Kevin M. Eckhardt. CDLA Maps: A JavaScript API for Sharing Digital Maps. A Master's Paper for the M.S. in I.S degree. July 2009. 84 pages. Advisor: Hugh A. Cayless

The popularity of Internet mapping APIs, such as the Google Maps API, has led to the creation of a multitude of web applications using geo-spatial information of all types. The Carolina Digital Library and Archives at the University of North Carolina at Chapel Hill (CDLA) has created close to 300 custom map tile sets from historic maps depicting locations throughout the state of North Carolina. These tile sets were designed to be displayed as overlays using the Google Maps API. This paper describes the design and implementation of a JavaScript API which will enable the CDLA to easily share these custom tile sets and allow scholars, educators, students, and others to include the CDLA's maps in their own web pages. Usage examples, an API reference, and the complete JavaScript source code are included.

Headings:

Application programming interfaces

Geographic information systems

Maps / Internet resources

JavaScript (Computer language)

CDLA MAPS: A JAVASCRIPT API FOR SHARING DIGITAL MAPS.

by
Kevin M. Eckhardt

A Master's paper submitted to the faculty
of the School of Information and Library Science
of the University of North Carolina at Chapel Hill
in partial fulfillment of the requirements
for the degree of Master of Science in
Information Science.

Chapel Hill, North Carolina

July 2009

Approved by

_____

Hugh A. Cayless

Table of Contents

1 Introduction

Maps have proven to be useful tools for many purposes, from navigation to planning to historical research. It is no wonder that web-based maps have become both a common and popular feature of sites on the Internet. Restaurants, retail stores, and other businesses use maps to help customers get directions or find the nearest location. Websites such as Google Maps and MapQuest allow users to view road maps, examine satellite imagery, and get driving directions around the world. Libraries and museums have digitized historical maps and made them available to many more users than were able to view the original artifacts.

Technological advances have changed the ways that websites are displayed on the Internet as well as the ways in which users can interact with them. Websites have evolved from simple, statically created pages to dynamically generated pages with high levels of user interaction. Web-based maps have exhibited a similar evolution from small single images to interfaces which allowed users to zoom and pan by reloading the displayed map view to today's mapping applications which provide real-time interaction with the map.

A number of JavaScript application programming interfaces (APIs) have been developed which allow users to create "mash-ups" by combining their own geo-spatial information with pre-existing maps. These APIs define the objects and methods used to create and interact with a web-based map. They support the creation of points, lines, polygons, and other items which can be overlaid on top of a map at specific locations.

The overlays can be "clickable" and used to display additional information or link to other pages or websites. In addition to simple geometric overlays, many mapping-APIs also allow users to create and display entire custom maps.

JavaScript mapping-APIs are available from many commercial mapping service providers including Google (http://code.google.com/apis/maps/), Yahoo! (http://developer.yahoo.com/maps/), and Microsoft (http://www.microsoft.com/maps/developers/). Open source mapping-APIs are also available, the most prominent one being OpenLayers (http://openlayers.org).

2 Web Map Use at CDLA

The Carolina Digital Library and Archives has begun to use web maps in collections created by its digital publishing initiative "Documenting the American South" (http://docsouth.unc.edu). "Going to the Show" (http://docsouth.unc.edu/gtts) and "North Carolina Maps" (http://www.lib.unc.edu/dc/ncmaps/) are the first two DocSouth collections to include web maps. They both use the Google Maps API to display custom map layers created from historic maps.

2.1 "Going to the Show"

"Going to the Show" (GttS) is a nationally-recognized digital humanities project which documents the social experience of movie-going during the silent film era in North Carolina. The project allows users to learn about the movie-going experience and the role that race played in that experience during the early 20th century by exploring an extensive collection of newspaper articles and advertisements, photographs, picture postcards, city directories, and other artifacts. The project includes a database of movie

theater locations, racial policies, and other information for over 200 communities in the state of North Carolina and a scholarly case study of early movie-going for the city of Wilmington.

One of the focal points of GttS is its use of Sanborn® Fire Insurance Maps. These small-scale (typically 50 foot to the inch) and highly detailed maps were created to aid in determining fire insurance premiums, but are now an excellent resource for visualizing and researching the historic development of cities throughout the United States. Sanborn® maps provide details about the construction materials of buildings along with the types and/or names of the businesses and organizations that used them. Maps were created and sold for individual cities. Each cities map set consisted of one or more individual map sheets. Small towns may have only a single sheet, while large cities could require over 50. New map sets were created for cities at approximately five year intervals.

For GttS the individual maps sheets were digitally "stitched" together to create a single map for the central business district and other select areas of over 40 North Carolina communities. These maps were then geo-referenced to match the areas shown by the maps to the geographic coordinates of the actual physical locations they depict. The maps are presented to site visitors using an interface based on the Google Maps API. This interface allows users to interactively explore the maps and see them displayed as an overlay on top of current road maps and aerial imagery.

GttS uses these maps to identify the locations of movie theaters and other businesses. This allows users not only to read about but also to visualize the spatial relationships between theater locations and the experience of going to the show. By

viewing Sanborn® maps from multiple years along with the current maps users can gain a sense of how a city has developed.

2.2 "North Carolina Maps"

"North Carolina Maps" (NCMaps) is a comprehensive collection of historic maps depicting the state of North Carolina. It is a collaborative effort between the North Carolina State Archives, the North Carolina Collection at UNC-Chapel Hill, and the Outer Banks History Center. Maps selected for this collection include "original, printed maps of North Carolina published prior to 1923; manuscript maps; maps from books and atlases published prior to 1923; and selected maps published by the state government after 1923" and "some maps showing larger regions, including many maps showing North Carolina with neighboring states, several maps depicting Southeastern states, and selected maps showing areas as large as all of North America, when these include particularly notable depictions of North Carolina" ("About North Carolina Maps", n.d.). The project, which is currently in its third and final year, will ultimately include over 2,000 digitized maps. A selection of maps from the collection have been geo-referenced and made available for display using the Google Maps API. Over 100 maps have been geo-referenced so far with plans to include at least one map for every county in North Carolina.

2.3 Custom Tile Sets

The geo-referenced maps from GttS and NCMaps are displayed inside Google Maps as custom Tile Layer Overlays. A Tile Layer Overlay consists of a set of images, or tiles, which are displayed on top of the map imagery provided by Google. These tiles

break the map image up into much smaller pieces which can be downloaded as needed to display sections of the map. The tile sets created for Tile Layer Overlays follow the structure used by Google to create the tiles for its own maps. The dimensions for each tile are 256 pixels wide by 256 pixels high. A different set of tiles is needed for each zoom level at which the map will be displayed. At the minimum zoom level, level 0, the entire world fits into a single tile. For each successive zoom level the region of the map represented by a single tile is split across four tiles: at zoom level 1 four tiles are used, positioned in a 2 x 2 grid; at zoom level 2 sixteen tiles are used, positioned in a 4 x 4 grid; and so on.

A custom tile set was created for each geo-referenced map using an Adobe Photoshop script. The script resizes the image as needed to match the dimensions of the region in the tile grid where it will be displayed and then creates the individual tile images. This is done for multiple zoom levels, starting with the zoom level that most closely matches the image's original dimensions and working down through each zoom level until the entire map can be displayed using between 1 and 4 tiles. Each tile is named using the x, y, and z coordinates which identify its zoom level and position in the tile grid.

3 The Benefits of an API

The creation of an API to manage the CDLA's map resources provides two major benefits: reduced maintenance costs and easier reuse of the map resources.

3.1 Reduced Maintenance Costs

The current implementations of the “Going to the Show” and “North Carolina Maps” websites require direct knowledge of the file locations and filename structure of the map layer tile images along with other resource specific metadata for each map resource that they use. Any change to a map resource’s tile set would also require changes to any site that utilizes that resource. Relocating the storage of map resource metadata from the individual applications to the API allows changes to be made to the tile sets without the need to modify the applications.

In addition to managing map resource metadata, the CDL Maps API performs the creation of the map layer objects which are used by the mapping-API to display the resource as a custom tile set overlay. Moving creation of the map layer object from the website’s JavaScript code to the CDLA Maps API simplifies the website’s code. The website needs only to request the desired map layer object from the API and add it to the web map. Changes to the map layer object creation process which do not impact other aspects of the mapping-API can be implemented within the CDLA Maps API without the need to modify the web applications.

3.2 Resource Reuse

The CDLA has currently created custom tile sets for almost 300 different maps - a labor intensive undertaking requiring many 100s of hours. The GttS and NCMaps projects have demonstrated the usefulness of these tile sets and spawned ideas for new projects using the maps as a resource. The API will enable the CDLA to easily reuse the map resources created for GttS and NCMaps in other collections. It also opens the door

to making the resources available to scholars, content creators, and other interested parties external to the organization.

4 Design

The CDLA Maps API acts as a layer of abstraction between the map resources and the web applications that use them. Functional needs for the CDLA Maps API fall into two high-level categories: administration and access. Administration deals with the creating and maintaining the metadata for the available map resources. Access deals with locating map resources and using them within a web map application.

4.1 Administration

Administrative tasks for the CDLA Maps API center around the creation and maintenance of the metadata associated with the individual map resources. This metadata is stored in a database. A secure web-based interface is used to assist with administrative tasks.

4.1.1 Metadata Scheme. In order to properly create the mapping-API layer objects used to display a map resource the API needs to keep track of the following information: the location and naming structure of the individual image files which make up the tile set, the minimum and maximum zoom levels for which tiles are available, and the geographic bounds of the map as displayed by the mapping API. To aid with searching the maps and identifying the content of an individual map resource the following data is also collected: the name of the map, the date when the map was published, a text description of map's contents, and the spatial extent of the area depicted by the map. Additional metadata (such as publisher, scale, projection, and physical size)

is available for the original map from which a map resource was created. Instead of replicating this metadata, which has no practical application in the API, a URL for the source map within the library's ContentDM repository is included. A full description of the metadata scheme is shown in Table 4-1.

Table 4-1. Map Resource Metadata Scheme

Field Name	Description
Map ID	Unique identifier for the map resource.
Title	Title of the map used to create the resource.
Description	Text description of the locations and information depicted by the map.
Date	Publication date of the source map.
Date Precision	The precision of the publication date. Permitted values are: day, month, year, decade, and century.
Spatial Extent	A description of the geographic area depicted by the map.
Display Extent	A description of the entire area "covered" by the map tiles when displayed within a web map. This includes the map legend, borders, etc. and not just the location depicted by the map.
Minimum Zoom	The minimum zoom level for the map based the zoom levels used by the Google Maps API.
Maximum Zoom	The maximum zoom level for the map based on the zoom levels used by the Google Maps API.
Tile Set URL	An URL pattern defining the location and naming structure of the map tiles. The placeholders {X}, {Y}, and {Z} should be used for the location of the tile's x, y, and z coordinates. For example: http://lib.unc.edu/maps/mapname/{Z}_{X}_{Y}.png
Source URL	Absolute URL for source map in the ContentDM repository.

4.1.2 Map IDs. Each map resource managed by the API must have a unique identifier. This identifier is referred to as the resource's "Map ID." The format of the Map ID is designed to be descriptive of the map resource that it represents. The Map ID consists of two parts: a location description and a content description. The location component is intended to provide a hierarchical description of the geographic area depicted by the map. The content component provides a brief description of the information presented by the map. A colon (:) is used as a separator between terms in each of the components. A double colon (::) is used to join the two components. A formal specification of the Map ID format is shown in Figure 4-1. Example Map IDs are shown in Table 4-2.

Certain individual map resources, such as the multiple Sanborn® Fire Insurance Maps available for a single city, are often utilized together. The concept of a Map Set was created to act as a shortcut for requesting these sets of related maps. Like individual resources, map sets also have a unique identifier. The API recognizes these special identifiers and properly associates them with the individual map resources contained within the set. Map Set IDs share the same namespace and format as Map IDs. By convention, a Map Set ID ends with the keyword "SET," though this is not a requirement.

Figure 4-1. Map ID Format Specification

```
                mapID ::= location-description "::" content-description
location-description ::= location-name [(":" sublocation-name)+]
        location-name ::= <term describing a geographic location>
     sublocation-name ::= <location-name located within the bounds of
                          the previous location-name or sublocation-
                          name>
 content-description ::= term [(":" term)+]
                term ::= (uppercase-letter | digit)+
    uppercase-letter ::= "A"..."Z"
               digit ::= "0"..."9"
```

Note: Modified BNF notation as defined in the Python user documentation.
http://docs.python.org/reference/introduction.html#notation

Table 4-2. Example Map IDs

Map ID **Map Name**
NC:BUNCOMBE::SOILSURVEY:1920 1920 Soil Survey map of Buncombe County, NC
NC:BUNCOMBE:ASHEVILLE::SANBORN:1917 1917 Sanborn® Fire Insurance Map for Asheville, NC
NC:BUNCOMBE:ASHEVILLE::SANBORN:SET Map set including the 1896, 1901, 1907, 1913, and 1917 Sanborn® Fire Insurance Maps for Asheville, NC

4.2 Access

Two key pieces are needed to provide web applications access to the map resources: the client-side JavaScript which is executed by web applications inside the user's browser and the server-side web service which provides access to the map resource metadata.

4.2.1 Client. The client-side component of the CDLA Maps API is a small JavaScript library. This library is used to request a map resource's metadata and create the map layer object used to display it. A web application that wants to display a map resource should include this library in its HTML source along with the libraries needed by the mapping-API that will be used to create the map.

Map resources are accessed by creating one of two objects provided by the library. The `cdla.maps.Map` object provides access to a single resource identified by its Map ID. The object contains methods for accessing the map resource's metadata and a method for creating a new instance of the map layer object used to display the resource on a web map.

The `cdla.maps.MapSet` object provides access to multiple map resources identified by a list of Map IDs and/or Map Set IDs. When using multiple map resources in a single web map it is often necessary to determine the minimum and maximum zoom levels and overall geographic bounds for the set of maps as a whole. The `cdla.maps.MapSet` object contains methods for accessing this aggregate information as well as accessing the `cdla.maps.Map` object for each of the map resources in the set.

Full specifications of the library methods and resource objects are provided in Appendix A.

4.2.2 Server. The server-side component of the CDLA Maps API is responsible for responding to requests for map resource metadata. Web applications using the API make requests for map resource metadata using `<script>` tags in the `<head>` section of their HTML documents. Requests are made to a specific URL on the API server. The URL includes a Map ID or Map Set ID to identify which map resource is being requested. The server responds with a JavaScript file that loads the resource metadata into the API's library. It should be noted that the API server does not directly provide the map layer tile image files, but instead provides a resource locator for them.

4.2.3 Access Control. The CDLA Maps API is intended to be freely available. Access control was not implemented to tightly restrict access but to provide a mechanism for monitoring usage of the service. URL-based API keys were selected as the access control method. This method requires a web application to provide a valid API key when requesting data from the service. API keys are matched to a specific domain name or URL. Access is not granted if the API key does not match with the URL of the web application making the request. This method relies on the referrer information included in the HTTP request and as a result it is not foolproof (Murray and Ort, 2007). Since the purpose of access control is monitoring and not securing the service this limitation is acceptable.

5 Implementation

Implementation of the API can be broken down into two major pieces: the server-side application and the client-side JavaScript library.

5.1 Server Application

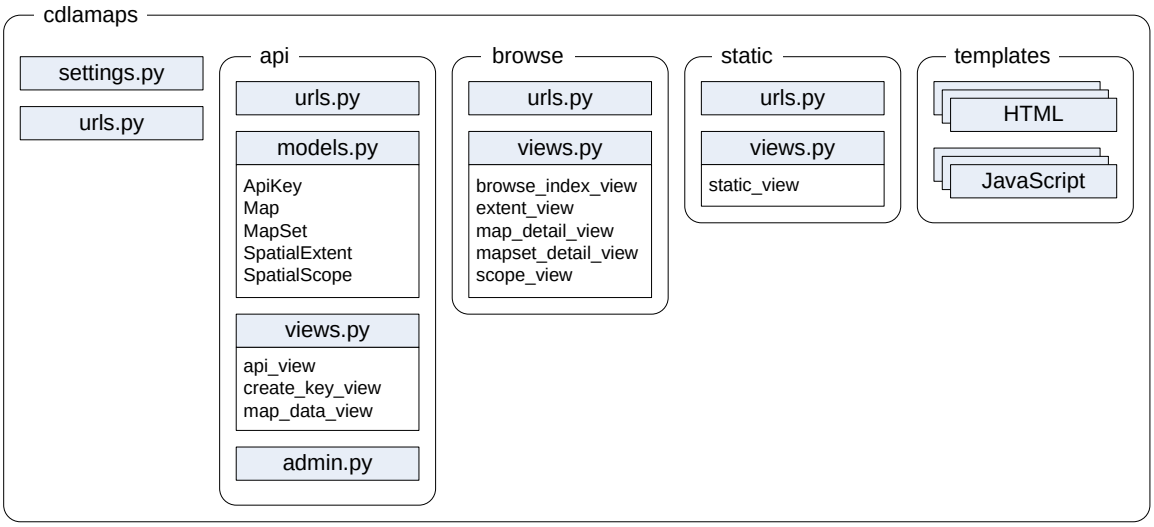
The server application is the most complex piece of the CDLA Maps API. It is responsible for: creating and maintaining map resources and their metadata, creating and verifying API keys, responding to requests for map resource metadata, and providing user documentation.

