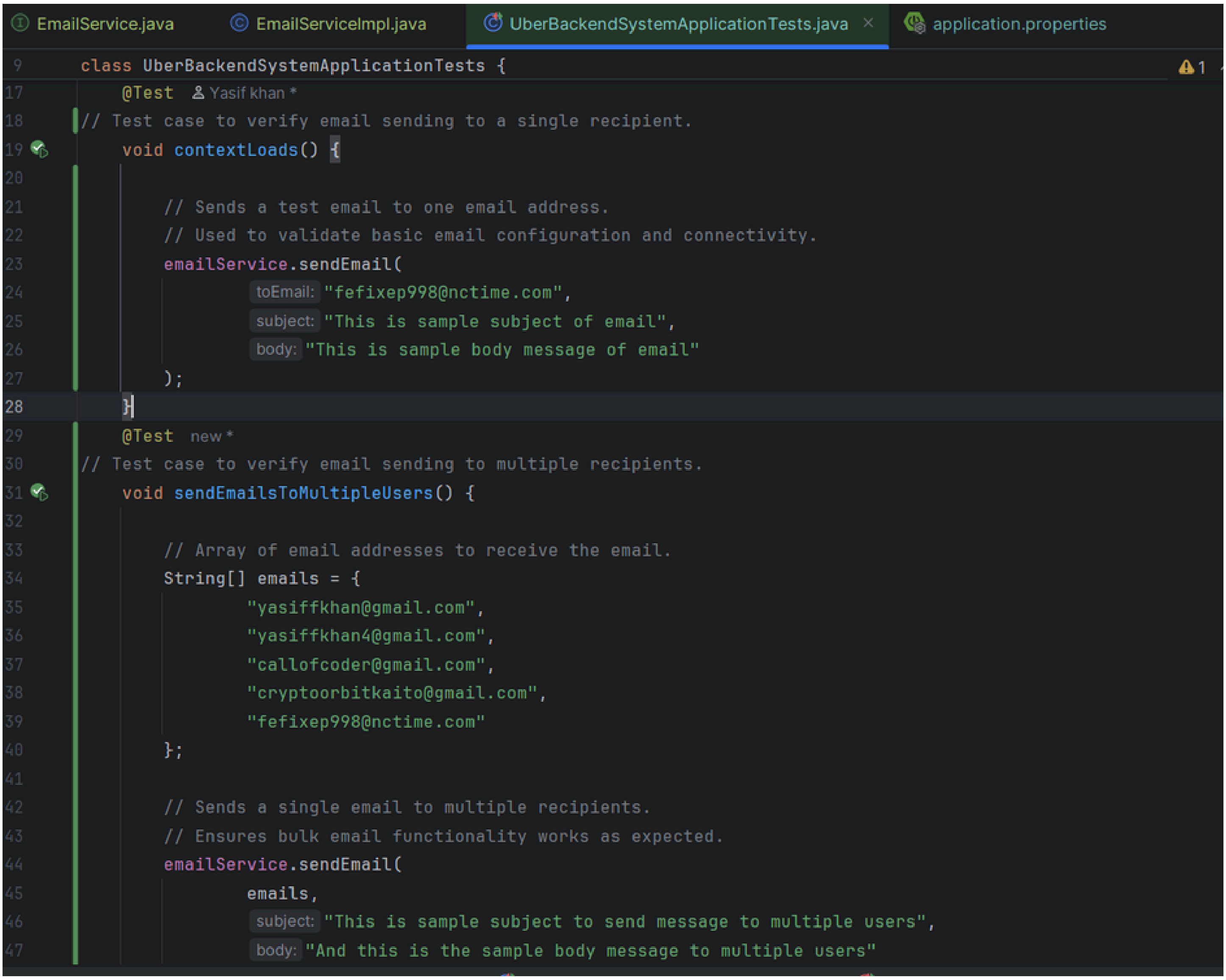
Day 10 - Project Cab booking



```
class UberBackendSystemApplicationTests {
    @Test  Yasif khan *
// Test case to verify email sending to a single recipient.
    void contextLoads() {

        // Sends a test email to one email address.
        // Used to validate basic email configuration and connectivity.
        emailService.sendEmail(
                toEmail: "fefixep998@nctime.com",
                subject: "This is sample subject of email",
                body: "This is sample body message of email"
        );
    }
    @Test  new *
// Test case to verify email sending to multiple recipients.
    void sendEmailsToMultipleUsers() {

        // Array of email addresses to receive the email.
        String[] emails = {
                "yasiffkhan@gmail.com",
                "yasiffkhan4@gmail.com",
                "callofcoder@gmail.com",
                "cryptoorbitkaito@gmail.com",
                "fefixep998@nctime.com"
        };

        // Sends a single email to multiple recipients.
        // Ensures bulk email functionality works as expected.
        emailService.sendEmail(
                emails,
                subject: "This is sample subject to send message to multiple users",
                body: "And this is the sample body message to multiple users"
```

