**Hands-On Lab: SOAP and REST Web Services in Java**

**1. Exploring SoapUI Tool Basic Features**

* **Objective:** Familiarize with SoapUI for testing SOAP and REST services.
* **Steps:**
  + **Installation:**
    - Download and install SoapUI.
  + **Create a SOAP Project:**
    - Import WSDL URL: http://www.dneonline.com/calculator.asmx?WSDL.
    - Test the Add operation by sending a request with parameters.
  + **Create a REST Project:**
    - Import base URL: https://jsonplaceholder.typicode.com.
    - Test GET request to /posts/1.

**6. WSDL - Web Services Development Using SOAP**

* **Objective:** Learn how WSDL is used to define and consume SOAP web services.
* **Steps:**
  + **WSDL Exploration:** View WSDL structure using SoapUI.
  + **Generate Client Code:** Use wsimport to generate Java client code:

CMD

wsimport -keep -s src http://www.dneonline.com/calculator.asmx?WSDL

* + **Java Client Code:**

java

import com.dneonline.Calculator;

import com.dneonline.CalculatorSoap;

public class SoapClient {

public static void main(String[] args) {

Calculator service = new Calculator();

CalculatorSoap port = service.getCalculatorSoap();

int result = port.add(10, 20);

System.out.println("Result: " + result);

}

}

**7. JAX-WS Implementation for SOAP**

* **Objective:** Implement a SOAP web service using JAX-WS.
* **Steps:**
  + **Add Dependencies:** Include JAX-WS API in your Maven pom.xml:

xml

<dependency>

<groupId>javax.xml.ws</groupId>

<artifactId>jaxws-api</artifactId>

<version>2.3.1</version>

</dependency>

* + **Define the Web Service Interface:**

java

import javax.jws.WebMethod;

import javax.jws.WebService;

@WebService

public interface CalculatorService {

@WebMethod

int add(int a, int b);

}

* + **Implement the Web Service:**

java

import javax.jws.WebService;

@WebService(endpointInterface = "CalculatorService")

public class CalculatorServiceImpl implements CalculatorService {

public int add(int a, int b) {

return a + b;

}

}

* + **Publish the Service:**

java

import javax.xml.ws.Endpoint;

public class SoapServer {

public static void main(String[] args) {

Endpoint.publish("http://localhost:8080/calculator", new CalculatorServiceImpl());

System.out.println("Service running at http://localhost:8080/calculator");

}

}

* + **Test the Service:** Use SoapUI to test the service at http://localhost:8080/calculator?wsdl.

**8. Introduction to REST API Features and Principles**

* **Objective:** Understand REST principles and features.
* **Steps:**
  + **REST Basics:** Discuss key principles like statelessness and resource representation.
  + **Create a REST API with Spring Boot:**
    - **Dependencies:**

xml

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

* + - **Implement a REST Controller:**

java

import org.springframework.web.bind.annotation.\*;

@RestController

@RequestMapping("/api")

public class PostController {

@GetMapping("/posts/{id}")

public Post getPost(@PathVariable int id) {

return new Post(id, "Title " + id, "Content for post " + id);

}

@PostMapping("/posts")

public Post createPost(@RequestBody Post post) {

post.setId(123);

return post;

}

@PutMapping("/posts/{id}")

public Post updatePost(@PathVariable int id, @RequestBody Post post) {

post.setId(id);

return post;

}

@DeleteMapping("/posts/{id}")

public void deletePost(@PathVariable int id) {

// Implement deletion logic

}

}

class Post {

private int id;

private String title;

private String content;

// Constructors, getters, and setters

public Post() {}

public Post(int id, String title, String content) {

this.id = id;

this.title = title;

this.content = content;

}

public int getId() { return id; }

public void setId(int id) { this.id = id; }

public String getTitle() { return title; }

public void setTitle(String title) { this.title = title; }

public String getContent() { return content; }

public void setContent(String content) { this.content = content; }

}

* + - **Test the API:** Run the Spring Boot application and use Postman to test the endpoints:
      * GET: http://localhost:8080/api/posts/1
      * POST: http://localhost:8080/api/posts with JSON body:

json

{

"title": "New Post",

"content": "Content for the new post"

}

* + - * PUT: http://localhost:8080/api/posts/1 with JSON body:

json

{

"title": "Updated Post",

"content": "Updated content"

}

* + - * DELETE: http://localhost:8080/api/posts/1

**9. SOAP vs REST**

* **Objective:** Compare SOAP and REST web services.
* **Steps:**
  + **Comparison:** Implement a simple SOAP service and a REST API that perform similar functions.
  + **Test and Compare:** Use the SOAP service at http://www.dneonline.com/calculator.asmx and the REST API created in Spring Boot for comparison.

