GeoServer Cookbook

Stefano Iacovella

Chapter No. 1
"Working with Vectors"
In this package, you will find:
The author’s biography
A preview chapter from the book, Chapter no.1 "Working with Vectors"
A synopsis of the book’s content
Information on where to buy this book

About the Author

Stefano Iacovella is a long-time GIS developer and consultant living in Rome, Italy. He also works as a GIS instructor.

He has a PhD in Geology. Being a very curious person, he developed a deep knowledge of IT technologies, mainly focusing on GIS software and related standards.

Starting his career as an ESRI employee, he was exposed to and became confident in proprietary GIS software, mainly the ESRI suite of products.

In the last 14 years, he has been involved with open source software, integrating it with proprietary software. He loves the open source approach, and really trusts in collaboration and sharing knowledge. He strongly believes in the open source idea and constantly manages to spread it, not only in the GIS sector but also in others.

He has been using GeoServer since the release of Version 1.5, configuring, deploying, and hacking it in several projects. Some of the other GFOSS projects he mainly uses and likes are GDAL/OGR, PostGIS, QGIS, and OpenLayers.

Prior to this cookbook, Stefano worked on GeoServer Beginner's Guide, which covers the basics of GeoServer.

For More Information:
www.packtpub.com/hardware-and-creative/geoserver-cookbook
When not playing with maps and geometric shapes, he loves reading about science (mainly physics and math), riding his bike, and having fun with his wife and two daughters, Alice and Luisa. You can contact him at stefano.iacovella@gmail.com or follow him on Twitter at @iacovellas.

Acknowledgments

I would like to thank all those people who helped me make this book a real thing.

A special mention for GeoServer's developers; they are the wonderful engine without which this software, and hence, this book too, would not exist.

I would like to thank all the people at Packt Publishing who worked on this book. They helped me a lot and were very patient and encouraging when I had difficulties meeting deadlines.

A special thanks to my technical reviewers; they constantly checked for my errors and omissions. This book has become better thanks to all their hard work.

Last but not least, I want to express my gratitude to Alessandra, Alice, and Luisa for their support and patience when I was working on this book.

For More Information:
www.packtpub.com/hardware-and-creative/geoserver-cookbook
GeoServer Cookbook

Until a decade ago, spatial data and the art of map building were considered tricky and complex—almost reserved for highly skilled and specialized professionals.

The advent of web mapping has changed the way geography is perceived by people. This has been largely powered by a few types of open source software that made it possible for everyone to collect, manage, and publish spatial data on the Internet.

GeoServer is one of these precious gems. Along with open JavaScript frameworks such as OpenLayers and Leaflet, it gives everyone the tools they need to create a powerful map server and lightweight applications that run in modern browsers. This allows users to show data visualizations, create online data editors, and do much more.

GeoServer is a complete and powerful type of server software that can publish data to a web mapping application. In recent releases, support for geoprocessing operations has been added, which lets you use GeoServer as a data processor.

Since the beginning, GeoServer has support to standards from Open Geospatial Consortium (OGC), which is kept compatible with its recent most relevant releases (for more information on OGC, have a look at http://www.opengeospatial.org).

This book will guide you through the details of configuring data for publication, creating geoprocessing tasks, and optimizing your server for optimal performance.

What This Book Covers

Chapter 1, Working with Vectors, discusses vector data publication with WFS. It covers how to use the data with a JavaScript client. It also explores how to use parametric SQL views and how to enhance performance with feature generalization.

Chapter 2, Working with Rasters, explores the different raster data types you can publish with GeoServer. It also covers the mosaic data creation and adding support for more formats that integrates GDAL libraries in your server.

Chapter 3, Advanced Styling, deals with the art of representing data on a map. Using the CSS module, you will explore how to create symbols for vector data and how to build a renderer for raster data.

Chapter 4, Geoprocessing, teaches you how to create data processing tasks on GeoServer. By following the recipes, you'll understand how you can use GeoServer to create complex server-side functions and use them in your client.

Chapter 5, Advanced Configurations, deals with some advanced configuration tasks. You'll explore database connection optimization, configuring the cache to improve performance, and Spatial Reference System (SRS) optimization.

For More Information:  
www.packtpub.com/hardware-and-creative/geoserver-cookbook
Chapter 6, *Automating GeoServer Configurations*, explores the GeoServer REST interface. Using the operation published in that interface, you will discover how to create automatic tasks to update your site's configuration.

Chapter 7, *Advanced Visualizations*, focuses on some visualization techniques. You will explore how to add time support in WMS, create animated maps, and export data to Google Earth.

Chapter 8, *Monitoring and Tuning*, teaches you the use of the control flow and how to monitor extensions for GeoServer. With the tools provided, you can control how the requests are filtered and queued to your server. You will learn how to save information about users' requests and analyze them to build reports.

Appendix, *OGC for ESRI Professionals*, discusses a brief comparison between GeoServer and ArcGIS for Server, a map server created by ESRI. The importance of adopting OGC standards when building a geographical information system is stressed. You will learn how OGC standards lets you create a system where different pieces of software cooperate with each other.

For More Information:
www.packtpub.com/hardware-and-creative/geoserver-cookbook
1

Working with Vectors

In this chapter, we will cover the following recipes:

- Using different WFS versions in OpenLayers
- Using WFS nonspatial filters
- Using WFS spatial filters
- Using WFS vendor parameters
- Filtering data with CQL
- Filtering data with CQL spatial operators
- Creating a SQL view
- Creating a parametric view
- Improving performance with pregeneralized features

Introduction

Vector data is probably the main source of spatial information that is used inside GeoServer to build maps. You may use the data both to render maps on the server side, that is, using the **Web Map Service (WMS)** standard, or have the client get the shapes and manipulate or render them in a map, that is, using the **Web Feature Service (WFS)** standard.

In this chapter, we will use both these standards, and we will focus on how to filter data and optimize configuration for better performance. We assume that you’re already comfortable with the standard WMS and WFS requests and you know how to configure a data store and a layer with the GeoServer web interface.

For More Information:

www.packtpub.com/hardware-and-creative/geoserver-cookbook
The recipes in this chapter use a few datasets. Configuring and publishing them is quite easy, so we are not covering these steps in detail. We will use publicly available data from NASA Earth Observatory (http://neo.sci.gsfc.nasa.gov) and Natural Earth (http://www.naturalearthdata.com). Configuration and publication of datasets is straightforward, and hence not covered in detail.

You should download the Blue Marble dataset from NASA Earth Observatory. In the home page, you can find it by navigating to the LAND section. Use the GeoTiff format with 0.1 degrees resolution to match the exercises provided in this book. You should publish this dataset as NaturalEarth:blueMarble for use with exercises that require a map that looks like the one in this book.

You also need a couple of datasets from Natural Earth: the datasets for countries and populated places in the 1:10,000,000 scale. Go to http://www.naturalearthdata.com/downloads/10m-cultural-vectors/ and download the datasets for countries and populated places in the shapefile format. Publish the countries' data as NaturalEarth:countries.

