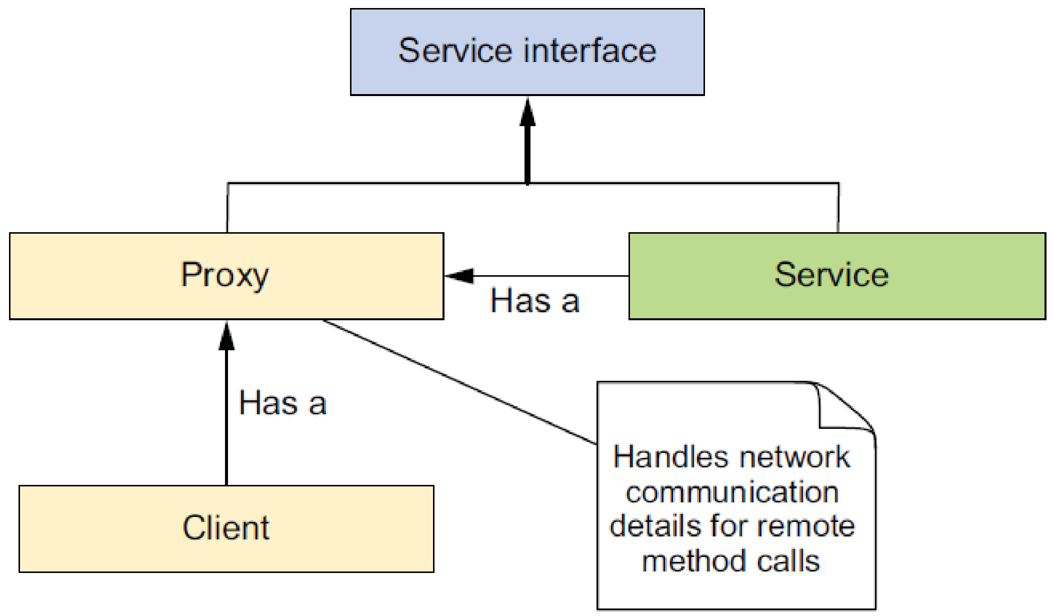
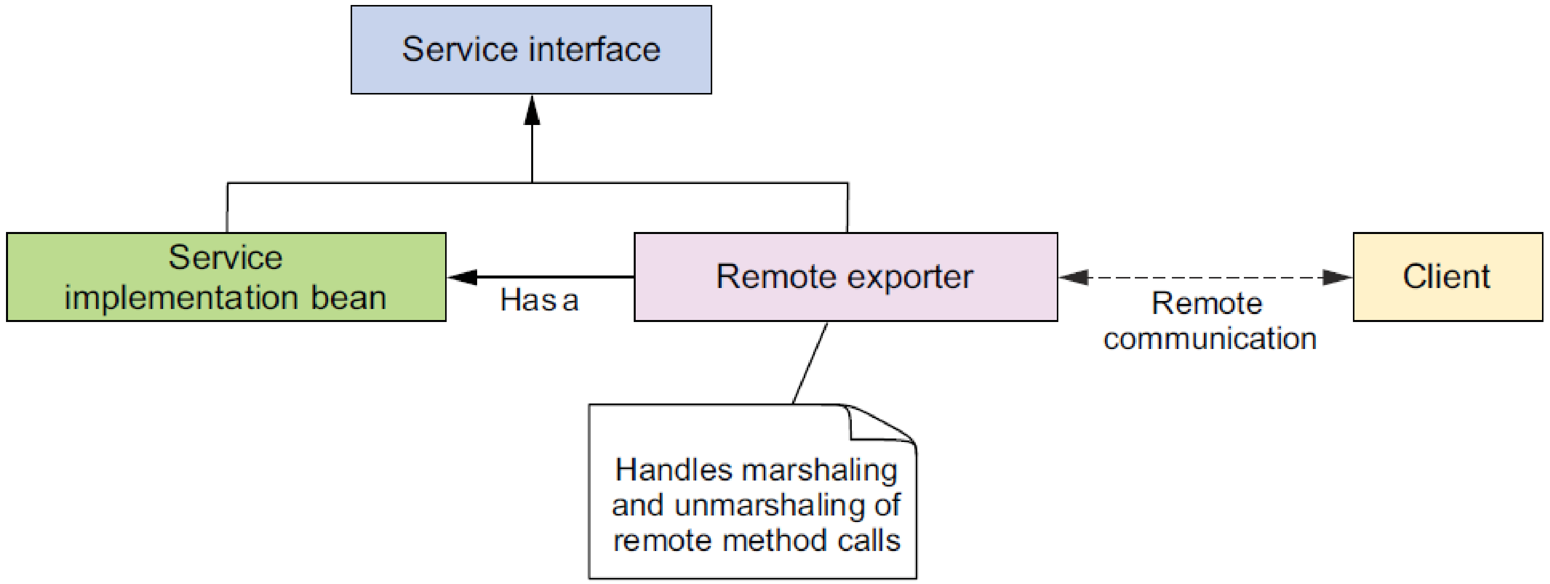
System Integration

- ✓ Remote Method Invocation
- ✓ Hessian
- ✓ Soap web services with JAX-WS
- ✓ REST web services
- ✓ Java Messaging System


Service interface
Proxy
Has a
Service
Has a
Client
Handles network communication details for remote method calls


Service interface
Service implementation bean
Has a
Remote exporter
Remote communication
Client
Handles marshaling and unmarshaling of remote method calls

RMI Exporting

```
@Bean
public RmiServiceExporter rmiExporter(MyService myService) {

RmiServiceExporter rmiExporter = new RmiServiceExporter();

        rmiExporter.setService(myService );
        rmiExporter.setServiceName(“ServiceName");
        rmiExporter.setServiceInterface(MyService.class);

return rmiExporter;
}
```

