Chapter No.11

"Monitoring the Performance and Health of GeoServer"
In this package, you will find:

The author’s biography

A preview chapter from the book, Chapter no.11 "Monitoring the Performance and Health of GeoServer"

A synopsis of the book’s content

Information on where to buy this book

About the Author

Colin Henderson is a spatial solutions architect with 14 years of experience working on solutions to complex spatial problems. He is currently the Geospatial Systems Capability Lead for Atkins, one of the world's leading design, engineering, and project management consultancies. Although experienced in a wide range of proprietary GIS software, his current focus is on specializing in the integration of open source software in complex enterprise environments. His most recent projects involve the integration of GeoServer with FME Server from Safe Software and the delivery of spatial web-mapping applications through Esri's ArcGIS for the Server platform and Latitude Geographics' Geocortex framework. Colin is the Technical Architect and Lead Developer of Atkins' open source-based spatial integration platform, CIRRUSmaps™, a solution built on the best breed of open source spatial software, including PostGIS and OpenLayers, with GeoServer at its heart, and designed from the ground-up for deployment in cloud environments.

A self-confessed techie, Colin enjoys digging deeper to understand technology and software, and then applying this learning to create innovative solutions to problems. When possible, he likes to "pay it forward" by helping others with their problems, through contributions on GIS Stack Exchange, in particular.

For More Information:

www.packtpub.com/web-development/angularjs-web-application-development-blueprints
I would like to dedicate this book to the memory of my grandfather, John Denis Stevens, who sadly passed away while I was writing and never got to see the finished product. He often liked to say "these computer things will never catch on!"

I would like to thank my parents for buying me my first computer. I really enjoyed breaking it and then trying to get it working again. Without the introduction, my curiosity of all things computers would never have happened.

Finally, I would like to thank my wife, Amy, and children, Evie and Max, for their patience during the writing process; I love you all very much.
Mastering GeoServer

Since its release in March 2002, GeoServer continued to mature and develop into a sophisticated open source web mapping server. It has a feature set that puts it on par with (some will argue that it even beats) the most popular commercial off-the-shelf web mapping servers. The key to a successful open source software project is to have a strong and talented group of developers and a vibrant community of active and engaged users. GeoServer has both of these key ingredients, which is why it is one of the most popular open source web mapping solutions available today. From large organizations such as Great Britain's national mapping agency (Ordnance Survey) handling large volumes of data to simple small-scale community websites, GeoServer can handle it all.

This book is intended as a natural follow-on from GeoServer Beginner's Guide by Stefano Lacovella and Brian Youngblood, also published by Packt Publishing. It is meant as an advanced guide to GeoServer, and is ideal for when you want to take GeoServer beyond the simple delivery of web maps into advanced uses such as spatial analysis. The book covers a variety of concepts such as installing production-ready and optimized servers, loading and managing spatial data, running complex spatial analysis, and manipulating the output.

What This Book Covers

Chapter 1, Installing GeoServer for Production, examines how GeoServer can be deployed in a production environment. Installation of GeoServer in Apache Tomcat on both Windows and Linux platforms is covered. The chapter ends by looking at different production-deployment architectures for failover and high availability.

Chapter 2, Working with Raster Data, addresses the different types of raster data that can be served by GeoServer and how to optimize them to serve at performance. Increasing the number of raster formats that can be served by implementing the GDAL extension is covered along with an approach to storing and serving very large coverage.

Chapter 3, Working with Vector Data in Spatial Databases, concentrates on storage and serving of vector data from spatial databases. PostGIS, Oracle, and Microsoft SQL Server databases are covered along with the use of SQL Views as layers.

Chapter 4, Using GeoServer to Serve Complex Features, describes how GeoServer can be used to deliver complex-featured schemas as a WFS service. In this chapter, we take a look at the concepts involved in complex schemas, discussing the difference between simple and complex features. To illustrate the concepts, we will take the Open Street Map data and publish it as an INSPIRE Annex I Road Transport Network schema.

For More Information:
www.packtpub.com/web-development/angularjs-web-application-development-blueprints
Chapter 5, *Using GeoServer as a Proxy*, takes a look at using GeoServer's cascaded services to act as a proxy to another WMS and/or WFS server. This capability is a little gem and often underutilized in production. We take a closer look and explore the different reasons why we might like to do it.

Chapter 6, *Controlling the Output of GeoServer*, takes a closer look at the technologies available to allow us to set styling for our layers. The chapter introduces us to the CSS styling extension that allows people familiar with this standard web technology to create gorgeous-looking maps. In addition to looking at how layers can be styled, we will also explore other areas where we can control GeoServer's output, such as responses to WMS GetFeatureInformation requests.

