Learning Consolidation

Implement CRUD by Using JPA Within a RESTful Service

Learning Objectives

- Explore JPA
- Explain the Spring Data JPA
- Implement a REST API using MySQL in the Data Layer

Java Persistence API

- The Java Persistence API (JPA) provides a mechanism for managing persistence and object-relational mapping.
- The JPA specification defines the object-relational mapping internally.
- JPA is based on the Java programming model.
- The relational mapping between classes is also maintained when it is persisted into the database.
- To store and retrieve data, simple SQL queries can be written in the application.
- All the Java classes that are to be persisted in the database are called **entities**. They are represented using the `@Entity` annotation present in the `javax.persistence` package.
- Each entity has a unique object identifier; the unique identifier, or primary key, of the entity is represented using the `@Id` annotation.

Spring Data JPA

- The Spring Data JPA makes it easy to implement a JPA-based data layer, or repositories, in a Spring REST API.
- Spring Data JPA reduces the need to write boilerplate or redundant code to exercise routine queries.
- Spring Data JPA provides built-in implementations for all these operations.
- The programmer can also write custom finder methods, which Spring can automatically implement.

Advantages of Using Spring Data JPA

No-code repository

The repository layer of the application can be free of code.

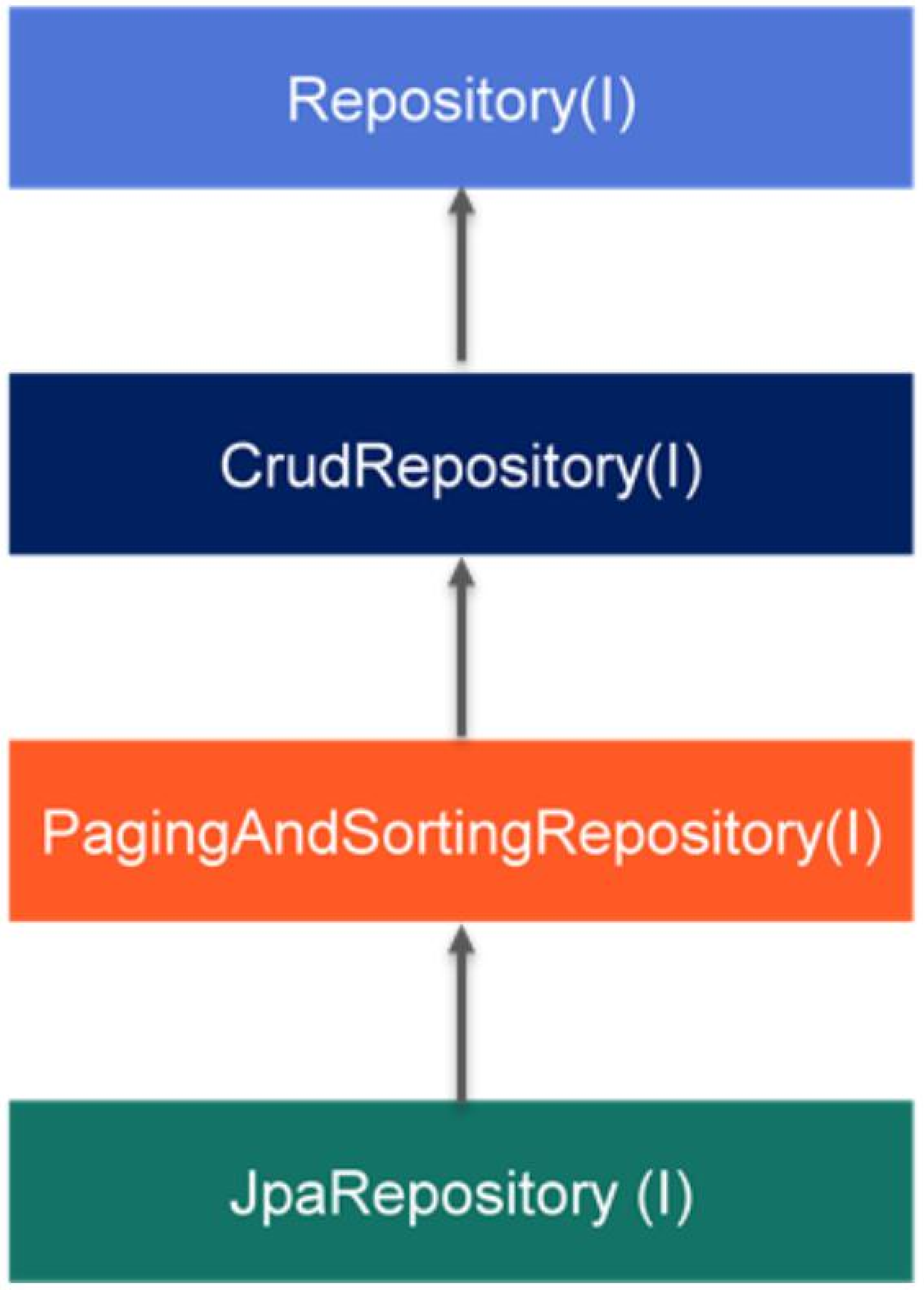
Reduced boilerplate code

It provides the default implementation for each method of the repository interfaces.

Generated Queries

Generates database queries based on the method name. If the query is complex, a method must be defined in the repository interface with a name that begins with findBy. Spring automatically parses the method name and creates a query for it.

