Web Frameworks for Python Geospatial Development

In this chapter, we will examine various options for developing web-based geospatial applications. We will also explore a number of important concepts used by web applications, and geospatial web applications in particular, as well as studying some of the important open protocols used by these applications and a number of available tools which you can use to implement your own web-based geospatial systems.

In particular, we will be:

- Examining the architecture used by web applications
- Learning about web application stacks
- Exploring the concept of a full stack web application framework
- Learning about web services
- Learning how map rendering can be implemented as a standalone web service
- Learning how tile caching can be used to improve performance
- Learning how web servers act as a frontend to a web application
- Exploring how JavaScript user interface libraries can enhance a web application's user interface
- Learning how slippy maps can be created using an application stack
- Examining the web frameworks available for developing geospatial applications using Python
- Learning about some of the important open protocols for working with geospatial data within a web application
- Exploring some of the major tools and frameworks available for building your own geospatial web applications

Web application concepts

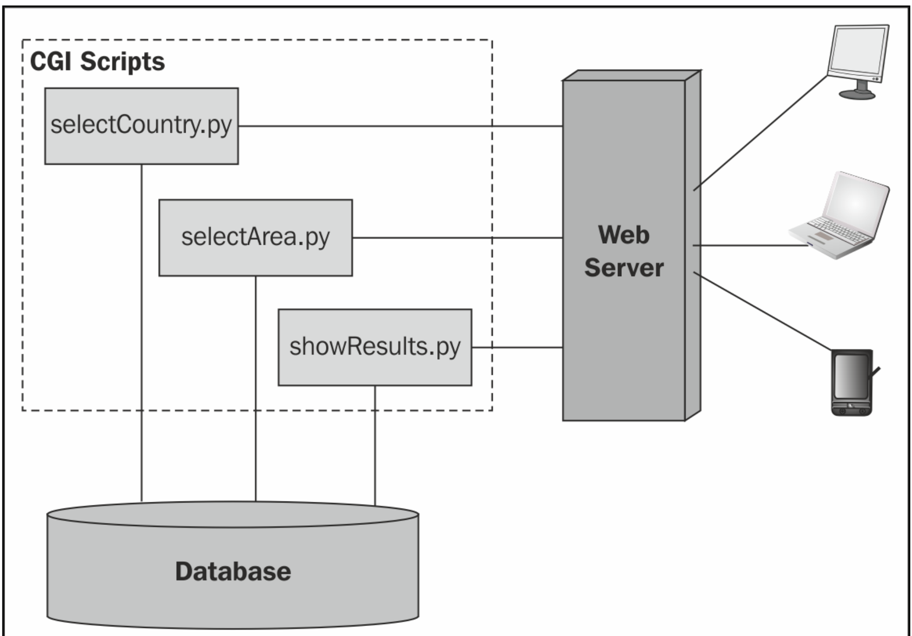
In this section we will examine a number of important concepts related to web-based application development in general, as well as concepts specific to developing geospatial applications that are accessed via a web server.

Web application architecture

There are many ways you can approach the development of a web-based application. You can write your own code, for example as a series of CGI scripts, or you can use one of the many web application frameworks which are available. In this section, we will look at the different ways web applications can be structured so that the different parts work together to implement the application's functionality.

A bare bones approach

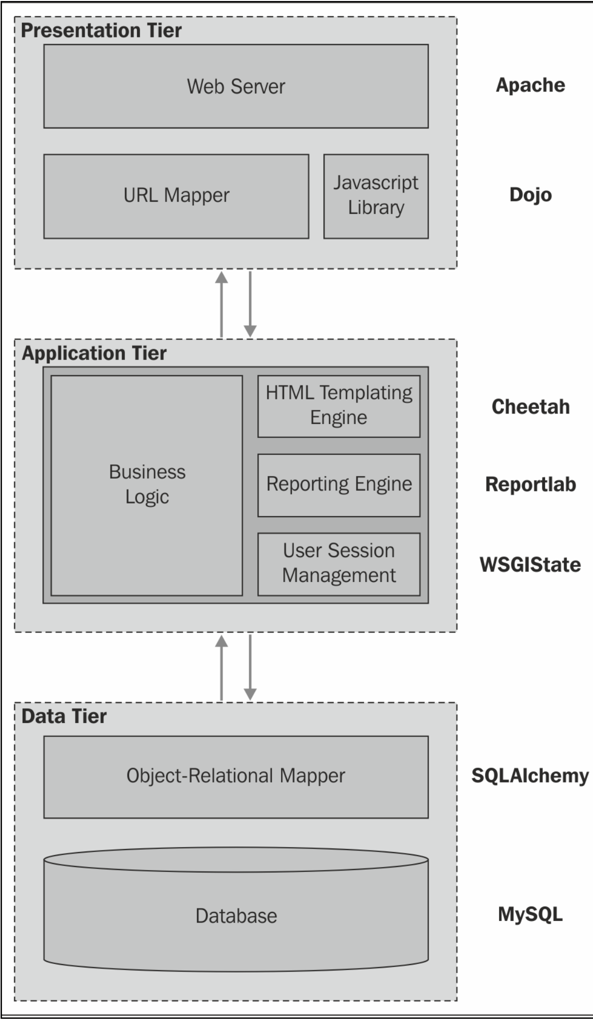
In *Chapter 7*, *Working with Spatial Data*, we created a simple web application named DISTAL. This web application was built using CGI scripts to provide distance-based identification of towns and other features. DISTAL is a good example of a "bare bones" approach to web application development, using nothing more than a web server, a database, and a collection of CGI scripts. The following diagram describes this approach:



The advantage of this approach is simplicity: you don't need any special tools or knowledge to write a web application in this way. The disadvantage is that you have to do all the low-level coding by hand. It's a very tedious and slow way of building a web application, especially a complex one with lots of features.

Web application stacks

To simplify the job of building web-based applications, you generally make use of existing tools that allow you to write your application at a higher level. For example, you might choose to implement a complex web application in the following way:



This stack of tools works together to implement your application: at the lowest level you have a **data tier** which deals with the storage of data. In this case, the application uses MySQL for the database and SQLAlchemy as an object-relational mapper to provide an object-oriented interface to this database. The **application tier** contains the application's business logic, as well as various libraries to simplify the job of building a stateful and complex web application. Finally, the **presentation tier** deals with the user interface, serving web pages to the users, mapping incoming URLs to the appropriate calls to your business logic, and using a sophisticated JavaScript library to build complex user interfaces within the user's web browser.

Different terms are sometimes used for these three tiers. For example, the data tier is sometimes called the **data access** layer, and the application tier is sometimes called the **business logic** layer. The concepts are the same, however.

Don't be too concerned about the details of this particular application's architecture, the main thing to realize is that there is a "stack" of tools all working together, where each tool makes use of the tools below it. Also, notice the complexity of this system: this application depends on a lot of different tools and libraries. Developing, deploying, and upgrading an application such as this can be challenging because it has so many different parts.

Web application frameworks

To avoid the complexity of mixing and matching so many different libraries, web developers have created various frameworks which combine tools to provide a complete web development system. Instead of having to select, install, and deploy ten different libraries, you can simply choose a complete framework that brings a known good set of libraries together, and adds its own logic to provide a complete "batteries included" web development experience. Most of these toolkits provide you with built-in logic to handle tasks such as:

- Defining and migrating your database schema
- Keeping track of user sessions and handling user authentication
- Building sophisticated user interfaces, often using AJAX to handle complex widgets within the user's browser

- Automatically allowing users to create, read, update, and delete records in the database (the so-called CRUD interface)
- Simplifying the creation of database-driven applications through standard templates and recipes

There is a lot more to these frameworks, but the important thing to remember is that they aim to provide a full stack of features which allow developers to quickly implement the most common aspects of a web application with a minimum of fuss. They aim to provide **rapid application development** (**RAD**) for web-based systems.

There are a number of Python-based web application frameworks available, including TurboGears, Django, Zope, Pyramid, and Web2py. Some of these frameworks also include extensions for developing geospatial web applications.

Web services

A web service is a piece of software that has an **application programming interface** (**API**) that is accessed via the HTTP protocol. Web services implement behind-the-scenes functionality used by other systems; they don't generally have an interface that allows end users to access them directly.

Web services are accessed via a URL; other parts of the system send a request to this URL and receive back a response, often in the form of XML or JSON encoded data, which is then used for further processing.

Types of Web Services

There are three main types of web services you are likely to encounter: RESTful web services, which use parts of the URL itself to tell the web service what to do, query string-based web services, and big web services which typically use the SOAP protocol to communicate with the outside world.

REpresentational State Transfer (REST) is a protocol that uses sub-paths within the URL to define the request to be made. For example, a web service might use the following URL to return information about a customer:

`http://myserver.com/webservice/customer/123`

In this example, `customer` defines what type of information you want, and `123` is the internal ID of the desired customer. RESTful web services are very easy to implement and use, and are becoming increasingly popular with web application developers.

The second major type of web service makes use of query strings. For example, you might update a customer's details using the following URL:

`http://myserver.com/webservice/update_customer?id=123&name=Tom`

Finally, a big web service has just one URL as the entry point for the entire web service. A request is sent to this URL as an XML-format message, and the response is sent back, also as an XML-formatted message. The SOAP protocol is often used to describe the message format and how the web service should behave. Big web services are popular in large commercial systems, despite being more complex than their RESTful equivalent.

Let's take a look at a simple but useful web service. This CGI script, called `greatCircleDistance.py`, calculates and returns the great-circle distance between two coordinates on the Earth's surface. Here is the full source code for this web service:

```
#!/usr/bin/python
import cgi
import pyproj
form = cgi.FieldStorage()
lat1 = float(form['lat1'].value)
long1 = float(form['long1'].value)
lat2 = float(form['lat2'].value)
long2 = float(form['long2'].value)
geod = pyproj.Geod(ellps="WGS84")
```

```
angle1,angle2,distance = geod.inv(long1, lat1, long2, lat2)
print 'Content-Type: text/plain'
print
print distance
```

This web service uses query parameters to supply the two coordinates, and the resulting distance (in meters) is returned as the body of the HTTP response. Since the returned value is a single number, there is no need to encode the results using XML or JSON; instead, the distance is returned as plain text.

