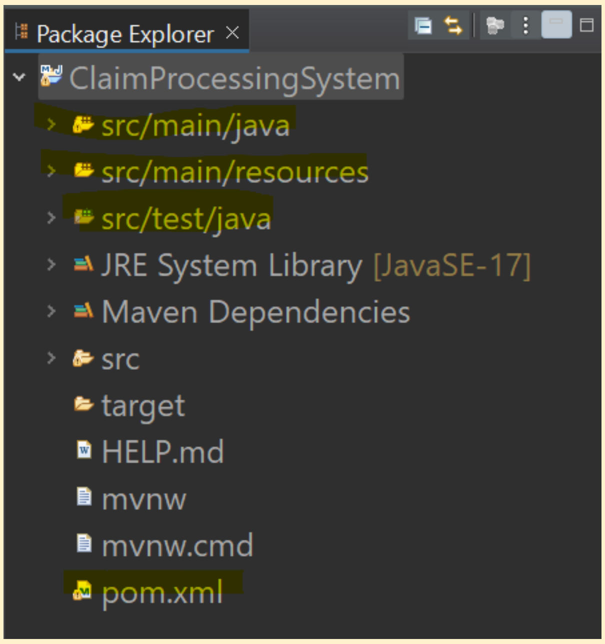
Claim Processing System Protype Code Explanation

Important Note: *This project may or may not work. The idea behind writing and sharing the code with you is to help you understand how different parts of the project connect and work together. It's a small prototype codebase. For example, we have used Log4j2 in one method only to demonstrate its functionality, but in an enterprise project, we would use every service/tool throughout the project. Additionally, real enterprise projects contain hundreds of thousands of lines of code, and it's nearly impossible to create an exact enterprise project.*

1. Project Overview:

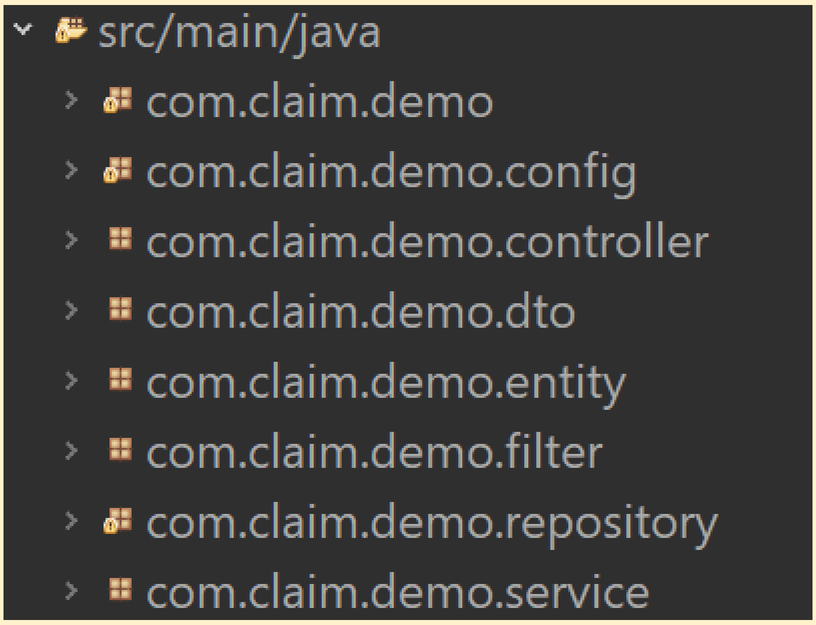


```
Package Explorer
ClaimProcessingSystem
  src/main/java
  src/main/resources
  src/test/java
  JRE System Library [JavaSE-17]
  Maven Dependencies
  src
  target
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
```

"ClaimProcessingSystem" is our main folder of claim processing system monolithic application and this is based on maven project.

- Inside this project we have 'src/main/java' that contain the main code of our application.
- 'src/main/resources' contains the application.properties file for setting the properties of different tool such as kafla redis etc etc
- 'src/test/java' contain the test classes and test cases for sonar coverage as well the testing whole application
- In the end we have 'pom.xml' file that contain dependencies to make it a complete springboot application

Our main focus would be on 'src/main/java',



```
src/main/java
    com.claim.demo
    com.claim.demo.config
    com.claim.demo.controller
    com.claim.demo.dto
    com.claim.demo.entity
    com.claim.demo.filter
    com.claim.demo.repository
    com.claim.demo.service
```

com.claim.demo: This is the main package where the application's entry point, usually the main method, is defined. This method is responsible for booting the application.

com.claim.demo.controller: This package contains the REST API controllers. Controllers handle incoming HTTP requests and respond with the appropriate HTTP responses. They act as the interface between the frontend and the backend services.

com.claim.demo.dto (Data Transfer Objects): DTOs are simple objects that should not contain any business logic. They are used to transfer data between processes, in this case, mainly between your controllers and services.

com.claim.demo.entity: This package includes the entity classes which map to the database tables through ORM (Object Relational Mapping). These classes typically contain Hibernate or JPA annotations that describe the database table structure.

