Chapter 9
Tile Serving

The previous four chapters explained techniques for creating and storing tiled images. In this chapter we will examine methods for sharing those tiled images with other users over a network.

9.1 Basics of HTTP

The Hypertext Transfer Protocol (HTTP) is one of the core standards of the Internet. It was originally designed for sharing interlinked documents but is now used for many other types of applications. Since HTTP is the basic application protocol of the Internet, there are considerable existing tools in place to support it. HTTP clients and/or servers are built into many applications and programming environments. Commercial and residential firewalls and proxies are designed to handle HTTP traffic. HTTP has a sophisticated security model with extensive infrastructure. What is most important to our designed of a tiled image server is that HTTP can be used to efficiently and securely share tiled images to both Web browsers and other applications.

The HTTP standard defines eight operations, but only two will be relevant to our work: GET and POST. The GET operation's intended use is to retrieve web content without causing side effects on the server. Consider a simple service to retrieve the current time from a time server. This would be best modeled with a GET operation, because there is no need to change any information on the time server. The POST operation is used to "post" data back to the server. For example, a web service that processed book orders would be modeled with the POST operation. This operation does cause side effects on the server because repeated executions of the post operation will result in multiple book orders. Since we are concerned with serving tiled images to users, not collecting data from users, we should use the "GET" operation to create our tiled image service. HTTP GET encodes query parameters by concatenating them on the end of the resource's Uniform Resource Locator (URL). For example, a URL to a time service might be:

J.T. Sample and E. Ioup, *Tile-Based Geospatial Information Systems: Principles and Practices*, DOI 10.1007/978-1-4419-7631-4_9,

```
http://www.sometimeservice.com/getthetime
```

Entering this URL into a web browser would return the current time encoded as an HTML document. Suppose that you want to retrieve the time for a different time zone. You could add a parameter called ”zone” to send the server a time zone you want. The resulting URL would look like this:

```
http://www.sometimeservice.com/getthetime?zone=UTC-6
```

An additional parameter could be added to specify the encoding for the response like this:

```
http://www.sometimeservice.com/getthetime?zone=UTC-6&
encoding=XML
```

9.2 Basic Tile Serving

In general, serving tiles requires a multi-step process. In the first step, users query a tile server of a list of available layers. They may also query the server for the availability of specific tiles from each layer and the image format of the specific tiles. In the final step, users request the actual tiled images.

In practice the interaction may be much simpler. Clients will usually not ask for the format of a tile. We have mandated that our tiled images be stored in browser compatible formats, either JPEG or PNG, so web browsers will be able to read them without querying first to see what format they use. Also, rather than use two queries to check if a tile is there and then to request it, clients will just request the tile, and accept the response that comes back. Many servers will provide a default tile to stand in for missing ones. This can be an empty, completely transparent tile, or a tile with a message that says the tile is not available.

The next two sections will present two slightly different schema for serving tiled images. They differ in the manner in which tiles are queried. The reader should not infer that the two different schema are incompatible or must be implemented on different systems. Each scheme is merely an interface to the same backing data store. Both schemes can be implemented on the same server. In practice, many different interfaces are implemented on tile stores to support as many clients as possible. Modern HTTP servers and Web service frameworks allow multiple interfaces to be implemented with very minimal overhead. In the final chapter, we will look at other methods for serving tiles to accommodate multiple client software packages and standardized protocols.

Each of the following algorithms will make use of the data store concept. The data store provides tiled images organized into layers. Individual tiles are queried by

Listing 9.1 Java DataStore abstract class.

```
public abstract class DataStore {

    //returns a list of layers availible from this DataStore
    public abstract String[] getLayersAvailible();

    //returns a list of TileRanges for the zoom levels availible from the named
        layer
    //null values in the array indicate a missing zoom level
    public abstract TileRange[] getTileRanges(String layerName);

    public abstract boolean tileExists(String layerName, int level, long row,
        long column);

    public abstract String getTileFormat(String layerName, int level, long row,
         long column);

    public abstract byte[] getTileImage(String layerName, int level, long row,
        long column);

}

class TileRange {

    int level;
    long mincol;
    long maxcol;
    long minrow;
    long maxrow;

}
```

layer, zoom level, row and column. The data store also provides information about the layers, such as what zoom levels and tile ranges are available for each layer, and whether or not a tile exists for a given address. Java and Python abstractions for the data store concept are shown in Listings 9.1 and 9.2

9.3 Tile Serving Scheme with Encoded Parameters

This section presents a tile scheme in which all parameters needed to query the tile server are encoded as HTTP GET parameters concatenated to the tile server's URL. A URL similar to the following will serve as the base URL for all tile requests:

`http://www.sometileserver.com/tiles`

If left without any parameters it will return a simple text-encoded list of available layers, the zoom levels and ranges of tiles available for each layer. The layer list response will appear as in Tables 9.1 and 9.2. This list can easily be parsed by client applications. Tile request parameters are given in Table 9.3.

