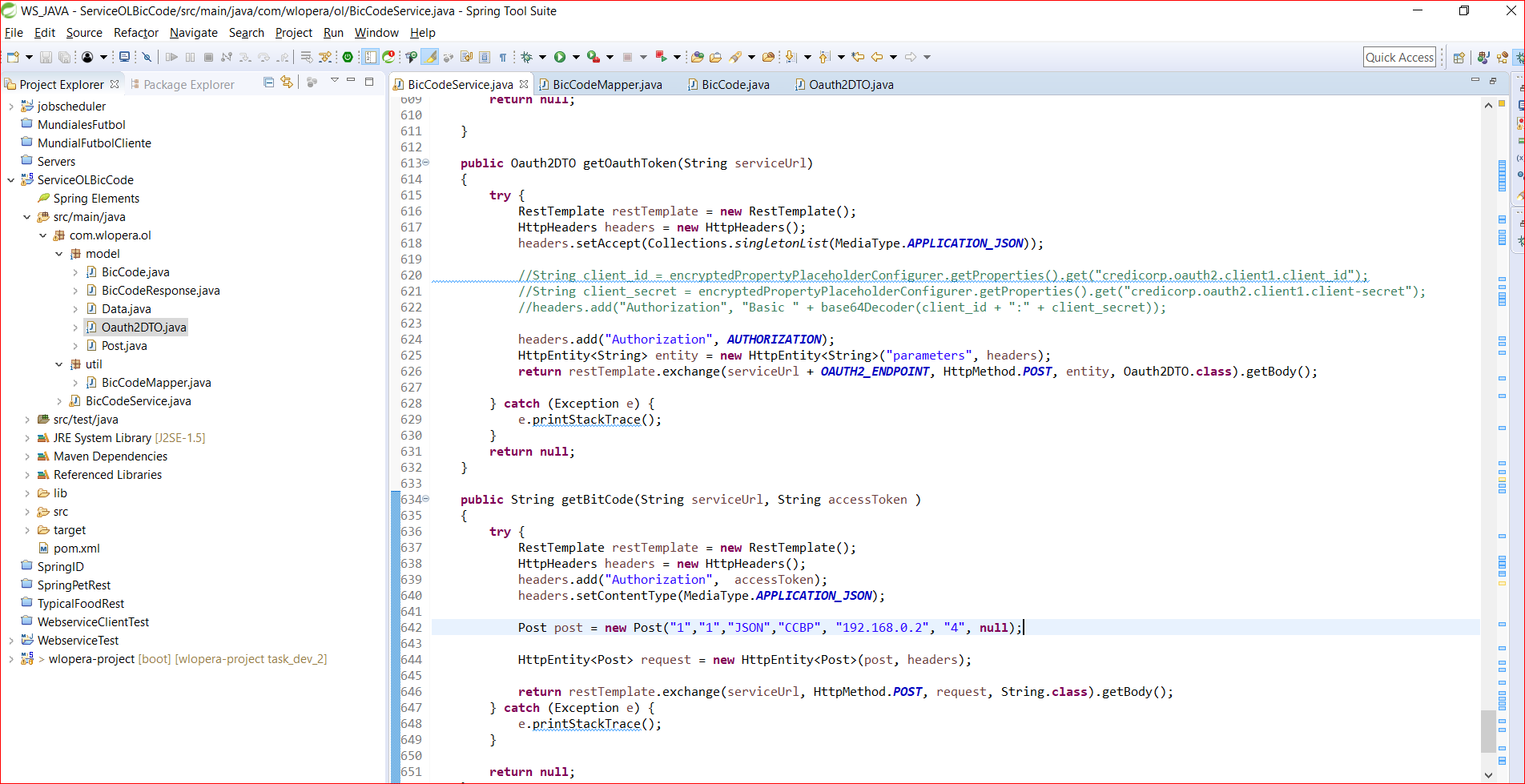
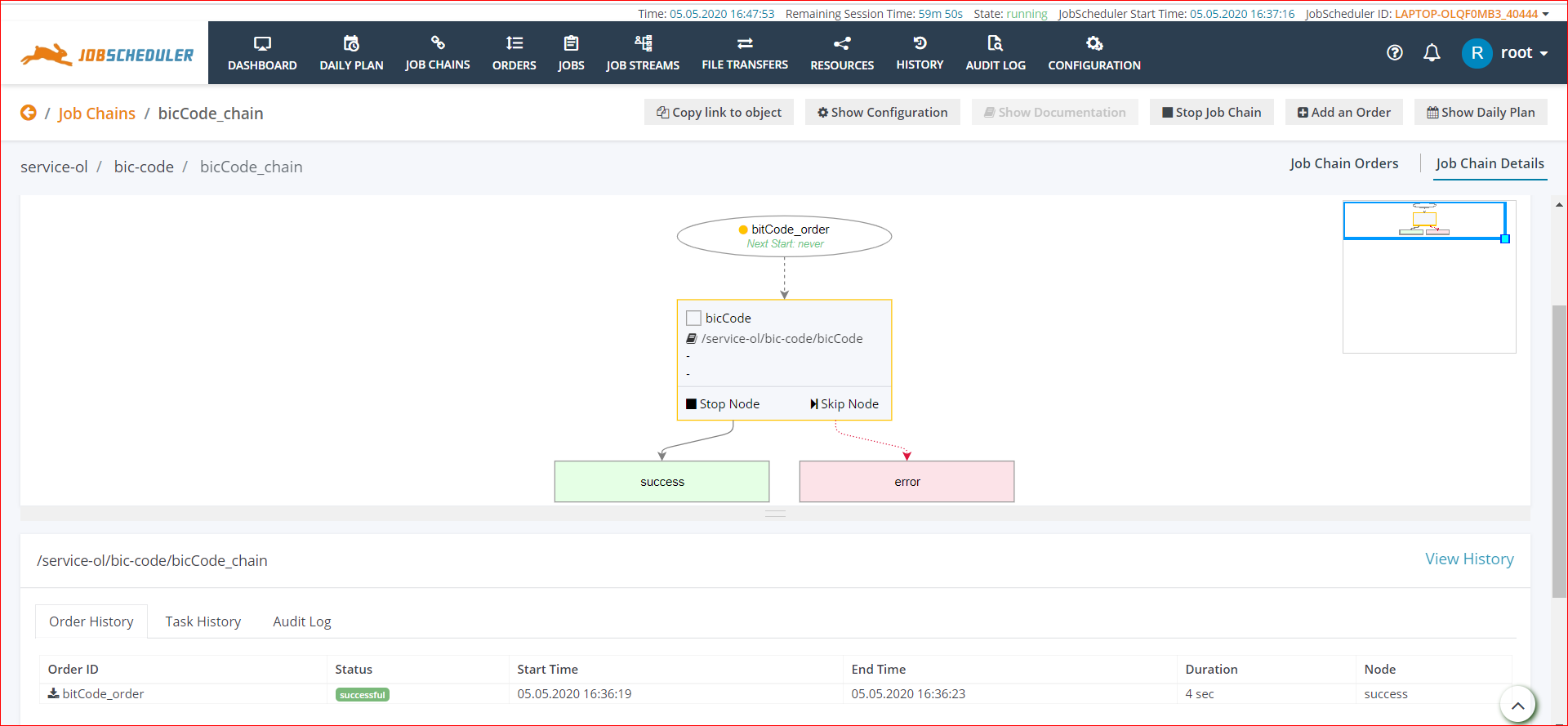
Integrar Java con JobScheduler - API Jobs (engine-job-api)

