Spring Boot -Core:

🔶 What Is Spring Boot Scheduling?

Spring Boot allows **automated execution of tasks** at scheduled times using the @Scheduled annotation — similar to cron jobs in Linux.

It is very useful when you want to:

- Run a background job (e.g., send email every hour)
- Clean up expired records
- Generate reports at fixed intervals
- Poll an API periodically

🔹 How to Enable Scheduling in Spring Boot

✅ Step 1: Add @EnableScheduling

This annotation **enables scheduling** in your application.

java

CopyEdit

```
@SpringBootApplication
@EnableScheduling
public class SchedulerApp {
    public static void main(String[] args) {
        SpringApplication.run(SchedulerApp.class, args);
    }
}
```

Without @EnableScheduling, Spring won't scan for @Scheduled methods.

✅ Step 2: Create a Scheduled Task

This is done using @Scheduled on any method inside a @Component-annotated class.

java

CopyEdit

```
@Component
public class MyScheduler {

    @Scheduled(fixedRate = 5000)
    public void printMessage() {
        System.out.println("Running scheduled task at " + new Date());
    }
}
```

🔹 Types of Scheduling in Spring Boot

Type	Explanation	Key
fixedRate	Runs every X ms, regardless of previous execution status.	Fast, non-blocking
fixedDelay	Waits until the previous execution finishes, then waits X ms to rerun.	Safe for long tasks
cron	Uses cron expressions for flexible scheduling (e.g., every Monday at 10 AM)	Highly customizable

---

🔸 Examples

🕐 1. Fixed Rate Example

java

CopyEdit

```
@Scheduled(fixedRate = 2000)
public void taskFixedRate() {
    System.out.println("Runs every 2 seconds: " + new Date());
}
```

👉 Runs the method **every 2 seconds**, no matter how long it takes.

🕒 2. Fixed Delay with Initial Delay

java

CopyEdit

```
@Scheduled(fixedDelay = 3000, initialDelay = 5000)
public void taskFixedDelay() {
    System.out.println("Runs after 5s initial delay, then every 3s after previous completes.");
}
```

⏰ 3. Cron Expression

java

CopyEdit

```
@Scheduled(cron = "0 * 19 * * ?")
public void taskCron() {
    System.out.println("Runs every minute between 19:00 and 19:59");
}
```

Cron format: second minute hour day month day-of-week
Example: 0 0/5 * * * ? → every 5 minutes

🔍 Cron Expression Reference (Simple)

Expression	Meaning
0 0 * * * *	Every hour
0 0 9 * * ?	Every day at 9 AM
*/10 * * * * *	Every 10 seconds
0 0/2 * * * ?	Every 2 minutes

💬 Common Interview Questions (With Answers)

🔸 1. How do you enable task scheduling in Spring Boot?

Add @EnableScheduling on the main class.

🔸 2. Difference between fixedRate and fixedDelay?

fixedRate	fixedDelay
Starts every X ms, regardless	Waits for completion + X ms delay
Good for quick tasks	Better for long or resource-heavy tasks

🔸 3. What is initialDelay?

It's the delay before the first execution of a scheduled method.

🔸 4. How do you schedule a task at specific time(s)?

Use @Scheduled(cron = "...") with a cron expression.

🔸 5. Can we schedule multiple tasks?

Yes, you can have multiple @Scheduled methods in the same or different classes.

🔧 Bonus Tips for Real Projects

- ❗ Always make sure the method is public and **returns void**.
- ✅ Combine scheduled tasks with services for clean code separation.
- 📈 Monitor scheduled tasks for failures (use logging, monitoring tools).
- 📏 For production-grade scheduling, consider Quartz Scheduler for more control and persistence.

📌 Conclusion

Spring Boot scheduling with @Scheduled is an **elegant and simple way** to run background or periodic tasks without any complex configuration. It’s one of the most commonly used features in real-world applications — and a **hot topic in interviews**.