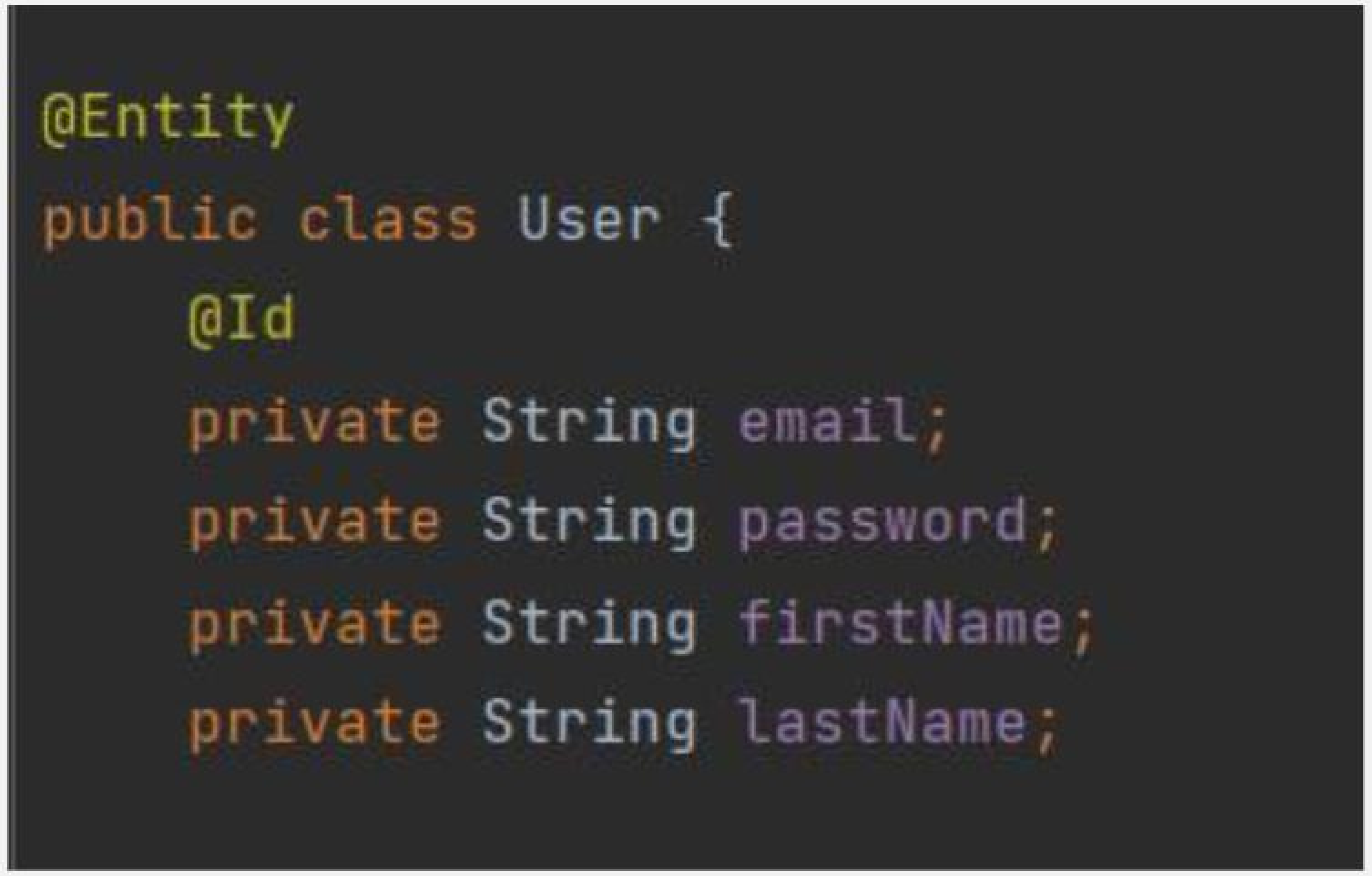
Types of Spring Data JPA Repositories



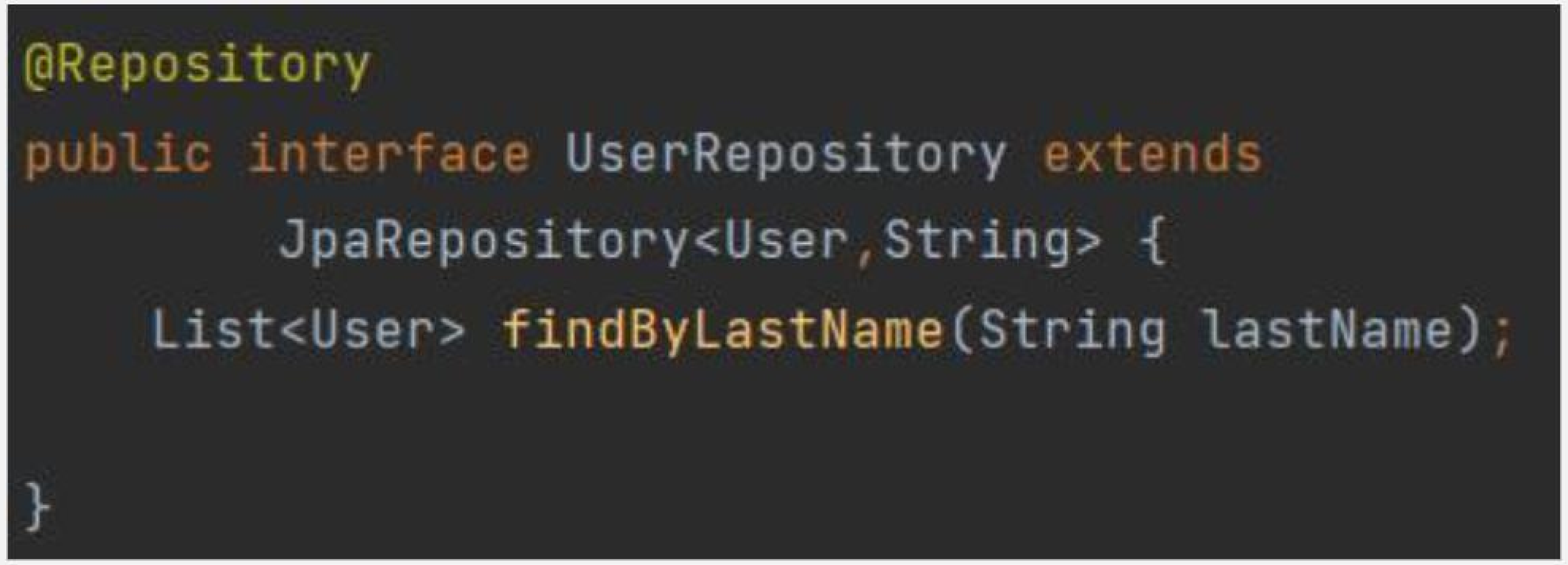
- The `Repository` is the top-level interface.
- `CrudRepository` is the child of Repository and offers standard create, read, update, and delete operations. It contains methods like findOne(), save(), delete(), etc.
- `PagingAndSortingRepository` is a child of `CrudRepository` and adds the findAll methods. It allows data to be sorted and retrieved in a paginated way.
- `JpaRepository` is a child of `PagingAndSortingRepository` and is a JPA-specific repository.
- Each interface provides functionality and can be used depending on requirements.

