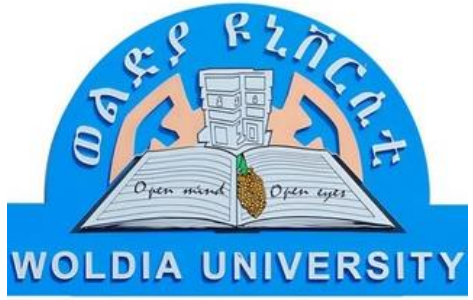

Woldia University
School of Computing

Department of Software Engineering

Course Title: **Web Service,**

Course Code: **SEng5127**

Chapter Three: Implementing Code-First SOAP Web Services

By : Demeke G.

AY:2018 E.C

Outline

- Overview of Java Web Technologies
- Exposing Plain Old Java Objects (POJOs) as Web Services
  - Applying JAX–WS Annotations to POJOs
  - Configuring and Deploying Web Services
- Implementing SOAP Clients in Java
  - Generating Client-Side Artifacts from WSDL
  - Authenticating and Authorizing Clients
- Augmenting SOAP-Based Services
  - Best Practices for SOAP Services
  - Implementing Policies for Security, Reliability, and Optimization

Learning Outcome

- ☛Identify and explain java technologies to implement Web Services
- ☛Explain the concept of the Code-First approach in SOAP web service
- ☛Expose Plain Old Java Objects (POJOs) as SOAP-based web services using JAX-WS annotations.
- ☛Generate client-side artifacts automatically from WSDL using tools like wsimport.
- ☛Develop SOAP clients capable of consuming and interacting with web services.
- ☛Monitor and analyze SOAP messages exchanged between client and server.

Introduction

- In 2017, **Java EE** transitioned to the Eclipse Foundation and was renamed **Jakarta EE**.
- **Java EE** (Java Platform, Enterprise Edition), now known as **Jakarta EE**, is a set of specifications and APIs for building large-scale, distributed, and robust enterprise applications in Java.
- **Jakarta EE** provides a framework for developing web applications, **microservices**, and **enterprise-level applications** with standardized solutions to common challenges like scalability, security, and persistence.
- A web application is an application accessible from the web.
- Servlet technology is used to create web application that resides at server side and generates dynamic web page.

Cont..

- A web application is composed of web components like **Servlet**, **JSP**, etc. and other components such as **HTML**.
- Servlet technology is **robust and scalable** because of java language.
- Java web applications can run on any platform with a JVM
- Java offers frameworks like **Spring**, **Hibernate**, and tools like **Tomcat** and **GlassFish** for efficient development.
- Java provides in-built security features like encryption, authentication, and secure socket layer (SSL) integration.
- Java applications can handle large amounts of traffic and data efficiently
- Servers like Apache Tomcat and GlassFish run Java web applications and manage request routing, application deployment, and scalability.

Cont..

Core Technologies in Java EE (Jakarta EE)

- **Servlets:** Core component for handling HTTP requests and responses and enables the creation of dynamic web applications.
- **JavaServer Pages (JSP)** : server-side technology that simplifies the creation of dynamic, platform-independent web pages by allowing developers to embed Java code directly into HTML.
- **Enterprise JavaBeans (EJB) :**is a server-side component architecture for modular, distributed, and transactional business applications in Java. Manages complex operations like distributed transactions and session management
- **Java Persistence API (JPA):** is a specification in Java for managing relational data in enterprise applications. JPA allows developers to interact with databases using Java objects without requiring extensive SQL code.

Cont..

- **Java Message Service (JMS)**
  - Provides messaging capabilities for asynchronous communication in distributed systems. Common JMS providers: RabbitMQ, Apache Kafka, **WildFly JMS**
  - A Java API for sending, receiving, and processing messages asynchronously.
- **Java API for RESTful Web Services (JAX-RS)**
  - Simplifies the development of RESTful web services.
  - Annotated-based API for creating lightweight web services.
- **Java API for XML Web Services (JAX-WS)**
  - Simplifies the creation of Simple Object Access Protocol (SOAP-based) web services.
- **Contexts and Dependency Injection (CDI)**
  - Manages the lifecycle of objects and their dependencies in enterprise applications. Promotes loose coupling and modularity.
- **JavaServer Faces (JSF)**
  - A framework for building component-based user interfaces for web applications
- **Java Transaction API (JTA)**
  - Manages transactions in enterprise applications, including distributed transactions.