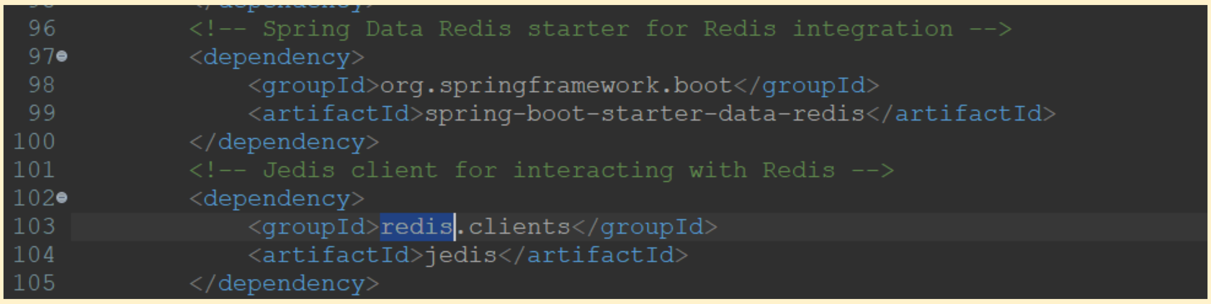
com.claim.demo.filter: This package likely contains filters used in the web layer for things like logging, authentication, or preprocessing requests before they reach the controllers.

com.claim.demo.repository: This package is expected to contain interfaces that extend JpaRepository or similar Spring Data interfaces. Repositories abstract the data layer, providing an elegant way to perform CRUD operations on the database.

com.claim.demo.service: Services contain the business logic of the application. They interact with repositories to fetch and store data, perform calculations, and handle business rules.

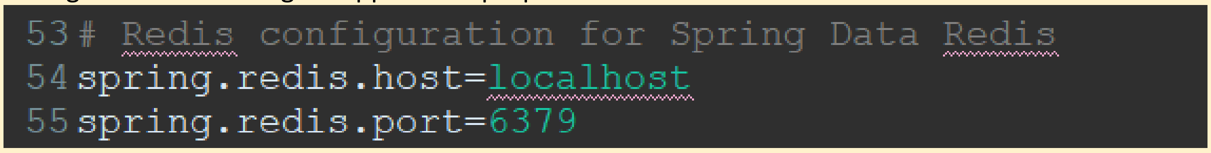
2. How Redis Cache is integrated:

Added Redis dependencies in the pom.xml.



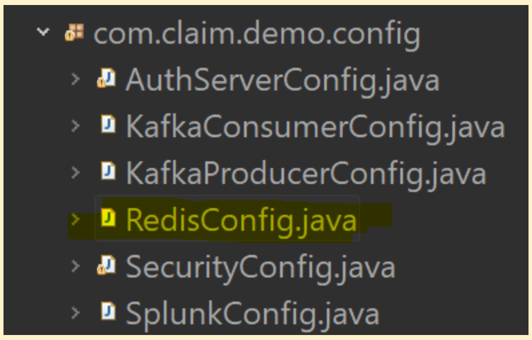
```xml
        <!-- Spring Data Redis starter for Redis integration -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-redis</artifactId>
        </dependency>
        <!-- Jedis client for interacting with Redis -->
        <dependency>
            <groupId>redis.clients</groupId>
            <artifactId>jedis</artifactId>
        </dependency>
```

Configured Redis settings in application.properties.



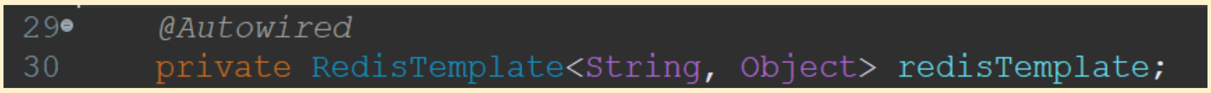
```properties
# Redis configuration for Spring Data Redis
spring.redis.host=localhost
spring.redis.port=6379
```

Implemented Redis configuration in a dedicated config file.



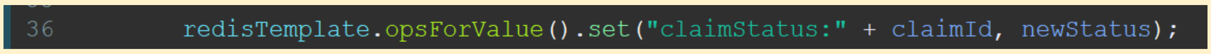
```
com.claim.demo.config
    AuthServerConfig.java
    KafkaConsumerConfig.java
    KafkaProducerConfig.java
    RedisConfig.java
    SecurityConfig.java
    SplunkConfig.java
```

Autowired RedisTemplate<String, Object> in com.claim.demo.service.ClaimService.



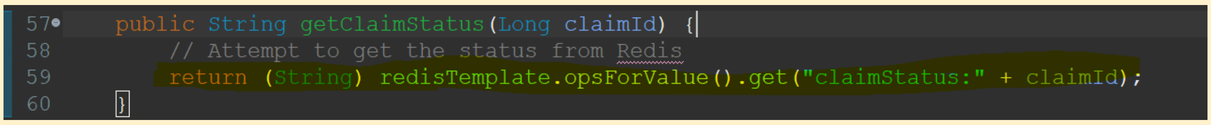
```java
    @Autowired
    private RedisTemplate<String, Object> redisTemplate;
```

Set data storage in Redis within com.claim.demo.service.ClaimService.updateClaimStatus and getClaimStatus methods.



```java
        redisTemplate.opsForValue().set("claimStatus:" + claimId, newStatus);
```

