SOAP exam platform documentation

1. Introduction

This document provides an overview of my SOAP project.
The platform consists of a client (hypothetically convertible into a smartphone app or something similar) that communicates with a server exposing a WSDL and providing three main operations. These operations allow:

- A user to request their own registered exam from the academic record;
- A user to request their entire university transcript;
- Enable professors to add new exams to students' academic records.

The platform requires both the client and the server to communicate securely with each other. The client sends encrypted, authenticated, signed messages with timestamps to prevent replay attacks. The server verifies the aforementioned aspects and responds with encrypted and signed messages, also including timestamps that the client verifies.

2. Technology stack and configurations

I have used Java as the programming language, MongoDB as the database, Maven for building the project and managing Java dependencies, and Spring (specifically Spring-WS) as the framework that implements the necessary technology stack for creating SOAP micro-services.

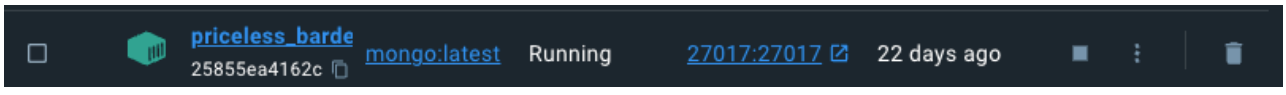
Let's explore each individual technology in more detail.

2.1 MongoDB

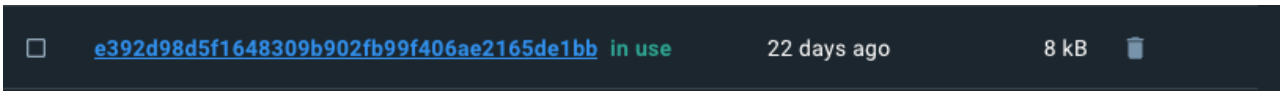
The project utilizes MongoDB as the database to store various registered votes and user data for the “UsernameToken authentication”. Specifically, the database stores username-password pairs. However, for security reasons, passwords are not stored in plain text but in hashed form. A highly robust and modern algorithm was chosen for this purpose, aiming to minimize collisions. Specifically, SHA3-512 has been used.
In this configuration the database is at localhost:27017. It runs in a docker.

Docker image:

Docker instance:

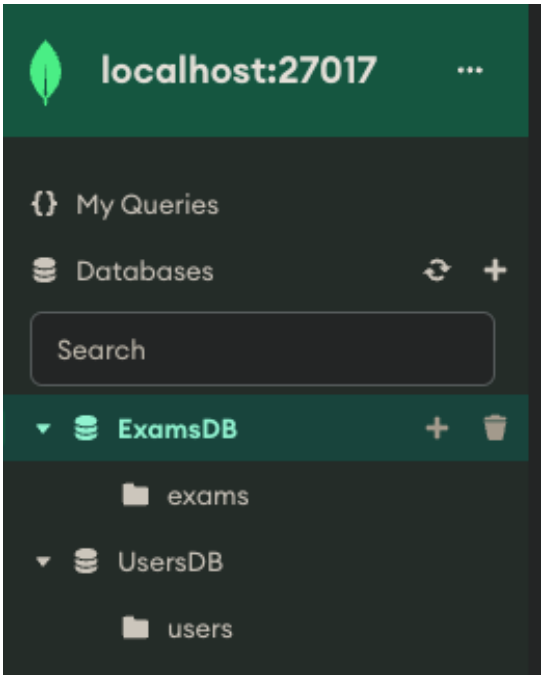


To ensure that Docker maintains the database memory, a mapped volume has been created in the host machine's file system:

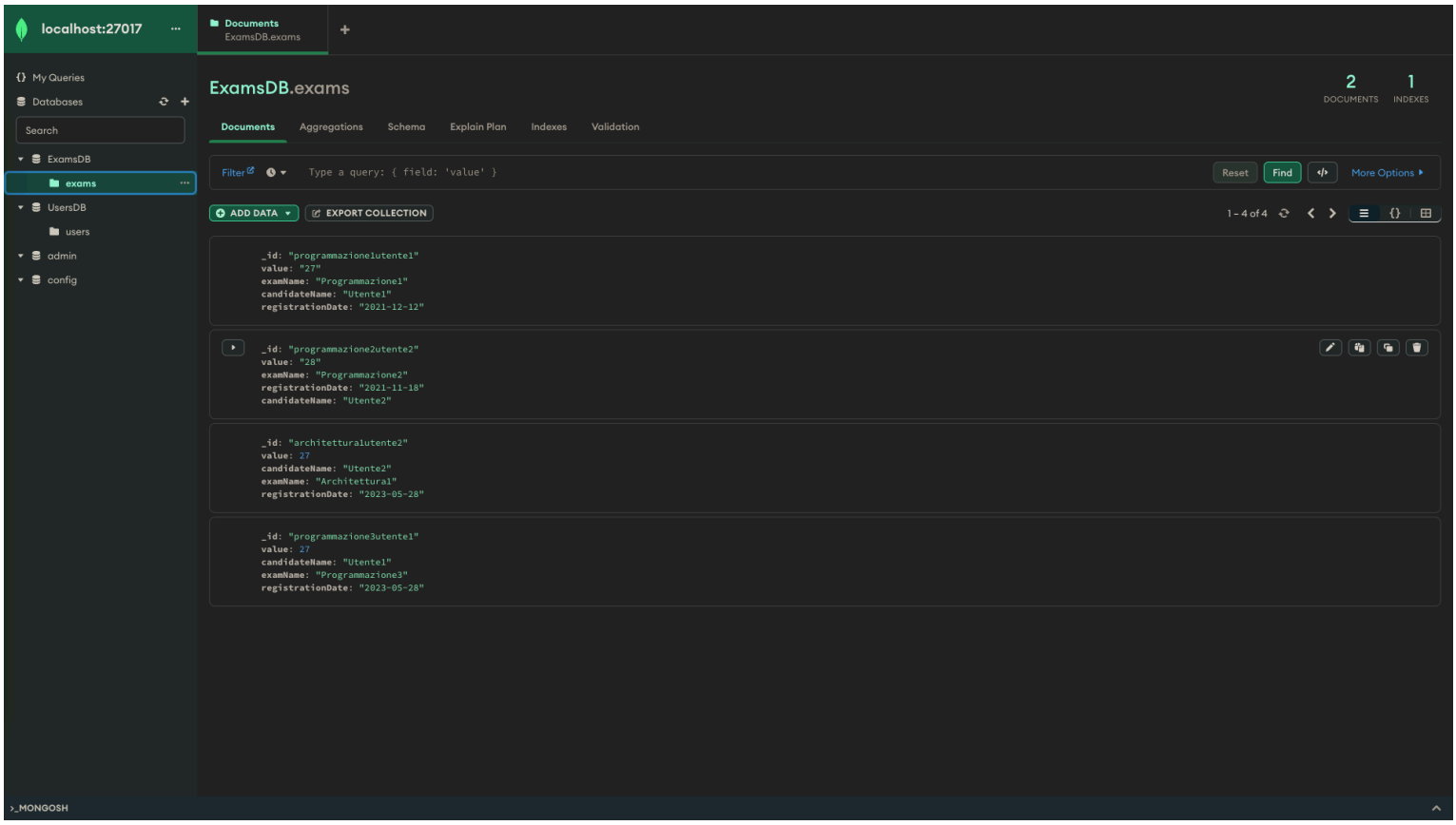


Of course, the port 27017 of the Docker container needs to be mapped to a port on the host machine. For clarity, in my case, I have mapped port 27017 to port 27017.

