



Community Experience Distilled

ArcGIS for JavaScript Developers by Example

A practical guide to get you creating powerful mapping applications using the rich set of features provided by the ArcGIS JavaScript API

Jayakrishnan Vijayaraghavan
Yogesh Dhanapal

[PACKT] open source*
PUBLISHING community experience distilled

www.allitebooks.com

ArcGIS for JavaScript Developers by Example

A practical guide to get you creating powerful mapping applications using the rich set of features provided by the ArcGIS JavaScript API

Jayakrishnan Vijayaraghavan

Yogesh Dhanapal

[PACKT] open source 
PUBLISHING community experience distilled

BIRMINGHAM - MUMBAI

ArcGIS for JavaScript Developers by Example

Copyright © 2016 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: April 2016

Production reference: 1250416

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

ISBN 978-1-78588-866-3

www.packtpub.com

Credits

Authors

Jayakrishnan Vijayaraghavan
Yogesh Dhanapal

Reviewer

Dobrin Ganev

Commissioning Editor

Amarabha Banerjee

Acquisition Editor

Ruchita Bhansali

Content Development Editor

Sumeet Sawant

Technical Editor

Shivani Kiran Mistry

Copy Editor

Yesha Gangani

Project Coordinator

Shweta H Birwatkar

Proofreader

Safis Editing

Indexer

Rekha Nair

Production Coordinator

Melwyn Dsa

Cover Work

Melwyn Dsa

About the Authors

Jayakrishnan Vijayaraghavan is an Esri-certified web developer with extensive experience in full stack web development, machine learning, and GIS. Working in variegated domains and geographies and through graduate and undergraduate studies in computer science and GIS, he has gained a solid grounding in geospatial technologies and in data science. He is a DAAD scholar and a winner of the UN-HABITAT special jury award. He's keen on developing intelligent and ubiquitous mapping systems by integrating Machine Learning concepts with GIS. He is also a novelist and poet too.

Yogesh Dhanapal has expertise in developing and delivering end-to-end web mapping application for key clients, and he is proficient in many web technologies. He also has many years of training and education in the geospatial domain. A hardcore programmer and GIS enthusiast, Yogesh is a Microsoft-certified solutions developer – web applications and Esri-certified web developer. He has gained expertise in applying GIS for transportation and petroleum domain and has extensive experience in customizing Esri roads and highways extension with JavaScript dojo modules. He is keen on developing cross-platform and web applications with a mobile-first approach.

We thank Timmons group, our employer and Mr. Matt McCracken, our project manager for his immense support and encouragement in this endeavor. We're also greatly obliged to the Packt team, especially, Preethi and Shivani for their meticulous feedbacks and enthusiasm in bringing out this book.

About the Reviewer

Dobrin Ganev is a software developer with years of experience in various development environments from finance to business process management. In recent years, he has been focused in geospatial development and data analytics using languages such as JavaScript, Python, Scala, and R. He has extensive knowledge with open source geospatial and the Esri platforms. Currently, he is focused on big data, and its applications across broad industries and sectors.

chorStream Inc. (<http://www.chorstream.com/>) is a software development firm focused on the use of big data and big data technologies to help clients work with and leverage large and diverse volumes of data founded in 2015. Mr. Ganev, as a co-founder, has worked with an accomplished team of professionals to create and bring to market an application's framework similar to a Web AppBuilder that end users are able to use to build custom and focused applications without having to have any development skills.

I would like to thank Packt Publishing for seeking and asking me to be part of this exciting book. I hope that the body of knowledge stored in this book will be a great asset to those who want to learn how they can leverage programming for GIS. I would also like to extend a warm thank you to Guillermo Paniagua, a colleague of mine, for his input and insight on GIS topics in this book, which helped me review this book. The endeavor to continually learn cannot be a solo activity, and the best results always come from team work and peer review.

www.PacktPub.com

eBooks, discount offers, and more

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 customercare@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://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

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

I dedicate this book to Mr. Vijayaraghavan V R, my father who sowed the first seeds of writing and teaching in me.

Table of Contents

Preface	vii
Chapter 1: Foundation for the API	1
Setting up the development environment	1
Browser, web server, and IDE	2
Web browser	2
Web server	2
IDE	4
Setting up an ArcGIS Developer account	4
Hello, Map – the jump-start code	4
Understanding the jump-start code	6
The API reference link	6
The AMD pattern of coding	10
The esri/map module	11
Setting the initial map extent	11
Brushing up some coordinate geometry	12
Quiz time!	13
Spatial reference systems	14
The quiz results	14
Getting the current map extent	15
The template generator for loading modules	19
Understanding dojo and AMD	20
The key components of AMD	21
The define method	21
The require method	22
Some awesome dojo modules	22
Dojo dom modules	23
Dojo event handler module	23
Dojo array module	23

Understanding ArcGIS Server and the REST API	24
Types of service	25
Working with the Service Catalog	26
Map server	27
Summary	31
Chapter 2: Layers and Widgets	33
<hr/>	
Data sources supported by the API	33
Flat file formats	34
KML	34
CSV files	34
ArcGIS Server	34
The concept of layers	35
Adding layers to a map	35
The functional classification of layers	36
Basemap layers	36
Functional layers	37
Graphics layers	38
Types of layers	38
The ArcGIS Tiledmap service layer	38
Spatial Reference	40
The ArcGIS DynamicMapService layer	42
Feature layers	47
Graphics layer	52
Map and layer properties	54
Map and layer events	55
Using Esri widgets – the genie's lamp	56
The BaseMapGallery widget	56
The Legend widget	57
Summary	58
Chapter 3: Writing Queries	59
<hr/>	
Developing the Wildfire application	60
Registering the application in the developer portal	61
Using a proxy in the application	64
Bootstrapping the application	66
Types of querying operations	67
Query task	67
Find task	68
Identify task	68
Building and executing a Query task	68
The QueryTask constructor	68
Constructor parameters	69
Instantiating the QueryTask object	70

Building the Query object	70
Querying by spatial geometry	72
Executing the query	73
Querying for Count	73
Query for Features	76
Query for Extent	79
Building and executing IdentifyTask	80
Instantiating IdentifyTask	80
Constructing the identify parameters object	80
Executing IdentifyTask	81
Building and executing a Find task	82
Instantiating a Find task	83
Building the Find parameters	83
Executing a Find task	84
Building a feature table	85
Building popups	85
Building InfoTemplates	86
Summary	87
Chapter 4: Building Custom Widgets	89
Creating a simple class	89
Configuring dojo	90
Developing a standalone widget	93
The dijit life cycle	94
Creating templated widgets	95
Widget folder structure	97
Guidelines for creating project folders	98
Creating a single point of entry	98
Defining dojoConfig	99
Modularizing the code	99
Providing support for internationalization	100
An overview of the widget folder structure	103
Building a custom widget	104
Modules required for the widget	105
Modules for the class declaration and OOPS	105
Modules for using HTML templates	105
Module for using event	105
Modules for manipulating dom elements and their styles	105
Modules for using the draw toolbar and displaying graphics	106
Modules for querying data	106
Modules for internationalization support	106
Using the draw toolbar	106
Initiating the draw toolbar	107
The draw operation	109

The draw-end event handler	109
Symbolizing the drawn shape	109
Executing the query	112
Initializing the QueryTask and Query object	112
Query event handlers	113
Summary	118
Chapter 5: Working with Renderers	119
<hr/>	
Working with colors	119
The RGB color model	119
The Esri color module	120
Working with symbols	121
SimpleLineSymbol	122
SimpleMarkerSymbol	123
ArcGIS symbol playground	124
SimpleFillSymbol	126
PictureMarkerSymbol	126
PictureFillSymbol	129
TextSymbol	129
Working with renderers	130
Choosing a renderer for a scenario	131
Developing a Stream Gauge application	131
The data source	132
Simple renderer	132
Applying unique value renderer	134
Class breaks renderer	136
HeatmapRenderer	137
DotDensityRenderer	138
BlendRenderer	138
SmartMapping	140
A classification method for classed renderers	140
Summary	141
Chapter 6: Working with Real-Time Data	143
<hr/>	
Background about the application	143
Visualizing map data	144
Building a hurricane tracking app	148
Symbolizing active hurricane layers	149
Adding a global wind data gauge	154
Tracking the latest active hurricanes	156
Getting a unique list of storms	157
Fetching the latest data and displaying on the grid	159
Refreshing feature layer	160

Creating a weather widget	161
The open weather API	161
Using the Geolocation API	162
Using geometry engine on input data	163
Displaying the weather data in the widget	164
Summary	167
Chapter 7: Map Analytics and Visualization Techniques	169
Building a demographics analytic portal	169
Basic statistical measures	170
Minimum	171
Maximum	171
Sum	171
Average	171
Standard deviation	171
Standardization	172
Statistical functionality provided by the API	172
StatisticDefinition module	172
Classification methods	174
Equal interval	174
Natural breaks	174
Quantile	175
Standard deviation	175
Concept of normalization	176
Feature layer statistics	176
Working with continuous and break renderers	180
ColorInfo	182
Selecting a color scheme	182
Creating a classed color renderer	185
opacityInfo	189
Using opacityInfo to create a classes opacity renderer	189
SizeInfo	191
RotationInfo	191
Multivariate mapping	192
Smart mapping	196
Summary	199
Chapter 8: Advanced Map Visualization and Charting Libraries	201
Charting with dojo	201
Dojo chart themes	202
Charting using the popup template	204
Types of 2D charts provided by dojox modules	206
Dojo charting methods	206
Defining your plot	207

Table of Contents

Defining the theme	207
Pushing the data	208
Chart plugins	208
Charting with D3.js	213
Creating a column chart with D3	214
D3 selections	215
D3 data	216
D3 scaling	217
Integrating SVG into D3 charts	218
Charting with Cedar	224
Loading Cedar libraries	224
Loading using the script tags	225
Loading using the AMD pattern	225
Summary	233
Chapter 9: Visualization with Time Aware Layers	235
Time aware layers	235
Need for time aware layers	237
Understanding time aware layers	237
Building the Drought app	238
Using the Time Slider	238
Steps to create a TimeSlider	239
Querying based on time using D3	243
Scaling and formatting time	244
D3 brush	244
Advanced spatio-temporal visualization with Cedar	251
Summary	254
Index	257

Preface

Web technologies are changing rapidly and so is the ArcGIS JavaScript API. Regardless of your development experience, ArcGIS offers an easy way to create and manage geospatial applications. It gives you access to mapping and visualization, analysis, 3D, data management, and support for real time data.

What this book covers

Chapter 1, Foundation for the API, endeavors to lay a firm foundation for the topics dealt with throughout the book. The basic environment needed to follow the explained topics further as well as to develop professional-looking code is set in this chapter. An introduction to dojo and the modular pattern of JavaScript coding is provided along with an explanation of basic ArcGIS concepts. Users are shown brief explanations with code snippets or diagrams about basic concepts wherever needed.

Chapter 2, Layers and Widgets, deals with the different types of layers used in the API along with the ideal context where each type is used. We will also be introduced to some of the most commonly used in-built widgets provided by Esri to use in our application.

Chapter 3, Writing Queries, will have an in-depth look into writing different types of queries, retrieving the results and displaying it. We will be developing a Wildfire app to understand how the types of query operations such as Identify, Find and Query task. We will also learn how to display a tabular information using a FeatureTable widget and format popup content using Infotemplates.

Chapter 4, Building Custom Widgets, will explain how to organize all the code into modularized widgets, and use it in our application. We will discuss how to configure dojo globally and how to provide support for internationalization. We will be extending the Wildfire app we developed in the previous chapter by constructing a spatial query that involves using the Draw toolbar.

Chapter 5, Working with Renderers, gives an in-depth treatment on the topic of colors, symbols, renderers, and the situations where each can be used effectively. This chapter will also deal with the nuances of data visualization techniques along with tips and tricks to create symbols and picture marker symbols easily. We will demonstrate the utility of three basic renderers: simple renderer, unique value renderer, and class breaks renderer by developing a Stream Gauge app.

Chapter 6, Working with Real-Time Data, will cover in detail what constitutes the real-time data, and it will also cover how to visualize data and get the most recently updated data. We will be building a hurricane tracking app to demonstrate this and will be adding a Global wind data gauge and a weather widget using geometry engine capability provided by the API and the geolocation feature provided by modern browsers.

Chapter 7, Map Analytics and Visualization Techniques, will take you a step closer toward becoming a map data scientist. We will cover a lot of ground in this chapter starting with a brush up of a few introductory statistics concepts. We will see the code in action and understand how statistics definition and feature layer statistics module can give us invaluable statistic measures, which can be used to render the map data meaningfully. We will then evaluate how to use the visual variables, such as colorInfo, opacityInfo, rotationInfo, and sizeInfo effectively in a renderer. We will use the knowledge gained to start building a demographics analytic portal.

Chapter 8, Advanced Map Visualization and Charting Libraries, will be using three different charting libraries such as dojo, D3.js and Cedar to extend the Demographics portal we started building in the previous chapter and more provide visual-analytical information to the users.

Chapter 9, Visualization with Time Aware Layers, will explain how to visualize spatiotemporal data using TimeSlider dijit, a custom D3.js timeslider as well a custom Time series Histogram by incorporating these on a Time-aware US Drought data.

What you need for this book

For this book, we'll need NotePad++/Brackets Editor, Google Chrome/Mozilla Firefox or any modern browser, Visual Studio Community Edition 2015, and Node.js for Windows.

Who this book is for

This book is for JavaScript developers who wish to develop amazing mapping applications using the rich set of features provided by the ArcGIS JavaScript API, but more than this, a spatial frame of mind will help the user go a long way.

Conventions

In this book, you will find a number of text styles 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: "After installing IIS, you can find the executables in the `IIS Express` folder inside the `Program Files` folder "

A block of code is set as follows:

```
<link rel="stylesheet" href="http://js.arcgis.com/3.15/esri/css/esri.css">
```

```
<script src="http://js.arcgis.com/3.15/"></script>
```


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


```
on(map, "layers-add-result", function (evt) {  
  console.log("1.", earthquakeLayer.id);  
  ...  
  console.log("5.", worldCities.layerInfos);  
});
```

Any command-line input or output is written as follows:

1. **Earthquake Layer**
2. `[Object,`
`Object,`
`...`
`Object]`
3. `esriGeometryPoint`
4. `1000`
5. `[Object, Object, Object]`

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: "Click on the **Add** button against the IIS Express application name, and then click on the **Install** button."

[ Warnings or important notes appear in a box 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.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

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

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 http://www.packtpub.com/sites/default/files/downloads/ArcGISForJavaScriptDevelopersByExample_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

Foundation for the API

You are probably reading this book because you want to integrate spatial capability into your web application using the ArcGIS JavaScript API and make it even more amazing, or maybe you're hoping to become a web mapping data scientist very soon. Whatever it is, we are with you. But don't you think we need a bit of groundwork before working on the actual projects? This chapter is all about that—laying a strong foundation for the concepts used later in this book. This chapter is by design diverse in its content, covering a lot of ground on the following topics:

- Writing your first mapping application using the API
- Brushing up on coordinate geometry, extents, and spatial reference systems
- Introducing dojo and the AMD pattern of coding
- Understanding ArcGIS Server and the REST API
- Setting up the development environment

Setting up the development environment

This book is a *by example* book, and we will be explaining the concepts with the applications that we'll develop. So, it's essential that you have the development environment up and running at the onset of this chapter. Most of the environments mentioned in the following sections are just our preferences and may not be mandatory to implement the code samples provided in this book. All the code samples have been targeted at running in a Windows-based OS and an **Integrated Development Environment (IDE)** named **Brackets**. If you have a different choice of OS and IDE, we welcome you to develop in the environment you're most comfortable with.

Browser, web server, and IDE

To develop, deploy, and execute any web application, we need the following components:

- Web browser
- Web server
- Integrated Development Environment (IDE)

Web browser

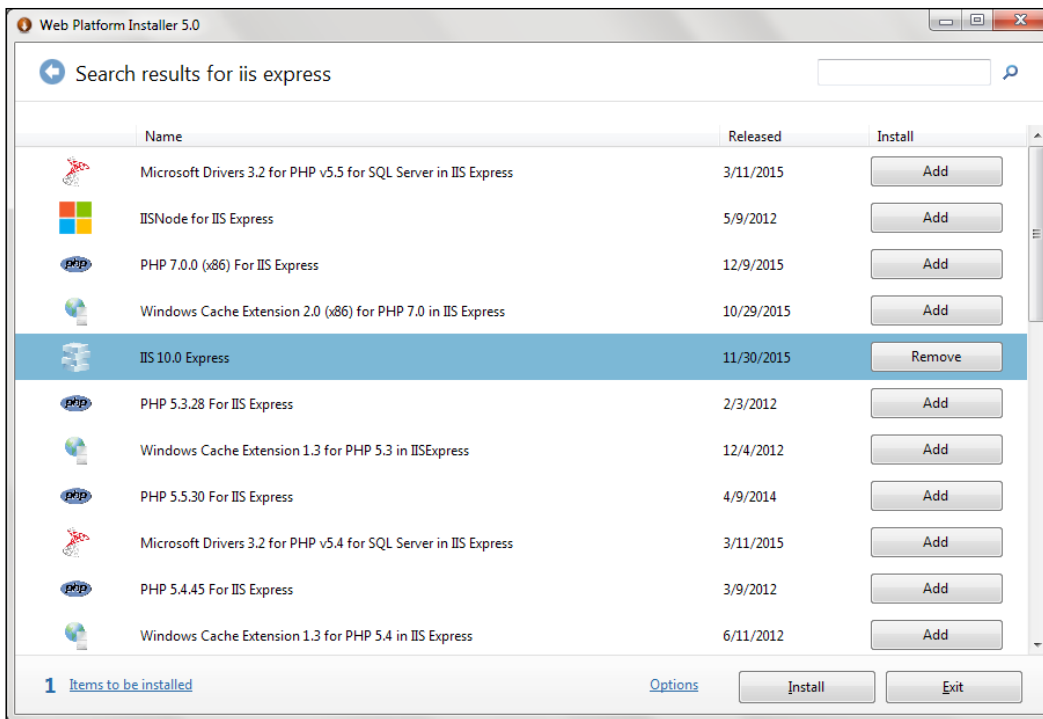
We have used Google Chrome throughout this book as it provides some great developer tools and HTML inspection tools. We think Mozilla too, is a great browser for development purposes.

Web server

Applications developed in this book are hosted using IIS Express. IIS Express is a lightweight webserver mainly used for hosting .NET web applications. Though, all the projects in this book are developed using pure HTML, CSS, and JavaScript, we will be using the Esri .NET resource proxy to access the ArcGIS online secured content and avoid cross domain issues.

