktuellen Entwicklungsstand und dient als Integrationsbereich für Feature-Banches. Änderungen, die in den Develop-Branch zusammengeführt werden, sollten funktionsfähig sein und eine grundlegende Qualitätssicherung durchlaufen haben.
- **Feature-Banches:** Für die Entwicklung neuer Features oder Funktionen werden Feature-Banches erstellt. Jeder Feature-Branch wird von der aktuellen Version des Develop-Banches abgezweigt und enthält den Code für das spezifische Feature. Feature-Banches werden normalerweise von einem Entwickler erstellt und getestet, bevor sie in den Develop-Branch zusammengeführt werden.

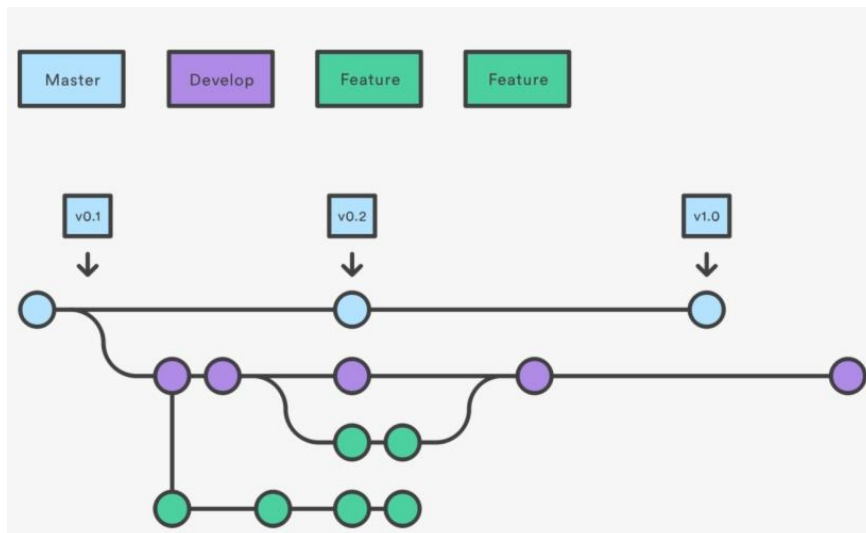


Abbildung 2: Git Work-Flow<sup>6</sup>

## 7. Vorgehensweise

Die Zusammenarbeit an einem Software-Projekt erfordert eine koordinierte Teamarbeit, um Effizienz, Qualität und Nachverfolgbarkeit zu gewährleisten. Einer der zentralen Komponenten für einen reibungslosen Teamwork ist die Nutzung der Versionsverwaltung GitLab.

Zu Beginn wurde das Projekt grob in Backend und Frontend unterteilt. Dabei übernahm Nicolas die Implementierung des Frontend mit Vue.js, während Naveen und Nina gemeinsam am Backend arbeiteten.

Viele zentrale Kernelemente wobei Frontend und Backend Hand in Hand arbeiten, wurde gemeinsam in virtuellen Meetings besprochen, diskutiert, implementiert, Lösungen gesucht und gepusht.

Während des ganzen Projektes wurde darauf geachtet mit Hilfe von Inline-Comments und Javadoc, die Codeabschnitte möglichst nachvollziehbar und simple zu erläutern, damit es keine Missverständnisse gibt.

## 8. Technologien

In diesem Kapitel werden die genutzten Technologien unseres Projektes erläutert und der Zusammenhang der einzelnen Komponenten analysiert. In den späteren Kapiteln wird genauer darauf eingegangen, welche Rolle die vorgestellten Technologien in unserem Projekt haben.

<sup>6</sup> (Abbildung Git Work-Flow , 2023)

Folgende Technologien wurden eingesetzt:

### **IntelliJ (IDE):**

IntelliJ IDEA ist eine integrierte Entwicklungsumgebung (IDE) für die Softwareentwicklung. Es bietet eine Vielzahl von Tools und Funktionen, um Entwicklern bei der effizienten Entwicklung von Code zu unterstützen. IntelliJ unterstützt verschiedene Programmiersprachen und Frameworks, einschliesslich Java, Kotlin, Scala und viele mehr.<sup>7</sup>

### **Docker:**

Docker ist eine Open-Source-Plattform, die es Entwicklern ermöglicht, Anwendungen in Containern zu erstellen, zu verpacken und bereitzustellen. Container sind leichte, eigenständige Ausführungsumgebungen, die Anwendungen und ihre Abhängigkeiten kapseln. Docker ermöglicht die einfache Bereitstellung von Anwendungen in verschiedenen Umgebungen, ohne dass auf dem Hostsystem spezifische Abhängigkeiten installiert werden müssen.<sup>8</sup>

### **Jakarta EE:**

Jakarta EE (ehemals Java Enterprise Edition) ist ein Framework für die Entwicklung von Unternehmensanwendungen in Java. Es bietet eine Reihe von Spezifikationen und APIs, die die Entwicklung von robusten, skalierbaren und sicheren Anwendungen erleichtern. Jakarta EE umfasst Technologien wie Servlets, Enterprise JavaBeans (EJB), JavaServer Faces (JSF), Java Persistence API (JPA) und viele mehr.

### **Spring:**

Spring ist ein leistungsfähiges Java-Framework für die Entwicklung von Unternehmensanwendungen. Es bietet eine umfangreiche Palette von Funktionen und Modulen, die die Entwicklung erleichtern und Best Practices fördern. Spring ermöglicht die Entwicklung von Anwendungen mit einer modularen, gut strukturierten Architektur und unterstützt verschiedene Technologien und Integrationen, einschließlich Datenbankzugriff, Webentwicklung, Sicherheit, Messaging und mehr.

### **Vue.js:**

Vue.js ist ein JavaScript-Framework für die Entwicklung von Benutzeroberflächen. Es ermöglicht die Erstellung interaktiver und reaktionsfähiger Webanwendungen. Vue.js bietet eine klare und einfache Syntax, die es Entwicklern erleichtert, Benutzeroberflächenkomponenten zu erstellen und sie in einer komponentenbasierten Architektur zu organisieren. Vue.js wird häufig in Kombination mit anderen Backend-Frameworks wie Jakarta EE oder Spring verwendet, um eine vollständige Webanwendung zu erstellen.

---

<sup>7</sup> (IntelliJ, 2023)

<sup>8</sup> (Docker, 2023)

## Maven:

Maven hat sich als ein beliebtes und weit verbreitetes Build-Management-Tool in der Java-Welt etabliert. Es vereinfacht die Projektkonfiguration, Abhängigkeitsverwaltung und Build-Prozesse erheblich und ermöglicht eine effiziente Entwicklung und Bereitstellung von Java-Anwendungen.

## 9. Architektur

Die Architektur eines Online-Shops umfasst die Organisation und Strukturierung der verschiedenen Komponenten, um eine effiziente und skalierbare E-Commerce-Anwendung zu erstellen. Bei der Verwendung von Spring Boot, Vue.js und Postgres können wir eine moderne und leistungsstarke Architektur realisieren. Dieses Kapitel dient als Übersicht über die einzelnen Komponenten und deren Zusammenspiel.

In den folgenden Abschnitten wird genauer auf die Architektur-Pläne eingegangen. Für das Frontend wurde Vue.js verwendet und Spring Boot für das Backend.

Wie in der Abbildung 2 erkennbar, exportiert REST API mit Spring Web MVC und interagiert mit der PostgreSQL Datenbank mit Hilfe des Spring DATA JPA. Der Client sendet HTTP-Requests und ruft HTTP-Responses ab und erhält die Daten auf den Komponenten, die der View übergeben werden.

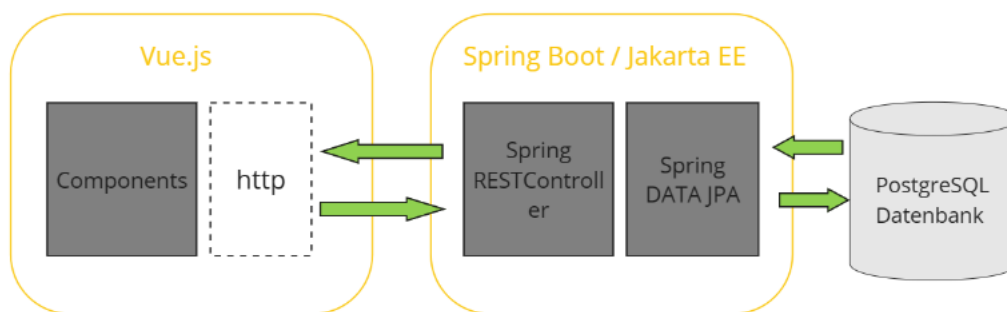


Abbildung 3: Spring Boot Architektur

### 9.1. Designentscheidungen

Der Online-Shop folgt dem Designansatz des Modularen Designs. Dabei wird die Anwendung in separate, unabhängige Module unterteilt. Jedes Modul erfüllt eine spezifische Funktion oder Aufgabe und kann eigenständig entwickelt, getestet, gewartet und wiederverwendet werden. Diese modulare Struktur bringt viele Vorteile mit sich wie die Wiederverwendbarkeit, Wartbarkeit, Kollaboration, Skalierbarkeit, Testbarkeit und Entkoppelung.

Bei der Umsetzung eines modularen Designs muss jedoch auf folgende Punkte geachtet werden:

- Definieren klarer Schnittstellen
- Minimierung von Abhängigkeiten
- Einhaltung des Singel-Responsibility-Prinzips
- Verwendung von Dependency Injection

Der Onlineshop wurde in folgende Module unterteilt:

*Tabelle 5: Projekt-Module*

Modul	Technologie	Details
Backend	Spring Boot, Jakarta EE	Kapitel 9.2
Datenbank	PostgreSQL, Docker	Kapitel 9.3
Frontend	Vue.js	Kapitel 9.4
Zahlungsabwicklung	Stripe	Kapitel 9.5

## 9.2. Backend

Das Backend wurde mit Spring in einem sogenannten Schichtenmodell aufgebaut. Das Schichtenmodell ist eine bewährte Praxis, um die Verantwortlichkeiten in der Anwendung zu organisieren und die Trennung von Aufgaben und Wiederverwendbarkeit des Codes zu fördern.

Darüber hinaus bietet Spring Mechanismen für Dependency Injection und andere Funktionen, um die Entwicklung von Anwendungen zu erleichtern und die Wartbarkeit zu verbessern.

In unserem Kontext mit Spring werden diese Schichten durch verschiedene Spring-Komponenten implementiert:

### Controller:

Ist die Schicht, welche die Benutzeranfragen entgegennimmt und darauf reagiert. In Spring wird der Controller mit der Annotation `@Controller` markiert. Er ist für die Verarbeitung der HTTP-Anfragen zuständig, extrahiert die relevanten Daten aus den Anfragen und ruft die entsprechenden Services auf, um die gewünschte Aktion auszuführen. Der Controller gibt normalerweise eine HTTP-Antwort zurück, die dem Client geliefert wird.

### Service:

Der Service enthält die Business-Logik und stellt spezifische Funktionen und Operationen zur Verfügung. In Spring wird der Service mit der Annotation `@Service` veranschaulicht. Der Service kapselt Operationen, die von Controllern aufgerufen werden und stellt eine abstrakte Schnittstelle (Interface) bereit, um bestimmte Aufgaben zu erledigen. Er interagiert mit den Repositories, um auf die Datenbank zuzugreifen und Daten zu verarbeiten.