In mongoDB there are two databases with one collection each one:

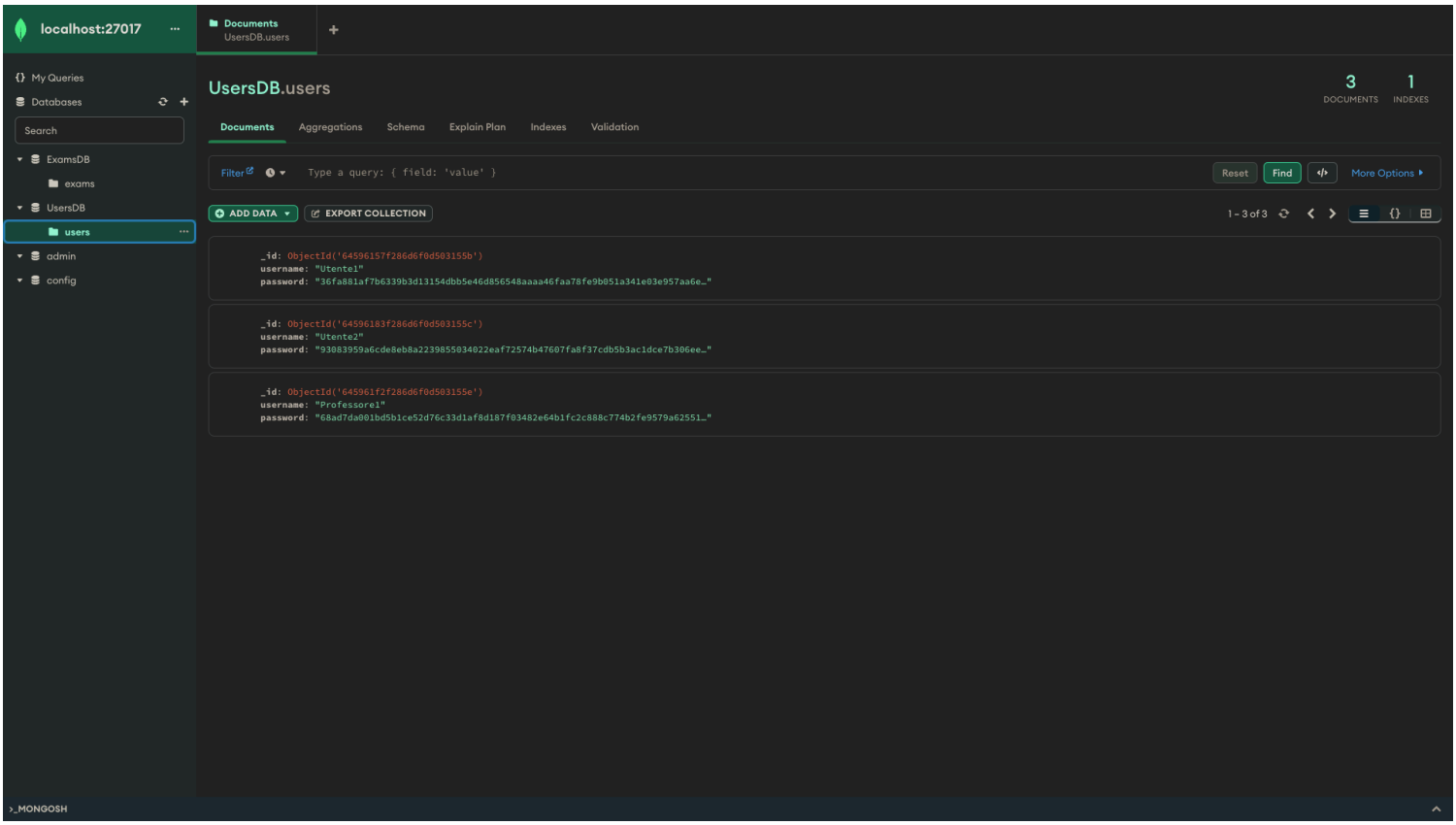


The first DB is the “ExamsDB” which contains a collection named “exams”. This collection contains all the registered exams:



The second DB is the “UsersDB” which contains a collection named “users”. This collection contains the username-password pair for each user.

As previously mentioned, the passwords stored in the "users" collection are not stored in plain text within the database. Instead, they are stored in hashed form using the highly robust SHA3-512 algorithm, which is resistant to collisions. Even in the event of a database compromise, the passwords would not be readable:



Since this is a demonstration project, I have created three simple users named “Utente1”, “Utente2" and “Professore1". Their respective passwords are “passwordutente1”, “passwordutente2”, and “passwordprofessore1”.

MongoDB is a popular NoSQL database that provides a flexible and scalable approach to data storage. It is designed to handle large volumes of unstructured or semi-structured data, making it suitable for a wide range of applications. MongoDB uses a document-oriented data model, where data is stored in flexible, JSON-like documents.

In the context of this project, MongoDB was chosen as the database due to its suitability for a demonstration project with few entity relationships. Given the nature of the project and the lack of complex data relationships, using MongoDB instead of a relational database was preferred. MongoDB's schema-less nature and ability to store data in a more flexible and agile manner align well with the requirements of the project.

Using MongoDB allowed for easy and efficient storage and retrieval of data, while avoiding the need to define strict schemas or perform complex joins between tables. This flexibility and simplicity of MongoDB made it a suitable choice for this demonstration project.

2.2 Maven

Maven is a powerful build automation and project management tool used primarily in Java-based projects. It provides a comprehensive infrastructure for managing project dependencies, compiling source code, packaging applications, and executing various build tasks.

At its core, Maven utilizes a declarative XML-based configuration file, known as the POM (Project Object Model), which describes the project structure, dependencies, build plugins, and other

essential project information. Maven uses this information to automate the build process and enforce project conventions.

One of the key features of Maven is its dependency management system. It simplifies the process of managing external libraries and frameworks by automatically resolving and downloading the required dependencies from remote repositories. Maven ensures that the project builds consistently across different development environments and allows for easy collaboration among developers.

Maven promotes a standardized project structure and provides a wide range of predefined build plugins that can be easily integrated into projects. These plugins enable tasks such as compiling source code, running tests, generating documentation, packaging the application into a distributable format, and deploying artifacts to remote repositories.

Overall, Maven offers a robust and standardized approach to building and managing Java projects, enhancing productivity, and ensuring reliable and reproducible builds.

Inside the POM of this project, it is possible to identify some interesting code snippets.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.0.6</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.soapproject</groupId>
    <artifactId>soap-project-web-service</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>soap-project-web-service</name>
    <description>Project SOAP</description>
    <properties>
        <java.version>17</java.version>
    </properties>
```

In the above image, you can see the declaration of the data of the XML file.
For example, it is possible to see the file encoding definition and indications of where to retrieve the namespaces that specify the tags found within the POM. Additionally, there are other minor properties such as the project name, version, and more.

The most interesting part for the project's functionality is the definition of various dependencies, which allows importing additional libraries into different classes of the project. You can see the dependencies for this project below.

Another interesting section of this file is inside the <plugin> tag.
In this section, it is evident that JAXB2 will be used to compile the project and translate Java objects into their corresponding XML objects (SOAP relies heavily on XML). To construct the XML objects and the WSDL, the file's pathname that serves as the source for everything needs to be provided. In our case, it is "exams.xdl," which we will discuss later.

You can see also this code below.

