



MapQuest Advantage API Developer Guide

.NET Interface Version

**Version 5.2.0
October 2007**

About This Document

Copyrights, Trademarks, and Legal Information

© 2007 MapQuest, Inc. All rights reserved. MapQuest, MapQuest.com, and Advantage are trademarks and/or registered trademarks of MapQuest, Inc. All other marks are the property of their respective owners.

THIS DOCUMENTATION IS PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND. IN NO EVENT SHALL MAPQUEST, ITS PARENT OR ITS LICENSORS BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES OF ANY KIND ARISING FROM ANY ERROR IN THIS DOCUMENTATION, INCLUDING WITHOUT LIMITATION ANY LOSS OR INTERRUPTION OF BUSINESS, PROFITS, USE, OR DATA.

The downloading, exporting, or reexporting of MapQuest products or any underlying information or technology must be in full compliance with all United States and other applicable laws and regulations. Any provision of a MapQuest product or documentation to the U.S. government is with restricted rights as described in the license agreement for that product.

MapQuest, MapQuest.com, the MapQuest logo, Advantage, Site Advantage, Advantage API, and Advantage Enterprise are either registered trademarks or trademarks of MapQuest.com, Inc. Other product and brand names are trademarks of their respective owners.

Table of Contents

1 Introduction	1
What Is Advantage API?	1
About the Developer Guide	1
Technical Support	3
Product Features	3
Advantage API Architecture	4
2 Client Interfaces	7
Client-Server Communication	7
Multiple Servers and Multi-Instance Servers	8
Client Interface Equality	8
Using the Java Interface.	10
Using MapQuest Classes	10
Getting and Setting Object Properties	10
Java Capitalization Conventions	11
Constants	11
Additional Java Help.	12
Security and Connectivity	12
Client-Server Encryption	13
Proxy Server Support	14
Socket Timeout Value	14
Firewalls and Ports	14
Learning Your Server Configuration	15
Getting Server Configuration in XML Format	15
Getting XML Information Without a MapQuest Client	17
Advantage API Connection Information	17
Database Connectivity in Advantage API	18
Standard Database Fields With Advantage API	18
Database APIs With Advantage API	19
Client API Constants	19

3 Geocoding	20
Geocoding Features20
Geocoding Concepts21
Geocoding Rules and Geocode Selectors23
Imperfect Matches23
Postal Code Levels24
Granularity Levels.25
Match Type Constants26
Summary of Granularity Codes and Match Types26
Confidence Levels and Quality Types27
MapQuest Address Objects30
Understanding Geocode Results30
Prepare For Multiple Matches and Imperfect Results32
Geocode Result Examples32
Geocode Example 132
Geocode Example 233
Geocode Example 333
Geocode Example 433
Basic Geocoding34
Advanced Geocoding.35
Defining Your Own Geocoding Rules37
Limiting Matches With A Geocode Selector38
Reverse Geocoding39
Geocoding Constants.39
4 Basic Mapping	41
Mapping Features41
Mapping Concepts.42
Map Data42
Map Scales and Zooming43
Automatic Map Data Selection.44
Display Types and Map Styles44
Advanced Map Styles In Advantage API.45
Sessions Overview.45
What Sessions Contain46

When to Use Server Sessions47
Creating Server Sessions48
Changing Server Sessions.49
Ending Server Sessions.49
Saving Session IDs in Web Applications50
Stateless Maps for Web Applications50
Be Careful When Updating Sessions51
Basic Mapping Classes52
MapState Objects52
Map Data Selection Classes53
Advanced Map Data Selection API53
Getting The Map Pool After Map Data Selection55
Getting Style Pool Names From Style Aliases.55
Basic Mapping API56
Basic Mapping With Server Sessions56
Creating Stateless Maps57
Downloading Map Images58
Customizing Map Format, Resolution, and Aliasing59
Customizing Map Appearance60
Overriding Styles With DTStyle60
Overriding Styles With DTStyleEx62
Overriding Styles With DTFeatureStyleEx63
User Interaction In Maps65
Map Zooming APIs65
Zooming Without Server Sessions65
Map Panning and Centering APIs.65
Panning With Stateless Maps66
Using Map Commands66
Using the BestFit Classes.67
Other Point-and-Click APIs68
Advanced Point-and-Click APIs68
Mapping Constants69

5 Displaying POIs 71

Finding and Creating POI Icons71
--	-----

Basic POI Classes72
Displaying Simple POIs.73
Templates for Using Multiple Tables in a Single Database80
Identifying Map Objects by Coordinates81
Using Points of Interest with Other APIs83
Customizing POI Appearance83
Getting Features From a Map83
Scaling and Centering a Map Around POIs84
Displaying POIs From Search Results84
Using Licensed POI Data With Searches84
Advanced Points of Interest APIs85
Using the Key Value in POIs85
Advanced Feature Types And DTs86
Independent Database Queries87
POI Constants89

6 Map Annotations 91

Drawing API Overview91
Map Annotation Classes92
Map Annotation Drawing Order92
Using Map Annotation Classes93
Drawing a Line Primitive93
Drawing an Ellipse Primitive94
Drawing a Symbol Primitive.96
Drawing a Rectangle Primitive.97
Drawing a Text Primitive98
Drawing a Polygon Primitive	100
Other Map Annotation APIs	101
Map Annotation Constants	102

7 Proximity Searching 105

Proximity Search Features.	105
Search API Overview	106
Defining What Data to Search	108
Database Searches.	108
DTs and Extra Criteria in Database Searches	108

Advanced Database Queries.	109
Feature Collection Searches	109
Map Coverage Searches	110
Setting Display Type Requirements for Searches	110
Using Search Criteria.	111
Radius Search Criteria	111
Rectangle Search Criteria	112
Polygon Search Criteria	113
Searching Near a Path With a Corridor Search.	114
Advanced Corridor Searching	115
Search Result Properties	115
Advanced Searching With Route Matrix API	116
Proximity Searching Constants	117

8 Routing 119

Routing Features	119
Routing API Overview	120
Specifying Locations for Routing	122
Customizing Routing.	122
Route Types	125
Fastest Routes	125
Shortest Routes.	125
Pedestrian Routes.	125
Optimized Routes.	125
SelectDataSetOnly Routes	125
Route Highlight API	126
Route Highlights With Server Sessions	126
Route Highlights With Stateless Maps	127
Routing Errors	128
Basic Routing Steps	129
Other Routing APIs	131
Route Matrix Calculation	131
Advanced Routing Location Specification	133
Specifying a Specific Route Pool	133
Getting Location Data After a Route	133

Routing Constants	134
9 Application Design	138
User Interface Recommendations	138
Geocoding User Interface	138
Mapping User Interface	138
Increasing Contrast Between Maps and POIs	138
Routing User Interface	139
Designing for Performance	139
Debugging Techniques	140
Common Session Mistakes	141
Appendix A: Display Types	143
DT Values for Developer Use	143
DT Values in Licensed Mapping Data	143
Common DT Ranges.	144
Common DTs With Details	144
Appendix B: Standard Icons	150

1 Introduction

What Is Advantage API?

MapQuest™ Advantage API™ is a hosted service that enables developers to integrate maps, driving directions, and other advanced location-based technology into Web applications and desktop applications. The MapQuest-hosted servers provide MapQuest functionality to client applications anywhere on the Internet. Customers manage their custom locations, called Points of Interest (POIs), using a suite of MapQuest Web applications called the Data Manager™.

Java, C++, and .NET development environments are supported through native installable APIs.

Other development environments are able to utilize the service either through the use of XML formatted request/response over HTTP, or the proprietary string-based format over HTTP that underlies the native API clients.

A sample Javascript API implementation of the object model is provided as-is that uses the XML protocol for communication.

Locations displayed on MapQuest maps require location coordinates called latitude and longitude. Advantage API requires coordinates of developer-specified POIs to be pre-calculated and stored before accessing them using the programming interface. The Data Manager calculates latitude and longitude coordinates from custom location data and then stores the information in MapQuest-hosted databases. Refer to the separate Data Manager Users Guide for details.

About the Developer Guide

The Developer Guide is intended for programmers whose applications use Advantage API to geocode addresses, generate and display maps, search locations, and/or calculate driving directions.

We assume that the reader is an experienced computer programmer. Although some tasks are relatively easy with the advanced features of Advantage API, advanced programming skills are necessary to create complete production-ready Web or desktop applications.

The reader should be familiar with programming in one of the supported client languages/interfaces. This document is published in four versions to correspond to the product's four client interfaces:

1. Developer Guide: C++ Interface.
2. Developer Guide: .NET Interface for C#, VB.NET, and ASP.NET.
3. Developer Guide: Java Interface.

IMPORTANT: You are reading the .NET interface version.

If you want a different version of this documentation or if you are not sure whether you have the latest release, check the Technical Support Web site, <https://trc.mapquest.com>.

Developer Guide Goals

- Orient developers with the Advantage API architecture and features.
- Provide overview and details of the MapQuest client interface.
- Provide overviews of each major part of the MapQuest API.
- Provide details and examples of the MapQuest API.
- Provide suggestions for application design and application debugging.
- Complement the API Reference documentation with “how to” information, explanations, suggestions, and warnings.

Using the Developer Guide With the API Reference

The Developer Guide does **not** replace the API Reference documentation provided with the product. You will typically consult both documents during development.

The API Reference contains details such as class inheritance, required parameters, return values, and the appropriate calling conventions for methods for each client interface. The API Reference includes all MapQuest classes, their methods, and their properties.

In contrast, the Developer Guide contains introductory, explanatory, and “how to” information to help developers to understand the APIs. The Developer Guide does not mention all MapQuest classes nor all the details for the ones it discusses. The majority of the Developer Guide is identical for all client interfaces, but you will find interface-specific information throughout the Developer Guide. For instance, you will find constants at the end of each chapter, special instructions in Chapter 2, “Client Interfaces,” and code examples for each interface.

Because the majority of this document is written to apply to all four client interfaces, there may be differences between Developer Guide text and the exact syntax of class names, properties, and methods within your client interface. Carefully read Chapter 2, “Client Interfaces,” for important details on this subject.

If you are ever in doubt of the exact syntax for a MapQuest API in your client interface, always consult the API Reference.

MapQuest provides the API reference in the following versions:

- C++
- .Net
- Java
- HTTP String Protocol

- XML Protocol

To find the API Reference, look within your MapQuest Directory (your installation directory) after product installation. Open the `clients` subdirectory, then open the folder of the same name as your interface (for instance, `java`), and then open the subdirectory called `docs`.

The most recent version of all documentation is on the Technical Support Web site at <https://trc.mapquest.com>.

Technical Support

If you have development questions, please contact MapQuest Technical Support or visit <https://trc.mapquest.com>. When contacting MapQuest, please provide your name, company name, telephone number, and a description of the problem, including the exact wording of error messages.

E-mail es_support@mapquest.com or call toll free at (800) 873-2418. International callers can call (303) 486-4090. Phone hours on weekdays other than holidays are from 9:00 a.m. to 8:00 p.m. (Eastern Standard Time).

Product Features

Advantage API allows developers to make sophisticated applications that can:

- Assign coordinates to a street address (or other location) to uniquely identify the location on the planet. For instance, assign coordinates to the address 123 Main Street in a specific city. This is *geocoding*.
- Display, interact with, and customize high-quality maps. This is *mapping*.
- Generate visual and/or textual navigational directions from one location to another, with optional intermediate destinations. This is *routing*.
- Search for custom locations within a given geographic area, calculate distances between locations, and perform other location searches using dynamic database queries. For example, users can search for the closest restaurants in their neighborhood. This is *proximity searching*.
- Create and manage sessions to improve context from one page view to the next, primarily used by Web applications. This is *session management*.
- Protect sensitive data and restrict client access to the MapQuest server to authorized client IDs, passwords, and IP addresses. This is *authentication* and *encryption*.
- Integrate the features above with other data for additional value. For instance, an application could filter hotel proximity search results based on room availability. Another application could search for businesses near a customer's wireless mobile location, examine purchasing profiles, and then offer targeted shopping recommendations.

Advantage API Architecture

MapQuest Server Engine. This server software provides the core MapQuest technologies mentioned above: geocoding, mapping, routing, and proximity searching.

MapQuest Clients. Software modules that request MapQuest functionality, usually as part of a Web application running on a server. These clients communicate via TCP/IP to the server from anywhere on the Internet.

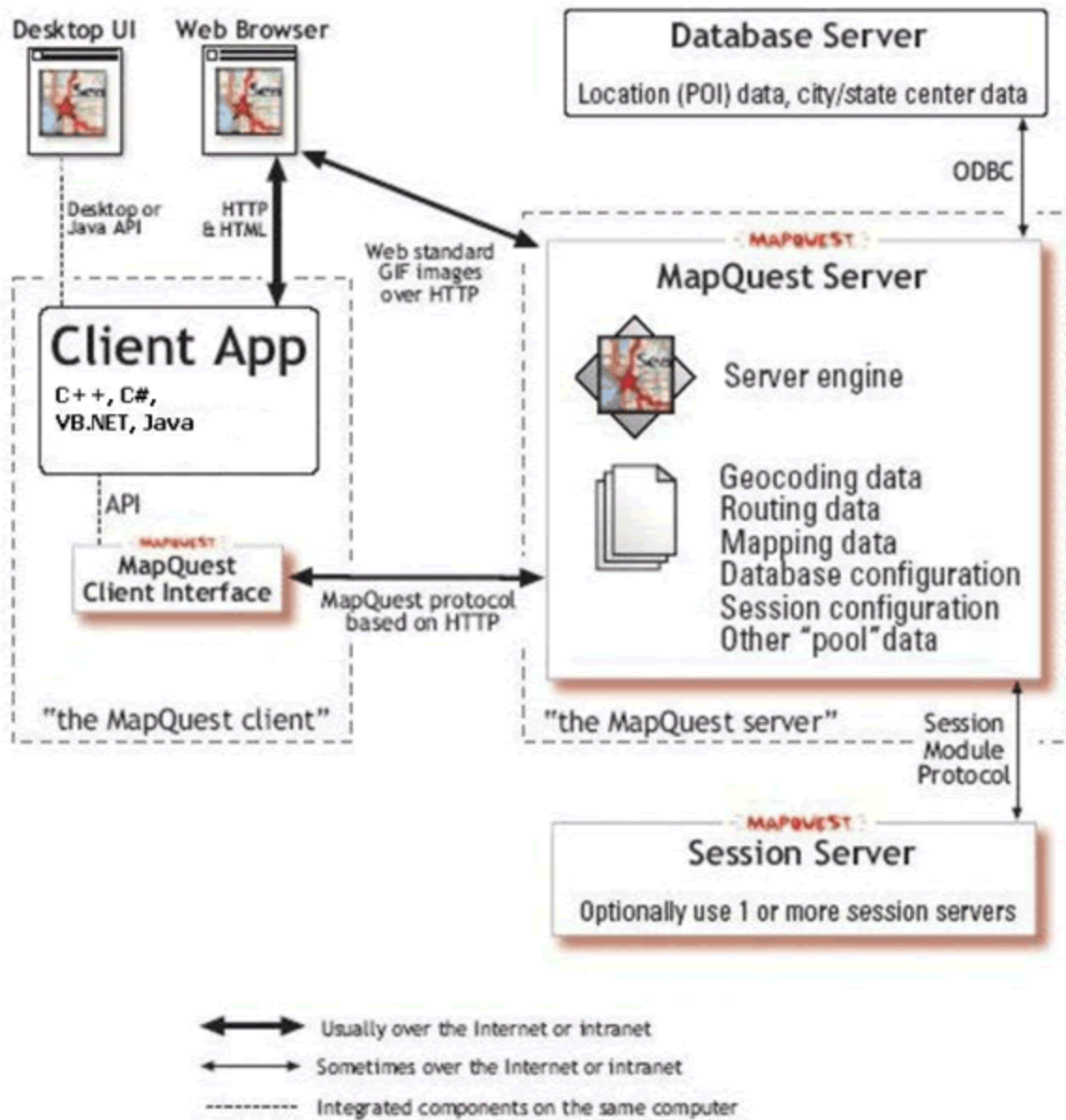
Database Server. A database server stores custom locations for proximity searching and displaying custom locations. Advantage API customers will use the Data Manager to manage their custom locations on this database server.

Session Server. A MapQuest server may employ one or more session servers to help manage special server files that track each user's current map state. The developer API is the same whether or not the MapQuest server uses session servers.

Utility Applications. Advantage API includes utilities to customize certain aspects of the product. These utilities run on Windows 2000 and Windows XP. Unix users must have a Windows-based computer available if they wish to use these tools. For more information about these tools, see each tool's separate documentation.

- **GRFEdit.** This tool prepares custom bitmap images to be bitmap icons by creating special image headers. As an Advantage API customer, if you simply want a standard GIF image centered on its representative location, simply send that GIF format image to Technical Support instead of using this tool.
- **Style File Editor.** This tool creates special style strings for advanced map style editing (line thickness, icons and fonts) in conjunction with the API.
- **GMFDraw.** This tool creates vector icons as an alternative to bitmap or raster icons. Use vector icons for higher map resolutions, as they scale better.

See the architecture diagram on the next page for a visual overview of how all these components interact.



Advantage API Architecture

2 Client Interfaces

Your desktop or Web application will need at least one computer that communicates with the Advantage API server. The application that you write is called a *client application* and it can run on multiple supported platforms.

To compile and link code with MapQuest client programming interfaces, you must have development tools compatible with at least one of the following interfaces:

- C++** Supported on Windows and Linux.
- Java** Supported on specific JDK releases for multiple OS platforms.
- .NET** Supported on Windows with C#, VB.NET, and ASP.NET.

Other HTTP and XML Protocol reference guides are provided to build request strings as the user sees fit. These options allow for the usage of any development environment that MapQuest does not provide a native language API client for.

For detailed compatibility requirements, contact MapQuest Technical Support. After installing the product, you can view the compiler requirements within the `readme.txt` file in the `clients` directory for each interface.

This document does **not** duplicate MapQuest client installation information. If you have not installed the client files yet, please refer to the Advantage API User Guide. If you have any questions during the installation process, please contact MapQuest Technical Support before proceeding.

Client-Server Communication

Each client connects to the server using a proprietary protocol based on HTTP (Hypertext Transfer Protocol) over network connections using the TCP/IP networking protocol.

The core MapQuest API functions translate API requests into a client request to the server, which may or may not be over a network. After the client interface receives a response from the server, the client interface returns the information back to your application using the MapQuest API.

Please refer to the diagram “Advantage API Architecture” on page 5 to view the system architecture.

Multiple Servers and Multi-Instance Servers

In this manual, most discussion of client-server communication refers only to one “server.” However, Advantage API customers will use multiple servers depending on the type of data. For instance, one server will be the server for geocoding, another for routing, and so on.

When preparing for a Advantage API server connection, your code will create a *client object* to manage the client-server connection. You will set properties in the client object to tell it the server’s name (the domain

name or IP address), the TCP/IP port (always 80 in Advantage API), and the server path (always "mq" in Advantage API).

To communicate with a MapQuest server, you will create special objects called *client objects*. MapQuest **strongly recommends** that you design your code such that you create one client object for each server or type of data. Create one client object for geocoding, one for mapping, one for routing, and one for proximity searching. You must then design your code so server requests use the appropriate client object.

Client Interface Equality

The four client interfaces for Advantage API are almost identical in design and functionality. Anything possible in one interface is possible in other client interfaces. If you are familiar with one MapQuest client interface and wish to work with another supported programming environment, you can learn the new client interface easily. For instance, if you are experienced with both C++ and Java and you learn the Advantage API C++ client interface, it will be easy to use the Advantage API Java client interface.

If you switch from one client interface to another, remember to download the appropriate Developer Guide version that provides information and code examples for your programming interface.

IMPORTANT: You are reading the .NET version.

If you want a different version of this documentation or if you are not sure whether you have the latest release, check the Technical Support Web site, <https://trc.mapquest.com>.

To illustrate similarities and differences among the client interfaces, Table 1 shows how you might specify a street address within each client interface.

Table 1, Comparison of Client Interfaces

Interface	Example
C++	<pre>OriginAddress.SetStreet("123 Main"); OriginAddress.SetCity("Pittsburgh"); OriginAddress.SetState("PA"); OriginAddress.SetPostalCode("15215"); OriginAddress.SetCountry("US");</pre>

Table 1, Comparison of Client Interfaces (*continued*)

Interface	Example
C#	<pre>OriginAddress.Street = "123 Main "; OriginAddress.City = "Pittsburgh"; OriginAddress.State = "PA"; OriginAddress.PostalCode = "15215"; OriginAddress.Country = "US";</pre>
Java	<pre>OriginAddress.setStreet("123 Main"); OriginAddress.setCity("Pittsburgh"); OriginAddress.setState("PA"); OriginAddress.setPostalCode("15215"); OriginAddress.setCountry("US");</pre>

For sample code for each client interface, see the `samples` directory within your *MapQuest Directory*, your local installation directory. The latest sample code is always on the Technical Support Web site: <https://trc.mapquest.com>.

This documentation refers to your *client directory* as the directory containing your MapQuest client interface files created by the installer. All client directories are in subdirectories of the `clients` subdirectory of your MapQuest Directory. For instance, if you are using the C++ client interface and your MapQuest Directory on UNIX is `/mq/`, your client directory would be `/mq/clients/c++`.

For detailed API Reference, see each client's API Reference, which includes object descriptions, object properties, object methods, and object inheritance information. You will find the documentation in your client directory's `docs` subdirectory. However, the latest documentation in all formats is always on the MapQuest Technical Support Web site: <https://trc.mapquest.com>.

Although the design of each interface is similar, the API usage is different in some important ways. The next section discusses details of **your** client interface.

Using the .NET Interface

Microsoft's .NET is a software architecture that allows software components made by different software vendors to be combined with and used with a variety of applications.

Advantage API supports .NET clients hosted on specific versions of the Windows OS, Microsoft Visual Basic.NET (VB.NET) and Microsoft Visual C# (C#) for desktop application development, and server-side Microsoft Active Server Pages .NET(ASP.NET) for Web application development. For more detailed requirements for .NET development environments, please contact MapQuest Customer Support.

Programming Code Styles in .NET

The programming code for different .NET implementations varies, even between Microsoft's own ASP.NET, C#, and VB.NET environments. In this documentation, some .NET code snippets are in C# and some are in VB.NET. There is a third style, ASP.NET, which is only used once in this manual to illustrate class usage in the following section.

The first line of multi-line code examples will indicate whether it is a VB.NET example or a C# example.

Refer to the sample code projects, which contains complete projects available for VB.NET, C#, and ASP.NET development environments using the .NET interface.

Using MapQuest Classes

When you use MapQuest class names in .NET interfaces, you reference them as a part of the MapQuest .NET interface. This is implemented differently among .NET interfaces.

Within VB.NET, you would create an Address object by creating an object of class Address, with code like:

```
Dim myAddress As New MQClientInterface.Address
```

Within C#, you would create an Address object using code like:

```
MQClientInterface.Address myAddress = new  
MQClientInterface.Address();
```

Within ASP.NET using script language VB, you would create an Address object using code like:

```
Dim myAddress As New MQClientInterface.Address
```

For all supported .NET environments, you must add a reference to the mapquest.dll. Default location is C:\mq\clients\dotnet\mapquest.dll.

Within your ASP.NET code, import Advantage API classes using the following command:

```
<%@ Import namespace="MQClientInterface" %>
```

Within your C# code, import Advantage API classes using the following command:

```
using MQClientInterface;
```

For all supported .NET environments, consult the API Reference documentation for more information on a class using the formal .NET class interface name, which includes the prefix "MQClientInterface".

For instance, for more information on the Address class, open the API Reference HTML documentation to the class listed as MQClientInterface::Address.

Getting and Setting Object Properties

Getting and setting object properties in the .NET interface give the impression of direct manipulation of the properties rather than "Get" or "Set" methods. However, this is not the case. The C# accessor language features are used to set and get the property. If this manual mentions setting an Address object's Street property, you will generally use code like:

```
myAddress.Street = "123 Main"
```

When consulting the API reference for more information, look up the object (for instance, the Address class would be MQClientInterface::Address) and look for the property within that object (for instance, Street).

Client Objects

To use any MapQuest APIs, your application will instantiate a *client object*, which is an object of class Exec. The primary MapQuest APIs are methods of the client object. Set the client object's basic connection information: its domain name (or IP address), its TCP/IP port, and path name. You would typically also use authentication methods ClientId and Password, which are discussed in more detail in the next section.

Here is some example C# code to prepare a client object:

```
MQClientInterface.Exec Client = new MQClientInterface.Exec();
Client.ServerName = "localhost";
Client.ServerPath = "mq";
Client.ServerPort = 9810;
Client.ClientId   = "99999";      ' see next section...
Client.Password   = "password";  ' see next section...
```

Here is some example VB.NET code to prepare a client object:

```
Dim Client As New MQClientInterface.Exec
Client.ServerName = "localhost"
Client.ServerPath = "mq"
Client.ServerPort = 9810
Client.ClientId   = "99999"      ' see next section...
Client.Password   = "password"  ' see next section...
```

MapQuest classes are automatically initialized when created. If you want to reuse objects and reinitialize them with default values, you can do so using their `Init` method. It is unnecessary to call `Init` unless you are re-using an object.

Constants

MapQuest constants in the .NET interface are referenced from classes with the same name as their type. For instance, the distance unit `Miles` can be used in your code as `MQClientInterface.DistanceUnits.MILES`.

The end of each chapter contains a list of constants discussed in that chapter with the exact syntax required by the .NET interface. To use a constant mentioned in this document, always consult the constants at the end of the appropriate chapter and/or consult the API Reference.

Additional .NET Help

You can learn more about using .NET technologies on Microsoft's Web site at <http://msdn.microsoft.com>. Also, read and compare the MapQuest sample code for VB.NET, C#, and ASP.NET environments.

Security and Connectivity

Client-Server Authentication

If a server is accessible from the Internet, it is vulnerable to unauthorized use. You can use the authentication features of Advantage API to decrease the risk of unauthorized use.

To prevent unauthorized applications from using the server, Advantage API provides several client-server authentication methods. At a minimum, the server can be configured to require customizable identifiers called client IDs. Your client ID strings might be numbers that designate a computer or human-readable names.

There are two additional options for client-server authentication that require and extend the basic client ID authentication:

- **Password authentication.** Client applications must provide a password for each server request. The server can assign each client ID a different password.
- **IP authentication.** Client applications must initiate network requests from specific Internet Protocol (IP) addresses or IP subnets.

Both password authentication and IP authentication can be used simultaneously with a client ID. If the trust levels vary among clients, the server can specify different authentication requirements for each client ID.

In Advantage API, client IDs are required but password and IP authentication are optional. To change authentication settings for your account, please contact Technical Support.

Configuring Authentication In Your Application

To set the client ID for your application, set the client object property `ClientId` before sending any requests to the server. To set the client ID password, set the `Password` property of the client object before sending any requests to the server.

IP-based authentication is configured entirely on the server. If your application is behind a firewall, your IP address to the public Internet may be different from the IP address configured on the physical computer. If in doubt of your network configuration, consult your system administrator.

For the strongest IP authentication, assign static IP addresses to clients and authorize only the exact IP addresses necessary. If it is impossible for you to use a small set of static IP addresses, the server can authorize entire subnets of IP addresses, but that is less secure than specifying specific IP addresses.

These authentication methods only authenticate the MapQuest client developer code with the Advantage API server. These APIs do not provide a general purpose user authentication system for applications built on top of Advantage API.

For more information about how clients and servers interact, see the figure “Advantage API Architecture” on page 5.

Client-Server Encryption

MapQuest clients communicate with the server using a proprietary protocol on top of the World Wide Web unencrypted protocol called HTTP. When clients make requests to the server and receive responses, data streams could contain confidential corporate or customer information. Advantage API provides optional data stream encryption to reduce the chance of data interception.

You can set encryption in your application by setting the client object property `Encryption` to the value: `MQClientInterface.EncryptionType.RANDOM_SHUFFLE`.

After setting this property, server requests are encrypted and the server will respond to each encrypted request with an encrypted response. Encryption is permitted on a per-request basis, so you can turn it off before or after making a request. You can change this property on the client without having to reinitialize the client or create a new session.

Server responses for certain types of URLs are never encrypted. Specifically, Web applications typically use *image URLs* that will always return unencrypted raw image data from an HTTP request. This is because the image will be retrieved and displayed by users' Web browsers rather than automatically decrypted by MapQuest libraries. If client-server encryption is set when **generating** an image URL, the image's **URL** will be encrypted even though the response will be unencrypted. Image URLs are discussed in detail in Chapter 4, “Basic Mapping.”

A similar exception for encryption exists for the client object method `GetServerInfoURL`. For details, see “Getting XML Information Without a MapQuest Client” on page 15.

Proxy Server Support

Advantage API supports HTTP proxies, which are required in some networks to connect to the Internet through firewalls. To use an HTTP proxy, set the client object properties `ProxyServerName` and `ProxyServerPort` properties. To use HTTP proxy “basic authentication,” also set the client object properties `ProxyUser` and `ProxyPassword`.

Socket Timeout Value

If you experience network or connection problems, you can change the network connection timeout value. Set the client object property `SocketTimeout` to the maximum seconds of inactive communication after which the network socket connection will be terminated.

The default value works for most applications, but you can change this property as appropriate for your application and network.

Firewalls and Ports

If your application or your users experience consistent network or connection problems through a corporate or home Internet firewall, the firewall might block required TCP/IP ports.

If a firewall blocks traffic for the port number used by the image URL (or any other client-based request to the server), the HTTP request will fail.

You cannot change the port number (80) used by Advantage API. If traffic on that port still appears to be blocked from a certain network, that network may require an HTTP proxy. Ask the network administrator for proxy details. If the client application runs behind this firewall, refer to “Proxy Server Support” on page 13 for MapQuest API configuration.

Learning Your Server Configuration

Getting Server Configuration in XML Format

Developers typically interact with the server data only by the names of configuration pools. A *pool* is a server resource of one type. There are pools for mapping, routing, geocoding, databases, and other resources. For example, a server *map pool* contains licensed map data that developers use to generate high-quality maps using the MapQuest API.

In many cases, there will be more than one pool of each type. For instance, one map pool might contain map data intended for street-level maps in large cities; another map pool might contain map data used when users “zoom out” to view the entire country. To refer to one of several pools of one type, you need to know the names of pools that you are interested in.

You can view the server’s configuration, including pool names, using the client object method `ServerInfo`. This method returns the installed coverage data (including pool names) if you pass 0 for its argument. If you pass 1 for its argument, it returns the data selector configuration, including geocode data selectors, map data selectors, and route data selectors.

`ServerInfo` returns data as a string formatted in XML, the Extensible Markup Language. XML is a data description language with flexible and adaptable information layout. It looks similar to HTML, but its format and tags are significantly more customizable. You are responsible for parsing the XML text. Much of the terminology used in the XML data is explained throughout the Developer Guide, but if you still have questions, please contact Technical Support.

Check coverage information and server pool names by calling `ServerInfo` with the argument 0 (zero). Below is some example coverage configuration data from a Advantage API server with mapping data. Note in particular the name properties of XML objects, for instance any line that contains the string “name=”.

