DreamCatch

Student:Bode Andreea-Nicoleta

**Group:30238**

Table of Contents

1. Requirements Analysis 3

1.1 Assignment Specification 3

1.2 Functional Requirements 3

1.3 Non-functional Requirements 3

2. Use-Case Model 4

3. System Architectural Design 5

4. UML Sequence Diagrams 8

5. Class Design 9

6. Data Model 10

7. System Testing 11

8. Bibliography 11

1. Requirements Analysis

# Assignment Specification

DreamCatch is a comprehensive sleep monitoring application that enables users to track the quality of their sleep over time. With DreamCatch, users can create custom categories and add entries with a short description and a set of tags that relate to their dreams. They can then evaluate the quality of their sleep based on three metrics: duration, energy level, and stress, which are scored on a scale of 1-5 for easy tracking.

The app allows users to aggregate these metrics into different categories and displays graphs for the current month and week for each quality measure when requested. In future deployments, the app will be able to monitor vitals through smart devices.

DreamCatch is designed with a layered architecture and uses an ORM and DI container. The data is stored in a database, and all user inputs are validated against invalid data before submission and saving in the database.

With its user-friendly interface, customizable categories, and comprehensive tracking metrics, DreamCatch is the perfect application for anyone looking to improve their sleep quality and track their dreams over time.

Within the application, the charts can be viewed in 2 colors depending on the average. Thus, if the average duration is greater than 4, then the chart will be colored green, otherwise it will be colored red. For energy, the chart will be colored green if the average is greater than 3, otherwise it will be red. For stress, if the average is greater than 2 it will be colored red, otherwise it will be green.

# Functional Requirements

Users can record their sleep by adding an entry with a short description and a set of tags that define the type of dream. Users can rate their sleep quality using three parameters: duration, energy level and stress. These values are recorded on a scale of 1 to 5 for easy tracking.

The app will allow users to select tag categories to better organize their sleep entries.

The application should be able to aggregate sleep quality values into different label categories to present charts from the current week and month for each quality measure, given that the user enters a requested category.

All application inputs will be validated against invalid data before submitting the data and saving it to the database.

The application will use the factory model for creating various reports/charts based on metrics. These reports should be easily accessible and easy to understand for the user.

The user interface should be intuitive and easy to navigate, providing a smooth and enjoyable user experience.

The application must ensure the privacy and security of user data, protecting against unauthorized access and data breaches.

It is required to use the Decorator Pattern for coloring the graphics. It is requested to implement the CQRS and Mediator Pattern architecture.

# Non-functional Requirements

* Performance

The application should be able to handle a large volume of data efficiently and respond to user requests in a timely manner, even during peak usage times.

* Security

The application should protect user data from unauthorized access, both in transit and at rest. It should also follow industry-standard security practices for data storage and transmission.

* Scalability

The system should be able to handle increasing amounts of data and user traffic as the user base grows. It should be designed to be scalable without significant downtime or disruption to users.

* Usability

The application should be easy to use and navigate for all types of users, regardless of technical skill level. It should also be accessible to users with disabilities.

* Reliability

The application should be highly available and reliable, with minimal downtime or disruptions. It should also have a robust backup and recovery system in place in case of data loss or system failure.

* Maintainability

The application should be easy to maintain and update, with clear and well-documented code. It should also be easy to troubleshoot and fix issues that arise.

* Compatibility

The application should be compatible with different operating systems and devices, including mobile and desktop platforms.

* Data Integrity

The application should ensure the accuracy and integrity of user data, with appropriate measures in place to prevent data corruption or loss.

* Privacy

The application should be compliant with applicable privacy laws and regulations, with appropriate measures in place to protect user privacy and data confidentiality.