Chapter 7, *Using GeoServer to Print Maps*, tells us that no web mapping server will be complete without the capability to generate printed output. In this chapter, we take a look at the community print extension that adds the capability to generate output to print through a flexible and powerful template capability. We learn how to install and configure the extension as well as create a print template and exploit the capability with an example using OpenLayers.

Chapter 8, *Integrating GeoServer in a Spatial Data Infrastructure*, explores the concepts behind SDI. This chapter shows us how GeoServer can be integrated within a complete production system to provide more than just a means of delivering styled maps for a web application.

Chapter 9, *GeoServer as a Spatial Analysis Platform*, explores the technologies available in GeoServer that allow us to perform server-side spatial analysis. GeoServer, in production, does not have to simply deliver maps for use in web applications. It is a powerful spatial analysis platform in its own right. First, the chapter explores the use of Web Processing Services (WPS), and it then moves on to show us how we can create our own services using GeoScript.

Chapter 10, *Enterprise Security and GeoServer*, demonstrates how GeoServer can be secured within a corporate environment utilizing standard corporate security technologies such as LDAP and Active Directory. Other options to secure GeoServer are also covered to show us how easy it is to lock down our web mapping servers.

Chapter 11, *Monitoring the Performance and Health of GeoServer*, is an important chapter because it shows us the tools available to monitor the health of our GeoServer instances. Maintaining a healthy GeoServer instance is crucial for a good user experience of applications using maps and the data served. This chapter will help us to understand when our servers perform sub optimally.

Chapter 12, *Optimizing GeoServer for Production*, is the final chapter of the book, and this is where we take a last look at the configuration of our server. The chapter goes about providing strategies and options to optimize the configuration of our servers. It also introduces us to some special considerations when running a cluster of GeoServers.

For More Information:  
www.packtpub.com/web-development/angularjs - web - application - development - blueprints
11

Monitoring the Performance and Health of GeoServer

With our GeoServer instances up, running, and secured, it is now time to consider whether they are performing well, and think about how we can keep an eye on them as they start to get utilized.

In this chapter, we will look at what tools exist within GeoServer to enable us to keep an eye on how things are going. We will understand how to configure them, and then look at how we can interpret the information generated to help us make decisions. We will also look at an approach to stress test our GeoServer instances to understand whether they are performing well.

By the end of this chapter, we will have covered the following topics:

- Understanding the importance of monitoring the health and performance of GeoServer
- Installing and configuring the Monitor extension
- Using JMeter to stress test GeoServer
- Understanding information to help make decisions

The importance of monitoring GeoServer

As with all business systems, it is important to monitor the health and performance of GeoServer. There are many reasons why we should do this, but for the key ones, we need to:

- Understand the amount of sustained load our server can handle
- Benchmark performance to monitor our server

For More Information:
www.packtpub.com/web-development/angularjs-web-application-development-blueprints
Monitoring the Performance and Health of GeoServer

- Understand what factors cause our server(s) to crash
- Know when our server is in danger of being overloaded
- Know when our server has been overloaded

There are numerous tools available that allow us to examine various web logfiles and gather statistics on how our server performs generally, but in a spatial context, we also need to gain some understanding of how well our server delivers map data. There are a number of online services that we can register our GeoServer with to perform this kind of monitoring. Services such as MapMeter from Boundless (http://boundlessgeo.com/solutions/mapmeter/) provide commercial monitoring solutions.

However, there are some open source tools that we can use to test and monitor the health of our GeoServer instances. The monitor extension for GeoServer allows us to track requests against GeoServer and store the results in a database where we can analyze them further. We can also use a test tool such as Apache JMeter™ to simulate different loads on our servers.

The GeoServer monitor extension

Installing the monitor extension is as straightforward as all the other GeoServer extensions. However, we have a choice to make about how we want to utilize the extension in production. The choice is in relation to how the data collected by the monitor extension is stored. There are two options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory (default choice)</td>
<td>Monitor data is persisted in memory, only for the last 100 requests</td>
</tr>
<tr>
<td>Hibernate</td>
<td>All the request data is persisted to a database store using Hibernate</td>
</tr>
</tbody>
</table>

The monitor extension configuration controls the storage mechanism; however, we need to consider the options now as it will determine how we install and configure the extension. The choice is a trade-off between having a complete history of the requests made against our GeoServer, but at the expense of storage requirements, versus having a limited history of requests, but without any storage requirement. If we expect our server to get very high volumes of traffic, then we might not want to persist the request information to storage as it will likely fill up very quickly. In a high-transaction environment, we might not be too concerned about historic requests as we will be more interested in how the server performs in real time.

For More Information:
www.packtpub.com/web-development/angularjs-web-application-development-blueprints
If we want to track the usage of our server over time, then there is no escaping the need to persist the request data to a database. For the purposes of this chapter, we will store the request data in a database and use the hibernate option.