JTA vs Local Transactions?

Servlet

- A technology used to create web application
- Servlets process client requests, execute business logic, and generate dynamic responses.
- An API that provides many interfaces and classes.
- Servlet is an interface that must be implemented for creating any servlet.
- Servlet is a class that extend the capabilities of the servers and respond to the incoming request.
- It can respond to any type of requests.
- Servlet is a web component that is deployed on the server to create dynamic web page.

Advantages of Servlets

- **Efficient**: Uses threads instead of creating new processes for each request.
- **Robust:** Servlets are managed by JVM so no need to worry about memory leak, garbage collection etc.
- **Scalability**: Can handle multiple client requests simultaneously.
- **Platform-Independent**: Runs on any server that supports the Java Servlet API (e.g., Tomcat, GlassFish).
- **Portability:** Can run on any servlet container that follows the Java Servlet specification.
- **Extensibility**: Can be integrated with other Java technologies like JSP, Hibernate, and Spring.
- **Secure:** Built on the Java platform, offering security features like encryption and authentication.

Servlets Tasks

- **Read the explicit data sent by the clients.**

  This includes an HTML form on a Web page or it could also come from an applet or a custom HTTP client program.

- **Read the implicit HTTP request data sent by the clients** . Like **cookies**, **media** types and compression schemes the browser understands

- **Process the data and generate the results**. Like talking to a database, invoking a Web service, or computing the response directly.

- **Send the explicit data to the clients.** This document can be sent in a variety of formats, including text (HTML or XML), binary (GIF images), Excel, etc.

- **Send the implicit HTTP response to the clients** This includes telling the browsers or other clients what type of document is being returned (e.g., HTML), setting cookies and caching parameters, and other such tasks.

- **Creating RESTful APIs.**

- **Managing sessions and user authentication.**

Http Request Methods

- Every request has a header that tells the status of the client.
- There are many request methods. **Get** and **Post** requests are mostly used.

HTTP Request	Description
GET	Asks to get the resource at the requested URL.
POST	Asks the server to accept the body info attached. It is like GET request with Extra info sent with the request.

The difference between Get and Post

GET	POST
only limited amount of data can be sent because data is sent in header.	large amount of data can be sent because data is sent in body.
Get request is not secured because data is exposed in URL bar.	Post request is secured because data is not exposed in URL bar.
Get request can be bookmarked	Post request cannot be bookmarked
Get request is idempotent. It means second request will be ignored until response of first request is delivered.	Post request is non-idempotent
Get request is more efficient and used more than Post	Post request is less efficient and used less than get.
Only ASCII characters allowed	No restrictions. Binary data is also allowed

Hibernate

- Hibernate ORM is a powerful, open-source Object-Relational Mapping (ORM) framework for Java.
- It simplifies the development of Java applications by providing an abstraction layer over the database,
- Allowing developers to interact with databases using Java objects rather than SQL queries.
- Hibernate implements the Java Persistence API (JPA) specification, making it a widely-used choice for data persistence in Java-based applications
- Supports multiple database dialects and simplifies database migration.
- Integrates with JTA, JDBC, or container-managed transactions.

Hibernate Annotations

@Entity: Marks a class as a persistent entity.

@Table: Specifies the table name.

@Id: Indicates the primary key.

@GeneratedValue: Specifies the strategy for primary key generation.

@Column: Maps a field to a database column.

@OneToOne, @OneToMany, @ManyToOne, @ManyToMany: Defines relationships between entities.