✅ Spring Boot – Sending Emails via SMTP

🔹 **Step-by-step Summary:**

1. **Add Dependency in pom.xml:**

xml

CopyEdit

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-mail</artifactId>
</dependency>
```

2. **Configure application.properties:**

properties

CopyEdit

```
spring.mail.host=smtp.gmail.com
spring.mail.port=587
spring.mail.username=your_email@gmail.com
spring.mail.password=your_app_password
spring.mail.properties.mail.smtp.auth=true
spring.mail.properties.mail.smtp.starttls.enable=true
```

🔸 *Use App Password (from Gmail settings) instead of your regular password.*

---

3. **Create EmailDetails Class:**

java

CopyEdit

```
public class EmailDetails {
    private String recipient;
    private String msgBody;
    private String subject;
    private String attachment; // optional
```

```
}
```

---

4. **Service Interface:**

java

CopyEdit

```
public interface EmailService {
   String sendSimpleMail(EmailDetails details);
   String sendMailWithAttachment(EmailDetails details);
}
```

---

5. **Service Implementation Using JavaMailSender:**

java

CopyEdit

```
@Service
public class EmailServiceImpl implements EmailService {

   @Autowired private JavaMailSender mailSender;
   @Value("${spring.mail.username}") private String sender;

   public String sendSimpleMail(EmailDetails details) {
      try {
         SimpleMailMessage message = new SimpleMailMessage();
         message.setFrom(sender);
         message.setTo(details.getRecipient());
         message.setSubject(details.getSubject());
         message.setText(details.getMsgBody());
         mailSender.send(message);
         return "Mail Sent Successfully";
      } catch (Exception e) {
```

```
            return "Error Sending Mail";
        }
    }
}
```

---

6. **Rest Controller:**

```
java
CopyEdit
@RestController
public class EmailController {

    @Autowired private EmailService emailService;

    @PostMapping("/sendMail")
    public String sendMail(@RequestBody EmailDetails details) {
        return emailService.sendSimpleMail(details);
    }
}
```

✅ **What to Say in Interview:**

"To send email in Spring Boot, I use spring-boot-starter-mail along with Gmail's SMTP. I configure credentials in application.properties, then create a service that uses JavaMailSender to send either a simple message or one with attachments. The whole flow is triggered via a REST controller."

🔧 How to Change the Default Port in Spring Boot (Complete Guide + Interview Q&A)

🔍 **Why Does Port Matter?**

When a Spring Boot app runs, it starts an embedded server (like Tomcat) by default on port **8080**.

But what if:

- Port 8080 is **already in use**?
- You want your app to run on a **different port** for a specific environment (e.g., dev/test/prod)?

That's when **changing the port** becomes essential.

💡 What Is a Port Number?

A **port number** is a virtual endpoint for sending or receiving data.

- Default HTTP: 80
- Default HTTPS: 443
- Default Spring Boot: 8080

⚠️ Only **one app** can use a port at a time.

✅ Ways to Change the Port in Spring Boot

Method	Use Case	Config Location
1. application.properties	Easiest, static	Code-level config
2. WebServerFactoryCustomizer	Dynamic, Java-based	Java class
3. Command Line / VM Option	Temporary, overrides others	Runtime only

---

🚀 Method 1: Change Port in application.properties

✅ **Step:**

In src/main/resources/application.properties, add:

properties

CopyEdit

server.port=7000

✅ **Run app:**

Spring Boot will now start at http://localhost:7000

📘 You can also use:

properties

CopyEdit

server.port=0

☝️ This tells Spring to pick a **random port** every time (useful for testing).

⚙️ **Method 2: Change Port Using Java (WebServerFactoryCustomizer)**

✅ **Create a Java Class:**

java

CopyEdit

```
import org.springframework.boot.web.server.*;
import org.springframework.stereotype.Component;


@Component
public class CustomPortConfig implements
WebServerFactoryCustomizer<ConfigurableWebServerFactory> {
   @Override
   public void customize(ConfigurableWebServerFactory factory) {
      factory.setPort(1000);  // Custom port
   }
}
```

✔️ Spring will pick this up at runtime and launch the app on port 1000.

💻 **Method 3: Change Port via Command Line / VM Options**

✅ **In terminal (Linux/macOS/Windows):**

bash

CopyEdit