Domain Layer

- The domain object that needs to be persisted in the database is annotated with `@Entity`.
- The primary key or object identity field is annotated with `@Id`.
- The annotations are part of the `javax.persistence` package.



```
@Entity
public class User {
    @Id
    private String email;
    private String password;
    private String firstName;
    private String lastName;
```

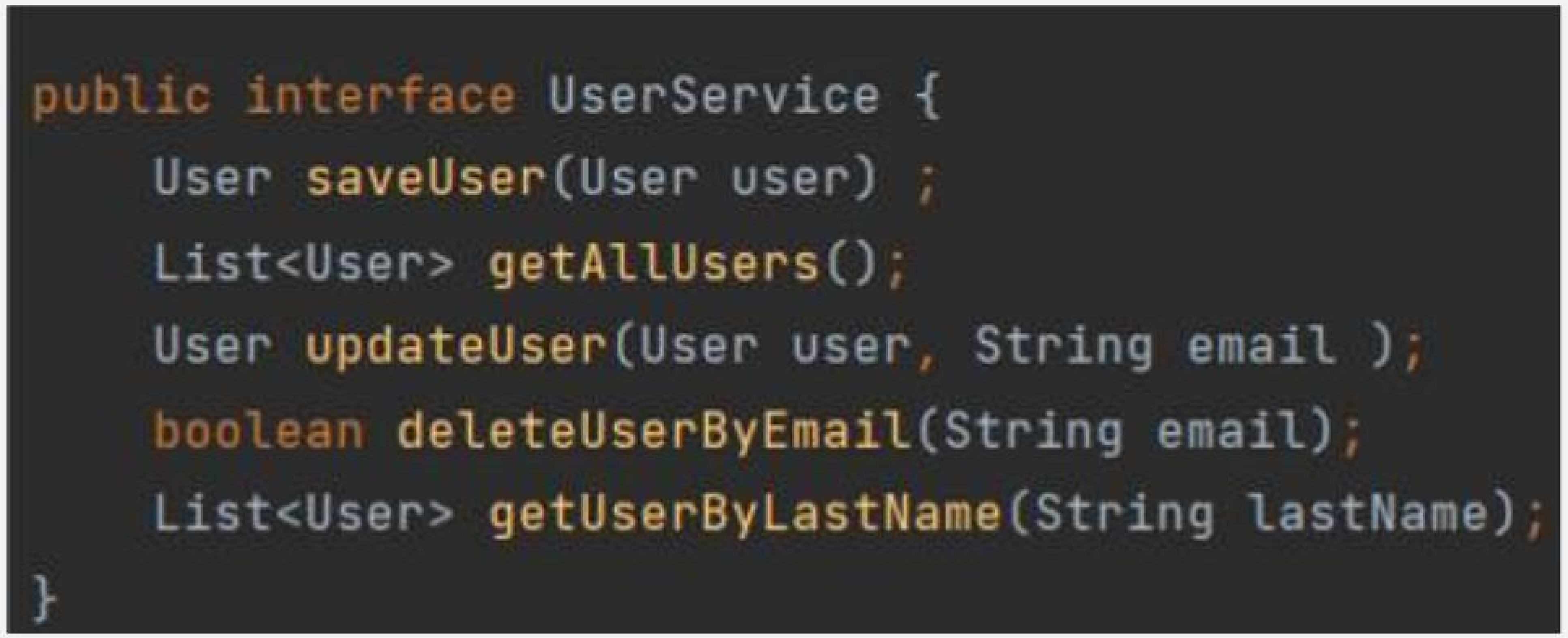


```
@Repository
public interface UserRepository extends
        JpaRepository<User,String> {
    List<User> findByLastName(String lastName);

}
```