## Installing the monitor extension

As we will use the hibernate option for storage, there are two files that we need to grab to install the monitor extension. The first is the core extension modules for monitor, which provides all the capabilities to log request data. The second is the Hibernate extension, which provides the storage mechanisms using the Hibernate ORM library for Java. To this end, we must download the two files and save them to a location on our server.

As with all other extensions, it is important to download the version of the extension matching the version of GeoServer that you installed. We have been using the stable version of GeoServer, which at the time of writing is Version 2.5.2. To download the monitor extension, go to the GeoServer download page for stable at http://www.geoserver.org/release/stable.

---

**Extensions**

<table>
<thead>
<tr>
<th>Vector Formats</th>
<th>Coverage Formats</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>App Schema</td>
<td>CSW</td>
</tr>
<tr>
<td>ArcSDE</td>
<td>WCS 2.0</td>
</tr>
<tr>
<td>CD2</td>
<td>WCS 2.0 CO</td>
</tr>
<tr>
<td>H2</td>
<td>WFS</td>
</tr>
<tr>
<td>MySQL</td>
<td></td>
</tr>
<tr>
<td>Oracle</td>
<td></td>
</tr>
<tr>
<td>Postgresenized Features</td>
<td></td>
</tr>
<tr>
<td>SQL Server</td>
<td></td>
</tr>
<tr>
<td>Terrastat</td>
<td></td>
</tr>
<tr>
<td>VPF</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output Formats</th>
<th>Services</th>
</tr>
</thead>
<tbody>
<tr>
<td>Excel</td>
<td>CSW</td>
</tr>
<tr>
<td>Image Map</td>
<td>WCS 2.0</td>
</tr>
<tr>
<td>OGR</td>
<td>WCS 2.0 CO</td>
</tr>
<tr>
<td>XMLT</td>
<td>WFS</td>
</tr>
<tr>
<td>DXF</td>
<td></td>
</tr>
<tr>
<td>JPEG Turbo</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Miscellaneous</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Chart Symbolizer</td>
<td></td>
</tr>
<tr>
<td>Control Flow</td>
<td></td>
</tr>
<tr>
<td>Cross Layer Filtering</td>
<td></td>
</tr>
<tr>
<td>CSS Styling</td>
<td></td>
</tr>
<tr>
<td>GeoSearch</td>
<td></td>
</tr>
<tr>
<td>CAS</td>
<td></td>
</tr>
</tbody>
</table>

- Monitor (Core, hibernate)
- Reporter (Core, hibernate)
- INSPIRE

---

*For More Information:*

www.packtpub.com/web-development/angularjs-web-application-development-blueprints
Under the Extensions section, there is a subgroup named Miscellaneous. Within this group, look for the Monitor (Core, Hibernate) entry. Click on the Core link to download the Monitor extension, and click on the Hibernate link to download the Hibernate libraries.

With both the modules downloaded, the method of installation is the same for both. We need to open a command line and enter the following two commands:

```bash
$ unzip geoserver-2.5.2-monitor-plugin.zip *.jar -d <tomcat-home>/webapps/geoserver/WEB-INF/lib
$ unzip geoserver-2.5.2-monitor-hibernate-plugin.zip *.jar -d <tomcat-home>/webapps/geoserver/WEB-INF/lib
```

These commands will copy all the extension files into our deployed GeoServer directory. We need to repeat the `unzip` commands for all of our instances of GeoServer, and then restart them all to activate the extension. If the extension installs correctly, then we will see a directory called `monitoring` inside our GeoServer data directory. This directory contains the configuration files for the monitor extension.

### Configuring the monitor extension

Inside the `monitoring` directory, there will be a number of configuration files that we can edit to set the behavior of the monitor extension to our liking. The following table shows the files and their purposes:

<table>
<thead>
<tr>
<th>Configuration file</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>db.properties</code></td>
<td>This provides the database connection settings, if using Hibernate</td>
</tr>
<tr>
<td><code>filter.properties</code></td>
<td>This provides the filter patterns the extension will use to select requests, but not to monitor</td>
</tr>
<tr>
<td><code>hibernate.properties</code></td>
<td>This provides the settings to control the behavior of Hibernate</td>
</tr>
<tr>
<td><code>monitor.properties</code></td>
<td>This provides the settings to control the behavior of the extension itself</td>
</tr>
</tbody>
</table>

The GeoServer documentation provides very good coverage of the configuration options, so we won't cover them all here. We'll head on over to [http://docs.geoserver.org/stable/en/user/extensions/monitoring/index.html](http://docs.geoserver.org/stable/en/user/extensions/monitoring/index.html) for more details. The default configuration files provide a configuration that is good for the vast majority of uses. Therefore, we shall only discuss some of the configuration settings to change how the extension behaves.