RMI Exporting

```
rmiExporter.setRegistryHost("rmi.spitter.com");
rmiExporter.setRegistryPort(1199);
```

RMI proxy

```
@Bean
public RmiProxyFactoryBean myService() {
RmiProxyFactoryBean rmiProxy = new RmiProxyFactoryBean();

        rmiProxy.setServiceUrl("rmi://localhost/MyService");
        rmiProxy.setServiceInterface(MyService.class);

        return rmiProxy;
}
```

Hessian Exporting

```
@Bean
public HessianServiceExporter
        hessianExportedMyService(MyService service){

 HessianServiceExporter exporter = new  HessianServiceExporter();

    exporter.setService(service);
    exporter.setServiceInterface(MyService.class);
    return exporter;
}
```

Hessian Proxy

```
@Bean
public HessianProxyFactoryBean myService() {

HessianProxyFactoryBean proxy = new HessianProxyFactoryBean();
proxy.setServiceUrl("http://localhost:8080/Spr1/my.service");
proxy.setServiceInterface(MyService.class);

return proxy;
}
```

Don’t forget to add the mapping in the dispatcher servlet

```
@Override
protected String[] getServletMappings() {
        return new String[] { "/", "*.service" };
}
```

JAX-WS Model Definition

@WebService

- annotation used to mark a class that defines a web service

@WebMethod

- annotation used to mark a web service operation

@WebParam

- annotation used as qualifier for the method parameters

JAX-WS Exporting

```
@Bean
public SimpleJaxWsServiceExporter jaxWsExporter() {
        SimpleJaxWsServiceExporter exporter =
                new SimpleJaxWsServiceExporter();

        exporter.setBaseAddress("http://localhost:8888/services/");

}
```

JAX-WS Proxy

```
@Bean
public JaxWsPortProxyFactoryBean myService() {
                    JaxWsPortProxyFactoryBean proxy = new
                                        JaxWsPortProxyFactoryBean();
proxy.setWsdlDocument(
                    "http://localhost:8080/services/MyService?wsdl");

          proxy.setServiceName(“myService");
          proxy.setPortName(“myServiceHttpPort");
          proxy.setServiceInterface(MyService.class);
          proxy.setNamespaceUri("http://ws");

          return proxy;
}
```

Representational **S**tate **T**ransfer

Usually the messages are formatted:

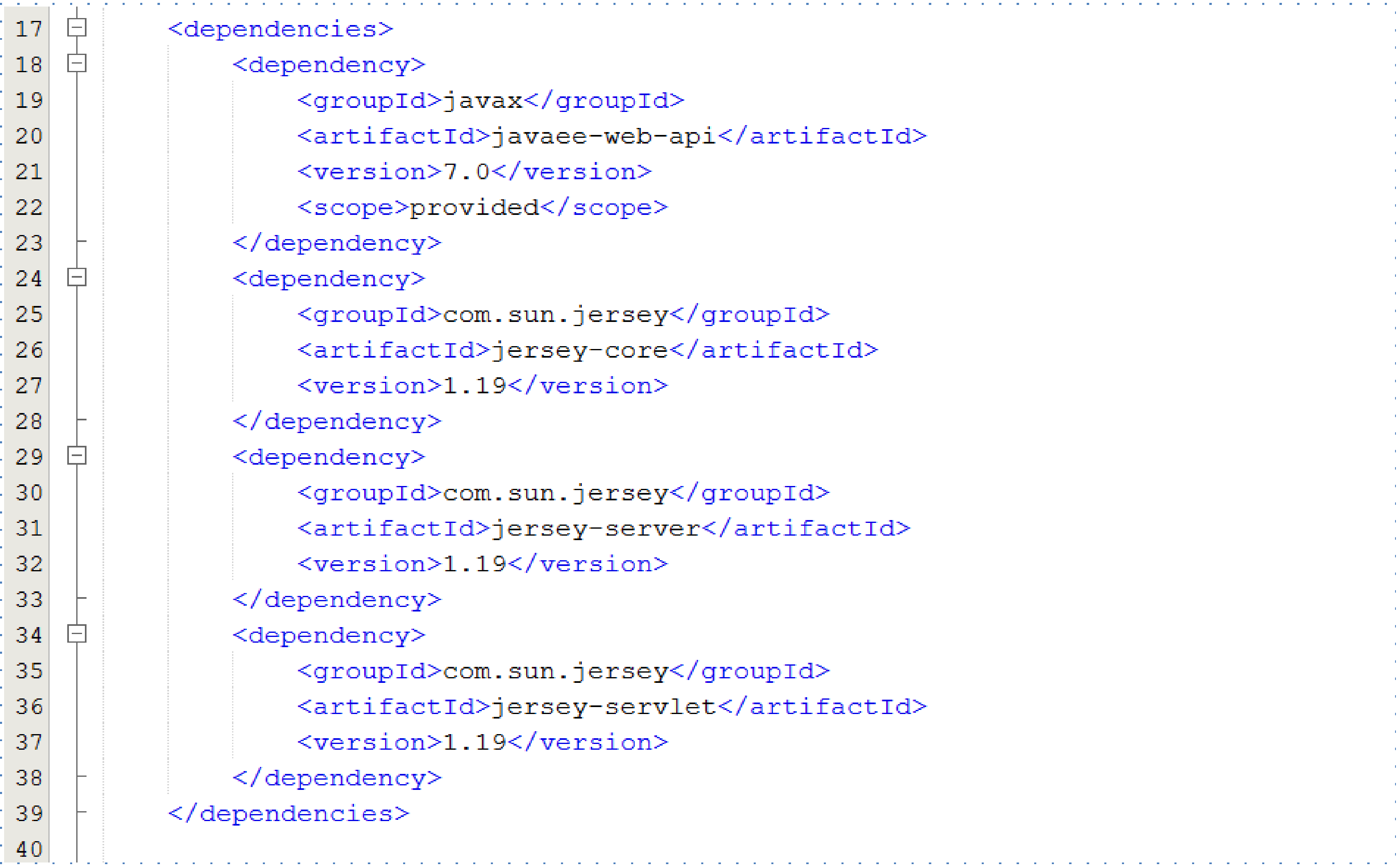
- JSON
- XML

HTTP is used to send the data similarly to SOAP
There is **no standard** of the responses and replies

