CITRUS

Reference Guide

Table of Contents

Citrus Simulator - Reference Documentation

Welcome to the Citrus simulator

This is a standalone simulator application for different message transports such as Http REST APIs, SOAP WebService interface and JMS messaging.

Clients are able to access the simulator endpoints and the simulator answers with predefined response messages. The simulator response logic is very powerful and enables us to simulate any kind of server interface.

The simulator uses the test framework Citrus www.citrusframework.org

NOTE: *This project is still under construction!*

Contributions and feedback is highly appreciated!

Authors

Christoph Deppisch

Version

0.7

www.citrusframework.org

Preconditions

The Citrus simulator requires some software installed on your localhost.

Java 8

The simulator is a Java application coded in Java 8. Following from that you need at least Java 8 to run it as a Spring Boot web application. Please make sure that you have Java development kit installed and set up. You can verify this with this command in a new terminal window.

```
> java -version
```

Build tools

The simulator uses Maven as build tool. If you only want to run a distribution artifact of the simulator you are fine with just Java on your machine. In case you want to build and maintain your simulator instance you need Maven to build your simulator application. We used Maven 3 when coding the simulator. You can verify the Maven installation on your host with this command:

```
> mvn -version
```

Browsers

The simulator provides a small web user interface when started. You can access this web UI with your browser. As we are in an early state in this project we do not invest much time in full cross-browser compatibility. We use Chrome and Firefox during development. So the simulator application is most likely to be 100% working on these two browsers. Of course other browsers might work without any limitations, too.

Installation

The Citrus simulator is started as a web application that uses Spring Boot. The easiest way to get started is to use a Maven archetype that creates a new project for you.

```
mvn archetype:generate -DarchetypeGroupId=com.consol.citrus.archetypes -DarchetypeArtifactId=
```

If you execute the command above the Maven archetype generator will ask you some questions about versions and project names. Once you have completed the generation you get a new Maven project that is ready to use. The project is created in a new folder on your machine. Switch to that folder and continue to build the project.

There are different simulator archetypes available. Please pick the most convenient archetype according to your project purpose.

- **citrus-simulator-archetypes-rest** Http REST simulator sample
- **citrus-simulator-archetypes-ws** SOAP web service simulator sample
- **citrus-simulator-archetypes-jms** JMS simulator sample

Build project

You can directly build the new project with

```
mvn install
```

This compiles, tests and packages the project. Now we can run the simulator:

```
mvn spring-boot:run
```

You will see the application starting up. Usually you will see some console log output. The web server should start within seconds. Once the application is up and running you can open your browser and point to ***http://localhost:8080***.

Now everything is set up and you can start to create some simulator scenarios.

Use simulator artifacts

The simulator project creates a web application artifact. After building you can find this WAR file in target/citrus-simulator-1.0.war

Name and version of that archive file may be different according to your project settings. You can start the simulator with Java

```
java -jar citrus-simulator-1.0.war
```

You will see the application starting up. Usually you will see some console log output. The web server should start within seconds. Once the application is up and running you can open your browser and point to ***http://localhost:8080***.

That's it you are ready to use the Citrus simulator.

Concepts

The Citrus simulator project has the primary focus to provide messaging simulation as a standalone server. Once started the simulator provides different endpoints (Http REST, JMS, SOAP Web Service, and so on) and waits for incoming requests. The time a request arrives at one of the endpoints the simulator maps the request to a predefined simulator scenario which is automatically executed immediately.

The mapping is done by extracting a mapping key from the request data. This can be a special request header or a XPath expression that is evaluated on the request payload.

The simulator scenario is capable of handling the request message and will return a proper response message to the calling client. With different scenarios defined the simulator is able to respond to different requests accordingly.

Each scenario defines a very special simulation logic by using Citrus test actions that perform once the scenario is triggered by incoming requests.

This way each incoming request is processed and a proper response message is provided to the client. You can define default and fallback scenarios for each simulator.

In addition to that the simulator provides a user interface where executed scenarios state success or failure. Also you can trigger scenarios manually.

