# Spring Boot Series – Day 4

Topic: Spring Boot Annotations  (Part 2)

## Why Go Beyond Basic Annotations?

✓ Spring Boot apps aren't just wired — they serve web requests, talk to databases, and handle real-time workloads.

✓ To do all this, Spring Boot provides more powerful annotations — for REST APIs, JPA, caching, and async tasks.

✓ These annotations don't just reduce boilerplate — they unlock powerful features in just a few words.

✓ Today, we’ll explore those real-world helpers that make your code production-ready.

## @RestController – Talk JSON, Not HTML

✅ Used to build RESTful APIs — it tells Spring that this class will return data, not views.
✅ It’s a combo of @Controller + @ResponseBody.
✅ Spring doesn't try to render a JSP or Thymeleaf view — it directly sends the return object as JSON.
✅ Perfect for APIs used by frontends or mobile apps.
Example: @RestController public class BookController { @GetMapping("/books") List<Book> getAll() {…} }

# Mapping Requests Made Simple

✓ @GetMapping, @PostMapping, @PutMapping, @DeleteMapping are modern shortcuts to map HTTP methods.

✓ They replace older verbose @RequestMapping(method = ...) style.

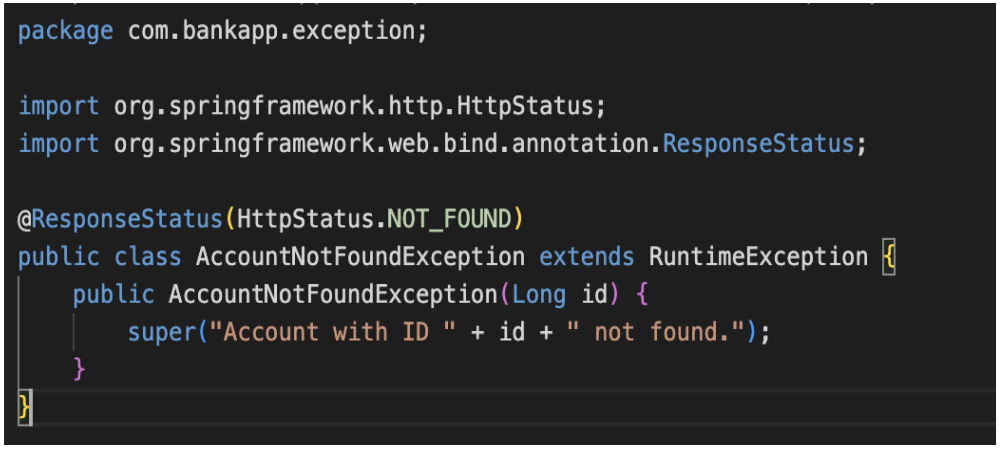
✓ Example: @PostMapping("/users") means this method handles POST requests to /users.

✓ Clean, readable, and follows REST principles.

## Making APIs Friendly – Errors, CORS, Validation

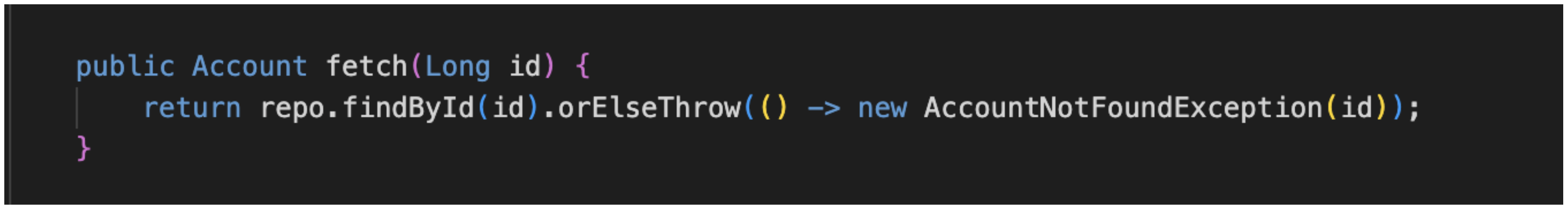
✓ **@ResponseStatus** lets you return custom HTTP status codes.