Strategic intent of this test class

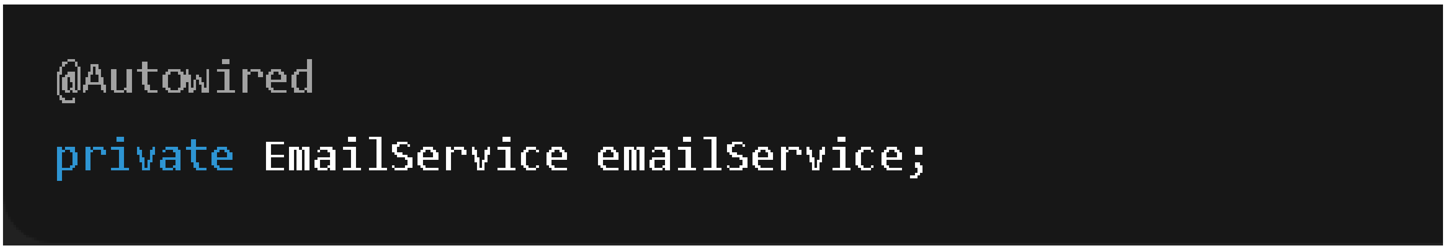
This test class exists to **validate end-to-end email functionality inside the Spring context**, not just method-level logic.

In simple terms:

- SMTP configuration
- Spring bean wiring
- EmailService integration
  are all being verified together.

This is closer to an **integration sanity check** than a pure unit test—and that's perfectly aligned for infrastructure components like email.

Dependency injection validation

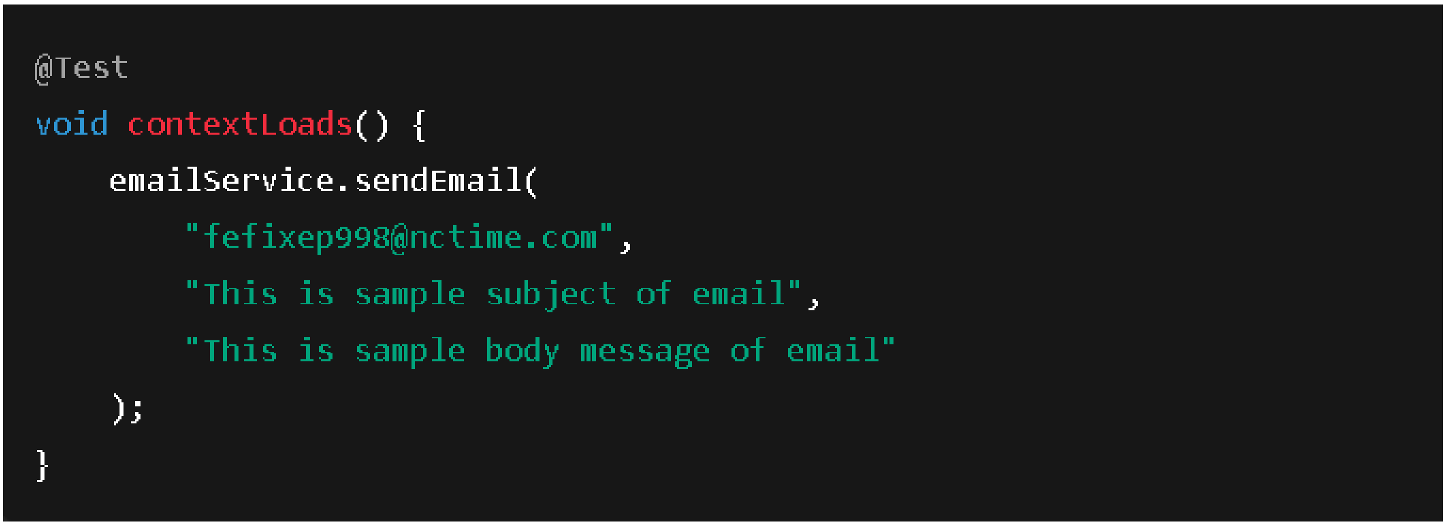


```
@Autowired
private EmailService emailService;
```

- Confirms that EmailService is correctly registered as a Spring bean
- Ensures all its dependencies (like JavaMailSender) are resolved
- If the context fails to load or mail config is broken, tests will fail early

This step alone provides **deployment-time confidence**.

Test 1: Single-recipient email validation



```
@Test
void contextLoads() {
    emailService.sendEmail(
        "fefixep998@nctime.com",
        "This is sample subject of email",
        "This is sample body message of email"
    );
}
```

