**GIT Operations**

1. **Create repository at github**
2. **Add the Remote to Your Local Git Repo**
   1. **Git remote add origin <mentioned url from github without this bracket>**
3. **Create or Initialise git in project folder**
   1. **Git init**
4. **Create branch**
   1. **Git checkout -b admin-controller branch**
5. **Add all changes**
   1. **Git add .**
6. **Commit changes with message**
   1. **Git commit -m “added admin controller with service class”**
7. **Push new branch to remote**
   1. **Git push -u origin admin-controller-branch**
8. **Merge with main branch**
   1. **Git checkout master**
   2. **Git merge admin-controller**
   3. **Git push**

**hierarchy and communication flow** of **Spring Boot Shopping Cart Application**, including:

* Users
* Products
* Cart
* Notification (SMS, Email, etc.)

**✅ 1. 🗂 Project Structure**

Below is the **Maven-style folder structure** with packages and key files:

shopping-cart/

├── src/

│ └── main/

│ ├── java/

│ │ └── com/

│ │ └── Flipkart/

│ │ └── shoppingcart/

│ │ ├── controller/

│ │ │ └── CartController.java

│ │ ├── service/

│ │ │ ├── CartService.java

│ │ │ ├── NotificationService.java (interface)

│ │ │ ├── SMSNotificationService.java

│ │ │ └── EmailNotificationService.java

│ │ ├── repository/

│ │ │ ├── UserRepository.java

│ │ │ └── ProductRepository.java

│ │ ├── entity/

│ │ │ ├── User.java

│ │ │ └── Product.java

│ │ └── ShoppingCartApplication.java

│ └── resources/

│ ├── application.properties

│ └── schema.sql / data.sql (optional)

├── pom.xml

**✅ 2. 💬 How the Components Communicate**

Here’s how each layer **talks to the others**:

**▶️ 1. CartController**

* Accepts HTTP request (from frontend or Postman)
* Calls CartService.addToCart(...)

@PostMapping("/cart/add")

public String addToCart(@RequestParam Long userId, @RequestParam Long productId, @RequestParam String notifyBy)

**▶️ 2. CartService**

* Calls UserRepository and ProductRepository to fetch user and product.
* Adds product to cart (you can maintain cart in DB or memory).
* Decides which NotificationService to use (SMS/Email/etc.)
* Sends notification.

java

CopyEdit

notificationService.sendNotification("Product added to cart", user.getUsername());

**▶️ 3. NotificationService (Interface)**

* Has method sendNotification(String message, String to)
* Implemented by multiple services:
  + SMSNotificationService
  + EmailNotificationService

Use @Qualifier or dynamic selection to choose between them.

**▶️ 4. UserRepository and ProductRepository**

* Extend JpaRepository or CrudRepository
* Fetch and save data from/to database

java

CopyEdit

User user = userRepository.findById(userId).orElseThrow();

Product product = productRepository.findById(productId).orElseThrow();

**▶️ 5. User and Product Entities**

* Represent database tables using @Entity
* Contain only fields, constructors, getters, and setters

**🔁 Overall Communication Flow Diagram (Logical)**

less

CopyEdit

Client (Postman/Frontend)

|

v

[ CartController ]

|

v

[ CartService ]

|

+--> [ UserRepository ] ---> [ User Table in DB ]

|

+--> [ ProductRepository ] ---> [ Product Table in DB ]

|

+--> [ NotificationService (Interface) ]

|

-------------------------

| |

[ SMSNotificationService ] [ EmailNotificationService ]

**✅ 3. Short Summary of Spring Annotations Used**

| **File** | **Annotation** | **Role** |
| --- | --- | --- |
| User, Product | @Entity | Map class to DB table |
| UserRepository, ProductRepository | @Repository | Data access layer |
| CartService, SMSNotificationService, etc. | @Service | Business logic |
| CartController | @RestController | REST API |
| ShoppingCartApplication | @SpringBootApplication | Main class |
| Interface Injection | @Autowired + @Qualifier | Inject beans |