For More Information:  
The db.properties file

The `db.properties` file contains the settings to use for database connections. Since we use the hibernate method of storage, we will need to make sure that the configuration settings inside this file are right for our needs. The first choice we need to make is the database that we will use to store the request data. By default, the monitor extension will use a Java H2 database. Hence, it will create some *.db files inside the monitoring directory. For our purposes, we will use a PostgreSQL database to store the request data. We need to change the settings in the file accordingly:

```java
Driver=org.postgresql.Driver
url=jdbc:postgresql://<server-name>:<port>/<db-name>
username=<username>
password=<password>
defaultAutoCommit=<true|false>
```

Most of these settings are self-explanatory as they are standard database connection settings. The key thing to notice is that the database connection is constructed using a JDBC connection string; we just need to replace the `<server-name>`, `<port>`, and `<db-name>` elements with our own values.

The filter.properties file

The `filter.properties` file instructs the monitor extension, which requests not to track. By default, it is configured to filter out all the requests to the web administration interface and monitor request API. The settings that control this are:

```
/web/**
/rest/monitor/**
```

Each line of the file should contain a pattern for a URL to be excluded from monitoring. The pattern should be everything in a standard request URL after `/geoserver`, and up to the ? separator for the start of query parameters. As it is a pattern, it is not necessary to write the entire request URL; for example, the `/web/**` pattern will filter out any requests with `web`. In other words, it will filter out all the web administration interface requests. So, if we do not want the monitor extension to log any requests to our WFS service, then we should add a new line to the file and put the value `/wfs`. 

For More Information:
www.packtpub.com/web-development/angularjs-web-application-development-blueprints
The hibernate.properties file

The hibernate.properties file contains the settings for the Hibernate module, and generally speaking, these settings should not really be changed. The default settings are good enough for all users and should only be changed if we have sufficient knowledge of the Hibernate library. However, we do need to change one setting in this file to make sure that the request data is persisted to our PostgreSQL database. The following two lines are required in the file:

```java
databasePlatform=org.hibernate.dialect.PostgreSQLDialect
database=POSTGRESQL
```

These two settings instruct the Hibernate library to use the PostgreSQL dialect in its database storage calls.

The monitor.properties file

The monitor.properties file contains the properties necessary for the core monitor extension to function. Most of the settings in this file can be left alone, but some should be tweaked. First up, we need to make sure that the storage mode is set to Hibernate:

```java
storage=hibernate
```

If we use the memory approach, then we set the value to memory. The next setting of interest is the one that controls the mode in which the monitor extension operates. The default configuration is:

```java
mode=history
```

The monitor extension can operate in two modes, with the choice determining when the extension persists data to the configured storage. One option is live, and the other is history. In the history mode (the default), the monitor extension persists the information about requests once the processing is complete and the request is fully satisfied. This mode is ideal when real-time information about requests is not necessary. It is worth noting that in this mode, the database is not stressed as much as in the live mode as fewer operations are performed. The live mode persists information about requests as they happen, which means that it is possible to get a real-time view on what the server does. The trade-off is that this mode will place additional strain on the database as it will perform more operations than in the history mode.
Another setting that we might need to consider changing is `maxBodySize`. This determines how much of the request body (in cases of POST requests) is stored. Setting this value too high will cause the storage to fill up quickly, so it is a trade-off between wanting to know details of the request made against how much space is available to store it. It can be useful to unbound this setting (use the value `-1`) when debugging issues with specific requests, but it is highly recommended to set this back to a limit afterwards. It is possible to disable logging of the request body entirely by setting the value to `0`. The value should be the number of characters to capture from the request body; for example, a value of `1024` will capture 1,024 characters.

It is important to make sure that the database body field has the same size as `bodySize` to ensure the data is stored correctly. This is only necessary when using database persistence.

### Checking whether the monitor extension is installed correctly

Once we have the configuration for monitor to our liking, we can go ahead and restart our instances of GeoServer. Once GeoServer restarts, we can check whether everything has been installed and configured correctly by logging in to the web administration console. If everything works as expected, then the left-hand side panel will contain a new section called **Monitor**:

<table>
<thead>
<tr>
<th>Monitor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity</td>
</tr>
<tr>
<td>Reports</td>
</tr>
</tbody>
</table>

Within the **Monitor** section, we should see two options: **Activity** and **Reports**.

---

**For More Information:**

Viewing the monitor extension activity and reports

The monitor extension provides two different views on the request data that it gathers. The first is the Activity view that provides a chart of the requests that have been made over a user-defined period. Clicking on the Activity link will open the Activity page, as shown here:

![Activity Chart]

The tabs across the top of the page allow us to look at different aggregates of data. For example, we can use the Monthly view to see requests over a period of several months. In the preceding example, we can see that between May 1, 2014 and October 23, 2014, there were three spikes of activity. The biggest spike was in June with peak requests of 9,500.