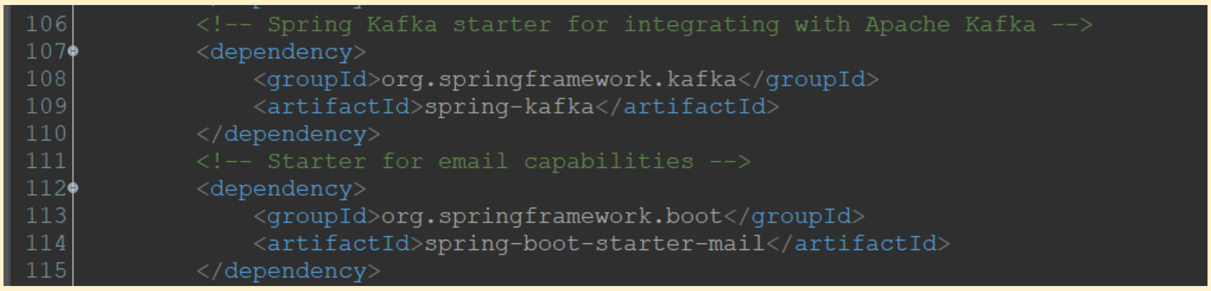
Fetching the data from redis cache in com.claim.demo.service.ClaimService.getClaimStatus(Long)



```java
    public String getClaimStatus(Long claimId) {
        // Attempt to get the status from Redis
        return (String) redisTemplate.opsForValue().get("claimStatus:" + claimId);
    }
```

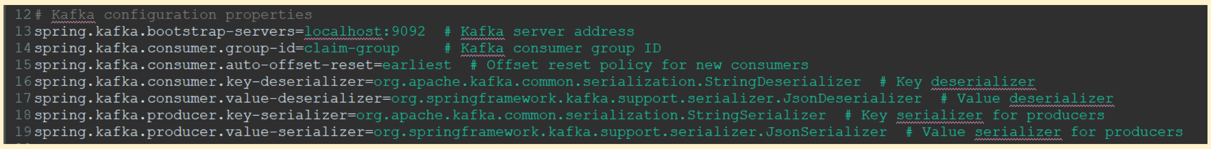
3. How Kafka is integrated:

Added Kafka and email-related dependencies to send notifications via email.

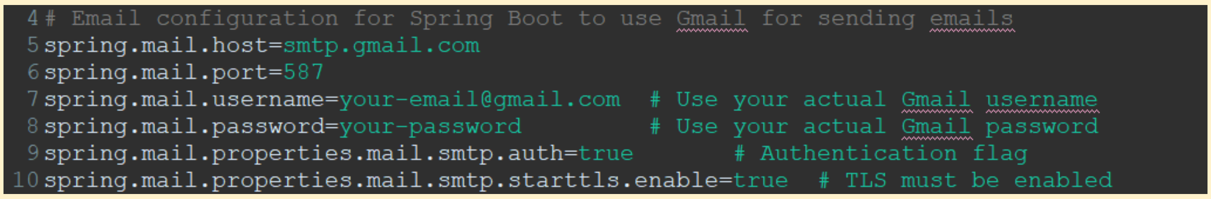


```xml
        <!-- Spring Kafka starter for integrating with Apache Kafka -->
        <dependency>
            <groupId>org.springframework.kafka</groupId>
            <artifactId>spring-kafka</artifactId>
        </dependency>
        <!-- Starter for email capabilities -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-mail</artifactId>
        </dependency>
```

Configured Kafka and email settings in application.properties.

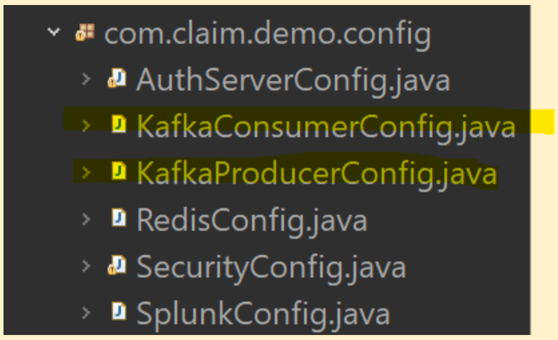


```properties
# Kafka configuration properties
spring.kafka.bootstrap-servers=localhost:9092  # Kafka server address
spring.kafka.consumer.group-id=claim-group     # Kafka consumer group ID
spring.kafka.consumer.auto-offset-reset=earliest  # Offset reset policy for new consumers
spring.kafka.consumer.key-deserializer=org.apache.kafka.common.serialization.StringDeserializer  # Key deserializer
spring.kafka.consumer.value-deserializer=org.springframework.kafka.support.serializer.JsonDeserializer  # Value deserializer
spring.kafka.producer.key-serializer=org.apache.kafka.common.serialization.StringSerializer  # Key serializer for producers
spring.kafka.producer.value-serializer=org.springframework.kafka.support.serializer.JsonSerializer  # Value serializer for producers
```