Annotation Hierarchy

1. Entity Class
   1. @Entity : class level, Map class to DB table

**Required Dependencies (With Use Case)**

| **Dependency** | **Maven Artifact** | **Why You Need It** |
| --- | --- | --- |
| ✅ Spring Web | spring-boot-starter-web | To build REST APIs (controllers, JSON request/response) |
| ✅ Spring Data JPA | spring-boot-starter-data-jpa | To interact with relational databases using JpaRepository, @Entity, etc. |
| ✅ H2 Database *(for dev/test)* | com.h2database:h2 | In-memory DB for testing. Auto-configured and web console available. |
| ✅ Lombok *(optional but helpful)* | org.projectlombok:lombok | Reduces boilerplate — generates getters, setters, constructors, etc. |
| ✅ Spring Boot DevTools *(optional)* | spring-boot-devtools | Enables live reload and faster development experience |
| ✅ Validation *(optional but good for form/input validation)* | spring-boot-starter-validation | To use annotations like @NotNull, @Size, @Email, etc. in DTOs/entities |

Solution

1. create following packages in com.flipkart.shoppingkart
   1. Controller: to manage API mappings
   2. Service: to write business logic
   3. Repository: to manage database connectivity
   4. Entity: to manage entities like class with variable declaration and setter getter
2. Create classes in respective package
   1. CartController
   2. userController
   3. productController
   4. CartService
   5. UserService
   6. ProductService
   7. NotificationService(Interface)
      1. EmailNotification
      2. SmaNotification
   8. UserEntity
   9. ProductEntity
   10. UserRepositoty(Interface)
   11. ProductRepository(Interface)
3. EntityPackage
   1. UserEntity
      1. @Data // this provide setter getter and reqd constructors its from Lombok dependency
      2. @Entity // Map this class to the database table
      3. **public** **class** User {
      4. @Id // make id as primary key
      5. @GeneratedValue(strategy = GenerationType.***IDENTITY***) // this will generate values for id automatically
      6. **private** Long id;
      7. **private** String name;
      8. **private** String username;
      9. **private** String password;
      10. }
   2. ProductEntity
      1. @Entity
      2. @Data
      3. **public** **class** Product {
      4. @Id
      5. @GeneratedValue(strategy = GenerationType.***IDENTITY***)
      6. **private** Long id;
      7. **private** String name;
      8. **private** Double price;
      9. }
4. Repository Package
   1. UserRepository
      1. @Repository // This will map the the interface as Spring managed DAO bean and handle exceptions
      2. **public** **interface** UserRepository **extends** JpaRepository<User,Long>{
      3. // JpaRepository will provide methods for
      4. //save, findById, count, deleteById, findAll(), etc
      5. }
   2. ProductRepository
      1. @Repository
      2. public interface ProductRepository extends JpaRepository<Product, Long>
      3. { }
5. Service Package
   1. NotificationService interface
      1. Public interface NotificationService{
      2. Void sendNotification(String msg, String to);
   2. EmailNotification
      1. @Service(“emailNotification”)
      2. Public class EmailNotfication implements NotificationService{
      3. Public void sendNotification(String msg, String to){
      4. Sysout(“email sent: “+msg+” to the: ”+to);
      5. }}
   3. SmsNotification
      1. @Service()
      2. Public class SmsNotification implement NotificationService{
      3. Public void sendNotification( String msg, String to ){
      4. Sysout(“Sms : “+msg+” sent to:”+to);
      5. }}
   4. CartService
      1. @Service
      2. **public** **class** CartService {
      3. @Autowired
      4. **private** ProductRepository productRepository;
      6. @Autowired
      7. **private** UserRepository userRepository;
      9. @Autowired
      10. @Qualifier("smsNotification")
      11. **private** NotificationService smsNotification;
      13. @Autowired
      14. @Qualifier("emailNotification")
      15. **private** NotificationService emailNotification;
      17. **public** **void** addToCart(Long userId, Long productId, String notifyBy)
      18. {
      19. User user = userRepository.findById(userId).orElseThrow();
      20. Product product = productRepository.findById(productId).orElseThrow();
      21. String msg = "Product "+product.getName()+" added to cart of user: "+user.getName();
      23. NotificationService notifier = notifyBy.equalsIgnoreCase("sms") ? smsNotification : emailNotification;
      24. notifier.sendNotification(msg, user.getName());
      25. }