If you want to run this CGI script, create the following program and name it `webServer.py`:

```
import BaseHTTPServer
import CGIHTTPServer
address = ('', 8000)
handler = CGIHTTPServer.CGIHTTPRequestHandler
server = BaseHTTPServer.HTTPServer(address, handler)
server.serve_forever()
```

Then create a sub-directory named `cgi-bin` in the same directory as your `webServer.py` program, and place your script in that directory. When you run this program, you will be able to access your CGI script at the following URL:

`http://127.0.0.1:8000/cgi-bin/greatCircleDistance.py`

Let's now look at a simple Python program, which calls this web service:

```
import urllib
URL = "http://127.0.0.1:8000/cgi-bin/greatCircleDistance.py"
params = urllib.urlencode({'lat1'  : 53.478948, # Manchester.
                           'long1' : -2.246017,
                           'lat2'  : 53.411142, # Liverpool.
                           'long2' : -2.977638})
f = urllib.urlopen(URL, params)
response = f.read()
f.close()
print response
```

Running this program tells us the distance in meters between these two coordinates, which happen to be the locations of Manchester and Liverpool in England:

```
% python callWebService.py
49194.46315
```

While this might not seem very exciting, web services are an extremely important part of web-based development. When developing your own web-based geospatial applications, you may well make use of existing web services, and potentially implement your own web services as part of your web application development.

Map rendering

We saw in *Chapter 8, Using Python and Mapnik to Generate Maps*, how Mapnik can be used to generate good looking maps. Within the context of a web application, map rendering is usually performed by a web service which takes a request and returns the rendered map image file. For example, your application might include a map renderer at the relative URL `/render` which accepts the following query parameters:

- `minX`, `maxX`, `minY`, `maxY`: The minimum and maximum latitude and longitude for the area to include on the map.
- `width`, `height`: The pixel width and height for the generated map image.
- `layers`: A comma-separated list of layers which are to be included on the map. The available predefined layers are: coastline, forest, waterways, urban, and street.
- `format`: The desired image format. Available formats are: png, jpeg, gif.

This hypothetical `/render` web service would return the rendered map image back to the caller. Once this has been set up, the web service would act as a black box providing map images upon request for other parts of your web application.

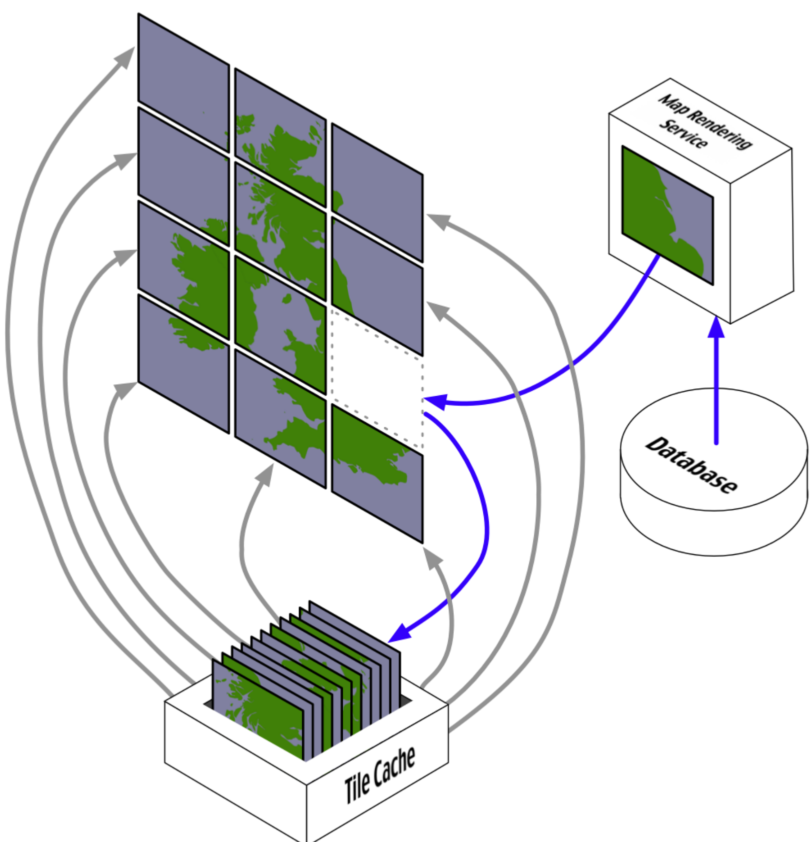
As an alternative to hosting and configuring your own map renderer, you can choose to use an openly available external renderer. For example, OpenStreetMap provides a freely-available map renderer for OpenStreetMap data at:

`http://staticmap.openstreetmap.de`

Tile caching

Because creating an image out of raw map data is a time and processor intensive operation, your entire web application can be overloaded if you get too many requests at any one time. As we saw with the DISTAL application in *Chapter 7*, *Working with Spatial Data*, there is a lot you can do to improve the speed of the map-generation process, but there are still limits on how many maps your application can render in a given time period.

Because map data is generally quite static, you can get a huge improvement in your application's performance by caching the generated images. This is generally done by dividing the world up into tiles, rendering tile images as required, and then stitching the tiles together to produce the desired map:

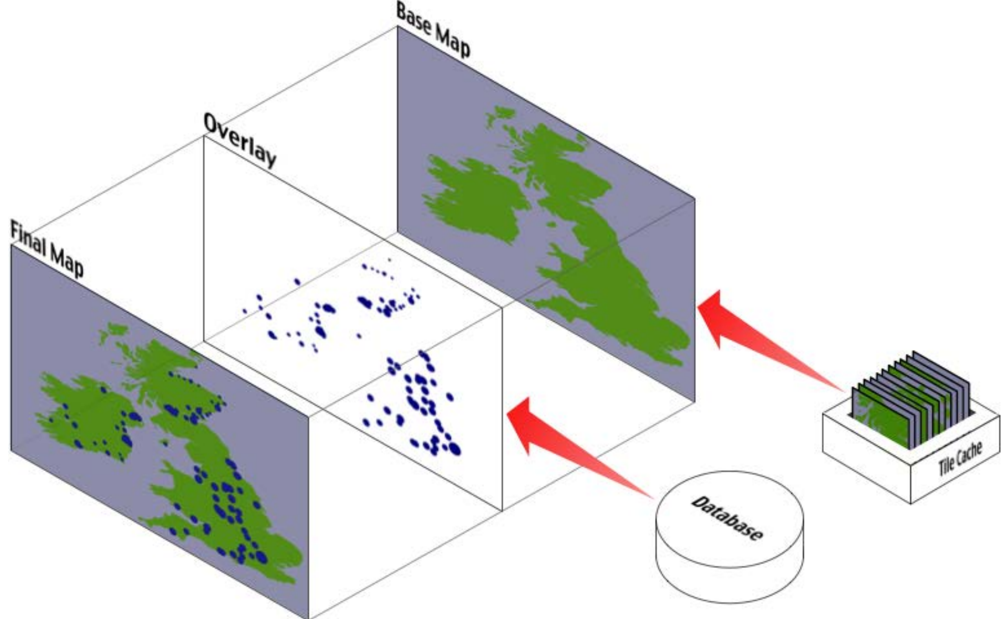


Tile caches work in exactly the same way as any other cache:

- When a tile is requested, the tile cache checks to see if it contains a copy of the rendered tile. If so, the cached copy is returned right away.
- Otherwise, the map rendering service is called to generate the tile, and the newly-rendered tile is added to the cache before returning it back to the caller.
- As the cache grows too big, tiles which haven't been requested for a long time are removed to make room for new tiles.

Of course, tile caching will only work if the underlying map data doesn't change. As we saw when building the DISTAL application, you can't use a tile cache where the rendered image varies from one request to the next.

One interesting use of a tile cache is to combine it with map overlays to improve performance even when the map data does change. Because the outlines of countries and other physical features on a map don't change, it is possible to use a map generator with a tile cache to generate the base map onto which changing features are then drawn as an overlay:

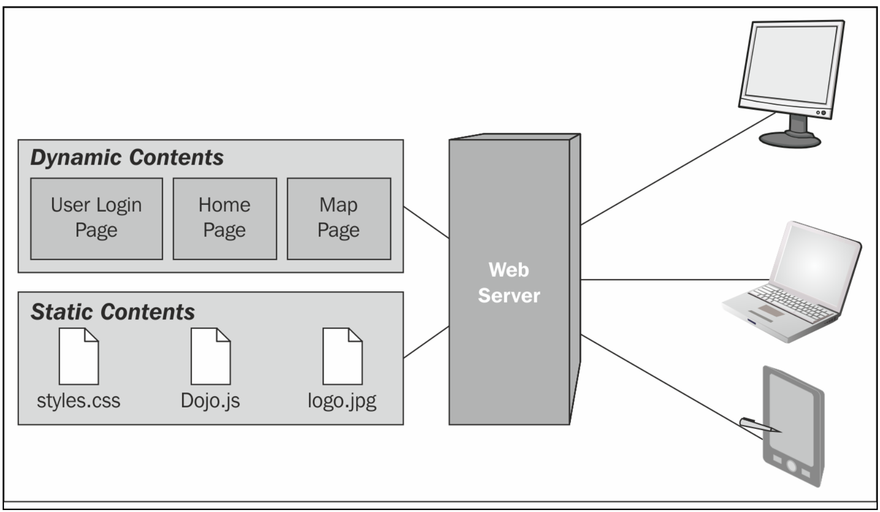


There are two possible ways of combining a base map with an overlay such as this. One way is to use Mapnik to combine the base map with the overlay, reading the base map image using a RasterDataSource, and displaying it via a RasterSymbolizer, and then make the combined map tiles available via a web service.