We will be using the populated places dataset to create a SQL view. To be able to create it, you should load the data in a spatial RDBMS. Our choice is PostGIS, as it is a very good option; it is powerful, easy to deploy, and free.

We won't cover how to install and configure a PostGIS installation. In fact, PostGIS is not an RDBMS, but a spatial plugin for PostgreSQL. So, you should first install the latter and then add the former. If this sounds new and somehow complicated for you, there are a lot of nice guides on the Internet that you can use for a quick start.

The procedure to install on Linux can be found at: http://trac.osgeo.org/postgis/wiki/UsersWikiInstall

For Windows, a good choice is downloading the binary packaged by Enterprise DB: http://www.enterprisedb.com/products-services-training/pgdownload#windows

---

**Using different WFS versions in OpenLayers**

You are probably comfortable with WMS and including WMS layers in a web application. When you need more control over your data, it's time to switch to WFS.

Unlike WMS, a WFS request brings you the actual raw data in the form of features. By working with the features directly, you are no more dealing with a static rendering of features, that is, a map; you can take fine control of the shapes by setting drawing rules on the client side.

---

For More Information:

www.packtpub.com/hardware-and-creative/geoserver-cookbook
WFS comes in several different flavors or, more precisely, in different protocol versions. GeoServer supports 1.0.0, 1.1.0, and 2.0.0. Each version differs in the formats supported and the query capabilities supported. For example, WFS 2.0.0 supports the use of temporal queries and joins.

If you are curious about the details of WFS and GML, check out the reference documents for WFS and GML in the OGC repository at http://www.opengeospatial.org/standards/is. Here, look for the following files:

- OpenGIS® Geography Markup Language Encoding Standard (GML)
- OpenGIS® Web Feature Service (WFS) Implementation Specification

The following screenshot shows the map you're aiming for:

We will build the code step by step. For your reference, check out the code bundle provided with the book and have a look at the ch01_wfsVersion.html and ch01_wfsVersion101.html files.

How to do it...

1. Create an HTML file and insert the following code snippet:

```html
<html>
<head>
  <title>Dealing with WFS version</title>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <body onload="init()"> 
    <div id="myMap"></div>
  </body>
</html>
```

2. Create a style for the map, defining the size and aspect:

```css
#myMap {
  clear: both;
  position: relative;
  width: 750px;
  height: 450px;
  border: 1px solid black;
}
```

3. Insert a reference to the OpenLayers library:

```html
<script type="text/javascript" src="http://openlayers.org/api/2.13.1/OpenLayers.js"></script>
```

4. Now start coding in JavaScript and add new map objects:

```javascript
function init() {
  map = new OpenLayers.Map({
    div: "myMap",
    // We don't want any layers as base map
    allOverlays: true,
  }
```

For More Information:

www.packtpub.com/hardware-and-creative/geoserver-cookbook
5. Add the Blue Marble layer, which is a standard WMS layer:

   ```javascript
   layers: [
     new OpenLayers.Layer.WMS("Blue Marble",
       "http://localhost/geoserver/wms",
       {layers: "NaturalEarth:blueMarble", format: "image/png", transparent: false}
   ),
   
   6. Now, add a vector layer using the WFS protocol:

   ```javascript
   new OpenLayers.Layer.Vector("countries", {
     strategies: [new OpenLayers.Strategy.BBOX()],
     protocol: new OpenLayers.Protocol.WFS({
       url: "http://localhost/geoserver/wfs",
       featureType: "countries",
       featureNS: "http://www.naturalearthdata.com/",
       // Mind the geometry column name
       geometryName: "geom"
     }),
   
   7. Insert a style to render the vector features:

   ```javascript
   styleMap: new OpenLayers.StyleMap({
     strokeWidth: 3,
     strokeColor: "#FF0000",
     strokeWidth: 1,
     fillColor: "#ee9900",
     fillOpacity: 0.3
   }},
   
   8. Zoom in to the map and center it on Europe:

   ```javascript
   center: [12.48, 42.60],
   zoom: 4
   
   map.addControl(new
   OpenLayers.Control.LayerSwitcher());

   </script>
   </head>

9. Save the file in a folder published on your server, such as TOMCAT_HOME/webapps/ROOT, and point your browser to it. You should get a map that looks like the one shown in the introduction to this recipe.

For More Information:
www.packtpub.com/hardware-and-creative/geoserver-cookbook
10. Now, switch the request to another version. We will use v1.1.0 and the JSON format:

```javascript
new OpenLayers.Layer.Vector("countries", {
    strategies: [new OpenLayers.Strategy.BBOX()],
    protocol: new OpenLayers.Protocol.WFS.v1_1_0({
        url: "http://localhost:8080/geoserver/wfs",
        featureType: "countries",
        featureNS: "http://www.naturalearthdata.com/",
        geometryName: "geom",
        outputFormat: "JSON",
    })
}),
```

11. Reload your document inside the browser and check that your map looks the same.

**How it works...**

We are using a JavaScript application to perform a WFS query on GeoServer. This is a common use case in the era of web mapping, besides using the OpenLayers framework to assist you in building a complicated request.

The HTML and CSS part of the script is quite easy. As you must have noticed, the core of this little program is the `init()` function, which is called at page loading.

We first create a `map` object and set the `allOverlays` variable to `true`. The default value of `false` makes it mandatory for a layer to be `basemap`; in this recipe, we don't want to have `basemap`, which is a layer that is always turned on in the map:

```javascript
allOverlays: true,
```

Then, we start to add data on the map. First, we use the raster data from the NASA Blue Marble dataset. We use the `OpenLayers.Layer.WMS` class; you just need to set a name and URL for the WMS service. The `format` and `transparent` parameters are optional, but they let you control the file produced by GeoServer. The code is as follows:

```javascript
layers: [
    new OpenLayers.Layer.WMS("Blue Marble",
        "http://localhost/geoserver/wms",
        {layers: "NaturalEarth:blueMarble", format:
            "image/png", transparent: false}
    ),
],
```

While we are using a raster dataset in this request, you can, of course, use vector data in the WMS request.

---

For More Information:

Then we create a new layer using the `OpenLayers.Layer.Vector` class, and this layer can use a different source data format:

```javascript
new OpenLayers.Layer.Vector("countries", {

We add a strategy, `BBOX`, to let `OpenLayers` query the server for data intersecting the current map extent:

```javascript
strategies: [new OpenLayers.Strategy.BBOX()],
```

With the `protocol` parameter, we set the data format. Of course, we use WFS, and this class defaults to the 1.0.0 version of the standard:

```javascript
protocol: new OpenLayers.Protocol.WFS({

We need to set some mandatory parameters when invoking the constructor for the WFS class. The `geometryName` parameter is optional, but it defaults to `the_geom` value. So, you need to set it to the actual name of the geometry column in your data. The code is as follows:

```javascript
url: "http://localhost/geoserver/wfs",
featureType: "countries",
featureNS: "http://www.naturalearthdata.com/",
geometryName: "geom"
```

WFS returns raw data, not an image map like WMS. So, you need to draw each feature yourself; you have to set some rules for feature drawing. Inside the `StyleMap` class, you set the color, line width, and other rendering parameters that will be used to represent features in the map, as shown in the following code:

```javascript
styleMap: new OpenLayers.StyleMap({
  strokeWidth: 3,
  strokeColor: "#FF0000",
  strokeWidth: 1,
  fillColor: "#ee9900",
  fillOpacity: 0.3
```

What happens when you load this app in your browser? You can use a browser extension to check the actual request sent to GeoServer.