Readers can install IIS Express either by installing Web Platform Installer or directly from the Microsoft downloads page, as illustrated in the following steps:

1. To install IIS Express using Web Platform Installer visit <https://www.microsoft.com/web/downloads/platform.aspx> to download Web Platform Installer.
2. Once downloaded, search for `IIS Express` in the search text. The search results will display the IIS Express application. Click on the **Add** button against the IIS Express application name, and then click on the **Install** button at the bottom of the page as shown in the following screenshot:



3. Installing IIS Express from Web Platform Installer ensures that we can get IIS Express' latest version against the direct download link, which we may not be able to provide the link for the latest version. When this book was written, the latest IIS Express direct download link was found at <https://www.microsoft.com/en-us/download/details.aspx?id=34679>.
4. After installing IIS, you can find the executables in the IIS Express folder inside the Program Files folder. The default location is usually `C:\Program Files\IIS Express`.
5. We will provide an executable batch (`.bat`) file within each project that helps to start the web server and host the project at the specified port.
6. You can find the following line of code in the executable file of each project we have developed for this book:


```
"C:\Program Files\IIS Express\iisexpress.exe" /path:<app location> /port:9098
```
7. The preceding line will host the application at port 9098. So, to access the app, you just need to use the URL—`http://localhost:9098/`.

IDE

The choice of IDE for developing JavaScript code is wide, and experienced developers already know what they need to use. We have used Brackets throughout this book as is our preferred choice of IDE.

Setting up an ArcGIS Developer account

For a few exercises in the book, you will require an ArcGIS Developer account. It's also an opportunity for you to explore various capabilities offered by ESRI for the developers. To set up a Developer account, just sign up for free at <https://developers.arcgis.com/en/sign-up/>.

Hello, Map – the jump-start code

If you're anything like us, you'd probably like to code your way to your first map right away. So here it is. Try adding these lines of code to a new HTML file in your Brackets IDE. You can also download the HTML source code, named B04959_01_CODE01, from the code repository and double-click on the HTML file to run it.

```
<!DOCTYPE html>
<html>

<head>
  <title>Hello, Map</title>
  <meta charset="utf-8" />
  <link rel="stylesheet" href="http://js.arcgis.com/3.15/esri/css/esri.css">
  <script src="http://js.arcgis.com/3.15/"></script>
  <script>
    var map;
    require(["esri/map"],
      function (Map) {
        map = new Map("mapDiv", {
          basemap: "national-geographic"
        });
      });
  </script>
</head>

<body>
  <div id="mapDiv" style="width:100%;height:600px;" />
</body>

</html>
```



While observing the preceding lines of code, you may have observed these two things:

- We didn't need any licensing, authentication, or key to run this code. In other words, the API is free. You just had to use the CDN link.
- We will be seeing this beautiful cartographic map in our browser as shown in the following screenshot:



- We encourage you to zoom or pan to location you want to see your map. If you haven't figured how to zoom/pan the map, we'll deal with it right away:
Left-click dragging or pressing any arrow key causes a pan and the level of detail doesn't change.
Shift + left-click drag, mouse scroll, a double click, or clicking on the + or - buttons on the map causes a zoom and the level of detail displayed changes.



There are other ways to achieve zooming/panning functionality. The ones mentioned here are just to gain a preliminary understanding.

Understanding the jump-start code

Let's try to understand the code we just saw. There are three concepts in this code that we'd like to explain. The first one deals with the reference links for the API or the **Content Delivery Network (CDN)** that we used to download the ArcGIS JavaScript API (v 3.15) and its associated style sheets. The second concept tries to introduce you to the pattern of coding employed, which is known as the **Asynchronous Modular Definition (AMD)** pattern. This is used by the latest version of dojo (v1.10). The next concept is about what you see in the browser when you ran the code – the map and the parameters we supplied to it.

The API reference link

First things first. We need to reference the API to develop an ArcGIS JavaScript API-based application. Esri is the organization that owns the API, yet the API is free and available for public use. The latest version of the API as of March 2016 was 3.15 and the corresponding dojo toolkit version was version 1.10.

The following libraries are the only ones you may probably need to reference to use ArcGIS JavaScript API's capabilities as well many dojo toolkit packages, such as `core`, `dojo`, `dijit`, `dgrid`, and so on:

```
<link rel="stylesheet" href="http://js.arcgis.com/3.15/esri/css/esri.css">
```

```
<script src="http://js.arcgis.com/3.15/"></script>
```

Refer to this link for complete documentation of the ArcGIS JavaScript API – <https://developers.arcgis.com/javascript/jsapi/>.

When you visit the preceding URL, you will see a web page providing complete documentation of the API with multiple tabs such as **API Reference**, **Guide**, **Sample Code**, **Forum**, and **Home**.

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:

- Log in or register to our website using your e-mail address and password.
- Hover the mouse pointer on the **SUPPORT** tab at the top.
- Click on **Code Downloads & Errata**.
- Enter the name of the book in the **Search** box.
- Select the book for which you're looking to download the code files.
- Choose from the drop-down menu where you purchased this book from.
- Click on **Code Download**.

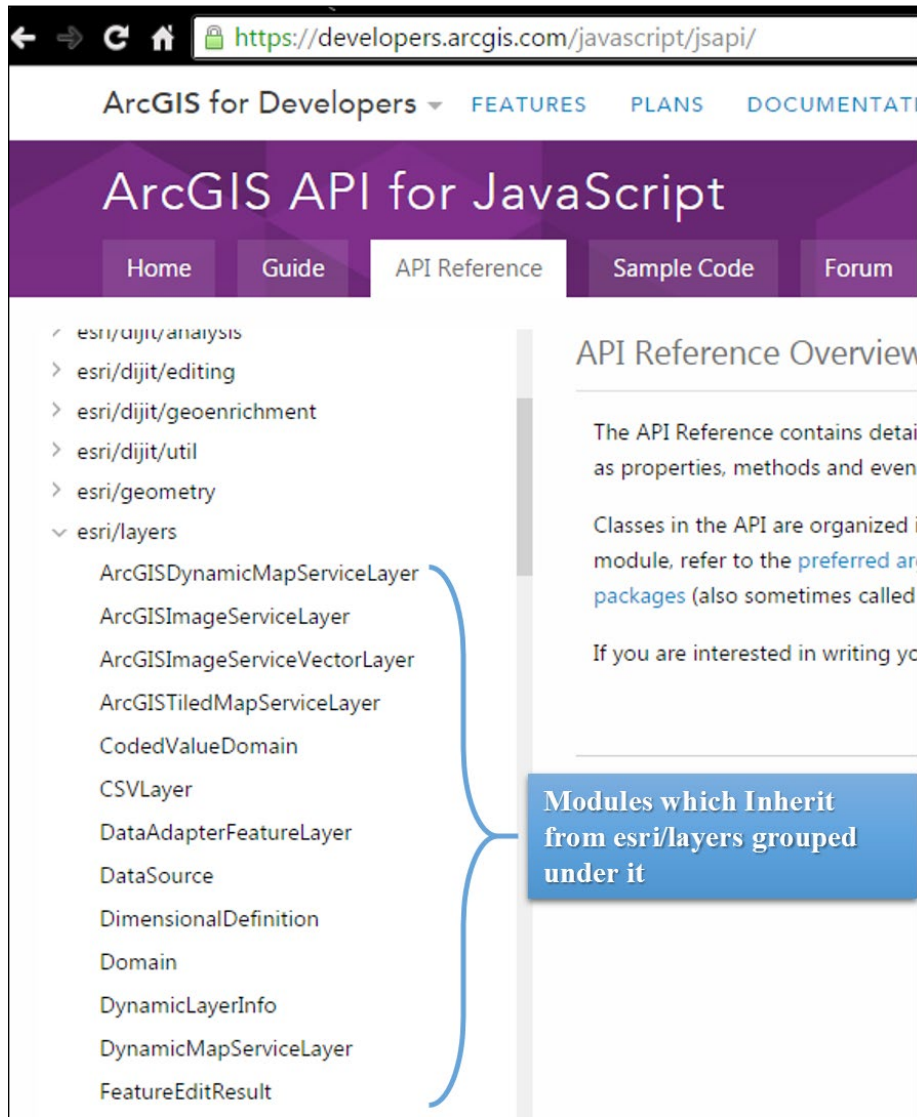


You can also download the code files by clicking on the Code Files button on the book's webpage at the Packt Publishing website. This page can be accessed by entering the book's name in the Search box. Please note that you need to be logged in to your Packt account.

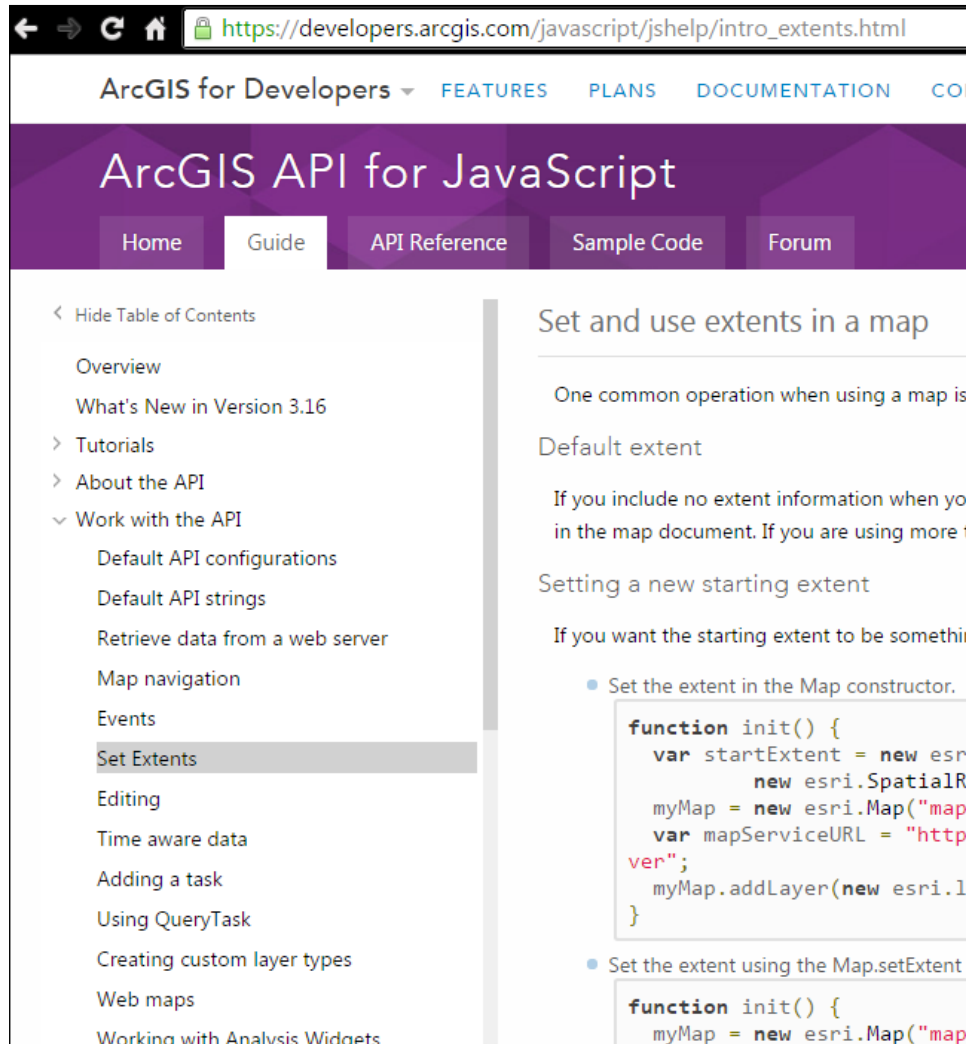
Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

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

The API reference lists all the modules available under the API as details, properties, methods, and events available for each module. The left pane groups most of the modules for easy reference. For example, the grouping named **esri/layers** has multiple modules that inherit from it. The following screenshot gives a snapshot of how the different modules that inherit from **esri/layers** are grouped:



The **Guide** section provides detailed instructions on important topics such as **Working with the Query Task**, **Working with ArcGIS Online Widgets**, and Working with symbols and renderers. The following screenshot shows a detailed guide on setting map extents:



The **Sample Code** tab is yet another useful section with hundreds of sample applications, which are used to demonstrate different concepts in the API. The best part of these sample codes is that they come with a sandbox facility, which you can use to play around with the code by modifying it.

The **Forum** tab redirects you to the following URL—<https://geonet.esri.com/community/developers/web-developers/arcgis-api-for-javascript>.

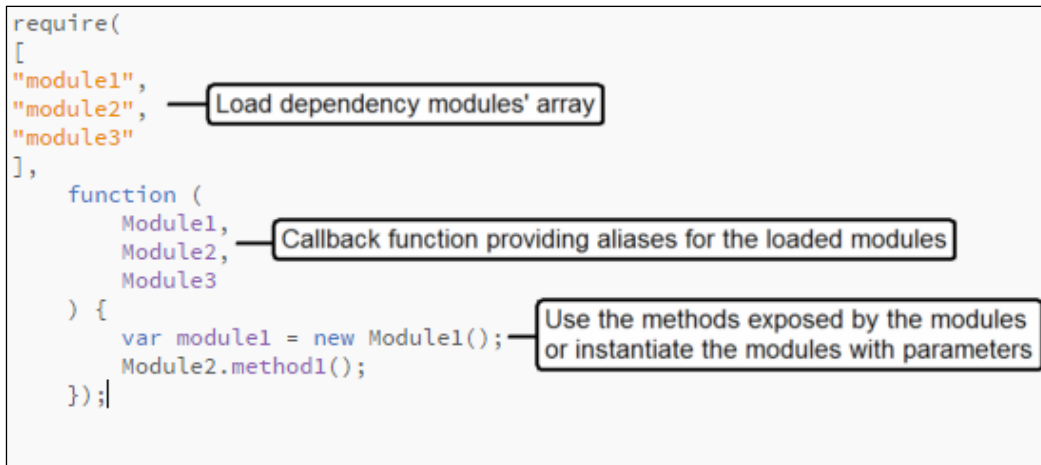
The GeoNet community forum is a great place to ask your questions and share your solutions to questions from developers like you.

Due to its close integration with the dojo framework, a working knowledge of the dojo toolkit is required and the reference documentation for it can be accessed at <http://dojotoolkit.org/reference-guide/1.10/>.

The AMD pattern of coding

If you've observed the code structure, it may look as follows:

```
require(
[
  "module1",
  "module2",
  "module3"
],
function (
  Module1,
  Module2,
  Module3
) {
  var module1 = new Module1();
  Module2.method1();
});
```



If you aren't familiar with this pattern of JavaScript coding, it is known as the AMD pattern of coding, and the ArcGIS API for JavaScript emphasizes on using this pattern of coding. In the initial chapters we will be introducing a lot about this to get ourselves familiarized with dojo and AMD. From the code structure you might have understood that the code *requires* certain modules and the function that loads these modules requires that they are in the same order. Some of the modules in our case were Esri modules (`esri/..`) and dojo modules (`dojo/..`). If you're wondering whether you could *require* custom defined modules, the answer is absolutely yes, and this will be a major part of our exercise in this book.

The esri/map module

The highlighted line in the code forms the crux of our jumpstart code:

```
var map = new Map("mapDiv", {  
    basemap: "national-geographic"  
});
```

The `map` module accepts two arguments. The first argument is the `div` container, which will contain the `map` object. The second argument is an optional object, which accepts a lot of properties that can be used to set the map's properties.

In our jumpstart code, the `basemap` property in the optional object sets one of the Esri's provided basemap codes named `national-geographic` to be displayed as the background map. We implore you to experiment with the other Esri provided basemaps, such as the following:

- `satellite`
- `dark-gray`
- `light-gray`
- `hybrid`
- `topo`

Setting the initial map extent

At times when the application opens up, you may want to zoom it to a particular area of interest, instead of showing the map at the world scale first, and then zoom your way to the area you want to see. To accomplish this, the `map` module provides a property to set its initial extent and also to programmatically change its extent any time you want.

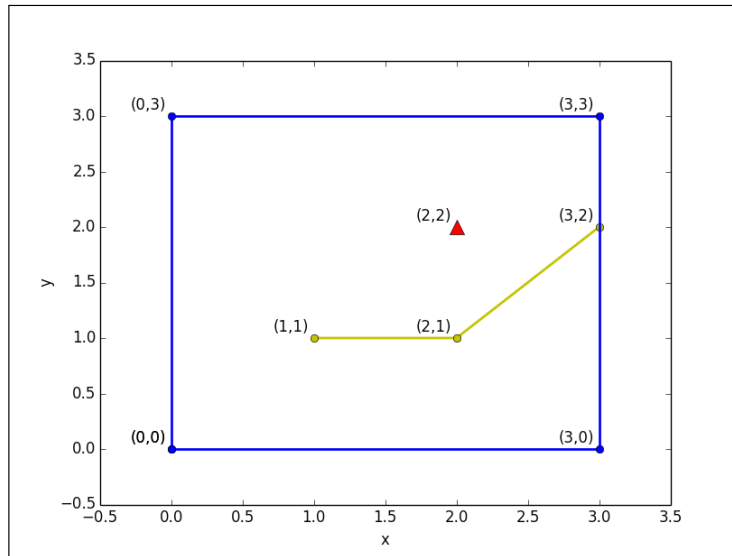
Before this, let's look at what an extent is in the context of a map.



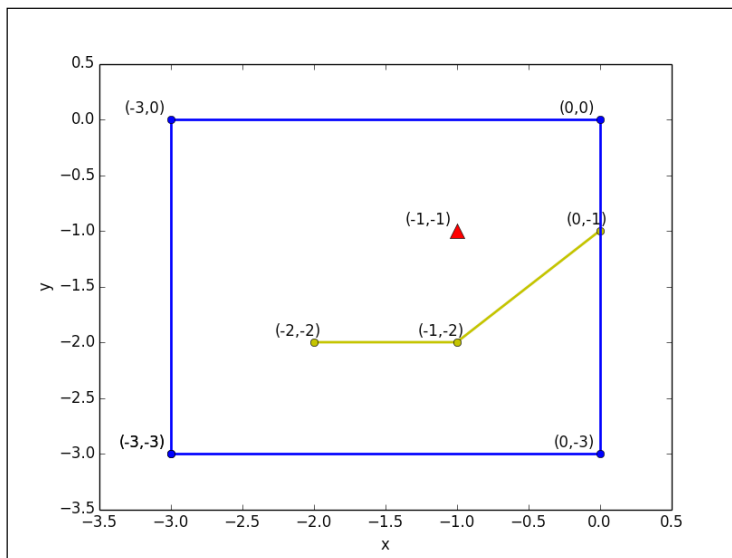
An extent is the minimum bounding rectangle that encloses an area of interest on the map.

