



NESSER-E-COMMERCE

Onlineshop-Website

Assaad Naes und Oliver Wallisch

Inhaltsverzeichnis

Einleitung	2
Datenstruktur	3
Datenbank	4
Das Frontend	5
Sprachen und Technologien	5
Der Aufbau des Onlineshops	6
Struktur der Homepage	6
HTTP-Anfragen	9
Produktaufruf und -anzeige	11
Gestaltung mit CSS	17
Einloggen und Registrierung	19
Die Profileseite	21
Der Warenkorb	21
Bezahlungsmethode	22
Media Queries	26
Kleine Anzeigen	27
Das Backend	28
Framework	28
Aufbau	29
Sicherheit	29
Diagramme	37
Sequenzdiagramm	37
Struktogramm	38
PHP-Code	40

Einleitung

In der heutigen Zeit ist eine E-Commerce-Website für jedes Unternehmen unerlässlich. Das **Nesser-E-Commerce Team** hat sich daher zum Ziel gesetzt, eine einzigartige und benutzerfreundliche Plattform zu schaffen, die unseren Kunden ein einfaches und angenehmes Einkaufserlebnis bietet.

Das Projekt umfasst die Entwicklung einer E-Commerce-Website, die hochwertige PC-Komponenten verkauft. Dabei war es uns besonders wichtig, auf Benutzerfreundlichkeit und die Kompatibilität der Website mit verschiedenen Bildschirmgrößen zu achten. Diese beiden Aspekte waren für unseren Online-Shop sehr wichtig.

Das Projekt gliedert sich hauptsächlich in drei Bereiche:

Frontend: Die Benutzeroberfläche, die die Interaktion mit der Website ermöglicht. Das Frontend übernimmt die Kommunikation zwischen dem Benutzer und der RESTful API.

Backend: Eine RESTful API, die die Geschäftslogik und Datenverarbeitung übernimmt und als Schnittstelle zwischen Frontend und Datenbank dient. Das Backend kümmert sich zudem um Sicherheitsanforderungen wie Benutzer-Login und -Registrierung.

Datenbank: Die Speicherung und Verwaltung der Produkt- sowie Benutzerdaten. Sie sorgt für eine effiziente Datenabfrage und -bearbeitung, insbesondere bei Bestellungen und Produktinformationen.

Insgesamt soll die Website nicht nur eine einfache Benutzererfahrung bieten, sondern auch die nötige Sicherheit und Skalierbarkeit aufweisen, um mit dem Wachstum des Geschäfts Schritt zu halten. Das Projekt verfolgt einen modularen Ansatz, der eine spätere Erweiterung und Anpassung an neue Anforderungen ermöglicht. Darüber hinaus legen wir großen Wert auf die Wartbarkeit des Codes, um zukünftige Updates und Anpassungen zu erleichtern.

Datenstruktur

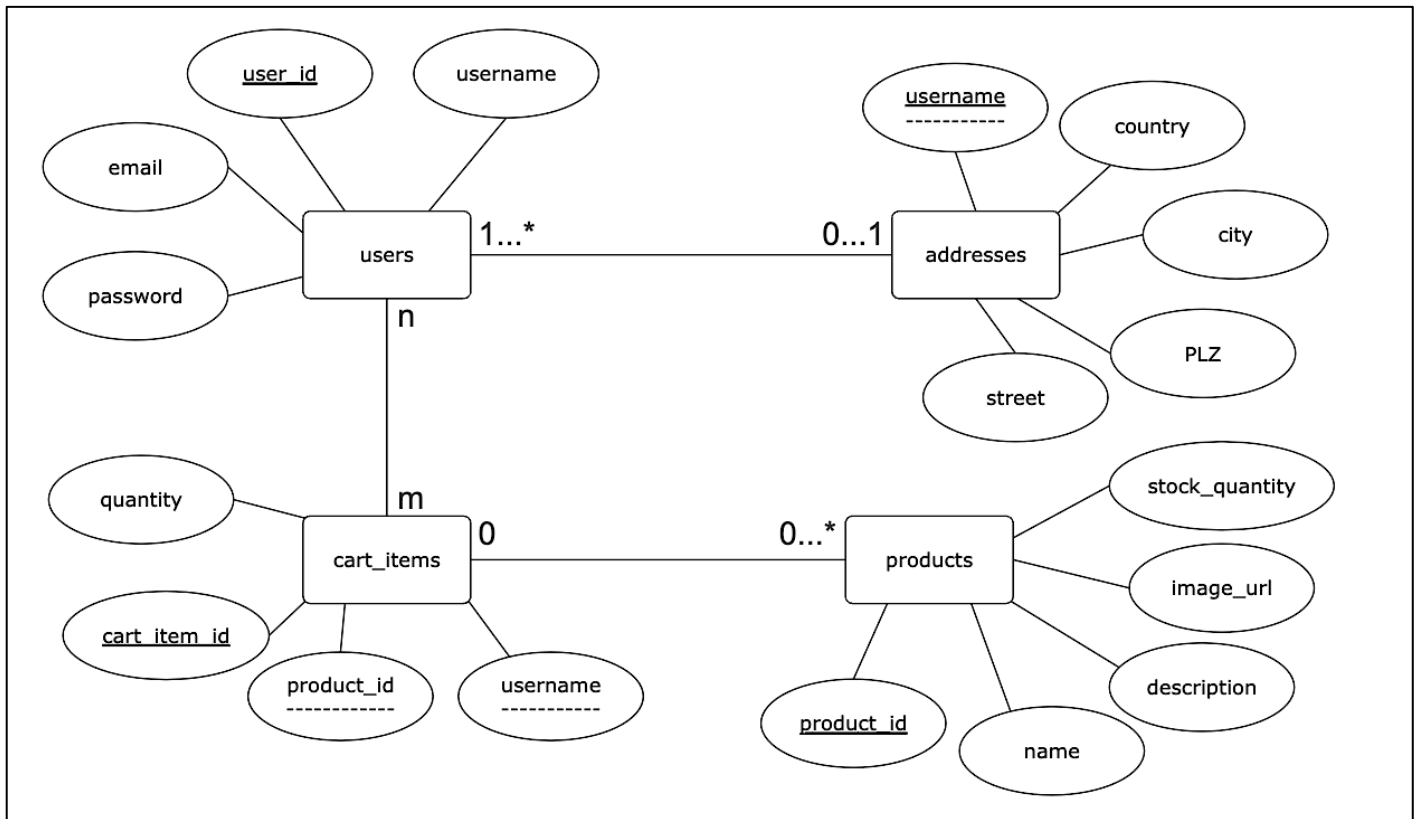


Diagramm 1 - UML-Diagramm

Im Onlineshop gibt es hauptsächlich vier Entitäten: **Produkte**, **Benutzer**, **Adressen** und **Warenkorb-Artikel**.

- **Produktentabelle:** Diese Tabelle enthält alle Produkte, die in unserem Onlineshop zum Verkauf angeboten werden. Jedes Produkt hat einen eindeutigen Identifier sowie Informationen wie Name, Beschreibung, Preis, Bild-Link und Lagerbestand.
- **Benutzertabelle:** In dieser Tabelle sind alle Benutzerdaten gespeichert, darunter der Benutzername, die E-Mail-Adresse und das verschlüsselte Passwort. Jeder Benutzer erhält einen eindeutigen Identifier (Benutzer-ID), der zur Zuordnung von Daten in anderen Tabellen dient.
- **Adressentabelle:** Diese Tabelle speichert die Adressen der Benutzer. Jeder Benutzer kann nur eine Adresse haben, aber eine Adresse kann mehreren Benutzern zugeordnet werden (z. B. wenn mehrere

Personen denselben Wohnort teilen). Die Adressen umfassen Felder wie Straße, Stadt, Postleitzahl, Land und Handynummer. Zusätzlich enthält die Tabelle den Fremdschlüssel **Benutzer_Id**, um eine Adresse einem bestimmten Benutzer zuzuweisen.

- **Warenkorb-Artikel:** Diese Tabelle enthält Informationen zu den Produkten, die sich im Warenkorb eines Benutzers befinden, sowie die Menge der jeweiligen Produkte. Die Zugehörigkeit der Warenkorb-Artikel zu einem Benutzer wird über die Benutzer-ID sichergestellt, die mit den entsprechenden Produktinformationen verknüpft wird.

Datenbank

Für die Entwicklung unseres Projekts haben wir uns für **PostgreSQL** als Datenbankmanagementsystem entschieden. Es ist ein leistungsstarkes relationale Datenbanksystem, das für seine Stabilität und Flexibilität bekannt ist. Es unterstützt eine Vielzahl von Features, die unsere Anforderungen an eine moderne Datenbank optimal erfüllen.

Indexes in einer Tabelle

Funktion

Ein Index in einer Datenbank dient dazu, die Geschwindigkeit von Datenbankabfragen zu verbessern. Er funktioniert ähnlich wie ein Inhaltsverzeichnis in einem Buch, das es ermöglicht, bestimmte Informationen schneller zu finden, ohne die gesamte Tabelle durchsuchen zu müssen. Ein Index wird für eine oder mehrere Spalten einer Tabelle erstellt und hilft dabei, Abfragen wie Suche, Sortierung und Filterung effizienter zu gestalten.

Beispiel

In unserer Tabelle **Warenkorb-Artikel** haben wir einen Index auf der Spalte `user_id` erstellt. Dadurch können wir die Artikel eines bestimmten Benutzers schneller finden und sortieren. Dies hat die

Abfragegeschwindigkeit und die allgemeine Effizienz des Systems erheblich verbessert.

Einsatz von Indexes

Indexes müssen mit Bedacht eingesetzt werden. Während sie die Geschwindigkeit von **SELECT-Anfragen** erheblich steigern, können sie **INSERT-, UPDATE- und DELETE-Anfragen** verlangsamen. Dies liegt daran, dass bei jeder Änderung der Tabelle (z. B. beim Hinzufügen, Aktualisieren oder Löschen von Datensätzen) der Index aktualisiert und neu organisiert werden muss, was zusätzliche Rechenressourcen erfordert.

Aus diesem Grund ist es wichtig, den Einsatz von Indexes sorgfältig zu planen. Es sollte genau abgewogen werden, ob ein Index für eine bestimmte Tabelle notwendig ist und einen klaren Mehrwert bietet, bevor er erstellt wird. Eine übermäßige oder unüberlegte Nutzung von Indexes kann die Performance der Datenbank negativ beeinflussen.

Das Frontend

Sprachen und Technologien

Die Webseite war nicht sehr komplex, und wir wollten den Code so klein wie möglich halten. Deshalb haben wir kein Framework benutzt.

Für die Gestaltung der Webseite haben wir **HTML** und **CSS** verwendet. Um Interaktionen und Funktionen hinzuzufügen, wurde **JavaScript** eingesetzt.

HTML (HyperText Markup Language) ist die Grundstruktur einer Webseite und definiert den Aufbau der Inhalte.

CSS (Cascading Style Sheets) wird verwendet, um das Design und das Aussehen einer Webseite zu gestalten, wie Farben, Positionen von die Elemente und Schriftarten.

JavaScript ist eine Programmiersprache, die Webseiten interaktiv und dynamisch macht.

Der Aufbau des Onlineshops

Der Onlineshop besteht aus insgesamt fünf Hauptseiten:

- **Homepage:** Die Startseite, auf der die Produkte präsentiert werden.
- **Login:** Eine Seite, auf der sich bestehende Benutzer anmelden können.
- **Register:** Eine Registrierungsseite für neue Benutzer, um ein Konto zu erstellen.
- **Einkaufskorb:** Eine Seite, auf der die im Warenkorb gespeicherten Artikel angezeigt werden, bevor die Bestellung abgeschlossen wird.
- **Adresse:** Eine Seite, auf der der Benutzer seine Adresse eintragen oder bearbeiten kann.