![Firebug](https://example.com/firebug.png)

*Firebug is a powerful extension for FireFox, and with Chrome, you can use the developer console.*
Using FireFox with Firebug, you should see a few requests upon loading the ch01_wfsVersion.html file. OpenLayers executes the POST WFS request with our parameters; you can see that the version is 1.0.0, the operation is GetFeature, and there is a bounding box filter defined in GML 2:

```
<?xml version="1.0" encoding="UTF-8"?>
  <wfs:Query typeName="feature-countries" xmlns:feature="http://www.naturalsearchdata.com/">
    <wfs:Filter xmlns:ogc="http://www.opengis.net/ogc">
      <ogc:BBOX>
        <ogc:PropertyName>geom</ogc:PropertyName>
        <gml:Envelope xmlns:gml="http://www.opengis.net/gml"
          srsName="EPSG:4326">
          <gml:lowerCorner>-53.43796875 3.04921875</gml:lowerCorner>
          <gml:upperCorner>78.39796875 82.15078125</gml:upperCorner>
        </gml:Envelope>
      </ogc:BBOX>
    </wfs:Filter>
  </wfs:Query>
</wfs:GetFeature>
```

Now, try to load the ch01_wfsVersion110.html file; the request is a little bit different. Of course, now the version is 1.1.0, but the filter looks different as well:

```
<?xml version="1.0" encoding="UTF-8"?>
  <wfs:Query typeName="feature-countries" xmlns:feature="http://www.naturalsearchdata.com/">
    <wfs:Filter xmlns:ogc="http://www.opengis.net/ogc">
      <ogc:BBOX>
        <ogc:PropertyName>geom</ogc:PropertyName>
        <gml:Envelope xmlns:gml="http://www.opengis.net/gml"
          srsName="EPSG:4326">
          <gml:lowerCorner>-53.43796875 3.04921875</gml:lowerCorner>
          <gml:upperCorner>78.39796875 82.15078125</gml:upperCorner>
        </gml:Envelope>
      </ogc:BBOX>
    </wfs:Filter>
  </wfs:Query>
</wfs:GetFeature>
```

You need to be aware that WFS 1.1.0 uses GML 3, which uses a different representation of geometry. In this case, OpenLayers hides the complexity of creating the correct geometry filter.

There's more...

You probably noted when downloading the Blue Marble dataset that the GeoTIFF file is quite a big file. To render this file, GeoServer must navigate the file contents and read blocks of pixels off the disk. To optimize data storage and enhance rendering speed, you can use GDAL tools to restructure the contents of the file for faster access.

For More Information:
www.packtpub.com/hardware-and-creative/geoserver-cookbook
If you have GDAL tools at your fingertips, you can check the metadata of the file:

```bash
$ gdalinfo BlueMarbleNG-TB_2004-12-01_rgb_3600x1800.TIFF
```

**Metadata:**

- AREA_OR_POINT=Area
- TIFFTAG_RESOLUTIONUNIT=1 (unitless)
- TIFFTAG_XRESOLUTION=1
- TIFFTAG_YRESOLUTION=1

**Image Structure Metadata:**

- INTERLEAVE=PIXEL

Now, let's transform the file using a compression method to reduce the file size and tile the dataset for faster access:

```bash
$ gdal_translate -of GTiff -co COMPRESS=DEFLATE -co TILED=YES BlueMarbleNG-TB_2004-12-01_rgb_3600x1800.TIFF blueMarble.tiff
```

Tiling organizes the file contents on disk into tiles, ideally locating blocks of pixels next to each other on disk. This optimization helps in performance when GeoServer is zoomed in to a small area of interest.

Then, we will add an overview to further hasten the data extraction:

```bash
$ gdaladdo -r cubic -ro blueMarble.tiff 2 4 8 16 32 64
```

An overview creates a small *summary* image, which can be used by GeoServer when zoomed out. By drawing using the overview, GeoServer can read fewer pixels off disk and avoid having to sample through the entire file.

By executing the `gdalinfo` tool again, you can check that these have actually been applied successfully:

```bash
...
```

**Metadata:**

- AREA_OR_POINT=Area
- TIFFTAG_RESOLUTIONUNIT=1 (unitless)
- TIFFTAG_XRESOLUTION=1
- TIFFTAG_YRESOLUTION=1

**Image Structure Metadata:**

- COMPRESSION=DEFLATE
- INTERLEAVE=PIXEL

For More Information:

www.packtpub.com/hardware-and-creative/geoserver-cookbook
Using WFS nonspatial filters

In the previous recipe, we just specified a layer to get features from. The OpenLayers strategy, BBOX, created a spatial filter to get only features intersecting the map extent. For common use, you may want to create filters yourself in order to extract specific sets of features from GeoServer.

In this recipe, we will build a map with four layers, each one containing countries according to their mean income. The data source for each layer is the countries' feature type, and we will apply different filters on them.

The resulting map looks like this:

For More Information:
www.packtpub.com/hardware-and-creative/geoserver-cookbook
How to do it...

1. Use the ch01_wfsVersion.html file of the previous recipe; rename it as wfsFilter.html in the same folder. Then, edit the JavaScript part as shown in the following code:

   ```javascript
   function init() {
     map = new OpenLayers.Map({
       div: "myMap",
       allOverlays: true,
     });
   }
   ```

2. Remove the Blue Marble layer; we will have only WFS layers here:

   ```javascript
   layers: [
     new OpenLayers.Layer.Vector("Low income Countries",
     {
       strategies: [new OpenLayers.Strategy.BBOX()],
       protocol: new OpenLayers.Protocol.WFS({
         url: "http://localhost/geoserver/wfs",
         featureType: "countries",
         featureNS: "http://www.naturalearthdata.com/",
         geometryName: "geom"
       })
     }),
   ],
   ```

3. Change the style for the first request; the features will be drawn with a pale brown fill and a dark outline:

   ```javascript
   styleMap: new OpenLayers.StyleMap({
     strokeWidth: 3,
     strokeColor: "#000000",
     strokeWidth: 1,
     fillColor: "#ffffff",
     fillOpacity: 1
   }),
   ```

4. Now, add a filter to only have low income countries in this layer:

   ```javascript
   filter: new OpenLayers.Filter.Logical({
     type: OpenLayers.Filter.Logical.AND,
     filters: [
       new OpenLayers.Filter.Comparison({
         type: OpenLayers.Filter.Comparison.EQUAL_TO,
       }),
     ]
   }),
   ```