HTTP Methods

- *Create*—POST
- *Read*—GET
- *Update*—PUT or PATCH
- *Delete*—DELETE

Using **Jersey**



```
<dependencies>
    <dependency>
        <groupId>javax</groupId>
        <artifactId>javaee-web-api</artifactId>
        <version>7.0</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>com.sun.jersey</groupId>
        <artifactId>jersey-core</artifactId>
        <version>1.19</version>
    </dependency>
    <dependency>
        <groupId>com.sun.jersey</groupId>
        <artifactId>jersey-server</artifactId>
        <version>1.19</version>
    </dependency>
    <dependency>
        <groupId>com.sun.jersey</groupId>
        <artifactId>jersey-servlet</artifactId>
        <version>1.19</version>
    </dependency>
</dependencies>

```

Defining a web operation

A **web operation** is defined with a method marked with the following annotations:

@Get or @Post

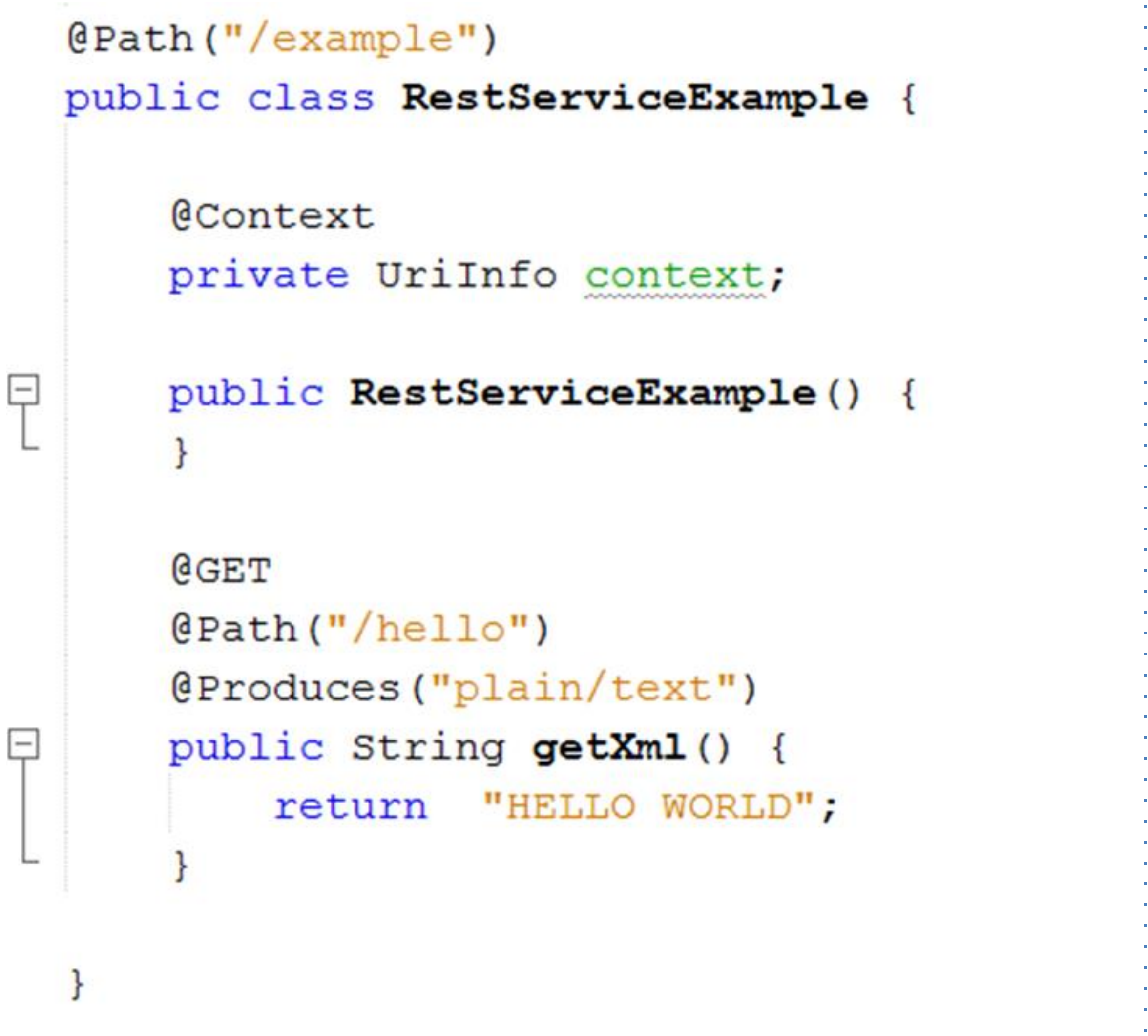
- to specify the HTTP transfer method

@Path

- to specify the web path at each the operation is accessible

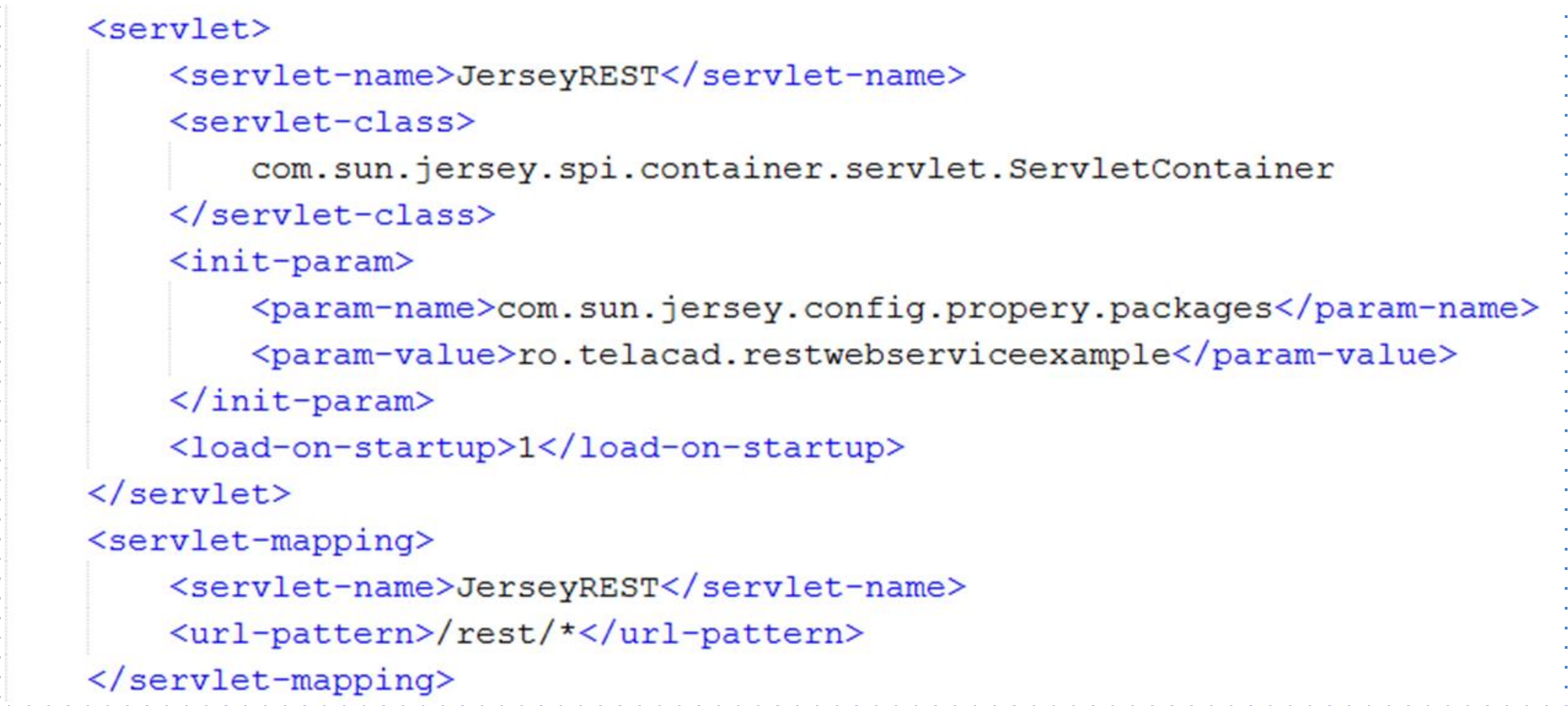
@Produces

- to specify the desired result of the response
E. g. JSON, XML, plain text



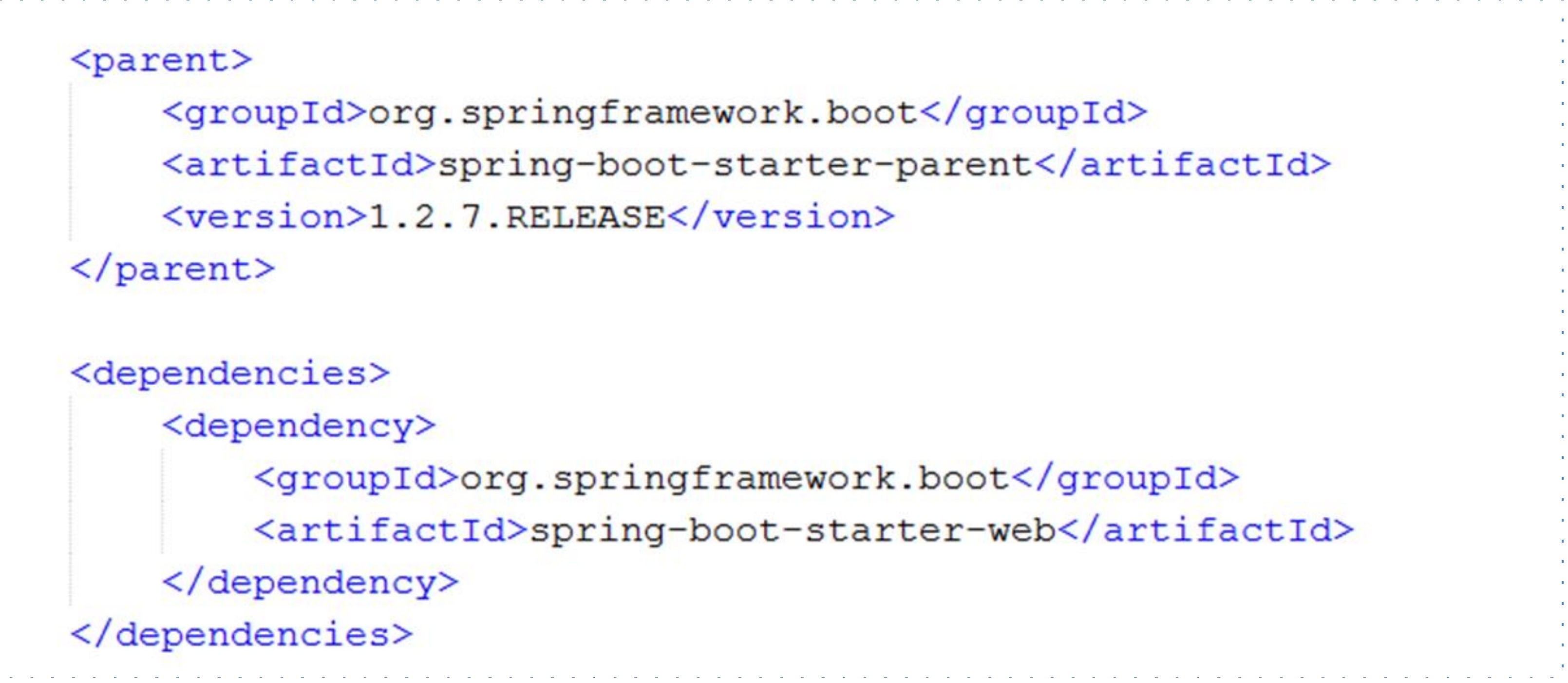
```
@Path("/example")
public class RestServiceExample {

    @Context
    private UriInfo context;

    public RestServiceExample() {
    }

    @GET
    @Path("/hello")
    @Produces("plain/text")
    public String getXml() {
        return  "HELLO WORLD";
    }

}
```

