JavaProjectsDocument

ControllerClass

Controller class
==========

1. interface venam , interface complicate use our controller class
2. class name must functionality name and end suffix controller add.
3. class only have that class related methods only.
4. single responsibility pattern compare to groupin which groupin is best practice and maintain code.
6. method name prifix with "handle" and action name .
7. controller class method return type RespnseEntity is best aproch ,
8. each method have single action only.
9. class name prefix with name customer, client or corporate afte functionality Name for example
"CustomerHomeController","UserPaymentController"
10. method name prefix with action and add name of the class prefix like " addCustomer" , "processUserPayment"
11.Exception Handling class also write in controller class.

example:
=======

java Copy code

```java
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class UserController {

    // Class level comment explaining the purpose of the UserController
    // This controller handles HTTP requests related to users.

    /**
     * Handles HTTP DELETE requests to delete a user by ID.
     * @param id The ID of the user to be deleted.
     * @return ResponseEntity indicating the success of the operation.
     */
    @DeleteMapping("/{id}")
    public ResponseEntity<User> handleDeleteUser(@PathVariable Long id) {
        // Invoke the service method to delete the user by ID
        userService.deleteUser(id);
        // Return ResponseEntity with HTTP status indicating successful deletion
        return new ResponseEntity<>(user, HttpStatus.NO_CONTENT);
    }
}
```

ServiceClass

Service
==========
SOLID principle
==========

S - single responsiblitiy that is class have only one functionality.
O - Open / Closed principle that is existing modules add new functioality that is add new class and existing module without affect any functionality.
L - liskov priciple that is superclass method implement class compolsory override and also override method use otherwise method not using it define another interface.
I - Interface segregation that is every interface have single method or multiple that functionality related group of method.
D - dependency injection which means loose coupling

RepositoryClass

repository
==========

just interface only create in entity class related.

Project Structure
==========

```
src/
├── main/
│   ├── java/
│   │   └── com/
│   │       └── example/
│   │           ├── controller/
│   │           │   ├── UserController.java
│   │           │   └── ProductController.java
│   │           ├── service/
│   │           │   ├── UserService.java
│   │           │   └── ProductService.java
│   │           ├── model/
│   │           │   ├── User.java
│   │           │   └── Product.java
│   │           └── repository/
│   │               ├── UserRepository.java
│   │               └── ProductRepository.java
│   └── resources/
│       ├── static/
│       │   └── css/
│       │       └── styles.css
│       ├── templates/
│       │   ├── user/
│       │   │   ├── user-form.html
│       │   │   └── user-details.html
│       │   └── product/
│       │       ├── product-form.html
│       │       └── product-details.html
│       └── application.properties
└── test/
    └── java/
        └── com/
            └── example/
                ├── controller/
                │   ├── UserControllerTest.java
                │   └── ProductControllerTest.java
                └── service/
                    ├── UserServiceTest.java
                    └── ProductServiceTest.java
```

EndPoint Name

EndPoint Name
==========

1.End point name compalsory noun that is name of things and also provide plurals forms like "users"

2.user is singular which means we don't write b'z in every case json provide collection of users so that complexity it will create.

Products:
==========

1.GET /products - Retrieve a list of products
2.GET /products/{id} - Retrieve a specific product
3.POST /products - Create a new product
4.PUT /products/{id} - Update an existing product
5.DELETE /products/{id} - Delete a specific product
6.GET /products/search?query={keyword} - Search for products based on a keyword

Sure, here are some common RESTful API endpoints for an eCommerce platform:

