

Introduction to API Mocking



Introduction to API Mocking

Contents

What is a Mock Service?	3
Why Should You Mock a Service?.....	3
When Should You Use A Mock?	4
The real service is not implemented	4
Services out of your control	5
Where Should You Use A Mock?.....	9
Where Should You Use A Mock?.....	11
SoapUI.....	11
When to use SoapUI for mocking a service.....	11
When not to use SoapUI.....	11
How do I create a mock in SoapUI?	11
Test the mock.....	20
View the WSDL.....	23
Add an item.....	24
Deploy to an application server.....	24
Conclusion.....	25
References	25

What is a Mock Service?

Before we start discussing reasons you may want to mock a service, let's define what a mock service is. No, it's not about being snarky... at least, not in this context.

According to the Cambridge Dictionaries Online:

“Not real but appearing or pretending to be
exactly like something”¹

So we are essentially talking about something that will behave as a real service, but will only mimic the behavior of the service - essentially an implementation that you have control over and are responsible for.

A mock service is not the same as a service simulation. A mock will only simulate a part, perhaps one specific interaction, of a system. A service simulator will simulate the entire system and behave in an expected way for all calls.

Why Should You Mock a Service?

So why would you want to create such a thing? The simplest possible answer to this is, “Because the real thing doesn't work,” but this may be a bit simplistic. A deeper reasoning of why the real service doesn't always work is needed. Some of those reasons will be explored in the following pages.

When Should You Use A Mock?

Not everything should be mocked. The actual reason why you want to use a mock is probably one or more of the following:

The real service is not implemented

You can't use the real service because it hasn't been implemented yet.

You are now faced with two options: serial or parallel development.

The serial option is to postpone the development of the system that uses the service until it is implemented and deployed. This may take a long time, and the business value of the service won't be possible to achieve until all the parts are implemented and working.

In a word, the serial approach is slow.

The other option is to do whatever it takes to develop in parallel. This means that the contract of the service to be implemented must be created as early as possible. It may not be perfect or complete, but it must be complete enough to create a mock service that behaves well enough to test. The service contract will probably have to evolve when problems are found or communicated. Both the service implementors and the client implementors will learn during the process of early testing.

Parallel development can help move a project forward, but it does come with some risks if issues are found with the service being mocked. But it's safe to say that updating a mock and changing development code is easier and less impactful than changing both after the application has gone to production. Mocking is your only viable solution if you want to be able to develop a client that uses a service that doesn't actually exist yet.

Mocking is your only
viable solution...

Services out of your control

Services out of your control are good candidates for mocking. It is often a problem if you don't have control over the data you need from a service when testing. This is commonly the case when performing automated testing.

Test data

The test data you need to be able to verify your part of the interaction may not be available. It is always difficult to test against a moving target where the test data varies with time.

An example might be an online storefront. Let's say that you are building a front end to an online storefront and the back end that holds the products is available behind a service. You want your automated test to purchase a product. This is no problem as long as the product is available in stock. But after a while, when you have executed the test many times, you will probably run out of items to sell.

Because you are testing the purchase path and not inventory results, you need to invent a way for your automated test to continue to run without worrying about inventory quantities. A mock that always returns that this product is available will make sure that the automated tests never fail due to lack of inventory.

Another example could be if you are building a service where you sell prepaid codes for mobile phones. You purchase the refill codes from different operators who only sell batches of prepaid codes. You would like to avoid buying a lot of new prepaid codes when developing your replenishing module, so you mock the public interface for each supplier and are able to develop your storefront and automate the testing without getting an enormous invoice from the operators.

Life cycle Out of Your Control

Often, the services you interact with are managed by someone else and you are not in control of their life cycle. All you have in your project is the contract the service implements. You can't decide when a new version should be deployed and the decision is left to the discretion of another team with completely different priorities than the priorities of your team. The solution to this case is to use a mock that implements the relevant parts of the service contract. When you are done with one call, mock the next call and implement your client using this new, extended mock. You are now independent of the service developers' life cycle handling.