## Repository:

Das Repository ist für den Zugriff auf die Datenbank zuständig. In Spring wird das Repository mit der `@Repository`-Annotation markiert. Es stellt eine Schnittstelle bereit, um Daten zu lesen, zu schreiben oder zu löschen und ermöglicht eine einfachere Interaktion mit der zugrunde liegenden Datenquelle. Das Repository abstrahiert die Datenbankoperationen und bietet Methoden zum Durchführen von Abfragen und zum Speichern von Daten.

Der Zusammenhang besteht darin, dass die Controller-Schicht die Benutzeranfrage entgegennimmt und die gewünschte Aktion auslöst. Die Controller nutzen die Serviceschicht, um die Geschäftslogik auszuführen und greifen auf die Repositories zu, um auf Daten zuzugreifen. Der Service wiederum kommuniziert mit dem Repository, um Datenoperationen durchzuführen und die Ergebnisse an den Controller zurückzugeben. Diese Zusammenhänge wurden in Abbildung 3 dargestellt.

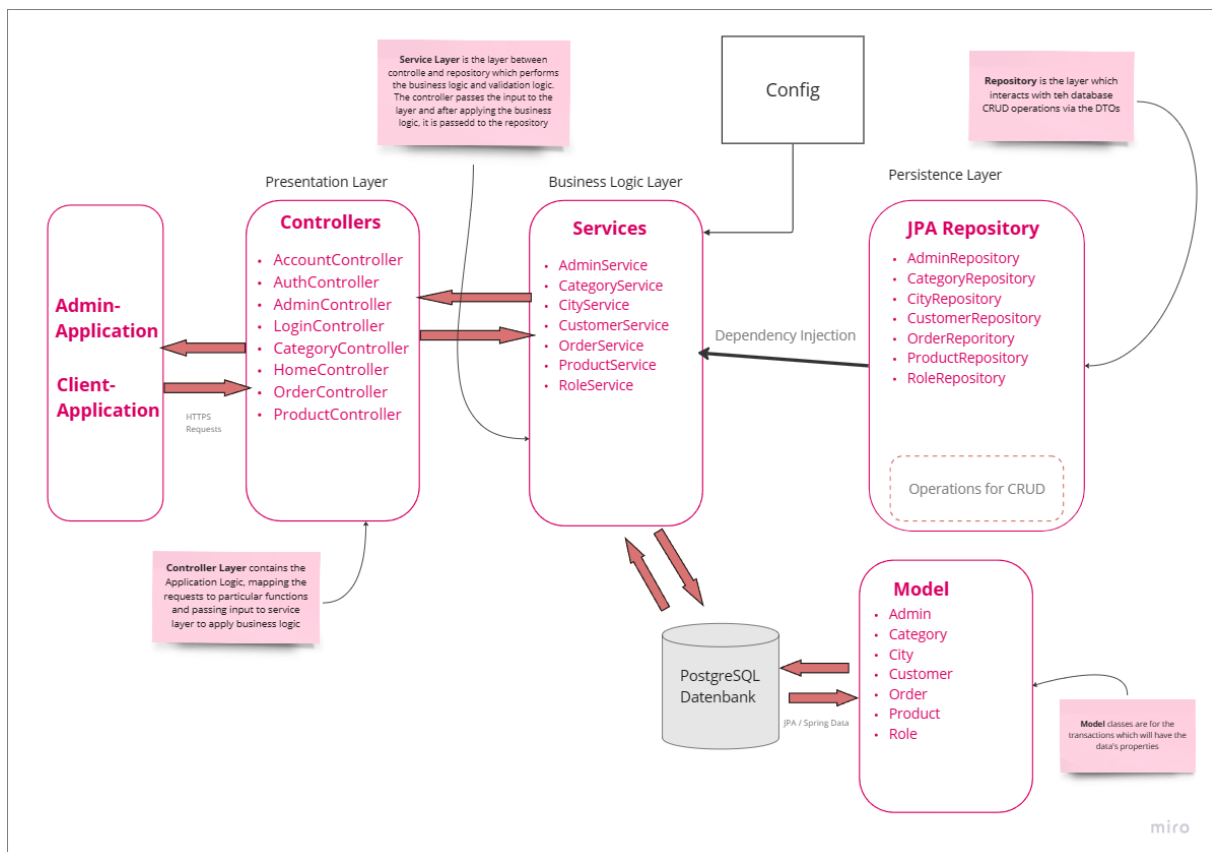


Abbildung 4: Controller-Service-Repository Architecture

### 9.2.1. Projektstruktur

In diesem Unterkapitel wird die Projektstruktur erläutert. Dabei wird noch auf die wichtigsten Abhängigkeiten (eng. Dependencies) eingegangen.

Maven definiert eine standardisierte Projektstruktur, die das Organisieren von Quellcode, Ressourcen und Konfigurationsdateien erleichtert. Das Hauptquellcodeverzeichnis befindet sich im Ordner "src/main/java", während die Ressourcen im Ordner "src/main/resources" platziert werden. Dabei

verwendet Maven die XML-basierte Datei "pom.xml" (Project Object Model), um Projektinformationen, Abhängigkeiten, Build-Konfigurationen und andere Metadaten zu speichern. Die "pom.xml" dient als zentrales Steuerungselement für das Maven-Projekt.

jea\_pibs

```
|— admin
| |— pom.xml
| |— src
| |   |— main
| |   |   |— java
| |   |   |   |— ch.ffhs.admin.admin
| |   |   |   |   |— config
| |   |   |   |   |— AdminConfiguration.class
| |   |   |   |   |— AdminDetails.class
| |   |   |   |   |— AdminServiceConfig.class
| |   |   |   |   |— controller
| |   |   |   |   |   |— CategoryController.class
| |   |   |   |   |   |— LoginController.class
| |   |   |   |   |   |— ProductController.class
| |   |   |   |   |   |— AdminApplication.class
| |   |   |   |   |— ressourcen
| |   |   |   |   |   |— application.properties
| |   |— test
| |   |   |— java
| |   |   |   |— ch.ffhs.admin.admin
| |   |   |   |   |— controller
| |   |   |   |   |   |— LoginControllerMockTest.class
|— customer
| |— pom.xml
| |— src
| |   |— main
| |   |   |— java
| |   |   |   |— ch.ffhs.customer
| |   |   |   |   |— config
| |   |   |   |   |— CustomerConfiguration.class
| |   |   |   |   |— CustomerDetails.class
| |   |   |   |   |— CustomerConfigService.class
| |   |   |   |   |— controller
| |   |   |   |   |   |— CategoryController.class
| |   |   |   |   |   |— LoginController.class
| |   |   |   |   |   |— OrderController.class
```

```

|   |   |   |   └─ ProductController.class
|   |   |   └─ CustomerApplication.class
|   |   └─ ressources
|   |       └─ application.properties
|   |           └─ data.sql
|   └─ test
|       └─ java
|           └─ ch.ffhs.admin.admin
|               └─ CustomerControllerTest.class
|               └─ ProductControllerTest.class
|
|
|
|   └─ infrastructure
|       └─ postgres-data
|           └─ docker-compose.yml
|
|
|
|   └─ library
|       └─ pom.xml
|       └─ src
|           └─ main
|               └─ java
|                   └─ ch.ffhs.library.library
|                       └─ dto
|                           └─ AdminDto.class
|                           └─ CategoryDto.class
|                           └─ CheckoutItemDto.class
|                           └─ CustomerDto.class
|                           └─ LoginDto.class
|                           └─ OrderDto.class
|                           └─ OrderItemsDto.class
|                           └─ ProductDto.class
|                           └─ StripeResponse.class
|                       └─ model
|                           └─ Admin.class
|                           └─ Category.class
|                           └─ City.class
|                           └─ Customer.class
|                           └─ Order.class
|                           └─ OrderItem.class
|                           └─ Product.class

```

```

|   |   |   |   └─ Role.class
|   |   |   └─ repository
|   |   |   └─ AdminRepository.class
|   |   |   └─ CategoryRepository.class
|   |   |   └─ CityRepository.class
|   |   |   └─ CustomerRepository.class
|   |   |   └─ OrderRepository.class
|   |   |   └─ OrderItemRepository.class
|   |   |   └─ ProductRepository.class
|   |   |   └─ RoleRepository.class
|   |   └─ service
|   |   └─ AdminService.class
|   |   └─ CategoryService.class
|   |   └─ CityService.class
|   |   └─ CustomerService.class
|   |   └─ ProductService.class
|   |   └─ RoleService.class
|   |   └─ impl
|   |   └─ AdminServiceImpl.class
|   |   └─ CategoryServiceImpl.class
|   |   └─ CityServiceImpl.class
|   |   └─ CustomerServiceImpl.class
|   |   └─ OrderServiceImpl.class
|   |   └─ OrderItemServiceImpl.class
|   |   └─ ProductServiceImpl.class
|   |   └─ RoleServiceImpl.class
|   └─ CustomerApplication.class
|   └─ ressources
|   └─ application.properties
|   └─ test
|   └─ java
|   └─ ch.ffhs.library.library.model
|   └─ AdminTest.class
└─ .gitignore
└─ README.md

```

### Spring Boot Dependencies:

Für die Entwicklung der API wurde wie bereits erwähnt Spring Boot verwendet. Ausserdem verwenden wir JPA und die PostgreSQL-Datenbank für die Persistenzseite des Backend.

Die erforderlichen Abhängigkeiten (eng. Dependencies) von Spring wurden in die pom.xml-Datei importiert.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Abbildung 5: Spring Boot Dependencies

### PostgreSQL Dependencies:

```
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
</dependency>
```

Abbildung 6: PostgreSQL Dependencies

Um das Setup zu vereinfachen, haben wir Spring Initializr verwendet, somit wurden die benötigten Abhängigkeiten schnell eingerichtet.

### 9.2.2. Endpoints

In der folgenden Abbildung wird eine Übersicht über die APIs präsentiert, welche Spring Boot exportiert. Die Abbildung wurde mit Hilfe von Swagger generiert welche wiederum von SpringDoc erstellt wird.

login-controller		^
PUT	/update-user/{id}	▼
POST	/register	▼
POST	/login	▼
GET	/users	▼
GET	/get-user/{id}	▼
DELETE	/delete-user/{id}	▼
order-controller		^
POST	/order/create-checkout-session	▼
POST	/order/add	▼
GET	/order/{id}	▼
GET	/order/all	▼
product-controller		^
GET	/products	▼
GET	/products/{id}	▼
GET	/products-in-category/{id}	▼
category-controller		^
GET	/categories	▼
GET	/categories/{id}	▼

Abbildung 7: Swagger Endpoints

### 9.2.3. Datasource

Unter `admin/src/main/resources` befinden sich `application.properties` mit den Applikationseigenschaften. Dabei kann man im Code-Ausschnitt sehen, dass die AdminApplication mit Port 8019 läuft. Die `application.properties` der CustomerApplication sieht abgesehen vom Port 8020 identisch aus und wird deshalb nicht dargestellt.

```

server.port=8019
server.servlet.context-path=/admin
spring.thymeleaf.check-template-location=true
spring.thymeleaf.prefix=classpath:/templates/
spring.thymeleaf.suffix=.html
spring.thymeleaf.mode=LEGACYHTML5
spring.thymeleaf.encoding=UTF-8
spring.thymeleaf.servlet.content-type=text/html
spring.thymeleaf.cache=false
#Datasource Postgres
spring.datasource.url=jdbc:postgresql://localhost:5432/postgres
spring.datasource.username=postgres
spring.datasource.password=postgres
spring.sql.init.mode=always
spring.jpa.generate-ddl=true
#Hibernate Properties
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
#Hibernate ddl auto (create, create-drop, validate, update)
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
#Swagger Properties
springdoc.api-docs.path=/api-docs
springdoc.swagger-ui.path=/swagger

```

Abbildung 8: application.properties (admin)

Hibernate wird automatisch die Datenbank-Tabellen erstellen, somit muss man lediglich im application.properties Konfigurations-File alle fehlenden Angaben manuell angeben.