The server application implementation uses the Django web framework (http://www.djangoproject.org). Django includes a built-in web-based administration package for managing data and provides support for geo-spatial data types, operations, and queries through use of the GeoDjango package (http://geodjango.org). PostgreSQL (http://www.postgresql.org) serves as the backing database. The PostgreSQL extension PostGIS (http://postgis.refractions.net) was used to add geo-spatial support to the database. These software packages were chosen for implementing the API due to their feature sets, prior experience using them, and available software support at the CDLA.

Django-based applications loosely follow the Model-View-Controller architectural pattern (Buschmann et al., 1996). Django uses Models to describe an application's data. Models are defined as Python classes and a model typically maps to a single table in the underlying database. Views process or retrieve data and prepare it for display or insertion into the database. Views are defined as Python methods. Display of information is accomplished through the use of a template system. The control aspect of the Django framework is performed by using regular expressions to match URLs with their respective view methods (Django Software Foundation, 2005-2009).

The CDLA Maps API consists of a project, *cdlamaps*, with three applications: *api*, *browse*, and *static*. This framework can be seen in Figure 5-1.

Figure 5-1. CDLA Maps Django Framework



The *api* application contains the data models for the map resources and views for handling API key creation and requests for map resource metadata. The *browse* application implements various views for exploring the available map resources. The *static* application is used to generate static pages such as the user documentation.

5.1.1 Map Resource Management. Four models are used to represent the map resources and their metadata: `Map`, `MapSet`, `SpatialExtent`, and `SpatialScope`. The `Map` model implements the map resource metadata scheme. The `MapSet` model defines Map Set IDs and the individual map resources that are part of that set. The `SpatialExtent` and `SpatialScope` models are used to represent the Spatial Extent field in the map resource metadata scheme.

The metadata scheme utilizes two geo-spatial data fields: Spatial Extent and Display Extent. Display Extent describes the bounding region of the map resource's tile

images. This region is rectangular in shape and therefore only four points need to be measured in order to define the polygon for the region. Spatial Extent describes the geographic region depicted by the map resource. These regions can often be irregular in shape and require a large number of points to define the polygon or multi-polygon that represents the region. The `SpatialExtent` model was created to allow administrators to select from a list of pre-defined geographic regions to use as the spatial extent value for a map resource. Public domain data sets obtained from NC OneMap (http://www.nconemap.com) were used to define spatial extent regions for the state of North Carolina, its three geographic regions, 100 counties, and over 500 towns and cities. These pre-defined spatial extents are based on modern boundaries and using them reduces the accuracy of the spatial extent field for a map resource based on a historical map. More accurate values could be achieved by measuring the actual spatial extent for each map resource. However, it is unlikely that the increased labor cost to do so would prove to be beneficial for this application.

The `SpatialScope` model provides an abstract representation of the relative size a spatial extent. The initial scopes defined for the API are ‘City’, ‘County’, ‘Region’, and ‘State’. Each scope is also assigned a numeric value to allow for a hierarchical comparison of scope definitions. Every spatial extent is assigned a spatial scope. This provides a mechanism for easily identifying maps that depict similar types of geographic regions.

Full definitions of these models can be found in Appendix C.

5.1.2 API Keys. API keys are freely available, but must be requested by users via an HTML form. To obtain an API key the user must provide a URL for the website or page where the API will be used along with a contact name and email address. Upon successful submission of the form an API key will be generated and displayed to the user. The generated API key can be used to validate a request for map resource data that originates from any webpage whose URL starts with the URL string that was used to create the key. If the provided URL was a domain name, then any page hosted by that domain can use the API key. If any path information was included in the URL, then the requesting page must reside in that directory or one of its sub-directories. If the domain name of the provided URL begins with the prefix 'www.' the API key will validate URLs both with and without the prefix. This is provided as a convenience since many servers are configured to respond to requests both with and without the 'www.' prefix in the domain name.

API keys are stored using the `ApiKey` model. The model includes fields for the key value itself, the URL associated with the key, the name and email address for the person who requested the key, a regular expression pattern used when verifying the key, and a counter for the number of times that data has been successfully requested using the key.

The API key is generated by calculating the SHA-1 hash (NIST, 2002) of a string consisting of the provided URL combined with a pre-defined "salt" string and a string representation of the current millisecond resolution date and time. This SHA-1 hash value is converted into its 40-character hexadecimal representation and used as the value of the key.

The API key must be included as a part of the URL for requests which require validation. When validating a request the server looks up the regular expression pattern associated with the provided key. This pattern is then matched against the URL string contained in the HTTP Referrer header of the HTTP request. If the URL matches the pattern the API key is considered valid. If the HTTP referrer is not provided or its value is empty the verification process will consider the API key to be valid by default. This is considered acceptable behavior since the API keys are used to track service usage and not to prevent access to the resources.

Full definitions of the `ApiKey` model and its associated methods can be found in Appendix C.

5.1.3 Resource Requests. The server must handle requests from client applications for both the JavaScript library and map resource metadata. Both request types are implemented by view methods within the Django framework.

The `api_view` method is used to respond to requests for the JavaScript library. This method checks that the API key contained in the URL for the request is valid. If the key is valid the method returns the text file containing the JavaScript library to the client. If the key is invalid the method returns JavaScript code which causes the browser to display an alert box informing the client that the provided API key was not valid for the requesting page.

The `mapdata_view` method is used to respond to requests for map resource metadata. URLs for this type of request include an API key and either a Map ID or Map Set ID. This method first checks to see if the API key is valid. If the key is invalid the method returns the same JavaScript alert code that is returned by the `api_view` for

invalid keys. If the key is valid then the method retrieves the metadata for the map resource identified by the requested ID from the database. The metadata is placed inside of a JSON object (http://www.json.org). If the requested ID is a Map Set ID then the JSON object will contain the metadata for each of the individual map resources that make up the map set. If no resource was found using the requested ID then the JSON object will be empty. This JSON object is returned to the client inside of a JavaScript file which loads the data into predetermined variables which can be accessed by the API's JavaScript library.

Full definitions of the `api_view` and `mapdata_view` methods can be found in Appendix C.

5.1.4 User Documentation. The server application also provides access to user documentation and an interface for browsing the available map resources. The user documentation is served as static HTML files. The map browsing interface is implemented using several view methods.

The index page for the browsing interface provides links for viewing available map resources for the state of North Carolina by geographic region, county, or city. Following one of these links loads a page created by the `scope_view` method. This method generates an alphabetical list of the spatial extents that have the requested spatial scope (city, county, or region) and a count of the available map resources contained within each spatial extent.

Selecting one of the spatial extents loads a page created by the `extent_view` method. This method generates a list of available map resources that are located within the selected spatial extent. Only map resources that have a scope less than or equal to the

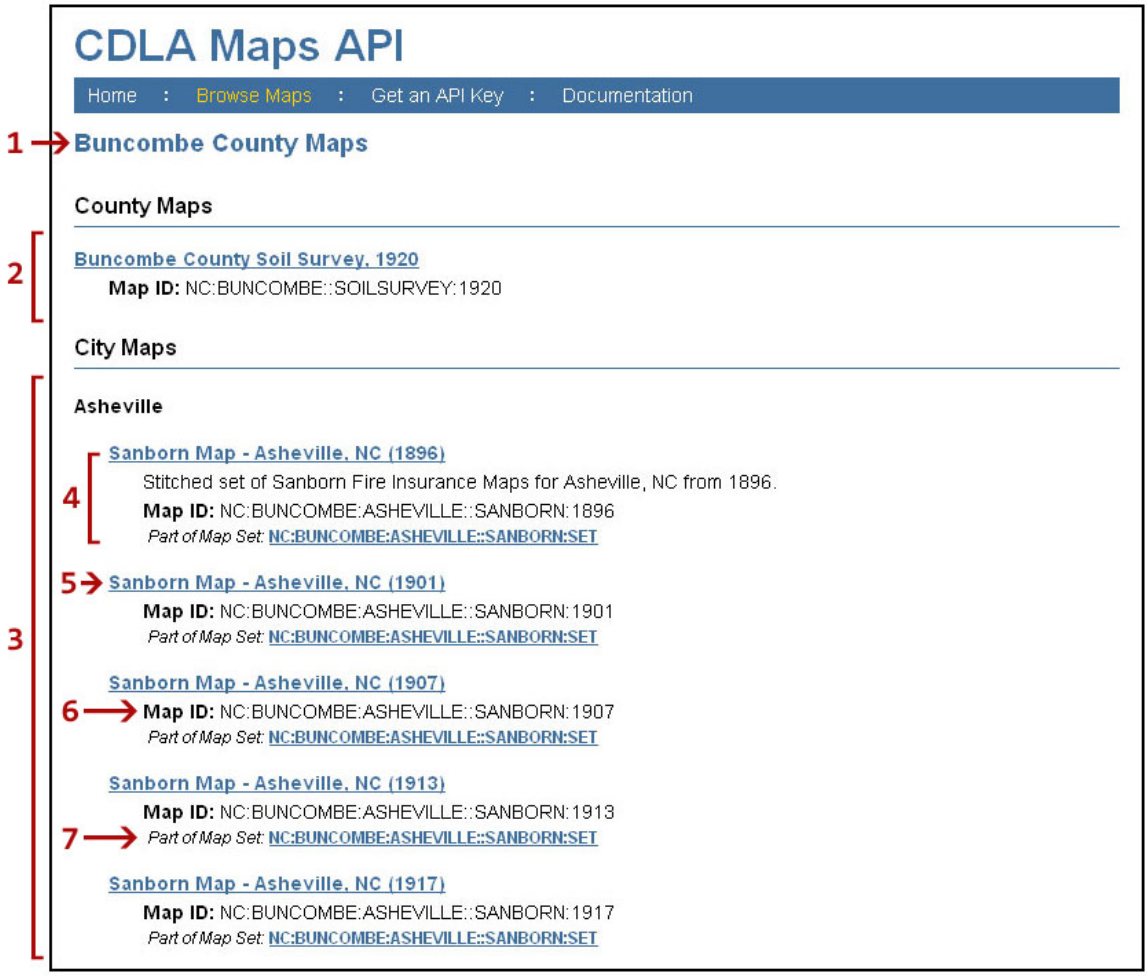
scope of the selected spatial extent will be included (i.e. if a county is selected then the list will include county and city maps, but not region maps). A sample extent view is shown in Figure 5-1.

Selecting one of the map resources loads a page created by the `map_detail_view` method. This method displays the selected map resource using the Google Maps API along with a list of the maps descriptive metadata.

If a map resource is part of a map set, this will be noted on both the `extent_view` and `map_detail_view` pages by providing a link to details about the map set. This page is generated by the `mapset_detail_view` method. The page displays the map set using the Google Maps API and provides information about each map resource in the set. A sample map set view is shown in Figure 5-2.

Full definitions of the browsing interface methods can be found in Appendix C.

Figure 5-1. Sample Extent View - Annotated



1. Name of the selected spatial extent
2. Maps having the selected spatial extent
3. Maps with a smaller spatial scope located within the spatial extent
4. A map resource
5. Link to more details about the map (`map_detail_view`)
6. The Map ID used to request the map using the JavaScript library
7. Indicates the map is part of a Map Set. Links to information about the Map Set (`mapset_detail_view`).

Figure 5-2. Sample Map Set View - Annotated

1. Name of the map set
2. Maps in the set displayed using the Google Maps API
3. Map Set metadata. Includes the Map Set ID which can be used to request the set using the JavaScript library
4. List of maps in the map set, organized by spatial scope and spatial extent
5. Checkbox can be used to turn display of the map resource's layer on or off

5.2 JavaScript Library

The JavaScript library is the client-side piece of the CDLA Maps API. It provides users with the methods needed to access map resources and create map layer objects. The complete code for the JavaScript library can be found in Appendix B.

5.2.1 Loading Metadata. The library does not initially include the metadata for the map resources. Metadata is requested for specific map resources and inserted into the library when the web page loads. A number of AJAX techniques are available for dynamically loading data. The typical method is to use the JavaScript `XMLHttpRequest` object to request the data from the server, however this method is restricted by JavaScript's same-origin policy. This policy only allows requests to be made using the same protocol, domain name, and port as the web page making the request (Zalewski, 2008). This will not work for the CDLA Maps API since it is intended to be used on third-party websites.

One method to work around the same-origin policy uses dynamically generated `<script>` tags, which are exempt from the same-origin policy, to load data from third-party servers. The data is sent in the form of JavaScript code. A common implementation of this method encodes the data as a JSON object (http://www.json.org) which is used as the argument to a callback function. The code requested by these dynamically generated script tags is not loaded synchronously. As a result, callback functions must be used to process the data after it has finished loading. Callback functions are common in AJAX applications (see http://doc.jquery.com/Ajax for examples), however they do add complexity to code that uses the API and as a result may serve as a barrier to entry for users who are not experienced programmers.

The CDLA Maps library uses a manual variation of the script tags method: when using the library a user includes a script tag to load the library along with one or more additional script tags to load the metadata for the map resources they intend to use. The JavaScript executed by these additional script tags loads the map resource metadata, contained within a JSON object, into variables which are accessible by the methods in the library. An example data loading script can be found in Appendix B.

5.2.2 Code Structure. The JavaScript library is intended to be used by developers of web sites both internal and external to the CDLA. In order to do this safely the library needs to ensure that other JavaScript code is not likely to interfere with its own operation and that it is not likely to interfere with other JavaScript code running on the page.

Placing the library inside its own namespace helps to protect the publicly accessible variables and methods by reducing the likelihood that the user or another library will accidentally overwrite them. JavaScript does not support true namespaces, but they can be emulated be defining variables and methods to be properties of a parent object assigned to a variable with the same name as the desired namespace (Resig, 2007, p. 49). `cdla.maps` was selected as the namespace for the API library. This establishes the first level `cdla` namespace which can be used to package the Maps API together with other potential future libraries.

The library uses closures to define private methods and variables that are not directly accessible outside of the library. Richard Cornford (2004) provides the following explanation of closures:

> The simple explanation of a Closure is that ECMAScript allows inner functions; function definitions and function expressions that are inside the function bodes of other functions. And that those inner functions are allowed access to all of the local variables, parameters and declared inner functions within their outer function(s). A closure is formed when one of those inner functions is made accessible outside of the function in which it was contained, so that it may be executed after the outer function has returned. At which point it still has access to the local variables, parameters and inner function declarations of its outer function. Those local variables, parameter and function declarations (initially) have the values that they had when the outer function returned and may be interacted with by the inner function. (para. 2)

This helps protect the library from external code and also prevents the library's private variables and methods from accidentally overwriting variables or methods that have been defined outside of the library.

A code quality tool can help to identify potential issues in code which, while syntactically correct, may be poorly structured. The code quality tool JSLint (http://www.jslint.com) was used to examine the library's code. JSLint found no errors in the final library code when run with the following options:

- `bitwise: true` – do not allow bitwise operators: `<<  >>  >>>  ~  &  |`
- `eqeqeq: true` – requires use of the type checking equality operators: `===  !==`
- `immed: true` – immediate function invocations must be wrapped in parentheses
- `newcap: true` – constructor functions must start with an intial capital letter
- `onevar: true` – allows only a single `var` statement in each function
- `regexp: true` – does not allow `.` in regular expression literals
- `undef: true` – requires variables to be declared before they are used
- `white: true` – requires strict whitespace rules
- `indent: 4` – 4 spaces should be used for indentation

The options used consist of the recommended options with the exception of `plusplus` and `nomen`. The `plusplus` option disallows the use of the `++` and `--`

operators because they could potentially cause confusion when reading the code. This option was disabled because the `++` operator is only used in `for` loop declarations where its meaning is clear. The `nomen` option does not allow leading underscores to be used in variable names. This option was disabled because the underscore was purposefully chosen to identify variables inside the `cdla.maps` namespace which are publicly accessible but are considered to be "private" to the library.

5.2.3 Error Messages. The library performs type checking on method arguments where possible. When invalid argument types are discovered an error message is thrown. An error message is also thrown if the user requests an invalid Map ID, Map Set ID, or API Type. These error messages are intended to be descriptive and helpful to the user in tracking down the source of the error.

5.2.4 Mapping-API Support. The CDLA's map resources were created with tile sets that conform to the structure used by the Google Maps API. Tiles created using this structure can be used with several other commercial mapping-APIs and the open source API OpenLayers. The JavaScript library was designed to support creating map layer objects for use with multiple mapping-APIs. The `cdla.maps.defaultAPI()` method can be used to specify which mapping API to use when creating the map layer objects. The mapping API can also be specified as an optional argument when requesting a map layer object via the `layer()` method of the `cdla.maps.Map` object. The library currently supports both the Google Maps and OpenLayers APIs. If no default API is configured Google Maps will be used as the default API.