```xml
    <dependencies>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web-services</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>

        <dependency>
            <groupId>wsdl4j</groupId>
            <artifactId>wsdl4j</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.ws</groupId>
            <artifactId>spring-ws-security</artifactId>
        </dependency>

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

        <dependency>
            <groupId>commons-codec</groupId>
            <artifactId>commons-codec</artifactId>
            <version>1.15</version>
        </dependency>

    </dependencies>
```

```xml
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>

            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>jaxb2-maven-plugin</artifactId>
                <version>3.1.0</version>
                <executions>
                    <execution>
                        <id>xjc</id>
                        <goals>
                            <goal>xjc</goal>
                        </goals>
                    </execution>
                </executions>
                <configuration>
                    <sources>
                        <source>${project.basedir}/src/main/resources/exams.xsd</source>
                    </sources>
                    <outputDirectory>${project.basedir}/src/main/java</outputDirectory>
                    <clearOutputDir>false</clearOutputDir>
                </configuration>
            </plugin>

        </plugins>
    </build>
```

2.3 Spring

Spring is a widely-used framework in the Java ecosystem that provides comprehensive support for building robust and scalable applications. It offers various modules and features that simplify the development process, enhance code quality, and promote good design practices.

When it comes to creating SOAP micro-services, Spring provides the Spring Web Services (Spring-WS) module. With Spring-WS, developers can leverage the power of the Spring framework to build SOAP-based web services. It offers a convenient way to define the service contracts, handle SOAP messages, and integrate with other Spring components.

Using Spring-WS, developers can annotate Java classes to define the SOAP endpoints, map incoming SOAP requests to specific methods, and generate SOAP responses. It provides flexibility in handling XML transformations, message validation, and data binding.

Spring-WS also integrates well with other Spring modules, such as Spring Dependency Injection (Spring DI) for managing dependencies, Spring Data for database interactions, and Spring Security for authentication and authorization in SOAP-based services.

In summary, Spring is a versatile framework that can be used to create SOAP micro-services by leveraging the Spring-WS module. It simplifies the development process, promotes code reuse, and provides robust support for handling SOAP messages and service contracts.

3.0 Comment on the "server.jks" and "client.jks" files

JKS (Java KeyStore) files are used in Java-based systems to store digital certificates and private keys. They play a crucial role in securing communication channels by enabling encryption, authentication, and integrity checks. JKS files are commonly used for SSL/TLS protocols and establish trust between parties. These files ensure secure connections and protect sensitive information transmitted over networks. Proper management and protection of JKS files are essential to maintain system security.

For this project, the "server.jks" file was created on the server-side. This file contains a generic client certificate (i.e., the client's public key) and certificates for individual users (i.e., the public keys of each user). Additionally, it also includes the server's private key, which is used to sign the response messages sent to the client.

The "client.jks" file contains the server's certificate and its own private key. Messages from the client to the server are signed with the server's public key, which is then decrypted by the server using its private key. The "client.jks" file also includes all the private keys of the various users, which they use to sign messages sent to the SOAP server with a very fine granularity, specific to each individual user rather than at the client level.

Here are the commands used to create what was just described.

Create the two crypto files and create the public-private key pairs for client and server:

- keytool -genkeypair -alias client -keyalg RSA -keysize 2048 -dname "CN=namespacetest.com -validity 365 -storetype JKS -keystore client.jks -storepass clientkeystore
- keytool -genkeypair -alias server -keyalg RSA -keysize 2048 -dname "CN=namespacetest.com -validity 365 -storetype JKS -keystore server.jks -storepass serverkeystore

Export the public certificate for the server and the client:

- keytool -export -file server.cert -keystore server.jks -storepass serverkeystore -alias server
- keytool -export -file client.cert -keystore client.jks -storepass clientkeystore -alias client

Import the public certificate of the server in the client and viceversa:

- keytool -import -file server.cert -keystore client.jks -storepass clientkeystore -alias serverpublic
- keytool -import -file client.cert -keystore server.jks -storepass serverkeystore -alias clientpublic

Generate all the user public-private keys pairs and export the public certificate:

- -genkeypair -alias Utente1 -keyalg RSA -keysize 2048 -dname "CN=namespacetest.com" -validity 365 -storetype JKS -keystore client.jks -storepass clientkeystore
- -genkeypair -alias Utente2 -keyalg RSA -keysize 2048 -dname "CN=namespacetest.com" -validity 365 -storetype JKS -keystore client.jks -storepass clientkeystore
- -genkeypair -alias Professore1 -keyalg RSA -keysize 2048 -dname "CN=namespacetest.com" -validity 365 -storetype JKS -keystore client.jks -storepass clientkeystore
- keytool -export -file utente1.cert -keystore client.jks -storepass clientkeystore -alias Utente1
- keytool -export -file utente2.cert -keystore client.jks -storepass clientkeystore -alias Utente2
- keytool -export -file professore1.cert -keystore client.jks -storepass clientkeystore -alias Professore1

Import all the certificate about the users in the server keystore:

- keytool -import -file utente1.cert -keystore server.jks -storepass serverkeystore -alias Utente1
- keytool -import -file utente2.cert -keystore server.jks -storepass serverkeystore -alias Utente2
- keytool -import -file professore1.cert -keystore server.jks -storepass serverkeystore -alias Professore1

N.B.: who create the users' keys pairs in the client.jks you must use as the password not the same of the keystore but the SHA3-512 of the password used for the user. This because we are using bot the "UsernameToken" and the "Signature" in Spring and in it, these element use both the "securityInterceptor.setSecurementUsername()" and the "securityInterceptor.setSecurementPassword()" and thus, it is necessary for both the alias value and the password used for signing to be the same as the (hashed) password and username values stored in the database, which are used for authentication through username-password.

4.0 The XSD file used to generate the Java classes that matched the WSDL

In Spring projects, a "contract-first" approach is used, where the structure of outgoing SOAP messages from the client and client responses is defined in an .XSD file. In our case, the file is named "exams.xsd" and it defines the structure of queries, responses, and some types belonging to a custom namespace. This XSD file is then used by JAXB2 to generate the WSDL and Java classes representing the described objects. The XSD file follows an XML structure.

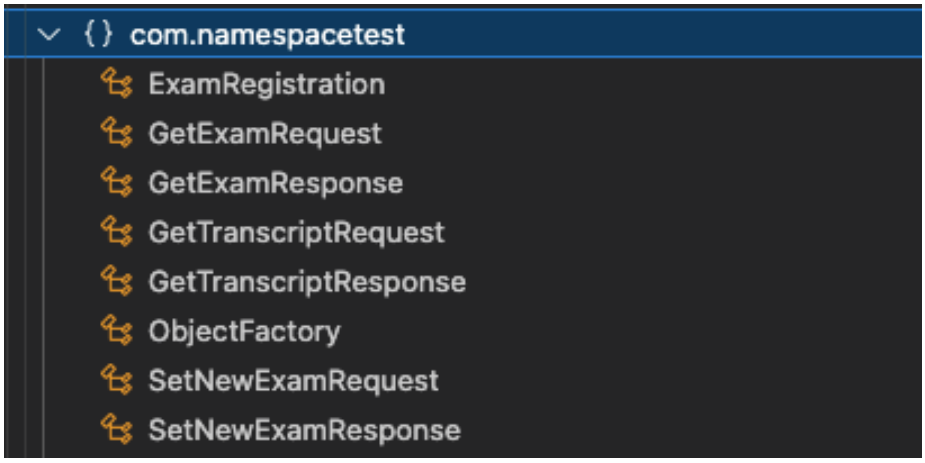
Within the file, a simple type has been defined that restricts the grade of a registered exam between 18 and 31 (where 31 represents the highest grade). Additionally, a complex type has been defined to represent an "Exam registration" stored in the database. This complex type consists of a grade, a student associated with the grade, the date of grade insertion in the transcript (automatically calculated on the Java side during a grade insertion request), and the name of the exam.