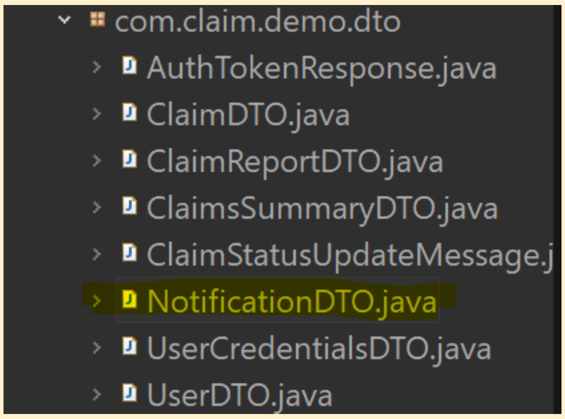


```properties
# Email configuration for Spring Boot to use Gmail for sending emails
spring.mail.host=smtp.gmail.com
spring.mail.port=587
spring.mail.username=your-email@gmail.com  # Use your actual Gmail username
spring.mail.password=your-password         # Use your actual Gmail password
spring.mail.properties.mail.smtp.auth=true        # Authentication flag
spring.mail.properties.mail.smtp.starttls.enable=true  # TLS must be enabled
```

Added KafkaProducerConfig and KafkaConsumerConfig in the config package, and NotificationDTO in the DTO package to manage notifications.



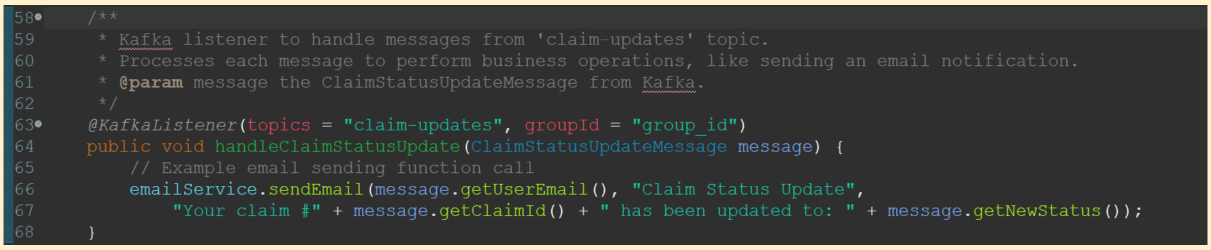
```
com.claim.demo.config
    AuthServerConfig.java
    KafkaConsumerConfig.java
    KafkaProducerConfig.java
    RedisConfig.java
    SecurityConfig.java
    SplunkConfig.java
```



```
com.claim.demo.dto
  AuthTokenResponse.java
  ClaimDTO.java
  ClaimReportDTO.java
  ClaimsSummaryDTO.java
  ClaimStatusUpdateMessage.j
  NotificationDTO.java
  UserCredentialsDTO.java
  UserDTO.java
```

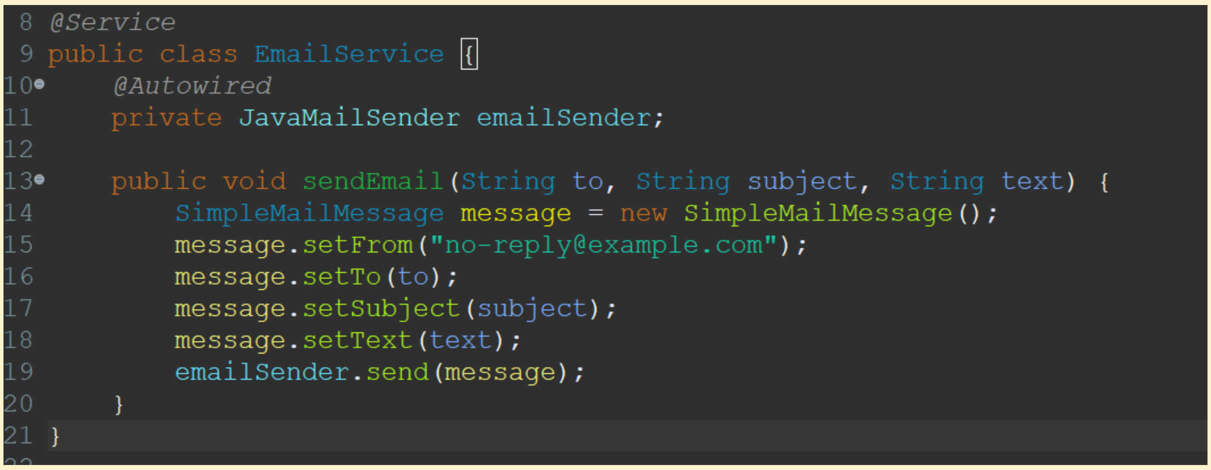
KafkaProducerConfig sends messages to a Kafka topic.

KafkaConsumerConfig consumes notifications and triggers an email send via sendEmail in com.claim.demo.service.EmailService when updates are published to the "claim-updates" Kafka topic.



```java
    /**
     * Kafka listener to handle messages from 'claim-updates' topic.
     * Processes each message to perform business operations, like sending an email notification.
     * @param message the ClaimStatusUpdateMessage from Kafka.
     */
    @KafkaListener(topics = "claim-updates", groupId = "group_id")
    public void handleClaimStatusUpdate(ClaimStatusUpdateMessage message) {
        // Example email sending function call
        emailService.sendEmail(message.getUserEmail(), "Claim Status Update",
            "Your claim #" + message.getClaimId() + " has been updated to: " + message.getNewStatus());
    }
```

