Eric Pimpler, Mark Lewin

Building Web and Mobile ArcGIS Server Applications with JavaScript

Second Edition

Build exciting custom web and mobile GIS applications with the ArcGIS Server API for JavaScript

Packt

Building Web and Mobile ArcGIS Server Applications with JavaScript

Second Edition

Build exciting custom web and mobile GIS applications with the ArcGIS Server API for JavaScript

Eric Pimpler
Mark Lewin

BIRMINGHAM - MUMBAI

Building Web and Mobile ArcGIS Server Applications with JavaScript

Second Edition

Copyright © 2017 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: February 2014

Second edition: October 2017

Production reference: 1201017

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78728-052-6

www.packtpub.com

Credits

Authors
Eric Pimpler
Mark Lewin

Reviewers
Ken Doman
Adeyemo Ayodele

Commissioning Editor
Aaron Lazar

Acquisition Editor
Chaitanya Nair

Content Development Editor
Zeeyan Pinheiro

Technical Editor
Ketan Kamble

Copy Editor
Safis Editing

Project Coordinator
Vaidehi Sawant

Proofreader
Safis Editing

Indexer
Aishwarya Gangawane

Graphics
Jason Monterio

Production Coordinator
Shantanu Zagade

About the Authors

Eric Pimpler is the founder and owner of GeoSpatial Training Services (`geospatialtraining.com`) and has over 20 years of experience in implementing and teaching GIS solutions using Esri, Google Earth/Maps, and open source technology. Currently, he focuses on ArcGIS scripting with Python and the development of custom ArcGIS Server web and mobile applications using JavaScript. He is the author of *Programming ArcGIS 10.1 with Python Cookbook*. Eric has a bachelor's degree in geography from Texas A&M University and a master's degree in applied geography with a specification in GIS from Texas State University.

Mark Lewin has been developing, teaching, and writing about software for over 16 years. His main interest is GIS and web mapping. Working for ESRI, the world's largest GIS company, he has been a consultant, trainer and course author, and a frequent speaker at industry events. He has subsequently expanded his knowledge to include a wide variety of open source mapping technologies and a handful of relevant JavaScript frameworks, including Node.js, Dojo, and JQuery. Mark now works within Oracle's MySQL curriculum team, focusing on creating great learning experiences for DBAs and database developers, but remains crazy about web mapping.

About the Reviewers

Ken Doman is a senior frontend engineer at GEO Jobe, a geographic information systems consultant company and ESRI business partner that helps public sector organizations and private sector businesses get the most out of geospatial solutions. Ken has worked for the municipal government and in the private sector. He has experienced many facets of GIS technology, from field data collection to mapping and data analysis to creating and deploying web mapping applications and solutions.

Ken is the author of *Mastering ArcGIS Server Development with JavaScript*. He has also reviewed several books for Packt Publishing, including Building Web and Mobile ArcGIS Server Applications with JavaScript and Spatial Analysis with ArcGIS by Eric Pimpler, ArcGIS for Desktop Cookbook by Daniela Christiana Docan, and ArcPy and ArcGIS by Silas Toms.

I'd first like to thank my wife, Luann, who puts up with my late nights reviewing books like this. I'd like to thank my current employer, GEO Jobe *GIS Consulting, as well as past employers like Bruce Harris and Associates, City of Plantation,* FL *and City of Jacksonville, TX for believing in me and letting me learn so much on the job. Finally, I'd like to thank my creator for putting me where I need to be.*

Ayodele Adeyemo is a geogeek with over four years experience, and is passionate about leveraging data and technology to provide solutions to human, social, and technological problems. He currently works with eHealth Africa in Nigeria with the Global Health and Informatics department to provide user data and technology to solve health issues across communities in Africa. He has experience in building and mentoring geospatial solutions as well as consulting for organizations on how to deploy sustainable geospatial solutions to support their agencies and projects.

Ayodele has worked as GIS Specialist at eHealth Africa and Founder and Latitudes Tech Limited. He has worked on the book *ArcPy and ArcGIS: Automating ArcGIS for Desktop and ArcGIS Online with Python.*

I'd like to thank my manager, Dami Sonoiki; my supervisor, Samuel Aiyeoribe; and my colleagues Samuel Okoroafor and Seun Egbinola.

www.PacktPub.com

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

https://www.packtpub.com/mapt

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at `https://www.amazon.com/dp/1787280527`.

If you'd like to join our team of regular reviewers, you can e-mail us at `customerreviews@packtpub.com`. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

Table of Contents

Preface 1

Chapter 1: Introduction to HTML, CSS, and JavaScript 8

- **Basic HTML page concepts** 8
  - DOCTYPE 9
  - Primary tags 10
  - Validating HTML code 11
- **JavaScript fundamentals** 14
  - Commenting code 14
  - Variables 15
  - JavaScript is case sensitive 16
  - Variable data types 17
  - Decision support statements 18
  - Looping statements 19
  - Functions 20
  - Objects 20
- **Basic CSS principles** 22
  - CSS syntax 23
    - Inline styling 26
    - Internal style sheet 26
    - External style sheet 27
- **Separating HTML, CSS, and JavaScript** 27
- **Summary** 29

Chapter 2: Creating Maps and Adding Layers 30

- **The ArcGIS API for JavaScript Sandbox** 31
- **Basic steps for creating an application with the ArcGIS API for JavaScript** 32
  - Creating HTML code for the page 32
  - Referencing the ArcGIS API for JavaScript 34
  - Loading API modules 35
  - Making sure the DOM is available 38
  - Creating the map 38
  - Creating the page content 40
  - Styling the page 40
  - The entire code 41

More about the map 42
Working with map service layers 44
Using the layer classes 46
Tiled map service layers 47
Dynamic map service layers 48
Adding layers to the map 50
Setting the visible layers from a map service 51
Setting a definition expression 52
Map navigation 53
Map navigation widgets and toolbars 53
Map Navigation with the mouse and keyboard 56
Getting and setting the map extent 57
Map events 58
Summary 60
Chapter 3: Adding Graphics to the Map 61
The four parts of a Graphic 63
Specifying graphic geometry 64
Symbolizing graphics 65
Assigning attributes to graphics 67
Changing graphic attributes in an InfoTemplate 67
Creating the graphic 68
Adding graphics to the GraphicsLayer 69
Multiple GraphicsLayers 69
Practice time 70
Summary 81
Chapter 4: The Feature Layer 82
Creating the FeatureLayer 84
Optional constructor parameters 84
Defining the display mode 85
The snapshot mode 86
The on-demand mode 86
The selection only mode 87
The auto mode 87
Setting a definition expression 87
Feature selection 88
Rendering FeatureLayer 89
Practice time 95
Summary 100
Chapter 5: Using Widgets and Toolbars 101

Adding toolbars to an application 102
Steps for creating a toolbar 103
Defining CSS styles 103
Creating the buttons 105
Creating an instance of the Navigation toolbar 106
User interface widgets 107
The BasemapGallery widget 108
Basemap toggle widget 110
Bookmarks widget 110
The Print widget 112
Layer List widget 113
Time to practice 113
Search widget 118
Gauge widget 118
Measurement widget 119
The Popup widget 120
Legend widget 121
OverviewMap widget 123
Scalebar widget 124
Directions 125
HistogramTimeSlider 126
HomeButton 127
LocateButton 128
TimeSlider 129
LayerSwipe 130
The Analysis widgets 131
Feature editing 132
FeatureService 132
The Editing widgets 133
The Editor widget 133
TemplatePicker widget 135
AttributeInspector widget 137
The AttachmentEditor widget 139
The Edit toolbar 141
Summary 141
Chapter 6: Performing Spatial and Attribute Queries 142
Introducing tasks in ArcGIS Server 142
Overview of attribute and spatial queries 143
The Query object 144
Setting query properties 145
Attribute queries 145

Spatial queries 147
Limiting the fields returned 148
Executing the query with QueryTask 148
Getting the results of the query 150
Practice time with spatial queries 151
Summary 160
Chapter 7: Identifying and Finding Features 161
Using IdentifyTask to access feature attributes 162
Introducing IdentifyTask 162
IdentifyParameters 162
IdentifyTask 163
IdentifyResult 164
Practice time - implementing identify functionality 165
Using FindTask to access feature attributes 171
FindParameters 171
FindTask 172
FindResults 172
Summary 173
Chapter 8: Turning Addresses into Points and Points into Addresses 174
Introduction to geocoding 175
Geocoding with a locator service in the ArcGIS API for JavaScript 175
Input parameter object 176
Input JSON address object 176
Input point object 177
Locator object 177
The AddressCandidate object 178
The geocoding process 178
The reverse geocoding process 179
Practice time with the locator service 179
The Search widget 187
Summary 190
Chapter 9: Directions and Routing 191
Routing task 192
Practice time with routing 195
The Directions widget 202
ClosestFacility Task 204
ServiceArea task 206
Summary 209

Chapter 10: Geoprocessing Tasks 210
Models in ArcGIS Server 211
Using the Geoprocessor task - what you need to know 212
Understanding the services page for a geoprocessing task 213
Input parameters 214
The Geoprocessor task 215
Executing the task 215
Synchronous tasks 216
Asynchronous tasks 216
Practice time with geoprocessing tasks 217
Summary 226
Chapter 11: Geometry Operations 227
The Geometry Service 227
Geometry Service operations 228
Using the Geometry Service 230
The Geometry Engine 234
Practice time with the Geometry Engine 235
Summary 239
Chapter 12: Integration with ArcGIS Online 240
Adding ArcGIS Online maps to your applications by using a webmap ID 240
Adding ArcGIS Online maps to your applications with JSON 243
Practice time with ArcGIS Online 244
Summary 251
Chapter 13: Creating Mobile Applications 252
Compact build of the API 253
Setting the viewport scale 254
Practice time with the compact build 254
Integrating the geolocation API 261
Practice time with the geolocation API 263
Summary 268
5 ppendix: Looking Ahead - Version 4 of the ArcGIS API for JavaScript 269
Steps for creating 2D maps 270
Accessing layers 272
New and changed layers 273
GraphicsLayer 274

FeatureLayer 274
MapImageLayer 275
VectorTileLayer 276
GroupLayers 277
SceneLayers 278
3D mapping and symbology 279
Scenes 279
Creating the map 280
Setting elevation data 282
Setting the camera 282
Specifying the environment 284
Local scenes 286
3D symbology and rendering 286
Summary 291
Index 292

Preface

ArcGIS Server is the predominant platform for developing GIS applications for the web. In the past, you could choose from a number of programming languages to develop web mapping applications with ArcGIS Server, including JavaScript, Flex, and Silverlight. However, the Flex and Silverlight APIs have now retired, leaving JavaScript as the preferred language for developing applications on this platform. Its advantages over other languages include the fact that you can use it to build both web and mobile applications and that it does not require the installation of a plugin: everything runs natively in the browser.

This book will teach you how to build web-based GIS applications using the ArcGIS API for JavaScript. Using a practical hands-on style of learning, you will learn how to build fully functional applications with ArcGIS Server and, in doing so, develop a skillset that is in high demand.

You will learn how to create maps and add geographic layers from a variety of sources, including tiled and dynamic map services. In addition, you'll learn how to add vector graphics to the map and stream geographic features to the browser using a feature layer. Most applications also include specific functionality implemented by ArcGIS Server as tasks. You'll learn how to use the various tasks provided by ArcGIS Server to perform common GIS operations, including queries, identification of features, finding features by attribute, geoprocessing tasks, and more. Finally, you'll learn just how easy it is to develop mobile applications with the ArcGIS API for JavaScript.

What this book covers

`Chapter 1`, *Introduction to HTML, CSS, and JavaScript*, covers the fundamental HTML, CSS, and JavaScript concepts before getting started with developing GIS applications with the ArcGIS API for JavaScript.

`Chapter 2`, *Creating Maps and Adding Layers*, shows how to create a map and add layers to the map. You will learn how to create an instance of the Map class, add layers of data to the map, and display this information on a web page. Map is the most fundamental class in the API as it provides the canvas for your data layers and any subsequent activities that occur in your application. However, your map is useless until you add layers of data. There are several types of data layers that can be added to a map, including tiled, dynamic, feature, and others. You will learn more about each of these layer types in this chapter as well.

`Chapter 3`, *Adding Graphics to the Map*, demonstrates how to display temporary points, lines, and polygons in `GraphicsLayer` on the map. The `GraphicsLayer` is a separate layer that always resides on top of any other layers and stores all graphics associated with the map.

`Chapter 4`, *The Feature Layer*, describes `FeatureLayer`. This inherits from the `GraphicsLayer`, but offers additional capabilities such as the ability to perform queries and selections. Feature layers are also used for online editing of features. Feature layers differ from tiled and dynamic map service layers because feature layers bring geometry information to the client computer to be drawn and stored by the web browser. Feature layers potentially cut down on round trips to the server. A client can request the features it needs and perform selections and queries on those features without having to request more information from the server.

`Chapter 5`, *Using Widgets and Toolbars*, covers out-of-the-box widgets that you can drop into your application for enhanced productivity. These include the basemap gallery. Bookmark, print, and overview map user interface components. In addition, the ArcGIS API for JavaScript also includes helper classes for creating toolbars within your applications, such as navigation and drawing toolbars.

`Chapter 6`, *Performing Spatial and Attribute Queries*, covers the ArcGIS Server Query Task that allows you to perform attribute and spatial queries against data layers in a map service that have been exposed. You can also combine these query types to perform a combination attribute and spatial query.

`Chapter 7`, *Identifying and Finding Features*, covers two common operations found in any GIS application: clicking a feature on the map to identify it, or performing a query to locate features with specific attribute values. In either case, information about particular features is returned. In this chapter the reader will learn how to use the IdentifyTask and FindTask objects to obtain information about features.

`Chapter 8`, *Turning Addresses into Points and Points into Addresses*, covers the use of the `Locator` task to perform geocoding and reverse geocoding. Geocoding is the process of assigning a coordinate to an address while reverse geocoding assigns an address to a coordinate.

`Chapter 9`, *Directions and Routing*, describes how to access ArcGIS Server network analysis services to perform analyses on street networks, such as finding the best route from one address to another, finding the closest school, identifying a service area around a location, or responding to a set of orders with a fleet of service vehicles.

`Chapter 10`, *Geoprocessing Tasks*, allows you to execute custom geoprocessing workflows that you define in ArcGIS Pro desktop software using Model Builder or code as Python scripts. Once published to ArcGIS Server as geoprocessing services, these can be accessed from within your web mapping applications. This is a very powerful feature, and we'll get into one in this chapter.

`Chapter 11`, *Geometry Operations*, describes how to use the ArcGIS Server Geometry Service, and its client-side counterpart, the Geometry Engine, to execute common geometric operations such as buffering a feature and reprojecting from one coordinate system to another.

`Chapter 12`, *Integration with ArcGIS Online*, details how you can use the ArcGIS API for JavaScript to access data and maps created with `ArcGIS.com`, which is a web site for working with maps and other types of geographic information. On this site, you will find applications for building and sharing maps. You will also find useful basemaps, data, applications, and tools that you can view and use along with communities you can join. For application developers the really exciting news is that you can integrate `ArcGIS.com` content into your custom developed applications using the ArcGIS API for JavaScript. This chapter shows you how.

`Chapter 13`, *Creating Mobile Applications*, describes how to build mobile GIS applications using the ArcGIS API for JavaScript. ArcGIS Server support is currently provided for iOS and Android devices. In this chapter, you'll learn about the compact build of the API that makes web mapping applications possible through the use of Dojo Mobile.

`Appendix`, *Looking Ahead at Version 4 of the ArcGIS API for JavaScript*, gives you a broad overview of what to expect in version 4 of the API. This is a complete reworking of the API that is being developed in parallel with version 3.x.

What you need for this book

To complete the activities in this book, you will need access to a web browser--preferably Google Chrome or Firefox. Each chapter contains practices designed to supplement the material presented. You will complete these practices using the ArcGIS API for JavaScript Sandbox to write and test your code. The sandbox can be found at `https://developers.arcgis.com/javascript/3/sandbox/sandbox.html`. All the practices will access publicly available instances of ArcGIS Server, so you will not need to install ArcGIS Server yourself.

Who this book is for

This book is written for application developers who want to develop web and mobile GIS applications using ArcGIS Server and the API for JavaScript. It is primarily oriented toward beginning and intermediate-level GIS developers or to more traditional application developers who may not have developed GIS applications in the past but who are now tasked with implementing solutions on this platform. No prior experience with ArcGIS Server, JavaScript, HTML, or CSS is expected, but it is certainly helpful.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
function computeServiceArea(evt) {
  map.graphics.clear();
  var pointSymbol = new SimpleMarkerSymbol();
  pointSymbol.setOutline = new
  SimpleLineSymbol(SimpleLineSymbol.STYLE_SOLID, new Color([255, 0,
  0]), 1);
  pointSymbol.setSize(14);
  pointSymbol.setColor(new Color([0, 255, 0, 0.25]));
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
function computeServiceArea(evt) {
  map.graphics.clear();
  var pointSymbol = new SimpleMarkerSymbol();
  pointSymbol.setOutline = new
  SimpleLineSymbol(SimpleLineSymbol.STYLE_SOLID, new Color([255, 0,
  0]), 1);
  pointSymbol.setSize(14);
  pointSymbol.setColor(new Color([0, 255, 0, 0.25]));
}
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "clicking the **Next** button moves you to the next screen".

Warnings or important notes appear like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of. To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Downloading the example code

You can download the example code files for this book from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you. You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.

5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Building-Web-and-Mobile-ArcGIS-Server-Applications-with-JavaScript-Second-Edition`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `https://www.packtpub.com/sites/default/files/downloads/Building-Web-and-Mobile-ArcGIS-Server-Applications-with-JavaScript-Second-Edition_ColorImages.pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title. To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy. Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material. We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Introduction to HTML, CSS, and JavaScript

There are certain fundamental concepts that you need to understand before you can get started with developing GIS applications with the ArcGIS API for JavaScript. For those of you already familiar with HTML, JavaScript, and CSS you may wish to skip ahead to the next chapter. However, if you're new to any of these concepts read on. We are only going to cover these topics at a very basic level. Just enough to get you started. For a more advanced treatment of any of these subjects there are many learning resources available including books and online tutorials. Please consult `Appendix` for a more comprehensive list of these resources.

In this chapter we will cover the following topics:

- Basic HTML page concepts
- JavaScript fundamentals
- Basic CSS principles

Basic HTML page concepts

Before we dive into the details of creating a map and adding layers of information you need to understand the context of where the code will be placed when you're developing applications with the ArcGIS API for JavaScript. The code you write will be placed inside an HTML page or JavaScript file. HTML files typically have a file extension of `.html` or `.htm` and JavaScript files have an extension of `.js`. Once you have created a basic HTML page you can then go through the necessary steps to create a basic map with the ArcGIS API for JavaScript.

The core of a web page is an HTML file. Coding this basic file is quite important as it forms the basis for the rest of your application. Mistakes that you make in the basic HTML coding can result in problems further down the line when your JavaScript code attempts to access these tags.

The following is a code example for a very simple HTML page. This example is about as simple as an HTML page can get. It contains only the primary HTML tags `<DOCTYPE>`,`<html>`, `<head>`, `<title>`, and `<body>`:

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Topographic Map</title>
   </head>
  <body>
      Hello World
  </body>
</html>
```

There are a number of flavors of HTML currently in use. Most new HTML pages developed today use HTML5, so we'll focus on HTML5 throughout the book, but without delving into many of its advanced features. Other versions of HTML you are likely to encounter in the wild include HTML 4.0.1 and XHTML 1.0.

DOCTYPE

The first line of your HTML page will contain the DOCTYPE. This is used to tell the browser how the HTML should be interpreted. We'll focus on HTML5 in this book so the first example you see in the following uses the HTML5 DOCTYPE. The two other common doctypes are HTML 4.01 Strict and XHTML 1.0 Strict:

- HTML5:

  ```
  <!DOCTYPE html>
  ```

- HTML 4.01 Strict:

  ```
  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">
  ```

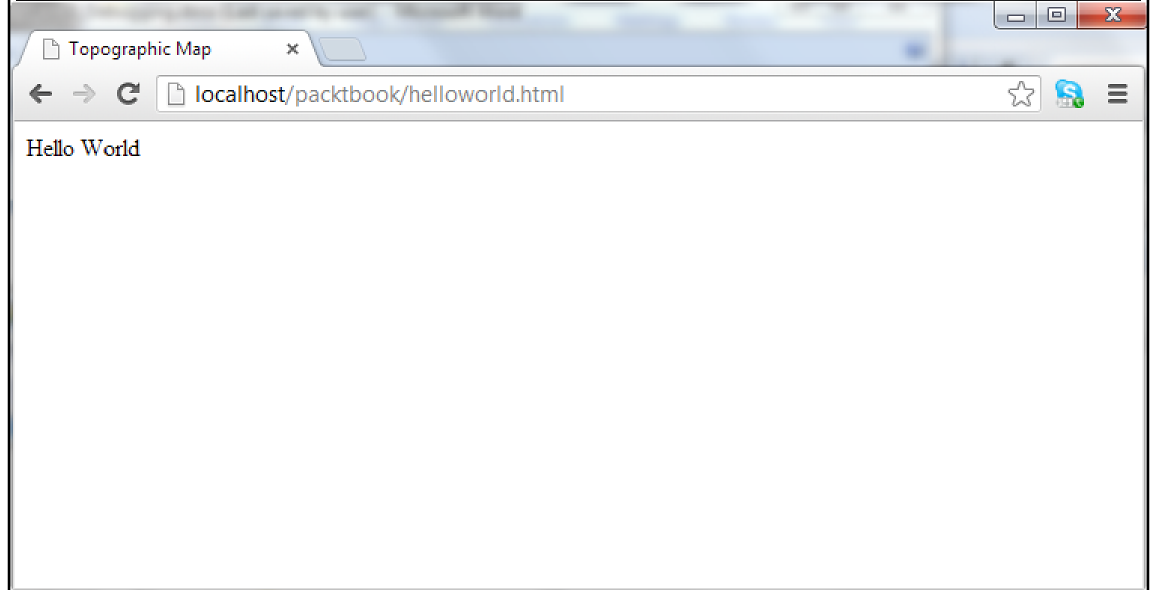
- XHTML 1.0 Strict:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

Primary tags

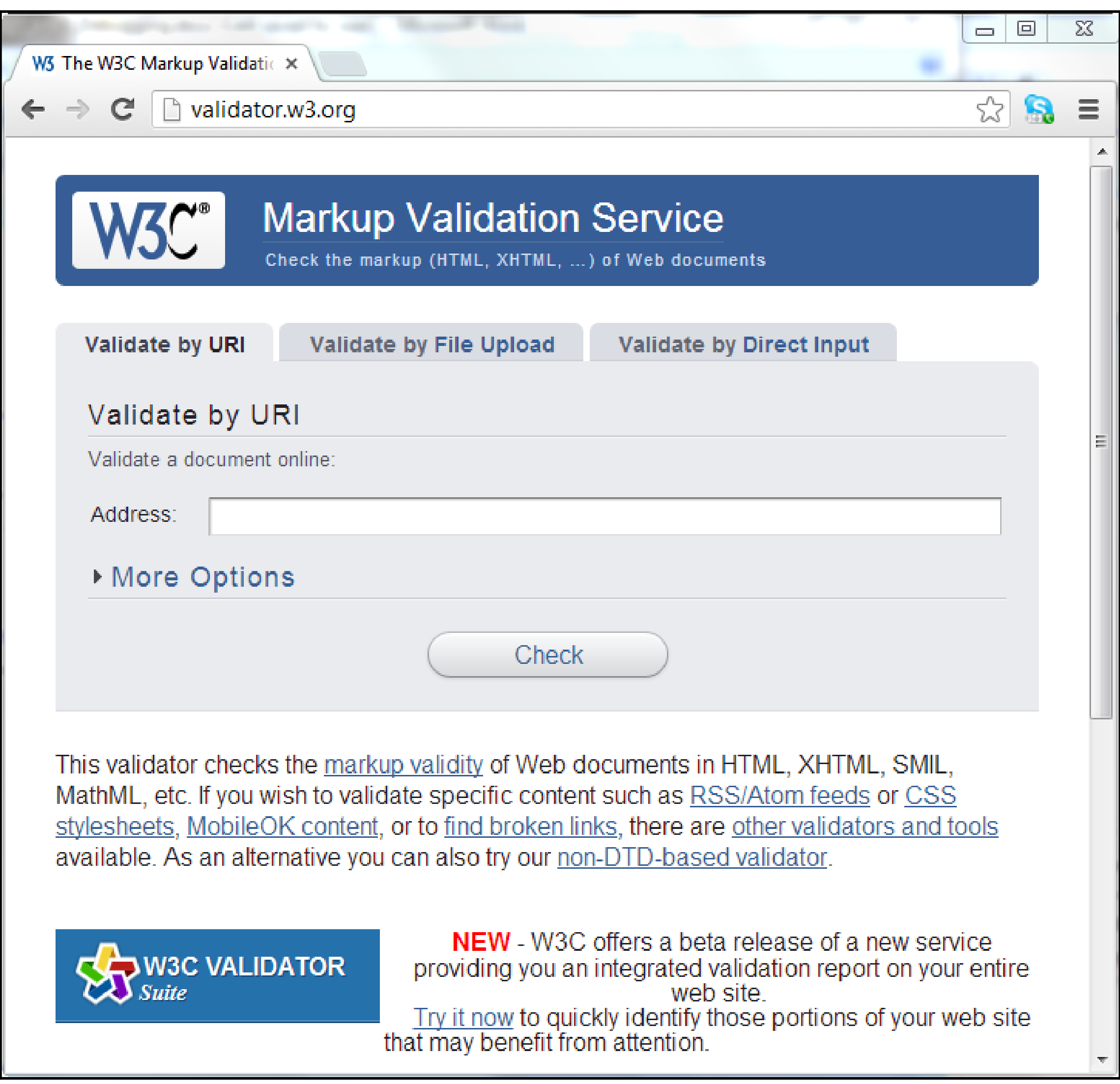
At a minimum, all your web pages will need to contain the `<html>`, `<head>`, and `<body>` tags. The `<html>` tag defines the whole HTML document. All other tags must be placed inside this tag. Tags that define how the web page will appear in the browser are placed inside the `<body>` tag. For instance, your mapping applications will contain a `<div>` tag inside the `<body>` tag that is used as a container for displaying the map.

Loading this HTML page in a browser would produce the content you see in the following screenshot. Most of the ArcGIS API for JavaScript code that you write will be placed between the `<head></head>` tags, either within a `<script>` tag or inside a separate JavaScript file. As you gain experience you will likely begin placing your JavaScript code inside one or more JavaScript files and then referencing them from the `<head>` section. We'll explore this topic later. For now just concentrate on placing your code inside `<script>` tags, within the `<head>` tags:

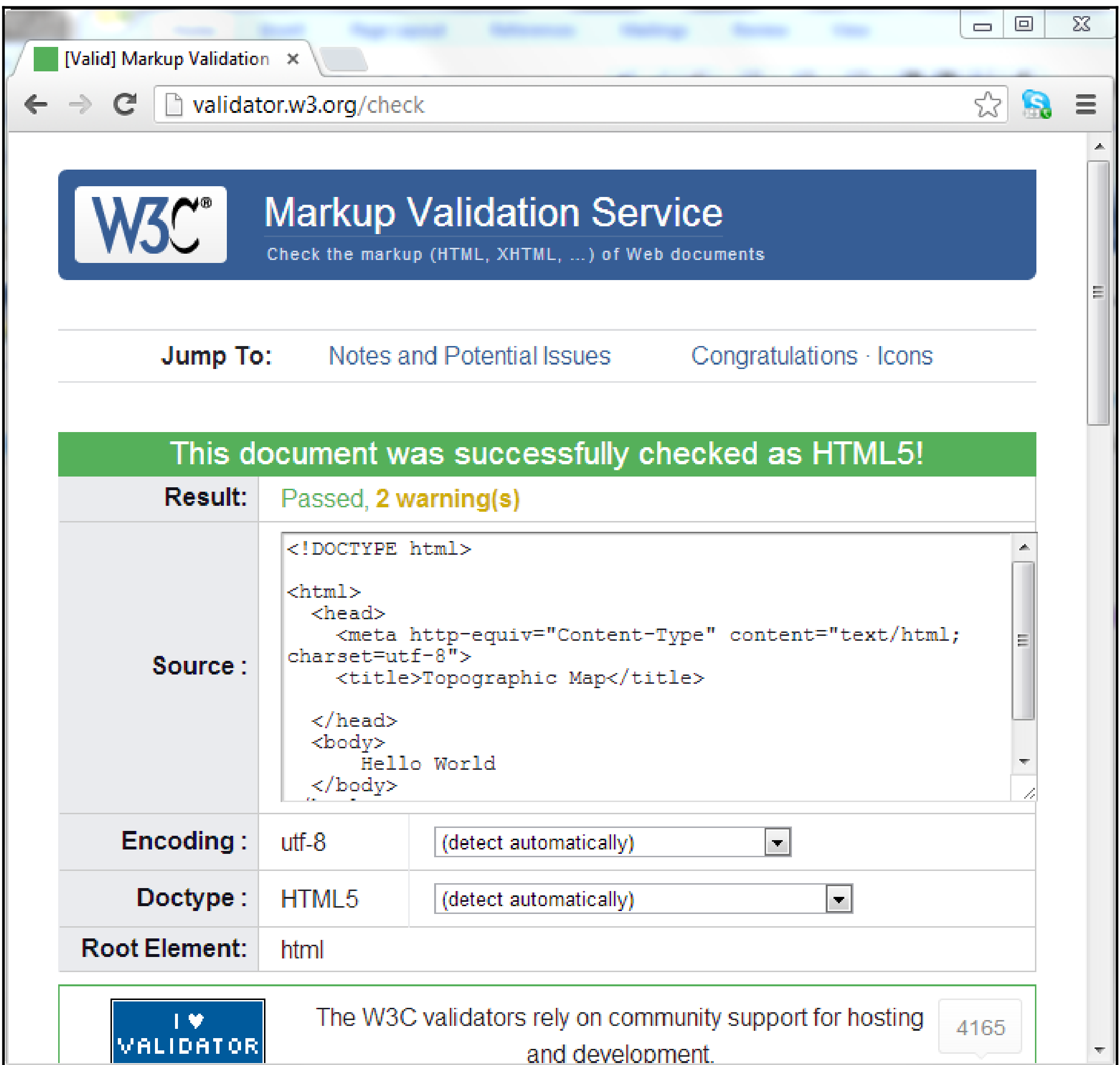


Validating HTML code

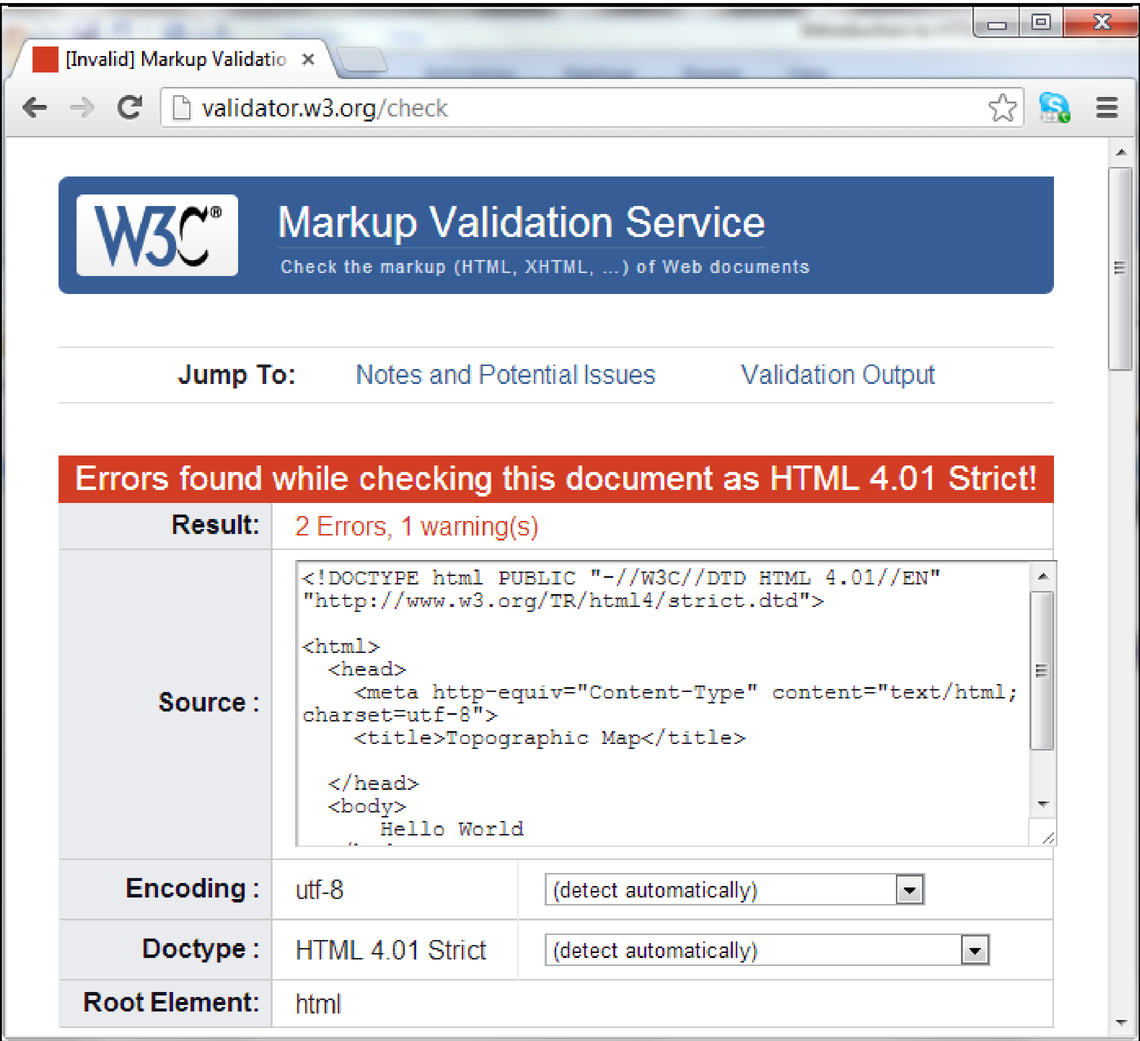
I've mentioned that it is very important that your HTML tags be coded correctly. This is all well and good you say, but how do I know my HTML has been coded correctly? Well, there are a number of HTML code validators that you can use to check your HTML. The W3C HTML validator (`http://validator.w3.org/`) shown in the following screenshot can be used to validate HTML code by **URI**, **File Upload**, or **Direct Input**:



Assuming that your HTML code successfully validates you will get a nice screen with a message indicating a successful validation as seen in the following screenshot:



On the other hand, it will identify any problems with a red error message as shown in the following screenshot. Errors are described in detail which makes it easier to correct problems. Often a single error can lead to many other errors so it is not uncommon to see a long list of error items. Don't panic if this is the case. Fixing one error often resolves many others:



To correct the errors in the preceding document you would need to surround the text **Hello World** with paragraph tags:

```
<p>Hello World</p>
```

JavaScript fundamentals

As the name implies, the ArcGIS API for JavaScript requires that you use the JavaScript language when developing your application. There are some fundamental JavaScript programming concepts that you will need to know before starting to build your application.

JavaScript is a lightweight scripting language that is embedded in all modern web browsers. Although JavaScript can certainly exist outside the web browser environment in other applications, it is best known for its integration with web applications.

All modern web browsers including Chrome, Firefox, Safari, and Edge have JavaScript embedded. The use of JavaScript in web applications enables the creation of dynamic applications that do not require round trips to the server to fetch data, and thus the applications are more responsive and feel like native applications. However, JavaScript can submit requests to the server for more (or more up-to-date) information, and is a core technology in the **AJAX** (**Asynchronous JavaScript and XML**) stack.

One common misconception regarding JavaScript is that it is actually a simplified version of Java. The two languages are actually unrelated: only the names and C-like syntax are similar.

Commenting code

It is best practice to always document your JavaScript code through the use of comments. At a minimum, this should include the author of the code, date of last revision, and the general purpose of the code. In addition, at various points throughout your code you should include comment sections that define the purpose of specific sections of the application. The purpose of this documentation is to make it easier for you or another programmer to quickly get up to speed in the event that the code needs to be updated in some way.

Any comments that you include in your code are not executed. They are simply ignored by the JavaScript interpreter. JavaScript can be commented in a couple of ways including single line and multi-line comments. Single line comments start with // and any additional characters you add to the line. The following code example shows how single line comments are created:

```
//this is a single line comment.  This line will not be executed
```

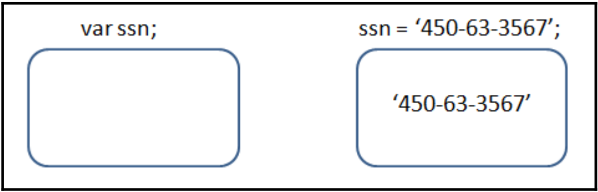
Multi-line comments in JavaScript start with /* and end with */. Any lines in between are treated as comments and are not executed. The following code example shows an example of multi-line comments (also called *block* comments):

```
/*
Copyright 2012 Google Inc. Licensed under the Apache License, Version 2.0
(the "License"); you may not use this file except in compliance with the
License. You may obtain a copy of the License at
http://www.apache.org/licenses/LICENSE-2.0 Unless required by applicable
law or agreed to in writing, software distributed under the License is
distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
KIND, either express or implied. See the License for the specific language
governing permissions and limitations under the License.
*/
```

Variables

Variables are a fundamental concept that you need to understand when working with any programming language. Variables are simply names that we use to associate with a data value. At a lower level, these variables are areas of space carved out in a computer's memory that store data.

You can think of a variable as a box that has a name and that contains some sort of data. When we initially create the variable it is empty until data is assigned. Basically, variables give us the ability to store and manipulate data. In the following figure, we create a variable called **ssn**. Initially this variable is empty but is then assigned a value of **450-63-3567**. The data value assigned to a variable can be of various types including numbers, strings, Booleans, objects, and arrays:



In JavaScript, variables are declared with the **var** keyword. In general, the names that you assign to your variables are completely up to you. However, there are certain rules that you need to adhere to when creating a variable:

- Variable names can contain both text and numbers, but should never start with a number. Always start your variable name with a letter or underscore.
- Spaces are not permitted. If you want your variable name to include multiple words, use *camel case* notation. This convention requires using lower case for the first word, and capitalizing the first letter of subsequent words. For example: `myVariableName`.
- Special characters such as percentage signs and ampersands are not allowed within variable names. Underscores, however, are permitted and can appear anywhere within the variable name.

Other than that, you are free to create variable names as you wish but you should try to assign variable names that describe the data that the variable will be assigned. It is also perfectly legal to declare multiple variables with the same `var` keyword as seen in the following code example:

```
var i, j, k;
```

You can also combine variable declarations with data assignments, as seen in the following examples:

```
var i = 10;
var j = 20;
var k = 30;
```

You may have also noticed that each JavaScript statement ends with a semicolon. The semicolon indicates the end of a statement in JavaScript and should always be included. Missing semicolons are the cause of many JavaScript programming errors.

JavaScript is case sensitive

One very important point that we need to be clear about is that JavaScript is a case-sensitive language. You must be very careful about this because it can introduce some difficult to track down bugs in your code. All variables, keywords, functions, and identifiers must be typed with a consistent capitalization of letters. This gets even more confusing when you consider that HTML is not case sensitive.

This tends to be a stumbling block for new JavaScript developers. I have created three variables, all with the same spelling, but because they do not follow the same capitalization pattern you end up with three different variables as follows:

```
var myName = 'Eric';
var myname = 'John';
var MyName = 'Joe';
```

Variable data types

JavaScript supports various types of data that can be assigned to your variables. Unlike other strongly typed languages such as .NET or C++, JavaScript is a loosely typed language. What this means is that you don't have to specify the *type* of data that will occupy your variable. The JavaScript interpreter does this for you on the fly. You can assign strings of text, numbers, Boolean `true/false` values, arrays, or objects to your variables.

Numbers and strings are pretty straight forward for the most part. Strings are simply text enclosed by either a single or a double quote. For instance:

```
var baseMapLayer = "Terrain";
var operationalLayer = 'Parcels';
```

Numbers are not enclosed inside quote marks and can be integers or floating point numbers:

```
var currentMonth = 12;
var layered = 3;
var speed = 34.35;
```

One thing we would point out to new programmers is that numeric values can be assigned to string variables through the use of single or double quotes that enclose the value. This can be confusing at times for some new programmers. For instance, a value of 3.14 without single or double-quotes is a numeric data type while a value of "3.14" with single or double quotes is assigned a string data type.

Other data types include Booleans that are simply true or false values, and arrays that are a collection of data values. An array basically serves as a container for multiple values. For instance, you could store a list of geographic data layer names within an array and access each element in the array individually as required.

Arrays allow you to store multiple values in a single variable. For example, you might want to store the names of all the layers you want to add to a map. Rather than creating individual variables for each layer you could use an array to store all of them in a single variable. You can then reference individual values from the array using an index number, or by looping through them with a `for` loop. The following code example shows how to create an array in JavaScript and reference its individual members via their index values:

```
var myLayers=new Array();
myLayers[0]="Parcels";
myLayers[1]="Streets";
myLayers[2]="Streams";
```

You could also simplify the creation of this array variable as seen in the following code example where the array has been created as a comma-separated list enclosed in brackets:

```
var myLayers = ["Parcels", "Streets", "Streams"];
```

Bear in mind that if you access array elements via their index, the index numbering is zero-based. This means that the first item in the array occupies position 0 and each successive item in the array is incremented by one:

```
var layerName = myLayers[0]; //returns Parcels
```

Decision support statements

An `if/else` statement in JavaScript and other programming languages is a control statement that allows for decision-making in your code. This type of statement performs a test based on an expression, that you specify at the top of the statement. If the test returns a value of `true` then the statements associated with the `if` block will run. If the test returns a value of `false` then the execution skips to the first `else if` block. This pattern continues until a value of `true` is returned in the test or the execution reaches the `else` statement. The following code example shows how this statement works:

```
var layerName = 'streets';
if (layerName == 'aerial') {
    alert("An aerial map");
}
else if (layerName == "hybrid") {
    alert("A hybrid map");
}
else {
    alert("A street map");
}
```

Looping statements

Looping statements give you the ability to run the same block of code over and over again. There are two fundamental looping mechanisms in JavaScript. The `for` loop executes a code block a specified number of times and the `while` loop executes a code block while a condition is `true`. Once the condition becomes `false`, the looping mechanism stops.

The syntax of a `for` loop is shown as follows. You'll note that it takes a start value for the loop counter that is an integer and a condition expression that, if it evaluates to `false`, will cause the loop to terminate. You should also supply an increment that increases the value of the loop counter on each iteration of the loop:

```
for (start value; condition statement; increment)
{
  the code block to be executed
}
```

In the following example, the start value is set to 0 and is assigned to a variable called `i`. The condition statement is when `i` is less than or equal to 10, and the value of `i` is incremented by 1 for each iteration of the loop using the `++` operator. The code in the body of the loop prints the value of `i` for the current iteration:

```
var i = 0;
for (i = 0; i <= 10; i++) {
    document.write("The number is " + i);
    document.write("<br/>");
}
```

The other basic looping mechanism in JavaScript is the `while` loop. This loop is used when you want to execute a code block while a condition is `true`. Once the condition is set to `false`, execution stops. `while` loops accept a single argument which is the condition expression that will be tested. In the following example, the code block will execute while `i` is less than or equal to 10. Initially `i` is set to a value of 0. At the end of the code block you will notice that `i` is incremented by one (`i = i + 1`):

```
var i = 0;
while (i <= 10)
{
    document.write("The number is " + i);
    document.write("<br/>");
    i = i + 1;
}
```

The `while` loop repeats until `i` reaches the value of 11.

Functions

Now let's cover the very important topic of functions. Functions are simply named blocks of code that execute when called. The vast majority of the code that you write in this course and in your development efforts will occur within functions that you define.

Best practice calls for you to split your code into functions that perform small, discrete tasks. These blocks of code are normally defined in the `<head>` section of a web page inside a `<script>` tag, but can also be defined in the `<body>` section. However, in most cases you will want your functions defined within the `<head>` section so that you can ensure that they are available once the page has loaded.

To create a function you need to use the `function` keyword followed by a function name that you define and any variables necessary for the execution of the function passed in as parameter variables. In the event that you need your function to return a value to the calling code, you will need to use the `return` keyword in conjunction with the data you want passed back.

Functions can also accept parameters which are just variables that are used to pass information into the function. In the following code example, the `multiplyValues()` function is passed two variables: `a` and `b`. This information, in the form of variables, can then be used inside the function which, in this instance, returns the product of `a` and `b`, which is assigned to the variable `x`:

```
var x;
function multiplyValues(a, b) {
   x = a * b;
   return x;
}
```

Objects

Now that we've gone through some basic JavaScript concepts we'll tackle the most important concept in this section. In order to effectively program mapping applications with the ArcGIS API for JavaScript, you need to have a good fundamental understanding of objects, so this is a critical concept that you need to grasp before moving forward.

The ArcGIS API for JavaScript makes extensive use of objects. We'll cover the details of this programming library in detail, but for now we'll focus on the high-level concepts. Objects are complex structures capable of aggregating multiple data values and actions into a single structure. This differs greatly from our primitive data types such as numbers, strings, and Booleans which can hold only a single value.

Objects are composed of both data and actions. Data, in the form of properties, contains information about an object. For example, with a `Map` object found in the ArcGIS Server JavaScript API, there are a number of properties including the map extent, graphics associated with a map, the height and width of the map, layers associated with the map, and so on. These properties contain information about the object.

Objects also have actions which we typically call *methods*, but we can also group constructors and events into this category. Methods are actions that a map can perform such as adding a layer, setting the map extent, or getting the map scale.

Constructors are special purpose functions that are used to create new instances of an object. With some objects it is also possible to pass parameters into the constructor to give more control over the object that is created. The following code example shows how a constructor is used to create a new instance of a `Map` object. You can tell that this method is a constructor because of the use of the `new` keyword which I've highlighted. The `new` keyword followed by the name of the object and any parameters used to control the new object defines the constructor for the object. In this particular case, we've created a new `Map` object and stored it in a variable called `map`. Three parameters are passed into the constructor to control various aspects of the `Map` object including the `basemap`, `center` of the map, and the `zoom` scale level:

```
var map = new Map("mapDiv", {
  basemap: "streets",
  center:[-117.148, 32.706], //long, lat
  zoom: 12
});
```

Events are actions that take place on the object and are triggered by the end user or the application. This would include events such as clicking on the map, panning the map, or a layer being added to the map.

Properties and methods are accessed via dot notation wherein the object instance name is separated from the property or method by a dot. For instance, to access the current map extent you would enter `map.extent` in your code. The following examples demonstrate how to access an object's properties:

```
var theExtent = map.extent;
var graphics = map.graphics;
```

The same is the case with methods except that methods have parentheses at the end of the method name. Data can be passed into a method through the use of parameters. In the following first line of code, we're passing a variable called `pt` into the `map.centerAt(pt)` method:

```
map.centerAt(pt);
map.panRight();
```

Basic CSS principles

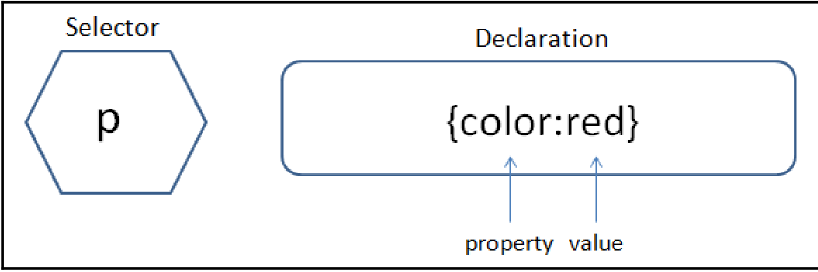
CSS or Cascading Style Sheets is a language used to describe how HTML elements should be displayed on a web page. For instance, CSS is often used to define common styling elements for a page or set of pages such as the font, background color, font size, link colors, and many other things related to the visual design of a web page:

```
<style>
       html, body { height: 100%; width: 100%; margin: 0; padding: 0; }
       #map{
         padding:0;
         border:solid 2px #94C7BA;
         margin:5px;
       }
       #header {
         border: solid 2px #94C7BA;
         padding-top:5px;
         padding-left:10px;
         background-color:white;
         color:#594735;
         font-size:14pt;
         text-align:left; font-weight:bold;
         height:35px;
         margin:5px;
         overflow:hidden;
       }
       .roundedCorners{
         -webkit-border-radius: 4px;
         -moz-border-radius: 4px;
         border-radius: 4px;
       }
       .shadow{

         -webkit-box-shadow: 0px 4px 8px #adadad;
         -moz-box-shadow: 0px 4px 8px #adadad;
         -o-box-shadow: 0px 4px 8px #adadad;      box-shadow: 0px 4px 8px
#adadad;
```

```
        }
</style>
```

CSS syntax

CSS follows certain rules that define what HTML element to select along with how that element should be styled. A CSS rule has two main parts; a selector and one or more declarations. The selector is typically the HTML element that you want to style. In the following example, the selector is `p`. A `<p>` element in HTML represents a paragraph. The second part of a CSS rule is one or more declarations, each of which consists of a property and a value. The property represents the style attribute that you want to change. In our example, we are setting the `color` property to red. In effect, what we have done with this CSS rule is to define that all text within our paragraph should be red `p {color:red}`:



Here is an example of a CSS rule:

```
p {color:red;text-align:center}
```

You can include more than one declaration in a CSS rule as you see in the preceding example. A declaration is always surrounded by curly brackets and each declaration ends with a semicolon. In addition, a colon should be placed between the property and the value. In this particular example, two declarations have been made; one for the color of the paragraph and another for the text alignment of the paragraph. Notice that the declarations are separated by a semicolon.

CSS comments are used to explain your code. You should get into the habit of always commenting your CSS code just as you would with any other programming language. Comments begin with a slash followed by an asterisk and end with an asterisk followed by a slash. In this way, they are identical to block comments in JavaScript. Everything in between those markers is assumed to be a comment and is ignored by the browser:

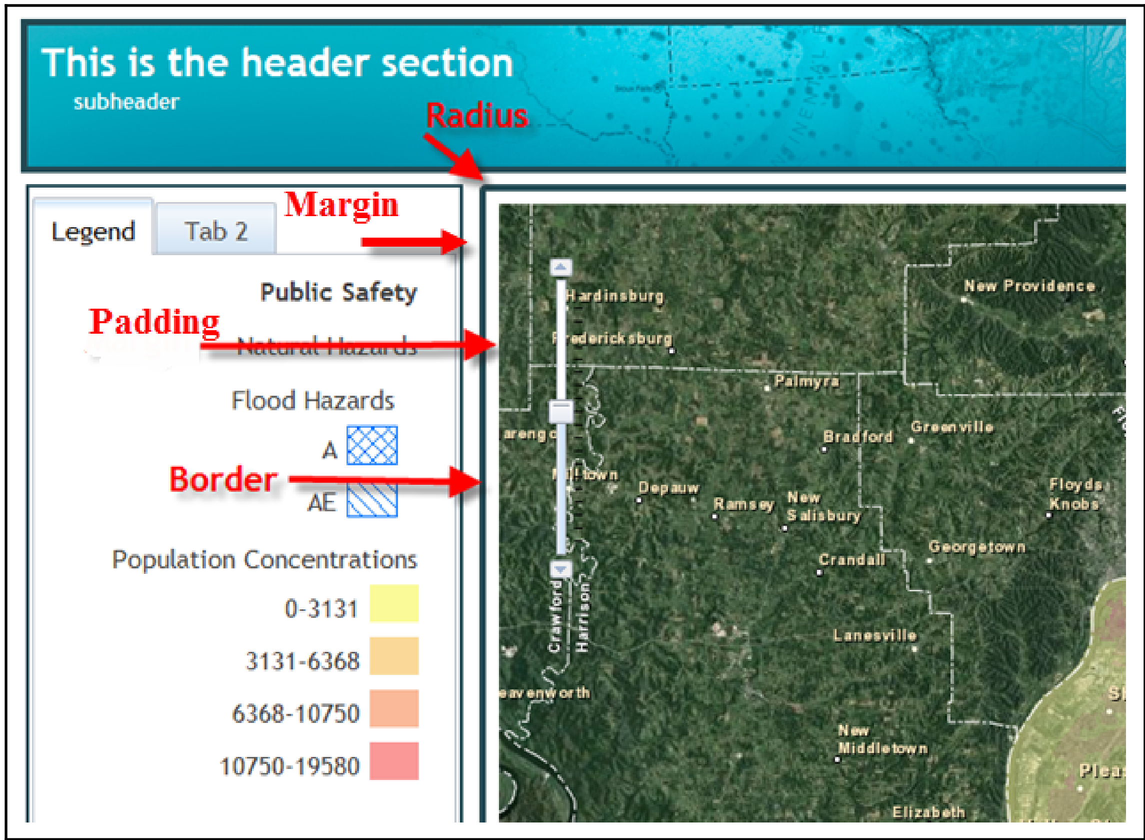
```
/*
h1 {font-size:200%;}
h2 {font-size:140%;}
h3 {font-size:110%;}
*/
```

In addition to specifying selectors for specific HTML elements, you can also use the `id` selector to define styles for any HTML elements with an `id` value that matches the `id` selector. An `id` selector is defined in CSS through the use of the pound sign (`#`) followed by an `id` value.

For example, in the following code example you see three `id` selectors: `rightPane`, `leftPane`, and `map`. In ArcGIS API for JavaScript applications, you almost always have a map. When you define a `<div>` tag that will serve as the container for the map, you specify an `id` and assign it a value which is often the word `map`. In this case, we are using CSS to define several styles for our map including a margin of 5 pixels along with a solid styled border of a specific color and a border radius:

```
#rightPane {
    background-color:white;
    color:#3f3f3f;
    border: solid 2px #224a54;
    width: 20%;
}
#leftPane {
    margin: 5px;
    padding: 2px;
    background-color:white;
    color:#3f3f3f;
    border: solid 2px #224a54;
    width: 20%;
}
#map {
    margin: 5px;
    border: solid 4px #224a54;
    -mox-border-radius: 4px;
}
```

This results in the following layout:



Unlike `id` selectors which are used to assign styles to a single element, `class` selectors are used to specify styles for a group of elements, all of which have the same HTML `class` attribute. A class selector is defined with a period followed by the class name. You may also specify that only specific HTML elements with a particular class should be affected by the style. Examples of both are shown in the code example as follows:

```
.center {text-align:center;}
p.center {text-align:center;}
```

Your HTML code would then reference the `class` selector as follows:

```
<p class="center">This is a paragraph</p>
```

This approach is useful if you want to *group* the styling of several HTML elements which are of different types.

There are three ways to insert CSS into your application: inline, or by using internal or external style sheets.

Inline styling

One way of defining CSS rules for your HTML elements is through the use of inline styles. This method is not recommended because it mixes style with presentation and is difficult to maintain. It is an option though in some cases where you only need to define a very limited set of CSS rules. To use inline styles, simply place the `style` attribute inside the relevant HTML tag:

```
<p style="color:sienna;margin-left:20px">This is a paragraph.</p>
```

Now let's put some emphasis on the *cascading* of cascading style sheets. As you now know, styles can be defined in external style sheets, internal style sheets, or inline. There is a fourth level that we didn't discuss which is the browser default. You don't have any control over that though. In CSS, an inline style has the highest priority, which means that it will override a style defined in an internal style sheet, an external style sheet, or the browser default. If an inline style is not defined then any style rules defined in an internal style sheet would take precedence over styles defined in an external style sheet. The caveat here is that if a link to an external style sheet is placed after the internal style sheet in HTML `<head>`, the external style sheet will override the internal sheet!

Internal style sheet

An internal style sheet moves all the CSS rules into a specific web page. Only HTML elements within that particular page have access to the rules. All CSS rules are defined inside the `<head>` tag and enclosed inside a `<style>` tag, as seen in the code example as follows:

```
<head>
     <style type="text/css">
        hr {color:sienna;}
        p {margin-left:20px;}
        body {background-image:url("images/back40.gif");}
    </style>
</head>
```

External style sheet

An external style sheet is simply a text file containing CSS rules and saved with a file extension of `.css`. This file is then linked into all web pages that want to implement the styles defined within the external style sheet through the use of the HTML `<link>` tag. This is a commonly used method for splitting out the styling from the main web page and gives you the ability to change the look of an entire website through the use of a single external style sheet.

These are the basic concepts that you need to understand with regard to CSS. You can use CSS to define styles for pretty much anything on a web page including backgrounds, text, fonts, links, lists, images, tables, maps, and any other visible objects.

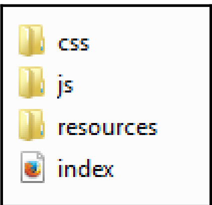
That's a lot to remember! Just keep in mind that style rules defined further down in the hierarchy override style rules defined higher in the hierarchy.

Separating HTML, CSS, and JavaScript

You may be wondering where all of this code is placed. Should you put all your HTML, CSS, and JavaScript code in the same file or split them into separate files? For very simple applications and examples, it is not uncommon for all the code to be placed into a single file with an extension of `.html` or `.htm`. In this case, the CSS and JavaScript code will reside in the `<head>` section of your HTML page.

However, the preferred way of creating an application is to separate the presentation from the content and behavior. The user interface items for your application should reside in an HTML page that contains only tags used to define the content of the application, along with references to any CSS (presentation) or JavaScript (behavior) files that are part of the application. The end result is a single HTML page and one or more CSS and JavaScript files. This would result in a folder structure similar to that shown in the following screenshot where we have a single file called `index.html` and several folders that hold CSS, JavaScript, and other resources such as images.

The **css** and **js** folders will contain one or more files:



CSS files can be linked into an HTML page with the `<link>` tag. In the following you will see a code example that shows you how to use the `<link>` tag to import a CSS file. Links to CSS files should be defined in the `<head>` tag of your HTML page:

```
<!DOCTYPE html>

<html>
  <head>
    <title>GeoRanch Client Portal</title>
    <meta name="viewport" content="initial-scale=1.0, user-scalable=no">
    <link rel="stylesheet" href="bootstrap/css/bootstrap.css">
  </head>
  <body>
  </body>
</html>
```

JavaScript files are imported into your HTML page with the `<script>` tag as seen in the following code example. These `<script>` tags can be placed in the `<head>` tag of your web page, as seen with the reference to the ArcGIS API for JavaScript near the end of the page just before the ending `</body>` tag, as has been done with the `creategeometries.js` file:

```
<!DOCTYPE html>
<html>
    <head>
        <title>GeoRanch Client Portal</title>
        <meta name="viewport" content="initial-scale=1.0, user-
scalable=no">
        <script src="https://js.arcgis.com/3.20/"></script>
    </head>
    <body>
        <script src="js/creategeometries.js"></script>
    </body>
</html>
```

Some developers recommend importing your JavaScript files close to the ending `</body>` tag. This is because, when rendering an HTML page, the browser considers each line of HTML in turn and does not proceed to the next line until the current one has been processed. If you load large external JavaScript files in the `<head>` section, then the user will just see a blank page until the script has fully loaded. Their rationale is that if these files are loaded last, the user will at least see something rendered on the page while they are being downloaded. In the authors' opinion, this is not usually a problem for most users because connection speeds are normally so good that even large external files are downloaded almost instantaneously. Do bear it in mind though.

Putting a `<script>` in the `<head>` section is recommended when using JavaScript libraries like Dojo that must be parsed before they can interact with HTML elements in the `<body>` section. That's why we always reference the ArcGIS API for JavaScript in the `<head>` section.

The Dojo Toolkit is an open source modular JavaScript library, designed to enable the rapid development of web applications. The ArcGIS API for JavaScript is built upon Dojo, so we will be talking about it extensively in this book. Find out more about Dojo here: `https://dojotoolkit.org/`.

Splitting your code into several files allows for a clean separation of your code and it should be much easier to maintain.

Summary

Before we can begin a detailed discussion of the ArcGIS API for JavaScript, you need to have an understanding of some of the fundamental HTML, CSS, and JavaScript concepts. This chapter has provided just that, but you will need to continue learning many additional concepts related to these topics. Right now you know just enough to be dangerous.

The visual appearance of your application is defined through the HTML and CSS code that you develop, while its functionality is controlled using JavaScript. These are very different skill sets and many people are good at one but not necessarily the other. Most application developers will focus on developing the functionality of the application through JavaScript and leave HTML and CSS to the designers! Nevertheless it is important that you have a good understanding of at least the basic concepts of all these topics. In the next chapter, we'll dive into the ArcGIS API for JavaScript and begin learning how to create the `Map` object and add dynamic and tiled map service layers to the map.

2
Creating Maps and Adding Layers

We all have to start somewhere when learning a new programming language or **Application Programming Interface** (**API**). The same applies to creating web mapping applications with the ArcGIS API for JavaScript. Not only do you need to understand some basic JavaScript concepts but you also need to have a grasp of HTML, CSS, and of course the ArcGIS API for JavaScript, which is actually built on top of the Dojo JavaScript framework. That's a lot to put on your plate at once, so in this chapter we'll have you create a very basic application that will serve as a foundation which you can build on in the coming chapters.

Mimicry is an excellent way to learn programming skills, so in this chapter I'm just going to have you type in the code that you see and we'll provide some explanation along the way. We'll save the detailed descriptions of the code for later chapters.

To get your feet wet with the ArcGIS API for JavaScript you're going to create a simple mapping application in this chapter that creates a map, adds a couple of data layers, and provides some basic map navigation capabilities.

Let's get started!

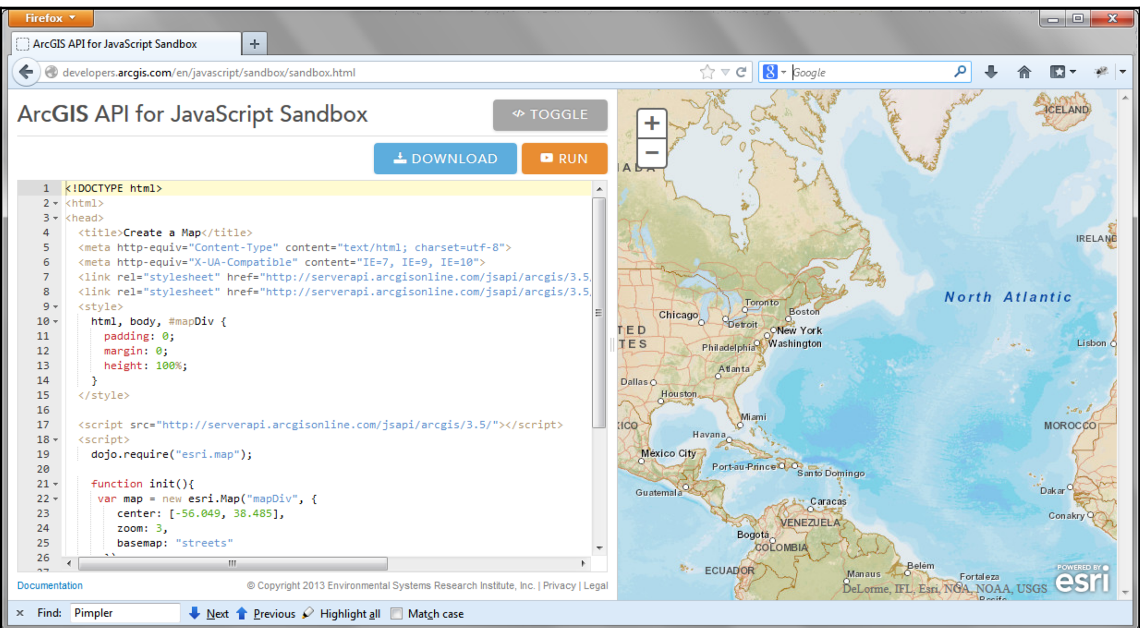
In this chapter we'll cover the following topics:

- The ArcGIS API for JavaScript Sandbox
- Basic steps for creating an application with the ArcGIS API for JavaScript
- More about the map
- Working with map service layers
- Tiled map service layers

- Dynamic map service layers
- Map navigation
- Working with the map extent

The ArcGIS API for JavaScript Sandbox

In this book you're going to use the ArcGIS API for JavaScript Sandbox to write and test your code. The Sandbox can be found at `https://developers.arcgis.com/javascript/3/sandbox/sandbox.html` and will appear as seen in the screenshot when loaded. You'll write your code in the left pane and click the **REFRESH** button to see the results in the right pane:



Basic steps for creating an application with the ArcGIS API for JavaScript

Now that we've got some of the basics of HTML, CSS, and JavaScript out of the way it's time to actually get to work and learn how to build some great GIS web applications! The material in this chapter will introduce you to some of the fundamental concepts that define how you create a map and add information to the map in the form of layers.

There are several steps that you'll need to follow for creating any GIS web application with the ArcGIS API for JavaScript. These steps will always need to be performed if you intend to have a map as part of your application. And I can't imagine that you wouldn't want to do that, given that you're reading this book! In a nutshell,there are several steps you need to follow:

- Create the HTML code for the page
- Reference the ArcGIS API for JavaScript and style sheets
- Load modules
- Make sure the DOM is available
- Create the map
- Define page content
- Style the page
- The entire code

The **HTML Document Object Model** (**DOM**) is a hierarchical representation of all the elements that make up a web page. JavaScript uses the DOM to inspect and change the values of these page elements at runtime.

Creating HTML code for the page

In the previous chapter, you learned the basic concepts of HTML, CSS, and JavaScript. Now you're going to start putting those skills to work. First, you need to create a simple HTML document which will ultimately serve as the container for your map. Since we're using the ArcGIS API for JavaScript Sandbox this step has already been done for you. However, I do want you to spend some time examining the code so that you have a good grasp of the concepts. In the following left pane of the Sandbox the code you see highlighted references the basic HTML code for the web page.

There's obviously some other HTML and JavaScript code in there as well, but the following code forms the basic components of the web page. This includes several basic tags including `<html>`, `<head>`, `<title>`, `<body>` and a few others:

```
<!DOCTYPE html>
  <html>
  <head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
  <meta name="viewport" content="initial-scale=1, maximum-scale=1,user-
scalable=no"/>
  <title>Simple Map</title>
  <!-- The following references the stylesheet for the ArcGIS API for
JavaScript UI components -->
  <link rel="stylesheet"
href="https://js.arcgis.com/3.21/esri/css/esri.css">
  <style>
    html, body, #map {
      height: 100%;
      margin: 0;
      padding: 0;
    }
  </style>
  <!-- The following references the ArcGIS API for JavaScript -->
  <script src="https://js.arcgis.com/3.21/"></script>
  <!-- The following script tag contains your custom application code -->
  <script>
    var map;
    require(["esri/map", "dojo/domReady!"], function(Map) {
      map = new Map("map", {
        basemap: "topo", //For full list of pre-defined basemaps, navigate
to http://arcg.is/1JVo6Wd
        center: [-122.45, 37.75], // longitude, latitude
        zoom: 13
      });
    });
  </script>
</head>
<body>
  <div id="map"></div>
</body>
</html>
```

Referencing the ArcGIS API for JavaScript

To begin working with the ArcGIS API for JavaScript you need to add references to the style sheet and API. In the Sandbox the following lines of code have been added inside the `<head>` tag:

```
<link rel="stylesheet" href="https://js.arcgis.com/3.21/esri/css/esri.css">
...
<script src="https://js.arcgis.com/3.21/"></script>
```

The `<script>` tag loads the ArcGIS API for JavaScript. At the time of this writing the version is currently `3.21`. When new versions of the API are released you'll want to update this number accordingly. The `<link>` tag loads the `esri.css` style sheet which contains styles specific to ESRI widgets and components.

Optionally, you can include a reference to one of the style sheets for a Dojo `dijit` theme. The ArcGIS API for JavaScript is built directly on the Dojo JavaScript framework. Dojo comes with four pre-defined themes that control the look of user interface widgets that are added to your application: `claro`, `tundra`, `soria`, and `nihilo`. In the following code example we are referencing the `claro` theme:

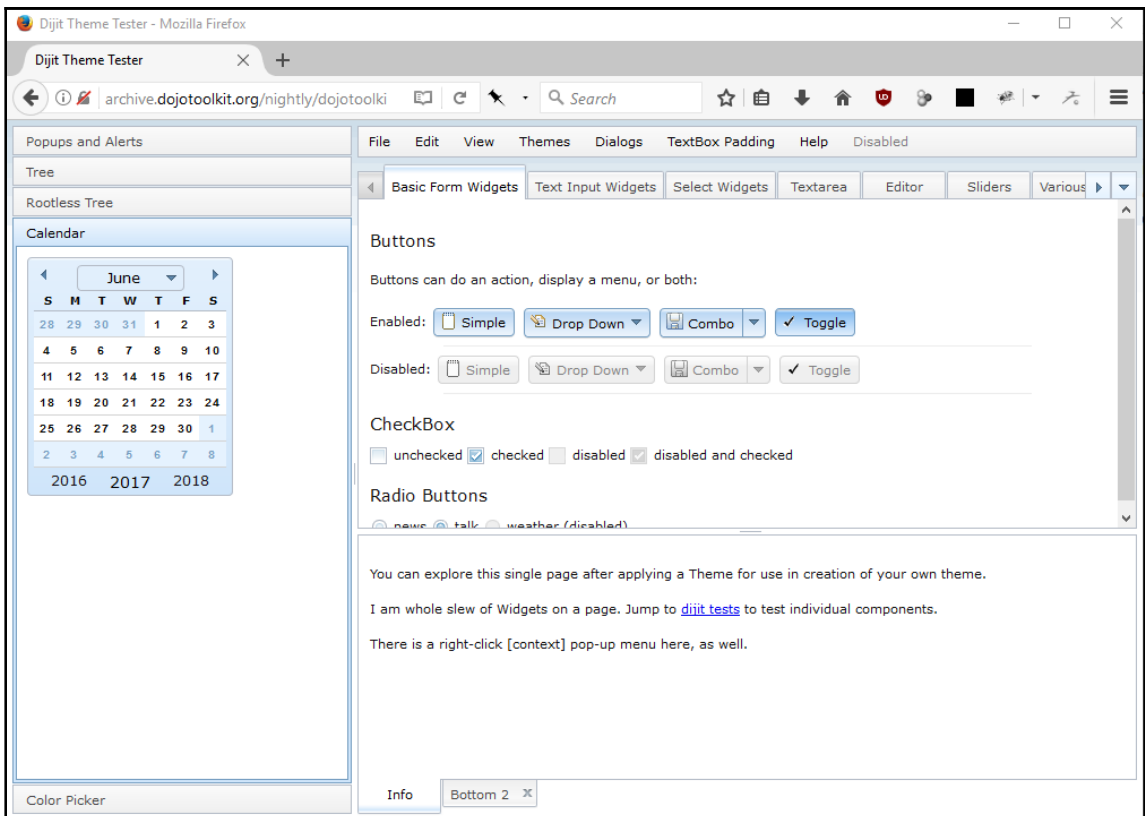
```
<link rel="stylesheet"
href="https://js.arcgis.com/3.21/dijit/themes/claro/claro.css">
```

The other available style sheets can be referenced as seen in the following code example. You don't *have* to use any of these style sheets, but if you intend to add Dojo user interface components (`dijit`) then you'll want to load a corresponding style sheet to control the styling of those components:

```
<link rel="stylesheet"
href="https://js.arcgis.com/3.20/dijit/themes/tundra/tundra.css">
<link rel="stylesheet"
href="https://js.arcgis.com/3.20/dijit/themes/nihilo/nihilo.css">
<link rel="stylesheet"
href="https://js.arcgis.com/3.20/dijit/themes/soria/soria.css">
```

In version 3.16 of the API, ESRI added a new theme called *Calcite*. At the time of writing, it is still in beta and does not support all the Dojo components included in the API. If you want to include it in your applications despite its limitations, check out the documentation here: `https://developers.arcgis.com/javascript/3/jshelp/css.html`.

The Dojo Toolkit provides a theme tester that you can use to get a feel for how each of the themes affect the display of the user interface components. The theme tester is located at `https://archive.dojotoolkit.org/nightly/dojotoolkit/dijit/themes/themeTester.html`:



Loading API modules

Before you can create a `Map` object you must first reference the module that provides the map functionality. This is accomplished through the use of the `require()` function.

Whether to use the older legacy style of Dojo or the new AMD is a source of frustration for many developers. AMD, or Asynchronous Model Definition, was introduced at version 1.7 of Dojo. The 3.4 release of the ArcGIS Server API for JavaScript was the first version to have all modules re-written. For the time being both legacy and AMD style will work just fine, but we advise you to write all your new applications using the AMD style. This is becoming increasingly important as the drop date for Dojo 2.0 gets nearer, because this new version will no longer support the legacy style. We'll follow that convention in this book but keep in mind that applications written prior to version 3.4 of the API still reflect the older style of coding.

Before you begin adding code to the Sandbox, please remove the following code you see listed. To ensure that you remove the correct code, look for the `<script>` tag that contains the call to the `require()` function. You need to remove everything between the opening `<script>` and closing `</script>` tags. You will write the JavaScript code to create the map yourself:

```
<script>
  var map;

  require(["esri/map", "dojo/domReady!"], function(Map) {
    map = new Map("map", {
      basemap: "topo", //For full list of pre-defined basemaps,
      navigate to http://arcg.is/1JVo6Wd
      center: [-122.45, 37.75], // longitude, latitude
      zoom: 13
    });
  });
</script>
```

The `require()` function is used to import resources into your web page. Various resources are provided by the ArcGIS API for JavaScript including the `esri/map` resource, which must be referenced before you can create a map or work with geometry, graphics, and symbols. Once you have a reference to `esri/map` you can use the `Map` constructor to create the map.

The modules you want to import must be contained within a new `<script>` tag. Add the following lines of code to the Sandbox inside the `<head>` tag:

```
<script>
 require([], function() {

 });
</script>
```

The `require()` function accepts an array of module names, followed by a (usually anonymous) `callback` function that will be executed when the resources that the modules represent are loaded and are available to the application.

The `callback` function's parameters are the alias names you will use to refer to the modules in your code. Therefore the first argument to the function will refer to the first module element in the array, the second argument will refer to the second module element in the array, and so on.

The argument names used inside the `require()` function's `callback` can be named anything you'd like. However, both `ESRI` and `Dojo` provide a list of preferred argument names and we recommend adhering to those lists to make it easier for others to understand your code.

The ArcGIS API for JavaScript preferred aliases can be found here: `https://developers.arcgis.com/javascript/3/jsapi/argument_aliases.html` and the Dojo preferred aliases can be found in a Google Spreadsheet here: `http://preview.tinyurl.com/y8ajhy7y`.

For example, in the following code that you add, we provide a reference to the `esri/map` resource, and then inside the anonymous function we provide a preferred argument of `Map`. Each module that you reference in the `require()` function will have an associated argument which will provide your code with access to that module, with one main exception, which we will cover as follows:

```
<script>
    require(["esri/map"], function(Map) {
    });
</script>
```

Making sure the DOM is available

When a web page loads, all the HTML elements that comprise the page are loaded and interpreted. This is known as the **DOM** (**Document Object Model**). Your JavaScript must not attempt to access any of these elements until all the elements have loaded. Obviously if your JavaScript code attempts to access an element that hasn't been loaded it will cause an error. To control this, Dojo has a `ready()` function that you can include inside the `require()` function, which will execute only after all the HTML elements and any modules have loaded. Alternatively, you can use the `dojo/domReady!` plugin to ensure that all the HTML elements have been loaded. We'll use the second method here:

```
<script>
require(["esri/map", "dojo/domReady!"], function(Map) {
    });
</script>
```

Note that, even though we have included the `dojo/domReady!` module in the list of modules we want to load, we have not provided an alias for it. This is because although we need the plugin to inform the application that the DOM is ready, we don't need to refer to it explicitly in our code. This is the exception to the rule we mentioned earlier.

Although it is certainly possible to add JavaScript code directly inside your HTML page, it is better practice to create a separate JavaScript file (`.js`). Most of the code that we write in this book will be done inside an HTML file for simplicity, but as your applications become more complex you'll want to adhere to the practice of writing your JavaScript code in a separate file or files.

Creating the map

The creation of a new map is done through `esri/Map`, which is a reference to the `Map` class found in the `esri/Map` module you imported in a previous step. Inside the `require()` function you're going to create new `Map` object using a constructor function. This constructor for the `Map` object accepts two parameters, including a reference to the `<div>` tag where the map will be placed on the web page as well as an options parameter that can be used to define various map setup options. The options parameter is defined as a **JSON** (**JavaScript Object Notation**) object that contains a set of key/value pairs.

Perhaps the most visible option is `basemap`, which allows you to select a pre-defined `basemap` from `ArcGIS.com`. Pre-defined `basemaps` include `streets`, `satellite`, `hybrid`, `topo`, `gray`, `oceans`, `national-geographic`, and `osm`.

A full list of pre-defined basemaps can be found here: `http://arcg.is/1JVo6Wd`.

The `zoom` option is used to define a starting zoom level for the map and is an integer value that corresponds to a pre-defined zoom scale level. The `minZoom` and `maxZoom` options define the largest and smallest scale zoom levels for the map, respectively. The `center` option specifies the center point of the map that will be displayed and takes a `Point` object containing a latitude/longitude coordinate pair. There are a number of additional options that you can pass in as parameters to the constructor for the `Map` object, to alter the behavior and appearance of the map.

First we'll create a global variable called `map` by adding the following highlighted line of code:

```
<script>
  var map;
  require(["esri/map", "dojo/domReady!"], function(Map) {
  });
</script>
```

Then, add the following highlighted code block to the `require()` function. This line of code is the constructor for the new `Map` object. The first parameter passed into the constructor is a reference to the id of the `<div>` tag in the DOM where the map will appear. We haven't defined this `<div>` tag yet, but we'll do so in the next step. The second parameter passed into the `Map` constructor is a JSON object that defines options including the geographic coordinate that will serve as the center of the map, the zoom level, and the `basemap: "topo"`:

```
 require(["esri/map", "dojo/domReady!"], function(Map) {
  map = new Map("map", {
    basemap: "topo",
    center: [-122.19, 37.94], // longitude, latitude
    zoom: 6
  });
});
```

Creating the page content

You need to create the HTML `<div>` tag that will serve as the container for the map. You always want to assign a unique id to the `<div>` tag so that your JavaScript code can reference that element in the page. If you're following along in the Sandbox, you'll see that this has already been done for you:

```
<body>
  <div id="map"></div>
</body>
```

In addition, if you specified a Dojo style sheet you will also want to define the `class` attribute for the `<body>` tag to reference it. This is not strictly necessary in this example because we are not going to be using any styleable `dijit`, but it can't hurt, so change the `<body>` tag to read as follows:

```
<body class="claro">
   <div id="map"></div>
</body>
```

Styling the page

You can add CSS styling information to the `<head>` tag that will modify the appearance of page elements. The Sandbox code includes the following styling, which makes the map occupy the entire HTML page:

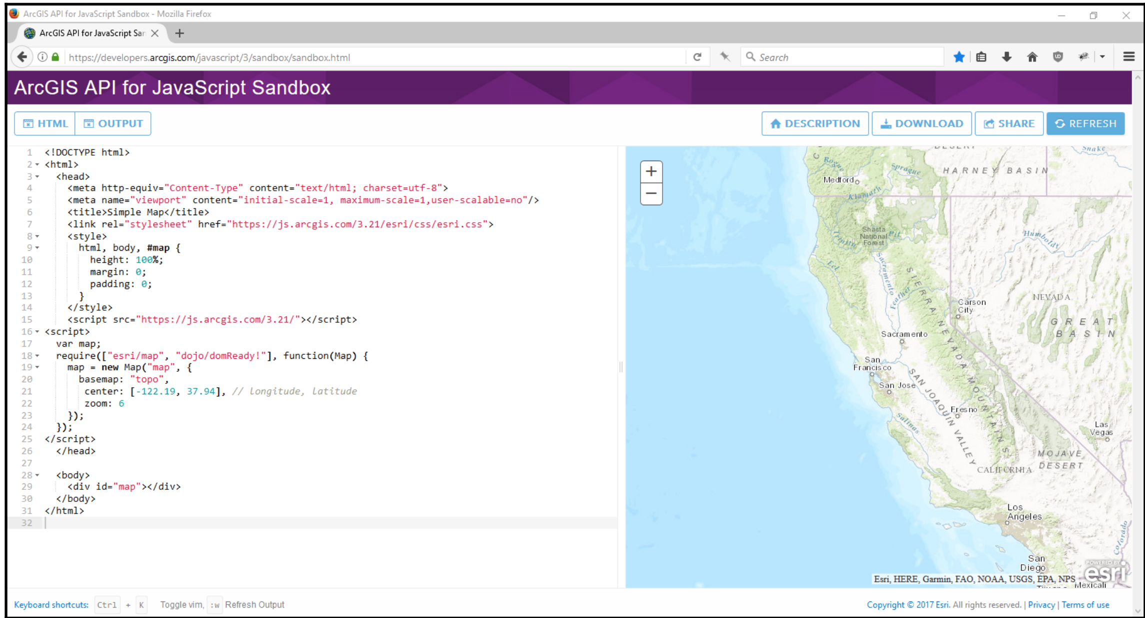
```
<style>
html, body, #map {
  height: 100%;
  margin: 0;
  padding: 0;
}
</style>
```

The entire code

The code for this simple application should appear as follows:

```
<!DOCTYPE html>
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
  <meta name="viewport" content="initial-scale=1, maximum-scale=1,user-
scalable=no" />
  <title>Simple Map</title>
  <link rel="stylesheet"
href="https://js.arcgis.com/3.21/esri/css/esri.css">
  <style>
    html,
    body,
    #map {
      height: 100%;
      margin: 0;
      padding: 0;
    }
  </style>
  <script src="https://js.arcgis.com/3.21/"></script>
  <script>
    var map;
    require(["esri/map", "dojo/domReady!"], function(Map) {
      map = new Map("map", {
        basemap: "topo",
        center: [-122.19, 37.94], // longitude, latitude
        zoom: 6
      });
    });
  </script>
</head>
<body>
  <div id="map"></div>
</body>
</html>
```

Execute the code by clicking the **Refresh** button and you should see the following map if everything has been coded correctly. If your application does not work as expected, check your code against the contents of the `basic_map.html` file in the `Chapter2` folder of the sample code:



More about the map

You'll need to follow the preceding process we described for every application that you build with the ArcGIS API for JavaScript.

The creation of the map can only happen when your HTML has finished loading, and all the required DOM elements and scripts are ready. You achieved this by using the `domReady!` plugin in your `require()` function. This ensured that the code in the function that was passed as the second argument only executes when everything is in place. This function is known as an *initialization function*:

```
require(["esri/map", "dojo/domReady!"], function(Map) {
  // initialization goes here
});
```

You use your initialization function to create your map, add layers, and perform any other setup routines necessary to start your application.

Creating a map is invariably one of the first things that you'll do and in this section we'll take a closer look at the various options you have when creating an instance of the `Map` class.

In object-oriented programming you create an instance (your own, working copy) of a class (like `Map`), by calling a special method on the class called the *constructor*. Constructors frequently take one or more parameters that can be used to set the initial state of an object.

The `Map` constructor accepts two parameters: the HTML page element (a `<div>`) where the map should be placed, and an options object that sets various properties on the newly-created map instance to define its behavior. This includes things like which basemap to use, the initial extent of the map, the display of navigation controls, how graphics appear during panning, what type of zoom slider control to use, and many more.

The second parameter in the constructor is always enclosed within braces. This tells the JavaScript interpreter that you are supplying a JavaScript object.

The JavaScript object syntax is known as JSON, and has many uses outside of JavaScript too.

Inside the braces, the map options (the object's properties) are specified as key/value pairs. If you are supplying more than one option, you must separate them with commas. In the following example, we are defining options for the map coordinates that will serve as the center of the map, along with a zoom level, and a basemap layer of `streets`:

```
var map = new Map("mapDiv", {
  center: [-56.049, 38.485],
  zoom: 3,
  basemap: "streets"
});
```

Working with map service layers

A map without data layers is like an artist with a blank canvas. The data layers that you add to your map give it meaning and set the stage for analysis. These data layers come from map services published by ArcGIS Server and, occasionally, other GIS servers. The two main types of layers available from ArcGIS Server are dynamic map service layers and tiled (or cached) map service layers.

Dynamic map service layers reference map services that create a map image on the fly and then return the image to the application. This type of map service may be composed of one or more layers of information. For example, the Demographics map service displayed in the following screenshot is composed of nine different layers representing demographic information at various levels of geography:

Demographics (MapServer)

View In: ArcMap ArcGIS Explorer ArcGIS JavaScript Google Earth

View Footprint In: Google Earth

Service Description:

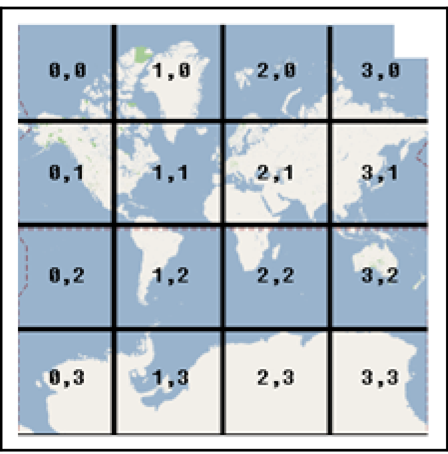
Map Name: Layers

Layers:

- Demographics/ESRI_Census_USA (0)
    - Census Block Points (1)
    - Census Block Group (2)
    - Counties (3)
        - Coarse Counties (4)
        - Detailed Counties (5)
    - States (6)
- ESRI_StreetMap_World_2D (7)
    - World Street Map (8)

While they can take somewhat longer to display in a client application since they must be generated "on the fly", dynamic map service layers are more versatile than tiled map service layers in that you can filter the features displayed by using layer definitions, set the visibility of various layers within the service, and define temporal information for the layer. For example, in the Demographics map service layer mentioned above, you might elect to only display Census Block Groups in your application. This is the type of versatility provided by dynamic map service layers that you don't get with tiled map service layers.

Tiled map service layers reference a predefined cache of map tiles instead of dynamically rendered images. The easiest way to understand the concept of tiled map services is to think about a grid that has been draped across the surface of a map. Each cell within the grid is the same size and will be used to cut the map into individual image files called tiles. The individual tiles are stored as image files on a server and retrieved as necessary depending on the map extent and scale. This same process is often repeated at various map scales. The end result is a cache of tilesets that have been generated for various map scales. When the map is displayed in the application it will appear to be seamless even though it is composed of many individual tiles:

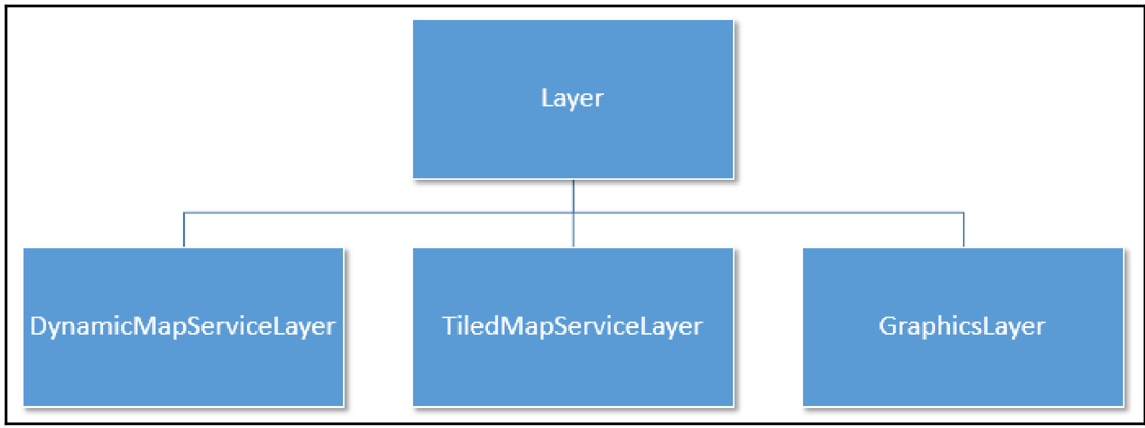


These tiled, or cached, map layers are often used as base maps such as imagery, street maps, and topographic maps, or for data layers that don't change often. Tiled map services tend to display faster since the server does not have to create images on the fly each time there is a request for a map.

Operational layers are then draped on top of the tiled base maps and these are usually dynamic layers. While they can be somewhat slower in terms of performance, dynamic map service layers have the advantage of being always up to date and your application is able to change their contents and appearance before they are rendered in the browser.

Using the layer classes

You can use the various layer classes in the API for JavaScript to reference map services hosted by ArcGIS Server and other map servers. All layer classes inherit from the `Layer` base class. The `Layer` class has no constructor so you can't create an actual object instance from this class. This base class simply defines properties, methods, and events that must exist in all classes that inherit from it:

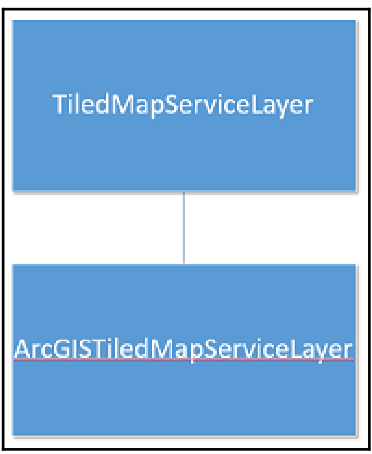


As indicated in the preceding figure, `DynamicMapServiceLayer`, `TiledMapServiceLayer`, and `GraphicsLayer` all inherit directly from the `Layer` class. (Note that there are several other layers that inherit from `Layer`, but many of those are for special use cases, so our diagram shows only these more commonly-used classes.)

`DynamicMapServiceLayer` and `TiledMapserviceLayer` also act as base classes. `DynamicMapServiceLayer` is the base class for dynamic map services while `TiledMapServiceLayer` is the base class for tiled map services. `Layer`, `DynamicMapServiceLayer`, and `TiledMapServiceLayer` are all base classes meaning that you can't specifically create an object from these classes in your application.

Tiled map service layers

As I mentioned, tiled map service layers reference a cache of predefined images that are tiled together to create a seamless map display. Because these tiles have been generated in advance, ArcGIS Server doesn't need to create them on the fly and this usually means performance is excellent. For this reason, they are often used as base maps which are cartographically rich and therefore expensive to render:



The `ArcGISTiledMapServiceLayer` class is used when referencing a tiled (cached) map service exposed by ArcGIS Server. This is one of several tiled map service layer types that inherit from the `TiledMapServiceLayer` class, but it's the only one shown in our diagram as it is by far the most commonly-used, and the only one we'll be talking about in this book.

Since this type of object works against a pre-generated cache of map tiles, performance is often excellent. The constructor for the `ArcGISTiledMapServiceLayer` requires an URL to the map service endpoint, along with an optional options object that allow you to assign an ID to the map service and control transparency and visibility. The following code example creates an option-less `ArcGISTiledMapServiceLayer` and then calls `Map.addLayer()` to add the layer to the map:

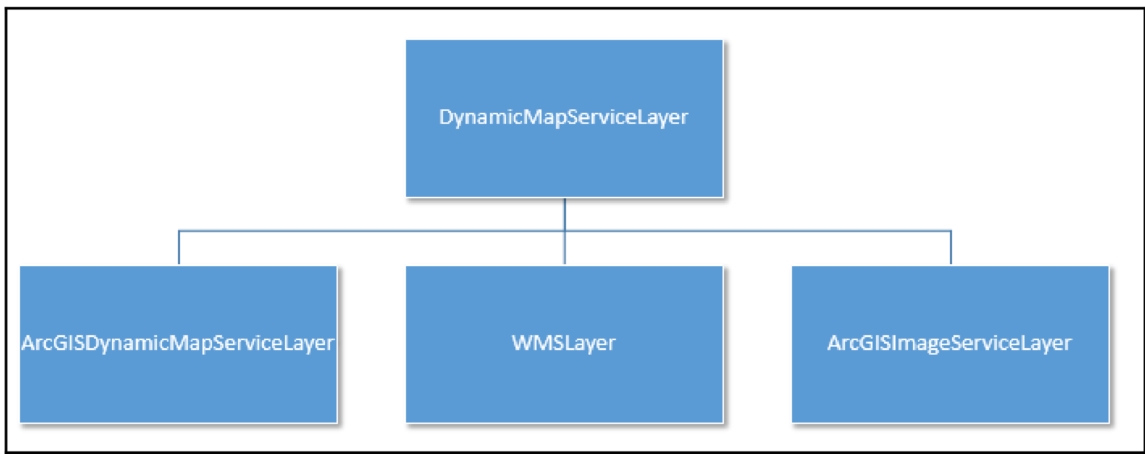
```
var basemap = new
ArcGISTiledMapServiceLayer("http://server.arcgisonline.com/ArcGIS/rest/serv
ices/World_Topo_Map/MapServer");
   map.addLayer(basemap);
```

Therefore, ArcGISTiledMapServiceLayer is used primarily for the fast display of cached map data: either basemaps or complex cartography that doesn't change very often. (If the data changes, you must recreate the cache). You can also control the levels at which the data will be displayed. For instance, you may want to display data from a more generalized tiled map service showing interstates and highways while your users are zoomed out at levels 0-6, and then switch to a more detailed view once the user zooms in further. You can also control the transparency of each layer added to the map.

Dynamic map service layers

As the name suggests, the ArcGISDynamicMapServiceLayer class is used to create dynamic maps served by ArcGIS Server, where all the of map imagery is rendered on the fly.

Just like the ArcGISTiledMapServiceLayer, the ArcGISDynamicMapServiceLayer constructor takes a URL endpoint for the map service along with optional parameters used to assign an ID to the service, specify the transparency of the map image, or set the initial visibility of the layer on or off:



The class name `ArcGISDynamicMapServiceLayer` can be somewhat misleading. Although it appears to reference an individual data layer this is in fact not the case. It refers to a map *service* rather than a data layer. Individual layers inside the map service can be turned on/off through the class `setVisibleLayers()` method.

The code for creating an instance of `ArcGISDynamicMapServiceLayer` looks very similar to the code for working with `ArcGISTiledMapServiceLayer`, as can be seen as follows.

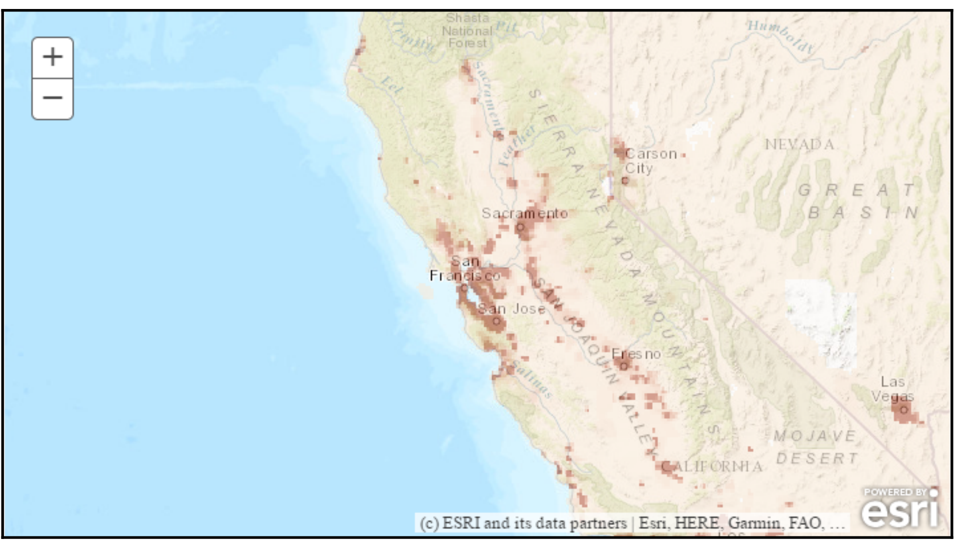
The constructor accepts a URL for the map service endpoint. The second parameter accepts an options object that you can supply to control transparency, visibility, and image parameters:

```
var operationalLayer = new
ArcGISDynamicMapServiceLayer("https://sampleserver1.arcgisonline.com/ArcGIS
/rest/services/Demographics/ESRI_Population_World/MapServer",{"opacity":0.5
});
map.addLayer(operationalLayer);
```

Add the preceding two lines of code to the ArcGIS API for JavaScript Sandbox, under the line of code that creates the map. Also ensure that you reference `esri/layers/ArcGISDynamicMapServiceLayer` in the `require()` function:

```
require(["esri/map", "esri/layers/ArcGISDynamicMapServiceLayer",
"dojo/domReady!"],
   function(Map, ArcGISDynamicMapServiceLayer) {
      map = new Map("map", {
         basemap: "topo",
         center: [-122.19, 37.94], // longitude, latitude
         zoom: 6
      });
      var operationalLayer = new
ArcGISDynamicMapServiceLayer("https://sampleserver1.arcgisonline.com
ArcGIS/rest/services/Demographics/ESRI_Population_World/MapServer",{"opacit
y":0.5});
      map.addLayer(operationalLayer);
});
```

Click the **Refresh** button in the Sandbox to see the dynamic layer added to the map as shown in the following screenshot:



With an instance of `ArcGISDynamicMapServiceLayer` you can perform a number of operations. Obviously you can create maps that display the data in the service, but you can also query data from layers in the service, control feature display through layer definitions, control individual layer visibility, display temporal information, export maps as images, control background transparency, and more.

Adding layers to the map

The `Map.addLayer()` method takes an instance of a layer (`ArcGISDynamicMapServiceLayer` or `ArcGISTiledMapServiceLayer` in our examples) as the first parameter, and an optional index that specifies where it should be placed. If you don't specify an index, the layer will be placed on top of all the other layers in the map, which is often what you want so that you can be sure it is visible.

In the following code example we create a new instance of `ArcGISDynamicMapServiceLayer`. We then call `Map.addLayer()`, passing in the new instance of the layer. The layers in the service will now be visible on the map:

```
var operationalLayer = new
ArcGISDynamicMapServiceLayer("http://sampleserver1.arcgisonline.com/ArcGIS/
rest/services/Demographics/ESRI_Population_World/MapServer");
map.addLayer(operationalLayer);
```

If you want to add more than one layer to the map in one go, you can use `Map.addLayers()`, which accepts an array of layer objects.

As well as being able to add layers to a map you can also remove them by using `Map.removeLayer(`*`layer`*`)` or `Map.removeAllLayers()`.

Setting the visible layers from a map service

You can control the visibility of individual layers within a dynamic map service layer using the `setVisibleLayers()` method. This only applies to dynamic map service layers, not tiled map service layers. This method takes an array of integers corresponding to the data layers in the map service. This array is zero-based so the first layer in the map service occupies position 0. In the **Demographics** map service illustrated in the following screenshot, **Demographics/ESRI_Census_USA (0)** occupies index 0:

Therefore, in the event that we'd like to display only the Census Block Points and Census Block Groups from this service we could use `setVisibleLayers()` as seen in the code example as follows:

```
var dynamicMapServiceLayer = new
ArcGISDynamicMapServiceLayer("http://sampleserver1.arcgisonline.com
   /ArcGIS/rest/services/Demographics/ESRI_Census_USA/MapServer");
dynamicMapServiceLayer.setVisibleLayers([0,1]);
map.addLayer(dynamicMapServiceLayer);
```

Setting a definition expression

In ArcGIS Desktop you can use a definition expression to specify a subset of all the features in a data layer that you want to display. A definition expression is simply a SQL query against the layer's data. Only the features whose attributes meet the query are displayed. For example, if you only wanted to display cities with a population greater than 1 million the expression would be something like `POPULATION > 1000000`. The ArcGIS API for JavaScript has a `setLayerDefinitions()` method that accepts an array of definitions that can be applied against `ArcGISDynamicMapServiceLayer` to control the display of features in the resulting map. The following code example shows how this is done:

```
var layerDefinitions = [];
layerDefinitions[0] = "POPULATION > 5000000";
layerDefinitions[5] = "AREA > 100000";
dynamicMapServiceLayer.setLayerDefinitions(layerDefinitions);
```

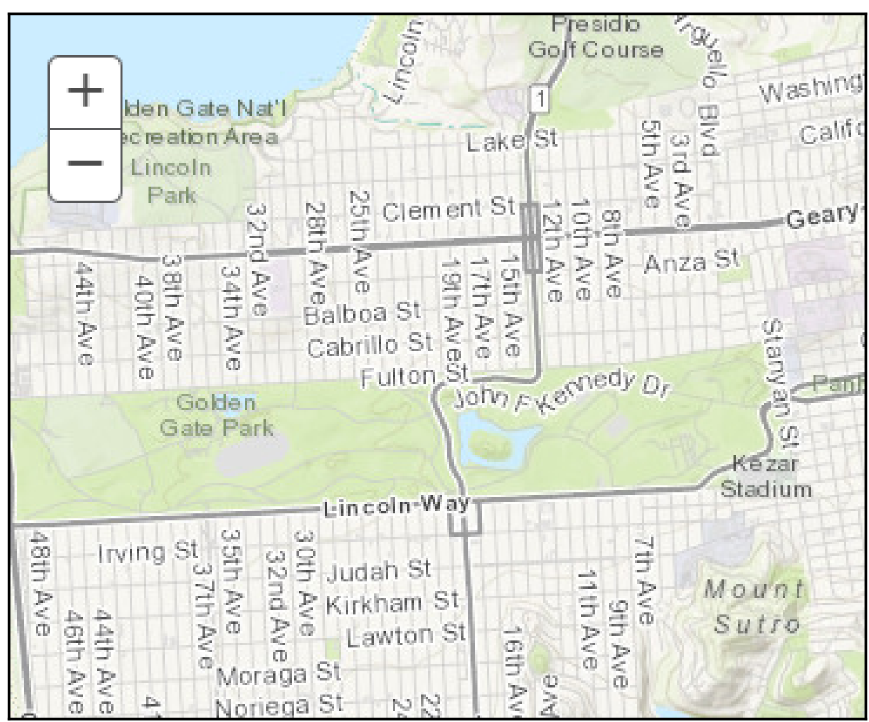

First, create an array that will hold multiple `WHERE` clauses which will serve as the definition expressions for each layer. In this case we are defining layer definitions for the first (index position 0) and sixth (index position 5) layers. Remember that the array is zero-based, so the first array is at index position 0. The `WHERE` clauses are stored in the array and then passed into the `setLayerDefinitions()` method. ArcGIS Server then renders the features that match the definition expressions for each layer.

Map navigation

Now that you know a little about maps and the layers that reside within those maps it's time to learn how to control map navigation in your application. In most cases your users will need to be able to navigate around the map through panning and zooming. The ArcGIS API for JavaScript provides a number of user interface widgets and toolbars that you can use to allow your user to change the current map extent. Map navigation can be controlled by either the mouse, the keyboard, or a combination of the two. Additionally, you can control map navigation programmatically.

Map navigation widgets and toolbars

The simplest way to provide map navigation control to your application is through the addition of various widgets and toolbars. By default, when you create a new map and add layers, a zoom slider is included with the map. This slider allows the user to zoom in and out of the map by clicking the plus or minus buttons respectively. The zoom slider is illustrated in the following screenshot. You don't have to write any code for the zoom slider to appear on your map. It is present by default:

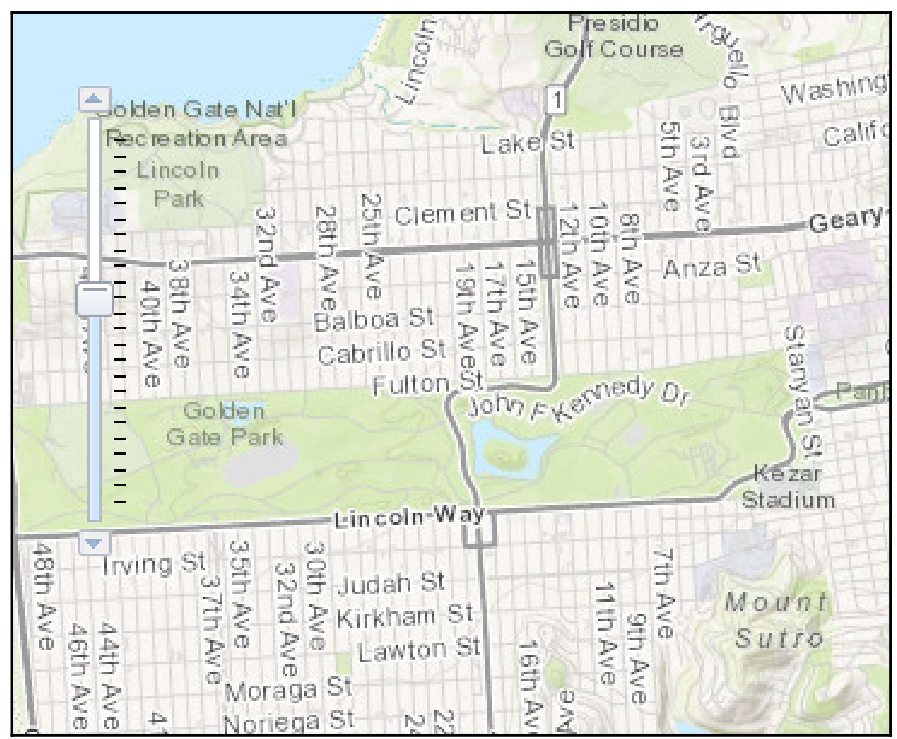


However, you can remove the slider if your application doesn't need it by setting the `slider` option to `false` within the map options object you pass into the `Map` constructor:

```
{ slider: false }
```

By default, the ArcGIS API for JavaScript displays the *small* version of the slider. If you want to give your users more fine-grained control over the map's zoom level, you can use the *large* slider instead:

```
{ sliderStyle: "large" }
```



You can also add pan buttons that will pan the map in the direction that the arrow points when clicked. By default pan buttons will not appear on the map. You must specifically set the `nav` option to `true` when creating your `Map` object:

```
{ nav: true }
```

The result of this is the addition of little directional buttons in the corners of the map:

The ArcGIS API for JavaScript also gives you the ability to add several types of toolbars to your application, including a navigation toolbar that contains buttons for zooming in and out, panning, full extent, next extent and previous extent. The topic of toolbar creation is covered in detail in a later chapter, so we'll save that discussion for then:

Map Navigation with the mouse and keyboard

Users can also control map navigation with the mouse and/or keyboard devices. By default, users can do the following:

- Drag the mouse to pan
- Mouse scroll forward to zoom in
- Mouse scroll backward to zoom out
- *SHIFT* + *Drag the mouse to zoom in* + drag the mouse to zoom in
- *Shift* + *Ctrl* + drag the mouse to zoom out
- *Shift* + click to recenter
- Double Click to zenter and zoom in
- *Shift* + double-click to center and zoom out
- Use arrow keys to pan
- Use + key to zoom in a level
- Use - key to zoom out a level

These options can be disabled using one of several `Map` methods. For example, to disable scroll wheel zooming you would use the `Map.disableScrollWheelZoom()` method. These navigation features can also be removed after the map has been loaded. This can be very useful. For example, let's say that you have a custom tool that requires the user to double click on the map to select a feature. You'll want to call `Map.disableDoubleClickZoom()` while your tool is being used, and `Map.enableDoubleClickZoom()` when your user has finished with it.

Getting and setting the map extent

One of the first things you'll want to master is getting and setting the map extent. By default the initial extent of a map within your application is the extent of the map when it was last saved in the map document file (`.mxd`) used to create the map service. In some cases this may be exactly what you want, but in the event that you need to set a map extent other than the default you have several options.

One of the optional parameters that can be defined in the constructor for the `Map` object is the `center` parameter. You can use this in conjunction with the `zoom` object to set the initial map extent. You'll see this illustrated in the following code example where we define a coordinate pair for the center of the map along with a zoom level of 3:

```
var map = new Map("mapDiv", {
        center: [-56.049, 38.485],
        zoom: 3,
        basemap: "streets"
      });
```

The initial extent of the map is not a required parameter, and thus if you leave out this information the map will simply use the default extent. This is shown in the following code example where only the ID of the container is specified in the map's constructor, and no map options object is supplied:

```
var map = new Map("map");
```

After a `Map` object has been created you can also use the `Map.setExtent()` method to change the extent by passing in an `Extent` object as seen in the code example as follows:

```
var extent = new Extent(-95.271, 38.933, -95.228, 38.976);
map.setExtent(extent);
```

Alternatively, you could set the `Extent` properties individually as seen in the code example as follows:

```
var extent = new Extent();
extent.xmin = -95.271;
extent.ymin = 38.933;
extent.xmax = -95.228;
extent.ymax = 38.976;
map.setExtent(extent);
```

There may be times when you are using multiple map services in your application. In this case, setting the initial map extent can be done either through the constructor for your map or by using the `Map.fullExtent()` method on one of the services. For example, it is common to use a map service that provides base layer capabilities containing aerial imagery along with a map service containing your own local operational data. The following code example uses the `fullExtent()` method to set the map's extent to the extent of the `myService2` layer:

```
map = new Map("mapDiv", {extent: myService2.fullExtent) });
```

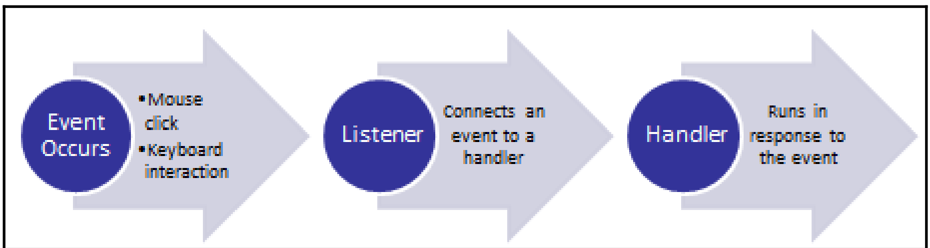
The current extent can be obtained either through the `Map.extent` property or the `onExtentChange` event. We'll talk about map events in a bit.

Note that the `Map.extent` property is read-only, so don't attempt to set the map extent directly through this property. Always use the `Map.setExtent()` accessor method.

Map events

In programming, events are actions that take place within an application. Normally these events are triggered by the end user and can include things like mouse clicks, mouse drags, keyboard actions, and others, but can also include the sending and receiving of data, changes in state of a user interface component, and many other scenarios.

The ArcGIS API for JavaScript is an asynchronous API that follows a publish/subscribe pattern wherein an application registers (publishes) events with listeners (subscribers). The following diagram illustrates this process. Listeners are responsible for monitoring the application for these events and then triggering a handler function that responds to the event. Multiple events can be registered to the same listener. The dojo `on()` function creates a listener that associates an event to a handler function:



As you'll recall the ArcGIS Server JavaScript API is built on top of Dojo. With Dojo, events are registered to handlers through the dojo `on()` method. This method takes three parameters. Take a look at the following code example to get a better understanding of how events are registered:

```
require(["esri/Map", "dojo/on"], function(Map, on) {
  // ...
  on(myMap, "click", displayCoordinates);
});
```

object   event   Javascript handler

In the preceding example, the first two parameters to `on()` specify the object and the type of event on that object that we want to listen to. In this case that means we are registering the `click` event found on the `Map` object. This event is fired every time the user clicks the mouse within the confines of the map. The final parameter, `displayCoordinates`, specifies the handler function that executes when the event is raised. The contents of this function is not shown, and it is something we have to write ourselves to respond to the map being clicked.

Therefore, each time the `click` event on the `Map` object is fired, it will trigger the `displayCoordinates` function which, once we've written it, will execute and report the current extent of the map. Although the events and the handlers they are registered to will change depending upon your circumstance, the method of registration is the same.

Each time an event occurs, an `Event` object is generated. This `Event` object contains additional event information such as the mouse button that was clicked or perhaps the key on the keyboard that was pressed. This object is automatically passed into the event handler where it can be examined. Notice in the following code example that the `Event` object is passed into the handler as a parameter. This is a dynamic object whose properties will change depending upon the type of event that was triggered.

```
function addPoint(evt) {
    alert(evt.mapPoint.x, evt.mapPoint.y);
}
```

There are many different events that are available on a number of different objects in the API. However, it is important to keep in mind that you do not have to register every event with a listener. Only those events that are necessary for your application should be registered. When an event occurs that hasn't been registered with a listener the event is simply ignored.

The `Map` object contains many different events that you can respond to including various mouse events, extent change events, basemap change events, keyboard events, layer events, pan and zoom events, and more. Your application can respond to any of these events. In the coming chapters we'll examine events that are available on other objects.

It is a good programming practice to always disconnect your listeners when they are no longer needed. Simply call the `remove()` method on the listener as shown as follows:

```
var mapClickEvent = on(myMap, "click", displayCoordinates);
mapClickEvent.remove();
```

Summary

We covered a lot of ground in this chapter. All applications created with the ArcGIS API for JavaScript require you to perform certain steps. These include defining references to the API and style sheet, loading modules, creating an initialization function, and so on. In the initialization function you will most likely create an instance of the `Map` class, add various layers, and perform other setup operations that need to be performed before the application is used. In this chapter you learned how to perform these tasks.

In addition, we examined the various types of layers that can be added to a map including tiled map service layers and dynamic map service layers. Tiled map service layers are pre-created and cached on the server and are most often used as basemaps in an application. Dynamic map service layers must be created on the fly each time a request is made and thus may take longer to generate. However, dynamic map service layers can be used to perform many types of operations including queries, setting definition expressions, and much more, and always show the latest version of the data.

You also learned how to provide navigation tools in your application through the use of widgets and toolbars. In addition, you learned how to programmatically control the map extent. Finally, we introduced the topic of events and you learned how to connect an event to an event handler, which is simply a JavaScript function that runs any time a particular event is triggered.

In the next chapter we'll look at how you can add graphics to your application.

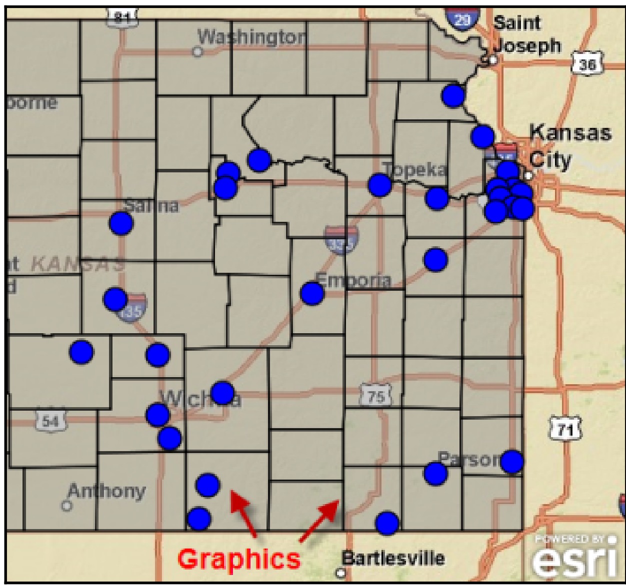
3

Adding Graphics to the Map

Graphics are points, lines, or polygons that are drawn on top of your map in a layer that is independent of any other data layers associated with a map service. Most people associate a graphic object with the symbol that is displayed on a map to represent the graphic. However, each graphic in ArcGIS Server can be composed of up to four objects including the geometry of the graphic, the symbology associated with the graphic, attributes that describe the graphic, and an `InfoTemplate` that defines the format of the `InfoWindow` that appears when a graphic is clicked. Although a graphic can be composed of up to four objects, it is not always necessary to include them all. The objects you choose to associate with your graphic will depend upon the needs of the application that you are building. For example, in an application that displays GPS coordinates on a map you might not need to associate attributes or display an `InfoWindow` for the graphic. However, in nearly all cases you will be defining the geometry and symbology for a graphic.

Graphics are temporary objects stored in a separate layer on the map that exists in-memory in the user's browser. They are displayed while an application is in use and removed when the session is complete. This layer, called a `GraphicsLayer`, hosts the graphics associated with your map. In `Chapter 2`, *Creating Maps and Adding Layers*, we discussed the various types of layers including dynamic map service layers and tiled map service layers. Just as with these other types of layers, `GraphicsLayer` also inherits from the `Layer` class. Therefore, all the properties, methods, and events found on the `Layer` class are also present on `GraphicsLayer`.

Graphics are displayed on top of any other layers that are present in your application. An example of point and polygon graphics is provided in the following screenshot. These graphics can be created by your users or drawn by the application in response to tasks that the application performs. For example, a business analysis application might provide a tool that allows the user to draw a free-hand polygon representing a potential trade area. The polygon graphic would be displayed on top of the map, and could then be used as an input to a geoprocessing task that pulls demographic information pertaining to the potential trade area:



Many ArcGIS Server tasks return their results as graphics. For example, the `QueryTask` can perform both attribute and spatial queries. The results of a query are then returned to the application in the form of a `featureSet` object which is simply an array of features. You can then iterate through each of the features in the array and render them as graphics on the map. Perhaps you'd like to find and display all of the land parcels that intersect the 100-year flood plain? A `QueryTask` could perform the spatial query and then return the results to your application where they would then be displayed as polygon graphics on the map.

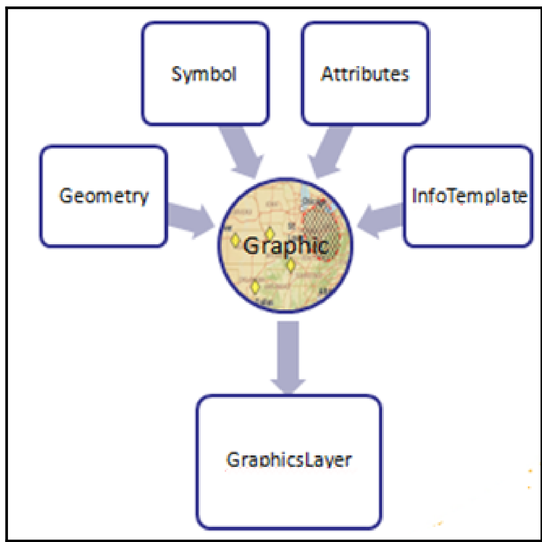
In this chapter we will cover the following topics:

- The four parts of a graphic
- Specifying graphic geometry
- Symbolizing graphics

- Assigning attributes to graphics
- Displaying graphic attributes in an `InfoTemplate`
- Creating the graphic
- Adding graphics to the `GraphicsLayer`

The four parts of a Graphic

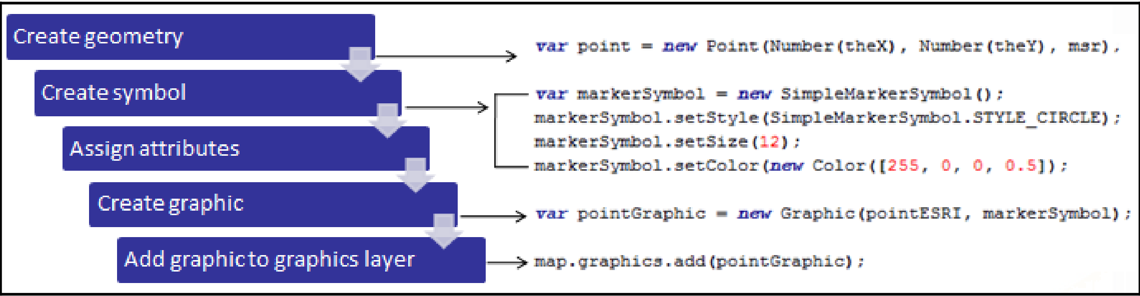
A graphic is composed of four items: **Geometry**, **Symbol**, **Attributes**, and an **InfoTemplate**, illustrated in the screenshot as follows:



A graphic has a geometric representation that describes its shape, and where it is located. However, a graphic can also have attributes which are name-value pairs that describe the graphic as well as an `InfoTemplate` that defines the format of the `InfoWindow` that appears when a graphic is clicked. After creation, graphic objects must be stored inside a `GraphicsLayer` object before they can be displayed on the map. This `GraphicsLayer` object functions as a container for all graphics that will be displayed.

All elements of a graphic are optional. However, the geometry and symbology of a graphic are almost always assigned. Without these two items there would be nothing to display on the map, and there isn't much point in having a graphic unless you're going to display it.

In the following you will see a code example showing the typical process for creating a graphic and adding it to the graphics layer. In this case we are applying the geometry of the graphic as well as a symbol for depicting the graphic. However, we haven't specifically assigned attributes or an `InfoTemplate` to this graphic:



Specifying graphic geometry

Graphics will almost always have a geometry component which is necessary for placement on the map. These geometry objects can be `Point`, `Multipoint`, `Polyline`, `Polygon`, or `Extent` and can be either created programmatically, or returned as the output from a task, such as a query.

Before creating any of these geometry types the required `esri/geometry` resource needs to be imported. The geometry resource contains classes for `Geometry`, `Point`, `Multipoint`, `Polyline`, `Polygon`, and `Extent`.

`Geometry` is the base class which is inherited by `Point`, `MultiPoint`, `Polyline`, `Polygon`, and `Extent`.

The following example shows how you can create a `Point` object:

```
new Point(-118.15, 33.80);
```

The `Point` class defines a location by an *x* and *y* coordinate, and can be in either map units or screen units.

Symbolizing graphics

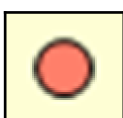
Each graphic that you create can be symbolized through one of the various symbol classes found in the API. Point graphics are symbolized by using the `SimpleMarkerSymbol` class and the available shapes include circle, cross, diamond, square, and X. You can also symbolize your points by using the `PictureMarkerSymbol` class which uses an image to display the graphic. Linear features are symbolized by using the `SimpleLineSymbol` class and can include solid lines, dashes, dots, or a combination. Polygons are symbolized by using the `SimpleFillSymbol` class and can be solid, transparent, or cross hatch. If you prefer to use an image in a repeating pattern for your polygons the `PictureFillSymbol` class is available. Text can also be added to a `GraphicsLayer` and is symbolized by using the `TextSymbol` class.

Points or multi-points can be symbolized by using the `SimpleMarkerSymbol` class which has various properties that can be set, including the style, size, outline, and color. Style is set through the `SimpleMarkerSymbol.setStyle()` method which takes the one of the constants seen as follows that corresponds to the type of symbol that is drawn (circle, cross, diamond, and so on).

```
STYLE_CIRCLE, STYLE_CROSS, STYLE_DIAMOND, STYLE_PATH, STYLE_SQUARE, STYLE_X
```

Point graphics can also have an outline color which is created by creating an instance of the `SimpleLineSymbol` class. You can also specify the size and color of the graphic:

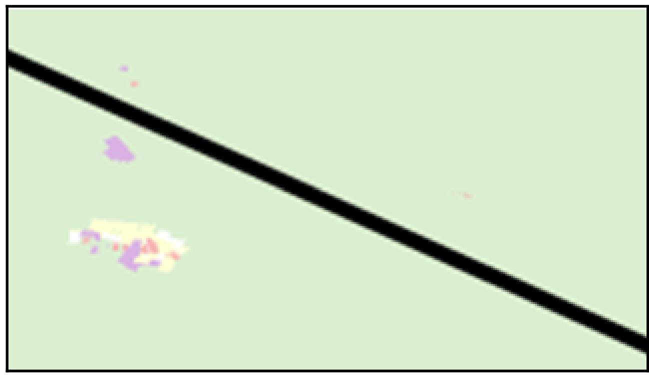
```
var markerSymbol = new SimpleMarkerSymbol();
markerSymbol.setStyle(SimpleMarkerSymbol.STYLE_CIRCLE);
markerSymbol.setSize(12);
markerSymbol.setColor(new Color([255,0,0,0.5]));
```



Linear features are symbolized with the `SimpleLineSymbol` class, and can consist either of a solid line or a combination of dots and dashes. Other properties of line symbols include color as defined with `dojo/Color` and a `width` property for setting the thickness of your line:

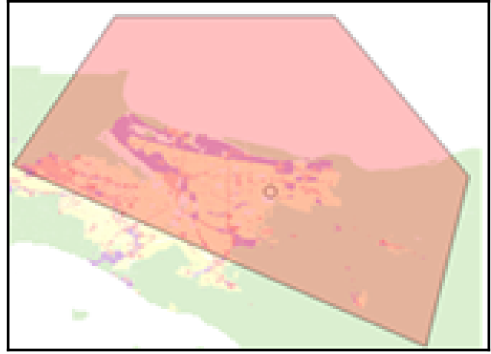
```
var polyline = new Polyline(msr);
//a path is an array of points
var path = [new Point(-123.123, 45.45, msr),.....];
polyline.addPath(path);
var lineSymbol = new SimpleLineSymbol().setWidth(5);
```

```
//create polyline graphic using polyline and line symbol
var polylineGraphic = new Graphic(polyline, lineSymbol);
map.graphics.add(polylineGraphic);
```



Polygons are symbolized by using the `SimpleFillSymbol` class which allows for the drawing of polygons in solid, transparent, or cross-hatch patterns:

```
var polygon = new Polygon(msr);
//a polygon is composed of rings
var ring = [[-122.98, 45.55], [-122.21, 45.21], [-122.13, 45.53],......];
polygon.addRing(ring);
var fillSymbol = new SimpleFillSymbol().setColor(new
Color([255,0,0,0.25]));
//create polygon graphic using polygon and fill symbol
var polygonGraphic = new Graphic(polygon, fillSymbol);
//add graphics to map's graphics layer
map.graphics.add(polygonGraphic);
```



You can also optionally specify an outline for the polygon by using a `SimpleLineSymbol` object.

Assigning attributes to graphics

The attributes of a graphic are the name/value pairs that describe that object. In many cases graphics are generated as the result of a task such as `QueryTask`. In such cases the geometry and attributes are derived from the query results and you just need to symbolize each graphic accordingly. The data columns associated with the layer you are querying become the attributes for the graphic. You can limit the range of attribute data returned by the query by setting properties on the task, such as `outFields`. If you are creating your graphics programmatically then you'll need to assign the attributes in your code using the `Graphic.setAttributes()` method seen in the code example that follows:

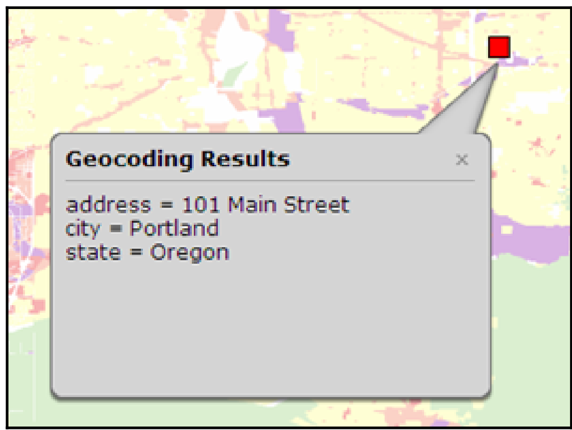
```
Graphic.setAttributes( {"XCoord":evt.mapPoint.x,
"YCoord".evt.mapPoint.y,"Plant":"Mesa Mint"});
```

Changing graphic attributes in an InfoTemplate

A graphic can also have an associated `InfoTemplate` that defines how the attribute data is displayed in a pop-up window. In the following code example, we create a point graphic, symbolize it, and then assign attribute information by using key/value pairs. In this instance we have keys that include address, city, and state. Each of the keys has a value. The variable that contains the attribute data is the third parameter we supply to the constructor for a new point graphic. An `InfoTemplate` defines the format of the pop-up window that appears and contains a title and an optional content template string:

```
var pointESRI = new Point(Number(theX), Number(theY),msr);
var markerSymbol = new SimpleMarkerSymbol();
markerSymbol.setStyle(SimpleMarkerSymbol.STYLE_SQUARE);
markerSymbol.setSize(12);
markerSymbol.setColor(new Color([255,0,0]));
var pointAttributes = {address:"101 Main Street", city:"Portland",
state:"Oregon"};
var pointInfoTemplate = new InfoTemplate("Geocoding Results");
//create point graphic using point and marker symbol
var pointGraphic = new Graphic(pointESRI, markerSymbol,
pointAttributes).setInfoTemplate(pointInfoTemplate);
//add graphics to maps' graphics layer
```

```
map.graphics.add(pointGraphic);
```



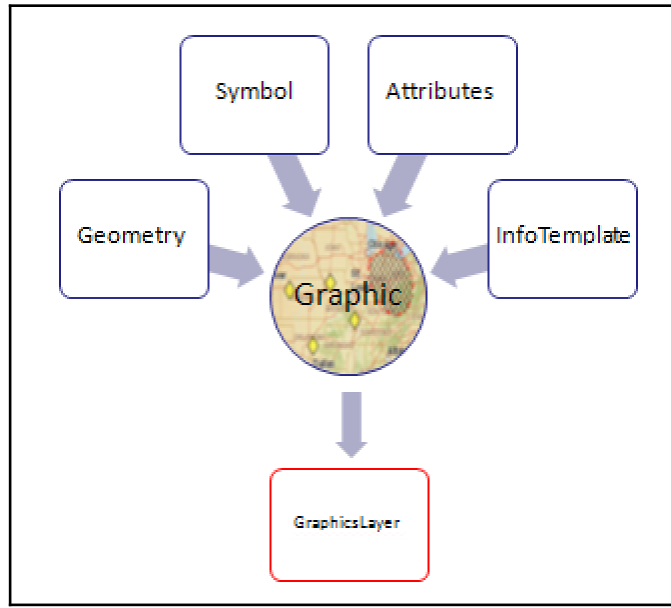
Creating the graphic

Once you've defined the geometry, symbology and attributes for your graphic, a new `Graphic` object can be created by passing this information to its constructor. Notice how in the following code example we create variables for the geometry (`pointESRI`), symbology (`markerSymbol`), and point attributes (`pointAttributes`) and then pass these as parameters to the constructor for our new graphic called `pointGraphic`. Then we apply the `InfoTemplate` by calling the new `Graphic` object's `setInfoTemplate()` method. Finally, we add the graphic to the map's `GraphicsLayer`:

```
var pointESRI = new Point(Number(theX), Number(theY, msr);
var markerSymbol = new SimpleMarkerSymbol();
markerSymbol.setStyle(SimpleMarkerSymbol.STYLE_SQUARE);
markerSymbol.setSize(12);
markerSymbol.setColor(new Color([255,0,0]));

var pointAttributes = {address:"101 Main Street", city:"Portland",
state:"Oregon"};
var pointInfoTemplate = new InfoTemplate("Geocoding Results");
//create the point graphic using point and marker symbol
var pointGraphic = new Graphic(pointESRI, markerSymbol,
pointAttributes).setInfoTemplate(pointTemplate);

//add graphics to map's graphics layer
map.graphics.add(pointGraphic);
```

Adding graphics to the GraphicsLayer

Before any of your graphics are displayed on the map you must add them to the `GraphicsLayer`. Each map has a `GraphicsLayer` which contains an array of graphics that is initially empty until you add the graphics:



The `GraphicsLayer` can contain any type of graphic objects, so you can mix points, lines, and polygons in the same layer. Graphics are added to the layer through the `add()` method and can also be removed individually through the `remove()` method. In the event that you need to remove all graphics simultaneously the `clear()` method can be used. `GraphicsLayer` also has a number of events that can be listened to, including `click`, `mouse-down`, and others, to add more interactivity to your application.

Multiple GraphicsLayers

The API supports adding multiple graphics layers to the map, making it much easier to organize different types of graphics. Layers can easily be removed or added as needed. For example, you can put polygon graphics representing counties in one graphics layer and point graphics representing traffic incidents in another graphics layer. Then you can display or hide either layer as needed.

Practice time

In this exercise you will learn how to create and display graphics on a map. We are going to create a thematic map showing population density by county for the State of Colorado. We will also introduce you to the `QueryTask`. As you will learn in a later chapter, *tasks* are special workflows that the ArcGIS Server can perform, and include things like spatial and attribute queries, identification of features, geocoding, and more. Finally, you will see how you can attach attributes to your graphic features and display them in an `InfoWindow`:

1. Open the JavaScript Sandbox at `https://developers.arcgis.com/javascript/3/sandbox/sandbox.html`.
2. Remove the JavaScript content from the `<script>` tag that is highlighted as follows:

```
<script>
  var map;

  require(["esri/map", "dojo/domReady!"], function(Map) {
    map = new Map("map", {
      basemap: "topo",  //For full list of pre-defined
basemaps,
      navigate to http://arcg.is/1JVo6Wd
      center: [-122.45, 37.75], // longitude, latitude
      zoom: 13
    });
  });
</script>
```

3. Create the variables that you'll use in the application:

```
<script>
  var map, defPopSymbol, onePopSymbol, twoPopSymbol,
threePopSymbol,
  fourPopSymbol, fivePopSymbol;
</script>
```

4. Add the `require()` function as seen in the highlighted code as follows:

```
<script>
  var map, defPopSymbol, onePopSymbol, twoPopSymbol,
threePopSymbol,
  fourPopSymbol, fivePopSymbol;
  require(["esri/map", "esri/tasks/query",
"esri/tasks/QueryTask",
  "esri/symbols/SimpleFillSymbol",
```

```
  "esri/InfoTemplate", "dojo/_base/Color", "dojo/domReady!"],
  function(Map, Query, QueryTask, SimpleFillSymbol,
InfoTemplate,
  Color) {
  });
</script>
```

We have covered the `esri/map` resource in a past exercise so no additional explanation should be necessary. However, the `esri/tasks/query` and `esri/tasks/QueryTask` resources are new, and we won't cover this in detail until a later chapter. However, in order to complete this exercise we need you to understand the basics. This `QueryTask` enables you to perform spatial and attribute queries against a data layer, and the `query` module enables you define an object to represent the query you want to execute.

5. Inside the `require()` function, you will need to create a `Map` object and add a basemap street layer by adding the following highlighted code . You will set the initial map extent to display the State of Colorado:

```
<script>
  var map, defPopSymbol, onePopSymbol, twoPopSymbol,
threePopSymbol,
  fourPopSymbol, fivePopSymbol;
  require(["esri/map", "esri/tasks/query",
"esri/tasks/QueryTask",
  "esri/symbols/SimpleFillSymbol", "esri/InfoTemplate",
  "dojo/_base/Color", "dojo/domReady!"],
  function(Map, Query, QueryTask, SimpleFillSymbol,
InfoTemplate,
  Color) {
    map = new Map("map", {
      basemap: "streets",
      center: [-105.498,38.981], // long, lat
      zoom: 6,
      sliderStyle: "small"
    });
  });
</script>
```

6. Inside the `require()` function, just under the code block that creates the `Map`, add the line of code to create a new polygon symbol which is transparent. This code creates a new `SimpleFillSymbol` and assigns it to the variable `defPopSymbol`. We use RGB values of `255,255,255,0` to ensure that the fill color will be completely transparent. This is accomplished through the value `0` which ensures that our coloring will be fully transparent. Later we will add additional symbol objects so that we can display a color-coded map of county population density. For now though, we simply want to create a simple symbol so that you can understand the basic procedure for creating and displaying graphics on a map:

```
map = new Map("mapDiv", {
  basemap: "streets",
  center: [-105.498,38.981], // long, lat
  zoom: 6,
  sliderStyle: "small"
});
defPopSymbol = new SimpleFillSymbol().setColor(new
Color([255,255,255, 0])); // transparent
```

In this next step you're going to get a sneak peek at how to use the `QueryTask`. We'll cover this task in detail in a later chapter. You use `QueryTask` to perform spatial and attribute queries on a data layer in a map service. In this exercise we are going to use the `QueryTask` to perform an attribute query against a county boundary layer provided by an ESRI map service.

7. Let's first examine the map service and layer that we will use in our query. Open a web browser and point to `http://sampleserver1.arcgisonline.com/ArcGIS/rest/services/Specialty/ESRI_StateCityHighway_USA/MapServer`:

ArcGIS Services Directory

Home > Specialty > ESRI_StateCityHighway_USA (MapServer)

Specialty/ESRI_StateCityHighway_USA (MapServer)

View In: ArcMap ArcGIS Explorer ArcGIS JavaScript Google Earth ArcGIS.com Map

View Footprint In: Google Earth

Service Description: This service provides census information for U.S. cities and states including total population, racial counts, and more. It also includes highwa
Server. ESRI has provided this example so that you may practice using ArcGIS APIs for JavaScript, Flex, and Silverlight. ESRI reserves the right to change or remov

Map Name: Layers

Legend

All Layers and Tables

Layers:

- ushigh (0)
- states (1)
- counties (2)

Tables:

Description: This service provides census information for U.S. cities and states including total population, racial counts, and more. It also includes highways.

Copyright Text: (c) ESRI and its data partners

Spatial Reference: 4326

This map service provides census information for U.S. states and counties and also includes a highway layer. In this exercise we are interested in the counties layer which has an index number of 2. Click the counties link to get detailed information about this layer. There are a lot of fields in this layer, but we are really only interested in a field that will allow us to query by state name and a field that gives us population density information. The `STATE_NAME` field gives us the state name for each county, and the `POP90_SQMI` field gives us population density for each county.

8. Return to the Sandbox. Under the line of code where we created our symbol, create the `QueryTask` by adding the following line of code just under the line that created the `defPopSymbol` variable. What this line does is create a new `QueryTask` object that points to the `ESRI_StateCityHighway_USA` map service that we just examined in our browser, and specifically points to the layer with index `2`, which is our county layer:

```
var queryTask = new
QueryTask("http://sampleserver1.arcgisonline.com/ArcGIS/rest/se
rvices/Specialty/ESRI_StateCityHighway_USA/MapServer/2");
```

9. Now you need to create the `Query` object which defines the nature of the query. Add the following line of code just under the line you just entered:

```
var query = new Query();
```

10. Now, we'll set some of the properties on our new `Query` object that will enable us to perform an attribute query. Add the following three lines of code just under the line that created the `query` variable:

```
var query = new Query();
query.where = "STATE_NAME = 'Colorado'";
query.returnGeometry = true;
query.outFields = ["POP90_SQMI"];
```

11. The `where` property is used to specify a SQL statement that will be executed against the layer. In this case, we're stating that we'd like to return only county records that have a state name of `Colorado`. Setting the `returnGeometry` property to `true` indicates that we want ArcGIS Server to return the geometries for all features that matched our query. This is necessary because we need to plot these features as graphics on top of the map and we need their geometries to do this. The `outFields` property is used to define which attribute fields we'd like returned. This information will be used later when we create the color coded map of county population density.
12. Finally, we'll use the `execute()` method on the `QueryTask` to perform the query against the layer we have indicated (counties) using the parameters defined in our `Query` object. Add the following line of code:

```
queryTask.execute(query, addPolysToMap);
```

In addition to passing the `Query` object into ArcGIS Server, we have also indicated that `addPolysToMap()` will serve as the callback function. This function executes after ArcGIS Server has performed the query and returned the results. It is up to the `addPolysToMap()` function to plot the records using the `featureSet` returned to it.

Before creating the `addPolysToMap()` callback function let's first discuss what the code will accomplish. The `addPolysToMap` function will take a single parameter: `featureSet`. When a `QueryTask` has executed, ArcGIS Server returns a `featureSet` object back to your code. A `featureSet` object contains the graphic objects returned by the query. Inside the `addPolysToMap` function you will see the line:

```
var features = featureSet.features;
```

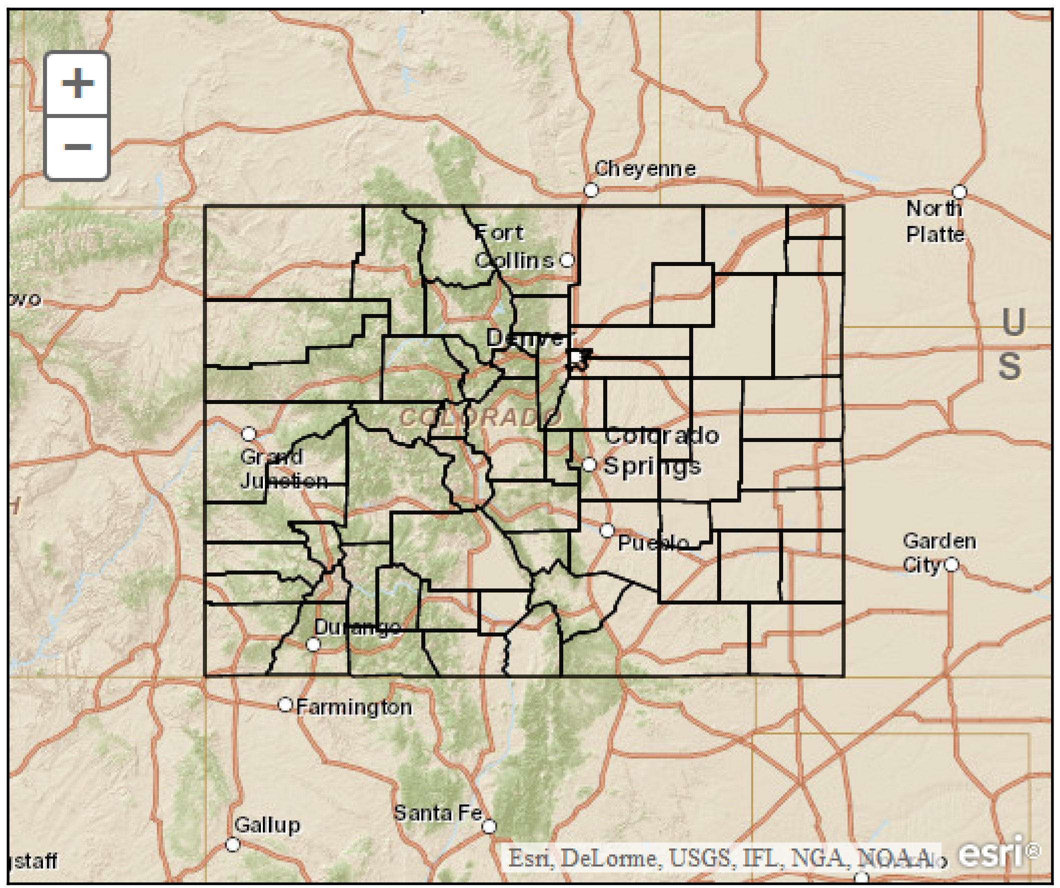
The `features` property returns an array of graphics objects, one for each feature returned by the query. We will use a loop construct to iterate through each of the graphics in the array and plot them on the map:

13. Create the callback function by adding the code block that follows under the line of code that executes the `QueryTask`:

```
function addPolysToMap(featureSet) {
  var features = featureSet.features;
  var feature;
  for (var i=0, il=features.length; i<il; i++) {
    feature = features[i];
    map.graphics.add(features[i].setSymbol(defPopSymbol));
  }
}
```

As I mentioned earlier, you have to add each graphic that you create to the `GraphicsLayer` object. This is done by calling the `GraphicsLayer` object's `add()` method, as you can see in the preceding code. You will also notice that we are attaching the symbol we created earlier to each of the graphics (county boundaries).

14. Execute the code by clicking the **Refresh** button and you should see the following output if your code is correct. Notice that each of the counties has been outlined with the symbol that we defined:



Now we're going to add additional code to the application that will color code each of the county polygons based on population.

15. Comment out the `defPopSymbol` variable inside the `require()` function and add five new symbols as follows:

```
//defPopSymbol = new SimpleFillSymbol().setColor(new
Color([255,255,255, 0])); //transparent
onePopSymbol = new SimpleFillSymbol().setColor(new
Color([255,255,128, .85])); //yellow
twoPopSymbol = new SimpleFillSymbol().setColor(new
```

```
Color([250,209,85, .85]));
threePopSymbol = new SimpleFillSymbol().setColor(new
Color([242,167,46, .85])); //orange
fourPopSymbol = new SimpleFillSymbol().setColor(new
Color([173,83,19, .85]));
fivePopSymbol = new SimpleFillSymbol().setColor(new
Color([107,0,0, .85])); //dark maroon
```

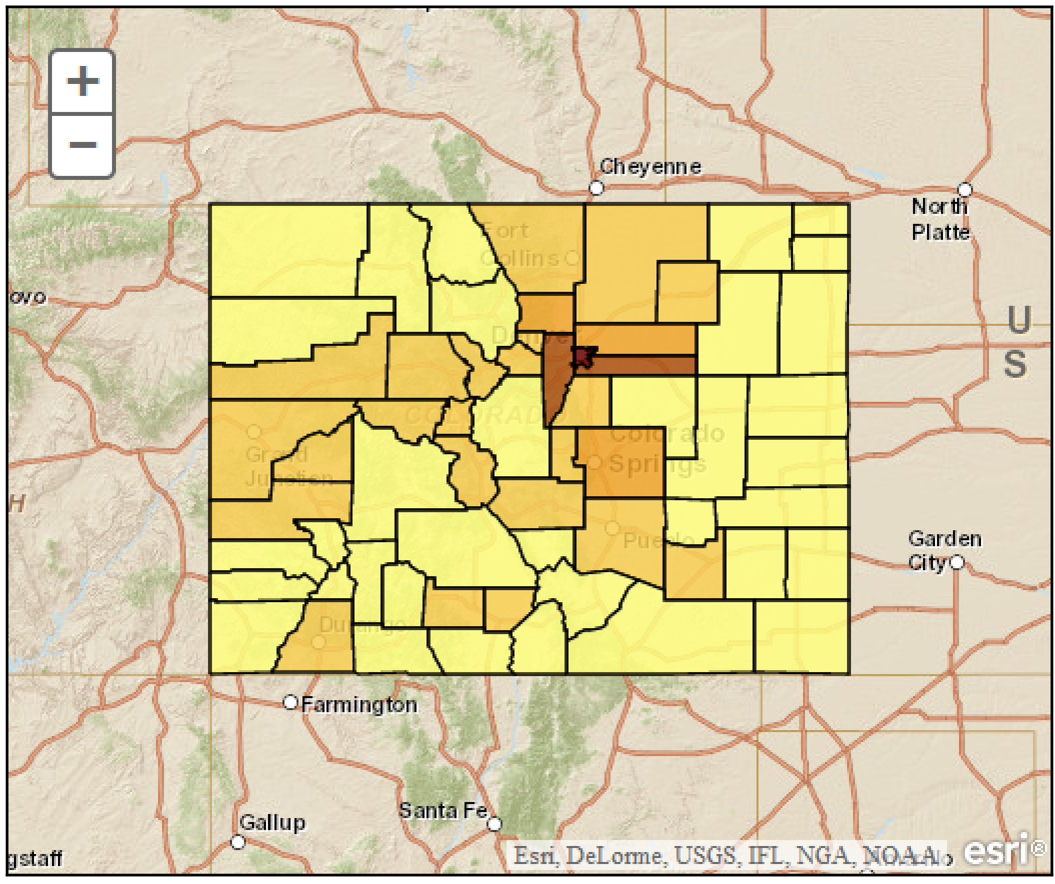
What we're doing here is basically creating a color ramp of symbols that will be assigned to each county based on population density. We are also applying a transparency value of .85 to each symbol so that we will be able to see through each of the counties. This will enable us to see the city names marked on the underlying base map.

16. Recall that earlier in the exercise we created `QueryTask` and `Query` objects, and that we defined an `outFields` property on `Query` to return the `POP90_SQMI` field. This will now come into play as we use the values returned in this field to determine the symbol applied to each county, based on the population density of that county. Update the `addPolysToMap` function as shown in the following and then we'll discuss what we've done:

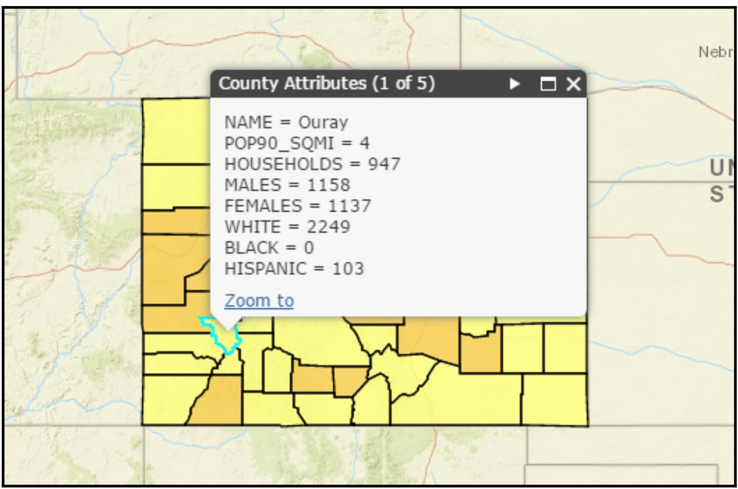
```
function addPolysToMap(featureSet) {
  var features = featureSet.features;
  var feature;
  for (var i=0, il=features.length; i<il; i++) {
    feature = features[i];
    attributes = feature.attributes;
    pop = attributes.POP90_SQMI;
    if (pop < 10) {
      map.graphics.add(features[i].setSymbol(onePopSymbol));
      } else if (pop >= 10 && pop < 95) {
map.graphics.add(features[i].setSymbol(twoPopSymbol));
      } else if (pop >= 95 && pop < 365) {
map.graphics.add(features[i].setSymbol(threePopSymbol));
      } else if (pop >= 365 && pop < 1100) {
map.graphics.add(features[i].setSymbol(fourPopSymbol));
      } else {
map.graphics.add(features[i].setSymbol(fivePopSymbol));
      }
   }
}
```

17. What we've done with this code block is obtain the population density information from each graphic and save it to a variable called `pop`. An `if/else` code block is then used to assign a symbol to the graphic based on the population density of that county. For example, a county with a population density (as defined in the `POP90_SQMI` field) of `400` would be assigned the symbol defined by `fourPopSymbol`. The for loop ensures that this is done for every graphic in the `featureSet.features` array.

    Execute the code by clicking the **Refresh** button and you should see the following output. Notice that each of the counties has been color coded using one of the symbols we defined, based on the population density:



In the next step you will learn how to attach attributes to graphics and display them in an `InfoWindow` when the user clicks on the graphic:



An `InfoWindow` is an HTML popup window that displays when you click a graphic on the map. Normally it contains the attributes of the clicked graphic, but it can also contain custom content that you specify as a developer. The content of these windows is specified through an `InfoTemplate` object which specifies a title for the window and the content to display in the window. The easiest way to create an `InfoTemplate` object is to use a wildcard for the content that will automatically insert all the fields of a dataset into the `InfoWindow`. We are going to add some additional output fields so that more content can be displayed in the `InfoWindow`.

18. Modify the line of code that sets the `Query` object's `outfield` property to include the fields highlighted in the code as follows:

```
query.outFields =
["NAME","POP90_SQMI","HOUSEHOLDS","MALES","FEMALES","WHITE","BL
ACK","HISPANIC"];
```

19. Add the following line of code just under the call to `QueryTask.execute`:

```
resultTemplate = InfoTemplate("County Attributes", "${*}");
```

20. The first parameter passed into the constructor (the `"County Attributes"` string) is the title for the window. The second parameter is a wildcard indicating that all the attribute's name/value pairs should be displayed. Therefore, the new fields we added to `Query.outFields` should all be included in the `InfoWindow` when a graphic is clicked.
21. Finally, we must assign the newly created `InfoTemplate` to each graphic. We call `Graphic.setInfoTemplate()` on each graphic as we process it in the `for` loop, but because we are going to use the same template for all the graphics in the layer, we can call the `setInfoTemplate()` method of `GraphicsLayer` instead.
22. Amend the `addPolysToMap()` function as shown:

```
function addPolysToMap(featureSet) {
  var features = featureSet.features;
  var feature;
  for (var i=0, il=features.length; i<il; i++) {
    feature = features[i];
    attributes = feature.attributes;
    pop = attributes.POP90_SQMI;
    if (pop < 10) {
      map.graphics.add(features[i].setSymbol(onePopSymbol));
    } else if (pop >= 10 && pop < 95) {
      map.graphics.add(features[i].setSymbol(twoPopSymbol));
    } else if (pop >= 95 && pop < 365) {
      map.graphics.add(features[i].setSymbol(threePopSymbol));
    } else if (pop >= 365 && pop < 1100) {
      map.graphics.add(features[i].setSymbol(fourPopSymbol));
    } else {
      map.graphics.add(features[i].setSymbol(fivePopSymbol));
    }
  }
  map.graphics.setInfoTemplate(resultTemplate);
}
```

23. Execute the code by clicking the **Refresh** button. Click any of the counties in the map and you should see an `InfoWindow` similar to the screenshot shown at the start of this activity.
24. You can view the solution code for this exercise in the `graphics.html` file in the `Chapter3` folder to verify that your code has been written correctly.

Summary

In this chapter you learned that graphics are often used to represent information that is generated as the result of actions performed within a working application. Frequently these graphics are returned as the result of a *task,* such as an attribute or spatial query. Graphics include points, lines, polygons, and text. These are temporary objects, displayed only during the current browser session. Each graphic can be composed of geometry, symbology, attributes, and an `Infotemplate`, and is added to the map through the use of a `GraphicsLayer` which sits on top of all the map service layers to ensure that the contents of the `GraphicsLayer` are always visible. In the next chapter we'll introduce you to the `FeatureLayer`, which can do everything that a `GraphicsLayer` can do and more!

4
The Feature Layer

The ArcGIS API for JavaScript offers an alternative approach to working with client-side graphic features: the `FeatureLayer`.

`FeatureLayer` inherits from the `GraphicsLayer` but also offers additional capabilities such as the ability to perform queries and selections and filtering of data using definition expressions. The `FeatureLayer` can also be used for web-based editing, as we will see in a later chapter.

The main way in which the `FeatureLayer` differs from tiled and dynamic map service layers is that it doesn't perform the rendering server side. Instead, it sends the geometries and their associated attributes to the browser. The browser renders the features from the geometries and stores all their attribute data in memory.

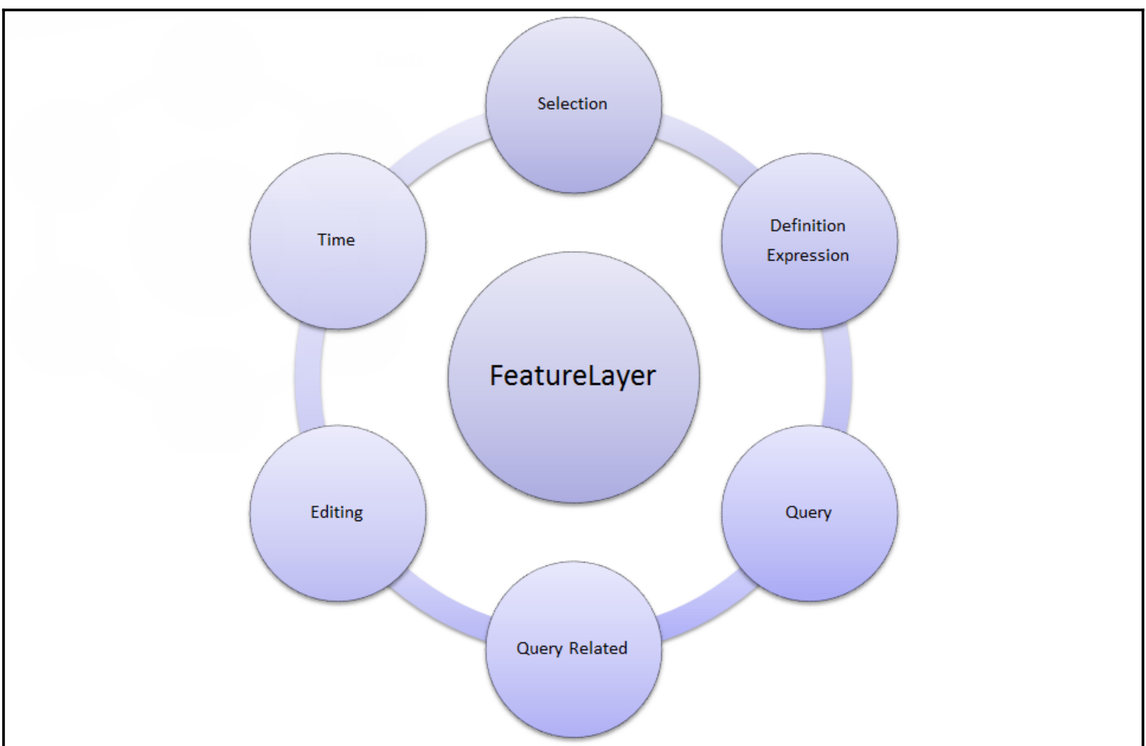
Compare this to the tiled and dynamic map service layers that we have already seen, which are rendered on the server as images and sent to the browser for display. If you want to retrieve attribute data from these services, you must query for it, which involves another round trip to the server.

Streaming the data from the ArcGIS server in this way can, therefore, improve the performance of your application by minimizing requests to the server. A client can request the features it needs and perform selections and queries on those features without having to request more information from the server.

As such, the `FeatureLayer` is especially appropriate for layers that respond to user interactions such as mouse clicks or hovers. The information required to process these interactions is already present in the client.

The catch is that if you're working with a `FeatureLayer` that contains a lot of features, it can take a long time to initially transport the features to the client. Luckily, the `FeatureLayer` supports several display modes that can help ease this burden of working with a large number of features. We'll examine each of these display modes in this chapter.

The `FeatureLayer` honors any definition expressions, scale dependencies, and other properties configured on the layer in the map service. Using `FeatureLayer`, you can access related tables, perform queries, display time periods, work with feature attachments, and do other useful things:



In this chapter, we will cover the following topics:

- Creating the `FeatureLayer`
- Defining the display mode
- Setting a definition expression

- Feature selection
- Rendering the `FeatureLayer`
- Practice time

Creating the FeatureLayer

The `FeatureLayer` must reference a layer from either a map service or a feature service. Use a map service if you just want to retrieve geometries and attributes from the server and symbolize them yourself. Use a feature service if you want to take advantage of symbols from the service's source map document. Also, use a feature service if you plan on performing web-based editing with the `FeatureLayer`. Feature layers honor any feature editing templates configured in the source map document.

The following code example shows how to create a `FeatureLayer` using its constructor. With tiled and dynamic layers, you simply provide a pointer to the REST endpoint of a map service, but with the `FeatureLayer`, you need to point to a specific *layer* in the service. In the following code example, we are creating a `FeatureLayer` from the first layer in the service, which is indicated by the number 0. The constructor for `FeatureLayer` also accepts options such as the display mode, output fields, info template, and others. Here, the display mode is set to "snapshot," which is the best access method for fairly small datasets. We'll cover the different display modes for `FeatureLayer` and when to use them in the next section:

```
var earthquakes = new
FeatureLayer("http://sampleserver3.arcgisonline.com/ArcGIS/rest/services/Ea
rthquakes/Since_1970/MapServer/0",{ mode: FeatureLayer.MODE_SNAPSHOT,
outFields: ["Magnitude"]});
```

Optional constructor parameters

In addition to specifying the layer (which is mandatory), you can also optionally pass in a JSON object that defines various options to the constructor. The constructor accepts many such options. We'll look at some of the more commonly used ones.

The `outFields` property restricts the fields that are returned with `FeatureLayer`. For performance reasons, it's best only to include the fields that the application needs rather than accepting the default of returning all fields.

In the following highlighted code, we've used the `outFields` property of the options object to tell the server to return only the `Name` and `Magnitude` fields:

```
var earthquakes = new
FeatureLayer("http://sampleserver3.arcgisonline.com/ArcGIS/rest/services/Ea
rthquakes/Since_1970/MapServer/0",{ mode: FeatureLayer.MODE_SNAPSHOT,
outFields: ["Name", "Magnitude"]});
```

The `refreshInterval` property defines how often (in minutes) to refresh the layer. Consider decreasing the value of `refreshInterval` if your underlying map service layer contains data that changes frequently:

```
var earthquakes = new FeatureLayer("
http://sampleserver3.arcgisonline.com/ArcGIS/rest/services/Earthquakes/Sinc
e_1970/MapServer/0",
{ mode: FeatureLayer.MODE_SNAPSHOT, outFields: ["Name","Magnitude"],
refreshInterval: 5});
```

To define the attributes and styling that should be displayed in an info window when a feature is clicked, you can set the `infoTemplate` property:

```
var infotemplate = new InfoTemplate("${Name}", "Magnitude: ${Magnitude}");
var earthquakes = new FeatureLayer("
http://sampleserver3.arcgisonline.com/ArcGIS/rest/services/Earthquakes/Sinc
e_1970/MapServer/0",{
  mode: FeatureLayer.MODE_SNAPSHOT,
  outFields: ["Name", "Magnitude"],
  refreshInterval: 5,
  infoTemplate:infotemplate
});
```

The most important option is probably the display mode, defined with the `mode` parameter, so we'll cover that in more detail.

Defining the display mode

When creating an instance of `FeatureLayer`, you need to specify a mode for retrieving features. Because the mode determines when and how features are brought from the server to the client, your choice can affect the behavior and performance of your application.

You can choose from the following modes:

- Snapshot mode: `MODE_SNAPSHOT`
- On-demand mode: `MODE_ONDEMAND`
- Selection mode: `MODE_SELECTION`
- Auto mode: `MODE_AUTO`

The snapshot mode

The snapshot mode retrieves all the feature from the map service layer and streams them to the client browser in one go. For this reason, you must carefully consider the size of your layer before using this mode. Generally, you should use this mode only with small datasets. Large datasets in snapshot mode can significantly degrade the performance of your application. The benefit of the snapshot mode is that since all features from the layer are sent to the client when the layer loads, there is no need to return to the server for additional data. This can improve your application's performance significantly.

ArcGIS imposes a limit of 1,000 features that may be returned at any one time, though this number is configurable through ArcGIS Server administration. This shouldn't be a problem because, as already mentioned, you will want to only use this mode when you're working with small datasets:

```
var earthquakes = new
FeatureLayer("http://sampleserver3.arcgisonline.com/ArcGIS/rest/services/Ea
rthquakes/Since_1970/MapServer/0",{ mode: FeatureLayer.MODE_SNAPSHOT,
outFields: ["Name", "Magnitude"]});
```

The preceding code example shows how to set the `FeatureLayer` to snapshot mode.

The on-demand mode

The on-demand mode retrieves features only as they are required by the application. What this means is that the server returns only the features within the current view extent. Therefore, every time a zoom or pan operation takes place, new features are streamed to the client from the server. This tends to work well with larger datasets that are not suitable for the snapshot mode.

It does require a round trip to the server to fetch the features each time the map extent changes, but for large datasets, this is preferable:

```
var earthquakes = new
FeatureLayer("http://sampleserver3.arcgisonline.com/ArcGIS/rest/services/Ea
rthquakes/Since_1970/MapServer/0",{ mode: FeatureLayer.MODE_ONDEMAND,
outFields: ["Name", "Magnitude"]});
```

The preceding code example shows how to set a `FeatureLayer` to the on-demand mode.

The selection only mode

Selection only mode does not initially request any features. Instead, features are returned only when a selection is made on the client. Selected features are streamed to the client from the server. These selected features are then held on the client. We will discuss how features are selected later on:

```
var earthquakes = new
FeatureLayer("http://sampleserver3.arcgisonline.com/ArcGIS/rest/services/Ea
rthquakes/Since_1970/MapServer/0",{ mode: FeatureLayer.MODE_SELECTION,
outFields: ["Name", "Magnitude"]});
```

The preceding code example shows how to set `FeatureLayer` to the selection only mode.

The auto mode

The auto mode makes the decision for you. However, it only works with hosted feature services, which we won't cover in detail until we discuss editing.

Setting a definition expression

Definition expressions are used to filter features that are streamed to the client based on the values of attribute fields. `FeatureLayer` contains a `setDefinitionExpression()` method that is used to create the filter expression. All features that meet the specified criteria will be returned for display on the map. You can build expressions using standard SQL syntax, as seen in the code example as follows:

```
featureLayer.setDefinitionExpression("Magnitude>7");
```

You can retrieve the current definition expression using the `FeatureLayer.getDefinitionExpression()` method, which returns a string containing the expression.

Feature selection

`FeatureLayer` also supports feature selection. Selected features are a subset of the features in a layer which can be used for viewing, editing, analysis, or as an input to other operations. Features are added to or removed from a selection set using either spatial or attribute criteria. The features in the selection set are often drawn with different symbologies to differentiate them from the other features in the layer.

To create a selection set, use the `selectFeatures(query)` method on `FeatureLayer`. A parameter is a `Query` object exactly like the one you have already seen with `QueryTask`. This is illustrated by the following code example:

```
var selectQuery = new Query();
selectQuery.geometry = geometry;
 featureLayer.selectFeatures(selectQuery,FeatureLayer.SELECTION_NEW);
```

We haven't covered the `Query` object in detail yet, but just like in `QueryTask`, you use it to define the input parameters for an attribute or spatial query.

The preceding screenshot shows a feature that has been selected. A selection symbol has been applied to the selected feature.

Any definition expression set on a layer, either through the application or on the layer inside the map document file, will be honored. Setting a symbol to use for the selected features is quite easy and simply involves creating a symbol and then using the `setSelectionSymbol()` method on `FeatureLayer`. Selected features will automatically be assigned this symbol. You can opt to define a new selection set, add features to an existing selection set, or remove features from a selection set through the provided constants, including `SELECTION_NEW`, `SELECTION_ADD`, and `SELECTION_SUBTRACT`. The following code example demonstrates how to define a new selection set:

```
fLayer.selectFeatures(selectQuery,FeatureLayer.SELECTION_NEW);
```

The first parameter is the `Query` object and the second is a constant that tells the API that we want to create a new selection set.

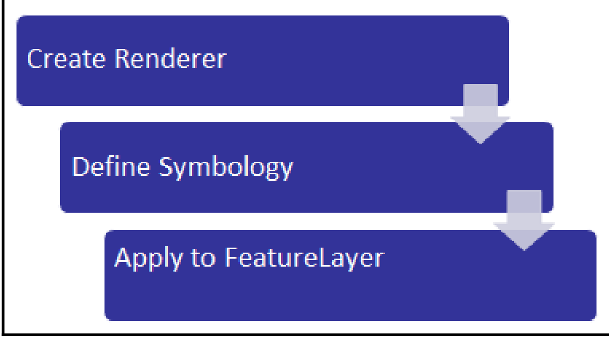
In addition, you can provide further parameters that define `callback` and `errback` functions to process the returned features or handle any errors, respectively.

Rendering FeatureLayer

You can use the `renderer` property to define a set of symbols for a `FeatureLayer` or a `GraphicsLayer`. These symbols can have different colors and/or sizes based on an attribute.

There are many different types of renderers in the ArcGIS Server API for JavaScript. Common ones include `SimpleRenderer`, `ClassBreaksRenderer`, `UniqueValueRenderer`, `DotDensityRenderer`, and `TemporalRenderer`. We'll examine each of these renderers in this section.

The rendering process will be the same regardless of the type of `renderer` you use. You first need to create an instance of the renderer, define the symbology for it, and then finally apply the renderer to the `FeatureLayer`. This rendering process is illustrated as follows:

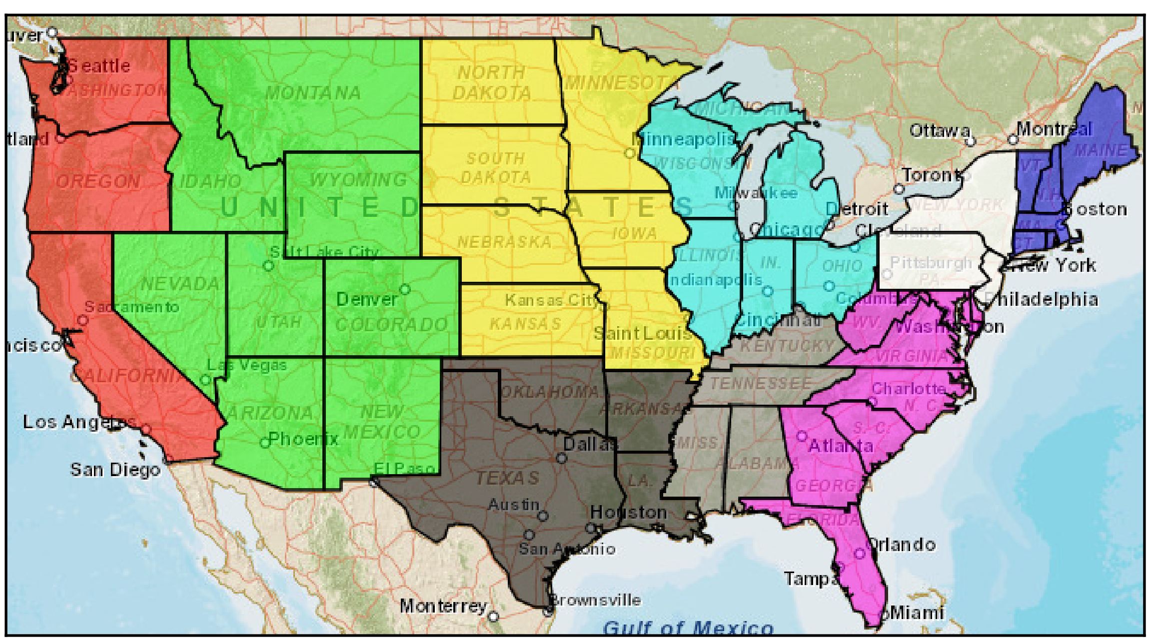


The simplest type of renderer is `SimpleRenderer`, which simply applies the same symbol for all graphics. All you need to do is define the symbol, associate it with the renderer, and then apply the renderer to your graphics layer:

```
var symbol = new SimpleMarkerSymbol();
 symbol.style = SimpleMarkerSymbol.STYLE_CIRCLE;
 symbol.setSize(10);
 symbol.setColor(new Color([255,0,0,0.5]));
 var renderer = new SimpleRenderer(symbol);
graphicsLayer.setRenderer(renderer);
```

The preceding example creates a red circle, 10-pixels high, and uses it to symbolize all graphics in the layer.

`UniqueValueRenderer` can be used to symbolize graphics based on a matching attribute, which is typically a field containing string data:



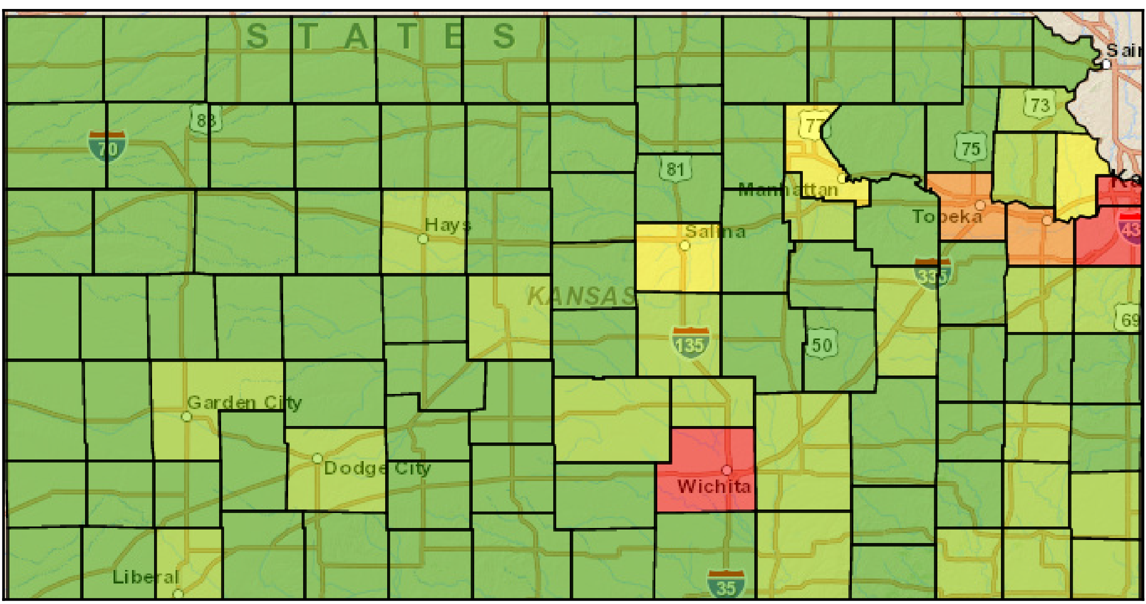
For example, if you have a states feature class you might want to symbolize each feature based on a region name. Each region would be associated with a different symbol. The following code example shows how you can create a `UniqueValueRenderer` programmatically and add values and symbols to it:

```
var renderer = new UniqueValueRenderer(defaultSymbol, "REGIONNAME");
renderer.addValue("West", new SimpleLineSymbol().setColor(new Color([255,
255, 0, 0.5])));
renderer.addValue("South", new SimpleLineSymbol().setColor(new Color([128,
0, 128, 0.5])));
renderer.addValue("Mountain", new SimpleLineSymbol().setColor(new
Color([255, 0, 0, 0.5])));
```

The `ClassBreaksRenderer` works with numeric attribute data. Each graphic will be symbolized according to the value of that particular attribute in accordance with numeric ranges that you define.

The breaks define the values at which the symbol will change. For example, with a parcel feature class, you might want to symbolize parcels based on values found in the `PROPERTYVALUE` field. You must first create a new instance of `ClassBreaksRenderer` and then define the breaks for the data. The `Infinity` and `-Infinity` values can be used as the lower and upper boundaries for your data if required, as shown in the following code example. Here, we use the `Infinity` keyword to signify a class break for any values greater than 250,000:

```
var renderer = new ClassBreaksRenderer(symbol, "PROPERTYVALUE");
renderer.addBreak(0, 50000, new SimpleFillSymbol().setColor(new Color([255,
0, 0, 0.5])));
renderer.addBreak(50001, 100000, new SimpleFillSymbol().setColor(new
Color([255, 255, 0, 0.5])));
renderer.addBreak(100001, 250000, 50000, new
SimpleFillSymbol().setColor(new Color([0, 255, 0, 0.5])));
renderer.addBreak(250001, Infinity, new SimpleFillSymbol().setColor(new
Color([255, 128, 0, 0.5])));
```



A `TemporalRenderer` provides render features according to time values. This type of renderer is often used for displaying historical information or near real-time data. The temporal renderer allows you to define how observations and tracks are rendered:

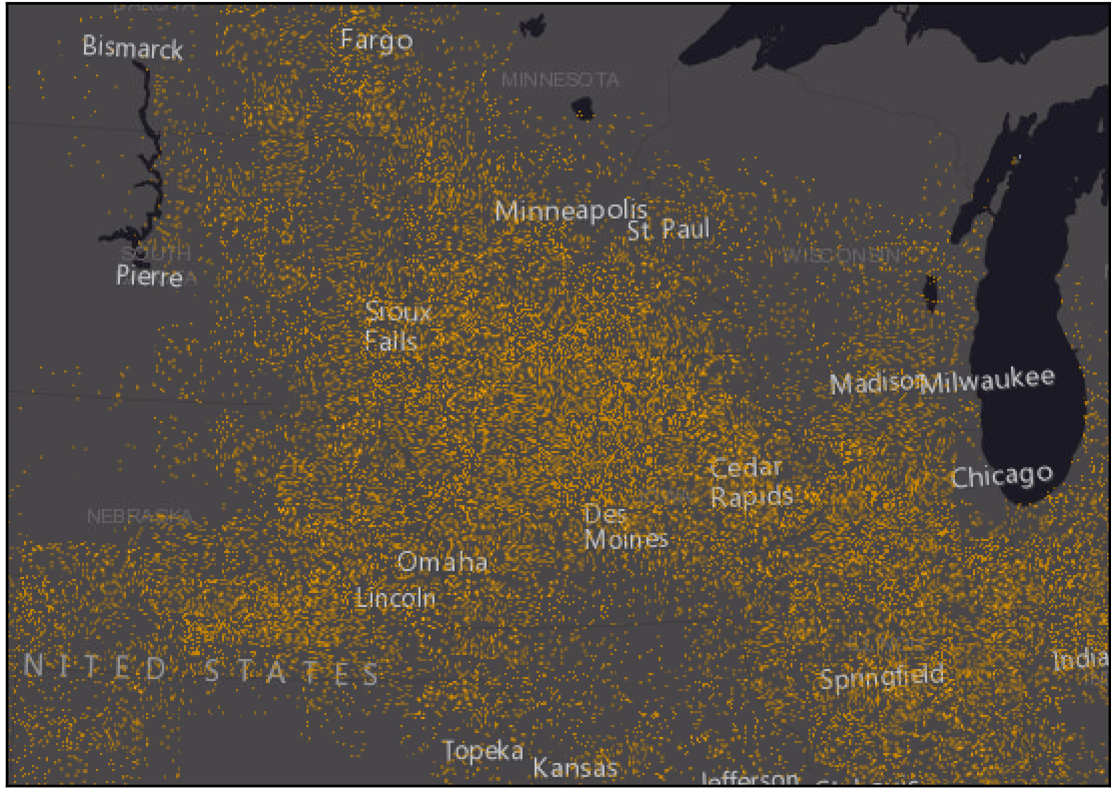
```
// temporal renderer
 var observationRenderer = new ClassBreaksRenderer(new
SimpleMarkerSymbol(), "magnitude");
 observationRenderer.addBreak(7, 12, new
SimpleMarkerSymbol(SimpleMarkerSymbol.STYLE_SQUARE, 24, new
SimpleLineSymbol().setStyle(SimpleLineSymbol.STYLE_SOLID).setColor(new
Color([100,100,100])),new Color([0,0,0,0])));

 observationRenderer.addBreak(6, 7, new
SimpleMarkerSymbol(SimpleMarkerSymbol.STYLE_SQUARE, 21, new
SimpleLineSymbol().setStyle(SimpleLineSymbol.STYLE_SOLID).setColor(new
Color([100,100,100])),new Color([0,0,0,0])));

 observationRenderer.addBreak(5, 6, new
SimpleMarkerSymbol(SimpleMarkerSymbol.STYLE_SQUARE, 18,new
SimpleLineSymbol().setStyle(SimpleLineSymbol.STYLE_SOLID).setColor(new
Color([100,100,100])),new Color([0,0,0,0])));

 observationRenderer.addBreak(4, 5, new
SimpleMarkerSymbol(SimpleMarkerSymbol.STYLE_SQUARE, 15,new
SimpleLineSymbol().setStyle(SimpleLineSymbol.STYLE_SOLID).setColor(new
Color([100,100,100])),new Color([0,0,0,0])));
```

```
 observationRenderer.addBreak(3, 4, new
SimpleMarkerSymbol(SimpleMarkerSymbol.STYLE_SQUARE, 12,new
SimpleLineSymbol().setStyle(SimpleLineSymbol.STYLE_SOLID).setColor(new
Color([100,100,100])),new Color([0,0,0,0])));

 observationRenderer.addBreak(2, 3, new
SimpleMarkerSymbol(SimpleMarkerSymbol.STYLE_SQUARE, 9,new
SimpleLineSymbol().setStyle(SimpleLineSymbol.STYLE_SOLID).setColor(new
Color([100,100,100])),new Color([0,0,0,0])));

 observationRenderer.addBreak(0, 2, new
SimpleMarkerSymbol(SimpleMarkerSymbol.STYLE_SQUARE, 6,new
SimpleLineSymbol().setStyle(SimpleLineSymbol.STYLE_SOLID).setColor(new
Color([100,100,100])),new Color([0,0,0,0])));var infos = [
   { minAge: 0, maxAge: 1, color: new Color([255,0,0])},
   { minAge: 1, maxAge: 24, color: new Color([49,154,255])},
   { minAge: 24, maxAge: Infinity, color: new Color([255,255,8])}
 ];var ager = new TimeClassBreaksAger(infos,
TimeClassBreaksAger.UNIT_HOURS);
 var renderer = new TemporalRenderer(observationRenderer, null, null,
ager);
 featureLayer.setRenderer(renderer);
```

The preceding code example illustrates how to create `TemporalRenderer` using a `ClassBreaksRenderer` and apply it to a `FeatureLayer`. The class breaks renderer is used to define symbols by magnitude; the larger the magnitude the larger the symbol.

Then a symbol `ager` is defined. The `ager` symbol determines how the feature's symbol changes as time progresses.

The final type of renderer that we'll discuss is `DotDensityRenderer`:



This renderer enables you to create dot density visualizations of data that show the concentration of discrete spatial phenomena. In the example, we're using it to display population density:

```
var dotDensityRenderer = new DotDensityRenderer({
   fields: [{
       name: "pop",
       color: new Color([52, 114, 53])
   }],
   dotValue: 1000,
   dotSize: 2
 });

 layer.setRenderer(dotDensityRenderer);
```

Here, we create `DotDensityRenderer` based on the `pop` field and define a dot value of 100 and size of 2. This will create one dot two pixels in size for each 1000 population.

There are other renderers that you might want to use for specific tasks, such as the `ScaleDependentRenderer`, which allows you to apply different symbols to features based on the scale at which they are being viewed, and the `HeatMapRenderer`, which is similar to the `DotDensityRenderer`, but uses a technique called Gaussian Blur to show the intensity of points.

Practice time

In this exercise, you will use `FeatureLayer` to set a definition expression on a layer, draw the features that match the definition expression as graphics, and respond to a hover event over the features.

Follow the given steps to complete the exercise:

1. Open the JavaScript sandbox at `https://developers.arcgis.com/javascript/3/sandbox/sandbox.html`.
2. Remove the JavaScript content from the `<script>` tag that I have highlighted, leaving an empty `<script></script>` block in which to write the required code:

```
<script>
  var map;

  require(["esri/map", "dojo/domReady!"], function(Map) {
    map = new Map("map", {
      basemap: "topo",  //For full list of pre-defined
basemaps,
      navigate to http://arcg.is/1JVo6Wd
      center: [-122.45, 37.75], // longitude, latitude
      zoom: 13
    });
  });
</script>
```

3. Create the variable that you will use to refer to the map in the `<script>` tag:

```
<script>
  var map;
</script>
```

4. Create the `require()` function that defines the resources you'll use in this application:

```
<script>
  var map;
  require(["esri/map",
  "esri/layers/FeatureLayer",
  "esri/symbols/SimpleFillSymbol",
  "esri/symbols/SimpleLineSymbol",
  "esri/renderers/SimpleRenderer",
  "esri/InfoTemplate",
  "esri/graphic",
  "dojo/on",
  "dojo/_base/Color",
  "dojo/domReady!"],
  function(Map, FeatureLayer, SimpleFillSymbol,
  SimpleLineSymbol, SimpleRenderer, InfoTemplate, Graphic, on,
Color) {
  }
  );
</script>
```

5. In your web browser, navigate to `http://sampleserver1.arcgisonline.com/ArcGIS/rest/services/Demographics/ESRI_Census_USA/MapServer/5`.

We will be using the `states` layer for this exercise. What we want to do is apply a definition expression to the `states` layer that will display only those states that have a median age greater than `36`. Those states with a median age greater than `36` will be displayed as graphics on the map, and an info window will be displayed containing the median age, median age for males, and the median age for females for that state when the user hovers the mouse over the states that meet the definition expression. In addition, the state boundary will be outlined in red. The fields we will use from the `states` layer include `STATE_NAME, MED_AGE, MED_AGE_M, and MED_AGE_F`.

6. Create the `Map` object as shown here:

```
function (Map, FeatureLayer, ...) {
    map = new Map("map", {
        basemap: "streets",
        center: [-96.095,39.726], // long, lat
        zoom: 4,
        sliderStyle: "small"
    });
}
```

7. Add a `Map.load` event handler that, when fired, triggers the creation of a `Map.Graphics.mouse-out` event handler that, in turn, clears any existing graphics and info windows:

```
map = new Map("map", {
    basemap: "streets",
    center: [-96.095, 39.726], // long, lat
    zoom: 4,
    sliderStyle: "small"
});

map.on("load", function () {
    map.graphics.on("mouse-out", function (evt) {
        map.graphics.clear();
        map.infoWindow.hide();
    });
});
```

8. Create a new `FeatureLayer` that points to the `states` layer you examined earlier. Set the feature retrieval mode to `MODE_SNAPSHOT`, define the output fields, and set the definition expression. Add the following code to your application:

```
map.on("load", function() {
     map.graphics.on("mouse-out", function(evt) {
       map.graphics.clear();
       map.infoWindow.hide();
     });
   });

var olderStates = new
FeatureLayer("http://sampleserver1.arcgisonline.com/ArcGIS/rest
/services/Demographics/ESRI_Census_USA/MapServer/5", {
    mode: FeatureLayer.MODE_SNAPSHOT,
    outFields: ["STATE_NAME", "MED_AGE", "MED_AGE_M",
"MED_AGE_F"]
});
olderStates.setDefinitionExpression("MED_AGE > 36");
```

Here, we used the `new` keyword to define a new instance of `FeatureLayer` that points to the `states` layer at the REST endpoint specified in the code. When defining a new instance of `FeatureLayer`, we included a couple of properties including the `mode` and `outFields`. The `mode` property can be either a snapshot, on-demand, or selection-only. Since the states layer contains a relatively small number of features, we can use the snapshot mode. The snapshot mode retrieves all features from the layer when it is added to the map, and therefore does not require any additional trips to the server. We are also specifying the `outFields` property, which is an array of fields that will be returned. We will be displaying these fields in an info window when the user hovers over the state. Finally, we set our definition expression on the layer to display only those features (states) where the median age is greater than `36`.

9. In this step, you create a symbol and apply `renderer` to the states features. You will also add the `FeatureLayer` to the map. Add the following lines of code just after the code you added in the last step:

```
var olderStates = new FeatureLayer("http://...", {
    mode: FeatureLayer.MODE_SNAPSHOT,
    outFields: ["STATE_NAME", "MED_AGE", "MED_AGE_M",
"MED_AGE_F"]
});
olderStates.setDefinitionExpression("MED_AGE > 36");

var symbol = new SimpleFillSymbol(SimpleFillSymbol.STYLE_SOLID,
    new SimpleLineSymbol(SimpleLineSymbol.STYLE_SOLID,
    new Color([255, 255, 255, 0.35]), 1),
    new Color([125, 125, 125, 0.35]));
olderStates.setRenderer(new SimpleRenderer(symbol));

map.addLayer(olderStates);
```

10. Using the output fields you defined earlier, create an `InfoTemplate`. Add the following lines of code to your application just after the lines you created in the last step. Note the inclusion of the output fields that are embedded inside brackets and preceded by a dollar sign:

```
var infoTemplate = new InfoTemplate();
infoTemplate.setTitle("${STATE_NAME}");
infoTemplate.setContent("<b>Median Age: </b>${MED_AGE_M}<br/>"
    + "<b>Median Age - Male: </b>${MED_AGE_M}<br/>"
    + "<b>Median Age - Female: </b>${MED_AGE_F}");
map.infoWindow.resize(245, 125);
```

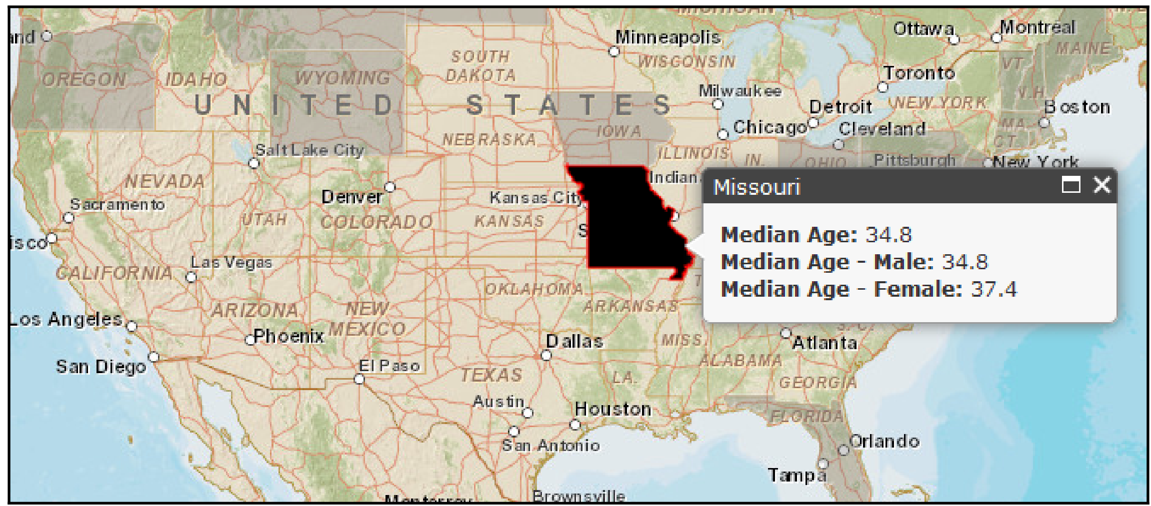
11. Add the following lines of code to create a graphic that will be displayed when the user hovers the mouse over a state:

```
var highlightSymbol = new
SimpleFillSymbol(SimpleFillSymbol.STYLE_SOLID,
    new SimpleLineSymbol(SimpleLineSymbol.STYLE_SOLID,
        new Color([255, 0, 0]),
        new Color([125, 125, 125, 0.35])
    )
);
```

12. The final step is to display the highlight and info template that we created in the last couple steps each time the user hovers over a state. Add the following code block after the last lines of code you entered, and then we'll discuss. Here, we are using `on()` to wire the `mouse-over` event to a handler function that will respond each time the event occurs. The `mouse-over` event handler, in this case, will clear any existing graphics from the `GraphicsLayer`, create the info template, create a highlighted symbol and add it to `GraphicsLayer`, and then show `InfoWindow`:

```
olderStates.on("mouse-over", function (evt) {
    map.graphics.clear();
    evt.graphic.setInfoTemplate(infoTemplate);
    var content = evt.graphic.getContent();
    map.infoWindow.setContent(content);
    var title = evt.graphic.getTitle();
    map.infoWindow.setTitle(title);
    var highlightGraphic = new Graphic(evt.graphic.geometry,
highlightSymbol);
    map.graphics.add(highlightGraphic);
    map.infoWindow.show(evt.screenPoint,
map.getInfoWindowAnchor(evt.screenPoint));
});
```

13. You may want to review the solution file (`featurelayer.html`) in the `Chapter4` folder of the sample code to verify that your code has been written correctly. Execute the code by clicking on the **Refresh** button and you should see the following output if everything has been coded correctly. You should see a map similar to the figure as follows. Move your mouse over one of the highlighted states to see the info window:



Summary

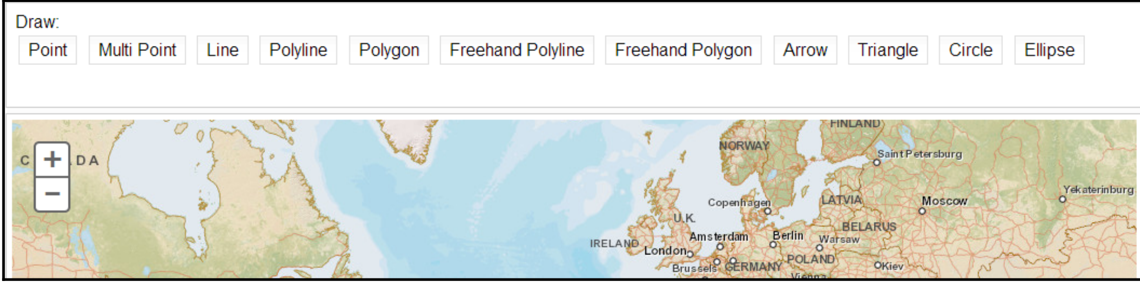
The `FeatureLayer` class is used for working with client-side graphics features. The `FeatureLayer` inherits from the `GraphicsLayer`, but it also offers additional capabilities such as the ability to perform queries and selections and supports definition expressions. The `FeatureLayer` can also be used for web editing. A `FeatureLayer` differs from tiled and dynamic map service layers because the feature layers bring geometry and attribute information across to the client computer to be drawn by the web browser. This can reduce the round trips to the server, and thereby improve the performance of your application. A client can request the features it needs, and perform selections and queries on those features without having to request more information from the server. The `FeatureLayer` is especially useful for layers that respond to user interactions such as mouse clicks or hovers. The ability to handle all that in the browser without making multiple requests of the server makes your application very responsive.

5
Using Widgets and Toolbars

As a GIS developer, your main objective is to build functionality specific to your application. Spending valuable time and effort implementing basic GIS tools that are common to many web mapping applications detracts from what should be your primary focus.

Fortunately, the API provides user interface widgets that you can drop directly into your application, and with a little configuration they are ready to go. These include map navigation tools, overview maps, legends, scale bars, and so on. The API also includes helper classes to help you build your own navigation and drawing toolbars. In this chapter you'll learn how easy it is to add these user interface components to an application.

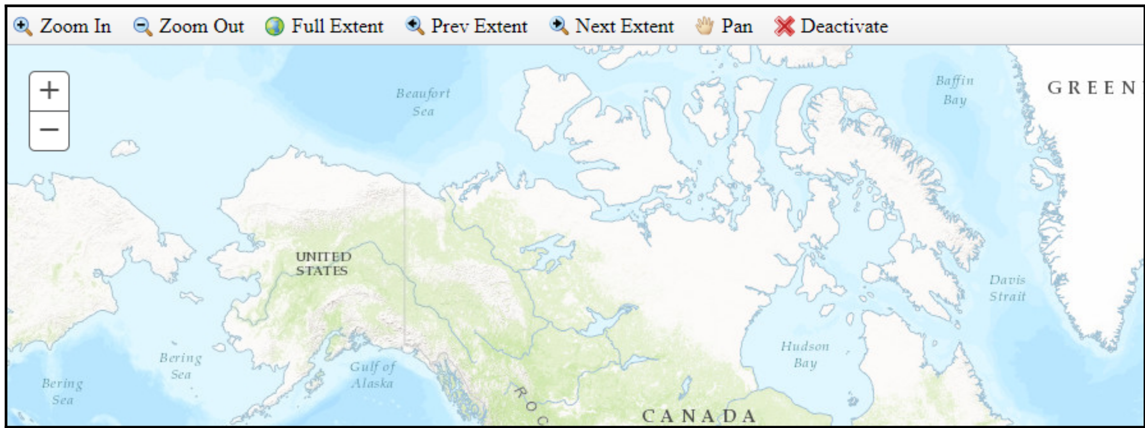
Let's start by examining one of the Esri sample applications that feature the **Draw** toolbar. Open a web browser and go to `https://developers.arcgis.com/javascript/3/samples/toolbar_draw/`:



At first glance you'd think that the Draw toolbar is simply a user interface component that you drop into your application. Not quite. Instead, it provides the functionality around which you can design your own interface or toolbar.

The helper class the API gives you is called `esri/toolbars/Draw`.

In this chapter, we'll have a look at another toolbar: `esri/toolbars/Navigation`, which you can see in the following action:



What both these helper classes do is save you the work of drawing zoom boxes, capturing mouse clicks and other user-initiated events. As any experienced GIS web developer can tell you, this is no trivial task. Incorporating these basic navigation capabilities in the helper classes provided with the API can easily save dozens of hours of development work.

In this chapter we'll cover the following topics:

- Adding toolbars to an application
- User interface widgets
- Feature editing

Adding toolbars to an application

There are four different toolbars that you can add to your application using helper classes provided by the API. The two we'll talk about in this chapter are Navigation and Draw. There is also an editing toolbar that can be used to edit features or graphics through a web browser. We'll discuss this toolbar in a later chapter. Bear in mind that when we talk about "toolbars" here, we are referring to the helper classes that assist you in building such toolbars, and not the toolbars themselves. The look and feel of your toolbar is down to you.

Note that the API includes another toolbar which provides support for measuring image services. That is beyond the scope of this book.

Steps for creating a toolbar

The navigation and draw toolbars are not simply user interface components that you can drop into your application. They are helper classes and there are several steps that you need to take to actually create your toolbar with the appropriate buttons. This "to do" list for your toolbars may seem a little intimidating but once you've done it once or twice it becomes pretty simple. The steps are listed in the following and we'll discuss each item in detail:

1. Define the CSS styles for each button.
2. Create the Buttons inside the toolbar.
3. Create an instance of `esri/toolbars/Navigation` or `esri/toolbars/Draw`.
4. Connect button events to handler functions.

Defining CSS styles

The first thing you'll need to do is define the CSS styles for each button that you intend to include on the toolbar. Each button on your toolbar will need an image, some text, or both along with a dimension for the button that you will specify within a `<style>` tag - either inline or linking out to a separate CSS file.

The following code shows the CSS definition of the buttons required for a navigation toolbar. Let's examine the **Zoom Out** button and follow it through the entire process to make things a little simpler. I've highlighted the **Zoom Out** button in the following code. As with all the other buttons we define the CSS selector to identify the button (in this case, all DOM nodes are decorated with the `zoomXXXIcon` class), an image to use for the button (`nav_zoomout.png`), and the width and height of the button:

```
<style>
@import "https://js.arcgis.com/3.21/dijit/themes/claro/claro.css";
        html,
        body,
        #map {
            height: 100%;
            margin: 0;
            padding: 0;
```

```
        }
        .zoominIcon {
            background-image: url(images/nav_zoomin.png);
            width: 16px;
            height: 16px;
        }

        .zoomoutIcon {
            background-image: url(images/nav_zoomout.png);
            width: 16px;
            height: 16px;
        }

        .zoomfullextIcon {
            background-image: url(images/nav_fullextent.png);
            width: 16px;
            height: 16px;
        }

        .zoomprevIcon {
            background-image: url(images/nav_previous.png);
            width: 16px;
            height: 16px;
        }

        .zoomnextIcon {
            background-image: url(images/nav_next.png);
            width: 16px;
            height: 16px;
        }

        .panIcon {
            background-image: url(images/nav_pan.png);
            width: 16px;
            height: 16px;
        }

        .deactivateIcon {
            background-image: url(images/nav_deactivate.png);
            width: 16px;
            height: 16px;
        }
</style>
```

Creating the buttons

You can lay out the buttons by using a `<div>` container with a `data-dojo-type` of `ContentPane` inside a `BorderContainer` as seen in the following code example. Each button references the appropriate CSS style the `iconClass` attribute.

In the case of the **Zoom Out** button in our example the `iconClass` attribute references `zoomoutIcon` which is the CSS style we defined earlier:

```
<body class="claro">
 <div id="navToolbar" data-dojo-type="dijit/Toolbar">
   <div data-dojo-type="dijit/form/Button" id="zoomin"
      data-dojo-props="iconClass:'zoominIcon'">Zoom In</div>
   <div data-dojo-type="dijit/form/Button" id="zoomout"
      data-dojo-props="iconClass:'zoomoutIcon'">Zoom Out</div>
   <div data-dojo-type="dijit/form/Button" id="zoomfullext"
      data-dojo-props="iconClass:'zoomfullextIcon'">Full Extent</div>
   <div data-dojo-type="dijit/form/Button" id="zoomprev"
      data-dojo-props="iconClass:'zoomprevIcon'">Prev Extent</div>
   <div data-dojo-type="dijit/form/Button" id="zoomnext"
      data-dojo-props="iconClass:'zoomnextIcon'">Next Extent</div>
   <div data-dojo-type="dijit/form/Button" id="pan"
      data-dojo-props="iconClass:'panIcon'">Pan</div>
   <div data-dojo-type="dijit/form/Button" id="deactivate"
      data-dojo-props="iconClass:'deactivateIcon'">Deactivate</div>
 </div>

 <div id="map"></div>
</body>
```

The preceding code block defines the buttons on the toolbar. Each button is created using a `Button` user interface control provided in the Dojo `dijit` namespace. All buttons are enclosed by a `<div>` of Dojo type `ContentPane`, which are in turn hosted by a Dojo `BorderContainer`.

Because we have declared these dijits in markup (using the `data-dojo-type` attribute) instead of programmatically, they are not created automatically when the page loads. The browser only understands how to build basic HTML elements, not those that Dojo supplies. It's Dojo's job to create those elements, which it does using the `dojo/parser`. When the page has finished loading, our Dojo code executes and calls `parser.parse()` to instantiate any dijits we've defined in the markup.

Creating an instance of the Navigation toolbar

Now that the visual interface for the buttons is complete, we need to create an instance of `esri/toolbars/Navigation` and wire up the events and event handlers. Creating an instance of the `Navigation` class is as easy as calling the constructor and passing in a reference to `Map` as seen in the following code example.

However, you'll first want to make sure that you add a reference to `esri/toolbars/Navigation`. The following code example adds references to the Navigation toolbar, creates the toolbar, connects click events to the buttons, and activates the tools. The relevant lines of code have been highlighted and commented so that you understand each section.

```
<script>
var map;
require([
  "esri/map",
  "esri/toolbars/navigation",
  "dojo/on",
  "dojo/parser",
  "dijit/registry",
  "dijit/Toolbar",
  "dijit/form/Button",
  "dojo/domReady!"
],
function (Map, Navigation, on, parser, registry) {
  parser.parse();
  var navToolbar;
  map = new Map("map", {
    basemap: "topo",
    center: [-56.953, 57.472],
    zoom: 3
  });
  navToolbar = new Navigation(map);
  on(navToolbar, "onExtentHistoryChange", extentHistoryChangeHandler);
  registry.byId("zoomin").on("click", function () {
    navToolbar.activate(Navigation.ZOOM_IN);
  });
  registry.byId("zoomout").on("click", function () {
    navToolbar.activate(Navigation.ZOOM_OUT);
  });
  registry.byId("zoomfullext").on("click", function () {
    navToolbar.zoomToFullExtent();
  });
  registry.byId("zoomprev").on("click", function () {
    navToolbar.zoomToPrevExtent();
  });
```

```
    registry.byId("zoomnext").on("click", function () {
      navToolbar.zoomToNextExtent();
    });
    registry.byId("pan").on("click", function () {
      navToolbar.activate(Navigation.PAN);
    });
    registry.byId("deactivate").on("click", function () {
      navToolbar.deactivate();
    });
    function extentHistoryChangeHandler () {
      registry.byId("zoomprev").disabled = navToolbar.isFirstExtent();
      registry.byId("zoomnext").disabled = navToolbar.isLastExtent();
    }
  });
  </script>
```

This example has demonstrated the steps to add a navigation toolbar to your web mapping application. By using the `esri/toolbars/Navigation` class, you avoid having to write your own code to draw and handle the extent rectangle for zooming in or out, or any other basic navigation functionality.

True, it's down to you to build the actual interface, so *toolbar* in this instance is a bit of a misnomer. It's more of a *toolbar helper*. However, as we have seen, it's pretty easy to create a nice user interface by using the different controls exposed by Dojo's Dijit library.

The `/esri/toolbars/Draw` class we mentioned at the beginning of this chapter provides similar assistance for creating tools that allow users to draw points, lines, and polygons on the map. The process for creating the Draw toolbar is very similar.

User interface widgets

The API for JavaScript comes with many out of the box widgets that you can drop into your application to add extra functionality without having to code it yourself.

User interface widgets all live in the `esri/dijit` namespace and include widgets for selecting basemaps, bookmarking map locations, printing maps, geocoding, measuring, adding legends, scalebars, overview maps and other map elements, and many, many more.

Widgets differ from the toolbars we have seen so far in this chapter, because they require less work from you, the developer. This is because, unlike toolbars, they provide not just the functionality, but the user interface as well.

For a full list of widgets, consult the ArcGIS API for JavaScript documentation at: `https://developers.arcgis.com/javascript/3/jshelp/`.

The BasemapGallery widget

The `BasemapGallery` widget displays a collection of basemaps from ArcGIS.com and/or a user-defined set of map or image services. When a basemap is selected from the collection, the current basemap is removed and the newly selected basemap appears. When adding your own basemaps to the basemap gallery, you must ensure that they have the same spatial reference as the other layers in the gallery. When using layers from ArcGIS.com this would be the Web Mercator reference with WKIDs of 102100, 102113, or 3857. While it's possible to include dynamic map services in your basemap gallery, we recommend using tiled layers for best performance.

When creating a `BasemapGallery` there are a number of parameters that you can supply to the constructor including the ability to show ArcGIS basemaps, define one or more custom basemaps for inclusion in the gallery, and a reference to the map where the gallery will be placed.

After creating the `BasemapGallery` widget you need to call the `startup()` method to prepare it for user interaction. This is required for all widgets that you declare programmatically.

If your widget doesn't appear, make sure that you have called its `startup()` method.