1. **Products**:
   - `GET /products` - Retrieve a list of products
   - `GET /products/{id}` - Retrieve a specific product
   - `POST /products` - Create a new product
   - `PUT /products/{id}` - Update an existing product
   - `DELETE /products/{id}` - Delete a specific product
   - `GET /products/search?query={keyword}` - Search for products based on a keyword
2. **Categories**:
   - `GET /categories` - Retrieve a list of categories
   - `GET /categories/{id}` - Retrieve a specific category
   - `POST /categories` - Create a new category
   - `PUT /categories/{id}` - Update an existing category
   - `DELETE /categories/{id}` - Delete a specific category
3. **Orders**:
   - `GET /orders` - Retrieve a list of orders
   - `GET /orders/{id}` - Retrieve a specific order
   - `POST /orders` - Create a new order
   - `PUT /orders/{id}` - Update an existing order
   - `DELETE /orders/{id}` - Delete a specific order
   - `GET /orders/{id}/items` - Retrieve items of a specific order
   - `POST /orders/{id}/items` - Add items to a specific order
   - `DELETE /orders/{id}/items/{itemId}` - Remove a specific item from an order
4. **Users**:
   - `GET /users` - Retrieve a list of users
   - `GET /users/{id}` - Retrieve a specific user
   - `POST /users` - Create a new user
   - `PUT /users/{id}` - Update an existing user
   - `DELETE /users/{id}` - Delete a specific user
5. **Authentication & Authorization**:
   - `POST /auth/login` - User login
   - `POST /auth/register` - User registration
   - `POST /auth/logout` - User logout
   - `GET /auth/profile` - Retrieve user profile
   - `PUT /auth/profile` - Update user profile
   - `PUT /auth/password` - Update user password
6. **Reviews & Ratings**:
   - `GET /products/{id}/reviews` - Retrieve reviews for a specific product
   - `POST /products/{id}/reviews` - Add a review to a specific product
   - `GET /products/{id}/ratings` - Retrieve ratings for a specific product
   - `POST /products/{id}/ratings` - Add a rating to a specific product

Segregation Principle

Segregation principle ??????

==========

1.Directly Implement Interface:

---

that class is only have that functionality gor for direct implement.example notification is a method implement email , email related functionality write. this is not have shared logic.

2.Use Common Class for Shared Logic:

---

that is common shared logic write and aggregation used to get common logic also override common logic.

Example

```
public interface ProductService {
    List<Product> getAllProducts();
    Product getProductById(String id);
    Product createProduct(Product product);
    Product updateProduct(String id, Product product);
    void deleteProduct(String id);
    List<Review> getProductReviews(String id);
    Review addReviewToProduct(String id, Review review);
    List<Rating> getProductRatings(String id);
    Rating addRatingToProduct(String id, Rating rating);
    List<Product> searchProducts(String query);
}

public interface RatingService {
    List<Rating> getProductRatings(String id);
    Rating addRatingToProduct(String id, Rating rating);
}

public interface ProductSearchService {
    List<Product> searchProducts(String query);
}

public interface ReviewService {
    List<Review> getProductReviews(String id);
    Review addReviewToProduct(String id, Review review);
}
```

Abstract Clsss

Abstract class when use
==========

abstract class use pannanumna common method functionality is not change extend any class like static variable provide common thing

same in abstract method after some dynamic method also add . by extend any class compalsory implement and alredy implemt abstract

method just using.

Java Doc

java doc
==========
class:
=========
write class name same as functionality name may need not write doc.

methods
==========
write method name same as action name however write complex logic place may write doc.

