

Dialogue SMS REST API Toolkit

Java Quick Start Guide

Date: March 2013

Version: 1.8.3

CONTENT

1	Sending Messages.....	3
1.1	Creating the Project	3
1.2	Importing the Package	6
1.3	Instantiating SendSmsClient	8
1.4	Sending a Message	8
1.4.1	Complete Code Sample.....	9
1.5	Processing the Response	10
1.5.1	Complete Code Sample.....	10
1.6	Multiple Messages	11
1.7	Multiple Recipients.....	11
1.8	Additional Parameters.....	12
1.8.1	ConcatenationLimit	12
1.8.2	UserTag.....	12
1.8.3	ScheduleFor	13
1.8.4	ValidityPeriod.....	13
1.8.5	ConfirmDelivery, ReplyPath, UserKey.....	13
1.8.6	SessionId, SessionReplyPath.....	14
1.9	Custom Parameters	14
1.10	Transports.....	15
1.10.1	Dependencies.....	15
1.11	RestOperations.....	16
2	Receiving Reports and Replies.....	18
2.1	Integrating the Server.....	18
2.2	Creating the Project	20
2.3	Importing the Package	23
2.4	Processing Message Reports	25
2.5	Processing Message Replies	27
3	Status Codes	30

1 Sending Messages

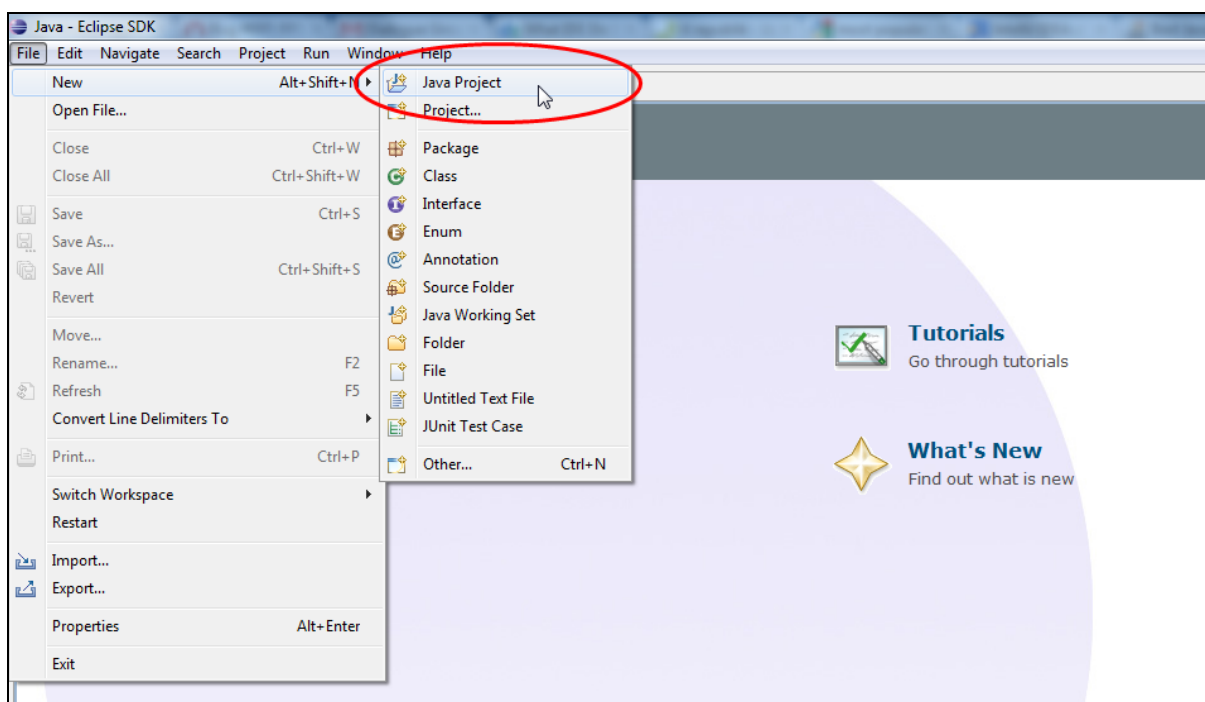
1.1 Creating the Project

This Java Quick Start Guide is aimed at novice Java developers; however you should know how to install and use Java on your system and be able to use an IDE (Integrated Developer Environment).

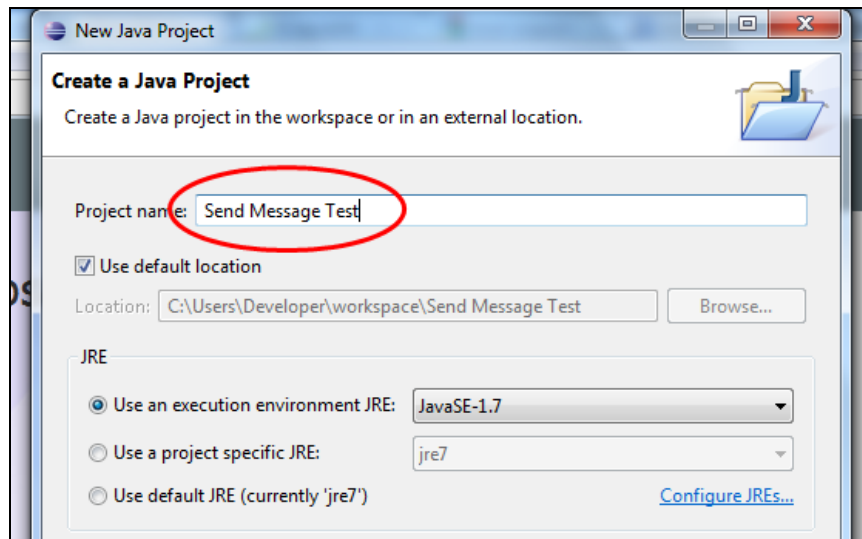
This guide uses a Java IDE called “Eclipse IDE for Java EE Developers”, which can be downloaded free of charge at <http://www.eclipse.org/downloads/>. You will also need the Java Runtime Environment (JRE) installed or, if you plan to implement a handler to receive reports and replies (as per section 2), the Java Development Kit (JDK). The toolkit library supports Java 1.5 or higher. JRE or JDK can be downloaded here: <http://www.oracle.com/technetwork/java/javase/downloads/>.

If you’re using another development environment you should be able perform the same tasks, although the steps required to achieve these tasks may be different.

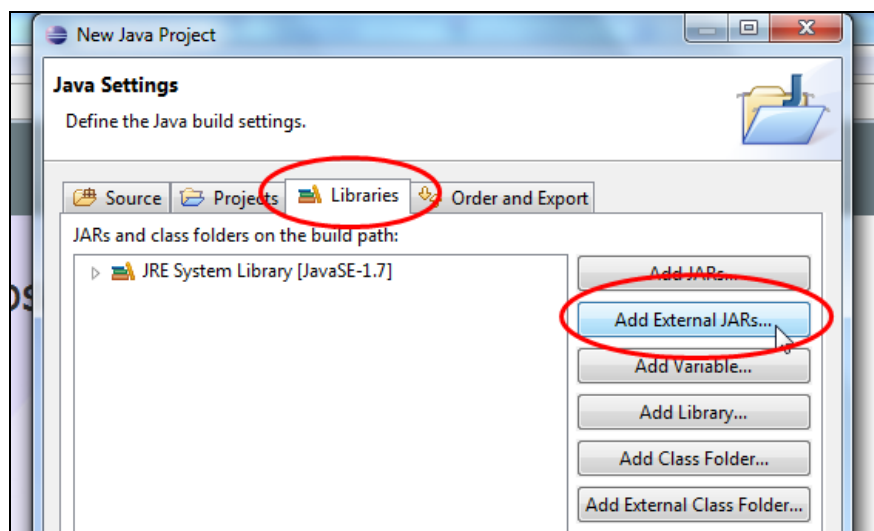
Using Eclipse create a new **Java Project**.



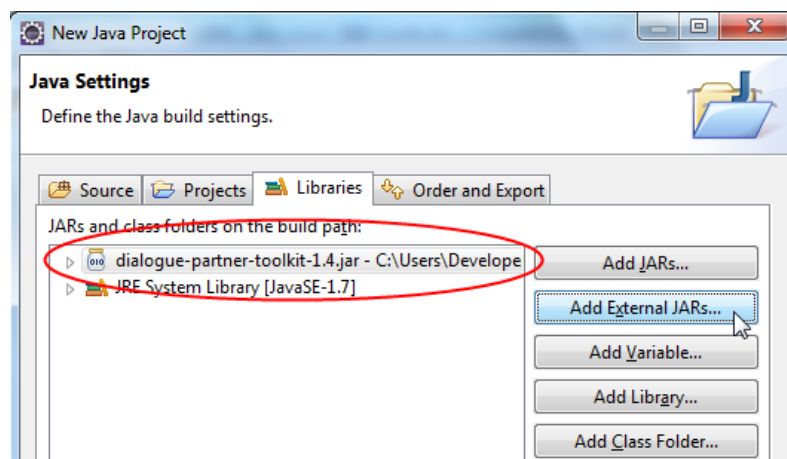
Give your project a name, optionally specify an alternate project location and select a JRE.



Click **Next >** and select the **Libraries** tab. Click the **Add External JARs...** button and navigate to the folder where you extracted **dialogue-partner-toolkit-1.4.2.zip**, containing **dialogue-partner-toolkit-1.4.2.jar** and select this file.