6 Usage Examples

The following examples illustrate the use of the CDLA Maps API to display a historical map resource as a layer using a mapping API. Unless indicated all examples will use Google Maps as the mapping API. The first example demonstrates the basic use case of displaying a single map resource. The second example displays the same resource using the OpenLayers mapping API. The third and fourth examples demonstrate use of the API for displaying more than one resource.

6.1 Single Map Resource

The simplest use case of the CDLA Maps API is to display a single map resource as an overlay. The basic steps to do this are:

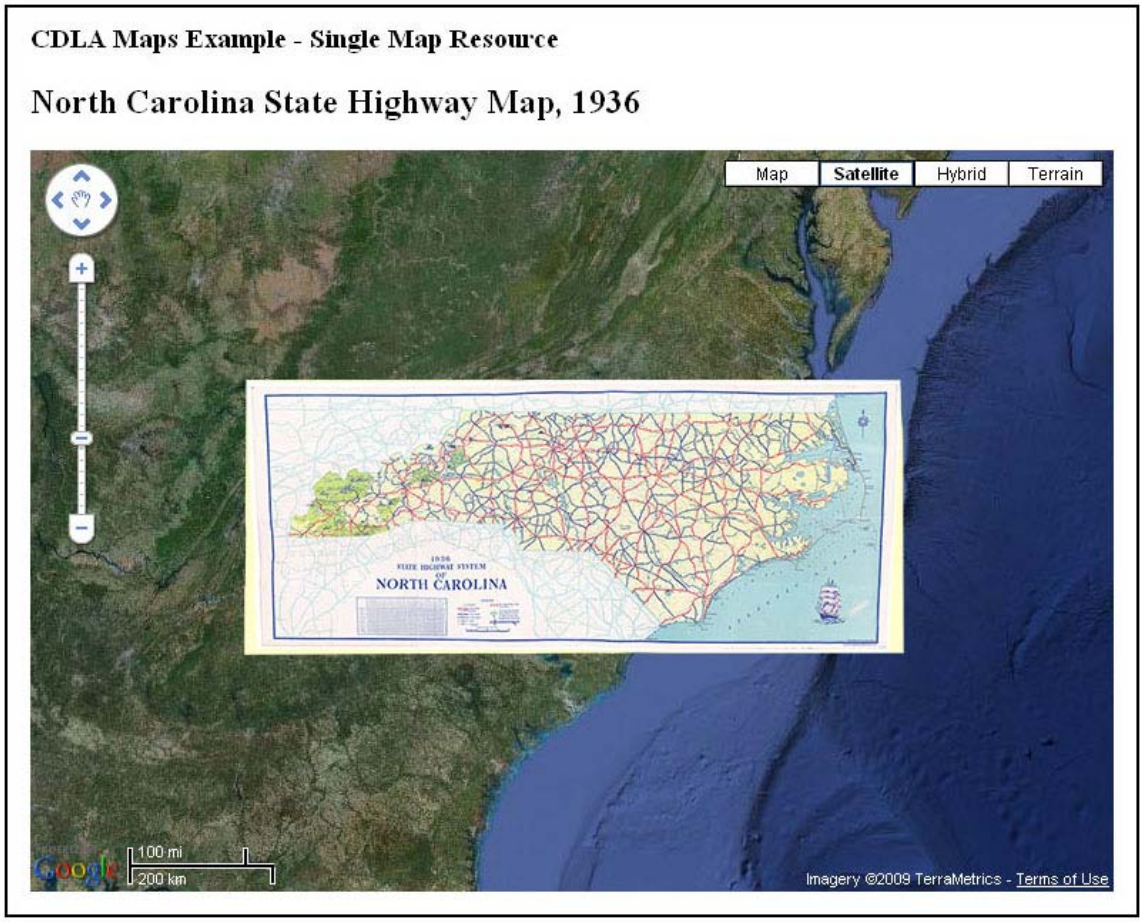
1. Create the Google Map object
2. Create the cdla.maps.Map object for the resource you wish to display.
3. Add the resource's layer object to the Google Map.

An example HTML document demonstrating this process can be seen in Figure 6-1. The document displays a map of North Carolina highways from 1936. The output of the document is shown in Figure 6-2.

Figure 6-1. HTML document for Single Map Resource

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8"/>
    <title>CDLA Maps API Example - Single Map Resource</title>
    <!-- load the Google Maps API -->
    <script type="text/javascript" src="http://maps.google.com/maps?file=api&amp;v=2
&amp;key=googlemapskey&amp;sensor=false"></script>
    <!-- load the CDLA Maps API -->
    <script type="text/javascript" src="http://server.unc.edu/cdlamaps/api/API_KEY">
    </script>
    <!-- load the map resource -->
    <script type="text/javascript"
         src="http://server.unc.edu/cdlamaps/api/mapdata/API_KEY/NC::HIGHWAYS:1936">
    </script>
    <script type="text/javascript">
    function init_page() {
      if (GBrowserIsCompatible()) {
        // create the Google Map object
        var map_options = {size: new GSize(760,520)};
        map = new GMap2(document.getElementById("mapcanvas"),map_options);

        // create the cdla.maps.Map object
        hwymap = new cdla.maps.Map("NC::HIGHWAYS:1936");

        // determine the maximum zoom level at which
        // the entire map layer is viewable
        zoom = map.getBoundsZoomLevel(hwymap.bounds());

        // make sure this zoom level is not less than
        // the minimum zoom level for the map layer
        if (zoom < hwymap.minZoom()) {
          zoom = hwmymap.minZoom();
        }

        // initialize the Google Map
        // - center the display on the map layer's location
        //   at our calculated zoom level
        // - use Google's satellite map for the base map
        // - configure the default user interface
        map.setCenter(hwymap.centroid(),zoom,G_SATELLITE_MAP);
        map.setUIToDefault();

        // add the map layer
        map.addOverlay(hwymap.layer());
      }
    }
    </script>
  </head>
  <body onload="init_page()" onunload="GUnload()">
    <h3>CDLA Maps Example - Single Map Resource</h3>
    <h2>North Carolina State Highway Map, 1936</h2>
    <div id="mapcanvas"></div>
  </body>
</html>
```

Figure 6-2. Single Map Resource



Looking more closely at the JavaScript contained in the HTML document shows some additional steps taken to determine where to set the initial viewpoint for the map. The `bounds()` method of the `cdla.maps.Map` object returns a `GLatLngBounds` object which identifies the geographic region containing tiles for the map resource. This object is passed to the Google map object's `getBoundsZoomLevel()` method to determine the maximum zoom level that will fully display the entire region. This is checked against the minimum zoom level of the map resource to ensure that tiles will be displayed for the resource. The `centroid()` method of the `cdla.maps.Map` object returns a `GLatLng`

object which identifies the center of the resource's geographic region. This object, along with the previously calculated zoom level, is passed to the Google map object's `setCenter()` method to change the map display to a centered view of the highway map resource.

6.2 Single Map Resource – Open Layers

This example replicates the use case in the previous example but uses the OpenLayers mapping API. The Google Maps API is also used in this example to provide the base layer for the map. When using the OpenLayers API with custom tile sets based on the tile structure used by the Google Maps API, such as those created by the CDLA, the map must be configured to use the Spherical Mercator projection. The additional configuration options can be seen in the HTML document shown in Figure 6-3. For more information on using the Spherical Mercator projection with OpenLayers see http://docs.openlayers.org/library/spherical_mercator.html.

Figure 6-3. HTML document for Single Map Resource - OpenLayers

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8"/>
    <title>CDLA Maps API Example - Single Map Resource - OpenLayers</title>
    <!-- load the OpenLayers API -->
    <script type="text/javascript" src="http://openlayers.org/api/OpenLayers.js">
    </script>
    <!-- load the Google Maps API to use as a base layer -->
    <script type="text/javascript" src="http://maps.google.com/maps?file=api&amp;v=2
&amp;key=googlemapskey&amp;sensor=false">
    </script>
    <!-- load the CDLA Maps API -->
    <script type="text/javascript" src="http://server.unc.edu/cdlamaps/api/API_KEY">
    </script>
    <!-- load the map resource -->
    <script type="text/javascript"
src="http://server.unc.edu/cdlamaps/api/mapdata/API_KEY/NC::HIGHWAYS:1936">
    </script>
```

```
    <script type="text/javascript">
    function init_page() {
      // create the OpenLayers map object
      // configure it to use the Spherical Mercator projection
      var map_options = {
        maxExtent: new OpenLayers.Bounds(-20037508.34,-20037508.34,
                                         20037508.34,20037508.34),
        numZoomLevels:20,
        maxResolution:156543.0339,
        units:'m',
        projection: new OpenLayers.Projection("EPSG:900913"),
        displayProjection: new OpenLayers.Projection("EPSG:4326")
      };
      var map = new OpenLayers.Map('mapcanvas', map_options);

      // create a base layer for the map
      // this example uses Google Maps Satellite map
      var gsat = new OpenLayers.Layer.Google(
        "Google Satellite",
        {type: G_SATELLITE_MAP, sphericalMercator: true}
      );
      map.addLayer(gsat);

      // set the default API type to OpenLayers
      cdla.maps.defaultAPI(cdla.maps.API_OPENLAYERS);
      // create the cdla.maps.Map object
      var hwymap = new cdla.maps.Map("NC::HIGHWAYS:1936");

      // add the map layer
      map.addLayer(hwymap.layer());

      // center the map on the entire map layer
      map.zoomToExtent(hwymap.bounds());

      // make sure tiles are visible for the layer
      if (map.getZoom() < hwymap.minZoom()) {
        map.zoomTo(hwymap.minZoom());
      }
    }
    </script>
  </head>
  <body onload="init_page()" onunload="GUnload()">
    <h3>CDLA Maps Example - Single Map Resource - OpenLayers</h3>
    <h2>North Carolina State Highway Map, 1936</h2>
    <div id="mapcanvas" style="height: 520px; width: 760px;"></div>
  </body>
</html>
```

6.3 Multiple Map Resources

This example demonstrates the use of a `cdla.maps.MapSet` object to display multiple map resources at the same time. Multiple map resources could be displayed by creating their `cdla.maps.Map` objects individually; however using a `cdla.maps.MapSet` object provides some helpful additional functionality. When displaying multiple map resources at the same time it is important to know the overall bounds, center point, minimum zoom level, and maximum zoom level for the entire set of map resources. The `cdla.maps.MapSet` object provides this information. The example HTML document shown in Figure 6-4 displays a 1917 Sanborn® Fire Insurance Map of Ashville, North Carolina overlaid on top of a Soil Survey map for Buncombe County from 1920. The output of the document is shown in Figure 6-5.

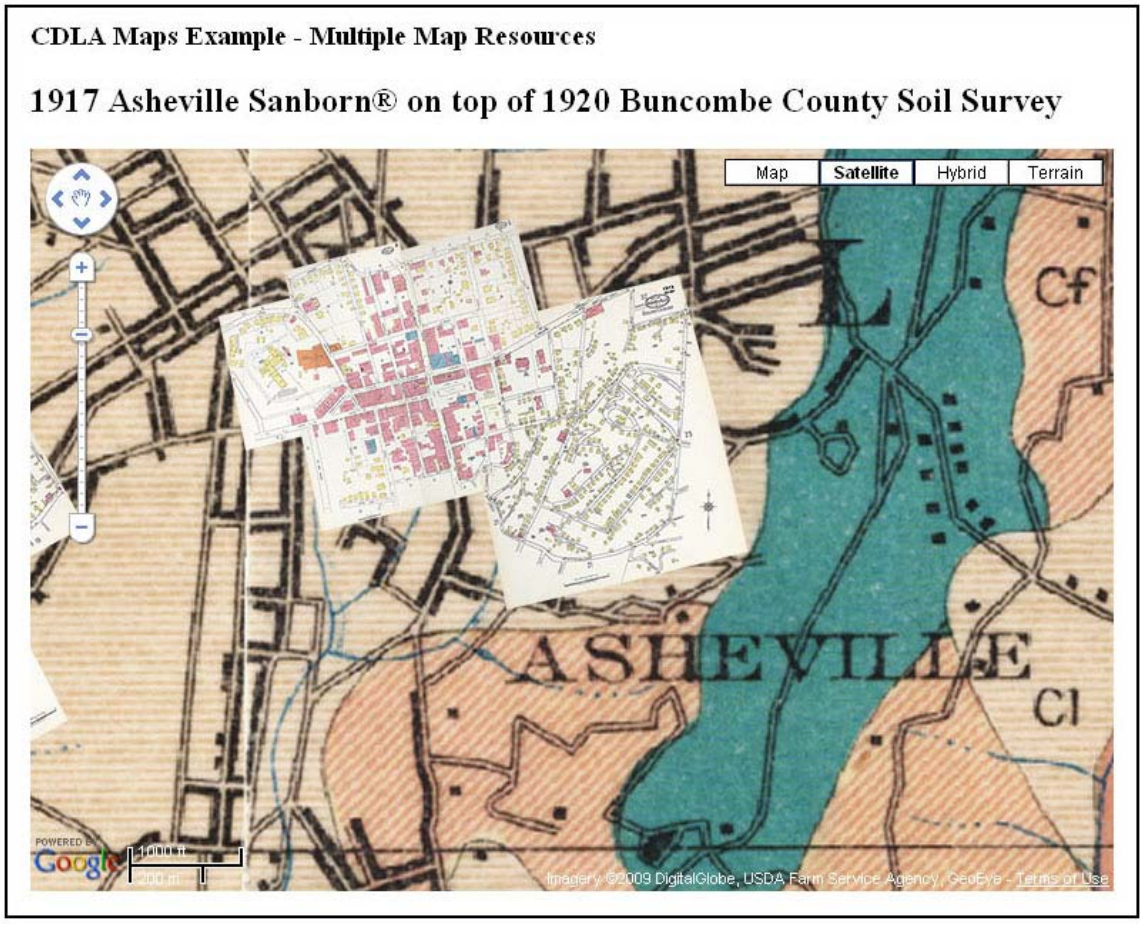
The `cdla.maps.Map` object's `layer()` method allows additional configuration parameters to be passed to the mapping API layer object's constructor. Here this is used to specify the z-ordering of the map resources by providing a Google Maps `GTileLayerOverlayOptions` object. The z-order of the layer objects tells the map what order to use when drawing the map layers. A higher z-order value indicates that the layer should be placed on top of layers with a lower z-order value. For this example the Sanborn® is displayed on top of the Soil Survey map.