Java Copy code

```java
/**
 * Service class for managing user operations.
 *
 * <p>This service provides methods to perform CRUD operations on user data,
 * such as creating, retrieving, updating, and deleting users.</p>
 *
 * @see User
 * @see UserRepository
 * @see UserService
 */
public class UserServiceImpl implements UserService {

    private final UserRepository userRepository;
    private final PasswordEncoder passwordEncoder;

    /**
     * Constructor to initialize the service with necessary dependencies.
     *
     * @param userRepository the user repository
     * @param passwordEncoder the password encoder
     */
    public UserServiceImpl(UserRepository userRepository, PasswordEncoder passwordEncoder) {
        this.userRepository = userRepository;
        this.passwordEncoder = passwordEncoder;
    }

    /**
     * Creates a new user in the system.
     *
     * @param user the user to be created
     * @return the created user
     */
    @Override
    public User createUser(User user) {
        validateUser(user);

        user.setPassword(passwordEncoder.encode(user.getPassword()));
        return userRepository.save(user);
    }

    /**
     * Retrieves a user by their ID.
     *
     * @param userId the ID of the user to retrieve
     * @return the retrieved user
     * @throws UserNotFoundException if the user is not found
     */
    @Override
    public User getUserById(Long userId) {
        return userRepository.findById(userId)
                .orElseThrow(() -> new UserNotFoundException("User not found with id: " + userId));
    }

    /**
     * Updates an existing user.
     *
     * @param userId the ID of the user to update
     * @param userDetails the new user details
     * @return the updated user
     * @throws UserNotFoundException if the user is not found
     */
    @Override
    public User updateUser(Long userId, User userDetails) {
        User existingUser = getUserById(userId);

        existingUser.setName(userDetails.getName());

        existingUser.setEmail(userDetails.getEmail());
        if (userDetails.getPassword() != null && !userDetails.getPassword().isEmpty()) {

            existingUser.setPassword(passwordEncoder.encode(userDetails.getPassword()));
        }
        return userRepository.save(existingUser);
    }

    /**
     * Deletes a user by their ID.
     *
     * @param userId the ID of the user to delete
     * @throws UserNotFoundException if the user is not found
     */
    @Override
    public void deleteUser(Long userId) {
        User existingUser = getUserById(userId);

        userRepository.delete(existingUser);
    }

    /**
     * Validates the user data before processing.
     *
     * @param user the user to validate
     * @throws ValidationException if the user data is invalid
     */
    private void validateUser(User user) {
        if (user.getName() == null || user.getName().isEmpty()) {
            throw new ValidationException("User name cannot be empty");
        }
        if (user.getEmail() == null || user.getEmail().isEmpty()) {
            throw new ValidationException("User email cannot be empty");
        }
        if (user.getPassword() == null || user.getPassword().isEmpty()) {
            throw new ValidationException("User password cannot be empty");
        }
    }
}
```

Common Comments

Java Copy code

```java
import java.util.*;
import java.text.SimpleDateFormat;

public class Example {

    // Initialize logger
    private static final Logger logger =
Logger.getLogger(Example.class.getName());

    public static void
main(String[] args) {
        // Initialize variables
        List<String> items = new
ArrayList<>();
        double totalPrice = 0;
        double discount = 10;
        double taxRate = 5;

        // Add some items
        items.add("Item1");
        items.add("Item2");

        // Calculate the total
amount
        totalPrice =
calculateTotal(items);

        // Apply discount to the
total price
        totalPrice *= (1 -
(discount / 100));

        // Check if the object is
null before proceeding
        if (items != null) {
            // Check if the list is
empty before processing
            if (!items.isEmpty()) {
                // Perform
operation
                System.out.println("Processing
items...");
            }
        }

        // Handle exception
        try {
            // Validate the input
parameters before processing
            validateInput(items,
taxRate, discount);

            // Calculate the tax
amount
            double taxAmount =
calculateTax(totalPrice, taxRate);

            // Update the total
price
            totalPrice +=
taxAmount;

            // Check if the index
is within bounds
            if (items.size() > 0) {
                // Log an
informational message
                logger.info("Items
processed successfully.");
            }
        } catch (Exception e) {
            // Display an error
message to the user
            System.err.println("An
error occurred: " +
e.getMessage());
        }

        // Check the string is null
or empty
        String str = "";
        if (str != null &&
!str.isEmpty()) {
            // Parse the input
string to extract data
            parseInput(str);
        }

        // Format the date to the
required pattern
        String formattedDate =
formatDate(new Date());

        // Update the status of the
order
        String status =
"Processed";
        updateStatus(status);

        // Fetch data from the
server
        fetchDataFromServer();

        // Check if the user has
the necessary permissions
        boolean hasPermission =
checkPermissions("admin");

        // Retry the operation if
it fails
        for (int i = 0; i < 3; i++)
{
            if (performOperation())
{
                break;
            }
        }

        // Initialize UI components
for the screen
        initializeUI();

        // Send a notification to
the user
        sendNotification("Processing
completed.");

        // Save the output to a
file
        saveToFile("output.txt");

        // Initialize resources for
database connection
        initializeResources();

        // Close resources to
prevent memory leaks
        closeResources();

        // Return the result to the
caller
        return totalPrice;
    }

    private static void
validateInput(List<String> items,
double taxRate, double discount) {
        // Check for null or empty
list
        if (items == null ||
items.isEmpty()) {
            throw new
IllegalArgumentException("Items
list cannot be null or empty.");
        }
        // Check the discount value
is within a valid range before
applying it
        if (discount < 0 ||
discount > 100) {
            throw new
IllegalArgumentException("Discount
must be between 0 and 100.");
        }
        // Ensure the tax rate is
not negative before calculating tax
        if (taxRate < 0) {
            throw new
IllegalArgumentException("Tax rate
cannot be negative.");
        }
    }

    private static double
calculateTotal(List<String> items)
{
        // Initialize the total
price with the price of the first
product added
        double total = 0;
        for (String item : items) {
            // Assuming each item
has a fixed price of 10 for example
purposes
            total += 10;
        }
        return total;
    }

    private static double
calculateTax(double totalPrice,
double taxRate) {
        // Calculate the tax amount
        return totalPrice *
(taxRate / 100);
    }

    private static void
fetchDataFromServer() {
        // Fetch data from the
server
    }

    private static boolean
checkPermissions(String role) {
        // Check if the user has
the necessary permissions
        return
"admin".equals(role);
    }

    private static boolean
performOperation() {
        // Perform the specified
operation
        return true;
    }

    private static void
initializeUI() {
        // Initialize UI components
for the screen
    }

    private static void
sendNotification(String message) {
        // Send a notification to
the user
    }

    private static void
saveToFile(String filename) {
        // Save the output to a
file
    }

    private static void
initializeResources() {
        // Initialize resources for
database connection
    }

    private static void
closeResources() {
        // Close resources to
prevent memory leaks
    }

    private static void
parseInput(String str) {
        // Parse the input string
to extract data
    }

    private static String
formatDate(Date date) {
        // Format the date to the
required pattern
        SimpleDateFormat formatter
= new
SimpleDateFormat("yyyy-MM-dd");
        return
formatter.format(date);
    }

    private static void
updateStatus(String status) {
        // Update the status of the
order
    }
}
```