Listing 9.2 Python DataStore abstract class

```
class DataStore:
    def getLayersAvailable(self):
        return []

    def getTileRanges(self):
        return []

    def tileExists(self):
        return False

    def getTileFormat(self):
        return ''

    def getTileImage(self):
        return ''

class TileRange:
    def __init__(self):
        self.level = 0
        self.mincol = 0
        self.maxcol = 0
        self.minrow = 0
        self.maxrow = 0
```

LAYER: layername
LEVEL: level_number, min_column, min_row, max_column, max_row

Table 9.1 Layer response list template.

LAYER: roads
LEVEL: 1, 0,0, 1,0
LEVEL: 2, 0 , 0, 3,1
LEVEL: 3, 0,0, 7, 3
LEVEL: 4,0,0,15,7
LEVEL: 5,0,0,31,15
LEVEL: 6,0,0,63,31
LEVEL: 7,0,0,127,63
LEVEL: 8,0,0,255,127
LAYER: imagery
LEVEL: 1,1,0,1,0
LEVEL: 2,2,1,2,1
LEVEL: 3,4,3,4,3
LEVEL: 4,8,6,8,6
LEVEL: 5,17,12,17,12
LEVEL: 6,34,24,34,24
LEVEL: 7,69,48,69,48
LEVEL: 8,139,96,139,97
LEVEL: 9,278,193,278,194
LEVEL: 10,556,387,556,388
LEVEL: 11,1112,775,1113,776
LEVEL: 12,2225,1551,2227,1553
LEVEL: 13,4451,3103,4455,3107
LEVEL: 14,8902,6207,8911,6214

Table 9.2 Example response list with two layers.

Parameter Name	Purpose	Possible Values
REQUEST	Specifies the type of request to be executed.	GETTILE TILEEXISTS GETFORMAT
LAYER	Specifies the name of the layer that will be used.	Any of the layer names listed in the layer list.
LEVEL	Specifies the zoom level number to be queried.	Any level number listed in the layer list.
ROW	Specifies the row of the tile be queried.	Any row value within the range list in the layer list.
COLUMN	Specifies the column of the tile be queried.	Any column value within the range list in the layer list.

Table 9.3 Tile request parameters.

Request types `GETTILE`, `TILEEXISTS`, and `GETFORMAT` are used to retrieve a tile image, check if the tile exists, and retrieve the format of the tiled image, respectively. An example query to retrieve a tiled image looks like the following:

```
http://www.sometileserver.com/tiles?REQUEST=GETTILE&
LAYER=imagery&LEVEL=6&ROW=24&COLUMN=34
```

Our first implementation of this scheme is in Java and uses the Java Servlet API, see Listing 9.3. The Java Servlet API provides a simple framework for creating HTTP based web services. Alternatively, our Python version of the same program uses the Web Server Gateway Interface (WSGI), see Listing 9.4. WSGI is a cross-platform interface used to support web applications in Python. WSGI provides similar functionality to the Java Servlet API.

9.4 Tile Serving Scheme with Encoded Paths

This scheme differs from the previous one in that the tile parameters are encoded in the URL path directly and not as query parameters. The template for the query method is as follows:

```
http://www.sometileserver.com/tiles/LAYERNAME/LEVEL/
ROW/COLUMN/REQUEST
```

For example, the previous tile query URL:

```
http://www.sometileserver.com/tiles?REQUEST=GETTILE&
LAYER=imagery&LEVEL=6&ROW=24&COLUMN=34
```

will now be:

```
http://www.sometileserver.com/tiles/imagery/6/24/34/
GETTILE
```

The method provides shorter query URLs, and mimics the paths of a computer's file system. If only the base path is entered, it will return a list of available layers and their tile ranges, exactly as with the previous scheme.

Listings 9.5 and 9.6 show implementations of this scheme. The method `getPathInfo()` is built into the Java Servlet API. It provides the path information after the address of the servlet. This is precisely the information we need to parse to get the path query values. The Python equivalent code uses the `environ ['PATH_INFO']` value to obtain the URL path.