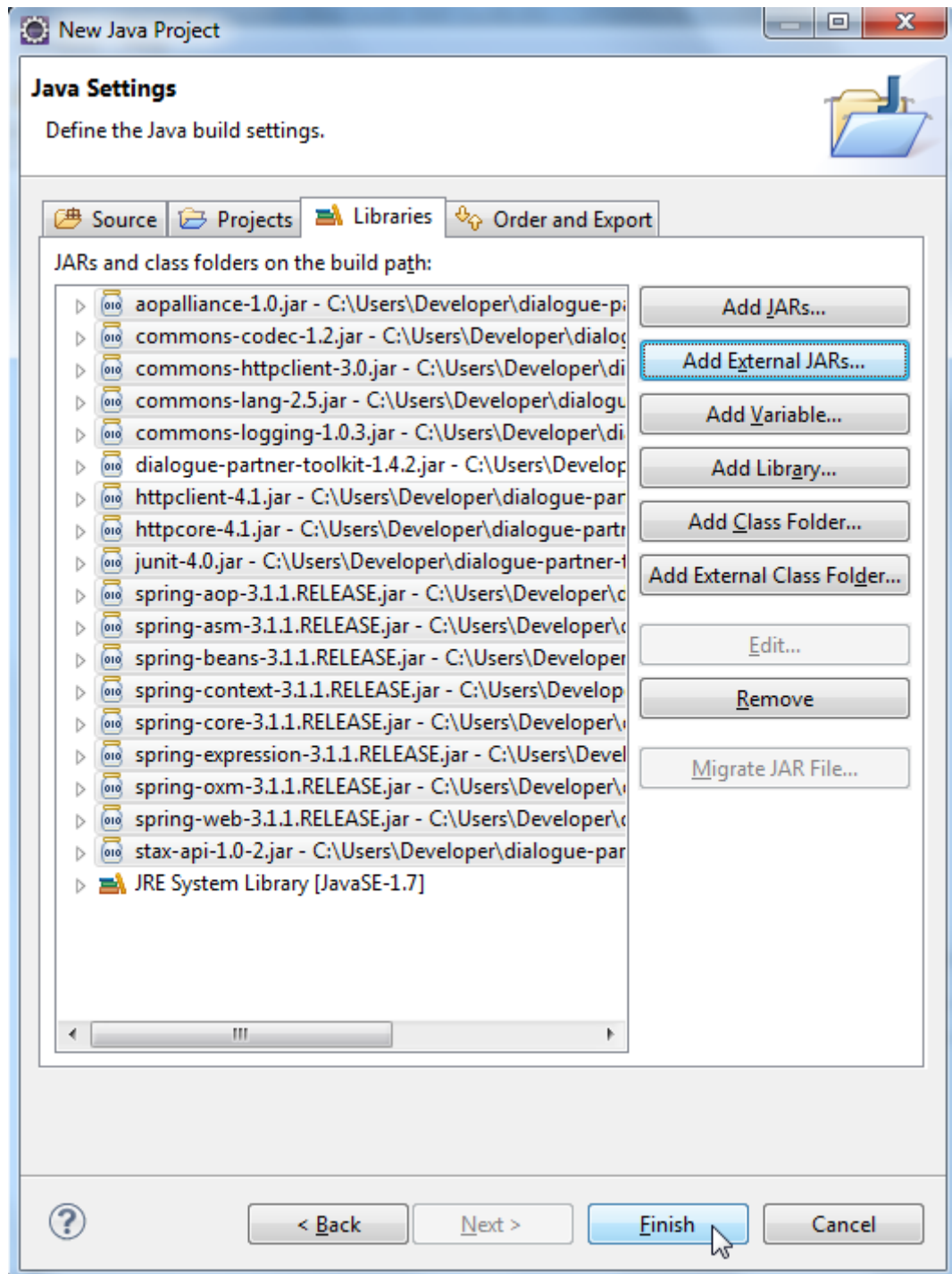


Confirm the file selector. The toolkit library is now listed.



The toolkit library is based on a number of standard libraries such as Spring RestTemplate for invoking REST web services and/or Apache client libraries for making HTTP requests.

To add these dependencies press **Add External JARs...** once more and navigate to the **lib** directory inside the extracted folder. Select all *.jar files (CTRL + A) and confirm the file selector.

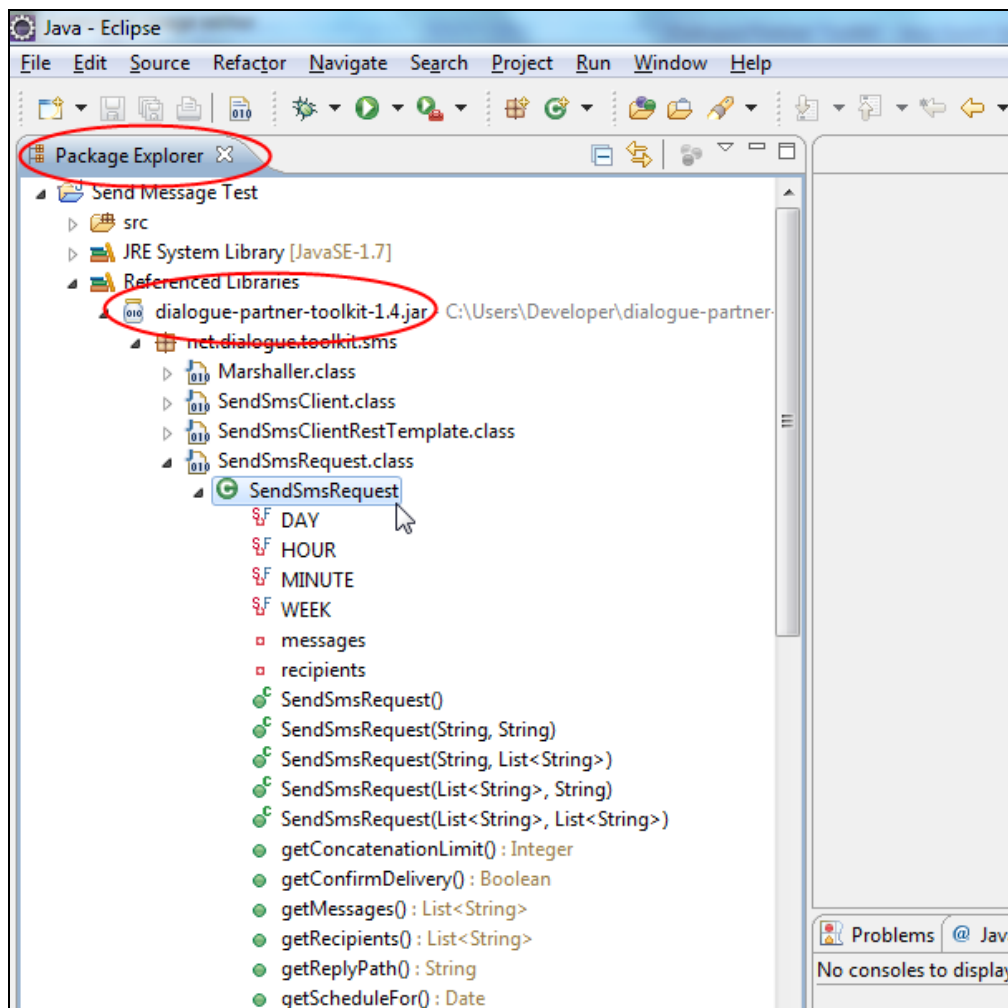


The dependencies are now listed in the dialog. Click **Finish** and wait until the project is created.

Not all libraries will be required at runtime. Depending on which transport you prefer to use (default, Apache Commons HttpClient, Apache HttpComponents Client or your own) you may not need commons-httpclient-3.0.jar or httpclient-4.1.jar and httpcore-4.1.jar.

If you use Java 1.6 or higher you do not need to include stax-api-1.0.2.jar, but you require this file on Java 1.5 since it does not bundle javax.xml.stream.XMLStreamWriter.

You can explore the toolkit library using the **Package Explorer** view by expanding Referenced Libraries > dialogue-partner-toolkit-1.4.2.jar > net.dialogue.toolkit.sms. Take a look at the SendSmsRequest object.



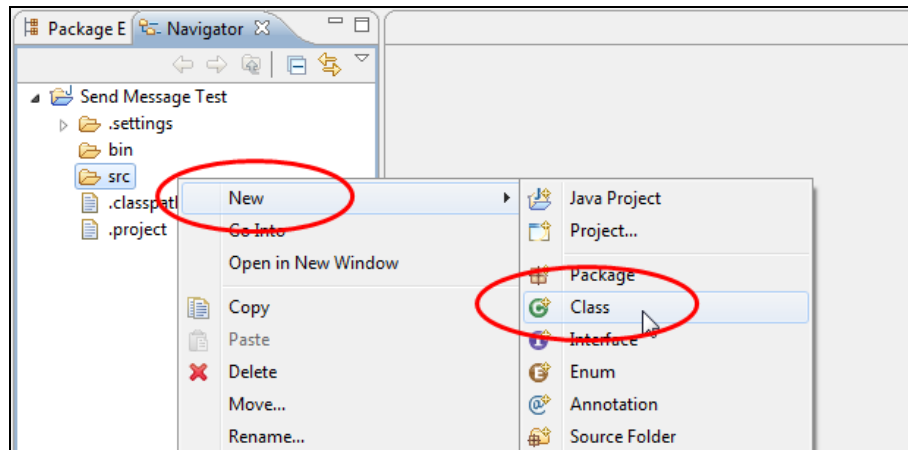
1.2 Importing the Package

The package required for sending messages is `net.dialogue.toolkit.sms`. Here are the objects of interest for sending messages:

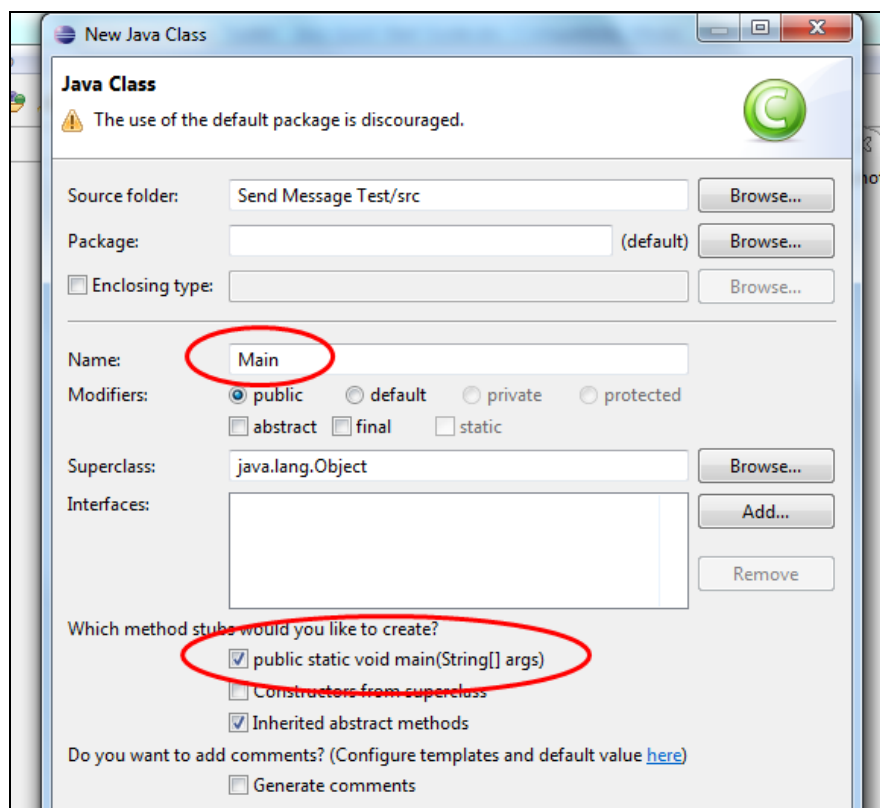
Object	Description
<code>SendSmsClient</code>	Allows sending of messages
<code>SendSmsClient.Builder</code>	Builder/helper for creating instances of <code>SendSmsClient</code> .