```
    <!-- Complex types definition -->
    <xs:complexType name="examRegistration">
        <xs:sequence>
            <xs:element name="value" type="tns:markValue"/>
            <xs:element name="examName" type="xs:string"/>
            <xs:element name="candidateName" type="xs:string"/>
            <xs:element name="registrationDate" type="xs:string"/>  <!-- date format: YYYY-mm-dd -->
        </xs:sequence>
    </xs:complexType>

    <xs:simpleType name="markValue">
        <xs:restriction base="xs:integer">
            <xs:minInclusive value="18"/>
            <xs:maxInclusive value="31"/>
        </xs:restriction>
    </xs:simpleType>

</xs:schema>
```

Within the XSD file, the custom namespace is specified, which will be used as a container for these types as well as for various requests and responses defined within this XSD file.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://namespacetest.com"
    targetNamespace="http://namespacetest.com"
    elementFormDefault="qualified"

>
```

JAXB2, based on this definition, creates a directory within the Java project with the name of the namespace. Inside this directory, all the classes for the defined requests and responses at the XSD level are generated and inserted.



5.0 The client side

The client within this project consists of a small Java program that includes a main method with several checks. For example, it verifies that the provided grade is between 18 and 31 and that the user inserting a grade is a "professor" user. It also ensures that all required arguments are passed. Additionally, the client allows the user to input a username and password for subsequent server-side authentication. The password provided by the user is transformed by calculating its hash using the SHA3-512 algorithm, and then it is checked on the server side. This client is complemented by two other classes: "ExamsClient" and "ExamsClientConfig". The former handles the marshalling of requests and responses (in other words, it maps which requests should be associated with which responses), while the latter contains client configurations (such as @Bean annotations), including the security interceptor.

The entry point is the “main” method:

```java
public class RunClient {
    Run | Debug
    public static void main(String[] args) throws Exception {
        String action;
        String username;
        String password;
        try {
            action = args[0];
            username = args[1];
            password = args[2];
        } catch (java.lang.ArrayIndexOutOfBoundsException exception) {
            System.out.println(x:"\n\nPlease specify an action to execute as first argument: you can choose between 'AddExam', 'RequireAnExam', 'RequireTranscript'.");
            System.out.println(x:"Please insert your username as second argument");
            System.out.println(x:"Please specify your password as third argument\n\n");
            return ;
        }

        // Configure the configurations for the client (example the security interceptor on the client side)
        ExamsClientConfig clientConfig = new ExamsClientConfig();
        password = DigestUtils.sha3_512Hex(password);
        clientConfig.setPassword(password);
        clientConfig.setUsername(username);
        if(action.compareTo(anotherString:"RequireAnExam") == 0) {
            clientConfig.setActionToExecute(actionToExecute:"getExamRequest");
        }
        else if(action.compareTo(anotherString:"RequireTranscript") == 0) {
            clientConfig.setActionToExecute(actionToExecute:"getTranscriptRequest");
        }
        else if(action.compareTo(anotherString:"AddExam") == 0){
            clientConfig.setActionToExecute(actionToExecute:"setNewExamRequest");
        }

        ExamsClient examsClient = clientConfig.getExamsClient();

```

In the first part show above, you can see that the main take the first three arguments from the input (in my case the STDIN is the terminal), If any of these arguments are not passed, the code captures the exception and provides an explanation to the user regarding what should be entered.

An clientConfig of type “ExamsClientConfig” is created and a password and username are seated into the class’ instance arguments. setPassword and setUsername are setter methods.

Based on the action requested by the user, another variable of the class instance is set, which will be used to set the part of the message to be encrypted.

Furthermore, an instance of the "ExamsClient" class is created based on the corresponding @Bean within the configuration class (i.e., the previously instantiated ExamsClientConfig).

In the subsequent lines of this main, the different cases are then handled, along with their respective checks and exception handling.

As we just mentioned, the client also consists of the ExamsClient class. An instance of this class (as seen earlier) is created in the main method using the configuration @Bean from the ExamsClientConfig class. This class, as previously mentioned, simply maps the requests to their corresponding responses.

```java
public class ExamsClient extends WebServiceGatewaySupport {
    public String username = "";
    public String password = "";


    public void setUsername(String username) {
        this.username = username;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public GetExamResponse getOneExam(GetExamRequest request)  {
        GetExamResponse examResponse = (GetExamResponse) getWebServiceTemplate().marshalSendAndReceive(request, new SoapActionCallback(soapAction:"http://namespacetest.com/getExamRequest"));

        return examResponse;
    }

    public GetTranscriptResponse getTranscript(GetTranscriptRequest request) {
        GetTranscriptResponse transcriptResponse = (GetTranscriptResponse) getWebServiceTemplate().marshalSendAndReceive(request, new SoapActionCallback(soapAction:"http://namespacetest.com/getTranscriptRequest"));

        return transcriptResponse;
    }

    public SetNewExamResponse getNewInsertedExam(SetNewExamRequest request) {
        SetNewExamResponse newExamResponse = (SetNewExamResponse) getWebServiceTemplate().marshalSendAndReceive(request, new SoapActionCallback(soapAction:"http://namespacetest.com/setNewExamRequest"));

        return newExamResponse;
    }
}

```

One important aspect to note is that since the endpoints (which we will see later) have been handled using @SoapAction, a new argument has been passed on each call of the "marshalSendAndReceive" function in this class. This argument creates a new "SoapActionCallback" that indicates which URL to contact for each call of the client-side functions. In other words, it maps the client-side functions to the addresses of the operations exposed by the server.

An important aspect of the client (as well as later for the server) are the configuration beans of the client itself. They have been placed in the "ExamsClientConfig" class. In the code shown below, you can see the typical Spring annotation for configuration classes, which is @Configuration. The other typical annotation for these classes is @Bean. An @Bean annotation effectively distinguishes an element that composes the client in this case. The variables and setter methods mentioned earlier, which are set in the "main" method, are also visible.

```java
@Configuration
public class ExamsClientConfig {
    private String username = "";
    private String password = "";
    private String actionToExecute = "";


    public void setUsername(String username) {
        this.username = username;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public void setActionToExecute(String actionToExecute) {
        this.actionToExecute = actionToExecute;
    }
```

This class is also of great interest for this project because it sets up the "Security interceptor" (in this case, a Wss4jSecurityInterceptor, which is currently the only one supported by Spring) that handles all the client-side security aspects. Let's take a closer look at it.

5.1 Detailed discussion of client-side security interceptor

The client-side security interceptor code will be shown below. It is a client component (thus an @Bean) that intercepts outgoing messages from the client and applies some rules and configurations on the messages.

```
57      @Bean
58      public Wss4jSecurityInterceptor securityInterceptorClient() throws IOException, Exception {
59          Wss4jSecurityInterceptor securityInterceptor = new Wss4jSecurityInterceptor();
60          securityInterceptor.setSecurementActions(securementActions:"UsernameToken Timestamp Signature Encrypt");
61          securityInterceptor.setSecurementUsername(this.username);
62          securityInterceptor.setSecurementPassword(this.password);
63          securityInterceptor.setSecurementSignatureCrypto(getCryptoFactoryBeanClient().getObject());
64
65          securityInterceptor.setSecurementEncryptionUser(securementEncryptionUser:"serverpublic");
66          securityInterceptor.setSecurementEncryptionCrypto(getCryptoFactoryBeanClient().getObject());
67          securityInterceptor.setSecurementEncryptionParts("{Content}{http://namespacetest.com}" + this.actionToExecute);
68
69          // Validate incoming response
70          securityInterceptor.setValidationActions(actions:"Timestamp Signature Encrypt");
71          securityInterceptor.setValidationSignatureCrypto(getCryptoFactoryBeanClient().getObject());
72          securityInterceptor.setValidationDecryptionCrypto(getCryptoFactoryBeanClient().getObject());
73          securityInterceptor.setValidationCallbackHandler(keyStoreCallbackHandlerClient());
74          securityInterceptor.setValidationTimeToLive(validationTimeToLive:10); //10 second time windows to consider the request valid
75
76
77          securityInterceptor.afterPropertiesSet();
78
79          return securityInterceptor;
80      }
```

The security actions to be applied on messages are: UsernameToken (authentication via username-password), addition of a timestamp to outbound messages (which allows to limit Replay attacks), addition of a signature to the messages (as previously mentioned these are signed with the private key of the specific user launching the action from the main, contained in the "client.jks") and encryption of the entire message.

