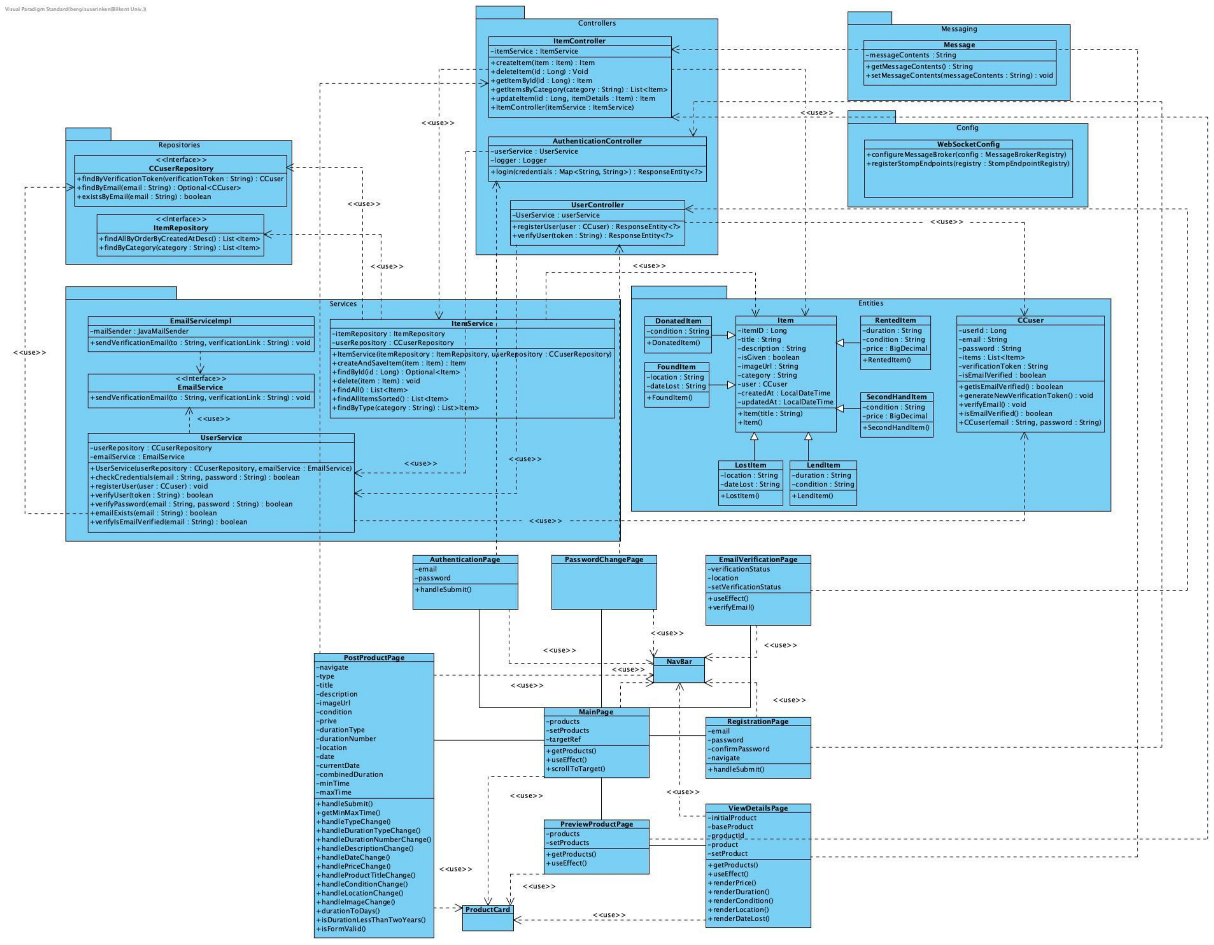
D5 – Solution Level Class Diagram and Design Patterns of CampusConnect
Team 09 – 01
December 3, 2023

1. Solution Level Class Diagram of CampusConnect



2. Design Patterns of CampusConnect

2.1. Facade – Structural Design Pattern

In CampusConnect, the ‘ItemService’, ‘UserService’, and ‘EmailService’ effectively act as facade classes, encapsulating complex business logic and data interactions behind simpler interfaces. This implementation aligns with the Facade design pattern.

- <u>ItemService:</u> This service simplifies interactions with `ItemRepository` and possibly ‘CCuserRepository’, offering straightforward methods like ‘createAndSaveItem’, ‘findById’, and ‘findAllItemsSorted’. It hides the complexities of item management (like database queries and business rules) from the controllers.