```
java -jar your-app.jar --server.port=9090
```

OR

✅ **In IntelliJ / Eclipse:**

- Go to: Run > Edit Configurations
- Add to **VM options**:

ini

CopyEdit

-Dserver.port=9090

This takes highest priority — **overrides both properties file and code**.

🌐 **Bonus: Change Context Path Too**

Default context path is /. You can change it:

properties

CopyEdit

server.servlet.context-path=/myapp

Now app URL: http://localhost:7000/myapp

📌 **Summary Table**

Method	Priority	Description
Command Line (--server.port)	🔝 High	Good for quick runtime change
application.properties	🔁 Medium	Best for stable config
Java Code (WebServerFactoryCustomizer)	⚙️ Low	When dynamic logic is needed

---

✅ **Sample Output**

scss

CopyEdit

Tomcat started on port(s): 7000 (http)

🎯 **Real-World Scenario**

Your team uses:

- **Port 8080** for development
- **Port 9090** for QA server
- **Port 443** for production

You manage this by:

- Setting different application.properties per environment OR
- Passing port via CI/CD command line

👨‍💼 **Interview Questions + Answers**

🔸 **Q1. How do you change the default port in Spring Boot?**

🟢 **Answer**:
You can change the default port (8080) by:

1. Adding server.port=7000 in application.properties
2. Using WebServerFactoryCustomizer interface
3. Using command line: --server.port=9090

🔸 **Q2. How to change the context path of a Spring Boot app?**

🟢 **Answer**:
Use this property:

properties

CopyEdit

server.servlet.context-path=/api

App URL becomes: http://localhost:8080/api

🔸 **Q3. What happens if you set server.port=0?**

🟢 **Answer**:
Spring Boot will choose a **random available port**. Useful for **testing** and **port conflicts**.

🔸 **Q4. Can two Spring Boot apps run on the same port?**

🔴 **Answer**: No. Only one service can bind to a specific port at a time.

🔸 **Q5. What is WebServerFactoryCustomizer used for?**

🟢 **Answer**:
It is an interface that lets you **customize the embedded server configuration** using Java code (e.g., set port, context path, SSL, etc.).

🚀 Spring Boot Transaction Management with @Transactional – Explained Simply

🧠 What is a Transaction?

A **transaction** is a **group of operations** that must either all succeed or all fail — **Atomicity**.

🧾 **Real-life Example**: Booking a flight

- Step 1: Save passenger info
- Step 2: Make payment
- ✅ Both succeed → Booking done
- ❌ Payment fails → Passenger info should also not be saved!

❓ Why Do We Need Transaction Management?

Without transaction management:

- Passenger info may get saved
- But if payment fails, there's **inconsistent data** in DB

✅ Spring solves this using **transaction management** so that:

- If all steps succeed → Commit to DB
- If any step fails → Rollback all changes (like it never happened)

💡 The Magic: @Transactional Annotation

📌 What does it do?

@Transactional tells Spring:

"Treat this method as a transaction – either everything inside succeeds, or everything rolls back."

You can put it on:

- Methods (most common)
- Class (applies to all methods in that class)

✅ **When is it Needed?**

- When you're performing **multiple DB operations** (insert/update/delete) that are related
- Common in **service layer** where business logic lives

🔨 **Example: Employee + Address Save**

Imagine saving both:

- An Employee
- And his Address

If either save fails (e.g., address is null), the entire operation must rollback.

🛠️ **Service Layer Example (With Transaction):**

java

CopyEdit

```
@Service
public class EmployeeService {

    @Autowired
    private EmployeeRepository employeeRepo;

    @Autowired
    private AddressService addressService;

    @Transactional
```

```
    public Employee addEmployee(Employee employee) throws Exception {
        Employee savedEmp = employeeRepo.save(employee);

        Address address = new Address();
        address.setAddress("Bangalore");
        address.setEmployee(savedEmp);

        // Save address
        addressService.addAddress(address);