SendSmsRequest	Input object (message parameters)
SendSmsResponse	Output object (submission results)
Sms	Contained once or more inside SendSmsResponse

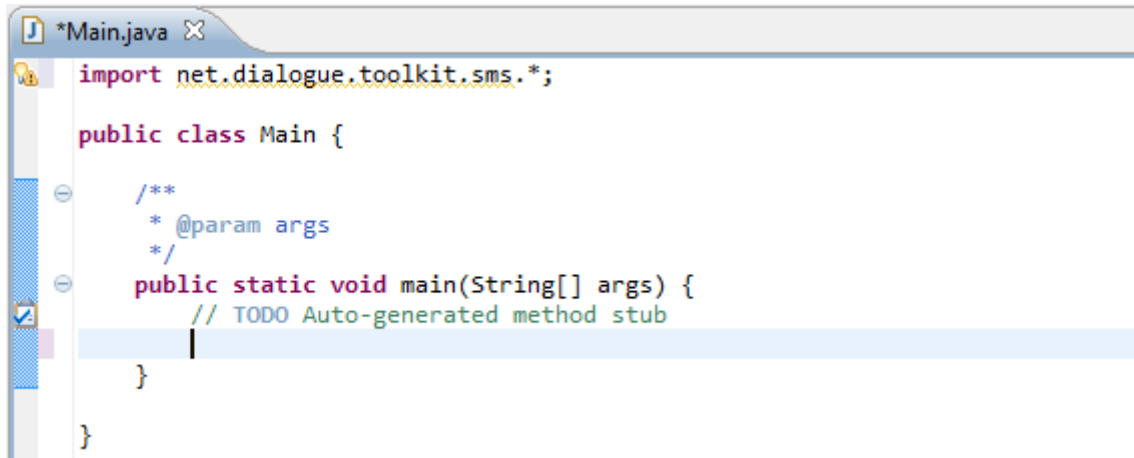
To use this package include `import net.dialogue.toolkit.sms.*;` in your source. However, you need to create a class: right-click in either **Package Explorer** or **Navigator** views, and select **New > Class**.



Name the class **Main** and select the option to have a main method stub automatically created. Click **Finish** to create the class.



Add the `import net.dialogue.toolkit.sms.*;` line at the top of the source code. The class including the import now looks as follows:



```

import net.dialogue.toolkit.sms.*;

public class Main {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub
    }

}

```

1.3 Instantiating SendSmsClient

While `SendSmsClient` can be instantiated directly we suggest you use the supplied `SendSmsClient.Builder` to facilitate construction and configuration of the `SendSmsClient` instance. The parameters required to build `SendSmsClient` are your API endpoint and API credentials (user name and password).

```

public static void main(String[] args) {

    SendSmsClient client = new SendSmsClient.Builder()
        .endpoint("endpoint")
        .credentials("user", "pass")
        .build();

}

```

Replace the yellow marked `endpoint`, `user` and `pass` by your API endpoint host name, user name and password, respectively. You will have received your API credentials as part of the Dialogue sign-up process. You will also have been given a document or link to a document, which contains the endpoint to use. Do not prefix the endpoint with `http://`.

By default secure communication via SSL is enabled; to disable SSL you can set the client's `Secure` property to `false` or add `.secure(false)` before the call to `.build()`; however we strongly recommend to leave secure communication enabled.

1.4 Sending a Message

To send a message, first instantiate the `SendSmsRequest` object, which defines various message parameters. For example:

```

SendSmsRequest request = new SendSmsRequest(
    "This is a test message",
    "44xxxxxxxxxx"
);


```


Replace the yellow marked `44xxxxxxxxxx` by your recipient number in international, normalized format, e.g. starting with 1 for US numbers or 44 for UK numbers.

To submit the message write:

```
try {
    SendSmsResponse response = client.sendSms(request);
} catch (HttpClientErrorException e) {
    System.out.println(e.getResponseBodyAsString());
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
```

Use `HttpClientErrorException.getResponseBodyAsString()` to extract an error description from the response body (the HTTP status code alone, e.g. “400 Bad Request” will not give you the reason why a submission failed).

Save your source file and run your application using the  button and that's it! You've just sent an SMS message.

1.4.1 Complete Code Sample

Here is the full code again:

```
import net.dialogue.toolkit.sms.*;

import org.springframework.web.client.HttpClientErrorException;
import java.io.IOException;

public class Main {

    public static void main(String[] args) {

        SendSmsClient client = new SendSmsClient.Builder()
            .endpoint("endpoint")
            .credentials("user", "pass")
            .build();

        SendSmsRequest request = new SendSmsRequest(
            "This is a test message",
            "44xxxxxxxxxx"
        );

        try {
            SendSmsResponse response = client.sendSms(request);
        } catch (HttpClientErrorException e) {
            System.out.println(e.getResponseBodyAsString());
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

1.5 Processing the Response

You should inspect the `SendSmsResponse` object returned by the `sendSms` call to see if the submission was successful or a failure. The response contains a list of `Sms` objects, each one corresponding to a submitted message. The `Recipient` property can be used to associate a submission result with the original recipient (in case multiple recipients were specified).

The `Successful` property of each `Sms` object is `true` if the submission succeeded (in which case the `Id` property contains the unique submission identifier); otherwise use properties `SubmissionReport` and `ErrorDescription` to get an error code and description, respectively.

Here is some code that inspects the `SendSmsResponse` object:

```
for (Sms sms : response.getMessages()) {
    if (sms.isSuccessful()) {
        System.out.printf("Submission to '%s' successful; " +
            "messageId: %s\n",
            sms.getRecipient(),
            sms.getId()
        );
    } else {
        System.out.printf("Submission to '%s' failed; " +
            "errorCode: %s, errorDescription: %s\n",
            sms.getRecipient(),
            sms.getSubmissionReport(),
            sms.getErrorDescription()
        );
    }
}
```

1.5.1 Complete Code Sample

Here is the full code again:

```
import net.dialogue.toolkit.sms.*;

import org.springframework.web.client.HttpClientErrorException;
import java.io.IOException;

public class Main {

    public static void main(String[] args) {

        SendSmsClient client = new SendSmsClient.Builder()
            .endpoint("endpoint")
            .credentials("user", "pass")
            .build();

        SendSmsRequest request = new SendSmsRequest(
            "This is a test message",
            "44xxxxxxxxxx"
        );
```

```

try {
    SendSmsResponse response = client.sendSms(request);

    for (Sms sms : response.getMessages()) {
        if (sms.isSuccessful()) {
            System.out.printf("Submission to '%s' successful; " +
                              "messageId: %s\n",
                              sms.getRecipient(),
                              sms.getId()
                              );
        } else {
            System.out.printf("Submission to '%s' failed; " +
                              "errorCode: %s, errorDescription: %s\n",
                              sms.getRecipient(),
                              sms.getSubmissionReport(),
                              sms.getErrorDescription()
                              );
        }
    }
} catch (HttpClientErrorException e) {
    System.out.println(e.getResponseBodyAsString());
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
}

```

1.6 Multiple Messages

You can submit multiple messages at once by adding further strings to the Messages property, a property of type `List<String>`. You can also use the corresponding constructor to create the request object with a string list.

```

SendSmsRequest request = new SendSmsRequest(
    Arrays.asList(
        "This is a test message", "This is another message"
    ),
    "44xxxxxxxxxx"
);

```

In the response you will find one `Sms` object for each message. The `foreach` loop used for processing the response in section 1.5 iterates over all returned `Sms` objects and prints the submission outcome for each message.

1.7 Multiple Recipients

You can target multiple recipients at once by adding further strings to the Recipients property, a property of type `List<String>`. You can also use the corresponding constructor to create the request object with a string list.

```

SendSmsRequest request = new SendSmsRequest(
    "This is a test message",

```

```
Arrays.asList(
    "44xxxxxxxxxx", "44xxxxxxxxxx"
);
```

In the response you will find one `Sms` object for each recipient. The `foreach` loop used for processing the response in section 1.5 iterates over all returned `Sms` objects and prints the submission outcome for each recipient.

1.8 Additional Parameters

There are other parameters that can be provided via properties of the `SendSmsRequest` object. You can call `System.out.println(request);` to dump all parameters in XML representation to the console for inspection.

1.8.1 ConcatenationLimit

By setting the `ConcatenationLimit` property, you enable long message concatenation. If the length of any message exceeds the default character limit the messaging gateway will send multiple concatenated messages up to the concatenation limit, after which the text is truncated. Concatenation works by splitting and wrapping the message in packets or fragments, each fragment being prefixed by the current fragment number and the total number of fragments. This allows the phone to know when it received all fragments and how to reassemble the message even if fragments arrive out of order.

The concatenation limit refers to the maximum number of message fragments, not the number of characters, e.g. a concatenation limit of “3” means no more than 3 SMS messages will be sent, which in total can contain up to 459 GSM-compatible characters. The concatenation overhead is 7 characters, so $160 - 7 = 153$ available characters per fragment $\times 3 = 459$ total characters.