Service Availability

You are developing a client and need to talk to a service. You have the contract and you know what you can (and should) expect. The problem is that the service is not always available. The supplier, be it internal or external, doesn't have good uptime for a test service, and it may go offline without notice.

In order to focus your testing on your code rather than theirs, you must be able to build your part of the system in parallel. If you can't create a fake service to test against, you may not be able to implement your part of the system.

Access to the Service

The service you need is only available from your production network, rather than the network the developers are using. This is very common in large organizations with multiple departments and hand-offs. Another common access problem is that you are missing the credentials needed to use a service. You don't have the proper username or password and are therefore blocked from using the service. Getting the proper credentials can sometimes be surprisingly difficult. Mocking the wanted service can be a good way to be in control without violating a user agreement or compromising security.

Expensive services

A service may be very expensive to use, especially if the supplier charges you for each call to the service. This may be acceptable if you only do manual testing, but automating the testing may be an issue if the cost is very high. You can solve this problem by mocking the service to avoid a charge when you call it.

Payment solutions

Implementing a credit card payment solution often requires an external service provider and a credit card number. Most likely, you are building a test that executes a purchase path but is not intended to test the credit card transaction itself since that is provided by a third party. In this case, using a mock can avoid passing real credit card transactions through while still allowing you to test your portion of this code.

Failing scenarios

The happy path works, but you need to be able to provoke different error scenarios. How could you test your entire implementation if these errors are unusual and hard to provoke? You need to be able to trigger the errors by sending known data that will create a failure. A mock can be a great solution that can generate an expected error so you can test how your application responds to an error condition.

Prototyping

Prototyping a service is very similar to giving your consumers early access.

Developing a prototype is always the best way to demonstrate your service's capabilities and design. It's always easier to discuss and improve something that actually exists and can be used. Customers can test whether they really are getting the response they need and find the areas that need improvements.

Prototyping without using fast tools is hard and will probably fail. It will be similar to developing the actual service - too slow. A good mocking tool can speed up the cycle time and therefore enable faster feedback.

Simulations

Mocking is usually about implementing a small portion of a large system. A small example can always be extended and simulating an entire system is obviously possible using the same tools. The same tooling that you use for creating a mock of a specific part of a system can be used to simulate the entire system and thus create a complete service simulator.

Validating that a system will deliver the planned business values is easier using a simulator compared to developing the entire system. Developing a large system may take considerable time. The simulator can later be removed in small steps when each part of a service is implemented.

Third-party consumers

A service supplier can allow their customers to have access to mocks as part of their testing offer. The supplier can develop mock suites and make them accessible. This would remove the need for the supplier to supply an online testing environment and thus reduce the supplier's maintenance and hosting costs.

The benefits for the developers that use these mocks would be that they don't have to be online and connect to the service through the Internet. This means lower network latency resulting in faster testing. The implementation of test automation always requires a stable supporting environment. Mocks are great for this, but you'll want to be in control over the mock and where it is executed.

Summary

There are many situations where mocking a Web service is the best option you have because you can't use the real service.

Before you start using a mocked service, make sure you know why you need to implement the mock. Mocks need maintenance and will add to your maintenance burden. Ask yourself, "is it worth the extra maintenance cost?" If your answer is yes, then go ahead and implement the mock. If you are unsure, postpone adding a mock to your project.

Where Should You Use A Mock?

Mocks should be used in testing and development environments, but maybe not in all testing environments.

Testing is often done in stages that build upon each other. If something isn't accepted in a specific stage, it isn't promoted to the next testing level.

A common set of test stages looks like:

- ◆ Development – Implement and test the actual functionality.
Mocks should be used to reduce development impediments.
- ◆ Automated Integration Test – First integration of different components of the system. A part of the continuous integration process. Mocks should be used to mimic the external services while making sure that the services are always available.
- ◆ Integration Test – Integrates the system with external systems.
Mocks may be used if an external service is hard to use. But it is preferable to avoid mocks as much as possible. The integration test stage may be automated or manual. External services must be very reliable if these tests are automated.