Simulator application

The simulator is based on Spring boot. This means we have a main class that loads the Spring boot application.

```
import com.consol.citrus.simulator.annotation.EnableRest;
import com.consol.citrus.simulator.annotation.SimulatorApplication;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

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

This class is the main entrance for all configuration and customization statements. First of all the class is annotated as **@SpringBootApplication** and **@SimulatorApplication**. This enables auto configuration for the simulator application. In addition to that we enable different aspects of the simulator by using further annotations provided. In the sample above we use the **@EnableRest** annotation for Http REST endpoint support.

There are multiple annotations available for different transport endpoints:

- **@EnableRest** Enables Http REST support
- **@EnableWs** Enables SOAP web services support
- **@EnableJms** Enables JMS support
- **@EnableEndpointComponent** Enables generic endpoint component support

Properties

The simulator is capable of loading configuration from system properties and property files. There are several properties that you can use in order to customize the simulator behavior. These properties are:

- **citrus.simulator.configuration.class** Java configuration class that is automatically loaded. (default is com.consol.citrus.simulator.SimulatorConfig)
- **citrus.simulator.template.path** Default path to message payload template files.
- **citrus.simulator.default.scenario** Default scenario name.
- **citrus.simulator.timeout** Timeout when waiting for inbound messages.
- **citrus.simulator.template.validation** Enable/disable schema validation.

You can set these properties as system properties or you can add a property file in following location:

- **META-INF/citrus-simulator.properties**

The simulator will automatically load these properties during startup.

Spring bean configuration

Citrus works with the Spring framework and the simulator is a Spring boot application. Therefore the configuration is done by adding and overwriting Spring beans in the application context. The simulator automatically loads Spring beans defined in following locations:

- **META-INF/citrus-simulator.xml** Xml Spring bean configuration file.
- **com.consol.citrus.simulator.SimulatorConfig** Java configuration class. You can

customize this class by defining the property **citrus.simulator.configuration.class**

All beans defined in there get automatically loaded to the simulator Spring application context.

Mapping key extractor

The mapping key extractor implementation decides how to map incoming request message to simulator scenarios. Each incoming request triggers a predefined scenario that generates the response message for the calling client. The simulator identifies the scenario based on a mapping key that is extracted from the incoming request.

There are multiple ways to identify the simulator scenario from incoming request messages:

- Message-Type: Each request message type (XML root QName) results in a separate simulator scenario
- REST request mappings: Identifies the scenario based on Http method and resource path on server
- SOAP Action: Each SOAP action value defines a simulator scenario
- Message Header: Any SOAP or Http message header value specifies a new simulator scenario
- XPath payload: An XPath expression is evaluated on the message payload to identify the scenario
- Json payload: An JsonPath expression is evaluated on the message payload to identify the scenario

Once the simulator scenario is identified with the respective mapping key the scenario get loaded and executed. All scenarios perform Citrus test logic in order to provide a proper response messages as a result. This way the simulator is able to perform complex response generating logic with dynamic values and so on.

The mentioned mapping key extraction strategies are implemented in these classes:

- **AnnotationRequestMappingKeyExtractor** Evaluates REST request mappings
- **SoapActionMappingKeyExtractor** Evaluates the SOAP action header
- **HeaderMappingKeyExtractor** Evaluates any message header
- **XPathPayloadMappingKeyExtractor** Evaluates a XPath expression on the message payload
- **JsonPayloadMappingKeyExtractor** Evaluates a JsonPath expression on the message payload

Of course you can also implement a custom mapping key extractor. Just implement the interface methods of that API and add the implementation to the simulator configuration as described later on in this document.

Default behavior

The default mapping key logic extracts the message type of incoming requests. This is done by evaluating a Xpath expression on the request payload that uses the root element of the message as the mapping key. Each message type gets its own simulator scenario.

Let's demonstrate that in a simple example. We know three different message types named **successMessage**, **warningMessage** and **errorMessage**. We create a simulator scenario for each of these message types with respective naming. Given the following incoming requests the simulator will pick the matching scenario for execution.

```
<successMessage>
    <text>This is a success message</text>
</successMessage>

<warningMessage>
    <text>This is a warning message</text>
</warningMessage>

<errorMessage>
    <text>This is a error message</text>
</errorMessage>
```

The simulator evaluates the root element name and maps the requests to the matching scenario. Each scenario implements different response generating logic so the simulator is able to respond to a **successMessage** in a different way than for **errorMessage** types.

Configuration

You can change the mapping key behavior by overwriting te default mapping key extractor in your simulator.

