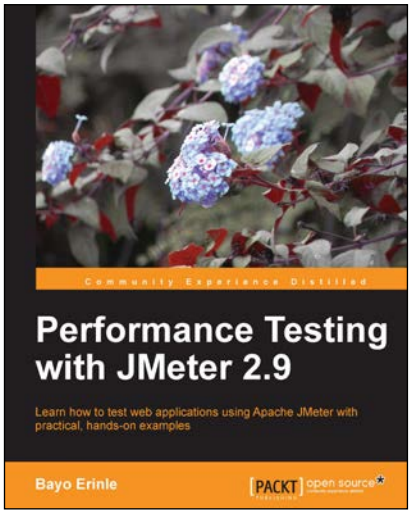
[PACKT] open source
community experience distilled
PUBLISHING

Performance Testing with JMeter 2.9

Bayo Erinle



Chapter No. 3
"Submitting Forms"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.3 "Submitting Forms"

A synopsis of the book’s content

Information on where to buy this book

About the Author

Bayo Erinle is a senior software engineer with over nine years' experience in designing, developing, testing, and architecting software. He has worked in various spectrums of the IT field, including government, finance, and health care. As a result, he has been involved in the planning, development, implementation, integration, and testing of numerous applications, including multi-tiered, standalone, distributed, and cloud-based applications. He is always intrigued by new technology and enjoys learning new things. He currently resides in Maryland, US, and when he is not hacking away at some new technology, he enjoys spending time with his wife Nimota and their three children, Mayowa, Durotimi, and Fisayo.

Performance Testing with JMeter 2.9

Performance Testing with JMeter 2.9 is about a type of testing intended to determine the responsiveness, reliability, throughput, interoperability, and scalability of a system and/or application under a given workload. It is critical and essential to the success of any software product launch and its maintenance. It also plays an integral part in scaling an application out to support a wider user base.

Apache JMeter is a free open source, cross-platform performance testing tool that has been around since the late 90s. It is mature, robust, portable, and highly extensible. It has a large user base and offers lots of plugins to aid testing.

This is a practical hands-on book that focuses on how to leverage Apache JMeter to meet your testing needs. It starts with a quick introduction on performance testing, but quickly moves into engaging topics such as recording test scripts, monitoring system resources, an extensive look at several JMeter components, leveraging the cloud for testing, and extending Apache JMeter capabilities via plugins. Along the way, you will do some scripting, learn and use tools such as Vagrant, Puppet, Apache Tomcat, and be armed with all the knowledge you need to take on your next testing engagement.

Whether you are a developer or tester, this book is sure to give you some valuable knowledge to aid you in attaining success in your future testing endeavors.

What This Book Covers

Chapter 1, Performance Testing Fundamentals, covers the fundamentals of performance testing and the installation and configuration of JMeter.

Chapter 2, Recording Your First Test, dives into recording your first JMeter test script and covers the anatomy of a JMeter test script.

Chapter 3, Submitting Forms, covers form submission in detail. It includes handling various HTML form elements, (checkboxes, radio buttons, file uploads, downloads, and so on), JSON data, and XML.

Chapter 4, Managing Sessions, explains session management, including cookies and URL rewriting.

Chapter 5, Resource Monitoring, dives into active monitoring of system resources while executing tests. You get to start up a server and extend JMeter via plugins.

Chapter 6, Distributed Testing, takes an in-depth look at leveraging the cloud for performance testing. We dive into tools such as Vagrant, Puppet, and AWS.

Chapter 7, Helpful Tips, provides you with helpful techniques and tips for getting the most out of JMeter.

3
Submitting Forms

In this chapter, we'll expand on the foundations we started building in *Chapter 2*, *Recording Your First Test*, and dive deeper into submitting forms in greater detail. While most of the forms you encounter while recording test plans might be simple in nature, some are a whole different beast and require you to pay them more careful attention. For example, more and more websites are embracing **RESTful** web services, and as such, you would mainly interact with JSON objects when recording or executing test plans for such applications. Another area of interest will be recording applications that make use of AJAX heavily to accomplish business functionality. Google, for one, is known to be a mastermind at this. Most of their products, including Search, Gmail, Maps, YouTube, and so on, use AJAX extensively. Occasionally, you might have to deal with XML response data; for example, extracting parts of it to use for samples further down the chain in your test plan. You might also come across cases when you need to upload a file to the server or download one from it.

