



# **Frankfurt University of Applied Sciences**

– Faculty of Computer Science and Engineering –

## **Project Cloud Computing**

LowTech GmbH

Milestone 3

Submitted by

**Timon Froussos**

Matrikelnummer: 1133510

# Contents

<b>1</b>	<b>Backend Implementation</b>	<b>1</b>
1.1	Django Project and App Structure . . . . .	1
1.2	Database Model Design . . . . .	1
1.2.1	Category Model . . . . .	2
1.2.2	Product Model . . . . .	2
1.2.3	Order Model . . . . .	2
1.2.4	OrderItem Model . . . . .	3
1.3	Database Migration . . . . .	3
1.4	API Integration . . . . .	3
1.5	Serializers in the API Application . . . . .	4
1.5.1	Product Serializer . . . . .	4
1.5.2	OrderItem Serializer . . . . .	4
1.5.3	Order Serializer . . . . .	4
1.5.4	Category Serializer . . . . .	5
1.6	Views in the API Application . . . . .	5
1.6.1	Product Views . . . . .	5
1.6.2	Order and OrderItem Views . . . . .	5
1.6.3	Category Views . . . . .	6
1.7	URLs in the API Application . . . . .	7
1.7.1	Default Router . . . . .	7
1.7.2	Additional Routes . . . . .	7
1.8	Django Admin Panel . . . . .	8
1.8.1	Customization . . . . .	8
1.9	Frontend Implementation with React . . . . .	8
1.9.1	Vite for Project Setup . . . . .	8
1.9.2	React Router for Navigation . . . . .	9
1.9.3	State Management with Zustand . . . . .	11
1.9.4	SCSS for Styling . . . . .	13
1.9.5	ApiService for Backend Communication . . . . .	13

# 1 Backend Implementation

## 1.1 Django Project and App Structure

The backend of our e-commerce platform is developed using **Django**, a high-level Python web framework that follows the Model-View-Template (MVT) architecture. The project is structured into two main applications:

- **backend**: Handles global settings and configurations for the project.
- **api**: Manages the REST API endpoints and facilitates communication with the frontend.

To set up the project, we initialized a Django project named `backend` and created the `api` application:

```
django-admin startproject backend
cd backend
python manage.py startapp api
```

Next, the required applications were registered in `settings.py` under the `INSTALLED_APPS` section:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rest_framework',
    'api',
    'corsheaders',
    'backend',
]
```

The `backend` app is responsible for managing general project settings and configurations, such as middleware, database connections, and authentication settings. Meanwhile, the `api` app is dedicated to handling RESTful communication by exposing endpoints for product management, order processing, and user authentication.

## 1.2 Database Model Design

The database schema was designed using Django's **Object-Relational Mapper (ORM)**, ensuring efficient interaction with the database while maintaining relational integrity. The primary models implemented in the `api` application include:

### 1.2.1 Category Model

The Category model organizes products into different categories. It consists of a single field:

```
class Category(models.Model):
    name = models.CharField(max_length=255)

    def __str__(self):
        return self.name
```

### 1.2.2 Product Model

Each product is stored in the Product model, which includes essential attributes such as name, description, price, stock quantity, and category association.

```
class Product(models.Model):
    name = models.CharField(max_length=255)
    description = models.TextField()
    price = models.FloatField()
    stock = models.IntegerField(default=0)
    category = models.ForeignKey(Category, on_delete=models.CASCADE)
    image = models.ImageField(upload_to='products/', null=True, blank=True)

    def __str__(self):
        return self.name
```

The category field establishes a **one-to-many relationship** with the Category model, ensuring that each product belongs to a specific category.

### 1.2.3 Order Model

The Order model represents customer transactions and includes information about the order status and payment method.