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<coverageInfo>
  <coverage name="NoData" search="1" map="1"
    geocode="0" route="0">
<scaleRange lo="1" hi="600000000" />
<projection format="MQGL">Q 37500000 0 37500000</projection>
<latLongBoundingBox ul_lat="90.000000" ul_long="-180.000000"
  lr_lat="-90.000000" lr_long="180.000000" />
<detailLevel>
  <scaleRange lo="1" hi=" 600000000" />
  <vendorCode>4</vendorCode>
  <dataVersion>1.0</dataVersion>
  <copyright>©2003 MapQuest.com, Inc.</copyright>
</detailLevel>
<style name="NoData"></style>
<style name="gdt"></style>
<style name="navt"></style>
</coverage>
<coverage name="gdt" search="1" map="1"
  geocode="1"route="1">
  <scaleRange lo="3000" hi="200000" />
  <projection format="MQGL">Q 37500000 0 37500000</projection>
  <latLongBoundingBox ul_lat="42.285627" ul_long="-80.528709"
```

```
    lr_lat="39.718842" lr_long="-74.679802" />
<detailLevel>
  <scaleRange lo="150000" hi=" 200000" />
  <vendorCode>2</vendorCode>
  <dataVersion>2002Q3</dataVersion>
  <copyright>© 2002 MapQuest.; © 2005 Tele Atlas, Inc.</copyright>
</detailLevel>
<detailLevel>
  <scaleRange lo="3000" hi=" 149999" />
  <vendorCode>2</vendorCode>
  <dataVersion>2002Q3
</dataVersion>
<copyright>©2002 MapQuest.com.; ©2005 Tele Atlas, Inc.</copyright>
</detailLevel>
<style name="NoData"></style>
<style name="gdt"></style>
<style name="navt"></style>
</coverage>
</coverageInfo>
```

Just like other client object requests, `ServerInfo` observes client authentication options and encryption settings.

Getting XML Information Without a MapQuest Client

The client object method `GetServerInfoURL` returns a URL to the MapQuest server that provides server information in the same XML format returned by `ServerInfo` and takes the same argument values.

The URL you get from this method is useful during development to confirm server settings from a plain Web browser, even from computers that do **not** contain the MapQuest client libraries. Of course, other software objects capable of HTTP connections can use this URL to confirm server availability and configuration.

Just like other client object requests, `GetServerInfoURL` observes client authentication options. However, the server response from `GetServerInfoURL` will ignore data stream encryption settings; the results from HTTP requests using that URL are always unencrypted.

When the server receives a request using that URL, it's considered a request from the client that generated the URL using `GetServerInfoURL`. The URL embeds the client ID (if present), password (if present), and server connection settings into the URL. If the server requires IP address authentication for that client ID, then all requests that use that URL must initiate from the appropriate IP address or subnet.

Advantage API Connection Information

You will need the following server connection information to use Advantage API. Create a different *client objects* (instances of the MapQuest class `Exec`) for each type of server data: geocoding, mapping, routing, and database access.

Table 2, Advantage API Connection Information

Data Type	Server	Port	Path	Default Selector
Geocoding	<code>geocode.access.mapquest.com</code>	80	<code>mq</code>	<code>mqgauto</code>
Mapping	<code>map.access.mapquest.com</code>	80	<code>mq</code>	<code>mqmauto</code>
Routing	<code>route.access.mapquest.com</code>	80	<code>mq</code>	<code>mqrauto</code>
Searching	<code>spatial.access.mapquest.com</code>	80	<code>mq</code>	<i>n/a</i>

The following information is provided in context in the rest of the Developer Guide, but is duplicated here for reference:

- You can display built-in map styles by supplying the name of *style aliases*. The available style aliases in Advantage API are: `bw`, `classic`, `default`, `style5`, and `european`.
- To support *reverse geocoding* with optional city and state lookup in the USA, use the geocode pool `us_postal`. For other countries, contact Technical Support.

Database Connectivity in Advantage API

In order to use your MapQuest-hosted databases, you must reference the data by a string known as a *database pool name*. Once your POI data has been pushed to production, you must specify the database pool name to access it.

You can use the test database pool name `"MQA.test"` for initial testing if desired. To use your own location data, use the following convention to identify your database pool name:

```
"MQA.MQ_YOURCLIENTID"
```

For instance, a customer with client ID 99999 would use the database pool name:

```
"MQA.MQ_99999"
```

As an example, suppose you wanted to use the MapQuest sample code project `SearchIt` with your data. In the sample, change the database pool name `"MQA.test"` to `"MQA.MQ_99999"`.

If you specified a table name in Data Manager, the database pool name must incorporate the table name appropriately. Append an underscore and then the table name to the database pool name as described above.

For instance, instead of the database pool name "MQA.MQ_99999", use the string "MQA.MQ_99999_mytablename".

For more details about uploading your files with Data Manager, consult the Data Manager User Guide.

Standard Database Fields With Advantage API

The following fields are defined in databases created on the servers:

S, T, I, Lat, Lng, Ic, N, RECORDID, address, city, county, state, postal, country, matchcode, ambiguous.

I = An automatically-generated incremented field. This is the *key field*.

RECORDID = A customer-defined ID field for the location.

N = Name field.

Lat = Latitude field.

Lng = Longitude field.

T = Display type (DT) field.

S = Spatial index (spatial ID) field.

Database APIs With Advantage API

Here are database-related APIs that you can use with location data on MapQuest-hosted databases.

- Use the robust proximity searching API. For searching APIs, refer to Chapter 7, “Proximity Searching.” For searching, always use a client object connected to the database server `spatial.access.mapquest.com`. You would typically display the results as simple Points of Interest (POI), as described in “Displaying Simple POIs” on page 68.
- Perform simple database lookups to extract MapQuest data or non-MapQuest data from hosted databases. Refer to “Independent Database Queries” on page 76.
- Dynamically display locations on maps. This is generally **not recommended** for performance reasons. Instead, perform an appropriate proximity search and then display the results as simple POIs using the APIs described in “Displaying Simple POIs” on page 68.

Client API Constants

NET developers should use the constants below exactly as shown.

Client Encryption Constants

Set the client object property `Encryption` to one of these constants:

`MQClientInterface.EncryptionType.NONE`

`MQClientInterface.EncryptionType.RANDOM_SHUFFLE`

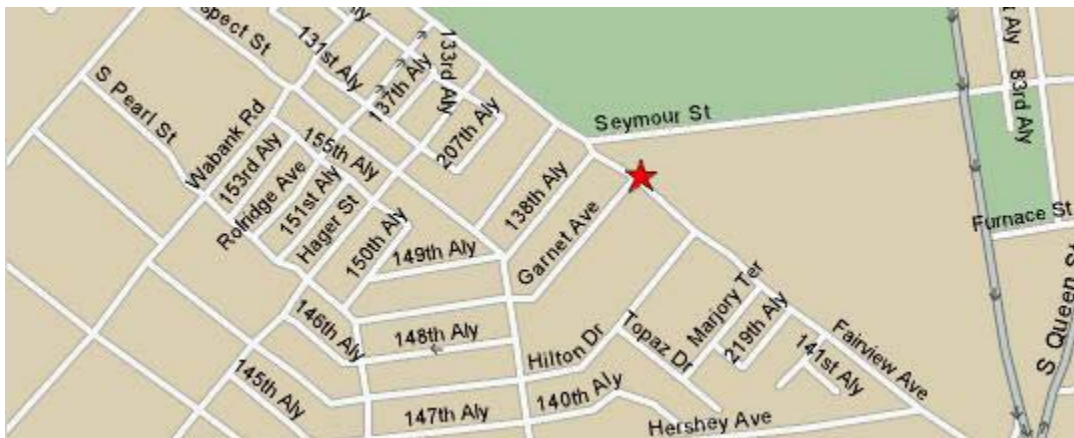
3 Geocoding

Geocoding an address (or another location) assigns location coordinates called latitude and longitude that uniquely identify a location on the planet. The server can calculate coordinates for street addresses, intersections, and the center points of regions like cities, states, and postal codes. The server's geocoding engine calculates these coordinates using geocoding data licensed from MapQuest.

Geocoding an address is required before plotting the address on a map, using it as an anchor point for searching, or generating driving directions with it.

This chapter discusses geocoding and introduces important concepts and terminology regarding MapQuest APIs for locations, coordinates, and addresses.

A Geocoded Location Displayed with an Icon



Geocoding Features

Applications using Advantage API can calculate the latitude and longitude of:

- Street addresses and intersections, the highest accuracy geocoding methods.
- Street blocks, including the nearest block to an invalid house number.
- Postal codes, including ZIP, ZIP+2, and ZIP+4 codes.
- City centers.
- US state and Canadian province centers.
- Country centers.
- Centers of other administrative areas that are used internationally.

If you geocode a location and exact coordinates cannot be found, the geocoding engine can estimate a location based on other information. For example, the server may not know the location of 123 Fake Street in Philadelphia, PA, 19146, but it might be able to provide coordinates for the center of the postal code 19146. If the postal code is not found, the geocoding engine might be able to provide coordinates for the center of Philadelphia, PA.

When the geocoding engine returns the center of a geographic area, the value is a *centroid*, a calculated “weighted center point” that is not necessarily within the most populous area or the commercial downtown area.

When a user wants to display a street address on a map, there are two steps for the Advantage API developer. First, use geocoding APIs to convert the street address into coordinates. Second, use mapping APIs to display a map, usually highlighting the location with an icon and/or text label.

Although the results of geocoding are often used to display a map centered on that location or to calculate driving directions using the routing API, you may use the coordinates in any way you choose.

The geocoding accuracy of valid street addresses depends strongly on the country, the region of the country, and the geocoding data available to the server.

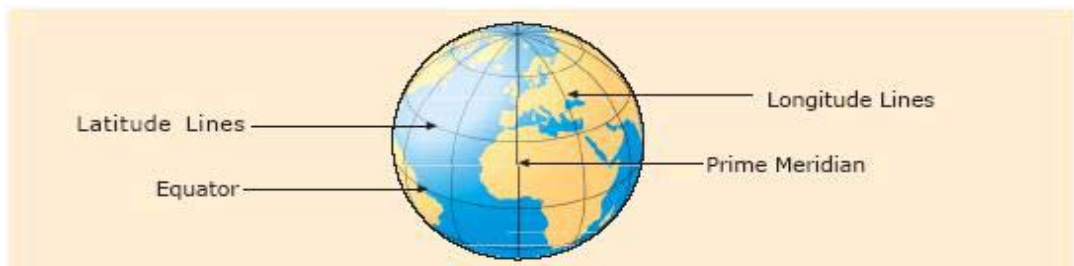
The server-based geocoding rules should work for the vast majority of developers. If you are sure that you need custom geocoding rules, you can override the default geocode selector on the server by defining your own *geocoding rules*. This approach is rarely used and not recommended for most developers.

Geocoding Concepts

Latitude and Longitude

Let us look at how the product represents any location on the planet using latitude and longitude coordinate numbers. *Latitude* represents north and south position on the planet; *longitude* represents east and west position on the planet centered around an arbitrary dividing line called the Prime Meridian. See the figure “Latitude and Longitude Lines Around Planet Earth” on page 20 for each of these elements.

Latitude and Longitude
Lines Around Planet Earth



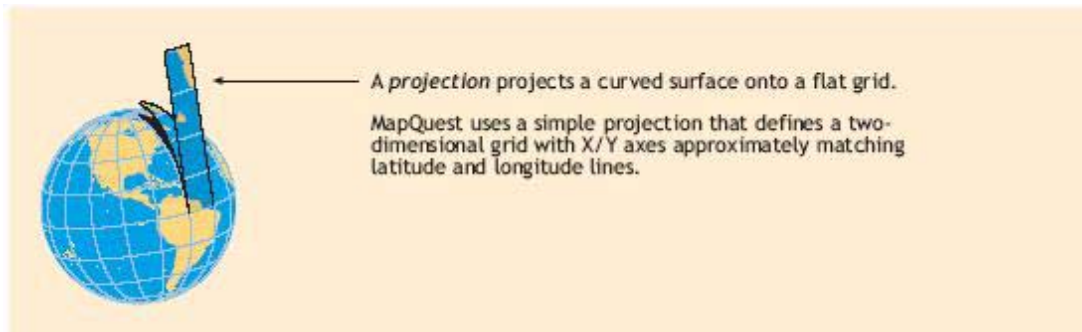
Readers familiar with Global Positioning System (GPS) receivers will recognize latitude and longitude coordinates as the standard coordinates for those devices.

Traditionally, there are two common ways to express latitude and longitude values:

1. **In degrees, minutes, and seconds.** A degree is 1/360 of a circle, a minute is 1/60th of a degree, and a second is 1/60th of a minute.
2. **In fractions of a degree.** For example, the latitude 40 degrees and 30 minutes is the same as 40.5 degrees, since one degree contains 60 minutes. You can also use precise decimal values expressed to several decimal places. MapQuest APIs uses this fractional degrees method.

Latitude and longitude lines are curved lines on a sphere (the Earth) but Advantage API treats the world as a flat surface to display maps. It does this using what's called a *projection* which stretches (and distorts) the curved surface of the Earth onto a flat surface. Advantage API uses a simple projection that treats latitude and longitude lines as axes on a grid. Refer to the figure "Using Latitude and Longitude for Projections" for an illustration of this effect.

Using Latitude and
Longitude for Projections

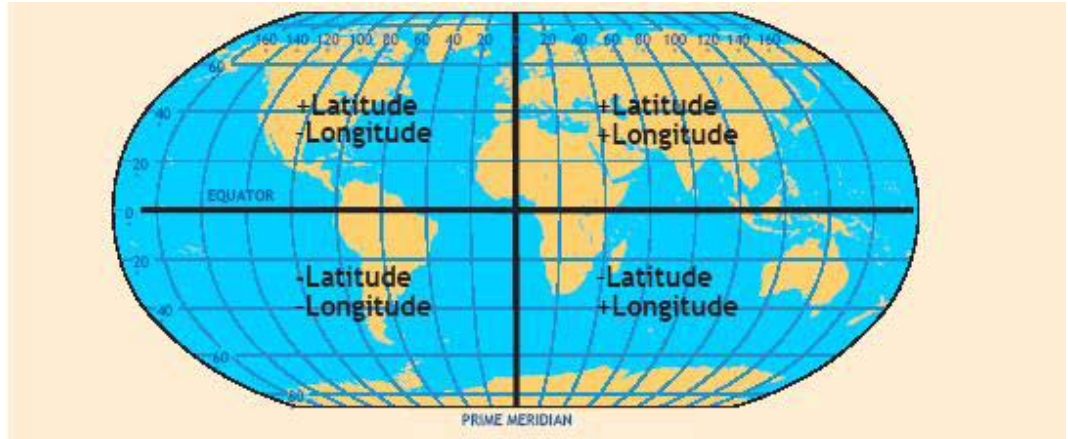


The center, at coordinates (0,0), represents the intersection of the equator and the Prime Meridian. See "The Quadrants of the Projected Coordinate System" on page 22 for an illustration.

As you can see in that figure, all of North America is in the upper left quadrant with positive latitude numbers and negative longitude numbers. For instance, Lancaster, Pennsylvania has a latitude of 40°2'16" North and 76°18'21" West. That becomes decimal values positive latitude 40.0377 and negative longitude -76.3058.

Developers with advanced cartographic knowledge should note that Advantage API uses multiple types of projections; API functions that perform pixel to latitude/longitude translation use the projection data on the server. To correct for the non-spherical shape of the planet, Advantage API uses datum NAD83 and the nearly-identical datum WGS84.

The Quadrants of the
Projected Coordinate
System



Geocoding Rules and Geocode Selectors

When you geocode a location, you should usually use the defaults in the form of server-based *geocode selectors* that are referenced by name. Advanced developers can provide their own *geocoding rules* that select different data sets and/or handle imperfect matches differently.

In either case, a geocoding request causes the server to run a series of geocoding rules that describe which geocoding data on the server to use and what type of matches to look for in priority order.

When the server finishes the request it will return no results, one location match, or multiple location matches if it was an ambiguous request. Ambiguous requests are common and should be handled by your application user interface. For instance, a user might enter the address 123 Main Street which might not match any addresses but offer the similar matches 123 South Main Street and 123 North Main Street. There are many other types of geocoding ambiguities and your application should typically let users choose from multiple choices.

Imperfect Matches

The MapQuest geocoding engine is very flexible and can geocode addresses even with limited or partially incorrect location information. For example, if the client provided a street but did not provide a house number, the server could calculate the total block range for the entire street. The server would provide coordinates that represent the entire block range, usually near the center of the block range.

The geocoding engine can also provide centers of geographic areas, such as cities, states, or postal codes. This is useful when the user did not provide a street address, or it was provided but not found.

The geocoding engine can also provide useful results when the client requests a location with slightly different information than the server's geocoding data. For instance, if a user provides "123 N. Main," the address can match the full street address "123 North Main Street." When allowing non-exact matches,

some geocoding data supports correcting directionals (North versus South), road types (Street versus Road), and sound-alike matches for street names, cities, states, and other administrative areas.

When the preferred information cannot be found, geocoding rules specify how to “fall back” to other useful results. For instance, should the server return multiple matches from an ambiguous request; or should the server estimate the location based on the nearest block to the address, or return the center of the likely postal code? In general, you should use the default geocoding rules defined in geocode selectors and not define your own geocoding rules.

Each location match includes a result code that includes confidence information about each *address element* (street name and number, city and state, and postal code). Use this information to decide whether the results are useful and/or how to use multiple matches within your user interface.

Postal Code Levels

Because there are sometimes multiple types of postal codes within one country, Advantage API supports four *postal code levels* for each region, although typically one or more are unused in any particular region. Level 1 represents the largest (least accurate) area and level 4 the smallest (most accurate) area. The geocoding engine attempts to assign a postal code of the highest accuracy (smallest area) to an address when it seems like a valid mailing address.

United States data contains the following types of postal codes:

- **ZIP+4 codes.** These are the most accurate postal codes and place valid USA mailing addresses on the correct block. In some cases, ZIP+4 codes represent individual high-volume mail recipients. ZIP+4 codes use postal code level 3.
- **ZIP+2 codes.** In urban areas, multiple blocks. In rural areas, usually larger geographic areas. ZIP+2 codes use postal code level 2.
- **ZIP codes.** Original 5-digit ZIP codes represent many different geographic sizes and represent less accurate geocoding than ZIP+4 or ZIP+2 codes. ZIP codes use postal code level 1.

Some data sets contain information about each postal code’s weighted centers. You can configure your geocoding rules to use this information to estimate address locations when it cannot find more accurate locations like exact street addresses.

Where other international geocoding data is available, MapQuest supports region-specific postal code variants. Of particular note is support for Canadian postal codes (6 digits, plus an optional space character delimiter) using postal code level 3. For other international data, contact Technical Support about supported postal code levels.

Granularity Levels

A *granularity level* is a qualitative measure of location accuracy of a geocoded address or search result. You generally will use these values to interpret the results of geocoding. For instance, to determine whether the server found the exact street address or whether it failed and could only find the center of the city in question.

For example, the highest granularity level is a numbered address, such as “123 Main Street” in a specific city. A low granularity level example would be a country.

There are five primary granularity levels, each with a corresponding *primary granularity code*. These granularity levels indicate the type of match desired or returned. There are granularity levels for exact location, intersection, block, administrative area, and postal code.

In addition to each level’s granularity code, there are *granularity subcodes* that further define the granularity. The granularity code and subcode are collectively referred to as *granularity codes*, represented with a letter for the primary granularity code and a number for the subcode (for instance, L1).

The following are the granularity levels and subcodes:

Location granularity. This is a match at the highest granularity level and the smallest geographic area. This granularity often corresponds to an exact street address, but also includes named locations and exact locations that are not street addresses. Depending on the geocoding data, street addresses with this granularity may have been interpolated on the matched block.

Intersection granularity. This is also a match at the highest granularity level. This granularity level indicates the intersection of two streets. Specify intersections with the @ symbol between street names.

Block granularity. One street block with no intersecting streets. This is a lower granularity than location or intersection matches.

- **Block subcode.** A street block matching the input street address but not the house number. Typically multiple matches are returned.
- **Nearest numbered block subcode.** The address range for this block is the closest to the provided address. This subcode can only be generated if the house number is input and the server finds matching street blocks with valid address ranges. Typically only one match is returned.
- **Representative block subcode.** The street block generally near the geographic center of all matching blocks. Typically only one match is returned.

Administrative area granularity. Administrative areas are the general term for governmental domains such as cities, counties, states, and countries. There are seven subcodes defined and their meanings are region-specific. Subcode 1 is the lowest granularity and typically represents a country center; subcode 7 is the highest granularity. Administrative area granularity is a lower granularity than all location, intersection, and block granularities. Typically, some subcodes are unused in each geocoding data set.

Postal code granularity. Postal code subcodes indicate the postal code’s likely geographic size, which often relates to the number of digits in the postal code. There are four subcodes for *postal code levels*. Subcode 1 represents the largest (least accurate) area and subcode 4 the smallest (most accurate) area. All postal code matches are a lower granularity than location, intersection, or block matches, but **not** necessarily higher or lower granularity than administrative area matches. Typically, some postal code levels are unused in each data set.

Match Type Constants

Advanced developers who override the default geocoding rules will use *match type constants* that correspond to granularity levels. Each set of granularity codes has a corresponding match type constant.

Summary of Granularity Codes and Match Types

See Table 3 for details of the granularity codes and match types. Within each primary granularity code, the table lists subcodes in granularity order from highest granularity to lowest granularity. However, groupings of primary granularity codes are **not** strictly ordered by granularity – for instance, both administrative areas and postal codes contain both high granularity and low granularity subcodes.

Table 3, Granularity Codes and Match Types

Granularity	Codes	Match Type	Meaning
Location	L1	LOC	A specific street address location.
Intersection	I1	INTR	An intersection of two or more streets.
Single Block	B1	BLOCK	The center of a single street block. House number ranges are returned if available.
Nearest Numbered Block	B3	NEARBLK	The center of a single street block whose numbered range is nearest to the input number. House number range is returned.
Representative Block	B2	REPBLK	The center of a single street block, which is located closest to the geographic center of all matching street blocks. No house number range is returned.
Admin Area 7	A7	AA7	Admin area, smallest. Unused in USA.
Admin Area 6	A6	AA6	Admin area. Unused in USA.
Admin Area 5	A5	AA5	Admin area. For USA, a city.
Admin Area 4	A4	AA4	Admin area. For USA, a county.
Admin Area 3	A3	AA3	Admin area. For USA, a state.
Admin Area 2	A2	AA2	Admin area. Unused in USA.

Table 3, Granularity Codes and Match Types (*continued*)

Admin Area 1	A1	AA1	Admin area, largest. For USA, a country.
Postal Code 4	Z4	PC4	Postal code, smallest. Unused in USA.
Postal Code 3	Z3	PC3	Postal code. For USA, a ZIP+4.
Postal Code 2	Z2	PC2	Postal code. For USA, a ZIP+2.
Postal Code 1	Z1	PC1	Postal code, largest. For USA, a ZIP.

Confidence Levels and Quality Types

When you get geocode results, you will typically examine each match to see how successful the match was. For each address element, the geocode results will contain a *confidence level code* with the general meaning of exact (A), good (B), approximate (C), or unused (X). It corresponds to the server's confidence that it matched that part of the address correctly.

Advanced developers who override the default geocoding rules should note that for every confidence level code except for “unused” (X), there is a corresponding *quality type* constant. Custom geocode rules will specify the minimum confidence level acceptable using a quality type constant

Within geocode results, a confidence level is assigned to each *address element*:

- **Street address and street name.** Street name, house number, road type, and directionals.
- **Administrative area names.** In USA data, these are city and state names.
- **Postal codes.** If the input address did not contain a postal code — even if one could be calculated from other address information — the confidence level will be Exact if we find one.

Table 4, Confidence Levels and Quality Types

Confidence Level	General Meaning	Search Quality Type	Match Code
Exact	Match locations that exactly correspond to the input location, as defined by the address elements specified.	EXACT	A
Good	Matches should be fairly similar, even if not exactly as specified.	GOOD	B
Approx	Matches should be somewhat similar to the input location as specified	APPROX	C
Unknown or unused	Confidence level has no meaning for this granularity level or was not used.	Not used when searching.	X

The confidence level of each address element is determined independently, and also each is independent of the overall granularity. No overall confidence level of the match is calculated.

For the general meaning of each confidence level, see Table 4, “Confidence Levels and Quality Types,” on page 26. Please refer to “Geocoding Constants” on page 37 for the exact syntax to use within your client interface.

When a high confidence level cannot be achieved, results may contain a lower confidence level, but never lower than the quality type you specified. For instance, if you specify the *exact* quality type for a location granularity search, you will get only exact results for the street address element, or no results if none were found. However, if you specify an *approximate* quality type, you may get results that are approximate, good, or exact.

Table 5 shows what typically constitutes exact, good, or approximate for each address element, although the details will vary by region and vendor

Table 5. Confidence Level Typical Meanings

Element	Level	Meaning
Street Name	Exact	Exact matches, although the geocoding engine allows standard variations of road types, directionals, numbered roads, and common abbreviations. Examples include Road vs. RD, North vs. N, Second vs. 2nd, and Mount vs. MT.
	Good	Good matches, ignoring differences in road type and directionals when they can be determined. Examples include Road vs. Street, North vs. South. This includes matches where the type and directional are supplied but not found in the match, or found in the match but not supplied.
	Approx	Sound-alike matches, partial matches, slight misspellings, and fuzzy matching. Details vary by geocoding data.
Admin Areas	Exact	Exact matches, although common abbreviations are acceptable. For instance, “Mount View” is the same as “Mt. View.”
	Good	Good matches, even if administrative area names do not exactly match. This includes sound-alike matches, partial, slight misspellings, or other fuzzy matching. Details vary by geocoding data.
	Approx	Matched administrative areas do not match the input administrative areas. This may occur when the postal code input determines the match.
Postal Code	Exact	The postal code matches at the granularity of the input. The matched postal code may be even more precise than the postal code specified in the input.

Table 5. Confidence Level Typical Meanings (*continued*)

Element	Level	Meaning
	Good	Postal code does not exactly match at the granularity of the input. However, the postal code likely matches the input address at a lower granularity level than at the requested granularity.
	Approx	Postal codes do not match. This may occur when the administrative area input determines the match.

MapQuest Address Objects

To geocode a location, first create an `Address` object and populate it with as much address information as you have. The `Address` object defines the address's street name, house number, city, county, state, postal code, and country. You can set address information using properties named `Street` (including street name, house number, road type, and directionals), `City`, `County`, `State`, `Country`, and `PostalCode`.

When you receive matches, they are in a collection of `GeoAddress` objects, which are similar to (but not a subclass of) `Address` objects. These objects contain all the properties of an `Address` plus the location's placement on the street segment, the geocoding status of the address (how successful it was), the latitude, and the longitude. These `GeoAddress` objects contain values indicating the quality of the match in the form of *geocode result codes*, which are described in more detail below.

Developers using international data are not restricted to USA administrative areas like cities, counties, and states. Both `Address` and `GeoAddress` classes support all seven levels of administrative areas, only some of which are used in each data set.

To get or set administrative areas by index number, get or set the `AdminArea` property of instances of class `Address` or `GeoAddress`. The interface for getting and setting these properties includes an administrative area index number (1-7). For more information about these classes, refer to the API Reference. If you do not know which administrative area indexes are supported in your international data, please contact MapQuest Technical Support.

Understanding Geocode Results

The server's geocoding engine executes geocoding rules that are defined for most developers by defaults on the server. The server executes each rule until either it finds matches as specified in the current geocoding rule or completely fails to find any matches using any rule.

For each geocoding rule, the server initially searches at the requested granularity (even if it's a low granularity) and at the exact confidence level (even if that confidence level is higher than the minimum requested).

For instance, if the rule requested a postal code granularity match, the server will ignore the street address, examine the postal code in the input address, and try to find an exact postal code match in the geocoding data.

If the server finds appropriate matches at that confidence level, even if there are fewer than the `MaxMatches` value, the server returns all matches it found and ignores the rest of the geocoding rules. Of course, the server will never return more than the matches specified in that rule's `MaxMatches` property.

If no matches at that confidence level were found, the search confidence level will decrease, and the server searches for alternatives at a lower confidence level if permitted by the quality type constant specified in the `GeocodeOptions` object. For instance, if no exact matches were found but you specified the quality type constant for "good," the server will now search for good matches.

The next geocoding rule will typically specify either a different data set, a lower confidence quality type, or lower granularity. For instance, if the server could not find the exact street addresses in its data, you might want to calculate the center of the likely postal code.

This process continues until matches are found for the current rule or there are no more rules to perform.

Geocode Result Codes

When geocoding is complete, the geocode results object includes the overall granularity at which the search stopped and a collection of results.

Each successful match contains a matched address and a *geocode result code*, a string of characters representing a match's granularity level and its confidence levels for the street name, administrative areas (in USA, city and state), and postal code.

See Table 6 for the meaning of the five characters in this string. Note that the order of confidence levels is arbitrary; it does not represent a preference or hierarchy among the returned values.

Table 6, Match Result Characters

Character Index	Meaning
1	Primary granularity code.
2	Granularity subcode.
3	Full street name and number confidence level.
4	Administrative area confidence level.
5	Postal code confidence level.

Prepare For Multiple Matches and Imperfect Results

Even when you expect that user addresses or addresses from custom data will not be ambiguous, you may receive 2 or more matches as results. In general, you should set `MaxMatches` to a number greater than 1 so you can prepare for such occurrences.

You can get ambiguous or duplicate results for a variety of reasons, and is even more likely to occur when using search confidence levels other than exact (using good or approximate). When the server finds more than one match, especially if the confidence levels are low, MapQuest strongly recommends that, when possible, you let the user choose which one is the correct one.

When display space allows, display the full addresses of geocode matches even if you assume geocoding was successful and the first match is perfect. This helps users detect misspellings or omitted address elements that resulted in unexpected “best guesses” by the server.

Geocode Result Examples

Here are some examples to explain how this would work. Let us geocode a specific address, 2224 Kater Street in Philadelphia, PA, postal code 19146, with the geocode rules specified as follows:

Table 7, Geocode Rule Example

Rule Number	Match Type	Quality Type	MaxMatches
1	Location	Exact	1
2	Location	Approximate	1
3	Block, Nearest	Good	1
4	Postal Code	Good	1

Geocode Example 1

If our input address exactly matches the geocoding data, the server will find one match on geocode rule number 1, and then end the search. The match might look like:

Address: 2224 KATER ST., PHILADELPHIA, PA 19146-1139
Result: L1AAA

The first two characters (L and 1) of the result code indicate the granularity of the match L1, the codes for a location granularity match. The following “A” characters indicate an exact match on the street address, the administrative areas, and the postal code.

Note that abbreviation differences like “St.” versus “Street” are minor enough to be permitted for “exact” matches. See Table 5, “Confidence Level Typical Meanings,” on page 27 for details.

Geocode Example 2

Let us suppose the user accidentally typed Kater Road instead of Kater Street. Assuming there were no Kater Road in the geocoding data, rule 1 would fail, but rule 2 would succeed with a match that looks like:

Address: 2224 KATER ST., PHILADELPHIA, PA 19146-1139
Result: L1BAA

The “B” in the match code indicates a “good” match of the street name and number, although the server has high confidence in the other elements of the address.

Geocode Example 3

Suppose the street name were input correctly but the user entered the invalid house number 222224, higher than any valid numbers on that street. Rule 1 and 2 would fail. However, rule 3 could succeed with a nearest block granularity match that would return the closest numbered block:

Address: [2500-2599] KATER ST., PHILADELPHIA, PA 19146-1139
Result: B3AAA

The first two characters of the result code (B3) indicate the codes for a nearest numbered block.

Geocode Example 4

If we omitted the street name and number altogether, a location match would be impossible, as would the block match. Since we provided the city, state, and postal code, rule 4 successfully matches the following address:

Address: PHILADELPHIA, PA 19146
Result: Z1XAA

The first two characters of the result code (Z1) indicate the granularity codes for a postal code match of the lowest accuracy in the USA, a 5 digit ZIP code. The “X” character in the result code indicates that the geocode rule that matched the address did not use the supplied street address, which was blank in this last example.

Basic Geocoding

The `Geocode` client object method takes either a `GeocodeOptionsCollection` or a `AutoGeocodeCovSwitch` object. For basic geocoding, pass simply a location and a collection for the results, which is a collection of `GeoAddress` objects. The server will use the default server *geocode selector*, which chooses the best geocoding data for typical use.

The following are the steps for basic geocoding:

1. Populate an `Address` object with the address to geocode.
2. Create a `LocationCollection` object for the results.
3. Call the `Geocode` client object method passing these two objects.
4. Check the number of matches in the results collection by getting the size of the `LocationCollection`, which contains a list of objects of class `GeoAddress`.
5. When the number of returns is zero, display a failure message and encourage users to check spelling, etc.
6. For each result, do something with it. For instance, display each result's address by getting properties such as `Street` and `City`.

Here is a code example illustrating this technique:

```
MQClientInterface.Address address = new
MQClientInterface.Address();
MQClientInterface.LocationCollection geocodeResults = new
MQClientInterface.LocationCollection();
address.Init();
address.Street      = "100 Penn St";
address.City        = "Pittsburgh";
address.State       = "Pa";
address.PostalCode  = "15215";
address.Country     = "US";
// In real code, catch exceptions from this server call...
geocodeClient.Geocode(address, geocodeResults);

if (geocodeResults.Size == 0)
{
    Console.WriteLine("ERROR - The address entered could not be geocoded");
}
else /*Location geocoded. Display the match(es).*/
{
    for (int i = 0, iCount = geocodeResults.Size; i < iCount;
        i++)
    {
        MQClientInterface.GeoAddress geoAddress =
        (MQClientInterface.GeoAddress)geocodeResults.GetAt(i);

        Console.WriteLine("Match # {0}", i+1);
    }
}
```



```
Console.WriteLine("\t{0}", geoAddress.Street);
Console.WriteLine("\t{0}", geoAddress.City);
Console.WriteLine("\t{0}", geoAddress.County);
Console.WriteLine("\t{0}", geoAddress.State);
Console.WriteLine("\t{0}", geoAddress.Country);
Console.WriteLine("\t{0}", geoAddress.PostalCode);
Console.WriteLine("\t{0}, {1}",
geoAddress.LatLng.Latitude, geoAddress.LatLng.Longitude);
Console.WriteLine("\t{0}\n", geoAddress.ResultCode);
    }
}
```

For complete working sample code and more comments within the code, please refer to the `GeocodeIt` sample code.

Advanced Geocoding

The geocoding rules within server-based *geocode selectors* will work for the vast majority of developers. If you are sure you need custom geocoding rules, override the default geocode selector on the server by defining your own geocoding rules. This approach is rarely used and is **not recommended** for most developers.

Geocoding rules specify what types of information are important, which geocoding data to use, and what types of matches would be best for your application. For each rule, you will define a `GeocodeOptions` object.

You will make multiple `GeocodeOptions` objects and add them in priority order to a collection of the class `GeocodeOptionsCollection`. When you pass that collection and other parameters to the `Geocode` client object method, the rules in the collection are examined in order.

Each geocoding rule specifies a granularity level for desired matches (as described in “Granularity Levels” on page 24) and a minimum confidence level using a quality type constant (as described in “Confidence Levels and Quality Types” on page 26).

The server stops searching when matches are found using the current rule. The server finds as many matches as possible, but never returns more matches than specified in the rule’s `MaxMatches` property.

Table 8 contains the most important properties within the `GeocodeOptions` class. For more information about this class, please refer to the API Reference.

Table 8, `GeocodeOptions` Properties

Property	Description
<code>CoverageName</code>	The name of a geocode pool on the server. Common examples include <code>tana</code> , <code>navt</code> , and <code>gaz</code> .
<code>MatchType</code>	A <i>match type</i> constant that corresponds to the granularity level of the desired match. Please refer to “Geocoding Constants” on page 37 for the exact syntax to use within your client interface.
<code>QualityType</code>	A <i>quality type</i> constant that specifies the minimum confidence necessary in each match. Please refer to “Geocoding Constants” on page 37 for the exact syntax to use within your client interface.
<code>MaxMatches</code>	The maximum matches to return. Must be at least 1.

Before you create geocoding rules, you must know what geocoding data is available on your server within *geocode pools*. If you do not already know how your server is configured for geocoding, you must find out by viewing your server status page as discussed in “Learning Your Server Configuration” on page 13.

You will typically create many geocoding rules. For instance, you might prefer exact matched street addresses when possible; failing that, use another data set for finding street names with no house numbers; failing that, use another data set to estimate the city center.

See also: “Geocode Result Examples,” p. 30.

Defining Your Own Geocoding Rules

In the following example, we geocode a street address and request a location match with a single geocoding rule to illustrate the basic geocoding API.

The following are the steps for defining your own geocoding rules:

1. Create geocode rules using one or more `GeocodeOptions` objects.
2. Set each rule’s `MatchType`, `QualityType`, and `MaxMatches` properties.
3. Add all the rules to a new `GeocodeOptionsCollection` object.
4. Populate an `Address` object with the address to geocode.
5. Create a `LocationCollection` object for the results.

6. Request geocoding with the `Geocode` client object method using the two collections and the `Address`.
7. Check the number of matches by the size of the `LocationCollection`, which contains a list of objects of class `GeoAddress`.
8. When the number of returns is zero, display a failure message and encourage users to check spelling, etc.
9. For each result, do something with it. For instance, display each result's address by getting properties such as `Street` and `City`.

Here is a code example illustrating this technique:

```
MQClientInterface.Address address = new
MQClientInterface.Address();
MQClientInterface.LocationCollection geocodeResults = new
MQClientInterface.LocationCollection();

address.Init();
address.Street      = "100 Penn St";
address.City        = "Pittsburgh";
address.State       = "Pa";
address.PostalCode  = "15215";
address.Country     = "US";
MQClientInterface.GeocodeOptions geocodeOptions = new
MQClientInterface.GeocodeOptions();
geocodeOptions.MatchType = MQClientInterface.MatchType.MT_LOC;
geocodeOptions.MaxMatches = 1;
geocodeOptions.QualityType =
MQClientInterface.QualityType.QT_EXACT;
geocodeOptions.CoverageName = "navt";

MQClientInterface.GeocodeOptionsCollection geocodeOptionsCollection
    =new MQClientInterface.GeocodeOptionsCollection();

geocodeOptionsCollection.add(geocodeOptions);
// In real code, catch exceptions from this server call...
Client.geocode(originAddress, geocodeResults,geocodeOptionsCollection);

if (geocodeResults.Size == 0)
{
```

```
Console.WriteLine("ERROR - The address entered could not be geocoded");
}
else /*Location geocoded. Display the match(es).*/
{
    for (int i = 0, iCount = geocodeResults.Size; i < iCount;
        i++)
    {
        MQClientInterface.GeoAddress geoAddress =
            (MQClientInterface.GeoAddress)geocodeResults.GetAt(i);

        Console.WriteLine("Match # {0}", i+1);
        Console.WriteLine("\t{0}", geoAddress.Street);
        Console.WriteLine("\t{0}", geoAddress.City);
        Console.WriteLine("\t{0}", geoAddress.County);
        Console.WriteLine("\t{0}", geoAddress.State);
        Console.WriteLine("\t{0}", geoAddress.Country);
        Console.WriteLine("\t{0}", geoAddress.PostalCode);
        Console.WriteLine("\t{0}, {1}",
            geoAddress.LatLng.Latitude, geoAddress.LatLng.Longitude);
        Console.WriteLine("\t{0}\n", geoAddress.ResultCode);
    }
}
```

For complete working sample code and more comments within the code, please refer to the `GeocodeIt` sample code.

Limiting Matches With A Geocode Selector

You can limit the number of matches returned by geocoding even if you are using a server-based geocode selector. Create an instance of `AutoGeocodeCovSwitch` and set its `MaxMatches` property to the maximum matches. The actual number of matches returned for a specific request may be smaller than this value.

When calling the `Geocode` client object method, pass the `AutoGeocodeCovSwitch` object as the final argument instead of a `GeocodeOptionsCollection` object.

Reverse Geocoding

The geocoding engine also provides *reverse geocoding*, which takes a latitude and longitude pair and returns a full street address. It is important to note that reverse geocoding uses mapping data, not geocoding

data, for the majority of the reverse geocoding process. Reverse geocoding is supported by most but not all data sets.

Find an address from latitude and longitude coordinates using the `ReverseGeocode` client object method, which takes a server map pool name, a location in a `LatLng` object, and for the results an empty `LocationCollection`.

For details about server map pool configuration, refer to “Learning Your Server Configuration” on page 13.

Items in the results collection are `GeoAddress` objects, although in current server releases this method returns no more than one result. If it fails, the collection is empty.

By default, `ReverseGeocode` stores the street address and postal code in the `GeoAddress`, but administrative areas (like city, state, or Canadian province) are empty. You can get administrative area names automatically by providing an optional parameter to `ReverseGeocode` that specifies what’s called a *geocode pool* on the server. If specified, `ReverseGeocode` geocodes the postal code and sets all the administrative area names.

Advantage API customers who want USA city and state lookup should supply the geocode pool `us_postal`. For other countries, contact Technical Support. For details about your server configuration, refer to “Learning Your Server Configuration” on page 13.

If your application potentially needs more than one match, instead use the robust search APIs discussed in Chapter 7, “Proximity Searching.”

Geocoding Constants

.NET developers should use the geocoding constants shown on the left using the syntax shown on the right.

Quality Type Constants

```
MQClientInterface.QualityType.EXACT  
MQClientInterface.QualityType.GOOD  
MQClientInterface.QualityType.APPROX
```

Match Type Constants

```
MQClientInterface.MatchType.LOC
MQClientInterface.MatchType.INTR
MQClientInterface.MatchType.NEARBLK
MQClientInterface.MatchType.BLOCK
MQClientInterface.MatchType.AA1 // Country
MQClientInterface.MatchType.AA2
MQClientInterface.MatchType.AA3 // State (USA) or Province
MQClientInterface.MatchType.AA4 // County (USA)
MQClientInterface.MatchType.AA5 // City
MQClientInterface.MatchType.AA6
MQClientInterface.MatchType.AA7
MQClientInterface.MatchType.PC1 // ZIP (USA)
MQClientInterface.MatchType.PC2 // ZIP+2 (USA)
MQClientInterface.MatchType.PC3 // ZIP+4 (USA) or Postal
MQClientInterface.MatchType.PC4
MQClientInterface.MatchType.POI
```

4 Basic Mapping

Your application can produce high-quality interactive maps, use data from leading map data vendors, add icons, and automatically select the best map data when you use multiple map data sets.

Developers can use MapQuest APIs to dynamically change the appearance of groups of map features (by type) or individual map features.

Interactive Map Example



Mapping Features

Map data selection. Let the server choose the best mapping data on the server for the requested map. For instance, one data set might be used for street-level views and another for national overview maps.

Panning. Provide latitude and longitude for the map center, or allow the user to click on the map to re-center the map at that location.

Zooming. View maps at an arbitrary “zoom” setting, choose customizable *zoom levels*, or let the user use a mouse to “draw” a rectangle to approximate the desired map scale.

Best fit. The server can automatically adjust center point and scale to best highlight any visible Points of Interest. For example, if a user searches for nearby hotels, the mapping engine can automatically calculate the center point and scale that “best fits” all search results.

Point-and-click APIs. APIs exist to allow your users to click with a mouse to pan, zoom, identify streets, and identify map features. You could allow users to click to display a street name or trigger an application-specific action.

Customizable map styles. Developers can programmatically change styles of entire groups of features or individual features. This includes changing colors, fonts, line widths, road shield appearance, or hiding features dynamically. You can also customize map styles for different contexts, like color display versus black-and-white printing.

Map icons. Display map features with unique icons and optional text labels. Points of Interest are discussed primarily in Chapter 5, “Displaying POIs.”

Map annotations. Draw simple shapes on maps to highlight map features, search parameters, or other custom data. For example, you could highlight a search region, boundaries of private property, or identify sub-map regions. For API information, refer to Chapter 6, “Map Annotations.”

Mapping Concepts

Before describing the details of mapping APIs, there are mapping concepts that are important to understand that are discussed in the following sections:

- Map Data.
- Map Scales and Zooming.
- Display Types and Map Styles.
- Automatic Map Data Selection.

Map Data

For each map data set, the server is configured with a *map pool*, which tells the server where to find the necessary map data. A *pool* is a server resource of one type. Most developers do not need to know the server’s exact map pool names because they let the server automatically choose the best map pool data. However, if you need to know your server’s exact map pool names, refer to “Learning Your Server Configuration” on page 13.

Although a single *map coverage* (mapping data set) can encapsulate multiple layers of map data for the same geographic area, whether a feature displays in a requested map is defined by *map style* data, not the mapping data itself.

Map Scales and Zooming

Every requested map has a *scale*, which is analogous to a “zoom” setting in a desktop computer program. Map scale is defined as the proportion of map units to real-world units. For example, if a map’s scale is expressed as 10,000, this means a scale ratio of 1:10,000, meaning one map unit represents 10,000 real-world units.

On a map where one inch equals 5 miles, the scale ratio is 1:316,800 because there are 316,800 inches in 5 miles. A typical city-level map might have a scale of 1:24,000, while a typical state-level map might have a scale of 1:2,000,000.

MapQuest APIs only use the number after the “1:” in the scale ratio. Remember that a map with a large scale number after the “1:” displays a large real-world area.

When the user “zooms in” or “zooms out” on a map, the user changes the map’s scale. Zooming in redisplay the map with a smaller *scale number*, for example zooming from a state map to a city map. Zooming out redisplay the map with a larger scale number, for example zooming from a state map to a national map.

You can let the server choose a scale when zooming in or out using pre-defined *zoom levels*, which are discussed in more detail later.

Display devices vary in *resolution*. For example, many monitors are 72 dots per inch (DPI) and your printer might be 300 DPI. To accommodate these differences, you can set the display resolution in dots per inch (DPI). The default is 72 DPI, but can be changed.

To illustrate the interaction of *scale* and *resolution*, if a map has a scale ratio 1:10,000 and display resolution 72 DPI, every 72 pixels on a map represents 10,000 real-world inches.

Advantage API also supports zooming to a specified rectangle, defined with latitude and longitude or X/Y pixel coordinates. The server displays the rectangle at a scale that allows it to fill the display area as much as possible.

There are also APIs for *panning*, which changes a map’s center point but does not change the map scale.

Automatic Map Data Selection

Servers generally have multiple map data sets with overlapping map data and the server can choose the best map data to display. For instance, if you have NAVTEQ, TANA, and NADB data installed, which should display street-level detail in a specific city? This decision process is called *map data selection* and is configured on the server.

To choose a coverage, the server examines the map’s center point, scale, and size; the preferred scale settings for each coverage; the geographic areas supported by each data set; and the data sets’ relative priority.

When map data selection is configured on the server, MapQuest client applications do not have to do anything to support it. When applications request a map from the server, the server simply selects the best map data.

However, map data selection is customizable, which is discussed in “Advanced Map Data Selection API” on page 50.

You can use MapQuest APIs to get the coverage name finally selected by map data selection. See “Map Data Selection Classes” on page 49 for details.

Display Types and Map Styles

Each map feature in the map data has a *display type* (DT), a number that identifies a type of map feature: a city, a lake, a highway, a dirt road, or other choices. You can think of it as a general classification of the map feature.

Display types from 0000 through 3071 identify common objects like roads and rivers. Developers may assign and use display type codes from 3072 through 3583 for custom Points of Interest. For example, if you have a database of hotels, you might assign display type 3583 to hotels. You would control the appearance of hotels by modifying the *map style* data for display type 3583.

If you plan to create non-point features or use display types associated with licensed map data, please refer to Appendix A, “Display Types.”

Developers use the default styles on the server for a map coverage by referencing the the name of *style pools* on the server. Each style pool loads its styles from *style files* that control the appearance of features of each display type at different scales. For example, a style file could specify that at scale ratios 1:2,000 to 1:2,999, interstate highways display as thin blue lines and city parks are not visible; at higher scales, interstate highways could display as thick blue lines and parks display as green polygons with a text label.

Advantage API developers cannot modify style files on the server. However, they can override styles using MapQuest APIs called *style objects*.

Advanced Map Styles In Advantage API

A map data set typically has multiple map styles, each of which is defined in a style pool on the server. Because you don’t know in advance what map will be chosen by map data selection, you typically reference them using a string called a *style alias* instead of a specific style pool name.

There are standard style aliases that are implemented in each data set. By using a style alias instead of a specific style pool name, you can request a default color map with the style alias "default" or a black-and-white map with the style alias "bw". There are also style aliases called "european" for European data and "classic" for a yellow background map. Note that if you do not specify a style alias, you’ll receive the default style alias, “style5”.

Style alias information on the server supports map data selection by mapping a general style name to a specific style file for each data set. Style aliases are **not** used by the map data selection algorithm to determine what map data to use; after the server chooses the data set, style aliases determine which style pool to use.

If you use style aliases, you can use MapQuest APIs to get the style pool name finally selected by map data selection. Refer to “Getting Style Pool Names From Style Aliases” on page 52 for details.

See also: “Customizing Map Appearance,” p. 56.

Sessions Overview

Interactive maps require the perception of continuity from one action to the next. A *session* describes the current map state so that programmers can save properties of a map from one action to the next.

Saving session information is generally easy for desktop and Java applications because they typically can save data easily in standard programming variables.

In contrast, HTML pages and Web applications are stateless, meaning they exist only for the time required for a page display. For the Web developer, this means that a Web page request contains no information about recent actions. Even if each Web page embedded recent map-related actions and data in each Web page, this would be a potentially large amount of data including database search queries, search results, and map annotations.

To simplify this process, Web applications can save session information on the server and refer to the data with a short string that identifies the session. Each page would merely save these *server session* strings in hidden form properties, URLs, Web browser cookies, or other storage mechanisms.

Although server sessions are the easiest way to use maps in many Web applications, you can create *stateless maps* using map image URLs called *direct URLs* that do not rely on MapQuest server sessions. Stateless maps are discussed in more detail later.

Desktop and Java applications can use all of the APIs mentioned above. However, since these types of applications typically can easily save session data locally and there are APIs that allow immediate downloading of map images, most applications like this choose not to use server sessions.

In all cases, developers instantiate an object of class `Session` to temporarily store everything necessary to draw the map. This object is called a *local session object*.

What Sessions Contain

Sessions encapsulate all information necessary to draw any map. This includes basic parameters like the map size and scale, but also map annotations, database queries used for POIs, simple POIs (including POI search results), and other options.

Add information to a local session object by creating objects of *session parameter object* classes and adding them with the session's `AddOne` method. All classes of session parameters are shown in Table 9.

Table 9, Session Objects

Information	Class Name	Details
Map state	<code>MapState</code>	<i>Required.</i> Coverage name, map size, scale, and center point.
DB queries	<code>DBLayerQueryCollection</code>	Database queries and results, used to display POIs on maps.
Features	<code>FeatureCollection</code>	A collection of map features to display, typically POIs.
Style changes	<code>CoverageStyle</code>	A collection of style objects, each of which overrides appearance settings defined in style pools on the server.
Annotations	<code>PrimitiveCollection</code>	Each item is a <i>drawing primitive</i> , discussed in Chapter 6, “Map Annotations.”
Map command	<code>MapCommand</code>	Subclasses of this class modify the map scale and center point. See “Using Map Commands” on page 62.
Advanced map data selection	<code>AutoMapCovSwitch</code>	An object that configures advanced map data selection.
Route highlights	Never directly accessed by developers. Never downloaded to a local session object.	Driving direction shapes added only using the routing API. These exist only within sessions stored on the server.

The `Session` method `AddOne` is unusual in the MapQuest API because most collections use an `Add` method (not an `AddOne` method) to append items. The session `AddOne` method enforces the requirement to have only one object of each class in the collection at any time. For instance, if there is already a `MapState` object in the session, adding another with `AddOne` will remove and destroy the previous one associated with the session.

If you want to get an object from the local session object, use the `GetObject` client object method, which takes a class name string as its argument.

In C#, you would simply use the class name as a string. For instance, to get a local session object's `MapState` object, you would use the code:

```
mapstate = (MQClientInterface.MapState)mqSession.GetObject("Mapstate");
```

When to Use Server Sessions

Most Web applications use server sessions if the user could potentially change a map, for instance if users can change zoom settings, pan the map, or display different POIs within the same geographic area.

You do not need to use server sessions if any of the following are true:

- Displaying a map is a one-time occurrence.
- You provide no user interaction for the map image itself.
- You do not display maps.
- Your application is a non-browser application. It is easier to save state information in these application types than it is for a Web application. However, using MapQuest server sessions might simplify programming code for some applications.

Creating Server Sessions

To create a server session, first create a *local session object* (an object of class `Session`), then configure it with parameters that you care about, and then call the client object method `CreateSessionEx`.

This will upload session information to the server, create a session object on the server with those parameters, and return a *session ID string* that identifies the session object stored on the server.

To display a map with a MapQuest server session:

1. Create an object of class `Session`.
2. Add *session parameter objects* using the session object's `AddOne` method. The only required session parameter object is a `MapState` object.
3. Call the `CreateSessionEx` client object method, which takes the local session object as its only argument.
4. Save the result, called a *session ID string*.

5. Web applications typically would pass the session ID string to the client object method `GetMapFromSessionURL`, which returns a URL that generates a map. Expose this URL within an HTML image tag.

Desktop applications typically would pass the session ID string to the client object method `GetMapImageFromSession`, which gets the image directly without generating an image URL.

Note that the local session object and the server session are **not** automatically synchronized. If you update any local session object parameters, you must update the server session with the new data. For details, see the next section, “Changing Server Sessions.”

A session-based URL embeds the session ID, but not the details of the session data. Because of this, you cannot generate multiple URLs from one session that represent **different** maps. If you want URLs that represent multiple maps, you must use multiple MapQuest server sessions. Alternatively, you could create multiple *stateless maps*, which do not use MapQuest server sessions.

Changing Server Sessions

Based on user interaction or other reasons, you might need to update settings of the session. For instance, if you are using sessions for maps and the user changes the map scale, your application will directly or indirectly change the session’s `MapState` session parameter.

When you create a MapQuest server session, you will receive a *session ID string* that identifies the server session. If you want to update some parameters of that server session, there are several API functions for this:

- To change zoom settings, use the zoom methods of your client object: `ZoomIn`, `ZoomOut`, `ZoomXY`, `ZoomToRectangleXY`, and `ZoomToRectangleLL`. These methods modify the server session and do not modify nor update a local session object.
- To pan the map, use the `Pan` method of your client object. This method modifies the server session and does not modify the local session object.
- For full flexibility, update any session parameters in a local session object and then call the `UpdateSessionEx` client object method. For instance, add session parameter objects such as a database query collection, and then update the server session with the new data.

If you use a method that modifies the server session directly, you can update your local session object with the most up-to-date session data using the `GetSessionEx` client object method.

Ending Server Sessions

If you are certain that a session is complete, delete the session on the server using the `DeleteSession` client object method. For instance, when a user “logs out” of a Web application or if a desktop application quits.

Sessions not used for a while may be deleted by the server. For the best server performance, delete sessions explicitly whenever appropriate.

If you try to access information for a session that has expired, the MapQuest API will throw an exception. Catch **all** exceptions from MapQuest API functions in production code and handle them appropriately.

Saving Session IDs in Web Applications

Web applications that use MapQuest server sessions to preserve map state information across map views must save the session ID string somewhere. Here are some options:

- Encode session ID strings in URLs.
- Save session ID strings in Web browser cookies.
- Save session ID strings in Web pages in hidden HTML form properties.
- Save session ID strings in some other persistent storage and link them to a user ID, Web cookie, or application-specific user or session.

Server sessions may expire after a certain amount of time. If sessions expire, MapQuest clients cannot access information in the session, so critical persistent data must be saved elsewhere. If this is a problem, you should use *stateless maps* as discussed in the next section.

Stateless Maps for Web Applications

Developers need not use MapQuest server sessions to display maps. MapQuest APIs can create map image URLs that embed all information that would be stored in a server session. These are called *stateless maps*.

The major advantage of a stateless map for a Web application is that its *direct URL* is valid for an extended period of time. Stateless maps create exactly the same high-quality map images as server session maps, but their URLs work forever if the server does not change its domain name (or IP address), port number, or other important configuration settings. The extended valid time for a map URL can be a major advantage for some Web applications.

The major downside of this approach is that you cannot extract any map state information from the direct URL itself. If any state information needs to be saved, you will have to save it yourself somehow. This is usually easy for non-browser applications, but may require extra work for Web applications.

For instance, if you want to use stateless maps but want to allow the user to click on “zoom in” or “zoom out” buttons, you will have to store the **old** map scale in each Web page so that your Web application knows what the **new** map scale should be for zooming in or zooming out.

Using stateless maps does not prevent user interaction in Web applications, but you must store interactive map settings in some other way:

- Encode map state information in your own encoded Web page URLs, which your application would use to generate map image URLs when needed.
- Store map state information in Web browser cookies.
- Store map state information in Web pages in hidden HTML form properties.
- Store map state information in some other persistent storage and link them to a user ID, Web cookie, or application-specific user or session.

Another downside of stateless maps is very long map URLs that can exceed the URL length for some browsers. Displaying a large number of POIs or a large number of map annotations will further increase the map URL length. URL length is rarely a problem for non-browser applications. However, Web applications that cannot predict the target Web browser might in rare situations produce map URLs too long to download.

If your application is not a Web application, you can use client object methods to download the map image immediately rather than get a map URL. That approach avoids all time limitation problems for map URLs. For more information, see “Downloading Map Images” on page 55.

See also: “What Sessions Contain,” p. 44; “Creating Stateless Maps,” p. 54.

Be Careful When Updating Sessions

When you update a local session object with data from the server, the MapQuest API replaces any session parameter objects already in the local session object. Remember to update your references/pointers to the new objects and delete unused session objects. Failure to do so can cause application errors, memory leaks that can lead to crashes, or immediate crashes in some client interfaces. This can affect all developers, and is especially a common issue for C++ developers.

This is true for developers using server sessions and those using stateless maps.

For example, suppose you have a private variable set to a `MapState` object and you add it to a local session object. If you use `GetSessionEx` or `UpdateSessionDirect` client object method, the MapQuest APIs will fill the local session object you provide with new session parameter objects.

If you provided a **new** local session object to these methods, remember to dispose of the old session object and set your private variable to the **new** `MapState` object.

If you provided your **original** local session object to these methods, the old `MapState` object will be destroyed during the update. The reference to old `MapState` object may be either irrelevant or invalid. Never access the old reference/pointer again, and set your private variable to the **new** `MapState` object.

To get an session object from a local session object, use the local session object method `GetObject` discussed in “What Sessions Contain” on page 44.

This affects all the following client object methods: `UpdateSessionEx`, `GetSessionEx`, `UpdateSessionDirect`, and the non-recommended methods `UpdateSession` and `GetSession`.

Basic Mapping Classes

This section describes the basic mapping classes. There are other map-related classes discussed in other sections and chapters. In particular, see Chapter 5, “Displaying POIs,” and Chapter 6, “Map Annotations.”

MapState Objects

A `MapState` object contains map state information necessary to display a map, such as size, scale, and center point. When you want to display a map, first create an object of class `MapState`. You will then set its properties `WidthInches`, `HeightInches`, and `MapScale`.

Alternatively, you can set the map size in pixels using the `MapState` properties `WidthPixels` and `HeightPixels`. Setting height and width like this assumes a display resolution of 72 dots per inch (DPI), but you can set the image size with respect to an arbitrary DPI using an optional argument when setting these properties. See the API Reference for details.

Note that specifying a DPI when setting (or getting) height and width only affects the height and width calculations but does **not** permanently change the DPI for a map. To actually change the display DPI setting for a map, you must change the DPI within the `DisplayState` object associated with the map. For details, see “Customizing Map Format, Resolution, and Aliasing” on page 55.

Set the map’s center with code that creates an object of class `LatLng`:

```
mapState.Center(new MQClientInterface.LatLng( 40.44569, -  
79.890393));
```

Add it to your local session object using the session’s `AddOne` method. See “Sessions Overview” on page 43 for more information about local session objects. To change the current map state, use the APIs discussed in “Changing Server Sessions” on page 46.

See also: “User Interaction In Maps,” p. 61.

Map Data Selection Classes

Developers do not have to do anything to support basic automatic map data selection. The server by default uses its default map data selection settings to select the best map data based on settings like the requested

map's boundaries, the supported map scales for each map data set, and the relative priority of each map data set.

Advanced Map Data Selection API

Adding an `AutoMapCovSwitch` object to a local session object allows advanced map data selection. To support automatic map data selection, the `Name` property of this object must match the `Name` setting in a `MapDataSelector` pool on the server.

Advantage API customers should refer to “Advantage API Connection Information” on page 16 for more server information.

By default, all installed map data sets are available during map data selection. You can remove some data sets from consideration by map data selection rules using the property `DataVendorCodes` of the `AutoMapCovSwitch` object. This property contains a collection of data vendors referred to by integer constants, not their map pool names.

This list can represent either a list of explicitly included data vendors or a list of data vendors explicitly removed from consideration when selecting a map coverage. The meaning of this collection is determined by the value of the `DataVendorCodeUsage` property (see below for details).

Please refer to “Mapping Constants” on page 64 for the exact syntax for data vendor codes within your client interface

To use this API, get the `DataVendorCodes` property, change the contents of the returned integer collection (of class `IntCollection`), and then set the `DataVendorCodeUsage` property appropriately. If the collection is empty, **no** data set limiting will occur, independent of the value of the `DataVendorCodeUsage` property.

You can change the map style using `AutoMapCovSwitch`. If you want to set a style alias or an explicit style pool name and still use automatic map data selection, you must set both the `Style` and `Name` properties properly. Do not set the `Name` property to the empty string, which will disable automatic map data selection. Instead, set `Name` to the name of a valid map data selector pool on the server and then set the `Style` property to the style alias or style pool. Advantage API customers should refer to “Advantage API Connection Information” on page 16 for data selector pool names.

The server has pre-defined *zoom levels* that provide standard map scale settings that users expect. These are used by the client object methods `ZoomIn` and `ZoomOut`, as well as the map commands `ZoomIn` and `ZoomOut` (discussed in more detail later). The defaults work for most developers, but you can override these settings in using the `ZoomLevels` property of `AutoMapCovSwitch`.

In the .NET client interface, `ZoomLevels` is implemented as a `ZoomLevels` attribute. It returns a collection, which you should modify directly; there is no `SetZoomLevels` method.

The `AutoMapCovSwitch` object contains all the properties shown in Table 10.

Table 10, `AutoMapCovSwitch` Properties

Property	Description
Name	<i>Required.</i> The name of a <code>MapDataSelectorPool</code> in your server initialization file. To disable map data selection, set Name to the empty string. Advantage API customers should refer to “Advantage API Connection Information” on page 16 for data selector pool names.
VendorCodeUsage	A flag indicating whether to include or exclude the specified vendor codes listed in the <code>VendorCodes</code> property. The possible settings are <code>Include</code> and <code>Exclude</code> . This property is ignored if the collection in <code>VendorCodes</code> property is empty. Please refer to “Mapping Constants” on page 64 for the exact syntax within your client interface.
ZoomLevels	A collection of zoom level scales used by the <code>ZoomIn</code> and <code>ZoomOut</code> client object methods. The defaults are 6000, 12000, 24000, 48000, 96000, 192000, 400000, 800000, 1600000, 3000000, 6000000, 12000000, 24000000, and 48000000.
Style	Set this to a style pool name or a style alias defined on the server. If you do not set this, the server’s defaults will be used. If you need the style pool finally chosen after map data selection, see “Getting Style Pool Names From Style Aliases” on page 52.

To customize map data selection:

1. Create an object of class `AutoMapCovSwitch`.
2. Set its `Name` property to the name of a `MapDataSelector` pool defined on the server. Advantage API customers should refer to “Advantage API Connection Information” on page 16 for data selector pool names.
3. Set its `Style` property to a style alias or style pool name.
4. Set its `ZoomLevels` property if desired.
5. Limit data vendor codes as mentioned above if desired. This is rarely used.
6. Add the `AutoMapCovSwitch` object to your local session object using the session’s `AddOne` method.

Although map data selection is strongly recommended, you can disable map data selection and manually specify the desired map data set. To do this, create an `AutoMapCovSwitch` object, set its `Name` property to

the empty string, add it to your local session object, and then set the desired map pool name in your MapState object's CoverageName property.

See also: "Sessions Overview," p. 43; "Automatic Map Data Selection," p. 41.

Getting The Map Pool After Map Data Selection

If you want to know which map pool was selected by map data selection, set all your session parameters, and then perform one of the following steps:

- If you use MapQuest server sessions, update the session with the `UpdateSessionEx` client object method, then get the session data with `GetSessionEx`, and then get the `CoverageName` property from your session's MapState object.
- If you use stateless maps, use the `UpdateSessionDirect` client object method and then get the `CoverageName` property from your session's MapState object.

To get a MapState object from a local session object, use the `GetObject` method discussed in "What Sessions Contain" on page 44.

Getting Style Pool Names From Style Aliases

The `AutoMapCovSwitch` property `Style` can contain a style alias such as "default" that can refer to different style pools depending on the data set selected during map data selection. If you specified a style alias in the `Style` property, in rare cases you might need the name of the style pool that was picked during map data selection.

If you want to know what style pool was selected by map data selection and style alias rules, you must create an `AutoMapCovSwitch` object as described in the previous section. Perform the following steps after setting session parameters, including the `AutoMapCovSwitch` property `Style`:

1. If you use MapQuest server sessions, update the session with the `UpdateSessionEx` client object method and then get the session data with `GetSessionEx`.
If you use stateless maps, use the `UpdateSessionDirect` client object method.
2. Get your session's `CoverageStyle` object using the client object method `GetObject`. For more information about the client object method `GetObject`, refer to "What Sessions Contain" on page 44
3. Check that `CoverageStyle` object's `Name` property for the style pool name.

Basic Mapping API

Basic Mapping With Server Sessions

The first step for basic maps is creating a local session object with appropriate options (see “What Sessions Contain” on page 44). You would then create a server session and generate a map URL from the session ID string.

You will design your map by adding options called *session parameter objects* to your local session object. The only required session parameter class is `MapState`, which sets the map size, scale, and center point. Please refer to Table 9, “Session Objects,” on page 44 for the complete list of session parameter objects.

These are the steps to set up a standard GIF image map and get its URL:

1. Create a `Session` object.
2. Create a `MapState` object and set its properties for size, scale, and center.
3. Add the `MapState` object to the session using the session’s `AddOne` method.
4. Add any other session parameter objects desired.
5. Web applications would get a map URL with the `GetMapFromSessionURL` client object method and expose the URL in an HTML image tag. Other applications would get the image data with `GetMapImageFromSession`.

The code snippet below illustrates this technique for a Web application using server session URLs and downloading the image directly:

```
MQClientInterface.MapState mapState = new
MQClientInterface.MapState();

mapState.WidthPixels  = 392;
mapState.HeightPixels = 245;
mapState.MapScale     = 48000;
mapState.Center       = new MQClientInterface.LatLng(40.44569, -79.890393);

MQClientInterface.Session mqSession = new MQClientInterface.Session();
mqSession.AddOne(mapState);

System.String sessionId;

// In real code, catch exceptions from this call...
sessionId = mapClient.CreateSessionEx(mqSession);
```

```
mapImage1 = mapClient.GetMapImageFromSession(sessionId);  
// Here, get the image URL using HTTP. Or, we could use  
// GetMapFromSessionURL instead of GetMapImageFromSession
```

For more information about downloading and displaying a map image, see “Downloading Map Images” on page 55. For full sample code with detailed code comments, see the `MapIt` sample code project.

Creating Stateless Maps

Applications can display maps without maintaining MapQuest server sessions. Learn more about the advantages and limitations of this approach in “Stateless Maps for Web Applications” on page 47.

Desktop and Java applications can immediately download a map described by a local session object if desired. For details, refer to the next section, “Downloading Map Images.”

In contrast, Web applications need to expose an image URL within an HTML image tag. Web applications that do not use server sessions typically encode their session data into a special kind of map URL called a *direct URL*. This URL embeds all information that would be stored in a MapQuest server session. A direct URL works for extended periods of time because it does not rely on a server session that could expire.

Create direct URLs with the client object method `GetMapDirectURLEx`, which takes a local session object. Be sure to customize your map by adding any desired session parameters to the local session object before calling `GetMapDirectURLEx`.

It is critical to understand that stateless maps do **not** use MapQuest server sessions, but your application must create a *local session object*, which is an instance of the class `Session`.

The basic steps to create stateless maps are:

1. Create an object of class `Session`.
2. Create a `MapState` object, and set its properties representing size, scale, and center point.
3. Add the `MapState` object to the session using the session’s `AddOne` method.
4. Add any other session parameter objects desired. For other options, see Table 9, “Session Objects,” on page 44.
5. Web applications would get a map URL with the `GetMapDirectURLEx` client object method and expose the URL in an HTML image tag. Other applications might get the image data with `GetMapImageDirect`.

For complete code examples using this approach, see the `MapIt` sample code project in the code section identified as “Process 2.”

Downloading Map Images

Web applications typically get a map URL (not the map image directly) and export it within an HTML `` tag with the map URL as the image source. The user's Web browser will eventually request the image directly from the Advantage API server.

In contrast, desktop applications typically would immediately download the image to the Advantage API client application. There are two different client object methods that do this:

- If you use server sessions, call the `GetMapImageFromSession` client object method to get the image data.

If you use don't use server sessions, call the `GetMapImageDirect` client object method to get the image data.

In the ASP.NET interface, you can get the image from a map URL using code like:

```
mapURL2 = MapClient.GetMapDirectURLEx(mqSession,DisplayState)
```

Alternatively, you can use the client object methods `GetMapImageDirect` and `GetMapImageFromSession` to immediately store image data into a byte array. The .NET version of the MapIt sample code uses `GetMapImageDirect` and `GetMapImageFromSession`.

Customizing Map Format, Resolution, and Aliasing

The `DisplayState` class lets you customize the map data format, display resolution of the map, and anti-alias settings.

If you want to use a map data format other than the default GIF format, create a `DisplayState` object and set its `ContentType` property. The map format choices are GIF (Graphical Interchange Format), PNG (Portable Network Graphics), Wireless Bitmap (WBMP), EPS (Encapsulated PostScript), and Adobe Illustrator EPS. Please refer to "Mapping Constants" on page 64 for the exact syntax to use within your client interface.

If your display resolution is not 72 dots per inch, set the `DPI` property in a `DisplayState` object with the correct resolution.

You can also change the anti-alias settings for a map using properties in this class. The default setting is to use anti-aliasing, which increases readability in many situations. To disable this behavior, set the `AntiAlias` property to `false`.

To use a `DisplayState` object, pass it to the client object methods that generate map URLs or download maps: `GetMapFromSessionURL`, `GetMapDirectURLEx`, `GetMapImageFromSession`, and `GetMapImageDirect`.

There are APIs that take a `DisplayState` object as a required or optional parameter, such as client object methods `LLToPix` and `PixToLL`. If you change the resolution to something other than 72 DPI, you **must** pass a `DisplayState` with the correct DPI to methods with a `DisplayState` parameter. This is true even when the parameter is listed as optional in the API Reference; the `DisplayState` object defines the display resolution for display coordinate calculations.

Note that if you increase the display resolution or use a vector-based image format like EPS, your maps will look more professional if you use only *drawing icons* (vector symbols) for any icons on your map.

Customizing Map Appearance

There are several ways to customize the appearance of map features other than displaying Points of Interest and Map Annotations, which are discussed in other chapters.

Developers can use APIs to override style pools on the server using *style objects*:

- Override styles for point features of one display type using the `DTStyle` class. This is the most common style object.
- Override styles of all feature types using the `DTStyleEx` class by using style settings created by the Style File Editor application.
- Override styles by identifying features by name or unique ID using the `DTFeatureStyleEx` class.

You can use zero, one, or more than one of each type of style object. To use them, you will add them to a `CoverageStyle` object, which serves as a collection of style objects.

Do not add multiple style objects with conflicting or potentially conflicting meaning to the same `CoverageStyle` object. When it cannot be helped, note that the server processes style objects in the order within the collection, overriding previous style changes.

Advantage API customers can rely only on the fonts "Helvetica" and "Lucida" on MapQuest-hosted mapping servers.

Note that text is not rendered on the server for non-bitmap image formats such as EPS. Because of this, all fonts in maps must be available on whatever computer displays or prints such images.

Overriding Styles With `DTStyle`

You can override style pool information in a map for all features of a display type (DT) using a `DTStyle` object. You can modify their icons, colors, text label fonts, label font sizes, label background colors, or visibility (to temporarily hide or show features).

`DTStyle` objects override the most common style attributes for point features. You can use `DTStyle` objects with line features and polygon features to set visibility (the `Visible` property) on or off, but no other line or polygon modifications are possible with `DTStyle`.

If you need to modify less common style attributes or if you need to modify line or polygon styles, you should instead use `DTStyleEx` or `DTFeatureStyleEx`.

Like other style object classes, `DTStyle` objects describe changes from styles defined in the server's style pools, including the appearance of developer-assigned DTs.

If you need two map features to have different icons, different colors, or different text label fonts, font sizes, or background colors, use different DTs for the features.

You can restrict modifications to take effect only at certain map scales. For example, you could make certain display types a brighter color within a given scale range, or even change visibility within a given scale range. You would do this by setting the `LowScale` and `HighScale` properties within the `DTStyle` object.

You must add the configured `DTStyle` object to a `CoverageStyle` object, which is essentially a collection of style objects. You then add the `CoverageStyle` object to your local session object.

When setting an icon, you must set the symbol name in the `SymbolName` property and the symbol type in the `SymbolType` property. The two symbol types are raster (*bitmap icons*) and vector (*drawing icons*).

Advantage API customers can use the default icons on the server described in Appendix B, "Standard Icons." If you want to use your own icons, you can create icons with the GMF Draw and GRF Edit tools included with the product. Send the results from those tools to Technical Support to install. Alternatively, if you want the icon centered on its representative point, you can send a standard GIF format image to Technical Support. Your code will specify an image name supplied by Technical Support.

You can customize the text label font and appearance by setting the properties `FontBoxBkgdColor`, `FontBoxMargin`, `FontBoxOutlineColor`, `FontColor`, `FontName`, `FontOutlineColor`, `FontSize`, `FontStyle`, and `LabelVisible`.

Here is a code snippet that illustrates modifying styles for display type (DT) 3072:

```
` This assumes we've set up a session object and MapState
MQClientInterface.DTStyle pointDTStyle = new
MQClientInterface.DTStyle();

pointDTStyle.DT = 3072; //the display type to change!

pointDTStyle.SymbolType = MQClientInterface.SymbolType.VECTOR;
pointDTStyle.SymbolName = "MQ00031";
pointDTStyle.LabelVisible = true;
pointDTStyle.Visible = true;

MQClientInterface.CoverageStyle coverageStyle = new
MQClientInterface.CoverageStyle();
```

```
coverageStyle.Add(pointDTStyle);  
mqSession.AddOne(coverageStyle);
```

For more complete examples, see the sample code `MapItWithPOI`.

Please refer to “Mapping Constants” on page 64 for the exact syntax to use for symbol types within your client interface.

If you are using MapQuest server sessions, remember to update the server session from the local session object using the `UpdateSessionEx` client object method before downloading a map from a URL or with `GetMapImageFromSession`.

Overriding Styles With `DTStyleEx`

If you want to modify feature types other than points or you want to use complex style descriptions, you can use the Style File Editor application to design complex styles to use with the `DTStyleEx` class. The Style File Editor application can export complex styles as *map style strings*.

The `DTStyleEx` class has all the functionality described for `DTStyle`, and additionally can modify or add styles for point, line, and polygon map features using map style strings.

You can restrict modifications to take effect only at certain map scales. For example, you could make certain display types be a brighter color at a given scale range, or even change visibility within a given scale range. Do this by setting the `HighScale` and `LowScale` properties within the `DTStyleEx` object.

You must add new `DTStyleEx` objects to a `CoverageStyle` object, which is essentially a collection of style objects. You would then add the `CoverageStyle` object to your local session object.

To achieve the same style effect seen in the Style File Editor:

1. Design a style using the Style File Editor application.
2. Use the mouse to right-click on the style.
3. Export the string to the clipboard.
4. In your code, create a `DTStyleEx` object.
5. Paste the string into your code in the style object's `StyleString` property. If you used any font names or symbol (icon) names, there will be quote signs within the style string. Most programming environments require you to appropriately “escape” the quote characters. Depending on the environment, this is done by replacing the quotes with two quotes (" ") or prepending the quote signs with a backslash (\ "). Failure to escape your quote characters will prevent your code from properly compiling.
6. Add the style object to a new `CoverageStyle` object.
7. Add the `CoverageStyle` object to your session object.

Here is an example of using a DTStyleEx object:

```
`This assumes we've set up a session object and MapState
MQClientInterface.DTStyleEx style = new
MQClientInterface.DTStyleEx();
style.DT = 1043;
style.StyleString = "Visible True Polygon Brush Color 255,0,0";

MQClientInterface.CoverageStyle coverageStyle = new
MQClientInterface.CoverageStyle();

coverageStyle.Add(style);
mySessionObject.addOne(coverageStyle);
```

```
CMQDTStyleEx* pStyle = new CMQDTStyleEx();
pStyle->SetDT(1043);
pStyle->SetStyleString("Visible True Polygon Brush Color 255,0,0");

CMQCoverageStyle* pStyles = new CMQCoverageStyle();

pStyles->Add(pStyle);
mySessionObject->AddOne(pStyles);
```

If you use MapQuest server sessions, remember to update the server session from the local session object using the UpdateSessionEx client object method before downloading a map from a URL or with GetMapImageFromSession.

Overriding Styles With DTFeatureStyleEx

The DTFeatureStyleEx class is very similar to DTStyleEx, but lets developers specify what map features to override by providing its name or unique ID (GEF ID) within a map data set.

You must add the configured DTFeatureStyleEx object to a CoverageStyle object, which is essentially a collection of style-modifying objects. You would usually add the CoverageStyle object to your local session object.

Here is a code snippet illustrating how to change the style of features by name:

```
' This assumes we've set up a session object and MapState
MQClientInterface.DTFeatureStyleEx style = new
MQClientInterface.DTFeatureStyleEx();
style.DT = 1043;
style.StyleString = "Visible True Polygon Brush Color 255,0,0";

FeatureSpecifier FS = new
MQClientInterface.FeatureSpecifier();
FS.AttributeType = MQClientInterface
.FeatureSpeciferAttributeType.NAME;
FS.AttributeValue = "Ohio River"
style.FeatureSpecifiers.add(FS);

MQClientInterface.CoverageStyle coverageStyle = new
MQClientInterface.CoverageStyle();

coverageStyle.Add(style);
mySessionObject.addOne(coverageStyle);
```

```
CMQDTFeatureStyleEx* pStyle = new CMQDTFeatureStyleEx();
pStyle->SetDT(1043);
pStyle->SetStyleString("Visible True Polygon Brush Color 255,0,0");

CMQFeatureSpecifier* pF = new CMQFeatureSpecifier();
pF->SetAttributeType(CMQFeatureSpeciferAttributeType::NAME);
pF->SetAttributeValue("Ohio River");
pStyle->GetFeatureSpecifiers().Add(pF);

CMQCoverageStyle* pStyles = new CMQCoverageStyle();

pStyles->Add(pStyle);
mySessionObject->AddOne(pStyles);
```

If you are using MapQuest server sessions, remember to update the server session from the local session object using the `UpdateSessionEx` client object method before downloading a map from a URL or with `GetMapImageFromSession`.

For more details about style strings, please refer to the section “Overriding Styles With DTStyleEx” on page 58.

User Interaction In Maps

Map Zooming APIs

If you use MapQuest server sessions, change zoom settings easily using simple client object methods: `ZoomIn`, `ZoomOut`, `ZoomXY`, `ZoomToRectangleXY`, and `ZoomToRectangleLL`.

All these change the map scale without changing the center point. The `XY` versions use pixel coordinates and the `LL` versions use latitude and longitude. After using these methods, re-download a map from its URL or use the client object method `GetMapImageFromSession` again.

If you need to update other session parameters at the same time, consider using the `UpdateSessionEx` client object method. Before calling `UpdateSessionEx`, change the map scale in your session's `MapState` object.

Alternatively, you can add a map command that changes map scale (see “Using Map Commands” on page 62) and then call `UpdateSessionEx`.

Zooming Without Server Sessions

If you are not using MapQuest server sessions, there are two ways to change a map's scale. One way is to set the new map scale in your session's `MapState` object and then call the client object method `UpdateSessionDirect` to update your session.

The other way is to add a map command (see “Using Map Commands” on page 62) to your local session object and then call `UpdateSessionDirect`.

In both cases, either regenerate your map image URL using `GetMapDirectURLEx` or re-download the map using `GetMapImageDirect`.

Map Panning and Centering APIs

If you are using MapQuest server sessions, change a map's center point easily using client object methods:

- To center the map to a latitude and longitude, use the `Center` method with the latitude and longitude parameters. This method in the .NET interface will detect the type of parameters that you pass to this method.
- To enable the user to re-center a map by clicking on a new center, determine the X/Y coordinates of the click and then use the `Center` method with X and Y parameters. This method in the .NET interface will detect the type of parameters that you pass to this method.
- To pan with a pixel coordinate offset, use the `Pan` method, which takes pixel coordinate offsets. This approach is useful for user interfaces with clickable arrows to pan up, down, left, or right.

After using any of these methods, re-download a map from its URL or use the client object method `GetMapImageFromSession` again.

If you need to update other session parameters at the same time, consider using the `UpdateSessionEx` client object method instead. Before calling `UpdateSessionEx`, change the center point in your session's `MapState` object.

Alternatively, you can add a map command that changes map center point (see “Using Map Commands” on page 62) and then call `UpdateSessionEx`.

Panning With Stateless Maps

If you are not using MapQuest server sessions, there are two ways to define a new center point. One way is to change the new center point in your session's `MapState` object and then call the client object method `UpdateSessionDirect` to update your session.

The other way is to add a map command (see “Using Map Commands” on page 62) to your local session object and then call `UpdateSessionDirect`.

In both cases, either regenerate your map image URL using `GetMapDirectURLEx` or re-download the map using `GetMapImageDirect`.

Using Map Commands

All developers can modify map scale and center point by adding *map commands* to the local session object and then updating their session.

All map commands affect the map state and cause map data selection rules to be examined, which could change the map coverage and style pool. Because of this, a map command will not take effect **until you update the session**. Updating the session gives the server a chance to re-calculate the map state, including the center point, map scale, coverage name, and style pool.

To use a map command, first instantiate a map command subclass: `BestFit`, `BestFitLL`, `Center`, `CenterLL`, `Pan`, `ZoomIn`, `ZoomOut`, `ZoomToRect`, or `ZoomToRectLL`. Add this object to your local session object using the session's `AddOne` method.

If you use MapQuest server sessions, update your session with the `UpdateSessionEx` client object method. Remember to call the client object method `GetMapFromSessionURL` to get a new map image or download the map image with `GetMapImageFromSession`.

If you are not using MapQuest server sessions, update your session with the `UpdateSessionDirect` client object method. Remember to call the client object method `GetMapDirectURL` to get a new map image or download the map image with `GetMapImageDirect`.

After the server processes a map command, the MapQuest API destroys and removes the map command from its local session object.

You can only add one map command to a session at a time. Process a new map command by updating the session before adding another map command.

Using the BestFit Classes

One of the most complex map commands is the `BestFit` class. It changes the map scale and center point to “best fit” a set of features and/or map annotations.

After adding POIs in a `FeatureCollection` (discussed more in Chapter 5, “Displaying POIs”) to a session, create a new `BestFit` map command object, set its `ScaleAdjustmentFactor` property to 1.2 and the `DTs` property to your POI display types in a collection of class `DTCollection`.

If you want to add map annotations (discussed more in Chapter 6, “Map Annotations”) to a map, you can “best fit” around those shapes in addition or instead of feature DTs. Create a new `BestFit` map command object, set its `ScaleAdjustmentFactor` property to 1.2, and set its `IncludePrimitives` property to true.

In both cases, you will finally add the map command to the session object using the session’s `AddOne` method and then update the session as mentioned above.

There is also a `BestFitLL` class, which is similar to the `BestFit` class. `BestFitLL` allows you automatically scale and center a map to fit an arbitrary set of locations stored in a `LatLngCollection`. It is useful when the locations used for `BestFit` calculations are **not** visible on the map (neither POIs nor map annotations).

Both `BestFit` and `BestFitLL` classes have a `KeepCenter` property. When set to true, the map command changes the map scale to include all points but does not change the map’s center point.

Both `BestFit` and `BestFitLL` classes have a `SnapToZoomLevel` property. If you are you using automatic map data selection and this property is set to true, the map command performs the `BestFit` command as described above and then adjusts the map scale to “zoom out” to the next zoom level. To customize zoom level map scales, refer to “Map Data Selection Classes” on page 49. This property has no effect if the map does not use automatic map data selection.

See also: Table 9, “Session Objects,” p. 44.

Other Point-and-Click APIs

There are other user interaction APIs that require you to determine the X/Y coordinates of the user interaction, typically the location of a mouse click. The way to get mouse click coordinates varies in every development environment. Consult your development environment documentation for more information.

To enable the user to click on a map to identify latitude and longitude, use the `PixToLL` method of your client object. `PixToLL` converts a collection of X/Y map coordinate points in a `PointCollection` to a collection of locations in a `LatLngCollection`.

Once you have latitude and longitude coordinates, you might want to search for information near that location:

- To identify visible map features and POIs, refer to “Identifying Map Objects by Coordinates” on page 70.
- To search for map features or POIs that may include objects not currently visible on a map, refer to Chapter 7, “Proximity Searching.”
- To estimate a street address, refer to “Reverse Geocoding” on page 36.

Advanced Point-and-Click APIs

Advanced programmers can get a road’s unique ID from mouse clicks using the `GetRoadGefIdXY` client object method. If you have latitude and longitude coordinates for a location, get a nearby road segment unique ID using the `GetRoadGefIdLL` client object method.

A map feature’s unique ID is only unique within one data vendor, not across data sets from multiple vendors. If you use the above method, you will need to know which map data set the map feature comes from, which may not be obvious if you use map data selection. Update the session with the client object method `UpdateSessionEx`, get the session data with the client object method `GetSessionEx`, and then get the map coverage name in the `MapState` object’s `CoverageName` property.

Mapping Constants

.NET developers should use the mapping constants shown below using the syntax required by the .NET interface.

Map Image Data Formats

These constants are used to override the default GIF image type. Set a `DisplayState` object’s `ContentType` property to one of the following values:

```
MQClientInterface.ContentType.AIEPS  
MQClientInterface.ContentType.EPS  
MQClientInterface.ContentType.GIF  
MQClientInterface.ContentType.PNG  
MQClientInterface.ContentType.WBMP
```


Limiting Map Data Selection By Vendor

These constants are used to override the data used by map data selection. Use these values with AutoMapCovSwitch objects, as discussed in “Map Data Selection Classes” on page 49.

```
MQClientInterface.DataVendorCode.USAGE_INCLUDE
MQClientInterface.DataVendorCode.USAGE_EXCLUDE

MQClientInterface.DataVendorCode.AND
MQClientInterface.DataVendorCode.CA
MQClientInterface.DataVendorCode.CRITCHLOW
MQClientInterface.DataVendorCode.DMTI
MQClientInterface.DataVendorCode.ETAK
MQClientInterface.DataVendorCode.GDT
MQClientInterface.DataVendorCode.MQ
MQClientInterface.DataVendorCode.NT
MQClientInterface.DataVendorCode.TA
MQClientInterface.DataVendorCode.TIGER
MQClientInterface.DataVendorCode.UNKNOWN
MQClientInterface.DataVendorCode.VOYAGER
MQClientInterface.DataVendorCode.LEADDOG
MQClientInterface.DataVendorCode.MDS
```

DTFeatureStyleEx Match Attributes

These constants are used to set properties for FeatureSpecifier objects, which are required to use the special features of the DTFeatureStyleEx class.

```
MQClientInterface.FeatureSpeciferAttributeType.GEFID
MQClientInterface.FeatureSpeciferAttributeType.NAME
```

5 Displaying POIs

You can add custom locations called Points of Interest (POIs) to your map. You can create custom icons for these locations on your maps and/or search for them using the proximity searching API. This chapter describes the APIs to add POIs to maps.

This chapter assumes that you have already read Chapter 4, “Basic Mapping.” If you have not done so, please do so now.

You can add two types of POIs to your maps:

- **Simple POIs.** Define each POI’s coordinates and an optional label. This type of POI includes results of proximity searches. This is the recommended POI approach.
- **Database POIs.** Store POI location data in a database and request that the server add some or all of these POIs to your map. The server searches the database to find POIs that appear within the map’s bounding rectangle. Database POIs are convenient to implement but are usually slower and more resource intensive than using simple POIs.

To perform searches for POIs within a circle, rectangle, or a corridor around a driving route, see Chapter 7, “Proximity Searching.”

NOTE: Some map coverages include features that people might consider POIs, like parks or national monuments, but they are implemented as standard map features.

Finding and Creating POI Icons

There are two types of icons supported by Advantage API:



Bitmap icons. These are also known as raster symbols.



Drawing icons. Use these for large icons or unusually high map resolutions. These icons are also called vector symbols.

You can easily use default icons such as stars, pushpins, and start/end markers. For the full list, refer to Appendix B, “Standard Icons.” You can view a few of these standard icons by running the sample code projects `MapItWithPOI` and `SearchAndDisplayIt`.

You can also create custom icons of either type, although bitmap icons are easier to create. If you want very large icons or unusually high map resolutions, use drawing icons. Advantage API customers can create both types of icons using the following tools that have their own separate documentation:

- **GRFEdit.** This tool prepares custom bitmap images to be bitmap icons by creating special image headers. This tool creates image headers as files ending with “.GRF”. Your code will specify the image as the symbol type `Raster`.
- **GMFDraw.** This tool creates drawing icons as files ending with “.GMF”. Your code will specify the server image as the symbol type `Vector`.

If you want to use your own icons with Advantage API, create one of the icon types mentioned above and send them to Technical Support to install. Alternatively, you can create one or more small GIF format images and send them to Technical Support.

Basic POI Classes

There are MapQuest classes that you need to know about to display POIs on maps. As mentioned in “Sessions Overview” on page 43, you can add *session parameter objects* to your local server session object that modify map appearance.

For each *simple POI* (individually defined POI), you will create a `PointFeature` object, which contains the location’s coordinates, the location’s display type, and an optional text label. You will add all such objects to `FeatureCollection` object, which you will usually add to your local session object using its `AddOne` method.

If you are displaying your POIs from a database query, you will **not** create `PointFeatures` to display each database POI. However, developers who use database POIs typically add one or more simple POIs for other reasons, such as displaying a star to indicate the user’s current location, the anchor for a recent search, or a geocoded address.

For each *display type*, you will create a *style object* (usually a `DTStyle` object) that specifies an icon, a text label, and text label appearance. Add all style objects to `CoverageStyle` object, which is essentially a collection of style objects. You will then add the `CoverageStyle` object to your local session object using the session’s `AddOne` method.

The two symbol types are raster (*bitmap icons*) and vector (*drawing icons*). In general, bitmap icons are easier to create. If you want very large icons or unusually high map resolutions, use drawing icons.

Typically, POIs share an icon image when they are the same general type. In such a case, you would assign a display type number to POIs of that type and configure one `DTStyle` object with the style information. If you need to assign new display types for point features, use the developer range 3072 through 3583. For other options, refer to Appendix A, “Display Types.”

Let’s use an example. Suppose a national monument Web site wanted to show four recommended hotels. The application would typically display five POIs: a star for the monument and four hotel icons. In this case, the monument would be one display type and the four hotels would be another display type. That application would create five `PointFeature` objects (one for each POI) and two `DTStyle` objects (one for each display type).

See also: “Display Types and Map Styles,” p. 42; “Customizing Map Appearance,” p. 56.

Displaying Simple POIs

See the previous section for an overview of the simple POI classes.

In order to create a simple POI, create a `PointFeature` object, set its `DT` property to the display type, and then set its latitude and longitude coordinates in its `CenterLatLng` property, which contains a `LatLng` object.

Here are the basic steps to add simple POIs to a map:

1. Create a `DTStyle` object.
2. Configure it with a display type and symbol details.
3. Add the `DTStyle` to a new `CoverageStyle` object.
4. Create a `PointFeature` object.
5. Set the point’s latitude and longitude.
6. Set the point’s display type.
7. Create a new `FeatureCollection` object.
8. Add the `PointFeature` to the `FeatureCollection`.
9. To create POIs, create more point features and add them to the feature collection.
10. Add the `FeatureCollection` to your local session object.
11. Generate the map URL or immediately download the image data as discussed in previous sections.

Here is a sample code snippet illustrating this technique with one simple POI at the center of the map, a common user interface to highlight a geocoded location:

```
// this assumes mapState & map data selection are set up
MQClientInterface.FeatureCollection featureCollection = new
MQClientInterface.FeatureCollection();

MQClientInterface.PointFeature pointFeature = new
MQClientInterface.PointFeature();

MQClientInterface.DTStyle pointDTStyle = new MQClientInterface.DTStyle();

pointDTStyle.DT = 3072; // the display type of the point!

pointDTStyle.SymbolType = MQClientInterface.SymbolType.VECTOR;

pointDTStyle.SymbolName = "MQ00031"; // standard red star

pointDTStyle.LabelVisible = true;
pointDTStyle.Visible = true;

MQClientInterface.CoverageStyle coverageStyle = new
MQClientInterface.CoverageStyle();

coverageStyle.Add(pointDTStyle);
pointFeature.DT = 3072; // must match the style object DT!

pointFeature.Name = "Hello";

//In this example, a point is displayed at the map center
pointFeature.CenterLatLng = mapState.Center;

featureCollection.Add(pointFeature);

mqSession.AddOne(featureCollection);
mqSession.AddOne(coverageStyle);
```

See the MapItWithPOI sample code for more complete code examples in each development environment.

Templates for Using Multiple Tables in a Single Database

If the MapQuest Advantage Enterprise server is configured with a DBPool that has a format/table layout identical to that of other tables not specifically called out in the server configuration, the following templated mechanism may be used to access the other tables singularly or together.

The code examples below use the DBLayerQuery object in VB.NET. Only the values set in the property really matter for these examples so they will be similar in any language.

```
Dim DBLayerQuery As New MQClientInterface.DBLayerQuery
```

Example 1 (querying the default table specified in the mqserver.ini):

'Use DBPool MQDATA to query the default table as configured on the server.'

DBLayerQuery.DBLayerName = "MQDATA"

Example 2 (querying another table contained in the database):

'Use DBPool MQDATA to query the table called 'hotels' instead of default table.

DBLayerQuery.DBLayerName = "MQDATA.hotels"

Example 3 (querying multiple tables contained in the database):

'Use DBPool MQDATA to query the tables 'hotels' , 'restaurants' and 'locations' in turn until the max matches requirement has been satisfied.

DBLayerQuery.DBLayerName = "MQDATA.hotels.restaurants.locations"

NOTE: A single query is built using the fields specified. Therefore, if doing a query on multiple tables, any Extra Criteria specified for the query

(i.e. DBLayerQuery.ExtraCriteria = "(Variable1 = '1')"

must be contained in each of the multiple tables or the query will fail. If you wish to query on different extra fields in each table, use the method in Example 2 to query each table with the desired criteria.

Identifying Map Objects by Coordinates

You can allow users to click on a POI, a street, or other map feature to identify it. There are two basic APIs to do this: the client object methods `IdentifyFeature` and `Search`.

The Search API can search many types of data, can perform four types of geographic searches, and can find map objects that are **not** currently displayed. Search can be used without the mapping API or MapQuest server sessions. The Search API is the subject of Chapter 7, “Proximity Searching.”

In contrast, the client object method `IdentifyFeature` is simple, limited in features, and requires that you use MapQuest server sessions. It is typically used to identify which POI was clicked by a user. However, it can also be used to search licensed map data sets for street names or other map features.

For example, a user could click on a POI that represents a business and an application could display the business address and phone number; another application might allow users to click on a street to display its name.

To use `IdentifyFeature`, first define the search criteria within an `IdentifyCriteria` object. These objects specify the center point coordinates for the search, the radius to search, and the maximum matches to return.

To limit the search to specific display types, create a `DTCollection` object and add all display types you want to find. You will pass the collection as a parameter to `IdentifyFeature`.

You can provide an empty `DTCollection`, which causes the server to search all display types that correspond to POIs and other point features; no roads, other line features, or polygons will be found.

If you want to search only standard road and bridge display types, add display types 1 through 511 to the collection. For other DT ranges, see Appendix A, “Display Types.”

The results are returned in a feature collection (class `FeatureCollection`), and you can iterate through the results.

If you are searching map data for built-in features and you do not carefully limit the display types to ones that are point features, results may be of the class `PointFeature`, `LineFeature`, or `PolygonFeature`.

The steps below identify a POI in response to a user pointing and clicking the map:

1. Create an `IdentifyCriteria` object.
2. Set its center point in X and Y coordinates.
3. Set the radius search distance.
4. Set the maximum number of features to be returned.
5. Perform the search using the client object method `IdentifyFeature`.
6. Check the size of the feature collection (the results). If the size is not zero, do something with each result.

Here is some code that illustrates this technique:

```
MQClientInterface.IdentifyCriteria criteria = new
MQClientInterface.IdentifyCriteria();
MQClientInterface.DTCollection dtcoll = new MQClientInterface.DTCollection();