For More Information:

www.packtpub.com/web-development/angularjs-web-application-development-blueprints
The second view that is available is Reports. Clicking on the Reports link will open the Reports page where a number of generated charts can be viewed:

![OWS Request Summary]

The OWS Request Summary report provides a summary of all the requests made against one of the standard OGC services, such as WFS, WMS, and WCS. The Overview tab shows the breakdown of requests across the different services. This information is static, but it provides a good overview of the spread of requests across the different OGC services.

**Going further with the request data**

As useful as the Activity and Reports views are, they do not tell us the whole story of what goes on with our server. The monitor extension captures a lot of information about each request made to GeoServer. If we use the Hibernate component to store the request data in a database (in our case, we use PostgreSQL), then we can also use standard database and analytics tools to gather more information.

For More Information:

www.packtpub.com/web-development/angularjs-web-application-development-blueprints
Monitoring the Performance and Health of GeoServer

For example, we can use the information captured about requests to find the 10 most popular layers. The following SQL for PostgreSQL will give us the answer:

```sql
SELECT a.name, COUNT(*) AS requests
FROM request_resources a, request b
WHERE a.request_id = b.id
AND b.service = 'WMS'
AND b.operation = 'GetMap'
GROUP BY a.name
ORDER BY requests DESC
LIMIT 10;
```

This `SELECT` statement joins the `request` table with the `request_resources` table using the `id` key, and then groups on `name` and provides count. The `ORDER BY a.name DESC` statement puts them into descending order and `LIMIT 10` returns us the 10 highest counts.

Another possibility, and arguably more interesting, is the ability to put the extents of each map request onto a map! To do this, we can make use of the SQL View functionality that we discovered in Chapter 3, Working with Vector Data in Spatial Databases. We can use PostGIS' geometry constructors to turn the bounding box information captured with each request into a geometry that we can show on the map. The following SQL will generate the geometry for each request:

```sql
SELECT *
, ST_SetSRID(ST_MakeBox2D(ST_MakePoint(minx, miny), ST_MakePoint(maxx, maxy))::geometry, trim(leading 'EPSG:' from crs)::integer)
FROM request
WHERE service = 'WMS' AND operation = 'GetMap'
```
This `SELECT` statement uses the `ST_SetSRID`, `ST_MakeBox2D`, and `ST_MakePoint` functions to generate polygons representing the bounds of each `GetMap` request on the WMS service. We can use this `SELECT` statement in `Edit SQL View`, as shown in the following screenshot:

![Edit SQL view](image)

This will create a layer showing us where the requests for our data are being made; we might be able to discern spatial patterns about map requests. When we start to get at the raw monitoring data, we can really learn a lot about the performance of our server as well as the usage pattern over time.

**Stress testing GeoServer**

With the monitoring extension installed, we now need to consider how we can go about testing the performance of our GeoServer, and then use the logs to analyze the results. To test the performance of our GeoServer instances, we will need a tool that will allow us to send multiple requests to map images and data in different locations and resolutions. We also need to be able to change the flow of requests and simulate multiple concurrent requests.

The web is full of tools that will allow us to simulate different mixes of requests to our servers to test their performance. When it comes to testing the performance of our GeoServer instance, the tool we choose to use is less important than the mix of requests that we decide to send to it. For the purposes of this chapter, we will use the Apache JMeter™ desktop application to simulate a heavy load on our servers.

**For More Information:**

www.packtpub.com/web-development/angularjs-web-application-development-blueprints
The Apache JMeter™ software is very capable and has a much broader scope for use than the example that we will use in this book. To learn more about Apache JMeter™, it is highly recommended that you read the documentation at http://jmeter.apache.org/usermanual/index.html.

To successfully stress test our GeoServer instances, we will have to complete a few configuration tasks first:

1. Generate test WMS bounding box requests.
2. Prepare an Apache JMeter™ workbench.
3. Execute the test.

Generating test WMS bounding boxes

Before we can begin to stress test our GeoServer instances, we need to come up with some WMS request parameters that we can use. If we can make the requests random within a given set of bounds, then so much the better. Fortunately for us, such a script exists in the world of Python. Frank Warmerdam created a Python script that will randomly generate WMS request parameters for random locations within a specified bounding box and at a range of zoom levels. The script itself can be downloaded from http://svn.osgeo.org/osgeo/foss4g/benchmarking/wms/2010/scripts/wms_request.py. The script accepts a number of parameters to generate the random bounding boxes:


The command-line options that can be specified are:

<table>
<thead>
<tr>
<th>Option</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>This option specifies the number of requests to randomly generate</td>
</tr>
<tr>
<td>region</td>
<td>This option specifies the bounding box of the region to randomly generate requests in</td>
</tr>
<tr>
<td>minres</td>
<td>This option specifies the minimum resolution to use for requests</td>
</tr>
<tr>
<td>maxres</td>
<td>This option specifies the maximum resolution to use for requests</td>
</tr>
<tr>
<td>maxsize</td>
<td>This option specifies the maximum size for a request, expressed as width and height in pixels</td>
</tr>
</tbody>
</table>