Struktur der Homepage

Die Homepage ist in verschiedene Teile und Strukturen aufgeteilt:

```
<main class="welcome-view">  
  <header> ...  
</header>  
  
  <section class="about-us"> ...  
</section>  
  
  <div class="images-slide-container"> ...  
</div>  
</main>
```

Das **Main**-Element enthält den **Header**, eine **Über-uns-Sektion** und ein **Animationselement**.

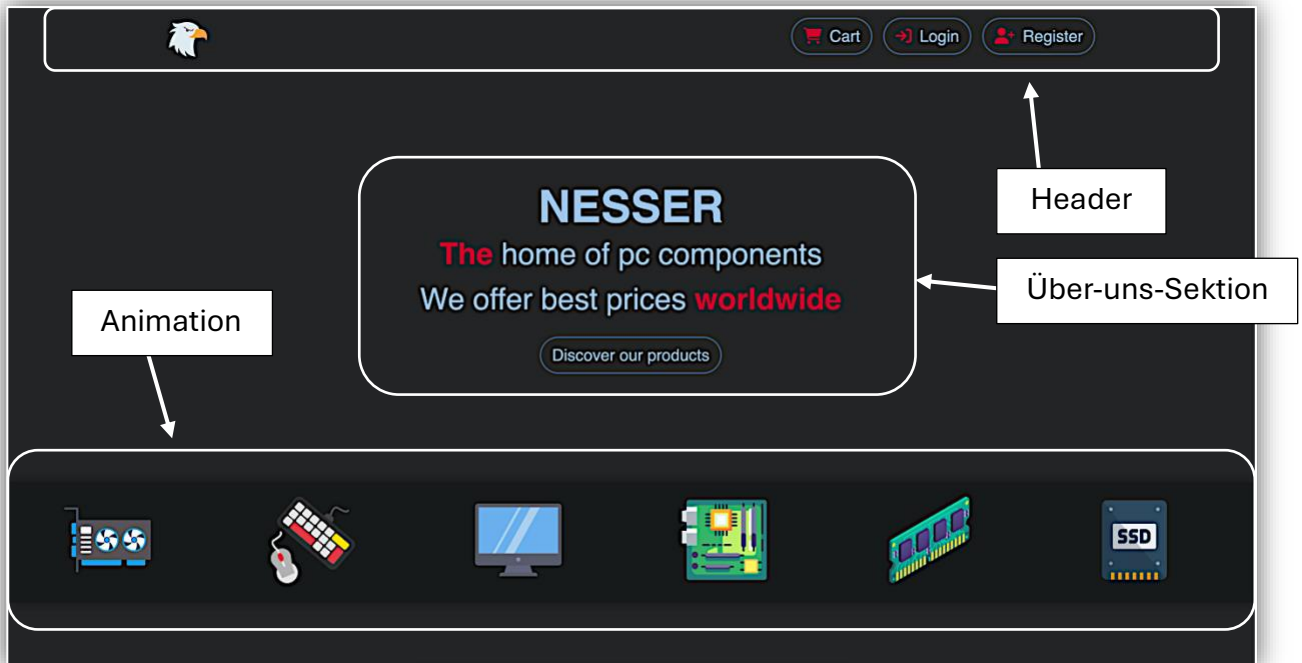
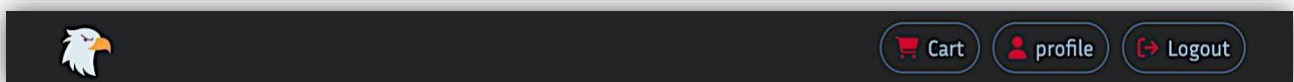


Bild 1 - Main

Dynamischer Header

Der Header ändert sich je nachdem, ob der Benutzer angemeldet ist oder nicht. Wenn der Benutzer angemeldet ist, sieht der Header wie folgt aus:



Technische Umsetzung des Headers

Der dynamische Header wird durch die Verwendung von zwei unterschiedlichen HTML-Dokumenten realisiert. Ein Dokument ist für nicht angemeldete Benutzer, während das andere speziell für angemeldete Benutzer erstellt wurde. Das Dokument für angemeldete Benutzer enthält zusätzliche Inhalte und Funktionen, die für eingeloggte Benutzer relevant sind.

Suchefeld und Produktansicht

Unter dem Hauptbereich (Main) folgen zwei weitere Elemente: das **Suchfeld** und die **Produktansicht**.

```
<div id="search"> ...
</div>

<section id="products-container">
</section>
```

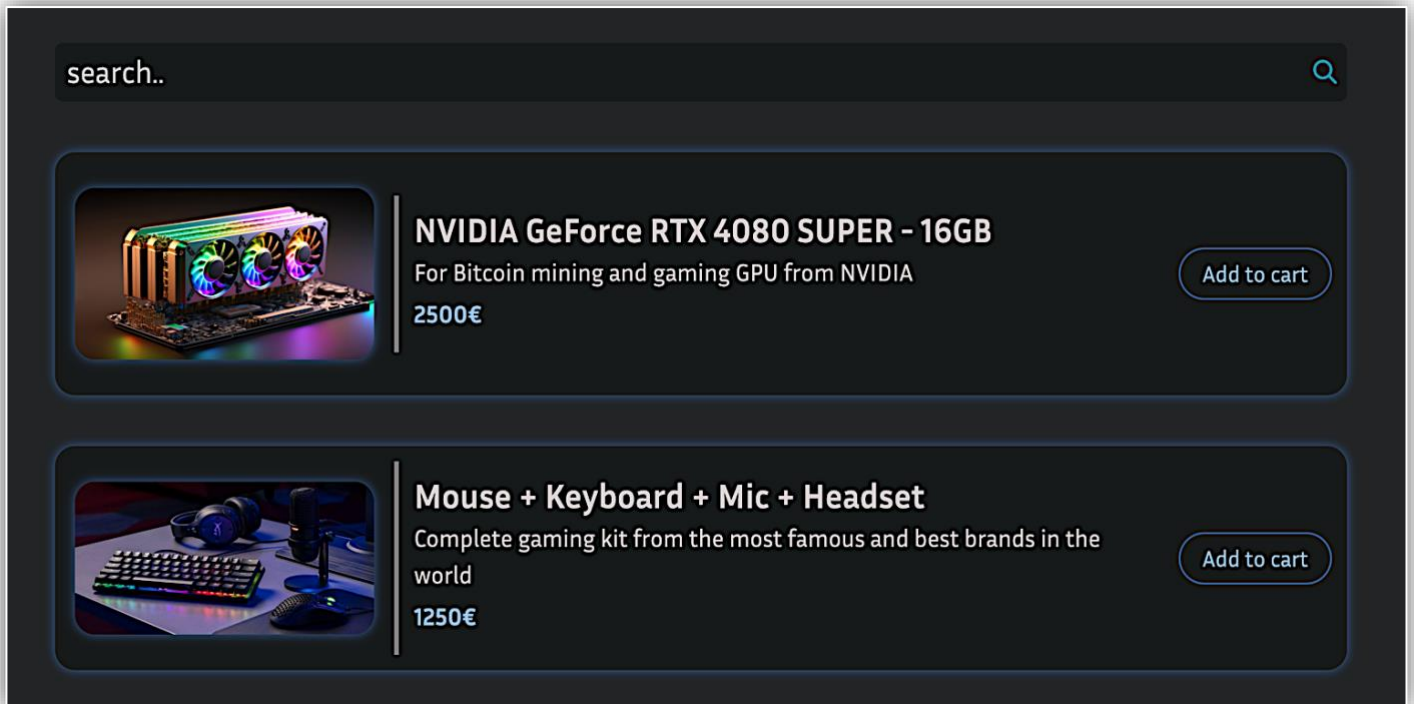



Bild 2 - Suchfeld und Produkte

Der Footer

Am Ende der Homepage befindet sich der Footer. Der Footer enthält nur einen Link mit dem Text "**Contact Us**", der auf unseren Instagram-Account verweist.

```
<footer>
  <a href="https://www.instagram.com/asaadnaes"
    class="fa-brands fa-instagram"
    target="_blank"> Contact us</a>
</footer>
```



Bild 3 - Der Footer

HTTP-Anfragen

Axios

Um HTTP-Anfragen mit JavaScript zu machen, gibt es verschiedene Varianten. In unserem Projekt haben wir **Axios** verwendet. Axios ist eine sehr bekannte JavaScript-Bibliothek, die es ermöglicht, einfach mit Backend-APIs zu kommunizieren.

Vorteile von Axios

1. **Einfache Handhabung:** Axios bietet eine einfache API, die das Senden und Empfangen von HTTP-Anfragen erleichtert.
2. **Automatische JSON-Transformation:** Axios wandelt automatisch JSON-Antworten des Servers in JavaScript-Objekte um, was den Umgang mit den Daten vereinfacht.

Anpassung von Axios-Konfiguration

Axios kann so konfiguriert werden, dass bestimmte Einstellungen für alle HTTP-Anfragen gelten.

In unserem Projekt wurde dazu eine JavaScript-Datei mit folgender Konfiguration erstellt:

```
const api = axios.create({
  baseURL: 'http://localhost:8080',
});
```

Hier wird eine Variable erstellt, die eine Axios-Instanz enthält. Diese Instanz verweist auf die Basis-URL des Backend-APIs (<http://localhost:8080>). Falls sich die URL des Backends ändert, muss die Basis-URL nur in dieser Konfigurationsdatei angepasst werden, anstatt in jeder einzelnen HTTP-Anfrage im Frontend.

Token-Handling und weitere Konfigurationen

Ein Beispiel von die weitere Konfigurationsmöglichkeiten, ist die Verwendung von **JWT-Tokens** (JSON Web Tokens), die bei bestimmten

Anfragen als Header gesendet werden. JWTs sind Token, die während des Login-Prozesses generiert werden. Sie müssen bei Anfragen für jedem gesicherten API-Endpunkt im Header mitgesendet werden, um die Authentizität des Benutzers zu bestätigen.

Bestimmte API-Endpunkte wie **Login**, **Register** und der Aufruf der **Produkte** müssen ohne Token erreichbar sein, da sonst kein Benutzer sich anmelden oder registrieren könnte. Diese Ausnahme kann in Axios durch eine gezielte Konfiguration umgesetzt werden.

Zunächst wird eine Liste mit regulären Ausdrücken (Regex-Mustern) erstellt, die die URLs der Endpunkte definieren, die keinen Token benötigen:

```
const noAuthPatterns = [/^\/login$/, /^\/register$/, /^\/products(\/.*)?$/];
```

Hier stehen:

- `^\/login$`: Für den Login-Endpunkt.
- `^\/register$`: Für den Registrierungs-Endpunkt.
- `^\/products(\/.*)?$`: Für den Produkte-Endpunkt und alle Unterrouen (z. B. Produktdetails).

Axios Interceptors für Anfragebearbeitung

Mit einem **Interceptor** wird jede HTTP-Anfrage, die über die erstellte Axios-Instanz (api) gesendet wird, abgefangen und vor dem Senden modifiziert:

```
api.interceptors.request.use(
```

Inerhalb der use-Methode kommen die konfigurationen für die Anfragen:

```
// Token wird aus dem lokalen Speicher abgerufen.  
const token = localStorage.getItem('token');  
// Testet, ob die Anfrage Authentifizierung erfordert.  
const requiresAuth = !noAuthPatterns.some((pattern) => pattern.test(config.url));
```

Authorization header hinzufügen, wenn für den Endpunkt Authentifizierung nötig ist:

```
// Wenn ein Token vorhanden ist und der Endpunkt Authentifizierung benötigt:  
if (token && requiresAuth) {  
  // Der Token wird als Authorization-Header hinzugefügt.  
  config.headers['Authorization'] = `Bearer ${token}`;  
}
```

Produktaufruf und -anzeige

Die Produkte werden über das REST API im Backend aus der Datenbank abgerufen. Der passende API-Endpunkt wird mit JS aufgerufen. Nachdem die Produkte abgerufen wurden, werden sie mit JS dynamisch in der Ansicht angezeigt.