For all these and more, we will explore some practical examples in this chapter and gain some helpful insights as to how to deal with these scenarios when you come across them as you prepare your test plans.

Capturing simple forms

We have already encountered a variation of form submission in *Chapter 2*, *Recording Your First Test*, when we submitted a login form to authenticate with the server. The form had two text fields for username and password respectively. That's a good start. Most websites requiring authentication will have a similar feel to them. HTML forms, however, span a whole range of other input types. These include checkboxes, radio buttons, select and multiselect drop-down lists, text areas, file uploads, and so on. In this section, we take a look at handling other HTML input types.

We have created a sample application we will be using throughout most of this chapter to illustrate some of the concepts we will be discussing. The application can be reached at `http://jmeterbook.aws.af.cm`. Take a minute to browse around and take it for a manual spin so as to have an idea what the test scripts we record will be doing.

Handling checkboxes

Capturing checkbox submission is similar to that of capturing textbox submissions, which we encountered earlier in *Chapter 2*, *Recording Your First Test*. Depending on the use case, there might be one or more related/unrelated checkboxes on a form. Let's run through a scenario for illustrative purposes. With your JMeter proxy server running and capturing your actions, perform the following steps:

1. Go to `http://jmeterbook.aws.af.cm/form1/create`.
2. Enter a name in the textbox.
3. Check off a hobby or two.
4. Click on **Submit**.

At this point, if you examine the recorded test plan, the `/form1/submit` post request has parameters for the following:

- `name`: This represents the value entered in the textbox
- `hobbies`: You can have one or more depending on the number of hobbies you checked on
- `submit`: This is the value of the **Submit** button

We can then build upon the test plan by adding a CSV Data Set Config to the mix to allow us to feed different values for the names and hobbies (see `handling-checkboxes.jmx`). Finally, we can expand the test plan further by parsing the response from the `/form1/create` sample to determine what hobbies are available on the form using a post processor element (for example, the regular expression extractor) and then randomly choosing one or more of them to submit. I'll leave that as an exercise for the reader. Handling the multiselect option is no different from this.

Handling radio buttons

Radio buttons are normally used as option fields on a web page; that is, they are normally grouped together to present a series of choices to the user, allowing them to select one per group. Things such as marital status, favorite food, and polls are practical uses of them. Capturing their submission is quite similar to dealing with checkboxes, except that we will have just one entry per submission for each radio group. Our sample at `http://jmeterbook.aws.af.cm/radioForm/index` has only one radio group, allowing users to identify their marital status. Hence, after recording this, we will only have one entry submission for a user.

1. Go to `http://jmeterbook.aws.af.cm/radioForm/index`.
2. Enter a name in the textbox.
3. Pick a marital status.
4. Click on **Submit**.

Viewing the HTML source of the page (right-click anywhere on the page and select **View Source**) will normally get you the "IDs" the server is expecting back for each option presented on the page. Armed with that information, we can expand our input test data, allowing us to run this same scenario for more users with varying data. As always, you can use a post-processor component to further eliminate the need to send the radio button IDs in your input feed. Handling a drop-down list is no different to this scenario. Handling all other forms of HTML input types; for example, text and text area fall under the categories we have explored thus far.

Handling file uploads

You may encounter situations where uploading a file to the server is part of the functionality of the system under testing. JMeter can also help in this regard. It comes with a built-in multipart/form-data option on post requests, which is needed by HTML to correctly process file uploads. In addition to checking the option to make a post request multipart, you will need to specify the absolute path of the file, in cases where the file you are uploading is not within JMeter's `bin` directory, or the relative path in cases where the file resides within JMeter's `bin` directory. Let's record a scenario illustrating this:

1. Go to `http://jmeterbook.aws.af.cm/uploadForm`.
2. Enter a name in the textbox.

3. Choose a file to upload by clicking on the **Choose File** button.
4. Click on **Submit**.

Note that files to be uploaded can't be larger than 1 MB.