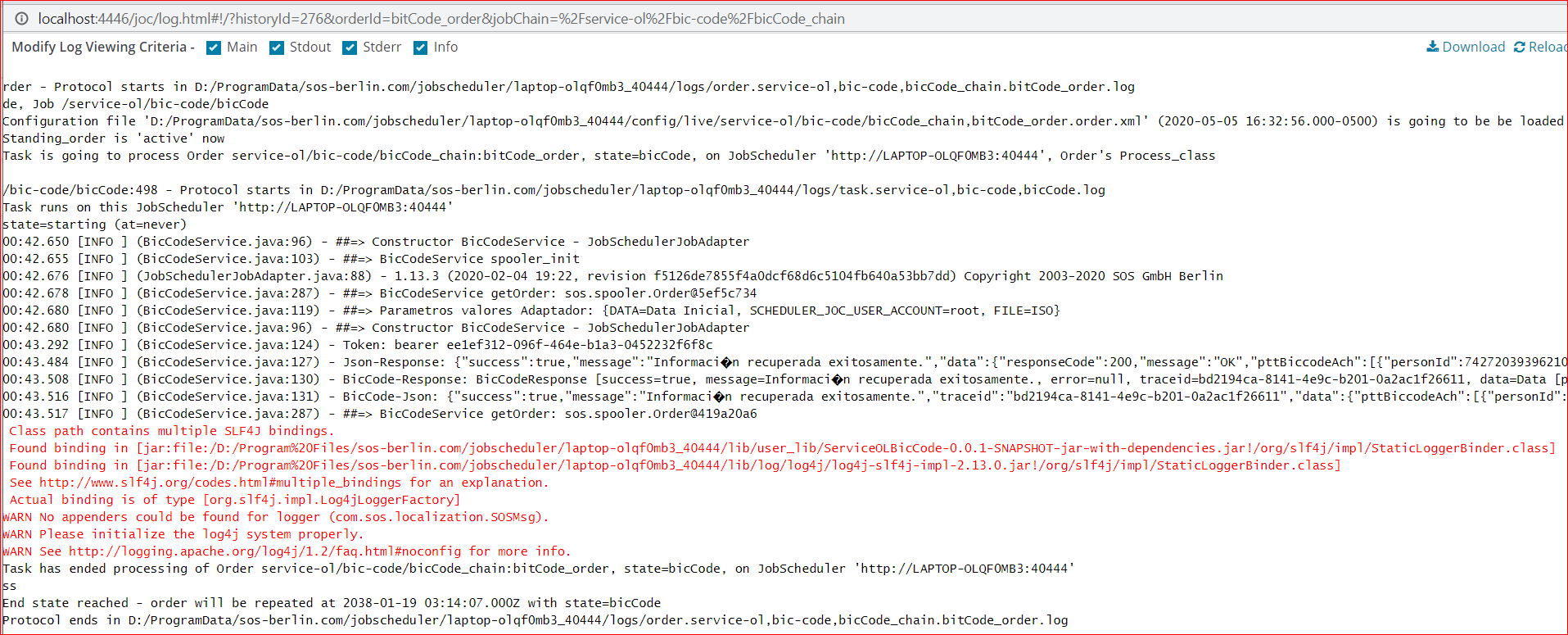
## Alcance

Este documento explica  cómo integrar el archivo JobScheduler .jar para trabajos con el API de Java (engine-job-api) en un proyecto maven.

Además, este documento proporciona un pequeño ejemplo para crear una clase de trabajo API de Java y cómo usarlo en un trabajo JobScheduler.