For More Information:

www.packtpub.com/web-development/angularjs-web-application-development-blueprints
We will generate 100 random WMS requests to give ourselves a good spread of requests. Perform the following steps:

1. Make sure that Python is installed before running this command.
2. Execute the following from a command-line shell:
   
   ```
   $ python wms_request.py -count 100 -region 0 0 700000 1300000 -minsize 400 400 -maxsize 800 800 -minres 1 -maxres 15 –srs 27700
   ```
3. This command will generate 100 WMS requests in CSV format inside the bounds of the British National Grid. Here are five examples generated from the command:
   ```
   493;595;126140.02,124150.64,130761.48,124708.41
   678;495;400974.82,1089828.8,403242.09,1091484.1
   582;543;633121.29,750590.72,638605.35,755707.28
   652;400;17348.862,386404.58,23686.324,390292.59
   416;481;471897.49,430970.91,472413.77,431567.85
   ```
4. The command will generate the CSV file using the value of `srs` as the filename, so in our case, this will be `27700.csv`. The file is stored in the same location as the command was run.
5. Before we can use this inside Apache JMeter™, we will have to tweak it a little bit. Notice how the first component of CSV contains a string similar to:
   ```
   493;595;126140.02
   ```
6. The first two components are the width and height for the WMS image request. We need to turn these into CSV components by replacing `;` with `,`
7. Load the CSV file into your favorite text editor and do a find and replace.
8. Save the file.

For More Information:

www.packtpub.com/web-development/angularjs-web-application-development-blueprints
Creating an Apache JMeter™ test workbench

Before we can get into the nitty-gritty of creating a test workbench for Apache JMeter™, we must download and install it. Apache JMeter™ is a 100 percent Java application, which means that it will run on any platform provided there is a Java 6 or higher runtime environment present. The binaries can be downloaded from http://jmeter.apache.org/download_jmeter.cgi, and at the time of writing, the latest version is 2.11. No installation is required; just download the ZIP file and decompress it to a location you can access from a command-line prompt or shell environment.

To launch JMeter on Linux, simply open shell and enter the following command:

```
$ cd <path_to_jmeter>/bin
$ ./jmeter
```

To launch JMeter on Windows, simply open a command prompt and enter the following command:

```
C:> cd <path_to_jmeter>\bin
C:> jmeter
```

After a short time, JMeter GUI should appear, where we can construct our test plan.

For ease and convenience, consider setting your system’s PATH environment variable to the location of the JMeter bin directory. In future, you will be able to launch JMeter from the command line without having to CD first.

The JMeter workbench will open with an empty configuration ready for us to construct our test strategy:

For More Information:  
www.packtpub.com/web-development/angularjs-web-application-development-blueprints
The first thing we need to do is give our test plan a name; for now, let's call it GeoServer Stress Test. We can also provide some comments, which is good practice as it will help us remember for what reason we devised the test plan in future.
To demonstrate the use of JMeter, we will create a very simple test plan. In this test plan, we will simulate a certain number of users hitting our GeoServer concurrently and requesting maps. To set this up, we first need to add **Thread Group** to our test plan. In a JMeter test, a thread is equivalent to a user:

![JMeter Test Plan](image)

In the left-hand side menu, we need to right-click on the **GeoServer Stress Test** node and choose the **Add | Threads (Users) | Thread Group** menu option. This will add a child node to the test plan that we right-clicked on. The right-hand side panel provides options that we can set for the thread group to control how the user requests are executed. For example, we can name it something meaningful, such as **Web Map Requests**.

For More Information:  
www.packtpub.com/web-development/angularjs-web-application-development-blueprints
In this test, we will simulate 30 users, making map requests over a total duration of 10 minutes, with a 10-second delay between each user starting. The number of users is set by entering a value for Number of Threads; in this case, 30. The Ramp-Up Period option controls the delay in starting each user by specifying the duration in which all the threads must start. So, in our case, we enter a duration of 300 seconds, which means all 30 users will be started by the end of 300 seconds. This equates to a 10-second delay between starting threads \( \frac{300}{30} = 10 \). Finally, we will set a duration for the test to run over by ticking the box for Scheduler, and then specifying a value of 600 seconds for Duration. By specifying a duration value, we override the End Time setting.
Next, we need to provide some basic configuration elements for our test. First, we need to set the default parameters for all web requests.