The second possibility is to combine the base map and the overlay directly within the web browser, using tools such as OpenLayers or Leaflet, which we will examine later in this chapter. This approach has the advantage of reducing the amount of map-rendering that you have to do on the server, at the disadvantage of increasing the amount of work that must be done at the browser level.

Web servers

In many ways a web server is the least interesting part of a web application: the web server listens for incoming HTTP requests from web browsers and returns either static content or the dynamic output of a program in response to these requests:



There are many different types of web servers, ranging from the pure-python SimpleHTTPServer included in the Python standard library, which we've used to serve the CGI scripts we've written, through more fully-featured servers such as CherryPy, Gunicorn, and of course the most popular industrial-strength web server of them all: Apache. Other more specialized web servers such as Nginx are also gaining in popularity.

One of the main consequences of your choice of web server is how fast your application will run. Obviously, a pure Python web server will be slower than a compiled high-performance server such as Apache. But there are more subtle differences as well. For example, even if you're using Apache, running a Python-based CGI script will cause the entire Python interpreter to be started up every time a request is received.

These types of issues make it extremely important that you design and implement your web application correctly. A slow web server doesn't just affect your application's responsiveness: if your server runs slowly, it won't take many requests to overload the server.

Another consequence of your choice of web server is how your application's code interacts with the end user. The HTTP protocol itself is **stateless**—that is, each incoming request is completely separate, and a web page handler has no way of knowing what the user has done previously unless you explicitly code your pages in such a way that the application's state is passed from one request to the next (for example, using hidden HTML form fields).

Because some web servers run your Python code only when a request comes in, there is often no way of having a long-running process sitting in the background that keeps track of the user's state or performs other capabilities for your web page handlers. For example, an in-memory cache might be used to improve performance, but you can't easily use such a cache with CGI scripts as the entire interpreter is restarted for every incoming HTTP request.

One of the big advantages of using a web application framework is that you don't need to worry about these sorts of issues: in many cases, the web framework itself will include a simple web server you can use for development, and provides a standard way of using industry-standard web servers when you deploy your application. The challenges of performance, keeping track of the user's state, and using long-running processes will all be solved for you by the web framework. It is, however, worthwhile to understand some of the issues involved in the choice of a web server, and to know where the web server fits within the overall web application. This will help you to understand what your web framework is doing, and how to configure and deploy it to achieve the best possible performance.

User interface libraries

While it is easy to build a simple web-based interface in HTML, users are increasingly expecting web applications to compete with desktop applications in terms of their user interface. Selecting objects by clicking on them, drawing images with the mouse, and dragging-and-dropping are no longer actions restricted to desktop applications.

JavaScript programs running within the web browser allow you to dynamically update the contents of the page, rather than having a fixed page that has to be reloaded every time something changes. These dynamic pages are often complemented by the use of **AJAX** (**Asynchronous JavaScript and XML**) to download data as required from the server. Working together, JavaScript and AJAX can be used to build sophisticated web applications that behave in ways impossible to achieve with static web pages.

While JavaScript is ubiquitous, it is also hard to program in. The various web browsers in which the JavaScript code can run all have their own quirks and limitations, making it hard to write code that runs the same way on every browser. JavaScript code is also very low level, requiring detailed manipulation of the web page contents to achieve a given effect. For example, implementing a pop-up menu requires the creation of a `<DIV>` element that contains the menu, formatting it appropriately (typically using CSS), and making it initially invisible. When the user clicks on the page, the pop-up menu should be shown by making the associated `<div>` element visible. You then need to respond to the user mousing over each item in the menu by visually highlighting that item and un-highlighting the previously highlighted item. Then when the user clicks, you have to hide the menu again before responding to the user's action.

All this detailed low-level coding can take weeks to get right, especially when dealing with multiple types of browsers and different browser versions. Since all you want to do in this case is have a pop-up menu that allows the user to choose an action, it just isn't worth doing all this low-level work yourself. Instead, you would normally make use of one of the available user interface libraries to do all the hard work for you.

These user interface libraries are written in JavaScript, and you typically add them to your website by making the JavaScript library file(s) available for download, and then adding the following line to your HTML page to import the JavaScript library:

```
<script type="text/javascript" src="library.js">
```

If you are writing your own web application from scratch, you would then make calls to the library to implement the user interface for your application. However, many of the web application frameworks that include a user interface library will write the necessary code for you, making even this step unnecessary.

There are many different types of user interface libraries which you can make use of. As well as general UI libraries such as JQuery UI and Twitter Bootstrap which provide a desktop-like experience, there are other libraries specifically designed for implementing geospatial web applications. We will explore some of these later in this chapter.

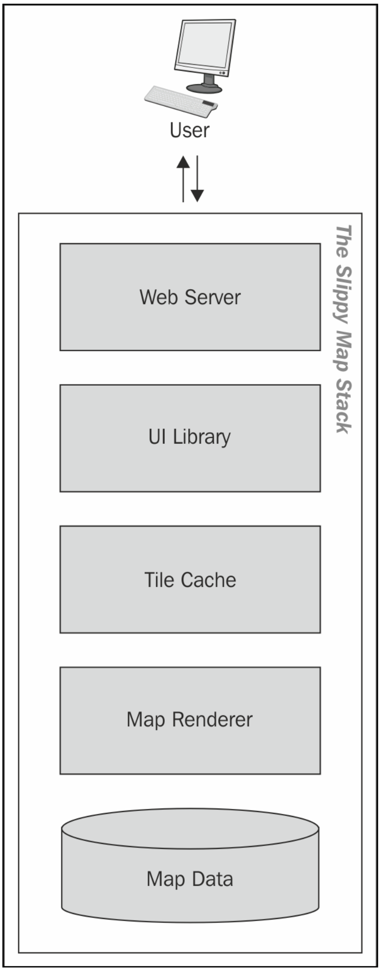
The slippy map stack

The slippy map is a concept popularized by Google maps: a zoomable map where the user can click-and-drag to scroll around and double-click to zoom in. Here is an example of a Google maps slippy map showing a portion of Europe:

Image copyright Google; map data copyright Europa Technologies, PPWK, Tele Atlas

Slippy maps have become extremely popular, and much of the work done on geospatial web application development has been focused on creating and working with slippy maps.

The slippy map experience is typically implemented using a custom software stack with the following components:



Starting from the bottom, the raw map data is typically stored in a shapefile or database. This is then rendered using a tool such as Mapnik, and a tile cache is used to speed up repeated access to the same map images. A user-interface library such as OpenLayers is then used to display the map in the user's web browser, and to respond when the user clicks on the map. Finally, a web server is used to allow web browsers to access and interact with the slippy map.

The geospatial web application stack

The slippy map stack is intended to display a slippy map within a web page, allowing the user to view a map but not generally to make any changes. A more comprehensive solution allows the user to not only view maps, but also to make changes to geospatial data from within the web application itself and perform other functions such as analyzing data and performing spatial queries. A complete geospatial web application stack would consist of a web application framework with an integrated slippy map stack and built-in tools for editing, querying, and analyzing geospatial data.

In many ways, the geospatial web application stack is the epitomé of geospatial web development: it allows for rapid development of geospatial applications with a minimum of coding and using existing libraries to do almost all the hard work. While you still need to understand how the web application framework (and its geospatial extensions) operate, and there are bugs and technical issues to be considered, these frameworks can save you a tremendous amount of time and effort compared with a "roll your own" solution.

Protocols

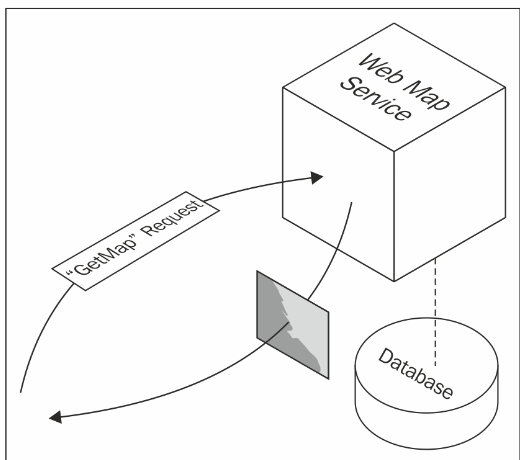
Because web-based applications are generally broken into multiple components, the way these components communicate becomes extremely important. It's quite likely that your web application will use off-the-shelf components or rely on existing components running on a remote server. In these cases, the protocols used to communicate between the various components are crucial to allow these various components to work together.

In terms of geospatial web applications, a number of standard protocols have been developed to allow different components to communicate. For example, the **Web Map Service** (**WMS**) protocol provides a standard way for a web service to receive a map-generation request and return the map image back to the caller.

In this section, we will examine some of the major protocols relating to geospatial web applications. While there are some tools that choose not to use these protocols, they are still frequently used in geospatial development, and so it is worthwhile becoming at least passingly familiar with them.

The Web Map Service protocol

The Web Map Service protocol defines the interface to a web service that creates map images upon request:



At a minimum, the Web Map Service needs to implement the following two HTTP requests:

- GetCapabilities

  This request returns information about the Web Map Service itself, in the form of an XML document describing the web service, including:

  - Which operations are supported by the web service
  - The maximum width and height of the generated map, in pixels
  - The maximum number of layers which can be included in the map
  - A list of the available map layers
  - A list of the various visual styles which can be applied to the map's features
  - A latitude/longitude bounding box defining the area of the Earth the Web Map Service can generate maps for
  - Which Coordinate Reference System is used by the map's data
  - The range of scale factors at which the map can be generated
  - A URL linking to the underlying map data

- GetMap

  This request generates and returns an actual map image based on the supplied parameters. The supplied parameters include:

  - A comma-separated list of the layers to include in the map
  - A comma-separated list of styles to apply to the map
  - A code indicating which Coordinate Reference System is used by the supplied bounding box parameters. For example, the code `CRS:84` indicates that the coordinates are longitude and latitude values using the WGS 84 datum
  - The bounding box defining the area of the Earth to be covered by the map
  - The width and height of the generated map image, in pixels
  - The image format to use for the generated map

  The GetMap request will return the generated map as an image file of the requested format. For example, if the request parameters included `FORMAT=JPEG`, the returned data would be a JPEG-format image.