- ◆ User Acceptance Test – Manual or exploratory testers should use the product. Mocks may be used if external services are hard to use. If external services are used, then it must be easy for the tester to verify them separately to avoid false negatives. A specific test may fail if the external system fails but this does not mean that there is a problem with the system under test.
- ◆ Acceptance Test – Mocks should not be used. It must be possible to verify the external systems without the system under test to avoid false negatives if a test fails.
- ◆ Production – Mocks should not be used here.

All these tests stages build on top of each other. The rule of thumb is that you want to remove as many moving parts as possible at early stages and add them gradually the closer to production you are. This means that using mocks early is usually a very good idea. And, in turn, using them late in the verification process is a really bad idea.

An example may be to have a mock in the final acceptance test environment. If you have a mock there, then you will not know until the product is in production if it works with the actual service. A bug found at this stage is unnecessarily expensive. Where you use mocked services depends on the situation.

An internal service may be good to mock early in the development and then remove the mock as soon as possible. An external service out of your control may be a good idea to mock to a very late stage in the development process.

The closer you are to production, the fewer mocks should be used. No mocks should be used in the last testing environment before production, and they should definitely never be used in production.

Where Should You Use A Mock?

There are many different tools for creating mocks. Building your own service is always an option, but this will need a more or less complete implementation and will require maintenance.

Tools like Mockito, JMock or EasyMock are designed for mocking in unit tests and will not be very useful if you want to mock a Web service based on SOAP or RESTful APIs.

Another option is to use a tool that supports creating a mock service. SoapUI is a great tool that will solve this problem.

SoapUI

When to use SoapUI for mocking a service

Use SoapUI when you need to prototype or fake an actual service and don't want or have a need for developing the actual service. If you are already using SoapUI for testing other Web services, it's a natural fit to use it for mocking as well.

When not to use SoapUI

Do not use SoapUI when you need to implement the actual service. SoapUI is meant for testing and prototyping and, as with all tools, should be used in the context and situation for which it is best suited. Other tools will be better suited for implementing production grade services.

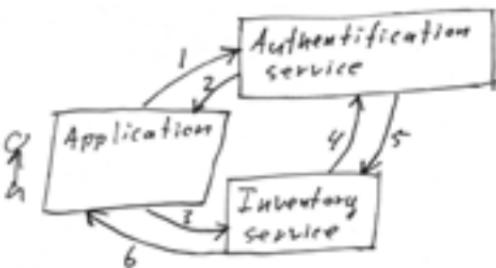
How do I create a mock in SoapUI?

The steps for creating a mock are similar if you have access to the service and want to create a mock to be in control over the availability or if all you have is the contract and sample requests and responses. It may be easier to explore the service you need to interact with if you have access to it, but it is not necessary. In this case I will walk you through an example where we don't have access to the service. All we have is access to a WSDL file describing the contract, a schema that defines

the rules for the contract, and finally a few response samples.

The example

Creating simple examples that are still realistic can be a difficult task. The example below describes a service where you need to identify yourself to add some data into a system. A schematic view could be something similar to this:



The user should authenticate with a username and password in an Authentication service.

1. A token is returned to the application if the user was successful authenticating.
2. The token is then sent to the Inventory service together with a request for adding an item.
3. The Inventory service should validate the request by sending the token to the Authentication service.
4. The Authentication service will tell the Inventory service if the supplied token was valid or not. The request for adding an item should only be executed if the token was verified as valid.
5. The Inventory service will respond to the application with the result of adding an item.

This could be the implementation of a single sign-on system. The user only needs to sign on in one system to get access to all other system. Another example where this would be valid is if a more or less unknown

service on the net wants to know who added a value. The user could then authenticate himself with Facebook and then get access to the service. This is just an example, but it is not an unrealistic one.

What would we need to mock for this example? To start with, we are only interested in mocking the services of the application the user is using. We are not interested in implementing the interaction between third-party services. This reduces the problem somewhat; we only need to handle the calls 1, 2, 3 and 6. The calls 4 and 5 are something we have to expect the third-party services to implement properly. They are out of our reach anyway.

