

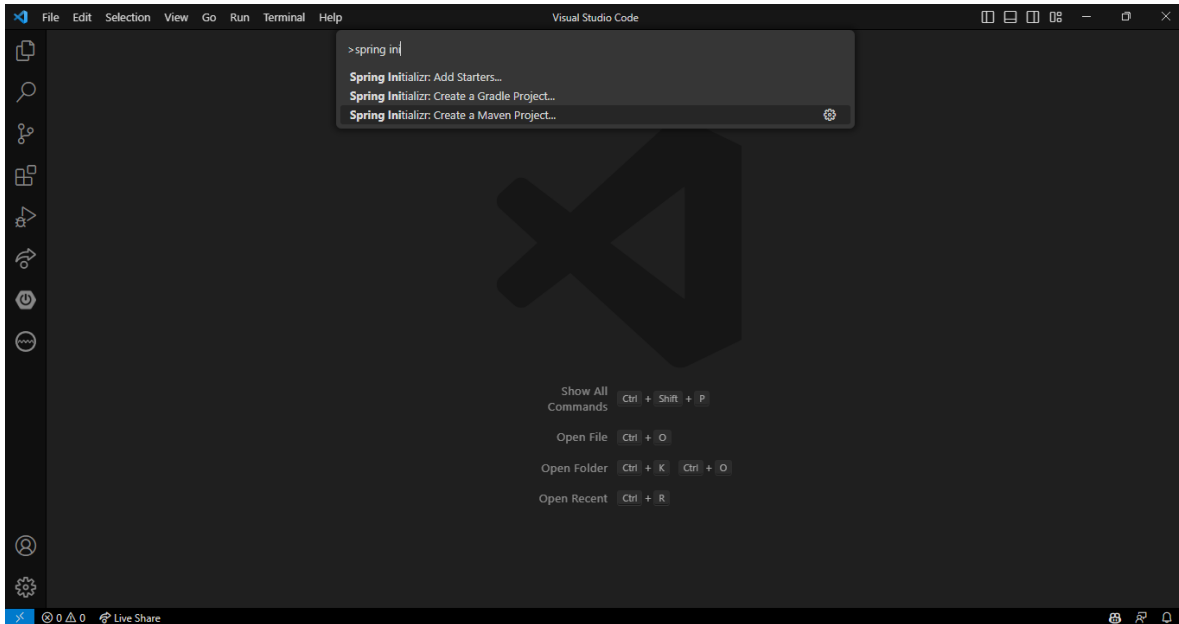
Servicios en Spring

Contenido

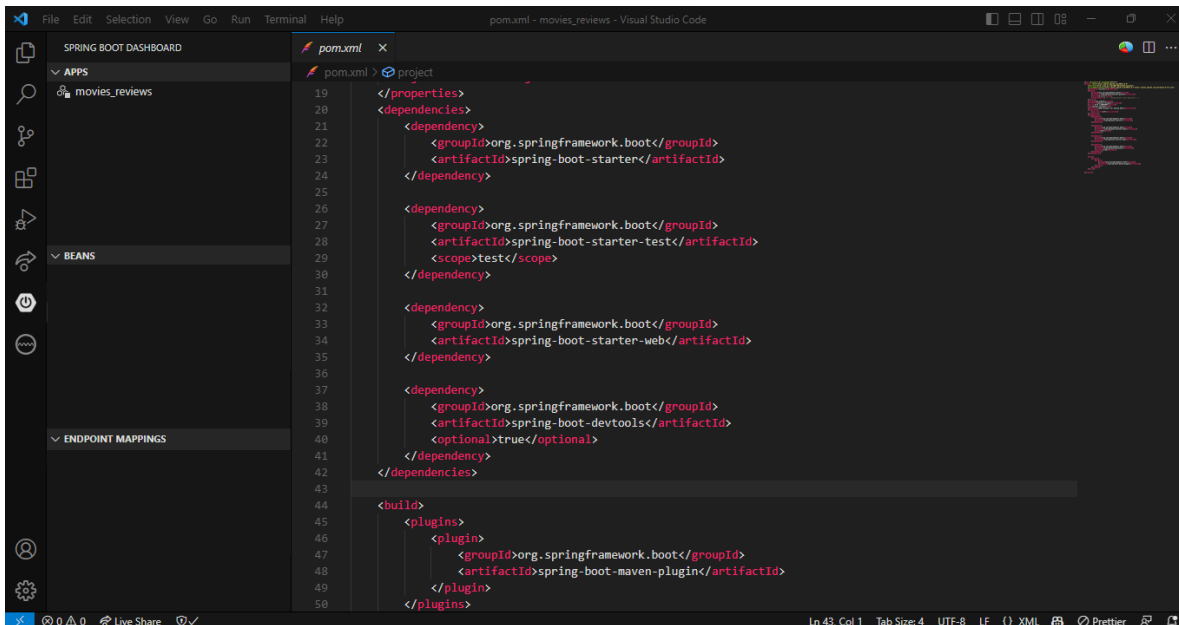
Configuración de entorno.....	2
Servicio REST	4
Servicio SOAP	7
Anotaciones	11

