**https://spring.io/guides/gs/producing-web-service/**

**Producing a SOAP web service**

This guide walks you through the process of creating a SOAP-based web service server with Spring.

What You Will build

You will build a server that exposes data from various European countries by using a WSDL-based SOAP web service.

|  |  |
| --- | --- |
|  | To simplify the example, you will use hardcoded data for the United Kingdom, Spain, and Poland. |

What You Need

* About 15 minutes
* A favorite text editor or IDE
* [JDK 1.8](http://www.oracle.com/technetwork/java/javase/downloads/index.html) or later
* [Gradle 4+](http://www.gradle.org/downloads) or [Maven 3.2+](https://maven.apache.org/download.cgi)
* You can also import the code straight into your IDE:
  + [Spring Tool Suite (STS)](https://spring.io/guides/gs/sts)
  + [IntelliJ IDEA](https://spring.io/guides/gs/intellij-idea/)

How to complete this guide

Like most Spring [Getting Started guides](https://spring.io/guides), you can start from scratch and complete each step or you can bypass basic setup steps that are already familiar to you. Either way, you end up with working code.

To **start from scratch**, move on to [Starting with Spring Initializr](https://spring.io/guides/gs/producing-web-service/#scratch).

To **skip the basics**, do the following:

* [Download](https://github.com/spring-guides/gs-soap-service/archive/main.zip) and unzip the source repository for this guide, or clone it using [Git](https://spring.io/understanding/Git): git clone <https://github.com/spring-guides/gs-soap-service.git>
* cd into gs-soap-service/initial
* Jump ahead to [Add the Spring-WS dependency](https://spring.io/guides/gs/producing-web-service/#initial).

**When you finish**, you can check your results against the code in gs-soap-service/complete.

Starting with Spring Initializr

You can use this [pre-initialized project](https://start.spring.io/#!type=maven-project&language=java&platformVersion=2.5.5&packaging=jar&jvmVersion=11&groupId=com.example&artifactId=producing-web-service&name=producing-web-service&description=Demo%20project%20for%20Spring%20Boot&packageName=com.example.producing-web-serv) and click Generate to download a ZIP file. This project is configured to fit the examples in this tutorial.

To manually initialize the project:

1. Navigate to [https://start.spring.io](https://start.spring.io/). This service pulls in all the dependencies you need for an application and does most of the setup for you.
2. Choose either Gradle or Maven and the language you want to use. This guide assumes that you chose Java.
3. Click **Dependencies** and select **Spring Web** and **Spring Web Services**.
4. Click **Generate**.
5. Download the resulting ZIP file, which is an archive of a web application that is configured with your choices.

|  |  |
| --- | --- |
|  | If your IDE has the Spring Initializr integration, you can complete this process from your IDE. |
|  | Both the pom.xml and build.gradle files need additional build information, which you will add in the next step. |

|  |  |
| --- | --- |
|  | You can also fork the project from Github and open it in your IDE or other editor. |

Add the Spring-WS dependency

The project needs to include spring-ws-core and wsdl4j as dependencies in your build file.

The following example shows the changes you need to make to the pom.xml file if you use Maven:

<dependency>

<groupId>wsdl4j</groupId>

<artifactId>wsdl4j</artifactId>

</dependency>

The following example shows the changes you need to make to the build.gradle file if you use Gradle:

dependencies {

implementation 'org.springframework.boot:spring-boot-starter-web'

implementation 'org.springframework.boot:spring-boot-starter-web-services'

implementation 'wsdl4j:wsdl4j'

jaxb("org.glassfish.jaxb:jaxb-xjc")

testImplementation('org.springframework.boot:spring-boot-starter-test')

}

Create an XML Schema to Define the Domain

The web service domain is defined in an XML schema file (XSD) that Spring-WS will automatically export as a WSDL.

