Documentation

Introduction:

In this shop, customers can browse and purchase a variety of products from the comfort of their own home. The website is designed to be easy to use and navigate, with a user-friendly interface that makes it simple to find and select the products they want.

As a new **customer**, you can create an account and start browsing the catalog of products right away. You can also log in daily to collect coins, which can be used to buy products. There will also be daily questions/puzzles and if you answer them correctly, you'll earn coins as a reward.

In addition to browsing and purchasing products, customers can also track their daily check-ins and rewards on the website. If you have any questions or issues, you can contact the customer service team for assistance.

As an **admin**, you can log in to manage the products and orders on the website. You can add new products and edit existing ones, as well as view customer orders and track inventory levels. You can also see the tokens for daily check-ins and edit them if necessary.

Architecture:

The architecture of this project is based on the Java Spring framework (Spring Boot) and utilizes Maven for dependency management. The back-end is built using Spring Boot, which provides a robust framework for building web applications.

The application uses PostgreSQL as the database management system, which is a popular and widely-used open-source database software. The back-end is designed to be highly scalable and fault-tolerant, with the ability to handle high traffic and large amounts of data.

The front-end is built using React, which is a popular JavaScript library for building user interfaces. We also use Fuse React Admin Template, which is an open-source admin template built with React and one of its key features is customizable architecture which helps easily customize the template's layout, theme, and functionality to match your specific needs.

Overall, the architecture of this project is designed to provide a robust and scalable solution for an online shop, with a focus on performance and ease of use.

Components:

Application-properties:

A popular library for controlling connections to databases in Java applications is the HikariCP connection pool. It offers a very flexible, fault-tolerant, and high-performance connection pool for managing connections to several databases, including PostgreSQL.

The spring.datasource.hikari.connection-timeout property is used to configure the maximum time (in milliseconds) that the service will wait for a connection to the database when using the HikariCP connection pool. We have set this parameter to 20000 (20 seconds), this can help to prevent the application from becoming unresponsive or crashing due to a lack of database connections.

Also, we are using spring.datasource.hikari.maximum-pool-size to configure the maximum number of concurrent connections that the service will allow to the database when using the HikariCP connection pool. This is set to 5 to prevent the application from overwhelming the database with too many concurrent connections, which can lead to performance degradation, timeouts, and other issues.

We have configured our URL for the PostgreSQL database to run on localhost on port 5432 and set the name of the database to „emt“ and the username/password for the connection to the database set to „postgres“ for every service. For server ports we are using 9090-9094.

Daily check-ins:

We have RESTful API for managing daily check-ins. The DailyCheckInController class contains endpoints for retrieving, creating, updating and deleting daily check-ins. The DailyCheckInService class performs the actual CRUD operations for daily check-ins. The DailyCheckIn class is the model class for daily check-ins, and DailyCheckInDTO and DailyCheckInCreationDTO are data transfer objects for daily check-ins. Finally, the DailyCheckInConverter class is used to convert between the model and DTO representations of daily check-ins.

The application also uses Lombok to generate getters, setters, equals, hashCode, and toString methods for the classes. We also use final variables and null-checking for null references.

Orders:

The Orders Management application is designed to manage orders, products within orders, and user interactions with their shopping cart. It allows for operations such as creating and editing orders, adding and removing products from a user's cart, and viewing order details, all of which are accessible via HTTP REST endpoints. Orders are tracked with a unique order ID and a tracking number for logistics purposes. Users have a credit balance that is verified when creating orders to ensure sufficient funds are available.

Key Components:

➢ **Main Application Class (OrdersManagementApplication.java):**

- Initializes the Spring Boot application.
- Scans components from'mk.ukim.finki.ordersmanagement','mk.ukim.finki.usersmanagement', and'mk.ukim.finki.productsmanagement'.

➢ **Controllers:**

- **OrderedProductController:**
  - Endpoints to manage activities related to products that users have ordered.
  - Includes operations to add and remove products from shopping cart and listing all ordered products.
- **OrderController:**
  - Endpoints to manage orders, such as creation, update, deletion, view, and status updates (cancel/confirm).

➢ **Services:**

- **OrderService:**
  - Contains business logic for order creation, edits, and status updates.
  - Manages financial validations and updates the user's credit balance.
- **OrderedProductService:**
  - Responsible for adding and removing products from the shopping cart.
  - Manages queries related to ordered products by user or order.