As seen from the example code these two schemes are very similar. The second scheme might be needed to provide compatibility with certain tiled image clients or with Internet caching systems that use URL patterns to determine what content to cache.

9.5 Service Metadata Alternatives

Each of the previous schemes returned text-formatted metadata about the layers available from its data store. Some users may prefer more formally defined formats. Two examples are XML and JSON. XML (Extensible Markup Language) and JSON (JavaScript Object Notation) both support structured, hierarchical storage of text information. Recall the layer list from previous section:

LAYER: roads
LEVEL: 1,0,0,1,0
LEVEL: 2,0,0,3,1
LEVEL: 3,0,0,7,3
LEVEL: 4,0,0,15,7
LEVEL: 5,0,0,31,15
LEVEL: 6,0,0,63,31
LEVEL: 7,0,0,127,63
LEVEL: 8,0,0,255,127

In XML, this might look like the following:

```
<Layer name="roads">
  <Level number="1" mincol="0" minrow="0" maxcol="1" maxrow="0" />
  <Level number="2" mincol="0" minrow="0" maxcol="3" maxrow="1" />
  <Level number="3" mincol="0" minrow="0" maxcol="7" maxrow="3" />
  <Level number="4" mincol="0" minrow="0" maxcol="15" maxrow="7" />
  <Level number="5" mincol="0" minrow="0" maxcol="31" maxrow="15" />
  <Level number="6" mincol="0" minrow="0" maxcol="63" maxrow="31" />
  <Level number="7" mincol="0" minrow="0" maxcol="127" maxrow="63" />
  <Level number="8" mincol="0" minrow="0" maxcol="255" maxrow="127" />
</Layer>
```

In JSON, it could be encoded as:

```
{
    "layer": "roads",
        "levels" : [
            {"1" : [0,0,1,0] },
            {"2" : [0,0,3,1] },
            {"3" : [0,0,7,3] },
            {"4" : [0,0,15,7] },
            {"5" : [0,0,31,15] },
            {"6" : [0,0,63,31] },
            {"7" : [0,0,127,63] },
            {"8" : [0,0,255,127] }
        ]
}
```

The benefit of these two formats is that they are well supported by many applications and programming environments. Most web browsers can automatically convert the JSON text into in-memory objects. Optional encodings can easily be added to our tile server schemes. We can simply add a parameter called "FORMAT" with the possible values "plaintext", "XML", "JSON." The tile server can then use this parameter to determine which format to output as a response.

9.6 Conclusions

In this chapter, we have covered how to serve tiled images via HTTP. Chapter 13 provides a discussion and examples of advanced tile serving to support a variety of client software systems. We have not covered the important topic of securing our tile server or its services. For more information on this topic we suggest [1] for further reading.