```
class Order(models.Model):
    orderNumber = models.CharField(max_length=100, unique=True)
    totalAmount = models.FloatField()
    status = models.CharField(
        max_length=20,
        choices=OrderStatus.choices,
        default=OrderStatus.PENDING
    )
    paymentMethod = models.CharField(
        max_length=20,
```

```

        choices=PaymentMethod.choices
    )

    def __str__(self):
        return self.orderNumber

```

The `status` and `paymentMethod` fields utilize Django's `TextChoices` feature, enforcing predefined values for order states (`PENDING`, `COMPLETED`, `CANCELLED`) and payment methods (`CREDIT_CARD`, `PAYPAL`, `BANK_TRANSFER`).

### 1.2.4 OrderItem Model

The `OrderItem` model defines the many-to-many relationship between `Order` and `Product`, specifying the quantity and price of each product within an order.

```

class OrderItem(models.Model):
    order = models.ForeignKey(Order, on_delete=models.CASCADE)
    product = models.ForeignKey(Product, on_delete=models.CASCADE)
    quantity = models.IntegerField()
    price = models.FloatField()

    def __str__(self):
        return f"{self.quantity} x {self.product.name} in Order {self.order.orderNumber}"

```

The `ForeignKey` relationships ensure that when an order is deleted, its associated items are also removed, maintaining referential integrity.

## 1.3 Database Migration

After defining the models, database migrations were created and applied using Django's migration framework:

```

python manage.py makemigrations api
python manage.py migrate

```

This step ensures that the defined models are translated into database tables.

## 1.4 API Integration

The `api` application exposes REST endpoints using the **Django REST Framework (DRF)**. This enables the frontend to interact with the backend through HTTP requests, facilitating CRUD operations on products, orders, and user authentication. The API design follows RESTful principles, ensuring a structured and scalable communication interface.

## 1.5 Serializers in the API Application

To facilitate data exchange between the backend and frontend, the api application utilizes **serializers** from the Django REST Framework (DRF). Serializers enable the transformation of complex Django model instances into JSON format and vice versa, ensuring efficient data transmission over RESTful APIs.

The implemented serializers correspond to the core models of the system: Product, Order, OrderItem, and Category. These serializers are defined as follows:

### 1.5.1 Product Serializer

The ProductSerializer is a ModelSerializer that converts Product model instances into JSON representations, including fields such as name, description, price, stock, and category.

```
class ProductSerializer(serializers.ModelSerializer):
    class Meta:
        model = Product
        fields = ['id', 'name', 'description', 'price', 'stock', 'category', 'image']
```

### 1.5.2 OrderItem Serializer

The OrderItemSerializer serializes individual items within an order. To provide detailed product information, it nests a ProductSerializer instance, allowing each OrderItem to include the associated product's attributes.

```
class OrderItemSerializer(serializers.ModelSerializer):
    product = ProductSerializer()

    class Meta:
        model = OrderItem
        fields = '__all__'
```

### 1.5.3 Order Serializer

The OrderSerializer manages the serialization of orders. Since an order consists of multiple order items, it integrates the OrderItemSerializer using the source parameter to reference the related OrderItem instances. The many=True argument ensures that multiple order items can be included in the response.

```
class OrderSerializer(serializers.ModelSerializer):
    order_items = OrderItemSerializer(source="orderitem_set", many=True, read_only=True)

    class Meta:
        model = Order
        fields = '__all__'
```

### 1.5.4 Category Serializer

The `CategorySerializer` handles the conversion of `Category` instances into JSON format, exposing all attributes of the `Category` model.

```
class CategorySerializer(serializers.ModelSerializer):
    class Meta:
        model = Category
        fields = '__all__'
```

## 1.6 Views in the API Application

In the api application, **views** are responsible for handling HTTP requests, processing data, and returning responses. The views are implemented using Django REST Framework's (DRF) `viewsets` and `generics`, which simplify the process of building RESTful APIs by automatically handling common actions like list, create, retrieve, update, and delete.

The main views in our application include views for handling products, orders, order items, and categories. These views are based on DRF's `ModelViewSet` and `ListCreateAPIView` classes, which provide functionality for interacting with the respective models.

### 1.6.1 Product Views

The `ProductListCreate` view allows clients to list all products and create new products. It uses the `ListCreateAPIView`, which provides both listing and creation functionality:

```
class ProductListCreate(generics.ListCreateAPIView):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
    parser_classes = (MultiPartParser, FormParser)
```

The `ProductViewSet` is a more advanced view that also provides the capability to search for products based on their name or description. This is achieved through the use of the `SearchFilter` from DRF:

```
class ProductViewSet(viewsets.ModelViewSet):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
    filter_backends = [filters.SearchFilter]
    search_fields = ['name', 'description']
```

### 1.6.2 Order and OrderItem Views

The `OrderViewSet` is responsible for handling orders, allowing for the listing, retrieval, updating, and deletion of orders. It also includes custom logic for creating orders. When an order is created, the related order items are processed and saved separately:

```

class OrderViewSet(viewsets.ModelViewSet):
    queryset = Order.objects.all()
    serializer_class = OrderSerializer

    def create(self, request, *args, **kwargs):
        data = request.data
        items = data.pop("items", [])
        order = Order.objects.create(**data)

        for item in items:
            OrderItem.objects.create(order=order, **item)

        return Response(OrderSerializer(order).data, status=status.HTTP_201_CREATED)

```

The `OrderItemViewSet` is similar to the `OrderViewSet` but operates specifically on individual order items:

```

class OrderItemViewSet(viewsets.ModelViewSet):
    queryset = OrderItem.objects.all()
    serializer_class = OrderItemSerializer

```

### 1.6.3 Category Views

The `CategoryViewSet` allows for the listing, creation, and management of categories, providing standard CRUD functionality through the `ModelViewSet`:

```

class CategoryViewSet(viewsets.ModelViewSet):
    queryset = Category.objects.all()
    serializer_class = CategorySerializer

```

Additionally, the `ProductListByCategory` view enables filtering products by category. It uses the `ListAPIView` class to return a list of products based on the specified `category_id`:

```

class ProductListByCategory(generics.ListAPIView):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
    filter_backends = (SearchFilter,)
    search_fields = ['category__id']

    def get_queryset(self):
        queryset = super().get_queryset()
        category_id = self.request.query_params.get('category_id', None)
        if category_id:
            queryset = queryset.filter(category__id=category_id)
        return queryset

```



## 1.7 URLs in the API Application

In the `api` application, the URLs define the routing of HTTP requests to specific views, enabling clients to interact with the backend. The routing is configured using Django's `path` function and DRF's `DefaultRouter`, which automatically generates routes for viewsets.

The main URLs are defined as follows:

### 1.7.1 Default Router

The `DefaultRouter` is used to register the viewsets for products, orders, order items, and categories. This automatically creates the necessary routes for standard CRUD operations:

```
router = DefaultRouter()
router.register(r'products', ProductViewSet)
router.register(r'orders', OrderViewSet)
router.register(r'orderitems', OrderItemViewSet)
router.register(r'categories', CategoryViewSet)
```

This results in the following routes:

- `/products/` for managing products,
- `/orders/` for managing orders,
- `/orderitems/` for managing order items,
- `/categories/` for managing categories.

### 1.7.2 Additional Routes

In addition to the automatically generated routes, custom URLs are defined for specific actions, such as listing products and searching for products by category. These routes are manually configured using the `path` function:

```
urlpatterns = [
    path('', include(router.urls)),
    path('products-list/', ProductListCreate.as_view(), name='product-list'),
    path('products/search/', views.ProductListByCategory.as_view(), name='product-search')
]
```

The `products-list/` route is mapped to the `ProductListCreate` view, which handles both listing and creating products, while the `products/search/` route maps to the `ProductListByCategory` view, which enables filtering products by category.

## 1.8 Django Admin Panel

The Django Admin Panel provides a powerful and customizable interface for managing the application's data. It allows administrators to interact with the models directly, facilitating tasks such as adding, editing, and deleting records without requiring direct database access.

In the `api` application, the admin panel is configured to manage the core models: `Product`, `Order`, `OrderItem`, and `Category`. The following configuration registers these models with the Django admin interface:

```
from django.contrib import admin
from .models import Product, Order, OrderItem, Category

admin.site.register(Product)
admin.site.register(Order)
admin.site.register(OrderItem)
admin.site.register(Category)
```

By registering the models, they are made available in the Django Admin Panel, allowing administrators to view and modify their data through a user-friendly interface. The admin panel automatically provides basic CRUD operations for each registered model, ensuring ease of use for the management of products, orders, and categories.

### 1.8.1 Customization

While the default admin interface is functional, it can be further customized by adding custom admin classes to modify the way models are displayed or edited. For example, fields can be arranged in a more intuitive layout, or search functionality can be added to improve usability. In this application, however, the default configuration suffices for basic administrative tasks.

## 1.9 Frontend Implementation with React

The frontend of the web application is built using the **React** library, which enables the development of dynamic and interactive user interfaces. Several tools and libraries were integrated to support the development process and ensure the scalability and maintainability of the application.

### 1.9.1 Vite for Project Setup

For the project setup, `Vite` was used as the build tool. `Vite` is a modern and efficient build tool that provides fast development cycles and efficient bundling. It is optimized for modern JavaScript frameworks and requires minimal configuration.

The `Vite` application was created using the following steps. First, the `Vite` project was initialized using the `npm create` command, specifying the `React` template with `TypeScript` support:

```
npm create vite@latest frontend --template react-ts
```

This command initializes a new React project named `frontend` with TypeScript support and the necessary configurations. The `react-ts` template provides an optimized setup for using React with TypeScript, allowing for type safety and better developer tooling throughout the project.

Once the project was created, the required dependencies were installed using `npm`:

```
cd frontend
npm install
```

Some of the key packages installed for the project include:

- `react` and `react-dom` for the React library and rendering React components in the DOM.
- `react-router-dom` for handling client-side routing and navigation within the application.
- `zustand` for state management, enabling global state sharing across components.
- `sass` to support SCSS, allowing for modular and maintainable styling.
- `typescript` to enable static type checking, improving code quality and providing better developer tooling.

After the installation, the development server can be started using the following command:

```
npm run dev
```

This command starts the Vite development server, which supports features such as fast hot module replacement (HMR), providing a responsive development environment. The application is now ready to be developed and tested locally.

Vite was chosen for its high performance, ease of use, and its ability to efficiently handle modern frontend frameworks like React with TypeScript.

### 1.9.2 React Router for Navigation

For routing within the application, `React Router` was implemented. `React Router` facilitates client-side navigation, allowing the application to load different views without requiring full page reloads. It supports both dynamic and static routes, enabling flexible management of URLs and components. By using `React Router`, the application can dynamically render different components based on the active route, ensuring a seamless user experience.

The routing setup is structured as follows:

```
<BrowserRouter>
  <Routes>
    <Route path="/" element={<Layout />}>
      <Route path="/Shop" element={<Shop />} />
      <Route path="/Cart" element={<Cart />} />
      <Route path="/Admin" element={<Admin />} />
      <Route path="checkout" element={<Checkout />} />
    </Route>
  </Routes>
</BrowserRouter>
```

```

    </Route>
  </Routes>
</BrowserRouter>

```

The `BrowserRouter` component wraps the entire routing structure, utilizing the HTML5 History API to manage URL changes without refreshing the page. Inside the `Routes` component, multiple `Route` elements define paths and their corresponding components. The main `Layout` component serves as a wrapper for all pages, ensuring a consistent layout structure across the application. The `Outlet` component within `Layout` is used to dynamically render the active route's component.

The `useNavigate` hook is used to programmatically navigate between routes. It allows for navigation based on specific events, such as button clicks or after certain conditions are met. An example usage of `useNavigate` is as follows:

```

const navigate = useNavigate();
navigate('/Shop');

```

This hook enables the developer to navigate to specific routes programmatically, enhancing the interactivity of the application.

The `Layout` component itself serves as the main structure for all the pages, ensuring that elements such as the top navigation bar, side drawer, modals, and notifications are consistent across all views. The `Layout` component is defined as follows:

```

const Layout: React.FC = () => {
  return (
    <div className={styles.root}>
      <TopBar isAdmin={true} />
      <Drawer />
      <div className={styles.content_container}>
        <Outlet />
      </div>
      <Modal />
      <Toast />
    </div>
  );
};

```

The `Layout` component includes the following key elements:

- **TopBar:** Displays a top navigation bar with the option to toggle administrative features.
- **Drawer:** A side navigation component that allows users to switch between various sections of the application.
- **content\_container:** A container where the content of the active route is displayed. The `Outlet` component is used to render the currently active route's component.
- **Modal:** A component for displaying modal dialogs, such as confirmation messages or forms.

- **Toast:** A notification component that provides brief feedback, such as success or error messages.

This setup ensures that the layout remains consistent across all pages while the content dynamically changes based on the active route. The combination of React Router for navigation and the Layout component for consistent structure allows for an efficient and user-friendly frontend application.

### 1.9.3 State Management with Zustand

For state management in the frontend, the application utilizes Zustand, a small, fast, and scalable state management library for React. Zustand provides a straightforward way to manage global state in the application, allowing various components to share and modify state without the need for prop drilling.

In the implementation, a custom store is created using `create` from Zustand, which defines the state and actions required by the application. The store maintains several pieces of state, including the list of products, the cart items, and categories. It also includes loading and error states to handle asynchronous requests.

The store is defined as follows:

```
import { create } from "zustand";
import { addToCart } from "../actions/addToCart";
import { removeFromCart } from "../actions/removeFromCart";
import { clearCart } from "../actions/clearCart";
import { incrementQuantity } from "../actions/incrementQuantity";
import { decrementQuantity } from "../actions/decrementQuantity";
import ApiService from "../../api/ApiService";

interface StoreState {
  products: Product[];
  cart: CartItem[];
  categories: Category[];
  isLoadingProducts: boolean;
  isLoadingCategories: boolean;
  error: string | null;
  fetchProducts: () => Promise<void>;
  fetchCategories: () => Promise<void>;
  addToCart: (productId: number, quantity?: number) => void;
  removeFromCart: (productId: number) => void;
  incrementQuantity: (productId: number, incrementBy?: number) => void;
  decrementQuantity: (productId: number, decrementBy?: number) => void;
  clearCart: () => void;
}

export const useStore = create<StoreState>((set) => ({
  // Initial state values
```

```

products: [],
cart: [],
categories: [],
isLoadingProducts: false,
isLoadingCategories: false,
error: null,

// Fetch products and categories asynchronously
fetchProducts: async () => {
  set({ isLoadingProducts: true, error: null });
  try {
    const data = await ApiService.get("/products");
    const products = data as Product[];
    set({ products, isLoadingProducts: false });
  } catch (error) {
    set({ error: error instanceof Error ? error.message : "An unknown error occurred",
  }
},

fetchCategories: async () => {
  set({ isLoadingCategories: true, error: null });
  try {
    const data = await ApiService.get("/categories");
    const categories = data as Category[];
    set({ categories, isLoadingCategories: false });
  } catch (error) {
    set({ error: error instanceof Error ? error.message : "An unknown error occurred",
  }
},

// Cart manipulation actions
addToCart: (productId, quantity = 1) =>
  set((state) => ({
    cart: addToCart(state.cart, productId, quantity), // Action for adding to the cart
  })),

removeFromCart: (productId) =>
  set((state) => ({
    cart: removeFromCart(state.cart, productId), // Action for removing from the cart
  })),

incrementQuantity: (productId, incrementBy = 1) =>
  set((state) => ({
    cart: incrementQuantity(state.cart, productId, incrementBy), // Action for incrementing
  })),

```