Produktaufruf mit Axios

Die von uns erstellte Axios-Instanz wird verwendet, um die Anfragen zu senden:

```
export async function getAll() {  
  try {  
    const response = await api.get(`/products`);  
    return response.data;  
  } catch (error) {  
    console.error(`Failed to retrieve products from the server. Error: ${error}`);  
    return [];  
  }  
}
```

async/await in JavaScript

async/await ist ein Feature von JavaScript, das es ermöglicht, Funktionen asynchron auszuführen. Das bedeutet, dass der Rest des Programms weiterlaufen kann, auch wenn die Funktion, die aufgerufen wurde, länger braucht, um fertig zu werden.

Mit `await` wird die Antwort der GET-Anfrage an den Server abgewartet, bevor der nächste Schritt ausgeführt wird.

Die Antwort wird in der Variable `response` gespeichert. Diese enthält die Daten, die vom Server zurückgegeben werden.

Wenn die Anfrage erfolgreich ist, werden die Daten (in unserem Fall Produktinformationen) zurückgegeben.

Falls ein Fehler auftritt:

Der Fehler wird in der Konsole angezeigt.

Statt der Daten wird eine leere Liste (`[]`) zurückgegeben, damit das Programm weiterlaufen kann, ohne abzustürzen.

Dynamische Anzeige von Produkten

Produkte statisch im HTML-Code zu erstellen ist aus mehreren Gründen ineffizient. Stattdessen wird JavaScript verwendet, um die Produkte dynamisch anzuzeigen.

Der Code funktioniert in mehreren Schritten:

1. Produkte aus dem Backend abrufen

Mit der Methode `getAll` werden die Produktdaten aus dem Backend abgerufen und in der Variable `products` gespeichert:

```
let products = await getAll();
```

2. HTML-Code für jedes Produkt erstellen

Der HTML-Code wird dynamisch in JavaScript als String erstellt. Variablen wie `product_id`, `image_url`, `name`, `description` und `price` werden in den String eingefügt, damit jede Produktkarte individuelle Daten enthält. Die folgende Methode nimmt ein Produkt als JavaScript-Objekt und erstellt daraus den entsprechenden HTML-Code:

```
function getProductHTML({ product_id, image_url, name, description, price }) {
    return `
        <article class="product">
            <div class="content">
                
                <div class="product-description">
                    <h1>${name}</h1>
                    <p>${description}</p>
                    <p class="price">${price}€</p>
                </div>
            </div>
            <div class="button-container">
                <button product-id=${product_id}
                    class="default add-to-cart-button">Add to cart</button>
            </div>
        </article>
    `;
}
```

Der getProductHTML methode erwartet ein JavaScript-Object der wie folgt aussieht:

```
product = {
    product_id: "001",
    image_url: "../images/cpu.jpeg",
    name: "CPU Core i9",
    description: "Sehr starkes CPU für Gaming",
    price: 999
}

getProductHTML(product);
```

Objekt-Dekonstruktion in der Methodenkopf

In der Methodendeklaration wird das Objekt direkt aufgesplittet:

```
function getProductHTML({ product_id, image_url, name, description, price })
```

Durch die geschweiften Klammern {} und die Variablennamen können die Attribute eines Objekts direkt verwendet werden, ohne ständig mit dem Objektnamen und einem Punkt darauf zugreifen zu müssen.

3. Produkte auf der Benutzeroberfläche anzeigen

Die Methode `generateProducts` nimmt die Liste der Produkte und fügt jedes Produkt in den dafür vorgesehenen HTML-Container ein.

Erklärungen zu den Schritten

- **Container auswählen:** Der Container im HTML-Dokument, in dem die Produkte angezeigt werden sollen, wird mit `document.getElementById` ausgewählt.

```
const container = document.getElementById("products-container");
```

- **Produkte durchlaufen:** Jedes Produkt in der Liste wird geprüft, und nur Produkte, deren `quantity` größer als 0 ist (Im Lager verfügbar), werden verarbeitet.

```
products.forEach(product => {  
    if (product.quantity !== 0) {  
          
    }  
});
```

- **HTML-Code erstellen:** Für jedes Produkt wird mit der Funktion `getProductHTML` der individuelle HTML-Code generiert.

```
const productHTML = getProductHTML(product);
```

- **HTML einfügen:** Der generierte HTML-Code wird mit `insertAdjacentHTML` in den Container eingefügt.
insertAdjacentHTML ist eine Methode in JavaScript, der HTML-Inhalt direkt in eine bestehende DOM-Struktur (Document Object Model) einfügt, ohne den gesamten Inhalt des Elements zu überschreiben.

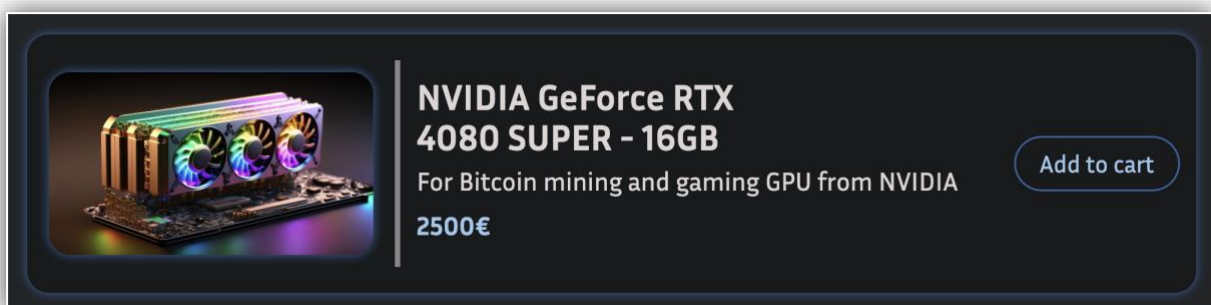
```
container.insertAdjacentHTML('beforeend', productHTML);
```

Mögliche Werte für Position:

1. **"beforebegin"**: Vor dem Ziel-Element selbst (außerhalb).
2. **"afterbegin"**: Direkt nach dem öffnenden Tag des Ziel-Elements (als erstes Kind).
3. **"beforeend"**: Direkt vor dem schließenden Tag des Ziel-Elements (als letztes Kind).
4. **"afterend"**: Nach dem Ziel-Element selbst (außerhalb).

Button-Eventhandler

Die Produktkarte enthält einen Button, mit dem ein Produkt in den Warenkorb hinzugefügt werden kann. Um den "Add to cart"-Button funktionsfähig zu machen, wurde ein Eventhandler für alle Buttons (von alle angezeigte Produkte) hinzugefügt.



Es wurde eine **Arrow-Funktion** erstellt, die den Button-Click verarbeitet. Arrow-Funktionen sind eine kompakte Schreibweise für Funktionen in JavaScript und haben eine kürzere Syntax als normale Funktionen.

Der Eventhandler für den "add to car"-Button läuft in mehreren Schritten ab:

- **Einloggen-Staus prüfen:** Zunächst wird überprüft, ob der Benutzer eingeloggt ist. Falls nicht, wird er auf die Login-Seite weitergeleitet, da nur angemeldete Benutzer Produkte in ihren Warenkorb hinzufügen können.


```
if (!isLoggedIn()) window.location.href = LOGIN_PAGE;
```

- Die Funktion isLoggedIn() überprüft, ob der Benutzer eingeloggt ist, und gibt true oder false zurück.
- LOGIN_PAGE enthält den Link zur Login-Seite. Mit window.location.href wird der Benutzer zu dieser Seite weitergeleitet.
- **Produkt-ID ermitteln:** Beim Erstellen der Produktkarten hat jeder Button ein Attribut namens product-id erhalten, das die ID des jeweiligen Produkts speichert. Diese ID wird beim Klick auf den Button ausgelesen, um das richtige Produkt in den Warenkorb zu legen.

```
const id = event.target.getAttribute("product-id");
```

- **Produkt zum Warenkorb hinzufügen:** Die Methode addItemToCart() wird aufgerufen. Sie benötigt drei Parameter:
- Den Benutzernamen (aus Sicherheitsgründen, die später in der Dokumentation erklärt werden).
- Die Produkt-ID (um den richtigen Produkt in dem Warenkorb hinzufügen).
- Die Menge (standardmäßig ist der Wert 1).

```
const status = await addItemToCart(username, id, 1);
```

- **Unter der Haube:** Die Methode addItemToCart() sendet eine **POST-Anfrage** an das Backend. Die notwendigen Informationen (Benutzername, Produkt-ID und Menge) werden im **Body** der Anfrage mitgeschickt.

```
const response = await api.post(`/cart/item/append`, {  
  username: username,  
  product_id: productId,  
  quantity: quantity  
});
```

- **Eventhandler-Einhängen:** Um den Eventhandler für alle Buttons hinzuzufügen, müssen zunächst alle relevanten Buttons ausgewählt werden.

```
const buttons = document.querySelectorAll(".add-to-cart-button");
```

In diesem Fall werden alle Buttons mit der Klasse “add-to-cart-button” selektiert.

Anschließend wird über alle Buttons iteriert, und jedem Button wird ein Eventhandler hinzugefügt:

```
buttons.forEach(el => el.addEventListener("eventType", handler));
```

EventType: Dieser definiert die Art des Events, auf das reagiert werden soll. In unserem Fall handelt es sich um ein click-Event.

Handler: Die Arrow-Funktion, die den Klick verarbeitet, wurde einer Variablen namens handler zugewiesen. Diese Variable wird dann als Eventhandler verwendet.

Dieser Ansatz ermöglicht eine flexible und dynamische Darstellung von Produkten, ohne den HTML-Code manuell anpassen zu müssen.

Gestaltung mit CSS

Die Webseite wurde mit **CSS** gestaltet, um ein ansprechendes und benutzerfreundliches Design zu gewährleisten. Da eine vollständige Beschreibung des gesamten CSS-Codes nicht notwendig ist, wird hier exemplarisch die Gestaltung der **Produktkarte** gezeigt. Die Produktkarte ist ein zentrales Element des Onlineshops und wurde so gestaltet, dass die wichtigsten Produktinformationen übersichtlich und visuell ansprechend dargestellt werden.

Das Beispiel zeigt, wie die Produktkarte gestaltet wurde, und veranschaulicht die wichtigsten Designentscheidungen für dieses zentrale Element der Webseite.

Wichtige Gestaltungspunkte der Produktkarte

1. Abgerundete Ecken und Schatten:

- Die Produktkarte hat **abgerundete Ecken** durch `border-radius: 20px`.
- Ein **Box-Schatten** (box-shadow) sorgt dafür, dass die Karte optisch hervorgehoben wird.

2. Zentrierung des Inhalts:

- Das Bild (img) wird zentriert und erhält ebenfalls **abgerundete Ecken** sowie einen Schatten, um einheitlich mit der Karte zu wirken.
- Das `display: block` und `margin: auto` stellen sicher, dass das Bild mittig im Container ausgerichtet ist.

3. Produktbeschreibung:

- Die Beschreibung im Container .product-description ist klar strukturiert.
 - **Produktname (h1)**: Wird durch eine größere Schriftgröße (`font-size: 1.8rem`) hervorgehoben.
 - **Preis (p.price)**: Hebt sich durch eine dickere Schrift (`font-weight: 700`) und eine kontrastierende Farbe ab.
- Eine Trennlinie (`border-left: 5px solid`) wurde zwischen dem Bild und der Beschreibung hinzugefügt, um die Bereiche visuell zu trennen.

4. Responsives Layout:

- Die Karte ist flexibel, da sie mit `width: 70%` arbeitet. Dadurch passt sie sich an verschiedene Bildschirmgrößen an, ohne das Design zu brechen.

5. Button-Container:

- Die Buttons sind in einem separaten Container (.button-container) positioniert, sodass sie optisch getrennt und leicht zugänglich sind.