      28. }
   5. UserService
      1. @Service
      2. **public** **class** UserService {
      3. @Autowired
      4. **private** UserRepository userRepository;
      6. **public** List<User> getUsers()
      7. {
      8. **return** userRepository.findAll();
      9. }
      11. **public** List<User> getUsersSorted()
      12. {
      13. **return** userRepository.findAll().stream().sorted(Comparator.*comparing*(User::getId)).collect(Collectors.*toList*());
      14. }
      16. **public** User getUserById(Long userId)
      17. {
      18. **return** userRepository.findById(userId).orElseThrow();
      19. }
      21. **public** User addUser(User user)
      22. {
      23. **return** userRepository.save(user);
      24. }
      26. **public** String deleteUser(Long userid)
      27. {
      28. **if**(userRepository.existsById(userid))
      29. {
      30. userRepository.deleteById(userid);
      31. **return** "user deleted from database";
      32. }
      33. **return** "no user found";
      34. }
      35. }
   6. ProductService
      1. @Service
      2. **public** **class** ProductService {
      4. @Autowired
      5. **private** ProductRepository productRepository;
      7. **private** Product product;
      9. **public** List<Product> getProducts()
      10. {
      11. **return** productRepository.findAll();
      12. }
      14. **public** Product getProduct( Long productId)
      15. {
      16. **return** productRepository.findById(productId).orElseThrow();
      17. }
      19. **public** Product addProduct(Product product)
      20. {
      21. **return** productRepository.save(product);
      22. }
      24. **public** String deleteProduct(Long productId)
      25. {
      26. **if**(productRepository.existsById(productId))
      27. {
      28. productRepository.deleteById(productId);
      29. **return** "product entry deleted";
      30. }
      31. **return** "product id does not exist";
      32. }
      33. }
   7. **Sdf**
6. **Make DataBase connection** (MySQL)
   1. Add MySQL Dependency
      1. <dependency>
      2. <groupId>mysql</groupId>
      3. <artifactId>mysql-connector-java</artifactId>
      4. <version>8.0.33</version>
      5. </dependency>
   2. Modify application.properties
      1. # MySQL DB connection
      2. spring.datasource.url=jdbc:mysql://localhost:3306/shopping\_app
      3. spring.datasource.username=root
      4. spring.datasource.password=root
      5. spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
      6. # JPA settings
      7. spring.jpa.hibernate.ddl-auto=update
      8. spring.jpa.show-sql=true
      9. spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect
7. Project is working fine now
8. **Version2**
9. But lets make it advance by introducing concccept of cart and foreign Key
10. Like now we will have 3 tables user,product and cart. Cart table has fields like
    1. Cart id(pk)
    2. Userid(FK)
    3. Productid(FK)
    4. Quantity
    5. Date
11. SO structure is like Every user has one cart, if user deleted then cart will also deleted (1:1 relationship)
12. Every cart has many Items (1:N relationship)
13. Multiple cart has one type of product(N:1 relationship)
14. Create Entity class Cart
    1. @Entity
    2. Public class Cart {
    3. @Id
    4. @GeneratedValue(strategy = GenerationType.IDENTITY)
    5. Private Long id;
    6. @ManyToOne // there will be many entries(rows) for one user\_id
    7. @JoinColumn(name=”user\_id”) //defining foreignKey user\_id
    8. Private User user;
    9. @ManyToOne
    10. @JoinColumn(name=”product\_id”)
    11. Private Product product;
    12. Private int quantity;
    13. Private LocalDataTime date;
    14. //setter and getter methods
15. Create Repository Interface CartRepository
    1. Public interface CartRepository cartRepository extends JpaRepository<Cart,Long>{
    2. Public List<Cart> findByProductId(Long productid);// this will automatically find the cart as per product id its like SQL query select \* from cart where productid=productid;
    3. Public List<Cart> findByUserId(Long userId);
    4. Public List<Cart> findByUserIdAndProductId(Long userId, Long productId)
    5. }
16. Modify addToCart() in CartService class
    1. Only when user and product is present then add item to cart
    2. If userid and product id is same then just add quantity to existing cart other wise add whole product
    3. At the end send notification
    4. Add methods like
       1. get all items in cart
       2. get cart of specific user
       3. get cart entries of particular product and all
    5. Get
17. **-----------Perform Unit Testing --------------------**