- *spring.datasource.username* und *spring.datasource.password* sind die gleichen Eigenschaften wie bei der Datenbank-Installation angegeben.
- Spring Boot benutzt Hibernate für die JPA Implementation und konfiguriert PostgreSQLDialect für die Datenbank.
- *Spring.jpa.hibernate.ddl-auto* wird für die Datenbank-Initialisierung verwendet. Dabei wird der Wert auf "update" gesetzt, damit Tabellen erstellt werden können und automatisch in der Datenbank mit den definierten Data Model korrespondieren. Jede Änderung des Models aktiviert ein Update der Tabellen

## 9.2.4. Entitäten

Entitäten in JPA (Jakarta Persistence API) sind nichts anderes als JavaBeans, die Daten darstellen, die in der Datenbank persistiert werden können. Eine Entität repräsentiert eine Tabelle, die in einer Datenbank gespeichert ist. Jede Instanz einer Entität stellt eine Zeile in der Tabelle dar.

Als Beispiel wurde im unteren Code-Ausschnitt die Entitäts-Klasse «Admin» abgebildet.

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity
@Table(name = "admins", uniqueConstraints = @UniqueConstraint(columnNames
= {"email","username"} ))

public class Admin {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "admin_id")
    private Long id;
    private String firstName;
    private String lastName;
    private String username;
    private String email;
    private String password;
    @ManyToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
    @JoinTable(name = "admins_roles", joinColumns = @JoinColumn(name =
"admin_id", referencedColumnName = "admin_id"),
        inverseJoinColumns = @JoinColumn(name = "role_id",
referencedColumnName = "role_id"))
    private Collection<Role> roles;
}
```

Übersicht wichtigster Annotations:

- **@Entity**: zeigt an, dass die Java-Klasse persistent ist
- **@Table**: biete Tabellen an, welche die Entität mappen
- **@Id**: wird für den Primary Key gebraucht
- **@GeneratedValue**: wird benötigt, um die "generation strategy" für den Primary Key zu definieren
- **@Column**: wird gebraucht, um eine Spalte in der Datenbank mit dem kommentierten Feld zu mappen

### 9.2.5. Repository Interface

Bei Spring Data JPA ist Spring für die Generierung der Implementierung und Registrierung der Spring-Komponenten verantwortlich. Um es zu nutzen, muss das JPA-Repository geerbt werden. Aufgrund dessen kann man die grundlegenden Methoden der CRUD-Repository-Schnittstelle (JPA-Repository) implementieren. Auch mit der @Query-Annotation kann man eigene Methoden implementieren.

Als Beispiel wurde im unteren Code-Ausschnitt die Repository-Klasse «AdminRepository» abgebildet.

```
@Repository
public interface AdminRepository extends JpaRepository<Admin, Long> {
    Admin findByUsername(String username);
}
```



## 9.2.6. Controller

In der Applikation wurden Controller benutzt, um APIs zur Verfügung zu stellen, um die CRUD-Operationen durchzuführen. Im unten aufgeführten Code-Ausschnitt wurde als Beispiel der CategoryController dargestellt.

```
/**
 * CategoryController, which is responsible for processing requests related
 * to categories
 */
@RestController
public class CategoryController {
    // this annotation injects the CategoryService which allows the
    // controller
    // to access the service to execute business logic related to
    // categories
    @Autowired
    private CategoryService categoryService;

    // injects CategoryRepository
    @Autowired
    private CategoryRepository categoryRepository;

    /**
     * method is called when an HTTP GET request is sent to the /categories
     * URL
     *
     * @return ResponseEntity with categories and 200
     */
    @GetMapping("/categories")
    public ResponseEntity<List<Category>> categories() {
        return new ResponseEntity<>(categoryService.findAll(),
        HttpStatus.OK);
    }

    /**
     * method is called when an HTTP GET request is sent to the
     * /categories/{id} URL
     * and is searching for a category by its ID
     *
     * @param id of the category
     * @return ResponseEntity with category by id and Http Status
     */
    @GetMapping("/categories/{id}")
    public ResponseEntity<?> findCategoryById(@PathVariable("id") Long id)
    {
        try {
            return new ResponseEntity<>(categoryService.findById(id),
            HttpStatus.OK);
        } catch (Exception e) {
            return new ResponseEntity<>(e.toString(), HttpStatus.CONFLICT);
        }
    }
}
```

### 9.2.7. Service

In der Applikation wurden Services benutzt, um die Komponenten der Geschäftslogik mit bestimmten Funktionen bereitzustellen. Im unten aufgeführten Code-Ausschnitt wurde als Beispiel der CustomerService dargestellt.

```
/**
 * The CustomerService interface thus defines the basic
 * operations required for managing customers
 */
public interface CustomerService {
    Customer save(CustomerDto customerDto);

    Customer findByUsername(String username);

    List<CustomerDto> findAllDto();

    Customer findCustomerById(Long id);

    Customer findCustomerByEmail(String email);

    CustomerDto update(CustomerDto customerDto);
}
```

## 9.3. Datenbank

Für das Projekt wurde die PostgreSQL-Datenbank, besser bekannt als "Posrgres", genutzt. PostgreSQL ist ein leistungsfähiges, objekt-relationales Open-Source-Datenbanksystem, das seit über 35 Jahren aktiv entwickelt wird.<sup>9</sup>

PostgreSQL ist bekannt für seine Zuverlässigkeit und Robustheit. PostgreSQL unterstützt die Transaktions-Eigenschaften ACID (Atomcity, Consistency, Isolation, Durability), um Datenkonsistenz und Ausfallsicherheit zu gewährleisten. Aufgrund seiner Erweiterbarkeit und Skalierbarkeit eignet sich die Datenbank perfekt für unser kleines Projekt.

### 9.3.1. Docker Container

Um die Portabilität der Datenbank zu gewährleisten, wird die PostgreSQL-Datenbank in einem Docker-Container ausgeführt. Durch die Verwendung von Docker wird die PostgreSQL-Datenbank in einem Container isoliert und kann unabhängig von der zugrundeliegenden Infrastruktur ausgeführt werden.

Zudem hilft Docker bei der Bereitstellung der Datenbank in der produktiven Umgebung. Der Container kann schnell gestartet werden und die Konfigurationen können ganz einfach über Umgebungsvariablen oder die Docker-Compose-Datei angepasst werden. In der unteren Abbildung sieht man einen Container mit einer PostgreSQL-Datebank Instanz.

---

<sup>9</sup> (PostgreSQL, 2023)



Abbildung 9: PostgreSQL-Container

### 9.3.2. Docker Compose File

Ein Docker Compose File wird verwendet, um die Konfigurationen und das Setup eines Docker-Containers zu definieren und zu verwalten. Es ermöglicht die Definition und Ausführung von Multi-Container-Anwendungen von Multi-Container-Anwendungen als Teil einer Docker-Umgebung.

Das Docker Compose File basiert auf der YAML-Syntax, die einfach zu lesen und zu schreiben ist. Mit Hilfe des Files können Anwendungen schnell und reproduzierbar erstellt werden. Es erleichtert somit die Zusammenarbeit im Team, da alle Teammitglieder denselben Konfigurationscode verwenden können, um den Container lokal bereitzustellen.

Im folgenden Code-Abschnitt sieht man den Inhalt des genutzten Docker Compose File für das Projekt. Die jeweils festgelegten Konfigurationen sind mit einem Kommentar (#) im Abschnitt beschrieben.