Figure 6-4. HTML document for Multiple Map Resources

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8"/>
    <title>CDLA Maps API Example - Multiple Map Resource</title>
    <!-- load the Google Maps API -->
    <script type="text/javascript" src="http://maps.google.com/maps?file=api&amp;v=2
&amp;key=googlemapskey&amp;sensor=false">
    </script>
    <!-- load the CDLA Maps API -->
    <script type="text/javascript" src="http://server.unc.edu/cdlamaps/api/API_KEY">
    </script>
    <!-- load the map resources -->
    <script type="text/javascript" src="http://server.unc.edu/cdlamaps/api/mapdata/A
PI_KEY/NC:BUNCOMBE::SOILSURVEY:1920">
    </script>
    <script type="text/javascript" src="http://server.unc.edu/cdlamaps/api/mapdata/A
PI_KEY/NC:BUNCOMBE:ASHEVILLE::SANBORN:1917">
    </script>
    <script type="text/javascript">
    function init_page() {
      if (GBrowserIsCompatible()) {

        // create the Google Map object
        var map_options = {size: new GSize(760,520)};
        map = new GMap2(document.getElementById("mapcanvas"),map_options);

        // create the cdla.maps.MapSet object
        var mapset = new cdla.maps.MapSet(["NC:BUNCOMBE::SOILSURVEY:1920",
                                           "NC:BUNCOMBE:ASHEVILLE::SANBORN:1917"]);

        // determine the maximum zoom level at which the entire mapset is viewable
        var zoom = map.getBoundsZoomLevel(mapset.bounds());

        // make sure this zoom level is not less than
        // the minimum zoom level for the map layer
        if (zoom < mapset.minZoom()) {
            zoom = mapset.minZoom();
        }

        // initialize the Google Map
        // - center the display on the map layer's location
        //   at our calculated zoom level
        // - use Google's satellite map for the base map
        // - configure the default user interface
        map.setCenter(mapset.centroid(),zoom,G_SATELLITE_MAP);
        map.setUIToDefault();

        // add the layers to the map
        // use z-ordering so that Soil Survey is below the Sanborn
        var options = { zPriority: 20 };
        map.addOverlay(
            mapset.maps()["NC:BUNCOMBE:ASHEVILLE::SANBORN:1917"].layer(
                cdla.maps.API_GOOGLE, options)
        );
```

```
        options = { zPriority: 10 };
        map.addOverlay(
            mapset.maps()["NC:BUNCOMBE::SOILSURVEY:1920"].layer(
                cdla.maps.API_GOOGLE, options)
        );
      }
    }
    </script>
  </head>
  <body onload="init_page()" onunload="GUnload()">
    <h3>CDLA Maps Example - Multiple Map Resource</h3>
    <h2>1917 Asheville Sanborn&reg; on top of 1920 Buncombe County Soil Survey</h2>
    <div id="mapcanvas"></div>
  </body>
</html>
```

Figure 6-5. Multiple Map Resources



6.4 Multiple Map Resources – Selectable

This example also uses a `cdla.maps.MapSet` object to display multiple map resources. This time the maps are selected using a Map Set ID instead of individual Map IDs for each map resource. The map resources selected for this example are a set of four Sanborn® Fire Insurance Maps for Hendersonville, North Carolina. These four maps are all at the same scale and all cover the same general area. Instead of displaying them all at the same time, JavaScript event handlers are combined with a set of radio buttons to allow the user to switch between the different maps.

A function is defined which handles changing which map resource is displayed. This function, `togglemaps()`, is passed the Map ID of the desired map resource as its argument. When called it displays the requested map resource and hides the others. Four radio button `<input>` elements, one for each map resource, are defined in the body of the HTML with Map IDs for values. The `onclick` attribute of each radio button is used to call the `togglemaps()` function with the button's value as the argument whenever that button is clicked.

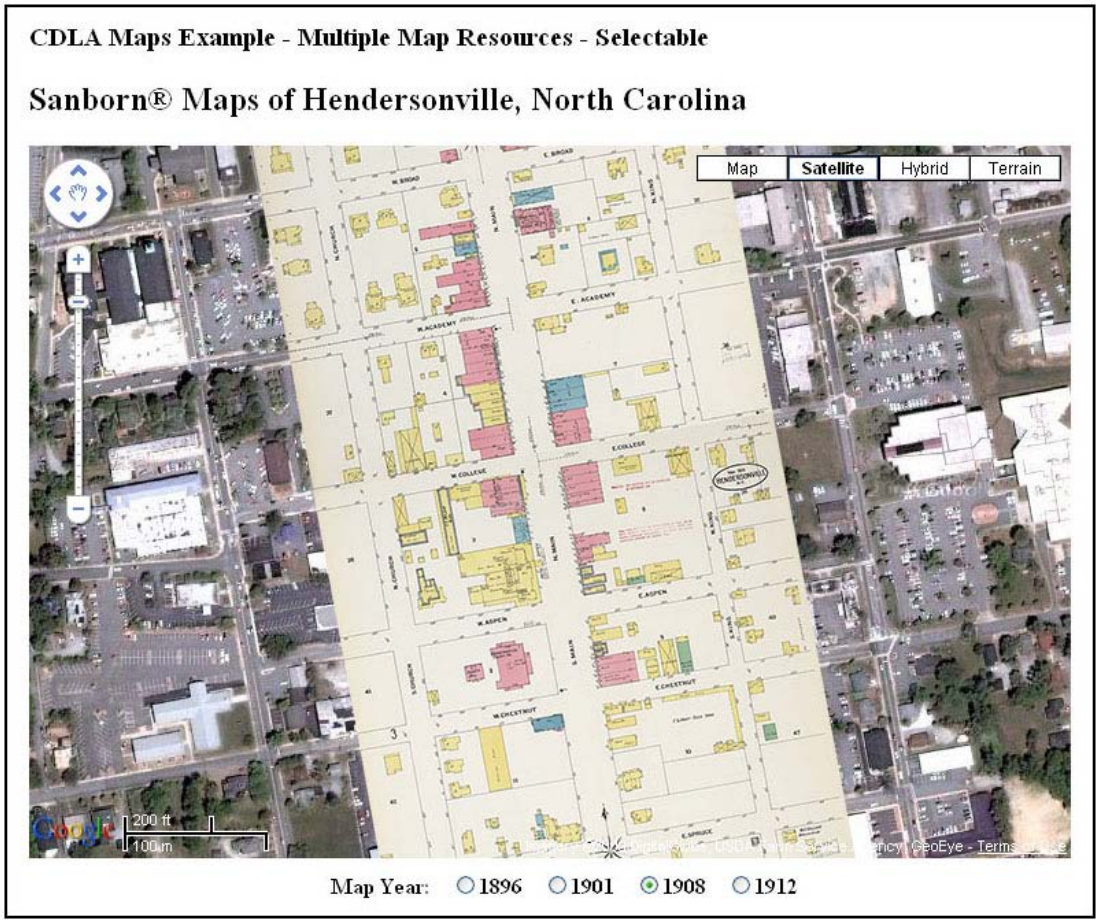
The HTML document for this example is shown in Figure 6-6 and the output is shown in Figure 6-7.

Figure 6-6. HTML document for Multiple Map Resources - Selectable

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8"/>
    <title>CDLA Maps API Example - Multiple Map Resource</title>
    <!-- load the Google Maps API -->
    <script type="text/javascript" src="http://maps.google.com/maps?file=api&amp;v=2
&amp;key=googlemapskey&amp;sensor=false"></script>
    <!-- load the CDLA Maps API -->
    <script type="text/javascript" src="http://server.unc.edu/cdlamaps/api/API_KEY">
    </script>
    <!-- load the map resources -->
    <script type="text/javascript" src="http://server.unc.edu/cdlamaps/api/mapdata/A
PI_KEY/NC:HENDERSON:HENDERSONVILLE::SANBORN:SET">
    </script>
    <script type="text/javascript">
    var togglemaps; // global so the event handler can access it

    function init_page() {
      if (GBrowserIsCompatible()) {

        // create the Google Map object
        var map_options = {size: new GSize(760,520)};
        map = new GMap2(document.getElementById("mapcanvas"),map_options);

        // create the cdla.maps.MapSet object
        var sanbornset = new cdla.maps.MapSet(
                                    ["NC:HENDERSON:HENDERSONVILLE::SANBORN:SET"]);

        // determine the maximum zoom level at which the entire mapset is viewable
        var zoom = map.getBoundsZoomLevel(sanbornset.bounds());

        // make sure this zoom level is not less than
        // the minimum zoom level for the map layer
        if (zoom < sanbornset.minZoom()) {
          zoom = sanbornset.minZoom();
        }

        // initialize the Google Map
        // - center the display on the map layer's location
        //   at our calculated zoom level
        // - use Google's satellite map for the base map
        // - configure the default user interface
        map.setCenter(sanbornset.centroid(),zoom,G_SATELLITE_MAP);
        map.setUIToDefault();

        // add the layers to the map
        // hide all except for the 1912 map
        var map_id;
        var sanbornmaps = sanbornset.maps();
        for (map_id in sanbornmaps) {
          map.addOverlay(sanbornmaps[map_id].layer());
          if (map_id !== "NC:HENDERSON:HENDERSONVILLE::SANBORN:1912") {
            sanbornmaps[map_id].layer().hide();
          }
        }
```

```
        // create the event handler
        togglemaps = function (display_id) {
          var map_id;
          for (map_id in sanbornmaps) {
            if (map_id === display_id) {
              sanbornmaps[map_id].layer().show();
            }
            else {
              sanbornmaps[map_id].layer().hide();
            }
          }
        }

      }
    }

    </script>
  </head>
  <body onload="init_page()" onunload="GUnload()">
    <h3>CDLA Maps Example - Multiple Map Resources - Selectable</h3>
    <h2>Sanborn&reg; Maps of Hendersonville, North Carolina</h2>
    <div id="mapcanvas"></div>
    <div style="text-align: center; margin: 10px; width: 760px; font-weight: bold;">
      <form>
        Map Year:
        <input type="radio" name="sanbornselector"
               value="NC:HENDERSON:HENDERSONVILLE::SANBORN:1896"
               style="margin-left: 15px;" onclick="togglemaps(this.value);">1896
        <input type="radio" name="sanbornselector"
               value="NC:HENDERSON:HENDERSONVILLE::SANBORN:1901"
               style="margin-left: 15px;" onclick="togglemaps(this.value);">1901
        <input type="radio" name="sanbornselector"
               value="NC:HENDERSON:HENDERSONVILLE::SANBORN:1908"
               style="margin-left: 15px;" onclick="togglemaps(this.value);">1908
        <input type="radio" name="sanbornselector"
               value="NC:HENDERSON:HENDERSONVILLE::SANBORN:1912"
               style="margin-left: 15px;" onclick="togglemaps(this.value);"
               checked="checked">1912
      </form>
    </div>
  </body>
</html>
```

Figure 6-7. Multiple Map Resources - Selectable



7 Next Steps

The CDLA Maps API is currently a functional prototype, but plans are in place to move towards a public release. The API will first be deployed in a pre-production setting on the CDLA's internal network. This will allow for further testing and refinement of the API in a controlled environment. Two focus areas for refinement are usability and performance. The server application will probably see the most change. The JavaScript library will remain largely in its current form but with support for additional mapping APIs.

Two important tasks for users of the API are creating API keys and locating map resources. Creating API keys is a simple process; however there is currently no automated method for a user to retrieve a lost key. Implementing this feature will provide a service to users and reduce the likelihood that a single user will create multiple keys for the same URL. Locating map resources must currently be done by browsing the available maps by geographic location. Additional "browse by" categories will be added along with a text search.

Database optimization and caching will be explored in order to improve performance. The most expensive database operations are the spatial queries used when generating pages for the map browsing interface. The results for these queries will only change if a map resource has been added, removed, or modified – an optimal scenario for caching. Improving the performance of the spatial operations will reduce load on the server and result in faster page loads for users.

On the administrative side, metadata for each of the almost 300 mapping resources will need to be added to the database. During this process the spatial extent and spatial scope models may need to be refined.

Once internal testing has completed successfully the API will be installed in the production environment and a small group of potential users will be recruited to serve as beta testers. This beta testing phase will be used to further refine the system before releasing the API to the general public.

References

"About North Carolina Maps" (n.d.). Retrieved July 14, 2009 from http://www.lib.unc.edu/dc/ncmaps/about.html

Bronn, J. (2008-2009). GeoDjango v1.1 documentation. Retrieved July 14, 2009 from http://geodjango.org/docs/

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). *Pattern-oriented software architecture: A system of patterns* (pp. 125-143). Chichester, England: Wiley.

Cornford, R. (2004). Javascript closures. Retrieved July 14, 2009 from http://jibbering.com/faq/faq_notes/closures.html

Crockford, D. (2002). JSLint: The JavaScript code quality tool. Retrieved July 14, 2009 from http://www.jslint.com

Django Software Foundation. (2005-2009). "Django appears to be a MVC framework, but…" Retrieved July 22, 2009 from http://docs.djangoproject.com/en/dev/faq/general/#django-appears-to-be-a-mvc-framework-but-you-call-the-controller-the-view-and-the-view-the-template-how-come-you-don-t-use-the-standard-names

Google. (2009). Google Maps API. Retrieved July 14, 2009 from http://code.google.com/apis/maps/

JSON.org (n.d.). Introducing JSON. Retrieved July 14, 2009 from http://www.json.org

National Institute of Standards and Technology. (2002). *Secure Hash Standard* (FIPS Publication No. 180-2). Retrieved July 22, 2009 from http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf

Microsoft Corporation. (2009). Bing Maps for Enterprise from Microsoft: Developer resources, online map APIs, SDKs, and map web services. Retrieved July 14, 2009 from http://www.microsoft.com/maps/developers/

Murray, G. and Ort, E. (2007) Restricting access to ajax services. Retrieved July 14, 2009 from http://java.sun.com/developer/technicalArticles/J2EE/usingapikeys/

NC Geographic Information Coordinating Council. (2009). NC OneMap. Retrieved July 14, 2009 from http://www.nconemap.com

Open Source Geospatial Foundation. (n.d.) OpenLayers: Free maps for the web. Retrieved July 14, 2009 from http://openlayers.org

PostgreSQL Global Development Group. (1996-2009). PostgreSQL. Retrieved July 14, 2009 from http://www.postgresql.org

Refractions Research. (2009) PostGIS. Retrieved July 14, 2009 from http://postgis.refractions.net

Resig, J. (2007). Pro JavaScript techniques [SpringerLink online edition]. doi:10.1007/978-1-4302-0283-7

Yahoo! Inc. (2009). Yahoo! Maps Web Services: The Yahoo! Maps Developer APIs. Retrieved July 14, 2009 from http://developer.yahoo.com/maps/

Zalewski, M. (2008). Browser security handbook, part 2. Retrieved July 14, 2009 from http://code.google.com/p/browsersec/wiki/Part2

Appendix A – API Documentation

CDLA Maps is a JavaScript API which lets you display geo-referenced historical maps from the collections of the Carolina Digital Library and Archives on your own web page. The API creates map layers which can easily be added to maps created using Google Maps and OpenLayers. Before you begin you will need to sign up for a free CDLA Maps API key.

Map IDs

The CDLA Maps API uses unique names, called Map IDs, to identify maps within the API. When you find a map resource that you want include on your page make note of the maps Map ID. Some maps, such as the Sanborn® Fire Insurance maps, are frequently used together. As a convenience a special Map ID, called a Map Set ID, can be used to refer to the entire map set. If a map is a part of a map set the Map Set ID will be listed alongside the Map ID on the map's information page.

Including the API in your page

To use the API on a page you will need to load it by including `<script>` tags in the `<head>` section of the HTML document. Two or more tags will be needed. The first tag loads the API itself:

```
<script type="text/javascript"
    src="http://server.unc.edu/cdlamaps/api/YOUR_API_KEY"></script>
```

Note: replace *YOUR_API_KEY* with your actual API key

You will also need to include one tag for each map or map set that you wish to use:

```
<script type="text/javascript"
    src="http://server.unc.edu/cdlamaps/api/mapdata/YOUR_API_KEY/MAP_ID">
</script>
```

Note: replace *YOUR_API_KEY* with your actual API key and *MAP_ID* with the Map ID/Map Set ID of the map/map set you wish to use.

The CDLA Maps API relies on methods of the mapping API you are using to create the map layer objects. Be sure that you load the mapping-API as well.

The API in Action

To display your chosen historical map you will first need to create a map interface using your chosen mapping API. Please refer to the mapping API's documentation if you are unsure how to do this. Next, use the CDLA Maps API to create a `cdla.maps.Map` object. This object contains metadata about the requested map - such as its name and a short description - along with information needed to create the map layer object used to display the map. Use the `layer()` method of the `cdla.maps.Map` object to obtain a map layer and then add that layer to the map interface you created earlier.

The following example uses the Google Maps API to display a 1920 Soil Survey map of Buncombe County, North Carolina.

```
// Create the Google Map
var map = new GMap2(document.getElementById("map_div");
map.setUIToDefault();

// initialize the map display
map.setCenter();

// Create the cdla.maps.Map object
var soilmap = new cdla.maps.Map("NC:BUNCOME::SOILSURVEY:1920");

// add the map to the google map
map.addOverlay(soilmap.layer());

// view the map centered at its minimum zoom level
map.setCenter(soilmap.centroid(), soilmap.minZoom());
```

For more information please refer to the API Reference.

API Reference

API Methods:

`cdla.maps.defaultAPI()`

Return the current default mapping API used to create map layers.

Arguments: none

Returns: `cdla.maps.API_TYPE`
The current default API type.

`cdla.maps.defaultAPI( api_type )`

Set the default mapping API used to create map layers.

Arguments: `api_type` (`cdla.maps.API_TYPE`)
A constant indicating the default mapping API to use.

Returns: `cdla.maps.API_TYPE`
The value of the newly set default API type.

API Constants:

The following constants can be used where a `cdla.maps.API_TYPE` is required.

`cdla.maps.API_GOOGLE` – The Google Maps API, version 2

`cdla.maps.API_OPENLAYERS` – The OpenLayers API

`cdla.maps.Bounds` Object Literal:

This object literal is used to define a bounding rectangle and is returned by the `bounds()` method of the `cdla.maps.Map` and `cdla.maps.MapSet` objects.

Properties:

`north` (Number) – The northern-most latitude of the bounding rectangle.

`south` (Number) – The southern-most latitude of the bounding rectangle.

`east` (Number) – The eastern-most longitude of the bounding rectangle.

`west` (Number) – The western-most longitude of the bounding rectangle.

`cdla.maps.Point` Object Literal:

This object literal is used to define a set of geographic coordinates and is returned by the `centroid()` method of the `cdla.maps.Map` and `cdla.maps.MapSet` objects.

Properties:

`lat` (Number) – The latitude value of the coordinate.

`lng` (Number) – The longitude value of the coordinate.

`cdla.maps.Map` Object:

This object represents a map resource. It includes methods for obtaining metadata for the resource and creating a layer object that can be used to display the map using a third-party mapping API.

Constructor:

`cdla.maps.Map( map_id )`

Arguments: `map_id` (String)
The unique Map ID of the resource being requested.