What this test proves

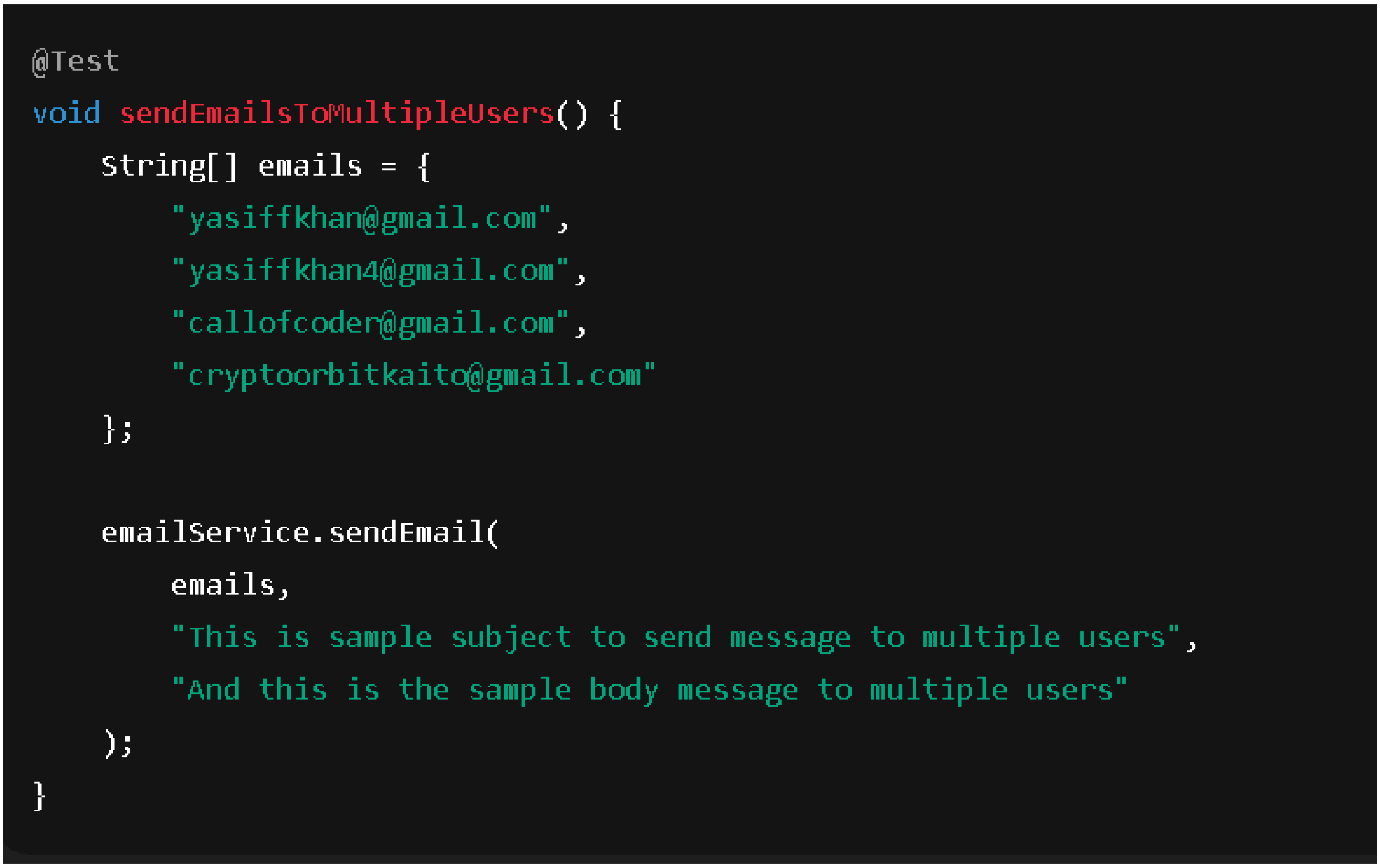
- Spring Boot application context loads successfully
- EmailService is usable immediately after startup
- SMTP credentials and mail server connectivity are working
- The single-recipient email path behaves as expected

This test acts as a **baseline smoke test** for email delivery.

From a delivery standpoint:

- If this test passes, transactional emails in real workflows are unlikely to fail due to misconfiguration.

Test 2: Multiple-recipient (broadcast) email validation



```java
@Test
void sendEmailsToMultipleUsers() {
    String[] emails = {
        "yasiffkhan@gmail.com",
        "yasiffkhan4@gmail.com",
        "callofcoder@gmail.com",
        "cryptoorbitkaito@gmail.com"
    };

    emailService.sendEmail(
        emails,
        "This is sample subject to send message to multiple users",
        "And this is the sample body message to multiple users"
    );
}
```

What this test validates

- Bulk email overload works correctly
- Array-based recipient handling is stable
- BCC logic (admin/system visibility) is exercised
- No runtime issues occur when sending to multiple recipients

This is critical for:

- admin notifications
- system alerts
- broadcast-style communication

Why this approach is enterprise-aligned

These tests intentionally do **not mock** the email sender because:

- Email is an infrastructure concern
- Failures usually come from configuration, not logic
- Catching SMTP or credential issues early saves production incidents

In real-world systems:

- Broken email configs often go unnoticed until users complain
- These tests proactively eliminate that risk

Observability and fault tolerance angle

Since the EmailService:

- logs success
- catches and logs failures

these tests also indirectly verify that:

- failures won’t crash the application
- email issues won’t block business flows

That’s a **resilience-first mindset**.

Big-picture takeaway

This test class demonstrates:

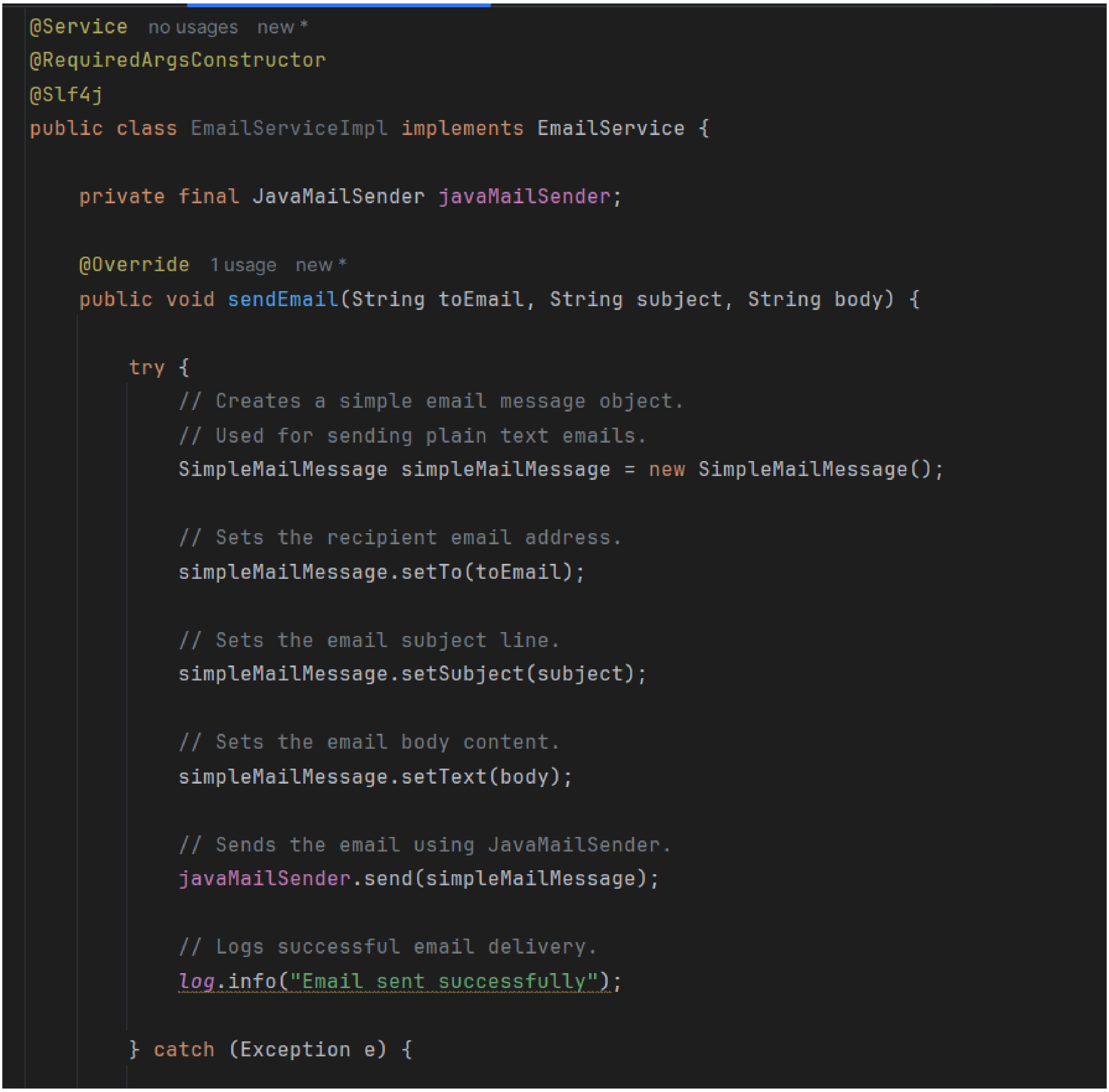
- proactive validation of external integrations
- confidence-driven development
- production-first thinking

It ensures that when features like:

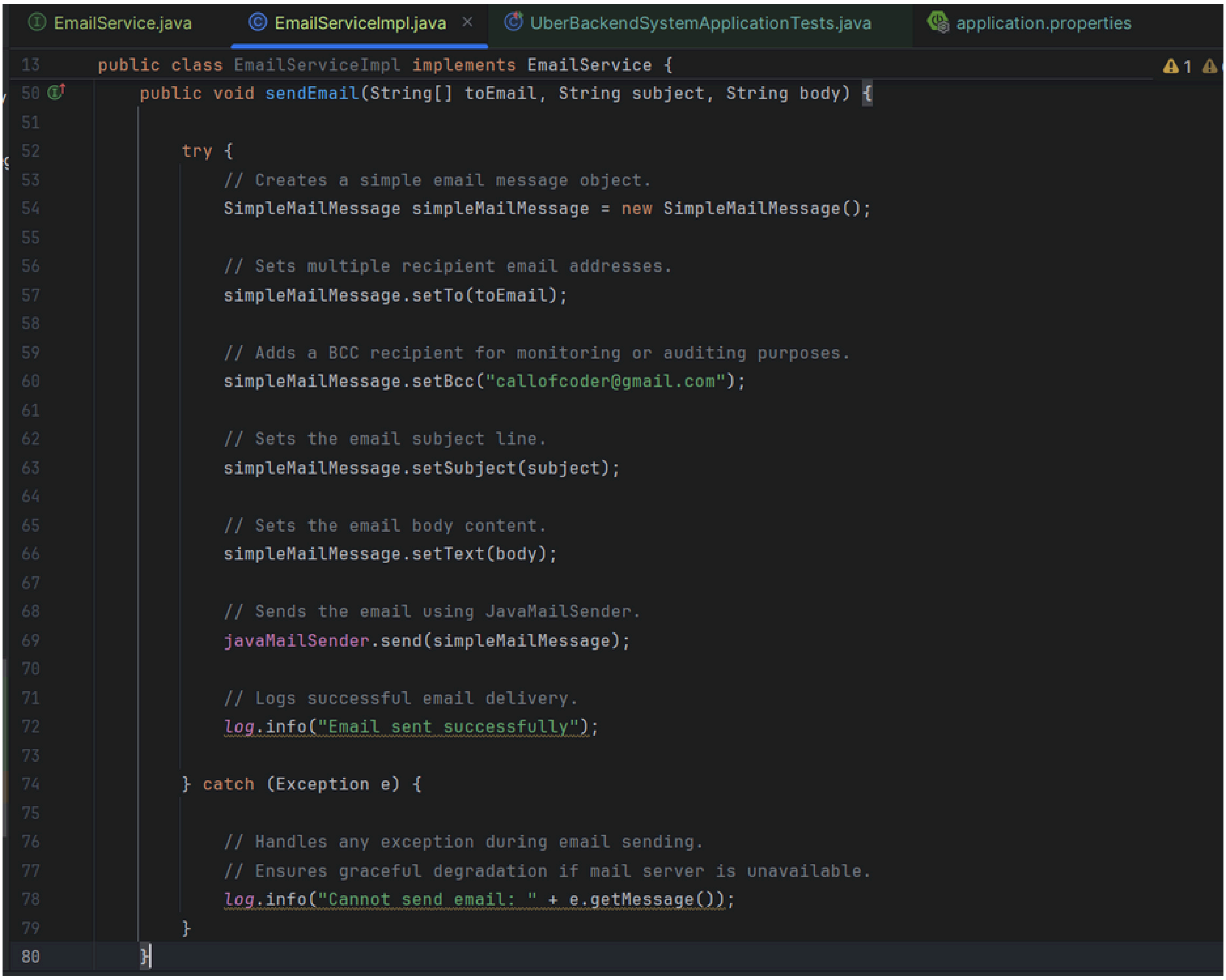
- account lock alerts
- ride notifications
- admin actions