MQClientInterface.FeatureCollection results = new
MQClientInterface.FeatureCollection();
criteria.Center = new MQClientInterface.Point(mouseClickX, mouseClickY));

criteria.Radius = 10;
criteria.MaxMatches = 10;
client.identifyFeature(sessId, criteria, results, dtcoll);
// Check the results collection...
```

If you use this API to identify streets and want to obtain the street's name, check the display type to confirm that it matches the display types of roads (1 through 511) before examining the feature's Name property.

See also: "Sessions Overview," p. 43; Chapter 7, "Proximity Searching," p. 93.

Using Points of Interest with Other APIs

There are several common ways developers use Points of Interest with other MapQuest APIs.

Customizing POI Appearance

You can customize the appearance of Points of Interest just like any other map feature. For instance, you can customize icons, colors, text label fonts, label font sizes, label background colors, or visibility (to temporarily hide or show features). Do this by creating style objects and adding them to your session's CoverageStyle object, which is essentially a collection of style objects.

The most common style object class is DTStyle, which overrides styles of all points of one display type. Using other classes, you can customize the appearance of display types and search for specific feature names and/or feature unique IDs (GEF IDs).

See "Customizing Map Appearance" on page 56 for details, or refer to the sample code project MapWithPOI.

Getting Features From a Map

There are two ways to get a list of features (including POIs) from a visible map.

1. To get the collection of features associated with a local session object, use the `GetObject` client object method to get its `FeatureCollection`, as discussed in “What Sessions Contain” on page 44. If you are using server sessions, you may need to first update your local session object with the `GetSessionEx` client object method.
2. To get all visible features from a map that uses MapQuest server sessions, use the client object method `GetDrawnFeatures`, which takes a session ID string and a feature collection to store the results. `GetDrawnFeatures` returns all previously-drawn point features, including developer-added POIs and point features from server map data.

The server must actually draw your session’s map with current settings for `GetDrawnFeatures` to return the correct set of points. It is **not** enough that the session has been updated and/or a map image URL created.

Also note that `GetDrawnFeatures` will not typically return all point features that could appear inside the map’s boundary rectangle. Some features have style settings that prevent them from drawing on a specific map. If you need a more thorough map data search, you should use the `Search` API discussed in Chapter 7, “Proximity Searching.”

Scaling and Centering a Map Around POIs

You can use the `BestFit` map command to scale and center a map to “best fit” a set of POIs. Please refer to “Using Map Commands” on page 62 for API details.

Displaying POIs From Search Results

You can programmatically search a geographic area for Points of Interest by defining a radius search, a rectangle search, a polygon search, or a corridor search around a path (typically routing results).

When you perform a proximity search, the search results are in the form of features within a `FeatureCollection` object. Although typically you will be interested in objects of class `PointFeature`, there are other types of features that you could get from a search.

If you searched only a database, all results will be of class `PointFeature`. If you searched licensed map data or feature collections that contained other feature types, results might be the class `PointFeature`, `LineFeature`, or `PolygonFeature`.

When you receive search results from the search API, display those features just as you would a collection of simple POIs, as discussed in the section “Displaying Simple POIs” on page 68.

See also: Chapter 7, “Proximity Searching,” p. 93.

Using Licensed POI Data With Searches

Developers can license a large set of Points of Interest (POIs) from MapQuest. This includes a variety of POIs from around the world. These include the most famous and popular tourist sites, government buildings, and other locations that may help orient your users. Please contact MapQuest Technical Support for NAVTEQ POI licensing information.

Like all other POIs with Advantage API, you must install the data into your database server with ODBC-compliant drivers.

Your application can use these POIs with MapQuest APIs for proximity searches. For instance, a city-owned Web site could show common landmarks to orient tourists in an unfamiliar city.

Here are some common ways to use licensed POI data:

- If your application geocodes an address, you could geocode the city’s center and then do a proximity search within 30 miles of the city center for POIs matching the user request.
- Display search results on a map using the APIs discussed in this chapter.
- Generate driving directions with the routing API using the latitude and longitude of the POI.
- Extract all the fields from the POI records and display them when a user clicks on them (or other user interface). Use the `GetRecordInfo` API described in “Independent Database Queries” on page 76.
- You can also use these POIs as a database layer, but you may only want to use them at a high scale (“zoomed in” to display a smaller area) to avoid displaying too many POIs displayed in each map, which would reduce server performance.

Advanced Points of Interest APIs

Using the Key Value in POIs

If you need to identify simple POIs after displaying them, you can store a unique identifier within each feature’s `Key` property. For POIs that were extracted from a database query, the API will automatically set this property to the key value as defined by the database pool.

This property has no inherent meaning to the MapQuest API or the server, but the value will be preserved, even when uploaded to the server and later retrieved using the `GetSessionEx` client object method.

Updating your local session object creates **new** session parameter objects, such as objects of class `MapState` and `FeatureCollection` that you added to the session. Because of this, you cannot simply save programming references to these objects and rely on them after an update. This affects all the following client object methods: `UpdateSessionEx`, `GetSessionEx`, `UpdateSessionDirect`, and the non-recommended methods `UpdateSession` and `GetSession`.

Because of this, you may need to find a POI in some way other than direct references after using one of these functions.

Set the `Key` property of special features to a special value. To find the special one(s), iterate over the objects in the `FeatureCollection` and get the `Key` property of each.

If you use the `Key` property, it is your responsibility to interpret the meaning of the property and to preserve uniqueness within a collection if that is required.

There is an additional property in all features called `SourceLayerName` that can be used in conjunction with the `Key` property. Since the key value is only unique within one data set, if you use multiple database tables for one search, you may want to distinguish which database table each result came from. The `SourceLayerName` property will contain the database table name.

If the found feature was from server map data, the `SourceLayerName` property will contain the map pool name.

If the found feature was a feature that you created yourself, the `SourceLayerName` property will be blank by default. You can set each feature's `SourceLayerName` property if desired before the search to help you process the results in conjunction with the `Key` property, which is also developer-assignable before each search.

See also: "Search Result Properties," p. 102.

Advanced Feature Types And DTs

You can add new features using APIs discussed in this chapter that are feature types other than points: polygon features and line features.

If you want to do this, the display type **must** match pre-defined ranges for each feature type. For a full list of DT ranges, see Appendix A, "Display Types."

For all feature types, you can choose to create features that match DTs of map features in licensed map data. This might be useful if you wanted to inherit default style behaviors for a new feature of a default

type. For instance, you could add a dirt road that isn't in the map data using a standard dirt road DT. Such custom roads will display, but are not used during routing or reverse geocoding.

For more information about built-in DTs, see Appendix A, "Display Types."

Before adding line and polygon features to a map, you might consider instead using drawing primitives, which are discussed in Chapter 6, "Map Annotations."

Independent Database Queries

You can create simple database queries that search or retrieve **any** record fields. All databases using this API must be configured on a Advantage API server in a database pool.

Developers typically use this API to get non-MapQuest database fields from records found using other APIs. For example, if a user clicks on a map POI that represents a store, the application might want to display the store's name, phone number, and store hours. Find the POI using the `IdentifyFeature` method, get the POI's *key value* (its unique ID within the database), and then finally do a simple database lookup on that record for all non-MapQuest fields such as phone number and store hours.

This a simple database interface, not a general purpose database solution. Never use `GetRecordInfo` with large SQL queries, large result sets, complex joins, or other complex queries. Large result sets are never supported and may cause memory problems on both the client application and the server.

To use this API, you should know a database pool name on the server and the database column/field names that you want to read.

You can optionally limit the search to specific records by *key field* values in the database table. The key field is defined on the server in the database pool.

You can optionally add additional database criteria if you can define it using SQL `WHERE` clauses. For example, to narrow the database query to a specific item with a specific value in the database column name `Type`, use an *extra criteria string* like `"Type=929"`. This string will be surrounded by parentheses and appended to the server-generated SQL query `WHERE` clause. For SQL syntax reference documentation, consult your database documentation or search the World Wide Web.

Advantage API customers should note that the key field is always the MapQuest-created record field called `Key`. This might be important if you need to look up details of a POI from its key value. If you instead want to search the record ID numbers that were provided as the first field in your Advantage API Data Manager input file, that is the database field `RecordID`. To find a record from this field, use an extra criteria string such as `"RecordID=12345"`.

Advantage API customers can use extra criteria strings with all user-defined database fields and the built-in database fields listed in "Database Connectivity in Advantage API" on page 16.

The steps for a typical independent database query:

1. Create an instance of `DBLayerQuery`.
2. Set its `DBLayerName` property to a database pool name.
3. If desired, set its `ExtraCriteria` property to an SQL fragment.
4. Create an instance of `StringCollection` for the database field names. For instance, if you wanted to get the application-specific field names `ID`, `X`, and `Y`, create a string collection containing the 3 strings `"ID"`, `"X"`, and `"Y"`. Leave the string collection empty to get all fields from the record.
5. Create a new instance of `RecordSet` for the results.
6. Create a new instance of `StringCollection` for record IDs, whether or not you intend to search for specific record IDs.
7. To search for specific records by *key field* values in that database pool, add the values to the new `StringCollection`. Add the values as strings even if the values represent integers or other numbers. For instance, to search for records IDs 9 and 29, add the strings `"9"` and `"29"`.
8. Call the client object method `GetRecordInfo` with the field names collection, the database query, and the two string collections.
9. Use the `RecordSet` attribute `EOF` to check whether there are results. If it returns false, there are more results.
10. Use the `RecordSet` method `EOF` to check whether there are more results. If it returns false, there are more results.
11. Use the `RecordSet` method `IsEOF` to check whether there are results. If it returns false, there are more results.
12. In the return `RecordSet` object, use the method `GetField` with a field name to get a field value for one or more fields.
13. Use the `RecordSet` method `MoveNext` to go to the next result.
14. Use the `RecordSet` method `IsEOF` to check whether there are more results. If it returns false, there are more results.

See the API Reference for other `RecordSet` class details like the methods `MoveFirst`, `MoveLast`, and attribute `BOF` (“is beginning of file”).

Catch any exceptions thrown by `RecordSet` methods in case your application accidentally moves beyond valid records or gets fields when no records exist.

POI Constants

.NET developers should use the POI constants shown below using the syntax required by the .NET interface.

```
MQClientInterface.DBFieldType.VARCHAR // String Field
MQClientInterface.DBFieldType.NUMERIC // Number Field
```

Symbol Types

These constants are used with style objects, such as `DTStyle`, as discussed in “Basic POI Classes” on page 67.

```
MQClientInterface.SymbolType.RASTER // Bitmap Icon
MQClientInterface.SymbolType.VECTOR // Drawing Icon
```

Colors

These constants are used for color properties of style object classes, as well as drawing primitive classes (mentioned in Chapter 6, “Map Annotations”).

```
MQClientInterface.ColorStyle.BLACK
MQClientInterface.ColorStyle.BLUE
MQClientInterface.ColorStyle.CYAN
MQClientInterface.ColorStyle.DARK_GRAY
MQClientInterface.ColorStyle.GRAY
MQClientInterface.ColorStyle.GREEN
MQClientInterface.ColorStyle.LIGHT_GRAY
MQClientInterface.ColorStyle.MAGENTA
MQClientInterface.ColorStyle.ORANGE
MQClientInterface.ColorStyle.PINK
MQClientInterface.ColorStyle.RED
MQClientInterface.ColorStyle.WHITE
MQClientInterface.ColorStyle.YELLOW
MQClientInterface.ColorStyle.INVALID_COLOR
```

Text Drawing Characteristics

These constants are used to set text properties for `DTStyle` classes as well as drawing primitives (mentioned in Chapter 6, “Map Annotations”).

```
MQClientInterface.FontStyle.NORMAL
MQClientInterface.FontStyle.BOLD
MQClientInterface.FontStyle.BOXED
MQClientInterface.FontStyle.OUTLINED
MQClientInterface.FontStyle.ITALICS
MQClientInterface.FontStyle.UNDERLINE
MQClientInterface.FontStyle.STRIKEOUT
MQClientInterface.FontStyle.THIN
MQClientInterface.FontStyle.SEMIBOLD
MQClientInterface.FontStyle.INVALID
```

6 Map Annotations

You can draw simple objects above or within a map's layers to highlight map features, proximity search parameters, or other custom data. For example, you could display the search radius used for a proximity search, demarcate boundaries of an important geographic region absent from map data, identify map and sub-map regions, or dramatically notate a map feature.

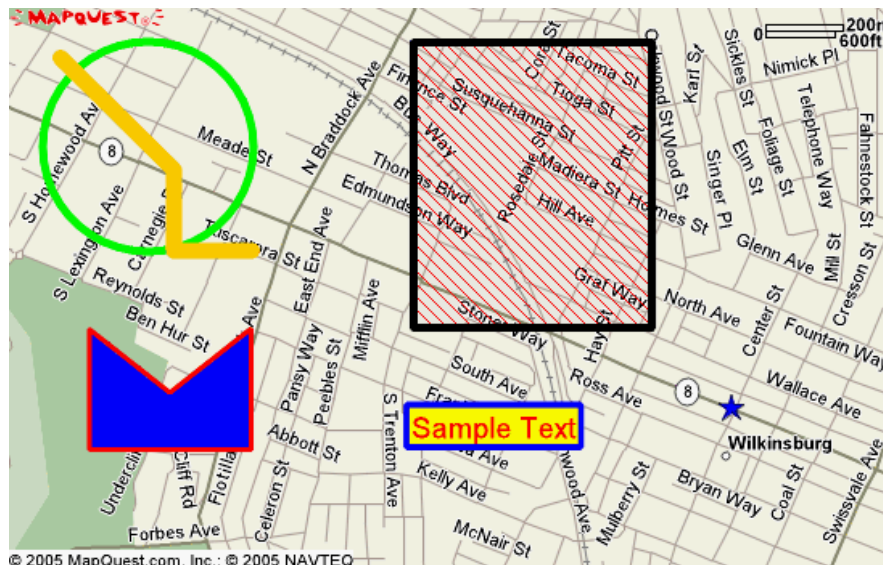
If you merely want to add small icons and/or text at a specific latitude and longitude, use POIs for added functionality. See Chapter 5, "Displaying POIs."

Drawing API Overview

You can display drawing objects called *drawing primitives* on top of other map information or embed these drawing objects between specific visual layers of map information. For instance, you could draw a polygon "on top of" streets but "below" highlighted driving directions.

Drawing primitives include ellipses (including circles), rectangles (including squares), icons, lines, polygons, and text. See the figure "Map Annotations In Multiple Drawing Orders" for examples. There is a different MapQuest object class for each type of drawing primitive.

Map Annotations
In Multiple
Drawing Orders



If your map annotations represent proximity to real-world locations, they move with the map if the user scrolls/pans the map or changes the map scale. Specify this drawing primitive as a *geographic coordinate type*, which is the default.

If the map annotations are independent of real-world locations, they might stay at the same pixel position if the user pans the map or changes the map scale. For example, if you always display your company logo in the upper right of the map, you can specify this drawing primitive as a *display coordinate type* by setting its `CoordinateType` property to `Display`. Please refer to “Map Annotation Constants” on page 90 for the exact syntax to use within your client interface.

In both cases, you can express the initial position using either X/Y screen coordinates or as latitude and longitude.

You may set each primitive’s opacity (transparency) in its `Opacity` property. Valid settings are from value 0 (invisible) to 255 (solid, the default).

Map Annotation Classes

You will instantiate one object for each drawing primitive using these classes:

Single lines or multi-segment lines:	<code>LinePrimitive</code>
Ellipses and circles:	<code>EllipsePrimitive</code>
Bitmap icons and drawing icons:	<code>SymbolPrimitive</code>
Text:	<code>TextPrimitive</code>
Rectangles and squares:	<code>RectanglePrimitive</code>
Polygons:	<code>PolygonPrimitive</code>

After you have created all your drawing primitives, add them to a collection of class `PrimitiveCollection` using the collection’s `Add` method.

`PrimitiveCollection` objects are *session parameter objects*, so add the collection to your local session object with your session’s `AddOne` method.

See also: “What Sessions Contain,” p. 44; “Basic Mapping API,” p. 53.

Map Annotation Drawing Order

Map annotations appear on top of all other layers by default. You can override this behavior and draw objects before or after certain layers of map data.

Map objects are drawn in general groupings called *map layers*, which are drawn in a predefined order, listed below in background to foreground order. Note that these layers **do not** inherently include drawing primitive classes that have similar names or appearance:

- **Polygons.** Includes rivers, parks, and urban area boundaries.
- **Route Highlights.** If you use the routing API and use MapQuest server sessions, the route's shapes can be displayed in this special map layer.
- **Roads.** Includes roads, highways, ferry lines, and pedestrian paths.
- **Icons.** Includes road shields and POI icons.
- **Text.** Includes road labels, city names, and POI labels.

Override the drawing order of a drawing primitive by setting its `property` to one of the following `DrawTrigger` constants: `AfterPolygons`, `AfterRouteHighlight`, `AfterText`, `BeforePolygons`, `BeforeRouteHighlight`, or `BeforeText`.

Please refer to “Map Annotation Constants” on page 90 for the exact syntax to use for these constants within your client interface.

You cannot explicitly place drawing primitives before or after road or icon layers. If this is what you want, choose either `AfterRouteHighlight` or `BeforeText`. Even if no route highlight is present, you can use the constants for `AfterRouteHighlight` and `BeforeRouteHighlight`.

Using Map Annotation Classes

Drawing a Line Primitive

To draw a single line or multi-segment line, create a `LinePrimitive` object and add it to your collection of drawing primitives in your `PrimitiveCollection`.

The `LinePrimitive` object defines the primitive's color, the line width (in thousands of an inch), and where to draw the line. The location is expressed either as X/Y screen coordinates, or as latitude and longitude. If you add more than two locations to a line primitive, the server will draw additional line segments to locations in the order within the collection.

To define the location and size, you will first get the primitive's collection of points, which is done differently if you use X/Y coordinates versus latitude and longitude.

To define points in latitude and longitude, get the `LatLngs` property and use its `Add` method to add points to the collection. To define points in X/Y pixel coordinates, get the `Points` property and use its `AddXY` or `Add` method to add points to the collection.

The default line is a black solid line with the width of 10 thousandths of an inch.

Here are the usual steps to draw a line primitive:

1. Create a `LinePrimitive` object.
2. Set the line's color (optional).
3. Set the line's width in thousandths of an inch (optional).
4. Set the line's style (optional).
5. Add your coordinate pairs to the points collection. See above for the two different approaches.
6. Add the line primitive to the primitive collection.
7. Add the primitive collection to the session.
8. If using server sessions, call the `UpdateSessionEx` client object method.

Here is a code snippet illustrating this technique, using X/Y coordinates:

```
MQClientInterface.LinePrimitive linePrimitive = new
MQClientInterface.LinePrimitive();
linePrimitive.Color = MQClientInterface.ColorStyle.ORANGE;
linePrimitive.Width = 140;
linePrimitive.Points.Add(MQClientInterface.Point(30, 30));
linePrimitive.Points.Add(MQClientInterface.Point(100, 100));
linePrimitive.Points.Add(MQClientInterface.Point(100, 150));
linePrimitive.Points.Add(MQClientInterface.Point(150, 150));

MQClientInterface.PrimitiveCollection primitives = new
MQClientInterface.PrimitiveCollection();

primitives.add(linePrimitive);
mySessionObject.addOne(primitives);

// see "Basic Mapping" chapter for session & map URL code...
```

See the sample code project `MapItWithPrimitives` for complete examples in each client interface, and please refer to “Map Annotation Constants” on page 90 for the exact syntax to use for all related constants within your client interface.

Drawing an Ellipse Primitive

To draw an ellipse or circle, create an `EllipsePrimitive` object and add it to your collection of drawing primitives in your `PrimitiveCollection`.

The `EllipsePrimitive` class defines the primitive's color, line and fill style, the line width in thousands of an inch, and its location.

To define the location and size, you will set the lower right (southeast) corner of the rectangle and the upper left (northwest) corner of the rectangle. The location is expressed in either X/Y screen coordinates or as latitude and longitude.

To define points latitude and longitude, set the properties `UpperLeftPointLatLng` and `LowerRightPointLatLng`. To define points in X/Y pixel coordinates, set the properties `UpperLeftPoint` and `LowerRightPoint`.

The default ellipse has a black solid line, is not filled, and has the width of 10 thousandths of an inch.

Here are the usual steps to draw an ellipse primitive:

1. Create an `EllipsePrimitive` object.
2. Set the lower right corner and the upper left corner of the enclosing rectangle. See above for two different approaches.
3. Set the ellipse's line width in thousandths of an inch.
4. Set the ellipse's color, width, and style.
5. Set the ellipse's fill color, and fill style (optional).
6. Add the ellipse to a primitive collection.
7. Add the primitive collection to your session.
8. If using server sessions, call the `UpdateSessionEx` client object method.

Here is a code snippet illustrating this technique, using X/Y coordinates:

```
MQClientInterface.EllipsePrimitive ellipsePrimitive = new
MQClientInterface.EllipsePrimitive();
ellipsePrimitive.Color = MQClientInterface.ColorStyle.GREEN;
ellipsePrimitive.CoordinateType =
MQClientInterface.CoordinateType.GEOGRAPHIC;
ellipsePrimitive.Width = 70;
ellipsePrimitive.UpperLeftPoint = new MQClientInterface.Point(20, 20);
ellipsePrimitive.LowerRightPoint = new MQClientInterface.Point(150, 150);

MQClientInterface.PrimitiveCollection primitives = new
MQClientInterface.PrimitiveCollection();
primitives.add(ellipsePrim);
mySessionObject.addOne(primitives);

// see "Basic Mapping" chapter for session & map URL code...
```

See the sample code project `MapItWithPrimitives` for complete examples in each client interface, and refer to “Map Annotation Constants” on page 90 for the exact syntax for related constants in your client interface

Drawing a Symbol Primitive

To draw an icon map annotation, create a `SymbolPrimitive` object and add it to your collection of drawing primitives in your `PrimitiveCollection`.

If you merely want to add small icons and/or text at a specific latitude and longitude, use POIs for added functionality. See Chapter 5, “Displaying POIs.”

The `SymbolPrimitive` object contains information to define the symbol file name and the location. The location is expressed either as X/Y pixel coordinates or as latitude and longitude.

To set its center point with latitude and longitude, set its `CenterLatLng` property. To set its center point with X/Y coordinates, set its `CenterPoint` property.

The default symbol type is raster (a bitmap icon), but you can set it to vector (a drawing icon) by setting its `SymbolType` property. Refer to “Map Annotation Constants” on page 90 for the exact syntax for related constants in your client interface.

Here are the usual steps to draw a symbol primitive:

1. Create a `SymbolPrimitive` object.
2. Set the symbol’s center point. See above for two different approaches.
3. Set the symbol’s file name.
4. Set the symbol’s type: vector or raster.
5. Add the symbol to the primitive collection.
6. Add the primitive collection to your session.
7. If using server sessions, call the `UpdateSessionEx` client object method.

Here is a code snippet illustrating this technique, using X/Y coordinates:

```
MQClientInterface.SymbolPrimitive symbolPrimitive = new
MQClientInterface.SymbolPrimitive();
symbolPrimitive.CoordinateType = MQClientInterface.CoordinateType.GEOGRAPHIC;
symbolPrimitive.CenterPoint = new MQClientInterface.Point(450, 250);
symbolPrimitive.SymbolName = "MQ00033";
symbolPrimitive.SymbolType= MQClientInterface.SymbolType.VECTOR;

MQClientInterface.PrimitiveCollection primitives = new
MQClientInterface.PrimitiveCollection();
primitives.add(symbolPrimitive);
mySessionObject.addOne(primitives);
```

```
// see "Basic Mapping" chapter for session & map URL code...
```

See the sample code project `MapItWithPrimitives` for complete examples in each client interface, and please refer to “Map Annotation Constants” on page 90 for the exact syntax to use for all related constants within your client interface.

Drawing a Rectangle Primitive

To draw a rectangle or square in a map, create a `RectanglePrimitive` object and add it to your collection of drawing primitives in your `PrimitiveCollection`.

The `RectanglePrimitive` class defines the rectangle color, line width in thousandths of an inch, line and fill style, and upper left (northwest) and lower right (southeast) corner coordinates. The coordinates are expressed either as X and Y screen coordinates, or as latitude and longitude.

To define points in X/Y pixel coordinates, set the properties `UpperLeftPoint` and `LowerRightPoint`. To define points using latitude and longitude, set the properties `UpperLeftPointLatLng` and `LowerRightPointLatLng`.

The default rectangle has a black solid line, is not filled, and has the width of 10 thousandths of an inch.

Here are the usual steps to draw a rectangle primitive:

1. Create a `RectanglePrimitive` object.
2. Set the rectangle’s upper left and lower right coordinates. See above for two different approaches.
3. Set the rectangle’s line style, and line color.
4. Set the rectangle’s line width in thousandths of an inch.
5. Set the rectangle’s fill color, and fill style (optional).
6. Add the rectangle primitive to the primitive collection.
7. Add the primitive collection to your session.
8. If using server sessions, call the `UpdateSessionEx` client object method.

Here is a code snippet illustrating this technique using X/Y coordinates:

```
MQClientInterface.RectanglePrimitive rectanglePrimitive = new
MQClientInterface.RectanglePrimitive();
rectanglePrimitive.Color = MQClientInterface.ColorStyle.BLACK;
rectanglePrimitive.CoordinateType =
MQClientInterface.CoordinateType.GEOGRAPHIC;
rectanglePrimitive.Width = 70;
```

```
rectanglePrimitive.FillStyle = MQClientInterface.FillStyle.FDIAGONAL;
rectanglePrimitive.UpperLeftPoint = new MQClientInterface.Point(250, 20);
rectanglePrimitive.LowerRightPoint = new MQClientInterface.Point(400, 200);

MQClientInterface.PrimitiveCollection primitives = new
MQClientInterface.PrimitiveCollection();

primitives.add(rectanglePrim);
mqSession.addOne(primitives);

// see "Basic Mapping" chapter for session & map URL code...
```

See the sample code project `MapItWithPrimitives` for complete examples in each client interface, and see “Map Annotation Constants” on page 90 for the exact syntax for relevant constants in your client interface.

Drawing a Text Primitive

To draw text, create a `TextPrimitive` object and add it to your collection of drawing primitives in your `PrimitiveCollection`.

If you want to create text next to a Point Of Interest (POI), it’s best to use *POI labels* instead of text primitives. See Chapter 5, “Displaying POIs.”

The `TextPrimitive` class defines the foreground and background colors, the text display area height and width, and the location of its upper left (northwest) corner. Specify coordinates using X/Y screen coordinates or with latitude and longitude.

To set its location with latitude and longitude, set its `UpperLeftLatLng` property. To set its location with X/Y coordinates, set its `UpperLeftPoint` property.

You can set the font to use for the text. You are responsible for verifying that the font is available on the server that generates the map image.

The default text primitive has black 14 point Arial text.

Here are the usual steps to draw a text primitive:

1. Create an `TextPrimitive` object.
2. Set the text’s foreground and background colors, and draw layer.
3. Set the text’s upper left coordinates. See above for two different approaches.
4. Set the text’s width and height.
5. Set the text’s font.

6. Set the label text in the `Text` property.
7. Add the text primitive to the primitive collection.
8. Add the primitive collection to your session.
9. If using server sessions, call the `UpdateSessionEx` client object method.

Here is a code snippet illustrating this technique, using X/Y coordinates:

```
MQClientInterface.TextPrimitive textPrimitive = new
MQClientInterface.TextPrimitive();
textPrimitive.Color = MQClientInterface.ColorStyle.RED;
textPrimitive.CoordinateType = MQClientInterface.CoordinateType.DISPLAY;
textPrimitive.BkgdColor = MQClientInterface.ColorStyle.YELLOW;
textPrimitive.BoxOutlineColor = MQClientInterface.ColorStyle.BLUE;
textPrimitive.Style = MQClientInterface.FontStyle.BOXED;
textPrimitive.FontName = "Helvetica";

textPrimitive.FontSize = 18;
textPrimitive.UpperLeftPoint = new MQClientInterface.Point(250,250);
textPrimitive.Width = 50;
textPrimitive.Text = "Sample Text";

MQClientInterface.PrimitiveCollection primitives = new
MQClientInterface.PrimitiveCollection();

primitives.add(textPrimitive);
mqSession.addOne(primitives);

// see "Basic Mapping" chapter for session & map URL code...
```

See the sample code project `MapItWithPrimitives` for complete examples in each client interface, and please refer to “Map Annotation Constants” on page 90 for the exact syntax to use for all related constants within your client interface.

Advantage API customers can rely only on the fonts "Helvetica" and "Lucida" on MapQuest-hosted mapping servers.

Note that text is not rendered on the server for non-bitmap image formats such as EPS. Because of this, all fonts in maps must be available on whatever computer displays or prints such images.

Drawing a Polygon Primitive

To draw a polygon, create a `PolygonPrimitive` object and add it to your collection of drawing primitives in your `PrimitiveCollection`. A polygon is similar to a `LinePrimitive`, but with the same number of points it will have an additional line segment from the last point to the initial point. Additionally, polygons can add a fill color and fill style.

The `PolygonPrimitive` class defines the primitive's color, the line width in one thousandths of an inch, the line color, line style, fill color, fill style, and the location of each vertex of the polygon.

The location of the vertices of the polygon are expressed either as X/Y screen coordinates or as latitude and longitude. You must add at least three points to a polygon primitive. The line segments will be drawn in the order you add the points, followed by the additional line segment to the initial point.

To define the points, you will must get the primitive's collection of points, which is done differently if you use X/Y coordinates or latitude and longitude:

To define points in latitude and longitude, get the `LatLngs` property and use its `Add` method to add points to the collection. To define points in X/Y pixel coordinates, get the `Points` property and use its `AddXY` or `Add` method to add points to the collection.

To draw a polygon primitive:

1. Create a `PolygonPrimitive` object.
2. Add the polygon's vertex values. See above for the two different approaches.
3. Set the polygon's line color, fill color, and fill style.
4. Set the polygon's line width in thousandths of an inch.
5. Add the polygon primitive to the primitive collection.
6. Add the primitive collection to your session.
7. If using server sessions, call the `UpdateSessionEx` client object method.

Here is a code snippet illustrating this technique:

```
MQClientInterface.PolygonPrimitive polygonPrimitive = new
MQClientInterface.PolygonPrimitive();
polygonPrimitive.Color = MQClientInterface.ColorStyle.RED;
polygonPrimitive.CoordinateType = MQClientInterface.CoordinateType.DISPLAY;
polygonPrimitive.FillColor = MQClientInterface.ColorStyle.BLUE;
polygonPrimitive.FillStyle = MQClientInterface.FillStyle.SOLID;
polygonPrimitive.Width = 42;
polygonPrimitive.Points.Add(new MQClientInterface.Point(50, 200));
```



```
polygonPrimitive.Points.Add(new MQClientInterface.Point(50, 275));
polygonPrimitive.Points.Add(new MQClientInterface.Point(150, 275));
polygonPrimitive.Points.Add(new MQClientInterface.Point(150, 200));
polygonPrimitive.Points.Add(new MQClientInterface.Point(100, 240));

MQClientInterface.PrimitiveCollection primitives = new
MQClientInterface.PrimitiveCollection();

mqSession.addOne(primitives);

// see "Basic Mapping" chapter for session & map URL code...
```

See the sample code project `MapItWithPrimitives` for complete examples in each client interface, and please refer to “Map Annotation Constants” on page 90 for the exact syntax to use for all related constants within your client interface.

Other Map Annotation APIs

Using the Key Value in Map Annotations

If you need to identify drawing primitives after drawing them, you can store a unique identifier within each primitive’s `Key` property. Most developers do not need this, but it is appropriate in some cases for identifying a drawing primitive so you can find it or modify it later.

This property has no inherent meaning to the MapQuest API or the server, but the value will be preserved, even when uploaded to the server and later retrieved using the `GetSessionEx` client object method.

Updating your local session object creates new session parameter objects, like objects of class `MapState` and `PrimitiveCollection` that you added to the session. Because of this, you cannot simply save object references to these objects and rely on them after an update. This affects all the following client object methods: `UpdateSessionEx`, `UpdateSessionDirect`, `UpdateSessionDirect`, and `GetSession`.

Therefore, you may need to find a map annotation in some other way. For example, suppose you create fifty complex polygon primitives, and two polygons were special in the context of your application because their appearance could be modified by the user somehow. Your application may want to change one `PolygonPrimitive` object and leave other drawing primitives untouched. Set the `Key` property of special objects to a special value. To find the special polygon, iterate over the objects in the `PrimitiveCollection` and get the `Key` property of each.

If you use the `Key` property, it is your responsibility to interpret the meaning of the property and to preserve uniqueness within a collection if required.

Scaling and Centering Maps to Fit Map Annotations

You can use the `BestFit` map command to scale and center a map to “best fit” map annotations you have added to your session. Please refer to “Using Map Commands” on page 62 for API details.

Map Annotation Constants

.NET developers should use the map annotation constants shown below using the syntax required by the .NET interface.

Layer Selection Constants

Set these constants in a drawing primitive’s `DrawTrigger` property to change the drawing layer.

```
MQClientInterface.DefaultTrigger.DRAW_BEFORE_POLYGONS
MQClientInterface.DefaultTrigger.DRAW_AFTER_POLYGONS
MQClientInterface.DefaultTrigger.DRAW_BEFORE_TEXT
MQClientInterface.DefaultTrigger.DRAW_AFTER_TEXT
MQClientInterface.DefaultTrigger.DRAW_BEFORE_ROUTE_HIGHLIGHT
MQClientInterface.DefaultTrigger.DRAW_AFTER_ROUTE_HIGHLIGHT
```

Colors

These constants are used for color properties of drawing primitive classes as well as the `DTStyle` classes used in POIs.

```
MQClientInterface.ColorStyle.BLACK
MQClientInterface.ColorStyle.BLUE
MQClientInterface.ColorStyle.CYAN
MQClientInterface.ColorStyle.DARK_GRAY
MQClientInterface.ColorStyle.GRAY
MQClientInterface.ColorStyle.GREEN
MQClientInterface.ColorStyle.LIGHT_GRAY
MQClientInterface.ColorStyle.MAGENTA
MQClientInterface.ColorStyle.ORANGE
MQClientInterface.ColorStyle.PINK
MQClientInterface.ColorStyle.RED
MQClientInterface.ColorStyle.WHITE
MQClientInterface.ColorStyle.YELLOW
MQClientInterface.ColorStyle.INVALID_COLOR
```

Text Drawing Characteristics

These constants are used to set text properties for drawing primitive classes as well as the `DTStyle` classes used in POIs.

```
MQClientInterface.FontStyle.NORMAL  
MQClientInterface.FontStyle.BOLD  
MQClientInterface.FontStyle.BOXED  
MQClientInterface.FontStyle.OUTLINED  
MQClientInterface.FontStyle.ITALICS  
MQClientInterface.FontStyle.UNDERLINE  
MQClientInterface.FontStyle.STRIKEOUT  
MQClientInterface.FontStyle.THIN  
MQClientInterface.FontStyle.SEMIBOLD  
MQClientInterface.FontStyle.INVALID
```

Pen Styles

These constants are used to set pen style properties for drawing primitives. On Windows, pen styles with Dash or Dot in their name work only with 1 pixel width lines; otherwise they show as solid lines.

```
MQClientInterface.PenStyle.SOLID  
MQClientInterface.PenStyle.DASH  
MQClientInterface.PenStyle.DOT  
MQClientInterface.PenStyle.DASH_DOT  
MQClientInterface.PenStyle.DASH_DOT_DOT  
MQClientInterface.PenStyle.NONE
```

Fill Styles

These constants are used to set solid fill properties for drawing primitives.

```
MQClientInterface.FillStyle.BDIAGONAL  
MQClientInterface.FillStyle.SOILD  
MQClientInterface.FillStyle.CROSS  
MQClientInterface.FillStyle.DIAG_CROSS  
MQClientInterface.FillStyle.FDIAGONAL  
MQClientInterface.FillStyle.HORIZONTAL  
MQClientInterface.FillStyle.VERTICAL  
MQClientInterface.FillStyle.NONE
```

Coordinate Types

These constants are used to set coordinate types for drawing primitives. The default is the geographic type.

```
MQClientInterface.CoordinateType.GEOGRAPHIC  
MQClientInterface.CoordinateType.DISPLAY
```

Text Alignment

These constants are used to set text alignment for `TextPrimitive` objects.