Depending on the location of the file you choose, you might encounter an error similar to the following:

```
java.io.FileNotFoundException: Argentina.png (No such file or directory)
  at java.io.FileInputStream.open(Native Method)
  at java.io.FileInputStream.<init>(FileInputStream.java:120)
  at org.apache.http.entity.mime.content.FileBody.writeTo(FileBody.
java:92)
  at org.apache.jmeter.protocol.http.sampler.
HTTPHC4Impl$ViewableFileBody.writeTo(HTTPHC4Impl.java:773)
```

Do not be alarmed! This is because JMeter is expecting to find the file in its `bin` directory. You will have to either tweak the file location in the recorded script to point to the absolute path of the file or place it in the `bin` directory or a subdirectory. For the sample packaged with the book, we have opted to place the files in a subdirectory of the `bin` directory (`$JMETER_HOME/bin/samples/images`). Examine the file `handling-file-uploads.jmx`.

Handling file downloads

Another common situation you may encounter will be testing a system that has file download capabilities exposed as a function to its users. Users, for example, might download reports, user manuals, and documentation from a website. Knowing how much strain this can put on the server could be an area of interest to stakeholders. JMeter provides the ability to record and test such scenarios. As an example, let's record a user retrieving a PDF tutorial from JMeter's website.

1. Go to `http://jmeterbook.aws.af.cm/`.
2. Click on the **Handling File Downloads** link.
3. Click on the **Access Log Tutorial** link.

This should stream a PDF file to your browser. You could add a View Results Tree listener and examine the response output after playing back the recording. You could also add a Save Responses to file listener and have JMeter save the contents of the response to a file you can later inspect. This is the route we have opted for in the sample recorded with the book. Files will be created in the `bin` directory of JMeter's installation directory. See `handling-file-downloads-1.jmx`. Also, using a Save Responses to file listener is useful for cases when you would like to capture the response, in this case a file, and feed it to other actions further on in the test scenario. For example, we could have saved the response and used it to upload the file to another section of the same server or a different server entirely.

Posting JSON data

REST (**REpresentational State Transfer**) is a simple stateless architecture that generally runs over HTTP/HTTPS. Requests and responses are built around the transfer of representations of resources. It emphasizes interactions between clients and services by providing a limited number of operations (`GET`, `POST`, `PUT`, and `DELETE`). `GET` fetches the current state of a resource, `POST` creates a new resource, `PUT` updates an existing resource, and `DELETE` destroys the resource. Flexibility is provided by assigning resources their own unique universal resource indicators (URIs). Since each operation has a specific meaning, REST avoids ambiguity. In modern times, the typical object structure passed between client and server is JSON. More information about REST can be found at `http://en.wikipedia.org/wiki/REST`.

When dealing with websites that expose RESTful services in one form or another, you will most likely have to interact with JSON data in some way. Such websites may provide means to create, update, and delete data on the server via posting JSON data. URLs could also be designed to return existing data in JSON format. This happens even more in most modern websites, which use AJAX to an extent, as we use JSON mostly when interacting with AJAX. In all such scenarios, you will need to be able to capture and post data to the server using JMeter. JSON, also known as **JavaScript Object Notation**, is a text-based open standard designed for human readable data interchange. You can find out more information about it at `http://en.wikipedia.org/wiki/JSON` and `http://www.json.org/`. For this book, it will suffice to know what the structure of a JSON object looks like. Here are some examples:

```
{"empNo": 109987, "name": "John Marko", "salary": 65000}
```

And:

```
[{"id":1,"dob":"09-01-1965","firstName":"Barry", "lastName":"White",
"jobs":[{"id":1,"description":"Doctor"}, {"id":2,"description":"Firem
an"}]}]
```

Some basic rules of thumb when dealing with JSON are as follows:

- `[]` – indicates a list of objects
- `{}` – indicates an object definition
- `"key": "value"` – define string values of an object, under a desired key
- `"key": value` – define integer values of an object, under a desired key

So the first example we saw shows an employee object, with employee number `109987`, whose name is `John Marko`, and who earns `$65,000`. The second sample shows a person named `Barry White`, born on `9/1/1965`, who is both a doctor and fireman.