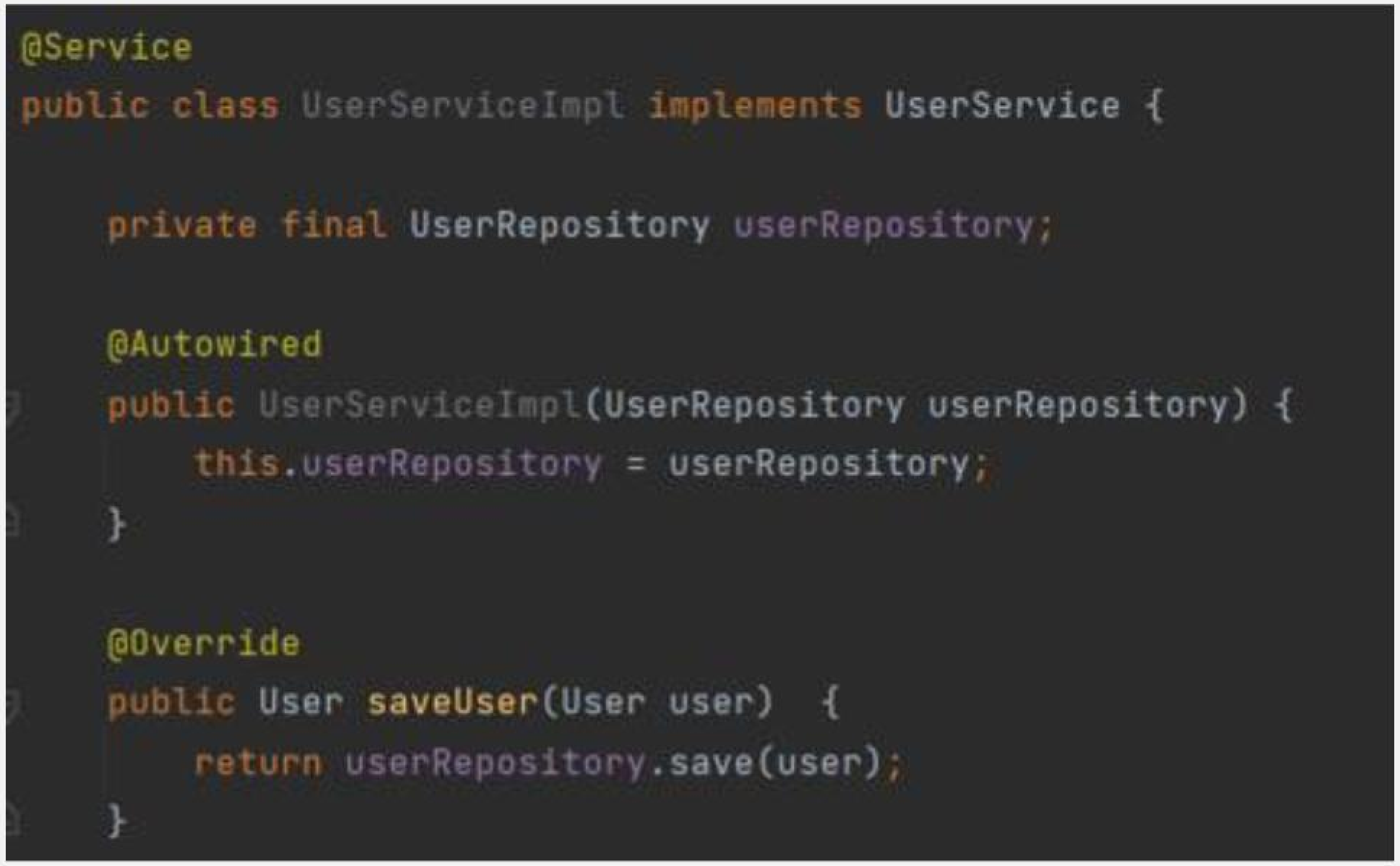
Repository Layer – User-Defined Methods

- User-defined methods can be created in the `UserRepository`.
- In the code, `findByLastName()` is a user-defined method, where `lastName` is an attribute of the domain class `User`.
- Here, "`findBy`" is a reserved phrase that Spring Data JPA will identify and provide the implementation for the method.