For More Information:
www.packtpub.com/hardware-and-creative/geoserver-cookbook
property: "income_grp",
value: "5. Low income"
}),

]}
})
})

5. We will repeat the previous steps for three other layers. Name the first as Lower middle income Countries:

    new OpenLayers.Layer.Vector("Lower middle income Countries", { 

6. The source of the layer is obviously the same; change the style with a fill as shown in the following line of code:

        fillColor: "#c2e699",

7. Then, change the filter to extract countries with a proper value:

        new OpenLayers.Filter.Comparison({
            type:
                OpenLayers.Filter.Comparison.EQUAL_TO,
            property: "income_grp",
            value: "4. Lower middle income"
        }),

8. Now, add a new layer for the upper-middle income countries; the only modified line of code is shown here:

        new OpenLayers.Layer.Vector("Upper middle income Countries", { 

        ...        fillColor: "#78c679",

        ...

        new OpenLayers.Filter.Comparison({
            type:
                OpenLayers.Filter.Comparison.EQUAL_TO,
            property: "income_grp",
            value: "3. Upper middle income"
        }),

9. Eventually, add a last layer for high income countries:

        new OpenLayers.Layer.Vector("High income Countries", { 

        ...        fillColor: "#238443",

For More Information:
www.packtpub.com/hardware-and-creative/geoserver-cookbook
10. The Filter is a bit more complex as we have two different values for high income countries:

```javascript
filter: new OpenLayers.Filter.Logical({
  type: OpenLayers.Filter.Logical.OR,
  filters: [
    new OpenLayers.Filter.Comparison({
      type: OpenLayers.Filter.Comparison.EQUAL_TO,
      property: "income_grp",
      value: "1. High income: OECD"
    }),
    new OpenLayers.Filter.Comparison({
      type: OpenLayers.Filter.Comparison.EQUAL_TO,
      property: "income_grp",
      value: "2. High income: nonOECD"
    })
  ]
})
```

11. Save the file and point your browser to it. You should get a map that looks like the one shown in the introduction to this recipe.

How it works...

The first part of the script is quite similar to that used in the previous recipe. We create a Map object and start adding layers to it.

To have the first layer containing only low income countries, we need to set a filter:

```javascript
filter: new OpenLayers.Filter.Logical({
```

The filter might contain more criteria, so we need to specify a logical operator to join more criteria. This is required only with a single-criteria filter, as shown in the following code:

```javascript
type: OpenLayers.Filter.Logical.AND,
```

Then, we set the filter type. In this case, an equality type, that is, only records with the value we specify, will be selected:

```javascript
filters: [
  new OpenLayers.Filter.Comparison({
    type: OpenLayers.Filter.Comparison.EQUAL_TO,
```

For More Information:

www.packtpub.com/hardware-and-creative/geoserver-cookbook
Eventually, we need to set the attributes on which the filter will be applied and the value to use for filtering records:

```json
    property: "income_grp",
    value: "5. Low income"
  }},
]}
```

To create the other three layers, we clone the same filter by setting a different value. We create four different layers from the same feature type.

Of course, we need to set a different style for each layer to have a proper distinct representation.

### Using WFS spatial filters

Filtering alphanumerical attributes is quite a common task. However, in a GIS application, you may also want to filter features according to geometric properties.

WFS includes a few spatial relationships that you can use to create a spatial filter. From a general point of view, you need an input shape, a relationship to be checked, and some target shapes to be filtered.

In this recipe, we use the `DWITHIN` spatial relationship to filter countries that are within a circular buffer.

For More Information:

You can find the full source code for this recipe in the code bundle available from the Packt site for this book. Look for the `ch01_wfsSpatialFilter.html` file.

How to do it...

1. Copy the file used in the first recipe to the `wfsSpatialFilter.html` file in the same folder. Then, alter the JavaScript part as shown in the following code:

```javascript
function init() {
    map = new OpenLayers.Map({
        div: "myMap",
        allOverlays: true,
        layers: [
            new OpenLayers.Layer.Vector("Filtered Countries", {
                strategies: [new OpenLayers.Strategy.Fixed()],
                protocol: new OpenLayers.Protocol.WFS({
                    url: "http://localhost/geoserver/wfs",
                    featureType: "countries",
                    featureNS: "http://www.naturalearthdata.com/",
                    geometryName: "geom"
                }),
                styleMap: new OpenLayers.StyleMap({
                    strokeWidth: 3,
                    strokeColor: "#000000",
                    strokeWidth: 1,
                    fillColor: "#78c679",
                    fillOpacity: 1
                })
            })
        ]
    });

2. Insert a spatial filter, as shown in the following code:

```javascript
filter: new OpenLayers.Filter.Logical({
    type: OpenLayers.Filter.Logical.AND,
    filters: [
        new OpenLayers.Filter.Spatial({
            type: OpenLayers.Filter.Spatial.DWITHIN,
            value: new OpenLayers.Geometry.Point(12, 42),
            distance: 8
        })
    ]
})
```
Working with Vectors

3. Save the file and point your browser to it. You should get a map that looks like the one shown in the introduction to this recipe.

How it works...

Not surprisingly, the code contained in the file is not so different from that used in the previous recipe. Indeed, we are performing the same task, which is filtering data, but now we want to use a different filter: a spatial filter.

You set a logical filter with an AND logical join:

```javascript
    filter: new OpenLayers.Filter.Logical(
        type: OpenLayers.Filter.Logical.AND,
    );
```

Then, you add a Filter.Spatial class. This is the magic of this recipe. The value parameter lets you choose the spatial relationship; in this case, we use DWITHIN, that is, all features within a specific distance from the source geometry will be selected:

```javascript
    filters: [
        new OpenLayers.Filter.Spatial( 
            type: OpenLayers.Filter.Spatial.DWITHIN,
        )
    ];
```

The source geometry is a point feature created with an OpenLayers class and specifies the latitude and longitude values:

```javascript
    value: new OpenLayers.Geometry.Point(12, 42),
```

Then, you have to set a distance. Please note that you can also set zero as a distance value; in this case, if you have a polygon geometry, only the feature that is contained in the polygon will be selected:

```javascript
    distance: 8
```

## Using WFS vendor parameters

The previous recipes used standard WFS requests. GeoServer also supports a few optional parameters that you can include in your requests. In this recipe, we will see how to ask GeoServer, which is reprojecting the data from the native SRS to another SRS, to use a vendor parameter.

Reprojection of data is a part of WFS 1.1.0 and 2.0.0, and GeoServer has provided support since 1.0.0 so that you can use it with any WFS version. The following screenshot is what we’re targeting in this recipe:

![Diagram](image-url)

You can find the full source code for this recipe in the code bundle available from the Packt site for this book; look for the `ch01_wfsReprojection.html` file.

For More Information:
Working with Vectors

How to do it...

1. Copy the file used in the first recipe to the wfsReprojection.html file in the same folder. Insert a new parameter for the Map object:
   ```javascript
   projection: "EPSG:3857",
   ```

2. Then, alter the JavaScript part when creating the WFS layer:
   ```javascript
   new OpenLayers.Layer.Vector("countries", {
     strategies: [new OpenLayers.Strategy.BBOX()],
     protocol: new OpenLayers.Protocol.WFS({
       url: "http://localhost/geoserver/wfs",
       featureType: "countries",
       featureNS: "http://www.naturalearthdata.com/",
       geometryName: "geom",
     }),
   
   srsName: new OpenLayers.Projection("EPSG:3857"),
   srsNameInQuery: true
   }),
   ```

3. Add a parameter to request data reprojection:
   ```javascript
   srsName: new OpenLayers.Projection("EPSG:3857"),
   srsNameInQuery: true
   ```

4. Save the file and point your browser to it. You should get a map that looks like the one shown in the introduction to this recipe.

How it works...

Using a vendor parameter is really straightforward; you just add it to your request. In our recipe, we want to use the countries' data that is stored in the EPSG:4326 projection, which is the geographical coordinates, and in EPSG:3857, which is the planar coordinates. First of all, we set the spatial reference system for the map:

```javascript
map = new OpenLayers.Map({
  div: "myMap",
  allOverlays: true,
  projection: "EPSG:3857",
});
```

Then, we create the WFS request for data by inserting the srsName parameter and assigning it the same projected coordinate system used for the map. The Boolean parameter srsNameInQuery is really important, as it defaults to false. If you don't set it, OpenLayers will not ask for reprojection when using WFS 1.0.0:

```javascript
srsName: new OpenLayers.Projection("EPSG:3857"),
srsNameInQuery: true
```
Let's see what happens when you load the page in your browser and the OpenLayers framework creates the WFS request. Use Firebug to capture the XML code sent to GeoServer with the GetFeature request. The Query element contains the srsName parameter that forces GeoServer to project data:

```xml
<wfs:Query typeName="feature:countries_1" srsName="EPSG:3857"
 xmlns:feature="http://www.naturalearthdata.com/"
><ogc:Filter xmlns:ogc="http://www.opengis.net/ogc">
  <ogc:BBOX>
    <ogc:PropertyName>geom</ogc:PropertyName>
    <gml:Box xmlns:gml="http://www.opengis.net/gml"
      srsName="EPSG:3857">
      <gml:coordinates decimal="." cs="," ts=" ">
        13325909.428711,-3545545.6572266
        16025909.428711,14065545.657227
      </gml:coordinates>
    </gml:Box>
  </ogc:BBOX>
</ogc:Filter>
</wfs:Query>
```

### Filtering data with CQL

Another vendor parameter is `cql_filter`. It allows users to add filters to requests using **Extended Common Query Language (ECQL)**.

In the previous recipes, you created filters using the OGC filter XML standard. ECQL lets you create filters in an easier text format and in a much more compact way. A CQL filter is a list of phrases similar to the `where` clauses in SQL, each separated by the combiner words AND or OR.

CQL was originally used for catalog systems. GeoServer uses an extension for CQL, allowing the full representation of OGC filters in the text form. This extension is called ECQL.

ECQL lets you use several operators and functions. For more information, you can read the documents available at the following URLs:

- [http://docs.codehaus.org/display/GEOTOOLS/ECQL+Parser+Design](http://docs.codehaus.org/display/GEOTOOLS/ECQL+Parser+Design)

**For More Information:**
In this recipe, we will create a map with a filter on `income_grp`, which is very similar to the previous one. Your result should look like the following screenshot:

You can find the full source code for this recipe in the `ch01_wmsCQLFilter.html` file.

**How to do it...**

1. Copy the file used in the first recipe to a new file and name it `wfsCQLFilter.html` in the same folder. Insert a new `sld` variable and populate it as shown in the following lines:

```html
<script type="text/javascript" src="http://openlayers.org/api/2.13.1/OpenLayers.js"></script>
<script type="text/javascript">
function init() {

var sld = '<StyledLayerDescriptor version="1.0.0">';

sld+= '<NamedLayer>
  <Name>NaturalEarth:countries</Name>
  <UserStyle>
    <IsDefault>1</IsDefault>
    <FeatureTypeStyle>

For More Information:
www.packtpub.com/hardware-and-creative/geoserver-cookbook
2. After the Blue Marble layer, add a new WMS layer:

```javascript
  layers: "NaturalEarth:countries",
  format: "image/png",
  transparent: true,
  CQL_FILTER: "income_grp = '1. High income: OECD' OR income_grp = '2. High income: nonOECD'",
  sld_body: sld
});
```

3. Save the file and point your browser to it. You should get a map that looks like the one shown in the introduction to this recipe.

---

How it works...

As with the vendor parameter for reprojection, the use of CQL filters is really easy. You can add one when you create the `Layer` object and insert the textual representation of the filter, that is, a string. In this filter, we want to select high income countries; two different values match the condition, so we use the logical operator OR to join them:

```
CQL_FILTER: "income_grp = '1. High income: OECD' OR income_grp = '2. High income: nonOECD'",
```
Then, we add an `sld_body` parameter and assign the content of the variable `sld` to it. This is not a filter requirement; we just want to override the default style for the countries' layers, so we use the WMS option to send a **Styled Layer Descriptor (SLD)** description of drawing rules to GeoServer:

```
sld_body: sld
```

Of course, we need to create an actual SLD document and insert it into the `sld` variable before creating the countries layer. We do this by adding a well-formed XML code line by line to the `sld` variable. You will note that we're creating exactly the same symbology used in the *Using WFS spatial filters* recipe:

```
var sld = '<StyledLayerDescriptor version="1.0.0">';
...
   sld+= '<PolygonSymbolizer'>;
   sld+= '<Stroke'>;
   sld+= '<CssParameter name="stroke">#000000</CssParameter>';  
   sld+= '<CssParameter name="stroke-width">1</CssParameter>';   
   sld+= '</Stroke>';  
   sld+= '<Fill'>;
   sld+= '<CssParameter name="fill">#FFFFCC</CssParameter>';  
   sld+= '<CssParameter name="fill-opacity">0.65</CssParameter>';   
   sld+= '</Fill>';  
   sld+= '</PolygonSymbolizer>';  
...
```

This is just a small peek at SLD. If you are curious about the standard, you can find the official papers for SLD at [http://portal.opengeospatial.org/files/?artifact_id=22364](http://portal.opengeospatial.org/files/?artifact_id=22364) and the XSD schemas at [http://schemas.opengis.net/sld/](http://schemas.opengis.net/sld/).

### Filtering data with CQL spatial operators

ECQL does not only let you create readable and powerful filters on feature attributes; obviously, it also lets you filter out geometric properties.