Brushing up some coordinate geometry

To understand about extent, a grasp of coordinate geometry would help. Line segments colored yellow will be referred to as *polyline* for our purposes. The blue lines represents *polygons* (a rectangle in our case):



Now, try to observe the difference in the coordinates between the preceding diagram and the following diagram:



Here are some notes about the preceding diagrams:

- The point is represented by just one pair of coordinates; (2, 2) in figure 1 and (-1, -1) in figure 2
- The polylines are represented by a sequence of coordinates
- The polygon is also represented by a sequence of coordinates, similar to the polyline

You might have figured out that apart from the coordinates and the axes, the shapes of both the figures are the same. This might mean two things:

- The diagram has shifted its x positions by -3 units and its y positions by -3 units
- Or it may mean that the origin has shifted its x and y positions by -3 units

The second possibility is more important for us to understand because it implies that the actual position of the diagram hasn't changed and only the origin or the coordinate axes has changed its position. So, in reference to the coordinate axes, the coordinates of the diagram shapes (the rectangle, point, and line) have also changed.



The same shape can have different coordinates based on the reference coordinate system. This kind of coordinate system is known as a **spatial reference** in the context of GIS.

Quiz time!

Let's test our knowledge. Try solving the following quiz:

Q1. What would be the coordinates of the point (with the triangle symbol) if the origin (the bottom-left corner of the rectangle) were (100000, 100000)?


Q2. Since the polygon and the polyline are both represented by a sequence of coordinates, how can we conclude whether the shape is a polygon or polyline given a sequence of coordinates?

Q3. How many coordinates are required to represent a rectangle?

Think about it and we'll give you the answers very soon.

Spatial reference systems

When displaying the world or a part of the world on a digital screen as a map, which is a two-dimensional surface just like our graph, we need to use a spatial reference system to identify the coordinates of locations on the map. There are numerous standard spatial reference systems in use. The bare minimum we need to know to proceed with using the API is that each reference system has a unique identification number that is recognized by the API. The complete parameters (such as datum used, origin coordinates, measurement units used, and so on) used to define a spatial reference can also be used to identify a particular spatial reference system.

 The unique ID with which an SRS is identified is known as **Well-known ID (wkid)**.
A string listing the parameters used to define a spatial reference system is known as **Well-known Text (wkt)**.

As you might have anticipated, each spatial reference system is associated with different measurement systems such as feet, meters, or decimal degrees.

For example, 4326 is the wkid of the global coordinate system known as **WGS 84**. The measurement unit for this reference system is decimal degrees.

102100 is the wkid of another global coordinate system whose measurement unit is meters.

The following URLs give a list of wkids and the corresponding wkt at <https://developers.arcgis.com/javascript/jshelp/pcs.html> and <https://developers.arcgis.com/javascript/jshelp/gcs.html>.

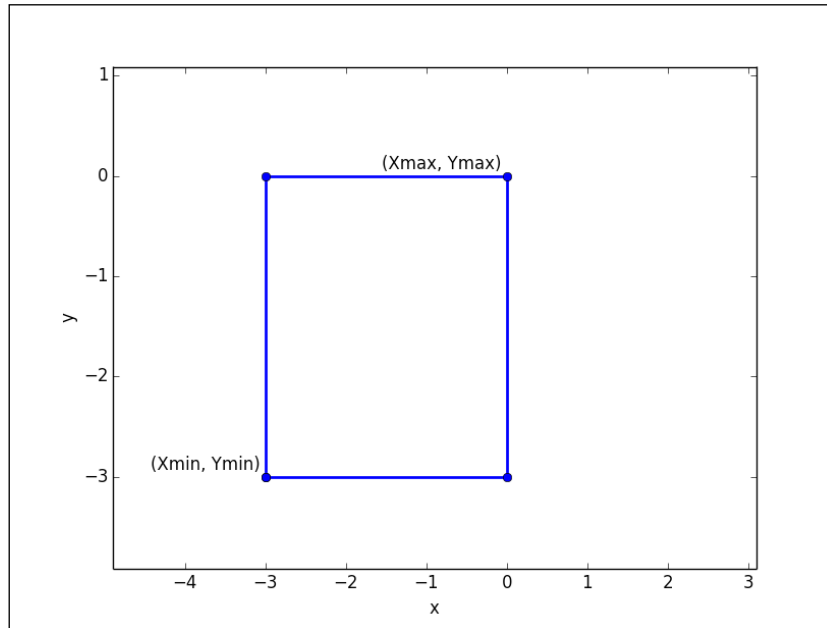
The quiz results

A 1. (100002, 100002) – relative to the origin, the point is 2 units away in the positive x-direction and 2 units away in the positive y-direction.

A 2. A sequence of coordinates can either be a polyline or a polygon unless mentioned explicitly in the geometry object. But a polygon has one property that makes it different from a polyline – the first and last coordinate must be the same. A polyline can have the same first and last coordinates, but not all polylines fulfil this criteria.

A 3. If your answer was 4, that's great! But if your answer was 2, you're awesome.

That's right. Just two coordinates are sufficient to define the rectangle, thanks to its perpendicularity property. The two coordinates could be any pair of diagonally opposite coordinates, but for the sake of the API, we will take the left-bottom coordinate and the upper-right coordinate. The bottom-left coordinate has the minimum x and y coordinate values among the 4 coordinate value pair, and the upper-right coordinate has the maximum x and y coordinate values:



Getting the current map extent

Zoom the map to the extent you want to set as the initial extent of the map. In the jump start code, the map variable is a global object since this is declared outside the require function:

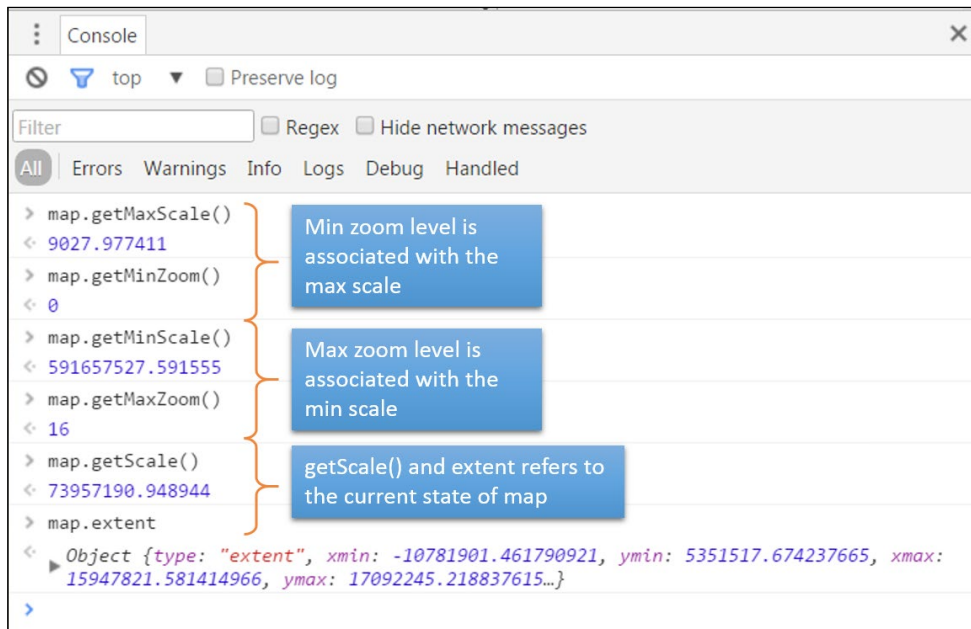
```
<script>
  var map; //Global variable
  require([
    "esri/map"
  ],
  function (
    Map
  ) {
    map = new Map("myMap", {
      basemap: "national-geographic"
    });
  });
</script>
```

```

    });
  });
</script>

```

This means that we can access the map's properties in the browser console. After zooming the map and the extent you want as the initial map extent, open the developer tools using the *Ctrl + Shift + I* command (in Chrome). In the JavaScript browser console, try to access the map properties, `getMaxScale()`, `getMinZoom()`, `getMinScale()`, `getMaxZoom()`, `getScale()`, and `extent`:



Scale is literally the factor with which the map measures are scaled down from the real-world measures. Maximum scale shows the greatest details on the map, and the minimum scale of the map shows the least detail. The values for `map.getMaxScale()` is smaller than that for `map.getMinScale()`, because the scale values represent reciprocal numbers. Hence $1/591657527 < 1/9027$ ($1/9027.977411$ and $1/591657527.59\dots$, respectively, in our instance).

Zoom levels, on the other hand, are the discrete scale levels at which the map is displayed. Most maps that involve `Basemaps` or `Tiledmaps` (which will be discussed in later chapters) can only be displayed at specific scale levels known as zoom levels. The minimum zoom level is mostly 0 and is associated with the maximum scale of the map.

`map.getScale()` gives us the current scale, and `map.extent` gives us the current extent of the map. We can use this `extent` object to set the extent of the map using the `setExtent()` method in the map. Refer to the API documentation for the `map` module and navigate to the `setExtent` method of the map. The `setExtent()` method accepts two parameters – the `Extent` object and an optional fit object. When we click on the hyperlinked `Extent` object, as provided in the document, it redirects us to the API documentation page for the `Extent` module:

The screenshot displays the Esri API documentation. The top section shows the `setExtent(extent, fit?)` method, which sets the extent of the map. It includes a description, a return type of `Deferred`, and parameters: `<Extent> extent` (Required) and `<Boolean> fit` (Optional). A sample code snippet is provided below.

The bottom section shows the `new Extent(json)` constructor, which creates a new `Extent` object using a JSON object. A blue callout box highlights the parameter `<Object> json` with the text "Extent can be passed as a json". The parameters section shows `<Object> json` (Required) as a "JSON object representing the geometry." A sample code snippet is also provided.

The left sidebar shows the navigation menu with the following items:

- esri
 - basemaps
 - Color
 - config
 - Credential
 - domUtils
 - Graphic
 - graphicsUtils
 - IdentityManager
 - IdentityManagerBase
 - ImageSpatialReference
 - InfoTemplate
 - InfoWindowBase
 - kernel
 - lang
 - Map

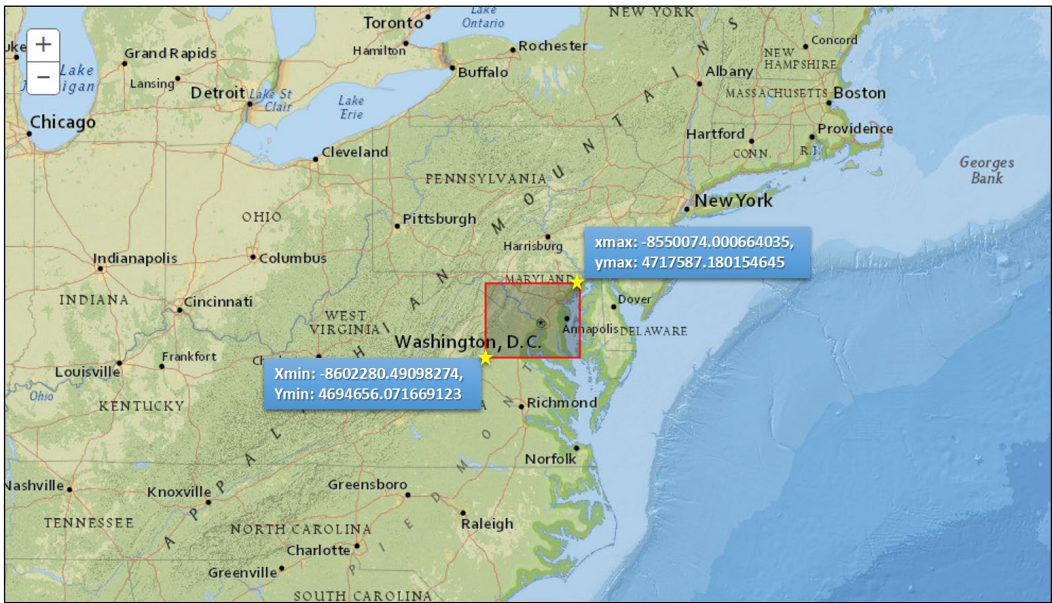
The bottom sidebar shows the "Table of Contents" with the following items:

- < Hide Table of Contents
- API Reference Overview
- Set up a development environment
- Preferred Argument Aliases
- > esri
 - > esri/arcgis
 - > esri/dijit
 - > esri/dijit/analysis
 - > esri/dijit/editing
 - > esri/dijit/geoenrichment
 - > esri/dijit/qtil
 - > esri/geometry
 - Circle
 - Extent

The constructor for `Extent` accepts a JSON object and converts it into an extent object. We can obtain this JSON object from the JSON string of the map's extent:

```
> map.extent |----> Map extent object
Object {type: "extent", xmin: -10691400.02030139, ymin: 4108957.3424341204, xmax:
-7350184.639900655, ymax: 5576548.285509114...}
  spatialReference: Object
    type: "extent"
    xmax: -7350184.639900655
    xmin: -10691400.02030139
    ymax: 5576548.285509114
    ymin: 4108957.3424341204
    __proto__: Object
> JSON.stringify(map.extent) |----> Map extent object as JSON
{"type":"extent","xmin":-10691400.02030139,"ymin":4108957.3424341204,"xmax":-73501
84.639900655,"ymax":5576548.285509114,"spatialReference":
{"wkid":102100,"latestWkid":3857}}"
```

The preceding image shows us the JSON string of the extent of the map that we have zoomed into. The following screenshot displays what the coordinates mean with respect to the map area we intend to zoom into (which is highlighted with the rectangle):



Now, we can copy the JSON object, create an `Extent` object, and assign it the `setExtent` method of the map. But before this, we need to import the `Extent` module (`esri/geometry/Extent`). The following screenshot explains how to implement this:

```
var map;
require([
  "esri/map",
  "esri/geometry/Extent"
]),
function (
  Map,
  Extent
) {
  map = new Map("myMap", {
    basemap: "national-geographic"
  });
  var extent = new Extent({
    "type": "extent",
    "xmin": -8602280.49098274,
    "ymin": 4694656.071669123,
    "xmax": -8550074.000664035,
    "ymax": 4717587.180154645,
    "spatialReference": {
      "wkid": 102100,
      "latestWkid": 3857
    }
  });
  map.setExtent(extent);
});
```

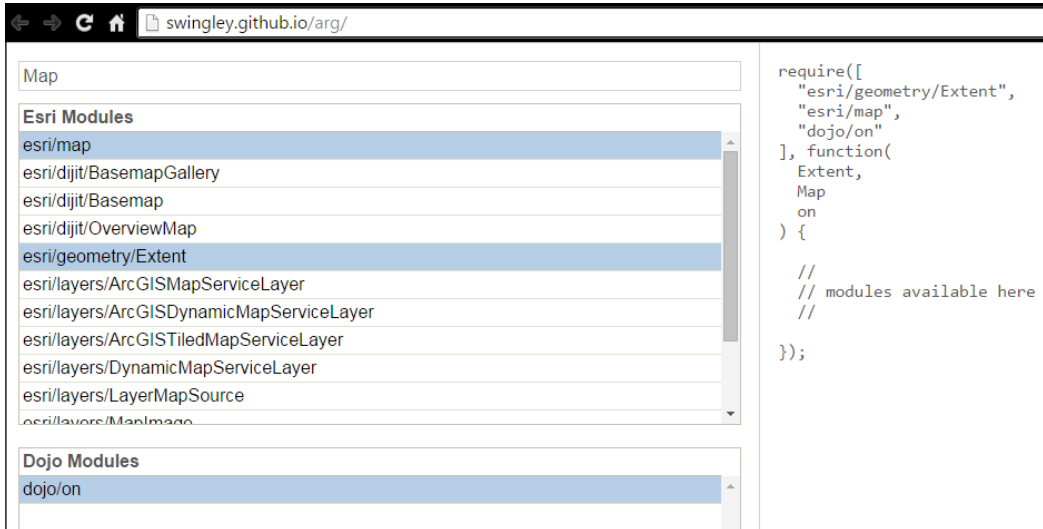


When we refresh the map now, the map will automatically zoom into the extent that we have set.

The template generator for loading modules

In the previous code where we successfully set the initial extent of the map, we had to use two modules: `esri/map` and `esri/geometry/Extent`. As the application grows, we may need to add many more modules to add additional functionality to the app. For a novice user, finding the module names from the API and incorporating them into the app might be cumbersome. This has been made easy using a web app template generator that can be found at <http://swingley.github.io/arg/>.

The following is a screenshot of the application:



The modules that are needed for our `require` function can be typed into the text box provided at the top of the application. There are two multi-selection list boxes: one listing the Esri modules and the other listing the dojo modules. Once we start typing the name of the modules required for our application, the list gets populated with the suggested modules matching the name we have typed. Once we select the module we need from either list box, it gets added to the list of modules in the `require` function, and an appropriate alias is added to the callback function as a parameter. Once all the required modules are selected, we can use the bare bone template being generated on the right side of the app. For setting the initial extent of the map, you may load the required modules by searching for the following names:

- Map (`esri/map`)
- Extent (`esri/geometry/Extent`)

Understanding dojo and AMD

As the name suggests, the AMD pattern of coding relies on modularizing your JavaScript code. There are lots of reason why you might need to start writing modular code or modules:

- Modules are written for a single purpose and are focused
- Modules are hence reusable
- Modules have a cleaner global scope

While there are many formats for writing modular JavaScript, such as CommonJS and ES Harmony, we will be dealing with AMD only because the latest versions of ArcGIS JavaScript API and the dojo toolkit upon which it is based uses the AMD style of coding. Dojo loader resolves the dependencies and loads the modules asynchronously while running the application.

The key components of AMD

In this section, we'll look at the `define` and `require` methods, which are the key components of AMD.

The `define` method

The `define` method defines a module. A module can have its own private variables and functions and only those variables and functions returned by the `define` function are exposed by other functions importing this module. An example for the `define` method is as follows:

```
define('DojoGreeting', [],
function () {
    var _dojoGreeting = 'Hello, from Dojo!';
    return _dojoGreeting;
});
```

Note the following in our code example:

- The first parameter in the `define` method is a module name or ID. This is optional. `dojoGreeting` is the name of our module.
- The second parameter is an array of dependencies for our module. We didn't need any dependencies for this module, so we just pass an empty array.
- The third parameter is a callback function that accepts any alias name for our dependencies that we might have loaded. Note that the alias names that are used as function parameters should be in the same order as it was defined in the dependency array. Since we didn't use any dependencies, we don't pass anything into this callback function.
- Inside the callback function, we can have as many privately scoped variables and functions as required. Any variable or function we'd like to expose from this module should be included in a `return` statement within the definition function.
- In our example, `_dojoGreeting` is a privately scoped variable that is returned by the `define` method.