Create an XSD file with operations to return a country’s name, population, capital, and currency. The following listing (from src/main/resources/countries.xsd) shows the necessary XSD file:

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:tns="http://spring.io/guides/gs-producing-web-service"

targetNamespace="http://spring.io/guides/gs-producing-web-service" elementFormDefault="qualified">

<xs:element name="getCountryRequest">

<xs:complexType>

<xs:sequence>

<xs:element name="name" type="xs:string"/>

</xs:sequence>

</xs:complexType>

</xs:element>

<xs:element name="getCountryResponse">

<xs:complexType>

<xs:sequence>

<xs:element name="country" type="tns:country"/>

</xs:sequence>

</xs:complexType>

</xs:element>

<xs:complexType name="country">

<xs:sequence>

<xs:element name="name" type="xs:string"/>

<xs:element name="population" type="xs:int"/>

<xs:element name="capital" type="xs:string"/>

<xs:element name="currency" type="tns:currency"/>

</xs:sequence>

</xs:complexType>

<xs:simpleType name="currency">

<xs:restriction base="xs:string">

<xs:enumeration value="GBP"/>

<xs:enumeration value="EUR"/>

<xs:enumeration value="PLN"/>

</xs:restriction>

</xs:simpleType>

</xs:schema>

Generate Domain Classes Based on an XML Schema

The next step is to generate Java classes from the XSD file. The right approach is to do this automatically during build time by using a Maven or Gradle plugin.

The following listing shows the necessary plugin configuration for Maven:

<plugin>

<groupId>org.codehaus.mojo</groupId>

<artifactId>jaxb2-maven-plugin</artifactId>

<version>2.5.0</version>

<executions>

<execution>

<id>xjc</id>

<goals>

<goal>xjc</goal>

</goals>

</execution>

</executions>

<configuration>

<sources>

<source>${project.basedir}/src/main/resources/countries.xsd</source>

</sources>

</configuration>

</plugin>

Generated classes are placed in the target/generated-sources/jaxb/ directory.

To do the same with Gradle, you first need to configure JAXB in your build file, as the following listing shows:

configurations {

jaxb

}

bootJar {

archiveBaseName = 'gs-producing-web-service'

archiveVersion = '0.1.0'

}

|  |  |
| --- | --- |
|  | The build files have tag and end comments. These tags make it easier to extract bits of it into this guide for a more detailed explanation. You do not need these comments in your own build file. |

The next step is to add the genJaxb task, which Gradle uses to generate Java classes. We need to configure gradle to find these generated Java classes in build/generated-sources/jaxb and add genJaxb as a dependency of compileJava task. The following listing shows the necessary addition:

sourceSets {

main {

java {

srcDir 'src/main/java'

srcDir 'build/generated-sources/jaxb'

}

}

}

task genJaxb {

ext.sourcesDir = "${buildDir}/generated-sources/jaxb"

ext.schema = "src/main/resources/countries.xsd"

outputs.dir sourcesDir

doLast() {

project.ant {

taskdef name: "xjc", classname: "com.sun.tools.xjc.XJCTask",

classpath: configurations.jaxb.asPath

mkdir(dir: sourcesDir)

xjc(destdir: sourcesDir, schema: schema) {

arg(value: "-wsdl")

produces(dir: sourcesDir, includes: "\*\*/\*.java")

}

}

}

}

compileJava.dependsOn genJaxb

Because Gradle does not have a JAXB plugin (yet), it involves an Ant task, which makes it a bit more complex than in Maven.

In both cases, the JAXB domain object generation process has been wired into the build tool’s lifecycle, so there are no extra steps to run.

Create Country Repository

In order to provide data to the web service, create a country repository. In this guide, you create a dummy country repository implementation with hardcoded data. The following listing (from src/main/java/com/example/producingwebservice/CountryRepository.java) shows how to do so:

package com.example.producingwebservice;

import javax.annotation.PostConstruct;

import java.util.HashMap;

import java.util.Map;

import io.spring.guides.gs\_producing\_web\_service.Country;

import io.spring.guides.gs\_producing\_web\_service.Currency;

import org.springframework.stereotype.Component;

import org.springframework.util.Assert;

@Component

public class CountryRepository {

private static final Map<String, Country> countries = new HashMap<>();

@PostConstruct

public void initData() {

Country spain = new Country();

spain.setName("Spain");

spain.setCapital("Madrid");

spain.setCurrency(Currency.EUR);

spain.setPopulation(46704314);

countries.put(spain.getName(), spain);