There are a few spatial operators that you can use in a CQL filter: **EQUALS, DISJOINT, INTERSECTS, TOUCHES, CROSSES, WITHIN, CONTAINS, OVERLAPS, RELATE, DWITHIN, and BEYOND**. With all these operators at your fingertips, you can really build complex filters.
In this recipe, we will use the **BEYOND** operator that is the inverse of **DWITHIN**, which we used in the recipe *Using WFS spatial filters*. With the filter, we will select the populated places located at least 1,000 km away from Rome, Italy. The result is shown in the following screenshot:

![Screenshot](image.png)

You can find the full source code for this recipe in the `ch01_wmsCQLSpatialFilter.html` file.

**How to do it...**

1. Create a projected table with populated places in PostGIS:
   ```sql
   gisdata=> CREATE TABLE populatedplaceswm AS SELECT name,
              ST_Transform(geom,3857) AS geom FROM populatedplaces;
   gisdata=> CREATE INDEX populatedplaceswm_0_geom_gist ON
              populatedplaceswm USING gist(geom);
   ```
2. Have a look at the latitude and longitude values for Rome using the following lines of code:

```sql
gisdata=> select ST_AsText(geom) from populatedplaces where name = 'Rome';
-----------------------------------------
POINT(12.481312562874 41.8979014850989)
```

3. Copy `wmsCQLFilter.html` to `wmsCQLSpatialFilter.html` and edit the JavaScript code. Change `sld` to `PointSymbolizer` with a red fill for points outside the buffer:

```html
sld+= '<PointSymbolizer'>
sld+= '<Graphic'>
sld+= '<Mark'>
sld+= '<WellKnownName>circle</WellKnownName>'
sld+= '<Fill'>
sld+= '<CssParameter name="fill">#FF0000</CssParameter>'
sld+= '</Fill'>
sld+= '</Mark'>
sld+= '<Size>3</Size>'
sld+= '</Graphic'>
sld+= '</PointSymbolizer>';
```

4. Add an `sld2` variable that holds drawing rules for points contained in the buffer area. You can reuse the code for the `sld` variable, changing the fill color to green:

```html
sld2+= '<PointSymbolizer'>
sld2+= '<Graphic'>
sld2+= '<Mark'>
sld2+= '<WellKnownName>circle</WellKnownName>'
sld2+= '<Fill'>
sld2+= '<CssParameter name="fill">#00FF00</CssParameter>'
sld2+= '</Fill'>
sld2+= '</Mark'>
sld2+= '<Size>3</Size>'
sld2+= '</Graphic'>
sld2+= '</PointSymbolizer>';
```

5. Set the SRS for the map to `EPSG:3857`:

```javascript
map = new OpenLayers.Map({
  div: "myMap",
  allOverlays: true,
  projection: "EPSG:3857",
  maxExtent: new OpenLayers.Bounds(-20037508, -20037508, 20037508, 20037508.34),
  maxResolution: 156543.0339,
  units: 'm',
```

For More Information:

www.packtpub.com/hardware-and-creative/geoserver-cookbook
6. Add a layer with a CQL filter for points located outside a 1,000-kilometre circular buffer from Rome:

```javascript
new OpenLayers.Layer.WMS("Far away places",
  "http://localhost/geoserver/wms",{
    layers: "NaturalEarth:populatedplaceswm",
    format: "image/png",
    transparent: true,
    cql_filter: "BEYOND(geom,POINT(1389413 5145697),1000000,meters)",
    sld_body: sld,
  })
```

7. Add a layer with a CQL filter for points located inside a 1,000-kilometre circular buffer from Rome:

```javascript
new OpenLayers.Layer.WMS("Near places",
  "http://localhost/geoserver/wms",{
    layers: "NaturalEarth:populatedplaceswm",
    format: "image/png",
    transparent: true,
    cql_filter: "DWITHIN(geom,POINT(1389413 5145697),1000000,meters)",
    sld_body: sld2,
  })
```

8. Save the file and point your browser to it. Your map will show a set of green points around Rome. As shown in the previous image, Rome is surrounded by red points.

**How it works...**

Using spatial operators in CQL filter is not really different than filtering other attributes. We used a map in a planar coordinate, in this case, the Web Mercator projection, because of an issue within GeoServer. Although you can define units that will be used to calculate distance in the BEYOND and DWITHIN operators, the interpretation depends on the data store (see http://jira.codehaus.org/browse/GEOS-937 for a detailed discussion). In PostGIS, the distance is evaluated according to the default units for the native spatial reference system of the data.

A spatial reference system (SRS) is a coordinate-based local, regional, or global system used to locate geographical entities. SRS defines a specific map projection as well as transformations between different SRS.

---

For More Information:

We need to create a new table with points projected in a planar coordinate. Then, we create an OpenLayers map object and set it to EPSG:3857. When using a projected SRS, we also need to set the map extent, resolution, and units:

```javascript
projection: "EPSG:3857",
maxExtent: new OpenLayers.Bounds(-20037508, -20037508, 20037508, 20037508.34),
maxResolution: 156543.0339,
units: 'm',
```

We then add two layers pointing to the same feature type, `populatedplaceswm`, and using two different spatial filters. The first one uses the `BEYOND` operator, passing it the coordinates of Rome expressed in meters, a 1000-kilometer distance, and the units:

```javascript
cql_filter: "BEYOND(geom,POINT(1389413 5145697),1000000,meters)",
```

Although the units are not evaluated when filtering features, the parameter is mandatory. Hence, you have to insert it, but you need to be aware of the native SRS of the data and calculate a proper value for the distance.

To override the default rendering of features, we set `sld_body` for the request to the SLD created in the JavaScript code:

```javascript
sld_body: sld,
```

To represent features inside the buffer, we create a similar layer, filtering features with the `DWITHIN` operator. The syntax is pretty similar to `BEYOND`; please ensure that you set the same point and distance to build the buffer area:

```javascript
cql_filter: "DWITHIN(geom,POINT(1389413 5145697),1000000,meters)",
```

Then, we set a different `sld_body` value to represent features with a different symbol:

```javascript
sld_body: sld2,
```

## Creating a SQL view

You probably know how to create a SQL view. Using views lets you represent data in different ways, extracting and transforming the data and avoiding data duplication.

With RDBMS, you can store views inside the database. In this case, a view is just a feature type for GeoServer, just like for a table.

You can also use a different approach with GeoServer, storing the SQL code inside your GeoServer configuration. This way, SQL views allow the execution of a custom SQL query on each request to the layer. This avoids the need to create a database view for complex queries.

For More Information:
www.packtpub.com/hardware-and-creative/geoserver-cookbook
We use PostGIS in this book. While it is one of the most powerful spatial databases available, not to mention that it is free to use, you may need to use other databases. GeoServer also supports Oracle, SQL Server, and MySQL with extension modules. You can use the recipes in this book with any of them; you only need to be careful with the SQL code. Code inserted in this book uses the ST_ functions that may have different syntax or be unavailable in other databases than PostGIS.

How to do it...