Now that we have covered a sample JSON structure, let's examine how JMeter can help with posting JSON data. The example website provides a URL to save the `Person` object. A person has a first name, last name, and date of birth attributes. In addition, a person can hold multiple jobs. So a valid JSON structure to store a person might look like the following code:

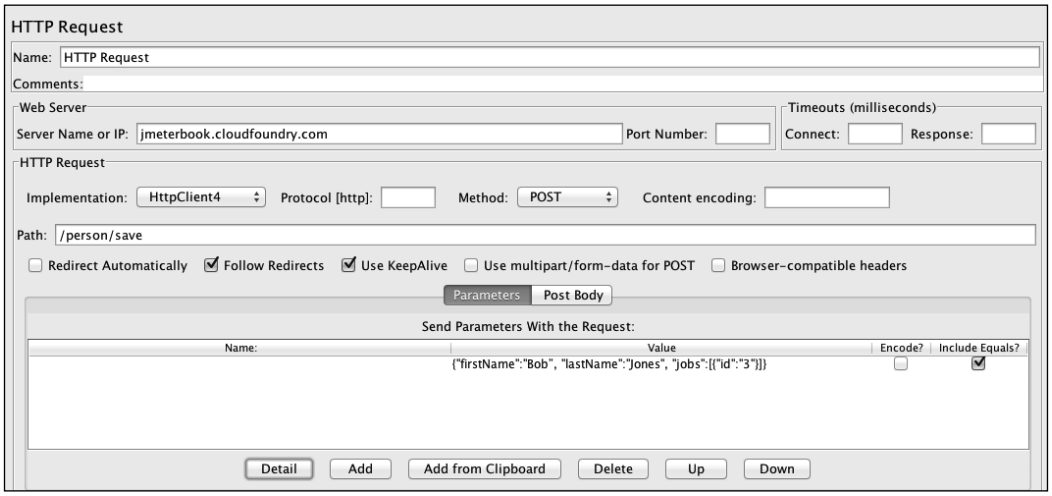
```
{"firstName":"Malcom", "lastName":"Middle", "dob": "2/2/1965",
"jobs":[{"id": 1, "id": 2}]}
{"firstName":"Sarah", "lastName":"Martz", "dob": "3/7/1971"}
```

Instead of recording, we will manually construct the test scenario for this case; we have intentionally not provided a form to save a person's entry so as to give you hands-on practice in writing test plans for such scenarios.

1. Launch JMeter.
2. Add a Thread Group to the **Test Plan** by right-clicking on **Test Plan** and navigate to **Add** | **Threads (Users)** | **Thread Group**.
3. Add a HTTP Request Sampler to the **Thread Group** by right-clicking on **Thread Group** and navigate to **Add** | **Sampler** | **HTTP Request**.
4. Under **HTTP Request**, change **Implementation** to **HttpClient4**.
5. Fill in the properties of the **HTTP Request** Sampler as:
    - **Server Name or IP**: `jmeterbook.aws.af.cm`
    - **Method**: `POST`
    - **Path**: `/person/save`

6. Under **Send Parameters with Request**, click on **Add** and fill in the attributes as follows:
    - **Name**: (leave blank)
    - **Value**: `{"firstName":"Bob", "lastName":"Jones", "jobs":[{"id":"3"}]}`
7. Add an HTTP Header Manager to the **HTTP Request Sampler** (right-click on **HTTP Request Sampler** | **Add** | **Config Element** | **HTTP Header Manager**).
    1. Add an attribute to **HTTP Header Manager** by clicking on it, and clicking on the **Add** button
    - **Name**: `Content-Type`
    - **Value**: `application/json`
8. Add a View Results Tree listener to the **Thread Group** by right-clicking on **Thread Group** and navigate to **Add** | **Listener** | **View Results Tree**.
9. Save the **Test Plan**.

If you have done everything correctly, your **HTTP Request** Sampler should look like the following screenshot:



Configuring the HTTP Request Sampler to post JSON

Now you should be able to run the test, and if all was correctly set, `Bob Jones` should now be saved on the server. You can verify that by examining the View Results Tree listener. The request should be green and in the **Response data** tab, you should see `Bob Jones` listed as one of the entries returned. Even better yet, you could view the last ten stored entries in the browser directly at `http://jmeterbook.aws.af.cm/person/list`.

Of course all other tricks we have learned thus far apply here as well. We can use a CSV Data Config element to parameterize the test and have variation in our data input. See *posting-json.jmx* for that. Regarding input data variation, since `jobs` are optional for this input set, it might make sense to parameterize the whole JSON string read from input feed to give you more variation.