Methods:

`title()`

Returns the map's title.

Arguments: none

Returns: String

`id()`

Returns the map's unique Map ID.

Arguments: none

Returns: String

`description()`

Returns a text description of the map.

Arguments: none

Returns: String

bounds([use_api [, api_type]])

Returns an object defining the bounds of the area containing the map tiles. (Note: this is not the geographic extent of the area depicted by the map, but a bounding rectangle for the entire area covered by the map when displayed as an overlay. This includes the map legend, borders, etc.)

Arguments: `use_api` (Boolean)
True: Return an API specific map bounds object.
False: Return a `cdla.maps.Bounds` object literal
Default: true
`api_type` (`cdla.maps.API_TYPE`)
A constant indicating the mapping API to use. If no API type is specified the default API will be used. This value only applies if `use_api` is true.

Returns: `use_api` is true: a `cdla.maps.Bounds` object
`use_api` is false: The object type returned depends on the API type.

api_type value	Return Object
cdla.maps.API_GOOGLE	GLatLngBound
cdla.maps.API_OPENLAYERS	OpenLayers.Bounds

centroid([use_api [, api_type]])

Returns an object defining the center point of the map's bounds.

Arguments: `use_api` (Boolean)
True: Return an API specific map point object.
False: Return a `cdla.maps.Point` object literal
Default: true
`api_type` (`cdla.maps.API_TYPE`)
A constant indicating the mapping API to use. If no API type is specified the default API will be used. This value only applies if `use_api` is true.

Returns: `use_api` is true: a `cdla.maps.Pount` object
`use_api` is false: The object type returned depends on the API type.

api_type value	Return Object
cdla.maps.API_GOOGLE	GLatLng
cdla.maps.API_OPENLAYERS	OpenLayers.LonLat

minZoom()

Returns the minimum zoom level for which tiles exist for this map resource. The value is based on the tile structure used by the Google Maps API.

Arguments: none

Returns: Number

`maxZoom()`

Returns the maximum zoom level for which tiles exist for this map resource. The value is based on the tile structure used by the Google Maps API.

Arguments: none

Returns: Number

`layer( [api_type [, options]] )`

Returns an API specific map layer object.

Arguments: `api_type` (`cdla.maps.API_TYPE`)
A constant indicating the default mapping API to use. If no API type is specified the default API will be used.
`options` (varies, mapping API specific)
Additional options to pass to the constructor for the map layer object. Use of this argument is an advanced technique.

Returns: The object type returned depends on the API type.

`api_type` value	Return Object
`cdla.maps.API_GOOGLE`	`GTileLayerOverlay`
`cdla.maps.API_OPENLAYERS`	`OpenLayers.Layer.TMS`

`cdla.maps.MapSet` Object:

This object represents a collection of map resources. It includes methods for obtaining metadata for set and the `cdla.maps.Map` objects for the maps in the set.

Constructor:

`cdla.maps.MapSet( map_ids )`

Arguments: `map_ids` (Array<String>)
List of Map IDs for the resources being requested. If a Map Set ID is included it will be expanded into its component maps.

Methods:

`count()`

Returns the number of maps in the map set.

Arguments: none

Returns: Number

bounds([use_api [, api_type]])

Returns an object defining the area containing the maps in the set. (Note: this is not the geographic extent of the area depicted by the maps, but a bounding rectangle for the entire area covered by all maps in the set when displayed as overlays. This includes the map legends, borders, etc. for each map.)

Arguments: `use_api` (Boolean)
True: Return an API specific map bounds object.
False: Return a `cdla.maps.Bounds` object literal
Default: true
`api_type` (`cdla.maps.API_TYPE`)
A constant indicating the mapping API to use. If no API type is specified the default API will be used. This value only applies if `use_api` is true.

Returns: `use_api` is true: a `cdla.maps.Bounds` object
`use_api` is false: The object type returned depends on the API type.

api_type value	**Return Object**
`cdla.maps.API_GOOGLE`	`GLatLngBound`
`cdla.maps.API_OPENLAYERS`	`OpenLayers.Bounds`

centroid([use_api [, api_type]])

Returns an object defining the center point of the map set's bounds.

Arguments: `use_api` (Boolean)
True: Return an API specific map point object.
False: Return a `cdla.maps.Point` object literal
Default: true
`api_type` (`cdla.maps.API_TYPE`)
A constant indicating the mapping API to use. If no API type is specified the default API will be used. This value only applies if `use_api` is true.

Returns: `use_api` is true: a `cdla.maps.Pount` object
`use_api` is false: The object type returned depends on the API type.

api_type value	**Return Object**
`cdla.maps.API_GOOGLE`	`GLatLng`
`cdla.maps.API_OPENLAYERS`	`OpenLayers.LonLat`

minZoom()

Returns the minimum zoom level for all maps in the set. The value is based on the tile structure used by the Google Maps API.

Arguments: none

Returns: Number

`maxZoom()`

Returns the maximum zoom level for all maps in the set. The value is based on the tile structure used by the Google Maps API.

Arguments: none

Returns: Number

`mapIds()`

Returns an array containing the map ID of every map in the set.

Arguments: none

Returns: Array<String>

`maps()`

Returns an object literal containing the `cdla.mapsMap` object of all maps in the set. This object can be used like an associative array indexed by Map ID.

Arguments: none

Returns: `{`*`map_id`*`: cdla.maps.Map, … }`

Appendix B – JavaScript Code

api.js – The API library

```
/* Copyright (c) 2009 Kevin M. Eckhardt
 *
 * Permission is hereby granted, free of charge, to any person obtaining a
 * copy of this software and associated documentation files (the "Software"),
 * to deal in the Software without restriction, including without limitation
 * the rights to use, copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit persons to whom the
 * Software is furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included
 * in all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
 * OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
 * THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS
 * IN THE SOFTWARE.
 */

(function () {

    var ns, _apis, verifyApiType, verifyMapId;

    // setup the api namespace
    if (!window.cdla) {
        window.cdla = {};
    }
    if (!window.cdla.maps) {
        window.cdla.maps = {};
    }

    // short name for the namespace
    ns = window.cdla.maps;


    /*************************/
    /*    API type constants    */
    /*************************/

    ns.API_GOOGLE = 1;
    ns.API_OPENLAYERS = 2;

    _apis = [ns.API_GOOGLE, ns.API_OPENLAYERS];

    // Google Maps can use either a short namespace prefix 'G'
    // or a long one 'google.maps.'
    ns._apiShortPrefix = true;

    // set the initial default API type
    if (!ns._apiType) {
        ns._apiType = ns.API_GOOGLE;
    }
```

```
/************************/
/*    Library methods    */
/************************/

ns.defaultAPI = function (api_type) {
    /* Set / return the default mapping API type
     *
     * Args:
     *   api_type - cdla.maps API type constant
     *
     * Returns:
     *   the current (or new) default api type constant
     */
    if (!api_type) {
        return ns._apiType;
    }
    verifyApiType(api_type);
    ns._apiType = api_type;
    return ns._apiType;
};


/*******************/
/*    Map object    */
/*******************/

ns.Map = function (map_id) {
    /*  Constructor
     *
     * Args:
     *   map_id - String.  Map ID for the requested resource
     */
    if (map_id.constructor !== String) {
        throw "cdla.maps error: map_id should be a String.";
    }
    map_id = map_id.toUpperCase();
    verifyMapId(map_id);

    this.map_id = map_id; // Store the map id to access the metadata
    this.layer_obj = {}; // storage for map layer objects
};


ns.Map.prototype.id = function () {
    return this.map_id;
};


ns.Map.prototype.title = function () {
    return ns._maps[this.map_id].title;
};


ns.Map.prototype.description = function () {
    return ns._maps[this.map_id].description;
};


ns.Map.prototype.bounds = function (use_api, api_type) {
    /* Returns the bounds of the map's tile set
     *
     * Args:
     *   use_api - Boolean.  If true, return a mapping API bounds object
     *       If false, return a cdla.maps.Bounds object literal
     *       Default value is true
     *   api_type - mapping api to use.  if not specified, default will be used.
     *       only applies if use_api is true
     */
```

```
        var GLLB, GLL, lb;

        // if use_api is not there, or not a boolean, then default to true
        if (typeof(use_api) !== "boolean") {
            use_api = true;
        }

        if (use_api) {
            // if no api specified, use the default
            if (!api_type) {
                api_type = ns._apiType;
            }

            // verify the api is valid and present
            verifyApiType(api_type, true);

            // create the bounds object
            switch (api_type) {

            case ns.API_GOOGLE:
                // use the proper constuctors based on the detected api prefix
                if (ns._apiShortPrefix) {
                    GLLB = GLatLngBounds;
                    GLL = GLatLng;
                }
                else {
                    GLLB = google.maps.LatLngBounds;
                    GLL = google.maps.LatLng;
                }
                return new GLLB(
                    new GLL(ns._maps[this.map_id].bounds.south,
                                ns._maps[this.map_id].bounds.west),
                    new GLL(ns._maps[this.map_id].bounds.north,
                                ns._maps[this.map_id].bounds.east)
                );

            case ns.API_OPENLAYERS:
                lb = new OpenLayers.Bounds(ns._maps[this.map_id].bounds.west,
                    ns._maps[this.map_id].bounds.south,
                    ns._maps[this.map_id].bounds.east,
                    ns._maps[this.map_id].bounds.north
                );
                // must transform bounds from geographic lat/lng
                // to spherical mercator for google-like tiles
                lb.transform(new OpenLayers.Projection("EPSG:4326"),
                             new OpenLayers.Projection("EPSG:900913"));
                return lb;
            }
        }
        return ns._maps[this.map_id].bounds;
    };


ns.Map.prototype.centroid = function (use_api, api_type) {
        /* Returns the centroid of the map's tile set
         *
         * Args:
         *   use_api - Boolean.  If true, return a mapping API point object
         *       If false, return a cdla.maps.Point object literal
         *   api_type - mapping api to use.  if not specified, default will be used.
         *       only applies if use_api is true
         */
        var GLL, pt;

        // if use_api is not there, or not a boolean, then default to true
        if (typeof(use_api) !== "boolean") {
            use_api = true;
        }
```

```
    if (use_api) {
        // if no api specified, use the default
        if (!api_type) {
            api_type = ns._apiType;
        }

        // verify the api is valid and present
        verifyApiType(api_type, true);

        // create the point object
        switch (api_type) {

        case ns.API_GOOGLE:
            // use the proper constuctors based on the detected api prefix
            if (ns._apiShortPrefix) {
                GLL = GLatLng;
            }
            else {
                GLL = google.maps.LatLng;
            }
            return new GLL(ns._maps[this.map_id].centroid.lat,
                           ns._maps[this.map_id].centroid.lng);

        case ns.API_OPENLAYERS:
            pt = new OpenLayers.LonLat(ns._maps[this.map_id].centroid.lng,
                                       ns._maps[this.map_id].centroid.lat);
            // must transform from geographic lat/lng
            // to spherical mercator for google-like tiles
            pt.transform(new OpenLayers.Projection("EPSG:4326"),
                         new OpenLayers.Projection("EPSG:900913"));
            return pt;
        }
    }
    return ns._maps[this.map_id].centroid;
};


ns.Map.prototype.minZoom = function () {
    return ns._maps[this.map_id].min_zoom;
};


ns.Map.prototype.maxZoom = function () {
    return ns._maps[this.map_id].max_zoom;
};


ns.Map.prototype.layer = function (api_type, options) {
    /*  Creates the mapping API layer object
     *
     * Args:
     *   api_type - cdla.maps api type constant
     *       If not specfied, the default will be used
     *   options - object literal containing additional layer options
     *             to pass to the API constructor
     *
     * Returns:
     *   a map layer object for the requested api type
     */

    var GCrC, GCr, GTL, GTLO, crc, tl_options, tlo_options, layer_bounds,
        getUrl, resolutions, zl;

    // if no api specified, use the default
    if (!api_type) {
        api_type = ns._apiType;
    }
```

```
// verify the api is valid and present
verifyApiType(api_type, true);

// make sure options is an object literal
if (!options || options.constructor !== Object) {
    options = {};
}

// check to see if we've already built the object
if (this.layer_obj[api_type]) {
    return this.layer_obj[api_type];
}

// build the layer object
switch (api_type) {

case ns.API_GOOGLE:
    // use the proper constuctors based on the detected api prefix
    if (ns._apiShortPrefix) {
        GCrC = GCopyrightCollection;
        GCr = GCopyright;
        GTL = GTileLayer;
        GTLO = GTileLayerOverlay;
    }
    else {
        GCrC = google.maps.CopyrightCollection;
        GCr = google.maps.Copyright;
        GTL = google.maps.TileLayer;
        GTLO = google.maps.TileLayerOverlay;
    }

    crc = new GCrC("Layer created by");
    crc.addCopyright(
        new GCr(
            1,
            this.bounds(true, api_type),
            ns._maps[this.map_id].min_zoom,
            "CDLA, UNC-Chapel Hill"
        )
    );
    tl_options = {};
    tlo_options = {};
    // check user options for tilelayer options
    if (options.opacity) {
        tl_options.opacity = options.opacity;
    }
    if (options.draggingCursor) {
        tl_options.draggingCursor = options.draggingCursor;
    }
    // check user options for tilelayeroverlay options
    if (options.zPriority) {
        tlo_options.zPriority = options.zPriority;
    }
    // add needed options
    tl_options.isPng = true;
    tl_options.tileUrlTemplate = ns._maps[this.map_id].tile_url;
    this.layer_obj[api_type] = new GTLO(
        new GTL(
            crc,
            ns._maps[this.map_id].min_zoom,
            ns._maps[this.map_id].max_zoom,
            tl_options
        ),
        tlo_options
    );
    return this.layer_obj[api_type];
```

```
    case ns.API_OPENLAYERS:
        layer_bounds = this.bounds(true, api_type);
        getUrl = function (bounds) {
            /* Builds the url for the map tile showing the requested bounds */
            var res, x, y, z, url;

            // check if requested bounds is within the bounds of the tile set
            if (!layer_bounds.intersectsBounds(bounds)) {
                return "http://kevorama.dyndns.org:8888/blank.png";
            }

            // calculate the google-like tile coordinates
            res = this.map.getResolution();
            x = Math.round((bounds.left - this.maxExtent.left) /
                           (res * this.tileSize.w));
            y = Math.round((this.maxExtent.top - bounds.top) /
                           (res * this.tileSize.h));
            z = this.map.getZoom();

            // replace the coordinate placeholders in the url pattern
            // with the actual values
            url = this.url;
            url = url.replace(/\{X\}/, x);
            url = url.replace(/\{Y\}/, y);
            url = url.replace(/\{Z\}/, z);
            return url;
        };
        // build the list of resolutions tiles are available for
        resolutions = [];
        for (zl = ns._maps[this.map_id].min_zoom;
             zl <= ns._maps[this.map_id].max_zoom + 1; zl++) {
            resolutions.push(20037508.34 * 2 / 256 / Math.pow(2, zl));
        }
        options.type = 'png';
        options.getURL = getUrl;
        options.isBaseLayer = false;
        options.resolutions = resolutions;
        options.serverResolutions = resolutions.slice();
        options.numZoomLevels = null;
        options.attribution =
            "Layer created by <a href='http://cdla.unc.edu'>CDLA, UNC-Chapel Hill</a>";
        this.layer_obj[api_type] = new OpenLayers.Layer.TMS(
            ns._maps[this.map_id].title,
            ns._maps[this.map_id].tile_url,
            options
        );
        return this.layer_obj[api_type];
    }
};


/**********************/
/*    MapSet object    */
/**********************/

ns.MapSet = function (map_ids) {
    /*  Constructor
     *
     * Args:
     *   map_ids - Array<String>.  List of Map or Map Set IDs
     */
    var mid, msid;
    this.map_ids = []; // storage for map ids
    this.map_obj = {}; // storage for Map objects

    if (map_ids.constructor !== Array) {
        throw "cdla.maps error: map_ids should be an Array.";
    }
```

```
    // expand any map set ids into their component maps
    for (mid = 0; mid < map_ids.length; mid++) {
        if (map_ids[mid].constructor !== String) {
            throw "cdla.maps error: map_ids should only contain Strings.";
        }
        map_ids[mid] = map_ids[mid].toUpperCase();
        if (ns._mapsets[map_ids[mid]]) {
            for (msid = 0; msid < ns._mapsets[map_ids[mid]].maps.length; msid++) {
                this.map_ids.push(ns._mapsets[map_ids[mid]].maps[msid]);
            }
        }
        else {
            this.map_ids.push(map_ids[mid]);
        }
    }

    // create the Map objects
    for (mid = 0; mid < this.map_ids.length; mid++) {
        this.map_obj[this.map_ids[mid]] = new ns.Map(this.map_ids[mid]);
    }
};


ns.MapSet.prototype.count = function () {
    return this.map_ids.length;
};


ns.MapSet.prototype.bounds = function (use_api, api_type) {
    /* Returns the aggregate bounds of the map tile sets
     *
     * Args:
     *   use_api - Boolean.  If true, return a mapping API bounds object
     *       If false, return a cdla.maps.Bounds object literal
     *   api_type - mapping api to use.  if not specified, default will be used.
     *       only applies if use_api is true
     */

    var GLLB, GLL, lb, mid, bounds, north = null, south = null, east = null, west = null;

    // if use_api is not there, or not a boolean, then default to true
    if (typeof(use_api) !== "boolean") {
        use_api = true;
    }

    // calculate the bounds
    for (mid = 0; mid < this.map_ids.length; mid++) {
        bounds = this.map_obj[this.map_ids[mid]].bounds(false);
        north = (north === null ? bounds.north : Math.max(north, bounds.north));
        south = (south === null ? bounds.south : Math.min(south, bounds.south));
        east = (east === null ? bounds.east : Math.max(east, bounds.east));
        west = (west === null ? bounds.west : Math.min(west, bounds.west));
    }

    if (use_api) {
        // if no api specified, use the default
        if (!api_type) {
            api_type = ns._apiType;
        }

        // verify the api is valid and present
        verifyApiType(api_type, true);

        // create the bounds object
        switch (api_type) {
```

```
        case ns.API_GOOGLE:
            // use the proper constuctors based on the detected api prefix
            if (ns._apiShortPrefix) {
                GLLB = GLatLngBounds;
                GLL = GLatLng;
            }
            else {
                GLLB = google.maps.LatLngBounds;
                GLL = google.maps.LatLng;
            }
            return new GLLB(
                new GLL(south, west),
                new GLL(north, east)
            );

        case ns.API_OPENLAYERS:
            lb = new OpenLayers.Bounds(west, south, east, north);
            // must transform bounds from geographic lat/lng
            // to spherical mercator for google-like tiles
            lb.transform(new OpenLayers.Projection("EPSG:4326"),
                         new OpenLayers.Projection("EPSG:900913"));
            return lb;
        }
    }
    return {north: north, south: south, east: east, west: west};
};


ns.MapSet.prototype.centroid = function (use_api, api_type) {
    /* Returns the centroid of the aggregate bounds of th map tile sets
     *
     * Args:
     *   use_api - Boolean.  If true, return a mapping API point object
     *       If false, return a cdla.maps.Point object literal
     *   api_type - mapping api to use.  if not specified, default will be used.
     *       only applies if use_api is true
     */
    var GLL, pt, bounds, lat, lng;
    // if use_api is not there, or not a boolean, then default to true
    if (typeof(use_api) !== "boolean") {
        use_api = true;
    }

    // calculate the centroid
    bounds = this.bounds(false);
    lat = (bounds.north + bounds.south) / 2;
    lng = (bounds.east + bounds.west) / 2;

    if (use_api) {
        // if no api specified, use the default
        if (!api_type) {
            api_type = ns._apiType;
        }
        // verify the api is valid and present
        verifyApiType(api_type, true);

        // create the point object
        switch (api_type) {

        case ns.API_GOOGLE:
            // use the proper constuctors based on the detected api prefix
            if (ns._apiShortPrefix) {
                GLL = GLatLng;
            }
            else {
                GLL = google.maps.LatLng;
            }
            return new GLL(lat, lng);
```

```
        case ns.API_OPENLAYERS:
            pt = new OpenLayers.LonLat(lng, lat);
            // must transform from geographic lat/lng
            // to spherical mercator for google-like tiles
            pt.transform(new OpenLayers.Projection("EPSG:4326"),
                         new OpenLayers.Projection("EPSG:900913"));
            return pt;
        }
    }
    return {lat: lat, lng: lng};
};


ns.MapSet.prototype.minZoom = function () {
    var min_zoom = null, mid;

    // calculate the minimum zoom
    for (mid = 0; mid < this.map_ids.length; mid++) {
        if (min_zoom === null) {
            min_zoom = this.map_obj[this.map_ids[mid]].minZoom();
        }
        else {
            min_zoom = Math.min(min_zoom, this.map_obj[this.map_ids[mid]].minZoom());
        }
    }
    return min_zoom;
};


ns.MapSet.prototype.maxZoom = function () {
    var max_zoom = null, mid;

    // calculate the maximum zoom
    for (mid = 0; mid < this.map_ids.length; mid++) {
        if (max_zoom === null) {
            max_zoom = this.map_obj[this.map_ids[mid]].maxZoom();
        }
        else {
            max_zoom = Math.min(max_zoom, this.map_obj[this.map_ids[mid]].maxZoom());
        }
    }
    return max_zoom;
};


ns.MapSet.prototype.mapIds = function () {
    // return a copy of the map_ids array
    return this.map_ids.slice();
};


ns.MapSet.prototype.maps = function () {
    // return a copy of the map_objs object
    var mid, maps = {};
    for (mid = 0; mid < this.map_ids.length; mid++) {
        maps[this.map_ids[mid]] = this.map_obj[this.map_ids[mid]];
    }
    return maps;
};




/*************************/
/*    Helper functions    */
/*************************/
```

```
    verifyApiType = function (api_type, is_present) {
        /* determine if api_type matches a valid api type constant
         *
         * Arg:
         *   api_type - the api type to check
         *   is_present - boolean.  if true, then verify the api type can be found
         *       default value is false
         */
        var a, valid = false;

        if (typeof(is_present) !== "boolean") {
            is_present = false;
        }

        for (a in _apis) {
            if (api_type === _apis[a]) {
                valid = true;
                break;
            }
        }
        if (!valid) {
            throw "cdla.maps error: Unknown API Type";
        }

        if (is_present) {
            // check for presence of the mapping API
            switch (api_type) {

            case ns.API_GOOGLE:
                // check both google maps API prefixes
                if (window.GMap2) {
                    ns._apiShortPrefix = true;
                }
                else if (window.google.maps.Map2) {
                    ns._apiShortPrefix = false;
                }
                else {
                    throw "cdla.maps error: Google Maps API not found";
                }
                break;

            case ns.API_OPENLAYERS:
                if (!window.OpenLayers) {
                    throw "cdla.maps error: OpenLayers API not found";
                }
                break;
            }
        }
    };

    verifyMapId = function (map_id) {
        /*
         * Checks to see if the map_id matches a known map in the loaded data
         * Throws an error if it is not
         */
        if (map_id.constructor !== String) {
            throw "cdla.maps error: MAP_ID is not a String.";
        }
        if (ns._mapsets[map_id]) {
            throw "cdla.maps error: Invalid Map ID - \n\t'" + map_id + "' is a Map Set ID.";
        }
        if (!ns._maps[map_id]) {
            throw "cdla.maps error: Unknown Map ID - \n\tMap '" + map_id +
                  "' has not been loaded or it does not exist.";
        }
    };

}());
```