Configuración de entorno

Se usó el **initializer** de VSCode, teniendo previamente instaladas las extensiones adecuadas de Java y Spring. Creando un proyecto Maven, en lenguaje Java versión 17.



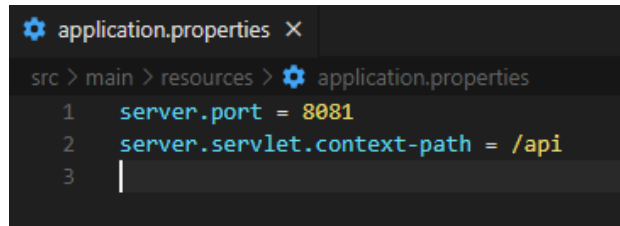
Se agregan las siguientes dependencias para configurar la aplicación web con Spring Boot y tener acceso a todas las funcionalidades proporcionadas por Spring MVC para el desarrollo de controladores y la gestión de peticiones HTTP.



spring-boot-starter-web: Esta dependencia proporciona todas las funcionalidades necesarias para desarrollar aplicaciones web en Spring Boot.

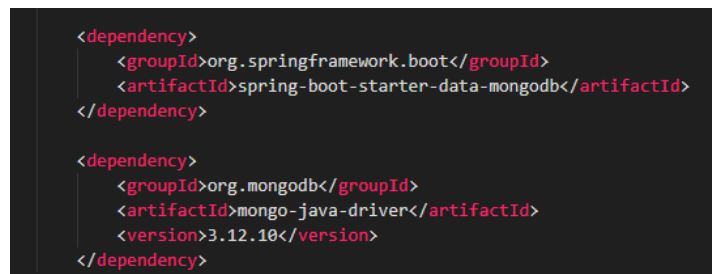
spring-boot-devtools: Esta dependencia es opcional y brinda herramientas de desarrollo adicionales para facilitar el ciclo de desarrollo en Spring Boot.

Se configura un puerto específico, un path para el servicio.

A screenshot of an IDE showing the 'application.properties' file. The breadcrumb navigation at the top reads 'src > main > resources > application.properties'. The file content shows two lines of configuration: 'server.port = 8081' and 'server.servlet.context-path = /api'. Line numbers 1, 2, and 3 are visible on the left margin.

```
src > main > resources > application.properties
1  server.port = 8081
2  server.servlet.context-path = /api
3
```

El motor de base de datos a usar será MongoDB, se usa un servicio en la nube, el cual es MongoDB Atlas. Para ello se agregan las siguientes dependencias.

A screenshot of an XML file showing two dependency declarations. The first declaration is for 'spring-boot-starter-data-mongodb' with groupId 'org.springframework.boot'. The second declaration is for 'mongo-java-driver' with groupId 'org.mongodb' and version '3.12.10'. The XML tags are color-coded: red for opening and closing tags, and black for the content.

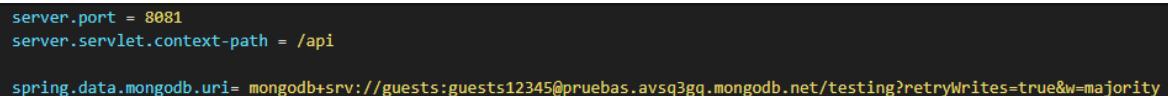
```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>

<dependency>
  <groupId>org.mongodb</groupId>
  <artifactId>mongo-java-driver</artifactId>
  <version>3.12.10</version>
</dependency>
```

spring-boot-starter-data-mongodb: Esta dependencia proporciona las funcionalidades necesarias para integrar MongoDB en una aplicación Spring Boot. Incluye el soporte para la capa de acceso a datos (repositorios, consultas, etc.) y la configuración básica para la conexión y gestión de la base de datos MongoDB.

mongo-java-driver: Esta dependencia es el controlador oficial de MongoDB para Java. Proporciona las clases y métodos necesarios para interactuar con una base de datos MongoDB, como la conexión, ejecución de consultas, inserción y actualización de documentos, entre otros.