For example, you could replace the value with `${json}`, and have the input CSV Data have entries such as:

```
json
{"firstName":"Malcom", "lastName":"Middle", "dob": "1/2/1971",
"jobs":[{"id": 1, "id": 2}]}
{"firstName":"Sarah", "lastName":"Martz", "dob": "6/9/1982"}
```

We'll leave that as an exercise for you. Although simplistic in nature, what we have covered here should give you all of the information you need to post JSON data when recording your test plans.

When dealing with RESTful requests in general, it helps to have some tools handy to examine requests, inspect responses, and view network latency, among many others. The following is a list of handy tools that could help:

- **Firebug** (an add-on that is available with Firefox, Chrome, and IE): `http://getfirebug.com/`
- **Chrome developer tools**: `https://developers.google.com/chrome-developer-tools/`
- **Advance REST Client** (the Chrome browser extension): `http://bit.ly/15BEKlV`
- **REST Client** (the Firefox browser add-on): `http://mzl.la/h8YMlz`

Reading JSON data

Now that we know how to post JSON data, let's take a brief look at how to consume it in JMeter. Depending on the use case, you might find yourself dealing more with reading JSON than posting it. JMeter provides a number of ways to digest this information, store it if needed, and use it further down the chain in your test plans. Let's start with a simple use case. The example website has a link that provides details of the last ten person entries stored on the server. It is available at `http://jmeterbook.aws.af.cm/person/list`.

If we were to process the JSON response and use the first and last name further down the chain, we could use a Regular Expression Extractor post processor to extract those. Let's create a test plan to do just that.

1. Launch JMeter.
2. Add a Thread Group to the **Test Plan** (right-click on **Test Plan** and navigate to **Add** | **Threads (Users)** | **Thread Group**).
3. Add a HTTP Request Sampler to the **Thread Group** by right-clicking on **Thread Group** and navigate to **Add** | **Sampler** | **HTTP Request**.
4. Under **HTTP Request**, change **Implementation** to **HttpClient4**.
5. Fill in the properties of the **HTTP Request Sampler** as follows:
    - **Server Name or IP**: `jmeterbook.aws.af.cm`
    - **Method**: `GET`
    - **Path**: `/person/list`
6. Add a Regular Expression Extractor as a child of the HTTP Request Sampler by right-clicking on **HTTP Request Sampler** and navigate to **Add** | **Post Processors** | **Regular Expression Extractor**.
7. Fill in the properties as follows:
    - **Reference Name**: `name`
    - **Regular Expression**: `"firstName":"(\w+?)",.+?,"lastName":"(\w+?)"`
    - **Template**: `$1$$2$`
    - **Match No**: `1`
    - **Default Value**: `name`

8. Add a Debug Sampler to the **Thread Group** by right-clicking on **Thread Group** and navigate to **Add** | **Sampler** | **Debug Sampler**.
9. Add a View Results Tree listener to the **Thread Group** by right-clicking on **Thread Group** and navigate to **Add** | **Listener** | **View Results Tree**.
10. Save the **Test Plan**.

The interesting bit here is the cryptic regular expression we are using here. It basically says to match words and store them in the variable defined as `name`. The `\w+?` regular expression instructs the pattern engine not to be greedy when matching and to stop on the first occurrence. The full capabilities of regular expressions are beyond the scope of this book, but we encourage you to master some as they will help you while scripting your scenarios. For now, just believe that it does what it says. Once you execute the test plan, you will be able to see the matches in the debug sampler of the View Results Tree. Here's a snippet of what you should expect to see:

```
name=firstName0lastName0
name_g=2
name_g0="firstName":"Larry","jobs":[{"id":1,"description":"Doctor"}],"
lastName":"Ellison"
name_g1=Larry
name_g2=Ellison
server=jmeterbook.aws.af.cm
```

Now let's shift gears to a more complicated example.