error.js – Script returned when an invalid API key is used

```
(function (){
    if (!window.cdlaerror) {
        window.cdlaerror = "An invalid cdla.maps API Key was used.  Please double check your API
Key.";
        alert(window.cdlaerror);
    }
})();
```

mapdata.js – Example data loading script

```
(function() {
  var ns, map_data, m;

  if (!window.cdla) {
      window.cdla = {};
  }
  if (!window.cdla.maps) {
      window.cdla.maps = {};
  }

  ns = window.cdla.maps;

  if (!ns._maps) {
      ns._maps = {};
  }
  if (!ns._mapsets) {
      ns._mapsets = {};
  }

  map_data = {
    "maps": [
      {
        "description": "1896 Sanborn Fire Insurance map set for Hendersonville, NC.",
        "title": "Sanborn Map - Hendersonville, NC (1896)",
        "max_zoom": 20,
        "bounds": {
          "west": -82.462512000000004, "east": -82.457504999999998,
          "north": 35.320058000000003, "south": 35.312435999999998
        },
        "tile_url": "http://docsouth.unc.edu/gtts/static/content/sanborn/Hendersonville/
1896/{Z}_{X}_{Y}.png",
        "min_zoom": 12,
        "centroid": {
          "lat": 35.316246999999997, "lng": -82.460008499999986
        },
        "id": "NC:HENDERSON:HENDERSONVILLE::SANBORN:1896"
      },
      {
        "description": "1901 Sanborn Fire Insurance map set for Hendersonville, NC.",
        "title": "Sanborn Map - Hendersonville, NC (1901)",
        "max_zoom": 20,
        "bounds": {
          "west": -82.462546000000003, "east": -82.457474000000005,
          "north": 35.320050999999999, "south": 35.312420000000003
        },
        "tile_url": "http://docsouth.unc.edu/gtts/static/content/sanborn/Hendersonville/
1901/{Z}_{X}_{Y}.png",
        "min_zoom": 12,
        "centroid": {
          "lat": 35.316235500000005, "lng": -82.460010000000011
        },
```

```
        "id": "NC:HENDERSON:HENDERSONVILLE::SANBORN:1901"
      },
      {
        "description": "1908 Sanborn Fire Insurance map set for Hendersonville, NC.",
        "title": "Sanborn Map - Hendersonville, NC (1908)",
        "max_zoom": 20,
        "bounds": {
          "west": -82.462500000000006, "east": -82.457453999999998,
          "north": 35.319481000000003, "south": 35.312412999999999
        },
        "tile_url": "http://docsouth.unc.edu/gtts/static/content/sanborn/Hendersonville/
1908/{Z}_{X}_{Y}.png",
        "min_zoom": 12,
        "centroid": {
          "lat": 35.315947000000001, "lng": -82.459976999999995
        },
        "id": "NC:HENDERSON:HENDERSONVILLE::SANBORN:1908"
      },
      {
        "description": "1912 Sanborn Fire Insurance map set for Hendersonville, NC.",
        "title": "Sanborn Map - Hendersonville, NC (1912)",
        "max_zoom": 20,
        "bounds": {
          "west": -82.465010000000007, "east": -82.457468000000006,
          "north": 35.319462999999999, "south": 35.312393999999998
        },
        "tile_url": "http://docsouth.unc.edu/gtts/static/content/sanborn/Hendersonville/
1912/{Z}_{X}_{Y}.png",
        "min_zoom": 12,
        "centroid": {
          "lat": 35.315928499999991, "lng": -82.46123900000002
        },
        "id": "NC:HENDERSON:HENDERSONVILLE::SANBORN:1912"
      }
    ],
    "map_sets": [
      {
        "maps": [
          "NC:HENDERSON:HENDERSONVILLE::SANBORN:1896",
          "NC:HENDERSON:HENDERSONVILLE::SANBORN:1901",
          "NC:HENDERSON:HENDERSONVILLE::SANBORN:1908",
          "NC:HENDERSON:HENDERSONVILLE::SANBORN:1912"
        ],
        "id": "NC:HENDERSON:HENDERSONVILLE::SANBORN:SET",
        "map_count": 4
      }
    ],
    "map_count": 4
  };

  for (m = 0; m < map_data.maps.length; m++) {
      ns._maps[map_data.maps[m].id] = map_data.maps[m];
  }

  for (m = 0; m < map_data.map_sets.length; m++) {
      ns._mapsets[map_data.map_sets[m].id] = map_data.map_sets[m];
  }
}());
```

Appendix C – Django Code

cdlamaps/settings.py

Note: Server names, passwords, and other items have been obscured for security reasons.

```
# Django settings for cdlamaps project.
import sys
sys.path.append('XXXXXX')

DEBUG = True
TEMPLATE_DEBUG = DEBUG

ADMINS = (
    ('Your Name', 'your_email@domain.com'),
)

MANAGERS = ADMINS

DATABASE_ENGINE = 'postgresql_psycopg2'
DATABASE_NAME = 'XXXXX'
DATABASE_USER = 'XXXXX'
DATABASE_PASSWORD = 'XXXXX'
DATABASE_HOST = 'XXXXX'
DATABASE_PORT = 'XXXXX'

# Local time zone for this installation.
TIME_ZONE = 'US/Eastern'

# Language code for this installation.
LANGUAGE_CODE = 'en-us'

SITE_ID = 1

# If you set this to False, Django will make some optimizations so as not
# to load the internationalization machinery.
USE_I18N = False

# Absolute path to the directory that holds media.
MEDIA_ROOT = 'XXXXX'

# URL that handles the media served from MEDIA_ROOT.
MEDIA_URL = '/cdlamaps/static/'

# URL prefix for admin media -- CSS, JavaScript and images.
ADMIN_MEDIA_PREFIX = '/cdlamaps/static/admin/'

# Make this unique, and don't share it with anybody.
SECRET_KEY = 'XXXXX'

API_KEY_SALT = 'XXXXX' # used when creating API keys
API_HOST = 'XXXXX' # used on template pages

# List of callables that know how to import templates from various sources.
TEMPLATE_LOADERS = (
    'django.template.loaders.filesystem.load_template_source',
    'django.template.loaders.app_directories.load_template_source',
)
```

```
MIDDLEWARE_CLASSES = (
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
)

ROOT_URLCONF = 'cdlamaps.urls'

TEMPLATE_DIRS = (
    # Put strings here, like "/home/html/django_templates" or "C:/www/django/templates".
    # Always use forward slashes, even on Windows.
    # Don't forget to use absolute paths, not relative paths.
    "/django/cdlamaps/templates",
)

INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.admin',
    'django.contrib.gis',
    'cdlamaps.api',
    'cdlamaps.browse',
    'cdlamaps.static',
)
```

cdlamaps/urls.py

```
from django.conf.urls.defaults import *
from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    (r'^api/', include('cdlamaps.api.urls')),
    (r'^browse/', include('cdlamaps.browse.urls')),
    (r'^admin/(.*)', admin.site.root),
    (r'^', include('cdlamaps.static.urls')),
)
```

cdlamaps/api/urls.py

```
from django.conf.urls.defaults import *

urlpatterns = patterns('cdlamaps.api.views',
    (r'^createkey/?', 'create_key_view', {}, 'cdlamaps-api-create'),
    (r'^mapdata/(?P<api_key>.*)/(?P<map_ids>.*)', 'mapdata_view', {}, 'cdlamaps-api-mapdata'),
    (r'^mapdata/(?P<map_ids>.*)', 'mapdata_view', {'api_key': None}, 'cdlamaps-api-mapdata'),
    (r'^(?P<api_key>.*)', 'api_view', {}, 'cdlamaps-api'),
)
```