Check the link for more information.



```java
public interface UserService {
    User saveUser(User user) ;
    List<User> getAllUsers();
    User updateUser(User user, String email );
    boolean deleteUserByEmail(String email);
    List<User> getUserByLastName(String lastName);
}
```



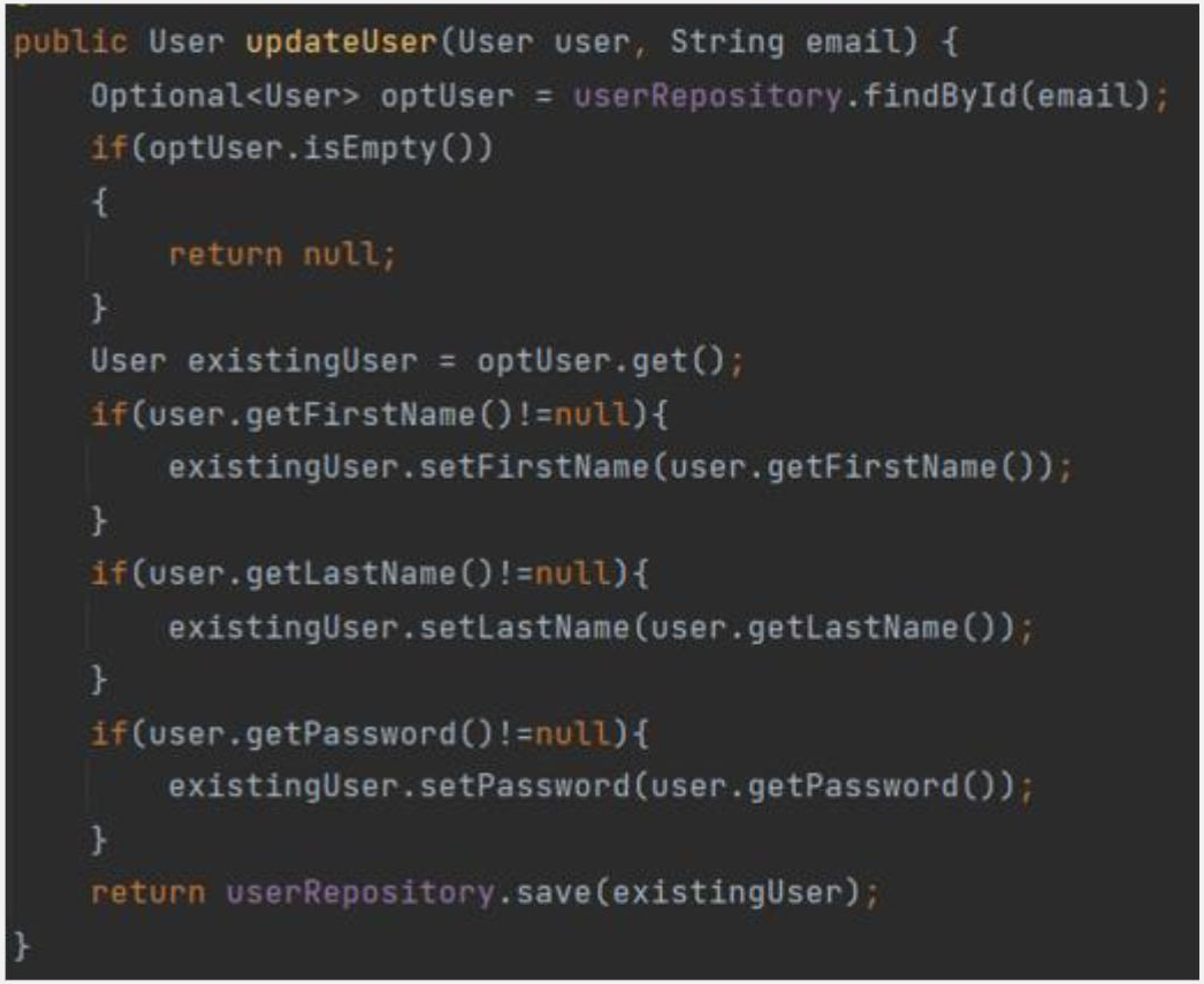
```java
@Service
public class UserServiceImpl implements UserService {

    private final UserRepository userRepository;

    @Autowired
    public UserServiceImpl(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    @Override
    public User saveUser(User user)  {
        return userRepository.save(user);
    }
```