The Web Map Service may also optionally implement the following request:

- GetFeatureInfo

  Returns more detailed information about the feature or features at a given coordinate within a rendered map image. The parameters used by this request include:

  - The map-generation parameters used to create a map image
  - The pixel coordinate of a desired point in the rendered map image

  Upon completion, this request returns information about the features or features at or near the given position in the rendered map image. The results are usually in XML format. Note that the exact information returned by a GetFeatureInfo request is not specified by the WMS specification.

For more information about the WMS protocol, you can find the complete specification on the open geospatial consortium's web site:

`http://www.opengeospatial.org/standards/wms`

WMS-C

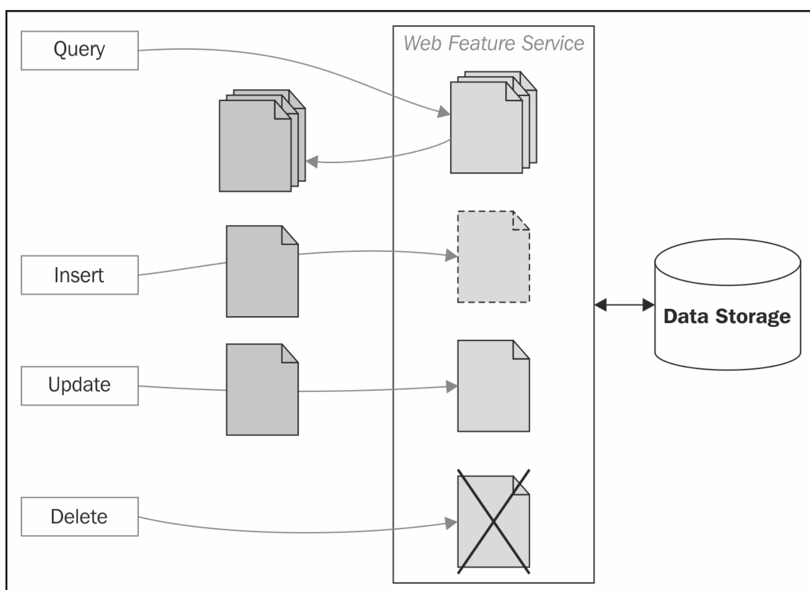
Because the WMS protocol generates arbitrary map images, it doesn't produce map tiles which can be easily cached. To get around this limitation, a set of recommendations were made to limit the way in which WMS operates, to make it more suitable for serving tiled images. This recommendation, known as WMS-C or the **WMS Tiling Client Recommendation**, ensures that the generated map images consist of fixed-size tiles. It also suggests extensions to the WMS protocol to make it clear that the rendered map images are map tiles.

More details about the WMS-C protocol can be found at:

`http://wiki.osgeo.org/wiki/WMS_Tiling_Client_Recommendation`

The Web Feature Service (WFS) protocol

The WFS protocol defines a web service that allows other parts of a web application to query and manipulate geospatial features independently of how those features are stored:



A web feature service represents geospatial features using **Geography Markup Language** (**GML**), which is an XML schema for storing and representing geographical information. GML is an international standard, allowing features to be represented and stored in a platform-agnostic way.

At a minimum, a web feature service needs to support the following requests, which allow client systems to access the WFS and retrieve features:

- `GetCapabilities`

  This request returns information about the web feature service itself, in the form of an XML document which describes:

  - The operations supported by the web service
  - Which types of features can be stored and retrieved by the web service
  - Which operations are supported by each type of feature

- `DescribeFeatureType`

  This request returns an XML document describing the structure of one or more types of features. This provides information about the attributes stored for each feature, as well as the way in which the feature itself is represented in the datastore.

- `GetFeature`

  This request queries the web feature service, returning features which match certain criteria. The caller can request which properties to retrieve and a maximum number of matching features to return, as well as both spatial and non-spatial query parameters.

The web feature service may also choose to support client systems adding, updating, and deleting features. This can be done in one of two ways: by allowing the client to lock one or more features before making a series of changes and then unlocking the features again, or simply by making the updates one at a time. The locking approach ensures that multiple processes don't update the same feature at the same time, although not all web feature service support locking.

The following requests support locking and non-locking changes to the datastore:

- `LockFeature`

  Lock one or more features so that other processes cannot make any changes to those features.

- `GetFeatureWithLock`

  Retrieve one or more features, and immediately lock the retrieved features.

- `Transaction`

  This request is used to add, update and delete features. It also allows previously-locked features to be unlocked, allowing other processes to make changes to those features.

Finally, the web feature service can optionally support external linking, where features (possibly stored within different web feature service) can be linked together. This is done through supporting the retrieval of nested features within the GetFeature request, and the separate GetGmlObject request which returns a given feature referred to by an XLink ID.

Web feature service are intended to abstract the storage and retrieval away from other parts of a web application, allowing different datastores to be used, and to allow information stored in separate places (possibly on separate servers) to be seamlessly combined. Unfortunately, the WFS protocol is quite complicated, relying heavily on complex XML schemas, which makes accessing and using a web feature service somewhat challenging. Despite this, the open and scalable nature of the WFS protocol does make it worthwhile. Depending on your requirements, you may wish to make use of this protocol in your applications, especially if you are trying to access or manipulate data stored externally.

More information about web feature service, including a complete specification, can be found at:

`http://www.opengeospatial.org/standards/wfs`

The TMS (Tile Map Service) protocol

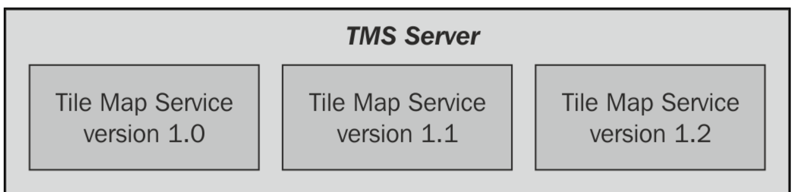
The TMS protocol defines the interface to a web service which returns map tile images upon request. The TMS protocol is similar to WMS, except that it is simpler and more oriented towards the storage and retrieval of map tiles rather than arbitrarily-specified complete maps.

The Open Geospatial Consortium has produced a new standard called **WMTS** (**Web Map Tile Service**) that theoretically supercedes the TMS protocol. However, TMS is simpler and is supported by all major libraries, unlike WMTS, so in this book we will concentrate on TMS rather than WMTS.

The TMS protocol uses RESTful principles, which means that the URL used to access the web service includes all of the information needed to complete a request. Unlike WMS, there is no need to create and submit complex XML documents to retrieve a map tile—all of the information is contained within the URL itself. This has the advantage of allowing standard HTTP caching to be used for map tiles.

Within the TMS protocol, a Tile Map Service is a mechanism for providing access to rendered map images at a given set of scale factors and using a predetermined set of spatial reference systems.

A single TMS server can host multiple Tile Map Services:

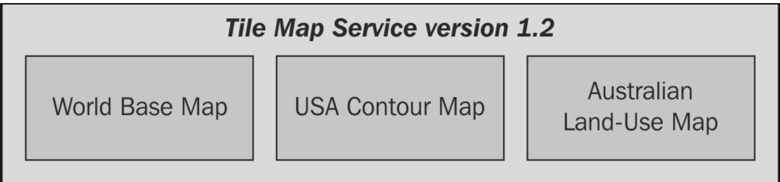


This is typically used to have different versions of a Tile Map Service available, so that new versions of the Tile Map Service can be implemented without breaking clients that depend on features in an older version.

Each Tile Map Service within a TMS server is identified by a URL that is used to access that particular service. For example, if a TMS server is running at `http://tms.myserver.com`, Version 1.2 of the Tile Map Service running on that server would generally reside at the sub-URL `http://tms.myserver.com/1.2/`. Accessing the top-level URL (that is, `http://tms.myserver.com`) returns a list of all the Tile Map Services available on that server:

```
<?xml version="1.0" encoding="UTF-8"/>
<Services>
  <TileMapService title="MyServer TMS" version="1.0"
  href="http://tms.myserver.com/1.0/"/>
  <TileMapService title="MyServer TMS" version="1.1"
  href="http://tms.myserver.com/1.1/"/>
  <TileMapService title="MyServer TMS" version="1.2"
  href="http://tms.myserver.com/1.2/"/>
</Services>
```

Each Tile Map Service provides access to one or more Tile Maps:

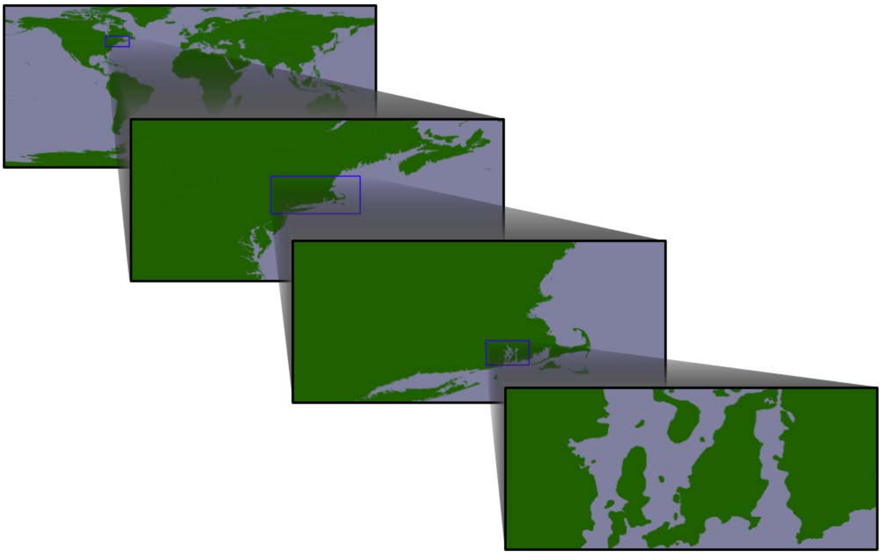


A Tile Map is a complete map of all or part of the Earth, displaying particular sets of features or styled in a particular way. The previous examples, of a worldwide base map, a contour map, and a land-use map, show how different Tile Maps might contain different sorts of map data or cover different areas of the Earth's surface. Different Tile Maps may also be used to make maps available in different image formats, or to provide maps in different spatial reference systems.