Declare the servlet definition in web.xml



```
<servlet>
    <servlet-name>JerseyREST</servlet-name>
    <servlet-class>
        com.sun.jersey.spi.container.servlet.ServletContainer
    </servlet-class>
    <init-param>
        <param-name>com.sun.jersey.config.propery.packages</param-name>
        <param-value>ro.telacad.restwebserviceexample</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>JerseyREST</servlet-name>
    <url-pattern>/rest/*</url-pattern>
</servlet-mapping>
```

Using Spring Boot to expose REST services



```xml
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.2.7.RELEASE</version>
</parent>

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

Three annotations are basic to know in order to define the REST interface:

@RestController

- used to define the rest interface class

@RequestMapping

- used to define the web path of an operation

@RequestParam

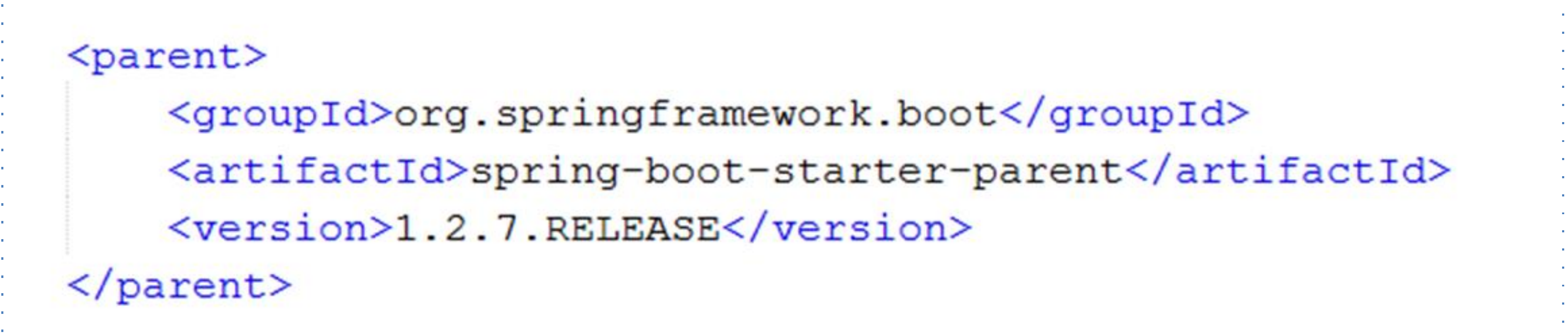
- used to mark a web parameter to the method paramter

```
@RestController
public class CatRestInterface{

        @RequestMapping(“/cat”)
        public Cat giveMeACat(

                @RequestParam(value=“name”,
                                   defaultValue=“Tom”) String name
                ){

                return new Cat(name);
        }

}
```

Finally …
create a Spring Boot Application main class

```
@SpringBootApplication
public class Main{
        public static void main(String [] args){
                SpringApplication.run(Main.class, args);
        }
}
```

Consuming a REST service



```xml
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.2.7.RELEASE</version>
</parent>
```

Consuming a REST service