It is also evident that the username and password for authentication are set using the "setSecurementUsername" and "setSecurementPassword" functions. It is important to note that the value passed to these functions is also used to retrieve the specific user's alias and password from the "client.jks" file. The alias represents the user's identity, and the password is used to access the alias's content, which includes the user's private key used for message signing. Therefore, it is crucial that the alias matches the user's username and that the alias's password in the keystore is the SHA3-512 hash of the user's password.

To handle the opening of a keystore in Spring, you can use a "@Bean" of type "CryptoFactoryBean". With this bean, we can open a JKS file and read its contents:

```
33      @Bean
34      public CryptoFactoryBean getCryptoFactoryBeanClient() throws Exception {
35          org.apache.xml.security.Init.init();
36
37          CryptoFactoryBean cryptoFactoryBean = new CryptoFactoryBean();
38
39          cryptoFactoryBean.setKeyStoreType(type:"jks");
40          ClassPathResource pathToFile = new ClassPathResource(path:"client.jks");
41          cryptoFactoryBean.setKeyStoreLocation(pathToFile);
42          cryptoFactoryBean.setKeyStorePassword(password:"clientkeystore");
43          cryptoFactoryBean.afterPropertiesSet();
44
45          return cryptoFactoryBean;
46      }
```

In the subsequent lines of code in the security interceptor, the alias is set to access the server's public key used for encrypting outgoing messages (the alias in client.jks is "serverpublic"), and the value passed from the main indicating the type of message to encrypt is also set (for example, getExamRequest instead of getTranscriptRequest).

From line 70 to line 74, there are the lines specifying the actions that the client-side security interceptor should take upon receiving a response message from the server. Specifically, it indicates that the timestamp contained in the response message is valid for 10 seconds, and the signature needs to be verified and the message needs to be decrypted. Decryption is performed using the client's private key (as the server encrypts the message towards the client with the client's public key), while the signature is verified using the server's public key (and thus the server's certificate, which was used to sign the message with its private key).

6.0 The endpoint class: the bridge between the client and the server

The `@Endpoint` class in a SOAP project with Spring-WS and Java serves as a key component for defining web service endpoints. It allows you to handle incoming SOAP requests and associate them with the appropriate methods within your endpoint class.

The `@SoapAction` annotation plays a crucial role by specifying the SOAP action URI associated with a particular method in your endpoint class. By mapping SOAP requests to methods based on the specified SOAP action, the framework can effectively route incoming SOAP messages to the corresponding methods for processing.

By combining the `@Endpoint` class with the `@SoapAction` annotation, you gain the ability to process incoming SOAP requests and direct them to the appropriate methods within your web service. This approach offers flexibility in handling SOAP requests and generating the appropriate responses for clients.

Specifically, the `@Endpoint` class contains what is referred to as "operations" in the SOAP world. Each operation is associated with a URL that is invoked by the client-side `ExamsClient` class, as seen previously, through the `SoapActionCallback`. Once the message is received by the operation, it is sent to the SOAP server for processing. Interceptors of various types may exist between the SOAP server and the operations. Some interceptors may perform simple tasks like logging requests, while others may be more complex. In this project, a security interceptor is also present on the server-side.

In this class, there is also the mapping of requests and responses from the perspective of the SOAP service.

The class' code is visible below.

```java
@Endpoint
public class ExamsEndpoint {
    public static final String NAMESPACE_URI="http://namespacetest.com";
    private ExamsRepository examsRepo;

    @Autowired
    public ExamsEndpoint (ExamsRepository examsRepo) {
        this.examsRepo = examsRepo;
    }

    @SoapAction(value = "http://namespacetest.com/getExamRequest")
    @ResponsePayload
    public GetExamResponse getASingleExam(@RequestPayload GetExamRequest request) {
        GetExamResponse response = new GetExamResponse();

        response.setResponseResult(examsRepo.findExamRegistration(request.getExamName(), request.getStudentID()));

        return response;
    }

    @SoapAction(value = "http://namespacetest.com/getTranscriptRequest")
    @ResponsePayload
    public GetTranscriptResponse getFullTranscript(@RequestPayload GetTranscriptRequest request) {
        GetTranscriptResponse response = new GetTranscriptResponse();

        response.setResponseResult(examsRepo.findAllexams(request.getStudentID()));

        return response;
    }

    @SoapAction(value = "http://namespacetest.com/setNewExamRequest")
    @ResponsePayload
    public SetNewExamResponse setNewExam(@RequestPayload SetNewExamRequest request) {
        SetNewExamResponse response = new SetNewExamResponse();
        response.setResponseResult(examsRepo.insertNewExam(request.getStudentID(), request.getExamName(), request.getMarkValue()));

        return response;
    }
}

```

7.0 The server side

The endpoint class that exposes the SOAP service operations has been referred to as the bridge between the client and the service. In reality, it is an integral part of the server-side. The server-side also consists of a main class used to start everything and a "WebServiceConfig" class, which is the specification of server-side components as @Bean (similar to what we saw on the client side). The service is hosted on a Tomcat server started through Maven. This is the typical Spring-WS environment. JAXB2 is used to generate all the necessary components (as seen earlier), Maven handles all the required technological components for the Spring/Java project (dependencies defined in the POM file), and upon startup, Spring creates and starts a Tomcat server. In our case, the Tomcat server is listening on port 8080.

```java
@SpringBootApplication
public class SoapProjectWebServiceApplication {

    Run | Debug
    public static void main(String[] args) {
        SpringApplication.run(primarySource:SoapProjectWebServiceApplication.class, args);
    }

}

```

An interesting aspect visible in the server's main class is the @SpringBootApplication annotation, which indicates that a Spring Boot application is being created.

7.1 Security aspects about the server side

As seen for the client side, the server side also handles the security aspect through a security interceptor. Here, too, it is a Wss4j security interceptor. The server receives encrypted messages from the client (encrypted with the server's public key), with a timestamp (valid for 10 seconds) and signed with the user's signature who invoked the operation (the private keys of the users, as explained before, are stored in the "client.jks"). Additionally, to verify the validity of the username/password, the server implements a "private" method that retrieves the hash sent by the client and compares it with the hash of the user stored in the MongoDB database.

The server also takes care of encrypting and signing (this time with its private key) the response messages it sends to the client.

```
96    @Bean
97    public Wss4jSecurityInterceptor securityInterceptor() throws IOException, Exception {
98
99        Wss4jSecurityInterceptor securityInterceptor = new Wss4jSecurityInterceptor();
100
101       // Validate incoming request
102       securityInterceptor.setValidationActions(actions:"UsernameToken Timestamp Signature Encrypt");
103       securityInterceptor.setValidationTimeToLive(validationTimeToLive:10); //10 second time windows to consider the request valid
104       securityInterceptor.setEnableSignatureConfirmation(enableSignatureConfirmation:true);
105       securityInterceptor.setValidationSignatureCrypto(getCryptoFactoryBean().getObject());
106       securityInterceptor.setValidationDecryptionCrypto(getCryptoFactoryBean().getObject());
107       CallbackHandler[] arrayCallbackHandlers = new CallbackHandler[2];
108       arrayCallbackHandlers[0] = securityCallbackHandler();
109       arrayCallbackHandlers[1] = keyStoreCallbackHandler();
110       securityInterceptor.setValidationCallbackHandlers(arrayCallbackHandlers);
111
112       //Encrypt the response
113       securityInterceptor.setSecurementEncryptionUser(securementEncryptionUser:"clientpublic");
114       securityInterceptor.setSecurementEncryptionParts(securementEncryptionParts:"{Content}{http://namespacetest.com}responseResult");
115       securityInterceptor.setSecurementEncryptionCrypto(getCryptoFactoryBean().getObject());
116
117       //Sign the response
118       securityInterceptor.setSecurementActions(securementActions:"Timestamp Signature Encrypt");
119       securityInterceptor.setSecurementUsername(securementUsername:"server");
120       securityInterceptor.setSecurementPassword(securementPassword:"serverkeystore");
121       securityInterceptor.setSecurementSignatureCrypto(getCryptoFactoryBean().getObject());
122
123       securityInterceptor.afterPropertiesSet();
124
125       return securityInterceptor;
126   }
```

The elements visible in the code above are essentially the same as those seen for the client-side security interceptor. One interesting thing is that on the server-side, to perform the verification of the username and password, a handler is invoked using the "securityCallbackHandler". Another interesting point is that since there are two handlers here (the one just mentioned and the keystoreCallbackHandler used to retrieve the server's private key), an array of handlers was created and added to the security interceptor.