Defines a custom exception with HTTP 404 status, thrown when an account is not found.



```java
package com.bankapp.exception;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(HttpStatus.NOT_FOUND)
public class AccountNotFoundException extends RuntimeException {
    public AccountNotFoundException(Long id) {
        super("Account with ID " + id + " not found.");
    }
}
```

If no account is found, this method throws AccountNotFoundException to trigger a 404 response.

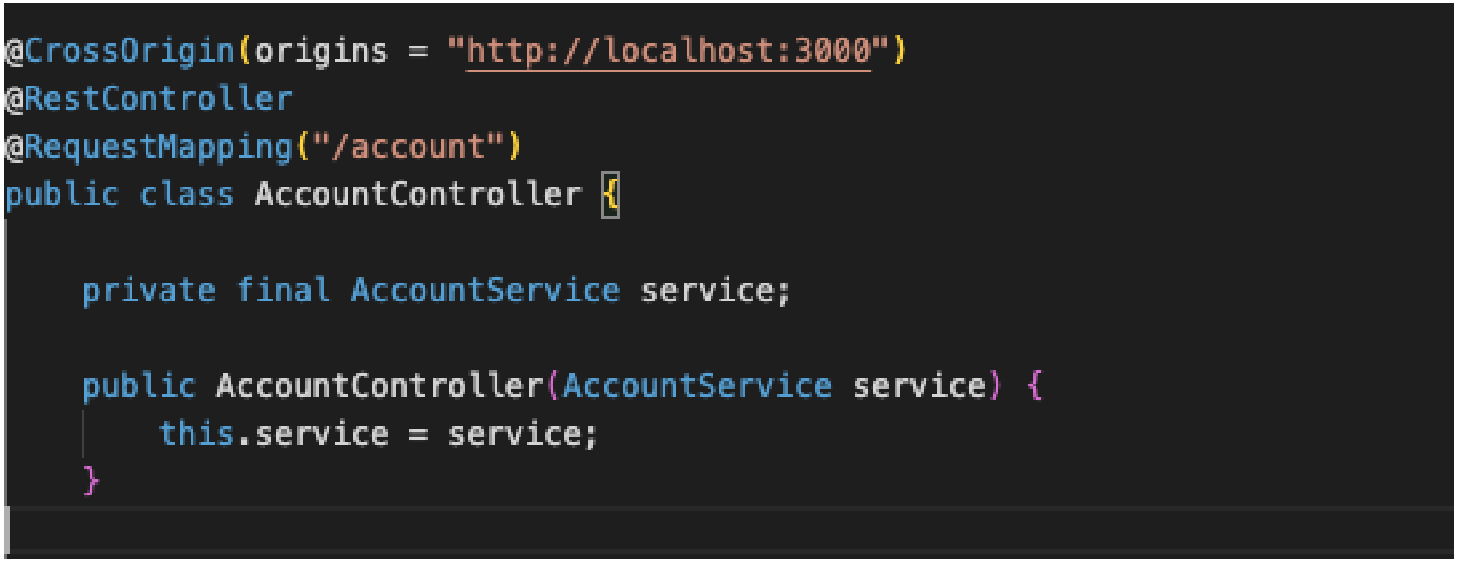


```java
    public Account fetch(Long id) {
        return repo.findById(id).orElseThrow(() -> new AccountNotFoundException(id));
    }
```

## Making APIs Friendly – Errors, CORS, Validation

✓ **@CrossOrigin** is vital for letting frontend apps (like React) call your backend.

Allows frontend apps (like React running on localhost:3000) to access these APIs without CORS errors.



