(IntelliJ IDEA, Eclipse, VS Code).

STEP 2: Configure Mail Server Properties

We need to tell Spring Boot how to connect to your SMTP (Simple Mail Transfer Protocol) server. We'll use Gmail's SMTP settings as an example, but you can replace them with your own mail server details.

Important Note for Gmail: If you use Gmail, you might need to generate an "App password" instead of using your regular Gmail password, especially if you have 2-Step Verification enabled. Go to your Google Account -> Security -> How you sign in to Google -> App passwords.

1. **Open application.properties:**
   - Navigate to src/main/resources/ and open application.properties.
2. **Add Mail Server Configuration:**
   - Add the following properties. **Replace your-email@gmail.com and your-app-password with your actual Gmail address and generated app password.**

```
# application.properties
# SMTP Host (e.g., Gmail's SMTP server)
spring.mail.host=smtp.gmail.com
# SMTP Port (587 for TLS, 465 for SSL)
spring.mail.port=587
# Your Gmail address
spring.mail.username=your-email@gmail.com
# Your Gmail App Password (or regular password if 2FA is off, not recommended)
spring.mail.password=your-app-password
# Enable authentication
spring.mail.properties.mail.smtp.auth=true
# Enable STARTTLS for secure connection
spring.mail.properties.mail.smtp.starttls.enable=true
# Optional: for debugging mail sessions
# spring.mail.properties.mail.debug=true
```

STEP 3: Create a Mail Service for Sending Emails

We'll create a service class that encapsulates the email sending logic using JavaMailSender.

- Create a new package com.example.mail.service in src/main/java/.
- Create a Java class named EmailService.java inside this package:

```
// src/main/java/com/example/mail/service/EmailService.java
package com.example.mail.service;

import jakarta.mail.MessagingException;
import jakarta.mail.internet.MimeMessage;
import org.springframework.beans.factory.annotation.Value; // Import for @Value annotation
import org.springframework.core.io.FileSystemResource; // For attachments
import org.springframework.mail.MailException; // Spring's mail exception
import org.springframework.mail.SimpleMailMessage; // For simple text emails
import org.springframework.mail.javamail.JavaMailSender; // Core Spring Mail interface
import org.springframework.mail.javamail.MimeMessageHelper; // For complex emails (HTML, attachments)
import org.springframework.scheduling.annotation.Async; // For asynchronous sending
import org.springframework.stereotype.Service;

import java.io.File;

/**
 * Service class for sending various types of emails.
 */
@Service
public class EmailService {

    private final JavaMailSender mailSender;

    @Value("${spring.mail.username}") // Inject the sender's email from properties
    private String senderEmail;

    // Spring automatically injects JavaMailSender configured from application.properties
    public EmailService(JavaMailSender mailSender) {
        this.mailSender = mailSender;
    }

    /**
```

```
     * Sends a simple text email.
     * @param to The recipient's email address.
     * @param subject The subject of the email.
     * @param text The plain text content of the email.
     */
    public void sendSimpleEmail(String to, String subject, String text) {
        try {
            SimpleMailMessage message = new SimpleMailMessage();
            message.setFrom(senderEmail); // Use the injected senderEmail
            message.setTo(to);
            message.setSubject(subject);
            message.setText(text);
            mailSender.send(message);
            System.out.println("Simple email sent successfully to " + to);
        } catch (MailException e) {
            System.err.println("Failed to send simple email to " + to + ": " +
e.getMessage());
            // In a real application, you might log the error, retry, or notify an admin.
        }
    }

    /**
     * Sends a rich HTML email with optional attachment.
     * @param to The recipient's email address.
     * @param subject The subject of the email.
     * @param htmlContent The HTML content of the email.
     * @param attachmentPath Optional path to a file to attach.
     */
    public void sendComplexEmail(String to, String subject, String htmlContent,
String attachmentPath) {
        try {
            MimeMessage message = mailSender.createMimeMessage();
            // true for multipart message (e.g., HTML + attachment)
            MimeMessageHelper helper = new MimeMessageHelper(message, true);

            helper.setFrom(senderEmail); // Use the injected senderEmail
            helper.setTo(to);
            helper.setSubject(subject);
            helper.setText(htmlContent, true); // true indicates HTML content

            if (attachmentPath != null && !attachmentPath.isEmpty()) {
```

```
            File attachment = new File(attachmentPath);
            if (attachment.exists()) {
                FileSystemResource file = new FileSystemResource(attachment);
                helper.addAttachment(file.getFilename(), file);
                System.out.println("Attached file: " + file.getFilename());
            } else {
                System.err.println("Attachment file not found: " + attachmentPath);
            }
        }

        mailSender.send(message);
        System.out.println("Complex email sent successfully to " + to);
    } catch (MessagingException | MailException e) {
        System.err.println("Failed to send complex email to " + to + ": " +
e.getMessage());
        e.printStackTrace(); // Print stack trace for more details on
MessagingException
    }
}

/**
 * Sends a simple text email asynchronously.
 * The method will return immediately, and email sending will happen in a
separate thread.
 * @param to The recipient's email address.
 * @param subject The subject of the email.
 * @param text The plain text content of the email.
 */
@Async // Marks this method to be executed in a separate thread
public void sendSimpleEmailAsync(String to, String subject, String text) {
    System.out.println("Attempting to send simple email asynchronously to " + to +
" from thread: " + Thread.currentThread().getName());
    sendSimpleEmail(to, subject, text); // Re-use the synchronous sending logic
}

/**
 * Sends a rich HTML email with optional attachment asynchronously.
 * @param to The recipient's email address.
 * @param subject The subject of the email.
 * @param htmlContent The HTML content of the email.
 * @param attachmentPath Optional path to a file to attach.
```

```
     */
    @Async
    public void sendComplexEmailAsync(String to, String subject, String
htmlContent, String attachmentPath) {
        System.out.println("Attempting to send complex email asynchronously to " + to
+ " from thread: " + Thread.currentThread().getName());
        sendComplexEmail(to, subject, htmlContent, attachmentPath); // Re-use the
synchronous sending logic
    }
}
```