```
MQClientInterface.TextAlignment.CENTER  
MQClientInterface.TextAlignment.LEFT  
MQClientInterface.TextAlignment.RIGHT  
MQClientInterface.TextAlignment.BASELINE  
MQClientInterface.TextAlignment.BOTTOM  
MQClientInterface.TextAlignment.TOP
```

7 Proximity Searching

You can search for important locations within a given geographic area, calculate distances between locations, and perform other location searches using dynamic database queries. For example, users can search for the closest restaurants in their neighborhood or bookstores within 10 miles of a driving route calculated with the routing API.

Because some developers use proximity searching without mapping the results, this chapter duplicates some Points of Interest (POI) class information from Chapter 5, “Displaying POIs.” Refer to that chapter and the API Reference documentation for more information about POI classes.

If you use MapQuest server sessions and you merely need to determine what on-screen map object a user clicked, use the `IdentifyFeature` API described in “Identifying Map Objects by Coordinates” on page 70.

Proximity Search Features

Most developers use proximity searching APIs to search for Points of Interest (POIs) stored in a database or a collection of simple POIs. Additionally, proximity searching APIs can search collections of features returned from previous searches or map data.

You can look for all POIs in a geographic area, or request only POIs with specific attributes. This flexibility allows applications to provide users with a user interface allowing them to “design” custom proximity searches. This is critical for applications with large numbers of locations with varied types and attributes.

There are two types of search criteria. *Geographic criteria* define the area to search. *Descriptive criteria* narrow the results of the search to those with certain attributes.

See the figure above for an example of how you might let users define descriptive criteria and geographic criteria.

You could offer even more geographic criteria choices, like choosing the number of miles from a location or let the user point-and-click on a map to choose new anchor points for a search.

Search API Overview

The following is an overview of the steps for a proximity search, followed by a summary of each step.

1. Define the geographic criteria.
2. Define one or more types of data to search: a collection of points, POIs from a database, or map data set features.
3. Narrow the search to specific display types or other descriptive criteria.
4. Perform the search using the `Search` client object method.
5. Iterate through the search results.
6. If the results are non-empty, extract feature information as needed.

Defining Geographic Criteria

Create a search criteria object by creating a class that defines the type of search criteria: a polygon search, a radius search, a rectangle search, or a corridor search around a path (usually routing results). You will specify the geographic area and the maximum number of matches to find. For details, see “Using Search Criteria” on page 98.

Defining Search Data

Database Search. If you want to search a database of Points of Interest, create an instance of class `DBLayerQuery` that specifies a database pool on the server. You can optionally narrow the search using SQL fragments. For example, narrow a restaurant POI search to those marked as Italian restaurants. For details, see “Database Searches” on page 96.

Feature Collection Search. If you need to search Points of Interest that are not stored in a database, search feature collections. The most common feature type is `PointFeature`, but all feature types are supported. For details, see “Feature Collection Searches” on page 97.

Map Data Set Search. If you want to find streets, parks, or other features within map data, you can specify a map coverage to search. For details, see “Map Coverage Searches” on page 97.

Narrowing the Search to Specific Display Types

Every POI and map data feature has a *display type* (DT) number that identifies what type of object it is: a city, a lake, a highway, a dirt road, or other choices. Narrow the display type range of items you are interested in finding by creating a `DTCollection` object and passing that to the `Search` client object method. For details, see “DTs and Extra Criteria in Database Searches” on page 96.

Performing the Search

Use the `Search` client object method, passing the objects mentioned above, and also pass a `FeatureCollection` that will contain the search results. You must have defined one or more types of data to search: POIs from a database, a collection of points, and/or map data set features.

If the “results” feature collection is non-empty before the search, the `Search` method will empty it before adding search results.

Iterating Through Search Results

In .NET, use the collection’s `Size` method to get the size and use the `GetAt` method to get each feature.

If the feature collection that contains the results of the search has a size equal to zero, display an appropriate error to the user; otherwise, iterate through the results collection.

Extracting Feature Information From Search Results

Some applications might want to get the name of the POI, its location, or other properties by directly getting properties from the object.

Many applications merely get the feature’s `Key` property. For POIs from databases, this key is defined within the POI database records and uniquely identifies the feature within that database. Simple POIs or other features that you create also can contain the `Key` property, and you can set that property as desired.

The `Key` property is empty within features found within licensed map data.

This property has no inherent meaning to the MapQuest API or the server, but the value will be preserved, even when uploaded to the server in a session feature collection and later retrieved using the `GetSessionEx` client object method.

Applications can perform their own database queries using non-MapQuest APIs using a POI’s unique key to access an unlimited number of feature details from their own databases. You can use the `GetRecordInfo` API to access information like this. For details, refer to “Independent Database Queries” on page 76.

Note that if you searched only a database, all results will be of class `PointFeature`. If you searched licensed map data or feature collections that contained other feature types, results might also be of the classes `LineFeature` or `PolygonFeature`.

Defining What Data to Search

There are three types of data you can search. You will use at least one type in each search, but you may search two or three types if desired:

- Database Searches.
- Feature Collection Searches.

- Map Coverage Searches.

Database Searches

If you want to search a database of Points of Interest, create an instance of `DBLayerQuery` to specify a database pool on the server.

The primary property in this object is `DBLayerName`, which should contain the name of a database pool on the server. By default, the database pool defines the database table within the database, but the database table can be overridden.

After you configure your `DBLayerQuery` object, you will add it to a query collection of class `DBLayerQueryCollection`. You can add multiple query objects to this collection to create maps with POIs from multiple database pools.

Databases for Advantage API customers have fixed database field names. Please refer to “Database Connectivity in Advantage API” on page 16 for details.

DTs and Extra Criteria in Database Searches

Although you can display all POIs in a database pool, you can narrow the database search as needed. For instance, you could display only POIs of desired display types, or use other criteria. You can define any criteria you want using SQL database query language expressions surrounded by parentheses. As a simple example, to narrow POI display to records with the value `TakesCreditCards` set to 1, add the extra criteria string `"TakesCreditCards=1"`.

For SQL syntax reference documentation, consult your database documentation or search the World Wide Web.

If the attribute you want to limit is the display type, you do not have to modify the database query. You can limit a search to certain display types by creating a *display type collection* (`DTCollection`), populate it with display type numbers you want, and pass it to the `Search` client object method. If desired, you can use both extra criteria strings and display type collections with a single search.

Some developers want to limit searches to certain POI display types and want one `DBLayerQuery` that will work with displaying and searching POIs. If this is the case for you, do not use `DTCollection` objects because the mapping display APIs do not accept `DTCollection` objects to limit display types. Instead, add an extra criteria string like `"DT=3583 OR DT=3582"`.

Here is an example of setting up a simple database query with extra criteria:

```
MQClientInterface.DBLayerQuery dbLayerQuery = new
MQClientInterface.DBLayerQuery();

dbLayerQuery.DBLayerName = "MQA.test";
```



```
dbLayerQuery.ExtraCriteria = "TakesCreditCards=1";

MQClientInterface.DBLayerQueryCollection dbLayerQueryCollection = new
MQClientInterface.DBLayerQueryCollection();

dbLayerQueryCollection.add(dbLayerQuery);
// Later, pass qColl to the Search client object method
```

Feature Collection Searches

If you need to search Points of Interest that are not stored in a database, you will search feature collections (class `FeatureCollection`).

The most common feature class is `PointFeature`, and these are often created by developers who display simple POIs on maps. However, `LineFeature` and `PolygonFeature` are also valid feature types within a feature collection.

To create a simple POI, create a `PointFeature` object, set its `DT` property with the display type, set its location in its `CenterLatLng` property, and optionally set a text label in its `Name` property. You can also set a unique key in its `Key` property, which might help identify the feature later if you want to get it back from your proximity search results, letting you re-query your own private database for more information about the POI if desired.

If you need to assign new display types, use the developer range 3072 through 3583. If you want to use line or polygon features, or you want to use DTs associated with features in licensed data, please refer to Appendix A, “Display Types.”

Map Coverage Searches

Although less common than proximity searching with Points of Interest, you can also search licensed map data for features. For instance, you could find individual streets, parks, lakes, or other features embedded within licensed map data.

You can search mapping data completely independent of map display, so even if center point, map scale, or style settings hide certain map features, the `Search` API will find them if they match your search requirements.

You might want to use this API if you wanted to find all features of a particular display type (for instance, major highways) within 5 miles of the user. For a list of display types, refer to Appendix A, “Display Types.”

To perform proximity searching with mapping data, pass the map pool name to the `Search` client object method when performing the search.

Setting Display Type Requirements for Searches

A common task is searching within a geographic area for specific display types. Usually, the search is within your own POI database, but you can search licensed map data or `FeatureCollection` objects too.

Display types are a general classification of the type of object. If you want to find DTs in licensed data, refer to “DT Values in Licensed Mapping Data” on page 125.

There is an easy-to-use mechanism to limit searches like this using a *display type collection*. Define an object of class `DTCollection` and add the desired display type numbers to the collection using the collection’s `Add` method.

When you call `Search`, pass the display type collection as an argument. See the API Reference for the calling convention details for your client interface.

Using Search Criteria

There are four types of geographic search criteria to describe an area to search:

- Radius Search Criteria.
- Rectangle Search Criteria.
- Polygon Search Criteria.
- Searching Near a Path With a Corridor Search.

Radius Search Criteria

The parameters of a radius search criteria object are a center point, a search radius, and the maximum matches to return from the search. You will pass the radius search criteria object as the first parameter to the `Search` client object method.

The search radius is in miles by default, but you can use kilometers using an optional *distance units* parameter when setting the `Radius` property. See the API Reference for details about the calling convention and constants.

To perform a proximity radius search with a database, complete the following steps:

1. Create a database query object of class `DBLayerQuery`.
2. Set its `DBLayerName` property.
3. Create an instance of `RadiusSearchCriteria`.
4. Set its center in latitude and longitude.
5. Set its search radius.

6. Set its maximum number of returned points.
7. Add the database query object to a new database query collection.
8. Perform the search using the client object method `Search`.
9. When the returned result collection has a length greater than zero, display results on a map or do anything that makes sense for your application.
10. If you used the `Key` property in features to store a unique identifier, check the `Key` property within each search result and query databases as appropriate.

For database steps, refer to “Database Searches” on page 96. For complete examples, refer to the sample code projects `SearchItAndDisplayIt` and `SearchIt`.

The following code snippet illustrates preparing a radius search criteria object:

```
MQClientInterface.RadiusSearchCriteria radiusSearchCriteria = new
MQClientInterface.RadiusSearchCriteria();

radiusSearchCriteria.Center = new MQClientInterface.LatLng(40.4445, -
79.995399);

radiusCrit.MaxMatches = 15;
radiusSearchCriteria.Radius = 5;
// Later, pass radiusCrit to the Search client object method
```

Rectangle Search Criteria

The parameters of a rectangle search criteria object are its upper left (northwest) corner coordinate, its lower right (southeast) corner coordinate, and the maximum matches to return from the search. You will pass the rectangle search criteria object as the first parameter to the `Search` client object method.

Some applications use a rectangle search instead of a radius search if they let a user choose the search area by either clicking-and-dragging or by clicking two corner points of a rectangle. If your map uses MapQuest server sessions, you can use the client object method `PixToLL` to convert pixel coordinates (from mouse clicks) to latitude and longitude.

If you ever need to search for POIs that overlap with your map display area but do not want to display them, this would be an good situation to use a rectangle search.

To perform a rectangle search with a database, complete the following steps:

1. Create a database query object of class `DBLayerQuery`.
2. Set its `DBLayerName` property.

3. Create an instance of `RectangleSearchCriteria`.
4. Set its corner coordinates in the properties `UpperLeft` and `LowerRight`.
5. Set the maximum number of returned points in the `MaxMatches` property.
6. Add the database query object to a new database query collection.
7. Perform the search using the client object method `Search`.
8. When the returned result collection has a length greater than zero, display results on a map or do anything that makes sense for your application.
9. If you used the `Key` property in features to store a unique identifier, check the `Key` property within each search result and query databases as appropriate.

The following code snippet illustrates preparing a rectangle search criteria object:

```
MQClientInterface.RectSearchCriteria rectSearchCriteria = new
MQClientInterface.RectSearchCriteria();

rectSearchCriteria.UpperLeft = new MQClientInterface.LatLng(40.517069, -
80.089632);

rectSearchCriteria.LowerRight = new MQClientInterface.LatLng(40.371931, -
79.901166);

rectSearchCriteria.MaxMatches = 20;
// Later, pass criteria to the Search client object method
```

For database steps, refer to “Database Searches” on page 96. For an example of processing search results, refer to the sample code projects `SearchIt` and `SearchItAndDisplayIt`.

Polygon Search Criteria

The parameters of a polygon search criteria object are a list of coordinates defining the boundaries of the polygon, and the maximum matches to return from the search. You will pass the polygon search criteria object as the first parameter to the `Search` client object method.

Some applications use a polygon search if their users need to select complex search areas by clicking three or more points of a polygon. If your map uses MapQuest server sessions, you can use the client object method `PixToLL` to convert pixel coordinates in a map to latitude and longitude.

Each location is a vertex in an implicitly closed polygon, so you do not need to repeat the first location to describe the polygon.

Add your new locations to the `LatLngCollection` object contained within the `ShapePoints` property of the search criteria.

To perform a polygon search with a database, complete the following steps:

1. Create a database query object of class `DBLayerQuery`.
2. Set its `DBLayerName` property.
3. Add the database query object to a new database query collection.
4. Create an instance of `PolygonSearchCriteria`.
5. Set its vertices using either X/Y coordinates or latitude and longitude.
6. Set its maximum number of returned points in the `MaxMatches` property.
7. Perform the search using the client object method `Search`.
8. When the returned result collection has a length greater than zero, display results on a map or do anything that makes sense for your application.
9. If you used the `Key` property in features to store a unique identifier, check the `Key` property within each search result and query databases as appropriate.

The following code snippet illustrates preparing a polygon search criteria object:

```
MQClientInterface.PolySearchCriteria polySearchCriteria = new
MQClientInterface.PolySearchCriteria();
polySearchCriteria.ShapePoints.AddLatLng(40.517069, -80.089632);
polySearchCriteria.ShapePoints.AddLatLng(41.517069, -83.089632);
polySearchCriteria.ShapePoints.AddLatLng(42.517069, -79.089632);
polySearchCriteria.MaxMatches = 15;
```

For database steps, refer to “Database Searches” on page 96. For an example of processing search results, refer to the sample code projects `SearchIt` and `SearchItAndDisplayIt`.

Searching Near a Path With a Corridor Search

Use a *corridor search* to search in the vicinity of a path, which typically is a route returned by the routing API. The parameters of a `CorridorSearchCriteria` object are the locations defining the path of the corridor, the width of the corridor around that path, and the maximum matches to return from the search. You will pass the corridor search criteria object as the first parameter to the `Search` client object method.

You must provide the shape points in a `LatLngCollection` object. Most developers get these shape points from the routing API. With the routing API, the results of routing are stored in an object of class

`RouteResults`. That object contains a `RouteShapes` property that contains the route shape points. For details, see “Routing API Overview” on page 106.

If you need to add new locations, modify the `LatLngCollection` object contained within the `ShapePoints` property of the search criteria.

Note that in current server software, corridor searching will find point features only (not lines or polygons). Also, point features are only found if they are in database data or custom `LocationCollection` objects; points will not be found in server map data.

The following is a code snippet that illustrates setting a corridor search criteria:

```
MQClientInterface.CorridorSearchCriteria crit = new
MQClientInterface.CorridorSearchCriteria();
crit.ShapePoints.AddLatLng(40.517069, -80.089632);
crit.ShapePoints.AddLatLng(41.517069, -83.089632);
crit.ShapePoints.AddLatLng(42.517069, -79.089632);
crit.MaxMatches = 15;
crit.CorridorWidth = 0.25;
```

Advanced Corridor Searching

By default, the server will *generalize* the shape points describing the corridor, a process which removes shape points if others are closer than a certain distance (the *buffer width*). Generalization usually decreases the time and server resources necessary to perform the search, and typically doesn’t affect the results.

To turn off generalization entirely, set the corridor search criteria object’s `CorridorExactLinks` property to true.

If you use route generalization, MapQuest recommends that you use the default buffer width (0.25 miles). If corridor searching is very slow for your application, you can experiment with increasing the buffer width. In general, do not decrease the value because it can increase time and server resources.

Valid buffer width values are between 0 to 2. The default distance unit is miles, but you can specify kilometers using an optional argument when setting `CorridorBufferWidth`.

Search Result Properties

After a search, the most common feature properties to get from the results are `Name` and `Key`. (See “Using the Key Value in POIs” on page 74.) However, returned features also may contain extra properties that provide information that may be useful for some applications. The most common example is showing the distance between found objects and the *anchor* (center) of a radius search. However, there are other types of information available depending on the type of search.

For radius and corridor searches with certain feature types, `Distance` property contains the distance from feature to the the radius anchor point or the corridor “center.” For complex features like lines and polygons, the distance is measured to the part of the feature that is closest to the anchor location. See the table “Post-Search Properties and Support for Feature Types and Search Criteria” on page 103 for other restrictions.

If the found feature is a line or polygon, you can additionally get the `ClosestLatLng` or `ClosestPoint` property. The “point” version returns the closest point in X/Y coordinates rather than latitude and longitude. These properties are only valid after radius searches; do not rely on them after corridor, rectangle, or polygon searches.

Additionally, if a line feature represents a road within map data, you can get properties called `LeftAddressHi`, `LeftAddressLo`, `RightAddressHi`, and `RightAddressLo`. These contain the street address ranges (from “lo” to “hi”) for the street blocks that begin and end that road. Note that the left and right notations do not guarantee the relative positions on the map that their names might imply.

Line features that are roads also have `LeftPostalCode` and `RightPostalCode` properties containing postal codes for the road endpoints.

See Table 11 for the complete list of supported extra properties for each search criteria and feature type. The text “roads” indicates support only for line features that represent roads found within server map data.

Table 11, Post-Search Properties and Support for Feature Types and Search Criteria

Feature Property	Polygon Search	Rectangle Search	Corridor Search	Radius Search
<code>Distance</code>	None	None	Points	All feature types
<code>ClosestLatLng</code>	None	None	None	Lines, polygons
<code>ClosestPoint</code>	None	None	None	Lines, polygons
<code>LeftPostalCode</code>	Roads	Roads	None	Roads
<code>RightPostalCode</code>	Roads	Roads	None	Roads
<code>LeftAddressHi/Lo</code>	Roads	Roads	None	Roads
<code>RightAddressHi/Lo</code>	Roads	Roads	None	Roads

Advanced Searching With Route Matrix API

You can use search APIs in conjunction with routing APIs to search for locations within a certain driving distance or driving time from an origin point.

You can calculate the driving distance or driving time between an origin point and a list of locations with the client object method `DoRouteMatrix`, described in “Route Matrix Calculation” on page 116.

If you can approximate a radius distance that would encapsulate all the POIs (or other locations) to search, you can use `DoRouteMatrix` to filter search results by maximum driving time or driving distance.

For example, here are the steps for finding locations whose driving time from an origin point is less than a certain amount of time:

1. Approximate the radius distance that would include all locations to search.
2. Use the search APIs with a radius criteria to find the locations around that distance an origin point. The search result locations will be features in a feature collection.
3. Create a new location collection.
4. Add to that collection a `GeoAddress` object containing the latitude and longitude of the origin location.
5. Add to that collection `GeoAddress` objects containing the latitude and longitude for each feature in the search results.
6. Create a one-to-many route matrix with `DoRouteMatrix` client object method, as described in “Route Matrix Calculation” on page 116

For each location, check if the time is less than the desired maximum time. If it is, perform whatever application-specific action you desire. For example, you might add this location to a map and/or list the driving times for the closest locations sorted by driving time.

Refer to the sample code project `SearchAndDisplayItWithRouteDistance` for an example of using `DoRouteMatrix` in conjunction with the search API to “search” for locations within a certain driving time or driving distance.

Proximity Searching Constants

.NET developers should use the search-related constants shown below using the syntax required by the .NET interface.

Distance Units

Used to describe miles or kilometers.

```
MQClientInterface.DistanceUnits.MILES  
MQClientInterface.DistanceUnits.KILOMETERS
```


8 Routing

Routing means calculating navigational directions between two locations, possibly with intermediate destinations. Advantage API provides a robust routing API and access to the world's best routing data from multiple vendors.

You can use the routing API on its own or in conjunction with the mapping API. You could highlight a driving route on a map or display textual driving directions.

Routing Example With
Route Highlight and Text
Narrative



1. Turn RIGHT on to SOUTH ST. Drive 0.2 miles.
2. Merge onto I-76 W (Portions toll). Drive 58.77 miles.
3. Take the exit number 286 toward US-222. Drive 1.06 miles.
4. Merge onto US-222 S. Drive 15.21 miles.
5. Merge onto US-30 W toward LANCASTER. Drive 7.58 miles.
6. Take the exit toward MOUNTVILLE. Drive 0.17 miles.

Routing Features

Show routes on maps. There are several ways to display routes on maps, including a very simple method if your application uses MapQuest server sessions.

Textual driving directions. The routing engine optionally provides textual driving directions that contain maneuver (turn-by-turn) directions with distances measured in either kilometers or miles.

Multiple location routing. Provide three or more locations in a route and the routing engine will calculate directions through intermediate destinations.

Optimized multi-destination routing. Advantage API supports intelligent routing when using more than 3 locations in a route. The server can generate an *optimized route*, which reorders intermediate destinations for the best possible route.

Routing using latitude and longitude. Specify locations using latitude and longitude to let the server find the closest drivable locations. This allows users to click on a map and generate navigational directions to or from that point.

Routing using street addresses. Using the geocoding API and the routing API together, you can provide driving directions to or from any street address.

Route customization. The routing engine allows generating different types of routes, including the shortest distance route, fastest time route, or pedestrian route. Other options include avoiding certain road types or changing the language of textual driving directions.

Automatic data selection. Request a route without manually choosing specific routing data on the server. Simply request a route and let the server choose the best vendor and coverage data. You can even let the server choose what data to use without requesting the route immediately — a helpful feature when routing data is split among multiple servers.

Route matrix calculation. Applications can now easily and efficiently automate calculations of driving distance and driving time between many locations.

Routing API Overview

For basic routing, developers request a route by setting a few simple options and setting 2 or more locations in the route. The server will use server-based *route selectors* to determine the best routing data for those locations. The route selector determines which data sets contain all of the locations. If the route selector finds more than one acceptable data set, the server chooses the highest priority data set among them, as defined by the server's route data selector file.

In general, a server will contain only one route data selector and will be configured as the default so you should not need to know the name of the *route data selector pool*.

Advantage API customers interested in route selector pool configurations should refer to “Advantage API Connection Information” on page 16.

The quality of the driving directions varies by vendor, by country, by region within the country, and the specific input locations. In addition, route pools have different coverage areas, so be sure to choose a route

pool capable of completing the route. This is especially important if your application handles addresses in multiple countries.

If you have questions about which routing data sets are most appropriate for your application, please contact MapQuest Technical Support.

Defining a Route Overview

Developers specify locations to the routing API as a `LocationCollection` object containing addresses of class `GeoAddress`. Usually, developers get a `GeoAddress` as a result from geocoding an address. Alternatively, you can specify locations using latitude and longitude coordinates.

In all cases, the first location is the origin, the last location is the destination, and other points are intermediate destinations in the order within the collection. For details, see “Specifying Locations for Routing” on page 108.

In order to use the routing API, you must create an object of class `RouteOptions`, even if you don’t override any options. Use this object to set options like route types such as fastest time or shortest distance. If you have more than 3 locations, you can generate an *optimized route*, which reorders intermediate destinations for the best possible route. The server provides a reordered list of your original locations.

For all options, refer to Table 12, “RouteOptions Properties,” on page 109.

Route Results Overview

After calculating a route with `DoRoute`, check the `ResultCode` property of the route results object that you passed to `DoRoute`. If the result matches the MapQuest constant for success, proceed to use, display, or map the results as desired.

If the result code is something else, the route was not successful. You can use that result code (and other error information) to determine what went wrong. For more details, see “Routing Errors” on page 113.

Displaying Route Highlights Overview

If you use MapQuest server sessions to display maps, displaying the driving directions visually on the map is easy when you provide `DoRoute` with a session ID string. See “Route Highlights With Stateless Maps” on page 112 for details.

If you do not use server sessions and you want to display route highlights, you can draw them using map annotations. See “Route Highlights With Stateless Maps” on page 112 for details.

The route highlights never include icons for origin and destination locations. The recommended approach to creating icons like those is to create simple POIs, discussed in Chapter 5, “Displaying POIs.”

Specifying Locations for Routing

The routing API calculates a route between an origin location and a destination location, with optional intermediate destinations. Most applications specify locations from results of geocoding full street addresses. You can also specify a location with any geocoding result, or use latitude and longitude coordinates.

A common application task is to calculate a route between a user's street address and a POI the user selected from results of a proximity search.

You must geocode all street addresses (or partial addresses) before passing them to the routing API. Prior geocoding allows your application to intelligently process geocoding ambiguities or errors in an application-specific way.

During geocoding, the best address match might be slightly different than the user-provided address or be returned with low confidence levels. If a user entered the address, MapQuest recommends that you display the full address of the best match returned from the geocoding engine. If the geocoding or routing behavior is incorrect, displaying this information will assist the user so he or she can try again.

The geocoding results are in `GeoAddress` objects. Once you geocode addresses for the route origin, intermediate locations, route destination, add them to a `LocationCollection` object in the order they will be visited. You will pass this location collection to the `DoRoute` function to calculate the route.

See also: Chapter 3, "Geocoding."; "Advanced Geocoding," p. 33.

Customizing Routing

The routing API requires that you create a `RouteOptions` object to pass to the routing API. You can optionally customize road type avoidance lists, narrative type (HTML, none, or default), the number of shape points returned for a single maneuver (one leg of the journey, usually on one road), and the route type (shortest, fastest, optimized, and others).

As an example of road type avoidance, you can specify to avoid highways whenever possible throughout the journey.

Advanced programmers with detailed knowledge of a map data set and who know specific road segment unique IDs (GEF IDs) can specify to avoid certain road segments. For road segments specified as *absolute avoidance*, the `DoRoute` client object method returns an error when the route cannot be completed without using one of the specified segments. For road segments specified as *attempt avoidance*, the segments in the list are used only if no other route is found.

To calculate a route, create an object of class `RouteOptions` and set any properties, all of which are optional. Refer to Table 12 for details of each property.

Table 12, RouteOptions Properties

Property	Description
RouteType	Optimize for shortest distance, fastest time, or pedestrian-only routes. Use the constants FASTEST, SHORTEST, PEDESTRIAN, or SELECT_DATASET_ONLY. The default is a fastest time route. Please refer to “Routing Constants” on page 118 for the exact syntax to use within your client interface. See below for more information about optimized and SELECT_DATASET_ONLY routes.
NarrativeType	The format of textual driving directions. Use the constants DEFAULT, HTML, or NONE. . The default is DEFAULT. Please refer to “Routing Constants” on page 118 for the exact syntax to use within your client interface.
NarrativeDistanceUnitType	Textual driving directions in miles or kilometers. Use the constants MILES or KILOMETERS. The default is MILES. Please refer to “Routing Constants” on page 118 for the exact syntax to use within your client interface.
MaxShapePointsPerManeuver	The number of shape points returned per maneuver (part of the journey). If zero, no shape point data will be returned. A higher number will result in more detailed route drawings, but at the expense of memory and speed. The default is zero. If you will draw route highlights and do not use server sessions, set this to 50 for typical route highlights.
AvoidAttributeList	A <code>StringCollection</code> containing strings that define the types of roads to avoid. Options are different for each data set. Please refer to “Routing Constants” on page 118 for the list of choices and the exact syntax to use within your client interface. The default is empty, which signifies that all road types are permitted.
CoverageName	The name of a route pool on the server. If omitted or the empty string, the server will use a route selector to choose the best data; the server will use its default route data selector unless you override this behavior in the <code>CovSwitcher</code> property.
Language	A string representing the language used for the routing narrative. Please refer to “Routing Constants” on page 118 for the exact syntax to use within your client interface. The default is English.

Table 12, RouteOptions Properties (continued)

Property	Description
<code>AvoidGefIdList</code>	A list of road unique IDs to avoid if possible. The default is an empty list, signifying no restrictions.
<code>AvoidAbsoluteGefIdList</code>	A list of road unique IDs to avoid. If the route is not possible, an error will be returned in the routing result code. The default is an empty list, signifying no restrictions.
<code>MaxGEFIDsPerManeuver</code>	The maximum number of road segment unique IDs to return for each maneuver in driving directions. The default is zero because most developers do not need them. If you need them, set this property to 65335. Note that this is different from visual shapes configured by <code>MaxShapePointsPerManeuver</code> .
<code>CovSwitcher</code>	The name of a route data selector pool on the server. If omitted or the empty string, the server will use its default route data selector. This property is rarely used because most servers only have one route data selector. This property is ignored if the <code>CoverageName</code> property is set to a non-empty string.

Route Types

The `RouteType` property in the `RouteOptions` object is very important for routing. Choose its value carefully, or let users choose the type of route they want. Please refer to “Routing Constants” on page 118 for the exact syntax to use for this constant within your client interface.

Fastest Routes

The Fastest route type requests the shortest time route. This is the default.

Shortest Routes

The Shortest route type requests the shortest distance route.

Pedestrian Routes

The Pedestrian route type requests a route only on roads that permit walking.

Optimized Routes

The Optimized route type requests an optimized route for use with 4 or more locations. The server keeps the first and last location in the same order but reorders intermediate locations for the best route. To get the new location order, check the routing results (`RouteResults` class) property `Locations`.

This route type works with fewer than 4 locations, but the optimization behavior is undefined. With fewer than 4 locations, instead specify the route type Fastest or Shortest.

SelectDataSetOnly Routes

The `SelectDataSetOnly` route type allows developers to use server-based route data selectors even when routing data is split among multiple servers. In that case, first perform a `SelectDataSetOnly` route request to determine the route pool. Get the selected route data set within the route results (class `RouteResults`) property `CoverageName`.

Next, request a route with a *specific route pool*, possibly from a different server and a different client object. For details, refer to “Specifying a Specific Route Pool” on page 117.

Route Highlight API

There are two methods of displaying the driving directions calculated by the MapQuest routing API, depending on how you use the mapping API:

- Route Highlights With Server Sessions.
- Route Highlights With Stateless Maps.

Route Highlights With Server Sessions

If you use MapQuest server sessions to manage map states between page views, you can provide `DoRoute` your session ID string to automatically add drawing shapes called *route highlights* to your map. Learn more about session ID strings in the section “Sessions Overview” on page 43.

The route shapes that are created exist only within the session on the server. They are not accessible by developers, nor are they ever downloaded by the `GetSessionEx` client object method. You cannot customize the shapes created using this API; route options like `MaxShapePointsPerManeuver` and `MaxGEFIDsPerManeuver` are ignored.

Using `DoRoute` with server sessions changes the center point and scale of the map to “best fit” the displayed route. Note that if your map was configured to “best fit” POIs on the map using the `BestFit` map command, POI “best fit” settings will be ignored.

Note that the route shapes do **not** include symbols for origin and destination locations. Refer to Chapter 5, “Displaying POIs,” for displaying simple POIs. For sample code demonstrating this technique with routing, refer to the project `RouteItWithServerSession`.

If necessary, you can delete the route highlight from a server session using the client object method `DeleteRoute`.

Displaying route highlights for maps is easiest when using server sessions. However, if you do not need MapQuest server sessions to maintain map state in a Web application, do **not** create server sessions just to simplify drawing route highlights. Also, do not generate drawing shapes or unique IDs (GEF IDs) during

routing unless your application uses them, because that would unnecessarily reduce overall server performance and increase the size of the server's response.

If you do not use MapQuest server sessions and you want to display route highlights, you can draw them using map annotations. See the next section for details.

Route Highlights With Stateless Maps

If you do not use server sessions, you can draw a route highlight on your stateless map using an instance of the map annotation class `LinePrimitive`. The `LinePrimitive` class is discussed in detail in Chapter 6, "Map Annotations."

If you are not already familiar with the map annotation APIs, see Chapter 4, "Basic Mapping," and Chapter 6, "Map Annotations."

By default, drawing shapes are **not** created during a route request. To generate the shapes, set the route option `MaxShapePointsPerManeuver` to 50 for typical route highlights.

To increase application and server performance, reduce the number of shape points in a route using the `LatLngCollection` class method `Generalize`.

This example combines all shape points within 0.01 miles to a single shape point. This reduces map drawing time and shortens a stateless map's URL. Shortening the map URL minimizes the possibility of exceeding some Web browser URL length limitations.

The following code snippet illustrates this technique:

```
// This assumes that routeResults contains the route results
MQClientInterface.LinePrimitive lpRtHlt = new
MQClientInterface.LinePrimitive();
lpRtHlt.Color           = MQClientInterface.ColorStyle.GREEN;
lpRtHlt.Key             = "RouteShape";
lpRtHlt.Style           = MQClientInterface.PenStyle.SOLID;
lpRtHlt.CoordinateType = MQClientInterface.CoordinateType.GEOGRAPHIC;
lpRtHlt.DrawTrigger     =
    MQClientInterface.DefaultTrigger.DRAW_AFTER_ROUTE_HIGHLIGHT;
lpRtHlt.Width           = 200;

// simplify the route by reducing points that are very close
lpRtHlt.LatLngs = routeResults.ShapePoints;
lpRtHlt.LatLngs.Generalize(0.01);
//Add the line primitive to a primitiveCollection
MQClientInterface.PrimitiveCollection pcMap = new
MQClientInterface.PrimitiveCollection();
```



```
pcMap.Add(lpRtHlt);  
mqSession.AddOne(pcMap);  
  
// scale the map using the BestFit map command!  
MQClientInterface.BestFit bfMap = new MQClientInterface.BestFit();  
bfMap.ScaleAdjustmentFactor = 1.2;  
bfMap.IncludePrimitives      = true;  
mqSession.AddOne(bfMap);
```

For more detailed sample code with code comments and beginning/end markers, please refer to the sample code `RouteItNoServerSession`.

The approach above draws the shapes for the entire route at once. If desired, you can get shapes for each maneuver in the journey for “turn by turn” sub-maps. For details, see “Getting Individual Steps of the Route” on page 115.

Routing Errors

After calculating a route with `DoRoute`, check the `ResultCode` property of the route results object that you passed to `DoRoute`. If the result matches the constant for success, then proceed to use, display, or map the results as desired.

If the result code is something else, the route was not successful. You can use the result code number to identify the cause (or one of the causes) of the calculation failure. See Table 13 for details.

Table 13, Routing Result Codes

Result Codes	Meaning
SUCCESS	Routing was successful.
INVALID_LOCATION	At least one location is invalid.
ROUTE_FAILURE	Unable to calculate route.
NOT_SPECIFIED	Unknown error.
NO_DATASET_FOUND	No suitable route data set could route all input locations.

Please refer to “Routing Constants” on page 118 for the exact syntax to use for these constants within your client interface.

You can get more error information by calling the `RouteResults` method `ResultMessages`. This method returns a `StringCollection` that contains more detailed error numbers and descriptive text in an ASCII string. For instance:

```
"100 Unable to use location #1 : Must have a valid Lat/Lng."
```

You can examine the error code number at the beginning of the string and define your own error messages as appropriate. See Table 14 for the most common error codes returned by `ResultMessages`.

Table 14, Routing `ResultMessages` Codes

Result Code	Meaning
100	Unable to use a location. The descriptive text usually provides details, such as which location number in the collection, and what was invalid, such as a latitude or longitude coordinate, or an invalid road segment GEF ID.
101	The <code>LocationCollection</code> must contain at least two locations.
102-199	At least one location is invalid.
200-299	Routing calculations failed.
300-399	At least one route option is invalid.
Other values	Other routing errors.

You can use the descriptive error messages as desired, but MapQuest recommends that your final application use the error codes and display application-specific error text to users.

Please refer to the routing sample code `RouteItWithServerSession` and `RouteItNoServerSession` for examples of using the routing error APIs.

Basic Routing Steps

The following are the steps for calculating a route between two or more locations from addresses:

1. Create `Address` objects and populate with address information.
2. Use the Geocoding API to geocode the locations into `GeoAddress` objects. (See Chapter 3, “Geocoding.”)
3. If geocoding succeeded, display the address match returned by the geocoder; otherwise, display an appropriate error to the user.
4. Create a `LocationCollection` object and populate it with geocoded locations in the following order: origin, optional midpoints, and destination.
5. Create a `RouteResults` object to store routing results.

6. Create a `RouteOptions` object.
7. If desired, change other route options. See Table 12 on page 109.
8. Calculate the route using the client object method `DoRoute`.
9. Look at the `RouteResults` object's result codes.
10. If there were errors, display them as appropriate.
11. If the route succeeded, display a map using the mapping APIs and/or display the text narrative.

For full sample code that illustrates these steps, please refer to the `RouteItNoServerSession` and `RouteItWithServerSession` sample code projects.

Getting Individual Steps of the Route

After completing a route, you can get information about each step in the journey. The server can describe each of these *maneuvers* as narrative text or shapes.

If you had only 2 input locations in the route (a beginning and an end), all the maneuvers are stored together in one collection. If you had 3 or more input locations, each set of maneuvers are organized by the larger sections of the journey between input locations. These larger sections of the journeys are represented by the MapQuest class `TrekRoute`.

You will find the maneuver information within the route results (class `RouteResults`) property `TrekRoutes`. That property contains a collection of `TrekRoute` objects which contain detailed information about each maneuver.

For instance, with only 2 input locations, the route results property `TrekRoutes` will contain exactly one `TrekRoute` object. The `TrekRoute` object contains a `Maneuvers` property which is a collection of `Maneuver` objects. Each one of these `Maneuver` objects can provide valuable information through its properties and methods.

Get the text narrative in the `Narrative` property.

- In the .NET interface, the `ShapePoints` property will contain the shape points for the entire route if you requested shapes with `MaxShapePointsPerManeuver`.
- Get the time and distance for each maneuver in the `Time` and `Distance` properties.
- For advanced developers who want a customized user interface that uses road attribute, turns, and sign information, check the API Reference for details of the `Maneuver` class. Some of these features require comparison to MapQuest constants. Refer to “Routing Constants” on page 118 for the exact syntax required.

For an example of how to use the MapQuest API to generate a text narrative, refer the sample code `RouteItNoServerSession` and `RouteItWithServerSession`.

Other Routing APIs

Route Matrix Calculation

The client object method `DoRouteMatrix` can automate calculations of driving distance and driving time between many locations provided in a location collection containing `GeoAddress` objects.

`DoRouteMatrix` will only return driving distance and driving time for each pair of locations examined. `DoRouteMatrix` will not return visual routes, route narratives, maps, route shapes, detailed error messages for each route, or other route results. If you expect to need that information or plan to re-generate that information eventually, you should call `DoRoute` for individual point-to-point routes rather than using `DoRouteMatrix`.

There are two ways to use `DoRouteMatrix`. In some situations, you might want a series of calculations from an origin point to many destinations, the results of which would be a *one-to-all route matrix*. To calculate an one-to-all matrix, set the `DoRouteMatrix` argument `AllToAll` to false and include the origin location first in the location collection that you will pass to `DoRouteMatrix`.

In other situations, you might want a larger matrix that describes the time and distance between every combination of two locations among a set of 3 or more locations, the results of which would be an *all-to-all route matrix*. For a all-to-all matrix, set `AllToAll` to true.

Just like a standard `DoRoute` request, you can define route options such as the route data set to use, roads to avoid, and whether distances should be returned as miles or kilometers. Configure these settings using a standard `RouteOptions` object configured similarly to how you would use it with the client object method `DoRoute`. However, the only route type currently supported is `Fastest`, which is the default. There are other many route options which will be ignored by `DoRouteMatrix` because they do not apply.

To store the results of `DoRouteMatrix`, create an empty (new or initialized) instance of class `RouteMatrixResults` and pass it to `DoRouteMatrix`. After calculating the route matrix, you can call the results object's methods and get its properties.

Before using these or other results, always check the `ResultCode` property, which indicates success, failure, or partial success. Refer to "Routing Constants" on page 118 for instructions on using these result code constants in your client interface.

If the result code is `Success`, every combination of origin and destination will yield valid time and distance information. If the result code is `PartialSuccess`, valid time and distance information is available for some combinations of locations, but some distances or times might be 0 to indicate an error for that origin and destination.

You can get more detailed error messages if the result code is `InvalidLocation`, `RouteFailure`, `ExceededMaxLocations`, `InvalidOption`, or `PartialSuccess`.

Check these messages by calling the `RouteMatrixResults` and `ResultMessages` and iterating through the messages. Each result message should contain a unique number followed by a text, such as “200 Unable to calculate route.”

You can programatically check for the numbers at the beginning if you want to check for the type of error. Message numbers 300 to 399 indicate invalid options. Message numbers 100 to 199 indicate one or more invalid locations. Message numbers 300 to 399 indicate route failures.

There are typically only one or two messages in the result messages collection. The number of messages in the result messages collection does not increase with the number of input locations.

Refer to the sample code project `SearchAndDisplayItWithRouteDistance` for an example of using `DoRouteMatrix` in conjunction with the search API to “search” for locations within a certain driving time or driving distance. Learn more on this subject in “Advanced Searching With Route Matrix API” on page 103.

Advanced Routing Location Specification

There may be situations where you want to use the routing API with one or more locations for which you have a latitude and longitude but not a street address. This might be useful if you have latitude and longitude from mouse clicks on maps, locations from dynamic databases, or cached latitude and longitude values from previous geocoding.

To get latitude and longitude coordinates from mouse clicks on maps, see “Other Point-and-Click APIs” on page 63.

To define a location from latitude and longitude, simply create a `GeoAddress` object and set the `LatLng` property with the latitude and longitude coordinates. You can then use that `GeoAddress` as a location for the routing API. You do **not** need to know or set the street segment unique ID (GEF ID) within `GeoAddress`.

In general, do **not** set the object’s `GEFID` property. The server will automatically find the closest street segment unique ID from the latitude and longitude.

See also: Chapter 3, “Geocoding,” p. 19; “Reverse Geocoding,” p. 36.

Specifying a Specific Route Pool

Although using server-based route selectors is strongly recommended, you can also specify a specific route pool on the server to use. If you do this, you will specify a route pool by name. If you need to learn more about the available route pools on your server, refer to “Learning Your Server Configuration” on page 13.

To do use a specific route pool, set the route pool name in the `CoverageName` property within the `RouteOptions` object.

You will typically only specify a specific route pool if you previously used the `SelectDatasetOnly` route type. For more details, refer to “SelectDataSetOnly Routes” on page 111.

Getting Location Data After a Route

After a route request, the server provides some information in the routing results (`RouteResults` class) property `Locations`. This is particularly useful if you have used the `Optimized` routing type because the locations have been reordered.

You can get the distance from each location to its representative road segment in the `DistAlong` property within each `GeoAddress` object in this collection. Note that the `DistAlong` property has a different meaning in geocoding and routing. The geocoding engine sets this property to the distance along the nearby road segment within the geocoding data. The server modifies this value in routing results to represent the distance from the location to the nearby road segment. The distance unit (miles or kilometers) is configured by the route option property `NarrativeDistanceUnitType`.

Although it is rare to need the GEF IDs for each location, get GEF IDs for each location in the `GeoAddress` object’s `GEFID` property. If it’s non-zero, it is the GEF ID for that road segment in the map data found during the route calculation.

Routing Constants

.NET developers should use routing constants with the exact syntax shown below.

Route Types

Used with the `RouteOptions` class to set the type of route to calculate.

```
MQClientInterface.RouteType.FASTEST
MQClientInterface.RouteType.SHORTEST
MQClientInterface.RouteType.PEDESTRIAN
MQClientInterface.RouteType.OPTIMIZED
MQClientInterface.RouteType.SELECT_DATASET_ONLY
```

Narrative Types

Used to set the `RouteOptions` class to set the type of textual directions.

```
MQClientInterface.NarrativeType.DEFAULT // default narrative
MQClientInterface.NarrativeType.HTML    // possibly with HTML
MQClientInterface.NarrativeType.NONE    // no text narrative
```

Distance Units

Used to describe miles or kilometers. For instance, as a routing preference for the words to use in textual narratives.

```
MQClientInterface.DistanceUnits.MILES  
MQClientInterface.DistanceUnits.KILOMETERS
```

Route Result Codes

Used to test the results of the Route client object method.

```
MQClientInterface.RouteResultsCode.SUCCESS  
MQClientInterface.RouteResultsCode.INVALID_LOCATION  
MQClientInterface.RouteResultsCode.ROUTE_FAILURE  
MQClientInterface.RouteResultsCode.NOT_SPECIFIED  
MQClientInterface.RouteResultsCode.NO_DATASET_FOUND
```

Heading Direction

Used with the Maneuver class to describe a direction for one step of a route.

```
MQClientInterface.Maneuver.HEADING_NULL  
MQClientInterface.Maneuver.HEADING_NORTH  
MQClientInterface.Maneuver.HEADING_NORTH_WEST  
MQClientInterface.Maneuver.HEADING_NORTH_EAST  
MQClientInterface.Maneuver.HEADING_SOUTH  
MQClientInterface.Maneuver.HEADING_SOUTH_EAST  
MQClientInterface.Maneuver.HEADING_SOUTH_WEST  
MQClientInterface.Maneuver.HEADING_WEST  
MQClientInterface.Maneuver.HEADING_EAST
```

Turn Types

Used with the `Maneuver` class to describe a turn for one step of a route.

```
MQClientInterface.Maneuver.TURN_TYPE_STRAIGHT
MQClientInterface.Maneuver.TURN_TYPE_SLIGHT_RIGHT
MQClientInterface.Maneuver.TURN_TYPE_RIGHT
MQClientInterface.Maneuver.TURN_TYPE_SHARP_RIGHT
MQClientInterface.Maneuver.TURN_TYPE_REVERSE
MQClientInterface.Maneuver.TURN_TYPE_SHARP_LEFT
MQClientInterface.Maneuver.TURN_TYPE_LEFT
MQClientInterface.Maneuver.TURN_TYPE_SLIGHT_LEFT
MQClientInterface.Maneuver.TURN_TYPE_RIGHT_UTURN
MQClientInterface.Maneuver.TURN_TYPE_LEFT_UTURN
MQClientInterface.Maneuver.TURN_TYPE_RIGHT_MERGE
MQClientInterface.Maneuver.TURN_TYPE_LEFT_MERGE
MQClientInterface.Maneuver.TURN_TYPE_RIGHT_ON_RAMP
MQClientInterface.Maneuver.TURN_TYPE_LEFT_ON_RAMP
MQClientInterface.Maneuver.TURN_TYPE_RIGHT_OFF_RAMP
MQClientInterface.Maneuver.TURN_TYPE_LEFT_OFF_RAMP
MQClientInterface.Maneuver.TURN_TYPE_RIGHT_FORK
MQClientInterface.Maneuver.TURN_TYPE_LEFT_FORK
MQClientInterface.Maneuver.TURN_TYPE_STRAIGHT_FORK
```

Maneuver Attributes

Used to with the `Maneuver` class to describe a direction for one step of a route.

```
MQClientInterface.Maneuver.ATTRIBUTE_POSSIBLE_SEASONAL_ROAD_CLOSURE
MQClientInterface.Maneuver.ATTRIBUTE_PORTIONS_TOLL
MQClientInterface.Maneuver.ATTRIBUTE_PORTIONS_UNPAVED
MQClientInterface.Maneuver.ATTRIBUTE_POSSIBLE_SEASONAL_ROAD_CLOSURE
MQClientInterface.Maneuver.ATTRIBUTE_GATE
MQClientInterface.Maneuver.ATTRIBUTE_FERRY
```

Routing Avoidance Types

Used with the `RouteOptions` class to set certain types of roads to avoid.

```
MQClientInterface.RouteOptions.AVOID_ATTRIBUTE_LIMITED_ACCESS
MQClientInterface.RouteOptions.AVOID_ATTRIBUTE_TOLL_ROAD
MQClientInterface.RouteOptions.AVOID_ATTRIBUTE_FERRY
MQClientInterface.RouteOptions.AVOID_ATTRIBUTE_UNPAVED_ROAD
MQClientInterface.RouteOptions.AVOID_ATTRIBUTE_SEASONAL
```


Routing Languages

Used with the `RouteOptions` class to set languages for textual driving directions.

```
MQClientInterface.RouteOptions.LANGUAGE_ENGLISH  
MQClientInterface.RouteOptions.LANGUAGE_FRENCH  
MQClientInterface.RouteOptions.LANGUAGE_GERMAN  
MQClientInterface.RouteOptions.LANGUAGE_ITALIAN  
MQClientInterface.RouteOptions.LANGUAGE_SPANISH  
MQClientInterface.RouteOptions.LANGUAGE_DANISH  
MQClientInterface.RouteOptions.LANGUAGE_DUTCH  
MQClientInterface.RouteOptions.LANGUAGE_NORWEGIAN  
MQClientInterface.RouteOptions.LANGUAGE_SWEDISH  
MQClientInterface.RouteOptions.LANGUAGE_IBERIAN_SPANISH  
MQClientInterface.RouteOptions.LANGUAGE_BRITISH_ENGLISH  
MQClientInterface.RouteOptions.LANGUAGE_IBERIAN_PORTUGUESE
```

Route Matrix Result Code Constants

Used with the client object method `DoRouteMatrix` to get results from this method from instances of the `RouteMatrixResults` class.