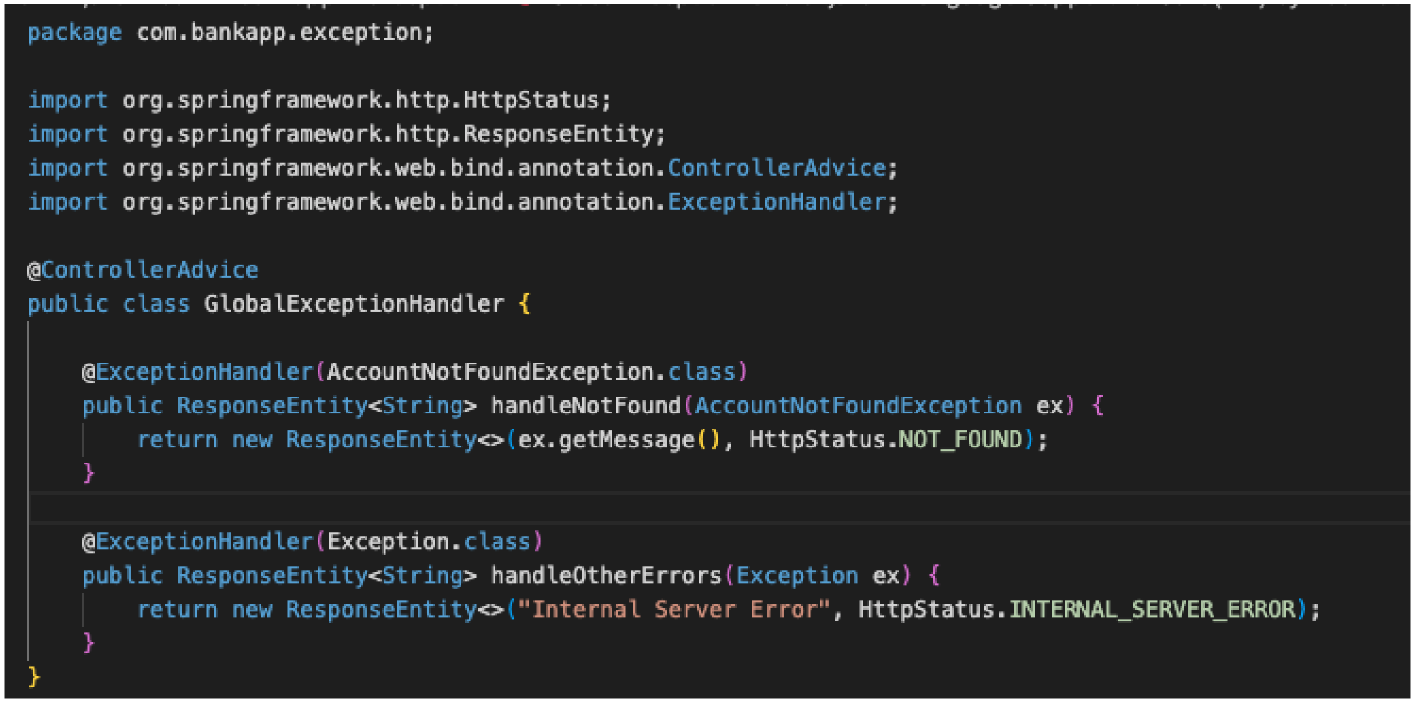
```java
@CrossOrigin(origins = "http://localhost:3000")
@RestController
@RequestMapping("/account")
public class AccountController {

    private final AccountService service;

    public AccountController(AccountService service) {
        this.service = service;
    }
```

@CodeByHaindavi | #SpringWithMe

# Making APIs Friendly – Errors, CORS, Validation

✓ **@ControllerAdvice + @ExceptionHandler** = global error handler. One place to catch all exceptions.

Centralized handler that catches exceptions like AccountNotFoundException and returns clean HTTP responses.