```
@Component
public class SimulatorAdapter extends SimulatorRestAdapter {
    @Override
    public MappingKeyExtractor mappingKeyExtractor() {
        HeaderMappingKeyExtractor mappingKeyExtractor = new HeaderMappingKeyExtractor();
        mappingKeyExtractor.setHeaderName("X-simulator-scenario");
        return mappingKeyExtractor;
    }
}
```

With the configuration above we use the *HeaderMappingKeyExtractor* implementation so the header name **X-simulator-scenario** gets evaluated for each incoming request message. Depending on that header value the matching scenario is executed as a result. The mapping key extractor is just a bean in the Spring application context. There is a default implementation but you can overwrite this behavior very easy in the simulator adapter configuration. Read more about how to add simulator adapter configuration classes in configuration chapters rest-config, ws-config or jms-config.

Simulator scenarios

The simulator provides the response generating logic by defining one to many scenarios that get executed based on the incoming request. The different scenarios on the simulator describe different response messages and stand for individual simulation logic. Each scenario is capable of receiving and validating the incoming request message. Based on that the scenario is in charge of constructing a proper response message.

First of all the scenario gets a name under which mapping strategies can identify the scenario. This name is very important when it comes to mapping incoming requests to scenarios. Besides that the scenario is a normal Java class that implements following interface *SimulatorScenario*

```
package com.consol.citrus.simulator.scenario;

public interface SimulatorScenario {
    ScenarioEndpoint scenario();
}
```

The simulator scenario has to provide message receive and send operations for the simulator endpoint. Fortunately there are default implementations that you can inherit from:

- **SimulatorRestScenario**
- **SimulatorWebServiceScenario**
- **SimulatorJmsScenario**
- **SimulatorEndpointScenario**

Each of these base scenario classes provide comfortable access to the endpoint so we can just add receive and send logic for generating the response message.

```
@Scenario("Hello")
public class HelloScenario extends SimulatorRestScenario {

    @Override
    protected void configure() {
        scenario()
            .receive()
            .payload("<Hello xmlns=\"http://citrusframework.org/schemas/hello\">" +
                        "Say Hello!" +
                    "</Hello>");

        scenario()
            .send()
            .payload("<HelloResponse xmlns=\"http://citrusframework.org/schemas/hello\">" +
                        "Hi there!" +
                    "</HelloResponse>");
    }
}
```

The scenario above is annotated with **@Scenario** for defining a scenario name. There is one single **configure** method to be implemented. We can use Citrus Java DSL methods in the method body in order to receive the incoming request and send back a proper response message. Of course we can use the full Citrus power here in order to construct different message payloads such as XML, JSON, PLAINTEXT and so on.

So we could also extract dynamic values from the request in order to reuse those in our response message:

```
@Scenario("Hello")
public class HelloScenario extends SimulatorRestScenario {

    @Override
    protected void configure() {
        scenario()
            .receive()
            .payload("<Hello xmlns=\"http://citrusframework.org/schemas/hello\">" +
                        "<user>@ignore@</user>" +
                    "</Hello>")
            .extractFromPayload("/Hello/user", "userName");

        scenario()
            .send()
            .payload("<HelloResponse xmlns=\"http://citrusframework.org/schemas/hello\">" +
                        "<text>Hi there ${userName}!</text>" +
                    "</HelloResponse>");
    }
}
```

In the receive operation the user name value is extracted to a test variable **${userName}**. In the response we are able to use this variable in order to greet the user by name. This way we can use the Citrus test power for generating dynamic response messages. Of course this mechanism works for XML, Json and Plaintext payloads.

Now you are ready to write different scenarios that generate different response messages for the calling client.

REST support

The simulator is able to handle REST API calls such as Http GET, POST, PUT, DELETE and so on. The simulator defines a special REST enabling annotation that we can use on the application class:

```
import com.consol.citrus.simulator.annotation.EnableRest;
import com.consol.citrus.simulator.annotation.SimulatorApplication;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

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

The **@EnableRest** annotation performs some auto configuration steps and loads required beans for the Spring application context in the Spring boot application.