```
MQClientInterface.RouteMatrixResultsCode.SUCCESS  
MQClientInterface.RouteMatrixResultsCode.INVALID_LOCATION  
MQClientInterface.RouteMatrixResultsCode.ROUTE_FAILURE  
MQClientInterface.RouteMatrixResultsCode.NO_DATASET_FOUND  
MQClientInterface.RouteMatrixResultsCode.NOT_SPECIFIED  
MQClientInterface.RouteMatrixResultsCode.INVALID_OPTION  
MQClientInterface.RouteMatrixResultsCode.PARTIAL_SUCCESS  
MQClientInterface.RouteMatrixResultsCode.EXCEEDED_MAX_LOCATIONS
```

9 Application Design

For some applications, knowing how to design your application and how to design the user interface are key parts of your final solution. This chapter discusses user interface recommendations, designing your code for performance, debugging techniques, and common problems during development.

User Interface Recommendations

Geocoding User Interface

When display space allows, display the full addresses of geocode matches even if you assume the geocoded address was successful and identical to the location requested. This helps users detect misspellings or omitted address elements that resulted in unexpected “best guesses” by the server.

Mapping User Interface

For the most part, users expect maps to look a certain way, even if users are not conscious of specific map conventions. For this reason, you are discouraged from dramatically changing the look and feel of maps using Advantage API.

Unconventional map appearance requires more time to visually process new road colors, fill colors, line widths, map symbols, and map feature labeling. If you make map style changes, strongly consider soliciting feedback from end users before finalizing any custom map styles or behaviors.

The most useful map style customizations are icons for custom Points of Interest and limited use of map annotations. Some applications may want to show or hide certain map features (either universally or within certain scale ranges) by changing map feature visibility for all objects of certain display types.

Increasing Contrast Between Maps and POIs

There are several approaches to increase contrast between maps and POIs:

- Use icons of varied size and/or icon depending on the context. For example, when a map is “zoomed in” to street-level, display a larger icon than when the map is “zoomed out” to a large or regional area.
- Consult with a graphic designer to design custom POI icons that visually stand out when displayed on the map.

- Hide unnecessary map feature objects by changing map feature visibility for objects based on display types.
- If maps will be on high resolution devices or are likely to be printed, increase the display resolution and/or use a vector-based image format like Encapsulated PostScript (EPS). See “Downloading Map Images” on page 55 for details.

See also: “Customizing Map Appearance,” p. 56.

Routing User Interface

The two standard routing user interfaces are a textual narrative of the route and a visual description called a route highlight. Where display space allows, display both the textual narrative of the route and the map with the route highlight.

Showing turn-by-turn sub-maps with route highlighting are an additional user interface option. If you do this, you might want them off by default and let the user choose that setting. This should be optional because it is always resource intensive for the server and some users find the user interface cluttered.

Also, remember to set the distance units for the textual narrative to miles or kilometers, whichever is more appropriate for your application. You could even set this option differently for each user. See Table 12, “RouteOptions Properties,” on page 109 for details.

Designing for Performance

For all applications with a high volume of server requests and/or a high network latency, you should be concerned about improving application performance.

There are things that developers can do to improve performance:

- Cache MapQuest API objects when you might need them again.
- If you are finished with a MapQuest API object but want to re-use it for a similar task, re-initialize it with the object’s `Init` method instead of destroying it. Note that you should generally not call `Init` on an object previously added to a collection because it will initialize the object within the collection.
- If you display maps in a non-browser application, cache the map image bitmap so you do not have to re-query the sever for the map image for minor display updates.
- If you continuously use a specific set of simple POIs (POIs not from a database), cache the `LocationCollection` instead of constantly re-creating it.
- POI display and searching are fastest when location collections are small and not in server-queried databases. Database-based POI display and proximity searches increase client-server response time

and reduce overall performance. If you have a very large set of POIs and you can do a “pre-search” to find POIs in the area of interest, then create simple POIs with the search results.

- If you geocode an address and there is a good chance that you might need the results again, save the important information from the `GeoAddress` (and the result code) so you do not have to repeat the MapQuest API requests.
- If you use the searching or routing API and might need the results again, cache them. Routing is extremely resource-intensive on the server.
- If you have more than 10,000 POIs, use *spatial IDs* in your POI data to improve performance in database POI display and searching.

Debugging Techniques

Here are some suggestions for developing bug-free applications:

- As with any programming code, add “assertion” code whenever possible to check for expected values.
- Check all return values from all MapQuest API calls that return result codes or potential error codes.
- Wrap exception-handling code around all MapQuest API functions and handle exceptions appropriately. There is only one exception symbol that will be thrown from MapQuest APIs, but the error message will vary to help diagnose problems.
- In debugging releases of your code, print out all errors to the user interface. For instance, if you are using the routing API, you can use the `ResultMessages` API to obtain user-readable messages. See “Routing Errors” on page 113.
- For geocoding problems, display the full addresses of matches in all returned `GeoAddress` objects, even if you assume the geocoded address was successful and identical to the location requested. Display multiple matches and all result codes in order to reduce confusion about ambiguous addresses or potentially flawed geocoding rules. During debugging, you might also consider displaying the geocode rules used during the geocoding.

Appendix A: Display Types

A *display type* (DT) is a number that identifies a map feature's type. For example, there is a different display type number for schools, dirt roads, highways, rivers, and lakes. Many display types are pre-defined in map data, but you can define your own display types to represent custom data like Points of Interest (POIs).

If you are going to create your own Points of Interest or additional map features for use with Advantage API, you must choose a display type for it. In most situations, you should use a display type not used by map data, so your features do not accidentally interact with default styles or behaviors defined on the server.

DT Values for Developer Use

If you want to use display types not currently used, refer to Table 15 and use values shown within the ranges appropriate for the feature type. For POIs, use the primary range 3072-3583. If you need more display types, use the secondary range listed in the table.

These values may change in future software or map data releases.

Table 15, Developer Display Type Ranges

Feature Type	Developer DT Range
Roads	256-511
Other lines	768-1023
Polygons	1280-1535
Points, including POIs	3072-3583 (primary range) 1792-2047 (secondary range)

DT Values in Licensed Mapping Data

If your application examines or uses DTs that are used by licensed map data, you need to know how to interpret these display types.

Applications may also use this information to augment licensed map data with new features with DTs that match DTs in map data. This might be useful if your new features had the same meaning as built-in features and you wanted all of the style behaviors of your new features to match the built-in appearance (color, line width, etc.) and behaviors (visibility at a given scale).

Important details of the DTs listed below:

- Map data may not use all DTs shown.
- Map data may use DTs that are not listed
- Map data may use undocumented DTs with meanings similar to those that are documented below.
- The *exact* meaning of a specific DT is different in every data set.
- These values may change in future software or map data releases.

Common DT Ranges

Although the values listed below are only rough guides, here are the DT ranges of common map features and their feature types. A feature's DT number must match the feature type shown for that DT range.

Table 16, Common Display Types, General Ranges

Feature Type	General DT Range
Lines: road and bridges	0-511
Lines: other	512-1023
Polygons	1024-1536
Points: data augmentation	1536-2047
Points: user points	3072-3583
Vendor-specific or reserved	All others

Common DTs With Details

Table 17, Common Display Types, Details

Roads and Bridges (Line Features)			
0000	DTR_LIMITEDACCESS	0015	DTR_ROADOTHERTHOROUGH
0001	DTR_PRIMARY	0050	DTR_ROAD1
0002	DTR_SECONDARY	0051	DTR_ROAD2
0003	DTR_OTHER	0052	DTR_ROAD3
0004	DTR_ACCESSRAMP	0053	DTR_ROAD4
0005	DTR_SERVICEDRIV	0054	DTR_ROAD5
0006	DTR_WALKWAY	0055	DTR_ROAD6

Table 17, Common Display Types, Details

0007	DTR_4WHEEL	0056	DTR_ROAD7
0008	DTR_FERRY	0057	DTR_ROAD8
0009	DTR_ALLEY	0058	DTR_ROAD9
0012	DTR_ROADSPECIAL	0059	DTR_ROAD10
0014	DTR_STAIRWAY		
Other Line Features			
0512	DTL_COASTLINE	0552	DTL_COUNTYBOUNDARY
0520	DTL_RAILROAD	0554	DTL_COUNTRYBOUNDARY
0540	DTL_OCEAN	0555	DTL_CANAL
0541	DTL_LAKE1	0556	DTL_CITYBOUNDARY
0542	DTL_LAKE2	0557	DTL_STBOUNDARYNONUS
0543	DTL_RIVER1	0600	DTL_AERIALTRAM
0544	DTL_RIVER2	0601	DTL_MISCGROUNDTRANS
0545	DTL_SUBJECT	0602	DTL_PIPEPOWERLINE
0546	DTL_BUILTUP	0603	DTL_CARLINE
0547	DTL_AIRPORT	0604	DTL_COGRAILROAD
0548	DTL_PARK	0605	DTL_CONTINENTALDIV
0549	DTL_STATEPARK	0606	DTL_SHRLINEINTERMIT
0550	DTL_NATIONALPARK	0766	DTL_GENERICLINE
0551	DTL_ZIPBOUNDARY	0767	DTL_FRAME (map frame)
Polygon Features			
1040	DTP_OCEAN	1063	DTP_CAMPGROUND
1041	DTP_LAKE1	1064	DTP_FEDERALLAND
1042	DTP_LAKE2	1065	DTP_SALTFLAT
1043	DTP_RIVER1	1066	DTP_CANAL

Table 17, Common Display Types, Details

1044	DTP_RIVER2	1067	DTP_QUARRY
1045	DTP_SUBJECT	1068	DTP_ISLAND
1046	DTP_BUILTUP	1069	DTP_NATNLGRASSLAND
1047	DTP_AIRPORT	1070	DTP_WILDLIFEREFUGE
1048	DTP_PARK	1071	DTP_NATNLSEASHORE
1049	DTP_STATEPARK	1072	DTP_WILDERNESSAREA
1050	DTP_NATIONALPARK	1073	DTP_RESERVOIR
1051	DTP_ZIP	1074	DTP_MARSH
1052	DTP_COUNTY	1075	DTP_GLACIER
1053	DTP_STATE	1076	DTP_LAKEINTERMITTENT
1054	DTP_COUNTRY	1077	DTP_LAKEDRY
1055	DTP_MILITARY&RESERVE	1078	DTP_RUNWAY
1056	DTP_GOLFCOURSE	1079	DTP_SHRLINEINTERMIT
1057	DTP_STADIUM	1080	DTP_WATERBODYGENERAL
1058	DTP_SHORELINEPERREN	1081	DTP_SUBURBS
1059	DTP_RRSTATION	1277	DTP_GENERICPOLY
1060	DTP_SHOPPINGCTR	1278	DTP_FRAME
1061	DTP_HOSPITAL	1279	DTP_LAND
1062	DTP_EMPLOYMENTCTR		
Point Features			
1540	DTT1_OCEAN	1715	DTT_SORORITYORFRAT
1541	DTT1_LAKE1	1716	DTT_TRAILERCOURT
1542	DTT1_LAKE2	1717	DTT_BRIDGE
1543	DTT1_RIVER1	1718	DTT_CHAMBERCOMMERCE
1544	DTT1_RIVER2	1719	DTT_COUNTYSEAT1
1545	DTT1_SUBJECT	1720	DTT_COUNTYSEAT2

Table 17, Common Display Types, Details

1546	DTT1_BUILTUP	1721	DTT_COUNTYSEAT3
1547	DTT1_AIRPORT	1722	DTT_COUNTYSEAT4
1548	DTT1_PARK	1723	DTT_COUNTYSEAT5
1549	DTT1_STATEPARK	1724	DTT_COUNTYSEAT6
1550	DTT1_NATIONALPARK	1725	DTT_DAM
1551	DTT_ZIP	1726	DTT_FIREHOUSE
1552	DTT_COUNTY	1727	DTT_NATGRASSLAND
1553	DTT_STATE	1728	DTT_WILDLIFEREFUGE
1554	DTT_COUNTRY	1729	DTT_NATPARKSERVICE
1580	DTT2_OCEAN	1730	DTT_NATSEASHORE
1581	DTT2_LAKE1	1731	DTT_WILDERNESSAREA
1582	DTT2_LAKE2	1732	DTT_INDIANRESERV
1583	DTT2_RIVER1	1733	DTT_POLICESTATION
1584	DTT2_RIVER2	1734	DTT_POSTOFFICE
1585	DTT2_SUBJECT	1735	DTT_MARSH
1586	DTT2_BUILTUP	1736	DTT_EXITNUMBER
1587	DTT2_AIRPORT	1737	DTT_FERRYCROSSING
1588	DTT2_PARK	1738	DTT_GLACIER
1589	DTT2_STATEPARK	1739	DTT_INTERCHANGE
1590	DTT2_NATIONALPARK	1740	DTT_AAASSTARREDPOI
1600	DTT_CITY1	1750	DTT_BUSTERMINAL
1601	DTT_CITY2	1751	DTT_TRAINSTATION
1602	DTT_CITY3	1752	DTT_SCHOOL
1603	DTT_CITY4	1753	DTT_HOSPITAL
1604	DTT_CITY5	1754	DTT_MONUMENT
1605	DTT_CITY6	1755	DTT_MARINA
1606	DTT_STATECAP1	1756	DTT_STADIUM

Table 17, Common Display Types, Details

1607	DTT_STATECAP2	1757	DTT_CONVENTIONCTR
1608	DTT_STATECAP3	1758	DTT_CIVICCTR
1609	DTT_STATECAP4	1759	DTT_FERRYPLAZA
1610	DTT_STATECAP5	1760	DTT_MUSEUM
1611	DTT_STATECAP6	1761	DTT_LIBRARY
1612	DTT_COUNTRYCAP1	1762	DTT_PARKANDRIDE
1613	DTT_COUNTRYCAP2	1763	DTT_CITYHALL
1614	DTT_COUNTRYCAP3	1764	DTT_COURTHOUSE
1615	DTT_COUNTRYCAP4	1765	DTT_RESTAREA
1616	DTT_COUNTRYCAP5	1766	DTT_PARKINGLOT
1617	DTT_COUNTRYCAP6	1767	DTT_TOURISTINFO
1700	DTT_CAMPGROUND	1768	DTT_GOLFCOURSE
1701	DTT_APARTMENT	1769	DTT_AMUSEMENT
1702	DTT_CEMETERY	1770	DTT_TOURISTATT
1703	DTT_RELIGIOUSINST	1771	DTT_ATM
1704	DTT_COUNTYHOME	1772	DTT_HOTEL
1705	DTT_CUSTODIAL	1773	DTT_RESTAURANT
1706	DTT_EMPLOYMENTCTR	1774	DTT_BANK
1707	DTT_FEDERALLAND	1775	DTT_THEATER
1708	DTT_PRISON	1776	DTT_GASSTATION
1709	DTT_GOVERNMENTCTR	1777	DTT_GROCERYSTORE
1710	DTT_LOOKOUTTOWER	1778	DTT_AUTOSERVICE
1711	DTT_MILITARYINSTALL	1779	DTT_BUSINESSFAC
1712	DTT_NURSINGHOME	1782	DTT_AUTOCLUB
1713	DTT_ORPHANAGE	1783	DTT_ENTERTAINMENT
1714	DTT_SHELTERMISSION	1784	DTT_RECREATION

Appendix B: Standard Icons

This appendix lists icons on the server that you can reference by name in your application. The table lists the symbol type of each icon as either raster (listed as R) or vector (listed as V) or both types (R&V). You will need the icon symbol type and the symbol name within your programming code. For the basic use of icons as Points of Interest icons, refer to “Overriding Styles With DTStyle” on page 56.

IMPORTANT: If any error is found in the requested style that prevents it from being displayed, a red triangle (see below) will be shown in its place to denote the error.



Table 18, Standard Icons On the Server


























Pushpins							
Icon	Description	Name	Type	Icon	Description	Name	Type
	Red pushpin	MQ00011	R		Green pushpin	MQ00012	R
	Blue pushpin	MQ00013	R		Yellow pushpin	MQ00014	R
	Cyan pushpin	MQ00015	R		Magenta pushpin	MQ00016	R
	Light Blue pushpin	MQ00017	R		Gray pushpin	MQ00019	R
Dots							
Icon	Description	Name	Type	Icon	Description	Name	Type
	Red dot	MQ00021	R		Green dot	MQ00022	R
	Blue dot	MQ00023	R		Yellow dot	MQ00024	R

Table 18, Standard Icons On the Server (*continued*)

	Cyan dot	MQ00025	R		Magenta dot	MQ00026	R
	Light blue dot	MQ00027	R		Gray dot	MQ00029	R

Stars

Icon	Description	Name	Type	Icon	Description	Name	Type
	Red star	MQ00031	R&V		Green star	MQ00032	R&V
	Blue star	MQ00033	R&V		Yellow star	MQ00034	R&V
	Orange star	MQ00035	R&V		Purple star	MQ00036	R&V
	White star	MQ00037	R&V		Black star	MQ00038	R&V
	Gray star	MQ00039	R&V		Gold star	MQ00040	R&V

Miscellaneous Icons (Colorized)













Icon	Description	Name	Type	Icon	Description	Name	Type
	Home	MQ00201	R		Religious	MQ00202	R
	Winery	MQ00203	R		Tree	MQ00204	R
	Briefcase	MQ00205	R		Hospital	MQ00206	R
	Grocery Store	MQ00207	R		Tourist Attraction	MQ00208	R
	Automatic Teller Machine	MQ00209	R		Automobile Club	MQ00210	R
	Camera	MQ00211	-R		Higher Education	MQ00212	R

Table 18, Standard Icons On the Server (*continued*)





































	Cinema	MQ00213	R		Performing Arts	MQ00214	R
	Film Strip (Cinema)	MQ00215	R		Swimmer	MQ00216	R
	Sports Center	MQ00217	R		Ski Resort	MQ00218	R
	Equestrian	MQ00219	R		Golf Course	MQ00220	R
	Silhouette of Person with Open Book (Library)	MQ00221	R		Rest Area	MQ00222	R
	Train Station	MQ00223	R		Border Crossing	MQ00224	R
	Bell	MQ00225	R		Open Book (Library)	MQ00226	R
	Pharmacy (Rx)	MQ00227	R		Bowling Center	MQ00228	R
	Police Station	MQ00229	R		Small Airport	MQ00230	R
	Planet Earth	MQ00231	R		Coat Hanger (Dry Cleaner)	MQ00232	R
	Fishing	MQ00233	R		Spider	MQ00234	R
	Bus Station	MQ00235	R		Business Facility	MQ00236	R
	Park and Ride	MQ00237	R		Park/Recreation Area	MQ00238	R
	Car Rental	MQ00239	R		Airport	MQ00240	R
	Bank	MQ00241	R		Campfire	MQ00242	R
	Court House/Convention Center/City Hall	MQ00243	R		Restaurant	MQ00244	R
	Gasoline/Petrol Station	MQ00245	R		Historic Site	MQ00246	R
	Hotel	MQ00247	R		Museum	MQ00248	R

Table 18, Standard Icons On the Server (*continued*)



















	Pizza Slice	MQ00249	R		Shopping	MQ00250	R
	Camping	MQ00251	R		Commuter Rail	MQ00252	R
	Automobile Service and Maintenance	MQ00253	R		Casino	MQ00254	R
	Parking Lot/Parking Garage	MQ00255	R		Ferry Terminal	MQ00256	R
	Nightlife	MQ00257	R		Motorcycle Dealership	MQ00258	R
	Marina	MQ00259	R		Tourist Information	MQ00260	R
	Ice Skating Rink	MQ00261	R		Government Building	MQ00262	R
	Urn (Museum)	MQ00263	R		Amusement Park	MQ00264	R
	Start	MQ09191	R		End	MQ09192	R

Table 18, Standard Icons On the Server (*continued*)

Miscellaneous Icons (Black & White)

































Icon	Description	Name	Type	Icon	Description	Name	Type
	Home	MQ00301	R		Religious	MQ00302	R
	Winery	MQ00303	R		Tree	MQ00304	R
	Briefcase	MQ00305	R		Hospital	MQ00306	R
	Grocery Store	MQ00307	R		Tourist Attraction	MQ00308	R
	Automatic Teller Machine	MQ00309	R		Automobile Club	MQ00310	R
	Camera	MQ00311	-R		Higher Education	MQ00312	R
	Cinema	MQ00313	R		Performing Arts	MQ00314	R
	Film Strip	MQ00315	R		Swimmer	MQ00316	R
	Sports Center or Complex	MQ00317	R		Ski Resort	MQ00318	R
	Equestrian	MQ00319	R		Golf Course	MQ00320	R
	Silhouette of Person with Open Book (Library)	MQ00321	R		Rest Area	MQ00322	R
	Train Station	MQ00323	R		Border Crossing	MQ00324	R
	Bell	MQ00325	R		Open Book (Library)	MQ00326	R
	Pharmacy (Rx)	MQ00327	R		Bowling Center	MQ00328	R
	Police Station	MQ00329	R		Small Airport	MQ00330	R
	Planet Earth	MQ00331	R		Coat Hanger (Dry Cleaner)	MQ00332	R

Table 18, Standard Icons On the Server (*continued*)





































	Fishing	MQ00333	R		Spider	MQ00334	R
	Bus Station	MQ00335	R		Business Facility	MQ00336	R
	Park and Ride	MQ00337	R		Park/Recreation Area	MQ00338	R
	Car Rental	MQ00339	R		Airport	MQ00340	R
	Bank	MQ00341	R		Campfire	MQ00342	R
	Court House/Convention Center/City Hall	MQ00343	R		Restaurant	MQ00344	R
	Gasoline/Petrol Station	MQ00345	R		Historic Site	MQ00346	R
	Hotel	MQ00347	R		Museum	MQ00348	R
	Pizza Slice	MQ00349	R		Shopping	MQ00350	R
	Camping	MQ00351	R		Commuter Rail	MQ00352	R
	Automobile Service and Maintenance	MQ00353	R		Casino	MQ00354	R
	Parking Lot/Parking Garage	MQ00355	R		Ferry Terminal	MQ00356	R
	Nightlife	MQ00357	R		Motorcycle Dealership	MQ00358	R
	Marina	MQ00359	R		Tourist Information	MQ00360	R
	Ice Skating Rink	MQ00361	R		Government Building	MQ00362	R
	Urn (Museum)	MQ00363	R		Amusement Park	MQ00364	R

Table 18, Standard Icons On the Server (continued)

Labeling Icons							
	Black eightball	MQ00400	R		Purple eightball	MQ00401	R
	White eightball	MQ00402	R		Red eightball	MQ00403	R
	Green eightball	MQ00404	R		Gold box	MQ00405	R
	Black box	MQ00406	R		Circle and box	MQ00407	R
none	Clear icon. Fully transparent graphic for use with text-only POIs.	MQ00408	R&V				

Glossary

address element

One of three basic parts of an address: street address (including street name and house number), administrative areas (including city and state), and postal code.

admin area

See *administrative area*.

administrative area

A general name for an area representing an official government or bureaucratic designation. Examples include cities, towns, counties, US states, and countries.

anchor

The center point for a radius search.

API

Application Program Interface. A set of routines, protocols, and tools for building software applications. An API provides the building blocks for a programmer to create a desktop application, Web application, or another type of software module.

ASCII

American Standard Code for Information Interchange. A standard for defining codes for information exchange between equipment produced by different manufacturers. ASCII represents English characters as numbers, with each letter assigned a number from 0 to 127.

block

A street segment, usually bounded by street intersections. For instance, if Main St. crossed only 1st Street and 2nd Street, the section of Main St. between those streets would be one block. Also called a *segment*.

bounding rectangle

A map's visible area defined by a rectangle on the projected two dimensional view of the world.

browser

A program that accesses and displays files available on the World Wide Web. Also known as a *Web browser*.

browser application

An application that display its user interface using HTML and a Web browser.

centroid

The weighted center of an asymmetrical area, used by Advantage API data to geocode postal codes or administrative areas such as cities, states, or Canadian provinces.

client

Software modules that request mapping functionality, usually as part of a Web application running on a server. Clients communicate via TCP/IP to the server and may or may not run on the same computer as the MapQuest server.

client object

An instance of the MapQuest object of class name `Exec` that contains the most important MapQuest API functions.

client directory

The directory containing your MapQuest client interface files, which were created by the installer if you chose to install client interfaces at the time. All client directories are in the `clients` subdirectory of your MapQuest Directory. For instance, if you are using the C++ client interface and your MapQuest Directory on UNIX is `/mq`, your client directory would be `/mq/clients/c++`.

confidence level

A confidence level returned with geocoding results allows you to interpret what the server does or does not know about the address match. The four confidence levels have the general meaning of exact, good, approximate, and unknown/unused. See also *quality type constant*.

coverage

Advantage API data from one vendor for a specific geographic area, and consists of one or more of the following data types: mapping, geocoding, routing, or map styles. Coverage is also used to refer to the geographic region for map data.

coverage area

A data set's polygon-defined geographic region containing valid data.

database

A collection of data arranged for ease and speed of search and retrieval.

database pool

Server configuration settings for databases used for Points of Interest (POIs). Developers will generally interact with database pools only by a string representing the name of the database pool.

database server

A software module that provides storage for and access to a database. You will need a database server to use geocoding or proximity searching. The database server may or may not run on the same computer as the MapQuest server.

descriptive search criteria

Search criteria that define and/or limit the types of objects to find. For instance, search criteria that finds restaurants could limit the search to Italian restaurants. Contrast with *geographic search criteria*.

directionals

Part of a street name that indicates a cardinal direction. Examples include North, South, East, and West. Some street addresses include directionals, for instance 33 North Main Street. The term also includes directionals that are abbreviated, such as 123 N. Main Street.

direct URL

A URL that is used to make a stateless map, which is a map that doesn't use MapQuest server sessions. Create a direct URL with client object method `GetMapDirectURLEx`.

display type

A number that identifies a map feature's general type. For example, there are different display types for schools, dirt roads, highways, rivers, and lakes. Many display types are defined in map data, but you can assign custom display types to represent custom Points of Interest.

DT

See *display type*.

Exec object

See *client object*.

feature

A general term for an object on a map. Features can be objects embedded in mapping data on the server, or be created with MapQuest classes such as `PointFeature`, `LineFeature`, or `PolygonFeature`.

GEF ID

See *unique ID*.

geocode result code

A string returned from the geocoding API, containing five characters representing a returned item's granularity level and confidence levels for the street name, administrative areas (in USA, city and state names), and postal code.

geocode pool

Server configuration settings for geocoding data. Developers will generally interact with geocode pools only by a string representing the name of the geocode pool.

geocoding

Geocoding an address (or another location) assigns location coordinates called latitude and longitude that uniquely identify a location on the planet.

geographic search criteria

Search criteria that define where a point is located. For example, within 10 miles of a location or within a specified rectangular area. Contrast with *descriptive search criteria*.

granularity codes

Two characters (a letter followed by a number) that represent a primary granularity level and a granularity subcode. Developers use granularity codes to interpret geocode results, specifically embedded in the five-character geocode result codes.

granularity level

A granularity level is a qualitative measure of accuracy of a geocoded address or search result. For example, the highest granularity level is a numbered address, such as "123 Main Street" in a specific city. A low granularity example would be a country, represented by its weighted center point.

HTML

HyperText Markup Language. HTML is a layout format for describing document contents and is the format of standard pages on the World Wide Web.

HTTP

Hypertext Transfer Protocol. The set of rules for requesting files on the World Wide Web. HTTP is based on the TCP/IP networking protocol.

IP address

An address defined by the Internet Protocol, which is part of the TCP/IP protocol. IP specifies the addressing scheme and format of data packets on a network. IP address are of the form X.Y.Z.A, with those letters corresponding to numbers 0 through 255. See also *TCP/IP*.

latitude

A coordinate indicating north and south position on the planet, centered around the equator. See also *longitude*.

link

See *segment*.

local session object

An instance of MapQuest class *Session*. Developers add objects to the local session object to define everything necessary to display a map, including the map center point, map scale, Points Of Interest, map annotations, database queries, and other information. MapQuest developers create a local session object independent of whether they use server sessions or use stateless maps.

longitude

A coordinate indicating east and west position on the planet, centered around an arbitrary dividing line called the Prime Meridian. See also *latitude*.

maneuver

One segment of a journey that corresponds to one line in a textual narrative like “Take California Highway 1 South for 350 miles.” A maneuver includes the transition from the previous maneuver, for instance turning right onto a street.

map data selection

The decision process on the server for which map data to display when multiple map data sets included overlapping coverage areas. For example, the server could show one coverage for street-level detail and another for national overview maps.

map feature

See *feature*.

map pool

Server configuration settings for mapping data. Developers will generally interact with map pools only by a string representing the name of the map pool.

map style file

A file that contains values to control the appearance of a map. Developers do not directly reference one, but instead refer to the name of a *style pool* on the server that uses it.

map styles

Map styles describe the appearance of a map at a particular scale. This includes how to draw and label highways, how thickly to draw roads, and where to find graphics that represent custom icons and logos. There are default map styles defined on the server in style pools, but you can override these styles using MapQuest APIs.

mapping

The server process of generating visual maps from map data.

MapQuest Directory

The installation directory for MapQuest files on your local disk. The recommended path is `/mq/` for UNIX and Linux, or `C:\mq\` for Windows. Most of the configuration files, examples, and sample code assume this default location.

MapQuest Enterprise engine

This server software provides the core MapQuest technologies: geocoding, mapping, routing, and proximity searching.

MapQuest Listener

This server software intercepts network requests from clients and queues requests for the MapQuest Enterprise engine.

match

One of several results from a geocoding request.

match code

See *geocode result code*.

match type constant

A special code used for specifying a granularity level when requesting geocoding of an address. See *granularity level*.

narrative

Textual driving directions returned by the MapQuest routing API.

nodata

A special map data set displayed when a requested map has a center point and map scales not supported by any map data set. This data set tells the user that there is no data available. This is useful because it allows users and developers to distinguish between sparse map data (no roads or other features visible) versus accidental zooming/panning beyond valid ranges. This feature requires map data selection to be enabled.

non-browser application

An application that does not display its user interface using HTML and a Web browser. This includes most desktop applications, such as those written in Visual Basic and C++. This also includes Java applications displayed with a Java user interface. However, it would not include Java server code whose purpose is to generate HTML code with image tags intended to be displayed by Web browsers.

ODBC

Open Database Connectivity. An API providing database access using a common interface, which typically requires software drivers to connect to databases from a specific vendor.

panning

Display a different part of a map by moving it north, south, east, west, or a combination of those directions, without changing the size or scale of the map.

pixel coordinates

See *X/Y coordinates*.

Points of Interest (POIs)

Custom locations that provide additional context on a map, and typically are displayed as icons with an optional text label.

pools

Pools are collections of server resources of a certain type. Most server resources are grouped into pools of identical items for increased performance. As a developer, you will refer to server resources such as map data and geocoding data by referring to a pool name, which is an identifying string for that server resource.

postal code levels

One of four postal code variants in a region, although some are typically unused in a data set. In USA data, there are three postal code levels that correspond to ZIP codes (level 1), ZIP+2 codes (level 2), and ZIP+4 codes (level 3). In Canadian data, there is only type of postal code, which is represented by postal code level 3.

proximity searching

A search within a geographic area for Points of Interest or other map features. The search area could be a radius around a location, in a certain bounding region, or within a certain distance along a path. For example, users could search for all video rental locations within 10 miles of their morning commute, or search a complex geographic region for any parks.

protocol

An agreed-upon format for transmitting data between two devices.

quality type constant

A constant used by developers during geocoding that corresponds to a minimum confidence in a match. The three quality type constants have the general meaning of exact, good, or approximate. See also *confidence level*.

road type

Part of a street name that includes words like Road, Street, Avenue, Lane, etc.

route pool

Server configuration settings for routing data. Developers will typically interact with route pools only by a string representing the name of the route pool

routing

The term for calculating navigational directions from one location to another location, with optional intermediate destinations.

scale

The proportion of map units to real-world units, analogous to a “zoom” setting in a desktop computer program. For example, if a map’s scale is expressed as 10,000, this means a scale ratio of 1:10,000, so one map unit represents 10,000 real-world units.

segment

A road segment that runs between two street intersections or dead-ends on one side. Synonymous with *block*.

server sessions

See *session management*.

server status page

An HTML page generated by your Advantage API server that displays your server configuration. It is generated through standard HTTP requests, and can be displayed with any Web browser.

session management

In general, HTML pages and Web applications are stateless, meaning they exist only for the time required for a page display. For the Web developer, this means that a Web page request contains no information about recent actions. A session object describes the current state of a map. Developers can choose to store or duplicate session information on the server, a process called creating server sessions.

session ID string

A string that represents a MapQuest server session object on the server. This is contrasted with a *local session object*, an instance of class `Session` that exists only in the client application.

spatial ID

Your custom location data can include *spatial IDs* to improve performance of proximity searches on extremely large POI data sets. The Spatial Indexing Tool, which is provided with the product, converts latitude and longitude to spatial IDs.

spatial index

See *spatial ID*.

SQL

Standard Query Language. A standard interactive and programming language for getting information from and updating a database. Queries take the form of a command language that lets you select, insert, update, find out the location of data, and so forth. There is also a programming interface.

stateless maps

Maps that do not use MapQuest server sessions. Stateless maps use a different type of image URL to retrieve the map image. A stateless map uses a *direct URL*, which encapsulates all necessary information into one long URL.

street type

See *road type*.

style file

See *map style file*.

style objects

MapQuest objects that override default map styles on the server. The most common style object class is `DTStyle`, but there are also the classes `DTStyleEx` and `DTFeatureStyleEx`.

style pool

Server configuration settings for map styles, which are configured on the server with map style files. Developers typically interact with style pools only by a string representing the name of the style pool.

TCP

Transmission Control Protocol. TCP enables two hosts to establish a virtual connection and exchange streams of data. TCP guarantees delivery of data and also guarantees that packets will be delivered in the same order in which they were sent. Software modules on the same computer can communicate via TCP; they do not need to be on separate computers. See also *TCP/IP*.

TCP/IP

Transmission Control Protocol/Internet Protocol. The suite of communications protocols used to connect hosts on the Internet. TCP/IP uses several protocols, the two main ones being TCP and IP. Software modules on the same computer can communicate via TCP/IP; they do not need to be on separate computers.

unique ID

A unique identifier of a street segment or other map feature within data from one data vendor. Unique IDs are not unique across data vendors. However, these values are shared across data sets from one vendor, for instance geocoding and map data from NAVTEQ. Unique IDs are called *GEF IDs* in some MapQuest APIs.

URL

A Uniform Resource Locator, a string of characters that specify a request for a resource, generally over the Internet or intranet. In MapQuest documentation, URLs usually refer to an HTTP protocol request for a resource from the Advantage API server. Because the MapQuest client-server protocol is based on HTTP, requests like getting map images, requesting the server status page, and other client-server requests are implemented with HTTP URLs.

WHERE clause

Part of an SQL query that further limits the results returned. For example, a WHERE clause might limit database query results to records that contain a certain ID number in a certain database column. See also *SQL*.

XML

Extensible Markup Language. XML is a data description language with flexible and adaptable information layout. It looks similar to HTML, but its format and tags are significantly more customizable.

X/Y coordinates

Coordinates on a displayed map, where (0,0) represents the upper left coordinate and the lower right coordinate is defined by pixels in the map. Note that if you change the display resolution (the dots per inch), you must take this into consideration when using functions that convert to or from X/Y coordinates from latitude and longitude.

ZIP code

A service mark in the United States for a system to expedite sorting and delivery of mail by assigning a five-digit number to each delivery area.

ZIP+2 code

A United States postal code area defined by its ZIP code plus 2 additional digits. A ZIP+2 code defines a smaller (more accurate) area than a ZIP code and larger (less accurate) area than a ZIP+4 code.

ZIP+4 code

A United States postal code area defined by its ZIP code plus 4 additional digits. A ZIP+4 code will usually defines a smaller (more accurate) area compared to both a ZIP code and a ZIP+2 code. Areas defined by a ZIP+4 code areas can be as small as a city block in a dense area, but may also correspond to a single high-volume mail recipient.

zoom level

A predefined map *scale* to which users can quickly “zoom in” or “zoom out.” The product defines default zoom levels to simplify zooming code. Developers can customize zoom levels using the `AutoMapCovSwitch` class.

Symbols

@ sign 24

A

abbreviations in geocoding 27, 30

AddOne method 44, 49, 51, 62, 80
 special behavior of 44

Address class 32, 34, 114

address element 23, 26
 in geocoding 26

address ranges
 in searches 103

addresses from Lat/Long 36

admin areas, See “administrative areas”

administrative areas 138

 granularity 24

 indexes

 international 28

 subcodes 24

AIEPS images 55

all-to-all route matrix 116

anchor 102, 138

anti-alias settings 55

API 138

API Reference 2

 comparison to Developer Guide 2

 location of 8

application design 122

approximate confidence, definition of 27

ASCII 114, 138

assertions 124

“at” sign 24

authentication 3, 11

 IP 11

 password 11

AutoGeocodeCovSwitch class 36

AutoMapCovSwitch class 44, 50, 51, 52, 149

avoidance

 absolute 108

- attempt 108
- by unique ID 110
- road segments 108
- road types 108, 109

avoiding road segments 110

B

- background colors 57
- best fit 111
 - with map annotations 90
- BestFit class 62, 63, 73, 90, 111
- BestFitLL class 62, 63
- bitmap icons 4, 67, 84
 - annotations 80
 - creating 66
- block 138
 - granularity 24
 - subcode 24
- bounding rectangle 138
- browser 138
- browser application 139
- browser limitations 48
- buffer width 102

C

- caching values for performance 123
- Canadian postal codes 23
- Canadian province 37
- center
 - of corridor searches 103
 - of radius searches 102
- Center class 62
- CenterLL class 62
- centroids 20, 139
- circle annotations 80
- cities 24
- class name strings 45
- client

- defined 139
- client directory 8, 139
- client ID 11, 15
- client object 6, 7, 139
- client objects
 - with Advantage API 16
- closest points
 - in radius searches 103
- ClosestPoint 103
- collections
 - iterating through 95
- colors 57
- confidence levels 29
 - detailed meanings of 27
- connection information
 - with Advantage API 16
- connection problems 13
- connection timeouts 13
- contrast
 - increasing 122
- cookies 47, 48
- coordinate type 80
- corridor search 101
- CorridorSearchCriteria class 101
- counties 24
- countries 24
- coverage 139
- coverage area 139
- coverage switching 41
 - advanced 50
 - disabling 51
- CoverageName 52
- CoverageStyle class 44, 52, 56
- crashes 48
- CreateSessionEx method 45

D

- Data Manager 1, 4
- data sets

- limiting in mapping 50
- data vendors
 - limiting in mapping 50
- database 140
 - pools 96
- database POIs
 - versus simple POIs 66
- database pool names in Advantage API 16
- database search
 - by key field 76, 77
- database server 4, 140
- database tables
 - overriding 96
- databases
 - searching 94
- DataVendorCodes 50
- datum
 - NAD83 21
 - WGS84 21
- DBLayerQuery class 94, 96
- DBLayerQueryCollection class 44, 96
- debugging 124
- DeleteSession method 46
- deleting sessions on server 46
- descriptive search criteria 93, 140
- development questions 3
- direct URLs 43, 47
- directionals 26, 27
- display coordinate type 80
- display resolution, See “resolution”
- display type 56, 57, 67, 68, 94, 140
 - and feature types 75
 - collection 96, 98
 - definition of 42
 - details of built-in values 125
 - developer range 42, 68, 97
 - ranges 125
 - standard range 42
- display type collection 96
- DisplayState class 55, 56

- DistAlong meaning changes 118
- distance after searches 103
- distance units 98, 102, 109, 123
- DoRoute method 107, 108, 111, 115
- DoRouteMatrix method 103, 104, 116, 117
- dots per inch 55
- DPI 41, 49
- drawing icons 4, 56, 67, 84
 - annotations 80
- drawing order
 - map annotation 80
- drawing primitives 79
 - See also “map annotations”
- DT, See “display type”
- DTCollection class 71, 96, 98
- DTFeatureStyleEx class 59
- DTStyle class 56, 57, 67, 68, 72
- DTStyleEx class 58, 59

E

- ellipse annotations 80
- EllipsePrimitive class 80, 82
- encryption 3, 12, 18
- EPS 55, 56
 - Adobe Illustrator EPS images 55
 - images 56, 87
- error
 - codes in routing 114
 - handling 47
 - messages in routing 114
- exact confidence, definition of 27
- exception
 - handling 124
- exceptions
 - catching 47
- Exec class 16
- Exec object, See “client object”
- extra criteria string 76
- ExtraCriteria property 77

F

- feature 141
- feature collections
 - searching 97
- feature types
 - and DTs 75
- FeatureCollection class 44, 67, 68, 71, 73, 75, 95, 98
- features
 - find by name 56
 - finding by unique ID 56
- firewalls 13
- font sizes 57
- fonts
 - with Advantage API 56, 87
- fuzzy matching 27

G

- GEF ID 114, 148
 - See also “unique ID”
- generalizing a route 102, 112
- GeoAddress class 28, 37, 107, 108, 114, 117
- Geocode method 32, 35
- geocode pool 34
- geocode result code 29
- geocode results 29
 - understanding 28
- geocode selectors 22
 - viewing server configuration 14
- GeocodeOptions class 33, 34
- GeocodeOptionsCollection class 33, 34
- geocoding 3, 19, 19–39, 141, 141–??
 - and routing 108
 - reverse 36
 - rules 23
- geocoding rules 20, 22, 33, 34
- geographic coordinate type 80
- geographic search criteria 93, 141
- GetDrawnFeatures method 73
- GetMapDirectURL method 62

GetMapDirectURLEx method 55, 61, 62
GetMapFromSessionURL method 55, 62
GetMapImageDirect method 55
GetMapImageFromSession method 55, 58, 59, 60, 61, 62
GetObject 45
GetRoadGefIdLL class 64
GetRoadGefIdXY class 64
GetServerInfo method 14
GetServerInfoURL method 15
GetSession method 49, 75, 89
GetSessionEx method 46, 48, 49, 64, 75, 89, 95, 111
GIF images 53, 55, 64
Global Positioning System 21
GMFDraw application 4, 67
good confidence, definition of 27
GPS 21
granularity
 administrative area 24
 block 24
 code 24
 codes 24
 intersection 24
 level 29
 location 24
 postal 25
 subcode 24
granularity level 24
GRFEdit application 4, 67

H

Helvetica font 56, 87
hidden HTML form properties 47, 48
hiding features and POIs, See “visibility”
HTML 142
HTML IMG tag 55
HTML pages
 managing state in 43
HTTP 6, 12, 142, 148
HTTP proxies 13

I

- icons 57, 80
 - icon layer 81
 - on the server 67
 - when displaying routing results 107
 - See also “bitmap icons” and “drawing icons”
- IdentifyCriteria class 71
- IdentifyFeature method 70, 71, 93
- image headers 4, 67
- image tags
 - in HTML 55
- IMG tag 55
- Init method 123
- IntCollection class 50
- international addresses 28
- international administrative areas 28
- international postal codes 23
- Internet firewalls 13
- intersection
 - granularity 24
 - specification 24
- intersections, specifying in geocoding 24
- IP address authentication 11
- IP, definition of 142
- iterating through collections 95

K

- keep center during best fit 63
- key field 17, 76, 77
- key value 76
- key values
 - in map annotations 89
 - in POIs 74
 - in searching 95, 99, 100, 101
- kilometers 102, 109, 123

L

- languages

- in routing 109
- Latitude 20
- latitude and longitude
 - routing with 117
- LatLng class 37
- LatLngCollection class 64, 101, 102
- layers, See “map layers” 81
- line annotations 80
- LineFeature class 71, 73, 95, 97
- LinePrimitive class 80, 81, 112
- link 142
- LLToPix method 56
- local session object 43, 45, 46, 48, 49, 53, 58, 80, 142
 - getting objects from 45
 - See also “session”
- location granularity 24
- LocationCollection class 32, 34, 37, 107, 108
- long URLs 48
- longitude 20

M

- maneuver 109, 142
- maneuvers 115
- map annotation
 - drawing order 80
- map annotations 79–93
 - best fit 90
 - defining initial position 80
 - key values 89
- map appearance
 - customizing 122
- map commands 50, 62
- map coverage 40
- map data searching 97
- map data selection 39, 41, 64
- map data selection, See “coverage switching” 143
- map feature 42
 - See “feature”
- map images

- from URLs 55
 - how to download 55
- map layers 81
- map pool 40, 97, 143
- map pool name of found POIs 75
- map selectors
 - viewing server configuration 14
- map size, setting in inches or pixels 49
- map style data
 - compared to mapping data 40
- map style files 143
- map style objects, See “style objects”
- map style strings 58
- map styles 42, 143
- map URLs 53
 - length issues 48
- MapCommand class 44
- MapDataSelector pool 50, 51
- mapping 3, 93–93, 143
- MapQuest architecture 3
- MapQuest Directory 8, 143
- MapQuest Enterprise clients 4
- MapQuest exceptions 47
- MapQuest Listener 144
- MapState class 44, 48, 49, 53, 61, 62, 89
- match 144
- match code 144
- match type 24, 34, 144
 - constants 38
- MatchType, See “match type”
- MaxMatches 29, 30, 33, 34
- memory leaks 48
- miles 102, 109, 123
- misspellings 122
- misspellings in geocoding 27
- multi-location routing 106
- multiple point routing 106

N

- NAD83 datum 21
- narrative types 109
- nearest numbered block subcode 24
- network problems 13
- non-bitmap images 56, 87
- non-browser application 144
- numbered roads in geocoding 27

O

- ODBC 144
- one-to-all route matrix 116
- opacity of drawing primitives 80
- optimized routes 106, 118
- overriding styles 56

P

- Pan class 62
- Pan method 46, 61
- panning 41, 144
- passwords 11, 15
- performance 123
- PixToLL method 56, 100
- PNG images 55
- POI 66–79, 145
 - labels 86
 - labels versus text primitives 86
- point
 - distance 103
- point features
 - modifying styles of 56
- point-and-click APIs 63
- PointCollection class 64
- PointFeature class 67, 68, 71, 73, 94, 95, 97
- Points of Interest
 - definition of 145
 - See also “POIs”
- POIs

- database 66
- display simple POIs
- key values 74
- licensed from MapQuest 74
- simple 66
- polygon annotations 80
- polygon layer 81
- PolygonFeature class 71, 73, 95, 97
- PolygonPrimitive class 80, 88, 89
- PolygonSearchCriteria class 101
- pools 40, 145
 - definition 13
- port number problems 13
- postal code
 - levels 23
 - subcodes 25
- postal code levels 25
- postal codes 23
 - Canadian 23
 - in searches 103
 - international 23
- postal granularity 25
- primary granularity code 24
- Prime Meridian 20
- PrimitiveCollection class 44, 80, 89
- projection 21
- protocol 145
- proxies 13
- proximity searching 3, 145
 - See also “searching”
- proxy servers 13

Q

- quality type 34
 - constants 37
- quality types 29
- QualityType, See “quality type”
- query objects
 - multiple 96

R

- radius search criteria 98
- RadiusSearchCriteria class 98
- raster icons 67
- raster symbols 66
- rectangle annotations 80
- rectangle search criteria 99
- RectanglePrimitive class 80, 85
- RectangleSearchCriteria class 100
- representative block subcode 24
- resolution 41, 49, 55
 - high 66, 123
 - versus scale 41
- ResultMessages 114
- reverse geocoding 16, 36
- ReverseGeocode method 37
- road layer 81
- road segment
 - avoidance 108
- road type 145
- road type avoidance 108
- road type variations in geocoding 27
- road types in geocoding 27
- route
 - generalizing 102, 112
 - searching around 101
 - searching near 101
- route data selection with no calculation 111
- route data selector 110
- route highlight layer 81
- route highlights 81, 107, 123
 - in sessions 44
 - with server sessions 111
 - with stateless maps 112
- route matrix 116
- route pool 106, 109, 146
 - manual selection 117
 - overriding 117
- route selectors 106
 - viewing server configuration 14

- route types 109
- RouteOptions class 107, 108, 115
- RouteResults class 114
- routing 105–122, 146
 - and geocoding 108
 - error messages 114
 - result messages 124
 - with latitude and longitude coordinates 117
 - with mouse clicks 117

S

- samples
 - location of 8
- scale 41, 146
 - versus resolution 41
- scale number 41
- scale ranges
 - restricting style changes to 57, 58
- scale ratio 41
- search based on drive time or distance 103
- search criteria
 - descriptive 93
 - geographic 93
 - radius 98
 - rectangle 99
- Search method 70, 95, 97, 98, 100, 101
- searching 3, 93–??
 - around a path 101
 - built-in map features 94
 - corridor 101
 - databases 96
 - displaying results 73
 - feature collections 94, 97
 - licensed map data 97
 - map data 97
 - narrowing a database search 96
 - streets 94
 - See also “proximity searching”
- segment 146

SelectDatasetOnly 118
SelectDataSetOnly in routes 111
server sessions 43, 105
 when to use 45
Session class 43, 44, 45
 See also “local session object”
session ID string 45, 46, 47, 73, 107, 111, 146
session management 3, 146
session parameter objects 44, 45, 53, 67, 80
 getting 45
session server 4
sessions 3
 common problems during updates 48
 deleting on server 46
 expiration of 47
 getting objects of 45
 in general terms 43
 persistent storage of 47
 updating 48, 62
shape points 109
shortest routes 109
simple POIs 67
 creating 68
 See “POIs”
 versus database POIs 66
snap to zoom level during best fit 63
socket timeouts 13
sound-alike matches 23, 27
Soundex matching, See “sound-alike matches”
SourceLayerName 75
spatial IDs 124
specific route pool 111, 117
spelling errors in geocoding 27
SQL 76, 94, 147
square annotations 80
stateless maps 43, 46, 47, 52
 advantages and disadvantages 47
states 24
street address ranges
 in searches 103

- street addresses 28
- streets
 - searching for 94
- StringCollection class 109
- style aliases 16, 42, 43, 51, 52
- style file 147
- Style File Editor application 4, 56
 - and creating style strings 58
- style modifying object 67
- style objects 42, 56, 57, 58, 59, 72, 147
- style pool 147
- style pool name 42, 51
- style pool names after map data selection 52
- style strings 4, 58
- style-modifying objects, See “style objects”
- styles
 - defaults on server 42
 - overriding 56
- subcode
 - block 24
 - definition of 24
 - nearest numbered block 24
 - postal code 25
 - representative block 24
- subcodes
 - administrative areas 24
- symbol types 84
- SymbolPrimitive class 80
- system architecture 5

T

- table names of found POIs 75
- TCP, definition of 148
- TCP/IP 6, 148
- Technical Support 3
- text annotations 80
- text labels 57
- text layer 81
- TextPrimitive class 80, 86

textual driving directions 109, 123

transparency, See “opacity”

trigger DT 81

turn-by-turn maps 123

U

unique ID 56, 59, 148

- avoidance during routing 110

- avoiding specific during routing 108

- from mouse clicks 64

- in routing 117

UpdateSession method 75

UpdateSessionDirect 49, 61

UpdateSessionDirect method 48, 49, 52, 61, 62, 75, 89

UpdateSessionEx method 46, 49, 59, 60, 61, 62, 64, 75, 82, 83, 87, 88, 89

updating sessions 62

URLs

- map images from 53, 55

- sessions and 47, 48

URLs, definition of 148

user interaction APIs 63

user interface design 122

utility applications 4

V

vector 4

vector icons, See “drawing icons”

vector image formats 56, 87

vector symbols 66

vendors

- limiting in mapping 50

visibility

- map feature 56, 58, 72, 122, 123

W

weighted center points 20, 23

WGS84 datum 21

WHERE clauses 76, 148

X

XML 14

Z

ZIP code 23, 149

ZIP+2 code 23, 149

ZIP+4 area 149

ZIP+4 code 23

zoom levels 39, 41, 50, 149

ZoomIn class 50, 62

ZoomIn method 46, 50, 61

zooming

- and map scales 41

- definition 41

ZoomLevels, See “zoom levels”

ZoomOut class 50, 62

ZoomOut method 46, 50, 61

ZoomToRect class 62

ZoomToRectangleLL method 46, 61

ZoomToRectangleXY method 46, 61

ZoomToRectLL class 62

ZoomXY method 46, 61