@Transient: Ignores a field for persistence

Hibernate Example

```
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Column;
@Entity
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(name = "name")
    private String name;
    @Column(name = "department")
    private String department;
    // Getters and Setters
}
```

Hibernate configuration hibernate.cfg.xml

```
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.driver_class">org.h2.Driver</property>
        <property name="hibernate.connection.url">jdbc:h2:mem:testdb</property>
        <property name="hibernate.connection.username">sa</property>
        <property name="hibernate.connection.password"></property>
        <property name="hibernate.dialect">org.hibernate.dialect.H2Dialect</property>
        <property name="hibernate.hbm2ddl.auto">update</property>
        <mapping class="com.example.Employee"/>
    </session-factory>
</hibernate-configuration>
```

Hibernate Example

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
public class MainApp {
   public static void main(String[] args) {
      SessionFactory factory = new  Configuration().configure ("hibernate.cfg.xml")
            .buildSessionFactory();
      Session session = factory.openSession();
      try {
         session.beginTransaction();
         Employee emp = new Employee();
         emp.setName("John Doe");
         emp.setDepartment("HR");
         session.save(emp); // Persist the entity
         session.getTransaction().commit();
         System.out.println("Employee saved!");
      } finally {
         session.close();
         factory.close();
      }   }}
```

Spring Boot Framework

- A powerful, open-source framework for building enterprise-level Java applications.
- It provides a comprehensive programming and configuration model, focusing on modern design patterns and simplifying Java development through dependency injection, aspect-oriented programming, and more.
- Spring is widely used in Java-based projects, offering solutions for various layers, including web, data access, messaging, and enterprise services.
- It creates stand-alone Spring applications that can be started using Java -jar.
- The main goal of Spring Boot is to reduce development, unit test, and integration test time.

Advantages of Using Spring Boot

- Reduces development time and increases productivity.
- Eliminates the need for manual configuration.
- Provides an opinionated default setup.
- Easy to test applications.
- Suitable for microservices architecture.
- Community support and extensive documentation.
- Simplifies dependency management (e.g., spring-boot-starter-web for web applications).

Spring Boot Architecture

- **Presentation Layer**: Handles user interfaces and requests

  (e.g., controllers).

- **Business Layer:** Handles computations, validations, and decision-making processes. i.e. **Services (@Service)**

- **Persistence Layer:** Manages interactions with the database and data repositories.
  - Repositories **(@Repository):**Provides CRUD operations using JPA, Hibernate, or other frameworks.
  - Maps database tables to Java objects.

- **Integration Layer:** Handles communication with external systems, APIs, or services
  - **RestTemplate**/WebClient: For consuming external **REST APIs.**
  - **Message Brokers**: For asynchronous communication (e.g., **RabbitMQ**, **Kafka**).

Spring Boot Starters

- Pre-defined dependency descriptors to simplify Maven/Gradle configurations.
- Examples:
  - **spring-boot-starter-web:** For building web applications (REST APIs).
  - **spring-boot-starter-data-jpa:** For database integration using JPA.
  - **spring-boot-starter-security:** For adding security features.
  - **spring-boot-starter-actuator:** spring-boot-starter-actuator
- Steps to setup the project.
  - Navigate to https://start.spring.io.
  - Choose either Gradle or Maven and the language you want to use
  - Click Dependencies and select **Spring Web**.
  - Click Generate.

Spring Boot Annotations

❑ Core Annotations

- **@SpringBootApplication**:Entry point for Spring Boot applications. Combines **@Configuration**, **@EnableAutoConfiguration**, and **@ComponentScan**.
- **@Configuration** Indicates that the class contains Spring configuration.
- **@Bean:**Marks a method as a bean producer, which Spring will manage in the application context.
  - A **bean** in Spring is simply an object that is **instantiated, assembled, and managed** by the **Spring IoC (Inversion of Control) container**.
  - A method-level annotation.

Web Layer Annotations

- **@RestController:Combines @Controller and @ResponseBody.** Used to create RESTful web services.
- **@Controller**:Marks a class as a Spring MVC controller.
- **@GetMapping, @PostMapping , @PutMapping, @DeleteMapping** : :Maps HTTP GET, POST, PUT and DELETE requests to a specific handler method.
- **@RequestMapping:**General-purpose mapping annotation for requests.
- **@RequestParam:**Binds query parameters or form data to method parameters.
- **@PathVariable:**Binds URI template variables to method parameters.
- **@RequestBody**: Binds the body of a request to a method parameter (e.g., JSON to a Java object).
- **@ResponseBody:**Indicates the return value of a method should be serialized and written directly to the response.
- **@CrossOrigin**: Enables cross-origin resource sharing (CORS).

Data Access Layer Annotations

@Entity: Marks a class as a JPA entity (maps to a database table).

@Table: Specifies the database table name for an entity.

@Id: Identifies the primary key of an entity.

@GeneratedValue: Specifies how the primary key is generated (e.g., AUTO, IDENTITY).

@Column: Configures a column in a database table.

@Repository: Marks a class as a repository for data access.

@Query: Used to define custom JPQL or SQL queries in a repository.

Dependency Injection (DI) Annotations

- **@Autowired**: Automatically injects dependencies by type.
- **@Qualifier:** Used with @Autowired to specify a particular bean when multiple beans of the same type exist.
- **@Primary:** Specifies a default bean when multiple candidates are available.
- **@Componen:** Marks a class as a Spring-managed component.
- **@Service:** A specialization of @Component for service-layer classes.
- **@Repository:** A specialization of @Component for data access classes.

Validation Annotations

- **@Valid:** Triggers validation for the annotated object.
- **@NotNull:** Ensures the annotated field is not null.
- **@Size:**Specifies size constraints for a string, collection, map, or array.
- **@Min:**Specifies the minimum value for a numeric field.
- **@Max:**Specifies the maximum value for a numeric field.

Security Annotations

- **@EnableWebSecurity:**Enables Spring Security's web security support.
- **@PreAuthorize:**Allows method-level security using expressions.
- **@Secured**:Specifies roles required to execute a method.

Example : Spring Boot Entity