```
SendSmsRequest request = new SendSmsRequest(
    Arrays.asList("This is a test message"),
    Arrays.asList("44xxxxxxxxxx")
);
request.setConcatenationLimit(10);
```

In the response you will find one `Sms` object for each message segment.

1.8.2 UserTag

By setting the `UserTag` property you can tag messages. You can use this for billing purposes when sending messages on behalf of several customers.

```
SendSmsRequest request = new SendSmsRequest(
    Arrays.asList("This is a test message"),
    Arrays.asList("44xxxxxxxxxx")
);
request.setUserTag("Customer1");
```

The `UserTag` property must not exceed 50 characters; longer values will make the submission fail.

1.8.3 ScheduleFor

Use the `ScheduleFor` property to delay sending messages until the specified date and time. The property is of type `Date`. When parsing dates from strings you should consider setting your time zone on `SimpleDateFormat`, so the correct time zone conversion is made.

This example sends the message on 1st December 2011 at 6:30 PM (relative to your machine's system time zone):

```
SendSmsRequest request = new SendSmsRequest(
    Arrays.asList("This is a test message"),
    Arrays.asList("44xxxxxxxxxx")
);

try {
    request.setScheduleFor(new SimpleDateFormat(
        "yyyy-MM-dd HH:mm:ss").parse("2011-12-01 18:30:00"));
} catch (ParseException e) {
    e.printStackTrace();
}
```

If the schedule date/time is in the past the message is sent immediately.

1.8.4 ValidityPeriod

Use the `ValidityPeriod` property to specify the maximum message delivery validity after which the message is discarded unless received. The property is of type `long` (the unit is milliseconds; however the lowest accepted value is 1 minute, i.e. 60000 ms). If not set the default validity period is applied.

This example sets the validity period to 1 week:

```
SendSmsRequest request = new SendSmsRequest(
    Arrays.asList("This is a test message"),
    Arrays.asList("44xxxxxxxxxx")
);
request.setValidityPeriod(1000L * 60 * 60 * 24 * 7 /*7 days*/);
```

This example sets the validity period to 10 days:

```
SendSmsRequest request = new SendSmsRequest(
    Arrays.asList("This is a test message"),
    Arrays.asList("44xxxxxxxxxx")
);
request.setValidityPeriod(1000L * 60 * 60 * 24 * 10 /*10 days*/);
```

The maximum validity period is 14 days, if you specify a longer validity period 14 days will be used.

1.8.5 ConfirmDelivery, ReplyPath, UserKey

The property `ConfirmDelivery` can be set to `true` to enable tracking of message delivery. If you enable `ConfirmDelivery` you must also set the `ReplyPath` property pointing to an HTTP event handler that you implement (see section 2). Optionally, set `UserKey` to an

arbitrary, custom identifier, which will be posted back to you. You can use this to associate a message submission with a delivery report.

```
SendSmsRequest request = new SendSmsRequest(
    Arrays.asList("This is a test message"),
    Arrays.asList("44xxxxxxxxxx")
);
request.setConfirmDelivery(true);
request.setReplyPath("http://www.myserver.com/mypath");
request.setUserKey("myKey1234");
```

Note that depending on that path setup some web servers may send redirects. For example, if mypath is a directory the web server may respond with a 302 redirect response indicating the post should go to mypath/. The messaging platform does **not** follow redirects, so you need to make sure that your reply paths do not use redirection.

1.8.6 SessionId, SessionReplyPath

The property SessionReplyPath points to an HTTP event handler that you have implemented (see section 2). The handler is invoked if the recipient replies to the message they have received. Optionally you can specify the SessionId property, which will be posted back to you. You can use this to associate a message submission with a reply.

```
SendSmsRequest request = new SendSmsRequest(
    Arrays.asList("This is a test message"),
    Arrays.asList("44xxxxxxxxxx")
);
request.setSessionReplyPath("http://www.myserver.com/mypath");
request.setSessionId("1234567890");
```

Note that depending on that path setup some web servers may send redirects. For example, if mypath is a directory the web server may respond with a 302 redirect response indicating the post should go to mypath/. The messaging platform does **not** follow redirects, so you need to make sure that your reply paths do not use redirection.

1.9 Custom Parameters

The Dialogue Messaging Gateway supports further parameters. To be able to use these parameters the SendSmsRequest object is based on Dictionary<string, string>. For example, to supply your own unique submission identifier you can write this code:

```
SendSmsRequest request = new SendSmsRequest(
    Arrays.asList("This is a test message"),
    Arrays.asList("44xxxxxxxxxx")
);
request.put("X-E3-Submission-ID", UUID.randomUUID().toString());
```

(If you use the same X-E3-Submission-ID parameter in a future SendSmsRequest object it will not perform a new message submission but return the previous SendSmsResponse

object. Do not confuse this parameter with the returned `Sms.Id` property – they are different.)

1.10 Transports

A transport is a component that is responsible for performing an HTTP request to the web service. The `SendSmsClient.Builder` allows you to create instances of `SendSmsClient` using different transports. To do this write:

```
SendSmsClient client = new SendSmsClient.Builder()
    .transport(transport)
    .endpoint("endpoint")
    .credentials("user", "pass")
    .build();
```

Where the `transport` parameter is an instance of `ClientHttpRequestFactory`. For convenience we have predefined three transports; the transport used by default is:

```
SendSmsClient.Builder.TRANSPORT_SIMPLE_CLIENT
```

This transport employs standard J2SE facilities to perform an HTTP request (it uses `URLConnection`); it will be supported by all standard Java VM implementations and does not rely on third-party library dependencies.

You may prefer to use other HTTP client libraries such as those provided by Apache. These are standards-based HTTP clients and include advanced features such as connection management and connection pooling.

This transport uses the Apache Commons HttpClient (3.x, see <http://hc.apache.org/httpclient-3.x/>):

```
SendSmsClient.Builder.TRANSPORT_COMMONS_CLIENT
```

Note the Commons HttpClient is now deprecated; Apache have stopped support and maintenance. However, as it's still used widely in the industry you may still want to use it, especially if it's already integrated in your platform.

Preferably use this transport, the newer Apache HttpComponents Client (4.x, see <http://hc.apache.org/httpcomponents-client-ga/>):

```
SendSmsClient.Builder.TRANSPORT_HTTP_COMPONENTS_CLIENT
```

You may provide a custom transport by passing an instance of `ClientHttpRequestFactory` to `transport()`.

1.10.1 Dependencies

Transports `TRANSPORT_COMMONS_CLIENT` and `TRANSPORT_HTTP_COMPONENTS_CLIENT` rely on third-party client libraries, which you can find in the `lib` directory:

`TRANSPORT_COMMONS_CLIENT` requires:

```
commons-httpclient-3.0.jar
commons-codec-1.4.jar (*)
commons-logging-1.1.1.jar (**)
junit-4.0.jar (***)
```

TRANSPORT_HTTP_COMPONENTS_CLIENT requires:

```
httpclient-4.1.jar
httpcore-4.1.jar
commons-codec-1.4.jar
commons-logging-1.1.1.jar
```

Note that there is no runtime requirement for these libraries to be present. If the required libraries are not found on the runtime class path TRANSPORT_COMMONS_CLIENT and TRANSPORT_HTTP_COMPONENTS_CLIENT will be set to TRANSPORT_SIMPLE_CLIENT.

(*) Actual dependency is on commons-codec-1.2.jar

(**) Actual dependency is on commons-codec-1.0.3.jar

(***) Actual dependency is on junit-3.8.1.jar

1.11 RestOperations

The `SendSmsClient` class uses an instance of `RestOperations` (which is part of Spring's `RestTemplate` framework) to communicate with the server. You may substitute this with your own implementation, for instance, by constructing `SendSmsClient` providing a custom instance of `RestOperations`.

The `SendSmsClient.Builder` instantiates the client using an instance of `RestOperations` implemented by `SendSmsClientRestTemplate`, which takes care of adding the necessary marshaller to correctly communicate with the server's protocol. If you want to implement your own `RestOperations` we suggest you use `SendSmsClientRestTemplate` as a base class, otherwise you need to provide your own marshalling to correctly serialize `SendSmsRequest` objects to XML and `SendSmsResponse` objects from XML.

This example shows how to set up a mock service, which does not perform any real submissions but still returns a successful submission response:

```
RestOperations restOperations =
    new SendSmsClientRestTemplate() {
        @SuppressWarnings({ "unchecked", "unused" })
        @Override
        public <T> T postForObject(
            String url,
            Object entity, Class<T> responseType,
            Object... uriVariables) throws RestClientException {
            SendSmsResponse response = new SendSmsResponse();
            SendSmsRequest request =
                ((HttpEntity<SendSmsRequest>) entity).getBody();
            for (String message : request.getMessages()) {
                for (String recipient : request.getRecipients()) {
                    Sms sms = new Sms();
                    sms.setId(UUID.randomUUID().toString());
```



```

        sms.setSubmissionReport(StatusCodes.
            TransactionCompleted.STATUS_SUCCESSFUL);
        sms.setRecipient(recipient);
        response.getMessages().add(sms);
    }
}
return (T) response;
};