STEP 4: Create a REST Controller to Trigger Email Sending

We'll create a simple REST controller to easily trigger email sending via HTTP requests.

- Create a new package com.example.mail.controller in src/main/java/.
- Create a Java class named MailController.java inside this package:

```
// src/main/java/com/example/mail/controller/MailController.java
package com.example.mail.controller;

import com.example.mail.service.EmailService;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

/**
 * REST Controller to trigger email sending for demonstration.
 */
@RestController
public class MailController {

    private final EmailService emailService;

    public MailController(EmailService emailService) {
```

```
        this.emailService = emailService;
    }

    /**
     * Endpoint to send a simple text email.
     * Example: GET http://localhost:8080/send-
simple?to=recipient@example.com&subject=Test&body=Hello
     * @param to Recipient email address.
     * @param subject Email subject.
     * @param body Email body text.
     * @return Confirmation message.
     */
    @GetMapping("/send-simple")
    public ResponseEntity<String> sendSimple(@RequestParam String to,
                                             @RequestParam String subject,
                                             @RequestParam String body) {
        emailService.sendSimpleEmail(to, subject, body);
        return ResponseEntity.ok("Simple email sending initiated to " + to);
    }

    /**
     * Endpoint to send a simple text email asynchronously.
     * Example: GET http://localhost:8080/send-simple-
async?to=recipient@example.com&subject=AsyncTest&body=HelloAsync
     * @param to Recipient email address.
     * @param subject Email subject.
     * @param body Email body text.
     * @return Confirmation message.
     */
    @GetMapping("/send-simple-async")
    public ResponseEntity<String> sendSimpleAsync(@RequestParam String to,
```

```
                                    @RequestParam String subject,
                                    @RequestParam String body) {
        emailService.sendSimpleEmailAsync(to, subject, body);
        return ResponseEntity.ok("Asynchronous simple email sending initiated to " +
to + ". Check console for thread info.");
    }

    /**
     * Endpoint to send a complex HTML email with an optional attachment.
     * Example: GET http://localhost:8080/send-
complex?to=recipient@example.com&subject=HTMLTest&body=<b>Hello</b><i>H
TML</i>&attach=true
     * @param to Recipient email address.
     * @param subject Email subject.
     * @param body Email HTML content.
     * @param attach Whether to include an attachment (true/false).
     * @return Confirmation message.
     */
    @GetMapping("/send-complex")
    public ResponseEntity<String> sendComplex(@RequestParam String to,
                                    @RequestParam String subject,
                                    @RequestParam String body,
                                    @RequestParam(defaultValue = "false") boolean
attach) {
        String htmlContent = "<html><body><h1>" + body + "</h1><p>This is an
<b>HTML</b> email from Spring Boot!</p></body></html>";
        String attachmentPath = null;
        if (attach) {
            // Create a dummy file for attachment
            // Ensure this file exists in src/main/resources or a path accessible to the
application
```

```
            // For demonstration, let's assume a file named 'dummy-attachment.txt' in
resources
            // You might need to create this file manually for the demo to work.
            // Path will be relative to the project root or classpath.
            attachmentPath = "src/main/resources/dummy-attachment.txt";
            // Or for a file directly in resources:
            // attachmentPath = getClass().getClassLoader().getResource("dummy-
attachment.txt").getFile();
        }
        emailService.sendComplexEmail(to, subject, htmlContent, attachmentPath);
        return ResponseEntity.ok("Complex email sending initiated to " + to + (attach ?
" with attachment." : "."));
    }

    /**
     * Endpoint to send a complex HTML email with an optional attachment
asynchronously.
     * Example: GET http://localhost:8080/send-complex-
async?to=recipient@example.com&subject=AsyncHTMLTest&body=<b>Hello</b>
<i>AsyncHTML</i>&attach=true
     * @param to Recipient email address.
     * @param subject Email subject.
     * @param body Email HTML content.
     * @param attach Whether to include an attachment (true/false).
     * @return Confirmation message.
     */
    @GetMapping("/send-complex-async")
    public ResponseEntity<String> sendComplexAsync(@RequestParam String to,
                                                   @RequestParam String subject,
                                                   @RequestParam String body,
                                                   @RequestParam(defaultValue = "false") boolean
```

```
attach) {
        String htmlContent = "<html><body><h1>" + body + "</h1><p>This is an
<b>HTML</b> email from Spring Boot (Async)!</p></body></html>";
        String attachmentPath = null;
        if (attach) {
            attachmentPath = "src/main/resources/dummy-attachment.txt";
        }
        emailService.sendComplexEmailAsync(to, subject, htmlContent,
attachmentPath);
        return ResponseEntity.ok("Asynchronous complex email sending initiated to "
+ to + (attach ? " with attachment." : ".") + " Check console for thread info.");
    }
}
```