```
@Entity
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
     public Long getId() {        return id;    }
    public void setId(Long id) {    this.id = id;    }
    public String getFirstName() {  return firstName;    }
    public void setFirstName(String firstName) {  this.firstName = firstName;    }
    public String getLastName() {  return lastName;    }
    public void setLastName(String lastName) {    this.lastName = lastName;    }
    public String getEmail() {  return email;    }
    public void setEmail(String email) {    this.email = email;    }
}
```

Example : Repository

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {
}
```

Methods Inherited from JpaRepository

```
<S extends T> S save(S entity);
Optional<T> findById(ID id);
boolean existsById(ID id);
Iterable<T> findAll();
Iterable<T> findAllById(Iterable<ID> ids);
long count();
void deleteById(ID id);
void delete(T entity);
void deleteAll(Iterable<? extends T> entities);
void deleteAll();
```

```
List<T> findAll(); // override returns List
List<T> findAll(Sort sort);
List<T> findAllById(Iterable<ID> ids);
<S extends T> List<S> saveAll(Iterable<S> entities);
void deleteInBatch(Iterable<T> entities);
void deleteAllInBatch();
T getOne(ID id);     // Lazy reference
<S extends T> List<S> findAll(Example<S> example);
<S extends T> List<S> findAll(Example<S> example,
Sort sort);
List<Customer> findByEmail(String email);
List<Customer> findByActiveTrueOrderByNameAsc();
```

Example : Service

```
@Service
public class CustomerService {
    private final CustomerRepository customerRepository;
    public CustomerService(CustomerRepository customerRepository) {
        this.customerRepository = customerRepository;
    }

    public List<Customer> getAllCustomers() {
        return customerRepository.findAll();
    }
    public Optional<Customer> getCustomerById(Long id) {
        return customerRepository.findById(id);
    }
}
```