After that we are ready to handle incoming REST API calls on the simulator. The simulator REST support provides a base REST scenario class.

```
@Scenario("Hello")
@RequestMapping(value = "/services/rest/simulator/hello", method = RequestMethod.POST)
public class HelloScenario extends SimulatorRestScenario {

    @Override
    protected void configure() {
        scenario()
            .receive()
            .payload("<Hello xmlns=\"http://citrusframework.org/schemas/hello\">" +
                        "Say Hello!" +
                    "</Hello>");

        scenario()
            .send()
            .payload("<HelloResponse xmlns=\"http://citrusframework.org/schemas/hello\">" +
                        "Hi there!" +
                    "</HelloResponse>");
    }
}
```

Now the base REST scenario provides the basic receive and send operations for the scenario. Also note that we can use **@RequestMapping** annotations for advanced REST request mapping.

Besides that you can have several REST related configuration options as described in the following sections.

REST configuration

Once the REST support is enabled on the simulator we have different configuration options. The most comfortable way is to add a **SimulatorRestAdapter** implementation to the classpath. The adapter provides several configuration methods.

```
public abstract class SimulatorRestAdapter implements SimulatorRestConfigurer {

    @Override
    public MappingKeyExtractor mappingKeyExtractor() {
        return new AnnotationRequestMappingKeyExtractor();
    }

    @Override
    public HandlerInterceptor[] interceptors() {
        return new HandlerInterceptor[] { new LoggingHandlerInterceptor() };
    }

    @Override
    public String urlMapping() {
        return "/services/rest/**";
    }
}
```

The adapter defines methods that configure the simulator REST handling. For instance we can add another mapping key extractor implementation or add handler interceptors to the REST API call handling.

Note The REST support is using a different default mapping key extractor. The **AnnotationRequestMappingKeyExtractor** is active by default and enables **@RequestMapping** related mapping on scenario classes. Read more about that in rest-request-mapping.

The **urlMapping** defines how clients can access the simulator REST API. Assuming the Spring boot simulator application is running on port 8080 the REST API would be accessible on this URI:

```
http://localhhost:8080/services/rest/*
```

The clients can send GET, POST, DELETE and other calls to that endpoint URI then. The simulator will respond with respective responses based on the called scenario.

You can simply extend the adapter in a custom class for adding customizations.

```
@Component
public class MySimulatorRestAdapter extends SimulatorRestAdapter {

    @Override
    public String urlMapping() {
        return "/my-rest-service/**";
    }
}
```

As you can see the class is annotated with **@Component** annotation. This is because the adapter should be recognized by Spring in order to overwrite the default REST adapter behavior. The custom adapter just overwrites the **urlMapping** method so the REST simulator API will be accessible for clients under this endpoint URI:

```
http://localhhost:8080/my-rest-service/*
```

This is the simplest way to customize the simulator REST support. We can also use the adapter extension directly on the Spring boot main application class:

```
import com.consol.citrus.simulator.annotation.EnableRest;
import com.consol.citrus.simulator.annotation.SimulatorRestAdapter;
import com.consol.citrus.simulator.annotation.SimulatorApplication;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
@SimulatorApplication
@EnableRest
public class Simulator extends SimulatorRestAdapter {

    @Override
    public String urlMapping() {
        return "/my-rest-service/**";
    }

    @Override
    public MappingKeyExtractor mappingKeyExtractor() {
        HeaderMappingKeyExtractor mappingKeyExtractor = new HeaderMappingKeyExtractor();
        mappingKeyExtractor.setHeaderName("X-simulator-scenario");
        return mappingKeyExtractor;
    }

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

So we have **@EnableRest** and REST adapter customizations combined on one single class.

Advanced customizations

For a more advanced configuration option we can extend the **SimulatorRestSupport** implementation.

```
import com.consol.citrus.simulator.annotation.EnableRest;
import com.consol.citrus.simulator.annotation.SimulatorRestSupport;
import com.consol.citrus.simulator.annotation.SimulatorApplication;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
@SimulatorApplication
public class Simulator extends SimulatorRestSupport {

    @Override
    protected String getUrlMapping() {
        return "/my-rest-service/**";
    }

    @Override
    public FilterRegistrationBean requestCachingFilter() {
        FilterRegistrationBean filterRegistrationBean = new FilterRegistrationBean(new Reques