Right-click on the **Web Map Requests** thread group node that we just created, and then navigate to **Add | Config Element | User Defined Variables**. This will add a new node in which we can specify the default HTTP request parameters for our test:

---

**User Defined Variables**

**Name:** MMS Request Defaults

**Comments:** Default MMS parameters common to all tests

<table>
<thead>
<tr>
<th>User Defined Variables</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>srs</td>
<td>EPSG:27700</td>
</tr>
<tr>
<td>format</td>
<td>imagejpeg</td>
</tr>
<tr>
<td>MMS</td>
<td>Services to apply</td>
</tr>
<tr>
<td>layers</td>
<td>OS/Geo, CSM, Police.uk_streetCrime</td>
</tr>
<tr>
<td>service</td>
<td>MMS</td>
</tr>
<tr>
<td>request</td>
<td>GetMap</td>
</tr>
<tr>
<td>version</td>
<td>1.1.1</td>
</tr>
</tbody>
</table>

---

For More Information:

In the right-hand side panel, we can specify any number of variables. We can use these as replacement tokens later when we configure the web requests that will be sent during our test run. In this panel, we specify all the standard WMS query parameters that we don't anticipate changing across requests. Taking this approach is a good practice as it means that we can create a mix of tests using the same values, so if we change one, we don't have to change all the different test elements.

To execute requests, we need to add Logic Controller. JMeter contains a lot of different logic controllers, but in this instance, we will use Simple Controller to execute a request. To add the controller, right-click on the Web Map Requests node and navigate to Add | Logic Controller | Simple Controller. A simple controller does not require any configuration; it is merely a container for activities we want to execute. In our case, we want the controller to read some data from our CSV file, and then execute an HTTP request to WMS. To do this, we need to add a CSV dataset configuration. Right-click on the Simple Controller node and navigate to Add | Config Element | CSV Data Set Config.

The settings for the CSV data are pretty straightforward. The filename is set to the file that we generated previously, containing the random WMS request properties. The path can be specified as relative or absolute. The Variable Names property is where we specify the structure of the CSV file. The Recycle on EOF option is important as it means that the CSV file will be re-read when the end of the file is reached. Finally, we need to set Sharing mode to All threads to ensure the data can be used across threads.

For More Information:  
www.packtpub.com/web-development/angularjs-web-application-development-blueprints
Next, we need to add a delay to our requests to simulate user activity; in this case, we will introduce a small delay of five seconds to simulate a user performing a map-pan operation. Right-click on the **Simple Controller** node, and then navigate to **Add** | **Timer** | **Constant Timer**:

<table>
<thead>
<tr>
<th>Constant Timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name: Constant Timer</td>
</tr>
<tr>
<td>Comments: Introduce a small delay between requests to simulate typical map-panning operations</td>
</tr>
<tr>
<td>Thread Delay (in milliseconds): $5000</td>
</tr>
</tbody>
</table>

Simply specify the value we want the thread to be paused for in milliseconds.

Finally, we need to add a JMeter sampler, which is the unit that will actually perform the HTTP request. Right-click on the **Simple Controller** node and navigate to **Add** | **Sampler** | **HTTP Request**. This will add an **HTTP Request** sampler to the test plan:
There is a lot of information that goes into this panel; however, all it does is construct an HTTP request that the thread will execute. We specify the server name or IP address along with the HTTP method to use. The important part of this panel is the Parameters tab, which is where we need to specify all the WMS request parameters. Notice that we used the tokens that we specified in the CSV Data Set Config and WMS Request Defaults configuration components. We use the \texttt{${token\_name}} token, and JMeter replaces the token with the appropriate value of the referenced variable.

We configured our test plan, but before we execute it, we need to add some listeners to the plan. A JMeter listener is the component that will gather the information from all the test runs that occur. We add listeners by right-clicking on the thread group node and then navigating to the Add | Listeners menu option. A list of available listeners is displayed, and we can select the one we want to add. For our purposes, we will add the Graph Results, Generate Summary Results, Summary Report, and Response Time Graph listeners. Each listener can have its output saved to a datafile for later review. When completed, our test plan structure should look like the following:

Before executing the plan, we should save it for use later.

**Choosing where to execute tests**

Before we go ahead and execute our test plan, we must consider from where we will run JMeter. Choosing where to run JMeter is important as it will impact how we interpret the results from the test. Broadly speaking, there are three locations we can choose to run JMeter:

- On the same server as GeoServer
- On a server on the same network as GeoServer
- On a web-connected desktop/laptop/server outside of the GeoServer network

For More Information:

www.packtpub.com/web-development/angularjs-web-application-development-blueprints
Monitoring the Performance and Health of GeoServer

The first option is useful for when we want to simply test our GeoServer’s performance at rendering map images; in other words, the pure response time from GeoServer. Without the variability of network contention getting in the way (because we will call the local host), we will get a good appreciation of the rendering time of our GeoServer instance.

However, this option is no good if we have GeoServer on a "headless" server that can’t display GUI. It is also worth noting that JMeter will itself consume resources, memory and CPU in particular, which can affect the validity of results. In this case, the second option will be the next best thing. By testing on the same network as the GeoServer instance, we will get a realistic reflection of the map image-rendering time of GeoServer, with only a minimal amount of overhead from the network contention.