Example : Service cont..

```
public Customer addCustomer(Customer customer) {
    return customerRepository.save(customer);
}
public Customer updateCustomer(Long id, Customer updatedCustomer) {
    return customerRepository.findById(id)
            .map(customer -> {
                customer.setFirstName(updatedCustomer.getFirstName());
                customer.setLastName(updatedCustomer.getLastName());
                customer.setEmail(updatedCustomer.getEmail());
                return customerRepository.save(customer);
            })
            .orElseThrow(() -> new RuntimeException("Customer not found"));
}

public void deleteCustomer(Long id) {
    customerRepository.deleteById(id);
}
```

Example : Controller

```
@RestController
@RequestMapping("/api/customers")
public class CustomerController {
    private final CustomerService customerService;
    public CustomerController(CustomerService customerService) {
        this.customerService = customerService;
    }
    @GetMapping
    public List<Customer> getAllCustomers() {
        return customerService.getAllCustomers();
    }
 @GetMapping("/{id}")
public ResponseEntity<Customer> getCustomerById(@PathVariable Long id) {
        return customerService.getCustomerById(id)
                .map(ResponseEntity::ok)
                .orElse(ResponseEntity.notFound().build());
```

Example : Controller cont…

```
@PostMapping
    public Customer addCustomer(@RequestBody Customer customer) {
        return customerService.addCustomer(customer);
    }
    @PutMapping("/{id}")
    public ResponseEntity<Customer> updateCustomer(@PathVariable Long id, @RequestBody Customer
updatedCustomer) {
        try {
            return ResponseEntity.ok(customerService.updateCustomer(id, updatedCustomer));
        } catch (RuntimeException e) {      return ResponseEntity.notFound().build();
        }    }
    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteCustomer(@PathVariable Long id) {
        customerService.deleteCustomer(id);
        return ResponseEntity.noContent().build();
    }}
```

Activity:1

You are given an Entity named Student with the following attribute . ID, FullName, sex, department,

- implement **4 layers**:
  - Entity, Repository, Service + Service Implementation, Controller

Configure application.properties

❖ Set the MySQL database connection details in your **application.properties as bellow**

```
# Datasource Configuration
spring.datasource.url=jdbc:mysql://localhost:3306/ur_database_name
spring.datasource.username=your_username
spring.datasource.password=your_password
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

# JPA Configuration
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
```

Running the Application

- Run the Spring Boot application.
- Test the endpoints using **Postman, Swagger**, or any API testing tool:
- use the **Springdoc OpenAPI** library, which is a popular choice for integrating Swagger UI with Spring Boot
- Steps to configure swagger:
  - **Add Maven Dependency**

```
<dependency>
        <groupId>org.springdoc</groupId>
        <artifactId>springdoc-openapi-ui</artifactId>
         <version>1.7.0</version>
</dependency>
```

  - Access Swagger UI http://localhost:8080/swagger-ui.html
    - GET /api/customers - List all customers.
    - GET /api/customers/{id} - Get a specific customer by ID.
    - POST /api/customers - Add a new customer.
    - PUT /api/customers/{id} - Update an existing customer.
    - DELETE /api/customers/{id} - Delete a customer.

Exposing Plain Old Java Objects (POJOs) as Web Services

- Decorating a standard Java class (POJO) with JAX-WS annotations to define it as a web service endpoint
- The primary annotation is **@WebService** at the class level, and **@WebMethod** for methods intended to be exposed as web service operations.
- Other annotations like @WebParam and **@WebResult** can be used to customize parameter and return value mapping. i.e.

```
import jakarta.jws.WebService;
import jakarta.jws.WebMethod;
@WebService
public class MyService {
@WebMethod
public String sayHello(String name) {
return "Hello, " + name + "!";
}
}
```

Cont..