cdlamaps/api/models.py

```
from django.contrib.gis.db import models
from django.template import defaultfilters
from django.db import IntegrityError
import re


class GeoModel(models.Model):
    '''
    Base model from which to build the other models
    '''
    # creation time
    ctime = models.DateTimeField(auto_now_add=True)
    # modification time
    mtime = models.DateTimeField(auto_now=True)
    # use the GeoManager to allow spatial queries
    objects = models.GeoManager()

    class Meta(object):
        abstract=True


class ApiKey(GeoModel):
    # default, autoincrement primary key
    id = models.AutoField(db_column='key_id', primary_key=True)
    # the url used to request the API key
    url = models.URLField(verify_exists=False, max_length=512)
    # name of requestor
    name = models.CharField(max_length=128, blank=True)
    # email address of requestor
    email = models.EmailField(max_length=256, blank=True)
    # the API key
    key = models.CharField(max_length=40)
    # RegExp pattern used to verify the key
    match_pattern = models.CharField(max_length=512)
    # number of successful API and Map Data requests made using this API key
    access_count = models.IntegerField(default=0)

    def __unicode__(self):
        return self.url


class SpatialScope(GeoModel):
    '''
    provides an abstract representation of the relative size a spatial extent
    e.g. 'City' or 'County'
    '''
    # default, autoincrement primary key
    id = models.AutoField(db_column='scope_id', primary_key=True)
    # name of the spacial scope
    name = models.CharField(max_length=100)
    # unique string identifier used in urls.  based on name.
    slug = models.SlugField(max_length=105, blank=True)
    # used to compare spacial scopes.  A larger value indicates the scope covers a larger
    # relative area. For example, 'County' has a value larger than 'City' but smaller
    # than 'State'
    level = models.IntegerField()

    def save(self, force_insert=False, force_update=False):
        '''
        ensure a slug is present and it is unique before saving
        '''
        if not self.slug:
            # create a slug from the extent name
            self.slug = defaultfilters.slugify(self.name)
```

```
        # make sure it is unique
        existing_slugs = SpatialScope.objects.filter(
                                        slug__startswith=self.slug).exclude(id=self.id)
        existing_slugs = existing_slugs.values_list('slug',flat=True)

        if existing_slugs:
            num_list = []
            p = re.compile('.+-(\d+)')
            for slug in existing_slugs:
                m = p.match(slug)
                if m:
                    num_list.append(int(m.group(1)))
            if num_list:
                new_slug_num = max(num_list)+1
            else:
                new_slug_num = 1
            self.slug += '-%d' % new_slug_num
        super(SpatialScope, self).save(force_insert, force_update)

    def __unicode__(self):
        return "%d - %s" % (self.level,self.name)

    class Meta:
        ordering = ['level']


class SpatialExtent(GeoModel):
    '''
    Describes a geographic area
    '''
    # default, autoincrement primary key
    id = models.AutoField(db_column='extent_id', primary_key=True)
    # name of the geographic area
    name = models.CharField(max_length=100)
    # unique string identifier used in urls.  based on name.
    slug = models.SlugField(max_length=105, blank=True)
    # the relative size of the geograhic area
    scope = models.ForeignKey(SpatialScope)
    # geometry field containing the bounds of the area
    bounds = models.MultiPolygonField(srid=4326)

    def save(self, force_insert=False, force_update=False):
        '''
        ensure a slug is present and it is unique for the scope before saving
        '''
        if not self.slug:
            # create a slug from the extent name
            self.slug = defaultfilters.slugify(self.name)
        # make sure it is unique for the scope
        existing_slugs = SpatialExtent.objects.filter(
                        scope=self.scope,slug__startswith=self.slug).exclude(id=self.id)
        existing_slugs = existing_slugs.values_list('slug',flat=True)
        if existing_slugs:
            num_list = []
            p = re.compile('.+-(\d+)')
            for slug in existing_slugs:
                m = p.match(slug)
                if m:
                    num_list.append(int(m.group(1)))
            if num_list:
                new_slug_num = max(num_list)+1
            else:
                new_slug_num = 1
            self.slug += '-%d' % new_slug_num
        super(SpatialExtent, self).save(force_insert, force_update)

    def __unicode__(self):
        return '%s (%s)' % (self.name, self.scope)
```

```
    class Meta:
        ordering = ['name','scope']


# Options used by the date_resolution field in the Map model
DATE_RESOLUTION_CHOICES = (
    ('D','Day'),
    ('M','Month'),
    ('Y','Year'),
    ('T','Decade'),
    ('C','Century')
)


class Map(GeoModel):
    # default, autoincrement primary key
    id = models.AutoField(db_column='map_id', primary_key=True)
    # the unique Map ID used to access the resource using the API
    api_id = models.CharField(max_length=512, unique=True, db_index=True)
    # Title of the map used to create the resource
    title = models.CharField(max_length=512, blank=True)
    # Text description of the locations and information depicted by the map.
    description = models.TextField(blank=True)
    # Publication date of the source map.
    date = models.DateField()
    date_resolution = models.CharField(max_length=1, choices=DATE_RESOLUTION_CHOICES)
    # the geographic area depicted by the map
    spatial_extent = models.ForeignKey(SpatialExtent)
    #  bounding box for the entire area "covered" by the map tiles
    #when displayed within a web map
    display_bbox = models.PolygonField(srid=4326)
    # minimum zoom level for the map based the zoom levels used by the Google Maps API
    min_zoom = models.IntegerField()
    # maximum zoom level for the map based the zoom levels used by the Google Maps API
    max_zoom = models.IntegerField()
    # URL pattern defining the location and naming structure of the map tiles
    tile_url = models.URLField(verify_exists=False, max_length=512)
    # URL for source map in the ContentDM repository
    source_url = models.URLField(verify_exists=False, max_length=512)

    # create a date string based on the date_resolution
    def _display_date(self):
        if self.date_resolution == 'D':
            return "%d/%d/%d" % (self.date.month, self.date.day, self.date.year)
        elif self.date_resolution == 'M':
            return "%d/%d" % (self.date.month, self.date.year)
        elif self.date_resolution == 'Y':
            return "%d" % self.date.year
        elif self.date_resolution == 'D':
            return "%d0s" % (self.date.year/10)
        elif self.date_resolution == 'C':
            return "%d00s" % (self.date.year/100)
        return ''
    display_date = property(_display_date)

    def save(self, force_insert=False, force_update=False):
        '''
        ensure api_id is unique among both Map and MapSet before saving
        '''
        if MapSet.objects.filter(api_id=self.api_id).count():
            raise IntegrityError("api_id should be unique among both Map and MapSet")
        super(Map, self).save(force_insert, force_update)

    def __unicode__(self):
        return self.api_id

    class Meta:
        ordering = ['api_id']
```

```
class MapSet(GeoModel):
    '''
    A set of associated maps that can be requested together
    '''
    # default, autoincrement primary key
    id = models.AutoField(db_column='mapset_id', primary_key=True)
    # the unique Map Set ID used to access the resources using the API
    api_id = models.CharField(max_length=512, unique=True, db_index=True)
    # name for the map set
    name = models.CharField(max_length=512, blank=True)
    # Text description of the map set.
    description = models.TextField(blank=True)
    # the maps in the map set
    maps = models.ManyToManyField(Map)

    def save(self, force_insert=False, force_update=False):
        '''
        ensure api_id is unique among both Map and MapSet before saving
        '''
        if Map.objects.filter(api_id=self.api_id).count():
            raise IntegrityError("api_id should be unique among both Map and MapSet")
        super(MapSet, self).save(force_insert, force_update)

    def __unicode__(self):
        return self.api_id

    class Meta:
        ordering = ['api_id']
```

cdlamaps/api/views.py

```
from cdlamaps.api import models as api_models
from django import forms
from django.shortcuts import render_to_response
from django.utils import simplejson
from django.conf import settings
import sha, re
from datetime import datetime


def create_key_view(request):
    '''
    displays and processes the API key request form
    '''
    # if we have data, bind it to the form
    if request.method == 'POST':
        form = ApiKeyForm(request.POST)
        # if valid, create the key and display it
        if form.is_valid():
            data = form.cleaned_data
            key = create_api_key(data)
            return render_to_response('mapsapi/displaykey.html', {
                'navcat': 'createkey',
                'url': data['url'], 'key': key.key
            })
    # no data so create a new empty form
    else:
        form = ApiKeyForm()

    return render_to_response('api/createkey.html', {
        'navcat': 'createkey',
        'form': form,
    })
```

```
def mapdata_view(request,api_key,map_ids):
    '''
    If the API key is valid:
        Renders a JavaScript file which loads the metadata for the
        requested Map IDs into the API library.
    Otherwise:
        Renders a JavaScript file which alerts the user that
        the API key was invalid

    Args:
        api_key - String.  The provided API key
        map_ids - String.  The requested Map or Map Set IDs
    '''
    if not valid_api_key(request,api_key):
        return render_to_response('api/error.js',mimetype='text/javascript')

    map_data = {
        'map_count': 0, 'maps': [], 'map_sets': [],
    }

    # can provide multiple IDs in same request by separating them with pipes
    map_ids = map_ids.upper().split('|')

    # process the ids
    for id in map_ids:
        mapset = None
        # if this is a MapSet ID, then get the list of Maps in the set
        try:
            mapset = api_models.MapSet.objects.get(api_id=id)
            map_set_info  = {
                'id': id,  'map_count': 0, 'maps': [],
            }
            map_list = mapset.maps.all()
        # if not, then treat it as a Map ID
        except api_models.MapSet.DoesNotExist:
            map_list = api_models.Map.objects.filter(api_id=id)

        # process each map
        for map in map_list:
            map_info = {
                'id': map.api_id,
                'title': map.title,
                'description': map.description,
                'tile_url': map.tile_url,
                'min_zoom': map.min_zoom,
                'max_zoom': map.max_zoom,
                'bounds': {},
                'centroid': {'lat': map.display_bbox.centroid.y,
                             'lng': map.display_bbox.centroid.x}
            }
            bounds = map.display_bbox.extent
            map_info['bounds']['north'] = bounds[3]
            map_info['bounds']['south'] = bounds[1]
            map_info['bounds']['east'] = bounds[2]
            map_info['bounds']['west'] = bounds[0]
            map_data['maps'].append(map_info)
            map_data['map_count'] += 1

            if mapset:
                map_set_info['map_count'] += 1
                map_set_info['maps'].append(map.api_id)

        if mapset and map_set_info['map_count'] > 0:
            map_data['map_sets'].append(map_set_info)

    return render_to_response('api/mapdata.js',
               {'json': simplejson.dumps(map_data)},mimetype='text/javascript')
```

```
def api_view(request,api_key):
    '''
    If the API key is valid:
        Renders the JavaScript API library
    Otherwise:
        Renders a JavaScript file which alerts the user that
        the API key was invalid

    Args:
        api_key - String.  The provided API key
    '''
    if not valid_api_key(request,api_key):
        return render_to_response('api/error.js',
                                  {'meta': request.META},mimetype='text/javascript')
    return render_to_response('api/api.js',
                              {'meta': request.META},mimetype='text/javascript')



###############
#    Forms    #
###############

class ApiKeyForm(forms.Form):
    '''
    Form used for API key requests
    '''
    # url to bind to the API key
    url = forms.URLField(max_length=512)
    # contacts name and email address
    name = forms.EmailField(max_length=512)
    email = forms.EmailField(max_length=512)



##########################
#    Helper Functions    #
##########################

def create_api_key(data):
    '''
    Creates an API key

    Args:
        data - ApiKeyForm.cleaned_data.  The data used to create the key

    Returns:
        cdlamaps.api.models.ApiKey - The newly created key
    '''
    apikey = api_models.ApiKey()
    apikey.url = data['url']
    apikey.email = data['email']

    # create the RegExp to used when validating the key
    match_pattern = re.escape(data['url'])
    # allow the "www" prefix to be optional
    if match_pattern.startswith(re.escape('http://www.')):
        match_pattern.replace(re.escape('www.'),"(www\\.)?",1)
    apikey.match_pattern = match_pattern

    # create the key
    key = sha.new(API_KEY_SALT)
    key.update(data['url'])
    key.update(datetime.today().isoformat())
    apikey.key = key.hexdigest()
    apikey.save()
    return apikey
```

```
def valid_api_key(request,api_key):
    '''
    Determine if the API key used is valid for the referring URL

    Args:
        api_key - String.  The API key to validate

    Returns:
        Boolean - True if the key is valid for the referring URL
                  False otherwise
    '''
    # if no referrer is available, then assume a file request and validate
    if 'HTTP_REFERER' not in request.META or not request.META['HTTP_REFERER']:
        return True

    url = request.META['HTTP_REFERER']

    # request came from this server, so validate
    if url.startswith(settings.API_HOST):
        return True

    # empy key is invalid
    if not api_key:
        return False

    api_key = api_key.lower()
    # api keys _should_ be unique, but just in case we'll pretend they're not
    key_list = api_models.ApiKey.objects.filter(key=api_key)
    for key in key_list:
        if re.match(key.match_pattern,url):
            key.access_count += 1
            key.save()
            return True
    return False
```

cdlamaps/api/admin.py

```
from cdlamaps.api import models
from django.contrib import admin

class SpatialScopeAdmin(admin.ModelAdmin):
    prepopulated_fields = {"slug": ("name",)}

class SpatialExtentAdmin(admin.ModelAdmin):
    prepopulated_fields = {"slug": ("name",)}

admin.site.register(models.SpatialScope, SpatialScopeAdmin)
admin.site.register(models.SpatialExtent, SpatialExtentAdmin)
admin.site.register(models.Map)
admin.site.register(models.MapSet)
admin.site.register(models.ApiKey)
```

cdlamaps/browse/urls.py

```
from django.conf.urls.defaults import *

urlpatterns = patterns('cdlamaps.browse.views',
    (r'^map/(?P<api_id>[\w:]+)$', 'map_detail_view', {}, 'cdlamaps-browse-map'),
    (r'^mapset/(?P<api_id>[\w:]+)$', 'mapset_detail_view', {}, 'cdlamaps-browse-mapset'),
    (r'^(?P<scope_slug>[-\w]+)/?$', 'scope_view', {}, 'cdlamaps-browse-scope'),
    (r'^(?P<scope_slug>[-\w]+)/(?P<extent_slug>[-\w]+)$', 'extent_view', {}, 'cdlamaps-browse-
extent'),
    (r'^$', 'browse_index_view', {}, 'cdlamaps-browse'),
)
```

cdlamaps/browse/views.py

```
from cdlamaps.api import models as api_models
from django.shortcuts import render_to_response, get_list_or_404, get_object_or_404
from django.template import RequestContext
from django.db.models import Q
from django.conf import settings


def browse_index_view(request):
    '''
    Renders an index page presenting options to browse maps by city, county, or region
    Page includes links to scope_view
    '''
    return render_to_response('browse/index.html', {'navcat': 'browse'},
                              RequestContext(request))


def scope_view(request,scope_slug):
    '''
    Renders a page containing a list of spatial extents having the reqested scope.
    A count of the maps contained within each extent is included
    Page includes links to extent_view

    Args:
        scope-name - String.  The name of the requested scope
    '''
    # get the requested scope
    scope = get_object_or_404(api_models.SpatialScope, slug=scope_slug)

    # get a list of all extents having the requested scope
    extent_list = get_list_or_404(api_models.SpatialExtent, scope=scope)
    scope_info = []
    for extent in extent_list:
        extent_dict = {}
        extent_dict['name'] = extent.name
        extent_dict['slug'] = extent.slug
        scope_list = {}

        # find the maps which intersect the extent.
        # Only include maps at or below the extents scope level
        for map in api_models.Map.objects.filter(
                            spatial_extent__scope__level__lte=extent.scope.level,
                            spatial_extent__bounds__intersects=extent.bounds):

            # use the cenroid of the map's extent as a check to see
            # if it is primarily located within this extent
            if extent.bounds.contains(map.spatial_extent.bounds.centroid):
                map_scope_name = map.spatial_extent.scope.name
                map_scope_level = map.spatial_extent.scope.level
```

```
                # increment the map count for this map's scope
                if map_scope_name not in scope_list:
                    scope_list[map_scope_name] = {
                        'name': map_scope_name,
                        'level': map_scope_level,
                        'map_count': 1
                    }
                else:
                    scope_list[map_scope_name]['map_count'] += 1
        extent_dict['map_counts'] = scope_list.values()

        # if this extent contains maps, then add it to the list for display
        if extent_dict['map_counts']:
            scope_info.append(extent_dict)
    return render_to_response('browse/scope.html',
                        {'navcat': 'browse', 'scope': scope, 'scope_info': scope_info},
                        RequestContext(request))


def extent_view(request,scope_slug,extent_slug):
    '''
    Renders a page containing a list of maps contained within the specified extent
    Page includes links to map_detail_view

    Args:
        scope_slug - String.  Slug identifying the scope of the requested extent
        extent_slug - String.  Slug identifying the requested extent
    '''
    # get the requested scope
    scope = get_object_or_404(api_models.SpatialScope, slug=scope_slug)
    # get the requested extent
    extent = get_object_or_404(api_models.SpatialExtent, slug=extent_slug, scope=scope)
    scope_list = {}
    # find the maps which intersect the extent.
    # Only include maps at or below the extent's scope level
    for map in api_models.Map.objects.filter(
                        spatial_extent__scope__level__lte=scope.level,
                        spatial_extent__bounds__intersects=extent.bounds):
        # use the cenroid of the map's extent as a check to see
        # if it is primarily located within this extent
        if extent.bounds.contains(map.spatial_extent.bounds.centroid):
            map_extent_name = map.spatial_extent.name
            map_extent_slug = map.spatial_extent.slug
            map_scope_name = map.spatial_extent.scope.name
            map_scope_slug = map.spatial_extent.scope.slug
            map_scope_level = map.spatial_extent.scope.level
            # add map to the list of maps which have its scope
            if map_scope_slug not in scope_list:
                scope_list[map_scope_slug] = {
                    'name': map_scope_name,
                    'slug': map_scope_slug,
                    'level': map_scope_level,
                    'locations': {}
                }
            if map_extent_slug not in scope_list[map_scope_slug]['locations']:
                scope_list[map_scope_slug]['locations'][map_extent_slug] = {
                    'name': map_extent_name,
                    'slug': map_extent_slug,
                    'maps': []
                }
            scope_list[map_scope_slug]['locations'][map_extent_slug]['maps'].append(map)
    for scope_slug in scope_list.keys():
        scope_list[scope_slug]['locations'] =
                                    scope_list[scope_slug]['locations'].values()
    maps = scope_list.values()
    return render_to_response('browse/extent.html',
                            {'navcat': 'browse', 'extent': extent, 'maps': maps},
                            RequestContext(request))
```

```
def map_detail_view(request,api_id):
    '''
    Renders a page that displays the requested map and it's metadata

    Args:
        api_id - String.  Map ID for the requested map
    '''
    # get the requested map
    map = get_object_or_404(api_models.Map,api_id=api_id)
    #  Build a hierarchical list of the map's location starting with
    # the name of its own extent and adding the names of the higher scoped
    # extents it is within
    location = [
        {
            'name': map.spatial_extent.name,
            'scope': map.spatial_extent.scope.name,
            'level': map.spatial_extent.scope.level
        }
    ]
    for extent in api_models.SpatialExtent.objects.filter(
                            scope__level__gt=map.spatial_extent.scope.level,
                            bounds__contains=map.spatial_extent.bounds.centroid):
        location.append({'name': extent.name, 'scope': extent.scope.name,
                         'level': extent.scope.level})
    return render_to_response('browse/map_detail.html',
                    {'navcat': 'browse', 'map': map, 'host': settings.API_HOST,
                     'location': location}, RequestContext(request))


def mapset_detail_view(request,api_id):
    '''
    Renders a page that displays the maps that make up the requested map set

    Args: api_id - String.  Map Set ID for the requested map set
    '''
    # get the requested map set
    mapset = get_object_or_404(api_models.MapSet, api_id=api_id)
    scope_list = {}
    # find the maps which intersect the extent.
    # Only include maps at or below the extents scope level
    for map in mapset.maps.all():
        map_extent_name = map.spatial_extent.name
        map_extent_slug = map.spatial_extent.slug
        map_scope_name = map.spatial_extent.scope.name
        map_scope_slug = map.spatial_extent.scope.slug
        map_scope_level = map.spatial_extent.scope.level
        # add map to the list of maps which have its scope
        if map_scope_slug not in scope_list:
            scope_list[map_scope_slug] = {
                'name': map_scope_name,
                'slug': map_scope_slug,
                'level': map_scope_level,
                'locations': {}
            }
        if map_extent_slug not in scope_list[map_scope_slug]['locations']:
            scope_list[map_scope_slug]['locations'][map_extent_slug] = {
                'name': map_extent_name,
                'slug': map_extent_slug,
                'maps': []
            }
        scope_list[map_scope_slug]['locations'][map_extent_slug]['maps'].append(map)
    for scope_slug in scope_list.keys():
        scope_list[scope_slug]['locations'] =
                                scope_list[scope_slug]['locations'].values()
    maps = scope_list.values()
    return render_to_response('browse/mapset_detail.html',
                              {'navcat': 'browse', 'mapset': mapset, 'maps': maps},
                              RequestContext(request))
```