➢ **Repositories:**

- **OrderRepository:**
  - For accessing the database and performing operations related to order entities.
- **OrderOrderedProductRepository:**
  - Manages the relation between an order and the products ordered within it.
- **OrderedProductRepository:**
  - Handles access to the OrderedProduct entities for operations like finding by user or product.

➢ **Entities:**

- **OrderOrderedProduct:**
  - Entity representing the association between Orders and Ordered Products.
- **OrderedProduct:**
  - Entity representing a product that has been ordered, with properties like price and quantity.

➢ **Value Objects:**

- **OrderId:**
  - A wrapper class to represent a unique identifier for an Order.
- **OrderedProductId:**
  - A unique identifier for an OrderedProduct.

➢ **Enums:**

- **OrderStatus:**
  - Represents the status of an order (RECEIVED, SHIPPED, CANCELLED).

➢ **Exceptions:**

- **InsufficientCreditException:**
  - An exception that is thrown when a user does not have enough credit to complete a purchase. (We use this for enhanced error handling)

➢ **DTOs:**

- **OrderedProductDTO:**
  - Data transfer object for carrying ordered product data between processes.
- **OrderDTO:**

        - Carries data related to an order.
    - **OrderCreationDTO:**
        - DTO used when creating a new order.

➢ **Converters:**

- **OrderedProductConverter:**
    - Converts OrderedProduct entities to DTOs and vice versa.
- **OrderConverter:**
    - Converts Order entities to DTOs.

Products:

The products management feature, designed to manage the lifecycle and attributes of products within the system. The feature allows users to view all products, search with pagination, find specific products by ID, create new products, update existing products, and delete products. It follows a common Spring Boot pattern utilizing controllers for HTTP handling, services for business logic, repositories for data access, entities for database mapping, and data transfer objects (DTOs) for client communication.

Key Components

➢ **ProductController**: Handles incoming HTTP requests at the "/products" root URL. It uses the ProductService and ProductConverter to execute operations and convert entities to DTOs needed for the client interface.

- findAll(): Returns a list of all product DTOs.
- findAllPaged(): Returns a paginated response of product DTOs based on given filters.
- findById(): Returns a single product DTO for the given product ID, or throws an exception if not found.
- create(): Creates a new product and returns its DTO.
- edit(): Updates an existing product and returns the updated DTO.
- delete(): Deletes a product based on the provided ID.

➢ **ProductService**: Encapsulates the business logic for product-related operations, interacting with the ProductRepository for data persistence.

- FindAll(), findAllPaged(), findById(): Wrappers for repository calls adding additional service-layer operations.
- Create() and edit(): These methods handle the creation and updating of products. They assign creation/modification timestamps and save the

product entity.
    - fillProperties(): Populates a product entity based on creation DTO fields and decodes and stores the base64 image.
    - delete(): Removes a product by ID.

➢ **ProductRepository**: Extends JpaRepository and provides a data access layer for the Product entity. Includes custom query methods like findAllPaged() which allows filtering based on various criteria.

➢ **Product**: The entity representing a product with fields such as dateCreated, dateModified, quantity, price, mage, title, description, and category. It inherits an ID from AbstractEntity.

➢ **Category**: An enumeration representing product categories like BOOKS, MOVIES, MUSIC.

➢ **ProductFilter**: A DTO holding filter criteria for searching products.

➢ **ProductDTO**: Contains a subset of Product entity fields, tailored for data transfer.

➢ **ProductCreationDTO**: Defines the necessary fields for creating or updating a product.

➢ **ProductConverter**: A component responsible for converting between Product entities and ProductDTO objects.

Users:

The users management includes API endpoints, services, domain models, data transfer objects (DTOs), converters, exceptions, and identifiers, and integrates Spring Security for user authentication.

Key Components

➢ **UsersManagementApplication.java:** Works as the entry point for the Spring Boot Application. It enables scheduling and sets the base package for component scanning which includes packages for user management and daily check-ins management functionalities.

Controllers

➢ **UserController.java**: Manages user-related actions such as CRUD operations and querying users based on filters or paginated methods. It also handles users' daily check-ins.