Using the BSF PostProcessor

When dealing with much more complicated JSON structures, you might find that the `Regular Expression Extractor` post processor just doesn't cut it. You might struggle to come up with the right regular expression to extract all the info you need. Examples of that might be deeply nested object graphs that have an embedded list of objects in them. At such times, a **BSF PostProcessor** will fit the bill. **BSF** (**Bean Scripting Framework**) is a set of Java classes that provide scripting language support within Java applications. This opens a whole realm of possibilities, allowing you to leverage the knowledge and power of scripting languages within your test plan while still retaining access to Java class libraries. Scripting languages supported within JMeter at the time of writing include AppleScript, JavaScript, BeanShell, ECMAScript, and Java to name a few. Let's jump right in with an example of querying Google's search service.

1. Launch JMeter.
2. Add a Thread Group to the **Test Plan** by right-clicking on **Test Plan** and navigate to **Add** | **Threads (Users)** | **Thread Group**.
3. Add a HTTP Request Sampler to the **Thread Group** by right-clicking on **Thread Group** and navigate to **Add** | **Sampler** | **HTTP Request**.
4. Under **HTTP Request**, change **Implementation** to **HttpClient4**.
5. Fill in the properties of the **HTTP Request Sampler** as follows:
   - **Server Name or IP**: `ajax.googleapis.com`
   - **Method**: `GET`
   - **Path**: `/ajax/services/search/web?v=1.0&q=Paris%20Hilton`
6. Add a BSF PostProcessor as a child of the **HTTP Request Sampler** by right-clicking on **HTTP Request Sampler** and navigate to **Add** | **Post Processors** | **BSF PostProcessor**:
   1. Pick JavaScript in the Language dropdown list
   2. In the **Scripts** text area, enter this:

```
// Turn the JSON into an object called 'response'
eval('var response = ' + prev.getResponseDataAsString());

// Create a variable called haveBoots_# containing the
number of matching URLs
// For each result, create a variable called haveBoots and
assign it the URL
vars.put("url_cnt", response.responseData.results.length);

//for each result, stop the URL as a JMeter variable
for (var i = 0; i <= response.responseData.results.length;
i++)
{
  var x = response.responseData.results[i];
  vars.put("url_" + i, x.url);
}
```

7. Add a Debug Sampler to the **Thread Group** by right-clicking on **Thread Group** and navigate to **Add** | **Sampler** | **Debug Sampler**.
8. Add a View Results Tree listener to the **Thread Group** by right-clicking on **Thread Group** and navigate to **Add** | **Listener** | **View Results Tree**.
9. Save the **Test Plan**.

Once saved, you can execute the test plan and see the full JSON returned by the request and the extracted values that have now been stored as JMeter variables. If all is correct, you should see values similar to the following:

```
url_0=http://www.parishilton.com/
url_1=http://en.wikipedia.org/wiki/Paris_Hilton
url_2=https://twitter.com/ParisHilton
url_3=http://www.imdb.com/name/nm0385296/
url_cnt=4
```

The BSF PostProcessor exposes a few variables that can be used in your scripts by default. In our preceding example, we have used two of them (`prev` and `var`). `prev` gives access to the previous sample result and `var` gives read/write access to variables. See a list of available variables at `http://jmeter.apache.org/usermanual/component_reference.html#BSF_PostProcessor`.

A quick run down of the code is as follows:

```
eval('var response = ' + prev.getResponseDataAsString());
```

Retrieves the response data of the previous sampler as a string and uses the JavaScript `eval()` function to turn it into a JSON structure. Take a look at the JavaDocs at `http://jmeter.apache.org/api/org/apache/jmeter/samplers/SampleResult.html` to see all the other methods available for the `prev` variable. Once a JSON structure has been extracted, we can call methods like we normally would in JavaScript.

```
vars.put("url_cnt", response.responseData.results.length);
```

This gets the size of the results that were returned and stores the result in a JMeter variable called `url_cnt`. The final bit of code iterates through the results and extracts the actual URLs and stores them into distinct JMeter variables `url_0` through `url_3`.

Handling the XML response

Yet another structure you may encounter as you build test plans is XML. Some websites may hand off XML as their response to certain calls. **XML** (**Extensible Markup Language**) allows you to describe object graphs in a different format than JSON does. For example, we could get our test application to return an XML representation of the person list we were working with earlier in this chapter by making a call to `http://jmeterbook.aws.af.cm/person/list?format=xml`. Describing XML in detail goes beyond the scope of this book, but you can find much more about it online. For our exercise, it will suffice just to know what it looks like. Have a look at the XML returned by the previous link.