Se agrega la URI generada para la conexión

A screenshot of an IDE showing the 'application.properties' file. It contains the same two lines as the previous screenshot, plus a third line for the MongoDB URI: 'spring.data.mongodb.uri= mongodb+srv://guests:guests12345@pruebas.avsq3gq.mongodb.net/testing?retryWrites=true&w=majority'.

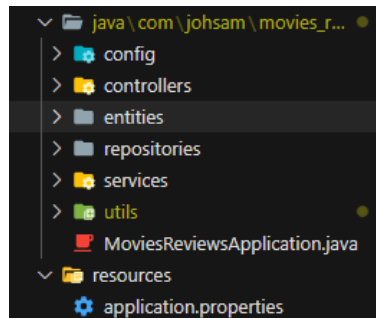
```
server.port = 8081
server.servlet.context-path = /api
spring.data.mongodb.uri= mongodb+srv://guests:guests12345@pruebas.avsq3gq.mongodb.net/testing?retryWrites=true&w=majority
```

Se usó **insomnia** para realizar las peticiones a los servicios.

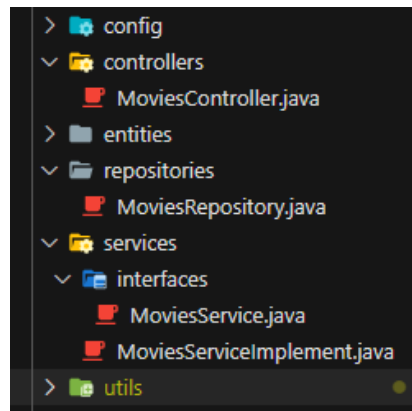
El objetivo es mostrar el proceso de creación de un CRUD de películas y de reseñas, el servicio REST se encargará de las películas, y el servicio SOAP de las reseñas de las películas.

Servicio REST

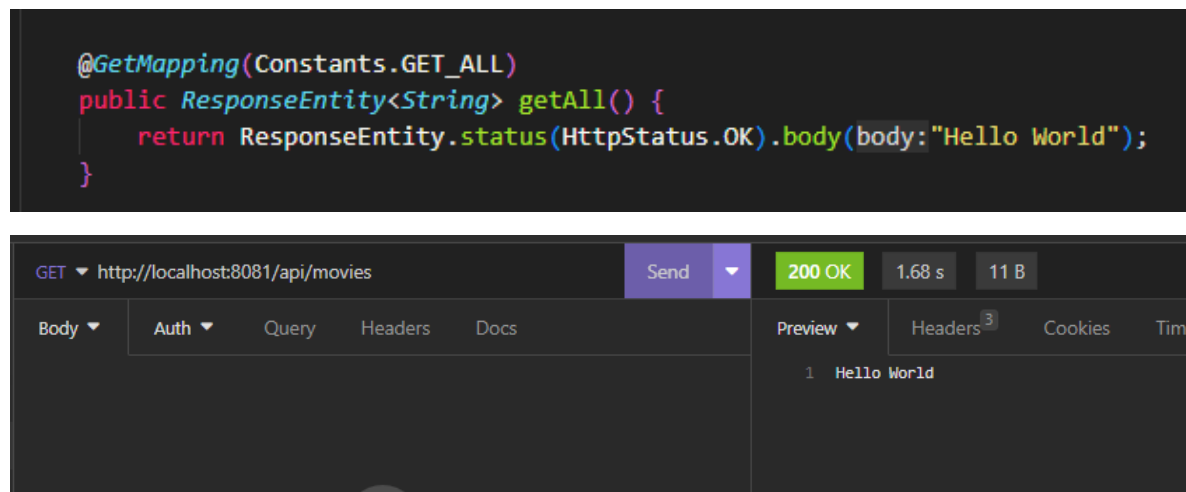
Se plantea esta estructura de carpetas inicial.



Se crean las clases correspondientes para la arquitectura planteada, para las películas.



Se crea una ruta de prueba para verificar el funcionamiento.