go live, the communication layer is already battle-tested.

In short, this is not “just a test”—
it’s a **risk-mitigation layer baked directly into the development lifecycle**.



```
@Service  no usages  new *
@RequiredArgsConstructor
@Slf4j
public class EmailServiceImpl implements EmailService {

    private final JavaMailSender javaMailSender;

    @Override  1 usage  new *
    public void sendEmail(String toEmail, String subject, String body) {

        try {
            // Creates a simple email message object.
            // Used for sending plain text emails.
            SimpleMailMessage simpleMailMessage = new SimpleMailMessage();

            // Sets the recipient email address.
            simpleMailMessage.setTo(toEmail);

            // Sets the email subject line.
            simpleMailMessage.setSubject(subject);

            // Sets the email body content.
            simpleMailMessage.setText(body);

            // Sends the email using JavaMailSender.
            javaMailSender.send(simpleMailMessage);

            // Logs successful email delivery.
            log.info("Email sent successfully");

        } catch (Exception e) {
```



```java
public class EmailServiceImpl implements EmailService {
    public void sendEmail(String[] toEmail, String subject, String body) {

        try {
            // Creates a simple email message object.
            SimpleMailMessage simpleMailMessage = new SimpleMailMessage();

            // Sets multiple recipient email addresses.
            simpleMailMessage.setTo(toEmail);

            // Adds a BCC recipient for monitoring or auditing purposes.
            simpleMailMessage.setBcc("callofcoder@gmail.com");

            // Sets the email subject line.
            simpleMailMessage.setSubject(subject);

            // Sets the email body content.
            simpleMailMessage.setText(body);

            // Sends the email using JavaMailSender.
            javaMailSender.send(simpleMailMessage);

            // Logs successful email delivery.
            log.info("Email sent successfully");

        } catch (Exception e) {

            // Handles any exception during email sending.
            // Ensures graceful degradation if mail server is unavailable.
            log.info("Cannot send email: " + e.getMessage());
        }
    }
```

Strategic intent of this service

This implementation represents a **centralized Email Notification Service**.
Its core objective is to **abstract email delivery logic** away from business workflows like:

- account creation
- account lock/unlock
- ride status notifications
- security alerts
- admin or system broadcasts

From an architectural perspective, this is exactly where email logic should live:
👉 *outside controllers, outside domain entities, and reusable across modules.*

Method 1: Single-recipient email

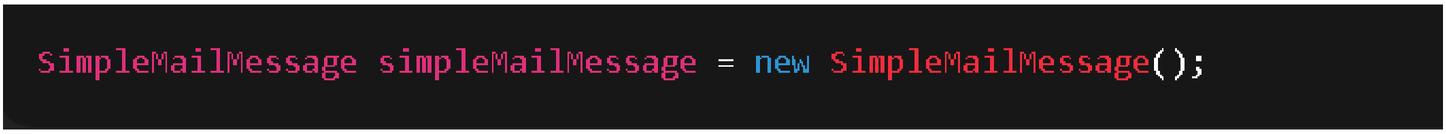
```java
@Override
public void sendEmail(String toEmail, String subject, String body)
```

What this method is optimized for

This variant is designed for **transactional, user-specific communication**, such as:

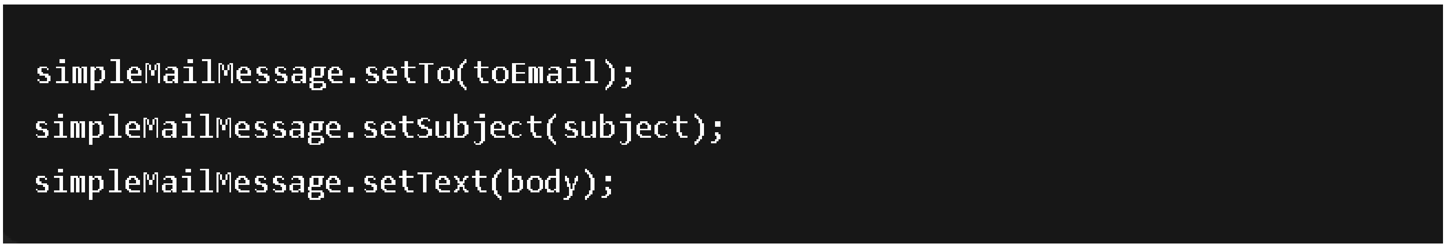
- OTP emails
- account lock notifications
- password reset alerts
- ride confirmation messages

Internal flow explained



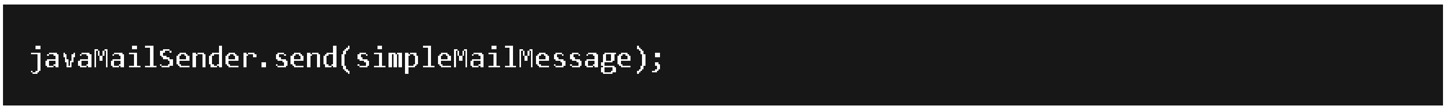
```
SimpleMailMessage simpleMailMessage = new SimpleMailMessage();
```

- Uses Spring’s lightweight mail abstraction
- Keeps implementation simple and synchronous
- Ideal for **plain-text transactional emails**
- Avoids overengineering with MIME unless truly needed



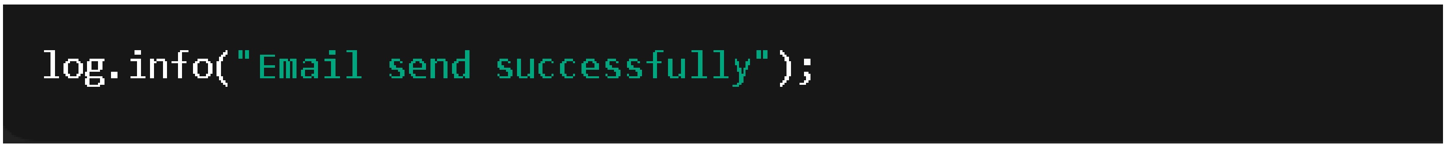
```
simpleMailMessage.setTo(toEmail);
simpleMailMessage.setSubject(subject);
simpleMailMessage.setText(body);
```

- Email metadata is cleanly separated:
  - recipient
  - subject
  - message content
- This promotes **clarity, readability, and maintainability**
- Makes future templating or localization straightforward



```
javaMailSender.send(simpleMailMessage);
```

- Delegates delivery to Spring’s mail infrastructure
- SMTP details remain fully externalized via configuration
- Keeps the service environment-agnostic (local, staging, prod)



```
log.info("Email send successfully");
```

- Adds **operational visibility**
- Helps during production monitoring and debugging
- Aligns with observability best practices

Error handling

```
catch (Exception e) {
    log.info("Cannot send email:"+e.getMessage());
}
```

- Prevents email failures from breaking core business flows
- Ensures **graceful degradation**
- Keeps user-facing APIs responsive even if SMTP fails

This is a deliberate tradeoff:
👉 *email is important, but it should not take the system down.*

Method 2: Multiple-recipient email (broadcast-style)

```
@Override
public void sendEmail(String[] toEmail, String subject, String body)
```

Why this overload exists

This method supports **bulk or system-level communication**, such as:

- admin notifications
- multi-user alerts
- system announcements
- audit or compliance emails

The overload avoids duplication and keeps intent explicit.

Key enhancement: BCC usage

```
simpleMailMessage.setBcc("callofcoder@gmail.com");
```

- Introduces **passive monitoring**
- Ensures a copy is always sent to a system/admin mailbox
- Useful for:
    - auditing
    - compliance
    - debugging delivery issues
- Prevents exposing admin emails to recipients

This is a **very enterprise-aligned practice**.

Everything else remains consistent

- Same message structure
- Same sending mechanism
- Same logging and fault tolerance

Consistency here reduces cognitive load and operational risk.