Country poland = new Country();

poland.setName("Poland");

poland.setCapital("Warsaw");

poland.setCurrency(Currency.PLN);

poland.setPopulation(38186860);

countries.put(poland.getName(), poland);

Country uk = new Country();

uk.setName("United Kingdom");

uk.setCapital("London");

uk.setCurrency(Currency.GBP);

uk.setPopulation(63705000);

countries.put(uk.getName(), uk);

}

public Country findCountry(String name) {

Assert.notNull(name, "The country's name must not be null");

return countries.get(name);

}

}

Create Country Service Endpoint

To create a service endpoint, you need only a POJO with a few Spring WS annotations to handle the incoming SOAP requests. The following listing (from src/main/java/com/example/producingwebservice/CountryEndpoint.java) shows such a class:

package com.example.producingwebservice;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.ws.server.endpoint.annotation.Endpoint;

import org.springframework.ws.server.endpoint.annotation.PayloadRoot;

import org.springframework.ws.server.endpoint.annotation.RequestPayload;

import org.springframework.ws.server.endpoint.annotation.ResponsePayload;

import io.spring.guides.gs\_producing\_web\_service.GetCountryRequest;

import io.spring.guides.gs\_producing\_web\_service.GetCountryResponse;

@Endpoint

public class CountryEndpoint {

private static final String NAMESPACE\_URI = "http://spring.io/guides/gs-producing-web-service";

private CountryRepository countryRepository;

@Autowired

public CountryEndpoint(CountryRepository countryRepository) {

this.countryRepository = countryRepository;

}

@PayloadRoot(namespace = NAMESPACE\_URI, localPart = "getCountryRequest")

@ResponsePayload

public GetCountryResponse getCountry(@RequestPayload GetCountryRequest request) {

GetCountryResponse response = new GetCountryResponse();

response.setCountry(countryRepository.findCountry(request.getName()));

return response;

}

}COPY

The [@Endpoint](https://docs.spring.io/spring-ws/sites/2.0/apidocs/org/springframework/ws/server/endpoint/annotation/Endpoint.html) annotation registers the class with Spring WS as a potential candidate for processing incoming SOAP messages.

The [@PayloadRoot](https://docs.spring.io/spring-ws/sites/2.0/apidocs/org/springframework/ws/server/endpoint/annotation/PayloadRoot.html) annotation is then used by Spring WS to pick the handler method, based on the message’s namespace and localPart.

The [@RequestPayload](https://docs.spring.io/spring-ws/sites/2.0/apidocs/org/springframework/ws/server/endpoint/annotation/RequestPayload.html) annotation indicates that the incoming message will be mapped to the method’s request parameter.

The [@ResponsePayload](https://docs.spring.io/spring-ws/sites/2.0/apidocs/org/springframework/ws/server/endpoint/annotation/ResponsePayload.html) annotation makes Spring WS map the returned value to the response payload.

|  |  |
| --- | --- |
|  | In all of these chunks of code, the io.spring.guides classes will report compile-time errors in your IDE unless you have run the task to generate the domain classes based on the WSDL. |

Configure Web Service Beans

Create a new class with Spring WS-related beans configuration, as the following listing (from src/main/java/com/example/producingwebservice/WebServiceConfig.java) shows:

package com.example.producingwebservice;

import org.springframework.boot.web.servlet.ServletRegistrationBean;

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.core.io.ClassPathResource;

import org.springframework.ws.config.annotation.EnableWs;

import org.springframework.ws.config.annotation.WsConfigurerAdapter;

import org.springframework.ws.transport.http.MessageDispatcherServlet;

import org.springframework.ws.wsdl.wsdl11.DefaultWsdl11Definition;

import org.springframework.xml.xsd.SimpleXsdSchema;

import org.springframework.xml.xsd.XsdSchema;

@EnableWs

@Configuration

public class WebServiceConfig extends WsConfigurerAdapter {

@Bean

public ServletRegistrationBean<MessageDispatcherServlet> messageDispatcherServlet(ApplicationContext applicationContext) {

