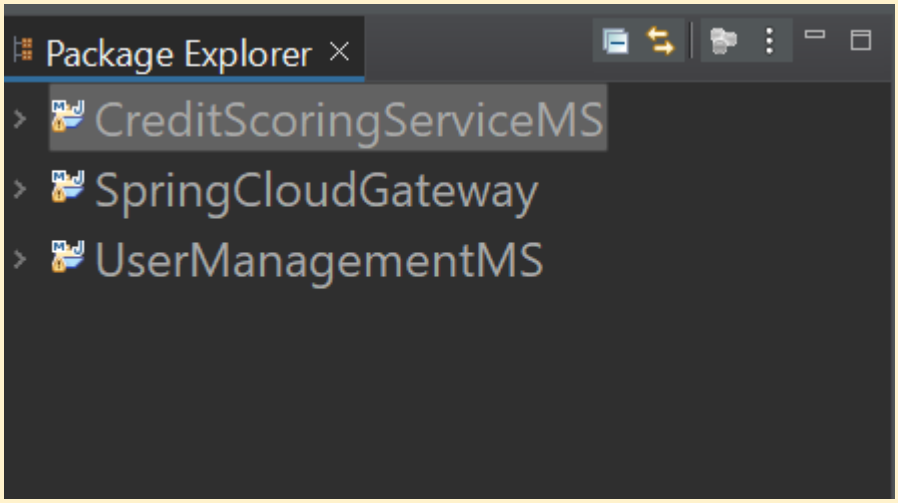
Credit Score Analysis Tool Protype Code Explanation

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:

We have created three microservices here, CreditScoringService, UserManagement and SpringCloudGateway(UI is directly connecting to this service and further this service is rerouting th requests to the different microservices).

NOTE: In Enterprise projects, there may have a lot of microservices since this is prototype based we just created three to understand you how things work.



The details for each microservice:

- Inside our projects 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' for each service,

Main package: 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.

config package: In this package, all the configuration and setup is done for different tools and services.

controller package: 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.

dto (Data Transfer Objects) package: 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.

entity package: 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.

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

repository Package: 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.

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

Note: We haven't implemented everything in every microservice. In CreditScoringServiceMS, we have implemented Redis Cache, Kafka, Email, Log4j2, and Splunk. In UserManagementMS, we have implemented login and registration with Oauth. In one scenario, we are consuming one UserManagementMS API by CreditScoringServiceMS. SpringCouldGateway MS is just for routing the APIs from UI to different microservices.

2. How Redis Cache is integrated in CreditScoringServiceMS:

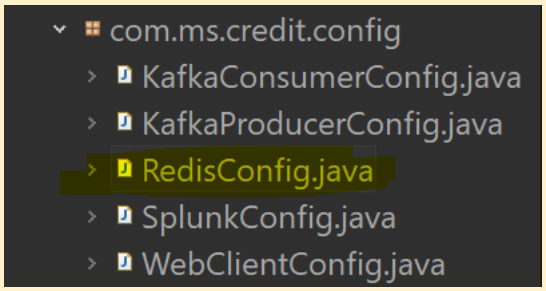
Added Redis dependencies in the pom.xml.

```
        <!-- 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.

```
# Redis configuration for Spring Data Redis
spring.redis.host=localhost
spring.redis.port=6379
# spring.redis.password=yourpassword  # Uncomment and set if Redis requires authentication
```

Implemented Redis configuration in a dedicated config file.



Autowired RedisTemplate<String, Object> in com.ms.credit.service.CreditScoreService.

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

Set data storage in Redis within
com.ms.credit.service.CreditScoreService.calculateCreditScore(CreditScoreDTO)

```
        redisTemplate.opsForValue().set(emailId, convertToEntity(creditScoreDTO));
```

Fetching the data from redis cache in
com.ms.credit.service.CreditScoreService.getCreditScoreByEmailId(int)

```
        CreditScore creditScore = (CreditScore) redisTemplate.opsForValue().get(emailId);
```

3. How Kafka is integrated in CreditScoringServiceMS:

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

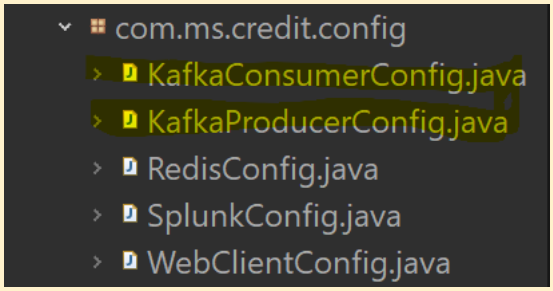
```
        <!-- Spring Kafka starter for integrating with Apache Kafka -->
        <dependency>
            <groupId>org.springframework.kafka</groupId>
            <artifactId>spring-kafka</artifactId>
        </dependency>
        <!-- Starter for email capabilities -->
        <dependency>
     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