2. Use-Case Model

O imagine care conține text, diagramă, linie, cerc

Descriere generată automat

**Use**-**Case**: Create dream entry

**Level**: User-goal level

**Primary** **actor**: User website

The user can select a dream type from the dropbox, add a description, enter values in the range [1,5] for energy, duration and stress. By pressing the Submit button, the data will be saved in the database. By pressing the ViewUsers button, they can see a table of all the data saved in the database, and by pressing the Charts button, they can see the 3 charts by date and stress, duration or energy. These charts are available for the week and for the current month. With the help of the drop down on the left side, choose "Weekly" if you want to display the graphs from the current week, respectively "Monthly" if you want to view the graphs from the current month.

3. System Architectural Design

**3.1 Architectural Pattern Description**

I used layer architecture. I have more packages than those captured in layer architecture, they can still be added to other packages to fully respect the architecture. I used Java and Spring, which is why my application is structured in packages: entity, controller, service, repository.

O imagine care conține text, diagramă, captură de ecran, Font

Descriere generată automat

The CQRS pattern is a design pattern that separates the responsibility of handling commands (write operations) and queries (read operations) into separate components, each with its own data model.

Command Pattern: The ICommand interface and its implementation classes (SaveDreamCommand and GetDreamCommand) encapsulate write operations as commands that are executed by the Mediator class. This is in line with the CQRS pattern, which separates write operations into commands.

Mediator Pattern: The IMediator interface and its implementation class (Mediator) are used to decouple the command handlers (SaveDreamCommand and GetDreamCommand) from the client code. This allows for the handling of commands to be centralized, thus simplifying the code and promoting modularity

Repository: The DreamRepository interface is used to provide a layer of abstraction over the persistence layer. This promotes separation of concerns by separating the data access code from the business logic.

DTO: The DreamDto class is used to transfer data between the various components of the application. This promotes loose coupling between components and facilitates the transformation of data from one form to another.

**3.2 Diagrams**

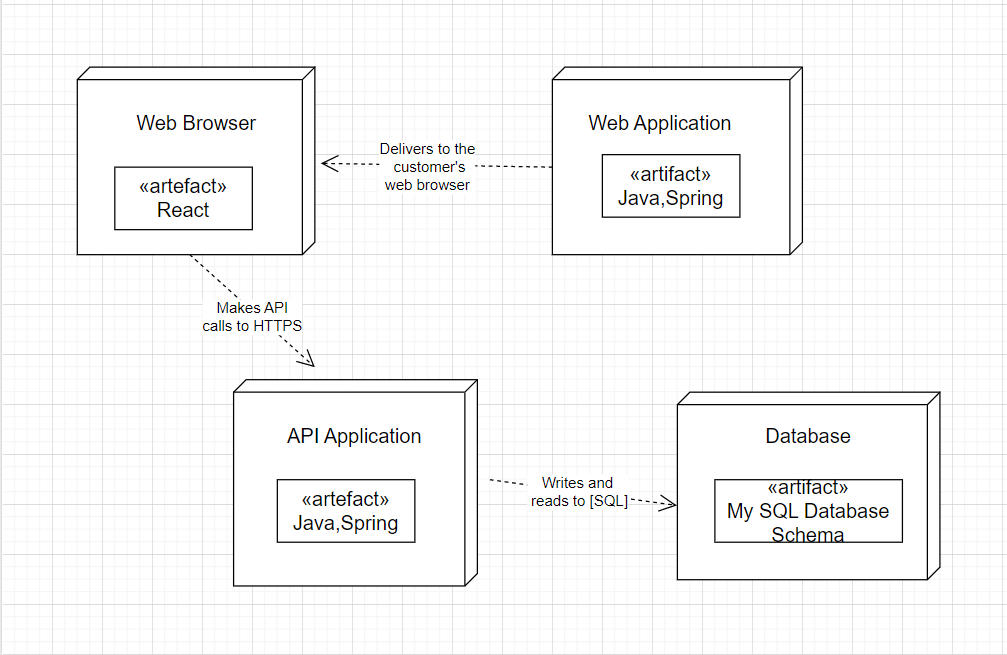
The conceptual architecture of the DreamCatch application is based on the layered architecture pattern, which separates the application into distinct layers, each with its own responsibilities and functionality. The layers in this architecture are: presentation layer( responsible for presenting the user interface to the user and handling user input. This layer communicates with the application layer to request and display data.), application layer (responsible for implementing the application's business logic, including processing user input and coordinating interactions between the presentation layer and the data access layer.), data access layer( responsible for accessing and manipulating data stored in the application's database.)