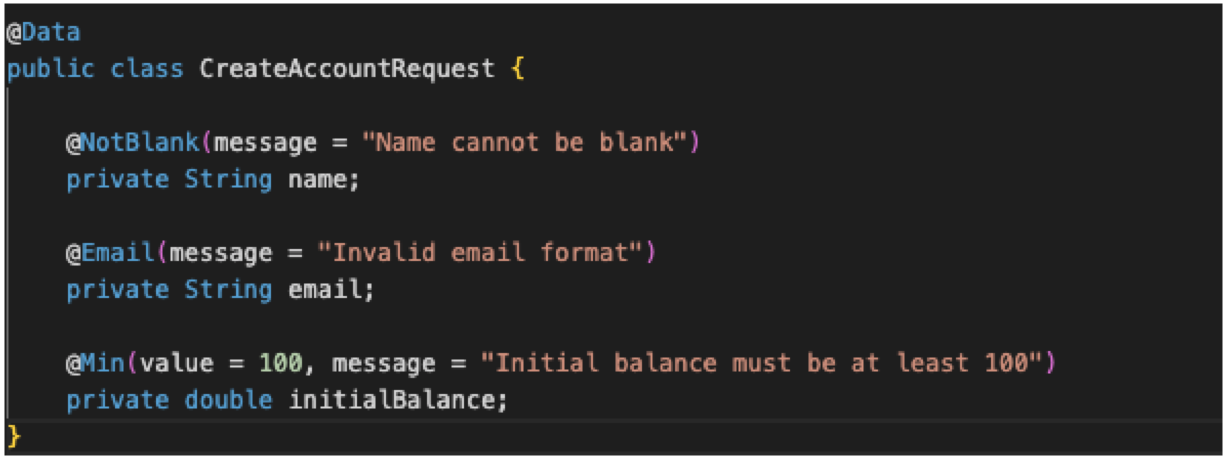


```java
package com.bankapp.exception;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;

@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(AccountNotFoundException.class)
    public ResponseEntity<String> handleNotFound(AccountNotFoundException ex) {
        return new ResponseEntity<>(ex.getMessage(), HttpStatus.NOT_FOUND);
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<String> handleOtherErrors(Exception ex) {
        return new ResponseEntity<>("Internal Server Error", HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

# Making APIs Friendly – Errors, CORS, Validation

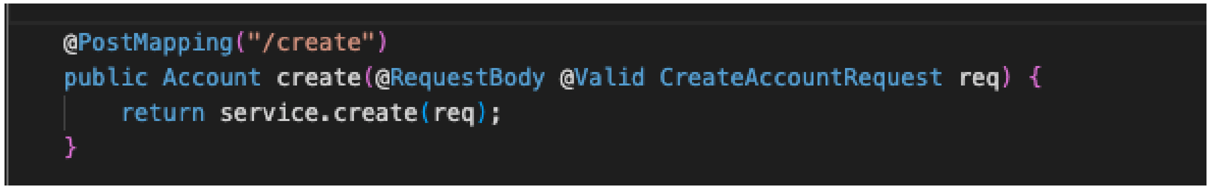
✓ **@Valid** ensures user input is validated — works with annotations like @NotNull or @Email on DTOs.

Defines validation rules for incoming account data using annotations like @NotBlank, @Email, and @Min.



```java
@Data
public class CreateAccountRequest {

    @NotBlank(message = "Name cannot be blank")
    private String name;

    @Email(message = "Invalid email format")
    private String email;

    @Min(value = 100, message = "Initial balance must be at least 100")
    private double initialBalance;
}
```

This controller method validates the request body before calling the service. If input is invalid, Spring auto-returns a 400 error with messages.

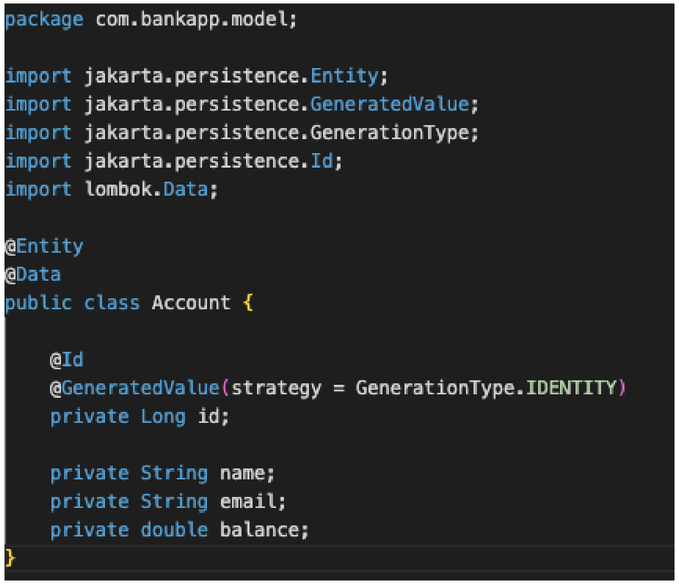