The require method

The `require` method uses custom defined modules or modules defined in an external library. Let's use the module we just defined with a `require` method:

```
require([
  'dojoGreeting', //load the module 'myModule'
  'dojo/dom',
  'dojo/domReady!'
], function (dojoGreeting, dom) {
  dom.byId('greeting').innerHTML = dojoGreeting;
});
```

That's about it. Pay close attention to the parameters of the `require` method:

- The first parameter is an array of module dependencies. The first module dependency is the custom module we just defined, `dojoGreeting`.
- The `dojo/dom` module lets us interact with the `dom` elements in HTML.
- `dojo/domReady!` is an AMD plugin that will wait until the DOM has finished loading before returning. Note that the plugin uses a special character "!" at the end. We need not assign an alias in the callback function since its return is meaningless. Hence this should be one of the last modules to be used in the dependency array.
- The callback function uses `dojoGreeting` and `dom` as the alias for the `dojoGreeting` and `dojo/dom` modules respectively. As mentioned earlier, we need not use an alias for `dojo/domReady!`.
- The `byId()` method of the `dom` module returns a reference of a `dom` node by its ID. It's very much equivalent to `document.getElementById()`, only that the `dom.byId()` works across all browsers.
- In our `register` method, we are assuming we have a `div` element with its ID as `greeting`.

Some awesome dojo modules

You have already been introduced to two `dojo` modules, namely `dojo/dom` and `dojo/domReady`. Now, it's time to get familiarized with some other awesome `dojo` modules, which you should try using wherever possible while writing an ArcGIS JS API application. Sticking to using pure `dojo` and `Esri JS` modules will have enormous kickbacks in terms of code integrity and cross-browser uniformity. What's more? `Dojo` has some pleasant surprises for you in terms of the commonly used JavaScript functionalities, some of which we are going to introduce very shortly.

Dojo dom modules

You've already used the `dojo/dom` module. But there are other `dojo dom` modules, which will let you manipulate and work with the `dom` nodes:

- `dojo/dom-attr`: This is the go-to module for anything related to `dom` attributes:
 - The `has()` method in the module checks whether an attribute is present in a given node
 - The `get()` method returns the value of the requested attribute or null if that attribute does not have a specified or default value
 - As you might have guessed, there is a `set()` method that you can use to set values to an attribute
- `dojo/dom-class`: This module provides most of the operations you need to do with CSS classes associated with the `dom` nodes
- `dojo/dom-construct`: The `dojo/dom-construct` module lets you construct `dom` elements easily

Dojo event handler module

The `dojo/on` module is an event handler module that is supported by most browsers. The `dojo/on` module could handle events from most types of object.

Dojo array module

You should prefer `dojo`'s array module over the native JavaScript array functions for a variety of reasons. `Dojo`'s array module is named `dojo/_base/array`.

`dojo/_base/array`

As you would expect from an array module, there's an iterator method known as `forEach()` as well as the `indexOf()` and `lastIndexOf()` methods. Now comes the best part. There's a `filter()` method that returns an array filtered by a particular condition. We find the `map()` method a gem since it not only iterates through an array but also allows us to modify the items in the callback function and return the modified array. Ever wanted to check whether each or at least one element of the array met a particular condition? Check out the `every()` and `some()` methods in this module.

This sample code explains two main methods of the dojo array module:

```
<script>
  require(["dojo/_base/array"],
    function (array, dom, domConst, on, domStyle) {
      var arr = ["Mon", "Tues", "Wednes", "Thurs", "Fri", "Satur", "Sun"];
      var daysOfWeek = array.map(arr, function (item) {
        return item + "day";
      });
      var weekdays = array.filter(daysOfWeek, function (item) {
        return item[0] != "S";
      });
      array.forEach(daysOfWeek, function (day, idx) {
        console.log("Day #" + (idx + 1) + " is " + day);
      })
    });
</script>
```

The preceding code prints the following to the browser's console window:

```
Day #1 is Monday
Day #2 is Tuesday
Day #3 is Wednesday
Day #4 is Thursday
Day #5 is Friday
Day #6 is Saturday
Day #7 is Sunday
```

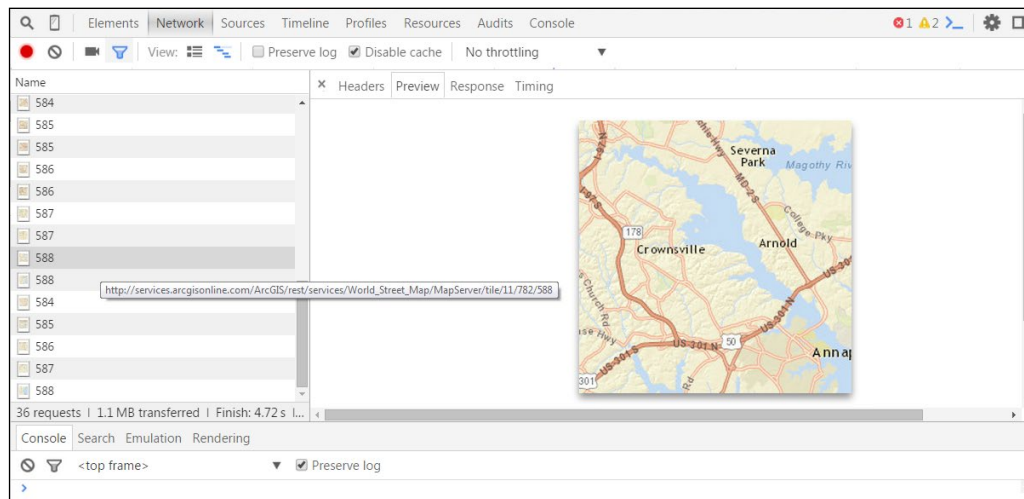
Understanding ArcGIS Server and the REST API

ArcGIS Server is an Esri product for enabling WebGIS by sharing geospatial data over the web. Our JavaScript API is capable of consuming a lot of services exposed by this server through the REST API. It simply means that all these services exposed by the ArcGIS Server is available through a URL. Now, let's look at how the REST API interface is so helpful to developers.

Types of service

When you ran the first code given in this book, you saw a cartographic map on the web page. The map you saw in the browser was actually a collection of images stitched together. You will realize this if you observed the **Networks** tab in the developer tools when you loaded the map. The individual images are called **tiles**. These tiles are also served by an ArcGIS MAP server. Here's a URL for one such tile: `http://server.arcgisonline.com/ArcGIS/rest/services/World_Street_Map/MapServer/tile/2/1/2`.

This just means that any resource published via ArcGIS Server and available to the API is through an URL, as shown in the following screenshot:



An ArcGIS service endpoint will have the following format: `<ArcGIS_Server_Name>/ArcGIS/rest/services/<Folder_Name>/<ServiceType>`.

ArcGIS Server provides a user interface to view these REST endpoints. This interface is popularly known as the **Service Catalog**.

The Service Catalog is something the developer needs to consult before planning to use a particular GIS service. The Service Catalog supports multiple formats such as JSON and HTML, HTML being the default format. If you're unable to view the Service Catalog, you need to contact your GIS Administrator to enable the service-browsing capability for the service you're interested in.

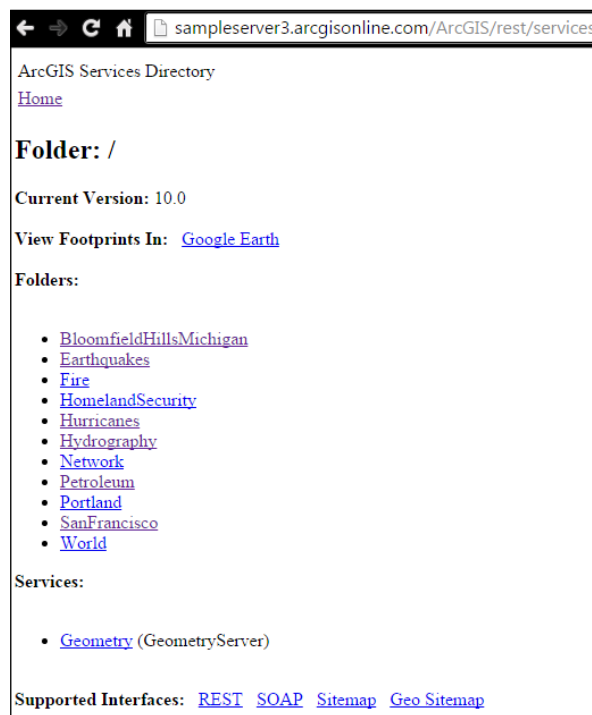
Working with the Service Catalog

Let's explore a sample GIS server provided by Esri named `sampleserver3.arcgisonline.com`.

To view the Service Catalog for any GIS server, the syntax is `<GIS Server Name>/ArcGIS/rest/services`.

So the URL we need to navigate to is: `http://sampleserver3.arcgisonline.com/ArcGIS/rest/services`.

You will see this screen in your browser:



The items of interest in the Service Catalog are the list of links below the **Folders** heading tag and the list of links under the **Services** heading tag. We encourage you to navigate to each of these links and see the kinds of service they expose. You will find the following types of service:

- **MapServer:** This serves geospatial data
- **FeatureServer:** This enables the editing functionality
- **ImageServer:** This serves image tiles

Didn't we mention that the Service Catalog supports multiple formats such as JSON? We encourage you to append a query string parameter, such as `?f=json`, at the end of the URL. To view the Service Catalog as an HTML, just remove the query string parameter from the URL.

Map server

A map server exposes GIS data as a REST endpoint.

Let's explore more about a particular map server named `Parcels` inside the folder `BloomfieldHillsMichigan`. Navigate to this URL: `http://sampleserver3.arcgisonline.com/ArcGIS/rest/services/BloomfieldHillsMichigan/Parcels/MapServer`.

The following heading tags are of particular interest to us: `layers`, `tables`, and `description`. Now, let's delve more into one of the layers in the map server. All three layers are worth navigating through. For the purpose of explanation, let's choose the first layer (Layer ID: 0), which can be navigated to directly using this URL: `http://sampleserver3.arcgisonline.com/ArcGIS/rest/services/BloomfieldHillsMichigan/Parcels/MapServer/0`.

All the heading tags listed in this URL are worth contemplating. We'll discuss some of these:

- **Geometry Type** describes the type of geometry of the particular layer. In our URL under investigation, it is named `'esriGeometryPoint'`, which means it is a point feature.
- Meta data such as `'Description'`, `'Copyright Text'`.
- Information about the Geographic Extent of the data under the tags `'Extent'` and `'Spatial Reference'`.
- The `Drawing Info` tag defines how the data is rendered on the map.
- `'Fields'` reveals the table schema of our layer. The actual field name is mentioned along with the type of the field and the alias name of the field. The alias and field type information is necessary to perform queries on the data. A field type of `'esriFieldTypeString'` and `'esriFieldTypeSmallInteger'` indicates that the field should be treated as a string and number respectively. `'esriFieldTypeOID'` is a special type of field that holds the unique Object ID of the features in the layer.

The Query endpoint

At the bottom of the page, there will be a heading tag named **Supported Operations** listing the links to the various endpoints exposed by this layer. There might a link with a text called **Query**. This link is the reason for our delving into ArcGIS Server and REST endpoints. Click on the link or navigate to it using this direct URL: <http://sampleserver3.arcgisonline.com/ArcGIS/rest/services/BloomfieldHillsMichigan/Parcels/MapServer/0/query>.

ArcGIS Services Directory
[Home](#) > [BloomfieldHillsMichigan](#) > [Parcels \(MapServer\)](#) > [Building Footprints](#)

Layer: Building Footprints (ID: 0)

Query Building Footprints:
PARCELID:

Filter Geometry:
Geometry Type:
Input Spatial Reference:
Spatial Relationship:
Relation:

Object Ids:
Where:
Time:

Result Options:
Return IDs only: True False
Return Geometry: True False
Max Allowable Offset:
Output Spatial Reference:
Return Fields (Comma Separated):
Format:

The UI provides us all possible ways that we can query with that particular layer (**Building Footprints**). The query operation seems to support both spatial as well as flat table SQL queries. As of now, let's just discuss the flat table queries. The **Where** field and the **Return Fields (Comma Separated)** are the ones that deal with flat table queries. The **Where** field accepts a standard SQL `where` clause as input, and the **Return Fields** accepts a comma-separated value of field names that needs to be the output. But we're explorers at this stage of development, and we just need to see the kind of data returned by this interface. Feed the following values into the corresponding textbox:

- **Where:** `1 = 1`
- **Return Fields:** `*`

Click on the **Query (GET)** button and scroll to the bottom of the screen.

The query literally returns all the layer data from all the fields from the database, but ArcGIS Server limits the results to 1000 features. Note that the browser URL has changed. The following URL is the REST GET request URL that was used to fire this query: `http://sampleserver3.arcgisonline.com/ArcGIS/rest/services/BloomfieldHillsMichigan/Parcels/MapServer/0/query?text=&geometry=&geometryType=esriGeometryPoint&inSR=&spatialRel=esriSpatialRelIntersects&relationParam=&objectIds=&where=1%3D1&time=&returnIdsOnly=false&returnGeometry=true&maxAllowableOffset=&outSR=&outFields=*&f=html`.

The following URL, removing all optional and undefined query parameters from the preceding URL, will also yield the same result: `http://sampleserver3.arcgisonline.com/ArcGIS/rest/services/BloomfieldHillsMichigan/Parcels/MapServer/0/query?where=1%3D1&outFields=*&f=html`.

Now let's analyze the result data a bit more by narrowing down our where clause. Note the **OBJECTID** field of the first feature among the results:

1. Remove the value in the where clause text box.
2. Enter the noted **OBJECTID** in the Object IDs text box. The object ID we noted was **5991** (but you could very well pick any).
3. There's a drop-down labeled format. Select the drop-down value named 'json'
4. Click on the **Query (GET)** button.

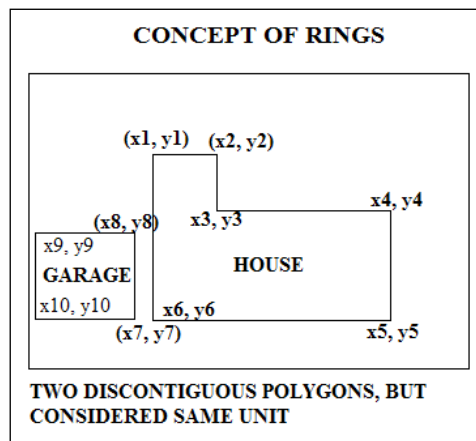
Or, here's the direct URL achieving the same operation: `http://sampleserver3.arcgisonline.com/ArcGIS/rest/services/BloomfieldHillsMichigan/Parcels/MapServer/0/query?objectIds=5991outFields=*&f=pjson`.

Now, the results look very detailed. What we are looking at is the data of a single feature. The JSON returns several features key value pairs with keys such as `displayFieldName`, `fieldAliases`, `geometryType`, `spatialReference`, `fields`, and `features`.

Let's look at the `feature` key value pair. The value for the `features` key is an array of objects. Each object has keys named `attributes` and `geometry`. The `attributes` hold the value of an object listing the key values of field names and its values. In our case, `PARCELID` is the fieldname and `"1916101009"` is its value:

```
▼ {
  "displayFieldName": "PARCELID",
  ▶ "fieldAliases": { ... }, // 1 item
  "geometryType": "esriGeometryPolygon",
  ▶ "spatialReference": { ... }, // 1 item
  ▶ "fields": [ ... ], // 1 item
  ▼ "features": [
    ▼ {
      ▼ "attributes": {
        "PARCELID": "1916101009"
      },
      ▼ "geometry": {
        ▼ "rings": [
          ▼ [
            13414403.970144361,
            397379.2099737525
          ],
          ▼ [
            13414379.790026248,
            397382.52001312375
          ],
          ▼ [
            13414381.669947505,
            397396.27985563874
          ],
        ]
      }
    }
  ]
}
```

The `geometry` object represents the polygon feature with an array of ring objects. And each ring is an array of floating point numbers. We earlier dealt with a polygon as just an array of coordinates. But ArcGIS Server treats a polygon as an array of rings. To understand the concept of rings, please look at the following illustration:



In the preceding illustration, we dealt with two disjoint polygons, but it is considered as a single unit in the real world, such as a house and garage. ArcGIS represents the polygon feature with two rings. The first rings consists of coordinates called $[[x1, y1], [x2, y2], \dots [x6, y6]]$, and the second ring consists of coordinates called $[[x7, y7], \dots [x10, y10]]$.

Summary

We used the ArcGIS JS API's CDN to access the API and tried to understand the map and Esri geometry modules. We tried to better understand extents and spatial references by brushing up our knowledge of coordinate geometry. We now know that an extent is just a minimum bounding rectangle that could be defined using two coordinates, and a spatial reference system is akin to the coordinate axes on a graph. We tried to look at some of the amazing modules that the dojo toolkit provides, which we must consider using in our code. ArcGIS Server exposes its GIS data and other resources as a REST API, that is, it is available as a URL. You also learned that a developer must always consult the Service Catalog before starting to consume any service through the API. we laid down our preferences in the way of development environment for working through projects in this book. The next chapter deals with the different types of layer used in the API and the ideal context where each type is used. We will also be introduced to some of the most commonly used in-built widgets provided by Esri, and we will use them in our application.

2

Layers and Widgets

The two basic components that make up our web mapping application are layers and widgets. A map object is similar to a canvas that holds all the layers, and users can interact with it, such as panning and zooming into the map. Layers are primarily associated with a particular data source. Widgets are composed of JavaScript logic and an HTML template (if it requires user interaction). Widgets can interact with the map or can function independently. Esri has developed a lot of general-purpose widgets, and these are bundled with the API. We will discuss how to use these widgets throughout this book. We will also see how to develop custom widgets in the next chapter. This chapter sets the starting point in the development of a full-fledged web mapping application displaying historical earthquake data. We will be gaining a strong foothold in the following topics as we progress through the chapter:

- Data sources supported by the API
- The concept of layers in the context of the API
- The functional classification of layers
- The different types of layers and their properties
- Featurelayers versus DynamicMapService versus graphics layer
- Using Esri's in-built widgets

Data sources supported by the API

The ArcGIS JavaScript API is a powerful and flexible client-side mapping software that provides support for integrating a variety of spatial data sources, which is currently in production. It also provides support for visualizing flat file formats, such as CSV, with some latitude and longitude information.

In order to leverage full capabilities provided by the ArcGIS JavaScript API, it is important to know the list of data sources it supports and the properties and methods it exposes.

The data sources supported by the ArcGIS JavaScript API as of version 3.14 can be broadly grouped as follows:

- ArcGIS Server services
- OGC compliant GIS services
- Flat file formats
- Custom web services (preferably REST services)

Let's review the different data source formats and understand how to get the necessary information about the data to consume in the ArcGIS JavaScript API.