The Service Layer

- The service layer performs the business logic necessary to provide functionality for the application.
- If a new user is registered, the information must be saved in the database, and the service layer utilizes the methods of the Repository layer to perform the save functionality.
- The programmer has no code to save the user object in the database explicitly. Still, Spring Data JPA implicitly provides functionality when the Repository layer extends the JpaRepository.

Service Layer – Update Operation

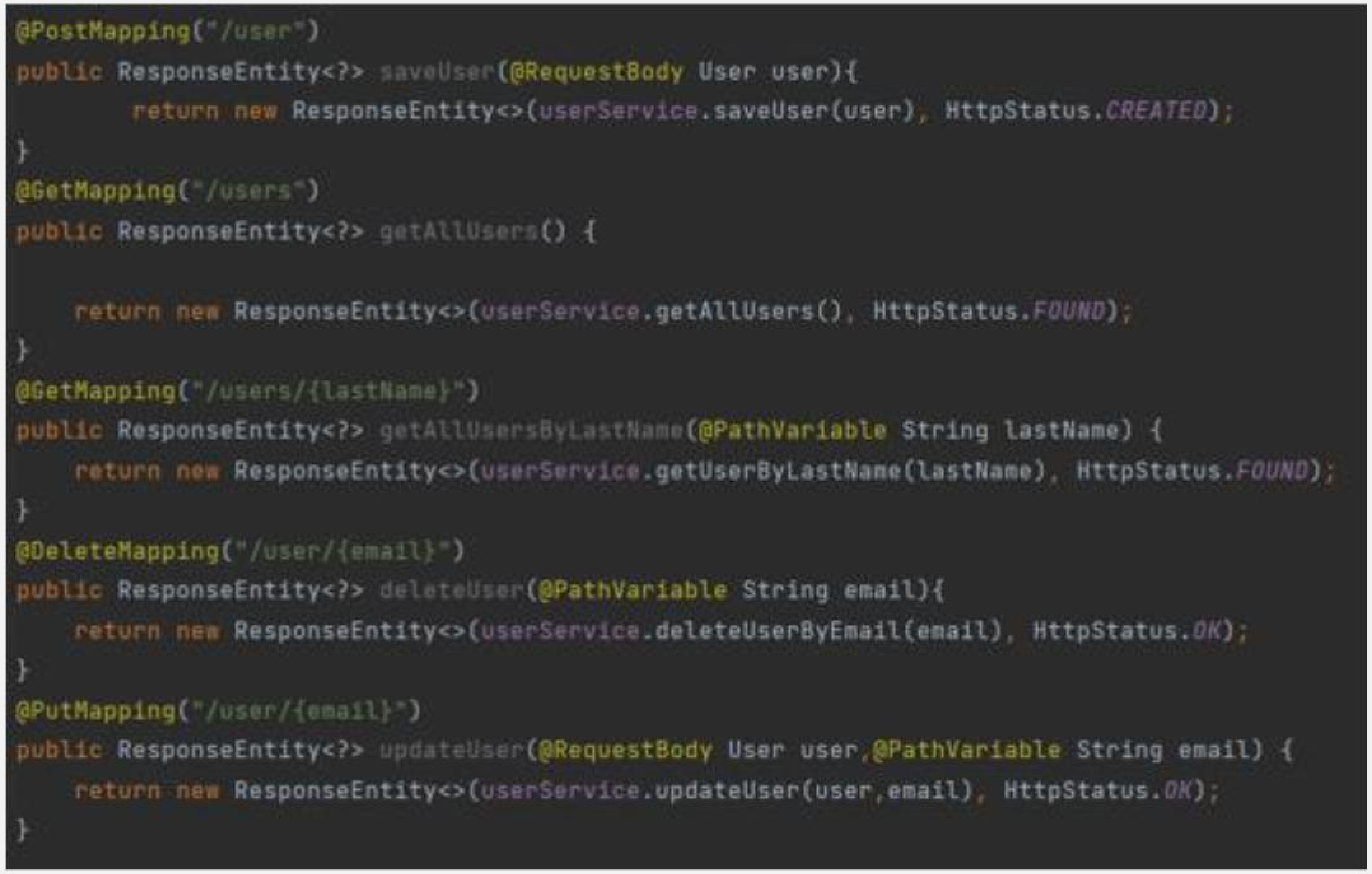
- When the details of a user need to be updated in the database, Spring Data JPA does not provide an update method.
- The programmer can write the required conditions, utilize the save method, and update the details in the database.



```java
public User updateUser(User user, String email) {
    Optional<User> optUser = userRepository.findById(email);
    if(optUser.isEmpty())
    {
        return null;
    }
    User existingUser = optUser.get();
    if(user.getFirstName()!=null){
        existingUser.setFirstName(user.getFirstName());
    }
    if(user.getLastName()!=null){
        existingUser.setLastName(user.getLastName());
    }
    if(user.getPassword()!=null){
        existingUser.setPassword(user.getPassword());
    }
    return userRepository.save(existingUser);
}
```