Design quality assessment

From a backend engineering lens, this implementation demonstrates:

- Clear separation of concerns
- Reusable service abstraction
- Defensive error handling
- Operational observability
- Scalable communication strategy

It also leaves clean extension points for:

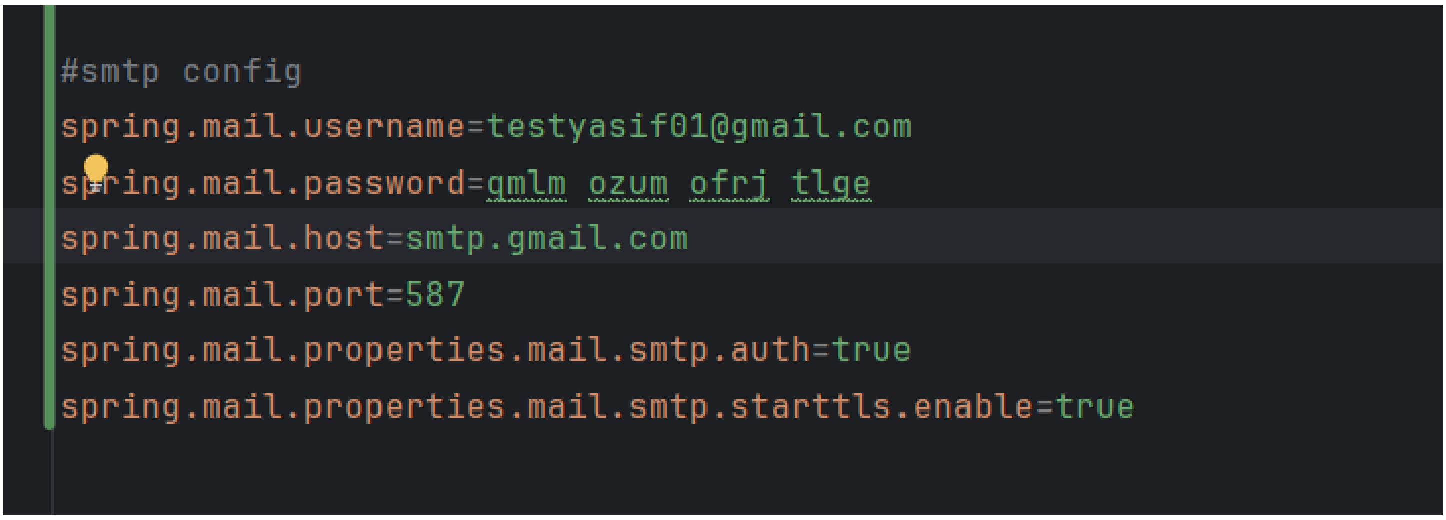
- async email delivery
- HTML/MIME emails
- template engines (Thymeleaf / FreeMarker)
- message queues (Kafka / RabbitMQ)

Big-picture takeaway

This email service is **not just about sending emails**.
It is about:

- protecting core workflows from external failures
- enabling reliable user communication
- supporting admin visibility
- laying groundwork for future scalability

In the context of an Uber-scale backend, this is exactly the level of foresight expected.



```
#smtp config
spring.mail.username=testyasif01@gmail.com
spring.mail.password=qmlm ozum ofrj tlge
spring.mail.host=smtp.gmail.com
spring.mail.port=587
spring.mail.properties.mail.smtp.auth=true
spring.mail.properties.mail.smtp.starttls.enable=true
```

Let’s break this down **cleanly, professionally, and with production context** 🧩✉️
This configuration is the **SMTP email setup** used by Spring Boot to send emails via **Gmail**.

📌 What this configuration does (Big Picture)

This block wires Spring Boot’s **JavaMailSender** with **Gmail SMTP**, enabling the application to send:

- transactional emails
- alerts & notifications
- admin/system communications

In short: **this is the backbone of your email infrastructure** 🚀

🔍 Line-by-line explanation

```
spring.mail.username=testyasif01@gmail.com
```

- This is the **sender email address**
- Gmail account from which all emails will be sent
- Used internally by JavaMailSender as the authenticated user

📌 *Think of this as the “from” identity of the system*

```
spring.mail.password=...
```

- This is **NOT the Gmail login password**
- This is a **Gmail App Password**
- Required because Gmail blocks normal passwords for SMTP access

✅ App passwords:

- Are safer
- Can be revoked anytime
- Are ideal for backend services

📌 *Enterprise-grade security practice*

```
spring.mail.host=smtp.gmail.com
```

- Gmail’s official SMTP server
- Routes outbound emails from your application

📌 *SMTP = Simple Mail Transfer Protocol*

```
spring.mail.port=587
```

- Port **587** is used for **STARTTLS**
- Industry standard for secure email transmission

📌 Why not 465?

- 465 → SSL (legacy)
- 587 → STARTTLS (modern & recommended)

spring.mail.properties.mail.smtp.auth=true

- Enables **SMTP authentication**
- Ensures Gmail verifies username + app password

📌 *Without this, Gmail will reject requests immediately*

spring.mail.properties.mail.smtp.starttls.enable=true

- Activates **TLS encryption**
- Encrypts email content during transmission

📌 This ensures:

- No plaintext emails
- No credential leaks
- Compliance with security standards

🔐 Security & Production Best Practices

You implemented this correctly, but in **real production systems**, improve it further 👇

✅ Move credentials to environment variables

```
spring.mail.username=${MAIL_USERNAME}
spring.mail.password=${MAIL_PASSWORD}
```

✅ Never commit passwords to GitHub

- Even app passwords are sensitive
- Use .env, AWS Secrets Manager, or Vault

🧠 How this connects to your code

This configuration powers:

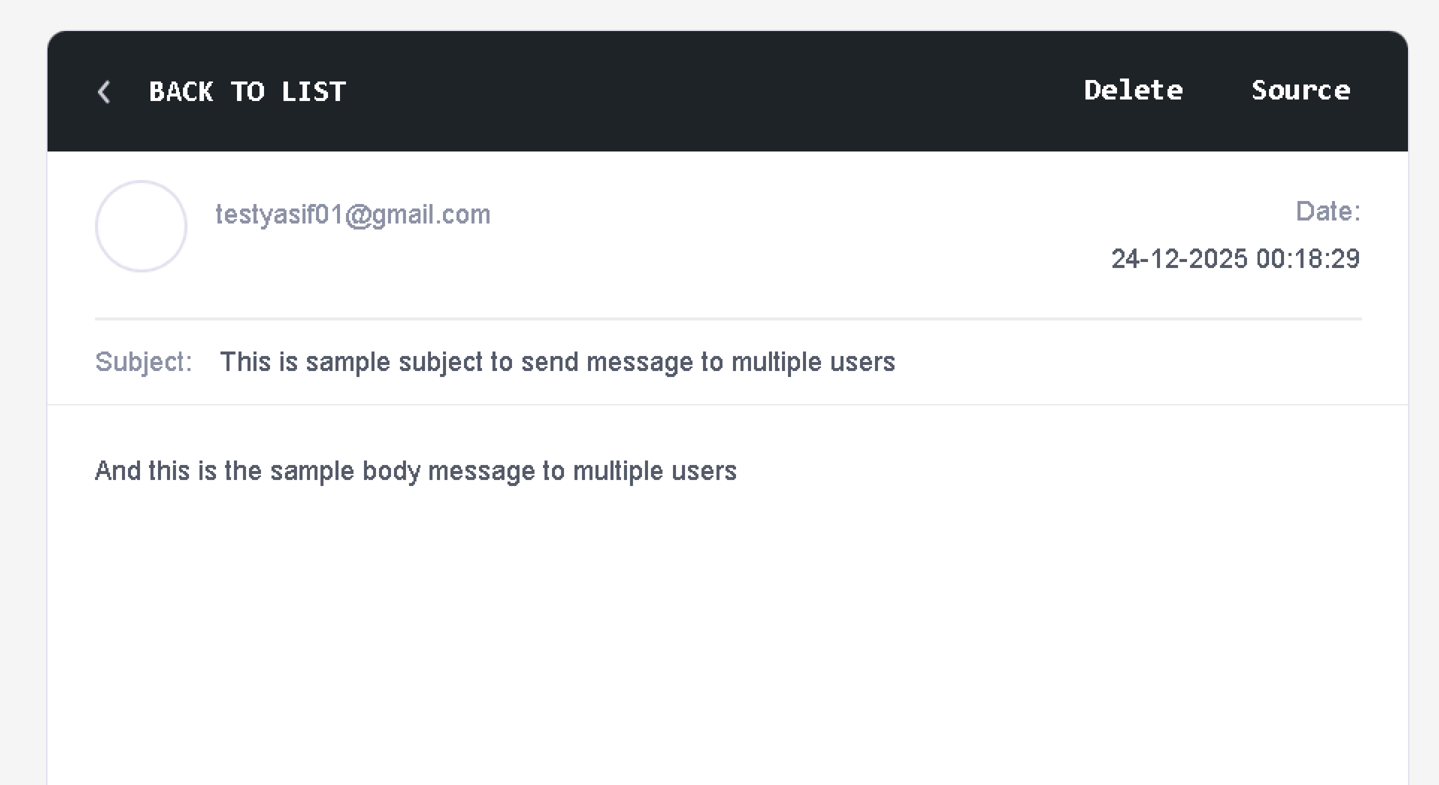
- JavaMailSender
- SimpleMailMessage
- Your EmailService.sendEmail() methods
- Your integration tests (@Test sendEmail())

📌 Without this config:
❌ No emails
❌ Tests fail
❌ Notifications break

💼 Professional takeaway (Interview-ready)

“I configured Spring Boot’s mail infrastructure using Gmail SMTP with STARTTLS, app-password-based authentication, and validated it through integration tests to ensure reliable transactional email delivery.”

That’s a **strong backend engineering statement** 💯

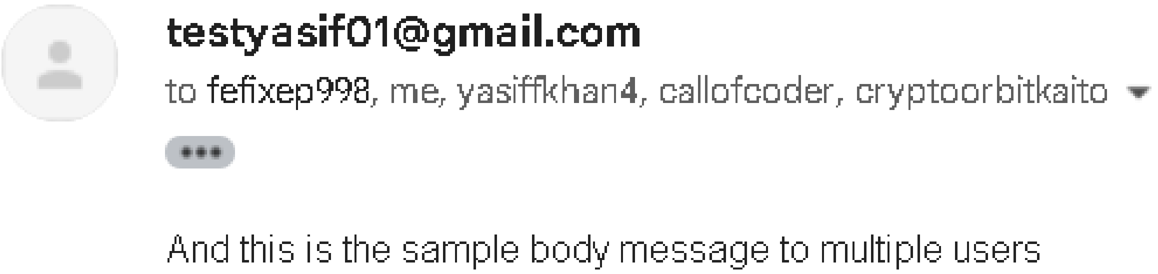


testyasif01@gmail.com

Date:
24-12-2025 00:18:29

Subject: This is sample subject to send message to multiple users

And this is the sample body message to multiple users



testyasif01@gmail.com

to fefixep998, me, yasiffkhan4, callofcoder, cryptoorbitkaito

And this is the sample body message to multiple users