**10. REST API Methods and HTTP Status Codes**

* **Objective:** Learn about different HTTP methods and status codes used in REST APIs.
* **Steps:**
  + **Methods:** Review GET, POST, PUT, DELETE, and PATCH methods.
  + **Status Codes:**
    - Implement and test HTTP status codes in your REST API:
      * **200 OK:** Successful GET or PUT request.
      * **201 Created:** Successful POST request.
      * **204 No Content:** Successful DELETE request.
      * **404 Not Found:** Resource not found.
      * **400 Bad Request:** Invalid input.

**11. Truly RESTful API, POSTMAN as an API Testing Tool**

* **Objective:** Develop a RESTful API that adheres to REST principles and test it using Postman.
* **Steps:**
  + **Enhance REST API:** Implement HATEOAS links in the REST API.

java

import org.springframework.hateoas.EntityModel;

import org.springframework.hateoas.Link;

import org.springframework.hateoas.server.mvc.WebMvcLinkBuilder;

@RestController

@RequestMapping("/api")

public class PostController {

@GetMapping("/posts/{id}")

public EntityModel<Post> getPost(@PathVariable int id) {

Post post = new Post(id, "Title " + id, "Content for post " + id);

EntityModel<Post> resource = EntityModel.of(post);

Link selfLink = WebMvcLinkBuilder.linkTo(WebMvcLinkBuilder.methodOn(PostController.class).getPost(id)).withSelfRel();

resource.add(selfLink);

return resource;

}

// Other methods remain the same

}

* + **Test with Postman:** Ensure that HATEOAS links appear in the API responses.

**12. JAX-RS Servlet Containers and JAX-RS Annotations**

* **Objective:** Learn about JAX-RS for RESTful web services and its annotations.
* **Steps:**
  + **Add Dependencies:**

xml

<dependency>

<groupId>javax.ws.rs</groupId>

<artifactId>javax.ws.rs-api</artifactId>

<version>2.1.1</version>

</dependency>

* + **Implement JAX-RS Application:**

java

import javax.ws.rs.ApplicationPath;

import javax.ws.rs.core.Application;

import java.util.HashSet;

import java.util.Set;

@ApplicationPath("/api")

public class RestApplication extends Application {

@Override

public Set<Class<?>> getClasses() {

Set<Class<?>> classes = new HashSet<>();

classes.add(HelloWorldResource.class);

return classes;

}

}

* + **Create a JAX-RS Resource:**

java

import javax.ws.rs.GET;

import javax.ws.rs.Path;

import javax.ws.rs.Produces;

import javax.ws.rs.core.MediaType;

@Path("/hello")

public class HelloWorldResource {

@GET

@Produces(MediaType.TEXT\_PLAIN)

public String sayHello() {

return "Hello, World!";

}

}

* + **Deploy and Test:** Deploy on a servlet container like Apache Tomcat and test the endpoint using Postman: http://localhost:8080/your-app/api/hello.

**13. JAX-RS Media Types**

* **Objective:** Handle different media types with JAX-RS.
* **Steps:**
  + **Handling Media Types:**
    - Modify your JAX-RS resource to support JSON and XML:

java

import javax.ws.rs.Consumes;

import javax.ws.rs.POST;

import javax.ws.rs.Produces;

import javax.ws.rs.Path;

import javax.ws.rs.core.MediaType;

@Path("/posts")

public class PostResource {

@POST

@Consumes(MediaType.APPLICATION\_JSON)

@Produces(MediaType.APPLICATION\_JSON)

public Post createPostJson(Post post) {

post.setId(123);

return post;

}

@POST

@Path("/xml")

@Consumes(MediaType.APPLICATION\_XML)

@Produces(MediaType.APPLICATION\_XML)

public Post createPostXml(Post post) {

post.setId(123);

return post;

}

}

* + - **Test with Postman:** Use appropriate Content-Type headers for JSON and XML requests.