```
version: '3.7'
services:
  postgres:
    # Official Postgres image from DockerHub
    image: postgres:latest
    environment:
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres
    logging:
      options:
        max-size: 10m
        max-file: "3"
    ports:
      # By default, a Postgres database is running on the 5432 port
      - '5432:5432'
    volumes:
      # You don't need to create the `db-data` folder, docker Compose will
      # do it for you
      # we share the folder `db-data` in our root repository, with the
      # default PostgreSQL data path
      - postgres-data:/var/lib/postgresql/data/
      # copy the sql script to create tables
      #- ./sql/create_tables.sql:/docker-entrypoint-
      initdb.d/create_tables.sql
    volumes:
      postgres-data:
```

### 9.3.3. Integration mit IDE

Um PostgreSQL mit dem Framework Jakarta EE zu nutzen, müssen zuvor einige Schritte in der gewünschten IDE ausgeführt werden.

Als erstes muss der PostgreSQL JDBC-Treiber heruntergeladen und dem Projekt hinzugefügt werden, um die Verbindung mit der Datenbank zu ermöglichen. Dabei besteht die Möglichkeit den Treiber manuell hinzuzufügen oder wie bei uns, gleich in das Maven-Build-Tool den Treiber als Abhängigkeit zu konfigurieren.

Danach wird für die Jakarta EE Anwendung die DataSource konfiguriert, darunter versteht man die logische Datenbankschnittstelle für eine Datenbankverbindung. Um eine Verbindung herzustellen werden folgende Daten benötigt:

- Host
- Port
- Datenbankname
- Benutzername und Passwort

#### **9.3.4. Data Model**

Das Data Model von Spring basiert auf dem Konzept des objektrelationalen Mapping (ORM). Dabei werden Datenbanktabellen in Java-Klassen abgebildet und die Datenbankoperationen werden über entsprechende Methodenaufrufe auf diesen Klassen ausgeführt. Das Data Model ermöglicht die Interaktion mit der Datenbank, ohne dass SQL-Code direkt geschrieben werden muss.

##### **Hibernate:**

Hibernate ist eine Open-Source-Implementierung der JPA-Spezifikation. Es ist ein ORM-Framework, das die JPA-Spezifikation umsetzt und zusätzlichen Funktionen und Erweiterungen anbietet. Hibernate erleichtert die Interaktion mit relationalen Datenbanken, indem es automatisch die Mapping-Beziehungen zwischen Java-Klassen und Datenbanktabellen verwaltet. Es ermöglicht das Speichern, Aktualisieren und Abfragen von Objekten in der Datenbank, ohne dass SQL-Code manuell geschrieben werden muss.

Aus diesem Grund haben auch wir Hibernate eingesetzt. Dabei wurden mit den hinterlegten Angaben der Entitäts-Klassen (siehe Kapitel 9.2.4 «Entitäten») folgendes Datenbankmodell erstellt:

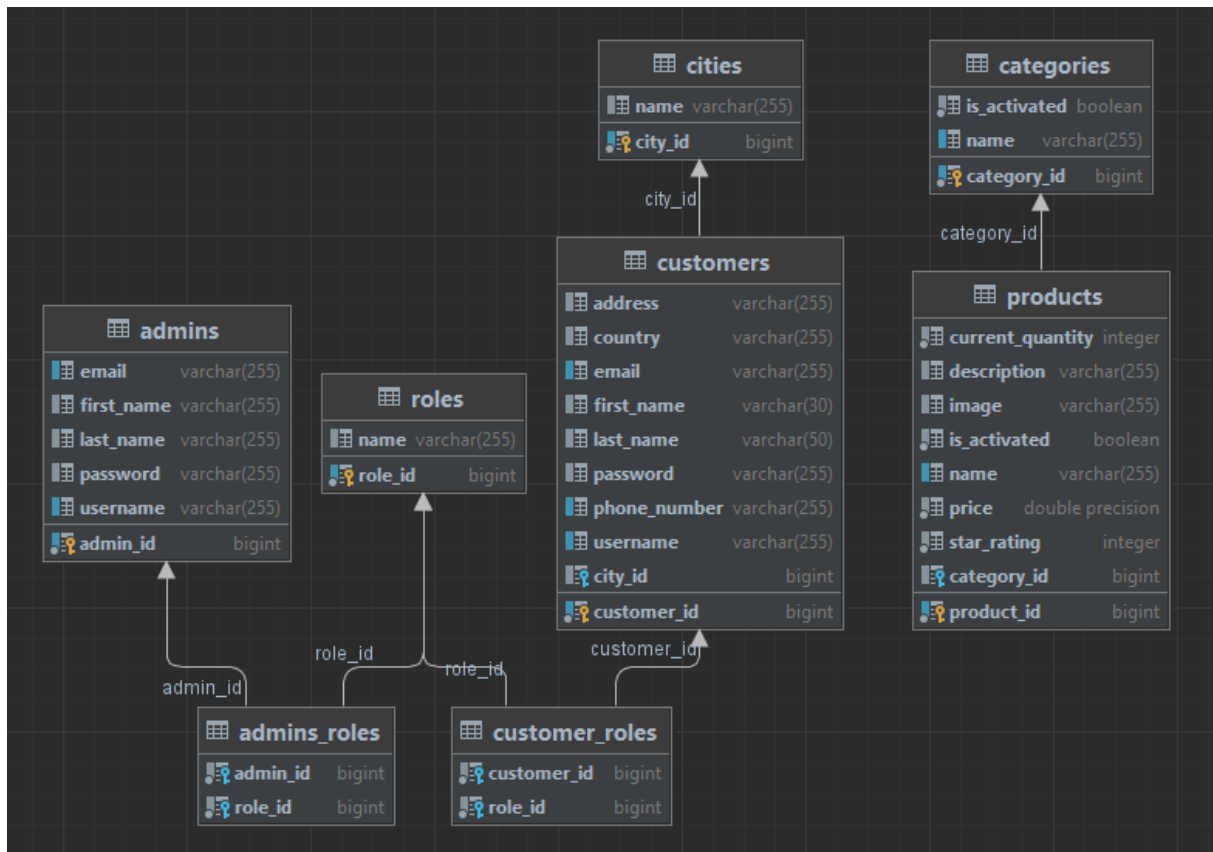


Abbildung 10: Data Model

Hier eine Übersicht über die Tabellen und ihre Beziehungen:

**Admins:** Diese Tabelle repräsentiert die Administratoren der Anwendung. Sie enthält Spalten wie: admin\_id, email, first\_name, last\_name, password und username.

**Categories:** Diese Tabelle enthält Kategorien von Produkten. Sie hat eine Beziehung zur Tabelle products über den Fremdschlüssel category\_id.

**Cities:** Diese Tabelle enthält Städtenamen und wird für die Kundenadressen verwendet.

**Customers:** Diese Tabelle repräsentiert Kunden der Anwendung. Sie enthält Informationen wie customer\_id, address, country, email, first\_name, last\_name, password, phone\_number, username und city\_id. Die Spalte city\_id ist ein Fremdschlüssel, der auf die Tabelle cities verweist.

**Products:** Diese Tabelle repräsentiert Produkte in der Anwendung. Sie enthält Spalten wie product\_id, current\_quantity, description, image, is\_activated, name, price, star\_rating und category\_id. Der Fremdschlüssel category\_id verweist auf die Tabelle categories, um die Kategorie jedes Produkts anzugeben.

**Roles:** Diese Tabelle enthält verschiedene Rollen, die den Administratoren und Kunden zugeordnet werden können.

**Admins\_roles und customer\_roles:** Diese Tabellen stellen die Many-to-Many-Beziehung zwischen Administratoren bzw. Kunden und Rollen dar. Sie enthalten Fremdschlüsselverknüpfungen auf die entsprechenden Tabellen.

Wie später dann ersichtlich ist, sind in der Applikation nicht alle Tabellen und Spalten in den DTOs implementiert worden. Aufgrund dessen, dass wir ein fiktives Unternehmen mit einem unproduktiven Online Shop sind. Somit wurde beispielsweise beim Customer-DTO viel Attribute ausgelassen, da wir auch noch keinen Lieferservice haben und somit die Adresse des Kunden nicht benötigen.

Aufgrund dessen wurde die Tabelle «cities» in der Anwendung nicht gebraucht und nur für den Zweck in der Datenbank implementiert, falls der Online-Shop produktiv geht.

### 9.3.5. Data Transfer Object (DTO)

Ein DTO (Data Transfer Object) wird verwendet, um Daten zwischen Frontend und Backend zu übertragen. Im unten aufgeführten Code-Ausschnitt wurde als Beispiel ein AdminDto dargestellt.

```
/**
 * This class is a data transfer object that is used in Spring-based
 * applications
 * to transfer data between the frontend and the backend
 */
@Data
@NoArgsConstructor
@AllArgsConstructor
public class AdminDto {

    private Long id;
    @Size(min = 1, max = 30, message = "Invalid first name! (1-30
characters")
    private String firstName;
    @Size(min = 1, max = 50, message = "Invalid last name! (1-50
characters")
    private String lastName;
    @Size(min = 1, max = 50, message = "Invalid username! (1-50
characters")
    private String username;
    private String email;
    @Size(min = 4, max = 30, message = "Invalid password! (8-30
characters")
    private String password;
}
```

## 9.4. Frontend Vue.js

### 9.4.1. Vue.js

In der heutigen Zeit wird eine starke Betonung auf interaktive und reaktive Benutzeroberflächen gelegt, die nahtlos mit dem Backend interagieren können. Traditionelle Ansätze zur Webentwicklung waren oft

durch komplexe JavaScript-Codebasen, unübersichtlichen Zustandsmanagement und eine eingeschränkte Wiederverwendbarkeit von Komponenten gekennzeichnet. Hier kommt Vue.js ins Spiel.

Vue.js ist ein fortschrittliches JavaScript-Framework für die Entwicklung von modernen, benutzerfreundlichen Webanwendungen. Es zeichnet sich durch seine Einfachheit, Flexibilität und Leistungsfähigkeit aus. Vue.js ermöglicht es Entwicklern, interaktive UI-Komponenten zu erstellen und diese nahtlos in eine Anwendung zu integrieren. Das Framework ist schlank und lässt sich leicht in bestehende Projekte integrieren.

Schlüsseleigenschaften:

- **Reaktive Datenbindung:** Eine der Hauptstärken von Vue.js liegt in seiner reaktiven Datenbindung. Änderungen im Zustand der Anwendung werden automatisch erkannt und die entsprechenden Teile der Benutzeroberfläche werden dynamisch aktualisiert. Dadurch entfällt die Notwendigkeit, manuell DOM-Manipulationen durchzuführen und vereinfacht die Handhabung von Datenflüssen erheblich.
- **Leichtgewichtig und skalierbar:** Vue.js wurde bewusst als schlankes Framework konzipiert, das sich leicht in Projekte integrieren lässt und eine hohe Leistung bietet. Es ist nicht überladen mit Funktionen, die möglicherweise nicht benötigt werden, und ermöglicht es Entwicklern, das Framework nach Bedarf zu erweitern.

#### **9.4.1.1. Projektstruktur**

In diesem Kapitel werden die verschiedenen Verzeichnisse und Dateien unserer Projektstruktur vorgestellt.

Die Projektstruktur basiert auf dem Hauptordner «src», der den Quellcode der Vue.js-Anwendung enthält. Hier sind die wichtigsten Verzeichnisse und Dateien:

- **Boot**
  - Axios.js
    - Definiert API-URLs für den Zugriff auf das Backend.
- **Layouts**
  - MainLayout.vue
    - Umfasst Webseitenkomponente, welche auf jeder Seite angezeigt werden.
- **Pages**
  - Auth
    - Beinhaltet alle Seiten, welche mit der Authentifizierung von Nutzern zu tun haben.
  - Category
    - Beinhaltet alle Seiten, welche mit Kategorien zu tun haben.
  - Checkout
    - Beinhaltet alle Seiten, welche mit dem Bestellprozess zu tun haben.
  - Products
    - Beinhaltet alle Seiten, welche mit Produkten zu tun haben.

- Router
  - Routes.ts
    - Definiert die URL für die verschiedenen Seiten.
- Stores
  - settingsStore.ts
    - Verwaltet Einstellungen wie z.B Dark Mode oder angemeldeter Nutzer.
  - shoppingCart.ts
    - Verwaltet den Warenkorb.

Ausserhalb des «src» Ordner befindet sich grösstenteils Konfigurationsdateien. Davon sind die Folgenden am relevantesten:

- Package.json
  - Beinhaltet allgemeine Informationen zu dem gesamten Projekt. Die wichtigsten davon sind die Abhängigkeiten, Skriptbefehle und die Version.
- Quasar.config.js
  - Beinhaltet Informationen von Quasar. Darunter sind Extras wie Icons und Fonts, und Plugins wie die Funktion für Notifikationen.

#### **9.4.1.2. Quasar**

Quasar ist eine leistungsstarke Erweiterung für Vue.js, die Entwicklern eine umfangreiche Sammlung von UI-Komponenten, Stilen und Funktionen bietet. Es wurde entwickelt, um die Entwicklung von Webanwendungen zu beschleunigen und gleichzeitig eine hohe Benutzerfreundlichkeit und Konsistenz zu gewährleisten.

Schlüsseleigenschaften:

- Umfangreiche UI-Komponenten: Quasar bietet eine umfangreiche Sammlung von vorkonfigurierten UI-Komponenten, die von Buttons und Formularelementen bis hin zu komplexen Datenvisualisierungen reichen. Diese Komponenten sind so gestaltet, dass sie sich nahtlos in Vue.js-Projekte integrieren lassen und eine konsistente Benutzererfahrung gewährleisten.
- Responsives Design: Quasar ermöglicht es Entwicklern, responsive Webanwendungen zu erstellen, die auf verschiedenen Geräten und Bildschirmgrößen optimal funktionieren. Die Erweiterung bietet eine Vielzahl von Grid-Systemen, Flexbox-Layouts und Medienabfragen, die es Entwicklern erleichtern, ihre Anwendungen an verschiedene Bildschirmgrößen anzupassen.

#### **9.4.1.3. Pinia**

Bei der Entwicklung von komplexen Webanwendungen mit Vue.js kann die Verwaltung des Anwendungs- und UI-Zustands zu einer Herausforderung werden. Traditionelle Ansätze wie die Verwendung von lokalen Daten oder globalen Ereignisbussen können zu Code-Komplexität und Problemen bei der Skalierbarkeit führen. Hier kommt Pinia ins Spiel.



Pinia ist eine leistungsstarke Erweiterung für Vue.js, die eine effektive Zustandsverwaltung in Webanwendungen ermöglicht. Es wurde speziell entwickelt, um die Komplexität der Zustandsverwaltung zu reduzieren und eine klare Struktur für die Handhabung von Datenflüssen in Vue.js-Projekten bereitzustellen. Pinia wurde speziell für die Verwendung mit Vue.js entwickelt

Schlüsseleigenschaften:

- Zentralisierter Zustand: Mit Pinia können Entwickler den Anwendungs- und UI-Zustand zentralisieren und in einem einzigen Speicher verwalten. Dies ermöglicht eine bessere Organisation des Codes und eine einfachere Verwaltung des Zustands in komplexen Anwendungen. Der zentrale Zustand kann über verschiedene Komponenten hinweg geteilt und aktualisiert werden.
- Strukturierte Zustandsverwaltung: Pinia fördert die Verwendung von Vuex-ähnlichen Konzepten wie Modulen, Aktionen, Mutationen und Getters, um den Zustand zu verwalten. Dies ermöglicht eine klare Trennung von Logik und Zustandsänderungen, was die Wartbarkeit und Lesbarkeit des Codes verbessert.
- Reactive: Pinia nutzt die reaktive Natur von Vue.js, um automatische Aktualisierungen des Zustands und der Benutzeroberfläche zu ermöglichen. Wenn sich der Zustand ändert, werden alle abhängigen Komponenten automatisch aktualisiert, ohne dass manuelle DOM-Manipulationen erforderlich sind.

Einsatz im Projekt

In unserem Projekt gibt es zwei verschiedene Pinia Stores, welche eingesetzt werden. Hier betrachten wir den «settingsStore.ts» als Beispiel. Dieser ist für die Verwaltung von allgemeinen Informationen zuständig.