```xml
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-web</artifactId>
    </dependency>
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
    </dependency>
</dependencies>
```

Consuming a REST service

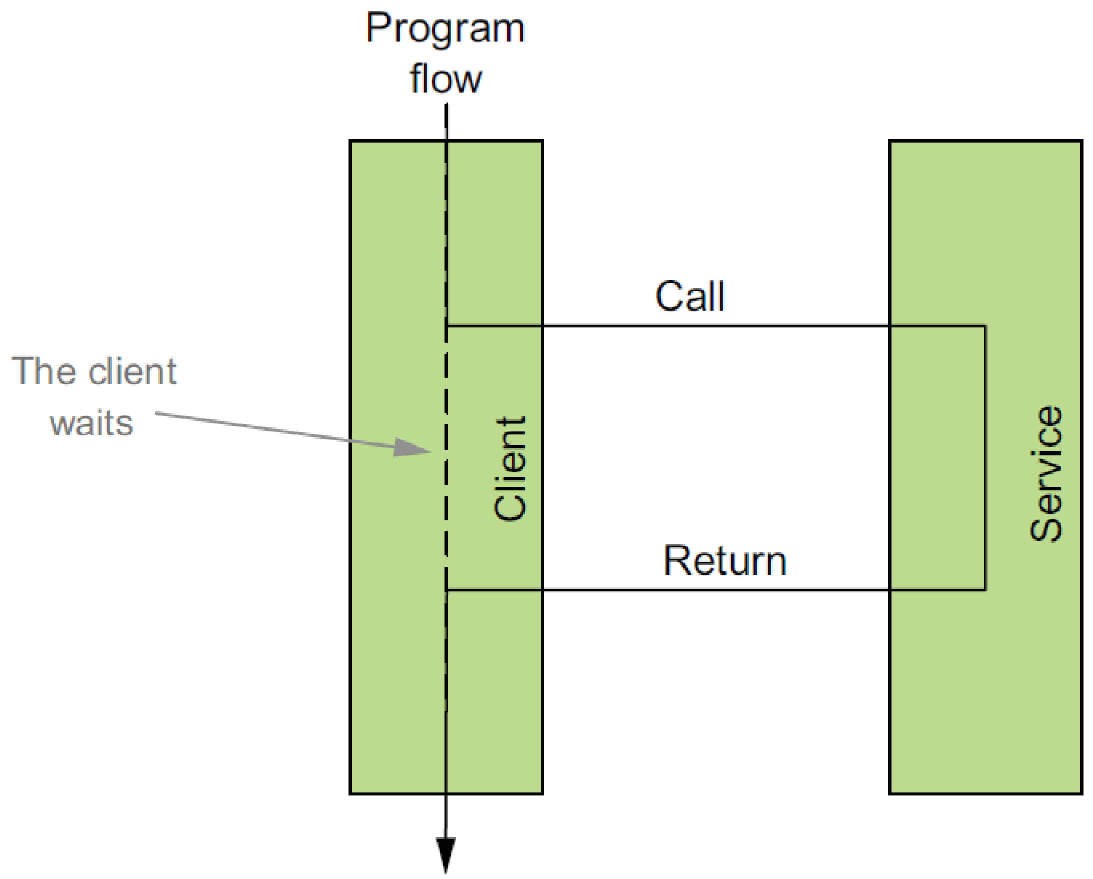
RestTemplate restTemplate = new **RestTemplate**();

String method = [“http://loaclhost:8080/cat](http://loaclhost:8080/cat)";
Cat cat = **restTemplate.getForObject(method, Cat.class);**

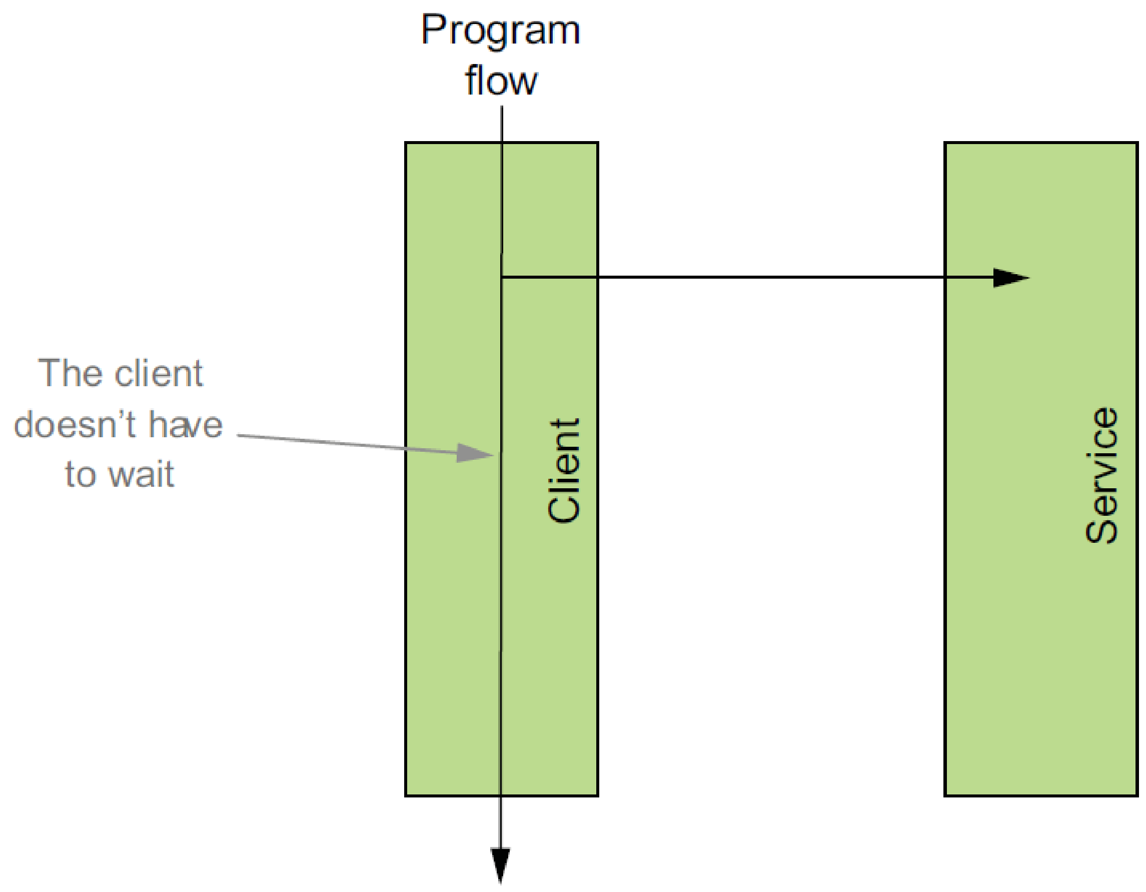
JMS – Java Messaging System

1. Asynchronous communication
2. Use a third party communication member
   - ActiveMQ, GlassFish ?
3. Guaranteed delivery
4. Decoupling
   - Implementation of sort of command pattern

Synchronous vs Asynchronous

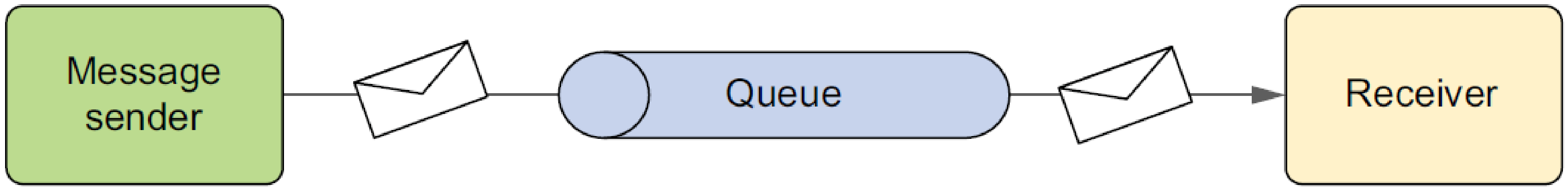