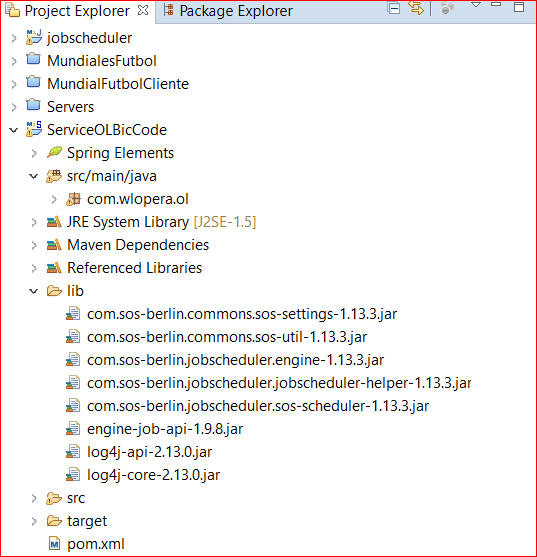




## Artefactos

* com.sos-berlin.commons.sos-settings-1.13.3.jar
* com.sos-berlin.commons.sos-util-1.13.3.jar
* com.sos-berlin.jobscheduler.engine-1.13.3.jar
* com.sos-berlin.jobscheduler.jobscheduler-helper-1.13.3.jar
* com.sos-berlin.jobscheduler.sos-scheduler-1.13.3.jar
* engine-job-api-1.9.8.jar
* log4j-api-2.13.0.jar
* log4j-core-2.13.0.jar

Muchas de las librerías tomadas de la instalación del JobSheduler:

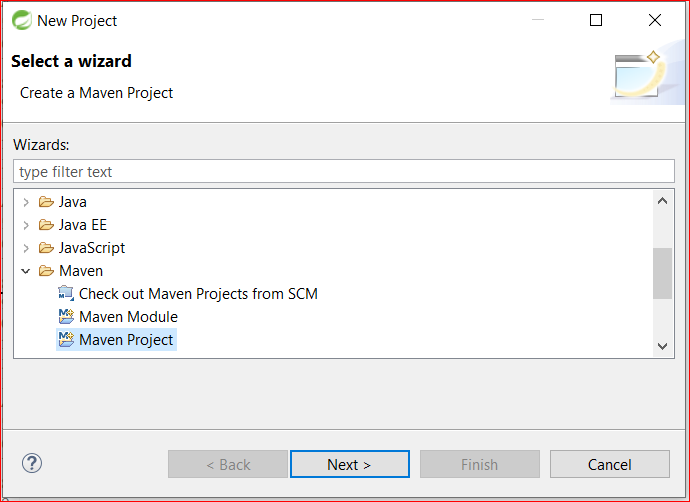


Se deben agregar al pom.xml de maven

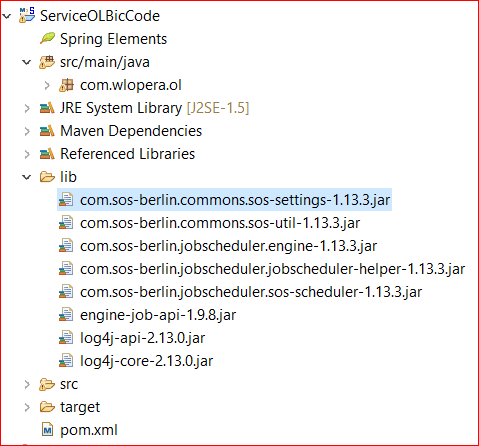
## Aplicativo:

## Proyecto JobScheduler – Api Job – Java, para consumo de un Servicio REST de Open Legacy. Consulta de Códigos BIC.

1. Crear proyecto maven



1. Agregar “*Source Folder*” => “src/main/java” : Código fuente
2. Crear carpeta Lib con los .jar básicos requeridos o “agregarlos en el pom.xml”



1. Agregar dependencias adicionales al pom.xml

<dependencies>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

<version>1.7.26</version>

</dependency>

<dependency>

<groupId>org.codehaus.jackson</groupId>

<artifactId>jackson-mapper-asl</artifactId>

<version>1.9.13</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-web</artifactId>