HibernetQuery

Query

---

default methods:

---

1. save(S entity):
2. findOne(ID id):
3. findAll():
4. existsById(ID id):
5. findAllById(Iterable<ID> ids):
6. count():
7. deleteById(ID id):
8. delete(T entity):

Using Method Names

If your query can be expressed using Spring Data's method naming conventions, you can simply define a method in your repository interface. For example:

java Copy code

```java
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    List<User> findByUsername(String username);
}
```

1. findByFirstNameAndLastName(String firstName, String lastName):
2. findByAgeGreaterThan(int age):
3. findBySalaryLessThan(double salary):
4. findByCity(String city):
5. findByState(String state):
6. findByCountry(String country):
7. findByFirstNameOrLastName(String firstName, String lastName):

8. findByAgeLessThanEqual(int age):
9. findByActiveTrue():
10. findByActiveFalse():
11. findByFirstNameIgnoreCase(String firstName):
12. findByLastNameIgnoreCase(String lastName):
13. findByFirstNameLike(String pattern):
14. findByLastNameLike(String pattern):
15. findByFirstNameStartingWith(String prefix):
16. findByLastNameEndingWith(String suffix):
17. findByDateOfBirthBetween(Date startDate, Date endDate):
18. findByFirstNameOrderByLastNameAsc(String firstName):
19. findByLastNameOrderByFirstNameDesc(String lastName):
20. findByAgeGreaterThanEqualAndCity(int age, String city):
21. findByStateAndCountry(String state, String country):
22. findByFirstNameNotNull():
23. findByLastNameNull():
24. findByDateOfBirthNotNull():
25. findByAgeIn(Collection<Integer> ages):
26. findBySalaryNot(double salary):
27. findByFirstNameAndActiveTrue(String firstName):
28. findByLastNameAndActiveFalse(String lastName):
29. findByCityOrState(String city, String state):
30. findByCountryNot(String country):
31. findByFirstNameContainingIgnoreCase(String keyword):
32. findByLastNameContainingIgnoreCase(String keyword):
33. findByAgeGreaterThanEqualAndAgeLessThan(int minAge, int maxAge):
34. findByDateOfBirthBefore(Date date):
35. findByDateOfBirthAfter(Date date):
36. findByDateOfBirthNotNullAndActiveTrue():
37. findByFirstNameAndLastNameAndAge(String firstName, String lastName, int age):
38. findByLastNameNotIn(Collection<String> lastNames):
39. findByAgeGreaterThanAndCityOrState(int age, String city, String state):
40. findBySalaryGreaterThanEqualAndSalaryLessThan(double minSalary, double

maxSalary):