```
export const useSettingsStore = defineStore('settings', {
  state: () => {
    const userPrefDarkMode = ref<boolean>(true);
    const currentUser = ref<number>(0);
    const isAdmin = ref<boolean>(false);
    const $q = useQuasar();

    if (localStorage.getItem('userPrefDarkMode')) {
      userPrefDarkMode.value = JSON.parse(
        <string>localStorage.getItem('userPrefDarkMode')
      );
    }

    watch(
      userPrefDarkMode,
      cb: (darkVal: boolean) => {
        localStorage.setItem('userPrefDarkMode', JSON.stringify(darkVal));
        $q.dark.set(darkVal);
      },
      {
        options: { deep: true }
      }
    );

    return { userPrefDarkMode, currentUser, isAdmin };
  }
});
```

Abbildung 11: Pinia Store Code Ausschnitt

In «Abbildung 11: Pinia Store Code Ausschnitt» können wir sehen das 3 verschiedene Werte verwaltet werden. die Funktionsrelevanten in diesem Beispiel sind der Nutzer und der Adminstatus. Diese Informationen werden dann auf verschiedenen Seiten benutzt, um das Verhalten der Seite zu bestimmen.

```

<q-btn v-if="settingsStore.isAdmin" :to="editLink">
  Edit
</q-btn>
<q-btn v-if="settingsStore.isAdmin" @click="deleteProduct">
  Delete
</q-btn>

```

Abbildung 12: Pinia Store Variablen Verwendung

In «Abbildung 12: Pinia Store Variablen Verwendung» können wir sehen, dass die gespeicherte Variable genutzt wird, um Admins den Zugriff auf die Bearbeitungs- und Löschfunktion von Produkten zu bieten, während diese Funktionen, normalen Nutzern nicht angezeigt werden.

#### 9.4.1.4. Axios

Moderne Webanwendungen erfordern oft die Kommunikation mit APIs, um Daten abzurufen oder zu senden. Die Verwaltung von HTTP-Anfragen kann jedoch eine komplexe Aufgabe sein, insbesondere wenn es um Fehlerbehandlung, Authentifizierung und das Verarbeiten von asynchronen Daten geht. Hier kommt Axios ins Spiel.

Axios ist eine leistungsstarke JavaScript-Bibliothek für das Ausführen von HTTP-Anfragen in Webanwendungen. Es wurde speziell für die Verwendung mit Vue.js entwickelt und bietet eine elegante API und eine Vielzahl von Funktionen für die Verarbeitung von HTTP-Anfragen. Axios unterstützt verschiedene Plattformen und bietet eine einheitliche Schnittstelle für die Kommunikation mit APIs.

Einsatz im Projekt:

In unserem Projekt werden zwei URLs definiert, welche als Grundlage für alle http Anfragen dienen. Durch diese Definition können diese überall im Projekt wiederverwendet werden.

```

5+ usages  ⚙ nicolas.dreier@students.ffhs.ch <nicolas.dreier@students.ffhs.ch>
const customerApi : AxiosInstance = axios.create({ baseURL: 'http://localhost:8020/shop' })
6+ usages  ⚙ nicolas.dreier@students.ffhs.ch <nicolas.dreier@students.ffhs.ch>
const adminApi : AxiosInstance = axios.create({ baseURL: 'http://localhost:8019/admin' })

```

Abbildung 13: Axios URL Definition

Ein Beispiel für den Einsatz sehen wir in «Abbildung 14: Axios Einsatz» auf der Seite «productOverview.vue». In diesem Beispiel werden alle Produkte aus dem Backend geholt und anschliessend dem Nutzer dargestellt. Falls bei der http Anfrage ein Fehler auftritt, wird dies dem Nutzer anhand der «Notify» Funktion von Quasar mitgeteilt.

```
customerApi.get( url: 'products')
  .then((response) => {
    apiData.value = response.data
  })
  .catch(() => {
    $q.notify( opts: {
      color: 'negative',
      position: 'top',
      message: 'Loading failed',
      icon: 'report_problem'
    })
  })
})
```

Abbildung 14: Axios Einsatz

## 9.5. Bestellprozess

### 9.5.1. Backend

Für die Bestellabwicklung wird der zuverlässige Zahlungsanbieter Stripe verwendet. Stripe ist eine weit verbreitete und beliebte Zahlungsplattform, die es Unternehmen ermöglicht, Online-Zahlungen sicher und einfach abzuwickeln. Mit Stripe kann man Zahlungen von Kunden akzeptieren, Abonnements verwalten und Zahlungsabläufe automatisieren.

Um sicherzustellen, dass die Integration mit Stripe reibungslos funktioniert, bietet Stripe die Test API an. Die Test API ermöglicht es Entwicklern, Stripe in einer Sandbox-Umgebung zu testen, ohne tatsächliche Zahlungen oder Transaktionen durchführen zu müssen. Dadurch kann man die Zahlungsabläufe, Funktionen und Fehlerbehandlung testen, bevor man sie in der Produktionsumgebung einsetzt.

Die Test API von Stripe stellt Testdaten und Testkarten zur Verfügung, mit denen man verschiedene Szenarien wie erfolgreiche Zahlungen, abgelehnte Zahlungen oder Karten mit Ablaufdaten simulieren kann. Man kann die Testintegration sowohl in der lokalen Entwicklungsumgebung als auch auf einem speziellen Staging-Server durchführen, um eine realistische Testumgebung zu schaffen.

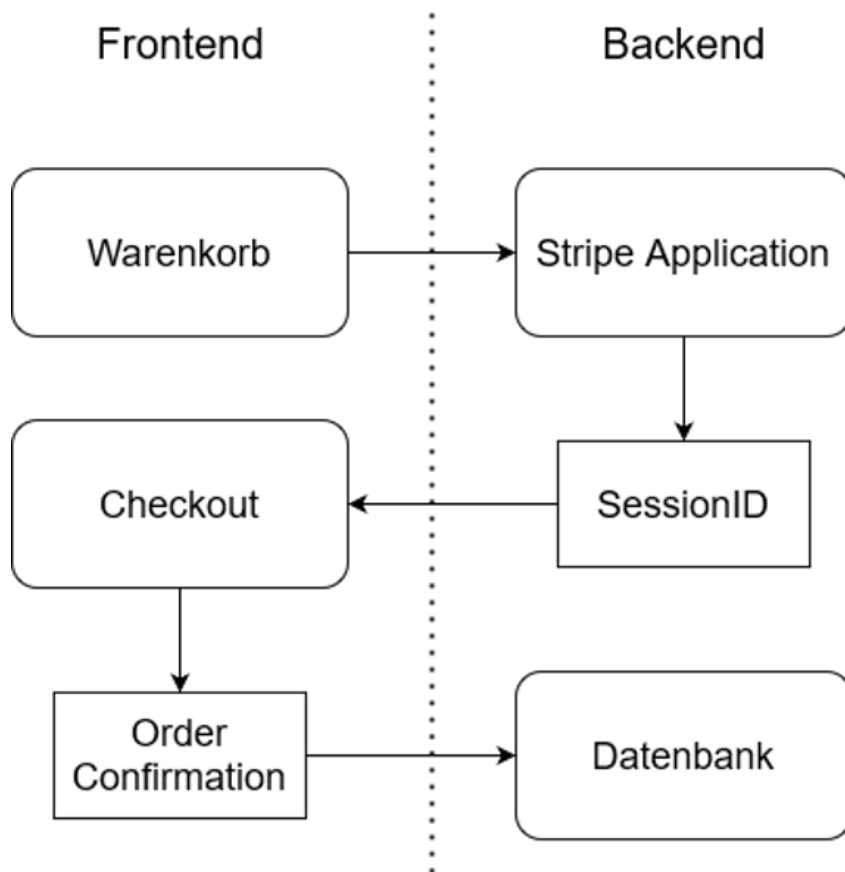


Abbildung 15: Stripe Ablauf Backend

Das Backend ist verantwortlich für die Verarbeitung eines POST-Requests und die Kommunikation mit der Stripe API. Beim Erhalt des Requests wird eine Anfrage an die Stripe API gesendet, um eine Session ID für den Zahlungsvorgang zu generieren. Diese Session ID wird anschließend an das Frontend zurückgegeben, um den Zahlungsprozess abzuschließen.

Sobald das Frontend die Session ID erhalten hat, kann es diese verwenden, um die Zahlungsinformationen an Stripe zu übermitteln und die Zahlung durchzuführen. Stripe führt dann die erforderlichen Zahlungsschritte aus und gibt eine entsprechende Antwort zurück.

Bei erfolgreicher Zahlung wird das Backend über eine entsprechende Benachrichtigung vom Frontend informiert. An diesem Punkt kann das Backend die erforderlichen Informationen der Bestellung aus dem POST-Request entnehmen und diese in die Datenbank speichern. Dadurch wird die Bestellung dauerhaft festgehalten und für weitere Verarbeitungsschritte verfügbar gemacht.

```

<dependency>
  <groupId>com.stripe</groupId>
  <artifactId>stripe-java</artifactId>
  <version>20.77.0</version>
</dependency>

<dependency>
  <groupId>com.google.code.gson</groupId>
  <artifactId>gson</artifactId>
</dependency>

```

Abbildung 16: Stripe MVN Dependance

Für die Einbindung von Stripe in das Projekt steht eine Stripe-Integration im MVN Repository zur Verfügung. Um diese Integration zu nutzen, ist es jedoch erforderlich, die GSON-Abhängigkeit hinzuzufügen. Die Stripe-Abhängigkeit ermöglicht es, JSON-Daten zu verarbeiten und verschiedene Funktionen bereitzustellen, einschließlich der Session-Verwaltung.

## Developers

[Overview](#) [API keys](#) [Webhooks](#) [Events](#) [Logs](#) [Apps](#)

### Your integration

4h 12h 24h **1w**

#### API requests

Successful	Failed
180	5



#### API error distribution

GET	POST	DELETE
0	5	0

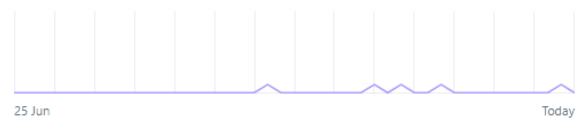


Abbildung 17: Stripe Sandbox API

In der Stripe-Testumgebung steht eine umfangreiche API zur Verfügung, die es ermöglicht, Testanfragen durchzuführen. Um diese API nutzen zu können, werden sowohl ein öffentlicher (Public) als auch ein privater (Private) API-Schlüssel bereitgestellt. Um Zugang zur Testumgebung und den entsprechenden API-Schlüsseln zu erhalten, ist die Erstellung eines kostenlosen Accounts erforderlich.

### 9.5.1.1. JPA-Modellierung

Im Datenbankmodell sind zwei Entitäten mit JPA modelliert. Die erste Entität ist "Order", die eine vollständige Bestellung repräsentiert. Sie speichert Informationen wie Datum, Gesamtsumme, Session-ID, Bestellelemente und den Benutzer, der die Bestellung aufgegeben hat.