Action Required: For the attachment example to work, create a simple text file named dummy-attachment.txt inside your src/main/resources directory. You can put any content in it, e.g., "This is a test attachment."

STEP 5: Main Spring Boot Application Class

This is the standard Spring Boot application entry point. To enable asynchronous email sending, we need to add @EnableAsync.

- Open your main application class (e.g., SpringMailDemoApplication.java in com.example.mail).

```
// src/main/java/com/example/mail/app/SpringMailDemoApplication.java
package com.example.mail;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.scheduling.annotation.EnableAsync; // Enables
asynchronous method execution
```

```
/**
 * Main Spring Boot application class for JavaMail demo.
 * @EnableAsync: Activates Spring's asynchronous method execution capability.
 */
@SpringBootApplication
@EnableAsync // This annotation is crucial for @Async to work
public class SpringMailDemoApplication {

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

STEP 6: Run the Application and Test Email Sending

1. **Ensure Mail Server Configuration is Correct:**
    - Double-check your application.properties for spring.mail.username and spring.mail.password. Make sure the password is an "App password" if you're using Gmail with 2-Step Verification. Incorrect credentials are the most common cause of errors.
2. **Create Dummy Attachment File:**
    - If you plan to test attachments, ensure you have created src/main/resources/dummy-attachment.txt with some content.
3. **Run the Spring Boot Application:**
    - Open your main application class (SpringMailDemoApplication.java).
    - Run it as a Java Application from your IDE, or use mvn spring-boot:run from the terminal in your project root.
    - You should see Spring Boot starting on port 8080.
4. **Test Email Sending:**
    - **Open your IDE's console for the spring-mail-demo application.** This is where you'll see logs about email sending status and asynchronous thread information.

- **Open the inbox of the recipient@example.com email address** you specified in the URLs.
- **Test Sending Simple Emails:**
  - Open your web browser and go to: http://localhost:8080/send-simple?to=your-recipient-email@example.com&subject=MyFirstSpringEmail&body=Hello%20from%20Spring%20Boot%20JavaMail!
  - **Observe:** Check your application console for "Simple email sent successfully" and the recipient's inbox for the email.
- **Test Sending Rich (HTML) Emails with Attachment:**
  - Open your web browser and go to: http://localhost:8080/send-complex?to=your-recipient-email@example.com&subject=HTMLWithAttachment&body=Welcome%20to%20HTML%20Email&attach=true
  - **Observe:** Check your application console and the recipient's inbox for an HTML email with the dummy-attachment.txt file.
- **Test Asynchronous Email Sending (Simple):**
  - Open your web browser and go to: http://localhost:8080/send-simple-async?to=your-recipient-email@example.com&subject=AsyncSimpleEmail&body=This%20is%20an%20async%20simple%20email.
  - **Observe:** You should immediately see "Asynchronous simple email sending initiated..." in your browser. In the console, notice that the "Attempting to send simple email asynchronously..." message appears from a different thread (e.g., task-1). The email should still arrive in the recipient's inbox.
- **Test Asynchronous Email Sending (Complex):**
  - Open your web browser and go to: http://localhost:8080/send-complex-async?to=your-recipient-email@example.com&subject=AsyncComplexEmail&body=Async%20HTML%20Test&attach=true

    - **Observe:** Similar to the simple async email, the browser response is immediate, and the email sending occurs in a background thread.

You have successfully implemented email sending functionality in a Spring Boot application using Spring JavaMail! You've learned how to:

- Configure Spring Boot for email sending using application.properties.
- Send simple plain text emails using SimpleMailMessage.
- Send rich HTML emails and include attachments using MimeMessageHelper.
- Implement asynchronous email sending using @Async for non-blocking operations.
- Understand basic error handling for mail operations.

This activity provides a solid foundation for integrating robust email capabilities into your Spring applications.