Flat file formats

The API provides native support to render flat file formats such as KML and CSV.

KML

Keyhole Markup Language (KML) is a spatial file format that was initially developed by Google and is currently maintained by OGC. It provides support for point, line, and polygon geometry, and even image overlays. KML is an XML well known for this versatility, but it is pretty verbose and is used in Google Maps. KML files can be opened in any text edit such as Notepad++.

CSV files

The CSV file is a plain text file format that stores tabular data with field values separated by commas. CSV files contain information about latitude and longitude or coordinate values such as X and Y coordinates in separate fields. A CSV file can be read by the API, and the location information can be converted to point to the location on the API.

ArcGIS Server

ArcGIS Server can be used to share spatial data over the Web. In our case, if we have data as shape files, personal geodatabases, file geodatabases, or enterprise geodatabases, we can use ArcGIS Server to serve the data over the Web as REST services. ArcGIS JavaScript is capable of consuming these services and displaying them onto the map. In case of other spatial formats, such as DWG, we can either use the ArcGIS desktop or **Feature Manipulation Engine (FME)**, which is a spatial ETL tool for converting into the Esri file format and publishing it via ArcGIS Server.

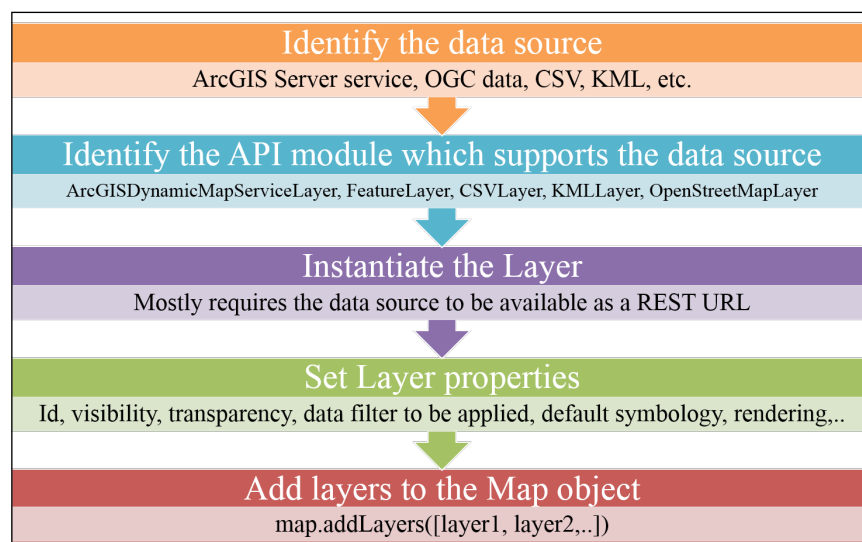
The concept of layers

If you had taken an introductory course in GIS, you'd be familiar with the classic image of GIS layers overlaid on each other. Layers, in the context of the API, are data resources available as REST endpoints or a JSON object. (That's right, you can construct a web map layer using a JSON string.) We will deal with the sources and types of these map layers very soon, but before that, let's list the most important considerations of any map layer:

- A layer is a container object for any data source
- Data can be added to the map object using the layer object
- Layers form a stack architecture – the first layer added is at the bottom
The norm is to have the *Basemap layer* at the bottom
- The map object has a special in-built layer to contain all map graphics
This is called the *graphics layer* and is at the top layer *always*
- All other functional layers are added in between
- The visibility of the layers can be turned on or off at any time

Adding layers to a map

Before dealing with the different types of layers, we will discuss how to add any layer to the map object, because the process is going to be the same for any layer type, and also it's very simple. In the following diagram, we can see all the types of layer:



There are two methods by which you can add any layer to the map object. Suppose `prjMap` is the name of the map object defined, and we need to add a layer; you can adopt one of these two methods:

- **Method 1:**

```
//Method 1
prjMap.addLayer(layer1);
/*layer1 is the layer object we would like to add to the map. */
```

- **Method 2:**

```
//Method 2
prjMap.addLayers([layer1]);
```

It's as simple as that! The second method is the preferred method, as there are certain widgets or functionalities that have to wait until all the layers in the map have been loaded. Using the second method will enable us to use an event handler that gets fired after all the layers are loaded. We will discuss about these event handlers toward the end of this chapter.

The functional classification of layers

Functionally, the different types of layer that could be added to a map can be classified as follows:

- Basemap or Tiledmap layers
- Functional layers
- Graphics layers

Let's discuss each of these independently.

Basemap layers

Basemap layers are layers that can be used as a reference background map. Usually, satellite imagery, topographical maps (maps showing elevation), or street maps serve this purpose. Basemaps are usually cached image tiles. This means that the Basemap is a static resource. Since they are static and are served as image tiles, we can't interact with (as in query or select) the features seen on the Basemap. And since this is the Basemap, this is the bottom-most layers as well as being the layer, that's added first to the map.

Now, the API provides different methods to add a basemap property to the map:

- Add the `basemap` property to the map object:

```
var map = new Map("mapDiv", {basemap: "streets"});
```
- Use the in-built basemap gallery provided by the API.
 This allows us to toggle between multiple basemaps, such as satellite imagery, Streets maps, Topographic maps, National Geographic maps, OpenStreetMaps, and so on.
- Create your own basemaps by adding Tiledmap layers to the map object (we'll discuss about Tiledmap layers very soon).

Download the project folder called `B04959_02_CODE_01` and open `index.html` to get a feel for the Basemap gallery widget:



Functional layers

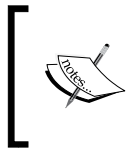
Functional layers display all the recent changes and are hence dynamic in nature as opposed to the relatively static nature of the Basemap or cached tile layers. Functional layers are layers you can interact with. The API provides options to perform different operations on most of these layers, such as:

- Selecting feature/s
- Retrieving the attributes and geometry of features
- Performing queries on the data
- Rendering features (applying styling to the features using different symbols, colors, width, and other graphic properties)
- Allowing create, update, and delete (CRUD) operations on features

Functional layers will be *reprojected-on-the-fly*, based on the spatial reference of the Basemap. This means that functional layers could be of a different spatial reference system than the Basemap and they'd still align with the Basemap, as the API will request the reprojected data of the functional layers from the server. There are different types of functional layer, such as dynamic layers and feature layers, which will be dealt with very soon.

Graphics layers

Graphics layers have the greatest versatility in terms of operations. Here, you can add as much data as you need to the attributes object. You can assign or modify its geometry (using the **Draw** toolbar or even programmatically), add symbology, query it (with functional layers, the query or update operations might be disabled), delete it, use it for selecting features from functional layers, or just use it as a redlining tool. But the graphics layer also has the shortest lifespan because it doesn't persist after a session—these are just stored on the client side. And due to these properties, it makes sense to have the graphics layer as the top-most layer, doesn't it?



A developer needs to be cautious about the spatial reference of the input data source when dealing with graphic layers. `esri/geometry/webMercatorUtils` is a handy module that lets us convert Web Mercator coordinates to geographic and vice versa.

Types of layers

We got a glimpse of the functional classification of layers. The API provides a host of modules to load layers from different data sources that generally fall into one of the functional classifications that we looked into. We are going to review some of the most important types of layers provided by the API and the methods and properties it exposes.

The ArcGIS Tiledmap service layer

This is the cached Tiledmap layer served by the ArcGIS Server:

Name	Value
Module Name	<code>esri/layers/ArcGISTiledMapServiceLayer</code>
Data Source Type	ArcGIS REST Service
Layer Type	BaseMap /Tiled Cache Layer
Response Type	Cached image tiles

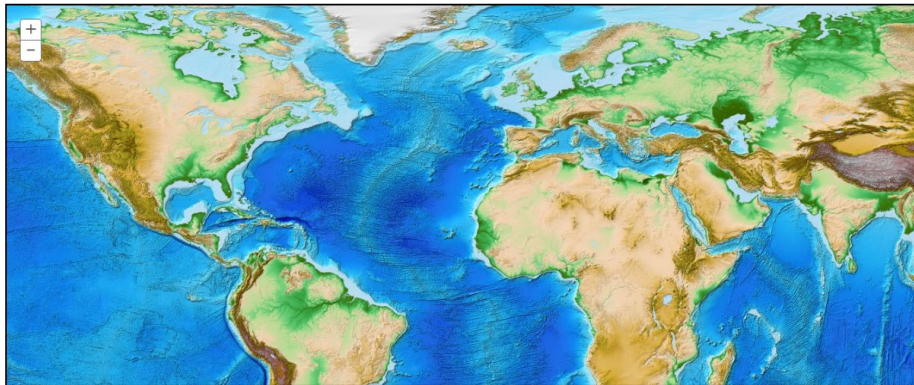
Name	Value
Constructor	<code>new ArcGISTiledMapServiceLayer(url, options?)</code>
Preferred Alias Name	<code>ArcGISTiledMapServiceLayer</code>



Preferred alias names

Preferred alias names provided by the API as part of the code convention and be accessed at https://developers.arcgis.com/javascript/jsapi/argument_aliases.html.

Why do we need to use a different Basemap when we already have a lot of options provided by Esri? Well, we found an aesthetic and visually informative tile map service from NOAA displaying a color shaded relief of the world's topography and bathymetry (ocean floor elevation differences):



You may consider using this as the Basemap for displaying any world-wide phenomena such as hazards or earthquakes. How do we do that? If you look at the constructor for this module, it looks for a required URL parameter and an optional options parameter.

The URL of the NOAA service we were talking about is `http://maps.ngdc.noaa.gov/arcgis/rest/services/etopo1/MapServer`.

Now, let's try to consume this as `ArcGISTiledMapLayer` (Code Reference: `B04959_02_CODE1.html`):

```
require([
  "esri/map",
  "esri/layers/ArcGISTiledMapServiceLayer",
```

```
"dojo/domReady!"
  ], function (
    Map, ArcGISTiledMapServiceLayer
  ) {
    var map = new Map("mapDiv");
    var tileMap = new ArcGISTiledMapServiceLayer("http://maps.ngdc.noaa.gov/arcgis/rest/services/etopo1/MapServer");
    map.addLayer(tileMap);
  });
```

That's all the code you need to write to see that beautiful map on your screen.

The service catalog of the Tiledmap service provides us with a lot of useful information that a developer should consider before using the Tiledmap service in the application. Let's consult the service catalog of the previously mentioned `ArcGISTiledMapServiceLayer`. In the screenshot of the service catalog provided in the next section, the developer can comprehend a lot of information about the data source:

- Spatial Reference
- TileInfo
- Initial Extent and FullExtent
- Min Scale and Max Scale
- Layers contributing to the tiles

Spatial Reference

Spatial Reference of the Tiledmap service or Basemap is one of the important properties overlooked by developers in the initial stages of coding. The **Spatial Reference** of the Tiledmap service is set as the spatial reference of the entire map. Operational layers, such as the dynamic map service and feature layers, added to the map that conforms to this is **Spatial Reference**, whatever their individual spatial reference is.

ArcGIS REST Services Directory [Login](#) | [Get Token](#)

[Home](#) > [services](#) > [etopo1 \(MapServer\)](#) [Help](#) | [API Reference](#)

[JSON](#) | [SOAP](#) | [WMS](#) | [WMTS](#)

etopo1 (MapServer)

Layers:

- [shaded_relief](#) (0)
- [dem](#) (1)

Spatial Reference: 4326 (4326)

Single Fused Map Cache: true

Tile Info:

- Height: 512
- Width: 512
- DPI: 96
- Levels of Detail: 16
 - Level ID: 0 [[Start Tile](#), [End Tile](#)]
 - Resolution: 0.351562499999999
 - Scale: 1.47748799285417E8
 - Level ID: 15 [[Start Tile](#), [End Tile](#)]
 - Resolution: 1.07288360595703E-5
 - Scale: 4508.93552506767
- Format: JPEG
- Compression Quality: 90.0
- Origin: X: -180.0
Y: 90.0
- Spatial Reference: 4326 (4326)

Initial Extent:

XMin: -136.19750976562503
YMin: 25.842480468749926
XMax: -43.00795898437499
YMax: 63.972949218749946
Spatial Reference: 4326

Full Extent:

XMin: -180.00833333333333
YMin: -90.00833333333328
XMax: 180.00833333333333
YMax: 90.00833333333334
Spatial Reference: 4326

Units: esriDecimalDegrees

Supported Image Format Types: PNG32,PNG24,PNG,JPG,DIB,T

Min Scale: 2.95828763795777E8

Max Scale: 4513.988705

Child Resources: [Info](#)

Supported Operations: [Export Map](#) [Identify](#) [Find](#) [Return U](#)

The spatial reference of the Basemap will persist for all the layers added on top of it

The section which differentiates a TiledMapService from a Dynamic Map Service

The minimum and maximum scale beyond which the tiles will not be visible

TileInfo

TileInfo provides information about the tiling scheme followed by `TiledMapService`. The **Level of Detail** can be used to set the zoom extent of the map.

Extent and Scale Info

Extent and scale info provides us information about the extent within which the tiles are visible.

Download the complete code from project folder `B04959_02_CODE_02` and see your beautiful Tiledmap in action.

The ArcGIS DynamicMapService layer

This module, as the name suggests, is a dynamically hosted resource from the ArcGIS Server REST API:

Names	Values
Module Name	<code>esri/layers/ArcGISDynamicMapServiceLayer</code>
Data Source Type	ArcGIS REST Service
Layer Type	Functional Layer
Response Type	Dynamically generated images
Constructor	<code>new ArcGISDynamicMapServiceLayer(url, options?)</code>

The dynamic map layer actually represents all the data exposed by the non-cached map service. For the same reason, dynamic map layers are a kind of composite layer because a map service generally has more than one layer.

We'll see what this means in a moment. We'll refer to the service catalog (yeah, it's a fancy term for the interface that appears when we navigate to the map service URL).

Open this URL of a map service in the browser — <http://maps.ngdc.noaa.gov/arcgis/rest/services/SampleWorldCities/MapServer>.

You will be able to see all the data layers exposed by the map service. So, when you consume this map service, all the data will be displayed on the map as part of a single `DynamicMapService` layer:

Map Name: World Cities Population

[Legend](#)

[All Layers and Tables](#)

Layers:

- [Cities](#) (0)
- [Continent](#) (1)
- [World](#) (2)

Description:



If you cannot see the service catalog for any service shown previously, it doesn't mean that the service is offline; it might be that service browsing is turned off on the production machine.

Make sure to try the URL by appending a query parameter named `f` with a value as `json`, for example, `{{url}}?f=json`.


Earlier, we discussed how to add `ArcGISTiledMapServiceLayer` to the map. The following code adds the `ArcGISDynamicMapService` layer upon the existing tiled layer:

```
require(["esri/map",
"esri/layers/ArcGISTiledMapServiceLayer",
"esri/layers/ArcGISDynamicMapServiceLayer",
"dojo/domReady!"
],
function (
Map,
ArcGISTiledMapServiceLayer,
ArcGISDynamicMapServiceLayer
) {
var map = new Map("mapDiv");
varshadedTiledLayer = new ArcGISTiledMapServiceLayer('http://maps.
ngdc.noaa.gov/arcgis/rest/services/web_mercator/etopo1_hillshade/
MapServer');

varworldCities = new ArcGISDynamicMapServiceLayer("http://maps.ngdc.
noaa.gov/arcgis/rest/services/SampleWorldCities/MapServer");

map.addLayers([shadedTiledLayer, worldCities]);
});
```

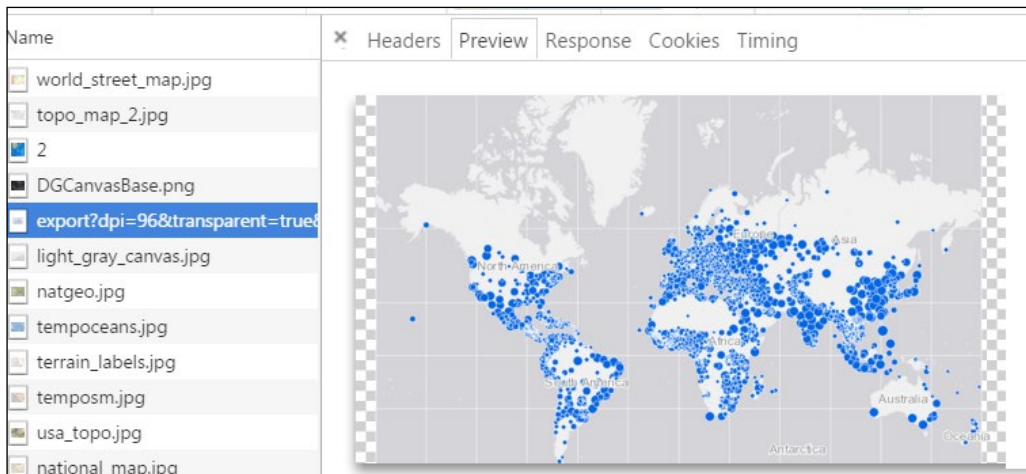
Now, if you would have observed, both `ArcGISDynamicMapServiceLayer` and `ArcGISTiledMapServiceLayer` consume a map service. So, how do we actually know which map service should be used as a Tiledmap service or which can be used as a DynamicMapService? You might have guessed it right. The service catalog is the answer. There is a particular heading in the service catalog that you have to look for in the map services that distinguishes a cached tile map service from the non-cached map service. This is called **TileInfo**.

 The attribute that differentiates cached tile map services from non-cached map services is called Tile Info.

TileInfo has information on the level of detail information. The levels of details determine the discrete scale levels at which the map will be displayed. These levels of details are also known as the zoom levels, and the markers in the zoom control of the map correspond with these zoom levels.

Now, there is a similarity with how Tiledmap service and DynamicMapService responses are served. Both are served as images. While Tiledmap services serve multiple image tiles for each extent, a DynamicMapService serves just one image for a given extent.


If you notice your **Networks** tab, there will be a `GET` request method named `export` appended to the DynamicMapService that we declared. This is the `GET` request that fetched the dynamic map image from the server:



Observe the name-value pairs in the query string of the preceding GET request. You'll notice the following fieldnames:

- The `dpi` fieldname defines the resolution of the image in dots per inch
- The `transparent` fieldname defines that the response image is transparent, and so, the background Basemap can be viewed
- The `format` fieldname has a value of `png`, which is the format of the response image
- The value for the `bbox` fieldname requests the extent (consisting of four coordinates — `Xmin`, `Ymin`, `Xmax`, and `Ymax`) for which the image is requested
- The value for the `bboxSR` fieldname defines the spatial reference in which the `bbox` coordinates were defined, and `imageSR` defines the spatial reference in which the response image is requested
- The value last fieldname called `f` defines the format of the response; it's an image of course

[



Exercise

Change the value for the `f` field name from `image` to `html` in the preceding GET request and see what you get.