```
@Entity
@Table(name = "orders")
public class Order {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    2 usages
    @Column(name = "created_date")
    private Date createdAt;

    2 usages
    @Column(name = "total_price")
    private Double totalPrice;

    2 usages
    @Column(name = "session_id")
    private String sessionId;

    2 usages
    @OneToMany(mappedBy = "order", fetch = FetchType.LAZY)
    private List<OrderItem> orderItems;

    2 usages
    @ManyToOne()
    @JsonIgnore
    @JoinColumn(name = "customer_id")
    private Customer user;
```

Abbildung 18: Order Model

Die zweite Entität ist "OrderItem" und repräsentiert ein Bestellelement. Neben dem Produkt enthält es auch Informationen wie den Preis, die Anzahl und das Bestelldatum.

```
@Entity
@Table(name = "orderitems")
public class OrderItem {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    3 usages
    @Column(name = "quantity")
    private @NotNull int quantity;

    3 usages
    @Column(name = "price")
    private @NotNull double price;

    3 usages
    @Column(name = "created_date")
    private Date createdAt;

    3 usages
    @ManyToOne
    @JsonIgnore
    @JoinColumn(name = "order_id", referencedColumnName = "id")
    private Order order;

    3 usages
    @OneToOne(fetch = FetchType.EAGER, cascade = {CascadeType.PERSIST, CascadeType.REFRESH, CascadeType.MERGE})
    @OnDelete(action = OnDeleteAction.CASCADE)
    @JoinColumn(name = "product_id")
    private Product product;
```

Abbildung 19: OrderItem Model

Diese beiden Entitäten werden auf der Datenbankebene verwaltet, wobei die Stripe API alle Transaktionsdaten selbst speichert. Es ist jedoch sinnvoll, die Daten auch auf unserer Seite zu sammeln, um Analysen und Berichte zu ermöglichen.

Während der Kommunikation über REST werden spezielle Data Transfer Objects (DTOs) verwendet, anstelle der tatsächlichen Modelle. Diese DTOs dienen dazu, die Daten zwischen dem Backend und dem Frontend auszutauschen und die erforderlichen Informationen zu übermitteln, ohne unnötige Daten preiszugeben oder die Sicherheit zu gefährden. Die DTOs sind speziell auf die Anforderungen der API-Endpunkte zugeschnitten und stellen sicher, dass nur relevante Daten übertragen werden.





```

public class StripeResponse {
    3 usages
    private String sessionId;

    no usages  Naveen
    public String getSessionId() { return sessionId; }

    no usages  Naveen
    public void setSessionId(String sessionId) { this.sessionId = sessionId; }

    1 usage  Naveen
    public StripeResponse(String sessionId) { this.sessionId = sessionId; }

    no usages  Naveen
    public StripeResponse() {
    }
}

```

Abbildung 21: StripeResponse

Nachdem die Anfrage zur Erstellung der Session an Stripe gesendet wurde, erhält das Frontend eine Stripe-Response als Rückgabewert. Der OrderService ermöglicht es, die Kontrolle über die zurückgegebenen Informationen zu behalten und gezielt nur die erforderlichen Daten, wie die Session-ID, an das Frontend weiterzugeben. Dadurch wird die Sicherheit und Privatsphäre gewährleistet, da sensible Zahlungsinformationen nicht unmittelbar an das Frontend gelangen.

### 9.5.1.3. Controller

Der REST-Controller stellt den API End Point dar.

```

@PostMapping("/create-checkout-session")
public ResponseEntity<> checkoutList(@RequestBody List<CheckoutItemDto> checkoutItemDtoList) throws StripeException {
    try {
        Session session = orderService.createSession(checkoutItemDtoList);
        StripeResponse stripeResponse = new StripeResponse(session.getId());
        return new ResponseEntity<>(stripeResponse, HttpStatus.OK);
    } catch (Exception e) {
        return new ResponseEntity<>(e.toString(), HttpStatus.BAD_REQUEST);
    }
}

```

Abbildung 22: Order Controller

Der Controller ermöglicht einen POST-Request mit den Bestellelementen abzusetzen. Diese werden an die Funktion zur Erstellung der Zahlungssitzung (Session) weitergeleitet. Im Anschluss wird die Session-ID als Rückgabewert zurückgegeben.

Die Stripe-API übernimmt die Verarbeitung der Zahlungssitzung und entscheidet auf Grundlage der Informationen in der Session, ob die Zahlung erfolgreich war oder nicht. Je nach Ergebnis leitet Stripe den Zahlungsprozess entsprechend weiter und führt die erforderlichen Aktionen aus.

Es ist wichtig zu beachten, dass sämtliche Transaktionen und Vorgänge in der Stripe-API aufgezeichnet werden. Dies bedeutet, dass alle relevanten Informationen, wie Zahlungsdetails, Bestelldaten und Statusinformationen, in der Stripe-API gespeichert und protokolliert werden. Dies dient der Nachverfolgbarkeit, dem Reporting und ermöglicht es, detaillierte Analysen über die Zahlungsvorgänge durchzuführen.

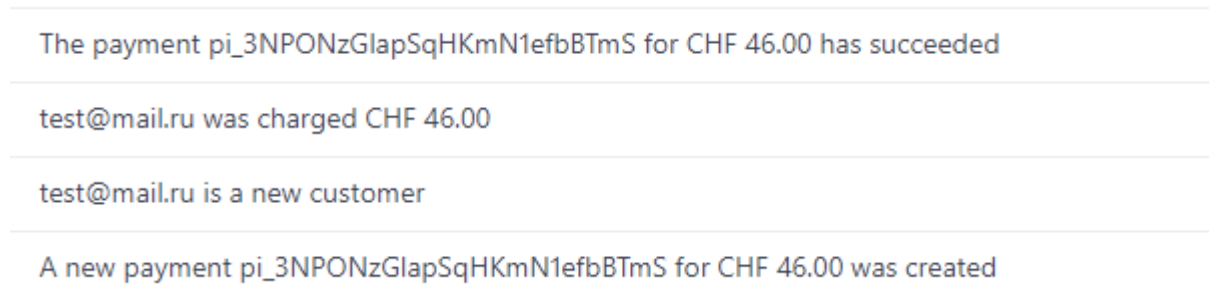


Abbildung 23: Stripe API Events

#### 9.5.1.4. Interne Datenbank

Für die interne Verarbeitung können Bestellungen auch in der Datenbank gespeichert werden. Hierbei wird zunächst eine "Order" erstellt, die die Session-ID und die UserID enthält. Anschließend wird die Liste der Bestellobjekte auf "OrderItems" abgebildet. Diese "OrderItems" werden mit der zuvor erstellten "Order" verknüpft. Dadurch können sowohl eine Bestellung als auch die dazugehörigen Produkte in der Datenbank gespeichert werden.

```
1 usage  Naveen
public Order placeOrder(String sessionId, CheckoutItemDto checkoutItemDto) {

    // create the order and save it
    Order newOrder = new Order();
    newOrder.setCreatedDate(new Date());
    newOrder.setSessionId(sessionId);
    newOrder.setUser(customerService.findCustomerById((long) checkoutItemDto.getUserId()));
    newOrder.setTotalPrice(checkoutItemDto.getPrice());
    return orderRepository.save(newOrder);
}
```

Abbildung 24: OrderService placeorder

Der Controller stellt einen entsprechenden Endpunkt bereit, über den diese Speicherung der Bestellungen erfolgen kann. Durch die Verwendung des Controllers wird die Kommunikation zwischen dem Frontend und dem Backend ermöglicht und die Daten können über REST-Schnittstellen übermittelt werden.

```

@PostMapping("/add")
public ResponseEntity<String> placeOrder(@RequestParam("sessionId") String sessionId, @RequestBody() List<CheckoutItemDto> checkoutItemList) {
    // place the order

    Order order = orderService.placeOrder(sessionId, checkoutItemList.get(0));

    for (CheckoutItemDto checkoutItemDto : checkoutItemList) {
        orderItemsService.addOrderedProducts(new OrderItem(order, productService.getProductById(checkoutItemDto.getProductId()), checkoutItemDto.getQuantity(), checkoutItemDto.getPrice()));
    }

    return new ResponseEntity<>("Order has been placed", HttpStatus.CREATED);
}

```

Abbildung 25: OrderService placeOrder

## 9.5.2. Frontend

### 9.5.2.1. Vue-Stripe

Vue-Stripe ist eine Erweiterung für Vue.js, die eine einfache und effiziente Integration von Stripe-Zahlungen in Vue.js-Anwendungen ermöglicht. Stripe ist ein beliebter Zahlungsdienst, der eine sichere und zuverlässige Abwicklung von Online-Zahlungen ermöglicht. Mit Vue-Stripe können Entwickler Stripe-Zahlungen nahtlos in ihre Vue.js-Anwendungen einbinden und Benutzern eine sichere und benutzerfreundliche Zahlungserfahrung bieten.

### 9.5.2.2. Einsatz im Projekt

Das Frontend übernimmt im Zahlungsprozess die Aufgabe, dem Backend den Warenkorb zu übermitteln und anschliessend den Nutzer auf die Zahlungswebseite weiterzuleiten. Auf der «Abbildung 26: Erstellen einer Bestellung im Frontend» ist zu erkennen, dass auf den Endpoint «http://localhost:8020/shop/order/create-checkout-session» mit dem Warenkorb als Request body eine Anfrage gestellt wird. Wenn diese erfolgreich ausgeführt wird, sendet das Backend eine generierte «sessionId», welche den Nutzer weiterleitet.

```

1 usage  nicolas.dreier@students.ffhs.ch <nicolas.dreier@students.ffhs.ch>
function createOrder() {
  customerApi.post( url: 'order/create-checkout-session', cartItems.value)
    .then(async (response) => {
      console.log(response.data)
      await router.push('/checkout/' + response.data.sessionId);
    })
    .catch(() => {
      $q.notify( opts: {
        color: 'negative',
        position: 'top',
        message: 'Register failed',
        icon: 'report_problem'
      })
    })
}
}

```

Abbildung 26: Erstellen einer Bestellung im Frontend

Die darauffolgende Seite repräsentiert den Zwischenschritt zur Zahlungsseite. Hier kann der Nutzer entweder zur Zahlung übergehen oder die Bestellung direkt abbuchen.

```

<stripe-checkout
  ref="checkoutRef"
  pk="pk_test_51MvTP6lapSqHKmNSs2LpkWKF7FWGw6PT0zvykLTxqB0h0tTgdU761QgoIJtmaC805eCnQsQtHjFRNG46KJr38Yl0086fUDvN2"
  :session-id="sessionId"
/>

```

Abbildung 27: Stripe Checkout Element

In «Abbildung 27: Stripe Checkout Element» ist das «stripe-checkout» Element von «vue-stripe» zu erkennen. Dieses erstellt anhand der generierten sessionId eine Zahlungsseite, auf der der User sicher seine Zahlung durchführen kann. In «Abbildung 28: Checkout Weiterleitung Funktion» ist zusätzlich zu erkennen, dass die sessionId über die URL an die Seite weiterkommuniziert wird.

```

methods: {
  submit() {
    // You will be redirected to Stripe's secure checkout page
    this.$refs.checkoutRef.redirectToCheckout();
  },
},
created() {
  this.sessionID = useRoute().params.sessionID; // Retrieve sessionId from route parameter
},

```

Abbildung 28: Checkout Weiterleitung Funktion

Nachdem der Nutzer die Zahlung absolviert hat, wird er auf die Seite «/payment/success/:sessionId» oder bei einer fehlgeschlagenen Zahlung auf die Seite «/payment/failed/:sessionId» weitergeleitet. Bei erfolgreicher Zahlung wird, wie in «Abbildung 29: Bestellung in Datenbank speichern» zu erkennen, eine Anfrage auf den Endpoint «order/add» gestellt, wodurch die Bestellung final in der Datenbank gespeichert wird.