SendSmsClient client = new SendSmsClient(restOperations);

```

You may use such an instance of `SendSmsClient` during development or testing purposes to prevent any accidental submissions.

For convenience we have provided the existing mock `RestOperation` instances `MockRestTemplate.SucceededMockRestTemplate` and `MockRestTemplate.FailingMockRestTemplate`, the first one returning only successful submission reports, the second one returning only submission failures (you can subclass this overriding `getSms()` to change the status code and error description used).

2 Receiving Reports and Replies

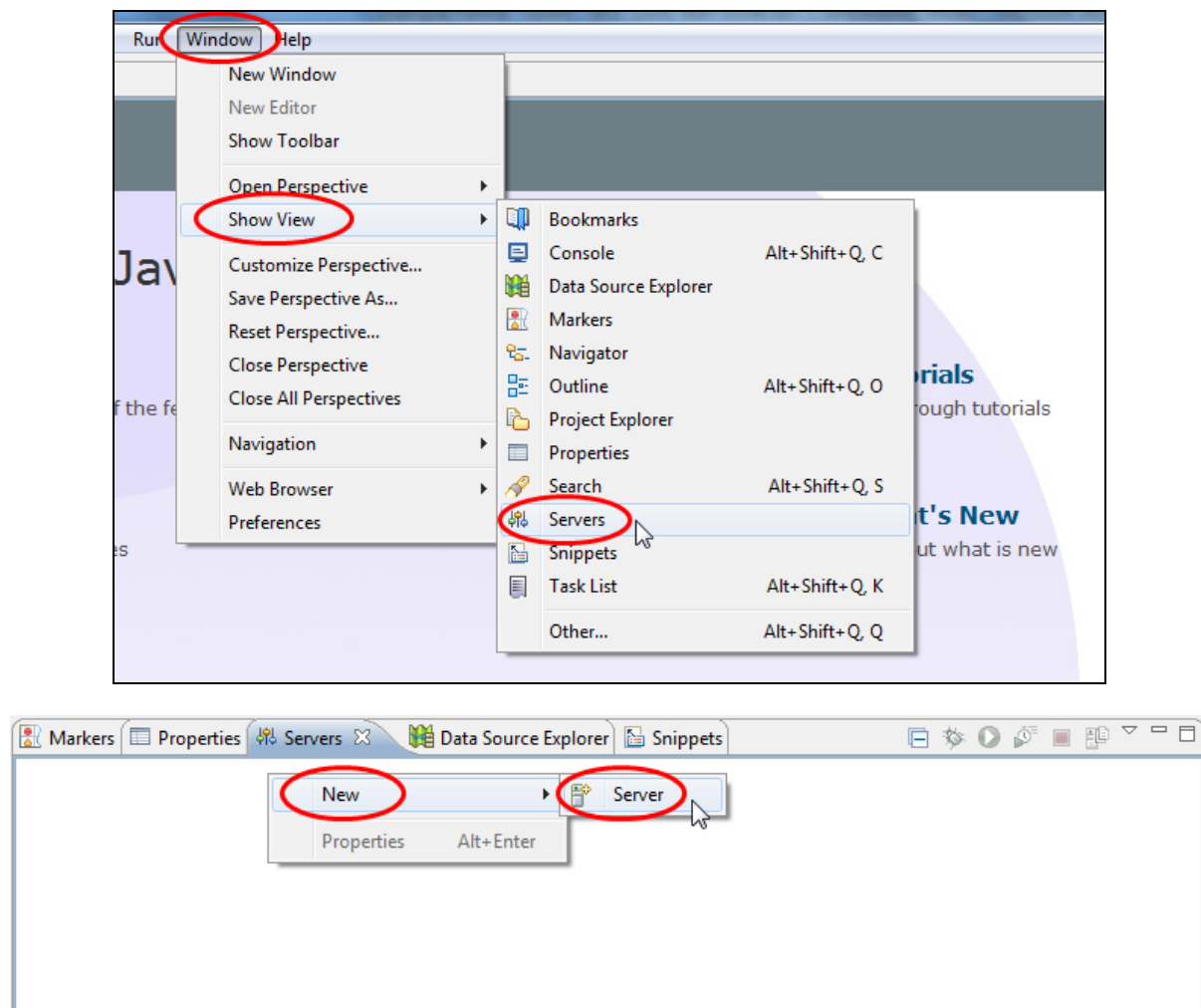
2.1 Integrating the Server

For implementing a handler to receive and process reports and replies this guide uses a Java IDE called “Eclipse IDE for Java EE Developers”, which can be downloaded free of charge at <http://www.eclipse.org/downloads/> (pick the Java EE version).

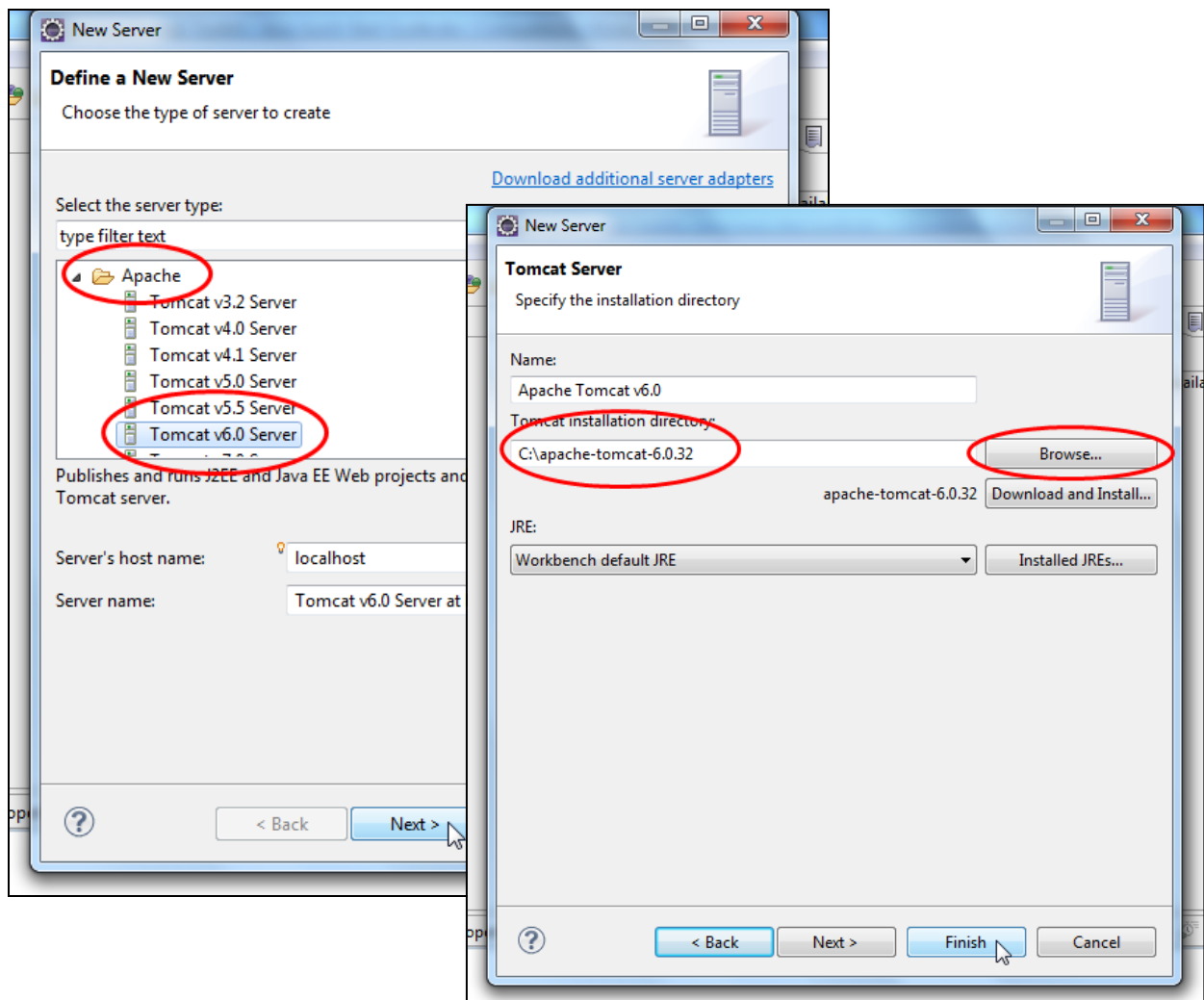
You will also need the Java Development Kit (JDK) installed. Note that you don't require the Java Enterprise Edition (Java EE) of the JDK. The toolkit library supports Java 1.5 or higher. The JDK can be downloaded here: <http://www.oracle.com/technetwork/java/javase/downloads/>.

The toolkit library does not rely on any particular web server, i.e. you're free to use any web or application server technologies that support the Servlet specification. This guide illustrates the use of Apache Tomcat, which you can download from <http://tomcat.apache.org/>.

First you need to integrate your Tomcat installation into the Eclipse IDE. Select menu **Window > Show View > Servers**. This shows the **Servers** view/panel (usually this is docked at the bottom side of the main window).



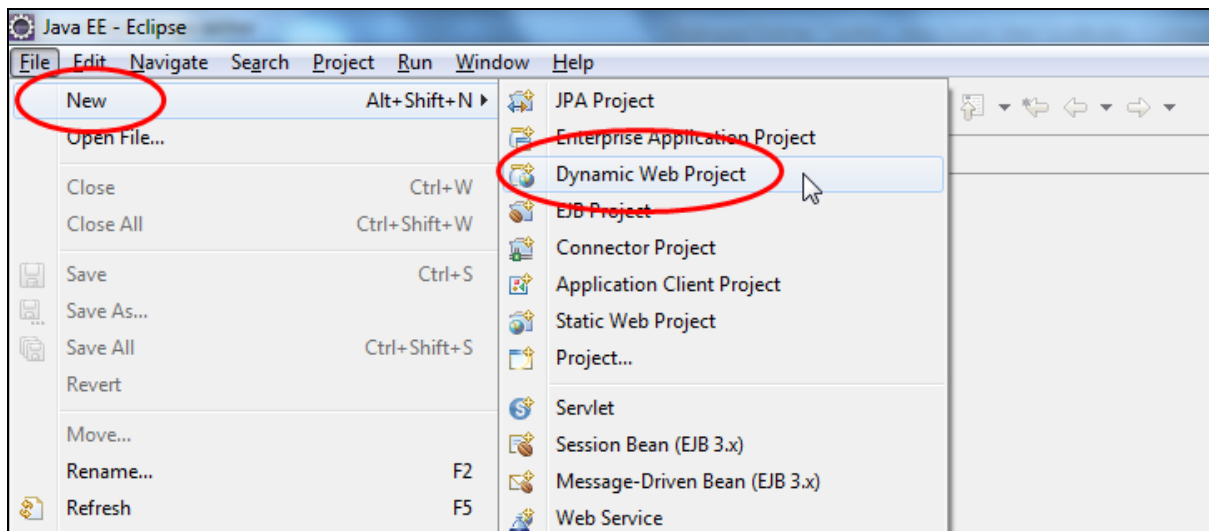
Right-click inside the view and select **New > Server**, which opens the **New Server** dialog. Expand the **Apache** group and select the version of Tomcat you have installed (during the creation of this guide we used version 6.0.32). Click **Next >** to continue.



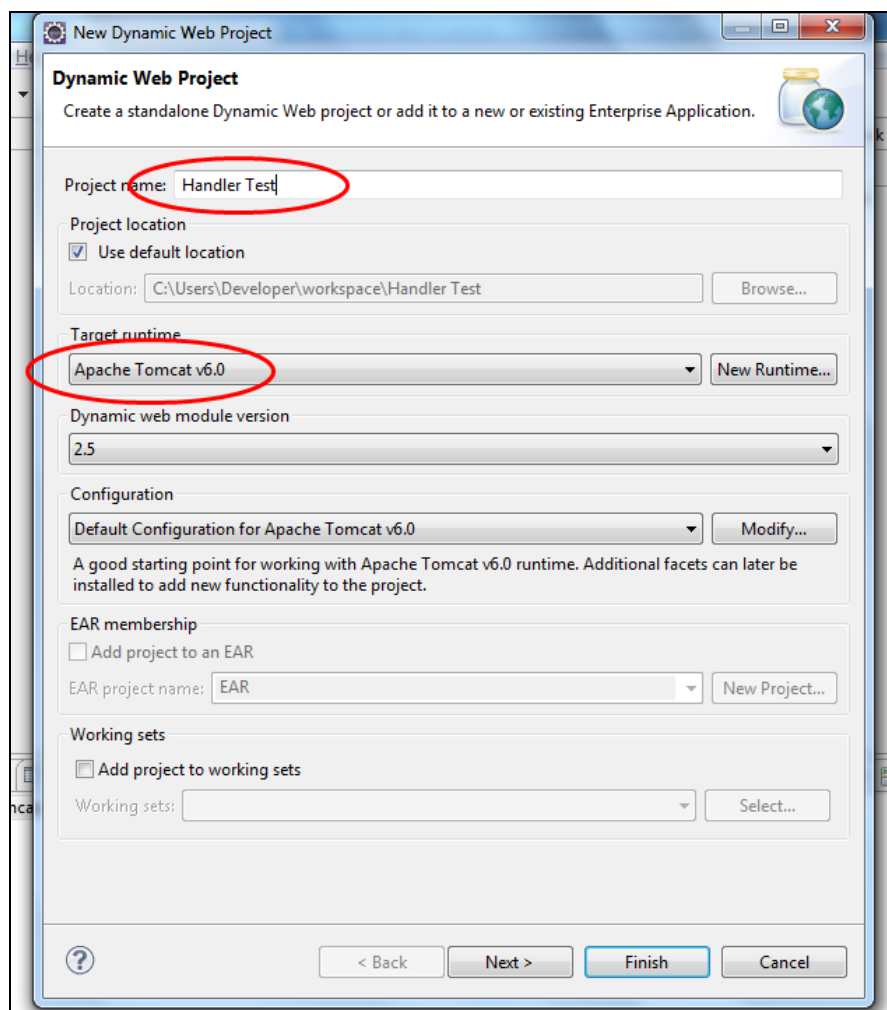
Click **Browse...** to select the Tomcat installation directory. Once selected it will appear in the input field to the left. Click **Finish**.