18. Add required dependencies like
    1. JUnit Jupiter (Aggregator)
    2. Mockito Core
    3. Mockito inline
    4. Springboot starter-test
19. Create Service Package in src.test.java
20. Create class UserServiceTest
    1. Use annotation @ExtendWith(MockitoExtension.class) for class so that we can use annotations like @InjectMocks or @Mock
    2. Inside class use annotation @InjectMocks to the object of class that we need to test. This will inject object of class
       1. @InjectMocks
       2. Private UserService userService;
    3. Use annotation @Mock to the field that we need for above class and we need to create fake data ie mocking data
       1. @Mock
       2. Private UserRepository userRepository
    4. Use annotation @Test to the methods that we need to test
       1. @Test
       2. Void getUserById\_userNotFound(){
       3. // here we are telling mockito that whenever there is call for method userRepository.findById (1L) that time return empty object. Without checking actual database. Optional is container object
       4. When(userRepository.findById(1L)).thenReturn(Optional.empty());
       5. // Now try with actual method of service class. We are expecting where no user found with provided id then it should through NoSuchElementException.
       6. When we are expecting exception then use assertion assertThrows()
       7. assertThrows(NoSuchElementExceeption.class,()->{
          1. userService.getUserById(1L); } );
       8. }
          1. @Test
          2. **void** getUserById\_notFound()
          3. {
          4. *when*(userRepository.findById(1L)).thenReturn(Optional.*empty*());
          5. *assertThrows*(NoSuchElementException.**class**, ()->{userService.getUserById(1L);});
          6. }
    5. Now test Method getUserByID() when user is present
       1. We know from userService lcass that if user is present with given ID the it should return that user object so we can verify three things
          1. User’s Name
          2. Verify that there is call for method userRepository.findByID
          3. Check userRepository.findById() should not return null value
       2. @Test
       3. Void getUserById\_UserFound()
       4. //create dummy user object to test
          1. User user = new User();
          2. User.setId(1L);
          3. User.setName(“Roshan”);
       5. // tell Mockito that whenever there is call for userRepository.findById(1L) then return this user object
          1. When(userRepository.findById(1L)).thenReturn(Optional.of(user)
       6. //Check with actual Method
       7. assertEquals(“Roshan”,userService.getUserById(1L).getName());
       8. verify(userRepository).findById(1L);
       9. assertNotNull(userService.getUserById(1L))
          1. @Test
          2. **void** getUserById\_UserFound()
          3. {
          4. User user = **new** User();
          5. user.setId(1L);
          6. user.setName("Roshan");
          7. *when*(userRepository.findById(1L)).thenReturn(Optional.*of*(user));
          8. //assertNotNull(userRepository.findById(1L));
          9. *assertEquals*("Roshan",userService.getUserById(1L).getName());
          10. *assertNotNull*(userService.getUserById(1L));
          11. *verify*(userRepository,*times*(2)).findById(1L);
          12. }
    6. Test getUsers() method
       1. We know that this method will return list of users so we can test following things
          1. Total number of users
          2. There is successful call for method findAll() from userRepository
       2. First create two users and add them in userList;
       3. Then tell Mockito that whenever there is call for findAll method() return userList
          1. When(userRepository.findAll()).thenReturn(userList);
       4. Check service class method
          1. assertEquals(2, userService.getUsers().size());
          2. verify(userRepository).findAll();
       5. actuial code as follows
          1. @Test
          2. **void** getUsers\_Test()
          3. {
          4. User user1 = **new** User();
          5. user1.setId(1L);
          6. user1.setName("Roshan");
          8. User user2 = **new** User();
          9. user2.setId(2L);
          10. user2.setName("Patil");
          12. List<User> userlist = List.*of*(user1,user2);
          14. *when*(userRepository.findAll()).thenReturn(userlist);
          15. *assertEquals*(2,userService.getUsers().size());
          16. //verify(userRepository, times(1)).count(); // this will not work
          17. *verify*(userRepository).findAll();
          18. }
21. Now test all methods from ProductService class

sdf