```java
    @PostMapping("/create")
    public Account create(@RequestBody @Valid CreateAccountRequest req) {
        return service.create(req);
    }
```

# JPA – Turning Classes into Tables

✓ **@Entity** marks a class to be saved to a DB table.
✓ **@Id** marks the primary key, and @GeneratedValue handles auto-increment.
✓ @Column customizes field-to-column mapping.
✓ No need to write SQL – just define the object, Spring + Hibernate do the rest.

*This entity maps to a database table — Spring Data JPA uses annotations to auto-generate schema and handle persistence.*

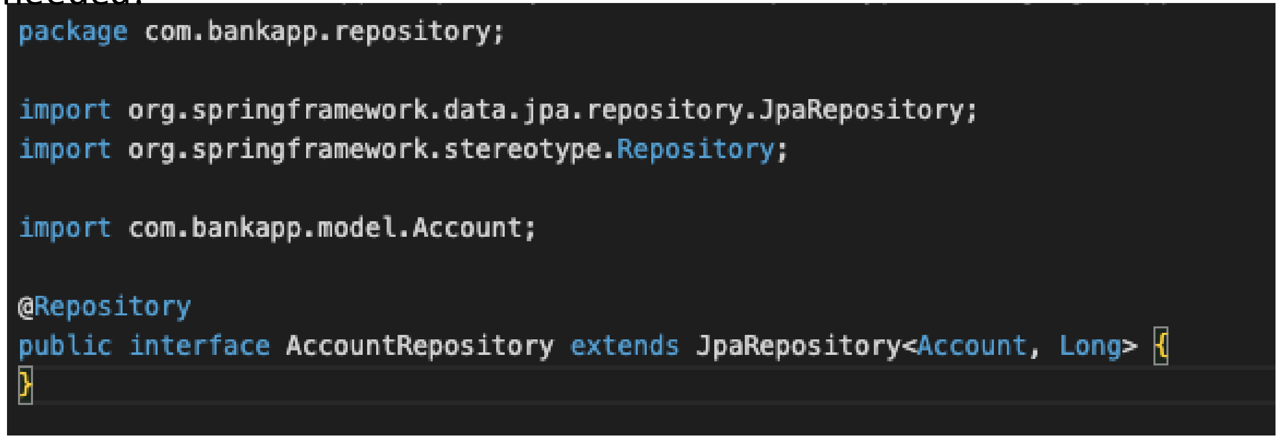


```java
package com.bankapp.model;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import lombok.Data;

@Entity
@Data
public class Account {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String email;
    private double balance;
}
```

# @Repository – Your Gateway to Data

✓ Marks a class that talks to the database.
✓ Works with Spring Data JPA interfaces like JpaRepository.
✓ Spring generates all the common DB methods for you: save, findAll, findById, delete…
✓ No need to write even a single SQL query unless you want to.

This interface connects to the database using Spring Data JPA — no SQL or implementation needed.

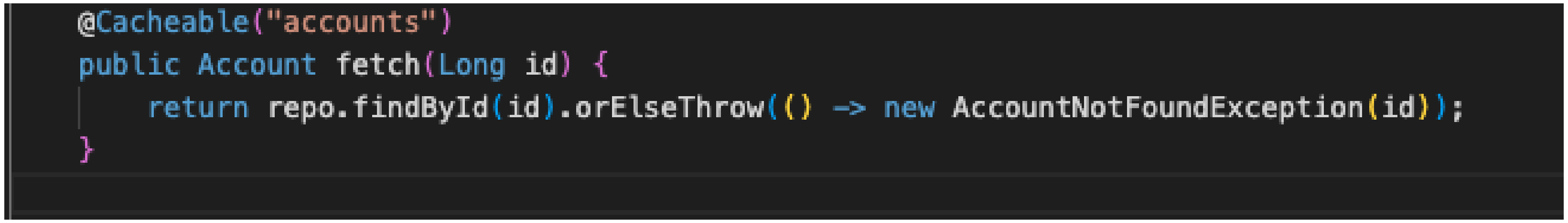


```java
package com.bankapp.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import com.bankapp.model.Account;