In addition to the layered architecture pattern, the DreamCatch application also utilizes the following architectural patterns:

Dependency injection: to manage dependencies between components and to enable modularity and testability.

Object-relational mapping (ORM): to facilitate the mapping of data between the application's object model and the database schema.

**Deployment Diagram**



**Component Diagram**

O imagine care conține text, diagramă, Plan, Desen tehnic

Descriere generată automat

**Package diagram**

O imagine care conține diagramă, text, Plan, Desen tehnic

Descriere generată automat

4. UML Sequence Diagrams

O imagine care conține diagramă

Descriere generată automat

5. Class Design

**5.1 Design Patterns Description**

Object-Relational Mapping (ORM) is a model that is used to facilitate data mapping between an application's object model and a database schema. The idea is to provide an abstraction layer between the application and the database, which makes it easier to work with data in a more object-oriented way. I implemented this using the Sprng framework.

Dependency Injection is a pattern that is used to manage dependencies between different components of a software application. The idea is to allow components to be loosely coupled so that they can be easily replaced or modified without affecting the rest of the application.

Another design pattern that is used in the DreamCatch application is the Repository pattern. This pattern is used to abstract away the details of data storage and retrieval. The repository provides a way for the application to interact with data storage without being tightly coupled to the specific implementation of the storage mechanism. In the DreamCatch application, the Repository pattern is used to abstract away the details of data storage and retrieval from the application layer, making it easier to swap out different data storage mechanisms in the future

Two React components are used to represent two different strategies for displaying charts based on the selected time period: WeeklyChartStrategy and MonthlyChartStrategy. These two components behave differently based on the type of chart selected (stress, duration, energy), but use the same basic structure to display the charts. This is an example of using the Strategy pattern, where a specific display component is used based on the selected strategy.

The chartFactory function is responsible for generating the corresponding chart based on the selected data type. It takes in the data for the chart and the chart type, and then creates and returns the component representing the corresponding chart. This is a typical approach of the Strategy pattern, where a function or method takes in a specific strategy and applies it in a specific way, without the need to change the basic logic of the function.

In the Home component, these components are used to display the corresponding charts based on the selected time period. Additionally, a handleViewChange method is used to switch the time period from weekly to monthly and vice versa.

Finally, the filtered data from the database is used to display the chart specific to the selected time period. This is an implementation of the Strategy pattern where display strategies can be changed without changing the basic display mode of the base components.

This code uses the Factory Pattern to generate the charts using the chartFactory function which creates different instances of Recharts components depending on the type of chart requested.

The Decorator Pattern is used to decorate graphical components. Specifically, the graph decorator receives a type of graph (duration, energy, stress), extracts the corresponding data from the chartData object, and calculates an average value for those data. This average value is then used to determine the color of the graph bars.

The decorator receives a graph component and replaces it with a decorated version that adds the functionality of coloring the bars according to the average values corresponding to the graph type. The decorated component is then used in various data display strategies (WeeklyChartStrategy and MonthlyChartStrategy) to create graphs for the filtered data corresponding to the selected time period.

The Decorator Pattern is used to add additional behavior to an existing component, without directly modifying it. This helps maintain modularity and flexibility in the graphical interface code.

In CQRS (Command Query Responsibility Segregation) architecture, the Mediator Pattern is used to handle the communication between the command and query sides of the system. The Mediator acts as a central point of control for executing commands and queries.

To use the Mediator Pattern in CQRS, we can create a Mediator class that implements an IMediator interface. The IMediator interface defines a single method "execute" that takes an ICommand or IQuery object as a parameter and returns the result.