- **@WebService:** marks a Java class as a web service endpoint interface (SEI) or a service implementation bean (SIB). This is the only required annotation for a basic JAX-WS service.
- **@WebMethod:** Used on a method to expose it as an operation in the generated WSDL. By default, all public methods are exposed, but this annotation provides explicit control.
- **@SOAPBinding:** Used to specify the style, use, and parameter style of the SOAP messages (e.g., Style.RPC or Style.DOCUMENT).
- **@XmlElement / @XmlRootElement:** These annotations help control the mapping of Java objects and their properties to XML in the SOAP messages

Configuring and Deploying Web Services

- After annotating the POJO, the web service needs to be configured and deployed to a web server or application server (e.g., Apache Tomcat, JBoss, WebSphere).
- Configuring and deploying web services involves several key stages to ensure functionality, security, and availability.
- Configuring and deploying web services involves packaging your application code and settings, preparing a server environment, and executing the deployment process to make the service accessible online.
- Configuring and deploying process is highly dependent on the specific technology stack and hosting platform being used.
- Deploying web services involves several essential stages:
  - **Packaging**
    Applications are packaged into deployable archives (e.g., WAR/EJB-JAR) containing code, libraries, and deployment descriptors.

Configuring and Deploying Web Services

- **Server Environment Setup :**The hosting server (Tomcat, WebLogic, IIS, etc.) must be properly configured for the deployment.
- **Security Configuration:** Set up authentication, authorization, and SSL/TLS certificates to protect data and control access.
- **Resource Configuration:** Set up required external resources such as databases, connection pools, and file system permissions.
- **Environment Variables:** Securely configure system-specific settings (e.g., database credentials, API keys) outside the application code.
- **Deployment Execution:** Install the packaged service manually or using automated tools/CI-CD pipelines.
- **Testing & Monitoring:** Verify correct operation, endpoint availability, and continuously monitor performance and errors.

Deployment Strategies & Best Practices

To deploy successfully and avoid problems

- Automate as much as possible (build → test → deploy) to reduce human error.
- Use version control (e.g. Git) so that changes are tracked and revertible
- Thorough testing (unit, integration, user acceptance) before release
- Monitoring and logging for production to catch bugs or performance issues quickly.
- Use environment-specific configuration outside the code (e.g. environment variables).
- Have a rollback plan: be ready to revert to a previous stable version if the release fails.
- If possible, deploy during low-traffic hours to minimize user disruption.

Implementing SOAP Clients in Java

- JAX-WS simplifies client development by generating client-side artifacts from the WSDL
- Use the wsimport tool to generate Java classes from the WSDL.
- Steps
- Create a new Spring Boot project
- Use this Jakarta-compatible(Your version) Maven dependencies:

```
<dependencies>
    <dependency>
        <groupId>com.sun.xml.ws</groupId>
        <artifactId>jaxws-rt</artifactId>
        <version>4.0.2</version>
    </dependency>
    <dependency>
        <groupId>jakarta.xml.bind</groupId>
        <artifactId>jakarta.xml.bind-api</artifactId>
        <version>4.0.0</version>
    </dependency>
</dependencies>
```

```
<dependency
<groupId>org.glassfish.jaxb</groupId>
        <artifactId>jaxb-runtime</artifactId>
        <version>4.0.3</version>
    </dependency>
```

Generating Client-Side Artifacts from WSDL

☛ Add **JAX-WS plugin** to generate client stubs from WSDLpom.xml (add inside <plugins>)

```
<plugin>
   <groupId>com.sun.xml.ws</groupId>
  <artifactId>jaxws-maven-plugin</artifactId>
     <version>4.0.2</version>
      <executions>
       <execution>
           <goals>
           <goal>wsimport</goal>
           </goals>
      ……..
```

```
<configuration>
<wsdlUrls>
<wsdlUrl>http://localhost:8081/ws/calculator.wsdl</wsdlUrl>
</wsdlUrls>

<packageName>com.example.soapclient</packageName>

 <extension>true</extension>

 </configuration>

 </execution>

  </executions>
</plugin>
```

Generate Stubs

- Run the command mvn clean install
- After running, generated classes will appear at :target/generated-sources/wsimport
- Configure the SOAP client bean (Spring Boot way)

```
@Configuration
public class SoapClientConfig {
 @Bean
public CalculatorSoap calculatorSoapClient() throws Exception {
  URL wsdlUrl = new URL("http://localhost:8081/ws/calculator.wsdl");
  Calculator service = new Calculator(wsdlUrl);
 return service.getCalculatorSoap();
 }}
```