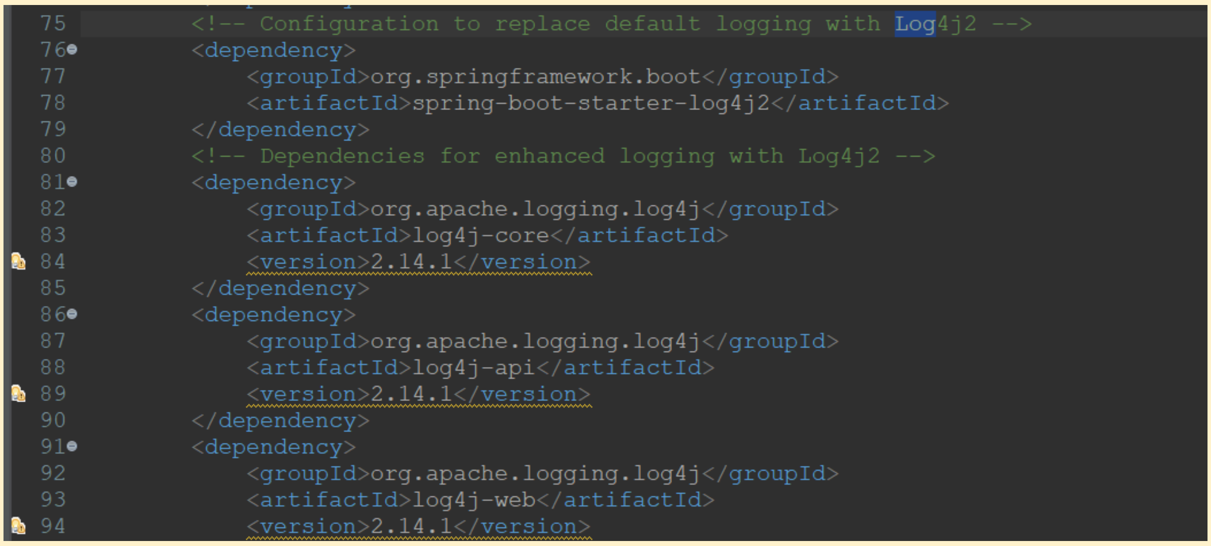
In ‘sendEmail’ inside com.claim.demo.service.EmailService, we are actually sending updates to the user



```java
@Service
public class EmailService {
    @Autowired
    private JavaMailSender emailSender;

    public void sendEmail(String to, String subject, String text) {
        SimpleMailMessage message = new SimpleMailMessage();
        message.setFrom("no-reply@example.com");
        message.setTo(to);
        message.setSubject(subject);
        message.setText(text);
        emailSender.send(message);
    }
}
```

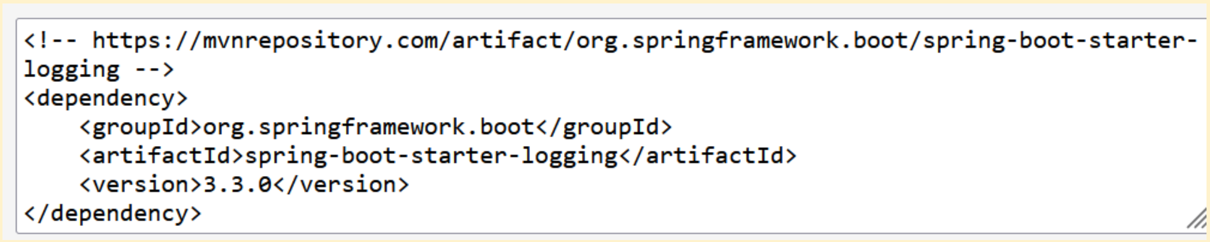
4. How log4j2 is integrated:

Replaced the default logging dependency with Log4j2 in the pom.xml.



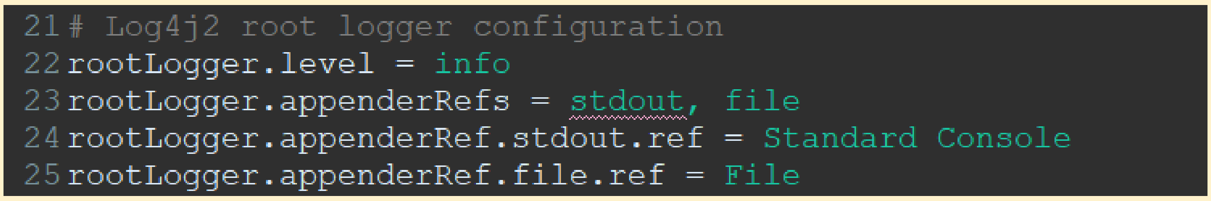
```
        <!-- Configuration to replace default logging with Log4j2 -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-log4j2</artifactId>
        </dependency>
        <!-- Dependencies for enhanced logging with Log4j2 -->
        <dependency>
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-core</artifactId>
            <version>2.14.1</version>
        </dependency>
        <dependency>
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-api</artifactId>
            <version>2.14.1</version>
        </dependency>
        <dependency>
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-web</artifactId>
            <version>2.14.1</version>
```

Removed dependency form pom.xml:



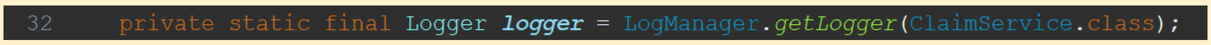
```
<!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-
logging -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-logging</artifactId>
    <version>3.3.0</version>
</dependency>
```

Configured Log4j2 settings in application.properties.