Einloggen und Registrierung

Die Seiten **Ei**loggen und **Reg**istrierung sind sehr ähnlich und haben fast das gleiche Layout. Beide Seiten bestehen aus mehreren **Eingabefeldern**, wie z.B. für den Benutzernamen, die E-Mail-Adresse und das Passwort. Außerdem gibt es jeweils **Buttons**, um das Formular abzusenden.

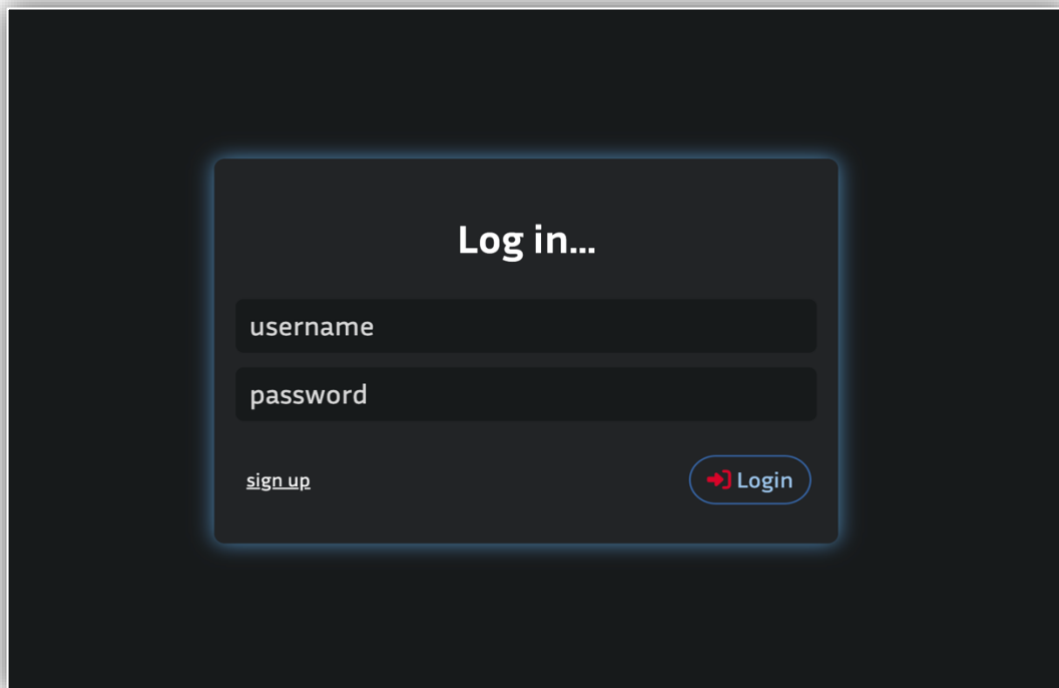
The image shows a dark-themed login interface. At the top, the text "Log in..." is centered. Below it are two input fields: the first is labeled "username" and the second is labeled "password". At the bottom left of the form area is a "sign up" link, and at the bottom right is a "Login" button with a red arrow icon.

Bild 4 - Einloggen-Seite

Es wird eine POST-Anfrage gesendet, wenn der **Login-Button** geklickt wird. Diese Anfrage enthält den **Benutzernamen** und das **Passwort**, die an das Backend weitergeleitet werden. Das Backend überprüft die gegebenen Daten und gibt das Ergebnis zurück. Das Frontend entscheidet basierend auf der Richtigkeit der Angaben, ob der Benutzer zur **Homepage** weitergeleitet wird oder eine **Fehlermeldung** angezeigt wird, falls das Backend ein Fehler zurückgibt.

Beim **Registrierungsprozess** werden mehrere Punkte überprüft:

- Wenn **der Benutzername** oder **die E-Mail** bereits existieren, wird eine Fehlermeldung angezeigt. Diese Überprüfung erfolgt durch das Backend, und anhand des zurückgegebenen Statuscodes

entscheidet das Frontend, ob eine Fehlermeldung angezeigt werden soll.

- **Die Passwörter** müssen übereinstimmen. Falls sie nicht übereinstimmen, wird auch eine Fehlermeldung angezeigt.
- Passwörter müssen ein bestimmtes Muster erfüllen. Wenn das Passwort nicht den Anforderungen entspricht, wird eine Meldung angezeigt. Wir haben eine Methode namens `isPasswordValid` erstellt, die prüft, ob das Passwort den Vorgaben entspricht.

```
function isPasswordValid(password) {  
    const minLength = 8;  
    const numbers = /[0-9]/g;  
    const upperCaseLetters = /[A-Z]/g;  
    const specialCharacters = /[$!&?=/];  
  
    return numbers.test(password) &&  
        upperCaseLetters.test(password) &&  
        specialCharacters.test(password) &&  
        password.length >= minLength;  
}
```

Die Methode überprüft, ob das Passwort: **mindestens 8 Zeichen lang ist, eine Zahl, einen Großbuchstaben und ein Sonderzeichen enthält.**

Sie verwendet dazu reguläre Ausdrücke und die `.test()`-Methode, um sicherzustellen, dass das Passwort diese Kriterien erfüllt. Die `.test()`-Methode überprüft ein gegebener String-Wert ein wert von der reguläre Ausdrücke beinhaltet und gibt ein Boolean-Wert zurück.

Wenn alle Bedingungen erfüllt sind, gibt die `isPasswordValid`-Methode `true` zurück, andernfalls `false`.

Die Profileseite

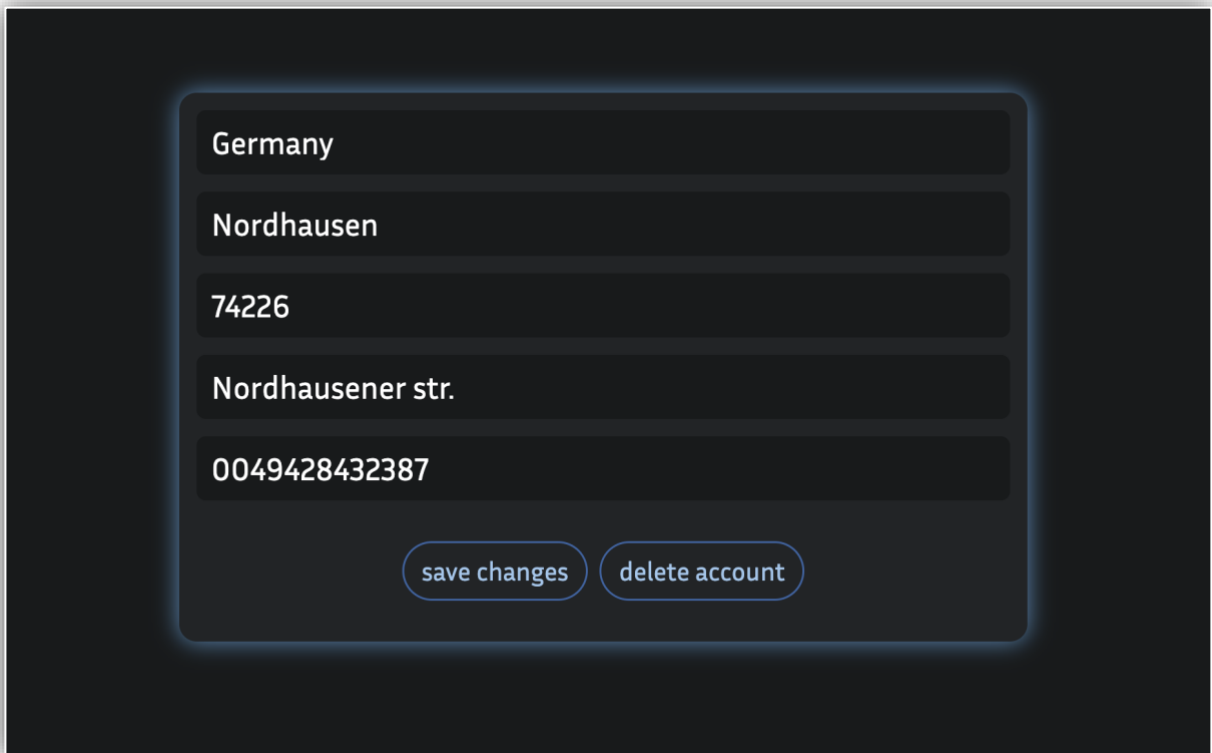
The image shows a dark-themed user profile interface. A central rounded rectangle contains five text input fields, each with a light blue border and a white glow. The fields are stacked vertically and contain the following text: 'Germany', 'Nordhausen', '74226', 'Nordhausener str.', and '0049428432387'. Below these fields are two rounded buttons with blue borders and white text: 'save changes' and 'delete account'.

Bild 5 – Profileseite

Die Profilseite zeigt die Adresse des Benutzers, die jederzeit bearbeitet werden kann. Außerdem hat der Benutzer die Möglichkeit, sein Konto zu löschen. Beim Löschen wird ein Backend-Endpunkt aufgerufen, der alle mit dem Benutzer verbundenen Daten in der Datenbank entfernt.

Nach der löschen vom Konto, wird der Benutzer direkt zu der homepage (ausgeloggt) weitergeleitet.

Der Warenkorb

Die Produkte im Warenkorb werden auf ähnliche Weise dynamisch generiert wie auf der Homepage. Auch das Preisfenster wird dynamisch aktualisiert, da sich der Gesamtpreis durch das Löschen des Produkts oder Ändern der Produktmengen im Warenkorb laufend ändern kann.

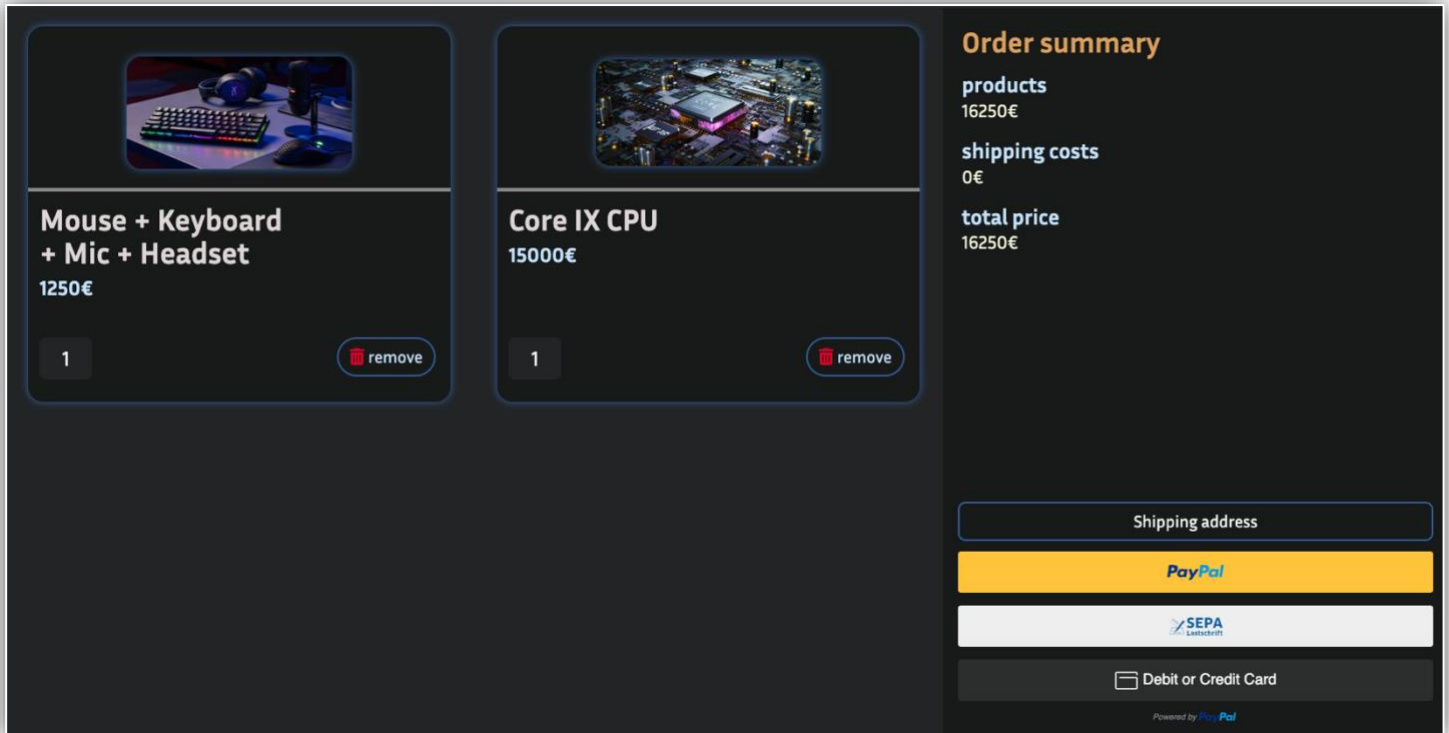


Bild 5 - Der Warenkorb

Der "Shipping Address"-Button führt den Benutzer zu seiner Profilseite, wo die Lieferadresse bearbeitet oder bestätigt werden kann.