MessageDispatcherServlet servlet = new MessageDispatcherServlet();

servlet.setApplicationContext(applicationContext);

servlet.setTransformWsdlLocations(true);

return new ServletRegistrationBean<>(servlet, "/ws/\*");

}

@Bean(name = "countries")

public DefaultWsdl11Definition defaultWsdl11Definition(XsdSchema countriesSchema) {

DefaultWsdl11Definition wsdl11Definition = new DefaultWsdl11Definition();

wsdl11Definition.setPortTypeName("CountriesPort");

wsdl11Definition.setLocationUri("/ws");

wsdl11Definition.setTargetNamespace("http://spring.io/guides/gs-producing-web-service");

wsdl11Definition.setSchema(countriesSchema);

return wsdl11Definition;

}

@Bean

public XsdSchema countriesSchema() {

return new SimpleXsdSchema(new ClassPathResource("countries.xsd"));

}

}COPY

* Spring WS uses a different servlet type for handling SOAP messages: [MessageDispatcherServlet](https://docs.spring.io/spring-ws/sites/2.0/apidocs/org/springframework/ws/transport/http/MessageDispatcherServlet.html). It is important to inject and set [ApplicationContext](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/context/ApplicationContext.html) to [MessageDispatcherServlet](https://docs.spring.io/spring-ws/sites/2.0/apidocs/org/springframework/ws/transport/http/MessageDispatcherServlet.html). Without that, Spring WS will not automatically detect Spring beans.
* Naming this bean messageDispatcherServlet does not replace Spring Boot’s [default DispatcherServlet bean](https://docs.spring.io/spring-boot/docs/2.4.3/reference/htmlsingle/#howto-switch-off-the-spring-mvc-dispatcherservlet).
* [DefaultMethodEndpointAdapter](https://docs.spring.io/spring-ws/sites/2.0/apidocs/org/springframework/ws/server/endpoint/adapter/DefaultMethodEndpointAdapter.html) configures the annotation-driven Spring WS programming model. This makes it possible to use the various annotations, such as [@Endpoint](https://docs.spring.io/spring-ws/sites/2.0/apidocs/org/springframework/ws/server/endpoint/annotation/Endpoint.html) (mentioned earlier).
* [DefaultWsdl11Definition](https://docs.spring.io/spring-ws/sites/2.0/apidocs/org/springframework/ws/wsdl/wsdl11/DefaultWsdl11Definition.html) exposes a standard WSDL 1.1 by using [XsdSchema](https://docs.spring.io/spring-ws/sites/2.0/apidocs/org/springframework/xml/xsd/XsdSchema.html)

|  |  |
| --- | --- |
|  | You need to specify bean names for [MessageDispatcherServlet](https://docs.spring.io/spring-ws/sites/2.0/apidocs/org/springframework/ws/transport/http/MessageDispatcherServlet.html) and [DefaultWsdl11Definition](https://docs.spring.io/spring-ws/sites/2.0/apidocs/org/springframework/ws/wsdl/wsdl11/DefaultWsdl11Definition.html). Bean names determine the URL under which the web service and the generated WSDL file are available. In this case, the WSDL will be available under http://<host>:<port>/ws/countries.wsdl. |

This configuration also uses the WSDL location servlet transformation: servlet.setTransformWsdlLocations(true). If you visit <http://localhost:8080/ws/countries.wsdl>, the soap:address will have the proper address. If you instead visit the WSDL from the public facing IP address assigned to your machine, you will see that address instead.

Make the Application Executable

Spring Boot creates an application class for you. In this case, it needs no further modification. You can use it to run this application. The following listing (from src/main/java/com/example/producingwebservice/ProducingWebServiceApplication.java) shows the application class:

package com.example.producingwebservice;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class ProducingWebServiceApplication {

public static void main(String[] args) {

SpringApplication.run(ProducingWebServiceApplication.class, args);

}

}COPY

@SpringBootApplication is a convenience annotation that adds all of the following:

* @Configuration: Tags the class as a source of bean definitions for the application context.
* @EnableAutoConfiguration: Tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings. For example, if spring-webmvc is on the classpath, this annotation flags the application as a web application and activates key behaviors, such as setting up a DispatcherServlet.
* @ComponentScan: Tells Spring to look for other components, configurations, and services in the com/example package, letting it find the controllers.