<version>3.0.2.RELEASE</version>

</dependency>

<dependency>

<groupId>com.google.code.gson</groupId>

<artifactId>gson</artifactId>

<version>2.8.6</version>

</dependency>

</dependencies>

1. Agregar Plugin para generar .jar con las dependencias de clases requeridas por el aplicativo (pom.xml)

<build>

<plugins>

<!-- any other plugins -->

<plugin>

<artifactId>maven-assembly-plugin</artifactId>

<executions>

<execution>

<phase>package</phase>

<goals>

<goal>single</goal>

</goals>

</execution>

</executions>

<configuration>

<descriptorRefs>

<descriptorRef>jar-with-dependencies</descriptorRef>

</descriptorRefs>

</configuration>

</plugin>

</plugins>

</build>

1. Crear clases

BicCodeService: Extiende del Adaptador de JobScheduler, clase principal donde están los métodos requeridos para interactuar con *JobScheduler* y los métodos requeridos con la lógica de consulta del *token* de conexión y del método que consulta los códigos BIC. (Esta clase debe depurarse)

BicCode: Parámetros de los códigos BIC

Oauth2DTO: Parámetros del token

Data: Parámetros con parte de la respuesta de los códigos BIC

BicCodeResponse: Respuesta del servicio de consulta de los códigos BIC

BicCodeMapper: Mapeo de datos Json <=> Objeto BicCodeResponse

**Notas Importantes del código:**

spooler\_process(): Método llamado por el JobScheduler para procesar la orden. (Se puede, entre otras cosas, agregar lógica de control y validación de la orden, por ejemplo: crear o modificar variables de la orden)

spooler\_task(): Método con acceso a la tarea (Se puede, entre otras cosas, tener acceso a los parámetros de la orden : spooler\_task.order().params())

spooler\_task\_before(): Método que permite tener acceso al proceso antes de generarse la tarea

spooler\_task\_after(): Método que permite tener acceso al proceso después de generarse la tarea

Oauth2DTO getOauthToken(String serviceUrl): Método que permite obtener el token de conexión a los servicios.

Utiliza las clases:

* RestTemplate: Cliente síncrono para realizar solicitudes HTTP, exponiendo una API de método de plantilla simple sobre bibliotecas de cliente HTTP subyacentes como JDK HttpURLConnection, Apache HttpComponents y otras.
* HttpHeaders: Una estructura de datos que representa encabezados de solicitud o respuesta HTTP, que asigna nombres de encabezado de cadena a una lista de valores de cadena.

RestTemplate restTemplate = new RestTemplate();

HttpHeaders headers = new HttpHeaders();

headers.setAccept(Collections.singletonList(MediaType.APPLICATION\_JSON));

headers.setAccept(Collections.singletonList(MediaType.APPLICATION\_JSON));

headers.add("Authorization", AUTHORIZATION);

HttpEntity<String> entity = new HttpEntity<String>("parameters", headers);

return restTemplate.exchange(serviceUrl + OAUTH2\_ENDPOINT, HttpMethod.POST, entity, Oauth2DTO.class).getBody();

getBitCode(String serviceUrl, String accessToken): Método que permite obtener los códigos BIC a partir del endpoint, el token de conexión y los parámetros de la petición.

Utiliza las clases:

* RestTemplate: Cliente síncrono para realizar solicitudes HTTP, exponiendo una API de método de plantilla simple sobre bibliotecas de cliente HTTP subyacentes como JDK HttpURLConnection, Apache HttpComponents y otras.
* HttpHeaders: Una estructura de datos que representa encabezados de solicitud o respuesta HTTP, que asigna nombres de encabezado de cadena a una lista de valores de cadena.
* Post: Clase con parámetros requeridos por la petición.

RestTemplate restTemplate = new RestTemplate();

HttpHeaders headers = new HttpHeaders();

headers.add("Authorization", accessToken);

headers.setContentType(MediaType.APPLICATION\_JSON);

Post post = new Post("1","1","JSON","CCBP", "192.168.0.2", "4", null);

HttpEntity<Post> request = new HttpEntity<Post>(post, headers);

return restTemplate.exchange(serviceUrl, HttpMethod.POST, request, String.class).getBody();

Nota: personid = null (Retorna todos los códigos BIC)