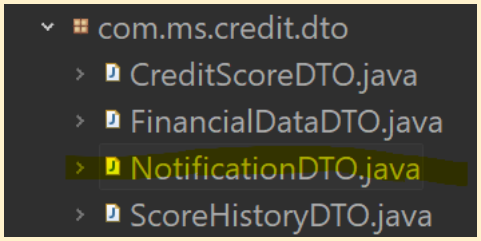
Configured Kafka and email settings in application.properties.

```
# 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
```

```
# 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.





KafkaProducerConfig sends messages to a Kafka topic.

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

```java
    /**
     * KafkaListener method configured to listen to 'credit-score-updates' topic.
     * Processes incoming messages by triggering email notifications using the EmailService.
     * @param message FinancialDataDTO object containing data from the consumed Kafka message.
     */
    @KafkaListener(topics = "credit-score-updates", groupId = "group_id")
    public void handleClaimStatusUpdate(FinancialDataDTO message) {
        // Sending an email about the credit score update.
        emailService.sendEmail(message.getEmailId(), "Credit Score Update",
            "Your Credit Score has been updated to: " + message.getCreditScore());
    }
```

In 'sendEmail' inside com.ms.credit.service.EmailService, we are actually sending updates to the user

```java
@Service  // Marks this class as a Spring managed service.
public class EmailService {
    // Autowires the JavaMailSender interface to use the configured mail sending infrastructure.
    @Autowired
    private JavaMailSender emailSender;

    /**
     * Sends an email to the specified recipient.
     * @param to The recipient's email address.
     * @param subject The subject line of the email.
     * @param text The body of the email.
     */
    public void sendEmail(String to, String subject, String text) {
        SimpleMailMessage message = new SimpleMailMessage();  // Creates a new email message obj
        message.setFrom("no-reply@example.com");  // Sets the sender's email address.
        message.setTo(to);  // Sets the recipient's email address.
        message.setSubject(subject);  // Sets the subject line of the email.
        message.setText(text);  // Sets the main text content of the email.
        emailSender.send(message);  // Sends the email through the JavaMailSender infrastructure
    }
}
```

4. How log4j2 is integrated in CreditScoringServiceMS:

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

```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>
        </dependency>
```

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.ms.credit.service.CreditScoreService class and getCreditScoreByEmailId method.

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

```
        if (creditScore == null) {
            logger.info("Fetching from DB");
            creditScore = creditScoreRepository.findTopByEmailIdOrderByDateDesc(emailId);
        } else {
            logger.info("Fetched from Redis");
```

5. How Splunk is integrated in CreditScoringServiceMS:

Added the Splunk repository in the pom.xml.

```
    <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.ms.credit.config.SplunkConfig.

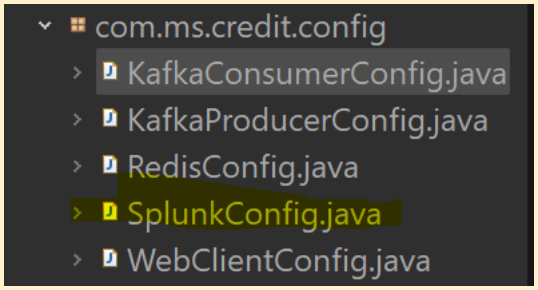
```
# Console appender configuration for logging
appender.console.type = Console
appender.console.name = Standard Console
appender.console.layout.type = PatternLayout
appender.console.layout.pattern = %d{yyyy-MM-dd HH:mm:ss} [%t] %-5level %logger{36} - %msg%n

# 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.ms.credit.config.SplunkConfig class



6. How Oauth2 is integrated in UserManagementMS:

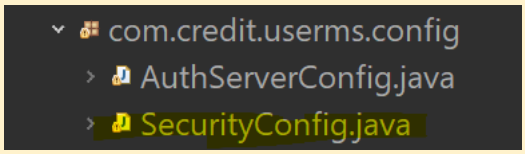
Added OAuth2 and Spring Security dependencies.

```
        <!-- 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>
        <!-- Auth0 Java JWT for working with JSON Web Tokens -->
        <dependency>
            <groupId>com.auth0</groupId>
            <artifactId>java-jwt</artifactId>
            <version>3.18.2</version>
        </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.