Se crea una estructura de respuesta para los servicios.

```
public class ResponseAPI {  
  
    @JsonProperty("message")  
    private String message;  
  
    @JsonProperty("response")  
    private Object response;  
  
    public ResponseAPI(String message, Object response) {  
        this.message = message;  
        this.response = response;  
    }  
  
}
```

Se crea el modelo de Peliculas con la notación `@Document` que se utiliza para mapear una clase Java a un documento en una colección específica de MongoDB. Esto permite realizar operaciones CRUD en la colección utilizando repositorios de Spring Data.

```
@Document(collection = "movies")  
public class Movie {  
  
    @Id  
    private String _id;  
  
    private String title;  
    private String synopsis;  
    private List<String> genre;  
    private int releaseYear;  
    private String director;  
    private String duration;  
  
}
```

Se implementa el repositorio como una interfaz que extiende de `MongoRepository`, que es un repositorio listo para usar con métodos CRUD y funcionalidades adicionales proporcionadas por Spring Data MongoDB.

```
import org.springframework.data.mongodb.repository.MongoRepository;  
  
@Repository  
public interface MoviesRepository extends MongoRepository<Movie, String>{  
  
}
```

Se implementa la interfaz de Servicio para las películas.

```
public interface MoviesService {  
  
    Movie create(Movie movie) throws Exception;  
  
    Movie getById(String id) throws Exception;  
  
    List<Movie> getAll() throws Exception;  
  
    Movie deleteById(String id);  
  
    Movie updateById(String id, Movie object);  
  
}
```

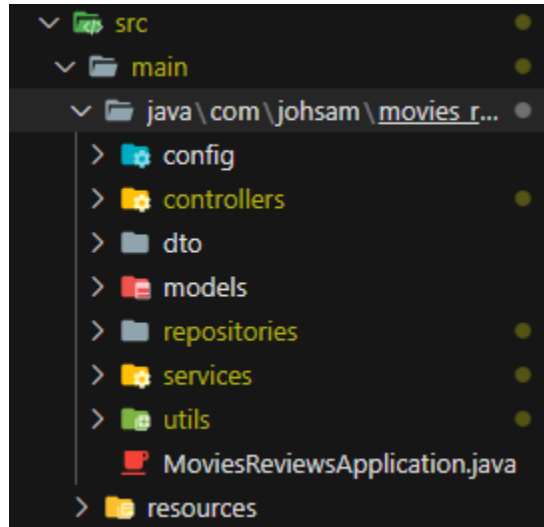
Se implementa el servicio, inyectando el repositorio, donde se aprecia el lanzamiento de excepciones que se tendrá con los mensajes de error correspondiente.

```
@Service  
public class MoviesServiceImpl implements MoviesService {  
  
    @Autowired  
    private MoviesRepository moviesRepository;  
  
    @Override  
    public Movie create(Movie movie) throws Exception {  
        Movie movieCreated = moviesRepository.save(movie);  
        if (movieCreated == null) {  
            throw new Exception("Movie not created");  
        }  
        return movieCreated;  
    }  
  
    @Override  
    public Movie getById(String id) throws Exception {  
        // TODO Auto-generated method stub  
        throw new UnsupportedOperationException("Unimplemented method 'getById'");  
    }  
}
```

Para el controlador se tendrá esta estructura, en el cual atraparé una excepción al intentar realizar cualquier acción. Respondiendo con un ResponseEntity.

```
@PostMapping(Constants.CREATE)  
public ResponseEntity<ResponseAPI> create(@RequestBody Movie movie) {  
    ResponseAPI response;  
  
    try {  
        Movie movieCreated = moviesService.create(movie);  
        response = new ResponseAPI(Constants.CREATED, movieCreated);  
        return ResponseEntity.status(HttpStatus.CREATED).body(response);  
    } catch (Exception e) {  
        response = new ResponseAPI(Constants.NOT_CREATED, e.getMessage());  
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(response);  
    }  
}
```

Al final esta es la estructura final, que se obtuvo de las carpetas. En utils, se creó un archivo de constantes los cuales ayudan a los mensajes de respuesta, y los path de las solicitudes que se reciben.