        return savedEmp;
    }
}
```

✅ If both employeeRepo.save() and addressRepo.save() succeed → data committed
❌ If address is null → Spring **rolls back** employee save too

---

🔥 **Common Interview Question:**

🧩 **Q:** What happens if there's an exception in the addAddress() method?

A: If it's a runtime exception (like NullPointerException), and the method is annotated with @Transactional, then **the whole transaction is rolled back** — no data is saved in the database.

📌 **Bonus Tip: Rollback on Checked Exceptions**

By default, @Transactional only rolls back for **unchecked exceptions**.

If you want rollback for **any exception**, add:

java

CopyEdit

@Transactional(rollbackFor = Exception.class)

🧠 Smart to mention this in interviews – shows you understand deeper control.

🤖 Do I Need @EnableTransactionManagement?

- If you're using spring-boot-starter-data-jpa, **NO need** to add it. Spring auto-configures.
- If you're not using JPA or need custom config, add:

java

CopyEdit

@EnableTransactionManagement

📌 How to Explain in Interview

🔔 Elevator Pitch:

"I use @Transactional to ensure that operations like saving employee and address are treated as one unit. If any step fails, the entire operation rolls back, keeping data consistent. Spring handles this elegantly using proxies and connection management under the hood."

🧠 Quick Q&A for Interview

Q1: Where do you apply @Transactional?
👉 Typically on **service layer methods**, since they contain business logic and interact with multiple repositories.

Q2: What if only one operation fails inside a transaction?
👉 All DB operations in the method will be **rolled back** automatically.

Q3: Will it rollback for all exceptions?
👉 No. By default, it rolls back only for **unchecked (runtime) exceptions**. To rollback for **checked exceptions**, specify rollbackFor = Exception.class.

Q4: What happens if you don't use @Transactional?
👉 Each save() will commit individually, leading to **partial data** and inconsistency if one fails.

🧪 Real-World Scenario

Imagine this method without @Transactional:

java

CopyEdit

Employee savedEmp = employeeRepo.save(emp);

Address address = null;

address.setAddress("Somewhere"); // Will throw NPE

addressRepo.save(address);

Result? Employee gets saved, Address doesn't → Bad!

But with @Transactional, nothing gets saved! ✅

🎯 Conclusion

"Using @Transactional, I ensure atomicity in my business logic. It's especially useful in real-world cases like booking systems, financial transactions, or any multi-step DB operation where data consistency is critical. I also understand how Spring handles rollback and how to extend it using rollbackFor."

✅ Mapping Entity to DTO using ModelMapper in Spring Boot

🚀 Why do we need DTOs?

In a typical Spring Boot app, we have:

- 👨‍💻 **Web Layer** (controllers, REST APIs)
- ⚙️ **Service Layer** (business logic)
- 🗄️ **Database Layer** (entities, repositories)

But the **Entity** (e.g. User) often contains:

- Sensitive info (e.g., password)
- Technical fields (e.g., database IDs, timestamps)
- Internal data we don't want to expose in APIs

💡 **Problem**: If we return an Entity directly from the API, we might accidentally expose sensitive data like **passwords** or **IDs**.

✅ Solution: Use DTO (Data Transfer Object)

- A **DTO** is a plain object created to **carry only required data** between layers (usually service → web).
- Think of it as a **"filter"** between your internal system and the outside world.

⚒️ How ModelMapper Helps?

- ModelMapper is a library that **automatically copies values** from one object to another when the field names match.
- Instead of writing boilerplate code (dto.setName(entity.getName())), ModelMapper does it in **one line**.

🧠 Interview-friendly Real-Life Example:

Imagine you're building a **User API**:

java

CopyEdit

```
@Entity
public class User {
    private int id;
    private String name;
    private String email;
    private String password;
}
```

You don't want to expose id and password in the API response. So, you create:

java

CopyEdit

```
public class UserDto {
    private String name;
    private String email;
}
```

✅ How to Use ModelMapper – Step-by-Step

🛠️ 1. Add ModelMapper dependency in pom.xml

xml

CopyEdit

```
<dependency>
  <groupId>org.modelmapper</groupId>
  <artifactId>modelmapper</artifactId>
  <version>3.1.1</version>
</dependency>
```

⚙️ 2. Define Bean in Main Class

java

CopyEdit

```
@SpringBootApplication
public class ModelMapperApp {
    public static void main(String[] args) {
        SpringApplication.run(ModelMapperApp.class, args);
    }

    @Bean
    public ModelMapper modelMapper() {
        return new ModelMapper();
    }
}
```

🖍️ 3. Modify Your Service

java

CopyEdit

```
@Service
public class UserServiceImpl implements UserService {

    @Autowired
    private UserRepository userRepository;
```

```
    @Autowired
    private ModelMapper modelMapper;

    @Override
    public UserDto getUser(int userId) {
        User user = userRepository.findById(userId).get();

        // magic happens here
        UserDto userDto = modelMapper.map(user, UserDto.class);

        return userDto;
    }
}
```

📦 4. Controller Example

```
java
CopyEdit
@RestController
@RequestMapping("/api/user")
public class UserController {

    @Autowired
    private UserService userService;