```

decrementQuantity: (productId, decrementBy = 1) =>
  set((state) => ({
    cart: decrementQuantity(state.cart, productId, decrementBy), // Action for decrem
  })),

  clearCart: () => set({ cart: clearCart() }), // Action for clearing the cart
});

```

In this implementation, each action that modifies the cart state, such as adding, removing, or changing the quantity of items, is handled by separate functions that are imported from the actions file. These functions, such as `addToCart`, `removeFromCart`, and `incrementQuantity`, encapsulate the logic for each operation, keeping the store itself clean and focused on state management.

By offloading the business logic to separate action functions, the store becomes more maintainable and testable, and the components remain focused on rendering the UI.

This structure allows for centralized management of the global state, ensuring that components that need access to shared state, such as the product list or the cart, can easily access and modify the state without unnecessary complexity.

#### 1.9.4 SCSS for Styling

Styling is managed using SCSS, a superset of CSS that includes features such as variables, nesting, and mixins. SCSS facilitates more structured and maintainable stylesheets. To avoid conflicts and ensure modularity, CSS Modules are used in combination with SCSS. This approach scopes styles locally to the components, preventing global style leaks and ensuring component-specific styling.

#### 1.9.5 ApiService for Backend Communication

To facilitate communication between the frontend and the backend, an `ApiService` class is implemented. This service acts as a wrapper around the `fetch` API, allowing the frontend to send HTTP requests to the backend and handle responses in a consistent manner.

The `ApiService` class defines static methods for sending GET, POST, PUT, and DELETE requests to the backend. The base URL and port for the API are dynamically configured through environment variables, ensuring that the application can easily switch between different environments, such as development and production. In case no environment variables are set, default values are provided.

The class is structured as follows:

```

class ApiService {
  private static baseUrl: string = import.meta.env.VITE_API_BASE_URL || "http://localhost";
  private static port: string = import.meta.env.VITE_API_PORT || "3000";

  private static get fullUrl(): string {
    return `${this.baseUrl}:${this.port}/api`;
  }
}

```

```

private static async handleResponse(response: Response): Promise<unknown> {
  if (!response.ok) {
    const errorText = await response.text();
    throw new Error('API-Error: ${response.status} - ${errorText}');
  }
  return response.json();
}

public static async get(endpoint: string, headers: Headers = {}): Promise<unknown> {
  try {
    const response = await fetch(`${this.fullUrl}${endpoint}`, {
      method: "GET",
      headers: {
        "Content-Type": "application/json",
        ...headers,
      },
    });
    return await this.handleResponse(response);
  } catch (error) {
    console.error("GET Request Error:", error);
    throw error;
  }
}

public static async post(endpoint: string, body: Record<string, unknown>, headers: Headers = {}): Promise<unknown> {
  try {
    const response = await fetch(`${this.fullUrl}${endpoint}`, {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
        ...headers,
      },
      body: JSON.stringify(body),
    });
    return await this.handleResponse(response);
  } catch (error) {
    console.error("POST Request Error:", error);
    throw error;
  }
}

public static async put(endpoint: string, body: Record<string, unknown>, headers: Headers = {}): Promise<unknown> {
  try {
    const response = await fetch(`${this.fullUrl}${endpoint}`, {
      method: "PUT",

```



```

        headers: {
            "Content-Type": "application/json",
            ...headers,
        },
        body: JSON.stringify(body),
    });
    return await this.handleResponse(response);
} catch (error) {
    console.error("PUT Request Error:", error);
    throw error;
}
}

public static async delete(endpoint: string, headers: Headers = {}): Promise<unknown>
    try {
        const response = await fetch(`${this.fullUrl}${endpoint}`, {
            method: "DELETE",
            headers: {
                "Content-Type": "application/json",
                ...headers,
            },
        });
        return await this.handleResponse(response);
    } catch (error) {
        console.error("DELETE Request Error:", error);
        throw error;
    }
}
}

```

The `ApiService` class provides a consistent and reusable way to interact with the backend by making API calls. It handles both the sending of requests and the processing of responses. The methods check the response status and, if the request was successful, parse the JSON body; otherwise, an error is thrown with the appropriate status code and message.

This service centralizes all API calls, making it easier to manage and maintain the communication layer of the application. It abstracts the complexities of the `fetch` API and simplifies error handling and response processing.

By using this service, the frontend components can remain focused on their UI responsibilities, while the `ApiService` manages the interaction with the backend.