The ICommand and IQuery interfaces represent the commands and queries that can be executed in the system. They both define a single method "handle" that takes a repository object as a parameter and returns the result of the command or query.

The repository object represents the data source for the command or query. It can be a database, a file system, or any other type of data source.

In the Mediator class, the execute method takes an ICommand or IQuery object as a parameter and calls its handle method, passing in the repository object. The result of the command or query is then returned to the caller.

Using the Mediator Pattern in CQRS architecture helps to decouple the command and query sides of the system, making it easier to maintain and extend the system over time. It also provides a centralized point of control for executing commands and queries, which can simplify the overall architecture of the system.

**5.2 UML Class Diagram**

O imagine care conține diagramă

Descriere generată automat

Front-end

O imagine care conține diagramă, linie, Plan, Desen tehnic

Descriere generată automat

*Back-end*

Entity package: Contains the Dream class, which represents the domain object. The ORM design pattern is used to map the Dream class to a relational database table.

Controller package: Contains the DreamController class, which handles incoming HTTP requests and delegates the work to the service layer.

Service package: Contains the DreamService class, which implements the business logic of the application. The Dependency Injection pattern is used to inject an instance of DreamRepository into the DreamService class.

Repository package: Contains the DreamRepository interface and its implementation, DreamRepositoryImpl. The DreamRepository interface defines the methods for interacting with the database, while DreamRepositoryImpl provides the actual implementation. The ORM pattern is used to map the Dream class to the database and the Dependency Injection pattern is used to inject an instance of EntityManager into the DreamRepositoryImpl class.

DTO package: Contains the DreamDto class, which is used to transfer data between layers. In this example, it is not directly related to any design pattern, but it is a common practice to use DTOs in web applications to avoid exposing domain objects to the client.

Decorator pattern is used to add additional behavior to the BarChartWrapper component. The ChartDecorator function takes a WrappedChart component as an argument, and returns a new component that wraps the WrappedChart component. This new component adds a color prop to the WrappedChart based on the average value of the chart data for a given type.

The Mediator Pattern is used to implement the CQRS (Command Query Responsibility Segregation) pattern. The CQRS pattern separates the write operations (commands) and the read operations (queries) into separate objects. The Mediator acts as a mediator between the commands and queries and the repositories (which handle the actual database operations).

By using the Mediator Pattern, the code becomes more modular and easier to maintain. The Mediator allows for loose coupling between the commands/queries and the repositories, making it easier to modify one without affecting the other. It also allows for better scalability and performance by allowing for separate scaling of the write and read operations.

6. Data Model

Dream Entity: This model represents a dream object and contains attributes such as dream id, title, description, date, and user id.

DreamDto: This model represents a data transfer object and contains fields such as dream id, title, description, date, and user id. It is used for data transfer between the frontend and backend of the application.

DreamRepository: This model represents the repository layer and contains methods for database operations such as insert, update, delete, and select.

DreamService: This model represents the service layer and contains methods for performing business logic on dream objects such as getting all dreams, getting dreams by user id, creating a new dream, updating an existing dream, and deleting a dream.

DreamController: This model represents the controller layer and contains RESTful endpoints for handling HTTP requests related to dreams such as getting all dreams, getting dreams by user id, creating a new dream, updating an existing dream, and deleting a dream.

7. System Testing

For the DreamCatch System, I did manual testing, I entered valid data to check if they are saved correctly in the database. With the help of the Stress, Duration and Energy buttons, you can see if the retrieved data are the same as those on the interface. Then, I entered invalid data, data validation was done both in the backend and in the frontend, and when invalid data is entered, a message will appear warning the user that they should enter a value between 1 and 5.

Link catre video: https://www.youtube.com/watch?v=FxCNeEGDsqM

8. Bibliography

1. <https://legacy.reactjs.org/>
2. <https://www.w3schools.com/java/>
3. <https://www.youtube.com/watch?v=vtPkZShrvXQ>
4. https://www.youtube.com/watch?v=4LZKnegAm4g&t=2270s