Implementing the example

This example will be implemented using the contract for a service. We don't have access to it. The WSDL file that defines the contract is called "MockExample.wsdl" and looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions
    xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://example.soapui.org/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns="http://schemas.xmlsoap.org/wsdl/"
    targetNamespace="http://example.soapui.org/"
    name="MockExample">
    <types>
        <xsd:schema>
            <xsd:import namespace="http://example.soapui.org/" />
        </xsd:schema>
    </types>
    <message name="login">
        <part name="parameters" element="tns:login"/>
    </message>
    <message name="loginResponse">
        <part name="parameters" element="tns:loginResponse"/>
    </message>
    <message name="InvalidCredentialsException">
```

```

<part name="fault" element="tns:InvalidCredentialsException"/>
</message>
<message name="addItem">
    <part name="parameters" element="tns:addItem"/>
</message>
<message name="addItemResponse">
    <part name="parameters" element="tns:addItemResponse"/>
</message>
<portType name="Service">
    <operation name="login">
        <input wsam:Action="http://example.soapui.org/Service/loginRequest"
               message="tns:login"/>
        <output wsam:Action="http://example.soapui.org/Service/loginResponse"
               message="tns:loginResponse"/>
        <fault message="tns:InvalidCredentialsException" name="InvalidCredentialsException"
              wsam:Action="http://example.soapui.org/Service/login/Fault/InvalidCredentialsException"/>
    </operation>
    <operation name="addItem">
        <input wsam:Action="http://example.soapui.org/Service/addItemRequest"
               message="tns:addItem"/>
        <output wsam:Action="http://example.soapui.org/Service/addItemResponse"
               message="tns:addItemResponse"/>
        <fault message="tns:InvalidSessionIdException" name="InvalidSessionIdException"
              wsam:Action="http://example.soapui.org/Service/addItem/Fault/InvalidSessionIdException"/>
    </operation>
</portType>
<binding name="ServicePortBinding" type="tns:Service">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
    <operation name="login">
        <soap:operation soapAction="" />
        <input>
            <soap:body use="literal" />
        </input>
        <output>
            <soap:body use="literal" />

```

```

        </output>
        <fault name="InvalidCredentialsException">
            <soap:fault name="InvalidCredentialsException"
use="literal"/>
        </fault>
    </operation>
    <operation name="addItem">
        <soap:operation soapAction="" />
        <input>
            <soap:body use="literal" />
        </input>
        <output>
            <soap:body use="literal" />
        </output>
        <fault name="InvalidSessionIdException">
            <soap:fault name="InvalidSessionIdException"
use="literal"/>
        </fault>
    </operation>
</binding>
<service name="MockExample">
    <port name="ServicePort" binding="tns:ServicePortBinding">
        <soap:address location="http://example.soapui.org:80/
MockExample"/>
    </port>
</service>
</definitions>

```

It refers to a schema called “MockExample.xsd” and it is defined as:

```

<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:tns="http://example.soapui.org/"
    xmlns:xss="http://www.w3.org/2001/XMLSchema" version="1.0"
    targetNamespace="http://example.soapui.org/">
    <xs:element name="InvalidCredentialsException" type="tns:Invalid
CredentialsException"/>
    <xs:element name="InvalidItemIdException" type="tns:InvalidItemId
Exception"/>
    <xs:element name="addItem" type="tns:addItem"/>
    <xs:element name="addItemResponse" type="tns:addItemResponse"/>
    <xs:element name="login" type="tns:login"/>
    <xs:element name="loginResponse" type="tns:loginResponse"/>
    <xs:complexType name="InvalidItemIdException">
        <xs:sequence>
            <xs:element name="message" type="xs:string" minOccurs-

```