Synchronous vs Asynchronous



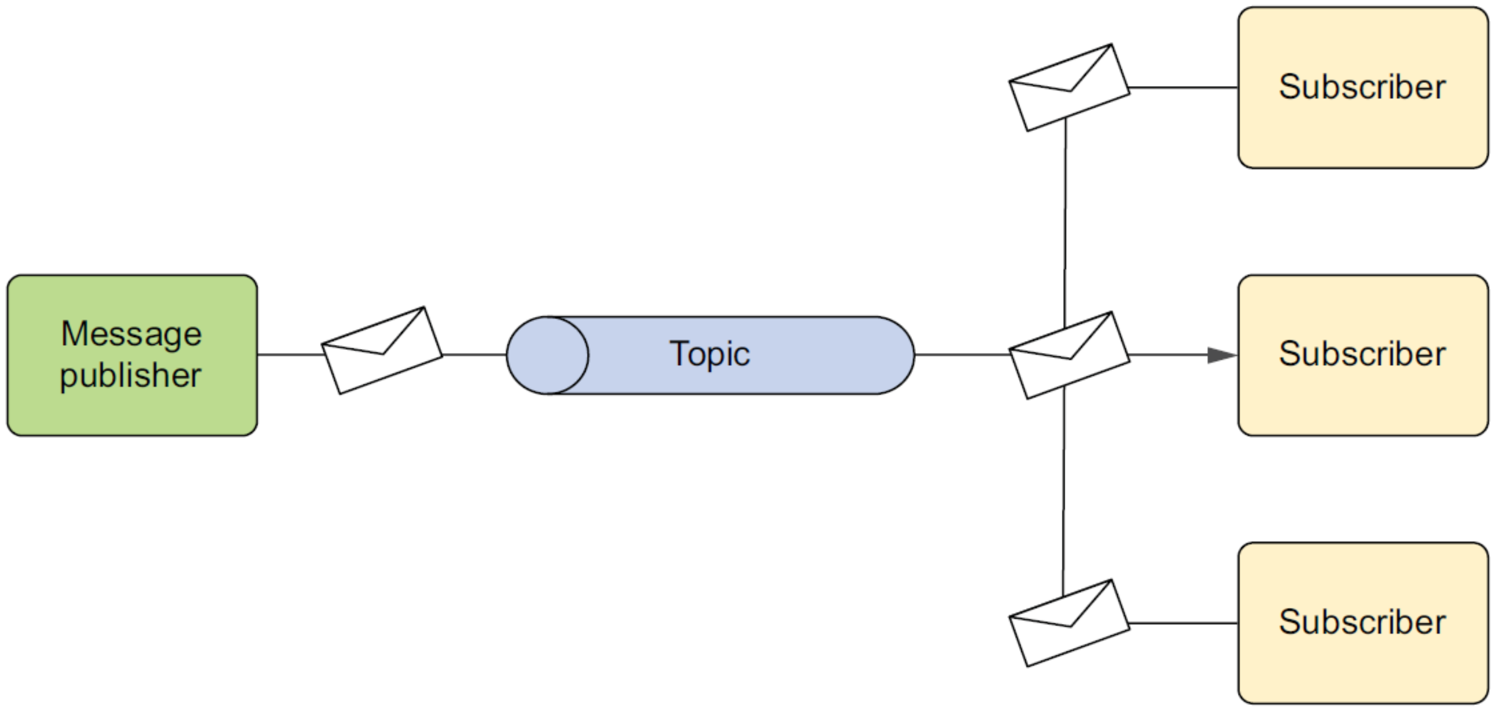
Queues vs Topics

A queue is a **producer – consumer** strategy



Queues vs Topics

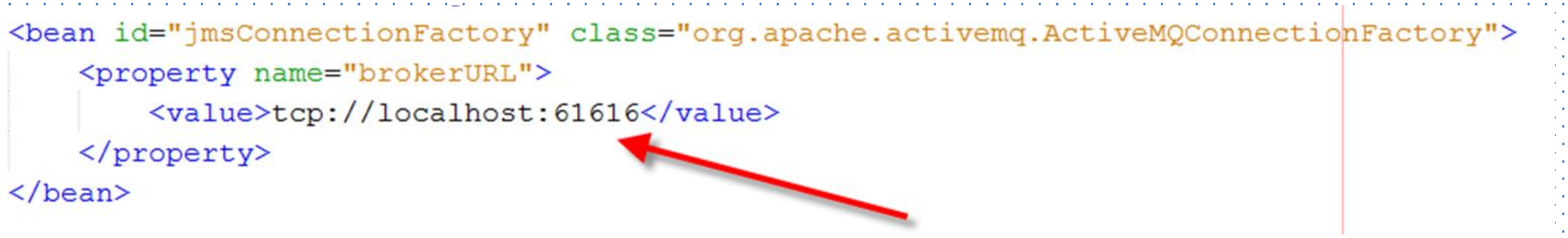
A topic is a **publisher – subscriber** strategy



Steps to produce a message

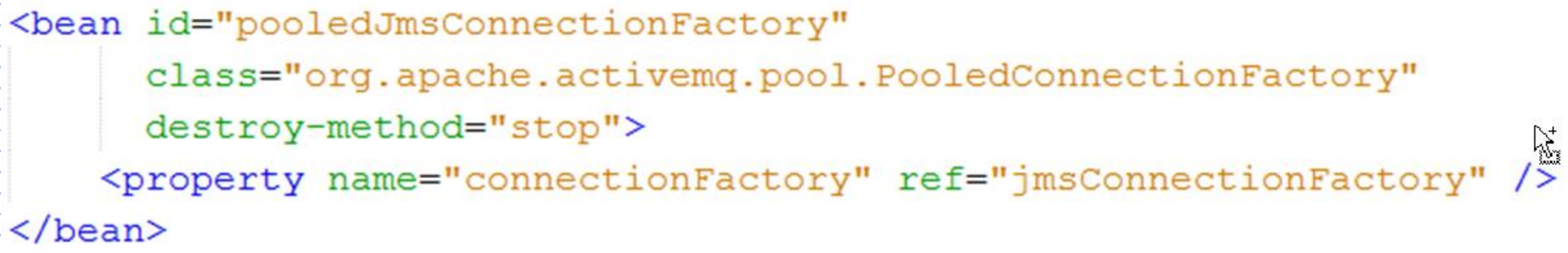
1. Define a JMS connection factory
2. Create a pool JMS connection factory
3. Create the resource – a topic or a queue
4. Create a JMS template bean
5. Send the message

Define the JMS connection factory



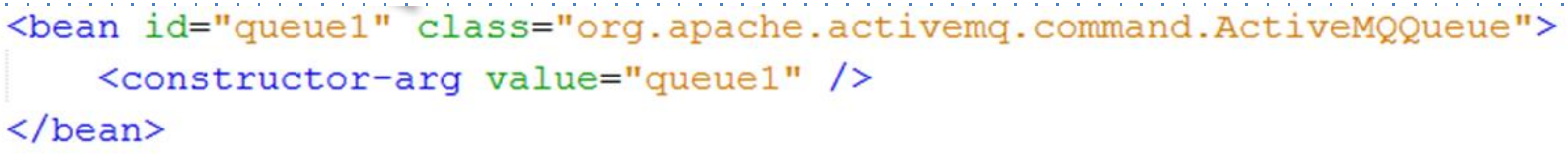
```
<bean id="jmsConnectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL">
        <value>tcp://localhost:61616</value>
    </property>
</bean>
```

Create a pool JMS connection factory