In the code snippet shown below, you can see the method (@Override) for adding the security interceptor to the SOAP message chain (similarly done on the client side), the creation of a messageDispatcherServlet responsible for receiving messages on the various exposed operations, and a @Bean used to create and expose the WSDL. In this case, the WSDL is exposed at the URL visible in the code.

```java
    @Override
    public void addInterceptors(List<EndpointInterceptor> interceptors) {
        try {
            interceptors.add(securityInterceptor());

        } catch (IOException e) {
            e.printStackTrace();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

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

    @Bean(name = "exams")
    public DefaultWsdl11Definition defaultWsdl11Definition(XsdSchema examsSchema) {
        DefaultWsdl11Definition wsdl11Definition = new DefaultWsdl11Definition();
        wsdl11Definition.setPortTypeName(portTypeName:"ExamsPort");
        wsdl11Definition.setLocationUri(locationUri:"http://localhost:8080/ws/exams.wsdl");
        wsdl11Definition.setTargetNamespace(ExamsEndpoint.NAMESPACE_URI);
        wsdl11Definition.setSchema(examsSchema);

        return wsdl11Definition;
    }

    @Bean
    public XsdSchema examsSchema() {
        return new SimpleXsdSchema(new ClassPathResource(path:"exams.xsd"));
    }
}
```

7.2 The exposed WSDL, the contract between the server and the client

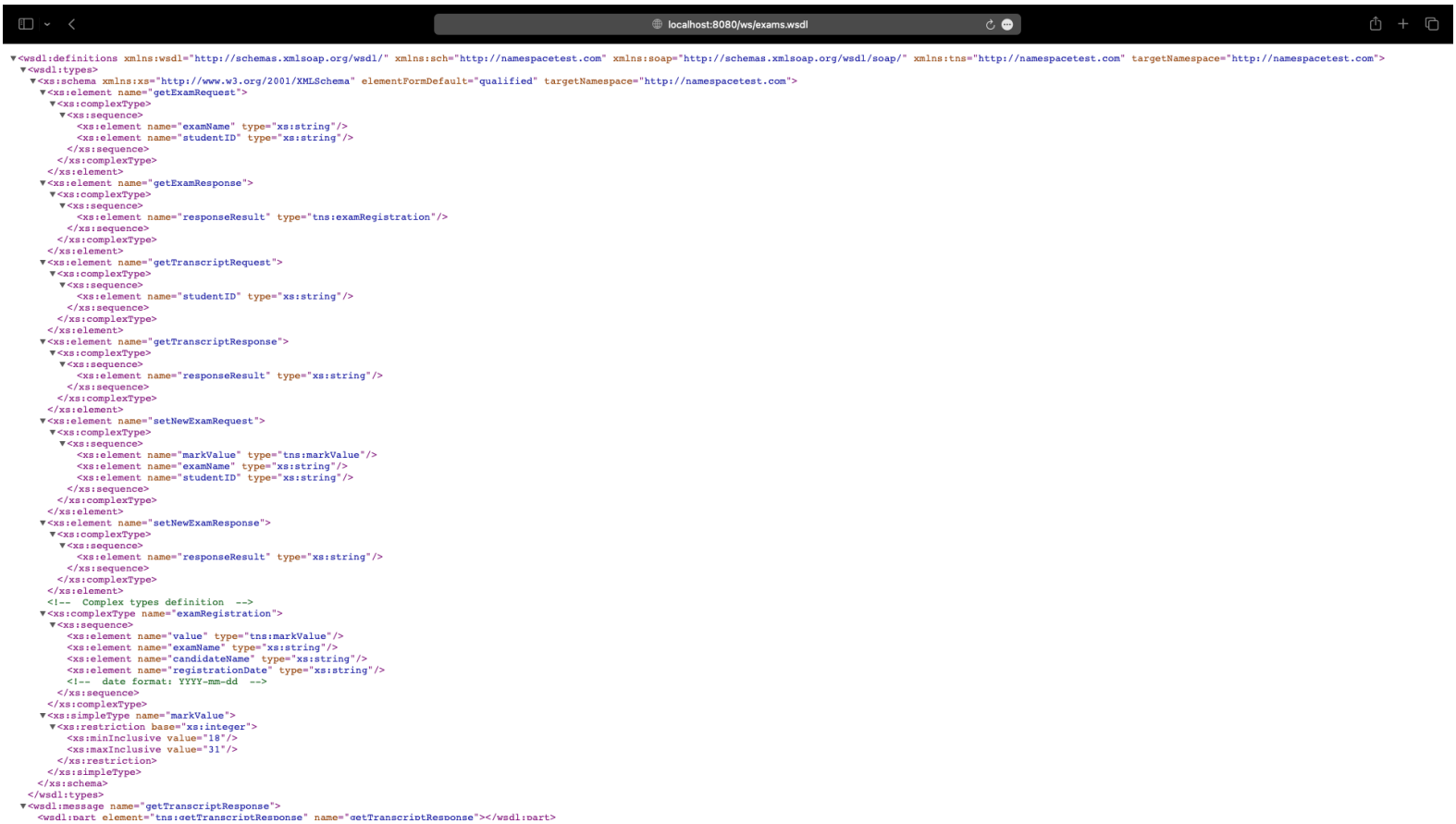
In the world of SOAP, WSDL stands for Web Services Description Language. It is an XML-based language used to describe the functionalities provided by a web service. WSDL serves as a contract between the service provider and the service consumer, defining the available operations, their input and output parameters, message formats, and communication protocols.

Essentially, a WSDL file provides a standardized way of communicating with a web service. It acts as a blueprint that allows clients to understand how to interact with the service, what data to send, and what responses to expect. It describes the structure and behavior of the service, making it easier for developers to integrate and consume the service in their applications.

By examining the WSDL, developers can generate client-side code that corresponds to the web service's operations and data types. This allows them to seamlessly interact with the service, sending requests and processing the responses in a standardized manner.

In summary, a WSDL file is a crucial component in SOAP-based web services as it provides a clear and standardized description of the service's capabilities, enabling seamless integration and interaction between clients and the service.

Below you can see part of the exposed WSDL of this project. You can also see that it is located as the same URL specify in the configuration class of the server discussed before.



7.3 The “ExamsRepository” class. The backed of this project

This class essentially handles the operations that lead to various results. In other words, it contains the functions that actually retrieve data on exams from the database and construct the output for the responses, which are then sent to the client through the operations.

With the @Component annotation Spring scan this class and add this functions as components of the server.

The code below shows a method that establishes a connection with MongoDB and establishes a connection with the "exams" collection. Specifically, it creates a cursor, which is an object that allows iterating over the documents.