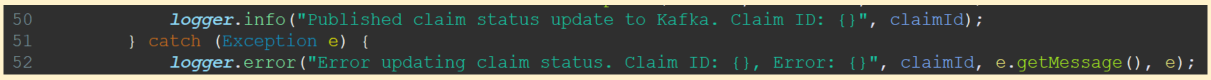


```
# Log4j2 root logger configuration
rootLogger.level = info
rootLogger.appenderRefs = stdout, file
rootLogger.appenderRef.stdout.ref = Standard Console
rootLogger.appenderRef.file.ref = File
```

We have added log4j2 in one place only just to show you how we can add log4j2 in the com.claim.demo.service.ClaimService class and updateClaimStatus method.



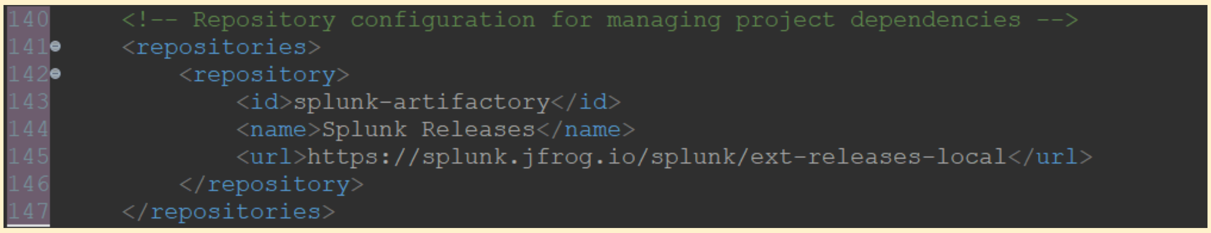
```
    private static final Logger logger = LogManager.getLogger(ClaimService.class);
```



```
            logger.info("Published claim status update to Kafka. Claim ID: {}", claimId);
        } catch (Exception e) {
            logger.error("Error updating claim status. Claim ID: {}, Error: {}", claimId, e.getMessage(), e);
```

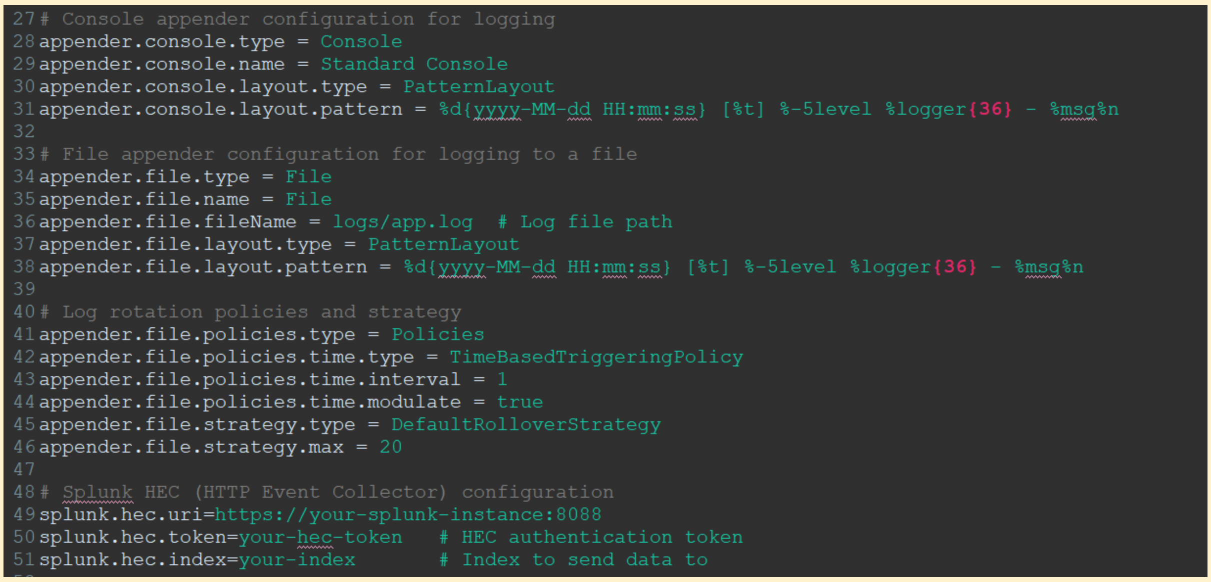
5. How Splunk is integrated:

Added the Splunk repository in the pom.xml.



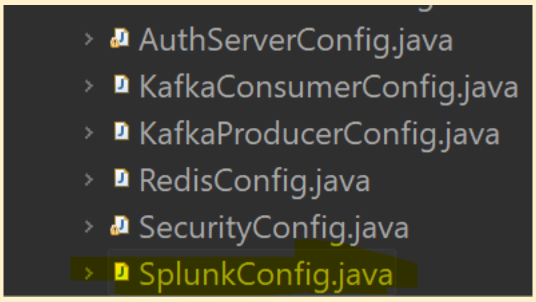
```
    <!-- Repository configuration for managing project dependencies -->
    <repositories>
        <repository>
            <id>splunk-artifactory</id>
            <name>Splunk Releases</name>
            <url>https://splunk.jfrog.io/splunk/ext-releases-local</url>
        </repository>
    </repositories>
```