Bezahlungsmethode

In dem Onlineshop stehen hauptsächlich **PayPal** und **Kreditkarte** als Zahlungsmethoden zur Verfügung. Die Kreditkartenzahlung ist bereits in der PayPal-Konfiguration integriert und muss nicht separat implementiert werden.

Wir haben uns für PayPal entschieden, da es eine sehr bekannte und vertrauenswürdige Zahlungsmethode ist. Außerdem bietet PayPal zahlreiche Integrationsmöglichkeiten, darunter auch eine **No-Code-Variante**. Diese ist jedoch nicht anpassbar und für größere Projekte weniger geeignet. Aus diesem Grund haben wir uns für die **Standard-Integration** entschieden. Dabei müssen über die PayPal-Dokumentation einige Optionen wie Programmiersprache, Währung und Land vor dem Code-Impelemntierung ausgewählt werden.

Implementierung von PayPal

Backend

Durch das Hinzufügen von zwei spezifischen API-Endpunkten für PayPal in dem Backend-Code, mit denen das Frontend kommuniziert.

- Der erste Endpunkt, `/api/orders`, erstellt eine Bestellung und gibt eine eindeutige Bestell-ID zurück.
- Der zweite Endpunkt, `/api/orders/{orderID}/capture`, führt den Bezahlprozess durch.

Frontend

Im Frontend wird ein **div-Element** mit einem bestimmten ID und Klassennamen benötigt:

```
<div id="paypal-button-container" class="paypal-button-container">
</div>
```

Dazu kommt ein JavaScript-Link, der mit PayPal kommuniziert und die **PayPal-Buttons** in dieses **div-Element** einfügt. Zusätzlich wird JavaScript-Code benötigt, um die Buttons mit Funktionen auszustatten und die Interaktion mit dem Backend zu ermöglichen.

Der Code für Frontend und Backend kann direkt aus der [PayPal-Dokumentation](#) kopiert und in die Applikation eingefügt werden. Es ist jedoch sehr sinnvoll, den Code zu verstehen, da bestimmte Anpassungen und Konfigurationen erforderlich sind.

Preisübergabe

Standardmäßig ist der Preis im Backend-Code **fest** einprogrammiert (hardcodiert). Daher muss der Gesamtpreis vom Frontend als Parameter im HTTP-Request an das Backend übergeben werden.

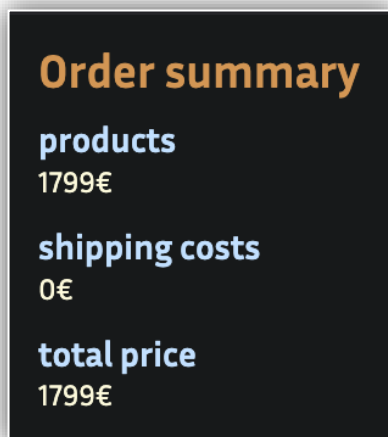
Um dies zu erreichen, haben wir eine Methode namens `getTotal`, die einen GET-Request an das Backend sendet, um den Gesamtpreis zu ermitteln:

```
const response = await api.get(`/cart/${username}/total`);
return response.data;
```


Als Nächstes muss der Gesamtpreis mit dem Request an den /api/orders-Endpunkt übergeben werden, damit er bei der Erstellung der Bestellung berücksichtigt wird. Somit kann PayPal erkennen, wie viel der Käufer bezahlen muss:

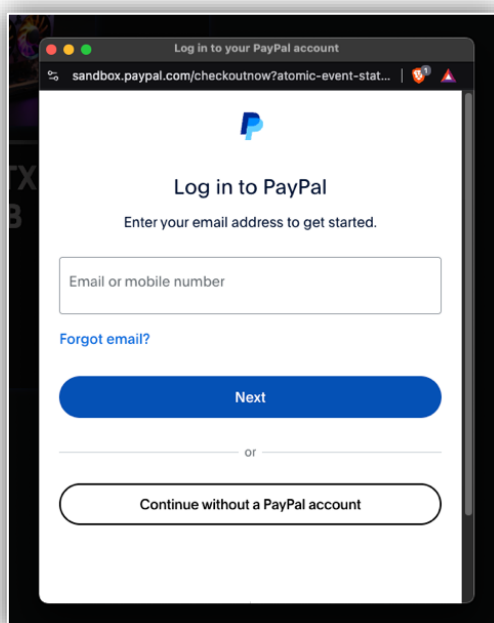
```
// use the "body" param to optionally
// pass additional order information
// like product ids and quantities
body: JSON.stringify({
  totalPrice: `${totalPrice}`
}),
```

Beispiel

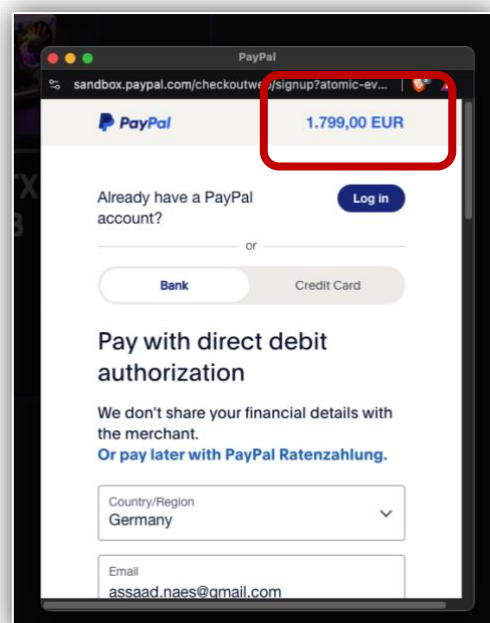


Der Gesamtpreis beträgt in diesem Beispiel 1799,00 €. Beim Klicken auf den PayPal-Button beginnt der Bezahlprozess:

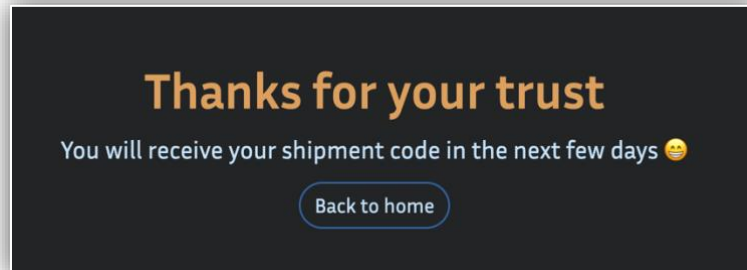
Einloggenprozess



Bezahlung mit dem richtigen Betrag



Bei erfolgreicher Bezahlung wird der Benutzer auf eine Danke-Seite weitergeleitet. Der Warenkorb wird anschließend über einen speziellen API-Endpunkt im Backend geleert.



```
// (3) Successful transaction -> Show confirmation or thank you message  
// Or go to another URL: actions.redirect('thank_you.html');  
clearCart(localStorage.getItem("username"));  
actions.redirect("thankyou.html");
```

Client-ID und Client-Secret

Um PayPal mitzuteilen, wer der Verkäufer ist, sind die Client-ID und das Client-Secret sehr wichtige Informationen, die unbedingt **sicher** behandelt werden müssen.

Während des Entwicklungsprozesses stellt PayPal eine **Sandbox-Umgebung** (Testumgebung) zur Verfügung, in der die **Client-ID** und das **Client-Secret** vorgegeben werden. Dies ermöglicht es Entwicklern, ihre Implementierung ohne Risiken zu testen. Da diese Sandbox-IDs und -Secrets keinem Benutzerkonto zugeordnet sind, werden keine echten Zahlungen durchgeführt.

Später kann der Verkäufer eine eigene Client-ID und ein eigenes Client-Secret erstellen und nutzen. Dafür ist jedoch ein **PayPal-Business-Konto** erforderlich.

Frontend: Im Frontend muss lediglich die Client-ID angegeben werden, damit PayPal weiß, an wen die Zahlung weitergeleitet werden soll.

Backend: Das Backend benötigt sowohl die Client-ID als auch das Client-Secret, um einen PayPal-Client zu erstellen, der die Zahlungsprozesse abwickelt.

Media Queries

Durch die Verwendung von CSS-Media-Queries kann die Webseite mit allen Bildschirmgrößen kompatibel gemacht werden, ohne dass sie auf Handys durcheinander dargestellt wird. Ein sehr gutes Entwicklungsmuster ist es, zuerst den CSS-Code für kleinere Bildschirme zu schreiben und dann für größere Bildschirme mithilfe von Media Queries zu gestalten. Ein Beispiel dafür ist unsere Homepage:

```
/*=====
=                MEDIUM VIEWS                =
=====*/
@media only screen and (min-width: 700px) { ...
}
```

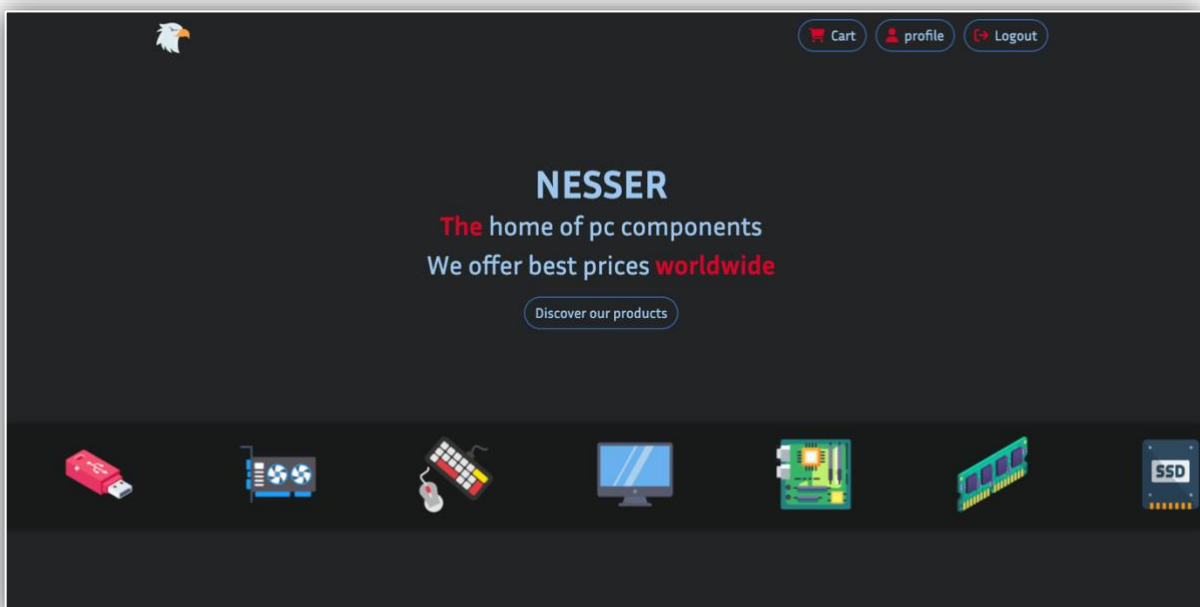
Für Bildschirmgrößen ab 700px haben wir bestimmte Gestaltungsversionen implementiert.