]

If you check out the API page, you will see that this module provides a lot of properties and methods. The following table shows some of the most important methods:

Method Name	Description
<code>exportMapImage(imageParameters?, callback_function?)</code>	This exports a map using values as specified by the <code>imageParameters</code> object. The callback function event returns the map image.
<code>refresh()</code>	This refreshes the map by making a new request to the server.
<code>setDPI(dotsPerInch)</code>	This enables setting the image resolution in dots per inch for the exported map.
<code>setLayerDefinitions(stringArray of Layerdefinitions)</code>	This enables us to filter the data displayed by the <code>DynamicMapService</code> .
<code>setVisibleLayers(Array_of_LayerIds)</code>	This makes visible only the layers whose IDs are passed in as the parameter.

Now, make sure you have the following requirements to display the DynamicMapService:

- Only display the `Cities` layer
- Provide a transparency of 0.5 for the dynamic map image
- Display only cities with a population greater than 1 million

The following snippet guides you in how to accomplish this (Code Reference: B04959_02_CODE2.html):

```
var worldCities = new ArcGISDynamicMapServiceLayer("http://maps.ngdc.noaa.gov/arcgis/rest/services/SampleWorldCities/MapServer", {
  "id": "worldCities",
  "opacity": 0.5,
  "showAttribution": false
});
worldCities.setVisibleLayers([0]);
worldCities.setLayerDefinitions(["POP > 1000000"]);
```



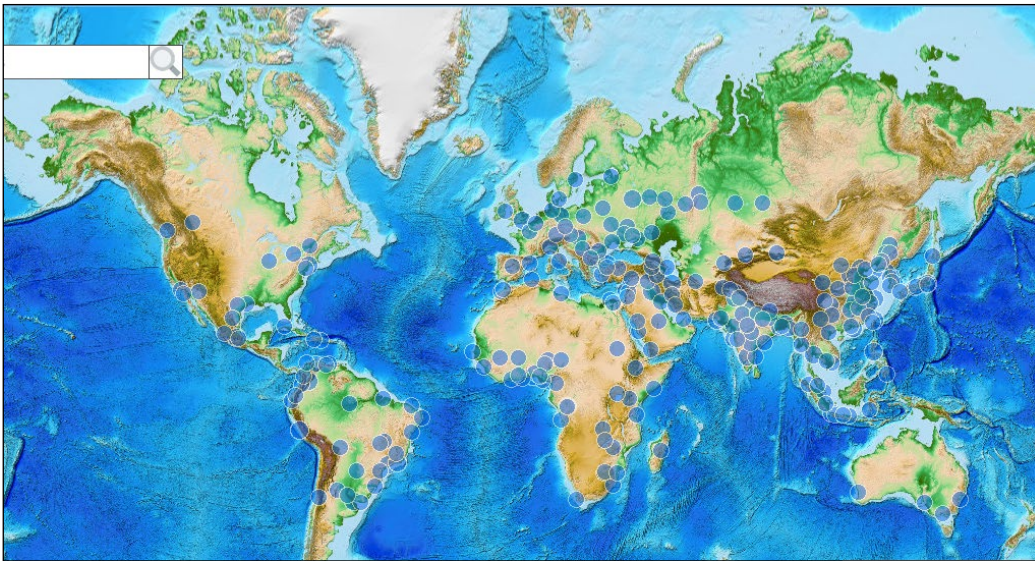
When the `showAttribution` property in the options object of any map object is set to `true`, all the attributions for the data source are shown in the bottom-right corner of the map.

The `setLayerDefinitions()` method accepts a string array of where clauses. While passing the layer definitions for a DynamicMapService, keep the following things in mind.

The index of the definition expression (where clause) should match the index of the layer on which the expression is applied. For example, if the `Cities` layer had an index of 5 in the preceding map service, the layer definition would look like this:

```
var layerDefinition = [];
layerDefinition[5] = "POP > 1000000";
worldCities.setLayerDefinitions(layerDefinition);
```

Once these conditions are met, the resulting map will look like this. The semi-transparent blue dots are the world cities with a population greater than one million:



Feature layers

A feature layer is an individual layer of a map service that has a geometry type. An individual layer in a map service could be a feature layer or even a raster layer; for example, http://sampleserver4.arcgisonline.com/ArcGIS/rest/services/Elevation/ESRI_Elevation_World/MapServer/1 and http://maps.ngdc.noaa.gov/arcgis/rest/services/web_mercator/hazards/MapServer/0 are both individual layers of a map service, but the former URL is a raster layer resource and the latter is a feature layer resource. The raster layer doesn't have a geometry type attribute in Service Catalog, whereas the feature layer has one of the geometry types point, multipoint, polyline, or polygon.

A feature layer is a very versatile entity because it supports advanced querying, selecting, rendering, and sometimes even editing facilities. A feature layer (or a raster layer) is identified using the index in the map service it belongs to:

Names	Values
Module Name	esri/layers/FeatureLayer
Data Source Type	ArcGIS REST Service
Layer Type	Functional Layer
Response Type	Feature Collection (Feature has geometry, attribute and symbology)
Constructor	<code>new FeatureLayer(url, options?)</code>

Adding the feature layer/s to the map is the same as adding a `DynamicMapService` layer or a `Tiledmap` service layer:

```
define([
  "esri/map",
  "esri/layers/FeatureLayer"
],
function(
  Map,
  FeatureLayer
){
  var map = new Map("mapDiv");
  var featureLayer1 = new FeatureLayer(featureLayer1URL);
  var featureLayer2 = new FeatureLayer(featureLayer2URL);
  map.addLayers([featureLayer1, featureLayer2]);
});
```

The `FeatureLayer` constructor

The `FeatureLayer` constructor has two arguments – the `FeatureLayer` URL and an optional `options` object. The `options` object provides a bunch of options to configure the `FeatureLayer` constructor. One of the most important `options` property is named `mode`.

The `mode` property defines how the feature layer is rendered on the map. Since feature layers stream the actual geometry of the feature, unlike the map service (which provides a dynamically generated image) or a `Tiledmap` service (which just serves pre-rendered cached tiles), the rendering of feature layers on a map has some performance considerations. There are four types of `mode` by which a feature layer can be rendered. The four modes are numeric values provided as constants by the API. If the callback function alias of the feature layer module is a feature layer, the four modes can be accessed using the following decorations:

- `FeatureLayer.MODE_SNAPSHOT`
 - This fetches all the features from the server once and resides on the client – a one-time overhead
 - This is updated when the additional filters are applied
- `FeatureLayer.MODE_ONDEMAND`
 - Features are fetched as needed
 - Continuous little chunks of overhead
 - Default `MODE`

- `FeatureLayer.MODE_SELECTION`
 - Only features selected using `selectFeatures()` method is displayed
- `FeatureLayer.MODE_AUTO`
 - This switches between `MODE_SNAPSHOT` or `MODE_ONDEMAND` (this choice is made by the API)
 - Best of both worlds

We will try to add a `FeatureLayer` constructor for historical earthquakes to the map. The map service providing these feature layers can be found at http://maps.ngdc.noaa.gov/arcgis/rest/services/web_mercator/hazards/MapServer.

The earthquakes layer is the fifth layer in the map service. But you can try other feature layers too. Here's a code snippet that lets you add a feature layer to the map object (Code Reference: `B04959_02_CODE3.html`):

```
define(["esri/map",
    "esri/layers/FeatureLayer",
    "dojo/domReady!"],
function (Map, FeatureLayer
) {
    varearthQuakeLayerURL = 'http://maps.ngdc.noaa.gov/arcgis/rest/
services/web_mercator/hazards/MapServer/5';
    earthQuakeLayer = new FeatureLayer(earthQuakeLayerURL, {
    id: "Earthquake Layer",
    outFields : ["EQ_MAGNITUDE", "INTENSITY", "COUNTRY", "LOCATION_NAME",
    "DAMAGE_DESCRIPTION", "DATE_STRING" ],
    opacity: 0.5,
    mode: FeatureLayer.MODE_ONDEMAND,
    definitionExpression: "EQ_MAGNITUDE > 6",
    });

    map.addLayers([earthQuakeLayer]);
});
```

The preceding code can be explained as follows:

- The `id` property assigns an ID to the feature layer
- The `opacity` property lets us define an opacity for the map
- The `definitionExpression` property is a where clause that lets us filter the features shown on the map
- An `outFields` property lets us define the fields provided by the feature layer

Here's a screenshot of the `FeatureLayer` superimposed over the `DynamicMapService` layer and the `Tiledmap` service layer. The semi-transparent colored circles represent the locations where any earthquake ever happened, which had a magnitude of more than 6 Richter scale:



When you pan the map or zoom around the map, the features are fetched and a corresponding `GET` request is fired, which fetches the features *on demand*. If you open the **Networks** tab in the developer console just after loading a feature layer, you will be able to understand a lot of things:

- The API uses the `query` method of the feature layer to fetch the features.
- In the query string, there will be query parameters, such as `geometry`, `spatialRel`, `geometryType`, and `inSR` which define the extent for which features need to be fetched. Other `FeatureLayer` constructor options, such as `outFields` and the `where` clause (corresponding to `definitionExpression`), can also be found in the query string.
- If you click on the **Preview** or **Response** tab, you will notice that the `GET` request fetches an array of features. Each feature has an attributes object and a geometry object. The attributes object will contain the field names mentioned in the `outFields` array and the corresponding field value of the particular feature:

The screenshot shows a REST client interface with a request and response. The request URL is `http://maps.ngdc.noaa.gov/arcgis/rest/services/web_mercator/hazards/MapServer/5/query?f=json&where=EQ_MAGNITUDE%20%3E%206&returnGeometry=true&spatialRel=esriSpatialRelIntersects&geometry=576822xmin%22%3A-10018754.11396947%2C%22ymin%22%3A10018754.171396947%2C%22xmax%22%3A-0.000004988163709640503%2C%22ymax%22%3A20037508.342788905%2C%22spatialReference%22%3A%7B%22wkid%22%3A102100%2C%22latestWkid%22%3A3857%7D%26geometryType=esriGeometryEnvelope&inSR=102100&outFields=OBJECTID%2C%2CEQ_MAGNITUDE_RANK%2C%2CDEATHS_AMOUNT_ORDER&outSR=102100`. The response is a JSON object with a `features` array. Annotations 1-5 highlight key parts: 1. The query URL, 2. The `MODE_ONDEMAND` parameter in the query string, 3. The `features` array in the response, 4. The first feature object, and 5. The `outfields` property in the feature object.

1. The query URL

2. The `MODE_ONDEMAND` parameter in the query string

3. All features matching the definitionExpression

4. The first feature object

5. Fields mentioned in the outfield property

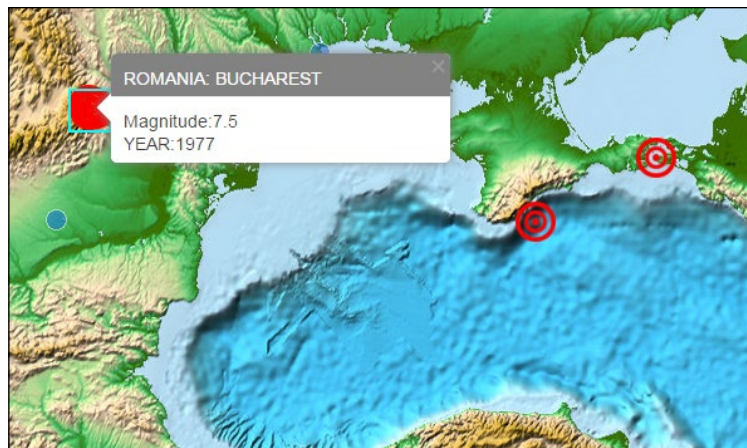
We'll deal with how to query and select the features in a feature layer in the next chapter. As of now, we are better off knowing what the following method does to the feature layer object:

Method	Description
<code>clear()</code>	Clears all graphics
<code>clearSelection()</code>	Clears the current selection
<code>getSelectedFeatures()</code>	Gets the currently selected features
<code>hide()</code>	Sets the visibility of the layer to <code>false</code>
<code>isEditable()</code>	Returns true if the <code>FeatureLayer</code> is editable

Method	Description
<code>setInfoTemplate (infoTemplate)</code>	Specify or change the <code>info</code> template for a layer
<code>setOpacity (opacity)</code>	Initial opacity of the layer (where 1 is opaque, 0 is transparent)
<code>show ()</code>	Sets the visibility of the layer to <code>true</code>

Infotemplates

Infotemplates provide a simple way to deliver an HTML popup displaying the information about a feature when we on click it. We will discuss Infotemplates in detail in the next chapter.



Graphics layer

We've already discussed about graphics layer a bit. We know that the map object, by default, contains a graphics layer, and it can be referenced using the `graphics` property of the map object. We can also create our own graphics layers and add them to the map. However, the default graphics layer provided by the map remains at the top.

Let's understand more about the graphics layer and the `Graphic` object that is added to the graphics layer. The graphics layer is a container for the `Graphic` objects.

A `Graphic` object has the following values:

- Geometry
- Symbol
- Attributes
- Infotemplate

Geometry

Geometry will have a type (point, multipoint, polyline, polygon, and extent), a spatial reference, and the coordinates making up the geometry.

Symbol

A symbol is a much more complex object because it is associated with the geometry it symbolizes. Also, the styling of the symbol is defined by the colors or picture used to fill up the symbol and the size of the symbol.

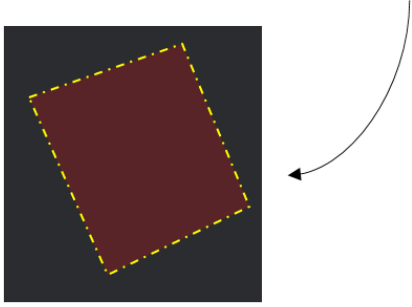
Let's review a snippet to understand this better. This is a simple snippet to construct a symbol for a polygon:

```

1  var tbDrawSymbol =
2      new SimpleFillSymbol(
3          SimpleFillSymbol.STYLE_SOLID,
4      new SimpleLineSymbol(
5          SimpleLineSymbol.STYLE_DASHDOT,
6          new Color([255, 255, 0]),
7          2
8      ),
9      new Color([255, 0, 0, 0.2])
10 );|

```

Symbol type used to style a polygon geometry
 The Fill type constant
 Symbol type for the outline of the polygon
 The Line type constant for the outline
 Color (yellow) and width of the outline
 Red Color fill for the polygon along with transparency



Attributes

The attributes of a graphic is a key-value pair object that stores information about the graphic.

InfoTemplate

`InfoTemplate` is the HTML template that can be used to display relevant information about a graphic when we click on it.

Map and layer properties

There are many common properties between the layers that give us relevant information about the layer. For example, properties such as `fullExtent`, `id`, `infoTemplates`, `initialExtent`, `layerInfos`, `maxRecordCount`, `maxScale`, `minScale`, `opacity`, `spatialReference`, `units`, `url`, and `visibleLayers` are the same for dynamic map layer as well Tiledmap layer, whereas properties such as `dynamicLayerInfos` and `layerDefinitions` are specific to the `DynamicMapService` layer. So, is the `tileInfo` property specific to Tiledmap layer?

Try to explore these properties by logging the properties to the console. For example, if you need to print a list of fields in a feature layer, use the `fields` property of the feature layer.

Here's a code snippet that logs certain information regarding the feature layer and the `DynamicMapService` layer to the console (code reference: `B04959_02_CODE5.html`):

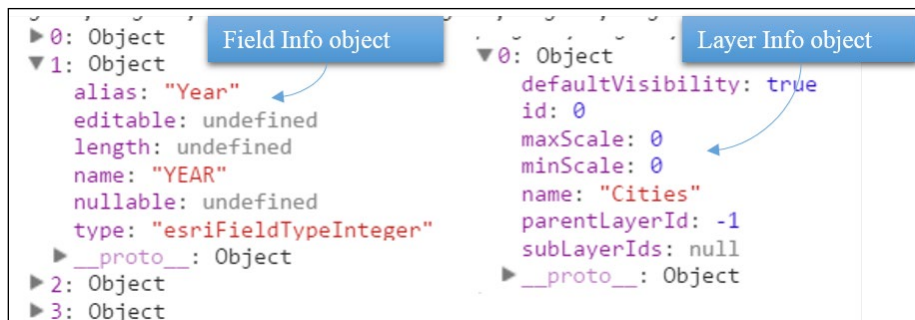
```
on(map, "layers-add-result", function (evt) {
  console.log("1.", earthquakeLayer.id);
  console.log("2.", earthquakeLayer.fields);
  console.log("3.", earthquakeLayer.geometryType);
  console.log("4.", earthquakeLayer.maxRecordCount);

  console.log("5.", worldCities.layerInfos);
});
```

The following is the screen output you will get in the console:

```
1. Earthquake Layer
2. [Object,
Object,
. . .
Object]
3. esriGeometryPoint
4. 1000
5. [Object, Object, Object]
```

`Featurelayer.fields` returns an array of field objects. Each object contains properties such as `alias`, `length`, `name`, `nullable`, and `type`. `DynamicLayer.layerInfos` returns an array of the `layerInfo` object. The `layerInfo` object provides information about the layer:



Map and layer events

Changing the extent of the map, adding a layer to a map, adding a group of layers to the map, or even clicking on the map or a mouse—the API has an event handler for all of it. While using the event, let's stick to dojo's on module to handle events. Find the prototype of handling events using dojo's "dojo/on" module:

```

1 ▼ require(["dojo/on"], function(on){
2   on(target, "event", function(e){
3     // handle the event
4   });
5 });
6 |

```

Target	Event	Description
Map	extent-change	Fires when the extent of the map has changed
Map	layers-add-result	Fires whenever you use the map.addLayers() method, after all the layers being added to the map are loaded
Map	load	This one is obvious
Map	basemap-change	
Feature layer	selection-complete	After selecting features from a feature layer

In the preceding code snippet, which logged out certain layer properties, you might have noticed that the entire code snippet was encompassed in an on statement:

```

on(map, "layers-add-result", function (evt) {
  console.log("1.", earthquakeLayer.id);
  ...
  console.log("5.", worldCities.layerInfos);
});

```

We needed to print out all the layer-related properties inside the `on` event since we need to wait until all the layers are loaded, or we will get an undefined for most of the properties. This particular event named `layers-add-result` is fired only after all the layer arrays added to the map is loaded.

Using Esri widgets – the genie's lamp

Widgets are the cornerstone of dojo. Widgets are UI components that can be built, configured, and extended in dojo to do a specific task. So, when someone provides us with a widget that accomplishes a task we need to do, all we have to do to instantiate it is configure it a bit and provide it with the container node reference where the widget should reside.