If a client system accesses the URL for a particular Tile Map Service, the server will return more detailed information about that service, including a list of the Tile Maps available within that service:

```
<?xml version="1.0" encoding="UTF-8"/>
<TileMapService version="1.2" services="http://tms.myserver.com">
  <Title>MyServer TMS</Title>
  <Abstract>TMS Service for the myserver.com server</Abstract>
  <TileMaps>
    <TileMap title="World Base Map"srs="EPSG:4326"profile="none"
    href="http://tms.myserver.com/1.2/baseMap"/>
    <TileMap title="USA Contour Map"srs="EPSG:4326"rofile="none"
    href="http://tms.myserver.com/1.2/usaContours"/>
    <TileMap title="Australian Land-Use Map"srs="EPSG:4326"
    profile="none"href="http://tms.myserver.com/1.2/ausLandUse"/>
    </TileMap>
  </TileMaps>
</TileMapService>
```

Client systems accessing rendered maps via a TMS Server will generally want to be able to display that map at various resolutions. For example, a world base map might initially be displayed as a complete map of the world, and the user can zoom in to see a more detailed view of a desired area:



This zooming-in process is done through the use of appropriate scale factors. Each Tile Map consists of a number of Tile Sets, where each Tile Set depicts the map at a given scale factor. For example, the first image in the previous illustration was drawn at a scale factor of approximately 1:100,000,000, the second at a scale factor of 1:10,000,000, the third at a scale factor of 1:1,000,000, and the last at a scale factor of 1:100,000. Thus, there would be four Tile Sets within this Tile Map, one for each of the scale factors.

If a client system accesses the URL for a given Tile Map, the server will return information about that map, including a list of the available Tile Sets:

```
<?xml version="1.0" encoding="UTF-8">
<TileMap version="1.2"
tilemapservice="http://tms.myserver.com/1.2">
  <Title>World Base Map</Title>
  <Abstract>Base map of the entire world</Abstract>
  <SRS>ESPG:4326</SRS>
  <BoundingBox minx="-180" miny="-90" maxx="180" maxy="90"/>
  <Origin x="-180" y="-90"/>
```

```
    <TileFormat width="256"height="256"mime-type="image/png"
    extension="png"/>
    <TileSets profile="none">
      <TileSet href="http://tms.myserver.com/1.2/basemap/0"
      units-per-pixel="0.703125"order="0"/>
      <TileSet href="http://tms.myserver.com/1.2/basemap/1"
      units-per-pixel="0.3515625"order="1"/>
      <TileSet href="http://tms.myserver.com/1.2/basemap/2"
      units-per-pixel="0.17578125"order="2"/>
      <TileSet href="http://tms.myserver.com/1.2/basemap/3"
      units-per-pixel="0.08789063"order="3"/>
    </TileSets>
  </TileMap>
```

Notice how each Tile Set has its own unique URL. This URL will be used to retrieve the individual tiles within the Tile Set. Each tile is given an x and y coordinate value indicating its position within the overall map. For example, using the previously mentioned Tile Map covering the entire world, the third Tile Set would consist of 32 tiles arranged as follows:

x = 0 y = 3	x = 1 y = 3	x = 2 y = 3	x = 3 y = 3	x = 4 y = 3	x = 5 y = 3	x = 6 y = 3	x = 7 y = 3
x = 0 y = 2	x = 1 y = 2	x = 2 y = 2	x = 3 y = 2	x = 4 y = 2	x = 5 y = 2	x = 6 y = 2	x = 7 y = 2
x = 0 y = 1	x = 1 y = 1	x = 2 y = 1	x = 3 y = 1	x = 4 y = 1	x = 5 y = 1	x = 6 y = 1	x = 7 y = 1
x = 0 y = 0	x = 1 y = 0	x = 2 y = 0	x = 3 y = 0	x = 4 y = 0	x = 5 y = 0	x = 6 y = 0	x = 7 y = 0

This arrangement of tiles is defined by the following information taken from the Tile Map and the selected Tile Set:

- The Tile Map uses the ESPG:4326 spatial reference system, which equates to longitude/latitude coordinates based on the WGS84 datum. This means that the map data is using latitude/longitude coordinate values, with longitude values increasing from left to right, and latitude values increasing from bottom to top.

- The map's bounds range from -180 to +180 in the x (longitude) direction, and from -90 to +90 in the y (latitude) direction.
- The map's origin is at (-180,-90)—that is, the bottom-left corner of the map.
- Each tile in the Tile Map is 256 pixels wide and 256 pixels high.
- This Tile Set has a units-per-pixel value of 0.17578125.

Multiplying the units-per-pixel value by the tile's size, we can see that each tile covers 0.17578125 * 256 = 45 degrees of latitude and longitude. Since the map covers the entire Earth, this yields eight tiles across and four tiles high, with the origin in the bottom-left corner.

Once the client software has decided on a particular Tile Set to use, and has calculated the x and y coordinates for the desired tile, retrieving that tile's image is a simple matter of concatenating the Tile Set's URL, the x and y coordinates, and the image file suffix:

```
url = tileSetURL + "/" + x + "/" + y + "." + imgFormat
```

For example, to retrieve the tile at coordinate (2, 3) from Tile Set number 1, you would use the following URL:

```
http://tms.myserver.com/1.2/basemap/1/2/3.png
```

Notice how this URL (and indeed, every URL used by the TMS protocol) looks as if it is simply retrieving a file from the server. Behind the scenes, the TMS server may indeed be running a complex set of map-generation and map-caching code to generate these tiles on demand—but the entire TMS server could just as easily be defined by a series of hardwired XML files and a number of directories containing pre-generated image files.

This notion of a **Static Tile Map Server** is a deliberate design feature of the TMS protocol. If you don't need to generate too many map tiles, or if you have a particularly large hard disk, you can pre-generate all the tile images and create a static TMS server by creating a few XML files and serving the whole thing behind a standard web server such as Apache.

While you might not implement your own dynamic TMS Server from scratch, you may well wish to make use of TMS servers in your own web applications, either by creating a Static Tile Map Server, or by using an existing software library that implements the TMS protocol such as the open-source TileCache server. TileCache will be discussed in the next section of this chapter.

The full specification for the TMS protocol can be found at:

`http://wiki.osgeo.org/wiki/Tile_Map_Service_Specification`

Tile-Numbering Issues with TMS

There is a subtle difference in the way the TMS protocol numbers map tiles compared with many other mapping systems. With TMS, the smallest Y value represents the bottom of the map, and the largest Y value represents the top. This is the opposite of many other tile-serving systems, including Google Maps and OpenStreetMap, where the lowest Y value is at the top of the map. This can lead to problems if your server outputs tiles in the Google Maps or OpenStreetMap style but your client is assuming tiles numbered according to the TMS protocol, or vice versa. If your tiles are appearing in the wrong order, you may need to flip the Y axis.

Tools

When discussing tools for developing geospatial web applications, it is worth remembering that all of the libraries and toolkits we have discussed in earlier chapters (SpatiaLite, MySQL, PostGIS, Mapnik, OGR, GDAL, Proj, Shapely, and so on) can also be used for web applications. In this section, we will add to this collection by examining some of the major Python libraries available for implementing tile caching and slippy maps. We will also look at some of the web application frameworks which support geospatial development.

Tile Caching

While there are several tools for implementing tile caching within a Python geospatial web application, many of them have not been updated for some time. Let's take a closer look at one that is in active development, TileStache, as well as the venerable TileLite, a lightweight but still-useful tile server written in Python.

TileStache

TileStache (`http://tilestache.org`) is a powerful and flexible Python-based map server that can act as a tile cache and much more. It was written as the successor to the earlier TileCache WMS server (`http://tilecache.org`) which still works but is no longer being developed.

TileStache can be run in four different ways:

1. As a WSGI application, either using the built-in werkzeug server, or as part of an external WSGI server such as Gunicorn.
2. As a CGI script.
3. As a `mod_python` module running within the Apache web server.
4. TileStache can also be called directly from your Python program.

TileStache can render map images using Mapnik, it can forward (and cache) map data from other WMS providers, and it can read geospatial data directly from shapefiles, spatial databases and GeoJSON data sources, among others.

As the tiles are rendered, they can be cached in a variety of ways, including storing them in files on the local disk, using the memory-based memcached daemon, and caching tiles on cloud services such as Amazon S3.

TileStache uses a URL scheme similar to the TMS protocol:

```
http://myserver.com/tilestache/[layer]/[zoom]/[x]/[y].[format]
```

where `layer` is the name of the desired layer, `zoom` is the desired zoom level, `x` and `y` are the coordinate for the desired tile, and `format` is the desired output format. Note that the returned data can either be a rendered image (in PNG or JPEG format), or raw geospatial data (in JSON format) that can then be displayed directly by the web browser, for example, by using a vector layer within the OpenLayers library.