        String urlMapping = getUrlMapping();
        if (urlMapping.endsWith("**")) {
            urlMapping = urlMapping.substring(0, urlMapping.length() - 1);
        }
        filterRegistrationBean.setUrlPatterns(Collections.singleton(urlMapping));
        return filterRegistrationBean;
    }

    @Override
    public HandlerMapping handlerMapping(ApplicationContext applicationContext) {
        SimpleUrlHandlerMapping handlerMapping = new SimpleUrlHandlerMapping();
        handlerMapping.setOrder(Ordered.HIGHEST_PRECEDENCE);
        handlerMapping.setAlwaysUseFullPath(true);

        Map<String, Object> mappings = new HashMap<>();
        mappings.put(getUrlMapping(), getRestController(applicationContext));

        handlerMapping.setUrlMap(mappings);
        handlerMapping.setInterceptors(interceptors());

        return handlerMapping;
    }

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

With that configuration option we can overwrite REST support auto configuration features on the simulator such as the **requestCachingFilter** or the **handlerMapping**. We can not use the **@EnableRest** auto configuration annotation then. Instead we extend the **SimulatorRestSupport** implementation directly.

REST request mapping

Usually we define simulator scenarios and map them to incoming requests by their names. When using REST support on the simulator we can also use request mapping annotations on scenarios in order to map incoming requests.

This looks like follows:

```
@Scenario("Hello")
@RequestMapping(value = "/services/rest/simulator/hello", method = RequestMethod.POST)
public class HelloScenario extends SimulatorRestScenario {

    @Override
    protected void configure() {
        scenario()
            .receive()
            .payload("<Hello xmlns=\"http://citrusframework.org/schemas/hello\">" +
                        "Say Hello!" +
                    "</Hello>");

        scenario()
            .send()
            .payload("<HelloResponse xmlns=\"http://citrusframework.org/schemas/hello\">" +
                        "Hi there!" +
                    "</HelloResponse>");
    }
}
```

As you can see the example above uses **@RequestMapping** annotation in addition to the **@Scenario** annotation. All requests on the request path **/services/rest/simulator/hello** of method **POST** will be mapped to the scenario. With this strategy the simulator is able to map requests based on methods, request paths and parameters.

The mapping strategy requires a special mapping key extractor implementation that automatically scans for scenarios with **@RequestMapping** annotations. The **AnnotationRequestMappingKeyExtractor** is active by default so in case you need to apply different mapping strategies you must overwrite the mapping key extractor in configuration adapter.

Web Service support

The simulator is able to handle SOAP Web Service calls as a server. The simulator defines a special SOAP enabling annotation that we can use on the application class:

```
import com.consol.citrus.simulator.annotation.EnableWebService;
import com.consol.citrus.simulator.annotation.SimulatorApplication;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

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

The **@EnableWebService** annotation performs some auto configuration steps and loads required beans for the Spring application context in the Spring boot application.

After that we are ready to handle incoming SOAP Web Service calls on the simulator. We can use the default scenario base class for SOAP Web Services.

```
@Scenario("Hello")
public class HelloScenario extends SimulatorWebServiceScenario {

    @Override
    protected void configure() {
        scenario()
            .receive()
            .payload("<Hello xmlns=\"http://citrusframework.org/schemas/hello\">" +
                        "Say Hello!" +
                    "</Hello>")
            .header(SoapMessageHeaders.SOAP_ACTION, "Hello");

        scenario()
            .send()
            .payload("<HelloResponse xmlns=\"http://citrusframework.org/schemas/hello\">" +
                        "Hi there!" +
                    "</HelloResponse>");
    }
}
```

The **SimulatorWebServiceScenario** automatically handles the SOAP envelope so we do not have to deal with that in the scenario receive and send operations. Also the scenario receive operation has access to the SOAP action of the incoming request call. Besides that we can also return a SOAP fault message as scenario outcome.

Let's move on with having a look at the SOAP related configuration options as described in the following sections.

Web Service configuration

Once the SOAP support is enabled on the simulator we have different configuration options. The most comfortable way is to add a **SimulatorWebServiceAdapter** implementation to the classpath. The adapter provides several configuration methods.