Listing 9.3 Java servlet code for tile serving with encoded parameters.

```
1  public class TileServlet extends HttpServlet {
2
3      DataStore dataStore;
4
5      public void doGet(HttpServletRequest request, HttpServletResponse response)
           {
6          String requestType = request.getParameter("REQUEST");
7          if (requestType.equalsIgnoreCase("GETTILE") ||requestType.
               equalsIgnoreCase("TILEEXISTS") || requestType.equalsIgnoreCase("
               GETFORMAT")) {
8              String layerName = request.getParameter("LAYER");
9              int level = Integer.parseInt(request.getParameter("LEVEL"));
10             long row = Long.parseLong(request.getParameter("ROW"));
11             long column = Long.parseLong(request.getParameter("COLUMN"));
12
13             if (requestType.equalsIgnoreCase("GETTILE")) {
14                 byte[] data = dataStore.getTileImage(layerName, level, row,
15                 column);
16                 if (data != null) {
17                     response.setContentType(dataStore.getTileFormat(layerName,
                           level, row, column));
18                     OutputStream os;
19                     try {
20                         os = response.getOutputStream();
21                         os.write(data);
22                         os.close();
23                         } catch (IOException e) {
24                         e.printStackTrace();
25                     }
26                 }
27             }
28             if (requestType.equalsIgnoreCase("TILEEXISTS")) {
29                 boolean val = dataStore.tileExists(layerName, level, row,
                       column);
30                 response.setContentType("text/plain");
31                 PrintWriter pw;
32                 try {
33                     pw = response.getWriter();
34                     pw.println(val);
35                     pw.close();
36                     } catch (IOException e) {
37                     e.printStackTrace();
38                 }
39                 return;
40             }
41             if (requestType.equalsIgnoreCase("GETFORMAT")) {
42                 String format = dataStore.getTileFormat(layerName, level, row,
                       column);
43                 response.setContentType("text/plain");
44                 PrintWriter pw;
45                 try {
46                     pw = response.getWriter();
47                     pw.println(format);
48                     pw.close();
49                     } catch (IOException e) {
50                     e.printStackTrace();
51                 }
52                 return;
53             }
54         }
55         //valid request type not found, send back layer list response
56         printLayerList(request, response);
57     }
58
59     private void printLayerList(HttpServletRequest request, HttpServletResponse
           response) {
60         PrintWriter pw;
61         try {
62             pw = response.getWriter();
63             String[] layers = dataStore.getLayersAvailible();
```

```
64              for (int i = 0; i &lt; layers.length; i++) {
65                  pw.println("LAYER:" + layers[i]);
66                  TileRange[] ranges = dataStore.getTileRanges(layers[i]);
67                  for (int j = 0; j &lt; ranges.length; j++) {
68                      if (ranges[j] != null) {
69                          pw.println("LEVEL:" + ranges[j].level + ","
70                          + ranges[j].mincol + "," + ranges[j].minrow
71                          + "," + ranges[j].maxcol + ","
72                          + ranges[j].maxrow);
73                      }
74                  }
75              }
76              pw.close();
77              } catch (IOException e) {
78              e.printStackTrace();
79          }
80      }
81  }
```

Listing 9.4 Python code for tile service with encoded parameters.

```
import cgi          # the cgi module is only used for some query string parsing
class TileWSGIApp:

    def __init__(self, datastore):
        self._datastore = datastore

    def doGet(self, environ, start_response):
        status = '200 OK'
        headers = [('Content-type', 'text/plain')]

        queryParams = cgi.FieldStorage(environ['wsgi.input'], environ=environ)

        requestType = queryParams.getfirst('REQUEST', '').lower()
        if (requestType in ['gettile', 'tileexists', 'getformat']):
            layerName = queryParams.getfirst('LAYER', '')
            level = int(queryParams.getfirst('LEVEL', -1))
            row = int(queryParams.getfirst('ROW', -1))
            column = int(queryParams.getfirst('COLUMN', -1))

            if (requestType == 'gettile'):
                data = self._datastore.getTileImage(layerName, level, row,
                    column)
                if (data != None):
                    headers = [('Content-type',
                                self._datastore.getTileFormat(layerName, level,
                                                              row, column))]

            elif(requestType == 'tileexists'):
                exists = self._datastore.tileExists(layerName, level, row,
                    column)
                if (exists):
                    data = 'True'
                else:
                    data = 'False'

            elif(requestType == 'getformat'):
                data = '%s' % self._datastore.getTileFormat(layerName, level,
                                                            row, column)

            else:
                data = self._getLayerList()
        else:
            data = self._getLayerList()

        start_response(status, headers)
        return [data]

    def _getLayerList(self):
        strbuf = cStringIO.StringIO()
        layers = self._datastore.getLayersAvailable()
        for layer in layers:
            strbuf.write('LAYER: %s\n' % layer)
            tileRanges = self._datastore.getTileRanges(layer)
            for tileRange in tileRanges:
                if (tileRange != None):
                    strbuf.write('LEVEL: %s,%s,%s,%s,%s\n' %
                                 (tileRange.level, tileRange.mincol,
                                  tileRange.minrow, tileRange.maxcol,
                                  tileRange.maxrow))
        return strbuf.getvalue()
```