2.2 Creating the Project

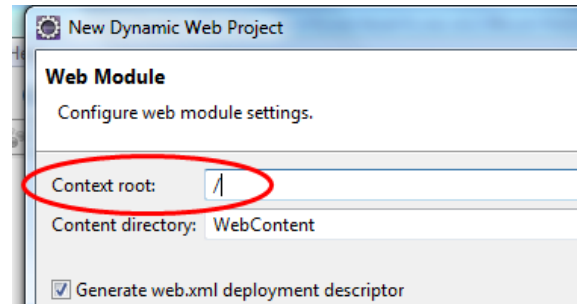
Using Eclipse create a new **Dynamic Web Project**.



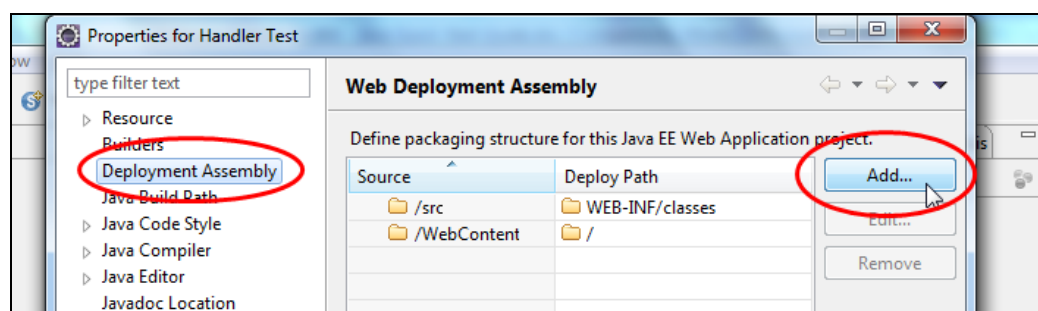
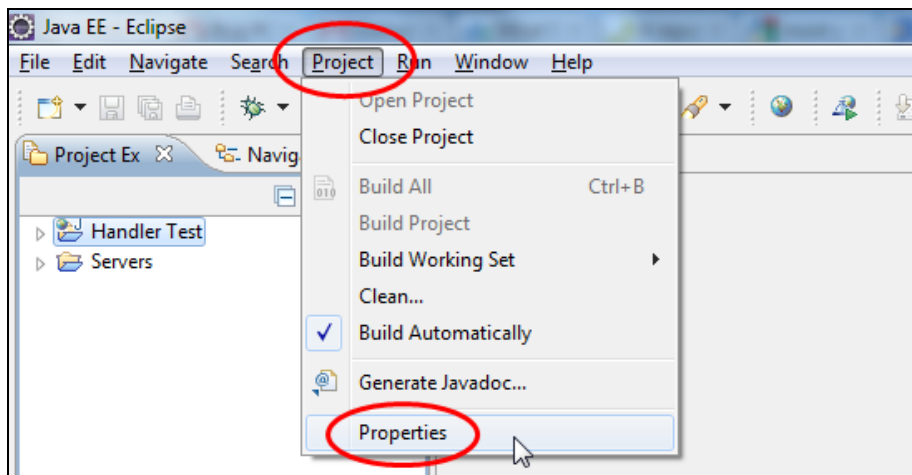
Give your project a name and optionally specify an alternate project location. Check that your Tomcat server is selected in the **Target runtime** drop-down.



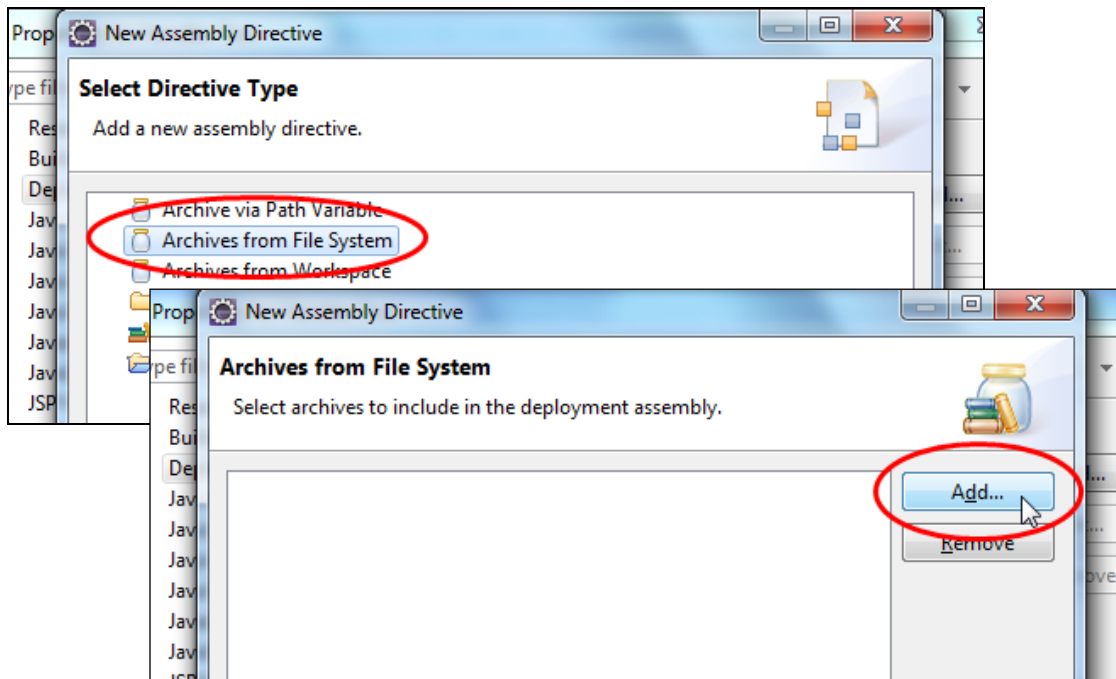
Click **Next >** twice until you are on the **Web Module** page. Change the **Context root** input field from “Handler_test” to “/” (enter a single forward slash, without quotes). Tick **Generate web.xml deployment descriptor**.



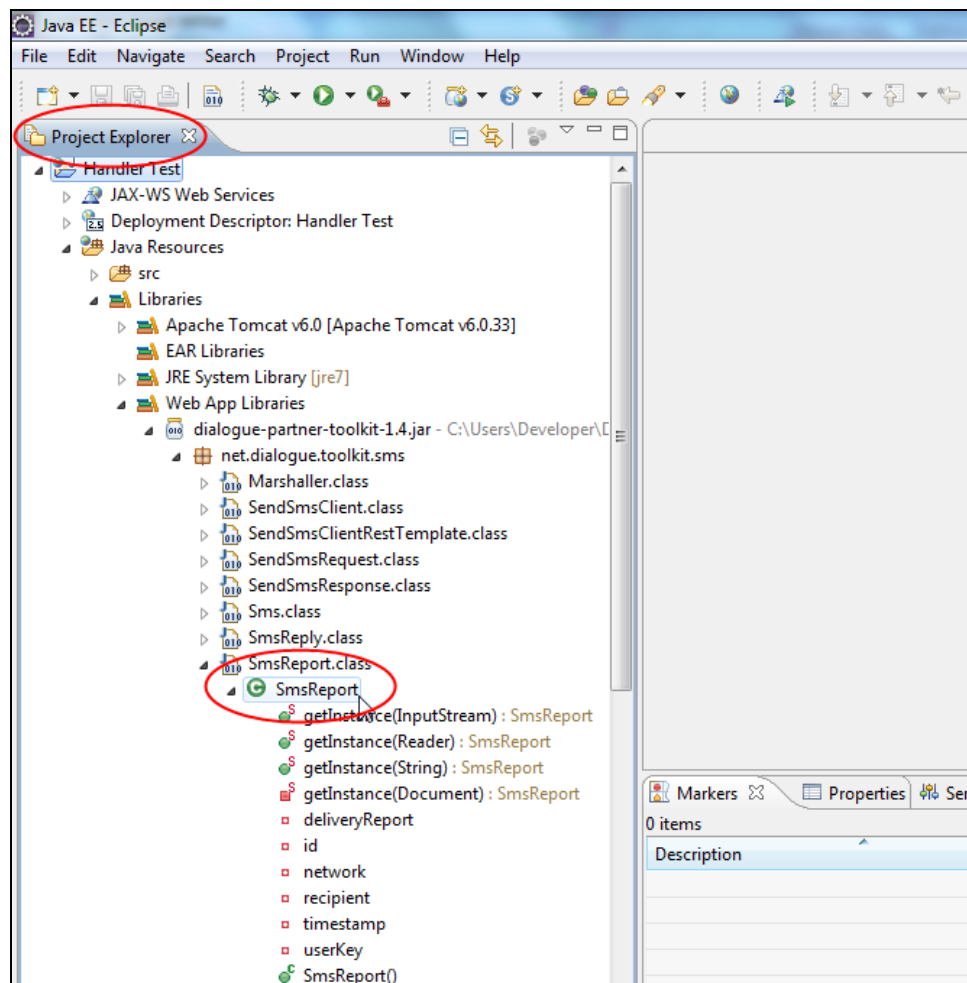
Click **Finish** and wait until the project is created. To add the toolkit library to your project open menu **Project > Properties** and under **Deployment Assembly** click the **Add...** button.



Next choose **Archives from File System** and click **Next >** and then **Add...** to navigate to the folder containing the client and select the JAR file. After adding the file click **Finish**, then **OK**. The library is now added to your project.



You can explore the toolkit library using the **Package Explorer** view by expanding Java Resources > Libraries > Web App Libraries > dialogue-partner-toolkit-1.4.2.jar > net.dialogue.toolkit.sms. Take a look at the SmsReport class.

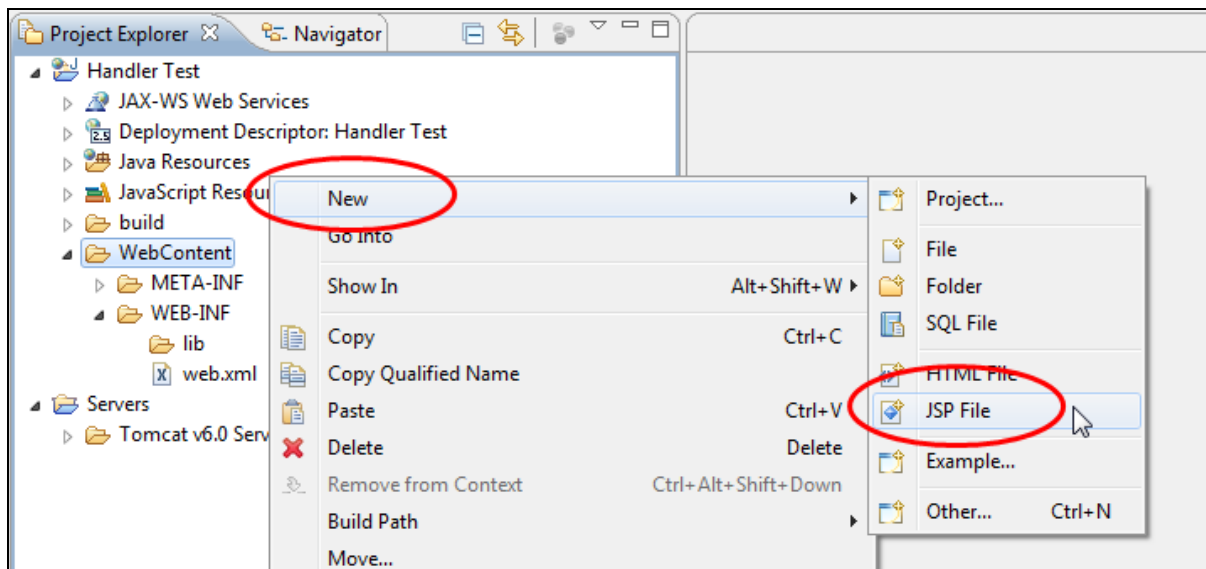


2.3 Importing the Package

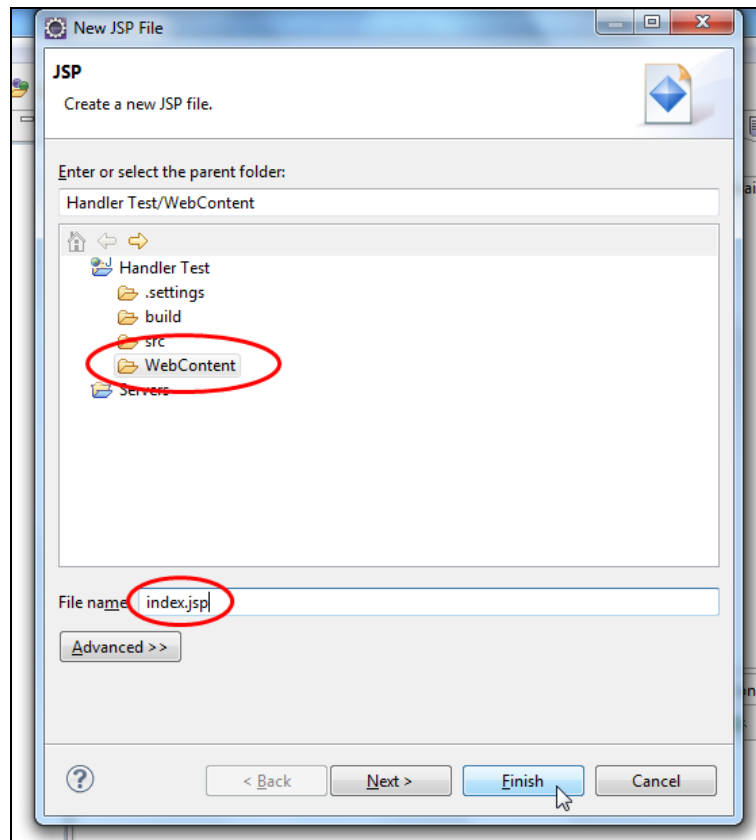
The package required for parsing reports or replies is `net.dialogue.toolkit.sms`. Here are the objects of interest for receiving reports and replies:

Object	Description
SmsReport	Utility object for parsing incoming reports (JSP)
SmsReply	Utility object for parsing incoming replies (JSP)

To use this package include `<%@ page import="net.dialogue.toolkit.sms.*" %>` at the top of your JSP file. However, you need to create a JSP first: right-click the **WebContent** folder in **Project Explorer** and select **New > JSP File**.



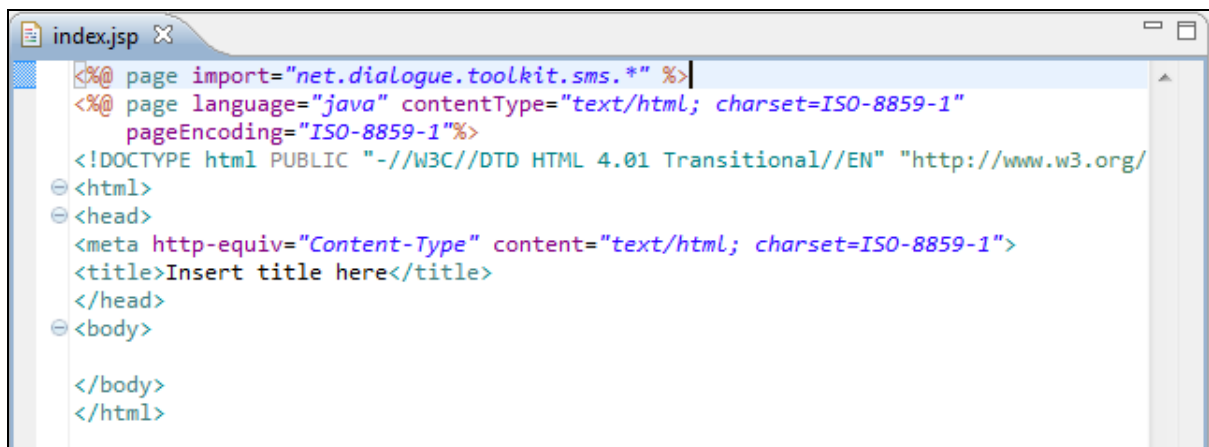
Make sure **WebContent** is selected as parent folder and name your file **index.jsp**. Click **Finish** to create the file.



Now add the import above the existing `<%@ page ... %>` directive on your page:

```
<%@ page import="net.dialogue.toolkit.sms.*" %>
```

Your JSP page should now look like this:



You can remove the `<!DOCTYPE>`, `<html>`, `<head>` and `<body>` elements. (These are generated by the IDE but are not used – our handler does not need to produce any visible HTML content.)

2.4 Processing Message Reports

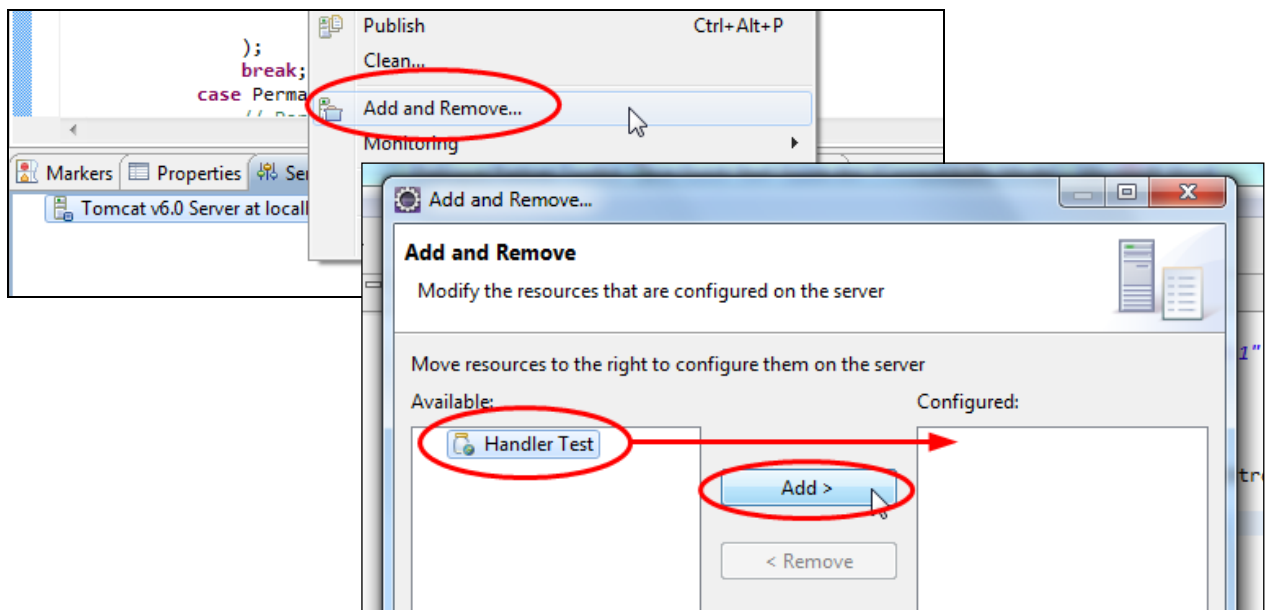
To process message report POST requests on your page write:


```
<%
if ("POST".equals(request.getMethod())) {
    SmsReport report = SmsReport.getInstance(request.getInputStream());
    switch (report.getState()) {
        case Delivered:
            // Message delivered
            break;
        case TemporaryError:
            // Temporary error but could still be delivered
            break;
        case PermanentError:
            // Permanent error, message was not delivered
            break;
    }
}
%>
```

You could add some logging using:

```
System.out.printf(
    "Message delivered: %s, %s, %s",
    report.getRecipient(), report.getUserKey(),
    report.getDeliveryReport()
);
```

Before you can run your project you must add it to the Tomcat server integration. To do this right-click the “Tomcat Server” entry inside the **Servers** view and select **Add and Remove...** from the context menu. Select your project under the **Available** list and click **Add >** to move it to the **Configured** list. Press **Finish** to confirm.



Save your JSP file and run your project using the  button in the **Servers** view. Open your web browser and point it to the address:

`http://localhost:8080/`

(8080 is the default Tomcat port; if your Tomcat instance uses a different port then change the URL accordingly.)

This displays a blank page. There's no parsing of an incoming report since this is a GET request.

To test the report processing download and install the `curl` command line utility from <http://curl.haxx.se>. Once installed you can fake a successful delivery report by running:

```
curl -H "Content-Type: application/xml" -d "<callback X-E3-Delivery-Report=\"00\" X-E3-ID=\"90A9893BC2B645918034F4C358A062CE\" X-E3-Loop=\"1322229741.93646\" X-E3-Network=\"Orange\" X-E3-Recipients=\"447xxxxxxxxx\" X-E3-Timestamp=\"2011-12-01 18:02:21\" X-E3-User-Key=\"myKey1234\"/>" -X POST http://localhost:8080/
```

(Make sure the port number 8080 coincides with your own.)

To fake a temporary error, change the X-E3-Delivery-Report value in the XML from 00 to something between 20 and 3F. To fake a permanent error, change the X-E3-Delivery-Report value to something between 40 and 7F.

That's it! You've successfully implemented a delivery report event handler.

Of course, you should use the `Recipient` and/or `UserKey` properties to associate the report with a previous submission. For example, prior to submission you might store a recipient and user key record in your own database. The record would also have a `DELIVERED` column which is left undefined or `NULL`. Upon receiving a report you may look up the record using the `Recipient` and `UserKey` properties and update the `DELIVERED` column to 'Yes' or 'No' to reflect the delivery state.

All properties exposed by `SmsReport` are:

Property	Description
Id	Unique report identifier (not that of the original submission).
Recipient	The original recipient of the submission.
DeliveryReport	Delivery report value, 00 to 1F indicating a successful delivery, 20 to 3F indicating a temporary error and 40 to 7F indicating a permanent error.
State	The delivery report value classified into <code>State.Delivered</code> , <code>State.TemporaryError</code> or <code>State.PermanentError</code> .
UserKey	Optional <code>UserKey</code> parameter from the message submission. Can be used to associate unique submissions with reports,

	even if the recipient is the same.
Timestamp	Date and time the report was received.
Network	Mobile operator network name.

Here is the full JSP page source code:

```
<%@ page import="net.dialogue.toolkit.sms.*" %>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%
    if ("POST".equals(request.getMethod())) {
        SmsReport report = SmsReport.getInstance(request.getInputStream());
        switch (report.getState()) {
            case Delivered:
                // Message delivered
                System.out.printf(
                    "Message delivered: %s, %s",
                    report.getRecipient(), report.getUserKey()
                );
                break;
            case TemporaryError:
                // Temporary error but could still be delivered
                System.out.printf(
                    "Temporary error: %s, %s, %s",
                    report.getRecipient(), report.getUserKey(),
                    report.getDeliveryReport()
                );
                break;
            case PermanentError:
                // Permanent error, message was not delivered
                System.out.printf(
                    "Permanent error: %s, %s, %s",
                    report.getRecipient(), report.getUserKey(),
                    report.getDeliveryReport()
                );
                break;
        }
    }
%>
```

2.5 Processing Message Replies

To process message reply POST requests on your page write:

```
<%
    if ("POST".equals(request.getMethod())) {
        SmsReply reply = SmsReply.getInstance(request.getInputStream());
        String sender = reply.getSender();
        String message = reply.getMessage();
        String sessionId = reply.getSessionId();
    }
%>
```

You could add some logging using:

```
System.out.printf("%s, %s, %s", sender, message, sessionId);
```

Run your project. Your web browser will open using an address which looks like this (the port number could vary):

<http://localhost:8080/>

This displays a blank page. There's no parsing of an incoming reply since this is a GET request.

To test the reply processing download and install the `curl` command line utility from <http://curl.haxx.se>.

Once installed, you can fake a message reply by running:

```
curl -H "Content-Type: application/xml" -d "<callback X-E3-Account-
Name=\"test\" X-E3-Data-Coding-Scheme=\"00\" X-E3-Hex-
Message=\"54657374204D657373616765\" X-E3-
ID=\"809EF683F022441DB9C4895AED6382CF\" X-E3-Loop=\"132223264.20603\" X-
E3-MO-Campaign=\"\" X-E3-MO-Keyword=\"\" X-E3-Network=\"Orange\" X-E3-
Originating-Address=\"447xxxxxxxx\" X-E3-Protocol-Identifier=\"00\" X-E3-
Recipients=\"1234567890\" X-E3-Session-ID=\"1234567890\" X-E3-
Timestamp=\"2011-11-25 12:14:23.000000\" X-E3-User-Data-Header-
Indicator=\"0\"/>" -X POST http://localhost:8080/
```

(Make sure the port number coincides with your own.)

That's everything required to implement a handler which can receive incoming replies. Here are all the properties exposed by `SmsReply`:

Property	Description
Id	Unique message identifier (not that of the original submission).
Sender	The original recipient of the submission (now the sender as it's a reply).
SessionId	Optional <code>SessionId</code> parameters from submission. Can be used to associate unique submissions with replies, even if the recipient is the same.
Message	The message text.
Timestamp	Date and time the message was received.
Network	Mobile operator network name.

Here is the full JSP page source code:

```
<%@ page import="net.dialogue.toolkit.sms.*" %>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%
    if ("POST".equals(request.getMethod())) {
        SmsReply reply = SmsReply.getInstance(request.getInputStream());
        String sender = reply.getSender();
        String message = reply.getMessage();
        String sessionId = reply.getSessionId();
        System.out.printf("%s, %s, %s", sender, message, sessionId);
    }
%>
```

3 Status Codes

The StatusCodes class lists common status codes used by submission and delivery reports. The codes are grouped into three categories represented by inner classes:

1. TransactionCompleted ("00" to "1F")
2. TemporaryError ("20" to "3F")
3. PermanentError ("40" to "5F")
4. RetryError ("60" to "7F")

Status codes are written in hexadecimal form; if you use any of the methods accepting an integer as status code make sure you correctly convert between systems. For example, 46 as hexadecimal value is PermanentError.STATUS_VALIDITY_PERIOD_EXPIRED, whereas 46 as integer value would be temporary status 2E.

To obtain an instance of StatusCodes you can either write this (recommended):

```
StatusCodes statusCodes = StatusCodes.INSTANCE;
```

or this (in case you want to subclass StatusCodes, e.g. to provide custom mappings):

```
StatusCodes statusCodes = new StatusCodes();
```

Uses of the class:

1. Check if a delivery report is of a particular status:

```
SmsReport report = ...
if(StatusCodes.PermanentError.STATUS_VALIDITY_PERIOD_EXPIRED
    .equals(report.getDeliveryReport())) {
}
```

2. Check if a status belongs to a certain range:

```
SmsReport report = ...
String statusCode = report.getDeliveryReport();
if(!StatusCodes.INSTANCE.isTransactionCompleted(statusCode)) {
}
```

3. Get a mapped status description for a status code:

```
SmsReport report = ...
String statusCode = report.getDeliveryReport();
if(!StatusCodes.INSTANCE.isTransactionCompleted(statusCode)) {
    String errorDescription =
        StatusCodes.INSTANCE.getDescription(statusCode);
}
```

The last point is particularly useful if you want to display status results on a web page since status descriptions are much more meaningful than status codes only.

Note that `SmsReport.getState()` uses methods offered by `StatusCodes` internally; it maps both `StatusCodes.isPermanentError()` and `StatusCodes.isRetryError()` to `State.PermanentError`.