Servicio SOAP

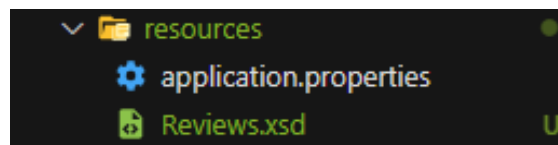
Para implementarlo, usaremos el enfoque contract-first, se comienza con el contrato WSDL, a partir del cual se generan las clases Java. Esto dentro del proyecto ya creado. Se agregan las siguientes dependencias.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web-services</artifactId>
</dependency>
<dependency>
  <groupId>wsdl4j</groupId>
  <artifactId>wsdl4j</artifactId>
</dependency>
<dependency>
  <groupId>javax.xml.bind</groupId>
  <artifactId>jaxb-api</artifactId>
  <version>2.3.1</version>
</dependency>
```

spring-boot-starter-web-services: Esta dependencia permite el desarrollo de servicios web en una aplicación Spring Boot. Incluye todas las bibliotecas necesarias para crear y consumir servicios web SOAP, utilizando la tecnología JAX-WS.

wsdl4j: Es una biblioteca Java que proporciona una API para trabajar con archivos WSDL (Web Services Description Language). WSDL es un estándar XML utilizado para describir la interfaz de un servicio web. La dependencia *wsdl4j* se utiliza para analizar, generar y manipular archivos WSDL en una aplicación Java.

jaxb-api: Esta dependencia proporciona la API de JAXB (Java Architecture for XML Binding), que es una tecnología de Java para mapear objetos Java a documentos XML y viceversa. JAXB simplifica el proceso de serialización y deserialización de objetos Java en formato XML, lo que es útil en el contexto de servicios web para convertir datos entre objetos y XML.



Como se hace uso de contrato primero, se requiere que se cree el dominio (métodos y parámetros) para el servicio. Se usa un archivo de esquema XML (XSD) que Spring-WS exportará automáticamente como WSDL. Se hace con el siguiente formato planteado para las reseñas.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://www.johsam.com/gs_ws"
  targetNamespace="http://www.johsam.com/gs_ws"
  elementFormDefault="qualified">

  <xs:complexType name="reviewSchema">
    <xs:sequence>
      <xs:element name="_id" type="xs:string" minOccurs="0" />
      <xs:element name="user" type="xs:string" />
      <xs:element name="rating" type="xs:double" />
      <xs:element name="review" type="xs:string" />
      <xs:element name="movie" type="xs:string" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="serviceStatus">
    <xs:sequence>
      <xs:element name="status" type="xs:string" />
      <xs:element name="message" type="xs:string" />
    </xs:sequence>
  </xs:complexType>

</xs:schema>
```

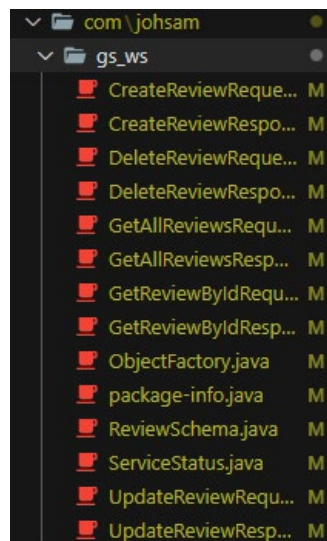
Se generan las clases de dominio a partir del archivo XSD definido. Se configura el plugin y se añade al pom.xml


```

<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>jaxb2-maven-plugin</artifactId>
  <version>1.6</version>
  <executions>
    <execution>
      <id>xjc</id>
      <goals>
        <goal>xjc</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <schemaDirectory>${project.basedir}/src/main/resources/</schemaDirectory>
    <outputDirectory>${project.basedir}/src/main/java</outputDirectory>
    <clearOutputDir>>false</clearOutputDir>
  </configuration>
</plugin>

```

Se ejecuta *mvn compile* para que las clases se generen automáticamente. Obteniendo:



Se crea de la misma manera que en la sección de REST el servicio y el repositorio para las reseñas. El modelo es el siguiente:

```

@Document(collection = "reviews")
public class Review {

    @Id
    private String _id;

    private String user;
    private double rating;
    private String review;
    private String movie;
}

```

Se genera el Endpoint, en el paquete de endpoint, el cual tendrá la clase *ReviewsEndpointSOAP*, que controla todas las solicitudes entrantes para el servicio.

```
@Component
@Endpoint
public class ReviewsEndpointSOAP {

    private static final String NAMESPACE_URI = "http://www.johsam.com/gs_ws";

    @Autowired
    private ReviewsService reviewsService;

    @PayloadRoot(namespace = NAMESPACE_URI, localPart = "getAllReviewsRequest")
    @ResponsePayload
    public GetAllReviewsResponse getAllReviews() {
        GetAllReviewsResponse response = new GetAllReviewsResponse();
        try {
            List<Review> reviewListFound = reviewsService.getAll();

            for (Review review : reviewListFound) {
                ReviewSchema reviewSchema = new ReviewSchema();
                BeanUtils.copyProperties(review, reviewSchema);
                reviewSchema.setId(review.getId());
                response.getReview().add(reviewSchema);
                response.setServiceStatus(getServiceStatus(HttpStatus.OK.toString(), Constants.SUCCESS));
            }
        } catch (Exception e) {
            response.setServiceStatus(getServiceStatus(HttpStatus.BAD_REQUEST.toString(), e.getMessage()));
        }
    }
}
```