41. findByActiveTrueAndCityNotNull():
42. findByActiveFalseAndStateNull():
43. findByDateOfBirthBetweenAndCountry(Date startDate, Date endDate, String country):
44. findByFirstNameIgnoreCaseOrLastNameIgnoreCase(String firstName, String lastName):
45. findByCityStartingWithIgnoreCase(String prefix):
46. findByStateEndingWithIgnoreCase(String suffix):
47. findByCountryIn(Collection<String> countries):
48. findByFirstNameNotNullAndLastNameNotNull():
49. findByAgeLessThanEqualAndCityOrState(int age, String city, String state):
50. findBySalaryGreaterThanEqualOrSalaryLessThanEqual(double minSalary, double maxSalary):

common criteria keywords

No.	Keyword	No.	Keyword	No.	Keyword	No.
1	And	26	TrueOrderBy	51	AfterAnd	76
2	Or	27	FalseOrderBy	52	BetweenAnd	77
3	Containing	28	InOrderBy	53	LessThanAnd	78
4	StartingWith	29	NotInOrderBy	54	GreaterThanAnd	79
5	EndingWith	30	BeforeOrderBy	55	StartingWithAnd	80
6	GreaterThan	31	AfterOrderBy	56	EndingWithAnd	81
7	LessThan	32	LessThanOrderBy	57	ContainingAnd	82
8	Between	33	GreaterThanOrderBy	58	BeforeOr	83
9	IsNotNull	34	BetweenOrderBy	59	AfterOr	84
10	IsNull	35	StartingWithOrderBy	60	BetweenOr	85
11	Not	36	EndingWithOrderBy	61	LessThanOr	86

Keyword
BeforeOrderBy
AfterOrOrderBy
BetweenOrOrderBy
LessThanOrOrderBy
GreaterThanOrOrderBy
StartingWithOrOrderBy
EndingWithOrOrderBy
IsNotNullOrderBy
IsNullOrderBy
NotOrderBy
TrueOrderBy

12	OrderBy	37	ContainingOrderBy	62	GreaterThanOr	87
13	IgnoreCase	38	AndNotNull	63	StartingWithOr	88
14	True	39	OrNotNull	64	EndingWithOr	89
15	False	40	IgnoreCaseAnd	65	ContainingOr	90
16	In	41	IgnoreCaseOr	66	BeforeAndOrderBy	91
17	NotIn	42	TrueAnd	67	AfterAndOrderBy	92
18	Before	43	TrueOr	68	BetweenAndOrderBy	93
19	After	44	FalseAnd	69	LessThanAndOrderBy	94
20	Like	45	FalseOr	70	GreaterThanAndOrderBy	95
21	NotNull	46	InAnd	71	StartingWithAndOrderBy	96
22	Null	47	InOr	72	EndingWithAndOrderBy	97
23	AndOrderBy	48	NotInAnd	73	BeforeOrOrderBy	98

FalseOrderBy

InOrderBy

NotInOrderBy

BeforeOrderBy

AfterOrderBy

LessThanOrderBy

GreaterThanOrderBy

StartingWithOrderBy

EndingWithOrderBy

ContainingOrderBy

AndNotNullOrderBy

OrNotNullOrderBy

24	OrOrderBy
25	IgnoreCaseOrderBy

49	NotInOr	74	AfterOrOrderBy	99	IgnoreCaseAndOrderBy
50	BeforeAnd	75	BetweenOrOrderBy	100	IgnoreCaseOrOrderBy

Using @Query Annotation

For more complex queries or when method naming conventions are not sufficient, you can write JPQL queries using the `@Query` annotation. Here's an example:

java Copy code

```java
@Repository
public interface UserRepository extends JpaRepository<User, Long> {

    @Query("SELECT u FROM User u WHERE u.username = :username")
    List<User> findByUsername(@Param("username") String username);
}
```

Explanation of Each Component

- **@Query**: Defines a custom query.
- **SELECT**: JPQL keyword to specify the selection of data.
- **u**: Alias for the `User` entity.
- **FROM**: JPQL keyword indicating the data source.
- **User**: The entity class being queried.
- **WHERE**: JPQL keyword to specify conditions.
- **u.email**: Field in the `User` entity.
- **= ?1**: Parameter placeholder for the first method argument.