@Repository
public interface AccountRepository extends JpaRepository<Account, Long> {
}
```

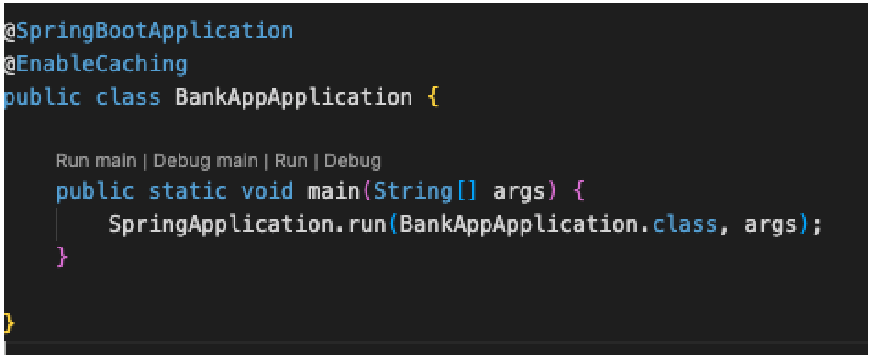
# Performance Boost – @Cacheable and Friends

- ✓ @EnableCaching turns on cache support.
- ✓ @Cacheable stores method results — so next call is instant.
- ✓ @CachePut updates cache after changes.
- ✓ @CacheEvict clears stale data.
- ✓ Used smartly, this can reduce DB load dramatically.

*The same fetch() method is now enhanced with @Cacheable — returning results from cache if already available.*



```
    @Cacheable("accounts")
    public Account fetch(Long id) {
        return repo.findById(id).orElseThrow(() -> new AccountNotFoundException(id));
    }
```



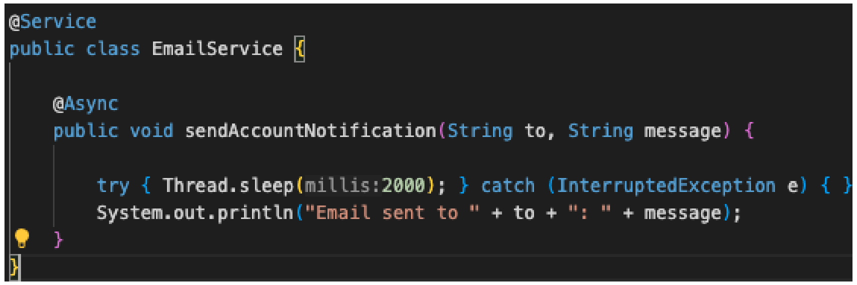
```
@SpringBootApplication
@EnableCaching
public class BankAppApplication {

    Run main | Debug main | Run | Debug
    public static void main(String[] args) {
        SpringApplication.run(BankAppApplication.class, args);
    }

}
```

@CodeByHaindavi | #SpringWithMe

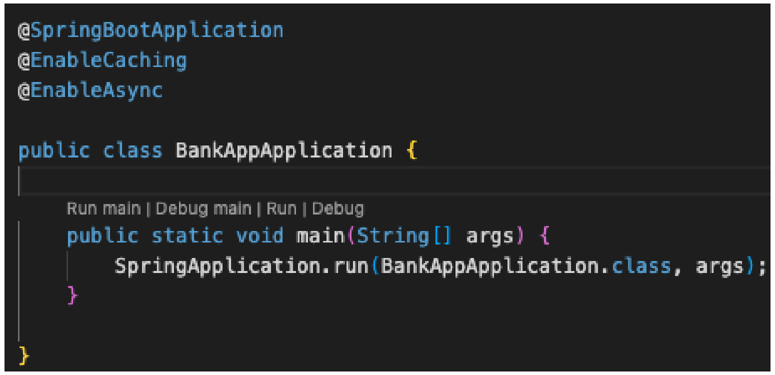
# Run in Background – @Async

- ✓ Some tasks don't need to block the user — like sending emails.
- ✓ @Async makes a method run in a separate thread.
- ✓ Add @EnableAsync to activate it globally.
- ✓ This keeps apps responsive and fast even during long tasks.

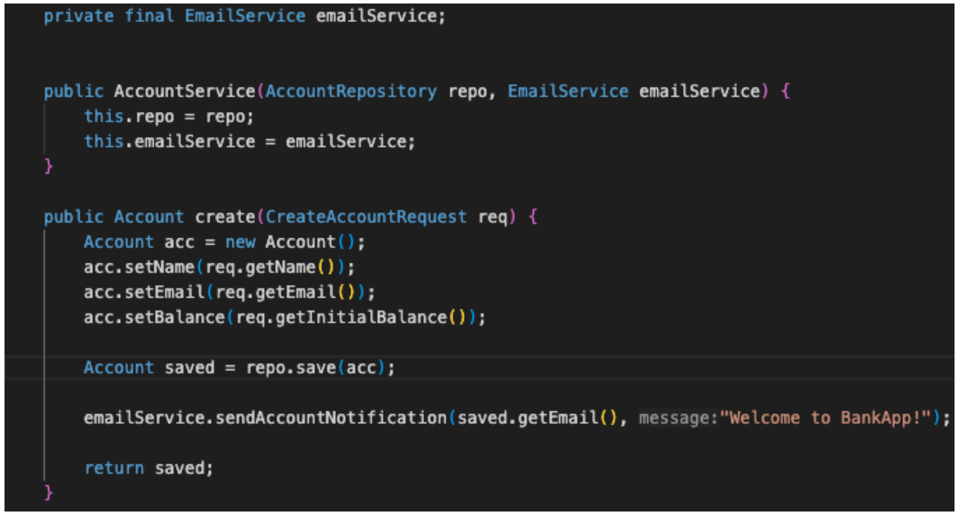
Enables background processing by making the email notification run on a separate thread.