```
public abstract class SimulatorWebServiceAdapter implements SimulatorWebServiceConfigurer {
    @Override
    public String servletMapping() {
        return "/services/ws/*";
    }

    @Override
    public MappingKeyExtractor mappingKeyExtractor() {
        return new XPathPayloadMappingKeyExtractor();
    }

    @Override
    public EndpointInterceptor[] interceptors() {
        return new EndpointInterceptor[] { new LoggingEndpointInterceptor() };
    }
}
```

The adapter defines methods that configure the simulator SOAP message handling. For instance we can add another mapping key extractor implementation or add endpoint interceptors to the SOAP service call handling.

The **servletMapping** defines how clients can access the simulator SOAP service. Assuming the Spring boot simulator application is running on port 8080 the SOAP service would be accessible on this URI:

```
http://localhhost:8080/services/ws/*
```

The clients can send SOAP calls to that endpoint URI then. The simulator will respond with respective SOAP responses based on the called scenario.

You can simply extend the adapter in a custom class for adding customizations.

```
@Component
public class MySimulatorWebServiceAdapter extends SimulatorWebServiceAdapter {

    @Override
    public String servletMapping() {
        return "/my-soap-service/**";
    }
}
```

As you can see the class is annotated with **@Component** annotation. This is because the adapter should be recognized by Spring in order to overwrite the default SOAP adapter behavior. The custom adapter just overwrites the **servletMapping** method so the SOAP simulator API will be accessible for clients under this endpoint URI:

```
http://localhhost:8080/my-soap-service/*
```

This is the simplest way to customize the simulator SOAP support. We can also use the adapter extension directly on the Spring boot main application class:

```
import com.consol.citrus.simulator.annotation.EnableWebService;
import com.consol.citrus.simulator.annotation.SimulatorWebServiceAdapter;
import com.consol.citrus.simulator.annotation.SimulatorApplication;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
@SimulatorApplication
@EnableWebService
public class Simulator extends SimulatorWebServiceAdapter {

    @Override
    public String servletMapping() {
        return "/my-soap-service/**";
    }

    @Override
    public MappingKeyExtractor mappingKeyExtractor() {
        return new SoapActionMappingKeyExtractor();
    }

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

So we have **@EnableWebService** and SOAP adapter customizations combined on one single class.

Advanced customizations

For a more advanced configuration option we can extend the **SimulatorWebServiceSupport** implementation.

```
import com.consol.citrus.simulator.annotation.EnableWebService;
import com.consol.citrus.simulator.annotation.SimulatorWebServiceSupport;
import com.consol.citrus.simulator.annotation.SimulatorApplication;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
@SimulatorApplication
public class Simulator extends SimulatorWebServiceSupport {

    @Override
    protected String getServletMapping() {
        return "/my-soap-service/**";
    }

    @Bean
    public ServletRegistrationBean messageDispatcherServlet(ApplicationContext applicationCon
        MessageDispatcherServlet servlet = new MessageDispatcherServlet();
        servlet.setApplicationContext(applicationContext);
        servlet.setTransformWsdlLocations(true);
        return new ServletRegistrationBean(servlet, getDispatcherServletMapping());
    }

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

With that configuration option we can overwrite SOAP support auto configuration features on the simulator such as the **messageDispatcherServlet**. We can not use the **@EnableWebService** auto configuration annotation then. Instead we extend the **SimulatorWebServiceSupport** implementation directly.

SOAP faults

The simulator is in charge of sending proper response messages to the calling client. When using SOAP we might also want to send back a SOAP fault message. Therefore the default Web Service scenario implementation also provides fault responses as scenario result.

```
@Scenario("GoodNight")
public class GoodNightScenario extends SimulatorWebServiceScenario {

    @Override
    protected void configure() {
        scenario()
            .receive()
            .payload("<GoodNight xmlns=\"http://citrusframework.org/schemas/hello\">" +
                        "Go to sleep!" +
                    "</GoodNight>")
            .header(SoapMessageHeaders.SOAP_ACTION, "GoodNight");

        scenario()
            .sendFault()
            .faultCode("{http://citrusframework.org}CITRUS:SIM-1001")
            .faultString("No sleep for me!");
    }
}
```

The example above shows a simple fault generating SOAP scenario. The base class **SimulatorWebServiceScenario** provides the **sendFault()** method in order to create proper SOAP fault messages. The simulator automatically add SOAP envelope and SOAP fault message details for you. So we can decide wheather to provide a success response or SOAP fault.

JMS support

The simulator is able to receive messages from any JMS message broker. The simulator will constantly poll a JMS destination (queue or topic) for incoming request messages. When the queue is of synchronous nature the simulator is able to send synchronous response messages. The simulator defines a special JMS enabling annotation that we can use on the application class:

```
import com.consol.citrus.simulator.annotation.EnableJms;
import com.consol.citrus.simulator.annotation.SimulatorApplication;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

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

The **@EnableJms** annotation performs some auto configuration steps and loads required beans for the Spring application context in the Spring boot application.