The main() method uses Spring Boot’s SpringApplication.run() method to launch an application. Did you notice that there was not a single line of XML? There is no web.xml file, either. This web application is 100% pure Java and you did not have to deal with configuring any plumbing or infrastructure.

Build an executable JAR

You can run the application from the command line with Gradle or Maven. You can also build a single executable JAR file that contains all the necessary dependencies, classes, and resources and run that. Building an executable jar makes it easy to ship, version, and deploy the service as an application throughout the development lifecycle, across different environments, and so forth.

If you use Gradle, you can run the application by using ./gradlew bootRun. Alternatively, you can build the JAR file by using ./gradlew build and then run the JAR file, as follows:

java -jar build/libs/gs-soap-service-0.1.0.jar

If you use Maven, you can run the application by using ./mvnw spring-boot:run. Alternatively, you can build the JAR file with ./mvnw clean package and then run the JAR file, as follows:

java -jar target/gs-soap-service-0.1.0.jar

|  |  |
| --- | --- |
|  | The steps described here create a runnable JAR. You can also [build a classic WAR file](https://spring.io/guides/gs/convert-jar-to-war/). |

Logging output is displayed. The service should be up and running within a few seconds.

Test the Application

Now that the application is running, you can test it. Create a file called request.xml that contains the following SOAP request:

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"

xmlns:gs="http://spring.io/guides/gs-producing-web-service">

<soapenv:Header/>

<soapenv:Body>

<gs:getCountryRequest>

<gs:name>Spain</gs:name>

</gs:getCountryRequest>

</soapenv:Body>

</soapenv:Envelope>

The are a few options when it comes to testing the SOAP interface. You can use something similar to [SoapUI](http://www.soapui.org/) or use command-line tools if you are on a \*nix/Mac system. The following example uses curl from the command line:

# Use data from file

curl --header "content-type: text/xml" -d @request.xml http://localhost:8080/ws

# Use inline XML data

curl <<-EOF -fsSL -H "content-type: text/xml" -d @- http://localhost:8080/ws \

> target/response.xml && xmllint --format target/response.xml

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"

xmlns:gs="http://spring.io/guides/gs-producing-web-service">

<soapenv:Header/>

<soapenv:Body>

<gs:getCountryRequest>

<gs:name>Spain</gs:name>

</gs:getCountryRequest>

</soapenv:Body>

</soapenv:Envelope>

EOF

As a result, you should see the following response:

<?xml version="1.0"?>

<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">

<SOAP-ENV:Header/>

<SOAP-ENV:Body>

<ns2:getCountryResponse xmlns:ns2="http://spring.io/guides/gs-producing-web-service">

<ns2:country>

<ns2:name>Spain</ns2:name>

<ns2:population>46704314</ns2:population>

<ns2:capital>Madrid</ns2:capital>

<ns2:currency>EUR</ns2:currency>

</ns2:country>

</ns2:getCountryResponse>

</SOAP-ENV:Body>

</SOAP-ENV:Envelope>COPY

|  |  |
| --- | --- |
|  | Odds are that the output will be a compact XML document instead of the nicely formatted one shown above. If you have xmllib2 installed on your system, you can curl -fsSL --header "content-type: text/xml" -d @request.xml <http://localhost:8080/ws> > output.xml and xmllint --format output.xml see the results formatted nicely. |

Summary

Congratulations! You have developed a SOAP-based service with Spring Web Services.

See Also

The following guides may also be helpful:

* [Consuming a SOAP web service](https://spring.io/guides/gs/consuming-web-service/)
* [Building an Application with Spring Boot](https://spring.io/guides/gs/spring-boot/)

Want to write a new guide or contribute to an existing one? Check out our [contribution guidelines](https://github.com/spring-guides/getting-started-guides/wiki).

|  |  |
| --- | --- |
|  | All guides are released with an ASLv2 license for the code, and an [Attribution, NoDerivatives creative commons license](https://creativecommons.org/licenses/by-nd/3.0/) for the writing. |