1. We will build a view that contains only European countries. Open your GeoServer web interface and switch to Layers:

   ![Layers](image)

   Manage the layers being published by GeoServer
   - Add a new resource
   - Remove selected resources

2. Select Add a new resource, and from the dropdown list, select the data store pointing to your RDBMS, PostGIS in our case. Instead of selecting a table from the list, select the Configure new SQL view... link:

   ![Add layer from](image)

   You can create a new feature type by manually configuring the attribute names and types. Create new feature type...
   On databases you can also create a new feature type by configuring a native SQL statement. Configure new SQL view...
   Here is a list of resources contained in the store "PostGISLocal". Click on the layer you wish to configure

3. In the form, insert EuropeanCountries as View Name and the following code as the SQL statement:

   ```sql
   SELECT ADMIN, 
   ST_UNION(COUNTRIES_EXP.GEOM) AS GEOM 
   FROM 
   (SELECT ADMIN, 
   (ST_DUMP(GEOM)).geom as geom 
   FROM 
   COUNTRIES 
   WHERE 
   REGION_UN = 'Europe') COUNTRIES_EXP
   ```

   For More Information:
   www.packtpub.com/hardware-and-creative/geoserver-cookbook
WHERE
ST_Intersects(COUNTRIES_EXP.GEOM,
  ST_GeomFromText('POLYGON((-11 37.40, -11 73.83, 27.28
  73.83, 27.28 37.40, -11 37.40))',4326)) = TRUE
GROUP BY ADMIN

4. Move to the bottom of the screen and select the **Guess Geometry type and srid** checkbox and click on **Refresh**. The **4326** EPSG code is properly detected, but you have to manually select **MultiPolygon** to avoid detecting the value of the polygon instead:

<table>
<thead>
<tr>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Refresh</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>SRID</th>
</tr>
</thead>
<tbody>
<tr>
<td>admin</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>geom</td>
<td>MultiPolygon</td>
<td>4326</td>
</tr>
</tbody>
</table>

5. Click on **Save**, and you will be brought to the publish layer form. Click on the button to calculate the native data extent and click on **Publish**. Move to **Layer Preview** and select the **EuropeanCountries** layer; your map should look like this one:

For More Information:
www.packtpub.com/hardware-and-creative/geoserver-cookbook
Creating a SQL view in GeoServer is not different from creating one in an RDBMS. You just have to build proper SQL code to filter and transform the feature.

It is not mandatory that source tables for your view are already published in GeoServer. You only need to have a data store properly configured to an RDBMS; you can’t use a SQL view against a shapefile or other file-based stores.

As you can see in the SQL code for this recipe, you can use any combination of the standard SQL and vendor extension. GeoServer does not evaluate the code, but it demands parsing and evaluation to the RDBMS.

You can use the aggregation and transformation function as we did. You need to return at least a proper geometrical field so that GeoServer can evaluate it and use it to configure a layer.

The view created from GeoServer is not stored inside your RDBMS. This may sound odd if you’re used to creating views in a database. Indeed, GeoServer views are a sort of virtual table. You can check this inside the data directory. Look for the workspace and find the featuretype definition, which is in the featuretype.xml file. You will find that your SQL query is just stored inside it:

```
<metadata>
  <entry key="JDBC_VIRTUAL_TABLE">
    <virtualTable>
      <name>EuropeanCountries</name>
      <sql>SELECT ADMIN,
      ST_UNION(COUNTRIES_EXP.GEOM) AS GEOM FROM
      (SELECT ADMIN,
      (ST_DUMP(GEOM)).geom as geom FROM
      COUNTRIES WHERE REGION_UN = 'Europe') COUNTRIES_EXP
      WHERE ST_Intersects(COUNTRIES_EXP.GEOM,
      ST_GeomFromText('POLYGON((-11 37.40, -11 73.83, 27.28
      73.83, 27.28 37.40, -11 37.40))',4326)) = TRUE GROUP BY ADMIN</sql>
  </virtualTable>
</entry>
```

For More Information:
www.packtpub.com/hardware-and-creative/geoserver-cookbook
Creating a parametric view

A parametric SQL view is based on a SQL query containing the named parameters with values provided dynamically in WMS and WFS requests.

How to do it...

1. We will now create a new view by extending the code of the previous step. What if you want to have a dynamic view that works for each continent? You need to start the same way as we did previously: select Add a new resource from the Layer menu.

2. Select the data store pointing to PostGIS or your preferred RDBMS. Instead of selecting a table from the list, select the Configure new SQL view... link:

3. In the form, insert ContinentView as View Name and the following code as the SQL statement:

```sql
SELECT ADMIN, GEOM FROM COUNTRIES WHERE CONTINENT = '%continent%'
```

4. Go to the SQL view parameters section and click on the Guess parameters from SQL link. Insert Africa as the default value.

For More Information:

www.packtpub.com/hardware-and-creative/geoserver-cookbook
5. At the bottom of the page, select the **Guess Geometry type and srid** checkbox and click on **Refresh**. The 4326 EPSG code is properly detected, but you have to manually select **MultiPolygon** instead of the detect value of **Polygon**.

6. Click on **Save**, and you'll be taken to the publish the layer form. Click on the button to calculate the native data extent, and then click on **Publish**. Move to **Layer Preview** and select the **ContinentView** layer. Your map now looks like this one:

![Map of Africa](image)

**For More Information:**
www.packtpub.com/hardware-and-creative/geoserver-cookbook
7. The preview map shows you the result of filtering with default values. Add \&viewparams=continent:Europe to your request URL and reload the map. You should now see a different map:

![Preview Map](image)

**How it works...**

A parametric view lets you define one or more filtering parameters at request time. In this recipe, instead of building five different views (one for each continent), we have a single view, and the calling app can define which continent is relevant in the request.

Building a parametric view is not different from a standard SQL view. You can start creating the code without any parameters to check that you have no syntax or logical errors.

Once you are done with the code, you can select which field should become a parameter. To let GeoServer recognize the parameters, you simply enclose them within % characters, as shown in the following code:

```
WHERE CONTINENT = '%continent%'
```

GeoServer recognizes the parameters and lets you set an optional default value. Please note that if you don’t set a default value, you always need to set a value for parameters when sending a request to GeoServer.

At run time, you set values for each parameter, adding the `viewparams` option to your request. As per the value of the option, you insert a set of `params-value` couples.
Improving performance with pregeneralized features

The feature type may contain a lot of coordinates. Also, if it's not a really big data, the simple vector datasets we use in this recipe contain a large number of vertices:

```
gisdata=> select sum(ST_NPoints(geom)) from countries;
sum
--------
548604
```

When you're creating a small-scale map, that is, a continent view, it does not make sense to have the GeoServer process all this data to output a really simplified rendering of the shapes. You are just wasting CPU time and degrading the performance of your server.

Having a simplified version of your data for general representation would be more practical, but you also want to have a detailed version of the data when your map goes to middle or large scale.

The idea behind the pregeneralized features module is to combine more versions of a feature type so that users of GeoServer can choose the one that is the best for each scale, while the user continues to use it as if it was a single feature type.