With that piece of configuration we are ready to handle incoming JMS messages on the simulator. Of course we need a JMS connection factory and other JMS related configuration options as described in the following sections.

JMS configuration

Once the JMS support is enabled on the simulator we have different configuration options. The most comfortable way is to add a **SimulatorJmsAdapter** implementation to the classpath. The adapter provides several configuration methods.

```
public abstract class SimulatorJmsAdapter implements SimulatorJmsConfigurer {
    @Override
    public ConnectionFactory connectionFactory() {
        return new SingleConnectionFactory();
    }

    @Override
    public String destinationName() {
        return System.getProperty("citrus.simulator.jms.destination", "Citrus.Simulator.Inbou
    }

    @Override
    public boolean useSoapEnvelope() {
        return false;
    }

    @Override
    public MappingKeyExtractor mappingKeyExtractor() {
        return new XPathPayloadMappingKeyExtractor();
    }
}
```

The adapter defines methods that configure the simulator JMS handling. For instance we can add another mapping key extractor implementation or enable automatic SOAP envelope handling.

The **destinationName** defines the incoming JMS destination to poll. The **connectionFactory** is mandatory in order to connect to a JMS message broker.

You can simply extend the adapter in a custom class for adding customizations.

```
@Component
public class MySimulatorJmsAdapter extends SimulatorJmsAdapter {

    @Override
    public String destinationName() {
        return "JMS.Queue.simulator.inbound";
    }

    @Override
    public ConnectionFactory connectionFactory() {
        return new ActiveMQConnectionFactory("tcp://localhost:61616");
    }
}
```

As you can see the class is annotated with **@Component** annotation. This is because the adapter should be recognized by Spring in order to overwrite the default JMS adapter behavior. The custom adapter just overwrites the **connectionFactory** and **destinationName** methods so the JMS simulator will connect to the ActiveMQ message broker and listen for incoming requests on that queue.

This is the simplest way to customize the simulator JMS support. We can also use the adapter extension directly on the Spring boot main application class:

```
import com.consol.citrus.simulator.annotation.EnableJms;
import com.consol.citrus.simulator.annotation.SimulatorJmsAdapter;
import com.consol.citrus.simulator.annotation.SimulatorApplication;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
@SimulatorApplication
@EnableJms
public class Simulator extends SimulatorJmsAdapter {

    @Override
    public String destinationName() {
        return "JMS.Queue.simulator.inbound";
    }

    @Override
    public ConnectionFactory connectionFactory() {
        return new ActiveMQConnectionFactory("tcp://localhost:61616");
    }

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

So we have **@EnableJms** and JMS adapter customizations combined on one single class.

Endpoint component support

We have seen how the simulator handles different transports such as Http REST, SOAP web services and JMS. Now the simulator is also able to handle other message transports such as mail communication, JMX mbean server, RMI invocations and much more. The simulator is able to deal with any kind of endpoint component that is supported in Citrus framework.

The generic endpoint support is added with **@EnableEndpointComponent** annotation.

```
import com.consol.citrus.simulator.annotation.EnableEndpointComponent;
import com.consol.citrus.simulator.annotation.SimulatorApplication;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

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

The **@EnableEndpointComponent** annotation performs some auto configuration steps and loads required beans for the Spring application context in the Spring boot application. Once we use that feature we can have any Citrus endpoint component as inbound source for simulator scenarios. This means we can have a mail server or a RMI server that is simulated with proper response messages.

As we are using generic endpoint components as inbound source we need to set the endpoint in the configuration. Read about that in the following sections.