Set up Splunk appender in com.claim.demo.config.SplunkConfig.



```
# Console appender configuration for logging
appender.console.type = Console
appender.console.name = Standard Console
appender.console.layout.type = PatternLayout
appender.console.layout.pattern = %d{yyyy-MM-dd HH:mm:ss} [%t] %-5level %logger{36} - %msg%n

# File appender configuration for logging to a file
appender.file.type = File
appender.file.name = File
appender.file.fileName = logs/app.log  # Log file path
appender.file.layout.type = PatternLayout
appender.file.layout.pattern = %d{yyyy-MM-dd HH:mm:ss} [%t] %-5level %logger{36} - %msg%n

# Log rotation policies and strategy
appender.file.policies.type = Policies
appender.file.policies.time.type = TimeBasedTriggeringPolicy
appender.file.policies.time.interval = 1
appender.file.policies.time.modulate = true
appender.file.strategy.type = DefaultRolloverStrategy
appender.file.strategy.max = 20

# Splunk HEC (HTTP Event Collector) configuration
splunk.hec.uri=https://your-splunk-instance:8088
splunk.hec.token=your-hec-token    # HEC authentication token
splunk.hec.index=your-index        # Index to send data to
```

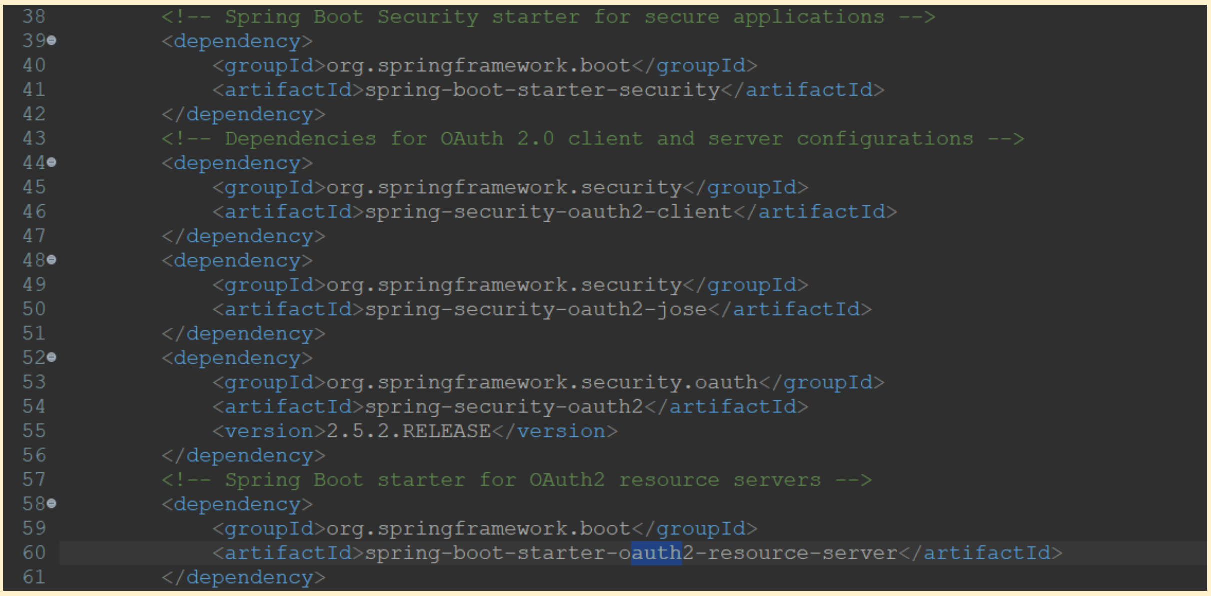
Then we have configured Splunk appender in com.claim.demo.config.SplunkConfig class



```
> AuthServerConfig.java
> KafkaConsumerConfig.java
> KafkaProducerConfig.java
> RedisConfig.java
> SecurityConfig.java
> SplunkConfig.java
```

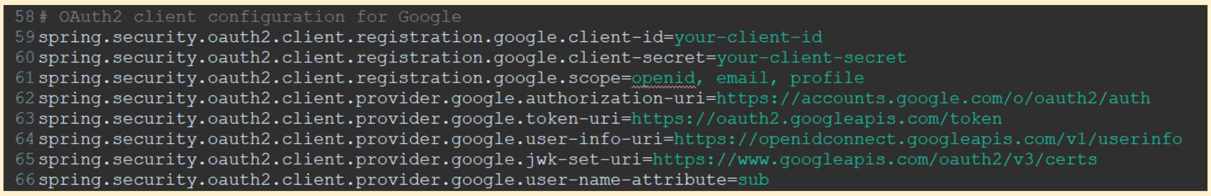
6. How Oauth2 is integrated:

Added OAuth2 and Spring Security dependencies.