Pregeneralized features are supported by an extension module. If you want to use this feature, you need to download the optional module. Please note that any extension module will follow the version of GeoServer. Installation is quite easy. Download the archive from http://geoserver.org/release/stable/ and extract the two JAR files into the WEB-INF/lib directory of the GeoServer installation.

How to do it...

1. Start creating three generalized versions of countries for use in this recipe. The following instructions work for PostGIS:

```
gisdata=> create table countries_0 as select admin, geom from countries;
gisdata=> CREATE INDEX countries_0_geom_gist ON countries_0 USING gist(geom);
gisdata=> create table countries_01 as select admin, ST_SimplifyPreserveTopology(geom,0.01) as geom from countries;
```

For More Information:

www.packtpub.com/hardware-and-creative/geoserver-cookbook
gisdata=> CREATE INDEX countries_01_geom_gist ON countries_01 
    USING gist(geom);

gisdata=> create table countries_05 as select admin, 
    ST_SimplifyPreserveTopology(geom,0.05) as geom from 
    countries;

gisdata=>CREATE INDEX countries_05_geom_gist ON countries_05 
    USING gist(geom);

gisdata=>create table countries_1 as select admin, 
    ST_SimplifyPreserveTopology(geom,0.1) as geom from 
    countries;

gisdata=>CREATE INDEX countries_1_geom_gist ON countries_1 
    USING gist(geom);

2. Create a new XML file, insert the following code snippet, and save it as geninfo_postgis.xml:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<GeneralizationInfos version="1.0">
    <GeneralizationInfo dataSourceNameSpace="NaturalEarth" 
        dataSourceName="PostGISLocal" featureName="GeneralizedCountries" 
        baseFeatureName="countries_0" geomPropertyName="geom">
        <Generalization dataSourceNameSpace="NaturalEarth" 
            dataSourceName="PostGISLocal" distance="1" 
            featureName="countries_01" geomPropertyName="geom"/>
        <Generalization dataSourceNameSpace="NaturalEarth" 
            dataSourceName="PostGISLocal" distance="5" 
            featureName="countries_05" geomPropertyName="geom"/>
        <Generalization dataSourceNameSpace="NaturalEarth" 
            dataSourceName="PostGISLocal" distance="10" 
            featureName="countries_1" geomPropertyName="geom"/>
    </GeneralizationInfo>
</GeneralizationInfos>
```

For More Information:  
www.packtpub.com/hardware-and-creative/geoserver-cookbook
3. Go to the data store section of the GeoServer web interface, click on **Add new data store**, and then select **Generalizing data store**:

![Vector Data Sources]

- Directory of spatial files (shapefiles) - Takes a directory of shapefiles and exposes it as a data store
- Generalizing data store - Data store supporting generalized geometries
- PostGIS - PostGIS Database
- PostGIS (JNDI) - PostGIS Database (JNDI)

4. Input **GeneralizedCountries** as the name for the data store, and then point to the location of the `geninfo_postgis.xml` file. Change the `org.geotools` string to `org.geoserver` in the two textboxes and click on **Save**:

![Connection Parameters]

- **RepositoryClassName** *
  - `org.geoserver.data.gen.DSFinderRepository`
- **GeneralizationInfosProviderClassName** *
  - `org.geoserver.data.gen.info.GeneralizationInfosProvider`
- **GeneralizationInfosProviderParam**
  - `file:data/geninfo_postgis.xml`
- **Namespace** *
  - `http://www.naturalearthdata.com/`

5. You can now see the feature type you defined in the XML file and click on the **Publish** button. You are now done. Switch to the layer's preview to check whether GeoServer is properly visualizing the data.

**How it works...**

If you look at the layer preview, you will not see any difference from the countries' layers. You will observe a faster rendering; this is because GeoServer is indeed extracting geometries from the simplified table. Let's check what is happening behind the scenes.
Set the log detail in GeoServer to `GEOTOOLS_DEVELOPER_LOGGING`, and then open your GeoServer log with the `tail` command:

```
$: tail -f /opt/Tomcat7042/webapps/geoserver/data/logs/geoserver.log
```

Now open the preview for `GeneralizedCountries`. After the map is shown, you should see some rows that state GeoServer first evaluates the geometry distance from the map's scale. Select the table that is more appropriate to extract the features from:

```
INFO [org.geotools.data.gen] - Hint geometry distance: 0.5681250333785898
INFO [org.geotools.data.gen] - Hint geometry distance: 0.5681250333785898
INFO [org.geotools.data.gen] - Using generalization: PostGISLocal countries_1 geom 0.1
```

In the log, the actual query that is performed on the database is present, and you can check whether it is created on a simplified version of the countries. As you can see, it is indeed created on a simplified version of the countries. Actually, the version with a higher generalization degree, which contains the more simplified features, is used:

```
DEBUG [org.geotools.jdbc] - CREATE CONNECTION
DEBUG [org.geotools.jdbc] - SELECT
   encode(ST_AsBinary(ST_Force_2D("geom")),'base64') as "geom" FROM
   "public"."countries_1" WHERE "geom" & & ST_GeomFromText('POLYGON ((-244.29375672340652 -121.77904978394649, -244.29375672340652 115.41315165161649, 244.29377198219652 115.41315165161649, 244.29377198219652 -121.77904978394649, -244.29375672340652 -121.77904978394649))', 4326)
   [org.geotools.jdbc] - CLOSE CONNECTION
```

From the database, you can check the total number of features in this table:

```
gisdata=> select sum(ST_NPoints(geom)) from countries_1;
   sum
  ------
  44085
```

Now, zoom in to your map and check what GeoServer writes in the log. When your map is centered on Europe, the map scale triggers GeoServer to use another table. If you inspect the log, you can indeed observe that now GeoServer is using the table `countries_05`:

```
INFO [org.geotools.data.gen] - Using generalization: PostGISLocal countries_05 geom 0.05
```

For More Information:

www.packtpub.com/hardware-and-creative/geoserver-cookbook
Check the database again; the total number of features is higher. However, you are using a fraction of them, as only a portion of the Earth is represented on the map, so you get finer detail without a slow rendering. This is attained using the following lines of code:

```
gisdata=> select sum(ST_NPoints(geom)) from countries_05;
      sum
     ------
      66042
```

Continue to zoom in until you see in the log that GeoServer is using the original table, `countries_0`. You are now using the entire detailed geometries, but in a relatively small area:

```
DEBUG [org.geotools.jdbc] - SELECT
    encode(ST_AsBinary(ST_Force_2D("geom")),'base64') as "geom" FROM
"public"."countries_0" WHERE "geom" && ST_GeomFromText('POLYGON ((6.016917772920612 4.379043245570911,
6.016917772920612 5.305575282428689, 7.925462806926788
5.305575282428689, 7.925462806926788 4.379043245570911,
1, 6.016917772920612 4.379043245570911))', 4326)
```
Where to buy this 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.