Endpoint component configuration

When using generic Citrus endpoints as simulator inbound source we need to configure those endpoint components. The most comfortable way is to add a **SimulatorEndpointComponentAdapter** implementation to the classpath. The adapter provides several configuration methods.

```
public abstract class SimulatorEndpointComponentAdapter implements SimulatorEndpointComponent


    @Override
    public abstract Endpoint endpoint(ApplicationContext applicationContext);

    @Override
    public boolean useSoapEnvelope() {
        return false;
    }

    @Override
    public MappingKeyExtractor mappingKeyExtractor() {
        return new XPathPayloadMappingKeyExtractor();
    }
}
```

The adapter defines methods that configure the endpoint component used as inbound source. As usual we can set the mapping key extractor implementation or add automatic SOAP envelope support.

More importantly we need to define an inbound endpoint that is used as source for scenarios. Let's have a simple endpoint component adapter example.

```
import com.consol.citrus.simulator.annotation.EnableEndpointComponent;
import com.consol.citrus.simulator.annotation.SimulatorEndpointComponentAdapter;
import com.consol.citrus.simulator.annotation.SimulatorApplication;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
@SimulatorApplication
@EnableEndpointComponent
public class Simulator extends SimulatorEndpointComponentAdapter {

    public static void main(String[] args) {
        SpringApplication.run(Simulator.class, args);
    }

    @Override
    public Endpoint endpoint(ApplicationContext applicationContext) {
        MailServer mailServer = new MailServer();
        mailServer.setPort(2222);
        mailServer.setAutoStart(true);

        return mailServer;
    }

    @Override
    public MappingKeyExtractor mappingKeyExtractor() {
        XPathPayloadMappingKeyExtractor mappingKeyExtractor = new XPathPayloadMappingKeyExtra
        NamespaceContextBuilder namespaceContextBuilder = new NamespaceContextBuilder();
        namespaceContextBuilder.getNamespaceMappings().put("mail", "http://www.citrusframewor
        mappingKeyExtractor.setNamespaceContextBuilder(namespaceContextBuilder);
        mappingKeyExtractor.setXpathExpression("/mail:mail-message/mail:subject");
        return mappingKeyExtractor;
    }
}
```

The custom adapter defines a Citrus mail server endpoint that should be used as inbound source. Any mail message that arrives at this mail server component will trigger a new simulator scenario then. Also we overwrite the mapping key extractor implementation so that the mail subject evaluates to the scenario that should be executed.

This configuration would lead us to a mail server that responds to incoming mail messages base on the mail subject. So we can have several simulator scenarios for different mail messages.

```
@Scenario("Hello")
public class HelloScenario extends SimulatorEndpointScenario {

    @Override
    protected void configure() {
        scenario()
            .receive()
            .payload("<mail-message xmlns=\"http://www.citrusframework.org/schema/mail/messag
                        "<from>user@citrusframework.org</from>" +
                        "<to>citrus@citrusframework.org</to>" +
                        "<cc></cc>" +
                        "<bcc></bcc>" +
                        "<subject>Hello</subject>" +
                        "<body>" +
                            "<contentType>text/plain; charset=utf-8</contentType>" +
                            "<content>Say Hello!</content>" +
                        "</body>" +
                    "</mail-message>");

        scenario()
            .send()
            .payload("<mail-response xmlns=\"http://www.citrusframework.org/schema/mail/messa
                        "<code>250</code>" +
                        "<message>OK</message>" +
                    "</mail-response>");
    }
}
```

The scenario implementation above uses the base class **SimulatorEndpointScenario** and listens for mail messages of subject **Hello**. The mail XML marshalling is automatically done by Citrus. This is the usual way how the Citrus mail component handles mail messages and responses. That means we can use the default Citrus features in our simulator, too. The scenario sends back a positive mail response to the calling client.

This is how we can use any Citrus endpoint component as simulator inbound source. This gives us the opportunity to support a huge set of message transports and message types in our simulator applications. Each incoming request on the endpoint component triggers a new simulator scenario.

Admin user interface

The simulator application is started as a Spring boot web application. Users can access an administrative user interface for reviewing the simulator status and executed scenarios and their outcome.

Open your browser and point to the simulator UI:

```
http://localhost:8080
```

You will see a simple web application that gives you information about the simulator.

Further reading

- Citrus home Citrus homepage with samples and further information.
- User guide gives you a detailed description of all Citrus features.
- Sample projects demonstrate typical simulator scenarios with different message transports
- Contributing explains how you can contribute to this project. Pull requests are highly appreciated!