```
var basemapGallery = new BasemapGallery({
    showArcGISBasemaps: true,
    map: map
}, "basemapGallery");
basemapGallery.startup();
```

The preceding example creates a basemap gallery using existing basemaps from `arcgis.com`.

If you want to include your own basemaps, you must create an instance of the `Basemap` class:

```
require([
  "esri/dijit/Basemap", ...
], function(Basemap, ... ) {
var myBasemaps = [];
var myBasemap = new Basemap({
layers: [myLayer1],
title: "My Layer",
thumbnailUrl: "images/myThumb.png"
  });
myBasemaps.push(myBasemap);
  ...
});
```

In the preceding code sample a new `Basemap` object is created with a title, thumbnail image, and an array containing a single layer. This `Basemap` is then pushed into an array. You can then add this array of `Basemap` objects to the `BasemapGallery`'s basemaps property:

```
var basemapGallery = new BasemapGallery({
    showArcGISBasemaps: true,
    basemaps: myBasemaps
```

```
    map: map
}, "basemapGallery");
basemapGallery.startup();
```

Basemap toggle widget

If you want to provide your users with just a choice of two basemaps, and the ability to toggle between them, consider the `BasemapToggle` widget.

Just specify the default basemap for your map in the usual way, and then the alternative basemap as part of the `BasemapToggle` definition:

```
require([
  "esri/map",
  "esri/dijit/BasemapToggle",
  "dojo/domReady!"
], function(
  Map, BasemapToggle
)  {

  map = new Map("map", {
    center: [-70.6508, 43.1452],
    zoom: 16,
    basemap: "topo"
  });
  var toggle = new BasemapToggle({
    map: map,
    basemap: "satellite"
  }, "BasemapToggle");
  toggle.startup();
```

Bookmarks widget

The `Bookmarks` widget is used to display a set of named geographic extents (bookmarks) to the end user. Clicking a bookmark in the widget will automatically set the extent of the map to that extent. The widget lets you add new bookmarks, delete existing bookmarks, and update bookmarks. Bookmarks are defined in JavaScript code as **JSON** (**JavaScript Object Notation**) objects. Their properties define the name, extent, and bounding coordinates of the bookmark. To add a bookmark to the widget you call `Bookmark.addBookmark()`:

```
require([
  "esri/map", "esri/dijit/Bookmarks", "dojo/dom", ...
], function(Map, Bookmarks, dom, ... ) {
var map = new Map( ... );
var bookmarks = new Bookmarks({
map: map,
bookmarks: bookmarksJSON
  }, dom.byId('bookmarks'));
  ...
});
```

In the preceding code example a new `Bookmarks` object is created. It is attached to the map, and a list of bookmarks in JSON format is added.

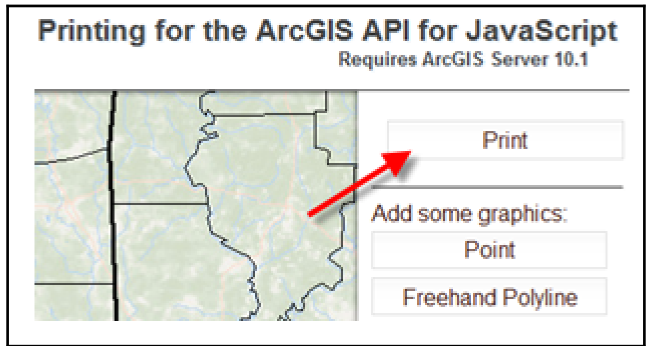
The following shows an extract from the JSON used to create the bookmarks:

```
var bookmarkJSON = {
    first: {
        "extent": {
            "xmin": -12975151.579395358,
            "ymin": 3993919.9969406975,
            "xmax": -12964144.647322308,
            "ymax": 4019507.292159126,
            "spatialReference": {
                "wkid": 102100,
```

```
                "latestWkid": 3857
            }
        },
        name": "Palm Springs, CA"
    },
    second: {
        "extent": {
            ...
        },
        "name": "Redlands, California"
    },
    third: {
        "extent": {
            ...
        },
        "name": "San Diego, CA"
    },
};
```

The Print widget

The `Print` widget is a much-welcomed tool that simplifies printing maps from web applications. It uses a default or user-defined layout for the map. This widget requires the use of an ArcGIS Server 10.1 or higher export web map task:




```
require([
  "esri/map", "esri/dijit/Print", "dojo/dom"...
], function(Map, Print, dom, ... ) {
var map = new Map( ... );
var printer = new Print({
     map: map,
     url:
"http://servicesbeta4.esri.com/arcgis/rest/services/Utilities/ExportWebMap/
```

```
GPServer
        /Export%20Web%20Map%20Task"
  }, dom.byId("printButton"));
  ...
});
```

The preceding code example creates a new `Print` widget. The `URL` property is used to point the widget to an ArcGIS Server export web map task and the widget is attached to an HTML element on the page.

Layer List widget

For many years, Esri resisted incorporating a Table of Contents control into the ArcGIS API for JavaScript. The premise was that they did not want to replicate the functionality of their desktop software on the web, because the end users of ArcGIS API for JavaScript applications would not be familiar with the GIS tools that practitioners use.

However, it appears that non-GIS users are equally happy to have a quick method for turning layers on and off in the map display, and so Esri finally gave us the `LayerList` control.

Time to practice

Let's practice using widgets by implementing the `LayerList` control.

Open the ArcGIS API for JavaScript Sandbox web page at `https://developers.arcgis.com/javascript/3/sandbox/sandbox.html`.

Leave the existing code intact, but change the initial map coordinates to longitude `-85.700` and latitude `38.240`, with a zoom level of `13`:

```
<script>
  var map;

  require(["esri/map", "dojo/domReady!"], function(Map) {
    map = new Map("map", {
      basemap: "topo",  //For full list ...
      center: [-85.700, 38.240], // longitude, latitude
      zoom: 13
    });
  });
</script>
```

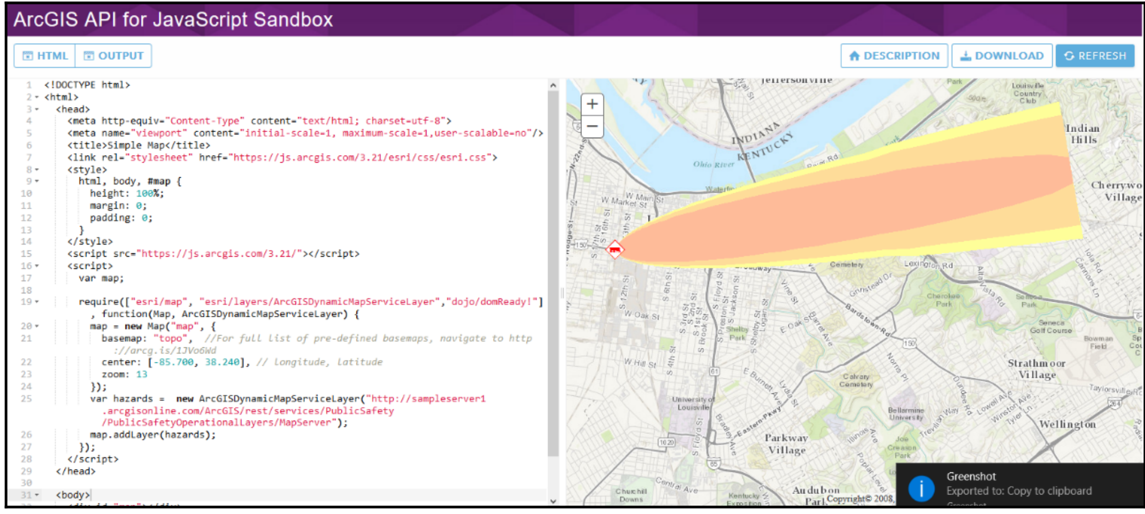
This centers the map on downtown Lousiville in Kentucky, US.

To give us a layer with some interesting sublayers to work within our layer list, we'll add the "Public Safety" layer from `http://sampleserver1.arcgisonline.com/ArcGIS/rest/services/PublicSafety/PublicSafetyOperationalLayers/MapServer`.

We'll need to add this as an `ArcGISDynamicMapServiceLayer`, because it is not a cached layer:

```
<script>
  var map;

  require(["esri/map",
"esri/layers/ArcGISDynamicMapServiceLayer","dojo/domReady!"], function(Map,
ArcGISDynamicMapServiceLayer) {
    map = new Map("map", {
      basemap: "topo",  //For full list of pre-defined basemaps,
navigate to http://arcg.is/1JVo6Wd
      center: [-85.700, 38.240], // longitude, latitude
      zoom: 13
    });
    var hazards =  new
ArcGISDynamicMapServiceLayer("http://sampleserver1.arcgisonline.com/ArcGIS/
rest/services/PublicSafety/PublicSafetyOperationalLayers/MapServer");
    map.addLayer(hazards);
  });
</script>
```

Check that everything is working fine:



Now, let's implement our `LayerList` widget. First, import the necessary modules:

```
require(["esri/map", "esri/layers/ArcGISDynamicMapServiceLayer",
"esri/dijit/LayerList", "dojo/domReady!"],
function(Map, ArcGISDynamicMapServiceLayer, LayerList) {
        map = new Map("map", { ...
```

Then reference the required CSS for the `LayerList` widget, so that it knows how to display. Let's use the `Claro` Dojo theme for this:

```
<link rel="stylesheet"
href="https://js.arcgis.com/3.21/dijit/themes/claro/claro.css">
    <script src="https://js.arcgis.com/3.21/"></script>
    <script>
      var map;

      require(["esri/map",
 ...
    </script>
  </head>

  <body class="claro">
    <div id="map"></div>
  </body>
```

Now we need to provide a `<div>` on the page in which to place the `LayerList` widget. Replace the contents of the `<body>` with the following:

```
  <body class="claro">
    <div class="container" data-dojo-type="dijit/layout/BorderContainer"
       data-dojo-props="design:'headline',gutters:false">
        <div id="layerListPane" data-dojo-type="dijit/layout/ContentPane"
       data-dojo-props="region:'right'">
            <div id="layerList"></div>
        </div>
        <div id="map" data-dojo-type="dijit/layout/ContentPane" data-dojo-
props="region:'center'"></div>
    </div>
  </body>
```

Because we're using Dojo markup here, which your web browser's rendering engine will just ignore, we have to tell Dojo to parse these elements for us when the page has loaded:

```
      require(["esri/map",
               "esri/layers/ArcGISDynamicMapServiceLayer",
               "esri/layers/ArcGISTiledMapServiceLayer",
               "esri/dijit/LayerList",
```

```
                "dojo/parser",
                "dojo/domReady!"],
        function(Map, ArcGISDynamicMapServiceLayer,
ArcGISTiledMapServiceLayer, LayerList, parser) {
        parser.parse();
        map = new Map("map", {
            basemap: "topo",
            center: [-85.700, 38.240], // longitude, latitude
            zoom: 13
        });
```

Now we need to add some CSS to make sure everything gets put in the right place. Replace the contents of the existing `<style>` tag with the following:

```
<style>
    html, body, .container, #map {
        height:100%;
        width:100%;
        margin:0;
        padding:0;
        margin:0;
        font-family: "Open Sans";
    }
    #map {
        padding:0;
    }
    #layerListPane{
        width:25%;
    }
</style>
```

Now we've got everything set up, we can add the `LayerList` widget to the map:

```
var layerList = new LayerList({
    map: map,
    layers: [{
        layer: hazards,
        id: "Hazards",
        showSubLayers: true
    }],
    showOpacitySlider: true
},"layerList");
layerList.startup();
```

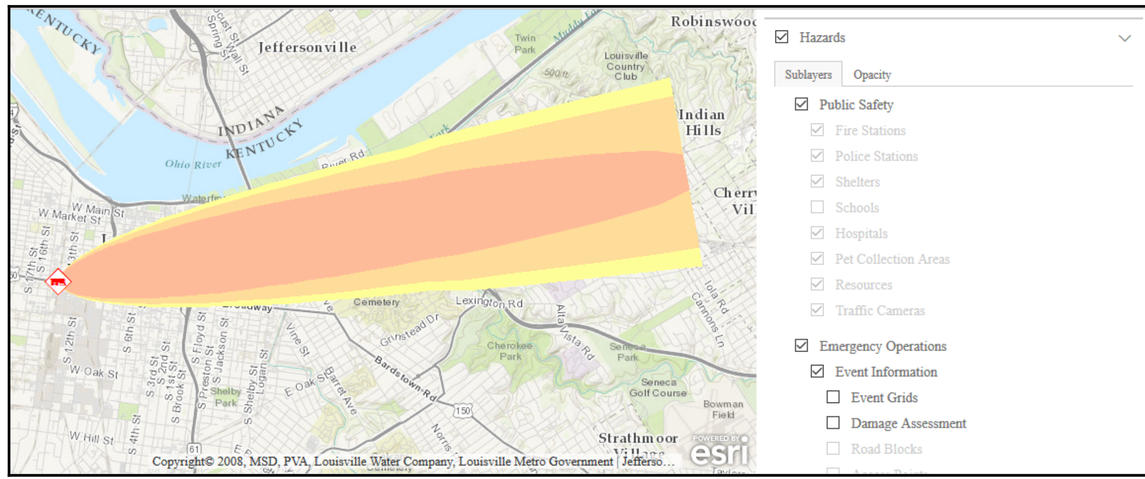
In the preceding code, we created an instance of the `LayerList` widget and passed it an options object. The most important options here are for the map, for which we want to display the layer list and layers. The `layers` property accepts an array of objects that represent the layers we want to include in the list. The properties we are setting for each layer are:

- `layer`: the layer itself
- `id`: the name of the layer as it should appear in the `LayerList` control
- `showSubLayers`: whether we want to show the individual layers that make up this map service as separate layers in the list

You'll also see that we're specifying the `showOpacitySlider` option on the `LayerList` control. This lets us present a slider control for each layer so that the user can adjust the transparency of the layer dynamically at runtime.

If we only wanted transparency for certain layers, we can use the `showOpacitySlider` option in the individual layer objects to override the behavior of the layer list control. Conversely, if we wanted to show or hide sublayers for all layers in the layer list, we could use this property on the control rather than in the individual layers.

Click the **Refresh** button in the Sandbox and ensure that the layer list displays correctly and that you are able to expand it to see the sub-layers. Note that some of the layers are grayed out until you zoom into the map. Enable and disable layers and verify that they are displayed or removed from the map accordingly. Then, switch to the opacity tab and adjust the transparency of the **Public Safety** map service:



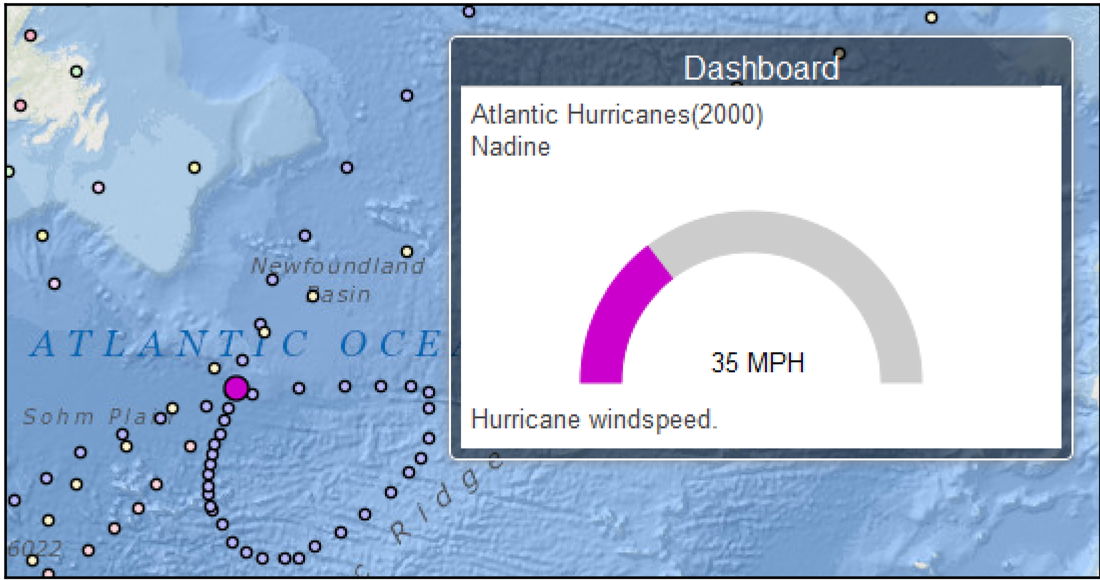
Search widget

The `Search` widget allows you to easily add geocoding functionality to your application. When you type the details of a location into the text box, the `Search` widget attempts to locate it on the map.

By default, the `Search` widget uses the ArcGIS Online World Geocoding service, but you can customize it to use your own sources. We'll talk about this widget in greater detail in `Chapter 8`, *Turning Addresses into Points and Points into Addresses*.

Gauge widget

The Gauge widget displays numeric data from a `FeatureLayer` or `GraphicsLayer` in a semi-circular gauge interface. You can define the attribute field to use for the numeric data that drives the gauge, a label field, the layer to reference, a maximum data value, the title and color for the gauge indicator, and more:



The following code demonstrates how to create a `Gauge` widget, using attribute data from a `FeatureLayer` (referred to in the code as `flHurricanes`):

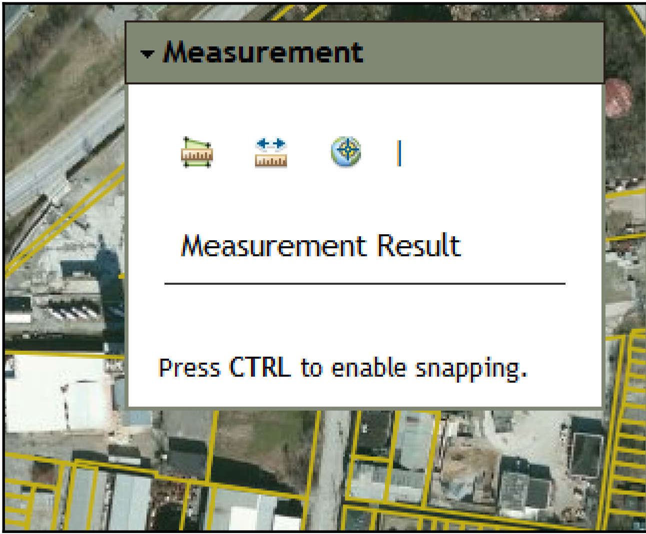
```
require([
  "esri/dijit/Gauge", ...
], function(Gauge, ... ) {
```

```
var gaugeParams = {
    "caption": "Hurricane windspeed.",
    "color": "#c0c",
    "dataField": "WINDSPEED",
    "dataFormat": "value",
    "dataLabelField": "EVENTID",
    "layer": flHurricanes,
    "maxDataValue": 120,
    "noFeatureLabel": "No name",
    "title": "Atlantic Hurricanes(2000)",
    "unitLabel": "MPH"
  };
var gauge = new Gauge(gaugeParams, "gaugeDiv");
  ...
});
```

A number of parameters are being passed into the constructor for `Gauge` including a caption, color, data field, layer, max data value, and others.

Measurement widget

The `Measurement` widget provides three tools that enable the end user to measure length and area as well as get the current mouse coordinates:



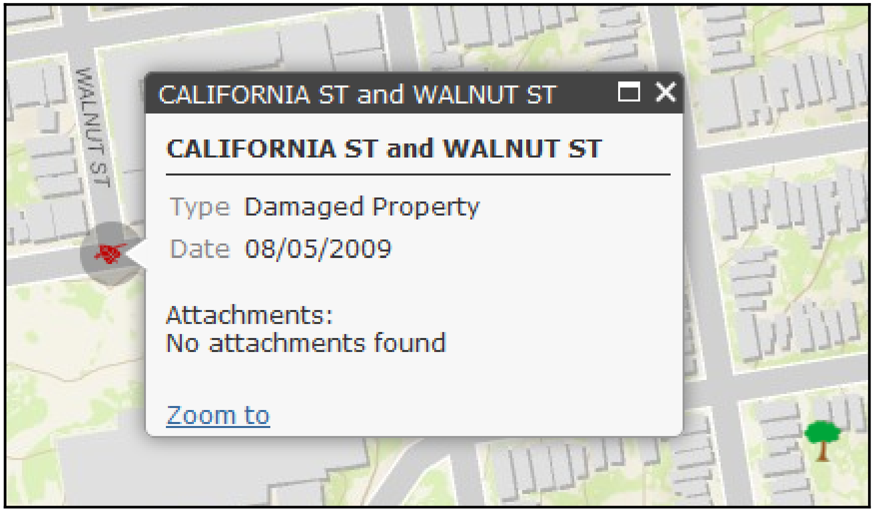
This widget also allows you to change the units of measurement:

```
var measurement = new Measurement({
  map: map
}, dom.byId("measurementDiv"));
measurement.startup();
```

The preceding code example shows how to create an instance of the `Measurement` widget and add it to the application.

The Popup widget

The `Popup` widget is functionally similar to the default Info Window in that it is used to display attribute information about features or graphics. In fact, starting at version 3.4 of the API this widget is now the default window for displaying attributes instead of the `InfoWindow`. However, it also contains additional functionality such as the ability to zoom to and highlight features, handling of multiple selections, and a maximize window button. The interface can also be styled using CSS. Please refer to the following screenshot for an example of the content that can be displayed in the `Popup` widget:



Starting at version 3.4, the Popup widget supports rendering text in a **right-to-left** (**RTL**) orientation to support RTL languages like Hebrew and Arabic. RTL support will automatically apply if the page direction is set to RTL using the `dir` attribute. The default value is **left-to-right** (**LTR**):

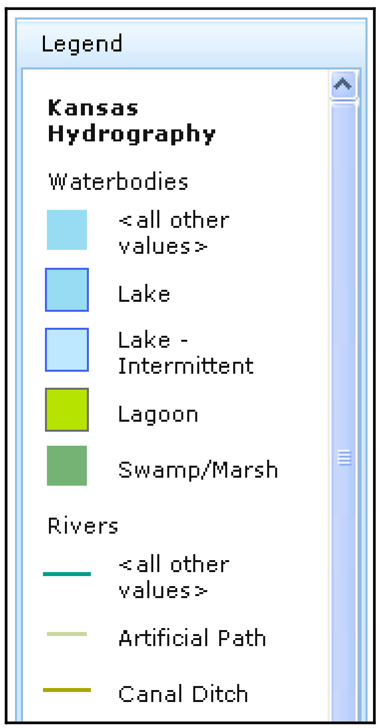
```
//define custom popup options
var popupOptions = {
  markerSymbol: new SimpleMarkerSymbol("circle", 32, null, new Color([0, 0,
0, 0.25])),
  marginLeft: "20",
  marginTop: "20"
};
//create a popup to replace the map's info window
var popup = new Popup(popupOptions, dojo.create("div"));
map = new Map("map", {
  basemap: "topo",
  center: [-122.448, 37.788],
  zoom: 17,
  infoWindow: popup
});
```

In the preceding code example a JSON `popupOptions` object is created to define the symbol, and margin of the popup. This `popupOptions` object is then passed into the constructor for the `Popup` object. Finally, the `Popup` object is passed into the `infoWindow` parameter which specifies that the `Popup` object should be used as the info window.

Legend widget

The Legend widget displays a label and symbols for some or all layers in the map. By default, it respects scale dependencies so that as you zoom in or out in the application the legend updates to reflect layer visibility at various scale ranges.

The Legend widget supports `ArcGISDynamicMapServiceLayer`, `ArcGISTiledMapServiceLayer`, and `FeatureLayer`, as well as some specialist layers like `CSVLayer` and `WMSLayer`, which we don't go into in this book:



When creating a new instance of the `Legend` widget you can specify various parameters that control the contents and display characteristics of the legend. The `arrangement` parameter can be used to specify the alignment of the legend within its container HTML as either left-aligned or right-aligned. The `autoUpdate` property can be set to `true` or `false` and if `true`, the legend will automatically update the legend if the map scale changes or when layers are added or removed from the map. The `layerInfos` parameter is used to specify a subset of layers to use in the legend, and `respectCurrentMapScale` can be set to `true` to trigger automatic legend updates based on scale ranges for each layer. Finally, you need to call the `startup()` method to display the newly created legend:

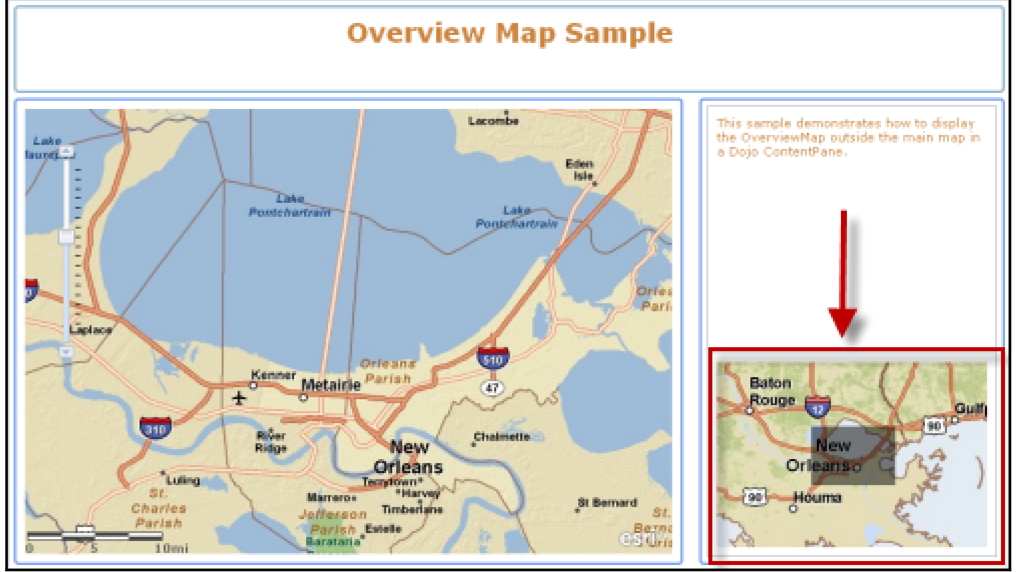
```
var layerInfo = dojo.map(results, function(layer,index){
   return {
     layer: layer.layer,
     title: layer.layer.name
   };
```

```
});
if(layerInfo.length > 0){
  var legendDijit = new Legend({
    map: map,
    layerInfos: layerInfo
  },"legendDiv");
  legendDijit.startup();
}
```

The preceding code example shows how to create a `Legend` widget and add it to an application.

OverviewMap widget

The `OverviewMap` widget is used to display the current extent of the main map within the context of a larger area. This overview map updates each time the main map extent changes. The extent of the main map is represented as a rectangle in the overview map. This extent rectangle can also be dragged to change the extent of the main map. An overview map can be displayed in a corner of the main map and also hidden from display when not in use. It can also be placed inside a `<div>` element outside the main map window or temporarily maximized for easy access to far away areas of interest:



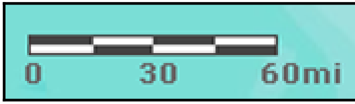
The `OverviewMap` widget takes a number of optional parameters in the constructor for the object. These parameters allow you to control things like where the overview map is placed in relation to the main map, the base layer to use for the overview map, the fill color for the extent rectangle, the appearance of a maximize button, and the initial visibility of the overview map:

```
var overviewMapDijit = new OverviewMap({map:map, visible:true});
overviewMapDijit.startup();
```

The preceding code example illustrates the creation of an `OverviewMap` widget.

Scalebar widget

The `Scalebar` widget is used to add a `Scalebar` to the map or a specific HTML node. The `Scalebar` displays units in either English or metric values. As of version 3.4 of the API it can show both English and metric values at the same time if you set the `scalebarUnits` property to `dual`. You can also control the `Scalebar` position through the `attachTo` parameter. By default, the `Scalebar` is positioned in the bottom left-hand corner of the map:

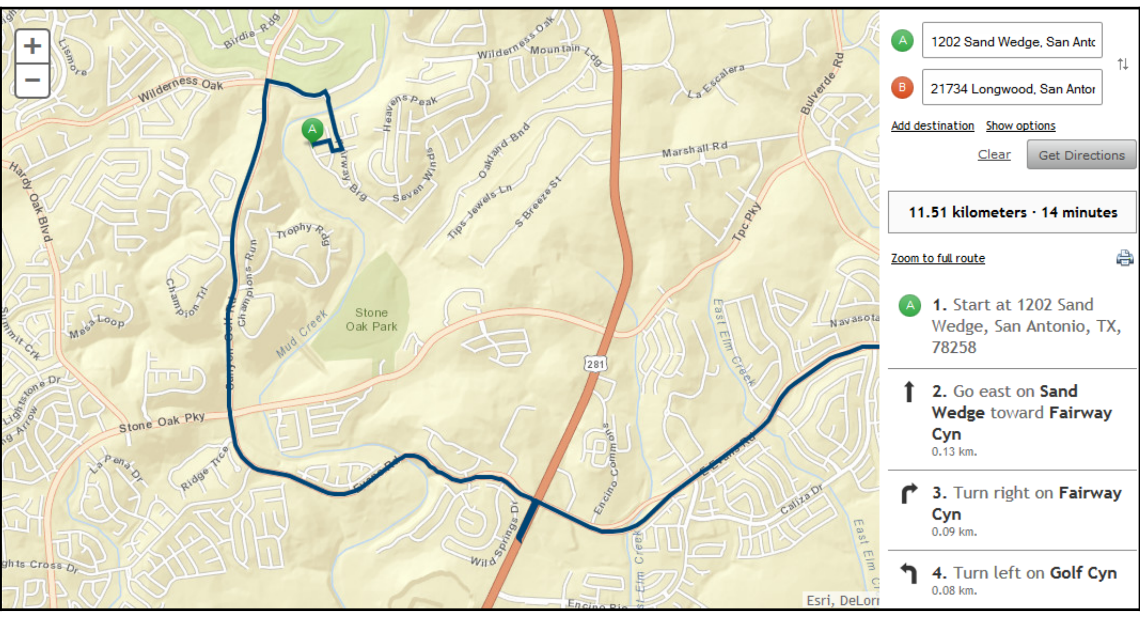



```
var scalebar = new esri.dijit.Scalebar({map:map, scalebarUnit:'english'});
```

The preceding code sample illustrates the creation of a `Scalebar` widget with units in English.

Directions

The `Directions` widget makes it easy to calculate directions between two or more input locations. The resulting directions, displayed in the following screenshot, are displayed with detailed turn-by-turn instructions and an optional map. If a map is associated with the widget the directions, route, and stops are displayed on the map. The stops displayed on the map are interactive, so you can click to display a popup with stop details or drag the stop to a new location to recalculate the route:

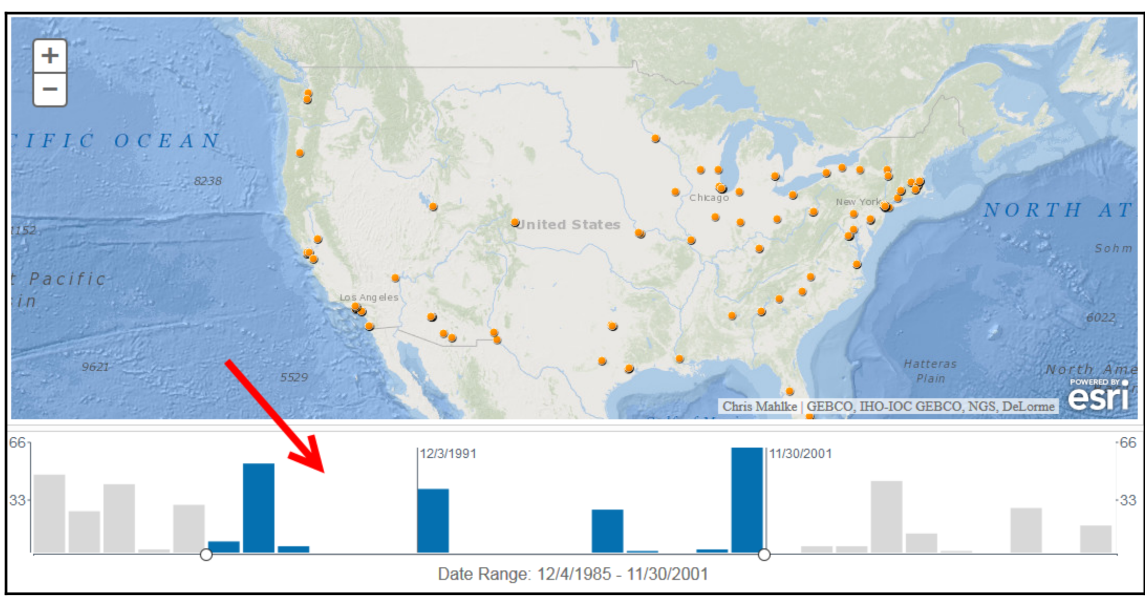



```
var directions = new Directions({
map: map
},"dir");

directions.startup();
```

The preceding code example shows the creation of a `Directions` widget. We'll look at the `Directions` widget in greater detail in a later chapter.

HistogramTimeSlider

The `HistogramTimeSlider` dijit provides a histogram chart representation of data for time-enabled layers on a map. Through the UI, users can temporally control the display of data with an extension to the `TimeSlider` widget:

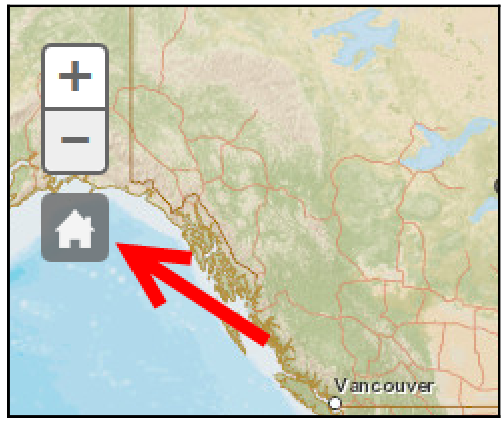



```
require(["esri/dijit/HistogramTimeSlider", ... ],
function(HistogramTimeSlider, ... ){
var slider = new HistogramTimeSlider({
dateFormat: "DateFormat(selector: 'date', fullYear: true)",
layers : [ layer ],
mode: "show_all",
timeInterval: "esriTimeUnitsYears"
  }, dojo.byId("histogram"));

map.setTimeSlider(slider);
});
```

In the preceding code example a `HistogramTimeSlider` object is created and associated with a map.

HomeButton

The `HomeButton` widget is simply a button that you can add to your application that returns the map to the initial extent:



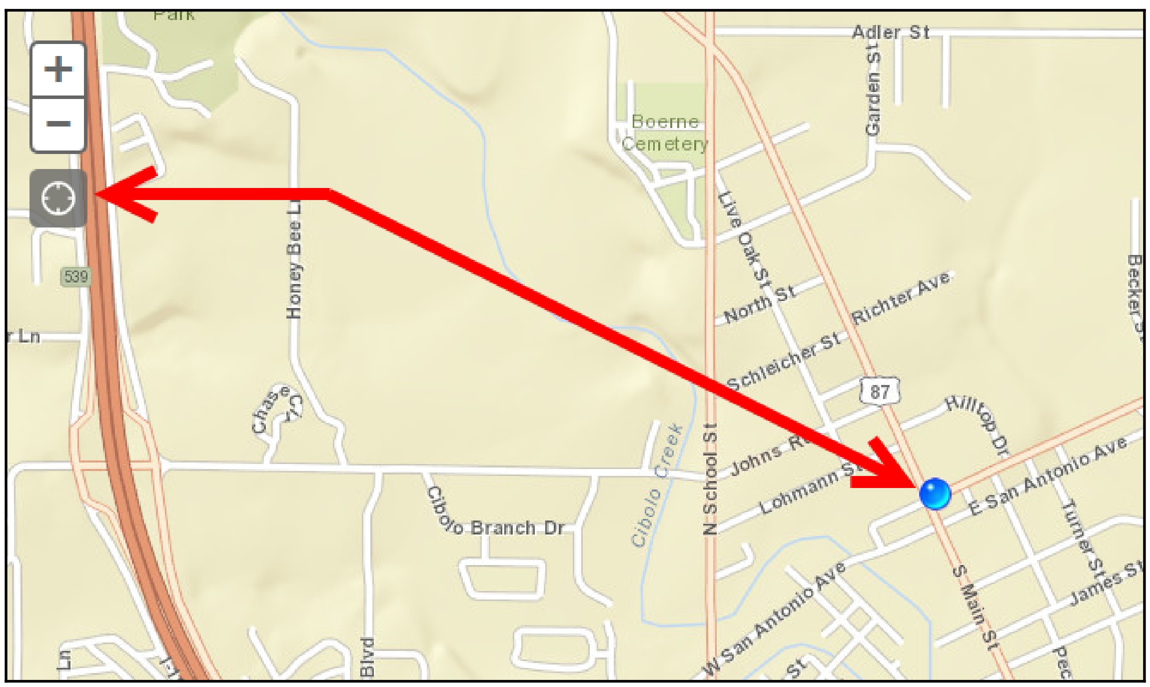

```
require([
      "esri/map",
"esri/dijit/HomeButton",
      "dojo/domReady!"
    ], function(
      Map, HomeButton
    )  {

var map = new Map("map", {
  center: [-56.049, 38.485],
  zoom: 3,
  basemap: "streets"
});

var home = new HomeButton({
  map: map
}, "HomeButton");
home.startup();

});
```

The preceding code example shows the creation of a `HomeButton` widget.

LocateButton

`LocateButton` can be used to find and zoom to the current location of the user. This widget uses the Geolocation API to find the user's current location. Once the location is found, the map zooms to that location. The widget provides options that allow the developer to define the following:

- HTML5 Geolocation Position options for finding a location such as `maximumAge` and `timeout`. The `timeout` property defines the maximum amount of time that can be used to determine the location of a device, while the `maximumAge` property defines the longest amount of time before a new location for the device will be found.
- The ability to define a custom symbol that will be used to highlight the user's current location on the map.
- The scale to zoom to when a location has been found:



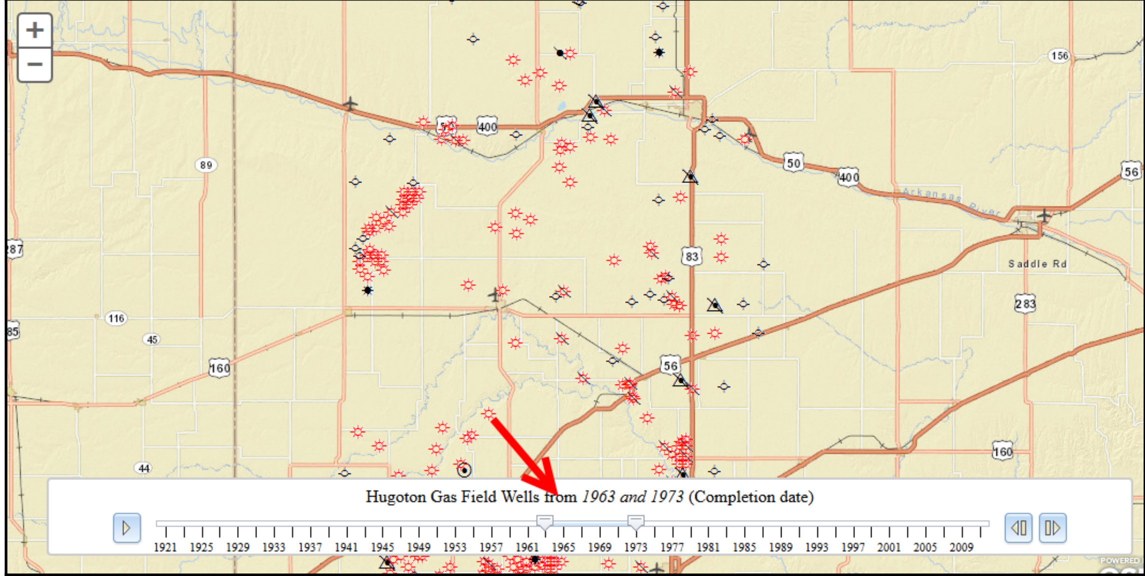
```
geoLocate = new LocateButton({
  map: map,
  highlightLocation: false
}, "LocateButton");
geoLocate.startup();
```

The preceding code example shows how to create an instance of the `LocateButton` and add it to the map.

Note that, as of version 3.19 of the ArcGIS API for JavaScript, your web app needs to be secure (accessed through HTPS) in order for the `LocateButton` to display. In version 3.18, this only applied to the Chrome web browser. Now it applies to all browsers.

TimeSlider

The `TimeSlider` widget is for visualizing time-enabled layers. The `TimeSlider` is configured to have two thumbs so only data within the time frame of the two thumb locations is displayed. The `setThumbIndexes()` method determines the initial location of each thumb. In this case a thumb is added at the initial start time and another thumb is positioned one time step up:

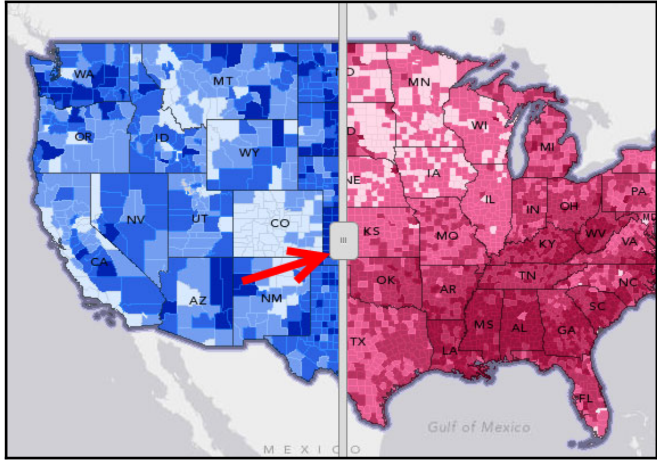


```
var timeSlider = new TimeSlider({
  style: "width: 100%;"
}, dom.byId("timeSliderDiv"));
map.setTimeSlider(timeSlider);

var timeExtent = new TimeExtent();
timeExtent.startTime = new Date("1/1/1921 UTC");
timeExtent.endTime = new Date("12/31/2009 UTC");
timeSlider.setThumbCount(2);
timeSlider.createTimeStopsByTimeInterval(timeExtent, 2,
"esriTimeUnitsYears");
timeSlider.setThumbIndexes([0,1]);
timeSlider.setThumbMovingRate(2000);
timeSlider.startup
```

The preceding code example illustrates how you can create an instance of the `TimeSlider` object and set various properties including the start and end times.

LayerSwipe

`LayerSwipe` provides a simple tool to show a portion of a layer or layers on top of a map. You can easily compare the content of multiple layers in a map by using this widget to reveal the contents of layer(s) on the map. The widget provides horizontal, vertical and scope viewing modes:



```
varswipeWidget = new LayerSwipe({
  type: "vertical",
  map: map,
  layers: [swipeLayer]
}, "swipeDiv");
swipeWidget.startup();
```

The preceding code example shows how to create an instance of `LayerSwipe` and add it to the map.

The Analysis widgets

A number of new analysis widgets were introduced with the 3.7 release of the ArcGIS API for JavaScript, and others have been added subsequently. Analysis Widgets provide access to the ArcGIS Spatial Analysis Service, which allows you to perform common spatial analyses on data, through the API.

Note that you will need an ArcGIS.com subscription to work with these widgets, and that you will require your account credentials because an analysis job will be deducted from your credit balance. Executing analysis tasks and hosting feature services are not available to personal account users.

The Analysis widgets include:

- `AnalysisBase`
- `AggregatePoints`
- `CreateBuffers`
- `CreateDriveTimeAreas`
- `DissolveBoundaries`
- `EnrichLayer`
- `ExtractData`
- `FindHotSpots`
- `FindNearest`
- `MergeLayers`
- `OverlayLayers`
- `SummarizeNearby`
- `SummarizeWithin`

Please refer to the `Working with Analysis Widgets` topic in the help documentation for more information on getting started with these widgets.

Feature editing

Simple feature editing is supported by the ArcGIS API for JavaScript when working against data stored in an enterprise geodatabase format. What this means is that your data needs to be stored in an enterprise geodatabase managed by ArcSDE.

Editing works on the concept of *last in wins*. For example, if two people are editing the same feature in a layer, and both submit modifications the last editor to submit changes will overwrite any changes made by the first editor. Obviously this could pose a problem in some cases so before implementing editing in your application you will need to examine how your data could be affected.

Other characteristics of editing include support for domains and subtypes, template style editing, and the ability to edit stand-alone tables and attachments. To use editing you will need to use a `FeatureService` and a `FeatureLayer`. Editing requests are submitted to the server using an HTTP Post request which in most cases will require the use of a proxy.

Web browsers allow applications to request data of, and retrieve data from, the servers on which they are hosted. You can install a proxy page on your web server to get around this limitation, which acts as an intermediary between the browser and remote servers. For more information, see the *Retrieve data from a web server* topic in the help documentation.

Editing support includes feature editing (including the creation and deletion of simple features) and modifying features through moves, cuts, union, or reshaping. You can also edit existing features and attach image, files, and comments to features.

FeatureService

Web editing requires a feature service to provide the symbology and feature geometry of your data. The feature service is just a map service with the feature access capability enabled. This capability allows the map service to expose feature geometries and their symbols in a way that is easy for Web applications to use and update.

Before you build a Web editing application, you need to create a feature service exposing the layers that you want to be edited. This involves setting up a map document and, optionally, defining some templates for editing. Templates allow you to pre-configure the symbology and attributes for some commonly-used feature types. For example, to prepare for editing streams, you might configure templates for *Major Rivers*, *Minor Rivers*, *Streams*, and *Tributaries*. Templates are optional, but they make it easy for the end user of the application to create common features.

Once your map is finished, you need to publish it to ArcGIS Server with the Feature Access capability enabled. This creates REST URLs, or endpoints, to both a map service *and* a feature service. You will use these URLs to reference the services in your application.

Feature services are accessible in the Web APIs through the `FeatureLayer` object which we covered in a previous chapter. Feature layers can work with either map services or feature services. However, when you use a `FeatureLayer` for editing purposes you must reference a feature service.

With the editing functionality your Web application first makes the changes to the FeatureLayer in the user's browser. You then call the `applyEdits()` method on the feature layer to apply the edits, which commits them to the database.

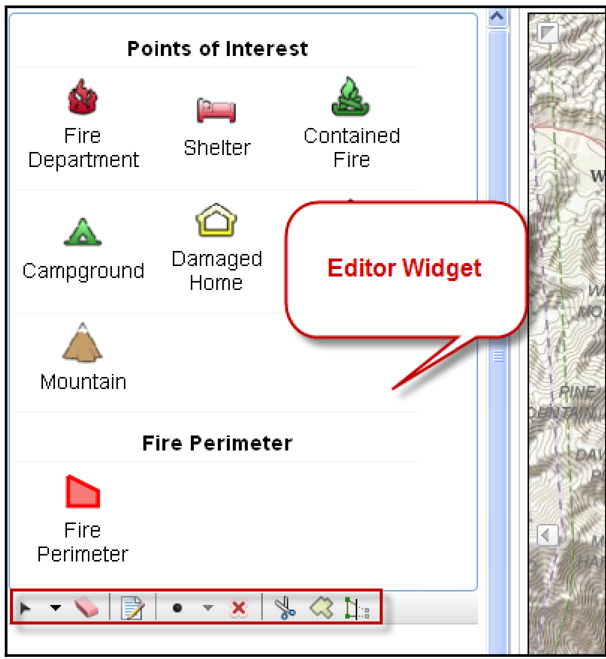
The Editing widgets

The ArcGIS API for JavaScript provides widgets to make it easier for you to add editing workflows to your Web applications. These widgets include the `Editor`, `TemplatePicker`, `AttributeInspector`, and `AttachmentEditor`. The `Editor` is the default editing interface and includes everything you need to edit a layer and also allows you to choose the number and types of tools available. `TemplatePicker` displays a pre-configured template containing symbols for each of the layers in your map document. This template style editing allows your users to pick a layer and begin editing, and is very simple and intuitive. The `AttributeInspector` provides an interface for editing the attributes of features and ensures valid data entry. Finally, the `AttachmentEditor` associates a downloadable file with a feature. We'll examine each of these widgets in more detail.

The Editor widget

The `Editor` widget, shown in the following screenshot, provides the default editing interface included with the API. It combines the functionality of the other widgets to provide everything that you need for editing a layer. You can choose the number and types of tools that are available on the widget.

The `Editor` widget saves your edits immediately after they are made; for example, as soon as you finish drawing a point. If you decide not to use the `Editor` widget, then when and how often to apply edits is down to you:

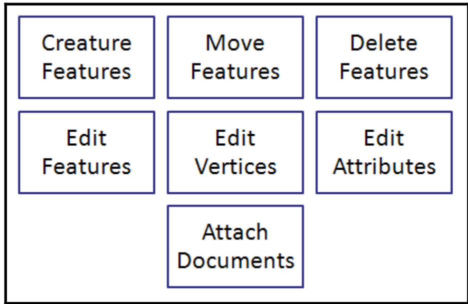


The following code shows how a new `Editor` object is constructed. The `params` object passed into the constructor is how the developer defines the functionality the editing application will include. In this case only the required options are defined. The required options are the map, the feature layers to edit, and the URL to a geometry service:

```
var settings = {
  map: map,
  geometryService: new
esri.tasks.GeometryService("http://servery/arcgis/rest/services/Geometry
      /GeometryServer"),
  layerInfos:[{
    featureLayer: myFeatureLayer
  }]
};
var params = {settings: settings};
var myEditingWidget = new Editor(params,'divEdit');
myEditingWidget.startup();
```

The Geometry Service is a map service that the ArcGIS Server administrator can enable. It does not depend on an underlying map document or another GIS resource. The Geometry Service allows you to perform certain operations such as measuring lengths and areas, re-projecting geometries, maintaining feature topology and more, by writing code that accesses the service. Web-based editing requires some of these capabilities, particularly those that involve re-shaping features, and uses a geometry service behind the scenes to fulfill them.

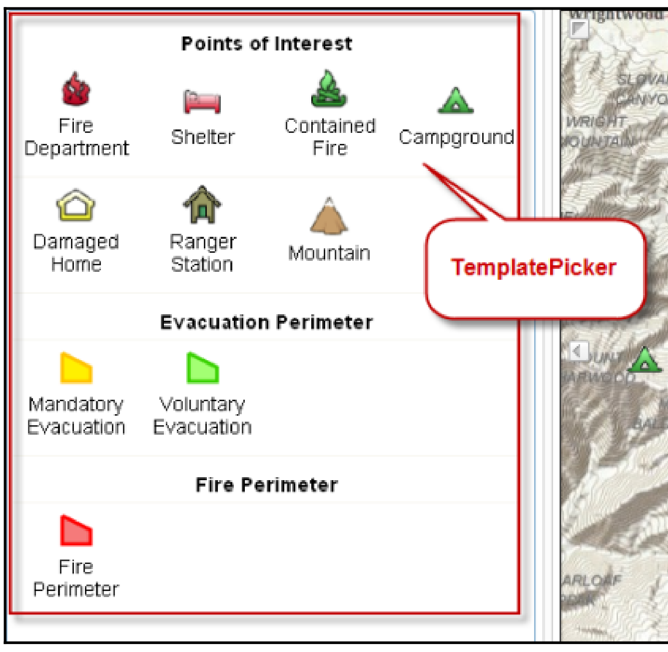
For most editing applications you should take advantage of the `Editor` widget. This widget allows you to perform all the functions you see listed as follows:



TemplatePicker widget

The `TemplatePicker` displays a pre-configured set of features to the user, with each symbolizing a layer in the service. Editing is initiated very simply by selecting a symbol from the template and then clicking on the map to add features.

The symbols displayed in the template come from the editing templates you defined in the feature service's source map or the symbols defined in the application. `TemplatePicker` can also be used as a simple legend:




```
function initEditing(results) {
   var templateLayers = dojo.map(results,function(result){
     return result.layer;
   });

   var templatePicker = new TemplatePicker({
     featureLayers: templateLayers,
     grouping: false,
     rows: 'auto',
     columns: 3
   },'editorDiv');
   templatePicker.startup();

   var layerInfos = dojo.map(results, function(result) {
     return {'featureLayer':result.layer};
   });

   var settings = {
       map: map,
       templatePicker: templatePicker,
```

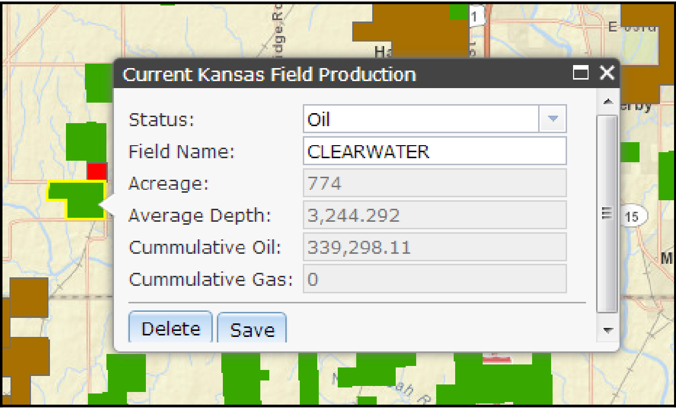
```
          layerInfos:layerInfos
        };
        var params = {settings: settings};
        var editorWidget = new Editor(params);
        editorWidget.startup();
    }
```

In preceding the code example a new `TemplatePicker` object is created and attached to the `Editor` widget.

AttributeInspector widget

The `AttributeInspector`, shown in the following screenshot, provides an interface for editing feature attributes over the Web. It also ensures that the data they enter is valid by matching the input to the expected data type. Domains are also supported. For example, if a coded value domain is applied to a field, the permitted values appear in a drop-down list, restricting the possibility of other values being entered. If a field requires a date value, a calendar appears, helping the user to supply a valid date.

The Editor widget instantiates the `AttributeInspector` by default, but if you are creating your own editing workflows, you might want to make use of it directly:



The `AttributeInspector` exposes all available attributes on the layer for editing. If you want to restrict the available attributes, you must code your own interface for entering and validating values:

```
var layerInfos = [{
  'featureLayer': petroFieldsFL,
  'showAttachments': false,
```

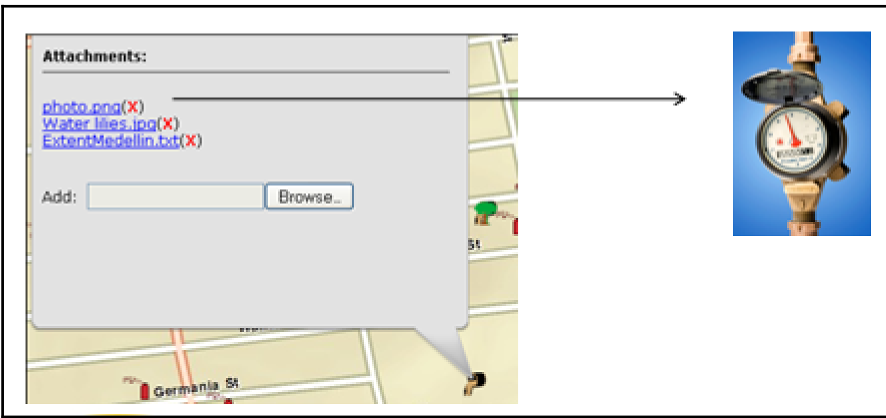
```
    'isEditable': true,
    'fieldInfos': [
    {'fieldName': 'activeprod', 'isEditable':true, 'tooltip': 'Current
  Status', 'label':'Status:'},
    {'fieldName': 'field_name', 'isEditable':true, 'tooltip': 'The name of
  this oil field', 'label':'Field Name:'},
    {'fieldName': 'approxacre', 'isEditable':false,'label':'Acreage:'},
    {'fieldName': 'avgdepth', 'isEditable':false, 'label':'Average Depth:'},
    {'fieldName': 'cumm_oil', 'isEditable':false, 'label':'Cummulative
  Oil:'},
    {'fieldName': 'cumm_gas', 'isEditable':false, 'label':'Cummulative Gas:'}
  ]
    }];

    var attInspector = new AttributeInspector({
      layerInfos:layerInfos
    }, domConstruct.create("div"));
    //add a save button next to the delete button
    var saveButton = new Button({ label: "Save", "class": "saveButton"});
   domConstruct.place(saveButton.domNode,   attInspector.deleteBtn.domNode,
  "after");
  saveButton.on("click", function(){
    updateFeature.getLayer().applyEdits(null, [updateFeature], null);
  });
  attInspector.on("attribute-change", function(evt) {
    //store the updates to apply when the save button is clicked
    updateFeature.attributes[evt.fieldName] = evt.fieldValue;
  });
  attInspector.on("next", function(evt) {
    updateFeature = evt.feature;
    console.log("Next " + updateFeature.attributes.objectid);
  });
  attInspector.on("delete", function(evt){
                evt.feature.getLayer().applyEdits(null,null,[feature]);
    map.infoWindow.hide();
  });

  map.infoWindow.setContent(attInspector.domNode);
  map.infoWindow.resize(350, 240);
```

In the preceding code example an `AttributeInspector` widget is created and added to the application. In addition, several event handlers, including `attribute-change`, `next`, and `delete` , are set up to handle various attribute changes.

The AttachmentEditor widget

In some situations, you may want to associate a downloadable file with a feature. For example, in a maintenance application you might want users to be able to click a feature representing a water meter and see a link to an image of the meter. In the ArcGIS Web APIs, an associated downloadable file like this is known as a feature attachment.

The `AttachmentEditor`, seen in the following screenshot, is a widget that helps users upload and view feature attachments. The `AttachmentEditor` includes a list of current attachments (with **Remove** button), as well as a Browse button that can be used to upload more attachments. The `AttachmentEditor` works well inside an info window, but can be placed elsewhere on the page:



In order to use feature attachments, attachments must be enabled on the source feature class. You can enable attachments for a `feature` class in ArcCatalog or the Catalog window in ArcGIS Pro. If the `Editor` widget detects that attachments are enabled, it will include an `AttachmentEditor`.

```
    var map;

        require([
          "esri/map",
          "esri/layers/FeatureLayer",
          "esri/dijit/editing/AttachmentEditor",
          "esri/config",

          "dojo/parser", "dojo/dom",

          "dijit/layout/BorderContainer", "dijit/layout/ContentPane",
  "dojo/domReady!"
```

```
        ], function(
          Map, FeatureLayer, AttachmentEditor, esriConfig,
          parser, dom
        ) {
          parser.parse();
          // a proxy page is required to upload attachments
          // refer to "Using the Proxy Page" for more information:
https://developers.arcgis.com
/en/javascript/jshelp/ags_proxy.html
          esriConfig.defaults.io.proxyUrl = "/proxy";

          map = new Map("map", {
            basemap: "streets",
            center: [-122.427, 37.769],
            zoom: 17
          });
          map.on("load", mapLoaded);

          function mapLoaded() {
            var featureLayer = new
FeatureLayer("http://sampleserver3.arcgisonline.com/ArcGIS/rest/services
                 /SanFrancisco/311Incidents/FeatureServer/0",{
              mode: FeatureLayer.MODE_ONDEMAND
            });

            map.infoWindow.setContent("<div id='content'
style='width:100%'></div>");
            map.infoWindow.resize(350,200);
            var attachmentEditor = new AttachmentEditor({},
dom.byId("content"));
            attachmentEditor.startup();

            featureLayer.on("click", function(evt) {
              var objectId =
evt.graphic.attributes[featureLayer.objectIdField];
              map.infoWindow.setTitle(objectId);
              attachmentEditor.showAttachments(evt.graphic,featureLayer);
              map.infoWindow.show(evt.screenPoint,
map.getInfoWindowAnchor(evt.screenPoint));
            });
            map.addLayer(featureLayer);
          }
        });
```

The preceding code shows how to create an `AttachmentEditor` and add it to the application.

The Edit toolbar

There may be times when you don't want to use the default editing workflow provided by the `Editor` widget. You might want to *create your own* as shown in the following image:

For example, you might not want changes to be applied immediately, which is what happens in the default workflow. Instead, you could request that a user confirm the change before it is committed.

You can use the `Edit` toolbar in these cases. The `Edit` toolbar is simply a JavaScript helper class that is part of the API. It helps with placing and moving vertices and graphics. This toolbar is similar to the `Navigation` and `Draw` toolbars that we examined earlier in the book. It provides all the underlying functionality of the Editor widget, but passes the responsibility for coding the interface to you, the developer.

Summary

Widgets and toolbars provide an easy way to add pre-built functionality to your application without having to write a lot of code. The wide array of available widgets has grown throughout the various releases of the API and it is expected that many new widgets will be available in future releases. Toolbars, though similar to widgets, are helper classes that provide the functionality you need to add navigation, drawing functionality, and editing tools to your application. However, it is up to you to define the appearance of your tools and how your users interact with them.

We'll cover specific widgets in greater detail in the forthcoming chapters.

In the next chapter you will learn how to create spatial and attribute queries using the `Query` and `QueryTask` classes and, use a widget to display attribute data.

6
Performing Spatial and Attribute Queries

The ArcGIS API for JavaScript `QueryTask` enables you to perform both attribute and spatial queries against data layers in a map service. You can also create queries that are a combination of the two. For example, you might need to find all land parcels with an appraised value of greater than $100,000, that intersect the 100-year floodplain.

In this chapter you will learn how to perform attribute and spatial queries using the `Query`, `QueryTask`, and `FeatureSet` objects in the ArcGIS API for JavaScript.

We will cover the following topics:

- Introducing tasks in ArcGIS Server
- Overview of attribute and spatial queries
- The `Query` object
- Executing the query with `QueryTask`
- Practice time with spatial queries

Introducing tasks in ArcGIS Server

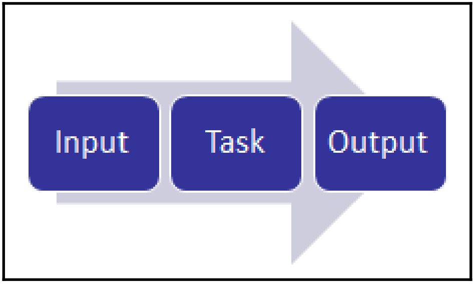
The ArcGIS API for JavaScript allows you to incorporate powerful GIS capabilities into your application by using the Task Framework.

Tasks enable you to perform spatial and attribute queries, find features based on text searches, geocode addresses, identify features, and perform various geometry operations including buffering and distance measurements. The tasks and their related objects reside in the `esri/tasks` namespace.

All tasks in the ArcGIS API for JavaScript follow the same general pattern, so that once you have learned how to use one type of task, you will easily be able to work with another.

At its most basic, the task workflow involves:

- Creating an object to represent the **Input** parameters for the task
- Executing the **Task**
- Unpacking the object returned by the task, containing the task **Output**:



The Task Framework

Overview of attribute and spatial queries

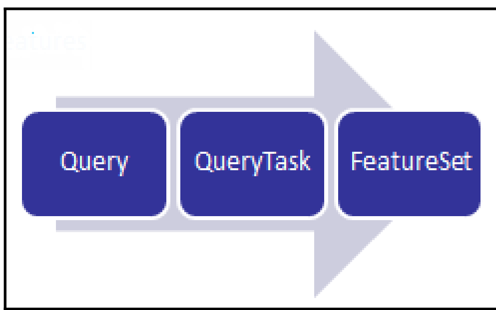
The `QueryTask` is no exception. It involves working with a series of objects to provide input to the task, execute the task, and dealing with the returned results. The input parameters for an attribute or spatial query are stored in a `Query` object that describes the nature of the query you want to perform. The `QueryTask` object executes the task using the information provided in the `Query` object. The results are returned in the form of a `FeatureSet` object. This contains an array of `Graphic` features which you can then display on the map.

The `Query` object, used as input to a `QueryTask`, contains properties including `geometry`, `where`, and `text`.

The `geometry` property is used to provide a geometry to use for a spatial query and is either a point, line, or polygon.

The `where` property is used to define an attribute query while the `text` property is used to perform a `SQL WHERE` *`field_name`* `LIKE` *`value`* query.

Other `Query` object properties include the ability to subset the fields that will be returned by the query, the output spatial reference for the returned geometry, and whether or not to return the geometries of the features in the result set:



The preceding diagram defines the object sequence you will use when creating attribute and spatial queries.

The Query object

In order for the `QueryTask` object to execute a query against a layer in a map service you need to define the query in a `Query` object.

The `Query` object will specify whether the query will be spatial, attribute, or a combination of the two.

Attribute queries can be defined by either the `where` or `text` properties. These properties are used to define a SQL attribute query. We'll look at the difference between `Query.where` and `Query.text` in a bit.

Spatial queries require that you set the `Query.geometry` property to define the input geometry that forms the basis of the spatial query.

Creating a new instance of the `Query` object is as simple as calling its constructor with no parameters:

```
var query = new  Query();
```

Setting query properties

At the bare minimum, you need to specify whether the query you want to create is attribute, spatial, or both.

Attribute queries

The `Query` object provides two properties that can be used in an attribute query: `Query.where` and `Query.text`.

In the following code example set the `Query.where` property so that only records where the `STATE_NAME` field are equal to `'Texas'` are returned. This is just a standard SQL query. Note that we have enclosed the word Texas in quotes. When performing an attribute query against a text column you need to enclose the text being evaluated with either single or double quotes. This isn't needed if you are performing an attribute query against columns containing other data types such as numbers or Booleans:

```
query.where = "STATE_NAME = 'Texas'";
```

You can also use the `Query.text` property to perform an attribute query. This is a shorthand way for creating a SQL `WHERE` clause using `LIKE`. The field used in the query is the **Display Field** for the layer defined in the map document. You can determine what the **Display Field** is for a layer by looking in the **ArcGIS Services Directory**. This is illustrated in the following screenshot where **ZONING_NAME** is the **Display Field**.

It is this display field that is queried using the `Query.text` property:

ArcGIS Services Directory

Home > Louisville > LOJIC_LandRecords_Louisville (MapServer) > Zoning

Layer: Zoning (ID: 2)

Display Field: ZONING_NAME

Type: Feature Layer

Geometry Type: esriGeometryPolygon

Description:

Definition Expression:

Copyright Text:

Min. Scale: 0

Max. Scale: 0

Default Visibility: True

Extent:

XMin: -85.9471222861492
YMin: 37.9969113880299
XMax: -85.4048460192834
YMax: 38.3802309070567
Spatial Reference: 4326

The following code example uses `query.text` to perform an attribute query that returns all features from a layer in a map service, where the value of the display field contains the state name entered by the user in a form field on the web page:

```
query = new Query();
query.returnGeometry = false;  // don't return geometries, just attributes
query.outFields = ['*'];       // return all attribute fields
query.text = dom.byId("stateName").value;
queryTask.execute(query, showResults);
```

Spatial queries

To perform a spatial query against a layer you'll need to pass in a valid geometry object to be used in the spatial filter along with a spatial relationship. Valid geometries include instances of `Extent`, `Point`, `Polyline`, and `Polygon`. The spatial relationship is set through the `Query.spatialRelationship` property and is applied during the query. The spatial relationship is defined through the use of one of the constant values described in the table as follows:

Constant	Description
SPATIAL_REL_CONTAINS	•Part or all of a feature from feature class 1 is contained within a feature from feature class 2.
SPATIAL_REL_CROSSES	•The feature from feature class 1 crosses a feature from feature class 2.
SPATIAL_REL_ENVELOPEINTERSECTS	•The envelope of feature class 1 intersects with the envelope of feature class 2.
SPATIAL_REL_INDEXINTERSECTS	•The envelope of the query feature class intersects the index entry for the target feature class.
SPATIAL_REL_INTERSECTS	•Part of a feature from feature class 1 is contained in a feature from feature class 2.
SPATIAL_REL_OVERLAPS	•Features from feature class 1 overlap features in feature class 2.
SPATIAL_REL_RELATION	•Allows specification of any relationship defined using the Shape Comparison Language.
SPATIAL_REL_TOUCHES	•The feature from feature class 1 touches the border of a feature from feature class 2.
SPATIAL_REL_WITHIN	•The feature from feature class 1 is completely enclosed by the feature from feature class 2

The following code example sets a `Point` object as the geometry passed into the spatial filter and sets the spatial relationship:

```
query.geometry = evt.mapPoint;
query.spatialRelationship = SPATIAL_REL_INTERSECTS;
```

The query defined in the code fragment is effectively requesting all features that intersect the point where the user clicked on the map.

Limiting the fields returned

Every column of information attached to the `FeatureSet` is additional data that must be passed from the server to the browser. Therefore, to improve performance, you should only include the fields that are needed by your application. You should also only return feature geometries if you intend to plot them on the map.

To specify the fields you require, create an array that contains the field names and assign it to the `Query.outFields` property as seen in the following code example. To return all fields you can use `outFields = ['*']`.

Feature geometry is returned by default. But if you don't need it, for example, if you intend to display only the data and not the location of the associated feature on the map, set `Query.returnGeometry = false`:

```
query.outFields = ["NAME", "POP2000", "POP2007"];
query.returnGeometry = false;
```

Executing the query with QueryTask

Once you've defined the nature of the query in a `Query` object you can then use `QueryTask` to execute the query.

First, create an instance of the `QueryTask` object. You create a `QueryTask` object by passing the URL of the layer against which the query will be executed to its constructor. Note from the code example that follows that the map service URL includes an index number which references a specific layer in the map service which will be queried:

```
myQueryTask = new
QueryTask("http://sampleserver1.arcgisonline.com/ArcGIS/rest/services/Demog
raphics/ESRI_CENSUS_USA/MapServer/5");
```

Once you have created the `QueryTask` object you can perform a query against its layer passing a `Query` object to its `QueryTask.execute()` method.

`QueryTask.execute()` accepts three parameters: the `Query` object, a function to call when the operation succeeds, and a function to call if the operation fails.

The syntax for `QueryTask.execute()` is provided as follows:

```
QueryTask.execute(objQuery,callback?,errback?)
```

Assuming that the query executes without error, the success `callback` function will be called and a `FeatureSet` object passed into the function. If an error occurs during execution of the query then an error `callback` function is executed. Both the success and error `callback` functions are optional. This always struck me as being a bit odd: what's the point in executing a query, if you are not interested in the results? But it's a good idea to check for any errors so that your application can fail gracefully.

At this point you may be wondering about these `callback` and `errback` functions. Most tasks in ArcGIS Server return an instance of `dojo.Deferred`. A `Deferred` object is a class that is used as the foundation for managing asynchronous threads in `Dojo`. Tasks in ArcGIS Server can be either synchronous or asynchronous.

Asynchronous and synchronous define how the client (the application using the task) interacts with the server and retrieves the results generated by the task. When a service is set to synchronous, the client waits for the task to finish. Typically, a synchronous task executes quickly (a second or two at most). An asynchronous task typically takes longer to execute, and the client doesn't wait for the task to finish. The end user is free to continue to use the application while the task executes. When a task finishes on the server it calls the `callback` function and passes the results into this function where they can then be used in some way. Often they are displayed on the map.

Let's take a look at a more complete code example. In the following, we first create a new variable called `myQueryTask` which points to layer 6 (the index numbers are zero based) in the `ESRI_CENSUS_USA` map service. We then create the `Query` object containing the input properties of the query and finally we use the `execute()` method on `QueryTask` to perform the query. The `execute()` method returns a `FeatureSet` object that contains the results of the query and these features are processed through a `callback` function called `showResults`. If an error occurs during execution of the task, the `errorCallback()` function will be called instead:

```
myQueryTask = new
esri.tasks.QueryTask("http://sampleserver1.arcgisonline.com/ArcGIS/rest/ser
vices/Demographics/ESRI_CENSUS_USA/MapServer/5");
// build query filter
myQuery = new esri.tasks.Query();
myQuery.returnGeometry = false;
myQuery.outFields = ["STATE_NAME", "POP2007", "MALES", "FEMALES"];
myQuery.text = 'Oregon';// execute query
myQueryTask.execute(myQuery, showResults, errorCallback);
function showResults(fs) {
    // do something with the results
    // they are returned as a FeatureSet object
}
```

```
Function errorCallback() {
    alert("An error occurred during task execution");
}
```

Getting the results of the query

As I mentioned earlier, the results of a query are returned to your application as a `FeatureSet` object. If you have requested the geometries of the features returned by the query (the default behavior), these are included in the `FeatureSet` object's `features` property as an array of `Graphic` objects. This makes it trivial to plot them on the map if you want to. This is illustrated in the screenshot as follows:

```
▼ featureSet: Object
    displayFieldName: "CITY_NAME"
  ▶ features: Array(23)
  ▶ fieldAliases: Object
  ▶ fields: Array(3)
    geometryType: "esriGeometryPoint"
  ▶ spatialReference: Object
    transform: null
    _ssl: undefined
  ▶ __proto__: Object
```

Each feature (of type `Graphic`) in the array can contain geometry, attributes, symbology, and an `InfoTemplate` as described in `Chapter 3`, *Adding Graphics to the Map*.

The following code example shows a `callback` function which is executed to process the `FeatureSet` when a query has completed execution:

```
Function addPolysToMap(featureSet) {
var features = featureSet.features;
var feature;
for (i=0, il=features.length; i<il; i++) {
feature = features[i];
   attributes = feature.attributes;
   pop = attributes.POP90_SQMI;
map.graphics.add(features[i].setSymbol(sym));
    }
}
```

Practice time with spatial queries

In this exercise you will practice performing spatial queries using the `Query`, `QueryTask`, and `FeatureSet` objects in the ArcGIS API for JavaScript. We will be querying and displaying records from t the Louisville, Kentucky land records layer.

Follow the given steps to complete the exercise:

1. Open the JavaScript Sandbox at `https://developers.arcgis.com/javascript/3/sandbox/sandbox.html`.
2. Remove all the JavaScript content from the `<script>` tag that I have highlighted as follows:

```
<script>
  var map;

  require(["esri/map", "dojo/domReady!"], function(Map) {
    map = new Map("map", {
      basemap: "topo",  //For full list of pre-defined
basemaps,
      navigate to http://arcg.is/1JVo6Wd
      center: [-122.45, 37.75], // longitude, latitude
      zoom: 13
    });
  });
</script>
```

3. Create variables to store the `map` object, the `QueryTask` and `Query` objects, a symbol, and an info template:

```
<script>
  var map, query, queryTask;
  var symbol, infoTemplate;
</script>
```

4. Add the `require()` function as seen in the highlighted code as follows:

```
 <script>
  var map, query, queryTask;
  var symbol, infoTemplate;

  require([
  "esri/map", "esri/tasks/query",
  "esri/tasks/QueryTask", "esri/tasks/FeatureSet",
  "esri/symbols/SimpleFillSymbol",
"esri/symbols/SimpleLineSymbol",
```

```
  "esri/InfoTemplate", "dojo/_base/Color", "dojo/on",
  "dojo/domReady!"
  ], function(Map, Query, QueryTask, FeatureSet,
  SimpleFillSymbol, SimpleLineSymbol,
  InfoTemplate, Color, on) {
  });
</script>
```

5. Inside the `require()` function create the `map` object. The map will be centered on the Louisville, KY area:

```
<script>
  var map, query, queryTask;
  var symbol, infoTemplate;

  require([
  "esri/map", "esri/tasks/query", "esri/tasks/QueryTask",
  "esri/tasks/FeatureSet", "esri/symbols/SimpleFillSymbol",
  "esri/symbols/SimpleLineSymbol", "esri/InfoTemplate",
  "dojo/_base/Color", "dojo/on", "dojo/domReady!"
  ], function(Map, Query, QueryTask, FeatureSet,
  SimpleFillSymbol, SimpleLineSymbol, InfoTemplate, Color, on)
{
    map = new Map("map",{
      basemap: "streets",
      center:[-85.748, 38.249], //long, lat
      zoom: 13
    });
  });
</script>
```

6. Create the symbol that will be used to display the results of the query:

```
<script>
  var map, query, queryTask;
  var symbol, infoTemplate;

  require([
  "esri/map", "esri/tasks/query", "esri/tasks/QueryTask",
  "esri/tasks/FeatureSet", "esri/symbols/SimpleFillSymbol",
  "esri/symbols/SimpleLineSymbol", "esri/InfoTemplate",
  "dojo/_base/Color", "dojo/on", "dojo/domReady!"
  ], function(Map, Query, QueryTask, FeatureSet,
SimpleFillSymbol,
  SimpleLineSymbol, InfoTemplate, Color, on) {
    map = new Map("map",{
      basemap: "streets",
      center:[-85.748, 38.249], //long, lat
```

```
      zoom: 13
    });
    symbol = new SimpleFillSymbol(SimpleFillSymbol.STYLE_SOLID,
    new SimpleLineSymbol(SimpleLineSymbol.STYLE_SOLID,
    new Color([111, 0, 255]), 2),
    new Color([255,255,0,0.25]));
    infoTemplate = new InfoTemplate("${OBJECTID}","${*}");
  });
</script>
```

7. Now, inside the `require()` function we are going to initialize the `queryTask` variable and then register the `QueryTask.complete` event. Add the following highlighted lines of code:

```
<script>
  var map, query, queryTask;
  var symbol, infoTemplate;

  require([
  "esri/map", "esri/tasks/query", "esri/tasks/QueryTask",
  "esri/tasks/FeatureSet", "esri/symbols/SimpleFillSymbol",
  "esri/symbols/SimpleLineSymbol", "esri/InfoTemplate",
  "dojo/_base/Color", "dojo/on", "dojo/domReady!"
  ], function(Map, Query, QueryTask, FeatureSet,
SimpleFillSymbol,
  SimpleLineSymbol, InfoTemplate, Color, on) {
    map = new Map("map",{
      basemap: "streets",
      center:[-85.748, 38.249], //long, lat
      zoom: 13
    });
    symbol = new SimpleFillSymbol(SimpleFillSymbol.STYLE_SOLID,
    new SimpleLineSymbol(SimpleLineSymbol.STYLE_SOLID, new
Color([111,
    0, 255]), 2), new Color([255,255,0,0.25]));
    infoTemplate = new InfoTemplate("${OBJECTID}", "${*}");
    queryTask = new
    QueryTask("http://sampleserver1.arcgisonline.com/ArcGIS
    /rest/services/Louisville/LOJIC_LandRecords_Louisville
    /MapServer/2");
    queryTask.on("complete", addToMap);
  });
</script>
```

The constructor for `QueryTask` must be a valid URL pointer to a data layer within a map service. In this case, we are creating a reference to the zoning layer in the `LOJIC_LandRecords_Louisville` map service. What this indicates is that we are going to perform a query against this layer. If you will remember from a previous chapter, `dojo.on()` is used to register events. In this case, we are registering the `complete` event for our new `QueryTask` object. This event fires when the query has completed, and in this case we'll handle this event using the `addToMap()` function specified as a parameter to `on()`.

8. Now we'll define the input parameters for the task by creating a `Query` object. In the first line we are creating a new instance of `Query`, and then we set the `Query.returnGeometry` and `Query.outFields` properties. Setting `Query.returnGeometry` to `true` indicates that ArcGIS Server should return the geometries of the features that matched the query so that we can plot them on the map. In `Query.outFields` we've specified a wildcard indicating that *all* fields associated with the zoning layer should be returned. Add the following highlighted lines of code just under the code you entered in the last step:

```
<script>
  var map, query, queryTask;
  var symbol, infoTemplate;

  require([
  "esri/map", "esri/tasks/query", "esri/tasks/QueryTask",
  "esri/tasks/FeatureSet", "esri/symbols/SimpleFillSymbol",
  "esri/symbols/SimpleLineSymbol", "esri/InfoTemplate",
  "dojo/_base/Color", "dojo/on", "dojo/domReady!"
  ], function(Map, Query, QueryTask, FeatureSet,
SimpleFillSymbol,
  mSimpleLineSymbol, InfoTemplate, Color, on) {
    map = new Map("map",{
      basemap: "streets",
      center:[-85.748, 38.249], //long, lat
      zoom: 13
    });
    symbol = new SimpleFillSymbol(SimpleFillSymbol.STYLE_SOLID,
    new SimpleLineSymbol(SimpleLineSymbol.STYLE_SOLID, new
Color([111,
    0, 255]), 2), new Color([255,255,0,0.25]));
    infoTemplate = new InfoTemplate("${OBJECTID}", "${*}");
    queryTask = new
    QueryTask("http://sampleserver1.arcgisonline.com/ArcGIS
    /rest/services/Louisville/LOJIC_LandRecords_Louisville
    /MapServer/2");
    queryTask.on("complete", addToMap);
```

```
        query = new Query();
        query.returnGeometry = true;
        query.outFields = ["*"];

      });
    </script>
```

9. Add a line of code just after the `map` object declaration that registers the `Map.click` event to a function called `doQuery`. The `doQuery` function will be passed the point on the map that was clicked by the user. This map point will be used as the geometry for the spatial query. In the next step we will create the `doQuery` function that will accept the point clicked on the map:

```
<script>
  var map, query, queryTask;
  var symbol, infoTemplate;

  require([
  "esri/map", "esri/tasks/query", "esri/tasks/QueryTask",
  "esri/tasks/FeatureSet", "esri/symbols/SimpleFillSymbol",
  "esri/symbols/SimpleLineSymbol", "esri/InfoTemplate",
  "dojo/_base/Color", "dojo/on", "dojo/domReady!"
  ], function(Map, Query, QueryTask, FeatureSet,
SimpleFillSymbol,
  SimpleLineSymbol, InfoTemplate, Color, on) {
    map = new Map("map",{
      basemap: "streets",
      center:[-85.748, 38.249], //long, lat
      zoom: 13
    });
    map.on("click", doQuery);

    symbol = new SimpleFillSymbol(SimpleFillSymbol.STYLE_SOLID,
    new SimpleLineSymbol(SimpleLineSymbol.STYLE_SOLID, new
Color([111,
    0, 255]), 2), new Color([255,255,0,0.25]));
    infoTemplate = new InfoTemplate("${OBJECTID}", "${*}");
    queryTask = new
    QueryTask("http://sampleserver1.arcgisonline.com/ArcGIS
    /rest/services/Louisville/LOJIC_LandRecords_Louisville
    /MapServer/2");
    queryTask.on("complete", addToMap);
    query = new Query();
    query.returnGeometry = true;
    query.outFields = ["*"];

  });
```

```
</script>
```

10. Now we'll create the `doQuery` function that handles the `Map.click` event. This function populates the `Query` object with the required geometry for a spatial query and then executes the query.
11. When the `Map.click` event gets fired, `doQuery` is passed an `event` object. The `event` object contains a `mapPoint` property which in turn contains a `Point`, which represents the point on the map where the user clicked.
12. We use this `Point` object to set the `Query.geometry` property. We will use this to locate the zoning parcel that contains the point.
13. Finally, we call `QueryTask.execute()` to perform the query. After the task completes, it will return a `FeatureSet` object that contains the features that satisfy the query criteria:

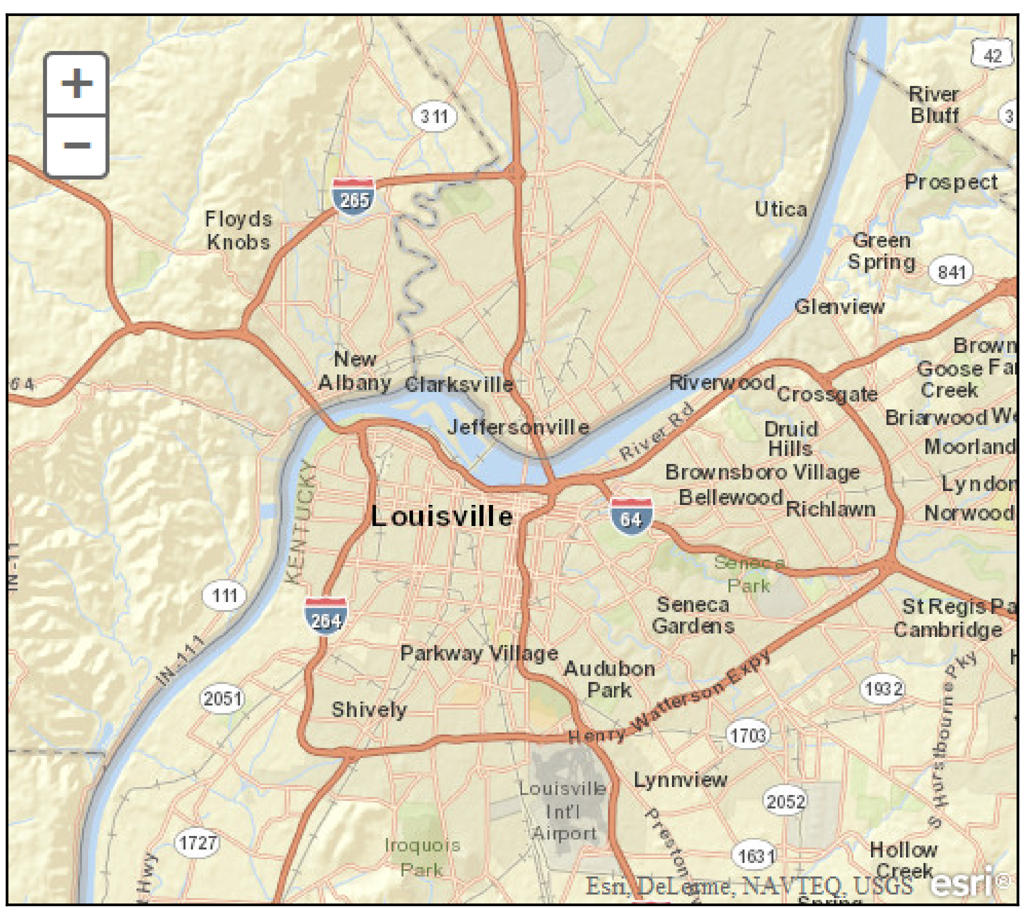
```
function doQuery(evt) {
    //clear currently displayed results
    map.graphics.clear();

    query.geometry = evt.mapPoint;
    query.outSpatialReference = map.spatialReference;
    queryTask.execute(query);
}
```

Now we need some way to access that object, because at the moment the ArcGIS Server has no way of returning it to our application. Add the following code block just under the closing brace for the `require()` function. Recall that we registered the `QueryTask.complete` event to trigger the `addToMap()` function. We haven't created this function yet, so let's go ahead and do that.

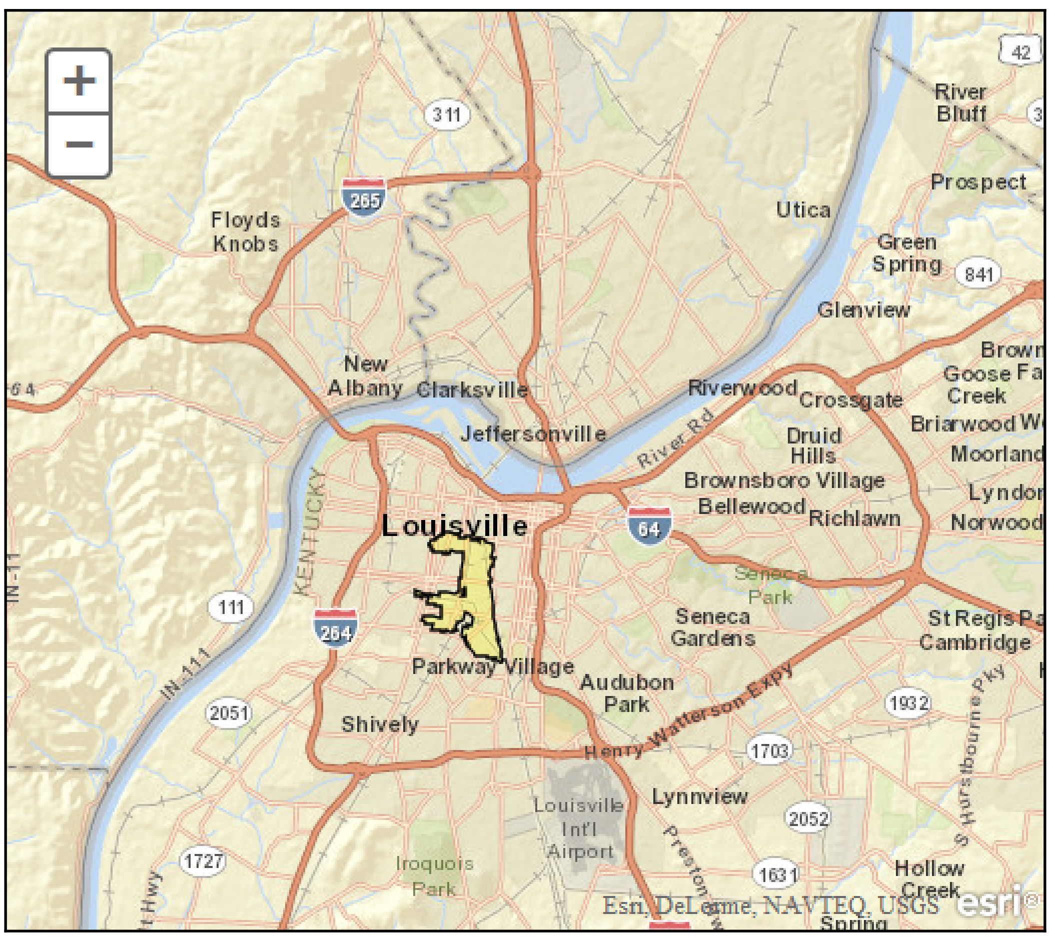
14. Create the `addToMap()` function. This function will accept a `FeatureSet` returned from the `QueryTask` and use it to display the features on the map. We'll also create an info template for each feature so that we can display its attributes:

```
function addToMap(results) {
  var featureArray = results.featureSet.features;
  var feature = featureArray[0];
map.graphics.add(feature.setSymbol(symbol).setInfoTemplate(info
Template));
}
```

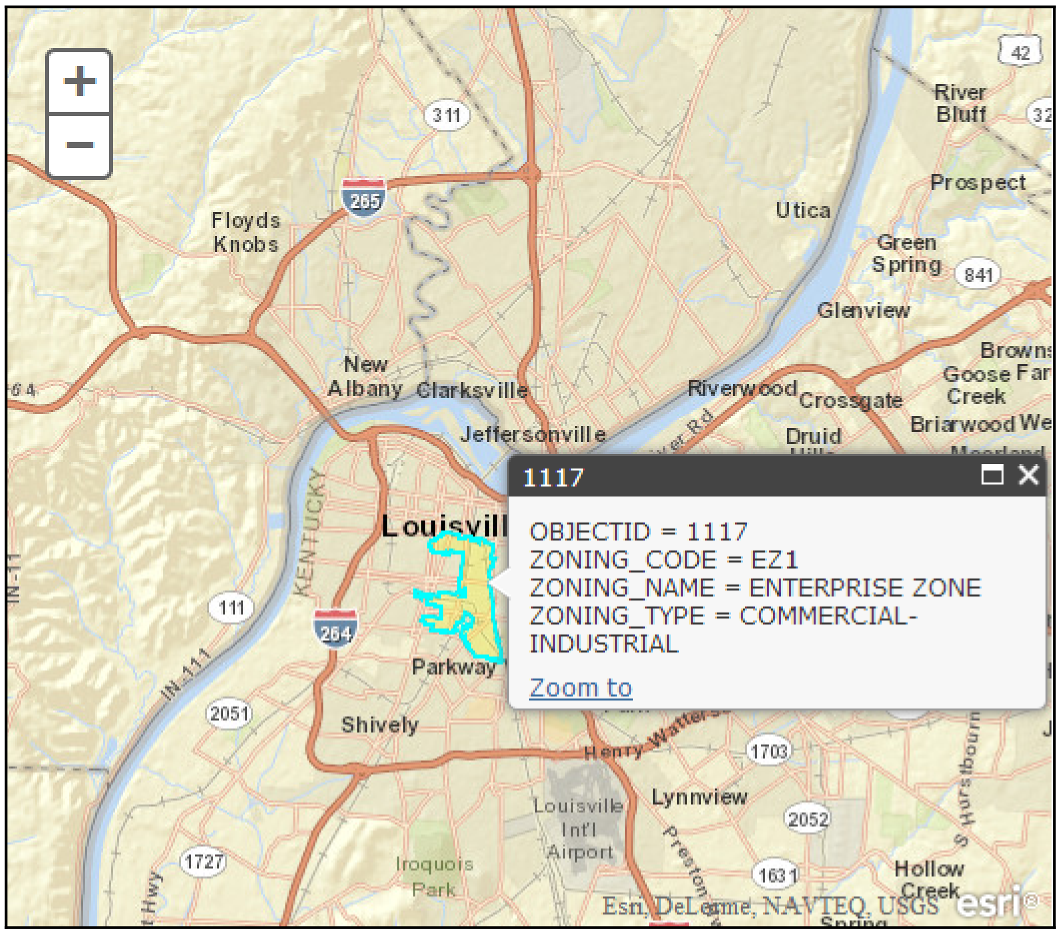
15. Click the **Run** button to execute the code. You should see the following map. If not, you may need to recheck your code for accuracy:



16. Click anywhere on the map to run the query. You should see the zoning polygon associated with the area you clicked:



17. Now, click the highlighted zoning polygon to display the attributes associated with that feature:



In this practice, you learned how to use the `Query` and `QueryTask` objects to create a spatial query which locates the zoning polygon that intersects the point where the user has clicked on the map.

Summary

In this chapter we introduced the concept of tasks in ArcGIS Server. ArcGIS Server provides a number of tasks for commonly used operations in a web mapping application, such as attribute and spatial queries.

To execute a query, you need a `QueryTask` object that references a layer within a map service that you will execute the query against. You also need a `Query` object to define the query itself.

Properties you might choose to set on the `Query` object include `where` and `text` properties to define attribute queries, a `geometry` property to define a spatial query, and an `outFields` property to subset the fields that should be returned.

When the ArcGIS Server has finished processing the query, it returns a `FeatureSet` object to your application by way of a `callback` function. You specify the name of this function when you execute the query.

The `callback` function is responsible for processing the `FeatureSet` and the array of `Graphic` objects it stores in its `feature` property, on the map.

In the next chapter you will learn how to use two additional tasks: `IdentifyTask` and `FindTask`. Both can be used to return the attributes of features and, as you will see, they are implemented in a very similar way to the `QueryTask` you worked within this chapter.

7
Identifying and Finding Features

Being able to identify features and find out information about them is a common requirement in web mapping applications. In this chapter we're going to cover two ArcGIS Server tasks related to returning feature attributes: `IdentifyTask` and `FindTask`.

`IdentifyTask` returns the attributes of features that have been clicked on a map. The attribute information is often presented in a pop-up window. As with the other tasks we have seen, the `IdentifyTask` object relies upon input parameters, in this case an object called `IdentifyParameters` that controls the results of the identify operation. These parameters include which layers in a service to use for the identify operation, and an acceptable distance from a feature for it to be considered as relevant to the map click location. An instance of `IdentifyResult` is used to hold the results of the task.
More often than not, the tasks that you can execute with the ArcGIS API for JavaScript replicate some of the most commonly used functions in ArcGIS Desktop. `FindTask` is one such operation. Just as in the desktop version of ArcGIS, this task can be used to find features in a layer that match a string value. Before executing a find operation with a `FindTask` object, you will need to set various parameters of the operation in an instance of `FindParameters`. `FindParameters` lets you specify things like the search text, the fields in which to search for that text, and more. Once supplied with a `FindParameters` object, `FindTask` then executes its tasks against one or more layers and fields and returns a `FindResult` object that contains the `layerID`, `layerName`, and feature that matched the search string.

In this chapter we will deal with the following topics:

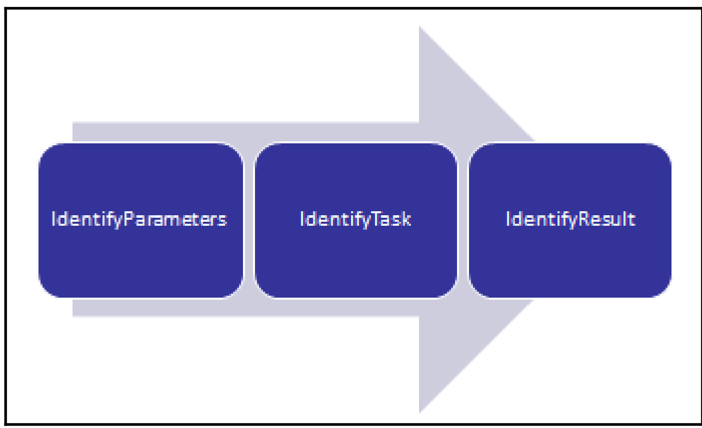
- Using `IdentifyTask` to access feature attributes
- Using `FindTask` to access feature attributes

Using IdentifyTask to access feature attributes

`IdentifyTask` allows a user to click on the map and return information about the feature or features they clicked on. In this section, you will learn how to use the various objects associated with `IdentifyTask` to achieve this.

Introducing IdentifyTask

As with the other tasks in ArcGIS Server, `IdentifyTask` functionality is separated into three distinct classes in the API: `IdentifyParameters`, `IdentifyTask`, and `IdentifyResult`. These three classes are illustrated in the following diagram:



IdentifyParameters

The input parameter object for `IdentifyTask` is `IdentifyParameters`. This object contains a number of properties that you can use to specify the parameters for the identify operation. These parameters include the geometry used to select features (`IdentifyParameters.geometry`), the layer on which to perform the identify (`IdentifyParameters.layerIds`), and a tolerance factor that specifies the distance from the map click location for a feature to be counted in the results (`IdentifyParameters.tolerance`).

To use the API's identify functionality, you first need to import the `IdentifyTask`, `IdentifyParameters`, and `IdentifyResults` modules into your application:

```
require(["esri/tasks/IdentifyTask", ... ], function(IdentifyTask, ... ){
... });
```

Before setting the various parameters on the `IdentifyParameters` object, you need to first create an instance of this object:

```
var identifyParams = new IdentifyParameters();
```

Now that you have an instance of `IdentifyParameters`, you can set its properties as seen in the following:

```
identifyParams.geometry = evt.MapPoint;  identifyParams.layerIds[0,1,2];
identifyParams.returnGeometry = true; identifyParams.tolerance = 3;
```

In most cases, an identify operation is performed using a point that the user has clicked on the map. You can obtain this using the point returned from the map click event as seen in the preceding code example. The layers which should be searched can be defined using an array of layer ids that are passed into the `IdentifyParameters.layerIds` property. The array should contain numeric values that reference the index numbers of the layers you want to search. You can obtain this information from the services directory. The `tolerance` property is especially important. This sets the distance in pixels around the geometry you're using for the identify operation. This geometry is usually a point, so you can think of the tolerance as a circular buffer around the point, with its radius being equal to the tolerance value you specify, in screen pixels. When the `IdentifyTask` is executed, any features from the layers to be identified that are within or intersect the circle are returned.

It's likely that you'll need to experiment with this tolerance value to obtain a value that is best for your application. If the value is set too low, you run the risk of the `identify` operation not identifying any features and, conversely, if the value is set too high, you may get too many features returned. It can be difficult to find the right balance and the tolerance value that works for one application may not work for another.

IdentifyTask

`IdentifyTask` performs the `identify` operation on one or more layers using the parameters specified in `IdentifyParameters`. As with the other tasks that we've examined, `IdentifyTask` needs a reference to the URL of the map service to use in the `identify` operation.

A new instance of `IdentifyTask` can be created with the following code example you see. The `IdentifyTask` constructor accepts just one parameter: the URL of the map service that contains the layer(s) against which the `identify` operation will be executed:

```
var identify = new
IdentifyTask("http://sampleserver1.arcgisonline.com/ArcGIS/rest/services/Sp
ecialty/ESRI_StatesCitiesRivers_USA/MapServer");
```

Once you've created a new instance of the `IdentifyTask` object, you can call the `IdentifyTask.execute()` method, which accepts an `IdentifyParameters` object along with optional success callback and error callback functions.

The following code is an example of where the `IdentifyTask.execute()` method is called. An instance of `IdentifyParameters` is passed as a parameter into the `IdentifyTask.execute()` method along with a reference to an `addToMap()` callback function which processes the results that are returned to it:

```
identifyParams = new IdentifyParameters();
identifyParams.tolerance = 3;
identifyParams.returnGeometry = true;
identifyParams.layerIds = [0,2];
identifyParams.geometry = evt.mapPoint;

identifyTask.execute(identifyParams, function(idResults) {
addToMap(idResults, evt); });

function addToMap(idResults, evt) {
    //add the results to the map
}
```

IdentifyResult

The `IdentifyTask` operation returns an array of `IdentifyResult` objects to the success `callback` function. Each `IdentifyResult` object contains the feature returned by the identify operation along with the layer id and the name of the layer in which the feature was found. The following code illustrates how an array of `IdentifyResult` objects can be processed:

```
function addToMap(idResults, evt) {
bldgResults = {displayFieldName:null,features:[]};
parcelResults = {displayFieldName:null,features:[]};

for (vari=0, i<idResults.length; i++) {
var idResult = idResults[i];
```

```
if (idResult.layerId === 0) {
if (!bldgResults.displayFieldName) {bldgResults.displayFieldName =
idResult.displayFieldName};
bldgResults.features.push(idResult.feature);
          }
else if (idResult.layerId === 2) {
if (!parcelResults.displayFieldName) {parcelResults.displayFieldName =
idResult.displayFieldName};
parcelResults.features.push(idResult.feature);
          }
        }
dijit.byId("bldgTab").setContent(layerTabContent(bldgResults,"bldgResults")
);
dijit.byId("parcelTab").setContent(layerTabContent(parcelResults,"parcelRes
ults"));

map.infoWindow.show(evt.screenPoint,
map.getInfoWindowAnchor(evt.screenPoint));
}
```

Practice time - implementing identify functionality

In this exercise, you will implement identify functionality in an application. This is a simple application that displays attribute information from buildings and land parcels in an info window when the user clicks the map.

Follow the given steps to complete the exercise:

1. Open the JavaScript sandbox at `https://developers.arcgis.com/javascript/3/sandbox/sandbox.html`.
2. Remove the JavaScript content from the `<script>` tag that I have highlighted as follows:

```
<script>
  var map;

  require(["esri/map", "dojo/domReady!"], function(Map) {
    map = new Map("map", {
      basemap: "topo",  //For full list of ...
      center: [-122.45, 37.75], // longitude, latitude
      zoom: 13
    });
  });
</script>
```

3. Create the variables that you'll use in the application:

```
<script>
   var map;
   var identifyTask, identifyParams;
</script>
```

4. Create the `require()` function that defines the resources you'll use in this application:

```
<script>
  var map;
  var identifyTask, identifyParams;
  require([
  "esri/map", "esri/dijit/Popup",
  "esri/layers/ArcGISDynamicMapServiceLayer",
  "esri/tasks/IdentifyTask",
  "esri/tasks/IdentifyResult",
  "esri/tasks/IdentifyParameters",
  "esri/dijit/InfoWindow",
  "esri/symbols/SimpleFillSymbol",
  "esri/symbols/SimpleLineSymbol",
  "esri/InfoTemplate", "esri/Color",
  "dojo/on", "dojo/domReady!"
  ], function (Map, Popup, ArcGISDynamicMapServiceLayer,
  IdentifyTask, IdentifyResult, IdentifyParameters,
  InfoWindow, SimpleFillSymbol, SimpleLineSymbol,
  InfoTemplate, Color, on) {

  });
</script>
```

5. Create a new instance of the `Map` object, and associate it with a `Popup`:

```
<script>
  var map;
  var identifyTask, identifyParams;
  require([
  "esri/map", "esri/dijit/Popup",
  "esri/layers/ArcGISDynamicMapServiceLayer",
  "esri/tasks/IdentifyTask",
  "esri/tasks/IdentifyResult",
  "esri/tasks/IdentifyParameters",
  "esri/dijit/InfoWindow",
  "esri/symbols/SimpleFillSymbol",
  "esri/symbols/SimpleLineSymbol",
  "esri/InfoTemplate", "esri/Color",
```

```
  "dojo/on", "dojo/domReady!"
  ], function (Map, Popup, ArcGISDynamicMapServiceLayer,
  IdentifyTask, IdentifyResult, IdentifyParameters, InfoWindow,
  SimpleFillSymbol, SimpleLineSymbol, InfoTemplate, Color, on)
{
    //setup the popup window
    var popup = new Popup({
      fillSymbol:
      new SimpleFillSymbol(SimpleFillSymbol.STYLE_SOLID,
      new SimpleLineSymbol(SimpleLineSymbol.STYLE_SOLID,
      new Color([255, 0, 0]), 2),
      new Color([255, 255, 0, 0.25]))
    }, dojo.create("div"));
    // create the map
    map = new Map("map", {
      basemap: "streets",
      center: [-83.275, 42.573],
      zoom: 18,
      infoWindow: popup
    });

  });
</script>
```

6. Create a new dynamic map service layer and add it to the map:

```
// create the map
map = new Map("map", {
    basemap: "streets",
    center: [-83.275, 42.573],
    zoom: 18,
    infoWindow: popup
});

var landBaseLayer = new
ArcGISDynamicMapServiceLayer(
"http://sampleserver3.arcgisonline.com
/ArcGIS/rest/services/
BloomfieldHillsMichigan/Parcels/MapServer",
{opacity:.55});
map.addLayer(landBaseLayer);
});
```

7. Add a `Map.click` event that will trigger a callback function called `executeIdentifyTask` that will respond when the map is clicked:

```
// create the map
map = new Map("map", {
basemap: "streets",
center: [-83.275, 42.573],
zoom: 18,
infoWindow: popup
});
var landBaseLayer = new
ArcGISDynamicMapServiceLayer("http://sampleserver3.arcgisonline
.com/ArcGIS/rest/services/BloomfieldHillsMichigan/Parcels/MapSe
rver",{opacity:.55});
map.addLayer(landBaseLayer);
map.on("click", executeIdentifyTask);
```

8. Create an `IdentifyTask` object:

```
identifyTask = new
IdentifyTask("http://sampleserver3.arcgisonline.com/ArcGIS/
 rest/services/BloomfieldHillsMichigan/Parcels/MapServer");
```

9. Create an `IdentifyParameters` object and set the properties shown in the code as follows:

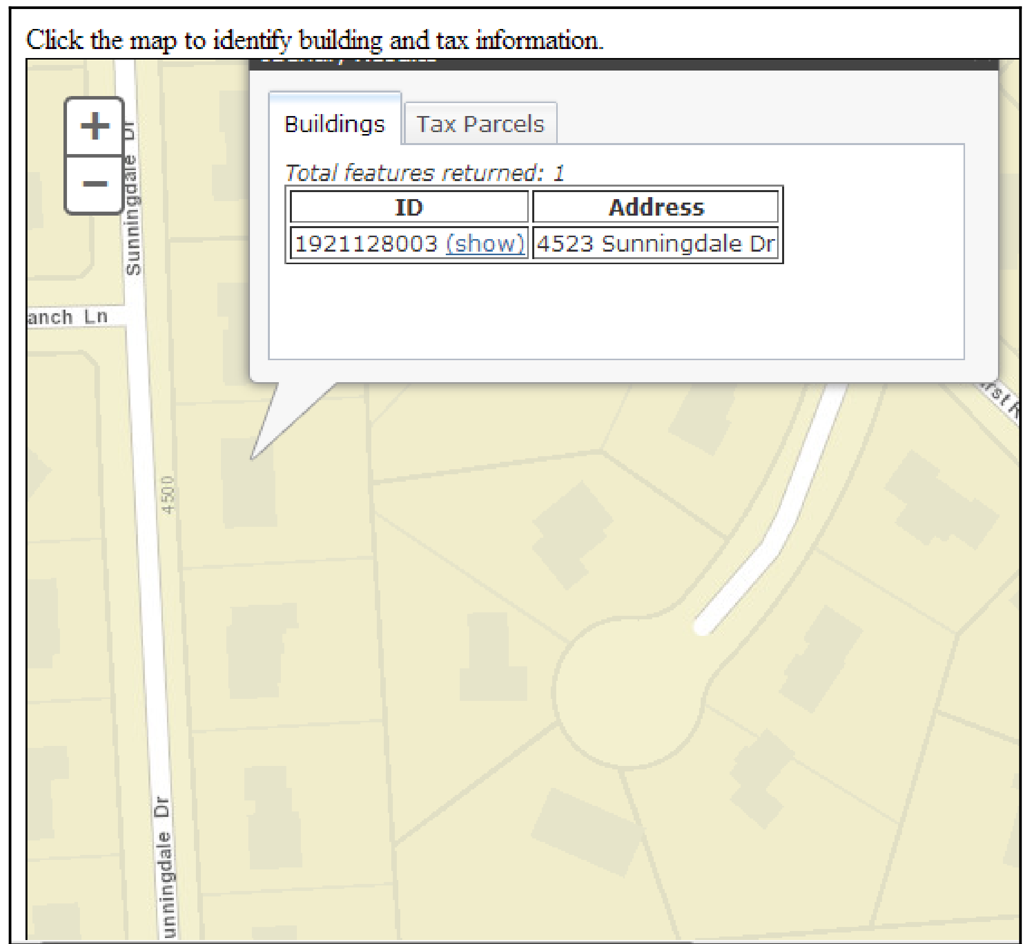
```
identifyTask = new
IdentifyTask("http://sampleserver3.arcgisonline.com/ArcGIS/rest
/services/BloomfieldHillsMichigan/Parcels/MapServer");

identifyParams = new IdentifyParameters();
identifyParams.tolerance = 3;
identifyParams.returnGeometry = true;
identifyParams.layerIds = [0,2];
identifyParams.layerOption =
IdentifyParameters.LAYER_OPTION_ALL;
identifyParams.width  =map.width;
identifyParams.height = map.height;
```

10. Create the `executeIdentifyTask()` function which is the callback function for the `Map.click` event. The `executeIdentifyTask()` function accepts one parameter, which is an instance of the `Event` object. Each event generates an `Event` object, which has various properties. In the case of a `Map.click` event, this `Event` object has a property that contains the point that was clicked. This can be retrieved with the `Event.mapPoint` property and is used to set the `IdentifyParameters.geometry` property. The `IdentifyTask.execute()` method also returns a `Deferred` object. You then add a `callback` function to this `Deferred` object that parses the results. Add the following code to create the `executeIdentifyTask()` function. Place it *outside* the `require()` function:

```
function executeIdentifyTask(evt) {
  identifyParams.geometry = evt.mapPoint;
  identifyParams.mapExtent = map.extent;
  var deferred = identifyTask.execute(identifyParams);
  deferred.addCallback(function (response) {
    // response is an array of identify result objects
    // Let's return an array of features.
    return dojo.map(response, function (result) {
      var feature = result.feature;
      feature.attributes.layerName = result.layerName;
        if (result.layerName === 'Tax Parcels') {
          var template = new InfoTemplate("", "${Postal
          Address} <br/> Owner of record: ${First Owner
Name}");
          feature.setInfoTemplate(template);
        }
        else if (result.layerName === 'Building Footprints') {
          var template = new InfoTemplate("", "Parcel ID:
          ${PARCELID}");
          feature.setInfoTemplate(template);
        }
        return feature;
      });
    });
  map.infoWindow.setFeatures([deferred]);
  map.infoWindow.show(evt.mapPoint);
}
```

11. You may want to review the solution file (`identify.html`) in the `Chapter7` folder of the sample code to verify that your code has been written correctly.

12. Execute the code by clicking the **Run** button and you should see the following output:



Using FindTask to access feature attributes

You can use `FindTask` to search a map service exposed by the ArcGIS Server REST API for a string value. The search can be conducted on a single field of a single layer, on many fields of a layer, or on many fields of many layers. As with the other tasks we've examined, the find operation relies upon complementary objects: `FindParameters`, `FindTask`, and `FindResult`. `FindParameters` serves as the input parameter object, which `FindTask` uses to accomplish its work, and `FindResult` contains the results returned by the task.

FindParameters

`FindParameters` is used to specify the search criteria for a find operation and includes a `searchText` property that is the text that will be searched for, along with properties that specify the fields and layers that will be searched. In addition, setting the `returnGeometry` property to `true` indicates that you want to return the geometry of the features that matched the find operation and this can be used to highlight the results on the map.

The following code example shows how to create a new instance of `FindParameters` and assign values to its properties. Before using any of the objects associated with a find operation, you'll need to import the `esri/tasks/FindTask` and `esri/tasks/FindParameters` modules. The `searchText` property defines the string value to search for, in the fields defined in the `searchFields` property. The layers that will be searched are defined in an array of layer index numbers assigned to the `layerIds` property. The index numbers correspond to the layers in the map service. The `geometry` property defines whether the feature geometries should be returned in the results. If you don't need the feature geometry (that is, you don't intend to plot or otherwise reference the geometry on the map) set the `geometry` property to false:

```
var findParams = new FindParameters();
findParams.searchText = dom.byId("ownerName").value;
findParams.searchFields = ["LEGALDESC","ADDRESS"]; //fields to search
findParams.returnGeometry = true;
findParams.layerIds = [0]; //layers to use in the find
findParams.outSpatialReference = map.spatialReference;
```

You can use the `contains` property to determine whether to look for an exact match of the search text or not. If `contains` is true, it searches for a value that *contains* the string in the `searchText` property. This is a case-insensitive search. If false, it searches for an exact match of the `searchText` string. The exact match is casesensitive.

FindTask

`FindTask`, illustrated in the following figure, executes a find operation against the layers and fields specified in `FindParameters` and returns a `FindResult` object that contains the records which satisfied the query:

```
findTask = new
FindTask("http://sampleserver1.arcgisonline.com/ArcGIS/rest/services/TaxPar
cel/TaxParcelQuery/MapServer/");
findTask.execute(findParams,showResults);

function showResults(results) {
    //This function processes the results
}
```

As with `QueryTask`, you must reference the map service URL that will be used in the find operation, but you with the `FindTask` you reference the entire map service and not a specific layer within the map service as you do for the `QueryTask`. The layers and fields within the map service that the find operation will use are defined in the `FindParameters` object. Once you have instantiated the `FindTask`, you can then call its `execute()` method, passing in the `FindParameters` object as the first parameter, and you can also define optional success and error callback functions. This is illustrated in the preceding code example. The success callback function is passed an instance of `FindResult,` which contains the results of the `find` operation.

FindResults

`FindResult` contains the results of a `FindTask` operation, including the associated features (which are graphics objects you can use to add the results to the map), the layer IDs and names the features belong to, and the field names that contained the search string:

```
function showResults(results) {
        //This function works with an array of FindResult that the task
returns
map.graphics.clear();
```

```
var symbol = new SimpleFillSymbol(SimpleFillSymbol.STYLE_SOLID, new
SimpleLineSymbol(SimpleLineSymbol.STYLE_SOLID, new Color([98,194,204]), 2),
new Color([98,194,204,0.5]));

        //create array of attributes
var items = array.map(results,function(result){
var graphic = result.feature;
graphic.setSymbol(symbol);
map.graphics.add(graphic);
return result.feature.attributes;

});
```

Having retrieved this information, it's up to you to do something with it! For example, you might want to display the attribute data in a Dojo `DataGrid`, or create some sort of report.

Summary

Returning attributes associated with features is one of the most common operations in GIS. ArcGIS Server has two tasks that can help with this: `IdentifyTask` and `FindTask`. `IdentifyTask` is used to return the attributes of a feature that has been clicked on the map. `FindTask` also returns feature attributes but performs a simple attribute query. This query helps you to locate features that contain text in the specified fields that satisfy the query. In the next chapter, you will learn how to locate features associated with an address, by using geocoding.

8

Turning Addresses into Points and Points into Addresses

One feature that is often required in web mapping applications is the ability to search for a street address or place name and have that turned into geographic coordinates that can be highlighted on the map, or manipulated in some other way. This process of turning addresses into points is called *geocoding*. Sometimes, users want to be able to click a map location and have your application tell them which physical address that location relates to. This is known as *reverse geocoding*.

Geocoding is accomplished in ArcGIS Server through the use of locator services which are consumed in the ArcGIS Server JavaScript API through the `Locator` class. As with the other tasks provided by ArcGIS Server, geocoding requires various input parameters, including an `Address` object for address matching or a `Point` object in the case of reverse geocoding. This information is then submitted to the locator service. If there is a match, the service returns an `AddressCandidate` object that contains the information you need to plot the address on the map.

In this chapter, we will cover the following topics:

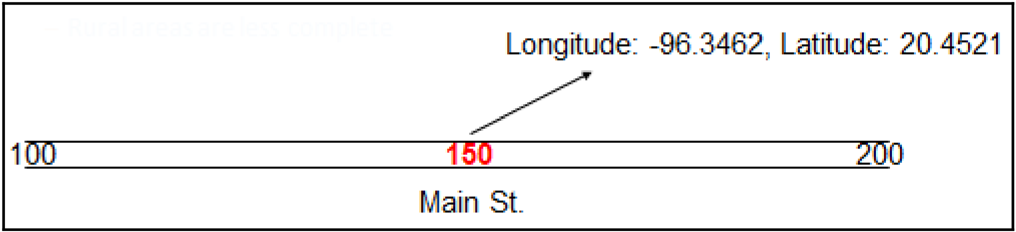
- Introduction to geocoding
- Geocoding with a locator service in the ArcGIS API for JavaScript
- The geocoding process
- The reverse geocoding process
- Practice time with the locator service

Introduction to geocoding

We'll first take a look at an example of geocoding, to give you a better feel for the process. If you have an address located at 150 Main St., you must first geocode the address to determine its geographic coordinates.

If 150 Main St lies on a street segment with an address range of 100 to 200 Main St., the geocoding process interpolates the location of 150 Main St. to be exactly halfway along this street segment. It then assigns 150 Main St. to the geographic location that corresponds to the point halfway between 100 and 200 Main St. This process is described in the following figure.

Now that you have the coordinates for the address, you can then plot it on the map:



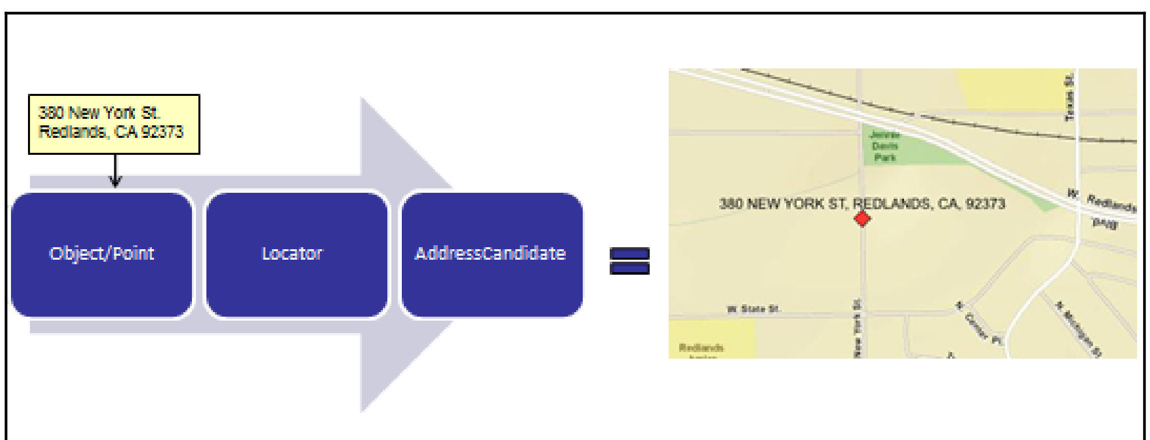
This is the most common way in which street addresses are geocoded. It works best in urban areas where addresses tend to be regularly spaced. However, it is less reliable in areas where addresses are not regularly spaced and for addresses in cul-de-sacs. It is notoriously unreliable in very rural locations.

There are other algorithms that can improve the quality of geocoding, but this is not something we'll get into here.

Geocoding with a locator service in the ArcGIS API for JavaScript

An ArcGIS Server locator service can perform both geocoding and reverse geocoding operations. Using the ArcGIS API for JavaScript, you submit an address to the `Locator` task and have it return the geographic coordinates for any matched locations.

The following figure illustrates this process. An address, defined by a JSON object in JavaScript, is input as a parameter to the `Locator` task object. The `Locator` task geocodes the address and returns the results in an `AddressCandidate` object which can then be displayed as a point on your map. This pattern is the same as the other tasks we've seen in previous chapters where an input object (`Address` object) provides input parameters to the task (`Locator`) which submits the job to ArcGIS Server. A result object (`AddressCandidate`) is then returned to a `callback` function for processing:



Input parameter object

The input parameter object for the `Locator` task is either a JSON address object for geocoding or a `Point` object for reverse geocoding. From a programmatic standpoint the creation of these objects differs. We'll discuss each of the objects in the next section.

Input JSON address object

In straightforward geocoding (address to point), your input to the `Locator` task is a JSON object representing the address or addresses you wish to search for. Each address is specified within this object as a series of name/value pairs which are the object's properties.

The exact properties you must define depend upon the nature of the locator service you are using. In the following example, we provide `street`, `city`, `state`, and `zip`:

```
var address = {
    street: "380 New York",
    city: "Redlands",
    state: "CA",
    zip: "92373"
}
```

Input point object

For reverse geocoding, the input to the `Locator` task is an `esri/geometry/Point` object. This is often generated by the user clicking the map. You retrieve the point at which the user clicked by listening to the `Map.click` event. This event provides a `Point` object that represents the location the user is interested in, and which you can then pass to the `Locator` task.

Sometimes the `Point` object will derive from some other operation that your application performs. For example, your application might select a location based on a query task and then require the address information relating to that location.

Locator object

The `Locator` class contains methods and events that can be used to execute a geocode or reverse geocode operation using the input data you provide.

The `Locator` task constructor needs a URL endpoint of a locator service running on an ArcGIS Server instance. The following code creates a `Locator` task using the `ESRI_Geocode_USA` locator service on ArcGIS Online:

```
var locator = new
Locator("http://sampleserver1.arcgisonline.com/ArcGIS/rest/services/Locator
s/ESRI_Geocode_USA/GeocodeServer")
```

Once you have an instance of `Locator`, you call the `addressToLocations()` method to geocode an address or the `locationToAddress()` method to perform a reverse geocode.

When the operation completes, an event is fired. In the case of an address geocode, the event is `address-to-locations-complete`. For reverse geocoding, the event is `on-location-to-address-complete`. Both events return `AddressCandidate` objects to the function that handles the event.

The AddressCandidate object

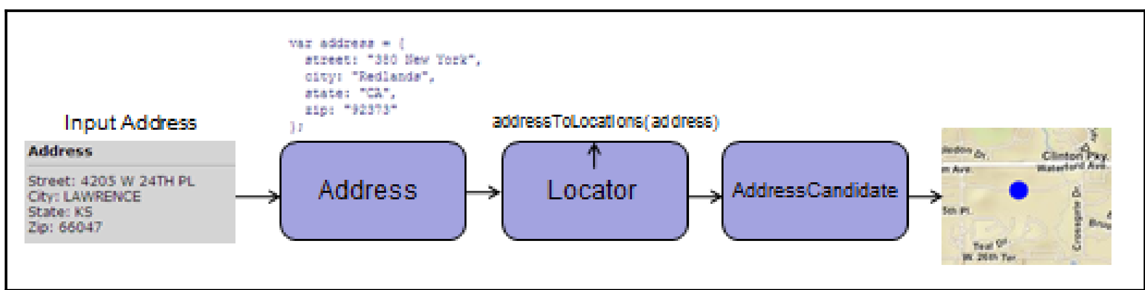
Each `AddressCandidate` object contains one result from the geocoding operation. This object has several properties, including the address, attributes, location, and score.

The `attributes` property contains name/value pairs of field names and values. The location is the *x*, *y* coordinate of the candidate address, and the `score` property is a numeric value between 0-100 that indicates the quality of the match. A higher score represents a better match.

The geocoding process

Let's summarize the process for geocoding an address with the ArcGIS API for JavaScript.

First, you create a `Locator` task object by referencing the URL of a locator task running on an ArcGIS Server instance. You then create the input address as a JSON object and submit it to the `Locator` task using the `addressToLocations()` method. When the operation completes, it returns an *array* of `AddressCandidate` objects which you can then plot on the map. It's up to you to decide which element in the array to use. Normally, it's the one with the best score:



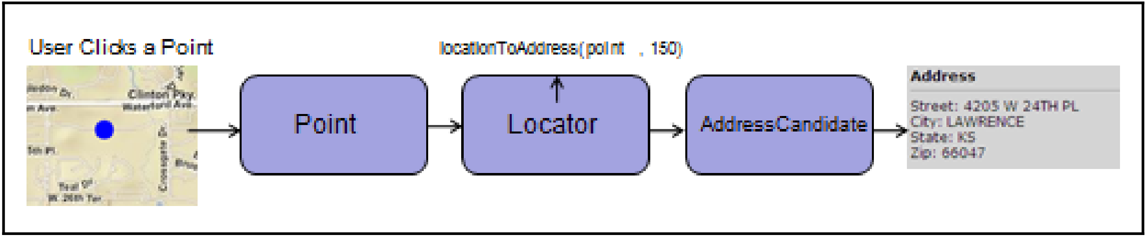
Note that, for batch geocoding operations, you can use the `addressesToLocations()` method, which allows you to specify multiple input addresses.

The reverse geocoding process

The reverse geocoding process is similar, but requires a different input and returns the results in a slightly different format.

This process also uses a `Locator` task object that references a URL to a locator service. You create a `Point` geometry object, either as the result of the user clicking the map, or some other event generated by your application.

You then submit the `Point` object to the `Locator` task using the task's `locationToAddress()` method, together along with a value that represents distance. The `distance` property, supplied in meters, determines the radius in which the `Locator` will attempt to find an address. If the locator service finds an address within the specified radius, it returns a single `AddressCandidate` object to your `callback` function:



Practice time with the locator service

In this exercise you will learn how to use the `Locator` task to geocode addresses using a locator service provided by ArcGIS Online at `https://geocode.arcgis.com`. You will enable a user to search for an address, find the top match, and display on the map.

Start by examining the contents of the `geocoding-begin.html` file in the `Chapter08` folder of the sample code in your text editor of choice. Some of the code for this exercise has been prewritten for you so that you can focus on the geocoding functionality. The application in its current state requests all the modules that the exercise requires, and displays a map and a text box that allows your users to search for an address, as shown in the following diagram:

Once you have obtained the `geocoding-begin.html` file from the sample code, open it in a web browser and then follow the given steps:

1. Click the **Locate** button next to the text box and you'll see that nothing happens. It is up to you to provide the address search functionality.
2. Visit the ArcGIS API for JavaScript Sandbox at `https://developers.arcgis.com/javascript/3/sandbox/sandbox.html`. Copy and paste the code from the `geocoding-begin.html` page so that it completely replaces the code currently in the Sandbox.

3. Add a variable declaration called `locator` that will store the `Locator` task object:

```
var map, locator;
```

4. Just under the code that creates the `Map` object, instantiate the `Locator` task by supplying the URL endpoint of the world geocoding service at ArcGIS Online:

```
locator = new Locator(
"https://geocode.arcgis.com/arcgis/rest/services/World/GeocodeS
erver");
```

5. Next, you need some way to trigger the geocoding functionality when the user clicks the **Locate** button next to the search text box. Enter the following line of code after the line that instantiates the `Locator` task. This looks up the "locate" button in the Dojo `dijit` registry and listens to the button's `click` event. When the button is clicked, it calls a function called `locate()`:

```
registry.byId("locate").on("click", locate);
```

6. Now you will define the `locate()` function. The purpose of this function is to take the search address entered by the user and package it in such a way that it can be supplied as an input parameter to the `Locator` task. The geocoding service that we're using allows a very simple address search by supplying the complete address to search for in a single object property called `SingleLine`. Alternatively, you could provide this information in multiple fields for increased accuracy, but we'll keep things simple here. For more information on the parameters accepted by this particular locator service, see: `https://developers.arcgis.com/rest/geocode/api-reference/geocoding-find-address-candidates.htm`. Our code will take the address from the text box and create an object from it, which can then be used in the `addresses` property of the object we supply to the task. We'll also specify that we want every field in the results returned to our application even though, in this example, we're only really interested in the geometry of the first result. Finally, we execute the task's `addressToLocations()` method using our object as an input parameter. Create the `locate()` function as shown in the following code:

```
function locate() {
    var address = {
        SingleLine: dom.byId("address").value
    };
    var options = {
        address: address,
        outFields: ["*"]
```

```
        };
        locator.addressToLocations(options);
    }
```

7. Next, we need to handle the `Locator` task's `address-to-locations-complete` event, so that our application knows when the results of the geocoding operation are available. Create the event handler with an anonymous `callback` function as follows:

```
    locator.on("address-to-locations-complete", function (evt) {

    });
```

8. The first thing we want to do within this event handler is to clear any existing geocoding results from the map:

```
    map.graphics.clear();
```

9. Upon successful completion, the `addressToLocations()` method returns an array of `AddressCandidate` objects. We're only interested in the first one in this exercise, so let's call that one `topMatch`:

```
    var topMatch = evt.addresses[0];
```

10. We want to display this result on the map, so we'll define a `SimpleMarkerSymbol` for it:

```
    var symbol = new
    SimpleMarkerSymbol(SimpleMarkerSymbol.STYLE_CIRCLE, 20,
        new SimpleLineSymbol(SimpleLineSymbol.STYLE_SOLID,
            new Color([0, 0, 0]), 2),
        new Color([255, 0, 0, 0.6]));
```

11. Our locator service returns geocoding results in the WGS84 geographic projection. Our ArcGIS Online Topo basemap's spatial reference however, is Web Mercator. So we need to reproject our result. There is a handy module called `esri/geometry/webMercatorUtils` we can use to convert geometries from geographic to Web Mercator and vice versa, and we'll use its `geographicToWebMercator()` method to perform the conversion:

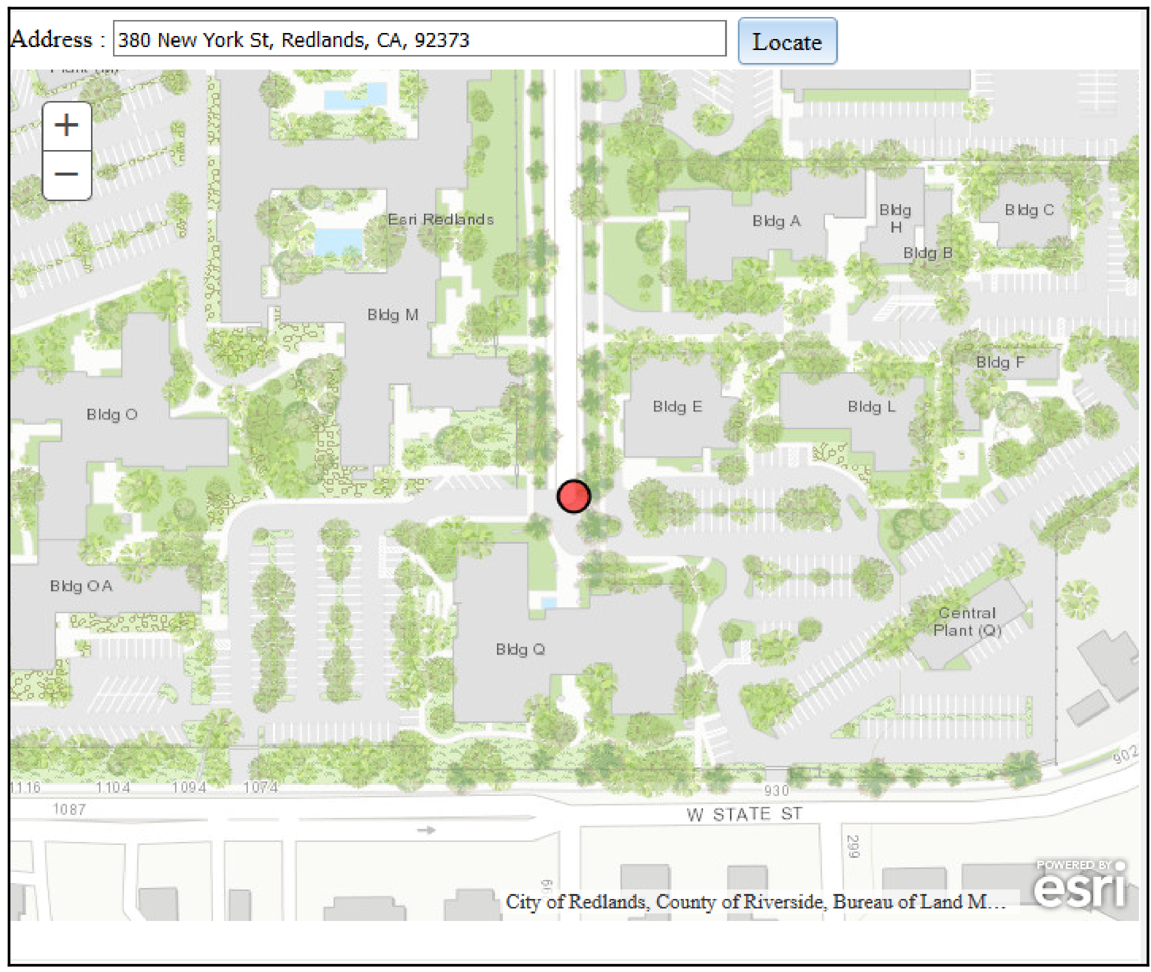
```
    var point =
    webMercatorUtils.geographicToWebMercator(topMatch.location);
```

12. Now we have a `Point` geometry, we can create a graphic from it, add it to the map, and then zoom and center the map to highlight our result. Enter the following code in the event handler:

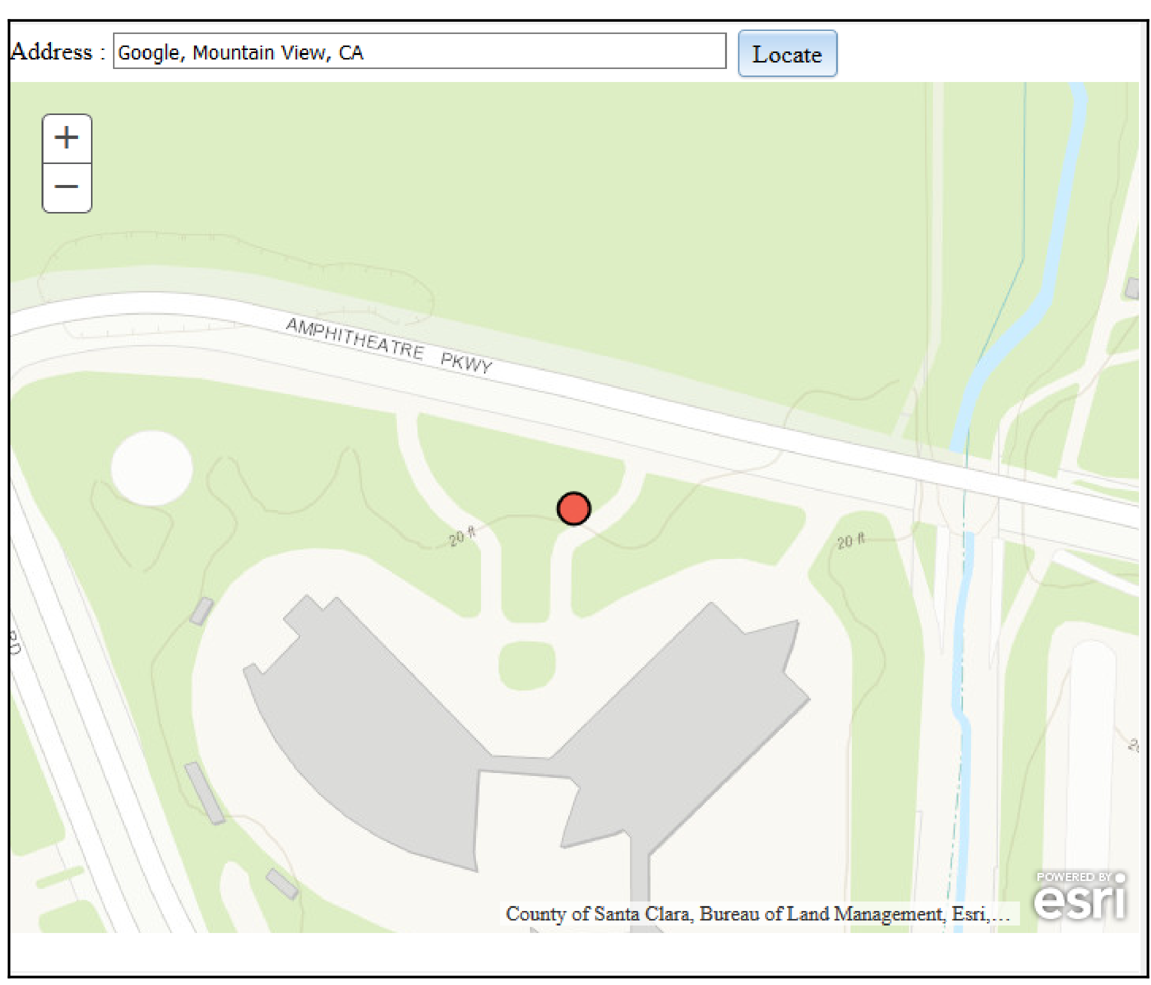
```
var locationGraphic = new Graphic(point, symbol);
map.graphics.add(locationGraphic);
map.centerAndZoom(locationGraphic.geometry, 18);
```

13. Click the **Refresh** button in the Sandbox and, superficially, our application appears exactly the same as it did at the start of the exercise:

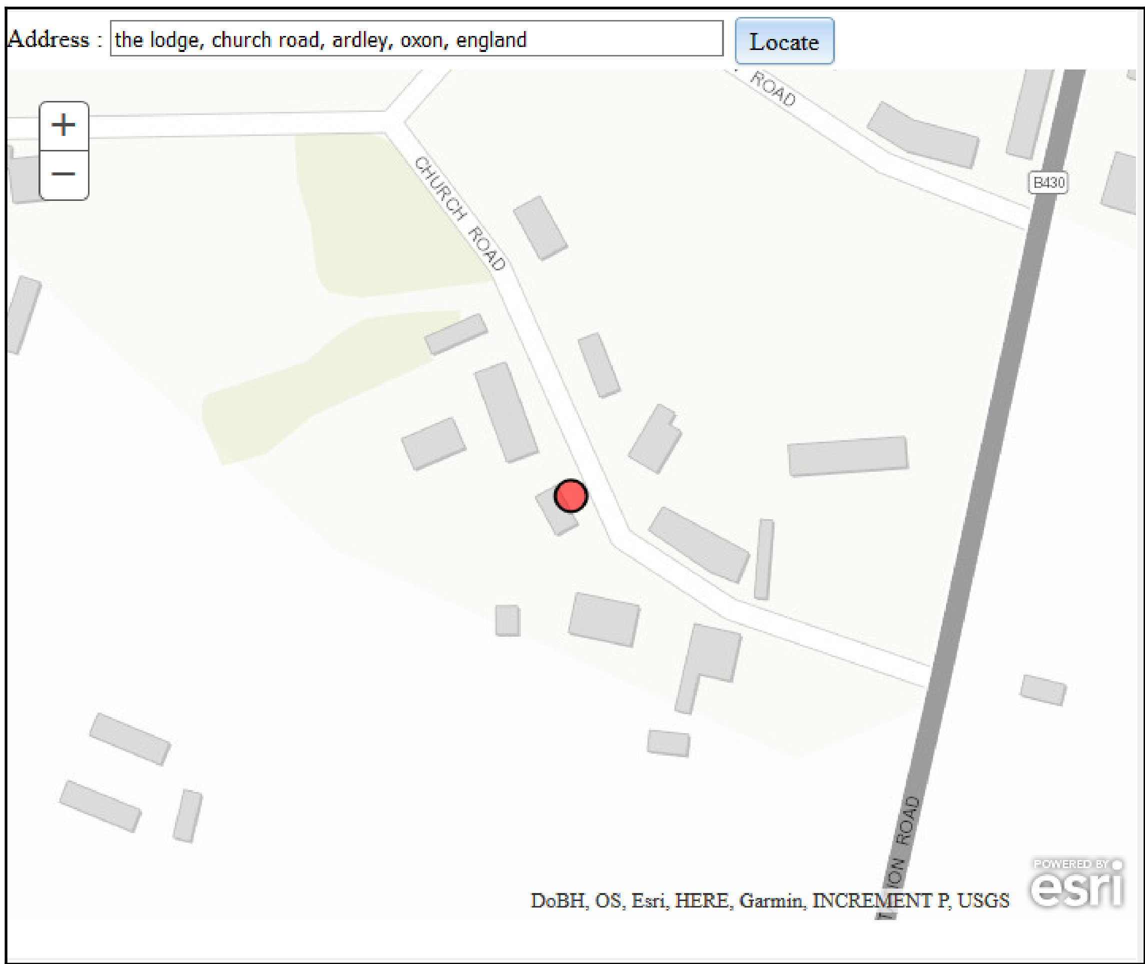
14. However, if we have written our code correctly, our geocoding functionality should now work. Keep the default address in the text box (Esri's head office in Redlands, California) and click the **Locate** button. You should see a marker appear in the middle of the map that is now centred over Esri's campus:



15. Try using the application to find other addresses or place names. For example, entering "**Google, Mountain View, CA**" locates Google's HQ:



16. Try searching for your home address. Depending on where you live, your results might vary. However, I (Mark Lewin) live in rural England where there are very few house numbers and all but the most experienced mailmen get hopelessly confused, and even so, my results were spot on:



The Search widget

Another, potentially much simpler way of incorporating geocoding functionality in your web mapping applications is to use the Search widget.

The Search widget replaces the earlier Geocoder widget which was deprecated in version 3.13 of the API, and offers additional functionality in that it can be used for searching across multiple data sources, not just locator services but feature layers too.

In addition, it provides a much nicer interface for the user in that it can provide suggestions while the user types their search criteria. For this to work, you need to be using a locator service running on an ArcGIS Server instance of 10.3 or later, with the `suggest` capability loaded.

The Search widget works with both geographic and Web Mercator spatial references. If your map uses another spatial reference, you must set a default geometry service so that the server can re-project the input geometries for you on the fly. You can set a default geometry service by using the `esri/config` module as follows:

```
require(["esri/config"], function(esriConfig) {
  esriConfig.defaults.geometryService =
"http://www.example.com/arcgis/rest/services/Utilities/Geometry/GeometrySer
ver";
});
```

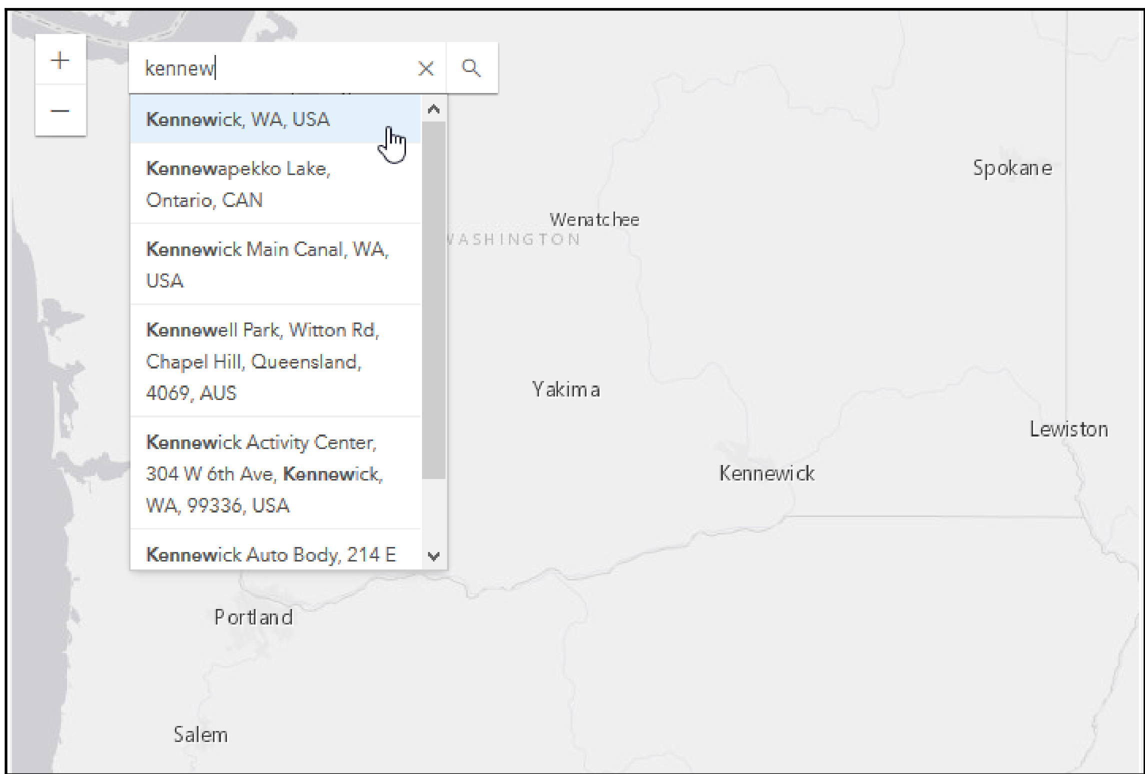
There are many properties and methods on the Search widget and we won't get into most of them here. However, if your application relies heavily on search and geocoding, you will find that the Search widget is very adaptable to your specific needs.

To add a very basic Search widget to your map, all you need to do is `require` the `esri/dijit/Search` module, then instantiate it with references to the map and the `div` on the page where the widget must appear:

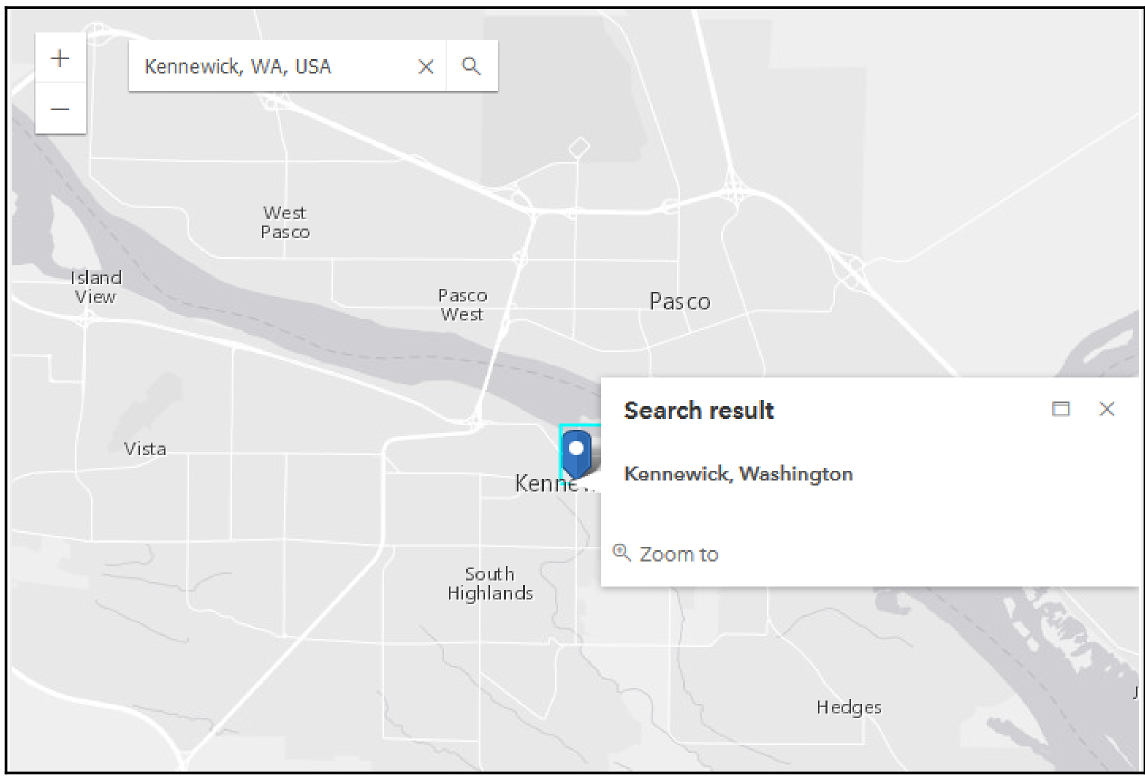
```
require([

  "esri/map",
  "esri/dijit/Search",
  "dojo/domReady!"