So, the good news is Esri provides us with in-built widgets that accomplish a lot of things, such as querying features, geocoding addresses (converting a text address into a location on a map), adding a widget to display the map legend, adding widgets to search for attributes, and even adding a widget to toggle between multiple basemaps. All the Esri built widgets can be located under `esri/dijits` in the table of contents section of the API reference page.

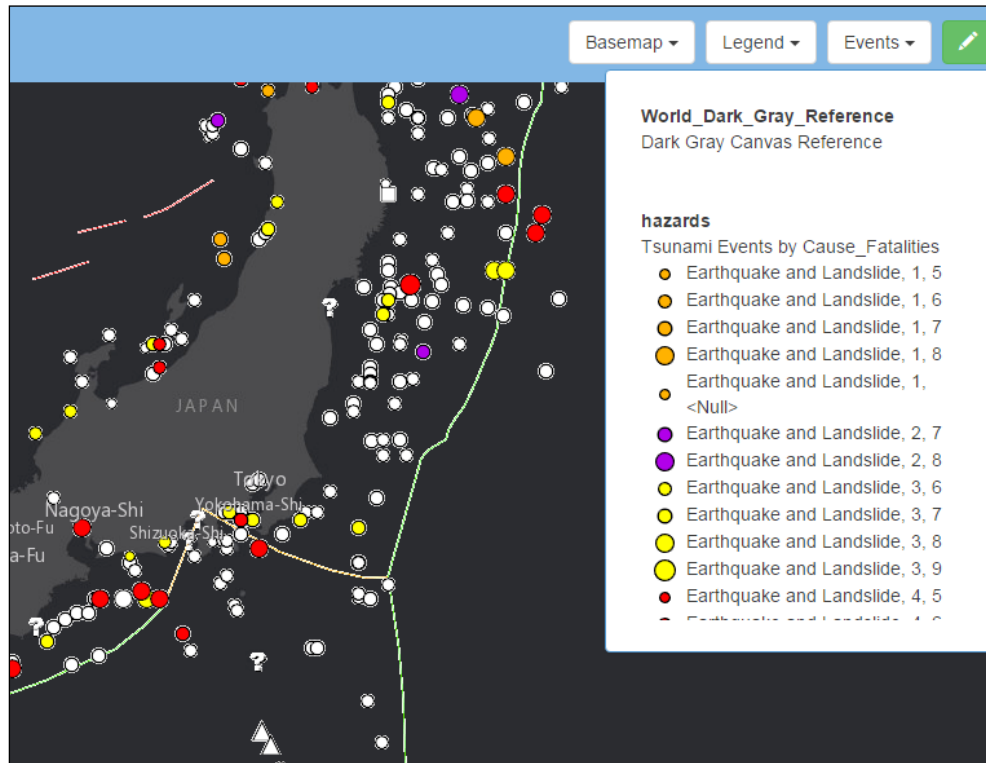
The BaseMapGallery widget

Well, you're not surprised that this widget does exist, right? We gave you a heads-up at the beginning of this chapter when we were dealing with `TiledMapLayers`. The `Basemap` layer widget provides us a with widget with which we can toggle a `Basemap` from a gallery of basemaps. See the following prototype code for integrating basemaps into our application (code reference: `B04959_02_CODE6`):

```
require(["esri/map",
  "esri/dijit/BasemapGallery"], function (Map,
  BasemapGallery) {
  var basemapGallery = new BasemapGallery({
    showArcGISBasemaps: true,
    map: map
  }, "basemapGalleryDiv");
});
```

The Legend widget

A map legend lists the layers in the map and the symbology used by all the layers. Constructing a legend by ourselves involves getting the `layerInfos` and `drawingInfos` and listing them in a `div` – the process sounds like a pain in the neck. Luckily for us, Esri provides us `dijit` (probably a portmanteau for `dojo` and `widget`) for constructing legends:



We use the following code to initiate the **Legend** widget (Code Reference: B04959_02_CODE6)

```
require(["esri/map",
"esri/dijit/Legend"], function (Map, Legend){
    on(map, "layers-add-result", function (evt) {
        var legendDijit = new Legend({
            map: map,
            }, "legendDiv");
        legendDijit.startup();
    });
});
```


Summary

We covered a lot of ground in this chapter. We tried to identify the process by which data is added to the map. We identified the data source, such as ArcGIS Server service, OGC data, CSV, KML, and so on. Then, we covered the API provided modules that support the display of, and further operations on, three major ArcGIS REST service data sources, namely the ArcGIS Tiledmap service layer, the ArcGIS DynamicMapService layer, and the feature layer. You also learned how to instantiate the layers and how to navigate their properties and events. We also dealt with a special kind of layer namely graphics layer, which is the top-most layer in the map and is used as a container object for all the graphics in the map. We got a taste of the plethora of in-built widgets provided by Esri. In the next chapter, we will have an in-depth look into writing spatial queries and retrieving the results. You will also learn how to use geometry services and the geometry engine to process geometric operations.

3

Writing Queries

"The art and science of asking questions is the source of all knowledge."

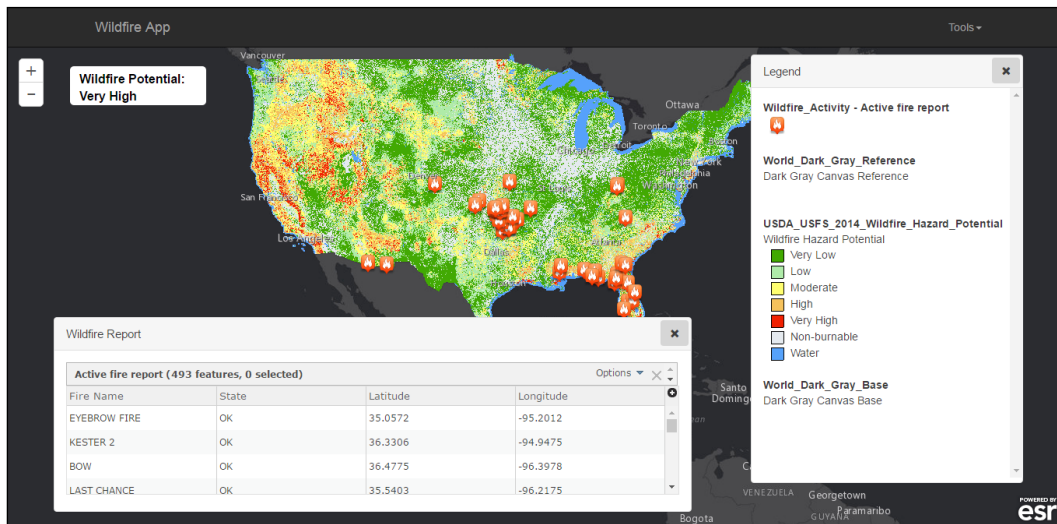
– Thomas Berger

Queries are the gateways to asking questions to the map through the API. They are considered as a *task* in the API terminology because the process of forming queries and getting the answers is a sequence of operations that must be carried out properly. In this chapter, we will be developing a Wildfire Location app to understand the following concepts:

- Building and executing the Query task
- Building and executing the Identify task
- Building and executing the Find task
- Promises, deferred, and the result objects for the Query, Find, and Identify tasks
- Using `FeatureTable.digit`
- Using `InfoTemplates`

Developing the Wildfire application

In this chapter, we will be developing an app that will display Active Wildfire Locations in the United States with a background map showing the Wildfire Potential for any location. We will also try to provide search/query functionalities by harnessing the components provided by the API. The following screenshot provides a rough rendition of our final application that we will have developed by the end of this chapter:



The application will have the following components:

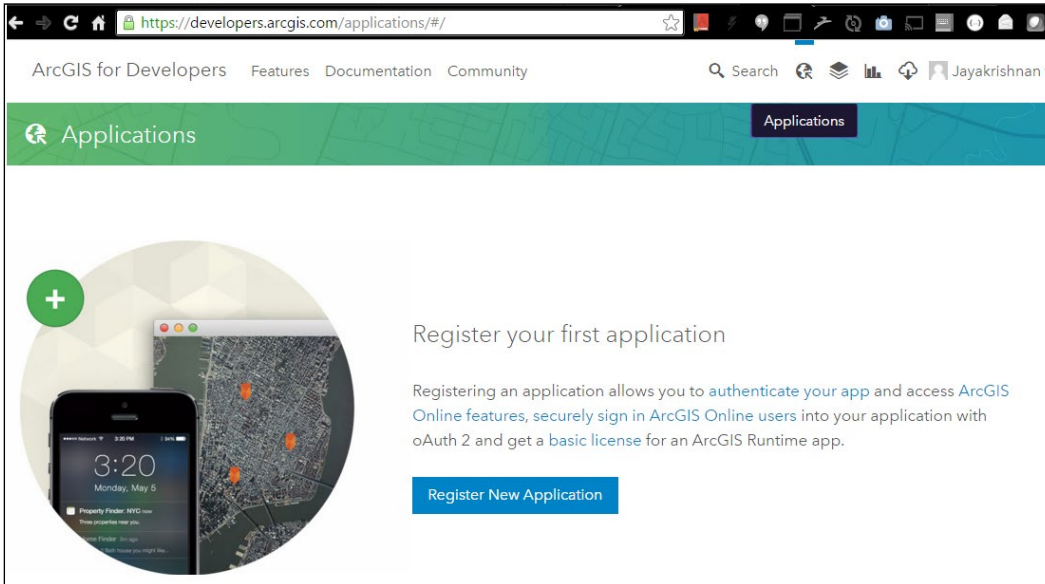
- Dark gray basemap
- Two operational map services, one displaying the Wildfire potential for the United States (raster data) and the other displaying Active Wildfire Locations (point data)
- A legend dijit (dojo widget) displaying the symbology of the layers added to the map
- A report widget that shows all the records of the Active Wildfire Locations

-
- A query widget where you can query Active Wildfire Locations based on the areal extent of Wildfire (this information is available in a field in the data)
 - A Find widget where you can enter any text, and all the States or Fire Names matching the search text will be fetched
 - A map click event that will identify and conspicuously display Wildfire Potential at the map click location
 - There are two operational data sources; one is the Wildfire Hazard Potential map service available at http://maps7.arcgisonline.com/arcgis/rest/services/USDA_USFS_2014_Wildfire_Hazard_Potential/MapServer and the other is Active Wildfire Data available at http://livefeeds.arcgis.com/arcgis/rest/services/LiveFeeds/Wildfire_Activity/MapServer
 - The latter map service is a *secured* map service, meaning that we need an ArcGIS Online account or an ArcGIS Developer account to use it. Apart from the preceding data sources, to access the vast pool of ArcGIS Online Data and the ones published in the Living Atlas of the World (<http://doc.arcgis.com/en/living-atlas/>), we need to do the following:
 - Register the app in ArcGIS Developer Portal and get a token for the app
 - Incorporate ArcGIS Proxy Code in our application

Registering the application in the developer portal

Using our ArcGIS Developer credentials (which we created as part of the *Setting up the development environment* section in *Chapter 1, Foundation for the API*), sign into the ArcGIS Developer portal (<https://developers.arcgis.com/>).

Next, navigate to the **Applications** page of the developer portal by clicking the appropriate icon as highlighted in the following screenshot. You can even do so by visiting <https://developers.arcgis.com/applications/>.



When we click on the **Register New Application** button, we will be prompted to enter the details about our application, as shown in the following screenshot. After providing the required details, if we click on the **Register New Application** button again, we will be led to another screen that displays the token for the app. This short-lived token can be used to access any secured ArcGIS Online map services. For example, try accessing the this in your browser—http://livefeeds.arcgis.com/arcgis/rest/services/LiveFeeds/Wildfire_Activity/MapServer.

You'll be redirected to a page that requires you to enter a token. When you provide the token that you got in the previous screen, you can see Service Catalog for the map service that we intend to see. The following screenshot explains this process:

The screenshot illustrates the process of registering a new application and logging in to the ArcGIS REST Services Directory. It is divided into three numbered steps:

- Step 1: New Application Details** - A form where you enter application information:
 - Title: LiveFeeds
 - Tags: Query
 - Description: An example of a short description explaining this application's purpose.
 A "Register New Application" button is at the bottom.
- Step 2: Generate Token** - A form where you enter credentials to generate a token:
 - Client ID: fy[]rk
 - Client Secret: ca[]357
 - Token: yk[]TlQfG91dFbl
 A "Generate Token" button is at the bottom.
- Step 3: ArcGIS Server REST API Login** - A page where you enter the token generated in Step 2:
 - Token: [] TOKEN
 A "Login" button is at the bottom.

The final screenshot shows the service catalog for the map service:

- ArcGIS REST Services Directory
- Home
- ArcGIS Server REST API Login
- To login to Services Directory when your site is federated to a portal, you must enter a token.
- To acquire this token, go to <https://www.arcgis.com> and enter 'http://livefeeds.arcgis.com/arcgis/rest' for the 'Webapp URL' parameter
- Token: [] TOKEN
- Login
- ArcGIS REST Services Directory
- Home
- Services > LiveFeeds > NOAA_METAR_current_wind_speed_direction (MapServer)
- JSON | SOAP
- LiveFeeds/NOAA_METAR_current_wind_speed_direction (MapServer)
- View In: [ArcGIS JavaScript](#) [ArcGIS.com Map](#) [ArcMap](#) [ArcGIS Explorer](#)
- View Footprint In: [ArcGIS.com Map](#)
- Service Description:
- Map Name: NOAA METAR current wind speed direction
- Legend
- All Layers and Tables
- Dynamic Legend
- Dynamic All Layers
- Layers:

 - Current Wind (0)

Using a proxy in the application

In this project, we need to use an Esri resource proxy to access secure ArcGIS Online data sources. The resource proxy is the server-side code that handles the request from the client to ArcGIS Server and forwards the response back from ArcGIS Server to the client. Esri has provided a proxy implementation that is specifically suitable for ArcGIS Server and ArcGIS Online. The Github code can be found at <https://github.com/Esri/resource-proxy>.

We will only be using the ASP.NET variant of the resource proxy that contains the following important files:

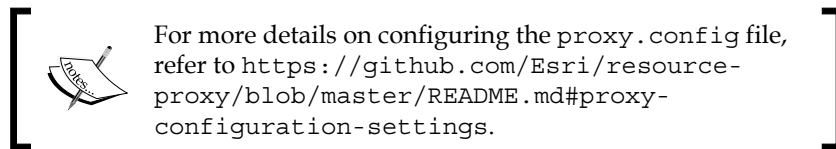
- proxy.ashx
- proxy.config
- Web.config

The proxy.ashx file contains the server-side code logic for making the request and forwarding the response back to the client. We need to configure proxy.config and include our ArcGIS Developer credentials in it. A sample proxy.config page is shown in the following screenshot:

```
<?xml version="1.0" encoding="utf-8" ?>
  <ProxyConfig allowedReferers="*" logfile="proxy_log_xml.log" mustMatch="false">
    <serverUrls>
      <serverUrl url="http://services.arcgisonline.com" matchAll="true" />
      <serverUrl url="http://server.arcgisonline.com/ArcGIS/rest/services"
        matchAll="true" />
      <serverUrl
        url="http://livefeeds.arcgis.com/arcgis/rest/services"
        tokenServiceUri="https://www.arcgis.com/sharing/generateToken"
        username="<username>"
        password="<password>"
        matchAll="true"
      />
    </serverUrls>
  </ProxyConfig>
```

To configure the `proxy.config` file, perform the following steps:

1. In the `proxy.config` file, modify the property values for `url`, `username`, and `password` in the `serverUrl` tag. For the `tokenServiceUri` property, the value should always be `https://www.arcgis.com/sharing/generateToken`.
2. For the `url` property, the value will be the location of the ArcGIS Server service. Specify either the specific URL (in this case, you will set `matchAll="false"`) or just the root URL (as shown in the preceding screenshot; in this case, the `matchAll` value will be `"true"`).



3. After configuring the proxy pages, we need to add a few more lines of code to our application. We need to load the `esri/config` module and use the following lines in our app code:

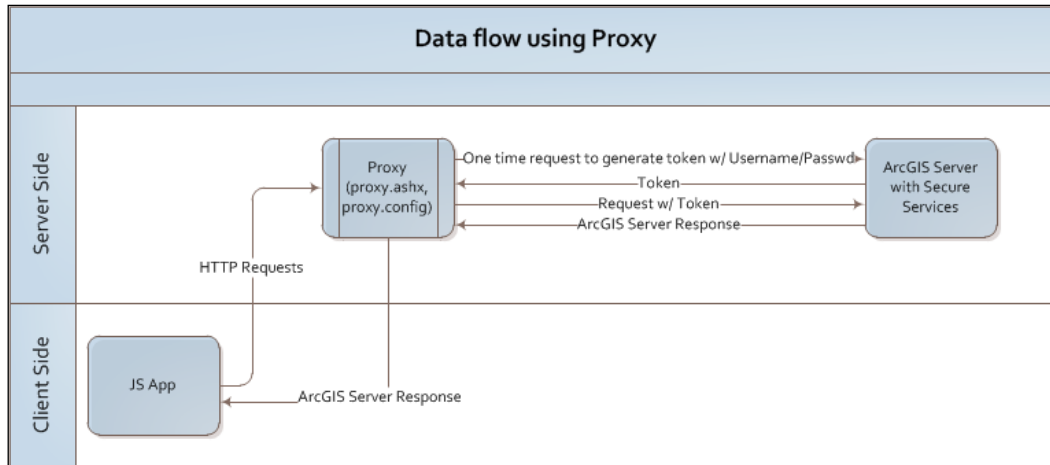
```
esriConfig.defaults.io.proxyUrl = "/proxy/proxy.ashx";
esriConfig.defaults.io.alwaysUseProxy = true;
```

In our application, the `proxy.ashx` page is located in the `proxy` folder at the application root. If the proxy pages are at a different application, we need to change the value for the `esriConfig.defaults.io.proxyUrl` variable. When we set the `esriConfig.defaults.io.alwaysUseProxy` value as `true`, all requests are handled by the proxy. If we need only specific URLs to be handled by the proxy, we may need to add a few more lines of code like this:

```
urlUtils.addProxyRule({
  urlPrefix: "route.arcgis.com",
  proxyUrl: "/proxy/proxy.ashx"
});
```

The `urlUtils` function is provided by the `esri/urlUtils` module.

The following diagram shows the flow of a HTTP REST request from the client to a secure ArcGIS Server service:



Bootstrapping the application

All the applications throughout this book are styled and bootstrapped using Bootstrap map libraries. The source code for these libraries can be found at <https://github.com/Esri/bootstrap-map-js>.