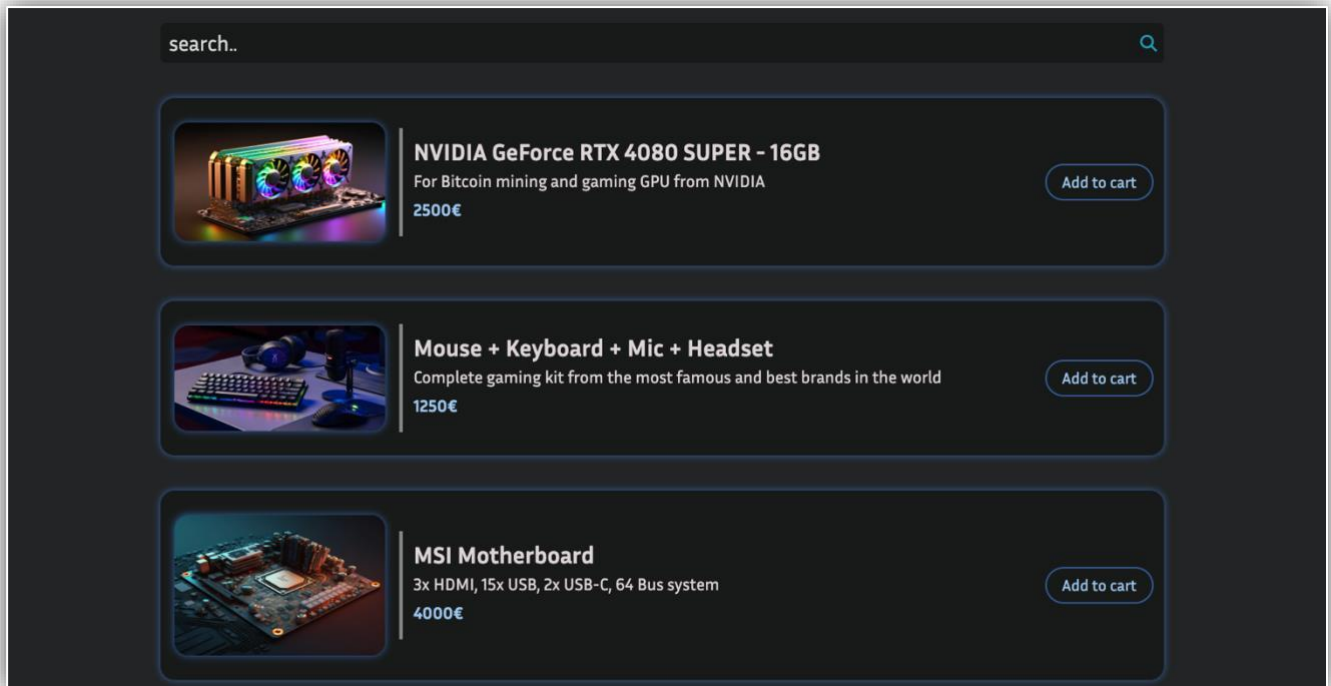
```
/*=====
=                LARGE VIEWS                =
=====*/
@media only screen and (min-width: 1100px) { ...
}
```

Für größere Bildschirme ab 1100px haben wir dann die Gestaltung für normale PC-Bildschirme angepasst.

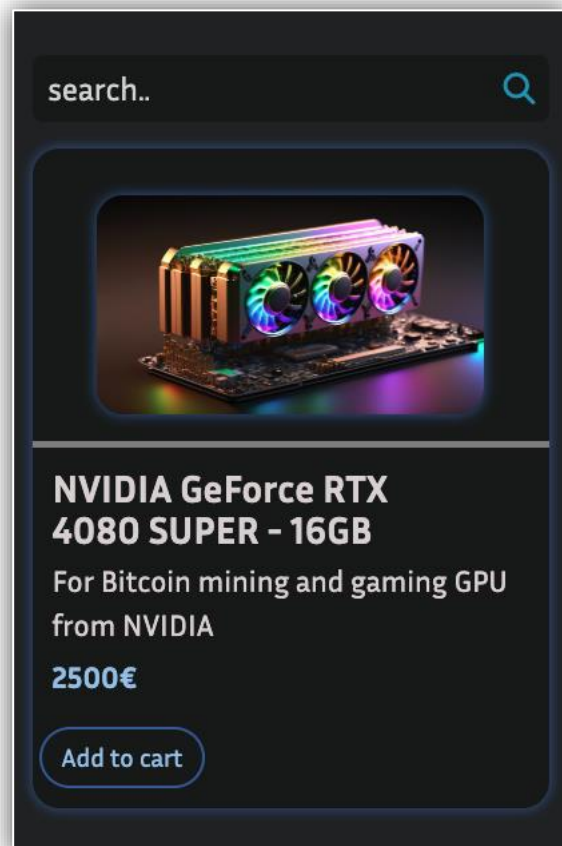
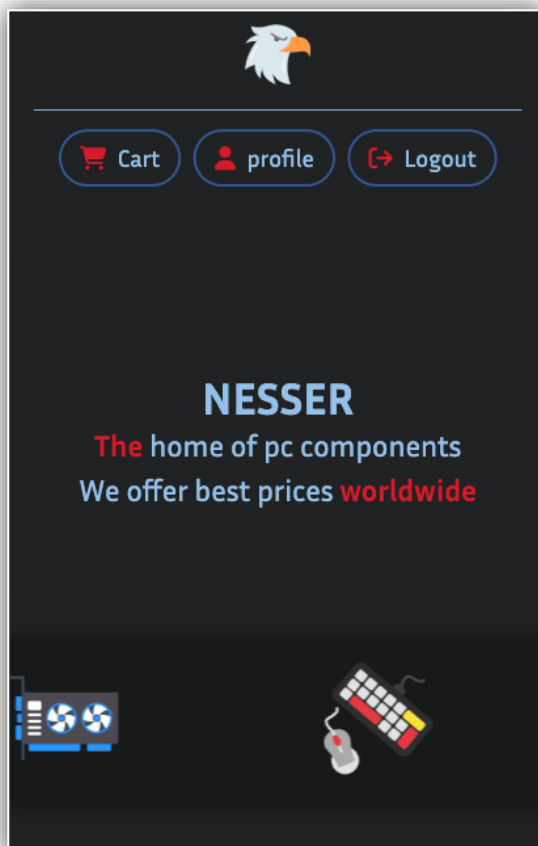
Die Homepage sieht auf unterschiedlichen Bildschirmen wie folgt aus:

Größe-Anzeigen





Kleine Anzeigen



Das Backend

Das Backend unseres Projekts sollte ursprünglich mit PHP entwickelt werden, da wir die Sprache kennenlernen sollten. Allerdings hatten wir im Team noch keine Erfahrung mit PHP, was die Umsetzung erschwerte. Wir haben unser Bestes gegeben und versucht, das Backend in PHP zu realisieren. Leider war dieser Ansatz nicht erfolgreich.

Um das Projekt nicht scheitern zu lassen, haben wir uns entschieden, das Backend stattdessen mit Java zu entwickeln. Diese Entscheidung ermöglichte es uns, das Projekt erfolgreich abzuschließen. Dennoch erkennen wir die Bedeutung von PHP und bedauern, dass wir es nicht einsetzen konnten.

Um unser Ziel, die Grundlagen von PHP zu verstehen, trotzdem zu erreichen, haben wir in der Projektdokumentation Methoden aus unserem Java-Code mit entsprechenden Umsetzungen in PHP verglichen. Dadurch konnten wir uns mit der Funktionsweise von PHP vertraut machen und einen Lerneffekt erzielen, auch wenn wir es nicht praktisch eingesetzt haben.

Framework

Spring Framework ist ein bekanntes Java-Framework, das die Entwicklung von Anwendungen vereinfacht, insbesondere für Webanwendungen. Es bietet viele Module, um gängige Funktionen abzudecken, wie Datenbankzugriff, Sicherheit und Web-Entwicklung.

Einige **Vorteile** von Spring sind:

- **Spring Boot:** Es vereinfacht die Erstellung von Web-Anwendungen durch vorgefertigte Konfigurationen und minimiert den notwendigen Code.
- **Spring Security:** Dieses Modul sorgt für Sicherheit, indem es Funktionen wie Authentifizierung und Autorisierung bietet, was besonders wichtig für den Schutz sensibler Daten ist.

- **Abhängigkeiten:** Durch die Nutzung von Abhängigkeiten (Dependencies) können gezielt nur die benötigten Funktionen und Bibliotheken in das Projekt integriert werden. Das verbessert die Skalierbarkeit, da neue Funktionen leicht hinzugefügt oder bestehende Komponenten angepasst werden können.

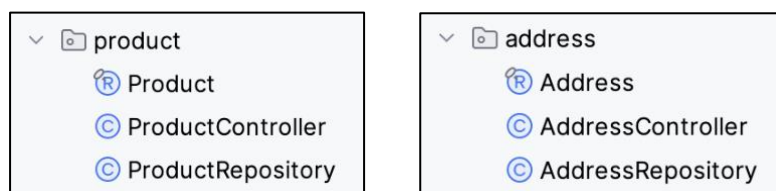
In unserem Projekt haben wir **Spring Boot** genutzt, um effizient ein sicheres und skalierbares Backend-API zu entwickeln. **Spring Security** spielte eine zentrale Rolle, um den Zugriff auf unsere API-Endpunkte zu schützen.

Aufbau

Die Backend-API wurde nach dem MVC-Muster (Model, View, Controller) entwickelt.

- **Model:** Zuständig für den Datenaustausch mit der Datenbank, wie das Lesen, Schreiben, Aktualisieren und Löschen von Daten.
- **View:** Die Ansicht, die dem Benutzer angezeigt wird.
- **Controller:** Beinhaltet die API-Endpunkte und fungiert als Schnittstelle zwischen der View (dem Benutzer) und dem Model.

Für jede Art von Operation wird eine separate Repository- und Controller-Klasse erstellt. Dadurch bleibt der Code gut strukturiert, getrennt und übersichtlich. **Beispielklassen:**



Sicherheit

Unser Online-Shop erfordert für bestimmte Endpunkte eine Authentifizierung und Autorisierung. Durch die Integration der Spring-Security-Abhängigkeit in die Anwendung wird standardmäßig ein Login-

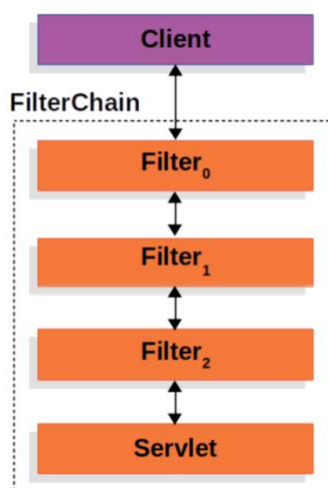
Mechanismus bereitgestellt, der den Zugriff auf alle Endpunkte bis zur erfolgreichen Anmeldung des Benutzers blockiert.

Allerdings bietet Spring Security keine Registrierungsmöglichkeiten und keine automatische Passwortverschlüsselung. Daher muss Spring Security konfiguriert werden, um die gewünschten Funktionen und Sicherheitsstandards zu erreichen.

Spring-Security-Filter

Jede Anfrage, die an das Backend gesendet wird, durchläuft mehrere Filter von Spring Security, bevor sie den Server erreicht. Diese Filter sorgen für verschiedene Sicherheitsmaßnahmen und Funktionen, die auf die eingehende Anfrage angewendet werden.

Die Reihe von Filter wird als FilterChain bezeichnet:



Die Abfolge dieser Filter wird als **FilterChain** bezeichnet.