], function (Map, Search) {
   var map = new Map("map", {
      basemap: "gray",
      center: [-120.435, 46.159], // lon, lat
      zoom: 7
   });
```

```
            var search = new Search({
               map: map
            }, "search");
            search.startup();

        });
```

The search box then displays and provides search suggestions as you type. For example, in the following output I start typing `kennew` to search for the town of **Kennewick** in Washington state, and see the following suggestions:



When I click on one of the search suggestions, the map displays a marker and a pop-up window that allows me to interact further with the results:



This is the default behavior and requires no further code than that already shown. However, this barely touches the surface of what this very useful widget is capable of, so I urge you to check out the documentation and samples.

Find out more about the Search widget in the ArcGIS API for JavaScript documentation at `https://developers.arcgis.com/javascript/3/jsapi/search-amd.html`.

Summary

The `Locator` task allows you to interact with an ArcGIS Server locator service to perform geocoding and reverse geocoding operations. It does this using the `addressToLocation()` and `locationToAddress()` methods. The first expects a JSON object that specifies the address to search for, and the second expects both a `Point` location around which to search and a numerical distance.

When you execute the appropriate method for the type of geocoding operation you require, the results are returned in `AddressCandidate` objects which you can process and plot the results.

This workflow should now seem quite familiar to you, as it is very similar to the other tasks you have used in the ArcGIS API for JavaScript.

In this chapter, we learned how to identify locations. In the next chapter, we'll be able to tell users how to reach those locations by using the API's direction and routing capabilities.

9
Directions and Routing

The ArcGIS API for JavaScript's directions and routing capabilities are provided by network analysis services running on ArcGIS Server. In order to create and publish a network service, you must have the Network Analyst extension in ArcGIS Pro. However, you can use third-party services in your web mapping application, such as those we reference in this chapter, without a license.

These services enable you to model transportation networks so that you can perform various analyses, such as finding the best route from one address to another, locating the closest school to your home, identifying a service area around potential sites for a new delivery department, or managing a delivery schedule with a fleet of service vehicles.

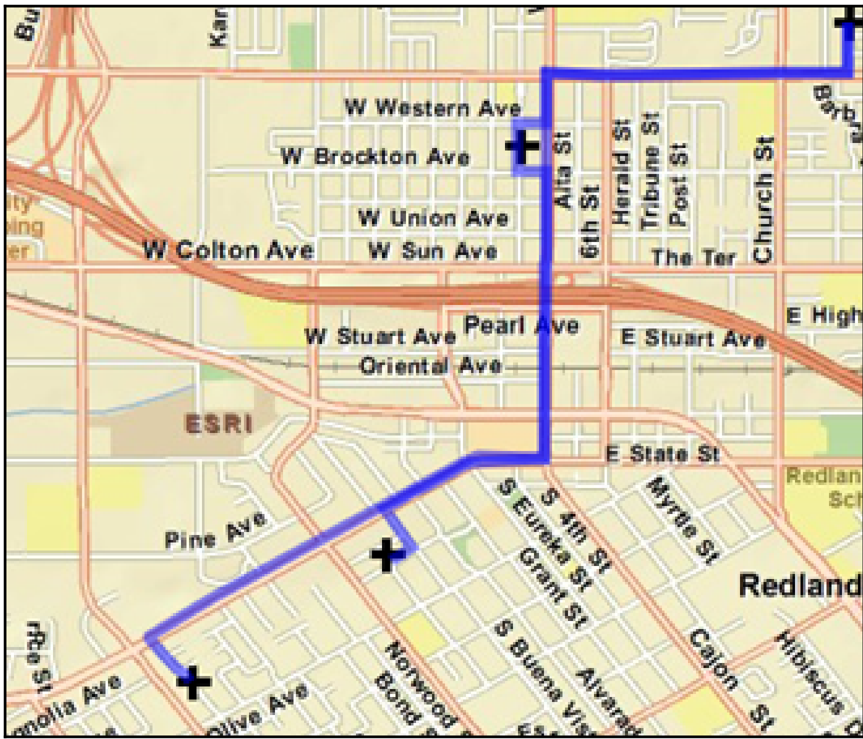
There are three main tasks that network analysis services can perform, that you access via the appropriate task objects in the ArcGIS API for JavaScript. These are routing, closest facility, and service area. We'll examine each of the service types in this chapter, and we'll also have a look at the Directions widget, which makes it easy to incorporate navigation functionality in your web maps.

In this chapter, we will cover the following topics:

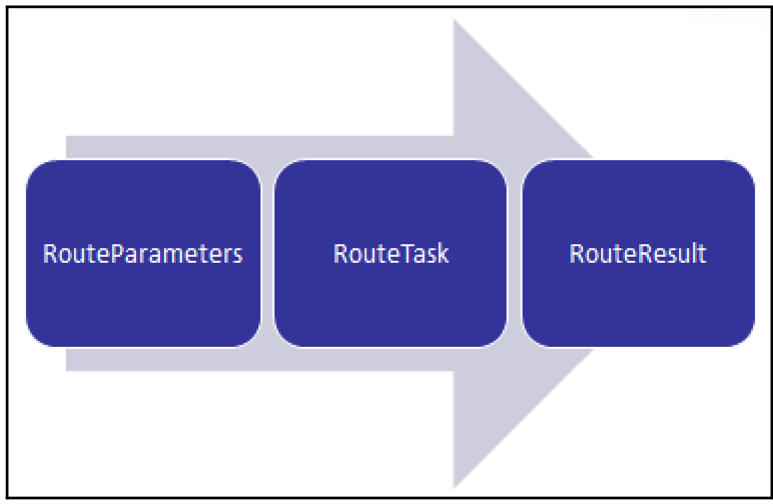
- Routing task
- Practice time with routing
- ClosestFacility task
- ServiceArea task
- Using the Directions widget

Routing task

You can use a `RouteTask` object to find routes between two or more locations, and optionally to get driving directions. The `RouteTask` uses a network analysis service to calculate the routes, which can be both simple (what's the quickest way to get from A to B?) and complex (involving multiple stops, barriers, and time windows). The `RouteTask` uses a least-cost path between multiple locations in a network. Impedance on the network can include time and distance variables:



As with the other tasks we have examined in this book so far, routing is accomplished through a series of objects including `RouteParameters`, `RouteTask`, and `RouteResult`:



The `RouteParameters` object provides the input parameters to `RouteTask` which executes the routing operation. Results are returned from ArcGIS Server in the form of a `RouteResult` object.

`RouteParameters` can include stop and barrier locations, impedance factors, whether or not to return driving directions, and many other options, too many to go into in detail here. We'll just cover the basics in this chapter, but we urge you to delve into the full capabilities of the Route Task by reading the API documentation.

The following code sample demonstrates how to create an instance of `RouteParameters`, add a series of stops and define the output spatial reference:

```
routeParams = new RouteParameters();
routeParams.stops = new FeatureSet();
routeParams.outSpatialReference = {wkid:4326};
routeParams.stops.features.push(stop1);
routeParams.stops.features.push(stop2);
```

Once you have got your `RouteParameters` set up correctly, you're ready to implement your `RouteTask` and instruct it to perform the analysis.

The constructor for `RouteTask` expects the URL of the network service endpoint to use for the analysis. Once you have instantiated the `RouteTask` you can kick things off by calling its `solve()` method and passing it the `RouteParameters`, a callback function for successful completion, and optionally a callback for any error condition:

```
routeParams = new RouteParameters();
routeParams.stops = new FeatureSet();
routeParams.outSpatialReference = {wkid:4326};
routeParams.stops.features.push(stop1);
routeParams.stops.features.push(stop2);
routeTask.solve(routeParams, routeSuccess, routeFailure);
```

If the operation is successful, the network analysis service returns a `RouteResult` object to the callback function you specified in the call to `RouteTask.solve()`. This is where you will write the code that unpacks the results and displays them to the user.

The type of data returned is largely dependent upon the options you set in the `RouteParameters` object. One of the most important properties on `RouteParameters` is the `stops` property. These are the points to be included in the analysis of the best route between points. Stops are defined as either an instance of `DataLayer` or `FeatureSet`, and are a set of stops to be included in the analysis.

The concept of barriers is also important in routing operations. Barriers restrict movement when planning a route. Barriers can include things like a car accident, construction work on a street segment, or other more permanent delays such as railroad crossings. Barriers are defined as either a `FeatureSet` or `DataLayer` and specified through the `RouteParameters.barriers` property. The following code example shows an example of how you can write code to create barriers in your `RouteParameters` object:

```
var routeParameters = new RouteParameters();

   //Add barriers as a FeatureSet
   routeParameters.barriers = new FeatureSet();
   routeParameters.barriers.features.push(
     map.graphics.add(
       new Graphic(
         evt.mapPoint,
         barrierSymbol
       )
     )
   );
```

If you set `RouteParameters.returnDirections` to `true`, you can return step-by-step directions in your `RouteResult`. You can also set properties that affect the language in which directions are output (`RouteParameters.directionsLanguage`), the units for road segment lengths, (`RouteParameters.directionsLengthUnits`), the verbosity of directions (`RouteParameters.directionsOutputType`), and more.

As well as directions, your results can include the route between points, the route name, and an array of stops.

You can also define if the task should fail if one stop is unreachable using the `RouteParameters.ignoreInvalidLocations` property. This property can be set to `true` or `false`. You can also introduce time into the analysis through properties such as `RouteParameters.startTime` which specifies the time the route begins and `RouteParameters.useTimeWindows` which defines that time windows should be used in the analysis.

Like we said, there's a lot you can do with the `RouteTask` and its `RouteParameters`. For more information, consult the online help: `https://developers.arcgis.com/javascript/3/jsapi/routetask-amd.html`.

Practice time with routing

In this exercise, you will learn how to implement routing in your applications. You'll create an instance of `RouteParameters`, add stops by allowing the user to click points on a map, and solve the route. The returned route will be displayed as a line symbol on the map:

1. Open the JavaScript Sandbox at `https://developers.arcgis.com/javascript/3/sandbox/sandbox.html`.
2. Remove all the JavaScript content from the `<script>` tag as shown in the following:

```
<script>
  var map;

  require(["esri/map", "dojo/domReady!"], function(Map) {
    map = new Map("map", {
      basemap: "topo",  //For full list of ...
      center: [-122.45, 37.75], // longitude, latitude
      zoom: 13
    });
  });
```

```
</script>
```

3. Create your `require()` function and import the following modules:

```
<script>
    require([
        "esri/map",
        "esri/urlUtils",
        "esri/tasks/RouteParameters",
        "esri/tasks/RouteTask",
        "esri/tasks/FeatureSet",
        "esri/symbols/SimpleMarkerSymbol",
        "esri/symbols/SimpleLineSymbol",
        "esri/graphic",
        "dojo/_base/Color"
    ],
    function (Map, urlUtils, RouteParameters, RouteTask,
        FeatureSet, SimpleMarkerSymbol,
        SimpleLineSymbol, Graphic, Color) {


    });
</script>
```

4. In the body of the `require()` callback, add the following lines of code. Note that this application requires access to resources on a different web server than the application is being served from. Generally speaking, this is not allowed, and results in a security exception. However, if the remote server supports **CORS** (**Cross Origin Resource Sharing**), there is no problem. The servers that host the map services we have been using so far all support CORS, but the network service we will use in this practice does not. Therefore, we must specifically tell our application that it's fine to access the host server, which we can do with the `esri/urlUtils` module as shown in the following:

```
urlUtils.addProxyRule({
    urlPrefix: "route.arcgis.com",
    proxyUrl: "/sproxy/"
});
```

The usual approach for production applications is to download and install a proxy page on your local web server, and configure it to allow resource sharing with specific servers. For more information on this approach and for a detailed explanation of CORS, see the help topic: `https://developers.arcgis.com/javascript/3/jshelp/ags_proxy.html`.

5. Inside the `require()` function, create the `Map` object as seen in the following and define variables to hold the route objects and the symbols you will use for display purposes:

```
<script>
    require([
        "esri/map",
        "esri/urlUtils",
        "esri/tasks/RouteParameters",
        "esri/tasks/RouteTask",
        "esri/tasks/FeatureSet",
        "esri/symbols/SimpleMarkerSymbol",
        "esri/symbols/SimpleLineSymbol",
        "esri/graphic",
        "dojo/_base/Color"
    ],
    function (Map, urlUtils, RouteParameters, RouteTask,
        FeatureSet, SimpleMarkerSymbol,
        SimpleLineSymbol, Graphic, Color) {
        urlUtils.addProxyRule({
           urlPrefix: "route.arcgis.com",
           proxyUrl: "/sproxy/"
        });

        var map, routeTask, routeParams;
        var stopSymbol, routeSymbol, lastStop;

        map = new Map("mapDiv", {
            basemap: "streets",
            center: [-123.379, 48.418], //long, lat
            zoom: 14
        });

    });
</script>
```

6. Just under the code block that created the `Map` object add an event handler for the `Map.click()` event. This should trigger a function called `addStop()`:

```
map = new Map("mapDiv", {
    basemap: "streets",
    center: [-123.379, 48.418], //long, lat
    zoom: 14
});
map.on("click", addStop);
```

7. Instantiate the `RouteTask` object. It should reference the Network Service shown:

```
routeTask = new RouteTask
("http://sampleserver3.arcgisonline.com/ArcGIS/rest/services/Ne
twork/USA/NAServer");
```

There are some very useful routing services on ArcGIS Online. Find out more at: `https://route.arcgis.com/arcgis/index.html`.

8. Instantiate the `RouteParameters` object. For its stops property, create and assign a new, empty `FeatureSet` object. Set its `outputSpatialReference` property to `4326`:

```
routeTask = new
RouteTask("https://route.arcgis.com/arcgis/rest/services/World/
Route/NAServer/Route_World");
routeParams = new RouteParameters();
routeParams.stops = new FeatureSet();
routeParams.outSpatialReference = {"wkid": 4326}
```

9. Add event handlers for completion of the `RouteTask.solve-complete()` event and `RouteTask.error()` event. The successful completion of a routing task should trigger the execution of a function called `showRoute()`. Any errors should trigger the execution of a function called `errorHandler()`:

```
routeParams = new RouteParameters();
routeParams.stops = new FeatureSet();
routeParams.outSpatialReference = { "wkid": 4326 };

routeTask.on("solve-complete", showRoute);
routeTask.on("error", errorHandler);
```

We could specify these handlers as callback functions when we execute `RouteTask.solve()`. We've opted for this approach here just to make the code a little easier to read.

10. Create symbol objects for start and end points of the route as well as the line that defines the route between those points. Add lines of code just under the ones you entered in the preceding step:

```
stopSymbol = new SimpleMarkerSymbol().setStyle(
    SimpleMarkerSymbol.STYLE_CROSS);
stopSymbol.setSize(15);
stopSymbol.outline.setWidth(4);
routeSymbol = new SimpleLineSymbol();
routeSymbol.setColor(
    new Color([0, 0, 255, 0.5]));
routeSymbol.setWidth(5);
```

11. Create the `addStop()` function which will be triggered when the user clicks the map. This function will accept an `Event` object as its only parameter. The point clicked on the map can be extracted from this object. This function will add a point graphic to the map, add the graphic to the `RouteParameters.stops` property, and on the second map, clicking it will call the `RouteTask.solve()` method, passing in an instance of `RouteParameters`:

```
function addStop(evt) {
    var stop = map.graphics.add(new Graphic(evt.mapPoint,
stopSymbol));
    routeParams.stops.features.push(stop);

    if (routeParams.stops.features.length >= 2) {
        routeTask.solve(routeParams);
        lastStop = routeParams.stops.features.splice(0, 1)[0];
    }
}
```

12. Create the `showRoute()` function, which accepts an instance of `RouteResult`. The only thing you need to do in this is function display the route from the `RouteResult` object passed into this function as a polyline in the `GraphicsLayer`:

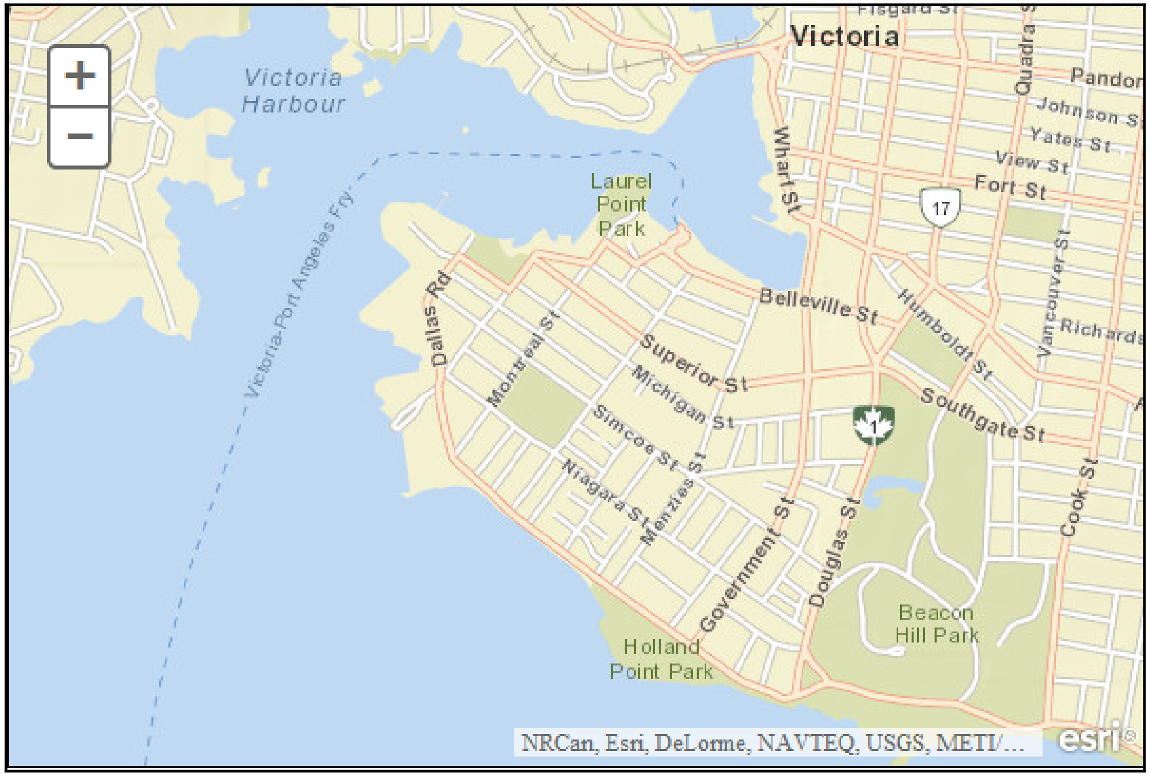
```
function showRoute(solveResult) {
map.graphics.add(solveResult.result.routeResults[0].route.setSy
mbol(routeSymbol));
}
```

13. Finally, handle any error conditions that might arise. The `errorHandler()` function should display an error message to the user and remove any left-over graphics from a previous successful execution:

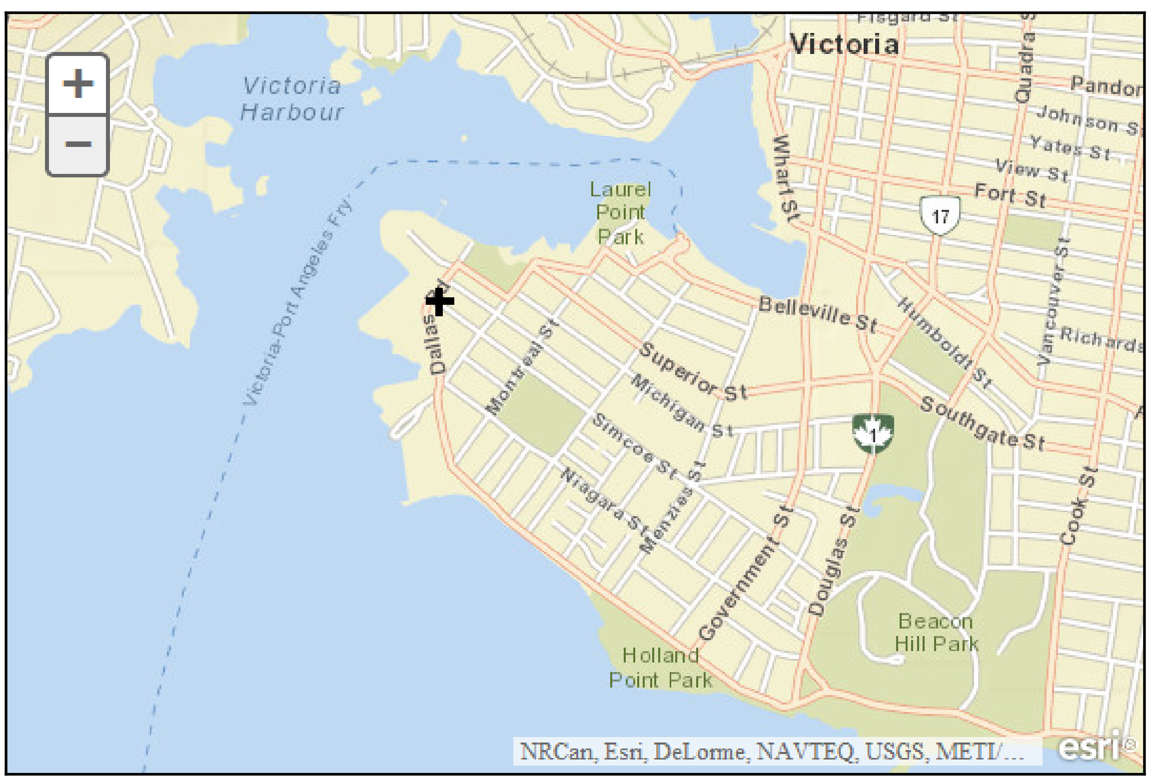
```
function errorHandler(err) {
    alert("An error occured\n" +
        err.message + "\n" +
        err.details.join("\n"));
    routeParams.stops.features.splice(0, 0, lastStop);
}
```

14. You may want to review the solution file (`routing.html`) in your `ArcGISJavaScriptAPI` folder to verify that your code has been written correctly.

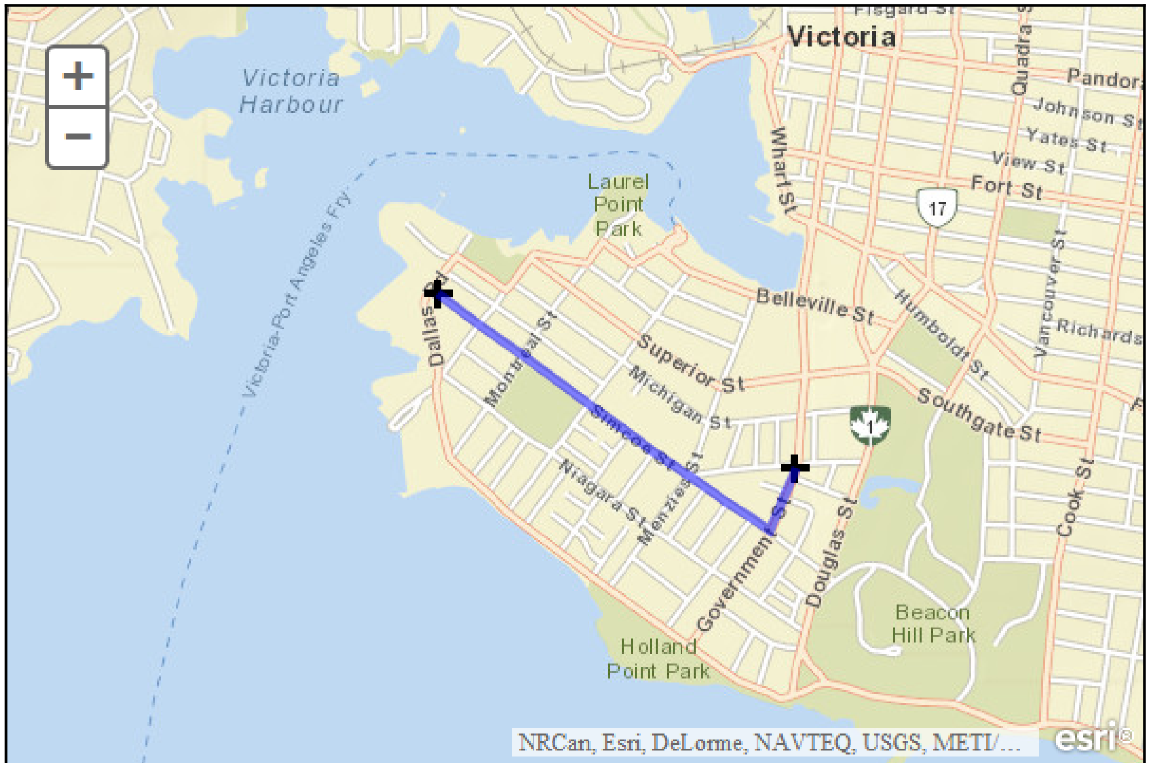
15. Click the **Refresh** button. You should see the following map. If not, you may need to recheck your code for accuracy:



16. Click somewhere on the map. You should see a point marker.



17. Click another point somewhere on the map. This should display a second marker along with the best route between the two points as seen in the following screenshot:



The Directions widget

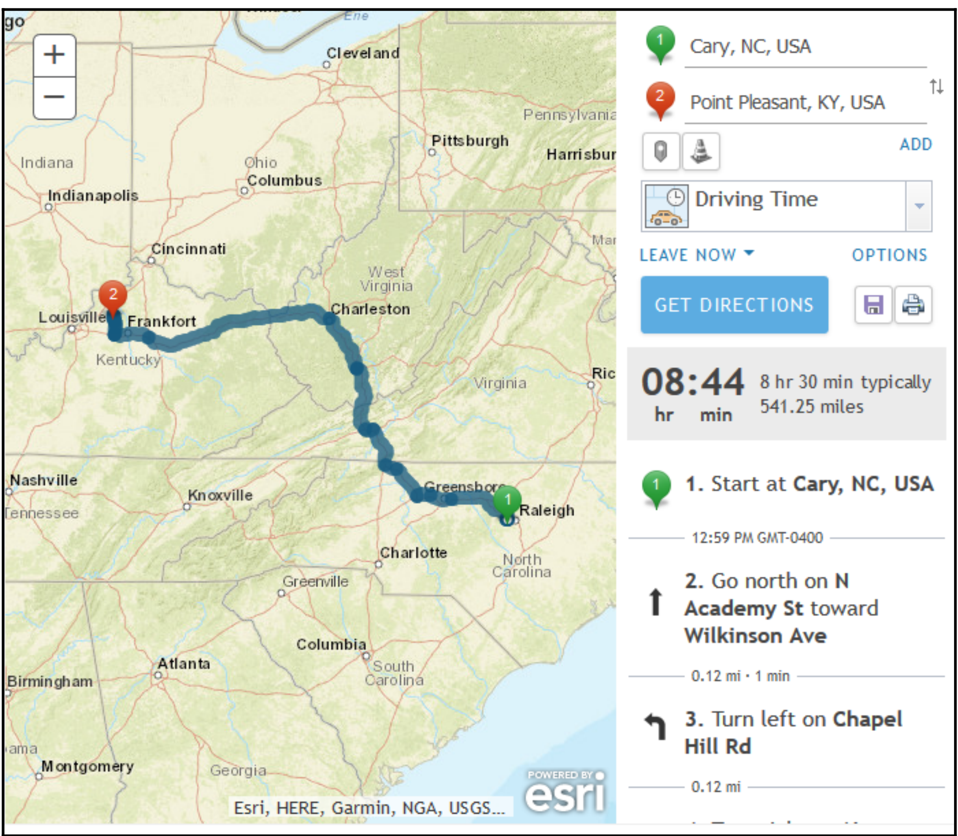
Now that you have seen what goes on behind the scenes, let us introduce you to a really useful widget that takes some of the complexity away when adding routing capabilities to your application.

By default, the Directions widget accesses the `http://route.arcgis.com/arcgis/rest/services/World/Route/NAServer/Route_World` network service to provide routing functionality, but you can configure it to use another network service if you wish. It also accesses another service to provide up-to-date traffic information.

The widget analyzes the network service to determine what capabilities it has, and automatically builds the user interface for you. It deals with gathering the input parameters, executing the task, capturing the results and displaying it to the user. All you have to do is add the `esri/dijit/Directions` module to your application, and instantiate it as follows:

```
var directions = new Directions({
    map: map,
    showSaveButton: true
},"dir");
directions.startup();
```

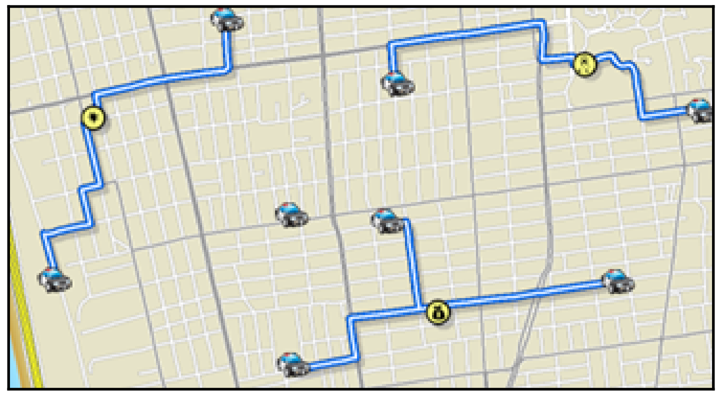
The minimal options I have set in the preceding constructor result in the following rather sophisticated interface, which provides full directions and the ability to save or print the result:



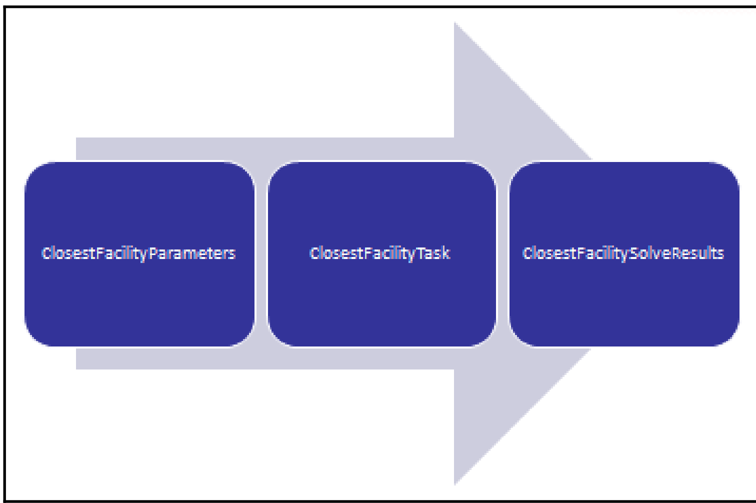
ClosestFacility Task

The `ClosestFacility` task helps you find the closest facilities around any locations (known as *incidents*) on a network.

When finding closest facilities, you can specify how many to find and whether the direction of travel is toward or away from them. The closest facility solver displays the best routes between incidents and facilities, reports their travel costs, and returns the directions to each facility.



The classes involved in solving closest facility operations include **ClosestFacilityParameters**, **ClosestFacilityTask**, and **ClosestFacilitySolveResults**:



`ClosestFacillityParameters` includes input parameters such as the default cutoff, whether or not to return incidents, routes, directions, and more. These parameters are used as input to the `ClosestFacilityTask` which contains a `solve()` method. Finally, results are passed from ArcGIS Server back to the client in the form of a `ClosestFacilitySolveResults` object.

The `ClosestFacilityParameters` object is used as input to `ClosestFacilityTask`.

Let's look at some of the more commonly-used properties of the `ClosestFacilityParameters` object.

The `incidents` and `facilities` properties are used to set the locations for the analysis.

The data returned by the task can be controlled through the `returnIncidents`, `returnRoutes`, and `returnDirections` properties, which are simply true or false values indicating whether the information that each property relates to should be returned in the results.

The `travelDirection` parameter specifies whether travel should be to or from the facility and `defaultCutoff` is the cutoff value beyond which the analysis will stop traversing. The following code example shows how to create an instance of `ClosestFacilityParameters` and apply various properties:

```
params = new ClosestFacilityParameters();
params.defaultCutoff = 3.0;
params.returnIncidents = false;
params.returnRoutes = true;
params.returnDirections = true;
```

Like the `RouteTask`, when you create a new instance of `ClosestFacilityTask` you will need to reference a network analysis service URL. Once created, the `ClosestFacilityTask` accepts the input parameters provided by `ClosestFacilityParameters` and submits them to a network analysis service using the `solve()` method.

The `solve()` method also accepts callback and error callback functions. For brevity, only the success callback function `processResults()` is used as follows:

```
cfTask = new
ClosestFacilityTask("http://<domain>/arcgis/rest/services/network/ClosestFa
cility");
params = new ClosestFacilityParameters();
params.defaultCutoff = 3.0;
params.returnIncidents = false;
params.returnRoutes = true;
params.returnDirections = true;
cfTask.solve(params, processResults);
```

The `ClosestFacilityTask` operation returns its results as a `ClosestFacilitySolveResult` object. This object contains various properties such as `incidents` (an array of points denoting the incidents), `facilities` (an array of points denoting the facilities), and `routes` (an array of graphics representing each route, which is returned when `ClosestFacilityParameters.returnRoutes` is set to `true`.

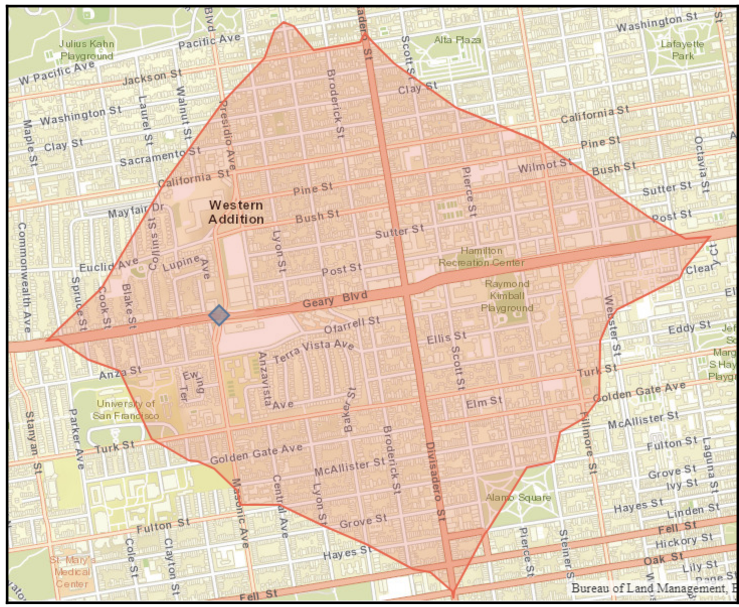
The `directions` property contains an array of `DirectionsFeatureSet` objects. Each includes text that describes turn by turn directions for getting from the incident to the facility, and the geometry of the route. Other information included in this object is the length of the route, time to travel along the route, and estimated time of arrival.

Other properties contained within `ClosestFacilitySolveResults` include an array of polylines representing the routes returned, an array of barriers that were included in the calculation, and any messages that the task returned during its operation.

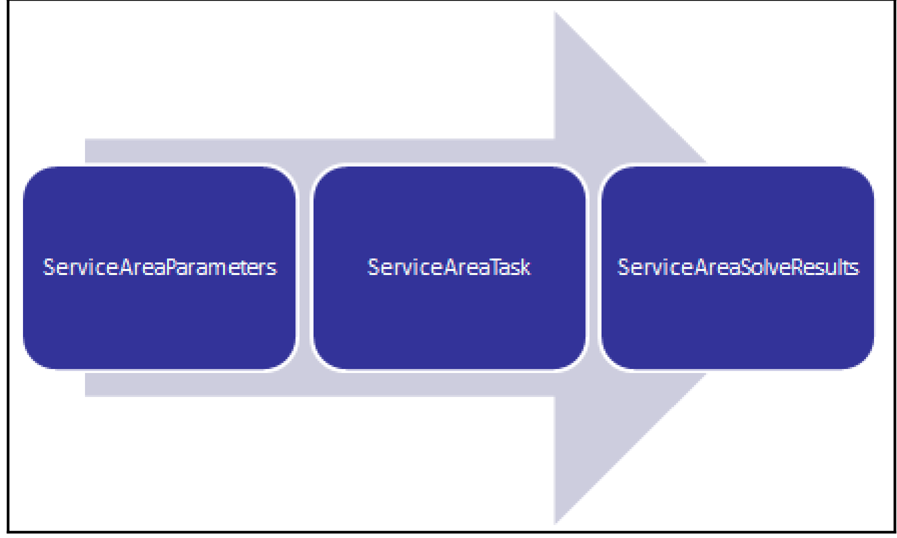
ServiceArea task

The `ServiceArea` task calculates how far around an input location you can expect to travel within a specified period of time. This is a particularly useful analysis when investigating possible locations for a new depot or similar facility where it is important to understand how wide an area your operatives can service.

The service area is defined in minutes and is a region that encompasses all accessible streets within that period:



The classes involved in service area operations include **ServiceAreaParameters**, **ServiceAreaTask**, and **ServiceAreaSolveResults**:



The ServiceAreaParameters object specifies the task's input parameters and includes properties for the default time period, the set of facilities involved, any barriers or other restrictions used in the analysis, and more.

To execute the task, you instantiate a ServiceAreaTask object and pass the ServiceAreaParameters object into its solve() method. Results are passed from ArcGIS Server back to the client in the form of a ServiceAreaSolveResults object.

Some of the more commonly used properties on the ServiceAreaParameters object are discussed as follows.

The defaultBreaks property is an array of numbers defining the service area. For instance, in the following code example a single value of 2 is provided, indicating that we'd like to return a 2-minute service area around the facility.

The returnFacilities property, when set to true, indicates that the facilities should be returned with the results.

Barriers, in the form of either points, polylines, or polygons can be specified in the barriers property.

You can analyze either to or from the facility by setting the travelDirection property. There are many other properties that can be set on ServiceAreaParameters, and we would urge you to read the documentation for a better idea of the task's capabilities.

The following code example demonstrates the use of the ClosestFacilityTask:

```
var params = new ServiceAreaParameters();
params.defaultBreaks= [2];
params.outSpatialReference = map.spatialReference;
params.returnFacilities = false;

var serviceAreaTask = new
ServiceAreaTask("https://sampleserver3.arcgisonline.com/ArcGIS/rest/service
s/Network/USA/NAServer/Service Area");
serviceAreaTask.solve(params,function(solveResult){
    var polygonSymbol = new SimpleFillSymbol(
        "solid",
        new SimpleLineSymbol("solid", new Color([232,104,80]), 2),
        new Color([232,104,80,0.25])
    );
arrayUtils.forEach(solveResult.serviceAreaPolygons, function(serviceArea){
    serviceArea.setSymbol(polygonSymbol);
    map.graphics.add(serviceArea);
});
```

```
}, function(err){
    console.log(err.message);
});
```

Summary

This chapter has introduced you to the various capabilities afforded by accessing a network service from within your application. You learned how to use the `RouteTask` to find directions from one point to one or more others. You saw how the Directions widget could make that much easier to implement and provide your users with a great interface. Then you saw how you could use the `ClosestFacilityTask` to find the closest points of interest (facilities) from one or more origin points (incidents), or use the `ServiceAreaTask` to calculate drive times within a specified time period, that might help you site a depot or other facility where response times are important.

All the tasks we have seen in this and previous chapters are great, but their capabilities are limited to those implemented by Esri, and sometimes that won't cut it. In the next chapter, you will learn how to work with geoprocessing tasks, which you can build yourself, publish as a service, and then consume within your web mapping applications.

10
Geoprocessing Tasks

Geoprocessing is the act of manipulating geographic and related data. Esri's ArcGIS Desktop software provides a large number of geoprocessing tools that you can chain together into models to accomplish specific workflows, like the one you see in the following diagram. For example, you might create a model that buffers one layer and then clips a second layer to it. You could use the model with a stream layer for the buffer and a vegetation layer for the clip. But, having created this model once, you can use it to perform the same operations on other layers without having to recreate it from scratch. You could also automate it as part of a batch-processing operation.

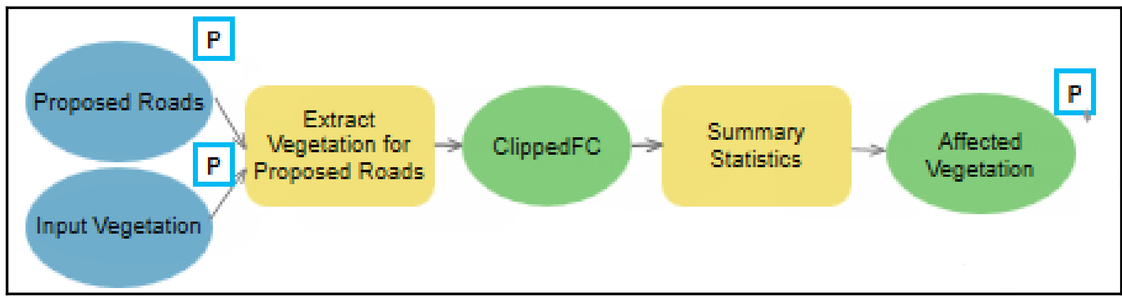
But where things get interesting from our perspective as ArcGIS API for JavaScript developers, is the ability to publish these geoprocessing tools as services in ArcGIS Server and consume them within our web mapping applications.

Think about that for a moment. That means we can create just about any GIS workflow we want, have the server execute it, and access its results in our application. That's real power, with very little code.

In this chapter, we will cover the following topics:

- Models in ArcGIS Server
- Using the `Geoprocessor` task - what you need to know
- Understanding the services directory page for a geoprocessing task
- The `Geoprocessor` task
- Executing the task

- Practice time with geoprocessing tasks

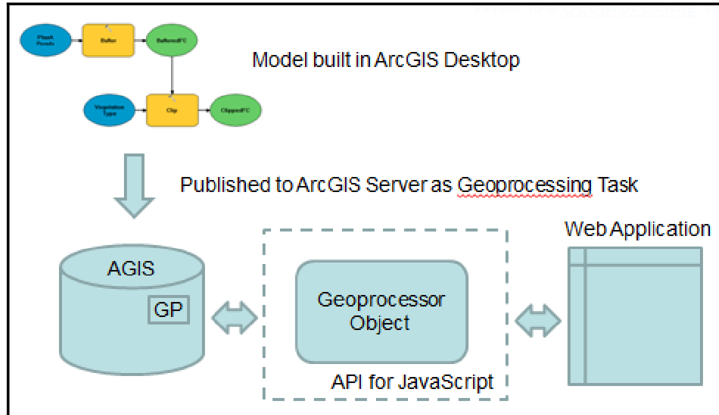


Models in ArcGIS Server

Models are built in ArcGIS Pro using `ModelBuilder`. Once built, you can publish these via ArcGIS Server, as geoprocessing services. Web applications can use the `Geoprocessor` task object in the ArcGIS API for JavaScript to execute the models and retrieve the results.

Although the client is responsible for feeding the correct parameters to the model and unpacking the results, the actual models are run on the ArcGIS Server. This is because they are computationally expensive and, to work their magic, require various ArcObjects that are not available on the client.

Geoprocessing jobs are submitted to the server by your application, using the `Geoprocessor` task object. The results are picked up after the service has completed. This process is illustrated in the following diagram:

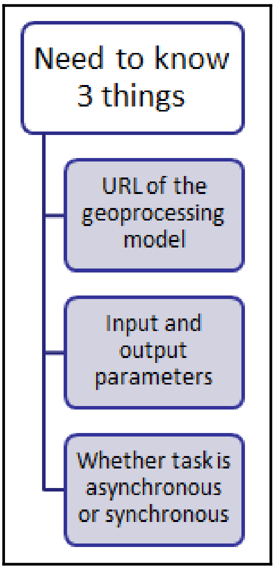


Using the Geoprocessor task - what you need to know

There are three things that you need to know when using a geoprocessing service:

- First, you need to know the URL where the model is located. An example URL is provided as follows. To make a model available through ArcGIS Server, you publish the toolbox that contains it. A toolbox can contain more than one model and so, therefore, can the geoprocessing service. For example, the following URL references the `PopulationSummary` model in the `ESRI_Population_World` geoprocessing service: `http://sampleserver1.arcgisonline.com/ArcGIS/rest/services/Demographics/ESRI_Population_World/GPServer/PopulationSummary`.
- You need to know what input parameters are required by the model, and what type of data is returned.
- Finally, you need to know whether the task is asynchronous or synchronous because this will affect how you write your code.

All of this information can be found on the services page for the geoprocessing task:



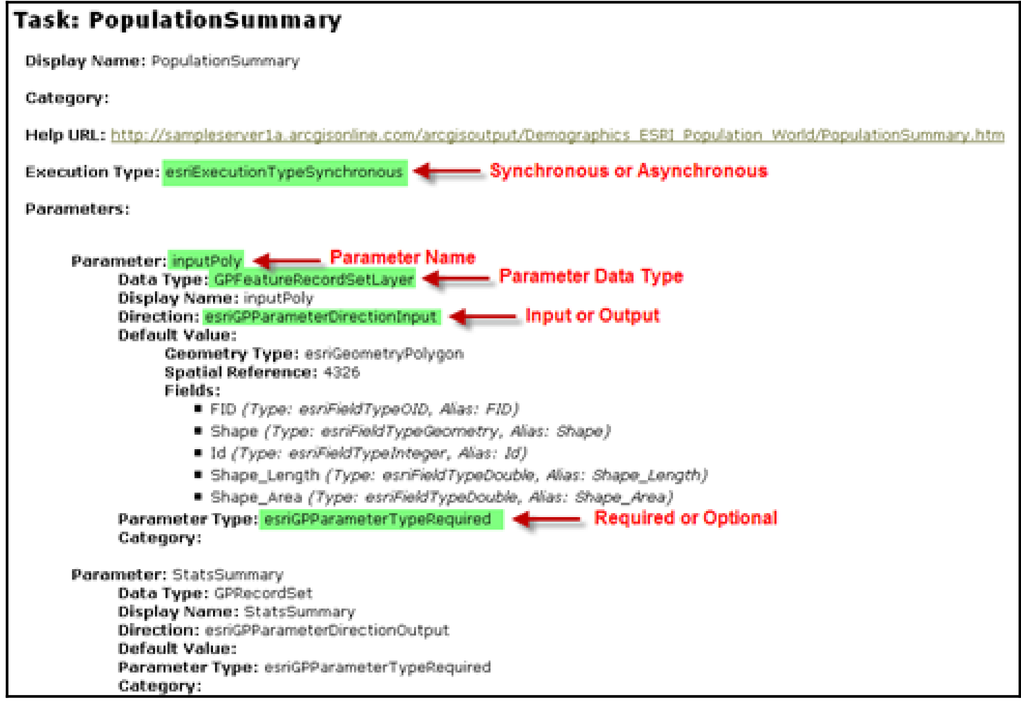
Understanding the services page for a geoprocessing task

The services directory page for a geoprocessing service provides all the information you need to include it in your application.

This includes the execution type which can be either synchronous or asynchronous. In the case of the `PopulationSummary` service seen in the following screenshot, it is a synchronous task. This means that once your application has executed, it won't do anything else until the results of the geoprocessing task have been returned. This type of execution is typically used only with tasks that you expect to execute quickly.

Asynchronous geoprocessing tasks are submitted as jobs and allow your application to continue functioning while the geoprocessing service is doing its work. When the task has completed, it notifies your application that processing is complete and the results are ready.

The other information you will need from the services directory page for the service includes the parameter names, the parameter data types, whether the parameter is an input or output type, and whether the parameter is required or optional:



Input parameters

Almost all geoprocessing tasks will require one or more input parameters. These parameters can be specified as either required or optional and are created as JSON objects. You'll see a code example showing how to create these JSON objects as follows. When creating parameters as JSON objects, you must remember to create them in the exact order that they appear on the services page. The parameter names must also be identical to those on the services page, as shown in the following example:

Parameter: Input_Observation_Point
Data Type: GPFeatureRecordSetLayer
Display Name: Input Observation Point
Direction: esriGPParameterDirectionInput
Default Value:
Geometry Type: esriGeometryPoint
Spatial Reference: 54003
Fields:
- FID *(Type: esriFieldTypeOID, Alias: FID)*
- Shape *(Type: esriFieldTypeGeometry, Alias: Shape)*
- OffsetA *(Type: esriFieldTypeDouble, Alias: OffsetA)*

Parameter Type: esriGPParameterTypeRequired
Category:

Parameter: Viewshed_Distance
Data Type: GPLinearUnit
Display Name: Viewshed Distance
Direction: esriGPParameterDirectionInput
Default Value: 15000 esriMeters
Parameter Type: esriGPParameterTypeRequired
Category:

Parameter name is important

Both parameters are required

The following code example is correct because the parameter names are spelled exactly as seen in the services page (and using the same case: the names are casesensitive) and they are provided in the correct order:

```
var params = {
    Input_Observation_Point: featureSetPoints,
    Viewshed_Distance: 250
};
```

In comparison, the following code example would be incorrect since the parameters are provided in reverse order:

```
var params = {
    Viewshed_Distance: 250,
    Input_Observation_Point: featureSetPoints
};
```

The Geoprocessor task

The `Geoprocessor` class in the ArcGIS API for JavaScript is what coordinates the input parameters, executes the geoprocessing task, and coordinates the collection of the task results. In this way, it is very similar to the other tasks you have worked with in this book, such as the `QueryTask`.

You kick off the geoprocessing operation by calling either `Geoprocessor.execute()` or `Geoprocessor.submitJob()` and passing in the required input parameters. We'll discuss the difference between these two methods later. After executing the geoprocessing task, the results are returned to the `Geoprocessor` object where they are processed by a callback function.

To create an instance of the `Geoprocessor` task, import the `esri/tasks/Geoprocessor` module and supply the URL of the geoprocessing service to the class constructor:

```
require(["esri/tasks/Geoprocessor"], function(Geoprocessor) {
    var url = "<gp service url>";
    var gp = new Geoprocessor(url);
});
```

Executing the task

The method you call on the `Geoprocessor` object to execute the task depends on whether the task is synchronous or asynchronous. This information is available to you in the services directory page for the geoprocessing service, under *Execution Type*. The Execution Type is set when the model is published as a service. As a developer, you don't have any control over the type after it has been published.

Just to remind you, synchronous execution requires that the client wait for the results before continuing with the application code. In asynchronous execution, a client submits a job, continues to run other functions, and checks back later for completion of the job. By default, the client checks back for completion every second until the job has finished.

Synchronous tasks

Synchronous tasks require that your application code submit a job and wait for a response before continuing. Because your end users must wait for the task to complete before continuing to interact with your application, only use synchronous tasks if you expect the geoprocessing service to return its results very quickly. If a task takes more than just a few seconds, consider republishing the service as asynchronous. Users quickly become frustrated with applications that appear to hang for longer than that.

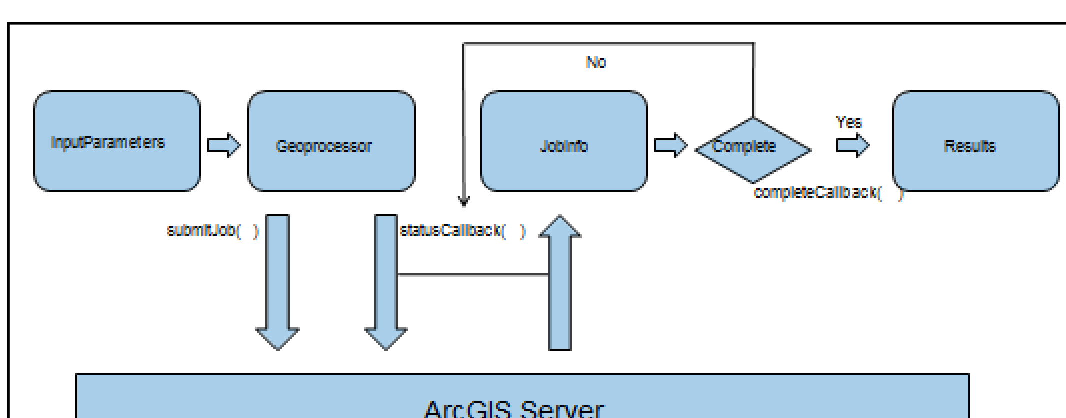
To execute a synchronous task, call the `Geoprocessor.execute()` method, passing in the required parameters and a callback function. The callback function is executed when the results of the operation are available, which are passed as a parameter to the callback function. The results are stored in an array of `ParameterValue`.

Asynchronous tasks

Asynchronous tasks require that you submit a job, allow your application to proceed with any other operations it must perform, and then check back in with ArcGIS Server to retrieve the results when the task is complete. The advantage of this approach is that it doesn't force your end users to wait for the results.

To execute an asynchronous task, call the `Geoprocessor.submitJob()` method. You will need to supply input parameters, success or error callback functions, and, optionally, a status callback function. The status callback function executes each time your application checks the server to see if the job has finished. By default, the status is checked once per second. However, you can change this interval by calling the `Geoprocessor.setUpdateDelay()` method. Every time the status is checked, the server returns `JobInfo` object that contains information indicating the status of the job. When `JobInfo.jobStatus` is set to `STATUS_SUCCEEDED`, the success callback function executes.

The following diagram illustrates this sequence of events:



Practice time with geoprocessing tasks

In this exercise, you will code a simple application that uses a particle tracking model to estimate where a bottle would end up after 180 days, if you dropped it into the ocean at a specific point. The services directory entry is shown as follows. You can see that this is a synchronous service which takes two input parameters: a `FeatureSet` that contains the point object representing the location where the bottle is dropped, and a `Double` representing the number of days it should be allowed to drift for. It returns a single output parameter, again of `FeatureSet` type, that represents the route the bottle would take:

Task: MessageInABottle

Display Name: MessageInABottle

Category:

Help URL: http://sampleserver1c.arcgisonline.com/arcgisoutput/Specialty_ESRI_Currents_World/MessageInABottle.htm

Execution Type: esriExecutionTypeSynchronous

Parameters:

Parameter: Input_Point
- **Data Type:** GPFeatureRecordSetLayer
- **Display Name:** Input Point
- **Direction:** esriGPParameterDirectionInput
- **Default Value:**
  - **Geometry Type:** esriGeometryPoint
  - **Spatial Reference:** 4326
  - **Fields:**
    - FID *(Type: esriFieldTypeOID, Alias: FID)*
    - Shape *(Type: esriFieldTypeGeometry, Alias: Shape)*
    - Id *(Type: esriFieldTypeInteger, Alias: Id)*
- **Parameter Type:** esriGPParameterTypeRequired
- **Category:**

Parameter: Days
- **Data Type:** GPDouble
- **Display Name:** Days
- **Direction:** esriGPParameterDirectionInput
- **Default Value:** 150
- **Parameter Type:** esriGPParameterTypeRequired
- **Category:**

Parameter: Output
- **Data Type:** GPFeatureRecordSetLayer
- **Display Name:** Output
- **Direction:** esriGPParameterDirectionOutput
- **Default Value:**
  - **Fields:**
    - FNode_ *(Type: esriFieldTypeInteger, Alias: FNode_)*
- **Parameter Type:** esriGPParameterTypeDerived
- **Category:**

1. Open the JavaScript Sandbox at `https://developers.arcgis.com/javascript/3/sandbox/sandbox.html`.
2. Remove the JavaScript content from the `<script>` tag that I have highlighted as follows:

```
<script>
  var map;

  require(["esri/map", "dojo/domReady!"], function(Map) {
    map = new Map("map", {
      basemap: "topo",  //For full list of ...
      center: [-122.45, 37.75], // longitude, latitude
      zoom: 13
    });
  });
</script>
```

3. Create your `require()` function to import the modules that we will use in this exercise:

```
<script>
    require(["esri/map",
        "esri/config",
        "esri/geometry/webMercatorUtils",
        "esri/tasks/Geoprocessor",
        "esri/symbols/SimpleMarkerSymbol",
        "esri/symbols/SimpleLineSymbol",
        "esri/Color",
        "esri/tasks/FeatureSet",
        "dojo/_base/array",
        "dojo/domReady!"], function (Map, esriConfig,
            webMercatorUtils, Geoprocessor, SimpleMarkerSymbol,
            SimpleLineSymbol, Color, FeatureSet, array) {
    });
</script>
```

4. Define variables to hold the `Map` and `Geoprocessor` objects, and instantiate the map with the following options:

```
<script>
    var map, gp;

    require(["esri/map",
        "esri/config",
        "esri/geometry/webMercatorUtils",
        "esri/tasks/Geoprocessor",
        "esri/symbols/SimpleMarkerSymbol",
        "esri/symbols/SimpleLineSymbol",
        "esri/Color",
        "esri/tasks/FeatureSet",
        "dojo/_base/array",
        "dojo/domReady!"], function (Map, esriConfig,
webMercatorUtils, Geoprocessor, SimpleMarkerSymbol,
SimpleLineSymbol, Color, FeatureSet, array) {

            map = new Map("map", {
                basemap: "satellite",
                center: [-43.682, 32.99],
                zoom: 3
            });
</script>
```

5. After the code that instantiates the `Map`, register an event handler called `executeGPTask` for the `Map.click` event:

```
map.on("click", executeGPTask);
```

6. Inside the `require()` function, create the new `Geoprocessor` task object and set the output spatial reference to Web Mercator (`wkid: 102100`):

```
gp = new
Geoprocessor("http://sampleserver1.arcgisonline.com/ArcGIS/rest
/services/Specialty/ESRI_Currents_World/GPServer/MessageInABott
le");
gp.setOutSpatialReference({ wkid: 102100 });
```

7. Now you'll create the `executeGPTask()` function that serves as the handler for the `Map.click()` event. This function will clear any existing graphics, create a new point graphic that represents the point where the user clicked the map, and execute the geoprocessing task. First, create the stub for the `executeGPTask()` function just under the line of code that instantiated the `Geoprocessor` object:

```
gp = new
Geoprocessor("http://sampleserver1.arcgisonline.com/ArcGIS/rest
/services/
    Specialty/ESRI_Currents_World/GPServer/MessageInABottle");
gp.setOutSpatialReference({ wkid: 102100 });

function executeGPTask(evt) {

}
```

8. Clear any existing graphics and create the new `SimpleMarkerSymbol` that will represent the point that is clicked on the map:

```
function executeGPTask(evt) {
    map.graphics.clear();

    var ptSymbol = new SimpleMarkerSymbol()
    ptSymbol.setStyle(SimpleMarkerSymbol.STYLE_SQUARE);
    ptSymbol.setColor(new Color([255, 255, 255, 1.0]));
    ptSymbol.setSize(15);
}
```

9. When the `Map.click()` event is triggered, a `MouseEvent` object is created and passed in to the `executeGPTask()` function. This object is represented in our code by the `evt` variable. In this step, you're going to create a new `Graphic` object using the `Event.mapPoint` property that contains the `Point` geometry returned from the map click and symbolizes it with the `SimpleMarkerSymbol` that you created in the last step. You'll then add this new graphic to the `GraphicsLayer` so that it can be displayed on the map:

```
function executeGPTask(evt) {
    map.graphics.clear();

    var ptSymbol = new SimpleMarkerSymbol()
    ptSymbol.setStyle(SimpleMarkerSymbol.STYLE_SQUARE);
    ptSymbol.setColor(new Color([255, 255, 255, 1.0]));
    ptSymbol.setSize(15);

    var graphic = new esri.Graphic(evt.mapPoint, ptSymbol);
    map.graphics.add(graphic);
}
```

10. Now create an array called `features` and place the `graphic` into the array. This array of graphics will eventually be passed into a `FeatureSet` object that will be passed to the geoprocessing task:

```
function executeGPTask(evt) {
    map.graphics.clear();

    var ptSymbol = new SimpleMarkerSymbol()
    ptSymbol.setStyle(SimpleMarkerSymbol.STYLE_SQUARE);
    ptSymbol.setColor(new Color([255, 255, 255, 1.0]));
    ptSymbol.setSize(15);

    var graphic = new esri.Graphic(evt.mapPoint, ptSymbol);
    map.graphics.add(graphic);

    var features = [];
    features.push(graphic);
}
```

11. Create a new `FeatureSet` object and add the array of graphics to the `FeatureSet.features` property:

```
function executeGPTask(evt) {
    map.graphics.clear();

    var ptSymbol = new SimpleMarkerSymbol()
    ptSymbol.setStyle(SimpleMarkerSymbol.STYLE_SQUARE);
    ptSymbol.setColor(new Color([255, 255, 255, 1.0]));
    ptSymbol.setSize(15);

    var graphic = new esri.Graphic(evt.mapPoint, ptSymbol);
    map.graphics.add(graphic);

    var features = [];
    features.push(graphic);
    var featureSet = new FeatureSet();
    featureSet.features = features;
}
```

12. Create a JSON object that will hold the input parameters for the geoprocessing task. The input parameters are `Input_Point` and `Days`. Remember that each input parameter must be provided exactly as shown in the services directory page, and in the same order. We define the `Input_Point` as a `FeatureSet` object which contains an array of graphics (only a single point graphic in this case), and `Days` should be assigned the numerical value of `180`. Finally, as this is a synchronous service, we'll call it with the `Geoprocessor.execute()` method, passing in the input parameters and the success callback (`displayResults`) and error callback (`showError`) functions. We'll create these callback functions next:

```
function executeGPTask(evt) {
    map.graphics.clear();

    var ptSymbol = new SimpleMarkerSymbol()
    ptSymbol.setStyle(SimpleMarkerSymbol.STYLE_SQUARE);
    ptSymbol.setColor(new Color([255, 255, 255, 1.0]));
    ptSymbol.setSize(15);

    var graphic = new esri.Graphic(evt.mapPoint, ptSymbol);
    map.graphics.add(graphic);

    var features = [];
    features.push(graphic);
    var featureSet = new FeatureSet();
    featureSet.features = features;
```

```
        var params = { "Input_Point": featureSet,
                       "Days": 180 };
        gp.execute(params, displayResults, showError);
    }
```

13. In the last step, we defined a callback function called `displayResults()` that will be called when the geoprocessing service completes successfully. Let's create this `displayResults()` function. The `displayResults()` function accepts two parameters: the result object and any messages that are returned. We'll ignore the messages for this example, but of course we are interested in the results! Just under the closing brace for the `executeGPTask()` function enter the following stub:

```
    function displayResults(results, messages) {

    }
```

14. We only expect this geoprocessing task to return a single feature, a polyline representing the path that the bottle would take. Let's dig into the `results` object and store that single feature in a variable called `feature` to make the result easier to work with:

```
    function displayResults(results, messages) {
        var feature = results[0].value.features[0];
    }
```

15. This feature will be a polyline, so create a suitable `SimpleLineSymbol` so that we can display it on the map, and assign the symbol to it before adding it to the map:

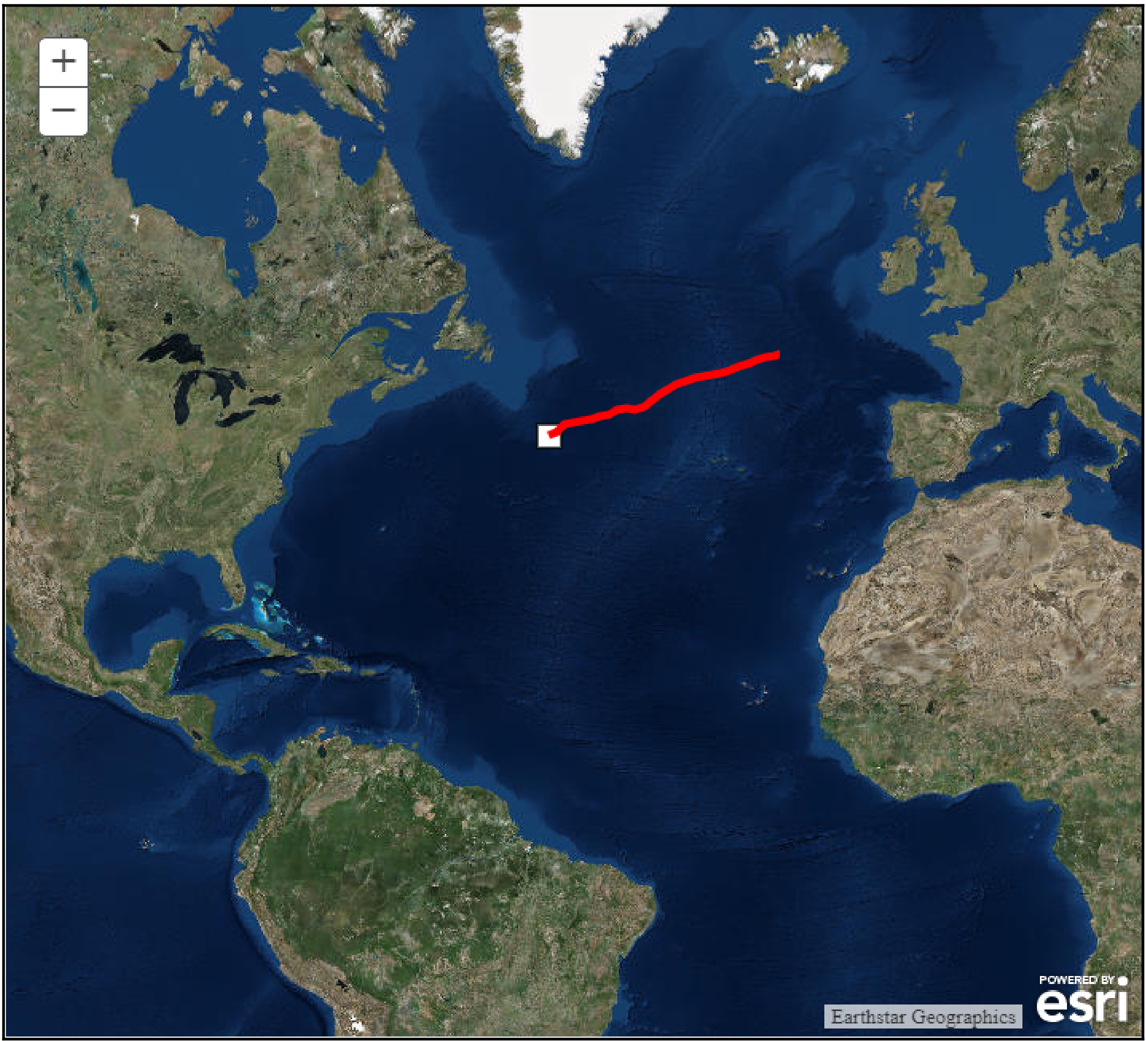
```
    function displayResults(results, messages) {
        var feature = results[0].value.features[0];
        var lineSymbol = new SimpleLineSymbol();
        lineSymbol.setColor(
            new Color([255, 0, 0]));
        lineSymbol.setWidth(5);
        feature.setSymbol(lineSymbol);
        map.graphics.add(feature);
    }
```

16. Finally, let's create the code for our error handler, called `showError()`. For now, let's just dump any error object this handler receives to the browser's console:

```
    function showError(err) {
        console.log(err);
    }
```

17. You may want to review the solution file (`geoprocessing.html`) in the `Chapter10` folder of the sample code to verify that your code has been written correctly
18. Click the **Refresh** button in the Sandbox. You should see the following map. If not, you may need to recheck your code for accuracy:

19. Click somewhere on the map. A square should appear at the point you clicked and the geoprocessing task executes. Be patient. Sometimes this can take several seconds. When it completes, you should see the path the bottle would take:



Summary

ArcGIS Server can expose geoprocessing models as services, which your ArcGIS API for JavaScript applications can access. These models execute on the ArcGIS Server due to their computationally intensive nature and the fact that they need ArcGIS components not available in the API to do their work. You submit these requests to the server via your application together with the required parameters, and the results are returned when the task completes.

Geoprocessing tasks can execute either synchronously or asynchronously and are configured to run as such by an ArcGIS Server administrator. As an application programmer, it is important for you to understand what type of geoprocessing service you are accessing since the method you must call to execute the task depends upon this information. You also need to pay careful attention to the parameters required by the service, and this information is available in the services directory. In the next chapter, you will learn how to use the Geometry Service. The Geometry Service allows you to perform certain common operations you would otherwise have to rely upon a geoprocessing service to provide.

11
Geometry Operations

In the last chapter, we looked at geoprocessing operations and the power they give you, as a developer, to define your own geospatial tasks using familiar desktop software, publish them as ArcGIS Server services, and consume them within your web mapping applications.

However, there are some geoprocessing operations that are very common. It would be reinventing the wheel to create geoprocessing tasks for them every time they are required by application developers. With that in mind, Esri has made some frequentlyused operations instantly available via the Geometry Service, and its more recent client-side counterpart, the Geometry Engine.

In this chapter we will cover the following topics:

- The Geometry Service
- The Geometry Engine
- Practice time with the Geometry Engine

The Geometry Service

Unlike all of the services we have seen so far, the Geometry Service is not dependent on some underlying resource. That is to say, it is not something that you must first create in ArcGIS Pro and then publish to ArcGIS Server.

You may say, and you would be correct, that you haven't had to do that for any of the services we have been using in this book. However, the point is that someone, somewhere, has done so. The basemaps you have been consuming in the web applications you have created so far all started off as a map document somewhere, and have been made available to you as the result of someone publishing that document to a publicly available ArcGIS Server instance. The geoprocessing services you have accessed were similarly conceived in Model Builder, or as a Python script.

This is not the case with the Geometry Service. To create a Geometry Service, all you have to do is get your ArcGIS Server administrator to enable it and you are good to go!

Once you have a running Geometry Service, check out its entry in the services directory. You will see that it offers a wide range of utility methods that support frequently used geometric operations:

ArcGIS Services Directory

Home > Geometry (GeometryServer) Help | API Reference

Geometry (GeometryServer)

Service Description: This is a sample service hosted by ESRI, powered by ArcGIS Server. ESRI has provided this example so that you may practice using ArcGIS APIs for JavaScript, Flex, and Silverlight. ESRI reserves the right to change or remove this service at any time and without notice.

Supported Interfaces: REST SOAP

Supported Operations: Project Simplify Buffer Areas And Lengths Lengths Label Points Relation Densify Distance Union Intersect Difference Cut TrimExtend Offset Generalize AutoComplete Reshape ConvexHull

Geometry Service operations

The full list of operations that the Geometry Service supports as of ArcGIS Server 10.3 is as follows:

- `Project`: Projects a set of geometries into a new spatial reference.
- `Simplify`: Alters the given geometries to make their definitions topologically legal with respect to their geometry type.
- `Buffer`: Creates a buffer polygon at a specified distance around the supplied geometries.
- `Areas` and `Lengths`: Calculates the areas and lengths of the input polygons.

- `Lengths`: Calculates the lengths of the supplied polylines.
- `Label Points`: Calculates an interior point for each polygon specified. You can use these points to label the polygon features.
- `Relation`: Computes the set of pairs of geometries from the input geometry arrays that belong to the specified relation. Both arrays are assumed to be in the same spatial reference. The relations are evaluated in 2D, and any z-coordinate values are ignored.
- `Densify`: Densifies geometries by plotting points between existing vertices.
- `Distance`: Measures the planar or geodesic distance between geometries.
- `Intersect`: This operation constructs the set-theoretic intersection between an array of geometries and another geometry.
- `Difference`: Constructs the set-theoretic difference between an array of geometries and another geometry.
- `Cut`: Splits an input polyline or polygon.
- `TrimExtend`: Trims or extends the input polylines using the user-specified guide polyline. When trimming features, the portion to the left of the cutting line is preserved in the output and the rest is discarded. If the input polyline is not cut or extended then an empty polyline is added to the output array.
- `Offset`: Constructs the offset of the input geometries. Requires a distance for the offset. If the specified distance is positive the constructed offset will be on the right side of the geometry and vice versa.
- `Generalize`: Generalizes the input geometries using the Douglas Peucker algorithm.
- `Autocomplete`: Constructs polygons that fill in the gaps between existing polygons and a set of polylines.
- `Reshape`: Reshapes a polyline or a part of a polygon using a reshaping line.
- `ConvexHull`: Returns the convex hull of the input geometry. The input geometry can be a point, multipoint, polyline, or polygon. The hull is typically a polygon but can also be a polyline or point.
- `FromGeoCoordinateString`: Converts an array of well-known strings into x, y-coordinates based on the conversion type and spatial reference supplied by the user. Only available with ArcGIS Server 10.3 or higher.

- `ToGeoCoordinateString`: Converts an array of x, y coordinates into well-known strings based on the supplied conversion type and spatial reference.
- `Union`: Constructs the set-theoretic union of the geometries in the input array. All inputs must be of the same geometric type.

Looking at that list, you'll probably recognize some operations that you envisage using quite frequently: operations such as `Buffer`, `Simplify`, or `Project`. Others may look a little arcane: `ConvexHull`, or `Autocomplete`, for instance.

The fact is that the ArcGIS API for JavaScript uses some of these operations internally to support its own tasks and widgets and these are included in the Geometry Service for the API's own convenience. For example, both the Measurement and Editor widgets both require a Geometry Service to function. However, the fact that they are there means that you can use them in your own applications without having to create a geoprocessing task that does the same thing.

Using the Geometry Service

Each of the operations listed in the previous section corresponds to a method on the `esri/tasks/GeometryService` class. To use an operation in your application, just create an instance of the `GeometryService`, and call the appropriate method.

Each method is unique in respect of the input parameters it expects from you, and the type of data you can expect it to return. In addition, each method requires a callback function that it will execute when the Geometry Service operation completes successfully and, optionally, an error callback that it will execute if something goes wrong.

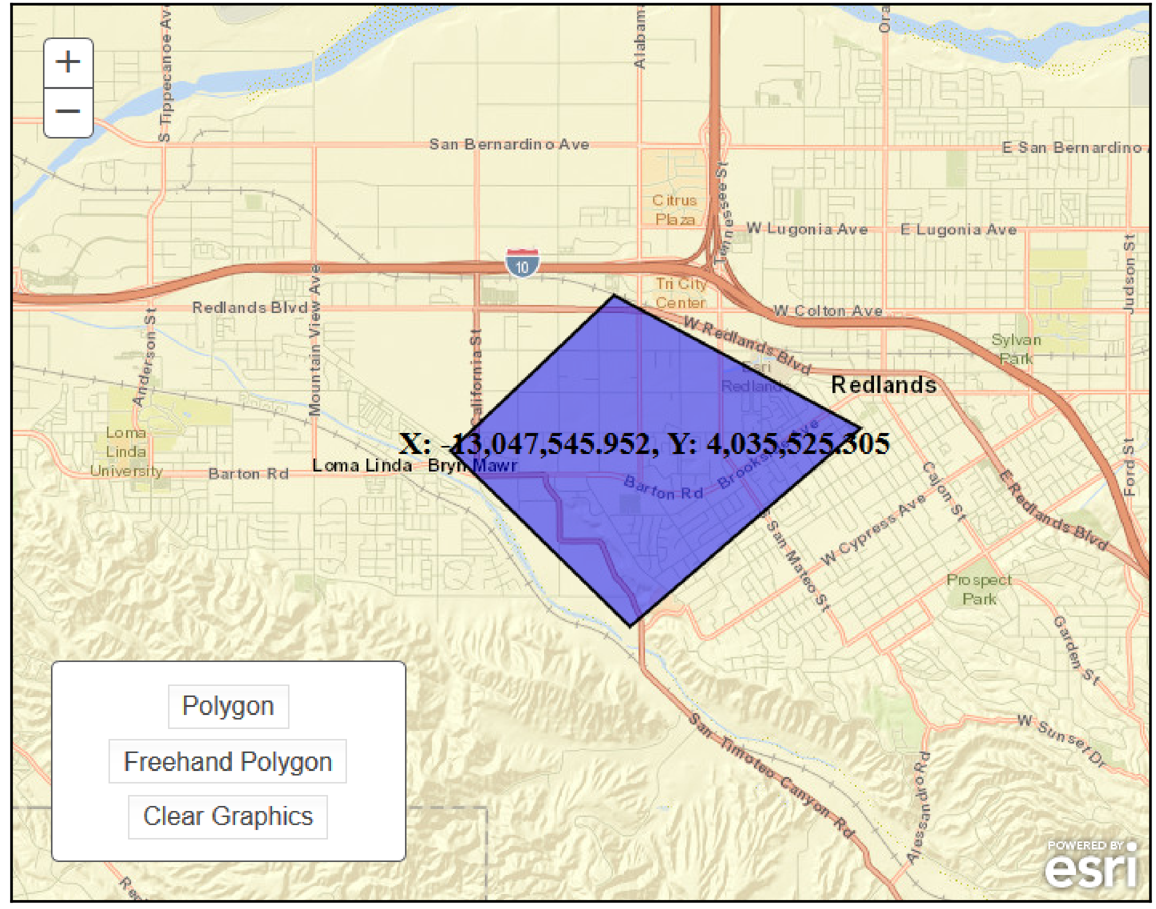
For details of each of the individual methods, check out the API documentation at `https://developers.arcgis.com/javascript/3/jsapi/`.

However, just to give you an idea of how you might use a Geometry Service operation in your applications, consider the following code extract, which allows the user to draw a polygon feature, and then uses the `simplify()` and `labelPoints()` Geometry Service methods to simplify the polygon and then create a point to anchor the label to:

```
// create and instantiate the map and perform other initialization
...

// create a toolbar for the map
toolbar = new Draw(map);
toolbar.on("draw-end", addToMap);
```

```
// activate a drawing tool when a button is clicked
registry.byId("btnDrawPolygon").on("click", function () {
    toolbar.activate(Draw.POLYGON);
});
registry.byId("btnDrawFreehand").on("click", function () {
});
registry.byId("btnClear").on("click", function () {
    map.graphics.clear();
});

geometryService = new
GeometryService("https://utility.arcgisonline.com/ArcGIS/rest/services/Geom
etry/GeometryServer");

function addToMap(evtObj) {
    map.graphics.clear();
    var geometry = evtObj.geometry;
    // add the drawn graphic to the map
    var symbol = new SimpleFillSymbol(
        SimpleFillSymbol.STYLE_SOLID,
        new SimpleLineSymbol(SimpleLineSymbol.STYLE_SOLID,
        new Color([0, 0, 0]), 2),
        new Color([0, 0, 255, 0.5]));
    var graphic = new Graphic(geometry, symbol);
    map.graphics.add(graphic);

    // simplify polygon to use it in the get label points request
    geometryService.simplify([geometry], getLabelPoints);
}

function getLabelPoints(geometries) {
    if (geometries[0].rings.length > 0) {
        geometryService.labelPoints(
         geometries, function (labelPoints) {
           // callback
            toolbar.deactivate();

            var font = new Font("20px",
               Font.STYLE_NORMAL,
               Font.VARIANT_NORMAL,
               Font.WEIGHT_BOLDER);
            array.forEach(labelPoints, function (labelPoint) {
                // create a text symbol
                var textSymbol = new TextSymbol(
                    "X: " + number.format(labelPoint.x) +
                    ", Y: " + number.format(labelPoint.y),
                    font,
                   new Color([0, 0, 0]));
```

```
                var labelPointGraphic = new Graphic(
                                 labelPoint, textSymbol);

                // add the label point graphic to the map
                map.graphics.add(labelPointGraphic);
            });
        });
    } else {
        alert("Invalid Polygon - must have at least 3 points");
    }
}
```



Another Geometry Service method you might find useful from time to time, is the `project()` method. If you have a bunch of features in one spatial reference that you want to convert to another, perhaps for the purposes of calculation or display, then the `project()` method will be your best friend.

This involves supplying a special type of object called `ProjectParameters` as the input to `project()`. The `ProjectParameters` object has the following properties:

- `geometries`: An array containing the geometries that you want to project.
- `outSR`: The destination `SpatialReference`.
- `transformation`: The **well-known ID** (**WKID**) or **well-known text** (**WKT**) of the datum transformation to apply to the projected geometries. This parameter is optional: if you don't supply one, ArcGIS Server uses a default transformation that may, or may not, be what you need. If you do specify a transformation, then you must set the `transformForward` property.
- `transformForward`: The direction of transformation. Forward is `true`, reverse is `false`.

The following code example specifies a datum transformation to project a point feature from NAD 1983 to WGS 1984. NAD 1983 is tied to the North American, Pacific (for Hawaii, Guam, and so on), and Marianas tectonic plates. WGS 1984 is tied to the **International Terrestrial Reference System** (**ITRF**) which is independent of the tectonic plates. Over time, the two coordinate systems have become increasingly different and often require re-projection to display correctly:

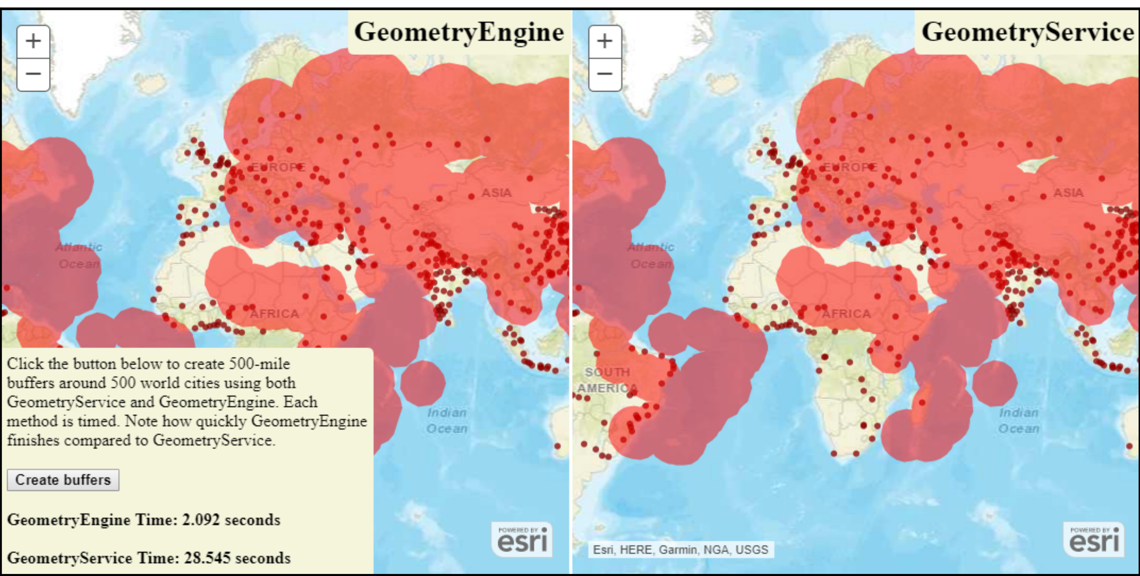
```
var transformation = 1188;  //NAD_1983_To_WGS_1984_1
require([
  "esri/tasks/ProjectParameters", ...
], function(ProjectParameters, ... ) {
  var params = new ProjectParameters();
  params.geometries = [point];
  params.outSR = outSR;
  params.transformation = transformation;
  params.transformationForward = true;
  gsvc.project(params);
  ...
});
```

The Geometry Engine

The Geometry Engine is a relatively recent addition to the API, having been introduced in version 3.13 and expanded upon with subsequent iterations. However, it's nothing short of a game-changer in respect of the performance of geometry operations.

The Geometry Engine is effectively a client-side implementation of the Geometry Service and, as such, does not need to make any network requests to the server. The result of this is much, much better performance. It includes nearly all the operations that the Geometry Service offers and adds a few more.

Esri has a great sample application that demonstrates the difference in performance between the Geometry Service and Geometry Engine. It uses both methods to generate a 500 mile buffer around 500 cities and records the time taken using each approach. In the following run, the Geometry Service took nearly half a minute to complete the task, but the Geometry Engine managed it in a shade over two seconds:



If you want to test the difference in performance yourself, you can access the application at `http://ekenes.github.io/esri-js-samples/ge-gs/`.

The Geometry Engine resides in the `esri/geometry` namespace and comes in two flavors: `geometryEngine` and `geometryEngineAsync`. The difference between the two is that `geometryEngineAsync` functions return a JavaScript *promise* that enables you to execute them as background threads in the browser and require that your browser has support for HTML5 web workers to achieve this. A promise is a construct that helps you avoid supplying callback functions as parameters to methods and is used widely in version 4 of the API, which we look at briefly in Appendix B.

We won't go into detail about web workers, JavaScript promises, or the `geometryEngineAsync` functions here. However, they are worth looking into, and the following Mozilla Developer Network articles should help. For information about web workers, see here: `http://preview.tinyurl.com/nuvtonm`. For promises, see `http://preview.tinyurl.com/k55xqy6`, and consult the `geometryEngineAsync` docs at `https://developers.arcgis.com/javascript/3/jsapi/esri.geometry.geometryengineasync-amd.html`.

In general, you should always use the Geometry Engine instead of the Geometry Service wherever possible.

Practice time with the Geometry Engine

In this practice, you will create an application that shows off the performance capabilities of the Geometry Engine by creating a buffer around the location of the user's cursor on the map and updating that buffer as the cursor moves.

1. Go to the ArcGIS API for JavaScript Sandbox and clear the contents of the following `<script>` block:

```
<script>
  var map;

  require(["esri/map", "dojo/domReady!"], function(Map) {
    map = new Map("map", {
      basemap: "topo",  //For full list of ...
      center: [-122.45, 37.75], // longitude, latitude
      zoom: 13
    });
  });
</script>
```

2. Create a variable called `map` to store the `esri/Map` object:

```
<script>
    var map;
</script>
```

3. Create the `require()` function to import the necessary modules as shown in the following:

```
<script>
    var map;

    require(["esri/map",
        "esri/Color",
        "esri/symbols/SimpleMarkerSymbol",
        "esri/symbols/SimpleFillSymbol",
        "esri/symbols/SimpleLineSymbol",
        "esri/graphic",
        "esri/geometry/geometryEngine",
        "dojo/on", "dojo/domReady!"],
        function (Map, Color, SimpleMarkerSymbol,
SimpleFillSymbol, SimpleLineSymbol, Graphic,
geometryEngine, on) {

   });
</script>
```

4. Write code to create the map in the `map` div. The map should be centered at lat:0, long:0, at zoom level 2, to show the entire world:

```
map = new Map("map", {
    basemap: "topo",
    center: [0, 0],
    zoom: 2
});
```

5. Underneath the code that creates the map, create a `SimpleMarkerSymbol` to show the location of the cursor, and a `SimpleFillSymbol` to represent the buffer:

```
var line = new SimpleLineSymbol();
line.setWidth(2);

var marker = new SimpleMarkerSymbol();
marker.setColor(new Color([0, 197, 255, 1]));
marker.setOutline(line);
marker.setSize(12);
```

```
var fill = new SimpleFillSymbol();
fill.setColor(new Color([255, 0, 0, 0.50]));
fill.setOutline(line);
```

6. Then, stub out an event handler for the `Map` object's `mouse-move` event. This will be fired every time the cursor moves over the map:

```
on(map, 'mouse-move', function (evt) {
});
```

Every time the mouse moves, we want to remove any existing graphics and then provide two new graphics: a marker representing the cursor location and a fill symbol representing a 500 mile buffer around that point.

7. Start by clearing the graphics layer every time the event handler is called:

```
on(map, 'mouse-move', function (evt) {
    map.graphics.clear();

});
```

8. Then, add a graphic that represents the current cursor location. This information is available in the `evt` object that is passed into the handler:

```
on(map, 'mouse-move', function (evt) {
    map.graphics.clear();
    var ptGraphic = new Graphic(evt.mapPoint, marker);
    map.graphics.add(ptGraphic);
});
```

The next thing we need to do is calculate the 500 mile buffer around the cursor location. The `geometryEngine` class has two methods we could use to achieve this: `buffer()` and `geodesicBuffer()`. The `buffer()` function creates a planar (or Euclidian) polygon at the specified distance around the input geometry, and the `geodesicBuffer()` function, as its name suggests, creates a geodesic buffer polygon. Geodesic buffers consider the spherical nature of the earth when creating the buffer, whereas planar buffers do not. Planar buffers appear as perfect circles when drawn on a projected flat map, while geodesic buffers appear as perfect circles only on a globe. The `geodesicBuffer()` function only works in WGS-84 (WKID: 4326) and Web Mercator (WKID: 102100 or 3857). If you're using those spatial references in your map, as we are here, you will get a more accurate result by using `geodesicBuffer()`.

9. Use the `geodesicBuffer()` function on the `geometryEngine` class to calculate a geodesic buffer within a 500 mile radius of the point that represents the location of the cursor:

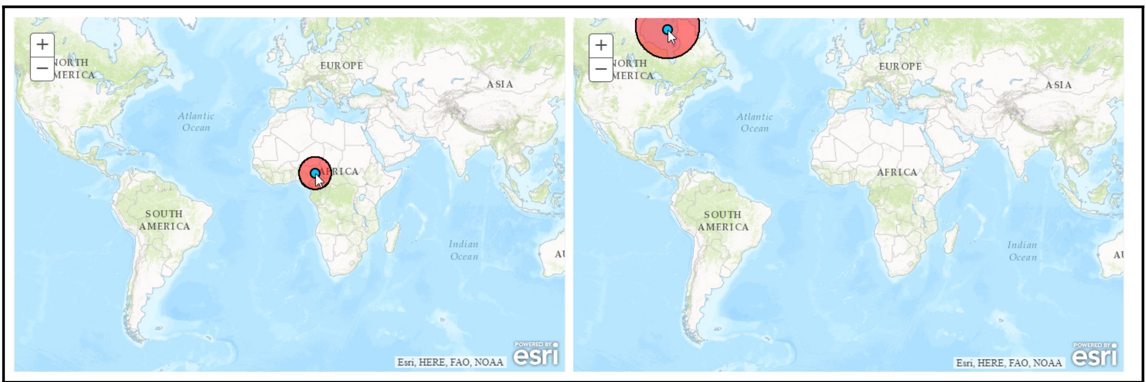
```
on(map, 'mouse-move', function (evt) {
    map.graphics.clear();
    var ptGraphic = new Graphic(evt.mapPoint, marker);
    map.graphics.add(ptGraphic);

    var bufferPoly =
geometryEngine.geodesicBuffer(ptGraphic.geometry, 500,
"miles");

});
```

10. Finally, create a graphic to represent the buffer created by the Geometry Engine and add it to the map:

```
on(map, 'mouse-move', function (evt) {
    map.graphics.clear();
    var ptGraphic = new Graphic(evt.mapPoint, marker);
    map.graphics.add(ptGraphic);

    var bufferPoly =
geometryEngine.geodesicBuffer(ptGraphic.geometry, 500,
"miles");
    var bufferGraphic = new Graphic(bufferPoly, fill);
    map.graphics.add(bufferGraphic);
});
```

11. Test your application by clicking **Refresh** in the Sandbox, and moving the cursor around the map.

Notice how the buffer updates almost instantaneously as the cursor moves. This effect would be impossible to achieve with the Geometry Service. Also, notice how the appearance of the buffer changes as you move the cursor closer to the poles. This is a very good demonstration of the sort of distortion that map projections invariably lead to:



Summary

In this chapter, you learned about the Geometry Service and the various capabilities it supports. In short, it helps you achieve sophisticated results for frequently required geometric operations without having to create geoprocessing services.

You also learned about the newer and vastly better-performing Geometry Engine, which gives you access to the same operations, without your application having to make round trips to the server.

So far, we have been digging deeper and deeper into the capabilities of the ArcGIS API for JavaScript and you should now have a pretty good understanding of the sort of things you can achieve with it. In the next chapter, we will take a step back and look at ArcGIS Online and how you can use the ArcGIS API for JavaScript to consume maps that you have created on the ArcGIS Online platform.

12

Integration with ArcGIS Online

ArcGIS Online is a website for working with maps and other types of geographic information. On this site, you will find applications for building and sharing maps. You will also find useful basemaps, data, applications, and tools that you can view and use, as well as communities you can join. For application developers, the really exciting thing about ArcGIS Online is that you can integrate content created on that platform into your ArcGIS API for JavaScript applications. In this chapter, you'll see how to go about this.

In this chapter, we will cover the following topics:

- Adding ArcGIS Online maps to your applications by using a `webmap` ID
- Adding ArcGIS Online maps to your applications by using JSON
- Practice time with ArcGIS Online

Adding ArcGIS Online maps to your applications by using a webmap ID

The ArcGIS Server API for JavaScript includes a couple of utility methods for working with maps from ArcGIS Online. Both methods are found in the `esri/arcgis/utils` module, usually aliased as `arcgisUtils`.

To create a map from an ArcGIS Online resource, you use the `arcgisUtils.createMap()` method. Before you can call `createMap()`, you need to tell the ArcGIS API for JavaScript which ArcGIS Online map you want to use in your application.

Every map in ArcGIS Online has a unique ID. This unique ID, called `webmap,` is vital information if you want to use that map in your ArcGIS API for JavaScript applications. To get the `webmap` ID for a map, just open the map in ArcGIS Online. The URL in the browser's address bar will contain the `webmap` ID for the map. You'll see how to access this ID in the practice for this chapter.

Once you have obtained the `webmap` ID for the ArcGIS Online map that you'd like to integrate into your custom JavaScript API application, you need to call the `getItem()` method, passing in the `webmap` ID. The `getItem()` method returns a `dojo/Deferred` object. `Deferred` is an object built specifically for tasks that may not complete immediately. It allows you to define success and failure callback functions that return control back to your application when the task completes.

In the following example, successful completion of the `getItem()` method passes an `itemInfo` object to the success callback function. This `itemInfo` object contains information about the web map that will be used to create the map from ArcGIS Online within your application. The following code sample illustrates this:

```
var agoId = "fc160a96a98d4052ae191cc486961b61";
var itemDeferred = arcgisUtils.getItem(agoId);

itemDeferred.addCallback(function(itemInfo) {
var mapDeferred = arcgisUtils.createMap(itemInfo, "map", {
mapOptions: {
   slider: true
   },
   geometryServiceURL:
"http://sampleserver3.arcgisonline.com/ArcGIS/rest/services/Geometry/Geomet
ryServer"
  });
mapDeferred.addCallback(function(response) {
map = response.map;
   map.on("resize", resizeMap);
  });
mapDeferred.addErrback(function(error) {
console.log("Map creation failed: ", json.stringify(error));
  });
itemDeferred.addErrback(function(error) {
console.log("getItem failed: ", json.stringify(error));
  });
}
```

We'll cover this entire function in two separate parts. For now, we'll examine the use of the `getItem()` method along with setting up callback functions for success or failure. These lines of code are highlighted in the preceding code example. In the first line of code, we create a variable called `agoId` and assign it the `webmap` ID that we'd like to use. Next, we call `getItem()`, passing in the `agoId` variable. This creates a new `dojo/Deferred` object which we assign to a variable called `itemDeferred` and use its `addCallback()` method to create a success callback function and `addErrback()` to create an error callback function.

The success function is passed an object (`itemInfo` in the following example) that contains information about the ArcGIS Online `webmap` we will use to create the map within our application. In the event of an error condition, the error callback function is called.

If `getMap()` returns good information about the `webmap`, we can add it to our application by using `createMap()`, as shown in the following code:

```
var agoId = "fc160a96a98d4052ae191cc486961b61";
var itemDeferred = arcgisUtils.getItem(agoId);

itemDeferred.addCallback(function(itemInfo) {
var mapDeferred = arcgisUtils.createMap(itemInfo, "map", {
mapOptions: {
   slider: true
   },
   geometryServiceURL:
"http://sampleserver3.arcgisonline.com/ArcGIS/rest/services/Geometry/Geomet
ryServer"
  });
mapDeferred.addCallback(function(response) {
map = response.map;
   map.on("resize", resizeMap);
  });
mapDeferred.addErrback(function(error) {
console.log("Map creation failed: ", json.stringify(error));
  });
itemDeferred.addErrback(function(error) {
console.log("getItem failed: ", json.stringify(error));
  });
}
```

The `createMap()` method is used to actually create the map from ArcGIS Online. This method takes an instance of `itemInfo` that is returned from a successful call to getItem(), or you can simply provide the `webmap` ID. As with any map that you create with the ArcGIS Server API for JavaScript you also need to provide a reference to a `<div>` container that will hold the map and any optional map options that you'd like to provide. Just as with the `getItem()` method we examined in the last slide, `createMap()` also returns a `dojo/Deferred` object that you can use to assign success and error callback functions. The success function accepts a `response` object that contains a `map` property that we use to retrieve the actual map. The error function runs when an error occurs that would prevent the creation of the map.

Adding ArcGIS Online maps to your applications with JSON

An alternative to creating a map using the web map ID is to create a map using a JSON object that represents the web map. This can be useful in situations where the application does not have access to the ArcGIS Online platform:

```
var webmap = {};
webmap.item = {
  "title":"Census Map of USA",
  "snippet": "Detailed description of data",
  "extent": [[-139.4916, 10.7191],[-52.392, 59.5199]]
};
```

Next, specify the layers that make up the map. In this snippet, the World Terrain basemap from ArcGIS Online is added, along with an overlay layer that adds additional information to the map such as boundaries, cities, water features, landmarks, and roads. An operational layer is added that displays the US census:

```
webmap.itemData = {
"operationalLayers": [{
  "url": " http://sampleserver1.arcgisonline.com/ArcGIS/rest/services/
Demographics/ESRI_Census_USA/MapServer",
  "visibility": true,
  "opacity": 0.75,
  "title": "US Census Map",
  "itemId": "204d94c9b1374de9a21574c9efa31164"
}],
"baseMap": {
  "baseMapLayers": [{
   "opacity": 1,
```

```
    "visibility": true,
    "url":
"http://services.arcgisonline.com/ArcGIS/rest/services/World_Terrain_Base/M
apServer"
    },{
    "isReference": true,
    "opacity": 1,
    "visibility": true,
    "url":
"http://services.arcgisonline.com/ArcGIS/rest/services/Reference/World_Refe
rence_Overlay/MapServer"
    }],
  "title": "World_Terrain_Base"
},
"version": "1.1"
};
```

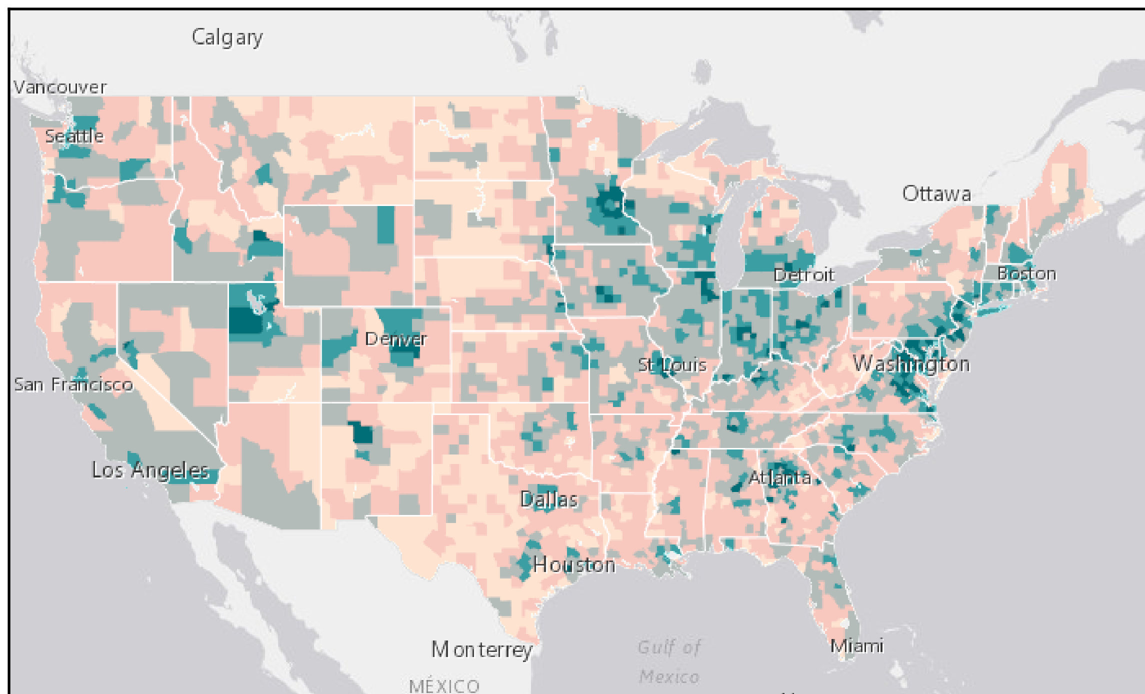
Once the web map is defined, use `createMap()` to build a map from the definition:

```
var mapDeferred = arcgisUtils.createMap(webmap, "map", {
mapOptions: {
slider: true
  }
});
```

Practice time with ArcGIS Online

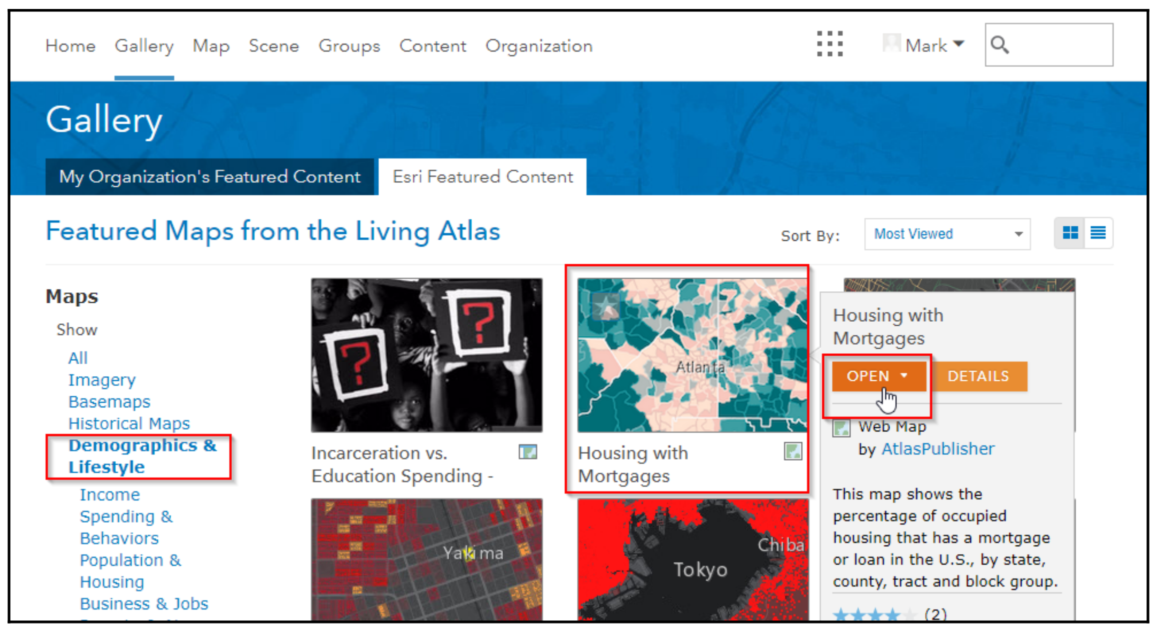
In this exercise, you will learn how to integrate ArcGIS Online maps into your applications. This simple application will display a public map showing the percentage of occupied housing that has a mortgage or loan in the US, by state, county, tract, and block group.

Darker colors represent higher proportions of borrowing:

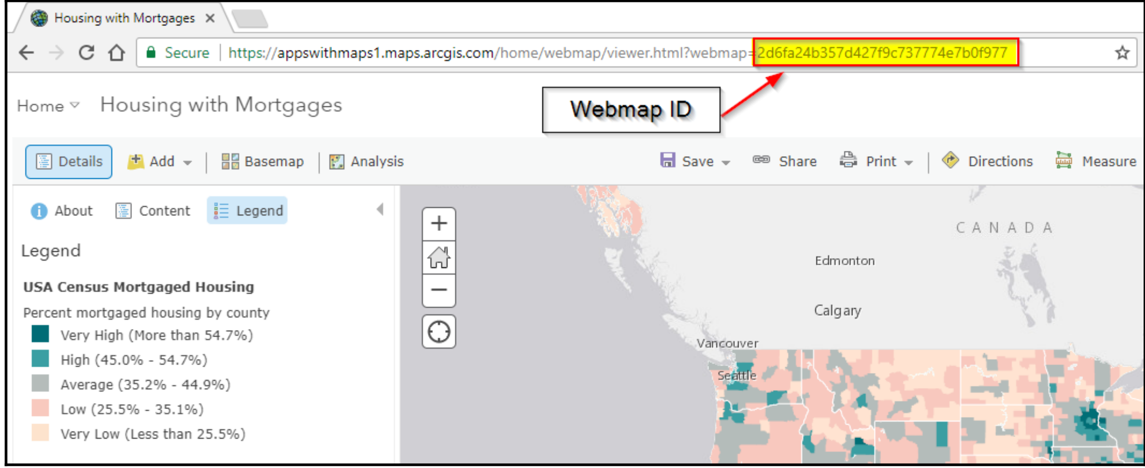


Before you begin coding the application, let's explore ArcGIS Online and see how you can find maps and retrieve their unique identifiers:

1. Visit `http://www.arcgis.com/features/free-trial.html` and sign up for a 21-day free trial of ArcGIS Online.
2. Fill in the details about your organization and then proceed to your organization's homepage. Click on the **Gallery** link at the top of the page.
3. On the **Gallery** page, click the **Esri Featured Content** tab.
4. In the **Esri Featured Content** tab, click the link for **Demographics & Lifestyle** under **Maps** on the left-hand side of the page.
5. Locate the *Housing with Mortages* map. Move the mouse over the thumbnail image and a popup appears with more information and the option to open the map. Click the **OPEN** button:



6. The map opens in the map viewer. Look at the browser's address bar and identify the portion of the URL that represents the **Webmap ID**, as shown in the following screenshot. Make a note of the ID, because you will need it in a bit:



7. Open the JavaScript Sandbox at `https://developers.arcgis.com/javascript/3/sandbox/sandbox.html`.

8. Remove the JavaScript content from the `<script>` tag that I have highlighted in the following code:

```
<script>
  var map;

  require(["esri/map", "dojo/domReady!"], function(Map) {
    map = new Map("map", {
      basemap: "topo",  //For full list of pre-defined ...
      center: [-122.45, 37.75], // longitude, latitude
      zoom: 13
    });
  });
</script>
```

9. Add the references below for the objects that we'll use in this exercise:

```
<script>
require([
        "dojo/parser",
        "dojo/ready",
        "dojo/dom",
        "esri/map",
        "esri/arcgis/utils",
        "esri/dijit/Scalebar",
        "dojo/domReady!"
      ], function(
parser,ready,dom,Map,arcgisUtils,Scalebar) {
    });
</script>
```

10. In this simple example, we're going to hard code the `webmap` ID into the application. Inside the `require()` function, create a new variable called `agoId` and assign it the `webmap` ID you obtained from the ArcGIS Online map viewer:

```
require([
"dojo/parser",
"dojo/ready",
"dojo/dom",
"esri/map",
"esri/arcgis/utils",
"esri/dijit/Scalebar",
"dojo/domReady!"
], function (parser, ready, dom, Map, arcgisUtils, Scalebar) {
    var agoId = "2d6fa24b357d427f9c737774e7b0f977";
});
```

The last two steps in this exercise deal with the `arcgisUtils.getItem()` and `arcgisUtils.createMap()` methods. Both these methods return what is known as a `dojo/Deferred` object. You need to have a basic understanding of `Deferred` objects or the code won't make a lot of sense.

As a reminder, `dojo/Deferred` helps you code operations that might take a while to execute. It allows you to define success and failure callback functions that will execute when the task completes. You define the success callback by using `Deferred.addCallback()` and the failure callback using `Deferred.errCallback()`. In the case of `getItem()`, a successful completion will pass in an `itemInfo` object to the success function. This `itemInfo` object will be used to create the map from ArcGIS Online inside your custom application. A failure to complete for some reason will result in the generation of an error being passed to the `Deferred.addErrback()` function.

11. Add the following code block to your application and then we'll discuss:

```
require([
    "dojo/parser",
    "dojo/ready",
    "dojo/dom",
    "esri/map",
    "esri/arcgis/utils",
    "esri/dijit/Scalebar",
    "dojo/domReady!"
], function (parser, ready, dom, Map, arcgisUtils, Scalebar) {

    var agoId = "2d6fa24b357d427f9c737774e7b0f977";
    var itemDeferred = arcgisUtils.getItem(agoId);

    itemDeferred.addCallback(function (itemInfo) {
         // empty handler
    });
    itemDeferred.addErrback(function (error) {
        console.log("getItem failed: ",
error);
    });

});
```

In the first new line of code, we call the `getItem()` function, passing in the `agoId` variable which references the map of interest from ArcGIS Online. This method returns a `Dojo/Deferred` object which is stored in a variable called `itemDeferred`.

The `getItem()` function gets details about the ArcGIS Online item (`webmap`). The object passed back to the callback is a generic object with the following specification:

```
{
   item: <Object>,
   itemData: <Object>
}
```

Assuming that the call to `getItem()` was successful this generic item object is then passed into our success handler, which we define within the `addCallback()` function. Inside the callback function, we then make a call to the `createMap()` method, passing in the `itemInfo` object, a reference to the map container, and any optional parameters that define the map functionality. Map parameters, in this case, include the presence of a navigation slider, and navigation buttons. The `createMap()` method then returns another `dojo/Deferred` object that is stored in the `mapDeferred` variable. In the next step, you'll define the code blocks that handle this `Deferred` object.

12. The object returned to the `mapDeferred.addCallback()` function takes the form you see in the following code:

```
{
   Map: <esri/Map>,
   itemInfo: {
          item: <Object>,
          itemData: <Object>
   }
}
```

13. Add the code below to handle the information returned:

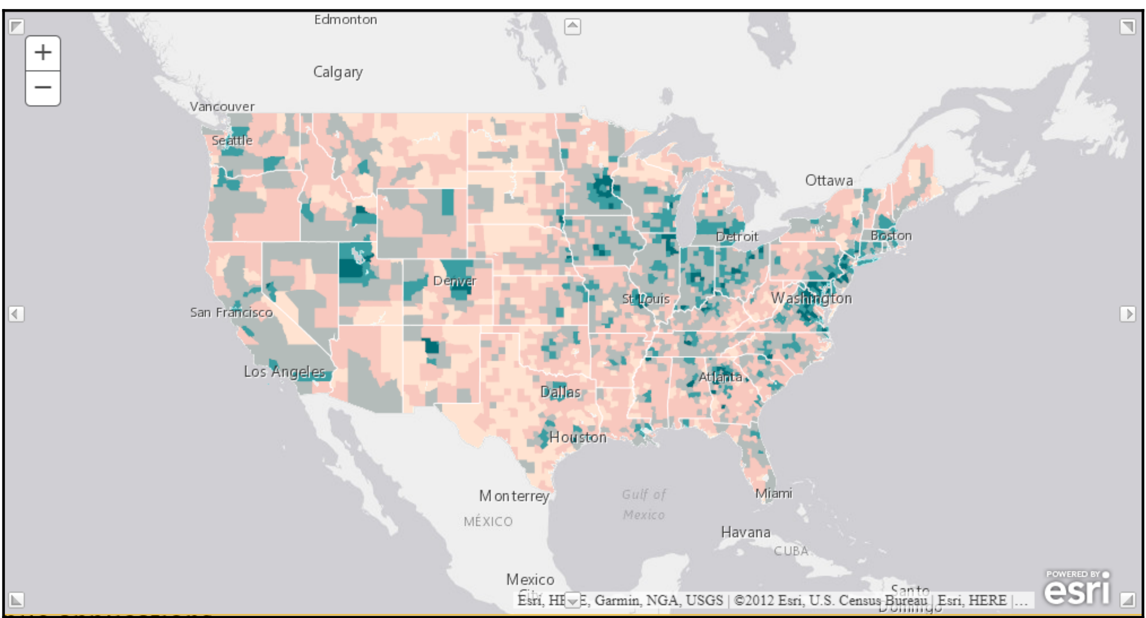
```
require([
    "dojo/parser",
    "dojo/ready",
    "dojo/dom",
    "esri/map",
    "esri/arcgis/utils",
    "esri/dijit/Scalebar",
    "dojo/domReady!"
```

```
], function (parser, ready, dom, Map, arcgisUtils, Scalebar) {

    var agoId = "2d6fa24b357d427f9c737774e7b0f977";
    var itemDeferred = arcgisUtils.getItem(agoId);

    itemDeferred.addCallback(function (itemInfo) {
        var mapDeferred =
         arcgisUtils.createMap(itemInfo, "mapDiv", {
            mapOptions: {
                slider: true,
                nav: true
            }
        });
        mapDeferred.addCallback(function (response) {
            map = response.map;
        });
        mapDeferred.addErrback(function (error) {
            console.log("Map creation failed: ",
error);
        });

    });
    itemDeferred.addErrback(function (error) {
        console.log("getItem failed: ",
error);
    });

});
```

The success function (defined in `mapDeferred.addCallback`) pulls the map from the response and assigns it to the map container.

14. You may want to review the solution file (`argisonline`) in the `Chapter12` folder of the sample code to verify that your code has been written correctly.

15. Click the **Run** button. You should see the following map. If not, you may need to recheck your code for accuracy:



Summary

ArcGIS Online is becoming increasingly important as a platform for creating and sharing maps and other resources. As a developer, you can integrate these maps into your custom applications. Each map has a unique identifier that you can use to reference the map, and the `esri/utils` module gives you access to methods you can call to retrieve details of the map and create an instance of it. Because it can take some time to return these maps from ArcGIS Online, the `getItem()` and `createMap()` methods return `dojo/Deferred` objects that provide callback functions for both success and failure. Once the maps have been successfully obtained from ArcGIS Online, they can then be used in your application just like any other map service. In the next chapter, you will learn how to use the ArcGIS API for JavaScript to create mobile applications.

13

Creating Mobile Applications

The ArcGIS Server API for JavaScript provides support for mobile platforms. Support is currently provided for iOS, Android, and Blackberry operating systems. The API is integrated with **Dojo Mobile**. In this chapter, you'll learn about the compact build of the API, which makes web mapping applications possible on mobile devices. You can also use other popular frameworks, such as **jQuery Mobile**, **Appcelerator**, and **PhoneGap** for development and related tools such as Bootstrap for responsive web design, but we'll focus on Dojo Mobile in this chapter.

Keep in mind that these frameworks are not the same things as the ArcGIS API for iOS or Android, which is what you'd use to build native applications that can be made available through an App Store. JavaScript API applications are rendered through the mobile device's browser.

We'll also cover the topic of the **geolocation API** and how it can be integrated into your ArcGIS Server applications. The geolocation API is a part of HTML5 and is used to get the location of a mobile device.

In this chapter, we will cover the following topics:

- Compact build of the API
- Setting the viewport scale
- Practice time with the compact build
- Integrating the geolocation API
- Practice time with the geolocation API

Compact build of the API

The ArcGIS API for JavaScript has a compact build that can be used to limit the footprint of the API resulting in quicker downloads for mobile devices.

The compact build only loads core Dojo objects that your application requires. For example, if you don't need the `Calendar` dijit then it's not loaded. If you need to use a code module that is not downloaded as part of the compact build, then you must use the `require()` function to load the specific module you want to use.

Using the compact version of the API is as simple as adding the word `compact` to the end of your reference to the API. You can see this in the following example. Using the API in a mobile application isn't any different from the techniques you've learned for creating web applications. However, you will need to learn some new techniques for creating user interfaces suitable for mobile applications. There are a number of good JavaScript mobile frameworks for accomplishing this task, including Dojo Mobile and jQuery Mobile. The mobile frameworks style the web content to make it look like a native mobile application. Safari browsers look like an iPhone application and Android browsers look like an Android application. Creating mobile user interfaces is beyond the scope of this text, but there are many good resources in print and online.

Although the compact builds result in a smaller initial footprint and are a great choice during development, you should ideally create an optimized build of the API when you are ready to deploy. This is a build that contains only the modules you require, and none that you do not. You can create such a build by using the Web Optimizer, but in order to do so you need either an ArcGIS Online account or ArcGIS for Developer account. You can find out more about the Web Optimizer at `https://developers.arcgis.com/javascript/3/jshelp/inside_web_optimizer.html`.

Setting the viewport scale

You will want to use the `viewport` `<meta>` tag to set some initial display characteristics for your application. The `<meta>` tag should be included in the `<head>` section of your web page. A value of `1.0` for `initial-scale` is recommended and will fill the entire viewport of the screen. Values can be set between `0` and `1.0`. If you don't set a width, your mobile browser will use the `device-width` when in portrait mode and if you don't set a height, the browser will use the `device-height` when in landscape mode:

```
    <meta name="viewport" content="width=device-width, initial-scale=1"
maximum-
    scale=1.0 user-scalable=0>
```

Practice time with the compact build

In this exercise, you will build the most basic mobile mapping application possible. We're simply going to use the compact build of the ArcGIS Server API for JavaScript to create a mapping application centered on the town of Banff, Alberta, in Canada. The application won't be able to do anything other than zoom and pan. There won't be any sort of user interface beyond just the map. The goal is just to illustrate the basic structure of a mobile application built with the API for JavaScript.

This exercise will be a little different than the exercises you've worked on in previous chapters. You won't use the ArcGIS API for JavaScript sandbox. Instead, you'll write your code in a text editor (I recommend Notepad++) and test it using a mobile emulator. This exercise will also require that you have access to a web server:

1. Before starting this exercise, you'll want to make sure you have access to a web server. If you don't have access to a web server or one isn't already installed on your computer, you can download the open source Apache web server from `http://httpd.apache.org/download.cgi`. Microsoft IIS is another commonly-used web server on Windows machines and there are many others that you can use as well. For the purposes of this exercise, I will assume that you are using the Apache web server.
2. A web server installed on your local computer will be referred to through a URL as `http://localhost`. This points to the `htdocs` folder under the Apache installation directory if you have installed Apache on a Windows platform. IIS typically uses `C:\inetpub\wwwroot` instead. Other web servers and different operating systems might expect your web files in different locations and you will need to consult the documentation. We will refer to this location as the document root in subsequent steps.

3. In the `Chapter13` folder of the sample code, you'll find a file called `mobile_map.html`. I have pre-written some of the code that you will use in this step, so that you can focus on adding referencing to the compact build as well as some other items related to mobile development. Use this file as your starting point and copy it to the document root.
4. Open `mobile_map.html` in your favorite text editor.
5. Reference the compact version of the API by adding the following highlighted lines of code to your application:

```
<head>
    <meta http-equiv="Content-Type" content="text/html;
    charset=utf-8">
    <!-- Add viewport -->
    <title>Mobile Map</title>
    <link rel="stylesheet"
    href="https://js.arcgis.com/3.21/esri/css/esri.css">

    <!-- Add reference to compact build of API -->
    <script src="https://js.arcgis.com/3.21compact/"></script>
    <script>
        // load modules and instantiate the map
    </script>
</head>
```

6. You will want to use the `viewport` `<meta>` tag attribute to set some initial display characteristics for your application. Use a value of `1.0` for the initial scale to fill the entire viewport of the screen and make this the maximum value too. Don't let users resize the application. You can achieve all this by adding the following markup near the top of the `<head>` tag:

```
<meta name="viewport" content="width=device-width, initial-
scale=1.0, maximum-
scale=1.0, user-scalable=no">
```

7. In the `<script>` tag, add the `require()` function, shown in the following code snippet, to import the required modules:

```
<script>
    require([
        "esri/map",
        "dojox/mobile",
        "dojo/domReady!"
    ],

        function (Map, mobile) {
```

```
        });
</script>
```

8. As is the case with a traditional web mapping application built with the API for JavaScript, you will need a `<div>` tag to hold the map for your mobile application. This has already been created for you. In a mobile application, you will want to style the map so that it takes up the entire viewport of the mobile application. This is accomplished in a `<style>` tag at the head of the page:

```
<head>
    <meta http-equiv="Content-Type" content="text/html;
    charset=utf-8">
    <!-- Add viewport -->
    <title>Mobile Map</title>
    <link rel="stylesheet"
    href="https://js.arcgis.com/3.21/esri/css/esri.css">
    <style>
        html,
        body,
        #map {
            height: 100%;
            margin: 0;
            padding: 0;
        }
    </style>
```

9. Add a line of code at the top of the `require()` function that hides the mobile device's browser address bar. When developing mobile applications, you want to maximize the amount of screen space available to you. You can use the Dojo Mobile `hideAddressBar()` function to do this:

```
function (Map, mobile) {

    mobile.hideAddressBar();

});
```

10. Then, instantiate the map, as shown in the following code snippet:

```
function (Map, mobile) {

    mobile.hideAddressBar();

    var map = new esri.Map("map", {
        basemap: "streets",
        center: [-115.570, 51.178],
        zoom: 12,
        slider: false
    });

});
```

11. Mobile devices can display their viewport in standard or landscape mode simply by turning the device. Your application will need to deal with these events as they occur. Add an `onorientationchange` event to the `<body>` tag. The `onorienationchange` event references a JavaScript function called `orientationChanged()` which we have not yet defined. We'll do that in the next step:

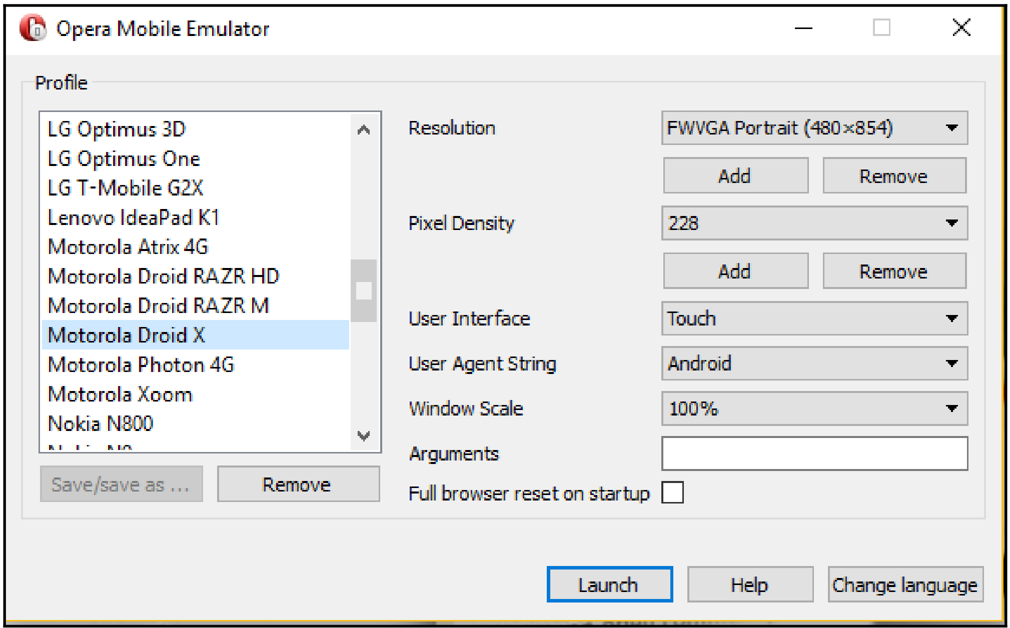
```
<body onorientationchange="orientationChanged()">
    <div id="map"></div>
</body>
```

12. Create the `orientationChanged()` JavaScript function, as shown in the following code snippet:

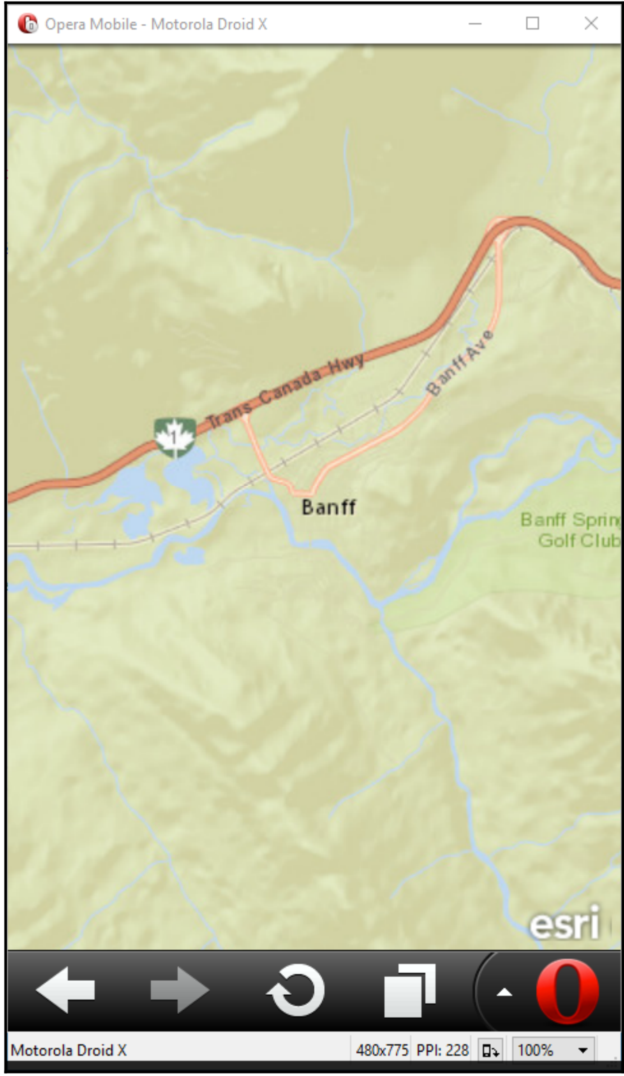
```
require([
    "esri/map",
    "dojox/mobile",
    "dojo/domReady!"
],

function (Map, mobile) {

    mobile.hideAddressBar();

    var map = new esri.Map("map", {
        basemap: "streets",
        center: [-115.570, 51.178],
        zoom: 12,
        slider: false
    });
```

```
        function orientationChanged() {
            if (map) {
                map.reposition();
                map.resize();
            }
        }