TileStache includes a built-in `preview` function that displays a slippy map in a web browser, making it easy to test and see how your map tiles are being rendered.

Configuration is done via JSON-format files, giving the developer a lot of flexibility in how TileStache works and what information it displays.

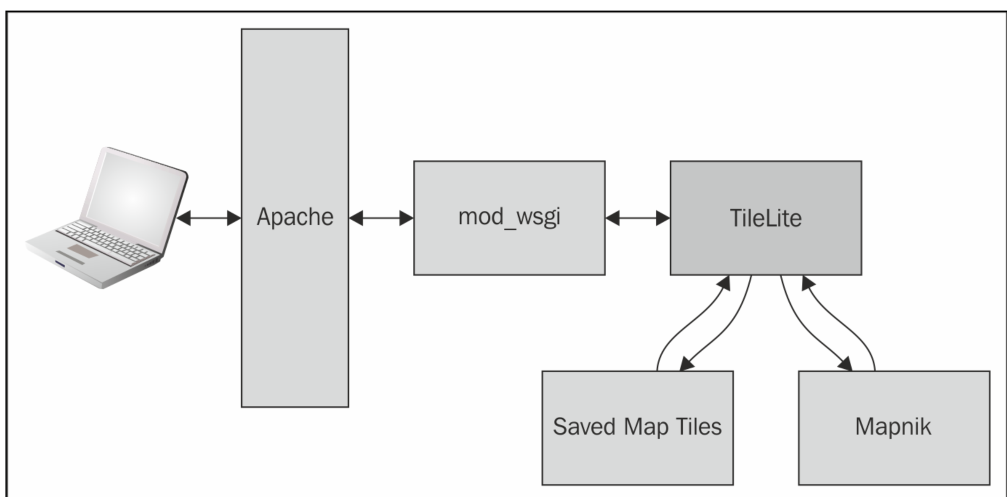
TileStashe takes a pragmatic approach to map tiling. Rather than conform to a standard, it provides a quick and powerful tile-caching system along with a simple URL scheme that allows these tiles to be retrieved by client software as required. For example, the OpenLayers library supports an XYZ layer which can retrieve tiles from TileStache.

TileLite

As the name suggests, TileLite (`http://bitbucket.org/springmeyer/tilelite`) is a lightweight tile server, written in Python. It serves tiles rendered using Mapnik, and uses WSGI for interacting with a web server. TileLite is easy to install and configure, and comes with its own server that you can use for development purposes. While it has not been updated in some time, it is still a useful lightweight tile server.

For deployment in a high-performance environment, you can combine TileLite with the mod_wsgi module to use TileLite within Apache. Because TileLite is a long-running process, it has a single Mapnik Map object that is held in memory to quickly produce map tiles on demand.

TileLite works with Mapnik, mod_wsgi, and Apache in the following way:



TileLite makes the map tiles available using the following URL scheme:

```
http://myserver.com/tileserver/[zoom]/[x]/[y].png
```

Because it is written in pure Python, it is easy to explore the TileLite source code and see how it works. It is fast and simple, and may well be suited to providing the tile caching needs of your geospatial web application.

User interface libraries

JavaScript code running on the user's web browser, in conjunction with AJAX technology, has made it possible to include complex user interfaces previously only seen on desktop-based GUI systems. Because of the complexity of the JavaScript code needed to achieve commonly-used parts of a user interface, a number of large and powerful UI libraries have been developed to simplify the task of building a complex web interface. Twitter Bootstrap, script.aculo.us, JQuery-UI, and YUI are examples of some of the more popular JavaScript user interface libraries.

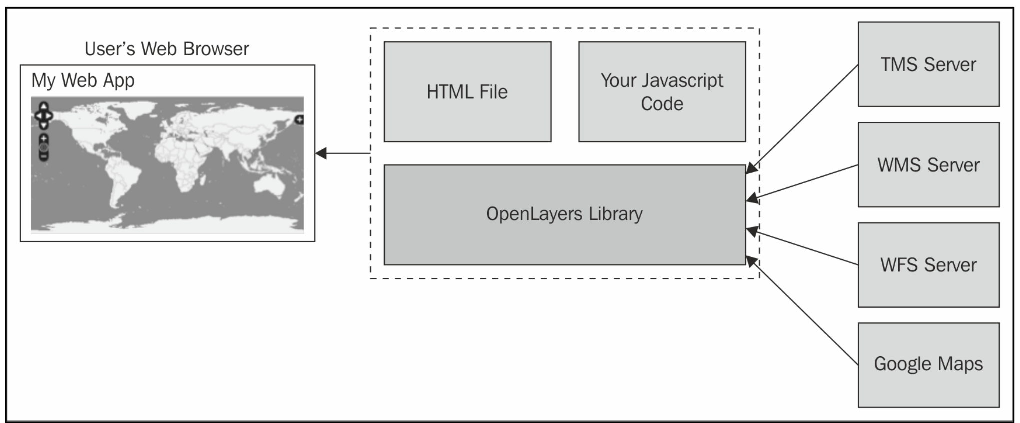
It is easy to forget that geospatial web applications are, first and foremost, ordinary web applications that also happen to work with geospatial data. Much of a geospatial web application's functionality is rather mundane: providing a consistent look and feel, implementing menus or toolbars to navigate between pages, user signup, login and logout, entry of ordinary (non-geospatial) data, reporting, and so on. All of this functionality can be handled by one of these general-purpose geospatial libraries, and you are free to either choose one or more libraries of your liking, or make use of the UI library built into whatever web application framework you have chosen to use.

These general-purpose user interface libraries, and the process of using them to implement non-geospatial functionality, has been covered by many other books and web sites. We will not look at them in depth here. Instead, we will take a closer look at two of the UI libraries specifically aimed at viewing or editing geospatial data via a slippy map interface: the fully-featured OpenLayers, and the simpler Leaflet library.

OpenLayers

OpenLayers (`http://openlayers.org`) is a sophisticated JavaScript library for building mapping applications. It includes a JavaScript API for building slippy maps, combining data from multiple layers, and includes various widgets for manipulating maps as well as viewing and editing vector-format data.

To use OpenLayers in your web application, you first need to create an HTML file to be loaded into the user's web browser, and write some JavaScript code which uses the OpenLayers API to build the desired map. OpenLayers then builds your map and allows the user to interact with it, loading map data from the various data source(s) you have specified. OpenLayers can read from a variety of geospatial data sources, including TMS, WMS, and WFS servers. All these various parts work together to produce the user interface for your web application in the following way:



> To use OpenLayers you have to be comfortable writing JavaScript code. This is almost a necessity when creating your own web applications. Fortunately, the OpenLayers API is very high level, and makes map-creation relatively simple.

Here is an example HTML page that displays a slippy map using OpenLayers:

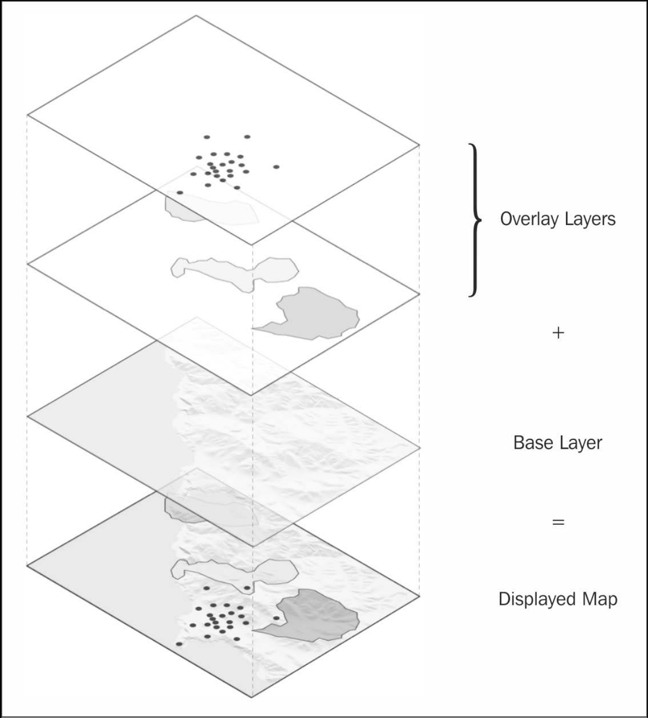
```
<html>
  <head>
    <script src="http://openlayers.org/api/OpenLayers.js">
    </script>
    <script type="text/javascript">
      function initMap() {
        var map = new OpenLayers.Map("map");
        var layer = new OpenLayers.Layer.WMS("Layer",
        "http://labs.metacarta.com/wms/vmap0",
        {layers: 'basic'} );
        map.addLayer(layer);
        map.zoomToMaxExtent();
      }
    </script>
  </head>
  <body onload="initMap()">
    <div style="width:100%; height:100%" id="map"></div>
  </body>
</html>
```

As you can see, the map uses a block-level element, in this case a `<div>` element, to hold the map. Initializing the map involves a short JavaScript function which defines the map object, adds a layer, and prepares the map for display.

Internally, the OpenLayers API uses a `Map` object to represent the slippy map itself, and one or more `Layer` objects which represent the map's data sources.

As with Mapnik layers, multiple OpenLayers layers can be laid on top of each other to produce the overall map image, and layers can be shown or hidden depending on the map's current scale factor.

There are two types of layers supported by OpenLayers: Base Layers and Overlay Layers. Base layers sit behind the overlay layers, and are generally used to display raster format data (images, generated map tiles, and so on). Overlay layers, on the other hand, sit in front of the base layers and are generally used to display vector format data, including points, lines, polygons, bounding boxes, text, markers, and so on:



Different `Layer` subclasses represent different types of data sources: `OpenLayers.Layer.TMS`, `OpenLayers.Layer.WMS`, `OpenLayers.Layer.Google`, and so on. In addition, the `OpenLayers.Layer.Vector` class represents vector-format data which can be loaded from a variety of sources, and optionally edited by the user. To have the vector layer read features from the server, you set up a `Protocol` object which tells OpenLayers how to communicate with the server. The most common protocol is HTTP, though several other protocols including WFS are also supported. The `Protocol` object usually includes the URL used to access the server.