```

curs="0"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="login">
    <xs:sequence>
        <xs:element name="username" type="xs:string" minOccurs-
curs="0"/>
        <xs:element name="password" type="xs:string" minOccurs-
curs="0"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="loginResponse">
    <xs:sequence>
        <xs:element name="return" type="xs:string" minOccurs-
curs="0"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="InvalidCredentialsException">
    <xs:sequence>
        <xs:element name="message" type="xs:string" minOccurs-
curs="0"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="addItem">
    <xs:sequence>
        <xs:element name="sessionId" type="xs:string" minOccurs-
curs="0"/>
        <xs:element name="name" type="xs:string" minOccurs="0"/>
        <xs:element name="description" type="xs:string" minOccurs-
curs="0"/>
        <xs:element name="cost" type="xs:float"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="addItemResponse">
    <xs:sequence>
        <xs:element name="return" type="xs:boolean"/>
    </xs:sequence>
</xs:complexType>
</xs:schema>

```

These two files define two methods, login and addItem.

Make sure that you have saved a copy of both files in the same directory.

- ◆ Start SoapUI.
- ◆ Create a “New SoapUI project” from the File menu.



Right click on the ServicePortBinding in the new project and select “Generate mock service.”



Click ok and use the default name for the mock “ServicePortBinding MockService.”



We shall start with two different possible responses for the login, “Valid” and “Invalid.” Remove the auto generated response “Response 1.” Right click on “login” and create a response called “Valid” by selecting “New MockResponse.”

A sample of a valid response is generated for you. It will look like this:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:exam="http://example.soapui.org/">
  <soapenv:Header/>
  <soapenv:Body>
    <exam:loginResponse>
      <!--Optional:-->
      <return>?</return>
    </exam:loginResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

We want to set the return value. Let us use the token “valid_token” to indicate that the login was successful. Replace the question mark with valid_token. The response should look like this now:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:exam="http://example.soapui.org/">
  <soapenv:Header/>
  <soapenv:Body>
```

```
<exam:loginResponse>
    <!--Optional:-->
    <return>valid_token</return>
</exam:loginResponse>
</soapenv:Body>
</soapenv:Envelope>
```

Next step is to create an invalid response. Right click on login again and create a new mock response called “Invalid.” Set the return value to “invalid_token” so the response looks like this:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:exam="http://example.soapui.org/">
    <soapenv:Header/>
    <soapenv:Body>
        <exam:loginResponse>
            <!--Optional:-->
            <return>invalid_token</return>
        </exam:loginResponse>
    </soapenv:Body>
</soapenv:Envelope>
```

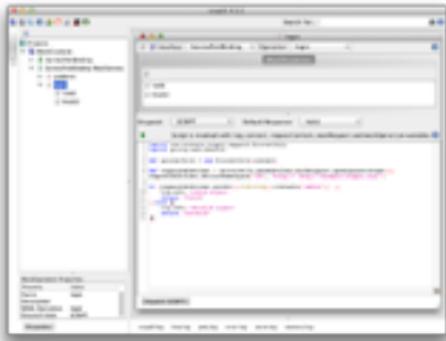
With two different responses available, it is time to add the logic when we should use the “Valid” and when we should use the “Invalid” response. We will implement this using a small Groovy script that looks for values in the login request.

Mark login and press enter to show the MockOperation editor.

Change the Dispatch type to SCRIPT and define the conditions using Groovy. In this case, use the script:

```
import com.eviware.soapui.support.GroovyUtils
import groovy.xml.XmlUtil
def groovyUtils = new GroovyUtils(context)
def requestXmlHolder = groovyUtils.getXmlHolder(mockRequest.getRequestContent())
requestXmlHolder.declareNamespace("ws", "http:// http://example.soapui.org/")
if (requestXmlHolder.getXml().toString().contains("admin")) {
    log.info "Valid Login"
    return "Valid"
} else {
```