    });
```

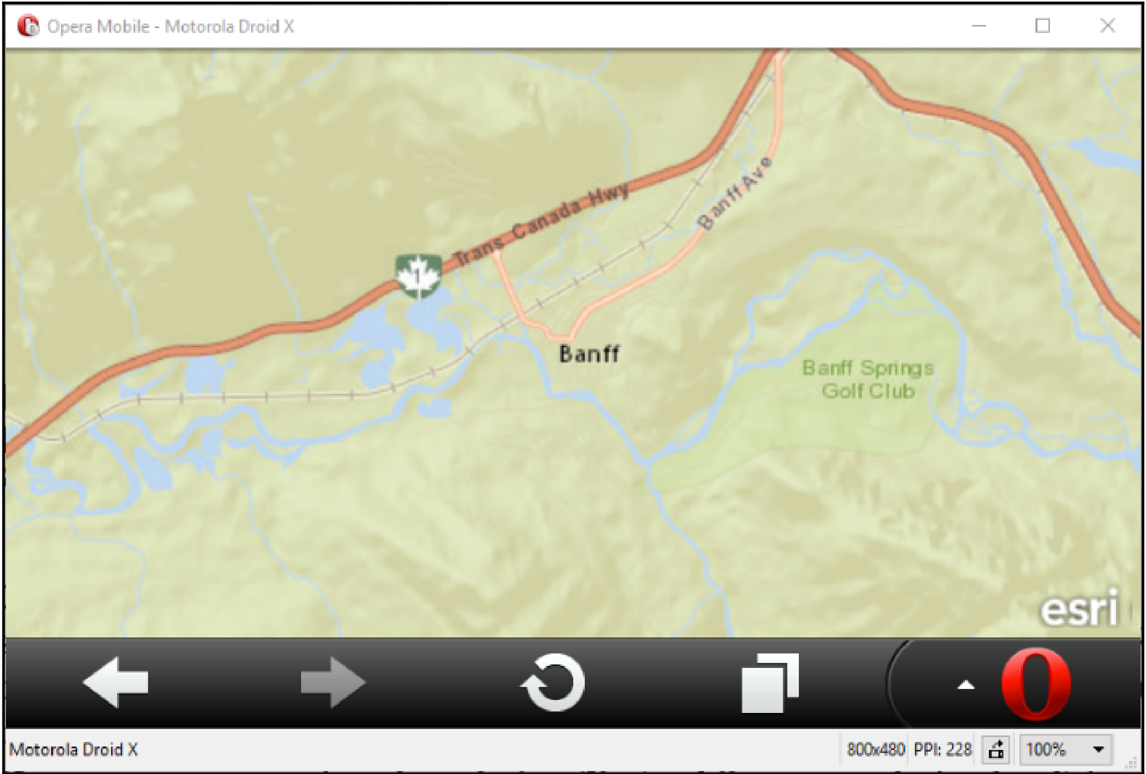
13. Save the file.
14. You now need an emulator. An emulator allows you to see how your application will look and behave on a mobile device. There are many to choose from, but I recommend the **Opera Mobile Classic Emulator** for simplicity. Visit `https://www.opera.com/developer/mobile-emulator` and download the appropriate installer for your operating system. Install the emulator and launch it.
15. Select a pre-configured device (such as iPhone 6) from the list on the left-hand side of the application and click **Launch**. If you want to test your application on a specific device and know the required details, you can add a new device:



16. Ensure that your file exists in the document root of your web server and that your web server is running. The file can then be accessed through a web browser using the URL `http://localhost:<port_number>/mobile_map.html`. Type the URL into the Opera Mobile browser's address bar and hit *Enter*. You should see a map appear as shown in the following screenshot:



17. You can use the mouse wheel and the *Ctrl* key to zoom in and out. Also, keep in mind that the ArcGIS Server API for JavaScript also supports gestures, so you can use a pinch gesture to zoom in and out as well. However, keep in mind that this will not work in the emulator. I have elected to remove the zoom slider from this application, but you can always re-enable it within the map options object.
18. Close the emulated application and, back in the Opera Mobile Emulator, select a **Landscape mode** under **Resolution**. Press **Launch**, and enter the URL of your hosted application in the browser address bar again. Verify that your map displays correctly in landscape mode:



This is about as simple as a mapping application can get, but hopefully it illustrates the basic characteristics of building a mobile mapping application.

Integrating the geolocation API

You can use the HTML5 geolocation API in your ArcGIS Server applications to get the location of a mobile device. You can also use the geolocation API to get the location from a standard web application, but this isn't nearly as accurate since it uses the IP address rather than GPS or cell tower triangulation.

This API has built-in security that requires explicit permission from the end user before this functionality can be used. Both mobile and web applications that use geolocation will prompt the user. This prompt will appear similar to the following screenshot:

Once you have the user's permission, using the API to retrieve location data is pretty straightforward.

The `Geolocation.getCurrentPosition()` method returns the current location of the mobile device. The `Geolocation.watchPosition()` method continually monitors the device location with a callback method being fired each time the position changes. So, if your application needs to be able to track the location of a device over time, then you'll want to use `watchPosition()` instead of `getCurrentPosition()`, which simply gets the location at a single point in time.

The following code contains a simple code example that uses the geolocation API. The first thing we do is check to see if the browser supports the geolocation API. This is done with the `navigator.geolocation` property, which returns a value of true or false. Generally, this will also prompt the user to allow the application to collect the current location and also make sure that the browser supports geolocation.

To see if your browser supports geolocation or any other HTML5 feature, go to `http://caniuse.com/`.

If the browser supports the geolocation API, and the end user gives permission to collect the location, then we call the `geolocation.getCurrentPosition()` method. The first parameter passed into this method indicates a success callback function that will be executed if the device is successfully located (`zoomToLocation()` in this example). You can also specify an error callback function (`locationError()` here).

If everything works okay, a `Position` object is passed to the success callback function. This `Position` object can then be examined to obtain the latitude/longitude coordinates of the location. That's what we've done in the following `zoomToLocation()` function. This function then obtains the latitude/longitude coordinates and plots the point on the map:

```
    if (navigator.geolocation){
      navigator.geolocation.getCurrentPosition(zoomToLocation,
locationError);
    }

    function zoomToLocation(location) {
      var symbol = new SimpleMarkerSymbol();
      symbol.setStyle(SimpleMarkerSymbol.STYLE_SQUARE);
      symbol.setColor(new Color([153,0,51,0.75]));
      var pt = esri.geometry.geographicToWebMercator(new
      Point(location.coords.longitude, location.coords.latitude));
      var graphic = new Graphic(pt, symbol);
```

```
    map.graphics.add(graphic);
    map.centerAndZoom(pt, 16);
  }
  function locationError(error) {
    switch (error.code) {
      case error.PERMISSION_DENIED:
        alert("Location not provided");
        break;
      case error.POSITION_UNAVAILABLE:
        alert("Current location not available");
        break;
      case error.TIMEOUT:
        alert("Timeout");
        break;
      default:
        alert("unknown error");
  break;
  }
  }
```

Practice time with the geolocation API

In this exercise, you will learn how to integrate the geolocation API into an ArcGIS Server API for a JavaScript application by performing the following steps:

1. Open the JavaScript sandbox at `https://developers.arcgis.com/javascript/3/sandbox/sandbox.html`.
2. Remove the JavaScript content from the `<script>` tag that I have highlighted in the following code snippet:

```
<script>
   var map;

   require(["esri/map", "dojo/domReady!"], function(Map) {
       map = new Map("map", {
           basemap: "topo",  //For full list ...
           center: [-122.45, 37.75],
           zoom: 13
       });
   });
</script>
```

3. Add the references in the following code snippet for the objects that we'll use in this exercise:

```
<script>
   require([
        "esri/map",
        "esri/symbols/SimpleMarkerSymbol",
        "esri/graphic",
        "esri/geometry/Point",
        "esri/geometry/webMercatorUtils",
        "dojo/_base/Color",
        "dojo/domReady!"
      ], function(Map, SimpleMarkerSymbol, Graphic, Point,
webMercatorUtils,
      Color) {
    });
 </script>
```

4. Create a new `Map` object centered on the San Diego, CA area with a `streets` basemap layer. This will serve as the default map location if the browser you are using doesn't support the geolocation API or if permission to access the current device location is not granted by the user:

```
var map;

require([
    "esri/map",
    "esri/symbols/SimpleMarkerSymbol",
    "esri/graphic",
    "esri/geometry/Point",
    "esri/geometry/webMercatorUtils",
    "dojo/_base/Color",
    "dojo/domReady!"
], function (Map, SimpleMarkerSymbol, Graphic, Point,
webMercatorUtils, Color)
{
    map = new Map("mapDiv", {
        basemap: "streets",
        center: [-117.148, 32.706],
        zoom: 12
    });
});
```

5. Create an `if` statement that checks if the browser supports the geolocation API and is given permission by the user to access the current device location. Do this by checking the value of the `navigator.geolocation` property. If the browser supports the geolocation API and permission is given by the end user, then this property will contain `true`. Do this check within an event listener that responds to the map's load event. Otherwise, things can run so quickly, you'll end up in the situation where your code is trying to draw graphics on the map before the map is ready:

```
map = new Map("mapDiv", {
          basemap: "streets",
          center:[-117.148, 32.706], //long, lat
          zoom: 12
        });
on(map, "load", function () {
    if (navigator.geolocation) {
       navigator.geolocation.
       getCurrentPosition(zoomToLocation, locationError);
    }
})
```

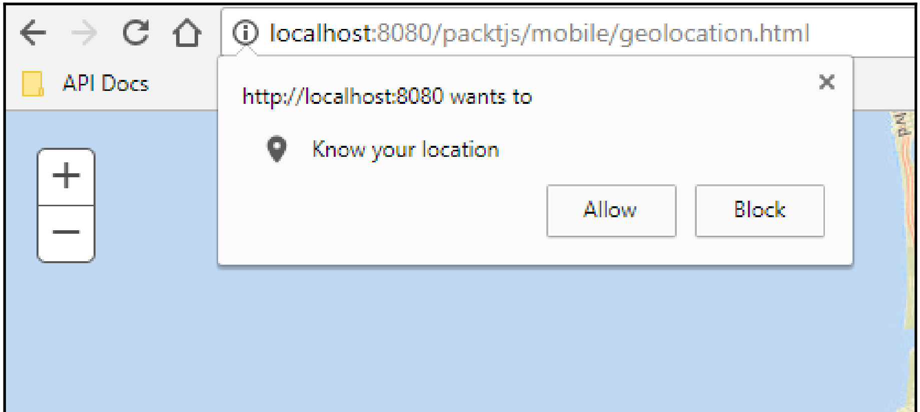
6. As you can see from the code you've added in the last step, the `geolocation.getCurrentPosition()` function defines two callback functions; one for success (`zoomToLocation`) and one for failure (`locationError`). In this step, you'll create the success callback function by adding the code block you see in the following code snippet:

```
 function zoomToLocation(location) {
   var symbol = new SimpleMarkerSymbol();
    symbol.setStyle(SimpleMarkerSymbol.STYLE_SQUARE);
    symbol.setColor(new dojo.Color([153,0,51,0.75]));
    var pt = webMercatorUtils.geographicToWebMercator(new
Point(location.coords.longitude,
location.coords.latitude));
    var graphic = new Graphic(pt, symbol);
    map.graphics.add(graphic);
    map.centerAndZoom(pt, 16);
  }
```

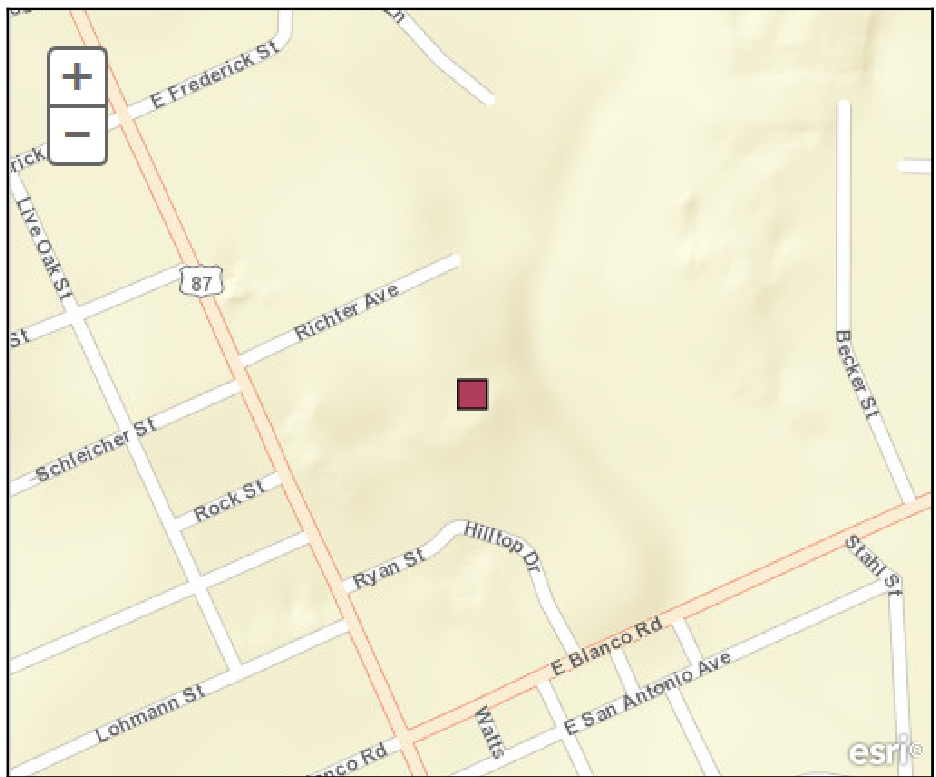
7. Now, let's add the error callback function called `locationError()`. This function will test for various types of error related to not being able to find the current location of the device. Add the function in the following code snippet, just after the success callback function you created in the last step:

```
function locationError(error) {
  switch (error.code) {
    case error.PERMISSION_DENIED:
      alert("Location not provided");
      break;
    case error.POSITION_UNAVAILABLE:
      alert("Current location not available");
      break;
    case error.TIMEOUT:
      alert("Timeout");
      break;
    default:
      alert("unknown error");
      break;
    }
  }
```

8. Click the **Refresh** button in the Sandbox. Initially, you should see a message similar to that displayed in the following screenshot:



9. Click **Allow**, **Share Location**, or whatever your chosen browser prompts you with. If the browser you are using supports the geolocation API, then a new map should be displayed with your current location represented by a symbol. Your location will obviously differ from mine. You will probably notice that the accuracy is pretty poor when accessed via a standard (non-mobile) browser. For an extra challenge, access the application via the mobile emulator that you used in the last practice and compare your results!



Summary

Mobile GIS applications are becoming very popular and the ArcGIS Server API for JavaScript can be used to quickly develop applications that are supported in both web and mobile applications. The API comes with built-in gesture support and works on iOS and Android platforms. The compact version of the API provides a smaller footprint that downloads quickly on mobile platforms and is recommended for development. In production, you should create an optimized build by using the Web Optimizer.

To make your application location-aware, you can use the HTML5 geolocation API, which supports both single-shot and continuous monitoring of the device location.

And that's just about it with regard to core API functionality. In the appendices, we'll provide what we hope is useful supplemental information relating to the Dojo elements that can help you lay out the components of your application, and a look ahead to version 4 of the ArcGIS API for JavaScript.

Looking Ahead - Version 4 of the ArcGIS API for JavaScript

Version 4 of the ArcGIS API for JavaScript is a radical reimagining of the API by Esri in order to achieve a number of aims. Why start over? Basically, because the API has grown greatly in capability and therefore in complexity. Esri has been under pressure to add more and more features and, as with any development project that outlives its original scope, the API has become a mass of complex and often contradicting classes, methods, and workflows.

If you have been reading this book thinking *Oh great! I've just spent hours of my life learning about* v3, *when v4 is out already and it's totally different!* then rest assured. Although it's undoubtedly possible that Esri will, at some point, pull the plug on v3.x and pressure developers to go and learn v4, that time is not now. The reason is that there is plenty of stuff that v4 can't do yet that v3 can, and does, very well. So, for now, and for an indeterminate period in the future, v3 is still very relevant and is still being developed. Esri continues to fix bugs and add great new features to v3, and shows no sign of slowing down at the time of writing. And although there are some major differences between versions, a lot of what you have learned in v3 is directly transferable to v4.

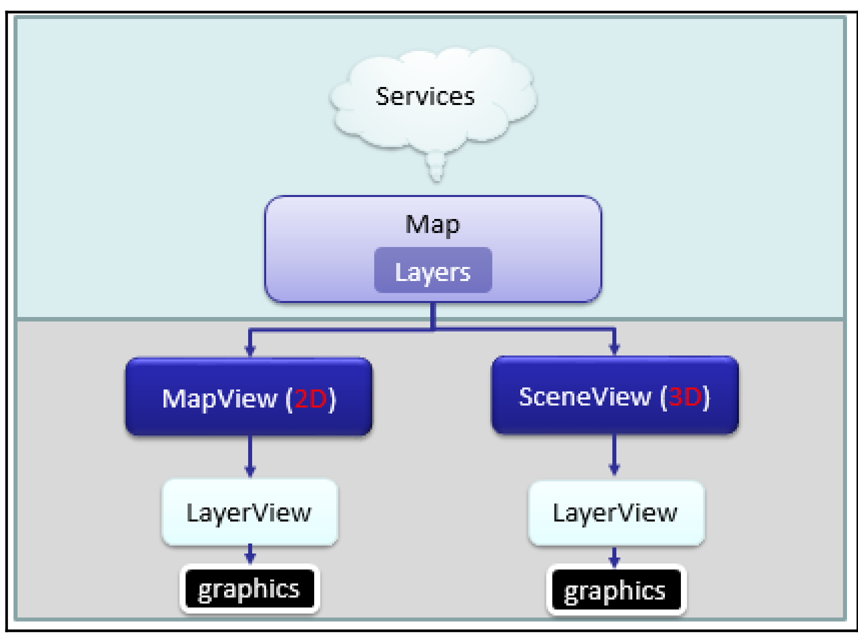
We can't possibly cover all the changes in v4 in an appendix - that requires a book in its own right. However, we can give you some idea of what features you can look forward to.

The main changes in v4, and the topics for this appendix, are:

- Steps for creating 2D maps
- New and changed layers
- 3D mapping and symbology

Steps for creating 2D maps

One of the main things that Esri wanted to introduce in version 4 of the ArcGIS API for JavaScript is support for 3D. This would have been a very messy addition to the API in its current format and therefore a new design was required. One of the most obvious changes is the way you create standard 2D maps in v4. In order to understand these differences, consider the following diagram:



The first thing to notice in version 4 of the ArcGIS API for JavaScript is that the map class exists purely as a container for GIS services. As a pure data source, it has no visible attributes and is therefore never actually rendered. This allows the API to deal with both 2D and 3D representations of the map and is very powerful anyway, because it lets us access the underlying layers in our service without ever having to draw a map. Instead, the visual representation of the map's data depends on views, and we have a different view object depending on whether we want to display that data in 2D (in which case we use a `MapView`) or 3D (in which case we use a `SceneView`). Each of these view objects controls how the data hosted by the map is displayed.

Let's see how this translates to creating a simple 2D map. Consider the following code:

```
<link rel="stylesheet" href="https://js.arcgis.com/4.4/esri/css/main.css">
<script src="https://js.arcgis.com/4.4/"></script>

<script>
require([
  "esri/Map",
  "esri/views/MapView",
  "dojo/domReady!"
], function(Map, MapView) {
  // Code to create the map and view will go here
  var map = new Map({
    basemap: "streets"
  });
  var view = new MapView({
    container: "viewDiv",   // Reference to the DOM node that will contain
the view
    map: map,                    // References the map object created in
step 3
   center: [-98.526077, 29.418965],
   zoom: 11
  });
});
</script>
```

First, we reference the Esri style sheet and v4.x of the ArcGIS API for JavaScript.

Then, we create a `<script>` tag and use Dojo's `require()` method to reference the Dojo and ArcGIS API modules we need. In this instance, it's the `Map` class (for the data) and the `MapView` (for 2D rendering). The `dojo/domReady!` plugin merely waits until the page loads before executing the JavaScript.

In the map object constructor, we define the data source. In this instance, that's the built-in `streets` layer hosted on ArcGIS Online.

In the `MapView` constructor, we specify which div in the page to render the map view, the map object that is the data source for the view, and then the zoom level and the center point of the initial extent.

So, all you really need to be aware of here is that the properties that you would have previously associated with the `Map` object, you now define within the `MapView`. You link the two by making the `map` property in the `MapView` constructor reference the `Map` object that "points" at the data that the view will display.

The biggest takeaway from this is that we now have the flexibility of treating the `Map` as a pure data source. We can access all the data associated with a map without ever having to draw it. So, for example, we can create half a dozen map objects, attach the layers that we're interested in, and not actually render them until we need them. When we want to display them, all we need to do is associate them with the appropriate view object.

To access the layers, you access properties on the `Map`. The `layers` property returns all operational layers in the map, `basemap` returns - you guessed it - the `basemap`, and `allLayers` returns both.

Remember that what you're getting back here is the data that comprises the layer and has nothing to do with how this is represented. We cannot stress this enough. However, it is a very powerful feature because it means you have direct access to this data if you wanted to show it somewhere else, such as in a chart control, or use it for integration with another application.

Accessing layers

Because of the way things are now separated into maps and views, there is no direct way for you to access feature data directly from the layer. Instead, you must access this data via the view, or more specifically, the `LayerView` object, which is the view's *displa*y of the data rather than the data itself. We'll now talk about how to access and work with feature data via the `LayerView`.

To access feature data via the `LayerView`, you first have to be sure that the `LayerView` has been loaded. You can do this by handling the promise returned by the `whenLayerView()` method of the `LayerView`'s. The type of object returned depends on the type of layer you're dealing with. If this is a feature layer, you get back a `FeatureLayerView` object and you can use its `queryFeatures()` method to access its data, as shown in the following code example:

```
var featureLayer = new FeatureLayer({
          url: ""https://services.arcgis.com/...",
          outFields: ["Name"]
        });

        map.add(featureLayer);
        view.whenLayerView(featureLayer).then(function (lyrView) {
          lyrView.watch("updating", function (val) {
            if (!val) {
               // count the total number of features
```

```
          lyrView.queryFeatures().then(function (results){
            // do something with the results
          }, function(err){
            console.log(err);
          });
        }
      });
    });
```

As well as using `queryFeatures()` to access the individual features of the `FeatureLayerView`, there are a couple of other really useful functions for working with feature layers:

- `queryExtent()` returns the extent of the features in the `LayerView`
- `queryFeatureCount()` gives you a count of the number of features in the `LayerView`
- `queryObjectIds()` gives you an array of object IDs of all features in the `LayerView`, which you can then use as the basis for further queries

New and changed layers

So far, version 4 of the ArcGIS API for JavaScript only supports a limited number of layers compared to v3.x. This will undoubtedly change over time. Some of the most interesting changes are listed as follows:

- `GraphicsLayer`
- `FeatureLayer`
- `MapImageLayer`
- `VectorTileLayers`
- `GroupLayers`
- `SceneLayers`

GraphicsLayer

The `GraphicsLayer` is just a clear layer that overlays the map and allows you to mark up the map with your own custom graphics. In this sense, it is very similar to what you've seen in v3.x. However, in v4 you can add graphics of different geometries to the same graphics layer and this has an important knock-on effect. Because the `GraphicsLayer` supports different geometries, you must define any renderers or popups against single graphics and not for the graphics layer as a whole. If you need this functionality, consider using a feature layer instead.

FeatureLayer

The `FeatureLayer` is the most able of all the layers and is probably the one that is used most often in ArcGIS Server web mapping applications. Functionally, it is very similar to what you are used to using.

You can create a `featureLayer` from the URL of a layer in the ArcGIS Server services directory:

```
var featureLayer = new FeatureLayer({
  url: "http://..."
});
```

You can do it from a portal item ID:

```
var featureLayer = new FeatureLayer({
  portalItem: {
   id: "myID"
   }
});
```

You can also do it from a feature collection:

```
var featureLayer = new FeatureLayer({
  objectIdField: "item_id",
  geometryType: "point",
  // Fields in the layer
  fields: [{
    name: "item_id",
    alias: "Item ID",
    type: "oid"
  }, {...}],
  // The renderer for the layer
  renderer: new SimpleRenderer({
    type: "simple",
```

```
      symbol: new SimpleMarkerSymbol({
        ...
      })
     }),
     popupTemplate: {
       title: "{title}",
       content: "{description}"
     },
     // Collection of graphics
     source: [graphic1, graphic2, graphic3]
   })
```

When we use a feature collection to create a feature layer, what we are doing is defining the source for a `FeatureLayer` manually. This basically involves providing a collection of graphics, defining the renderer for those graphics, as well as the attribute fields and popup information. The benefit of this approach is that you can use a single renderer for multiple graphics as well as a single popup. You can also query this `FeatureLayer` the same way as you would query a `FeatureLayer` based on an ArcGIS Server map service. This is a powerful concept which, although introduced in v3 of the API, is much easier to work with in v4.

MapImageLayer

If you've worked with v3 of the API, `MapImageLayer` is probably one of the few layer names that won't be familiar to you. However, the nature of the layer will be very familiar indeed, because this is the `ArcGISDynamicMapServiceLayer`, mercifully renamed for version 4 of the ArcGIS API for JavaScript.

Unlike the tiled image service that you typically use for basemaps where all of the tiles are precached, the `MapImageLayer` serves up an image for the entire map extent, which it creates dynamically.

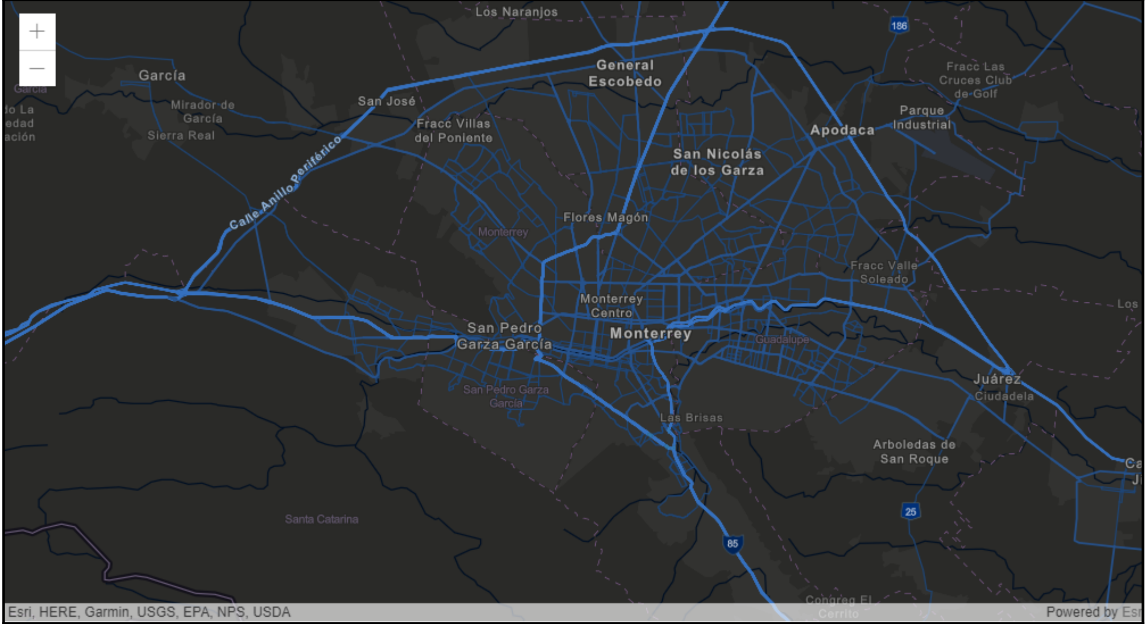
One dramatic improvement from the previous versions of the API is the ease in which you can define the visibility and any definition expressions for the sublayers in the map service. This was quite a procedure before and is now simple and intuitive, as can be seen in the following code example:

```
var myMILayer = new MapImageLayer({
  url: // url,
  sublayers: [{
    id: 0,
    visible: true
  }, {
    id: 1,
```

```
      visible: true
    }, {
      id: 2,
      visible: true,
      // filter layer contents with definition expression
      definitionExpression: "pop2000 > 5000000"
    }, {
      id: 3,
      visible: false
    }]
  });
```

VectorTileLayer

The `VectorTileLayer` is new and is effectively a way to style vector data dynamically within the browser without sacrificing performance. Basically, the `VectorTileLayer` accesses a cached tile and renders it in vector format, producing very sharply-defined and attractive maps:



Each layer is rendered using styling information separate from the tiles. This means that one set of vector tiles may be styled numerous ways without having to generate a new image cache for each style. This saves space and speeds up the process for creating new map styles.

Style sheets may contain multiple options for rendering the same type of feature. In a street layer, for example, major highways might have three symbology options. The symbology can be changed on the client without having to make a request for a new tile from the server. This enables maps to be dynamic on the client while still taking advantage of map caches.

To create a `VectorTileLayer`, you do not reference the map service, but one of its style files (in JSON format) instead. Because they are based on the Mapbox vector tile specification, they play very well with Mapbox data too, making it a snap to include Mapbox data in your ArcGIS Server application. To do this, you need to provide a Mapbox developer token as shown in the following code example:

```
VectorTileLayer.ACCESS_TOKEN = "developer_access_token";
var tileLayer = new VectorTileLayer({
  url: "mapbox://styles/mapbox/streets-v8"
});
var map = new Map({
  layers: [tilelayer]
});
```

Instances of a `VectorTileLayer` sound like magic, and to some degree they are! But be aware that at the time of writing they have some limitations. Consult the documentation for details.

GroupLayers

A `GroupLayer` isn't a way to access a web service; instead, it allows you to group together many layers of different types and deal with them all collectively. A typical use case is to control the visibility of a group of layers, rather than treating them individually. For example, you may have a couple of feature layers, a `MapImageLayer`, and a graphics layer all related to property zoning information and want to be able to toggle their visibility on or off all in one go. A group layer is perfect for such a task:

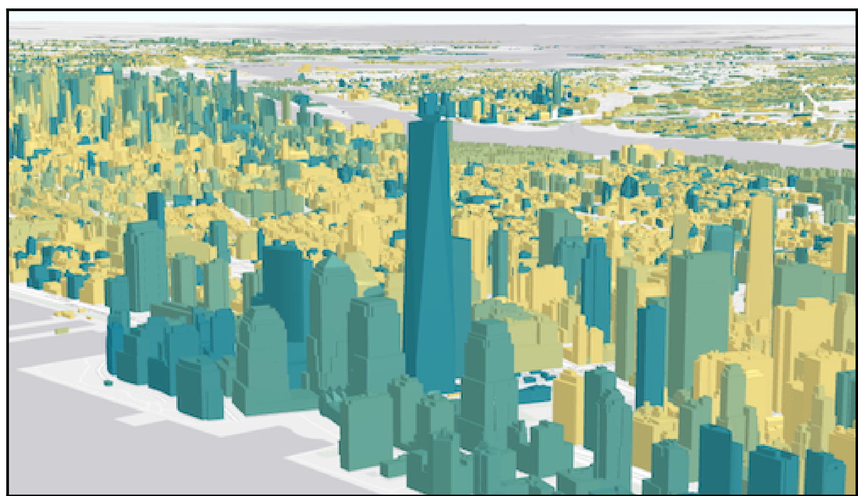
```
var layers = {
  new FeatureLayer({...}},
  new FeatureLayer({...}),
  new GraphicsLayer({...})
}
```

```
var groupLayer = new GroupLayer({ layers });
var map = new Map({
  basemap: "streets",
  layers: [groupLayer]
});
```

SceneLayers

The `SceneLayer` is a layer type designed for on-demand streaming and displaying large amounts of data in a `SceneView`, which we'll cover when we talk about 3D mapping in the next section. `SceneLayer` supports two geometry types: point and 3D objects (for example Buildings).

The `SceneLayer` displays data published to a Scene Service. Scene Services can hold large volumes of features in an open format that is suitable for web streaming. The `SceneLayer` loads these features progressively, starting from coarse representations before refining them in higher detail as necessary for close-up views:



You can add a `SceneLayer` in exactly the same way as any other layer, as shown in the following code example:

```
sceneLayer = new SceneLayer({
  url: "http://scene.arcgis.com/arcgis/rest/services/Hosted/
              Building_Hamburg/SceneServer/layers/0";
});
```

You can then visualize them using a range of different symbologies and renderers. A `SceneLayer` also supports visual variables, which allow you to easily visualize numeric data in the layer with continuous color. For example, the service used in the preceding image represents building features containing a numeric attribute storing the number of residents in each building. The renderer for the layer uses the color visual variable to shade each feature, along with a continuous yellow to red color ramp based on the value of the given field. Buildings with a high occupancy rate are shaded with red, whereas buildings with a low rate are yellow. Buildings with values between the low and high values are assigned intermediate colors.

3D mapping and symbology

Without a doubt, the most exciting thing about version 4 of the ArcGIS API for JavaScript is its support for 3D mapping.

Working with a 3D view is much like working with a 2D view. For example, both views share the same implementation for layers, renderers, tasks, geometry, symbology, popups, and navigation, but 3D adds 3D-specific concepts such as environment (atmosphere and lighting) and the camera. This extra 3D capability is what we'll cover in this section.

Scenes

A scene is symbolized 3D geospatial content that includes a multiscale basemap, a collection of 2D and 3D layers, styles, and configurations that allow you to visualize and analyze geographic information in an intuitive and interactive 3D environment. You can author these scenes either in the ArcGIS Online or Portal Scene Viewer software, or on the desktop using ArcGIS Pro.

You can choose between a global or local scene to best display your data, such as global weather patterns, shipping lanes, or underground utilities. The main differences between local scenes and global scenes are that local scenes can display data that has a spatial reference in a local **projected coordinate system** (**PCS**) (but not, currently, a geographic coordinate system); the terrain and layers are projected on a planar surface rather than on a sphere. Local scenes are best used for displaying or analyzing data at the local or city scale that have a fixed extent in which you work. Local scenes are helpful for urban planning and visualization where you can view campus facilities or building developments. You can navigate underground and interact with subsurface 3D data, such as utility networks or earthquake data.

Publishing these scenes creates a *scene service*, which can be brought into your web mapping application just like any other map service in version 4 of the ArcGIS API for JavaScript.

Creating the map

The steps for creating a 3D map are very similar to those you used to create a 2D map, but with a few extra properties you can play with.

First, you define the map, specifying a basemap. Because the map will be rendered in 3D, it will help to have some elevation data, which you define in the map's `ground` property.

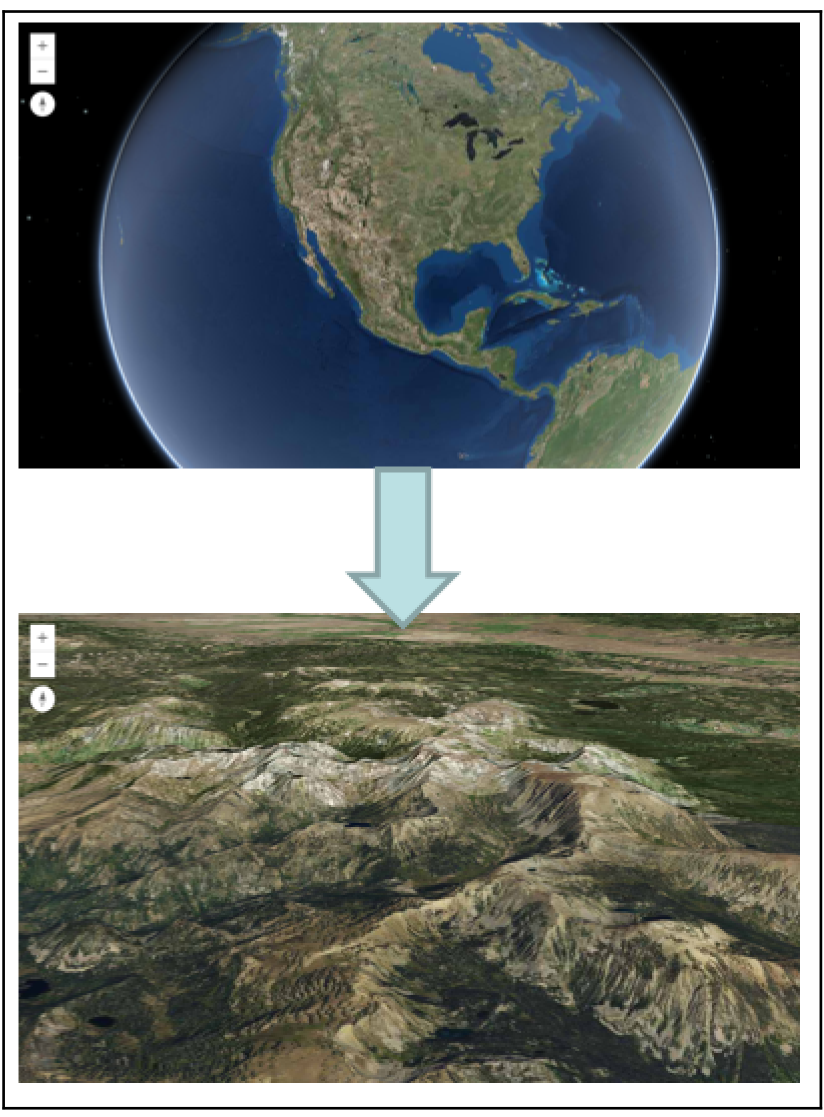
Then, you must create the view. Instead of using `MapView` like you would in a 2D map, you use `SceneView`. You can specify a `viewingMode` which is global for large extents (rendering the earth as a globe), or local for more discrete areas. For global mode, the layers must be in WGS84 or Web Mercator projection or reprojectable to the same, whereas for local mode you can use any projected coordinate system. The default `viewingMode` if you don't specify it will be `global` if the view's spatial reference is `WGS84` or `WebMercator`, and `local` otherwise.

You also have the option to set properties for camera and elevation, which will allow you to position the initial viewpoint. Finally, you can add further layers and visualize them using special 3D symbology.

The following code example shows how to create a map and specify the basemap and elevation data (using the `ground` property):

```
require([
  "esri/Map",
  "esri/views/SceneView",
  "dojo/domReady!"
], function(Map, SceneView) {
  var map = new Map({
    basemap: "satellite",
    ground: "world-elevation"
  });
  var view = new SceneView({
    container: "sceneDiv",
    map: map,
    center: [-98.5, 29.4],
    scale: 50000000,
    viewingMode: 'global'
  });
});
```

Here, we're just using Esri's World Elevation Data Service. Then, we define a `SceneView` with an initial extent based on a center point and scale, and set the viewing mode to `global`. In global view mode, you can see the curvature of the earth, as shown in the first screenshot as follows. As we zoom in, the elevation data becomes visible, as shown in the second screenshot:



Now, we'll dig into some of those options in more detail. We'll start with the map's `ground` property, then look at some of the properties of the `SceneView` class such as environment and camera, and then move on to adding and symbolizing layers.

Setting elevation data

The API uses a special type of layer called an `ElevationLayer` for showing elevation data in 3D maps. You can either use Esri's World Elevation Data service for this, in which case you just set the map's `ground` property to a string reading word hyphen evaluation, or you can use your own Image Services for this.

The map's `ground` property accepts a collection of elevation layers and you can add these using the `ground` property's `layer.add()` method, as shown in the following example:

```
var layer = new ElevationLayer({
  url: "//elevation3d.arcgis.com/arcgis/rest/services/
  WorldElevation3D/Terrain3D/ImageServer"
});
map.ground.layers.add(layer);
```

Don't be tempted to add these elevation layers as operational layers. They won't work if you do. Always specify them in the map's `ground` property.

Setting the camera

The `SceneView` class is very similar to the `MapView` class you have already worked with, but it has a couple of special properties for working with 3D data. The main ones are `camera` and `environment`. Let's look at `camera` first.

You can think of the `camera` property as a literal camera that is located at a specific point on your map, pointing in a specific direction and tilted at a specific angle.

There are a couple of different ways you can define these options in the camera object. The following code example demonstrates how to specify the camera position using an array where the first element is longitude, the second latitude, and the third elevation in meters. It also defines the heading of the camera in degrees. The heading is zero when north is the top of the screen. It increases as the view rotates clockwise. The angles are always normalized between 0 and 360 degrees:

```
var view = new SceneView({
  camera: {
    position: [
```

```
          -122, // longitude
            38, // latitude
         50000  // elevation in meters
       ],
       heading: 95
     }
   });
```

The following example uses the `x`, `y`, and `z` properties in an object literal to specify longitude, latitude, and elevation respectively. It also specifies the tilt of the camera in degrees with respect to the surface as projected down from the camera position. Tilt is zero when looking straight down at the surface and 90 degrees when the camera is looking parallel to the surface:

```
var view = new SceneView({
  camera: {
    position: {
      x: -100, // lon
      y: 45,   // lat
      z: 10654 // elevation in meters
    },
    tilt: 65
  }
});
```

To move the camera to a new position, it's easiest just to clone the existing camera object using its `clone()` method and then just update the values that have changed. You can then set the camera property of the `SceneView` to point to this new instance of the camera object, but if you do, the transition will be jarring. You can make this transition smoother by using the API's built-in animation capability. Just call the `goTo()` method belonging to `SceneView` and pass in the new camera object, or positional properties, or even a graphic or extent to animate to.

The following code clones the existing camera object and then zooms smoothly to it by using `goTo()`:

```
var newCam = view.camera.clone();
newCam.heading = 180;
view.camera = newCam;
view.goTo(newCam);
```

The following code shows how you can chain a sequence of animations together by using the promise returned by the `goTo()` method when an animation completes:

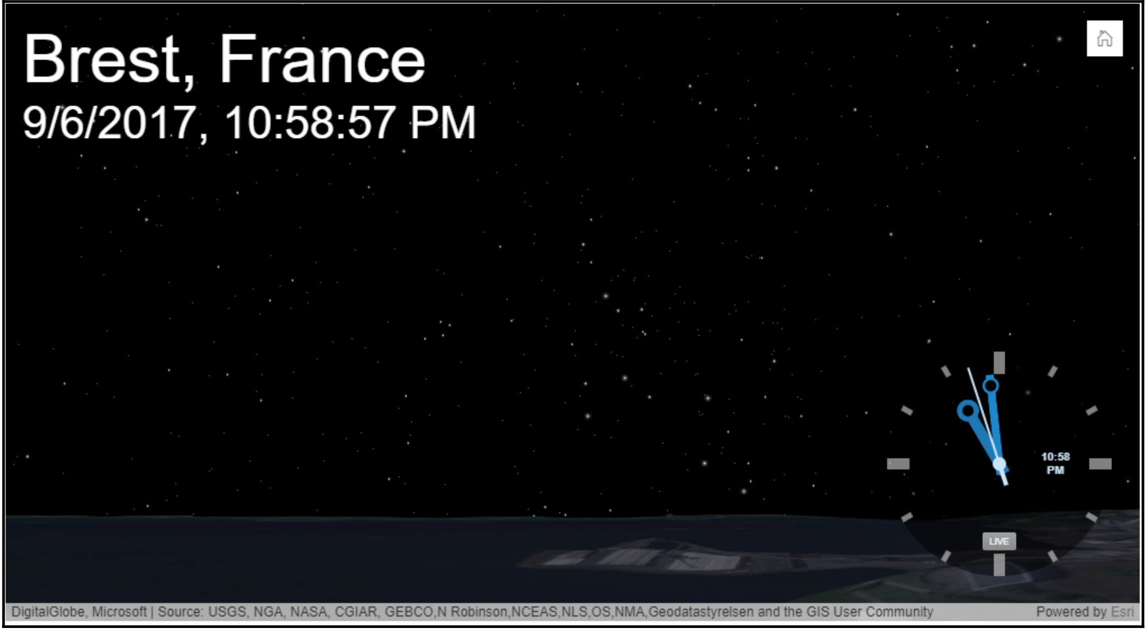
```
view.goTo(graphic1)
    .then(function() {
```

```
      return view.goTo(graphic2);
    })
    .then(function() {
      return view.goTo(graphic3);
    });
```

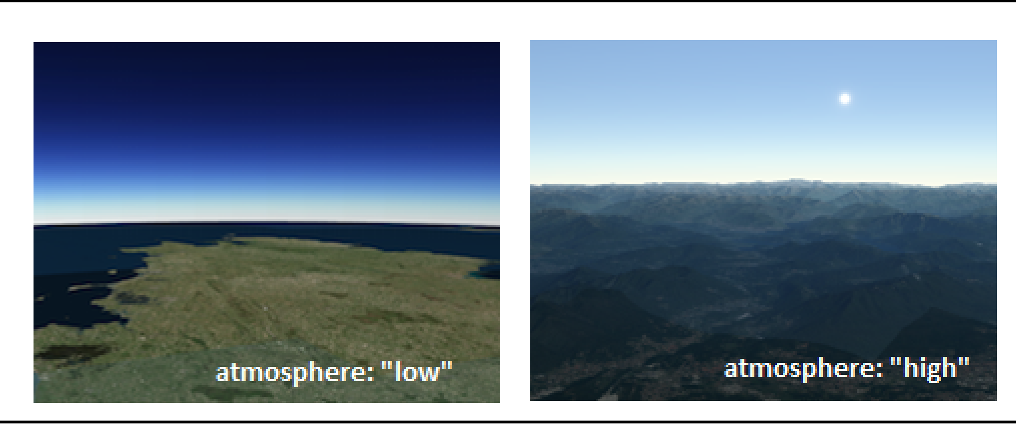
Specifying the environment

The environment property has three main properties, the main one being `lighting`, which lets you control the position of the sun and allows you to add really nice shadow effects to buildings and other features. We'll look at this property in more detail in a bit.

Another is `starsEnabled`, which defines whether or not to show the stars. If you choose to include stars, their position is accurate depending on the current date or whichever date you define in the view, which is quite remarkable when you think about it!



The last is `atmosphereEnabled`, which lets you decide whether or not to visualize the atmosphere. If you want to, you have a choice of high-quality visualization or low-quality visualization. The quality of the atmosphere can have significant effects on performance, so use high-quality only if you know your users have high-end machines. The difference between the two is shown on the slide:

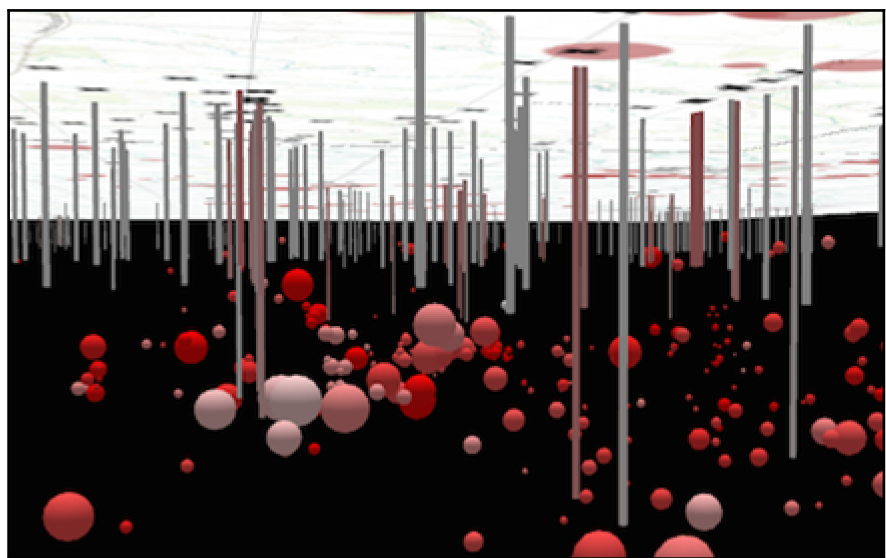


The environment's lighting can be altered by changing the values of four different properties. The first is the date and time, which will determine the sun's position in relation to the earth and also determines the position of the stars if you have enabled them using the `starsEnabled` property. Whether this is automatically updated when the camera position changes is determined by the `cameraTrackingEnabledProperty`. The whole idea of simulating the sun is really all about creating some nice shadow effects and this is the default behavior. However, if you don't want to show shadows, set the `directShadowsEnabled` property to false. The last property is `ambientOcclusionEnabled`. If you enable this, it will try and calculate how light should be reflected off surfaces; this requires a high-end machine to work well, so it is disabled by default and is always turned off on mobile devices:

Local scenes

So far, we have only looked at global scenes, which are great for larger extents, but not so useful for smaller extents. Whereas global scenes render the earth as a globe, local scenes render the surface on a flat plane and allow you to navigate not just on the scene, but also under it, to show underground features such as this sample of Kansas oil wells following image. If you don't want users to be able to do this, you can prevent it and other operations by setting the `SceneView`'s constraints property.

You can only render a local scene if the view's spatial reference is in a projected coordinate system and all of its layers are either in the same spatial reference or reprojectable to it, so cached layers in a different spatial reference won't work. Geographic coordinate systems are not supported at the time of writing, but Esri claims they are working on this capability. To limit the extent of the scene, pass the required extent to the `clippingArea` property of the `SceneView`.



3D symbology and rendering

Now that we have 3D maps, we need 3D symbols so that we can highlight features of interest. Version 4 of the ArcGIS API for JavaScript supports the following main 3D symbol classes, which can be easily identified in the documentation by the fact that they have "3D" somewhere in their name:

- Points (`PointSymbol3D`)
- Lines (`LineSymbol3D`)
- Polygons (`PolygonSymbol3D`)
- Mesh (`MeshSymbol3D`)
- Labels (`LabelSymbol3D`)

All these symbols inherit from the `Symbol3D` class. The only one that might seem unfamiliar is Mesh, which relates to symbolizing features from a scene layer and we'll get onto those in a bit.

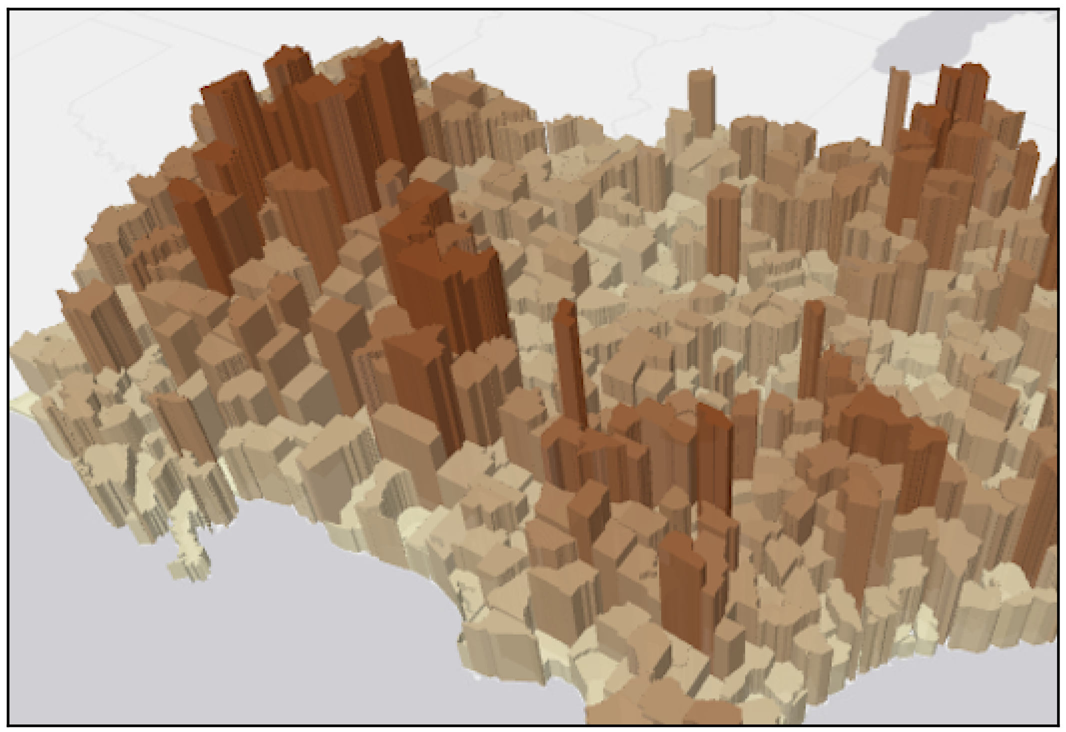
Each of these main symbol classes must be associated with a symbol layer, as demonstrated in the following table:

Symbol Layer	Symbol	Geometry	Description	Size	Example
IconSymbol3DLayer	PointSymbol3D, PolygonSymbol3D	Point, Polygon	flat	points/pixels	
ObjectSymbol3DLayer	PointSymbol3D, PolygonSymbol3D	Point, Polygon	volumetric	meters	
LineSymbol3DLayer	LineSymbol3D, PolygonSymbol3D	Polyline, Polygon	flat	points/pixels	
PathSymbol3DLayer	LineSymbol3D	Polyline	volumetric	meters	
FillSymbol3DLayer	PolygonSymbol3D, MeshSymbol3D	Polygon, Mesh	flat	-	
ExtrudeSymbol3DLayer	PolygonSymbol3D	Polygon	volumetric	meters	
TextSymbol3DLayer	LabelSymbol3D	Point, Polyline, Polygon	flat	points/pixels	Text

For example, the `ExtrudeSymbol3DLayer` renders polygon geometries by extruding them upward from the ground, creating a 3D volumetric object. This is done with a `PolygonSymbol3D` in a `SceneView`. `MapView` does not support 3D symbols.

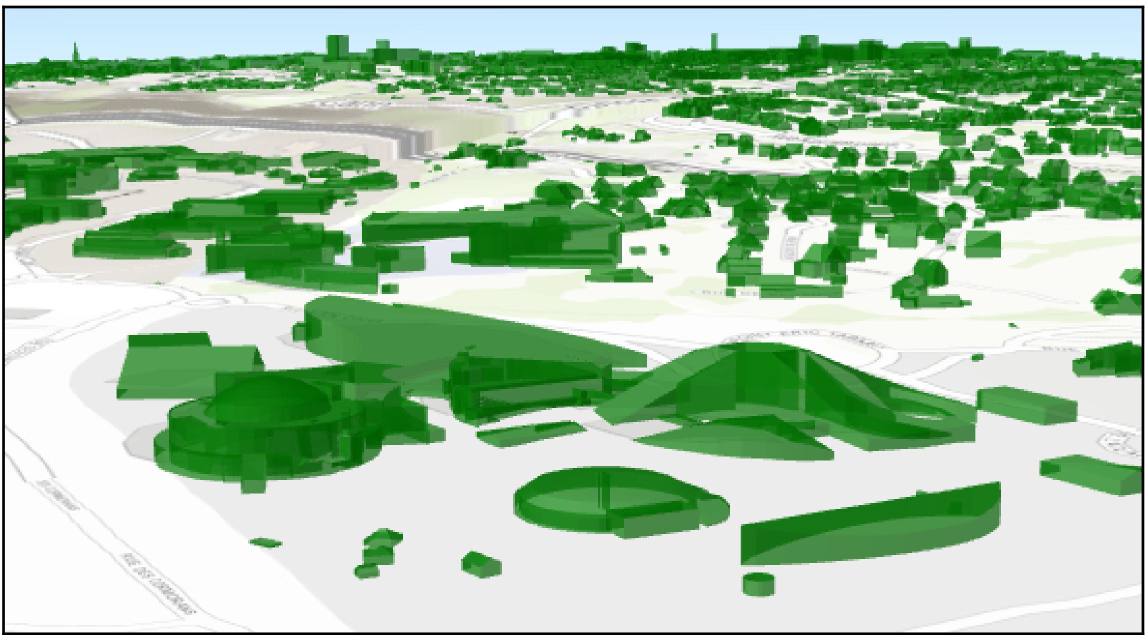
The color of the symbol is set in the `material` property. The height of the extrusion is always defined in meters with the `size` property. Extrusion height and color can also be data-driven by adding size and/or color visual variables to any renderer that uses this symbol layer.

An `ExtrudeSymbol3DLayer` must be added to the `symbolLayers` property of a `PolygonSymbol3D`. Multiple symbol layers may be used in a single symbol. The following image shows a polygon `FeatureLayer` representing building footprints. The features are symbolized with a `PolygonSymbol3D` containing an `ExtrudeSymbol3DLayer`. The extrusion is based on the height of buildings:



`MeshSymbol3D` uses `FillSymbol3DLayer` as its symbol layer. `MeshSymbol3D` is used to render 3D mesh features in a `SceneLayer` in a 3D `SceneView`. The following image shows a `SceneLayer` whose graphics are styled with a `MeshSymbol3D` containing a `FillSymbol3DLayer`.

3D mapping in version 4 of the ArcGIS API for JavaScript uses the same renderers as are present in v3.x of the API. However, a new property has been added to the `SimpleRenderer` object to support what Esri calls visual variables.



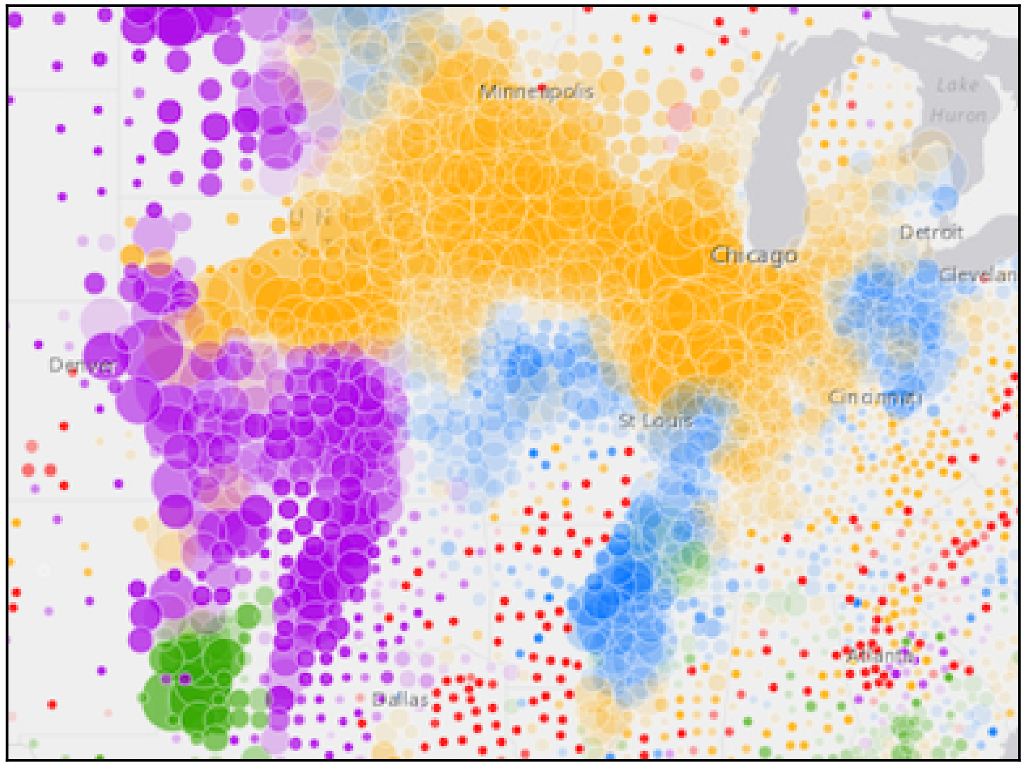
The following code creates this effect:

```
var symbol = new MeshSymbol3D({
  symbolLayers: [new FillSymbol3DLayer({
    material: { color: "green" }
  })]
});
sceneLayer.renderer = new SimpleRenderer({
  symbol: symbol
});
```

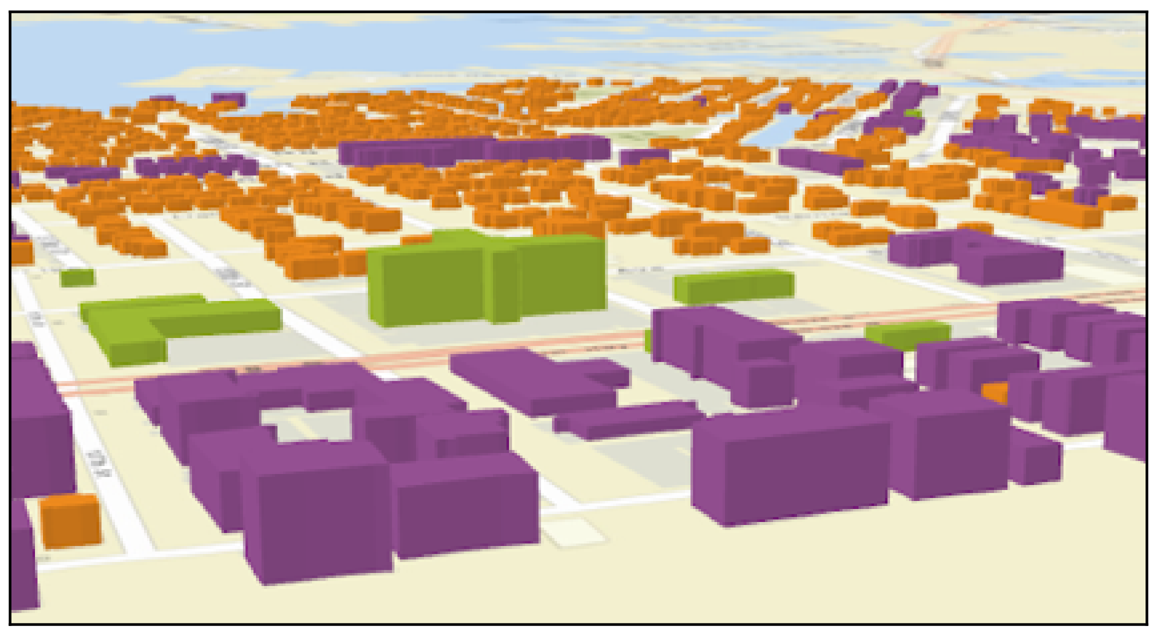
Visual variables define the parameters for data-driven geographic visualizations of numeric data. They allow you to easily map continuous ramps of color, size, opacity, and/or rotation to the minimum and maximum data values of one of the layer's numeric attribute fields.

The `visualVariables` property of the renderer contains an array of objects called *visual variables*. Each object must indicate the type of visual variable to apply (for example color, size, opacity, rotation), the numeric field from which to drive the visualization, and the ramp of visual values to map to the data.

Visual variables allow you to easily create stunning visualizations based on thematic attributes (for example population, education, rank, money, magnitude, and so on) in either 2D or 3D. This is accomplished by mapping data values from a numeric field attribute to color, size, and/or opacity values. The renderer then takes over and resizes or shades features based on the value of the given field and its position relative to the minimum and maximum values. The following example uses three visual variables (size, color, and opacity):



The size visual variable can be used to visualize the true sizes of features (for example tree canopy, road width, building height, and so on) based on their size in the real world. This can be particularly powerful when working in a 3D `SceneView`. The following example shows a layer of building footprints that uses visual variables to extrude each feature to the true height of the buildings based on data stored in an attribute field:



Summary

In this appendix, we looked at the rationale behind version 4 of the ArcGIS API for JavaScript and allayed some concerns about our existing 3.x skills becoming obsolete in the short term. We also considered some of the amazing new features that the API supports, including 3D mapping, vector tile layers, and more. There are plenty more we didn't cover, such as better, easier-to-style widgets, integration with other frameworks like JQuery, and better support for ArcGIS Online and Portal installations, but we would urge you to consult the documentation for further information.

That brings us to the end of the book. We hope we have given you a good enough basis for using the ArcGIS API for JavaScript to encourage you to create some truly outstanding web mapping applications!

Index

2

2D maps
  creating 270
  layers, accessing 272

3

3D
  Camera, setting 282
  Elevation Data, setting 282
  environment, specifying 284
  local scenes 286
  map, creating 280
  mapping 279
  rendering 286
  scenes 279
  symbology 279, 286, 290

A

AJAX (Asynchronous JavaScript and XML) 14
analysis widgets 131
Apache web server
  URL 254
Appcelerator 252
application programming interface (API) 30
ArcGIS API for JavaScript Sandbox
  API modules, loading 35
  application, creating 32
  code, executing 41
  DOM, availability checking 38
  HTML code, creating for page 32
  map, creating 38
  Page Content, creating 40
  page, styling 40
  reference 34
  URL 37, 180
  using 31
ArcGIS API
  geocoding, with locator service for JavaScript 175
  URL 107, 113, 197
ArcGIS Online maps
  adding, to application 240, 243
  adding, to applications with JSON 243
ArcGIS Online
  using 244
ArcGIS Server
  models 211
  tasks 142
asynchronous tasks 216
AttachmentEditor widget
  Edit toolbar 141
attribute
  about 63
  overview 143
auto mode 87

B

basemap toggle widget 110
BasemapGallery widget 108
basemaps
  URL 39
bookmarks widget 110

C

Calcite
  URL 35
ClosestFacility Task 204, 206
code
  splitting, into separate files 27
compact build
  about 253
  using 254
  viewport scale, setting 254

CORS (Cross Origin Resource Sharing) 196
CSS files 28
CSS principles
  about 22
  CSS syntax 23
  external style sheet 27
  inline styling 26
  internal style sheet 26
custom build
  using 260

D

definition expression
  setting 52, 87
Directions widget
  about 125, 202
  URL 202
display mode
  auto mode 87
  defining 85
  on-demand mode 86
  selection only mode 87
  snapshot mode 86
Document Object Model (DOM) 32, 38
Dojo Mobile 252
Dojo Toolkit
  URL 35
Draw toolbar
  URL 101
Dynamic Map Service Layers 48

E

Editing widgets
  about 133
  AttachmentEditor widget 139
  AttributeInspector Widget 137
  Editor widget 133
  TemplatePicker Widget 135

F

feature editing
  about 132
  Editing widgets 133
  FeatureService 132
feature selection 88, 89
FeatureLayer
  about 274
  creating 84
  optional constructor parameters 84
  rendering 89, 95
FindTask
  about 172
  FindParameters 171
  FindResults 172
  used, for accessing feature attributes 171
functions 20

G

Gauge Widget 118
geocoding
  about 175
  input parameter object 176
  JSON address object, inputting 176
  locator object 177
  point object, inputting 177
  process 178
  with locator service 175
geolocation API
  about 252
  integrating 261
  using 263
Geometry 63
Geometry Engine
  about 234
  reference 235
  using 235
Geometry Service
  about 227
  operations 228
  using 230
geoprocessing task
  about 215
  input parameters 214
  services page 213
  using 212, 217
Graphic Geometry
  specifying 64
graphic
  adding, to GraphicsLayer 69
  attributes 63

attributes, assigning 67
attributes, displaying in InfoTemplate 67
creating 68
creating, on map 70
displaying, on map 70
geometry 63
symbol 63
symbolizing 65
GraphicsLayer 274
GroupLayers 277

H

HistogramTimeSlider 126
HomeButton widget 127
HTML Page
code, validating 11
concepts 8
DOCTYPE 9
primary tags 10

I

IdentifyTask
about 162
Identify functionality, implementing 165
identify operation, performing 163
IdentifyParameters 162
IdentifyResult 164
used, for accessing feature attributes 162
InfoTemplate 63
International Terrestrial Reference System (ITRF) 233

J

JavaScript files 28
JavaScript Sandbox
URL 31
JavaScript
case sensitive 16
code, commenting 14
decision support statements 18
functions 20
fundamentals 14
looping statements 19
objects 20
variable data types 17
variables 15
jQuery Mobile 252
JSON (JavaScript Object Notation)
about 38, 110
ArcGIS Online maps, adding to applications 243

L

Layer Classes
using 46
Layer List widget
about 113
implementing 113
layers
FeatureLayer 274
GraphicsLayer 274
GroupLayers 277
MapImageLayer 275
SceneLayers 278
VectorTileLayers 276
LayerSwipe 130
left-to-right (LTR) 121
Legend Widget 121
LocateButton 128
locator object
about 177
AddressCandidate object 178
locator service
URL 179
using 179, 186
looping statements 19

M

Map Events 58
Map Navigation
about 53
Map Extent, obtaining 57
Map Extent, setting 57
toolbars 53
widgets 53
with keyboard 56
with mouse 56
Map Service Layers
Definition Expression, setting 52
Dynamic Map Service Layers 48
Layer Classes, using 46

layers, adding 50
Map Navigation 53
Tiled Map Service Layers 47
Visible Layers, setting 51
working with 44
map
exploring 42
MapImageLayer 275
Measurement Widget 119

O

objects 20
on-demand mode 86
Opera Mobile Classic Emulator
URL 258
optional constructor parameters 84
OverviewMap Widget 123

P

PhoneGap 252
Popup widget 120
Print widget 112
projected coordinate system (PCS) 279
promise
URL 235
Public Safety
URL 114

Q

Query object
about 144
Attribute Queries 145
Fields returned, limiting 148
properties, setting 145
Spatial Queries 147
QueryTask
query, executing 148
results, obtaining 150

R

reverse geocoding process 179
right-to-left (RTL) 121
RouteTask
URL 195
using 192, 195

S

Scalebar Widget 124
SceneLayers 278
scenes 279
Search Widget 118, 187
selection only mode 87
ServiceArea task 206
snapshot mode 86
spatial queries
overview 143
performing 151
Symbol 63
synchronous tasks 216

T

tasks
asynchronous tasks 216
executing 215
synchronous tasks 216
Tiled Map Service Layers 47
time
practicing, with routing 195
practising, with routing 202
TimeSlider widget 129
toolbars
adding, to application 102
buttons, creating 105
creating 103
CSS styles, defining 103
instance, creating of Navigation toolbar 106

U

user interface widgets
about 107
analysis widgets 131
basemap toggle widget 110
BasemapGallery widget 108
bookmarks widget 110
Directions widget 125
Gauge Widget 118
HistogramTimeSlider 126
HomeButton widget 127
Layer List widget 113

LayerSwipe 130
Legend Widget 121
LocateButton 128
Measurement Widget 119
OverviewMap Widget 123
Popup widget 120
Print widget 112
Scalebar Widget 124
Search Widget 118
TimeSlider 129

V

variable data types 17
variables 15
VectorTileLayers 276
viewport scale
setting 254
Visible Layers
setting, from Map Service 51

W

W3C HTML validator
URL 11
Web Optimizer
URL 253
web workers
URL 235
webmap ID
using 240, 243
well-known ID (WKID) 233
well-known text (WKT) 233