Native Queries

You can also execute native SQL queries using the `@Query` annotation with the `nativeQuery` attribute set to `true`. Here's an example:

java Copy code

```java
@Repository
public interface UserRepository extends JpaRepository<User, Long> {

    @Query(value = "SELECT * FROM users WHERE username = :username", nativeQuery =
    List<User> findByUsername(@Param("username") String username);
}
```

Explanation of Each Component

- **@Query**: Annotation to define a custom query.
- **value =**: Indicates the beginning of the query string.
- **"SELECT * FROM users WHERE email = ?1"**: The SQL query string.◇ **SELECT ***: SQL keyword to specify the selection of all columns.
- **FROM**: SQL keyword indicating the data source.
- **users**: The database table name.
- **WHERE**: SQL keyword to specify conditions.
- **email**: Column in the `users` table.
- **= ?1**: Parameter placeholder for the first method argument.
- **nativeQuery = true**: Specifies that this is a native SQL query.

Return Type
=======

1. **Entity**:
- **Example**: `User findById(Long id);`
- **Purpose**: Returns a single entity object. Use this when you expect the query to return exactly one result.

- **Optional<Entity>**:
◇ **Example**: `Optional<User> findById(Long id);`
◇ **Purpose**: Wraps the result in an `Optional` to handle the case when the result might be null (i.e., no entity found). This is useful for avoiding `NullPointerException` and for explicitly handling the absence of a result.

- **List<Entity>**:
  - ◇ **Example**: `List<User> findByLastName(String lastName);`
  - ◇ **Purpose**: Returns a list of entities. Use this when you expect multiple results.

- **Page<Entity>**:
  - ◇ **Example**: `Page<User> findByLastName(String lastName, Pageable pageable);`
  - ◇ **Purpose**: Returns a paginated list of entities. Useful for implementing pagination in your application.

- **Slice<Entity>**:
  - ◇ **Example**: `Slice<User> findByLastName(String lastName, Pageable pageable);`
  - ◇ **Purpose**: Similar to `Page`, but only fetches a slice of data (partial page) without needing the total count. Useful for infinite scrolling scenarios.

- **Stream<Entity>**:
  - ◇ **Example**: `Stream<User> findAllByAgeGreaterThan(int age);`
  - ◇ **Purpose**: Returns a Java 8 stream of entities, allowing for lazy processing of large datasets.

- **Collection<Entity>**:
  - ◇ **Example**: `Collection<User> findByCity(String city);`
  - ◇ **Purpose**: Returns a collection of entities. Similar to `List` but more general.

- **Set<Entity>**:
  - ◇ **Example**: `Set<User> findDistinctByLastName(String lastName);`
  - ◇ **Purpose**: Returns a set of entities. Ensures that the results are unique.

- **Entity Projection (Interface)**:
  - ◇ **Example**: `List<UserNameOnly> findBy();`
  - ◇ **Purpose**: Returns a projection interface, allowing for partial retrieval of entity data.

• **DTO (Data Transfer Object)**:
◇ **Example**: `List<UserDTO> findBy();`
◇ **Purpose**: Returns a DTO object, often used to shape the data returned from the query to meet the specific needs of the client.

Why Use These Return Types?
◇ **Entity**: Directly interact with the entity in the database. Suitable for straightforward CRUD operations.
◇ **Optional<Entity>**: Handle potential absence of data gracefully.
◇ **List<Entity>**: Retrieve multiple entities when you expect more than one result.
◇ **Page<Entity>**: Efficiently manage large datasets by breaking them into pages. Ideal for pagination.
◇ **Slice<Entity>**: Fetch portions of data without needing the total count. Good for scenarios like infinite scrolling.
◇ **Stream<Entity>**: Process large datasets in a memory-efficient way.
◇ **Collection<Entity>**: More generic than `List`, but serves a similar purpose.
◇ **Set<Entity>**: Ensure uniqueness of results.
◇ **Entity Projection**: Optimize performance by retrieving only the necessary fields.
◇ **DTO**: Tailor the query results to fit the specific requirements of the client, enhancing performance and reducing data transfer overhead.