```
        log.info "Invalid Login"  
        return "Invalid"  
    }  
  
}
```



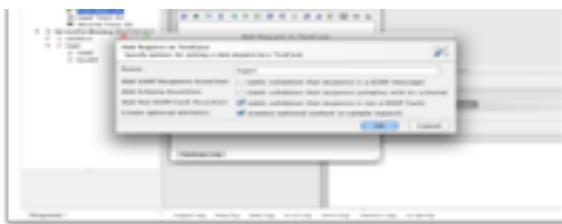
This script will always return a valid response if the request contains “admin.” This may be a bit generous, but remember, this is just an example of how a mock can be created. A better example would have some kind of list with valid user names and valid passwords. We are almost done with our first mock. The next step is to test it.

Test the mock

Create a test suite, right click on the “MockExample” and select “New TestSuite.”



Name the new test suite “Login” and add a test case to it. Right click on the test suite “Login” and select “New TestCase.” Name the new test case “Valid credentials.” Finally, create a test step for this test case. Expand the login suite and valid credentials until you see the test steps. There are none at the moment. Add a test request step by right clicking on “Test Steps.” Call it “logon” and bind it to the method “ServicePortBinding → logon.” Check that “Add Not SOAP Fault Assertion:” is checked and that “Create optional elements:” is checked.



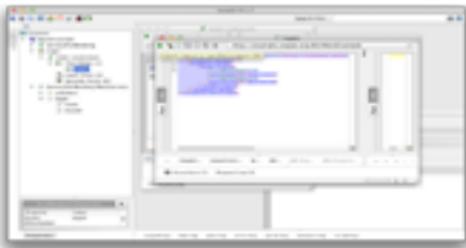
A sample request is created. You need to set some parameters to be able to login.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:exam="http://example.soapui.org/">
    <soapenv:Header/>
    <soapenv:Body>
        <exam:login>
            <!--Optional:-->
            <username?></username>
            <!--Optional:-->
            <password?></password>
        </exam:login>
    </soapenv:Body>
</soapenv:Envelope>
```

You want to change the optional element `username` from a question mark to “admin” and the optional element `password` from a question mark to “secret123.”

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:exam="http://example.soapui.org/">
  <soapenv:Header/>
  <soapenv:Body>
    <exam:login>
      <!--Optional:-->
      <username>admin</username>
      <!--Optional:-->
      <password>secret123</password>
    </exam:login>
  </soapenv:Body>
</soapenv:Envelope>
```

To run this step, click on the arrow in the upper left corner



This will unfortunately fail. There isn't any service running at “`http://example.soapui.org:80/MockExample`.” We need to change where we expect the service to be executed and start the mock to be able to execute the test against it.

Click on the drop down with the address and select the `mockService-PortBinding`. Next step is to start the mock, right click on “`ServicePort-Binding MockService`” and start the mock.

The result should look like this:



We see two things here. First of all, we notice a successful execution of the test step. Secondly, we notice that the mock is running minimized in the bottom of SoapUI.

View the WSDL

The contract for our service can be viewed from a browser. All you need to do is to point the browser to <http://localhost:8088>



Follow the link “ServicePortBinding MockService” to see the WSDL file for this example.



Add an item

The second part of this example is to add an item to the system. It is, however, identical to the first example in that it is all about accepting a request, writing a dispatcher to select the proper response and so forth.

Deploy to an application server

You can save mocks created with SoapUI as a WAR file and deploy on any servlet container. If you want to deploy your mock on Tomcat, all you have to do is to drop a WAR from SoapUI into the Web apps directory.

Right clicking on the SoapUI project and select “Deploy As WAR” creates a WAR. Save the WAR and deploy it as any other Web application in Tomcat.

Conclusion

Creating test doubles, or mocks, of a Web service is something you want to do when you have to be in control over a service during development and testing. Creating a mock using SoapUI is easy if you either have access to the actual service or have the contract for the service.

Mocks should not replace the actual implementation in production.

Mocks should be used during testing to avoid problems with mandatory services.

Mock only the interfaces your application is using, do not mock any expected services between third-party services.

References

1 Definition of Mock, http://dictionary.cambridge.org/dictionary/british/mock_2

3

Hidden Gems in SoapUI Pro

(That you should use today)

1 WSDL Refactoring 2 SQL Query Builder 3 Security Generator



Download Free Trial
SoapUI 14-day evaluation

About SmartBear Software

More than one million developers, testers and operations professionals use SmartBear tools to ensure the quality and performance of their APIs, desktop, mobile, Web and cloud-based applications. SmartBear products are easy to use and deploy, are affordable and available for trial at the website. Learn more about the company's award-winning tools or join the active user community at <http://www.smartbear.com>, on [Facebook](#) or follow us on Twitter [@smartbear](#) and [Google+](#).