As well as setting the protocol, you also supply the format used to read and write data. Supported formats include GML, GeoJSON, GeoRSS, OSM, and WKT, among others. The `Format` objects also support on-the-fly reprojection of vector data so that data from different sources, using different map projections, can be combined onto a single map.

In addition to the map itself, OpenLayers allows you to use various `Control` objects, either embedded within the map or shown elsewhere on the web page. `Control` objects include simple push-buttons, controls for panning and zooming the map, controls which display the map's current scale, controls for showing and hiding layers, and various controls for selecting, adding, and editing vector features. There are also `invisible` controls which change the behavior of the map itself. For example, the `Navigation` control allows the user to pan and zoom by clicking-and-dragging on the map, and the `ArgParser` control tells OpenLayers to scan the URL's query string during page load for arguments which adjust how the map is initially displayed.

OpenLayers is a very powerful tool for building geospatial web interfaces. Even if you don't use it directly in your own code, many of the web application frameworks which support geospatial web development (including TurboGears, Mapfish, and GeoDjango) use OpenLayers internally to display and edit map data.

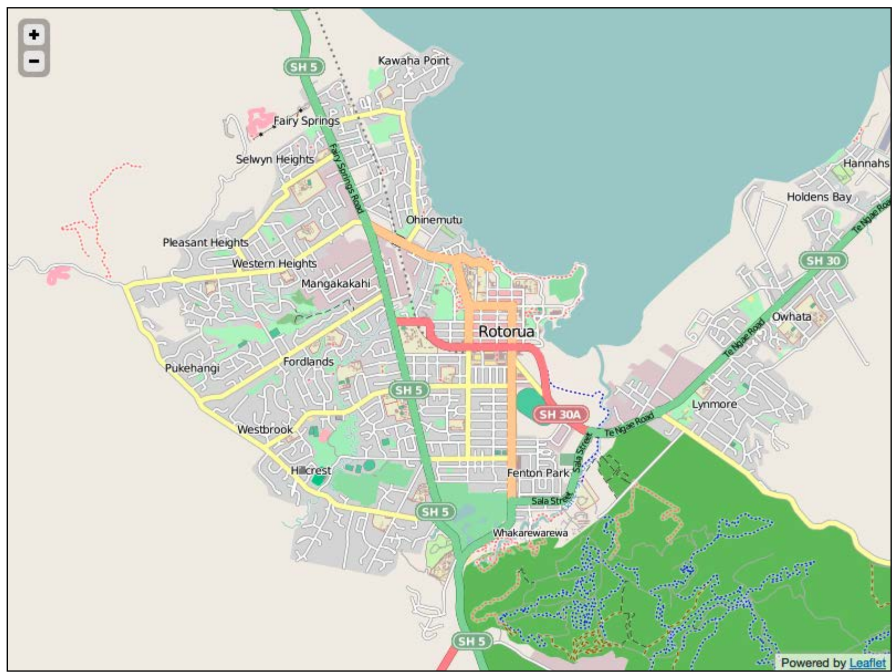
Leaflet

Leaflet (`http://leafletjs.com`) is another JavaScript library for including slippy maps within a web application. It is a lean and simple library, and very easy to use. Here, for example, is an HTML page that displays a map using Leaflet:

```
<html>
  <head>
    <title>Example Map</title>
    <link rel="stylesheet" href="http://cdn.leafletjs.com/
     leaflet-0.4.5/leaflet.css"/>
    <!--[if lte IE 8]>
```

```
    <link rel="stylesheet" href="http://cdn.leafletjs.com/leaflet
    0.4.5/leaflet.ie.css" />
    <![endif]-->
    <script src="http://cdn.leafletjs.com/leaflet
0.4.5/leaflet.js"></script>
    <script type="text/javascript">
        function onLoad() {
          var map = L.map("map");
          map.setView([-38.139, 176.244], 13);
          var layer =
L.tileLayer("http://tile.openstreetmap.org/{z}/{x}/{y}.png");
          map.addLayer(layer);
      }
    </script>
  </head>
  <body onload="onLoad()">
      <div id="map" style="width:800px; height:600px">
      </div>
  </body>
</html>
```

Running this page results in the following slippy map being displayed:



Leaflet makes it easy to include multiple layers on a map (including overlaying vector data on top of a base map), and is easy to configure to work with any of the standard protocols for accessing map tiles. You can also add markers and popups to your maps, and various controls that respond to mouse clicks and other UI events. As well as being lean and simple, Leaflet also works well with mobile browsers, making it in many ways an ideal JavaScript library for working with slippy maps.

Web application frameworks

In this section, we will examine three of the major Python-based web application frameworks that also support geospatial web application development.

All three of these frameworks are highly usable and in active development. While we will select a particular framework to use in the latter chapters of this book, any one of these three would be a suitable choice for your geospatial web development, and which one you decide on is largely a matter of personal preference.

GeoDjango

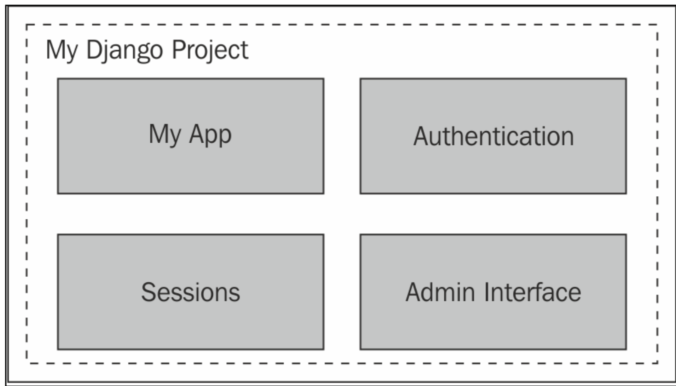
Django (`http://djangoproject.org`) is a rapid application development framework for building database-oriented web applications using Python. GeoDjango is a set of extensions to Django which add geospatial capabilities to the Django framework.

To understand how GeoDjango works, it is first necessary to understand a little about Django itself. Let's start by taking a closer look at Django.

Understanding Django

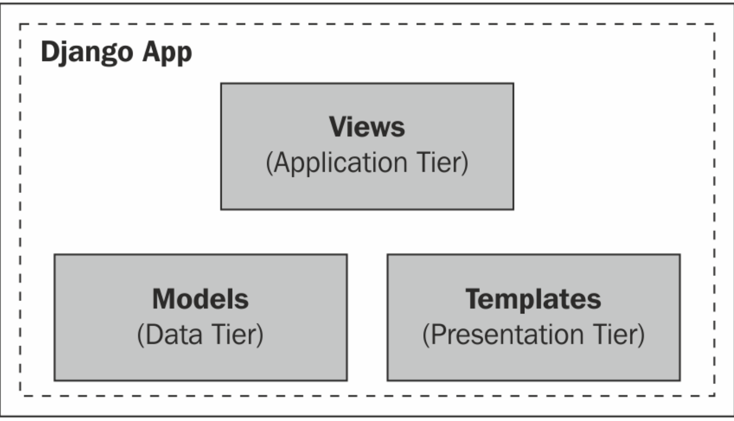
The Django framework is highly respected, and used to power thousands of web applications currently deployed across the internet. The major parts of Django include an object-relational mapper, an automatically-generated admin interface, a flexible URL mapper, and an HTML templating engine. Putting these elements together, Django allows you to quickly build sophisticated web applications to implement a wide variety of database-oriented systems.

A Django project consists of a number of apps, where each app implements a standalone set of functionality:



When creating a web application, you define your own app, and will typically make use of one or more predefined apps that come with Django. One of the most important predefined apps is the admin interface, which allows you to administer your web application, view and edit data, and so on. Other useful predefined apps implement persistent sessions, user authentication, site maps, user comments, sending emails, and serving static data. A large number of user-contributed apps are also available, providing features such as database migration (south), background task queuing (celery), image processing (django-imagekit), and much more.

Internally, each app consists of three main parts:



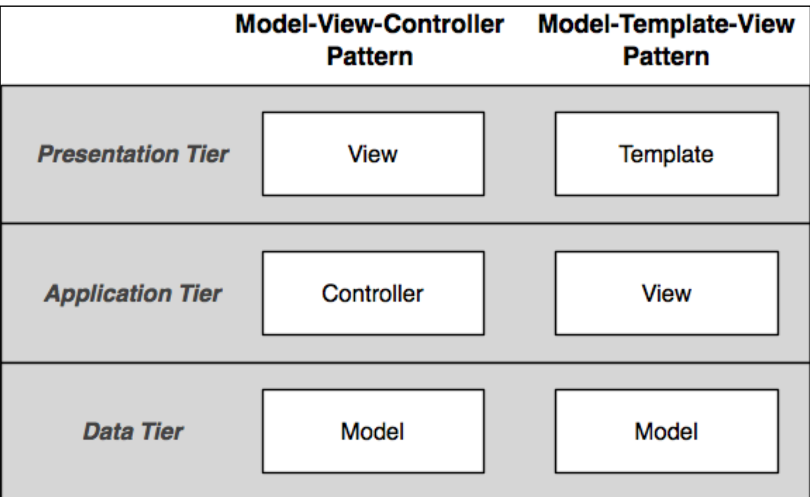
The models represent the app's data tier. This contains everything related to the application's data, how it is structured, how to import it, how to access it, how data is validated, and so on.

The templates make up the app's presentation tier. These describe how information will be presented to the user.