```

customerApi.post( url: 'order/add', shoppingCart.objectCart, config: {
  params: {
    sessionId: sessionId
  }
} )
.then(async (response) => {
  console.log(response.data)
})
.catch(async () => {
  $q.notify( opts: {
    color: 'negative',
    position: 'top',
    message: 'Order failed',
    icon: 'report_problem'
  })
  await router.push('/payment/failed/' + sessionId);
})

```

Abbildung 29: Bestellung in Datenbank speichern

## 10. Deployment View

Die Deployment View visualisiert die Zusammenhänge der einzelnen Komponenten der Anwendung und wie sie miteinander interagieren.

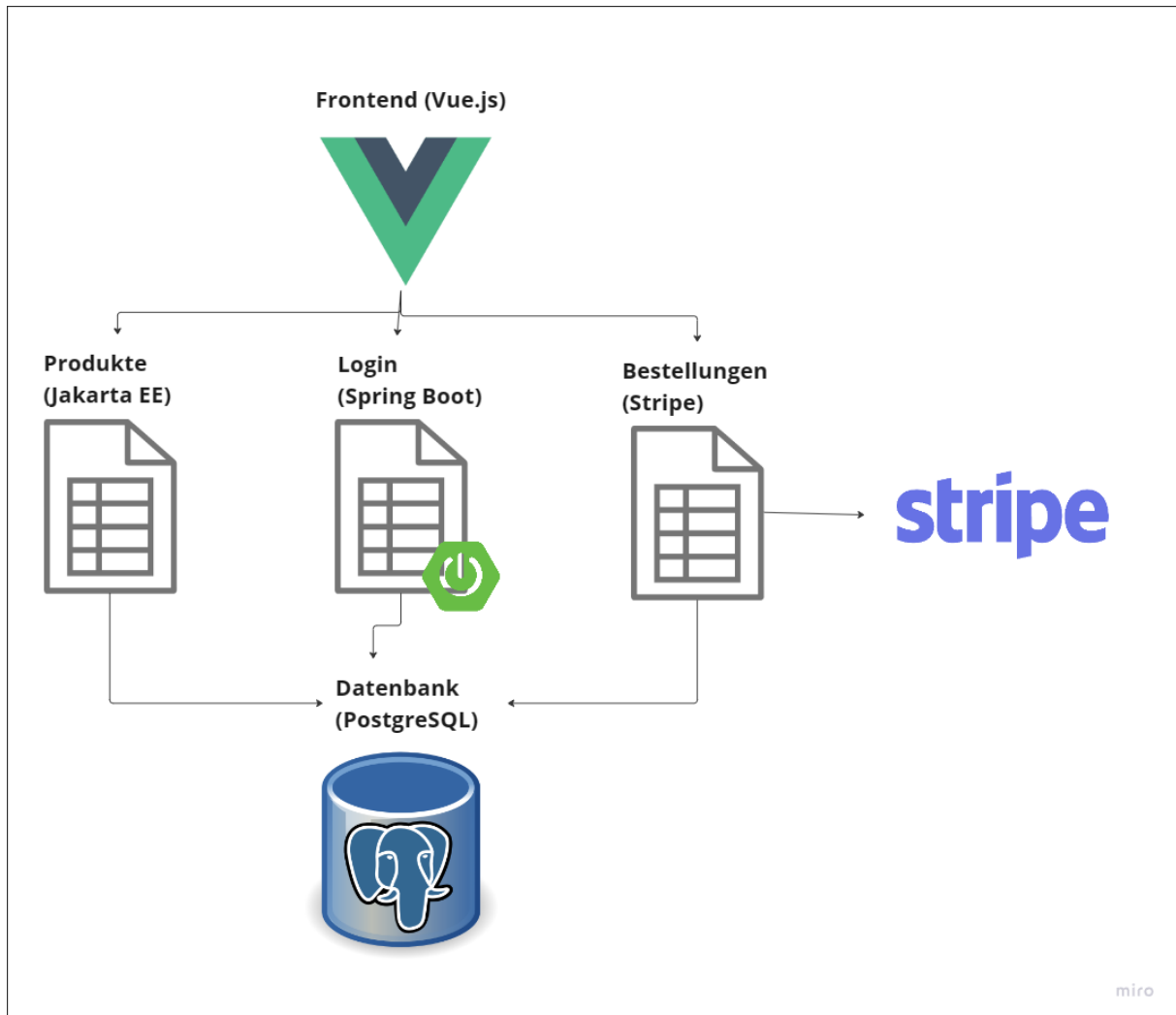


Abbildung 30: Deployment View

## 11. Vorarbeiten

Um das Projekt erfolgreich starten zu können, müssen zuvor einige Vorarbeiten erledigt werden.

Zuvor muss sichergestellt sein, dass die erforderliche Software heruntergeladen ist:

- **Docker:** ist eine freie Software zur Isolierung von Anwendungen mit Hilfe von Container Virtualisierung.
- **IDE (Integrierte Entwicklungsumgebung):** ist Software für die Anwendungsentwicklung, die gängige Entwicklertools in einer zentralen grafischen Oberfläche vereint

- **Node.js:** ist eine plattformübergreifende Open-Source-JavaScript-Laufzeitumgebung, die JavaScript-Code außerhalb eines Webbrowsers ausführen kann. Damit kann zum Beispiel ein Webserver betrieben werden.<sup>10</sup>
- **NPM:** ist ein Paketmanager für die JavaScript-Laufzeitumgebung Node.js. Mit dem Command: `npm install` kann NPM installiert werden.

## 12. Installationsanleitung

Bevor die Applikation gestartet werden kann, muss das Git Repository<sup>11</sup> in die IDE geklont werden. Danach muss als erstes das docker-compose-File gestartet werden, um ein Docker-Image der Postgres-Datenbank zu erstellen. Alle wichtigen Angaben wie User, Passwort und Port-Nummern sind dabei bereits angegeben und müssen nicht beachtet werden (siehe Abbildung 31).

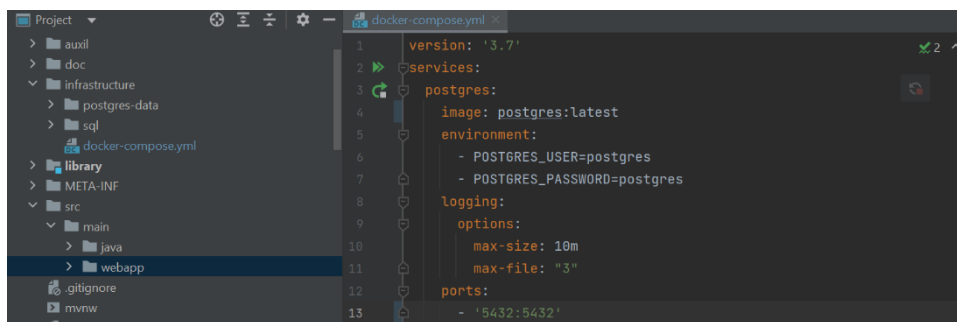


Abbildung 31: Docker-Compose-File starten

Nachdem das Docker-Compose-File gestartet wurde, kann man auf Docker prüfen, ob das Datenbank-Image erfolgreich angelegt wurde.

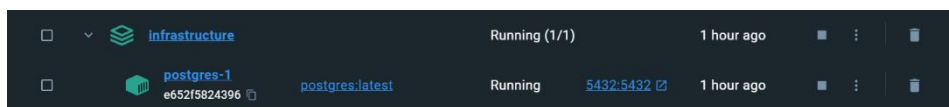


Abbildung 32: Docker Container

Danach kann man im Database-Toolfenster die Datenbank-Eigenschaften ansehen und ändern. Im Dialogfeld folgende Eigenschaften angeben, um auf die Datenbank zuzugreifen (Siehe Abbildung 33).

<sup>10</sup> [Download | Node.js \(nodejs.org\)](https://nodejs.org/) (zuletzt aufgerufen am 02.07.2023)

<sup>11</sup> [https://git.ffhs.ch/nina.aegerter/jea\\_semesterprojekt\\_gruppea.git](https://git.ffhs.ch/nina.aegerter/jea_semesterprojekt_gruppea.git) (zuletzt aufgerufen am 04.06.2022)

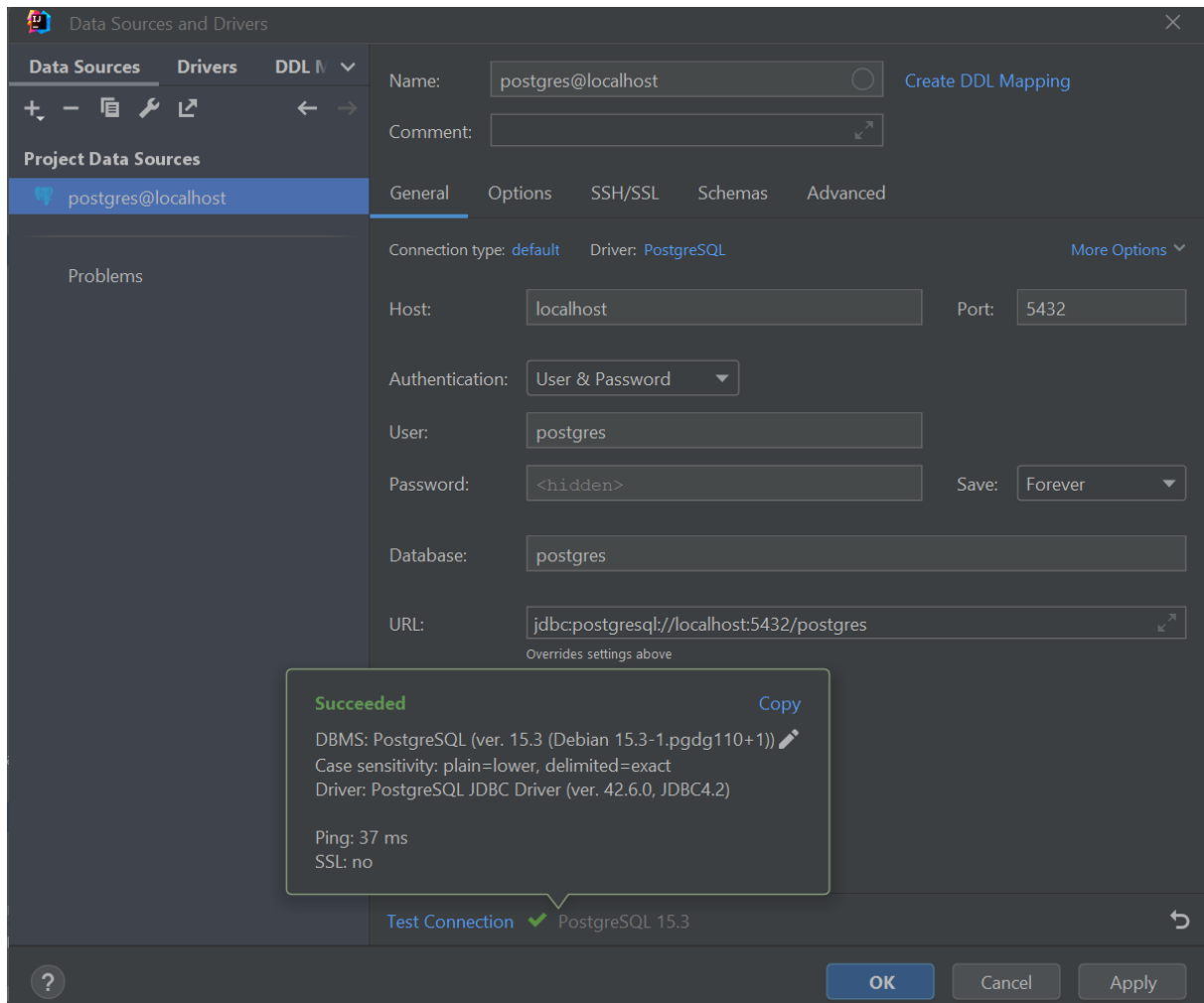


Abbildung 33: Konfiguration der Datenbank

Als Port Nummer wurde der Postgres-Standard Port 5432 verwendet. Zudem müssen noch die User- und Passwort-Angaben hinzugefügt werden.

Nun werden die Spring Applikationen gestartet. IntelliJ bietet ein Interface für die Verwaltung der Microservices an.

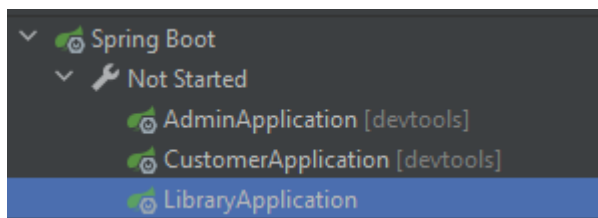


Abbildung 34: IntelliJ Services Verwaltung

Alternativ können auch die jeweiligen Main Klassen ausgeführt werden:

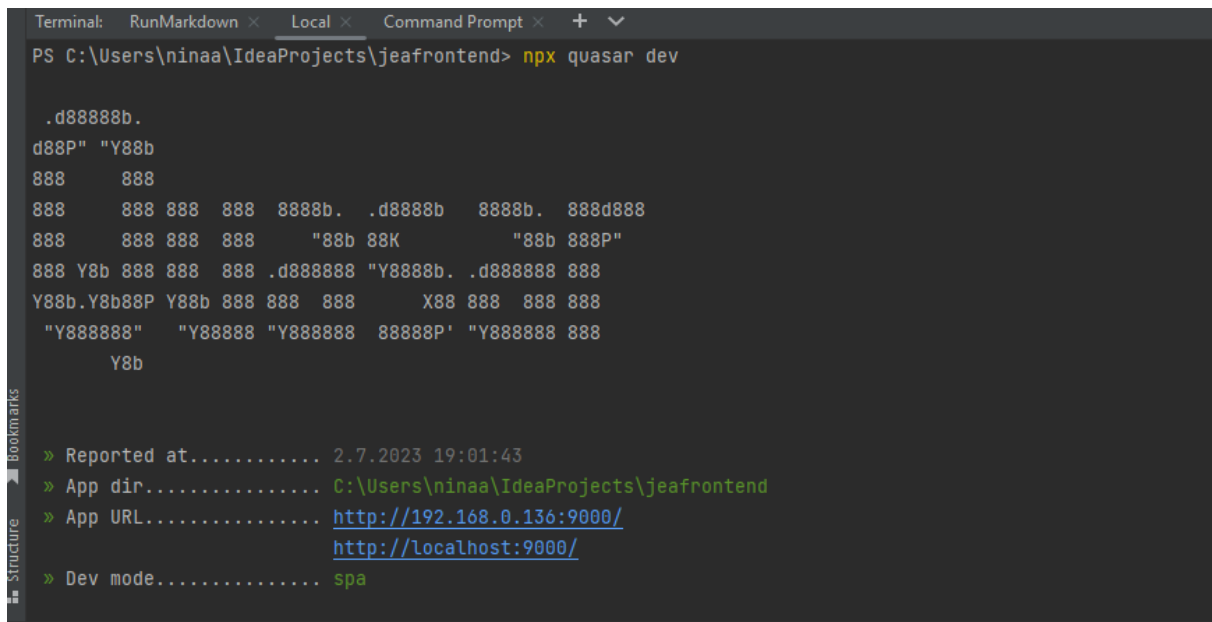
src/main/java/ch/ffhs/customer/customer/CustomerApplication

src/main/java/ch/ffhs/library/library/LibraryApplication

src/main/java/ch/ffhs/admin/admin/AdminApplication

In einem weiteren Schritt muss das Frontend-Repository<sup>12</sup> geklont werden und in ein weiteres IDE-Projekt implementiert werden.

Danach muss wie auf Abbildung ersichtlich, im Terminal das Kommando **(npx) quasar dev** eingegeben werden.