Se configuran los Beans del servicio SOAP:

```
@Configuration
@EnableWs
public class WSConfig extends WsConfigurerAdapter {

    @Bean
    public XsdSchema reviewSchema() {
        return new SimpleXsdSchema(
            new ClassPathResource("review.xsd"));
    }

    @Bean
    public ServletRegistrationBean messageDispatcherServlet(ApplicationContext applicationContext) {
        MessageDispatcherServlet servlet = new MessageDispatcherServlet();
        servlet.setApplicationContext(applicationContext);
        servlet.setTransformWsdlLocations(transformWsdlLocations:true);
        return new ServletRegistrationBean(servlet, "/ws/*");
    }

    @Bean(name = "review")
    public DefaultWsd11Definition defaultWsd11Definition(XsdSchema reviewSchema) {
        DefaultWsd11Definition definition = new DefaultWsd11Definition();
        definition.setPortTypeName(portTypeName:"ReviewPort");
        definition.setLocationUri(locationUri:"/ws");
        definition.setTargetNamespace(targetNamespace:"http://www.johsam.com/gs_ws");
        definition.setSchema(reviewSchema);
        return definition;
    }
}
```

Se prueba el servicio

The screenshot shows a REST client interface with a POST request to `http://localhost:8081/api/ws`. The status is **200 OK** with a response time of 205 ms and a body size of 549 B. The XML view displays the following request and response:

```
1 <?xml version='1.0' encoding='UTF-8'>
2 <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
3   xmlns:gs="http://www.johsam.com/gs_ws">
4   <soapenv:Header/>
5   <soapenv:Body>
6     <gs:createReviewRequest>
7       <gs:review>
8         <gs:user>Johan</gs:user>
9         <gs:rating>4.2</gs:rating>
10        <gs:review>Fantabulous</gs:review>
11        <gs:movie>649c40bd13b4351845146c23</gs:movie>
12      </gs:review>
13    </gs:createReviewRequest>
14  </soapenv:Body>
15 </soapenv:Envelope>
```

```
1 <?xml version='1.0' encoding='UTF-8'>
2 <SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
3   xmlns:ns2="http://www.johsam.com/gs_ws">
4   <SOAP-ENV:Header/>
5   <SOAP-ENV:Body>
6     <ns2:createReviewResponse>
7       <ns2:serviceStatus>
8         <ns2:status>200 OK</ns2:status>
9         <ns2:message>Created</ns2:message>
10      </ns2:serviceStatus>
11      <ns2:review>
12        <ns2:_id>649c412f13b4351845146c24</ns2:_id>
13        <ns2:user>Johan</ns2:user>
14        <ns2:rating>4.2</ns2:rating>
15        <ns2:review>Fantabulous</ns2:review>
16        <ns2:movie>649c40bd13b4351845146c23</ns2:movie>
17      </ns2:review>
18    </ns2:createReviewResponse>
19  </SOAP-ENV:Body>
20 </SOAP-ENV:Envelope>
```

El proceso de implementación SOAP fue basado en: [Creating a SOAP Web Service with Spring | Baeldung](#)

Anotaciones

- Al momento de implementar SOAP se creó en una rama aparte.
- Uno de los problemas que mas se tuvo fue la generación de la solicitud, en cuanto a como se estructuraba y los encabezados.
- En la primera implementación se obtuvo un error al realizar las solicitudes, lo cual se respondía con la siguiente información.

No adapter for endpoint [public com.johsam.gs_ws.GetAllReviewsResponse com.johsam.movies_reviews.endpoints.ReviewsEndpointSOAP.getAllReviews()]: Is your endpoint annotated with @Endpoint, or does it implement a supported interface like MessageHandler or PayloadEndpoint?

El cual después de inventar varias opciones usando ChatGPT y foros, se decidió implementarlo nuevamente.

- Se creó una segunda rama de SOAP, la cual se implementó desde cero paso a paso, sin embargo, se llegó al mismo error. La solución final realizada fue cambiar las versiones, ya que en un foro se mencionaba que posiblemente era problemas de configuración Maven. Así que se cambió a Java 8, y se ejecutó correctamente.