- <u>UserService:</u> This service handles user-related operations by interfacing with ‘CCuserRepository’. Complexities such as user registration, credential validation, and email verification are abstracted away. Controllers utilize easy-to-understand methods like ‘registerUser’, ‘checkCredentials’, and ‘verifyUser’, ensuring a clear separation from the underlying detailed logic.

- EmailService (‘EmailServiceImpl’): It provides a simplified way to send emails, wrapping the functionality of ‘JavaMailSender’. The service offers methods like ‘sendVerificationEmail’, abstracting the intricacies of email configurations and operations from the controllers.

By using these services as facades, we have achieved a cleaner, more maintainable architecture in the CampusConnect application. They reduce the controller layer's complexity and improve the system's overall organization by clearly separating concerns.

2.2. Singleton – Creational Design Pattern

In our CampusConnect application, we've strategically used Spring's annotations to enforce the Singleton design pattern for our service, repository, and controller components, bringing significant benefits:

- Service Classes (EmailService, UserService, ItemService, MessagingService): When we annotate these classes with @Service, Spring manages them as singleton beans. This means that regardless of how many times these services are injected or used across the application - in controllers, other services, or utility classes - Spring ensures that only one instance of each service exists. This is particularly useful for maintaining a consistent state and shared resources (like database connections or configuration settings). For example, the EmailService handles all email operations consistently throughout the application, ensuring that there is no duplication or conflict in email handling logic.

- Repository Classes (ItemRepository, UserRepository): Annotated with @Repository, these classes are also treated as singletons. They provide an abstraction layer over the data access logic, allowing our services to interact with the database in a consistent and centralized manner. The singleton nature here ensures that there's a single point of interaction with the database tables related to items and users, which aids in maintaining the integrity and consistency of our data access operations.

- Controller Classes (ItemController, UserController, AuthenticationController): These classes, typically annotated with @RestController, are controllers that handle HTTP requests. Spring treats them as singletons, creating only one instance of each controller. This is efficient since controllers are stateless by nature and a single instance can handle all incoming requests. It also simplifies the routing and handling of HTTP requests as each controller is a unique endpoint in our application.

In our CampusConnect application, the implementation of singleton services and repositories through Spring's annotations offers significant advantages. Firstly, it ensures consistent state management across the application, which is crucial for maintaining data integrity and providing consistent user experiences. This uniformity is especially important given the stateful nature of services like `UserService`, which handle sensitive user data. Secondly, the singleton approach greatly enhances efficient resource management. By avoiding the need for multiple instances of each component, we significantly reduce resource consumption, thereby boosting the overall performance of the application. Lastly, the ease of managing a single instance of each component simplifies both configuration and maintenance. Changes made to a singleton are uniformly reflected throughout the application, streamlining updates and ensuring consistency in application behavior. These benefits collectively contribute to making CampusConnect not only more efficient and reliable but also easier to manage and scale.

2.3. Repository – Architectural Design Pattern

In our CampusConnect application, we have embraced the Repository design pattern to efficiently manage data access. This pattern is crucial for separating the data access logic from the business logic, ensuring a clean separation of concerns. By implementing repositories such as ‘ItemRepository’ and ‘UserRepository’, we abstract the complexities of direct database interactions. This abstraction not only simplifies querying and manipulating data but also enhances maintainability and consistency across the application. The repositories provide a unified interface for data operations like ‘findAll()’, ‘findById()’, and ‘save()’, allowing other parts of the application to interact with the data layer without needing to know about the underlying database details. This decoupling also greatly aids in testing, as it allows for easy mocking of the data layer. Additionally, the integration with Spring Data JPA streamlines the implementation, reducing boilerplate code and increasing efficiency. Overall, the Repository design pattern plays a pivotal role in maintaining clean, testable, and maintainable code in CampusConnect.

2.4. Model View Controller – Architectural Design Pattern

In our CampusConnect application, we have adopted the Model View Controller (MVC) design pattern to effectively structure and manage the application. This pattern segregates the application into three interconnected components, enhancing maintainability and scalability. The ‘Model’ components, represented by our domain entities like ‘Item’ and ‘User’, encapsulates the application's data and business logic. The ‘View’ is handled by the frontend, built with React, presenting data to users and enabling interaction. The ‘Controller’, realized through classes like ‘ItemController’ and ‘UserController’, acts as an intermediary, handling incoming requests from the view, manipulating the model, and sending back data or responses. This separation of concerns allows for cleaner, more organized code, easier debugging and testing, and the flexibility to evolve the frontend and backend independently, making MVC an ideal choice for the structure of CampusConnect. MVC works in alignment with the Repository design pattern: MVC organizes the flow and processing of data, while the Repository deals with how data is fetched and stored.