```java
@Service
public class EmailService {

    @Async
    public void sendAccountNotification(String to, String message) {

        try { Thread.sleep(millis:2000); } catch (InterruptedException e) { }
        System.out.println("Email sent to " + to + ": " + message);
    }
}
```



```java
@SpringBootApplication
@EnableCaching
@EnableAsync

public class BankAppApplication {

    Run main | Debug main | Run | Debug
    public static void main(String[] args) {
        SpringApplication.run(BankAppApplication.class, args);
    }

}
```



```java
    private final EmailService emailService;


    public AccountService(AccountRepository repo, EmailService emailService) {
        this.repo = repo;
        this.emailService = emailService;
    }

    public Account create(CreateAccountRequest req) {
        Account acc = new Account();
        acc.setName(req.getName());
        acc.setEmail(req.getEmail());
        acc.setBalance(req.getInitialBalance());

        Account saved = repo.save(acc);

        emailService.sendAccountNotification(saved.getEmail(), message:"Welcome to BankApp!");

        return saved;
    }
```

## 💼 Interview Insight

How do @ControllerAdvice and @ExceptionHandler help in building robust REST APIs?
• They enable centralized exception handling across multiple controllers.
• @ExceptionHandler methods catch specific exceptions like AccountNotFoundException, MethodArgumentNotValidException, etc.
• With @RestControllerAdvice, JSON error responses can be customized globally (instead of duplicating try-catch logic).
• Internally, Spring uses ExceptionHandlerExceptionResolver to map exceptions to handlers.

Why do we extend JpaRepository in Spring Boot, and what benefits does it provide?
• Extending JpaRepository gives you built-in CRUD operations like save(), findById(), delete() without writing SQL.
• It also supports custom finder methods using naming conventions (e.g., findByEmail).

## 📌 Hands-On Tip

✅ Create two beans of the same type and try injecting them without @Qualifier.
✅ Write a custom exception and return it from a controller — then handle it globally using @ControllerAdvice.
✅ Temporarily remove @EnableAutoConfiguration (or comment out @SpringBootApplication) and observe how the app behaves.

## 📌 Recap – Day 4

✅ We moved beyond wiring annotations and explored how Spring Boot annotations make real-world features easy and powerful.
✅ You learned how to handle web requests, validate inputs, manage exceptions, and build cleaner APIs using annotations like @RestController, @Valid, @ExceptionHandler, and more.
✅ We mapped Java objects to database tables using @Entity, accessed data with @Repository, and injected values using both @Value and @ConfigurationProperties.
✅ We also saw how to boost performance with @Cacheable, run tasks asynchronously with @Async, and connect frontend-backend with @CrossOrigin.

👉 Coming up next: Spring Boot Actuator✨