```
Terminal: RunMarkdown x Local x Command Prompt x + v
PS C:\Users\ninaa\IdeaProjects\jeafrontend> npx quasar dev

.d88888b.
d88P" "Y88b
888 888
888 888 888 888 8888b. .d8888b 8888b. 888d888
888 888 888 888 "88b 88K "88b 888P"
888 Y8b 888 888 888 .d888888 "Y8888b. .d888888 888
Y88b.Y8b88P Y88b 888 888 888 X88 888 888 888
"Y888888" "Y88888 "Y888888 88888P' "Y888888 888
Y8b

» Reported at..... 2.7.2023 19:01:43
» App dir..... C:\Users\ninaa\IdeaProjects\jeafrontend
» App URL..... http://192.168.0.136:9000/
http://localhost:9000/
» Dev mode..... spa
```

Abbildung 35: Quasar dev

Abbildung 36: Quasar

Danach laufen Customer- und Admin-Applikationen auf den Ports:

<http://localhost:8020/shop/>

<http://localhost:8019/admin/>

Durch den Aufruf der folgenden URLs im Browser kann überprüft werden ob die Services laufen:

<http://localhost:8020/shop/swagger>

<http://localhost:8019/admin/swagger>

Das Frontend ist unter <http://localhost:9000/> erreichbar.

## 13. Fazit

Bereits bei den ersten Coding Phasen haben wir festgestellt, dass es besser ist die Aufgabenstellung genau durchzulesen und sich dann Notizen und Gedanken zum Vorgehen zu machen und nicht gleich ohne Ziel einfach los «Coden».

Des Weiteren musste die Projekt-Struktur mehrmals überarbeitet werden, da mit grösseren Code-Abschnitten die eigentliche Funktion der Klasse unterging und so festgestellt wurde, dass es wieder an der Zeit für ein Refactoring ist und eine neue separate Klasse mehr Sinn ergibt.

---

<sup>12</sup> <https://git.ffhs.ch/nicolas.dreier/jeafrontend.git> (zuletzt aufgerufen am 02.07.2023)



Auch hätte man Zeit sparen können, indem die Kommentare immer aktuell zu halten und alles genau zu dokumentieren. Jedoch gab es somit auch den Vorteil, dass die Gedankengänge zum verfassten Code nochmals repetiert werden mussten und somit der Lerneffekt grösser war. Dies ermöglichte uns auch Fehler in der Logik zu finden, die wir vorher übersehen hätten.

Das Herzstück des Projekts, die Verknüpfung der beiden Projekte – Frontend und Backend - bereitete einiges an Aufwand. Wir mussten uns damit vertraut machen und viele Dinge auch durch Ausprobieren erlernen.

Trotz der Hürden und Schwierigkeiten sind wir aber mit dem Projekt zufrieden. Zum Ersten konnten wir es pünktlich und funktionsfähig abschliessen, und zum Zweiten konnten wir sehr viel von dem Projekt mitnehmen und lernen, besonders was das Planen und Verteilen von Aufgaben angeht. Beim nächsten Mal sollte der ganze Ablauf schon viel geschmeidiger vonstattengehen und die grössten Schwierigkeiten des Projektes vermieden werden können. Unter diesen Kriterien definieren wir es als ein erfolgreiches Projekt!

## 14. Verzeichnis

### 14.1. Literaturverzeichnis

*Abbildung Git Work-Flow* . (02. Juli 2023). Von [https://www.nicepng.com/png/detail/955-9558351\\_github-branching-workflow-diagram-git-flow.png](https://www.nicepng.com/png/detail/955-9558351_github-branching-workflow-diagram-git-flow.png) abgerufen

Bloomenthal, A. (02. Juli 2023). *Investopedia*. Von <https://www.investopedia.com/terms/e/ecommerce.asp> abgerufen

Broy, M., & Kuhrmann, M. (2020). Anforderungsanalyse und Anforderungsmanagement. In *Einführung in die Softwaretechnik*. Springer Verlag.

*bwl-lexikon.de*. (02. Juli 2023). Von <https://www.bwl-lexikon.de/wiki/stakeholderanalyse/> abgerufen

*Docker*. (02. Juli 2023). Von <https://www.docker.com/> abgerufen

*Informationstechnologie im KMU*. (02. Juli 2023). Von <https://www.it-im-kmu.com/lastenheft/> abgerufen

*IntelliJ*. (02. Juli 2023). Von JET BRAINS: <https://www.jetbrains.com/idea/> abgerufen

*Moodle*. (28. Juni 2023). Von <https://moodle.ffhs.ch/mod/choicegroup/view.php?id=4295523> abgerufen

*Node.js*. (02. Juli 2023). Von <https://nodejs.org/en/download> abgerufen

*PostgreSQL*. (02. Juli 2023). Von <https://www.postgresql.org/> abgerufen

### 14.2. Abbildungsverzeichnis

Abbildung 1: E-Commerce .....	6
Abbildung 2: Git Work-Flow .....	13
Abbildung 3: Spring Boot Architektur .....	15
Abbildung 4: Controller-Service-Repository Architecture .....	17
Abbildung 5: Spring Boot Dependencies.....	21
Abbildung 6: PostgreSQL Dependencies.....	21
Abbildung 7: Swagger Endpoints .....	22
Abbildung 8: application.properties (admin) .....	23
Abbildung 9: PostgreSQL-Container .....	27
Abbildung 10: Data Model .....	29
Abbildung 11: Pinia Store Code Ausschnitt.....	33
Abbildung 12: Pinia Store Variablen Verwendung .....	34
Abbildung 13: Axios URL Definition .....	34
Abbildung 14: Axios Einsatz .....	35
Abbildung 15: Stripe Ablauf Backend .....	36
Abbildung 16: Stripe MVN Dependence.....	37
Abbildung 17: Stripe Sandbox API .....	37
Abbildung 18: Order Model.....	38
Abbildung 19: OrderItem Model .....	39

Abbildung 20: OrderService createSession (secret key durchgestrichen) .....	40
Abbildung 21: StripeResponse .....	41
Abbildung 22: Order Controller .....	41
Abbildung 23: Stripe API Events .....	42
Abbildung 24: OrderService placeorder .....	42
Abbildung 25: OrderService placeOrder .....	43
Abbildung 26: Erstellen einer Bestellung im Frontend .....	43
Abbildung 27: Stripe Checkout Element.....	44
Abbildung 28: Checkout Weiterleitung Funktion .....	44
Abbildung 29: Bestellung in Datenbank speichern.....	44
Abbildung 30: Deployment View .....	45
Abbildung 31: Docker-Compose-File starten .....	46
Abbildung 32: Docker Container .....	46
Abbildung 33: Konfiguration der Datenbank.....	47
Abbildung 34: IntelliJ Services Verwaltung .....	47
Abbildung 35: Quasar dev .....	48

### 14.3. Tabellenverzeichnis

Tabelle 1: Stakeholder.....	7
Tabelle 2: Funktionale Anforderungen .....	8
Tabelle 3: Nicht-Funktionale Anforderungen .....	9
Tabelle 4: Rechtliche Anforderungen .....	11
Tabelle : Projekt-Module.....	16

### 14.4. Glossar

JPA	Jakarta Persistence API
ORM	Object-relational Mapping, Objektrelationale Abbildung
DTO	Data Transfer Object
IDE	Integrated Development Environment