Controller Layer

- The adjacent image shows all the handler methods to process the GET, POST, PUT, and DELETE requests.



```java
@PostMapping("/user")
public ResponseEntity<?> saveUser(@RequestBody User user){
        return new ResponseEntity<>(userService.saveUser(user), HttpStatus.CREATED);
}
@GetMapping("/users")
public ResponseEntity<?> getAllUsers() {

    return new ResponseEntity<>(userService.getAllUsers(), HttpStatus.FOUND);
}
@GetMapping("/users/{lastName}")
public ResponseEntity<?> getAllUsersByLastName(@PathVariable String lastName) {
    return new ResponseEntity<>(userService.getUserByLastName(lastName), HttpStatus.FOUND);
}
@DeleteMapping("/user/{email}")
public ResponseEntity<?> deleteUser(@PathVariable String email){
    return new ResponseEntity<>(userService.deleteUserByEmail(email), HttpStatus.OK);
}
@PutMapping("/user/{email}")
public ResponseEntity<?> updateUser(@RequestBody User user,@PathVariable String email) {
    return new ResponseEntity<>(userService.updateUser(user,email), HttpStatus.OK);
}
```

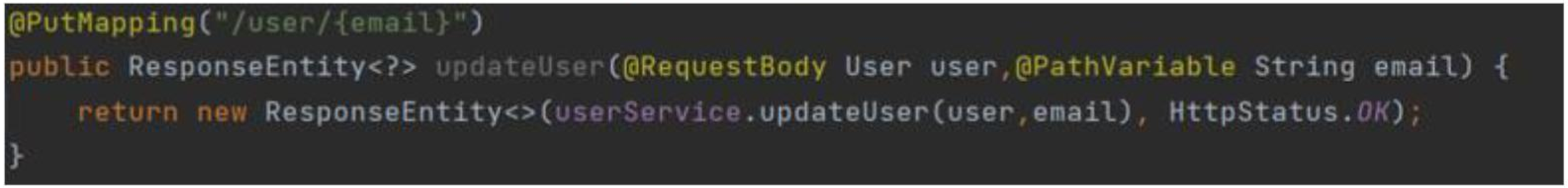
@PathVariable and @RequestBody

`@PathVariable`

- `@PathVariable` is a Spring annotation that indicates a method parameter should be bound to a URI template variable.

`@RequestBody`

- The client sends data along with the request; this data is present in the request body.
- On the application side, the data in the request body must be read and deserialized into domain objects. This is done using the `@RequestBody` annotation.



```
@PutMapping("/user/{email}")
public ResponseEntity<?> updateUser(@RequestBody User user,@PathVariable String email) {
    return new ResponseEntity<>(userService.updateUser(user,email), HttpStatus.OK);
}
```

- To update the user details, the user data must be fetched based on email and then updated; thus, the email is passed in the URI as a path variable.
- The client passes the user data to be updated in the request body.