The first method, "findExamRegistration," is responsible for searching for a registered exam in the database based on a username (potentially a student ID or something similar) and the name of an exam. It reconstructs a primary key for the search. The primary key is formed by concatenating the subject name with the username, both converted to lowercase. Once the reference exam is identified (using its key), the corresponding element is returned. The object returned by this method is of type “ExamRegistration”, which is the complex type defined in the XSD and therefore in the WSDL.

A similar approach is taken with the "findAllExams" method, which takes a username as input and retrieves all exams associated with that user from the database. The check is performed using a regular expression that searches for the username as a substring of the key. In this case, the returned object is a formatted string that enhances the readability of the exam transcript.

```java
@Component
public class ExamsRepository {
    private Map<String, ExamRegistration> exams = new HashMap<>();

    private MongoCollection<Document> findInDB() {

        MongoClient mongoClient = MongoClients.create(connectionString:"mongodb://localhost:27017"
        MongoDatabase database = mongoClient.getDatabase(databaseName:"ExamsDB");
        MongoCollection<Document> collection = database.getCollection(collectionName:"exams");

        return collection;
    }


    public ExamRegistration findExamRegistration (String examName, String studentID) {
        Assert.notNull(examName, message:"The examName musn't be null");
        Assert.notNull(studentID, message:"The studentID musn't be null");

        String keyBuilding = examName.toLowerCase() + studentID.toLowerCase();
        System.out.println("This is the builded keyBuilding: " + keyBuilding);

        MongoCollection<Document> collection = findInDB();
        Bson regexComparison = regex(fieldName:"_id", ".*" + keyBuilding + ".*");
        FindIterable<Document> cursor = collection.find(regexComparison);
        Document examFromDB = cursor.first();

        ExamRegistration retrievedExam = new ExamRegistration();
        retrievedExam.setCandidateName(examFromDB.get(key:"candidateName").toString());
        retrievedExam.setExamName(examFromDB.get(key:"examName").toString());
        retrievedExam.setRegistrationDate(examFromDB.get(key:"registrationDate").toString());
        retrievedExam.setValue(Integer.parseInt(examFromDB.get(key:"value").toString()));

        return retrievedExam;
    }

```

The same principle applies to the method that allows adding an exam to the database. Here, the interesting part is that the insertion date is automatically calculated in the YYYY-MM-DD format and written to the database. Only professors can add exams, and the check is performed on the client-side using a regex that verifies if the username contains the word "professor" (assuming that professor usernames have a slightly different format in a real-world context).

7.4 The JUnit test class for the “ExamsRepository” class

In order to test the "ExamsRepository" class, which is responsible for the server-side operations, I have written a JUnit test class with assertions.

As visible from the screenshot below, the test code ran successfully and produced the expected results in the assertions.

```java
class SoapProjectWebServiceApplicationTests {

    @Test
    void getExam() {
        ExamsRepository repoTest= new ExamsRepository();

        String test1ExamName = "Programmazione1";
        String test1StudentID = "Utente1";
        String date = "2021-12-12";

        ExamRegistration result = repoTest.findExamRegistration(test1ExamName, test1StudentID);

        System.out.println("Questo è il candidate name: " + result.getCandidateName());
        assertTrue(result.getCandidateName().equals(test1StudentID));
        assertTrue(result.getExamName().equals(test1ExamName));
        assertTrue(result.getValue() == 27);
        assertTrue(result.getRegistrationDate().equals(date));
    }

    @Test
    void getTranscript() {
        ExamsRepository repoTest = new ExamsRepository();
        String test1StudentID = "utente2";

        String result = repoTest.findAllexams(test1StudentID);

        assertTrue(result.contains(s:"Programmazione2, Mark: 28, Registration date:"));
    }

    @Test
    void insertNewExam() {
        ExamsRepository repoTest = new ExamsRepository();

        String result = repoTest.insertNewExam(studentID:"Utente2", newExam:"Architettura1", value:27);

        assertTrue(result.equals(anObject:"Exam registration successfully registered."));

        ExamRegistration result2 = repoTest.findExamRegistration(examName:"architettura1", studentID:"utente2");
        assertTrue(result2.getCandidateName().equals(anObject:"Utente2"));
        assertTrue(result2.getExamName().equals(anObject:"Architettura1"));
        assertTrue(result2.getValue() == 27);
    }
}
```

8.0 WS-Policy

In Spring, declaring WS-Policy can be cumbersome as it requires integrating Apache CXF with Spring. For this reason, I have chosen not to include them explicitly. However, in this chapter, I will describe how some WS-Policy statements could be written.

As a possible policy to consider, I thought it could be useful to require the use of HTTPS as the communication protocol instead of HTTP. While it's true that data is already encrypted using WS-Security in SOAP, having an additional security layer at the application protocol level that encompasses SOAP provides an extra assurance.

```
<wsp:Policy xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
        xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
        xmlns:https="http://schemas.xmlsoap.org/ws/2004/09/policy/http">

  <sp:TransportBinding>
    <wsp:Policy>
      <https:TransportToken/>
```

```
    <sp:AlgorithmSuite>
      <wsp:Policy>
        <sp:Basic256/>
      </wsp:Policy>
    </sp:AlgorithmSuite>
    <sp:IncludeTimestamp/>
  </wsp:Policy>
 </sp:TransportBinding>

</wsp:Policy>
```

<sp:TransportBinding> specifies that the policy applies to the transport layer.

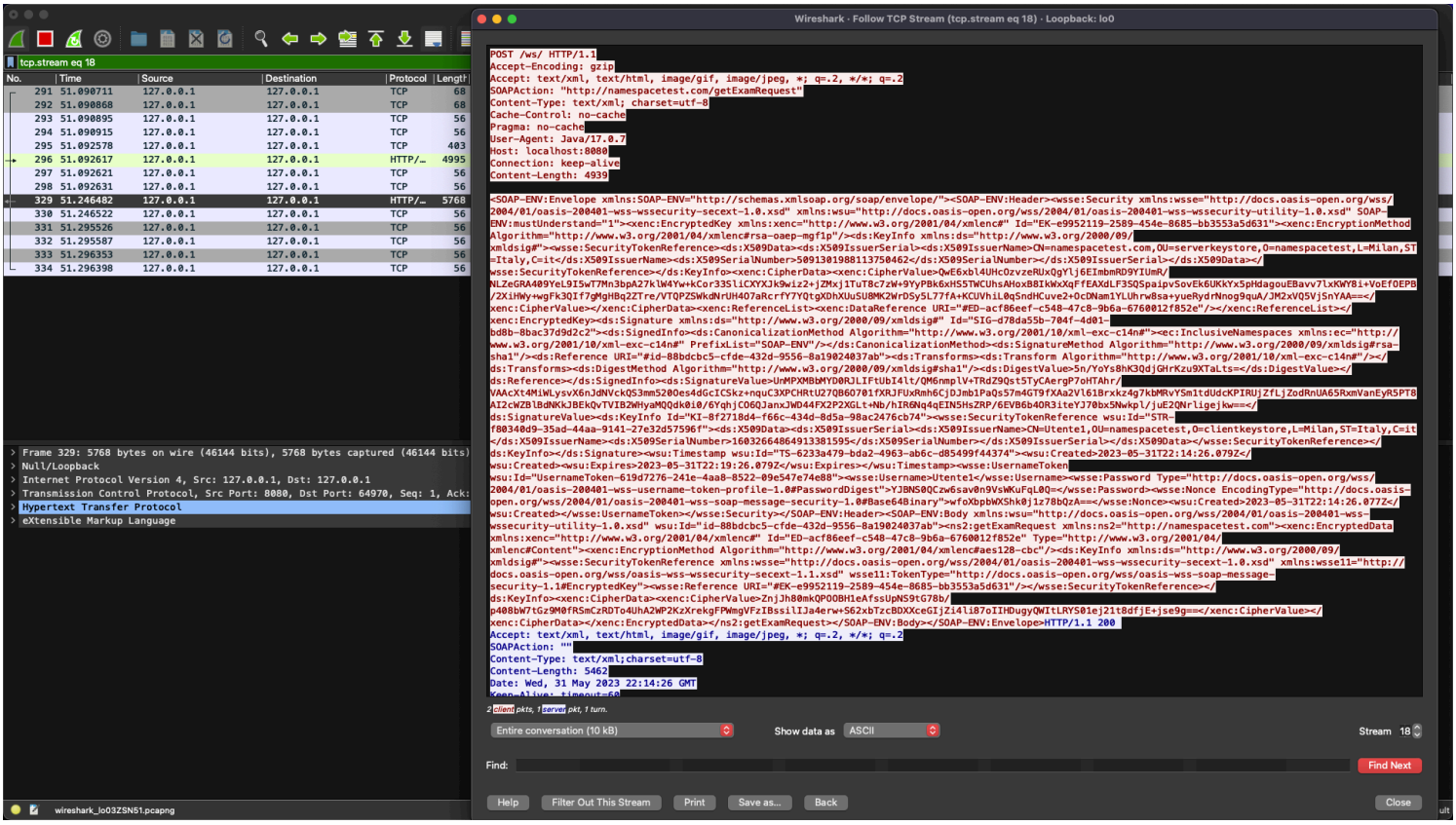
<https:TransportToken/> indicates that a transport-level security token is required, which implies the use of HTTPS.