cdlamaps/static/urls.py

```
from django.conf.urls.defaults import *

urlpatterns = patterns('cdlamaps.static.views',
    (r'^docs/?$', 'static_view', {'navcat': 'docs', 'template_name': 'index'},'cdlamaps-docs-
index'),
    (r'^docs/(?P<template_name>.*)$', 'static_view', {'navcat': 'docs'},'cdlamaps-docs'),
    (r'^$', 'static_view', {'navcat': 'home', 'template_name': 'index'},'cdlamaps-home'),
)
```

cdlamaps/static/views.py

```
from django.shortcuts import render_to_response
from django.template import RequestContext


def static_view(request, template_name, navcat):
    if navcat == 'docs':
        template_name = 'docs/'+template_name
    template_name = "static/%s.html" % template_name
    return render_to_response(template_name,{'navcat': navcat},RequestContext(request))
```

cdlamaps/templates/base.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>cdla.maps :: {% block title %}{% endblock %}</title>
<link rel="stylesheet" type="text/css" href="{{ MEDIA_URL }}css/cdlamaps.css"/>
{% block extrahead %}{% endblock %}
</head>
<body {% block body_attr %}{% endblock %}>
<div id="pagewrap">
  <div id="header">CDLA Maps API</div>
  <div id="navbar">
    <ul>
      <li><a{% ifequal navcat "home" %} class="active"{% endifequal %}
              href="{% url cdlamaps-home %}">Home</a></li>
      <li class="divider">:</li>
      <li><a{% ifequal navcat "browse" %} class="active"{% endifequal %}
              href="{% url cdlamaps-browse %}">Browse Maps</a></li>
      <li class="divider">:</li>
      <li><a{% ifequal navcat "createkey" %} class="active"{% endifequal %}
              href="{% url cdlamaps-api-create %}">Get an API Key</a></li>
      <li class="divider">:</li>
      <li><a{% ifequal navcat "docs" %} class="active"{% endifequal %}
              href="{% url cdlamaps-docs-index %}">Documentation</a></li>
    </ul>
  </div>
  <div id="content">
    {% block content %}{% endblock %}
  </div>
</div>
</body>
</html>
```

cdlamaps/templates/api/api.js

This file contains the JavaScript library. See Appendix B.

cdlamaps/templates/api/error.js

This file contains the invalid API alert script. See Appendix B.

cdlamaps/templates/api/mapdata.js

```
(function() {
    var ns, map_data, m;

    if (!window.cdla) {
        window.cdla = {};
    }
    if (!window.cdla.maps) {
        window.cdla.maps = {};
    }
    ns = window.cdla.maps;

    if (!ns._maps) {
        ns._maps = {};
    }

    if (!ns._mapsets) {
        ns._mapsets = {};
    }

    map_data = {{ json|safe }};

    for (m = 0; m < map_data.maps.length; m++) {
        ns._maps[map_data.maps[m].id] = map_data.maps[m];
    }

    for (m = 0; m < map_data.map_sets.length; m++) {
        ns._mapsets[map_data.map_sets[m].id] = map_data.map_sets[m];
    }
}());
```

cdlamaps/templates/api/createkey.html

```
{% extends "base.html" %}

{% block title %}Create API Key{% endblock %}

{% block content %}
<h2>Create an API Key</h2>

<div class="intro">
  In order to use the CDLA Maps API you will need to create an API key for your site.
</div>
<form method="post">
<div class="formrow">
  <div class="label{% if form.errors.url %} error{% endif %}">URL</div>{{ form.url }}
  {% if form.errors.url %}
    <div class="label"> </div><div class="error">{{ form.errors.url }}</div>
  {% endif %}
</div>
<div class="formrow">
  <div class="label{% if form.errors.name %} error{% endif %}">Name</div>{{ form.name }}
  {% if form.errors.name %}
    <div class="label"> </div><div class="error">{{ form.errors.name }}</div>
  {% endif %}
</div>
<div class="formrow">
  <div class="label{% if form.errors.email %} error{% endif %}">Email</div>{{ form.email }}
  {% if form.errors.email %}
    <div class="label"> </div><div class="error">{{ form.errors.email }}</div>
  {% endif %}
</div>
<div class="formrow">
  <div class="label"> </div><input type="submit" value="Create Key"/>
</div>
</form>
{% endblock %}
```

cdlamaps/templates/api/displaykey.html

```
{% extends "base.html" %}

{% block title %}Your API Key{% endblock %}

{% block content %}
<h2>Your API Key</h2>
<div class="formrow">
  <div class="label">URL: </div><div class="text">{{ url }}</div>
</div>
<div class="formrow">
  <div class="label">API Key: </div><div class="text">{{ key }}</div>
</div>
{% endblock %}
```

cdlamaps/templates/browse/index.html

```
{% extends "base.html" %}

{% block title %}Browse Maps{% endblock %}

{% block content %}
<h2>Browse Maps</h2>
<div class="intro">
  A large number of maps are available for you to use.
  Browse the collection to find a map that will meet your needs.
</div>
<div class="linkblock">
  <a href="{% url cdlamaps-browse-scope scope_slug="region" %}">Browse by Region</a>
  <div>
    Find maps from one of North Carolina's three geographic regions.
  </div>
</div>
<div class="linkblock">
  <a href="{% url cdlamaps-browse-scope scope_slug="county" %}">Browse by County</a>
  <div>
    Find county-wide maps and maps of cities and towns within that county.
  </div>
</div>
<div class="linkblock">
  <a href="{% url cdlamaps-browse-scope scope_slug="city" %}">Browse by City</a>
  <div>
    Find maps of a specific city or town.
  </div>
</div>
{% endblock %}
```

cdlamaps/templates/browse/scope.html

```
{% extends "base.html" %}

{% block title %}Browse Maps By {{ scope.name }}{% endblock %}

{% block content %}
<h2>Browse Maps By {{ scope.name }}</h2>
{% for extent in scope_info %}
<div class="linkblock">
  <a href="{% url cdlamaps-browse-extent scope_slug=scope.slug, extent_slug=extent.slug %}">
      {{ extent.name }}</a>
  <div>
  {% for escope in extent.map_counts|dictsortreversed:"level" %}
    {% if not forloop.first %},{% endif %}
    {{ escope.name }} maps: {{ escope.map_count }}
  {% endfor %}
  </div>
</div>
{% endfor %}
{% endblock %}
```

cdlamaps/templates/browse/extent.html

```
{% extends "base.html" %}

{% block title %}
  {% ifequal extent.scope.slug "city" %}
    {{ extent.scope.name }} of {{ extent.name }} Maps
  {% else %}
    {{ extent.name }} {{ extent.scope.name }} Maps
  {% endifequal %}
{% endblock %}

{% block content %}
<h2>
  {% ifequal extent.scope.slug "city" %}
    {{ extent.scope.name }} of {{ extent.name }} Maps
  {% else %}
    {{ extent.name }} {{ extent.scope.name }} Maps
  {% endifequal %}
</h2>
{% for scope in maps|dictsortreversed:"level" %}
<h3 class="hr">{{ scope.name }} Maps</h3>
  {% for loc in scope.locations|dictsort:"name" %}
    {%ifnotequal extent.scope.slug scope.slug %}
      <h4{% if not forloop.first %} class="subhr"{% endif %}>{{ loc.name }}</h4>
    {% endifnotequal %}
    {% for map in loc.maps|dictsort:"title" %}
      <div class="linkblock{%ifnotequal extent.scope.slug scope.slug %} indent{% endifnotequal
%}">
        <a href="{% url cdlamaps-browse-map api_id=map.api_id %}">{{ map.title }}</a>
        {% if map.description %}<div>{{ map.description }}</div>{% endif %}
        <div><b>Map ID:</b> {{ map.api_id }}</div>
        {% if map.mapset_set.count %}
        <table class="mapsets">
          <tr>
            <td class="label">Part of Map Set{{ map.mapset_set.count|pluralize }}:</td>
            <td class="list">
            {% for mapset in map.mapset_set.all %}
              <a href="{% url cdlamaps-browse-mapset api_id=mapset.api_id %}">
                {{ mapset.api_id }}</a>{% if not forloop.last %}<br/>{% endif %}
            {% endfor %}
            </td>
          </tr>
        </table>
        {% endif %}
      </div>
    {% endfor %}
  {% endfor %}
{% endfor %}
{% endblock %}
```

cdlamaps/templates/browse/map_detail.html

```
{% extends "base.html" %}

{% block title %}View Map :: {{ map.title }}{% endblock %}

{% block extrahead %}
<script type="text/javascript"
       src="http://maps.google.com/maps?file=api&amp;v=2&amp;sensor=false&amp;key=XXXXX">
</script>
<script type="text/javascript"
        src="{{ host }}{% url cdlamaps-api api_key="YOUR_API_KEY" %}">
</script>
<script type="text/javascript"
        src="{{ host }}{% url cdlamaps-api-mapdata api_key="YOUR_API_KEY" map_ids=map.api_id %}">
</script>
<script type="text/javascript">
  function init_page() {
    if (GBrowserIsCompatible()) {
      var map_options = {size: new GSize(760,400)};
      map = new GMap2(document.getElementById("mapcanvas"),map_options);
      map.setUIToDefault();

      maplayer = new cdla.maps.Map("{{ map.api_id }}");
      zoom = map.getBoundsZoomLevel(layer.bounds());

      if (maplayer.maxZoom() > G_NORMAL_MAP.getMaximumResolution()) {
        G_NORMAL_MAP.getMaximumResolution = function() { return maplayer.maxZoom(); };
      }
      if (maplayer.maxZoom() > G_SATELLITE_MAP.getMaximumResolution()) {
        G_SATELLITE_MAP.getMaximumResolution = function() { return maplayer.maxZoom(); };
      }
      if (maplayer.maxZoom() > G_HYBRID_MAP.getMaximumResolution()) {
        G_HYBRID_MAP.getMaximumResolution = function() { return maplayer.maxZoom(); };
      }
      map.setCenter(maplayer.centroid(),zoom,G_SATELLITE_MAP);
      map.addOverlay(maplayer.layer());
    }
  }
</script>
{% endblock %}

{% block body_attr %}onload="init_page();"  onunload="GUnload();"{% endblock %}

{% block content %}
<h2>{{ map.title }}</h2>
<div id="mapcanvas"></div>
<table class="mapinfo">
  <tr>
    <td class="label">Title</td>
    <td>{{ map.title }}</td>
  </tr>
  {% if map.description %}
  <tr>
    <td class="label">Description</td>
    <td>{{ map.description }}</td>
  </tr>
  {% endif %}
  <tr>
    <td class="label">Map ID</td>
    <td>{{ map.api_id }}</td>
  </tr>
  {% if map.mapset_set.count %}
  <tr>
    <td class="label">Map Set{{ map.mapset_set.count|pluralize }}</td>
    <td>
      {% for mapset in map.mapset_set.all %}
        <a href="{% url cdlamaps-browse-mapset api_id=mapset.api_id %}">{{ mapset.api_id }}</a>
```

```
      {% if not forloop.last %}<br/>{% endif %}
      {% endfor %}
    </td>
  </tr>
  {% endif %}
  <tr>
    <td class="label">Date</td>
    <td>{{ map.display_date }}</td>
  </tr>
  {% for loc in location|dictsort:"level" %}
  <tr>
    <td class="label">{{ loc.scope }}</td>
    <td>{{ loc.name }}</td>
  </tr>
  {% endfor %}
{% endblock %}
```

cdlamaps/templates/browse/mapset_detail.html

```
{% extends "base.html" %}

{% block title %}
    Map Set :: {% firstof mapset.name mapset.api_id %}
{% endblock %}

{% block extrahead %}
<script type="text/javascript"
src="http://maps.google.com/maps?file=api&amp;v=2&amp;sensor=false&amp;key=XXXXX">
</script>
<script type="text/javascript"
        src="{{ host }}{% url cdlamaps-api api_key="YOUR_API_KEY" %}">
</script>
<script type="text/javascript"
      src="{{ host }}{% url cdlamaps-api-mapdata api_key="YOUR_API_KEY" map_ids=mapset.api_id %}">
</script>
<script type="text/javascript">
  function init_page() {

    var map_options = {size: new GSize(760,400)};
    map = new GMap2(document.getElementById("mapcanvas"),map_options);
    map.setUIToDefault();

    maplayerset = new cdla.maps.MapSet(["{{ mapset.api_id }}"]);
    zoom = map.getBoundsZoomLevel(maplayerset.bounds());

    if (maplayerset.maxZoom() > G_NORMAL_MAP.getMaximumResolution()) {
      G_NORMAL_MAP.getMaximumResolution = function() { return maplayerset.maxZoom(); };
    }
    if (maplayerset.maxZoom() > G_SATELLITE_MAP.getMaximumResolution()) {
      G_SATELLITE_MAP.getMaximumResolution = function() {
                                                  return maplayerset.maxZoom(); };
    }
    if (maplayerset.maxZoom() > G_HYBRID_MAP.getMaximumResolution()) {
      G_HYBRID_MAP.getMaximumResolution = function() { return maplayerset.maxZoom(); };
    }
    map.setCenter(maplayerset.centroid(),zoom,G_SATELLITE_MAP);

    maplayers = maplayerset.maps();
    var map_id
    for (map_id in maplayers) {
      map.addOverlay(maplayers[map_id].layer());
    }
  }
```

```
  function toggleMapLayer(map_id, status) {
    if (status) {
      maplayers[map_id].layer().show();
    }
    else {
      maplayers[map_id].layer().hide();
    }
  }
</script>
{% endblock %}

{% block body_attr %}onload="init_page();"{% endblock %}

{% block content %}
{% if mapset.name %}
<h2>{{ mapset.name }}</h2>
{% else %}
<h2>Map Set - {{ mapset.api_id }}</h2>
{% endif %}
<div id="mapcanvas"></div>
<table class="mapinfo">
  {% if mapset.description %}
  <tr>
    <td class="label">Description:</td>
    <td>{{ mapset.description }}</td>
  </tr>
  {% endif %}
  <tr>
    <td class="label">Map Set ID:</td>
    <td>{{ mapset.api_id }}</td>
  </tr>
</table>
{% for scope in maps|dictsortreversed:"level" %}
<h3 class="hr">{{ scope.name }} Maps</h3>
  {% for loc in scope.locations|dictsort:"name" %}
    {%ifnotequal extent.scope.slug scope.slug %}
      <h4{% if not forloop.first %} class="subhr"{% endif %}>{{ loc.name }}</h4>
    {% endifnotequal %}
    {% for map in loc.maps|dictsort:"title" %}
      <div class="linkblock{%ifnotequal extent.scope.slug scope.slug %} indent{% endifnotequal
%}">
        <input type="checkbox" checked="checked" value="{{ map.api_id }}"
               onclick="toggleMapLayer(this.value, this.checked)" />
        <a href="{% url cdlamaps-browse-map api_id=map.api_id %}">{{ map.title }}</a>
        {% if map.description %}<div>{{ map.description }}</div>{% endif %}
        <div><b>Map ID:</b> {{ map.api_id }}</div>
      </div>
    {% endfor %}
  {% endfor %}
{% endfor %}
{% endblock %}
```

cdlamaps/templates/static/

The templates in this directory contain static pages for the “home” index page and API documentation and are not included. See Apendix A to view the API documentation.