```xml
<bean id="pooledJmsConnectionFactory"
        class="org.apache.activemq.pool.PooledConnectionFactory"
        destroy-method="stop">
    <property name="connectionFactory" ref="jmsConnectionFactory" />
</bean>
```

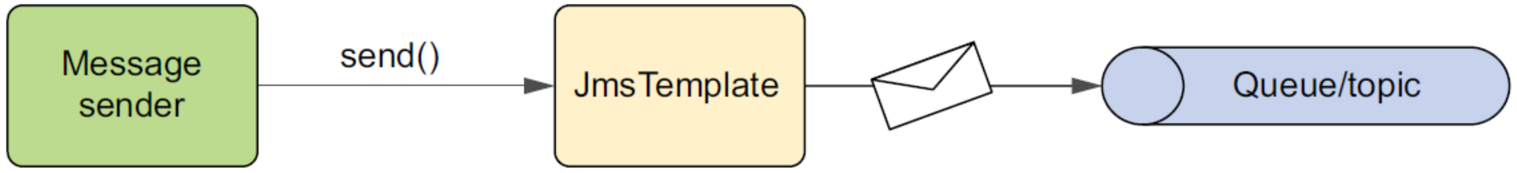
Create the resource



```xml
<bean id="queue1" class="org.apache.activemq.command.ActiveMQQueue">
    <constructor-arg value="queue1" />
</bean>
```

Create a JMS template bean

```
<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
    <property name="connectionFactory" ref="pooledJmsConnectionFactory" />
</bean>
```



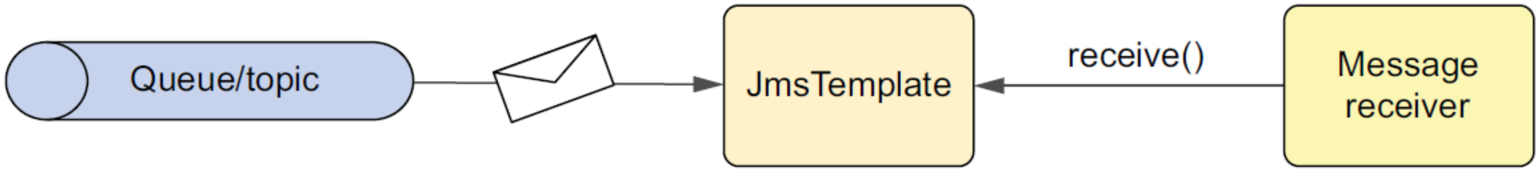
Sending the message

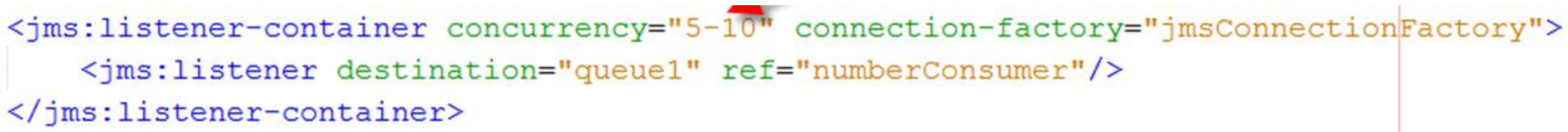
```
@Autowired
JmsTemplate template;

@Autowired
Destination myQueue;

template.convertAndSend(myQueue, “message”);
```

Steps to retrieve a message

1. Create a message listener – **MessageListener**
   - Override the **onMessage** method

2. Create a listener container





```xml
<jms:listener-container concurrency="5-10" connection-factory="jmsConnectionFactory">
    <jms:listener destination="queue1" ref="numberConsumer"/>
</jms:listener-container>
```