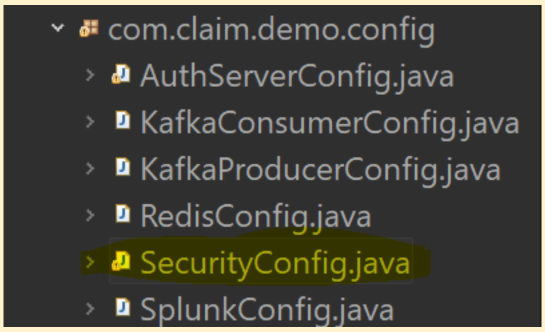
```
        <!-- Spring Boot Security starter for secure applications -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-security</artifactId>
        </dependency>
        <!-- Dependencies for OAuth 2.0 client and server configurations -->
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-oauth2-client</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-oauth2-jose</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.security.oauth</groupId>
            <artifactId>spring-security-oauth2</artifactId>
            <version>2.5.2.RELEASE</version>
        </dependency>
        <!-- Spring Boot starter for OAuth2 resource servers -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
        </dependency>
```

Configured OAuth-related properties in application.properties.



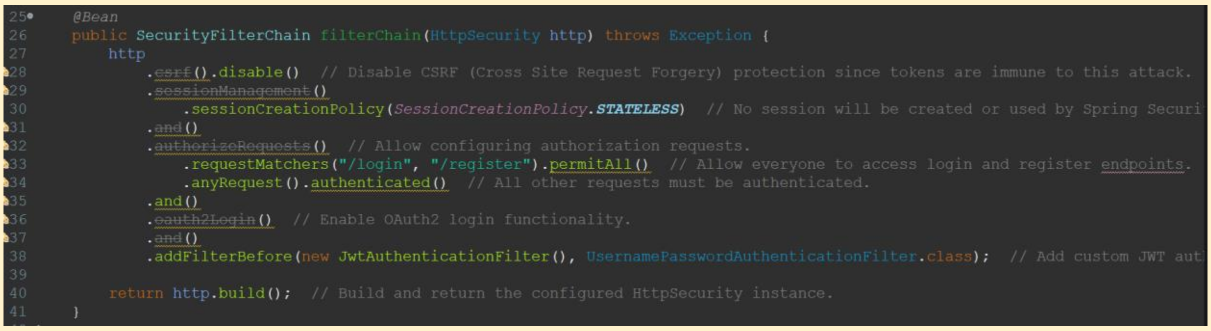
```
# OAuth2 client configuration for Google
spring.security.oauth2.client.registration.google.client-id=your-client-id
spring.security.oauth2.client.registration.google.client-secret=your-client-secret
spring.security.oauth2.client.registration.google.scope=openid, email, profile
spring.security.oauth2.client.provider.google.authorization-uri=https://accounts.google.com/o/oauth2/auth
spring.security.oauth2.client.provider.google.token-uri=https://oauth2.googleapis.com/token
spring.security.oauth2.client.provider.google.user-info-uri=https://openidconnect.googleapis.com/v1/userinfo
spring.security.oauth2.client.provider.google.jwk-set-uri=https://www.googleapis.com/oauth2/v3/certs
spring.security.oauth2.client.provider.google.user-name-attribute=sub
```

Implemented security configuration and token validation excluding /login and /register APIs in the security config file.



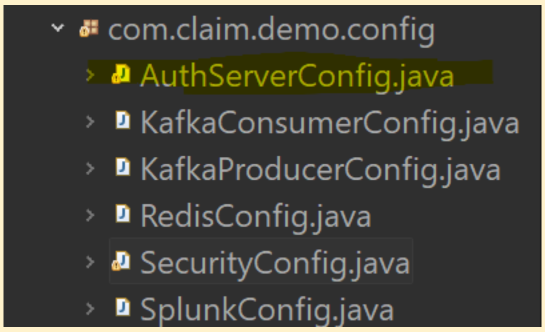
```
com.claim.demo.config
    AuthServerConfig.java
    KafkaConsumerConfig.java
    KafkaProducerConfig.java
    RedisConfig.java
    SecurityConfig.java
    SplunkConfig.java
```

Inside security config file, we are validating the tokens except /login and /register API



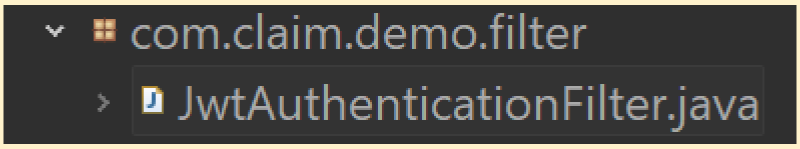
```
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .csrf().disable()  // Disable CSRF (Cross Site Request Forgery) protection since tokens are immune to this attack.
            .sessionManagement()
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS)  // No session will be created or used by Spring Securi
            .and()
            .authorizeRequests()  // Allow configuring authorization requests.
                .requestMatchers("/login", "/register").permitAll()  // Allow everyone to access login and register endpoints.
                .anyRequest().authenticated()  // All other requests must be authenticated.
            .and()
            .oauth2Login()  // Enable OAuth2 login functionality.
            .and()
            .addFilterBefore(new JwtAuthenticationFilter(), UsernamePasswordAuthenticationFilter.class);  // Add custom JWT aut

        return http.build();  // Build and return the configured HttpSecurity instance.
    }
```

We also created AuthServerConfig file to add Oaut2 and token related logic



```
com.claim.demo.config
    AuthServerConfig.java
    KafkaConsumerConfig.java
    KafkaProducerConfig.java
    RedisConfig.java
    SecurityConfig.java
    SplunkConfig.java
```

We created JwtAuthenticationFilter file to validate the tokens before passing Apis to the service methods



```
com.claim.demo.filter
    JwtAuthenticationFilter.java
```