Listing 9.5 Java code for tile serving with path encoded parameters.

```
public class TileServlet2 extends HttpServlet {

  DataStore dataStore;

  public void doGet(HttpServletRequest request, HttpServletResponse response) {
      //get any parts of the request path, after the servlet address, but not
          including query parameters
      String path = request.getPathInfo();
      if (path == null) {
          printLayerList(request, response);
          return;
      }
      if (path.startsWith("/")) {
          path = path.substring(1);
       }
      String[] queryVals = path.split("/");
      if (queryVals.length != 5) {
          printLayerList(request, response);
          return;
      }

      String layerName = queryVals[0];
      int level = Integer.parseInt(queryVals[1]);
      long row = Long.parseLong(queryVals[2]);
      long column = Long.parseLong(queryVals[3]);
      String requestType = queryVals[4];

      if (requestType.equalsIgnoreCase("GETTILE")) {
          byte[] data = dataStore.getTileImage(layerName, level, row, column);
          if (data != null) {
              response.setContentType(dataStore.getTileFormat(layerName,
              level, row,
              column));
              OutputStream os;
              try {
                  os = response.getOutputStream();
                  os.write(data);
                  os.close();
                  } catch (IOException e) {
                  e.printStackTrace();
              }
          }
      }
      if (requestType.equalsIgnoreCase("TILEEXISTS")) {
          boolean val = dataStore.tileExists(layerName, level, row, column);
          response.setContentType("text/plain");
          PrintWriter pw;
          try {
              pw = response.getWriter();
              pw.println(val);
              pw.close();
              } catch (IOException e) {
              e.printStackTrace();
          }
          return;
      }
      if (requestType.equalsIgnoreCase("GETFORMAT")) {
          String format = dataStore.getTileFormat(layerName, level, row,
          column);

          response.setContentType("text/plain");
          PrintWriter pw;
          try {
              pw = response.getWriter();
              pw.println(format);
              pw.close();
```

```
66                  } catch (IOException e) {
67                      e.printStackTrace();
68              }
69              return;
70          }
71
72          //valid request type not found, send back layer list response
73          printLayerList(request, response);
74      }
75
76      private void printLayerList(HttpServletRequest request, HttpServletResponse
            response) {
77          PrintWriter pw;
78          try {
79              pw = response.getWriter();
80              String[] layers = dataStore.getLayersAvailible();
81              for (int i = 0; i &lt; layers.length; i++) {
82                  pw.println("LAYER:" + layers[i]);
83                  TileRange[] ranges = dataStore.getTileRanges(layers[i]);
84                  for (int j = 0; j &lt; ranges.length; j++) {
85                      if (ranges[j] != null) {
86                          pw.println("LEVEL:" + ranges[j].level + ","
87                          + ranges[j].mincol + "," + ranges[j].minrow
88                          + "," + ranges[j].maxcol + ","
89                          + ranges[j].maxrow);
90                      }
91                  }
92              }
93              pw.close();
94              } catch (IOException e) {
95              e.printStackTrace();
96          }
97      }
98  }
```

Listing 9.6 Python code for tile serving with path encoded parameters.

```
class TileWSGIApp2:

    def __init__(self, datastore):
        self._datastore = datastore

    def doGet(self, environ, start_response):
        status = '200 OK'
        headers = [('Content-type', 'text/plain')]

        path = environ['PATH_INFO']

        if (path == None):
            data = self._getLayerList()
        else:
            if (path[0] == '/'):
                path = path[1:]

            queryVals = path.split("/")

            if (len(queryVals) != 5):
                data = self._getLayerList()
            else:
                layerName = queryVals[0]
                level = int(queryVals[1])
                row = int(queryVals[2])
                column = int(queryVals[3])
                requestType = queryVals[4].lower()

                if (requestType == 'gettile'):
                    data = self._datastore.getTileImage(layerName, level, row,
                        column)
                    if (data != None):
                        headers = [('Content-type',
                                    self._datastore.getTileFormat(layerName,
                                        level,
                                                                  row, column))
                                                                      ]

                elif(requestType == 'tileexists'):
                    exists = self._datastore.tileExists(layerName, level, row,
                        column)
                    if (exists):
                        data = 'True'
                    else:
                        data = 'False'

                elif(requestType == 'getformat'):
                    data = '%s' % self._datastore.getTileFormat(layerName,
                        level,
                                                                row, column)

                else:
                    data = self._getLayerList()

        start_response(status, headers)
        return [data]

    def _getLayerList(self):
        strbuf = cStringIO.StringIO()
        layers = self._datastore.getLayersAvailable()
        for layer in layers:
            strbuf.write('LAYER: %s\n' % layer)
            tileRanges = self._datastore.getTileRanges(layer)
            for tileRange in tileRanges:
                if (tileRange != None):
                    strbuf.write('LEVEL: %s,%s,%s,%s,%s\n' %
```

```
62                                  (tileRange.level, tileRange.mincol,
63                                   tileRange.minrow, tileRange.maxcol,
64                                   tileRange.maxrow))
65          return strbuf.getvalue()
```

References

1. Wells, C.: Securing AJAX applications. O'Reilly (2007)