Ein besonders wichtiger Filter in dieser Kette ist der **UsernamePasswordAuthenticationFilter**. Dieser Filter verarbeitet die Anmeldedaten (Benutzername und Passwort) aus einer HTTP-Anfrage, überprüft deren Gültigkeit und authentifiziert den Benutzer.

Spring-Security-Konfiguration

In unserem Projekt haben wir umfangreiche Anpassungen vorgenommen, um die Sicherheitsanforderungen unserer Web-Applikation zu erfüllen.

Da eine vollständige Beschreibung der Konfiguration den Rahmen der Dokumentation sprengen würde, konzentrieren wir uns hier nur auf die wichtigsten Einstellungen.

CORS-Konfiguration

Moderne Webanwendungen bestehen oft aus einem Frontend und einem Backend, die auf unterschiedlichen Domains gehostet sind. Beispielsweise könnte eine Anwendung aus folgenden Teilen bestehen:

- **Frontend:** <http://frontend.com>
- **Backend:** <http://api.com>

Wenn das Frontend eine Anfrage an das Backend stellt, prüft der Browser, ob diese Anfrage erlaubt ist.

Ohne eine spezielle Konfiguration durch Cross-Origin Resource Sharing (CORS) würde der Browser die Anfrage blockieren, um die Anwendung vor potenziellen Sicherheitsrisiken wie **Cross-Site-Scripting (XSS)** oder **Cross-Site Request Forgery (CSRF)** zu schützen.

Da das Frontend unserer Webapplikation mit dem Backend kommunizieren muss, ist es notwendig, CORS zu konfigurieren, um diese Anfragen gezielt zu erlauben:

```
CorsConfigurationSource corsConfigurationSource() {  
    CorsConfiguration configuration = new CorsConfiguration();  
    configuration.setAllowedOrigins(List.of("*"));  
    configuration.setAllowedMethods(List.of("GET", "POST", "PUT", "DELETE", "OPTIONS"));  
    configuration.setAllowedHeaders(List.of("Authorization", "Content-Type"));  
    UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();  
    source.registerCorsConfiguration(pattern: "**", configuration);  
    return source;  
}
```

- **CorsConfiguration**-Instanz wird verwendet, um die erlaubten Regeln zu definieren.
- Der **SetAllowedOrigins** erlaubt Anfragen von den definierten Domänen. Ein "*" steht für alle Domänen.
- **SetAllowedMethods** definiert welche HTTP-Methoden erlaubt sind.
- **SetAllowedHeaders** gibt die erlaubten HTTP-Header an.
- Um die Konfiguration mit bestimmten **URL-Pfaden** zu verknüpfen, wird der **UrlBasedConfigurationSource** verwendet. "/"**" bedeutet,

dass die Konfiguration auf alle Endpunkte im Backend angewendet werden sollen.

JWT-Konfiguration

Bevor JWT zu erklären muss zu erst den Unterschied zwischen Session-based und Token-based Authentifizierung veranschaulicht werden.

Session-based Authentifizierung: Der Server erstellt nach dem Login eine Session-ID und speichert diese in einer serverseitigen Datenbank. Die Session-ID wird beim Client (meist als Cookie) gespeichert und bei jeder Anfrage an den Server mitgeschickt. Der Server prüft die ID in seiner Datenbank, um die Sitzung zu validieren.

Ein Nachteil tritt auf, wenn mehrere Server im Einsatz sind (z. B. in einer Lastverteilungsumgebung mit einem Load Balancer). Wenn der Load Balancer eine Anfrage an einen Server weiterleitet, der die entsprechende Session-ID nicht kennt (weil sie auf einem anderen Server gespeichert ist), kann der Benutzer gezwungen sein, sich erneut anzumelden. Dieses Problem erfordert zusätzliche Maßnahmen, wie das Synchronisieren der Sessions zwischen den Servern oder die Nutzung von zentralen Speichermöglichkeiten.

Token-based Authentifizierung: Der Server erstellt nach dem Login ein Token, dass beim Client gespeichert wird (im Local Storage als Beispiel). Der Token wird bei jeder Anfrage gesendet und enthält alle benötigten Informationen, sodass der Server keine Sitzungsdaten speichern muss, was der Problem löst.

JSON-Web-Token (JWT):

Ist ein kompakter, sicherer Token, der aus drei Teile besteht:

- **Header:** Metadaten (z. B. Signaturalgorithmus).
- **Payload:** Daten wie Benutzerinformationen oder Berechtigungen.
- **Signature:** Sicherstellung, dass der Token nicht manipuliert wurde (z. B. durch HMAC-SHA256).

JWT wird oft in Token-based Authentication verwendet und kann serverseitig überprüft werden, ohne einen Zustand zu speichern.

Implementierung von JWT

Um JWT in Spring Security zu integrieren, muss ein **JWT-Filter** erstellt und in die Filterchain von Spring Security eingebunden werden, damit alle Anfragen durch den JWT-Filter laufen.

```
.addFilterBefore(jwtFilter, UsernamePasswordAuthenticationFilter.class)
```

Die Methode **addFilterBefore** wird dafür verwendet. Wie der Name schon sagt, fügt diese Methode einen Filter vor einem anderen Filter ein. Das bedeutet, dass der eingefügte Filter (in diesem Fall `jwtFilter`) die Anfragen vor dem angegebenen Filter (`UsernamePasswordAuthenticationFilter`) verarbeitet.

Das ist in unserem Fall sehr wichtig, da der `jwtFilter` überprüft, ob ein gültiger Token vorhanden ist. Falls der Token gültig ist, muss die Anfrage nicht mehr nach einem Benutzernamen und Passwort authentifiziert werden.

Token

Ein JWT-Token sieht wie folgendes aus:

eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJhc3NhYWVzMDIzIiwiaWF0IjoxNzM0MDM2ODI3LCJleHAiOiE3MzQzOTY4Mjd9.o5Kqge1Mw9nMTBzavKqAnlaevTDeLOufDLaWc7LWhM

Und beinhaltet folgende Informationen:

HEADER: ALGORITHM & TOKEN TYPE	PAYLOAD: DATA	VERIFY SIGNATURE
<pre>{ "alg": "HS256" }</pre>	<pre>{ "sub": "assaadnaes023", "iat": 1734036827, "exp": 1734396827 }</pre>	<pre>HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), your-256-bit-secret) <input type="checkbox"/> secret base64 encoded</pre>

Registrierung und Passwortverschlüsselung

Um die Registrierung zu ermöglichen, wurde im UserController-Klasse ein API-Endpoint erstellt, der ein User-Objekt vom Frontend erwartet. Das User-Objekt stammt aus der User-Klasse, die wichtige Informationen (ID, Benutzername, Email und Passwort) des Benutzers enthält:

```
@Data 10 usages
public class User {
    private int user_id;
    private String username;
    private String email;
    private String password;
}
```

In Spring Boot sind **Annotations** spezielle Markierungen, die dem Framework Anweisungen geben, wie bestimmte Klassen oder Methoden behandelt werden sollen. Sie vereinfachen die Konfiguration und Aktivierung von Funktionen, ohne dass viel zusätzlicher Code

geschrieben werden muss. Der @Data als Beispiel Annotation fügt alle nötigen Getters und Setters für alle definierten Variablen in dieser Klasse, somit kann der geschriebene Code minimiert werden.

Register-Endpoint

```
@PostMapping("/register")
public ResponseEntity<String> register(@RequestBody User user) {
    userRepository.register(user);
    logger.info("User: {} registered", user.getUsername());
    return new ResponseEntity<>(body: "User registered successfully", HttpStatus.CREATED);
}
```

Die @PostMapping-Annotation gibt an, dass die Methode eine POST-Anfrage für den Endpoint /register erwartet. Das bedeutet, wenn das Frontend eine POST-Anfrage an den Endpoint <http://localhost:8080/register> sendet, wird die Methode register aufgerufen.

Die Methode gibt ein **ResponseEntity** zurück. ResponseEntity in Java (Spring) ist eine Klasse, die eine HTTP-Antwort repräsentiert. Sie enthält

den **Statuscode** sowie den Antwortinhalt (z. B. einen **String**) und ermöglicht so eine flexible Konfiguration der Antwort.

Die **@RequestBody**-Annotation teilt Spring Boot mit, dass der Inhalt der Anfrage im Body des Requests (in diesem Fall das User-Objekt) extrahiert und an die Methode übergeben werden soll.

Das User-Objekt wird an die `userRepository`-Klasse weitergegeben, um die notwendigen Informationen in der Datenbank zu speichern. Danach wird das Erstellen eines neuen Benutzers in den **Logs** vermerkt, und eine passende `ResponseEntity`-Antwort wird an das Frontend zurückgesendet.

Der Verarbeitung in der Repository-Klasse

```
public void register(User user) { 1 usage
    user.setPassword(encoder.encode(user.getPassword()));

    jdbcClient.sql("INSERT INTO users (username, email, password) " +
        "VALUES (?, ?, ?) " +
        "RETURNING username") StatementSpec
        .params(user.getUsername(), user.getEmail(), user.getPassword())
        .query(String.class) MappedQuerySpec<String>
        .single();
}
```

Als erstes wird das eingegebene Passwort von dem Benutzer verschlüsselt, bevor es in der Datenbank gespeichert wird.

Zunächst wird das vom Benutzer eingegebene Passwort verschlüsselt, bevor es in der Datenbank gespeichert wird. Der **encoder** ist ein Objekt des **BCryptPasswordEncoder**, das ganz oben in der Klasse definiert wurde:

```
BCryptPasswordEncoder encoder = new BCryptPasswordEncoder( strength: 12);
```

Der **BCryptPasswordEncoder** in Java wird verwendet, um Passwörter sicher zu verschlüsseln. Es nutzt eine **One-Way-Verschlüsselung**, was bedeutet, dass die Passwörter nicht entschlüsselt werden können.

Die **Strength** bei der Objekterstellung gibt die Anzahl der Iterationen bei der Verschlüsselung an – höhere Werte bedeuten stärkere Sicherheit, aber auch mehr Rechenaufwand.

Als nächstes werden die Informationen des Benutzers mithilfe eines Insert-Statements in der Datenbank gespeichert. Dabei werden die Werte aus dem Benutzerobjekt über **Prepared Statements** hinzugefügt, um SQL-Injection-Angriffe zu vermeiden.

Das jdbcClient-Objekt gehört zur Klasse JdbcClient, die eine Verbindung zur Datenbank ermöglicht. Die Datenbankeigenschaften (URL, Benutzername und Passwort) werden in den *Application Properties* der Anwendung definiert.

Fallst das Komplette Prozess erfolgreich gelaufen ist, ist der Benutzer registriert und kann sich direkt mit seinem Benutzernamen und Passwort registrieren.

Im Fall von Fehlern

Für den Fall, dass aus irgendeinem Grund eine Exception auftritt, haben wir eine Klasse namens **GlobalExceptionHandler** erstellt. Diese Klasse fängt spezifische Exceptions ab und verarbeitet sie entsprechend dem jeweiligen Szenario.