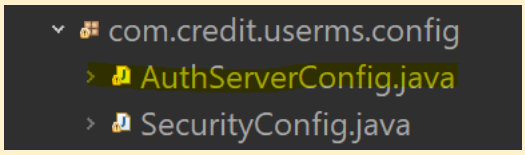


Inside security config file, we are validating the tokens except /login and /register API

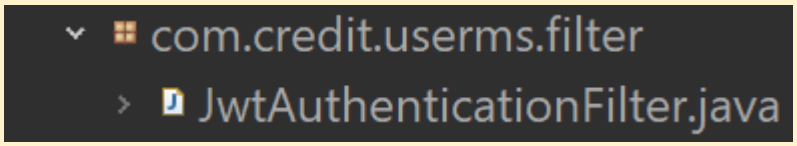
```
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .csrf().disable()  // Disable CSRF (Cross Site Request Forgery) protection since tokens are immune to this attac
            .sessionManagement()
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS)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

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

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



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



7. How Microservices communicating with each other:

In this prototype project, microservices communicate by consuming REST APIs. We have added a scenario where CreditScoringServiceMS consumes an API from UserManagementMS.

First, we created the /{userId} API in the com.credit.userms.controller.UserController of UserManagementMS. This API will be consumed by CreditScoringServiceMS.

```java
    @GetMapping("/{userId}")
    public ResponseEntity<String> getUserById(@PathVariable Long userId) {
        UserDTO userDTO = userService.getUserDetails(userId);
        if (userDTO != null) {
            return ResponseEntity.ok(userDTO.getEmail()); // Credit Scoring Service is getting this emailId
        }
        return ResponseEntity.notFound().build();
    }
```

Also add the service method for this API in com.credit.userms.service.UserService file.

```java
    public UserDTO getUserDetails(Long userId) {
        User user = [illegible]Repository.findById(userId).orElse(null);
        if (user != [illegible]) {
            return new UserDTO(user.getUsername(), user.getEmail(), user.getRole());
        }
        return null;
    }
```

Now in order to consume this '/{userId} by CreditScoringServiceMS,

Add webflux dependency in the pom.xml in CreditScoringServiceMS

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

Inside com.ms.credit.client.UserManagementClient [In CreditScoringServiceMS], we are calling UserManagementMS API as below

```java
    // URL endpoint for the user management service.
    private final String USER_SERVICE_URL = "http://user-management-service/users";

    /**
     * Retrieves user details from the user management service using a reactive WebClient.
     * @param userId The ID of the user whose details are to be fetched.
     * @return a Mono that emits the user details as a string or an error signal if an error occurs.
     */
    public Mono<String> getUserDetails(int userId) {
        return webClient.get()  // Create an HTTP GET request.
                        .uri(USER_SERVICE_URL + "/{userId}", userId)  // Append the user ID to the URL and set the variable in
                        .retrieve()  // Extract the response body automatically.
                        .bodyToMono(String.class);  // Convert the response body to a Mono that emits strings.
    }
```

This above method is called by the service methods and controllers so this way our microservices communicate with each other.

8. How are we using spring cloud gateway in SpringCloudGateway microservice:

Added spring cloud gateway and other security related dependecies in pom.xml

```xml
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-gateway</artifactId>
            <version>4.1.2</version>
        </dependency>
```

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

Added security-related configurations for validation before directing requests to the appropriate microservices. In the springSecurityFilterChain method, we authenticate requests for the userManagementMS and creditScoringServiceMS microservices. We can include additional microservices as required.

```
com.spring.cloudgateway.config
    CorsConfig.java
    GatewaySecurityConfig.java
```

```
@EnableWebFluxSecurity
public class GatewaySecurityConfig {

    @Bean
    public SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
        http
            .authorizeExchange()
            .pathMatchers("/users/**").authenticated()
            .pathMatchers("/credit/**").authenticated()
            .anyExchange().permitAll()
            .and().csrf().disable()
            .oauth2Login(); // Example for OAuth2
        return http.build();
    }
```

Now added CorsConfig file for rerouting the requests.

```
com.spring.cloudgateway.config
    CorsConfig.java
    GatewaySecurityConfig.java
```

```
    @Bean
    public CorsWebFilter corsWebFilter() {
        CorsConfiguration config = new CorsConfiguration();
        config.setAllowCredentials(true);
        config.addAllowedOrigin("http://allowed-origin.com");
        config.addAllowedHeader("*");
        config.addAllowedMethod("*");

        UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
        source.registerCorsConfiguration("/**", config);

        return new CorsWebFilter(source);
    }
```