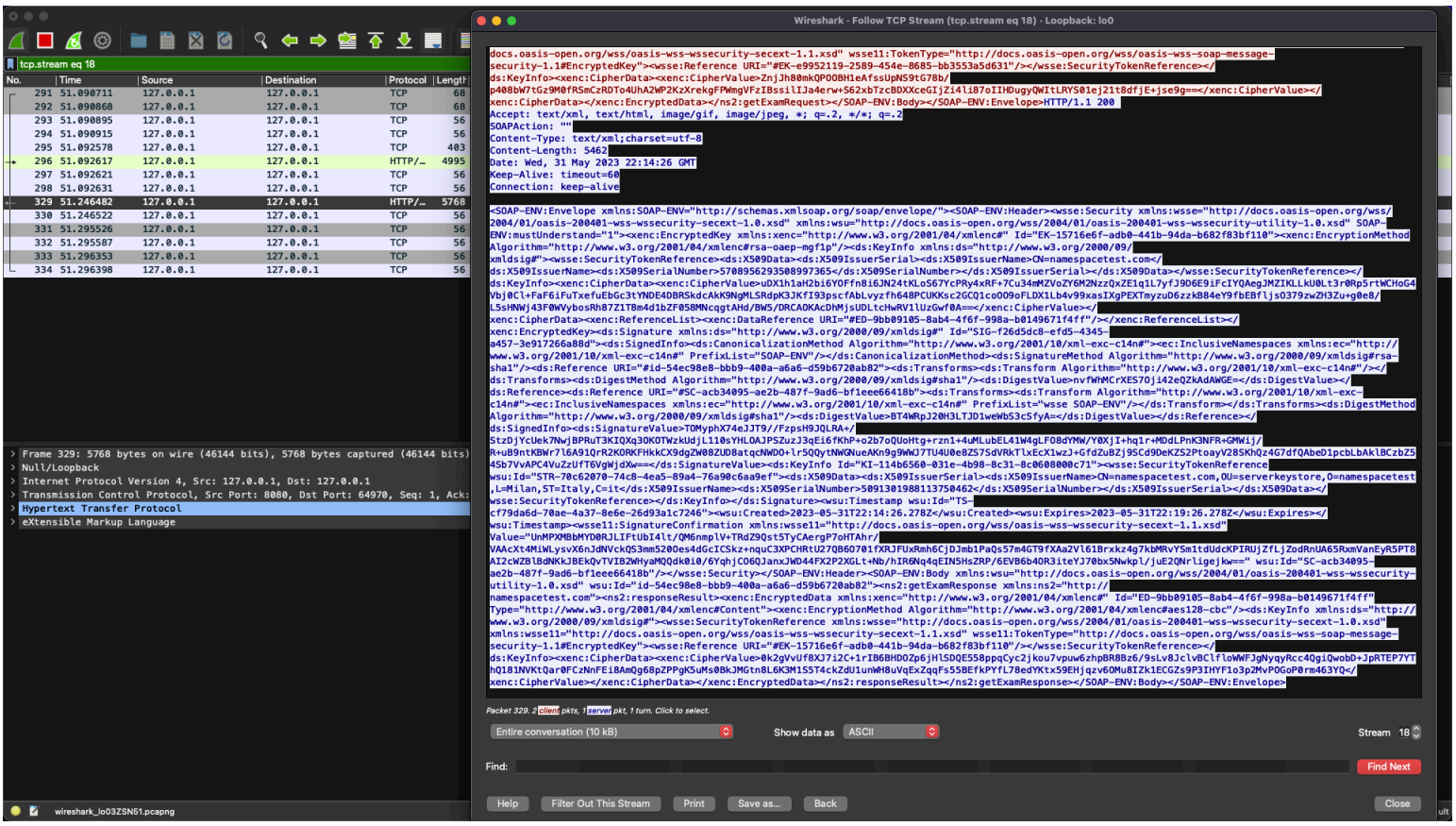
<sp:AlgorithmSuite> specifies the encryption algorithm suite to be used. In this example, it uses the "Basic256" algorithm for secure communication.

<sp:IncludeTimestamp/> requires the inclusion of a timestamp in the messages to ensure temporal validity.

9.0 Traffic sniffing with Wireshark in order to verify the encryption and the message structure

As an additional test to verify the expected message structure and ensure the encryption of request and response contents, I wanted to monitor the exchanged packets between the client and server by sniffing the traffic on the loopback interface (since the communication in this illustrative project takes place entirely on localhost). Below, I will provide some screenshots of the intercepted messages.




tcp.stream eq 18
No. Time Source Destination Protocol Length
291 51.090711 127.0.0.1 127.0.0.1 TCP 68
292 51.090868 127.0.0.1 127.0.0.1 TCP 68
293 51.090895 127.0.0.1 127.0.0.1 TCP 56
294 51.090915 127.0.0.1 127.0.0.1 TCP 56
295 51.092578 127.0.0.1 127.0.0.1 TCP 403
296 51.092617 127.0.0.1 127.0.0.1 HTTP/… 4995
297 51.092621 127.0.0.1 127.0.0.1 TCP 56
298 51.092631 127.0.0.1 127.0.0.1 TCP 56
329 51.246482 127.0.0.1 127.0.0.1 HTTP/… 5768
330 51.246522 127.0.0.1 127.0.0.1 TCP 56
331 51.295526 127.0.0.1 127.0.0.1 TCP 56
332 51.295587 127.0.0.1 127.0.0.1 TCP 56
333 51.296353 127.0.0.1 127.0.0.1 TCP 56
334 51.296398 127.0.0.1 127.0.0.1 TCP 56
Frame 329: 5768 bytes on wire (46144 bits), 5768 bytes captured (46144 bits)
Null/Loopback
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 8080, Dst Port: 64970, Seq: 1, Ack:
Hypertext Transfer Protocol
eXtensible Markup Language
wireshark_lo03ZSN51.pcapng
Wireshark · Follow TCP Stream (tcp.stream eq 18) · Loopback: lo0
Packet 329. 2 client pkts, 1 server pkt, 1 turn. Click to select.
Entire conversation (10 kB)
Show data as ASCII
Stream 18
Find:
Find Next
Help
Filter Out This Stream
Print
Save as...
Back
Close