Ein Beispiel hierfür ist, wenn ein Benutzer versucht, sich mit einem bereits existierenden Benutzernamen oder einer E-Mail-Adresse zu registrieren, die bereits in der Datenbank gespeichert ist. In diesem Fall wird eine **DuplicateKeyException** ausgelöst:

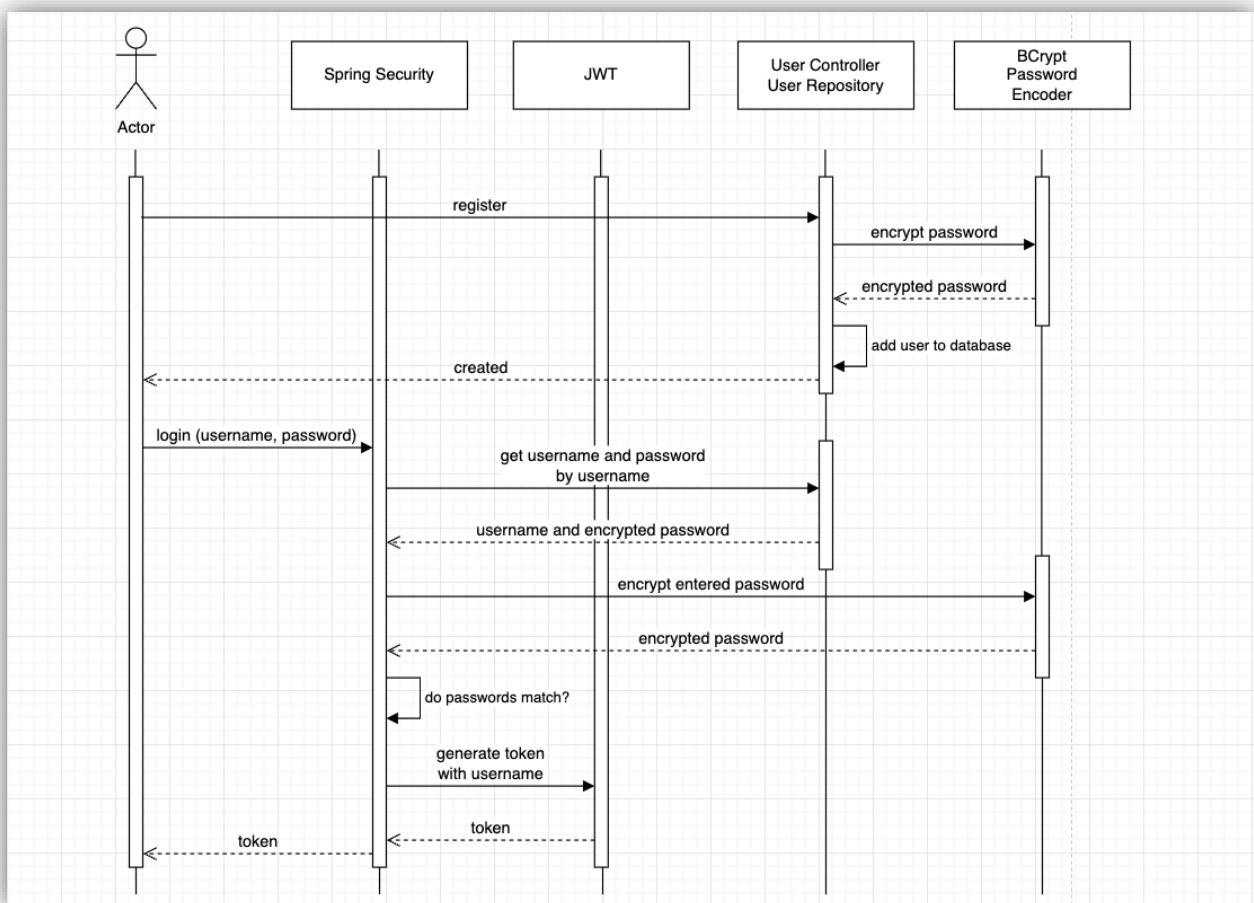
```
@ExceptionHandler(DuplicateKeyException.class)
public ResponseEntity<String> handleDuplicateKeyException(DuplicateKeyException e) {
    logger.warn( message: "Request failed. {}", e.getMessage());
    return new ResponseEntity<>( body: "Already exists!", HttpStatus.CONFLICT);
}
```

Der **ExceptionHandler** fängt diese Exception ab und gibt ein **ResponseEntity**-Objekt zurück. Dieses enthält eine entsprechende Fehlermeldung, die den Benutzer darauf hinweist, dass der Benutzername oder die E-Mail-Adresse bereits existiert.

Diagramme

Sequenzdiagramm

Der Registrierung- und Login-Prozess mit JWT und Spring Security wird im folgenden Sequenzdiagramm auf einfache Weise veranschaulicht:



Registrierungsprozess

- Der Benutzer ruft als Erstes den Register-API-Endpoint mit den notwendigen Informationen auf.

- Die **UserRepository-Klasse** verschlüsselt das eingegebene Passwort mit dem **BcryptPasswordEncoder**, der ein verschlüsseltes Passwort zurückgibt. Anschließend wird der Benutzer in die Datenbank eingefügt.

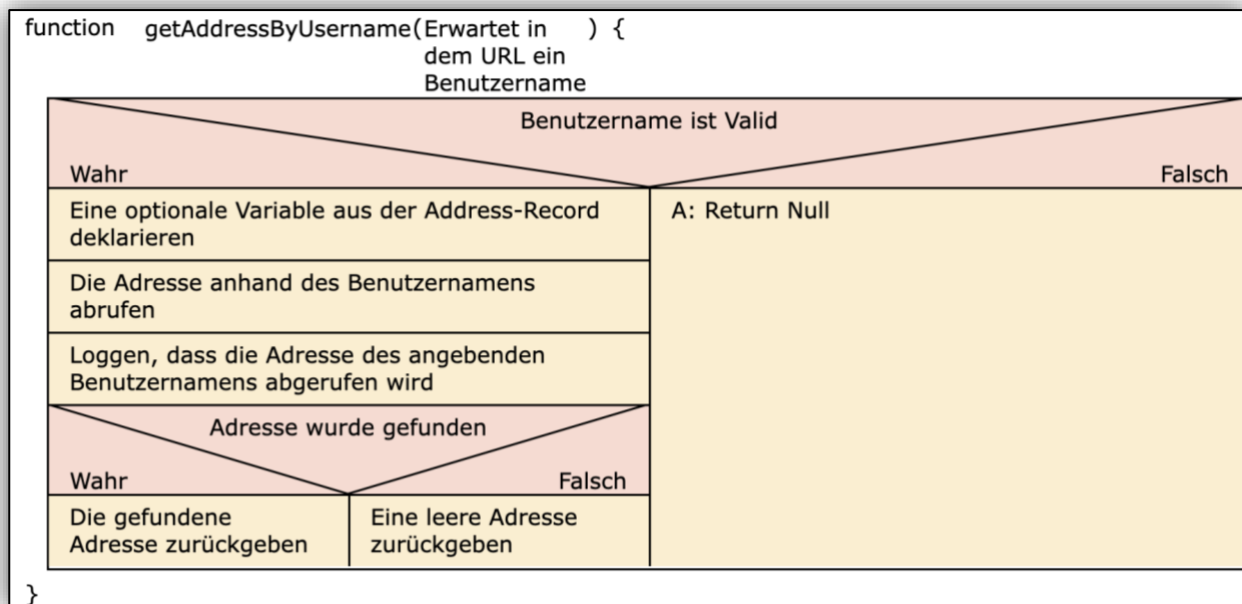
Einloggenprozess

- Der Benutzer gibt seinen **Benutzernamen** und das **Passwort** ein.
- **Spring Security** empfängt die Anmeldedaten und ruft über das Repository den Benutzernamen und das Passwort aus der Datenbank ab, sofern der Benutzername existiert. Falls der Benutzername nicht existiert, wird der Prozess abgebrochen, und der Benutzer erhält eine entsprechende Fehlermeldung.
- Da das Passwort in der Datenbank verschlüsselt ist, wird das vom Benutzer eingegebene Passwort ebenfalls verschlüsselt, um einen Vergleich durchführen zu können.
- **Spring Security** vergleicht die beiden Passwörter. Falls sie übereinstimmen, wird ein **JWT-Token** erzeugt, der den Benutzernamen enthält. Der Token wird schließlich an das Frontend zurückgesendet.

Struktogramm

```
@GetMapping("/{username}")
public Address getAddressByUsername(@PathVariable String username) {
    if (jwtService.validateUsername(username)) {
        Optional<Address> address = addressRepository.findByUsername(username);
        logger.info(message: "Getting address of user: {}", username);
        return address.orElseGet(() -> new Address( username: "", country: "", city: "", plz: 0, street: "", phone_number: ""));
    }
    return null;
}
```

Die Methode **getAddressByUsername** wird mithilfe eines Struktogramms veranschaulicht:



Ein **Sicherheitsproblem** in dem Projekt bestand darin, dass ein Benutzer mit Hilfe von API-Test-Tools wie **Postman** den Login-Endpoint aufrufen konnte. Nach einem erfolgreichen Login erhielt der Benutzer sein **JWT-Token** im Klartext. Mit diesem Token konnte er anschließend unberechtigte Aktionen durchführen, indem er es im Bearer-Token-Header einfügte.

Ein Beispiel:

Der aktuell angemeldete Benutzer, **Oliver**, versucht, die Adresse eines anderen Benutzers, **Assaad**, zu erhalten. Dafür verwendet er sein JWT-Token und ruft den Endpoint <http://localhost:8080/address/assaad> auf. In der bisherigen Implementierung hätte Oliver dadurch die Adresse von Assaad abrufen können, obwohl er dazu nicht berechtigt ist.

Lösungsansatz:

Um dieses Problem zu lösen, wurde der **Benutzername des angemeldeten Benutzers in das JWT-Token integriert**. Nun wird bei jedem Zugriff auf den Adress-Endpoint geprüft, ob der Benutzername im JWT-Token mit dem Benutzernamen im Pfad der Anfrage übereinstimmt. Falls die beiden Benutzernamen nicht identisch sind, wird die Anfrage abgelehnt.

Da JWT-Tokens **serverseitig** generiert und signiert werden, können sie nicht manipuliert werden. Diese Anpassung stellt sicher, dass unberechtigte Zugriffe auf die Daten anderer Benutzer effektiv verhindert werden. Das Sicherheitsproblem wurde damit erfolgreich behoben.

PHP-Code

```
@PostMapping("/item/append")
public ResponseEntity<String> addItemToCart(@RequestBody CartItem cartItem) {
    if (jwtService.validateUsername(cartItem.username())){
        cartItemRepository.save(cartItem);
        logger.info("message: \"Added Item to cart. username {}\", cartItem.username());
        return new ResponseEntity<>("Item added successfully", HttpStatus.ACCEPTED);
    }
    return new ResponseEntity<>("User not authorized", HttpStatus.UNAUTHORIZED);
}
```

Die **addItemToCart**-Methode fügt ein Produkt in den Warenkorb eines Benutzers hinzu. Die Methode könnte in **PHP** folgendermaßen aussehen: Zunächst müssen die Objekte **jwtService**, **cartItemRepository** und **der Logger** initialisiert werden. Dazu müssen die entsprechenden Variablen deklariert werden:

```
protected $jwtService;
protected $cartItemRepository;
protected $logger;
```

Um ihnen eine Objektinstanz zuzuweisen, basiert dies in vielen modernen Frameworks über dem Konstruktor:

```
public function __constructor(JwtService $jwtService, CartItemRepository
    $cartItemRepository, LoggerInterface $logger) {
    $this->jwtService = $jwtService;
    $this->cartItemRepository = $cartItemRepository;
    $this->logger = $logger;
}
```

Der Hauptunterschied zu Java ist der Zugriff auf Kind-Objekte anhand eines Pfeils nach rechts (->) statt mit einem Punkt (.). Der restliche Code sieht so aus:

```
public function addItemToCart(CartItem $cartItemRequest) {
    $cartItem = $cartItemRequest->getJson();

    if ($this->jwtService->validateUsername($cartItem->username)) {
        $this->cartItemRepository->save($cartItem);
        $this->logger->info("Added Item to cart. username: " . $cartItem
            ->username);
        return response()->json(['message' => 'Item added successfully'], 202);
    }

    return response()->json(['message' => 'User not authorized'], 401);
}
```

Ganz wichtig: Jede PHP-Datei muss mit `<?php` anfangen und mit `?>` enden.

Wir konnten durch dieses Projekt viel Neues lernen. Obwohl wir, wie bereits erwähnt, PHP in diesem Projekt nicht einsetzen konnten, haben wir dennoch erfolgreich versucht, das Projekt mithilfe anderer Sprachen und Technologien abzuschließen und zum Erfolg zu führen.

Für Fragen stehen wir gerne zur Verfügung

Assaad Naes & Oliver Wallisch