The views make up the application tier, and hold the application's business logic. A view is a Python function responsible for accepting incoming requests and sending out the appropriate response. Views typically make use of the model and template to produce their output.

Make sure that you don't confuse Django's model-template-view architecture with the **model view controller** (**MVC**) pattern commonly used in software development. The two are quite distinct, and describe the different tiers in the web application stack in very different ways. While the model in both Django and MVC represents the data tier, Django uses the view to hold the application logic, and separates out the presentation using templates. MVC, on the other hand, allows the view to directly specify the presentation of the data, and uses a controller to represent the application's business logic.

The differences between these two design patterns can be summarized as follows:



	Model-View-Controller Pattern	Model-Template-View Pattern
Presentation Tier	View	Template
Application Tier	Controller	View
Data Tier	Model	Model

There is a lot more to Django than can be covered in this brief introduction, but this is enough to understand how GeoDjango extends the Django framework.

GeoDjango

GeoDjango builds on Django's capabilities to add comprehensive support for building geospatial web applications. In particular, it extends the following parts of the Django system:

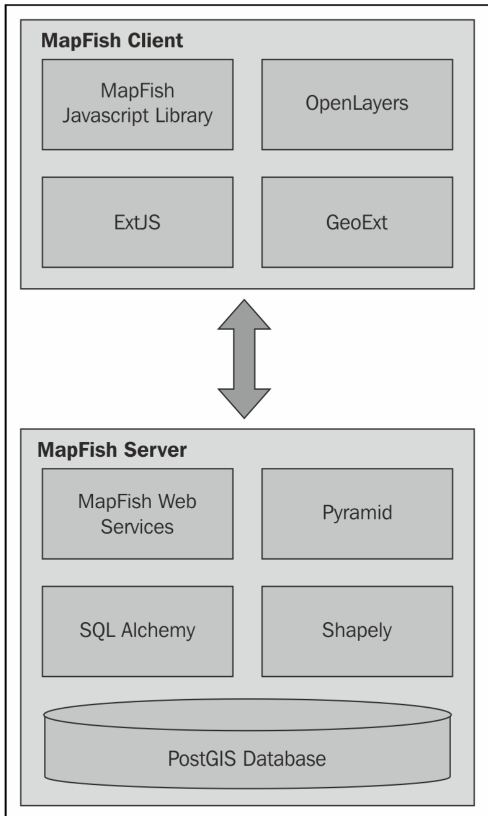
- The Model
    - The Django model is extended to support geospatial data types and spatial queries
    - As geospatial features are read from the database, the object-relational mapper automatically converts them into GEOS objects, providing methods for querying and manipulating these features in a sophisticated way similar to the interface provided by Shapely
    - The Model can import data from any OGR supported vector data source into the GeoDjango database
    - GeoDjango can use *introspection* to see which attributes are available in a given OGR data source, and automatically set up the model to store and import these attributes
- The Template
    - Django's HTML templating engine is extended to allow for the display of geospatial data using an embedded OpenLayers slippy map
- The Admin Interface
    - Django's admin interface is extended to allow the user to create and edit geospatial data using OpenLayers. The vector data is displayed on top of a base map using either OpenStreetMap data or a less detailed WMS source called *Vector Map Level 0*.

All told, the GeoDjango extension makes Django an excellent choice for developing geospatial web applications. We will be working with GeoDjango in much more detail in the remaining chapters of this book.

Mapfish

Mapfish (`http://mapfish.org`) is an extension to the Pyramid web application framework in much the same way as GeoDjango is an extension to Django. Pyramid is the successor to Pylons, a popular and flexible web framework. Pyramid brings together a number of third-party tools to implement a complete web development framework, supporting features such as a model view controller architecture, URL mapping, form handling, sessions, user accounts, and various options for deploying your web application, as well as internationalization, testing, logging, and debugging tools. Out of the box, Pyramid supports both a number of HTML templating libraries, as well as the SQLAlchemy and ZODB object-relational mappers.

Mapfish builds on Pyramid to create a complete geospatial web application framework. Mapfish itself is broken into two portions: a server portion and a client portion:



The MapFish server uses PostGIS, SQLAlchemy and Shapely to provide an object-oriented layer on top of your geospatial data as part of a Pyramid application. The server also implements a number of RESTful web services, using a custom protocol known as the MapFish Protocol. These allow the client software to view and make changes to the underlying geospatial data.

The Mapfish client software consists of a JavaScript library that provides the user interface for a Mapfish application. The Mapfish JavaScript library builds on OpenLayers to produce a slippy map, along with the ExtJS and GeoExt libraries to provide user-interface widgets and common behaviors such as feature selection and editing, searching, enabling and disabling layers, and so on.

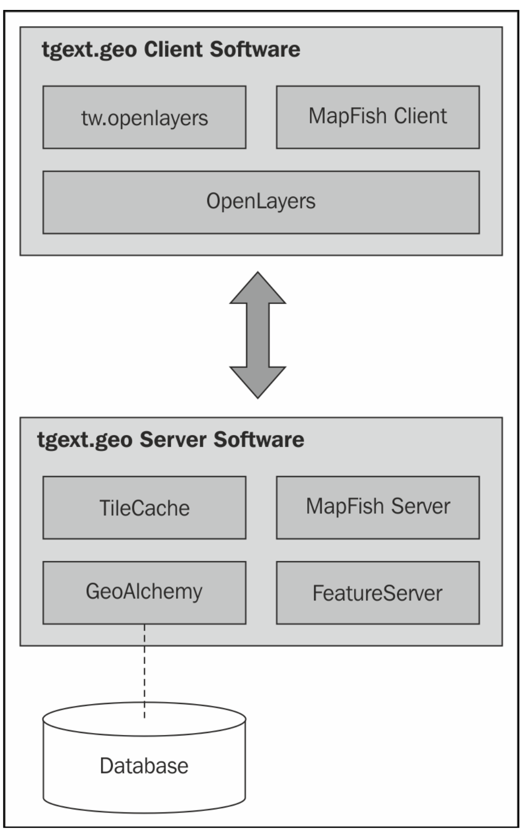
Putting all this together, Mapfish allows developers to build complex user interfaces for geospatial web applications, along with server side components to implement features such as geocoding, spatial analysis, and editing geospatial features.

TurboGears

Just as GeoDjango is an extension to the existing Django web framework and Mapfish is an extension to Pyramid, the TurboGears web framework (`http://turbogears.org`) also has an extension designed to make it easy to implement your own geospatial web applications. This extension is named **tgext.geo**, and comes bundled with TurboGears.

TurboGears is a sophisticated and extremely popular web application framework built upon Pylons and using a number of standard components including the SQLAlchemy object-relational mapper, the Genshi templating system, and the ToscaWidgets user interface library. TurboGears also uses the MVC architecture to separate the application's data, business logic, and presentation.

The tgext.geo extension to TurboGears makes use of several existing third-party libraries, rather than trying to implement its own functionality. As with Mapfish, tgext.geo consists of both client and server components:



tgext.geo uses the ToscaWidgets tw.openlayers wrapper around the OpenLayers library to make it easy to embed a slippy map into your TurboGears application. Alternatively, you can use OpenLayers directly, or use the MapFish client libraries if you prefer.

On the server side, tgext.geo consists of four parts:

- TileCache is used to serve externally-generated map tiles and display them as a background for your own map data, without wasting bandwidth or time regenerating the map tiles every time they are needed.
- GeoAlchemy (`http://geoalchemy.org`) provides an object-relational mapper for geospatial data stored in a PostGIS, MySQL, or SpatialLite database. As the name suggests, GeoAlchemy is built on top of SQLAlchemy.

- MapFish RESTful web services provide a simple way for the client code to read and update the geospatial data held in the database.
- FeatureServer (`http://featureserver.org`) provides a more feature-complete interface to the application's geospatial data. FeatureServer implements the WFS protocol, as well as providing a RESTful interface to geospatial data in a number of different formats including JSON, GML, GeoRSS, KML, and OSM.

The tgext.geo extension to TurboGears makes it possible to quickly build complete and complex geospatial applications on top of these components using the TurboGears framework.

Summary

In this chapter, we have surveyed the geospatial web development landscape, examining the major concepts behind geospatial web application development, some of the main open protocols used by geospatial web applications, and a number of Python-based tools for implementing geospatial applications that run over the internet.

We have seen that:

- A web application stack allows you to build complex but highly structured web applications using off-the-shelf components
- A web application framework supports rapid development of web-based applications, providing a "batteries included" or full stack development experience
- Web services make functionality available to other software components via an HTTP-based API
- A map renderer such as Mapnik can be used to build a web service that provides map-rendering services to other parts of a web application
- Tile caching dramatically improves the performance of a web application by holding previously-generated map tiles, and only generating new tiles as they are needed
- Web servers provide the interface between your web application and the outside world

- A user-interface library, often combined with AJAX technology, runs in the user's web browser and provides a sophisticated user interface not possible with traditional HTML web pages
- The slippy map, popularized by Google Maps, is the ubiquitous interface for viewing and manipulating geospatial data
- Complete geospatial web application stacks, developed using web application frameworks, can implement sophisticated geospatial features including data manipulation, searching, and analysis with far less development effort than would be required using a "roll your own" solution
- Geospatial web protocols such as WMS, WFS, and TMS allow different components to communicate in a standard way
- Tile caching can be implemented in Python using either TileStache or TileLite
- Existing general-purpose user interface libraries such as Twitter Bootstrap, script.aculo.us, JQuery UI, and YUI can all be used in geospatial applications to implement the non-spatial portions of the user interface
- OpenLayers and Leaflet are JavaScript libraries for implementing slippy maps, allowing the user to view and edit geospatial data
- GeoDjango, MapFish, and tgext.geo are extensions to existing web application frameworks, providing complete web application development environments for building complex geospatial web applications

In the next chapter, we will start to build a complete mapping application using PostGIS, Mapnik, and GeoDjango.