➢ **RegisterController.java**: Provides an endpoint for user registration.
➢ **LogoutController.java**: Provides a simple endpoint for user logout, invalidating the current user session.

Services

➢ **UserService.java**: Delivers core business logic for user management including registration, searching, creating, editing, deleting users, and handling daily check-ins.

➢ **UserDailyCheckInsService.java**: Manages daily check-ins for users, allowing binding the check-ins to users, claiming, and resetting check-ins.

➢ **RoleService.java**: Handles role-related operations, such as finding and creating roles, including role privileges.

➢ **RolePrivilegeService.java**: Specifies the operations for managing role privileges.

➢ **PrivilegeService.java**: Manages the creation and retrieval of privileges.

Quiz:

The core functionality of this application encompasses managing quiz questions and answers, including CRUD operations, exception handling for not found entities, and converting between entities and DTOs to interact with a front-end client conveniently.
Transactional operations ensure the integrity of the persistent states during operations like creating and editing quiz content.

- ➢ **QuizQuestionController.java**: A REST controller that handles HTTP requests related to quiz questions. Offers endpoints to list, find, create, update, and delete quiz questions. Utilizes QuizQuestionService and QuizQuestionConverter to process business logic and convert entities to DTOs respectively.

➢ **QuizAnswerController.java**: A REST controller stub for quiz answers; currently associates with QuizAnswerService.

➢ **QuizQuestionService.java**: Handles the business logic for managing quiz questions. Includes finding, creating, editing, deleting questions, and submitting of quiz results, updating users' scores accordingly.

➢ **QuizQuestionAnswerService.java**: Manages the association between a quiz question and its answers, including get-or-create logic for quiz question answers.

➢ **QuizAnswerService.java**: Provides services to find, create, and get-or-create quiz answers based on provided DTOs.

Repositories:

QuizQuestionRepository.java, QuizAnswerRepository.java, QuizQuestionAnswerRepository.java: Interfaces extending JpaRepository to provide CRUD operations on the database for quiz questions, quiz answers, and their associations.

Entity Classes:

QuizQuestion.java, QuizAnswer.java, and QuizQuestionAnswer.java: JPA Entities representing quiz questions, answers, and the relationship between a question and its answer. They are annotated with @Entity and mapped to database tables.

ID Classes:

QuizQuestionId.java, QuizQuestionAnswerId.java, and QuizAnswerId.java: Wrapper classes that extend DomainObjectId to act as unique identifiers for their respective entities.

Exception Class:

QuizQuestionNotFoundException.java: An exception thrown when a quiz question does not exist in the repository.

Enumerations:

Topic.java and Difficulty.java: Enums defining possible values for topics and difficulty levels of quiz questions.

DTO Classes:

Data Transfer Objects (DTOs) like QuizQuestionDTO and QuizAnswerDTO, are used for transferring data between processes, typically from the service layer to the controller.

Converters:

QuizQuestionConverter.java and QuizAnswerConverter.java: These components convert the quiz entities to DTOs and vice versa, facilitating the separation of internal entity representation from the API exposed model.

Shared Kernel:

The shared-kernel module's primary goal is to provide a set of common utilities and base types that ensure consistency and reuse across different parts of a complex

application, following best practices in DDD for forming a language and sharing kernel across various bounded contexts.

➢ **StringUtils.java**: A utility class that provides string manipulation methods, such as converting a string to lower case only if it's not null and returning null for empty or null strings.

➢ **NullableUtils.java**: A utility class featuring methods to safely perform operations on objects that may be null. It includes functions to execute a transformation if an object is not null, supply an alternative if it is, or return an alternative direct value if the object is null.

➢ **ValueObject.java**: An interface marker for value objects in the domain model, which are objects that don't have a conceptual identity.

➢ **DomainObjectId.java**: A base class for domain object identifiers. It encapsulates a unique id string, likely meant to be used as a primary key in persistence. The class is annotated with @Embeddable, suggesting the ID can be embedded in entity classes.

➢ **DomainObject.java**: A basic interface to mark classes as domain objects, making them serializable.

➢ **AbstractEntity.java**: A generic abstract class that all domain entities can extend. It requires an identifier of type DomainObjectId. This base class ensures that all entities have an ID and aids in implementing DDD principles and reinforcing object identity within the application.

Created by:

- **Bojan Bozhilov**
- **Marko Ichev**