Use the client in a service

```
@Service
public record CalculatorService(CalculatorSoap calculatorSoap) {
  public int add(int a, int b) {     return calculatorSoap.add(a, b);    }
  public int subtract(int a, int b) {    return calculatorSoap.subtract(a, b);    }
  public int multiply(int a, int b) {   return calculatorSoap.multiply(a, b);    }
   public int divide(int a, int b) {     return calculatorSoap.divide(a, b);    }
}
```

Use the client in a REST controller..

```
@RestController
@RequestMapping("/api/calc")
 public class CalcController {
private final CalculatorService calculatorService;
public CalcController(CalculatorService calculatorService){
this.calculatorService = calculatorService;
  }
@GetMapping("/add")
public int add(@RequestParam int a, @RequestParam int b) {
return calculatorService.add(a, b) }
@GetMapping("/multiply")
 public int multiply(@RequestParam int a, @RequestParam int b) {
 return calculatorService.multiply(a, b);    }
}
```

Authenticating and Authorizing Clients

Authenticating

- Authentication is the act of validating a user, device, or application's claim of identity. It ensures that only legitimate users can access a system
- Common authentication methods include:
  - **Passwords/Credentials**: The most common method, where a user provides a secret string of characters.
  - **Multi-Factor Authentication (MFA):** Requires the user to provide two or more verification factors, such as a password and a one-time PIN sent to their mobile device, adding an extra layer of security.
  - **Biometrics:** Uses unique biological traits like fingerprints or retina scans for identity verification.
  - **Certificates:** Relies on digital certificates issued by a trusted authority to verify the client's identity, a method often used in client-server interactions.
  - **Tokens**: Uses physical or digital tokens (like JSON Web Tokens or JWTs) to grant access for a limited time or number of sessions after initial login.

Authorization

- After a client's identity has been verified through authentication, authorization determines their access rights and permissions within the system.
- Common authorization models and mechanisms include:
  - **Role-Based Access Control (RBAC):** Permissions are assigned based on a user's organizational role (e.g., administrator, editor, guest). A doctor might have access to all patient records, while a receptionist is limited to basic contact information.
  - **Attribute-Based Access Control (ABAC):** Grants permissions based on a set of attributes related to the user, the resource, and the environment, allowing for more granular control than RBAC.
  - **Access Control Lists (ACLs):** These lists define rules that specify which users or system processes are granted or denied access to specific resources or data.
  - **OAuth 2.0:** An open standard framework widely used for authorization, allowing third-party applications to access resources on behalf of a user without needing their full credentials. It provides limited access via tokens

Augmenting SOAP-Based Services

- It means improving, extending, modernizing, or adapting existing SOAP web services without modifying or replacing the original SOAP implementation.
- SOAP services are often legacy, running in large enterprises—banks, hospitals, governments. They still work, but are old, rigid, and hard to change.
- Augmentation lets you keep legacy SOAP systems running, while adding modern features around them.
- Replacing SOAP services is often not possible due to Expensive, Requires specialized skills, External integration

Areas Where SOAP Services Are Commonly Augmented

☛We can augment SOAP services in seven major ways:

1) Add Modern Interfaces (REST / JSON / GraphQL)

- **SOAP uses:** XML, WSDL, WS-* protocols
- **Modern applications prefer:** REST, JSON, JWT authentication, Microservices, Mobile-friendly APIs

2) Add Authentication & Authorization (OAuth2 / JWT)

- **Legacy SOAP may use:** Basic auth ,WS-Security Username, Token MTOM/PKI,Certificates
- **Modern systems use:**Oauth2, JWT, OpenID Connect

3) Add Logging, Monitoring, and Tracing

- Legacy SOAP systems often lack performance metrics, logging and distributed tracing

4) Add Caching Layer

- SOAP calls are often slow, due to heavy XML serialization.

5) Add Orchestration Layer (Combine Multiple SOAP Services)

- data is split across multiple SOAP services: Augmentation lets us create a single aggregated API that calls all