The third option is the one that we will choose if we want to see how well our GeoServer instance performs in a real-world scenario. By running JMeter from a web-connected client, we will introduce real-world performance issues such as Internet congestion. The results should give us a realistic view of the sort of performance our users can expect to see.

**Executing the test profile**

Executing the test plan is very straightforward. We just need to hit the big green arrow button on the toolbar. Once we do this, the test will begin its execution, and we can examine the listeners as the test is in progress to see what happens in real time:

For More Information:

www.packtpub.com/web-development/angularjs-web-application-development-blueprints
In this case, we can see the **Graph Results** listener plotting the results of the test as it runs. Notice that we didn't yet reach our maximum number of users running on the server. The number on the far right of the toolbar shows us how many threads of the total are running; in this case, there are only 21 out of the maximum 30 threads running. As the test progresses, this number will gradually increase at a rate of one every 10 seconds, as we specified in our **Ramp Up** property.

We included the **Graph Results** listener in our test plan to demonstrate the visualization of requests. However, in practice, when doing a load test such as this, we will not actually include the **Graph Results** listener, as it is resource intensive.

---

**Analyzing the results of the stress test**

Once the test run is complete, we will be left with a range of results to examine. The listeners we add to our test plan will determine the kind of information we can analyze. The key information for us to analyze from our test run are **Summary Report** and **Response Time Graph**.

As its name suggests, **Summary Report** provides us with a summary of the responses received during the test. From it, we can ascertain details such as the average, minimum, and maximum response times for requests as well as an indication of the throughput of our server. Throughput is a good indicator of how well our server will handle the load as it tells us how many requests per second it can handle:

<table>
<thead>
<tr>
<th>Label</th>
<th># Samples</th>
<th>Average</th>
<th>Min</th>
<th>Max</th>
<th>Std Dev</th>
<th>Error %</th>
<th>Throughput</th>
<th>Kbps</th>
<th>Avg. Odds</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetReq HTTP</td>
<td>5186</td>
<td>2554</td>
<td>35</td>
<td>4046</td>
<td>263.05</td>
<td>0.01%</td>
<td>8.7 reqs</td>
<td>659.21</td>
<td>77803.7</td>
</tr>
<tr>
<td>TOTAL</td>
<td>5186</td>
<td>2554</td>
<td>35</td>
<td>4046</td>
<td>263.05</td>
<td>0.01%</td>
<td>8.7 reqs</td>
<td>659.21</td>
<td>77803.7</td>
</tr>
</tbody>
</table>

The summary report gives us a very good baseline measure that we can use to compare results against. For example, the results from this test were performed on a newly configured server without any users accessing it. They show that this particular server can handle a throughput of 8.7 requests per second. We can take this as a baseline indicator of what we can expect our server to do, and we can then do regular monitoring and tests to see if the server starts to deviate from this baseline significantly. If it does, then that will be an indication that there is a potential problem with the server or its configuration.
The **Response Time Graph** output is also a useful indicator of how well our server is processing data, and if we deploy JMeter in a web-connected client, then it will be a good indicator of the real-world response times that we can reasonably expect our server to deliver.

The graph can be structured over specific intervals to see how response times vary. If we make the graph match our ramp-up values, we can expect to see spikes in the response times every 10 seconds as a new user connects to the server. Over time, we can expect to see the server settle down and start to become a little more consistent in its response times. In an ideal world, we would like to see a graph that is relatively flat as that will indicate a server that is optimized well and not susceptible to spikes in load.

Of course, just as JMeter provided us all the information during the stress test, we also logged all the requests through the monitor extension. We can now go back to GeoServer and take a look at the activity and reports, or go direct to the database and run some queries to see how well GeoServer responded to the requests.
Summary

In this chapter, we discussed the reasons why we should monitor the health of our GeoServer instances. We explored how to install and configure GeoServer's monitor extension so that GeoServer logs the map requests made to it. We also looked at how Apache JMeter™ can be used to construct and execute test plans to place loads on our servers so that we can analyze the results and gain an understanding of how well our servers perform.

Performance testing is the key to having an optimized, responsive, and healthy GeoServer implementation. The real art of testing is devising tests in such a way that they are representative of the real-world scenario in which we think our server will be used. For example, it is not much use to simply set the test to create 100 concurrent users immediately and start hammering the server. By gradually introducing the load, and introducing small delays between each request to simulate user habits, we will get a much better reflection of how our server will perform in a real-world setting.

The formula is very simple: well-performing GeoServer = happy users!

Now we know how we can run performance tests and monitor the condition of our GeoServer instance, it is time to look at how we can further optimize it. In the next chapter, we will look at some different ways to further optimize our instances.
Where to buy this book

You can buy Mastering GeoServer from the Packt Publishing website:

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.