Now that you know what XML looks like, let's get going with a sample test plan that deals with retrieving an XML response and extracting variables from it. Have a look at the XML we will be parsing at `http://search.maven.org/remotecontent?filepath=org/springframework/spring-test/3.2.1.RELEASE/spring-test-3.2.1.RELEASE.pom`. Our goal is to extract all the `artifactId` elements (deeply nested within the structure) into variables that we can then use later in our test plan, if we choose.

1. Launch JMeter.
2. Add a Thread Group to the **Test Plan** by right-clicking on **Test Plan** and navigate to **Add** | **Threads (Users)** | **Thread Group**.
3. Add a HTTP Request Sampler to the **Thread Group** byright-clicking on **Thread Group** and navigate to **Add** | **Sampler** | **HTTP Request**.
4. Under **HTTP Request**, change **Implementation** to **HttpClient4**.
5. Fill in the properties of the **HTTP Request Sampler** as follows:
    - **Server Name or IP**: `search.maven.org`
    - **Method**: `GET`
    - **Path**: `/remotecontent?filepath=org/springframework/spring-test/3.2.1.RELEASE/spring-test-3.2.1.RELEASE.pom`
6. Add a Save Responses to a file listener as a child of the **HTTP Request Sampler** by right-clicking on **HTTP Request Sampler** and navigate to **Add** | **Listener** | **Save Responses to a file**.
7. Fill in the properties of the **Save Responses** as follows:
    - **Filename prefix**: `xmlSample_`
    - **Variable name**: `testFile`
8. Add a XPath Extractor as a child of the **HTTP Request Sampler** by right-clicking on **HTTP Request Sampler** and navigate to **Add** | **Post Processors** | **XPath Extractor**.
9. Fill in the properties of the **HTTP Request Sampler** as follows:
    - **Reference name**: `artifact_id`
    - **XPath query**: `project/dependencies/dependency/artifactId`
    - **Default value**: `artifact_id`
10. Add a Debug Sampler to the **Thread Group** by right-clicking on **Thread Group** | **Add** and navigate to **Sampler** | **Debug Sampler**.

11. Add a View Results Tree listener to the **Thread Group** by right-clicking **Thread Group** and navigate to **Add** | **Listener** | **View Results Tree**.
12. Save the **Test Plan**.

Once saved, you will be able to execute the test plan and see the `artifact_id` variables in the View Results Tree listener. The only new element we have used here is the XPath Extractor post processor. This nifty JMeter component allows you to use the XPath query language to extract values from a structured XML or (X)HTML response. As such, we can extract an element deeply nested in the structure with this simple query: `project/dependencies/dependency/artifactId`.

This will look for the tail element (`artifactId`) of the query string within the following structure:

```
<project...>
  ...
  <dependencies>
    <dependency>
      <groupId>javax.activation</groupId>
      <artifactId>activation</artifactId>
      <version>1.1</version>
      <scope>provided</scope>
    </dependency>
    ...
  </dependencies>
</project>
```

This will return `activation`. That is exactly the information we are interested in. Now you know just how to get at the information you need when dealing with XML responses.

Summary

In this chapter, we have gone through the details of how to capture form submission in JMeter. We covered simple forms, with checkboxes and radio buttons. The same concepts covered in those sections can be applied to other input form elements such as text areas and comboboxes. We then explored how to deal with file uploads and downloads when recording test plans. Along the way, we addressed working with JSON data, both posting and consuming it. This exposed us to two powerful and flexible JMeter post processors, Regular Expression Extractor and BSF PostProcessor. Finally, we took a look at how to deal with XML data when we encounter it. For that, we covered yet another post processor JMeter offers, XPath Extractor PostProcessor. You should now be able to use what we have learned so far to accomplish most tasks you need to accomplish with forms while planning and scripting your test plans.

In the next chapter, we will dive into managing sessions with JMeter.

Where to buy this book

You can buy Performance Testing with JMeter 2.9 from the Packt Publishing website: `http://www.packtpub.com/performance-testing-with-jmeter-2-9/book`.

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

www.PacktPub.com

For More Information:
www.packtpub.com/performance-testing-with-jmeter-2-9/book