    @GetMapping("/get/{id}")
    public ResponseEntity<UserDto> getUser(@PathVariable("id") int id) {
        return ResponseEntity.ok(userService.getUser(id));
    }
```

}

✅ **What to Say in Interviews:**

"In a typical Spring Boot application, I avoid exposing entity classes directly through APIs, especially when dealing with sensitive fields like passwords. Instead, I use DTOs to expose only required data.

To make the mapping clean and automatic, I use **ModelMapper**, which copies data between objects based on field names. It avoids boilerplate code and helps maintain a clean separation between layers.

This improves **security**, **encapsulation**, and keeps the API response lightweight."

🔁 **Bonus: If Interviewer Asks More**

Q: What if field names are different?

🧠 You can customize ModelMapper:

java

CopyEdit

modelMapper.typeMap(User.class, UserDto.class)

.addMapping(User::getFullName, UserDto::setName);

Q: Is ModelMapper better than MapStruct?

✅ **ModelMapper** is dynamic and good for quick setups.
✅ **MapStruct** is compile-time safe and faster in large-scale apps.
📌 *In interviews, mention both to show awareness.*

🏆 **Summary (1-liner to end with confidence):**

"Using ModelMapper with DTOs in Spring Boot ensures data security, keeps API responses clean, and avoids repetitive boilerplate mapping logic."

🌱 What Is Validation in Spring Boot?

Validation means **checking if user input is correct or acceptable**.

🛡️ For example:

- Name should not be empty
- Age should be between 18 and 60
- Email must be in valid format

Spring Boot makes this **super easy** using the **Hibernate Validator**, the official implementation of **Bean Validation (JSR 380)**.

🎯 **Why Is It Important?**

Without validation:

- Your app may **crash** or behave **unexpectedly**
- Hackers might **inject invalid data**
- Users may face **bad UX**

💬 **Interview Answer Tip**:

"Validation helps protect your application from invalid or harmful input and improves both user experience and system integrity."

✅ **Basic Setup (3 Steps)**

🧱 **1. Add the Dependency in pom.xml**

xml

CopyEdit

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-validation</artifactId>

</dependency>

🎯 This gives us all validation annotations like @NotNull, @Email, @Min, etc.

📦 **2. Create a Validated Bean**

java

CopyEdit

```
@Data
public class GeekEmployee {

    @NotNull(message = "Employee ID cannot be null")
    private Long empId;

    @NotBlank(message = "Name is required")
    private String name;

    @Min(value = 18, message = "Minimum age is 18")
    @Max(value = 60, message = "Maximum age is 60")
    private int age;

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

    @Pattern(regexp = "^[0-9]{5}$", message = "Postal code must be 5 digits")
    private String postalCode;
}
```

🧪 3. Use @Valid in Controller

java

CopyEdit

```
@PostMapping("/add")
public ResponseEntity<String> addEmployee(@Valid @RequestBody GeekEmployee employee) {
    return ResponseEntity.ok("Employee added!");
}
```

✨ Boom! Spring Boot automatically checks all fields based on annotations.

🧠 **Must-Know Validator Annotations (With Tips)**

Annotation	Use It When…
@NotNull	Value must be present (can be empty though)
@NotEmpty	Value must not be null or empty
@NotBlank	For strings – must not be just spaces
@Email	Must be a valid email format
@Min/@Max	Numbers must be within a range
@Size	String/list must be within size limits
@Pattern	Must match a custom regex pattern

📌 Pro Tip: You can **combine** these:

java

CopyEdit

@Email

@NotBlank

private String email;

🚨 **How to Handle Validation Errors (Like a Pro)**

If validation fails, Spring throws a MethodArgumentNotValidException.

You can handle it **globally** using @ControllerAdvice:

java

CopyEdit

@ControllerAdvice

public class GlobalExceptionHandler extends ResponseEntityExceptionHandler {

    @Override

    protected ResponseEntity<Object> handleMethodArgumentNotValid(

        MethodArgumentNotValidException ex, HttpHeaders headers,

```
        HttpStatus status, WebRequest request) {

        Map<String, Object> errorBody = new HashMap<>();
        errorBody.put("timestamp", Instant.now());
        errorBody.put("status", status.value());
        errorBody.put("errors", ex.getBindingResult()
                .getFieldErrors()
                .stream()
                .map(FieldError::getDefaultMessage)
                .collect(Collectors.toList()));

        return new ResponseEntity<>(errorBody, headers, status);
    }
}
```

🔥 Now the client gets a clean, readable error response:

```
json
CopyEdit
{
  "timestamp": "2025-07-19T10:25:00Z",
  "status": 400,
  "errors": [
    "Email cannot be null",
    "Postal code must be 5 digits"
  ]
}
```

🎤 **What to Say in an Interview:**

"In Spring Boot, I use Hibernate Validator for user input validation using annotations like @NotNull, @Email, and @Size.
I annotate my DTOs or model classes, and use @Valid in the controller to trigger validation

automatically.
If validation fails, I handle it using @ControllerAdvice and customize the error response for better UX and debugging."

🏆 **BONUS: Impress the Interviewer**

- ✅ Say: “Using annotations makes the code **declarative and readable**.”
- ✅ Say: “It enforces **robust input contracts** between frontend and backend.”
- ✅ Say: “I use custom validation messages to give users meaningful feedback.”
- 🚀 Extra: “For complex validations (e.g., cross-field), I use @AssertTrue or custom constraint annotations.”