Once you download the required libraries, we will need to add the following CSS and JavaScript libraries to our application:

```
<head>
<!-- Bootstrap-map-js& custom styles -->
<link href="css/lib/bootstrap.min.css" rel="stylesheet">
<link rel="stylesheet" type="text/css"
href="css/lib/bootstrapmap.css">
<link rel="stylesheet" href="//netdna.bootstrapcdn.com/font-
awesome/4.0.3/css/font-awesome.css">
</head>
<body>

<script src="http://code.jquery.com/jquery-1.10.1.min.js"></script>
<script src="js/lib/bootstrap.min.js"></script>
</body>
```

Once these libraries are added, we need to add one more JavaScript file as a dojo module and not as a script reference. In our application, the JavaScript library under discussion is located at `/js/lib/bootstrapmap.js`.

When adding this library as a module in the require function, we need to omit the file extension. The following screenshot illustrates this statement:

```
require([
  "js/lib/bootstrapmap",
  "esri/config",
  "dojo/domReady!"],
  function (
    BootstrapMap,
    esriConfig) {
    var map = BootstrapMap.create("mapDiv", {
      basemap: "dark-gray",
      showAttribution: false,
      wrapAround180: true
    });
  });
```

So, instead of using the `esri/map` module, we will be using the `bootstrapmap` module to create the map. The `bootstrapmap` module accepts all the properties and methods that the `esri/map` provides, since the `bootstrapmap` module is just a wrap around the `esri/map` module.

Types of querying operations

Various types of querying operations are possible on the ArcGIS Server provided data. We will be dealing with the three most important querying operations provided by the API in this chapter:

- Query task
- Find task
- Identify task

Query task

Query task lets us operate just one layer, so the constructor for a Query task requires us to provide the URL of a feature layer. Query task lets us query the data using the attributes (field values; for example, query cities whose population is greater than 2 million) or using the location (for example, find all the gas stations that are within the current extent of the map or a custom-drawn extent). When the number of features satisfying the query conditions is greater than the limit set by the server (the `maxRecordCount` setting in ArcGIS Server), we can use a feature named *paging* to retrieve all the features on a batch mode.

Find task

Find task can operate on multiple layers in a map service and multiple fields. Find task basically searches for a given text in all the fields throughout all the layers in a given map service. When we don't know which field we are searching for and thus can't construct a proper SQL where clause to query the data, this is an ideal operation to rely upon.

Identify task

Identify task is predominantly a location-based search operation that returns all the data from all the layers in a given map service that intersect with a given geometry (such as a map-click point).

In all the preceding tasks, we can restrict the fields or layers upon which the search operation is being performed. The following matrix summarizes all the options available with the three different types of query operations:

	Find	Query	Identity
Attribute Based Search	TRUE	TRUE	FALSE
Location Based Search	FALSE	TRUE	TRUE
Supports Multiple Layers in a Service	TRUE	FALSE	TRUE
Supports Paging	FALSE	TRUE	FALSE

Building and executing a Query task

Query tasks are designed to query `featureLayer`. Thus, to instantiate `querytask`, we need to provide the URL of `featurelayer`. In version 3.15 of the API, the module is named `esri/tasks/QueryTask`.

The QueryTask constructor

The syntax for the `QueryTask` constructor is as follows:

```
newQueryTask(url, options?)
```

The example for the `QueryTask` constructor is as follows:

```
require([
  "esri/tasks/QueryTask", ...
], function(QueryTask, ... ) {
  varqueryTask = new QueryTask("<Feature Layer URL>")
});
```

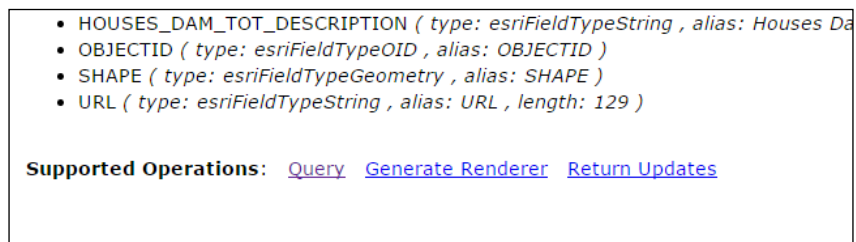
Constructor parameters

The URL of a feature layer that has query functionality is enabled, to verify that the query functionality on a feature layer is enabled, we have to visit the Service Catalog of the map service and check that `Query` is among the supported operations for the feature layer we'd like to query upon.

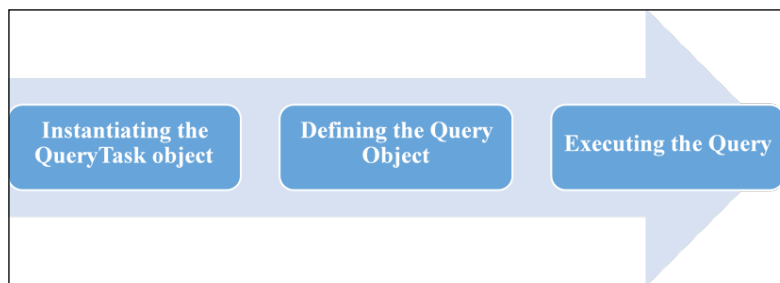
For example, in the Active Wildfire map service that we were dealing with, we'd like to query the layer containing data about Active Wildfire layers. There's just one layer in the map service, hence the layer index for the feature layer is 0.

The URL of the feature layer is `http://livefeeds.arcgis.com/arcgis/rest/services/LiveFeeds/Wildfire_Activity/MapServer/0`.

When we visit this link and scroll down to the bottom of the page where the **Supported Operations** section is found, we will see the `Query` operation being listed there:



Executing a query using a `Query` task involves the following steps:



Instantiating the QueryTask object

A Query task is based on the Active Wildfire feature layer. Hence, we will be using the feature layer's URL to instantiate the QueryTask object. The following lines of code explain how to instantiate QueryTask:

```
var wildFireActivityURL = "http://livefeeds.arcgis.com/arcgis/rest/  
services/LiveFeeds/Wildfire_Activity/MapServer/0";  
var queryTask = new QueryTask(wildFireActivityURL);
```

Building the Query object

The QueryTask object just defines which layer or data you want to query, but we need to use the Query object to define what the actual query is. The Query object is provided by the `esri/tasks/Query` module.

The Query object does the following:

- It forms a SQL `where` clause to query by attributes.
- It uses a spatial geometry to perform the query.
- It indicates the spatial relationship with which the query has to be executed.
- It requests an array of feature fields from the server.
- It indicates whether the query result needs to return geometric information.

The Query object has a property named `where`. This property accepts SQL's `where` clauses and fetches the data that satisfies the `where` clause.

The format of the `where` clause is as follows:

```
query.where = "<Query Expression 1><AND/OR><Query Expression 2> ...<AND/  
OR><QueryExpression n>"
```

Where `Query Expression` is "`<FieldName><operator><value>`";. And `<FieldName>` is the name of the field in the feature that we would like to query.

`<operator>` is a kind of SQL operator, such as `LIKE`, `=`, `>`, `<`.

The following snippet demonstrates the use of the `where` clause:

```
query.where = "STATE = 'OK' OR STATE = 'WY'";
```

When we would like to retrieve all the features from the `feature` class, the `where` clause needs to be set to a **truthy** expression, such as `1=1`. A truthy expression is something that evaluates to `true` under all circumstances.

You can use a truthy expression to retrieve all features:

```
query.where = "1=1";
```



In practice, the number of features returned using this expression is determined by the server setting known as `MaxRecordCount`, as shown in the following screenshot. The default value is 1000. This limit can be changed in the ArcGIS server settings.

When evaluating strings, do remember to enclose the string value within single quotes:

```
locQuery.where = "STATE_NAME = 'OK'";
```

The fields required in the output feature set can be passed as an array of field names to the query object parameter called `outFields`:

```
query.outFields = ["FIRE_NAME", "STATE ", "LATITUDE", "LONGITUDE"];
```

We can indicate whether we need the geometric information of the features by passing the value `true` or `false` to the query object parameter called `returnGeometry`:

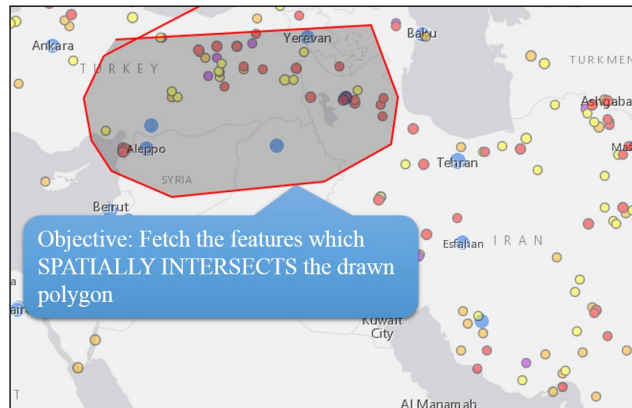
```
query.returnGeometry = true;
```

The following screenshot shows how to construct a complete `Query` object that can retrieve all the features from the feature layer set in the `Query` task object:

```
var query = new Query();
query.outFields = ["FIRE_NAME", "STATE", "LATITUDE",
"LONGITUDE"];
query.returnGeometry = false;
query.where = "1=1";
```

Querying by spatial geometry

We can fetch features from a feature layer that has a spatial relationship with another input geometry. We need to define what the spatial relationship between the input geometry and the features to be retrieved are. When not defined, the default spatial relationship becomes an intersection. This means that we are trying to fetch features that intersect with an input geometry.



There are other types of spatial relationship provided by the Query object as constants:

- `Query.SPATIAL_REL_CONTAINS`: This retrieves all the features that are entirely contained by the input geometry
- `Query.SPATIAL_REL_INTERSECTS`: This is the default spatial relationship where all features that intersect with the input feature are fetched
- `Query.SPATIAL_REL_TOUCHES`: Here, all the features that touch the input geometry are fetched

Normally, the input geometry maybe a selected feature from another feature, class, or geometry from the `draw` object, or in our case, the current map's extent, as shown in the following code snippet:

```
var query = new Query();
query.outFields = ["FIRE_NAME", "STATE", "LATITUDE", "LONGITUDE"];
query.returnGeometry = false;
query.where = "1=1";
query.geometry = map.extent;
query.returnGeometry = false;
```

Executing the query

When we need to execute a query and retrieve the results, we need to invoke the query execution methods in the Query task object. A Query task can be executed for getting the actual features that satisfy the query. In some cases, we may only need the count of features that satisfy the query or the spatial extent of the query result. There are five types of query operation that can be performed on the query task object:

- Query for Features
- Query for Count
- Query for Extent
- Query for Object IDs
- Query for Relationship

All operations accept the Query object as the first argument and return a deferred object. Let's understand the use of the three most important query task operations: Query for Count, Query for Extent and Query for Features.

Querying for Count

When we just want the count of the features that satisfy the query condition, we can use this operation. The following screenshot shows the Query for Count operation on a set of Query Features, given a Query object (with a Query Extent). The result will be the count of features that satisfy the Query object:

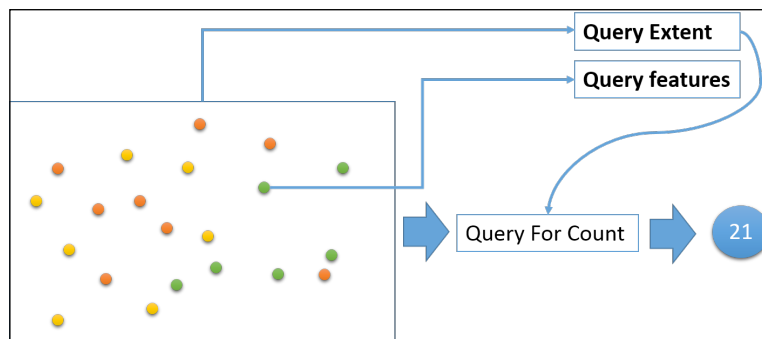
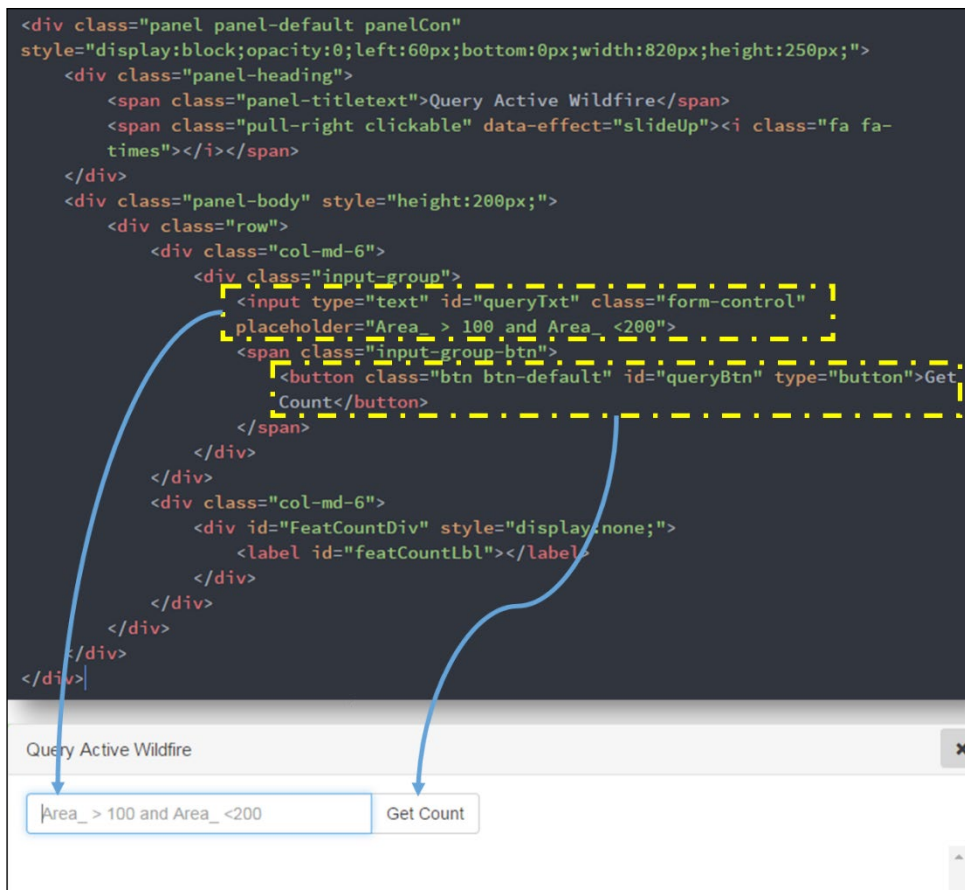


Diagram illustrating Feature Querying by Count

When we use the `executeForCount()` method of the query task, we still use the query object as the method argument. This can be an attribute query, a spatial query, or a combination of both. The main purpose of this method is to quickly assess the number of features a query operation returns. Sometimes, this may be the only information you need to display to the user.

Let's go ahead and create a UI to fetch the count of features that satisfy our query condition. The following screenshot shows a bootstrap panel with a text box to input the query text and the **Get Count** button. We have also provided another `div`, which is hidden. The `div` contains a label that shows the count of features.



Query execution should happen on clicking the **Get Count** button. When no input is provided in the query text box, the query will evaluate to a truthy expression; that is, the count of all the features within the map extent will be returned. The following code accomplishes just that:

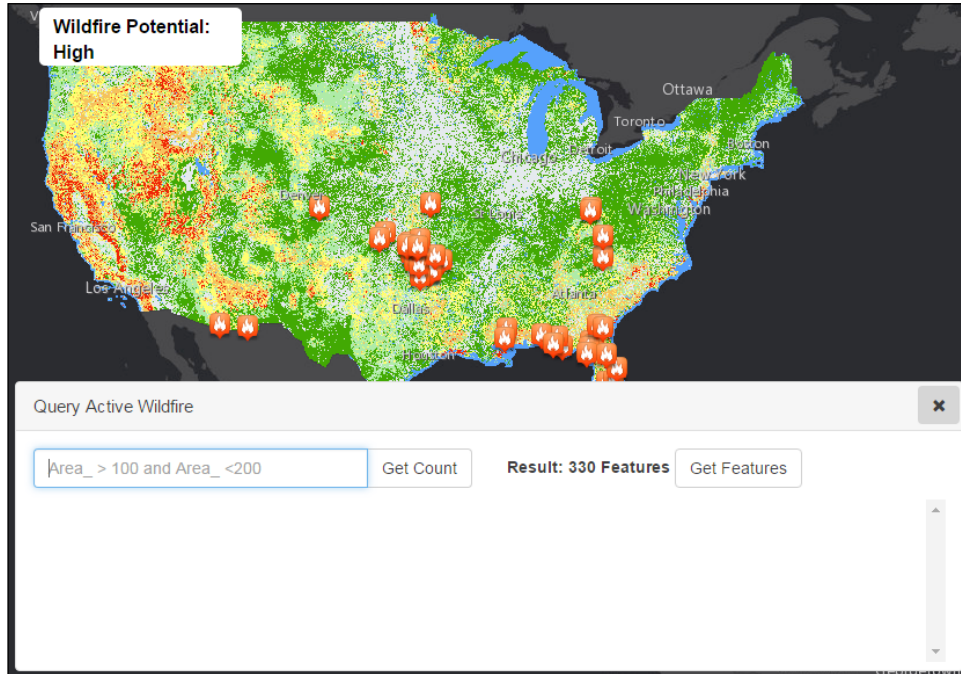
```
on(dom.byId("queryBtn"), "click", function () {
  query.outFields = ["FIRE_NAME", "STATE", "LATITUDE", "LONGITUDE"];
  query.returnGeometry = false;
  query.where = dom.byId("queryTxt").value || "1=1";
  query.geometry = map.extent;
  var queryCountDeferred = queryTask.executeForCount(query);
  queryCountDeferred.then(function (count) {
    dom.byId("FeatCountDiv").style.display = "block";
    dom.byId("featCountLbl").innerHTML = "Result: " + count + " Features";
  }, function (err) {
    console.log(err);
  });
});
```

In the preceding code snippet, `on` is the event handler module provided by `dojo` (`dojo/on`). The `byId()` method of the `dom` module is used to get the reference of the `dom` element with the ID—`queryBtn`. We are executing the preceding piece of code on the `click` event of `queryBtn`. Notice that in the highlighted code, we handle the situation when we receive no input from the query textbox. The `executeForCount()` method returns a `Deferred`. When the `Deferred` object is resolved, the `.then()` method is used to trigger the callback. Within the `.then` method, we have defined two functions; the first function is fired when the operation is successful, and the second function is fired when the operation throws an error. We can also use the `execute-for-count-complete` event on the `queryTask` object to retrieve the results.

The `result` object just returns the count number.

Refer to the following API documentation to get more information on the result object returned by this method—<https://developers.arcgis.com/javascript/jsapi/querytask-amd.html#event-execute-for-count-complete>.

The result of our operation on the map will look like the following screenshot:

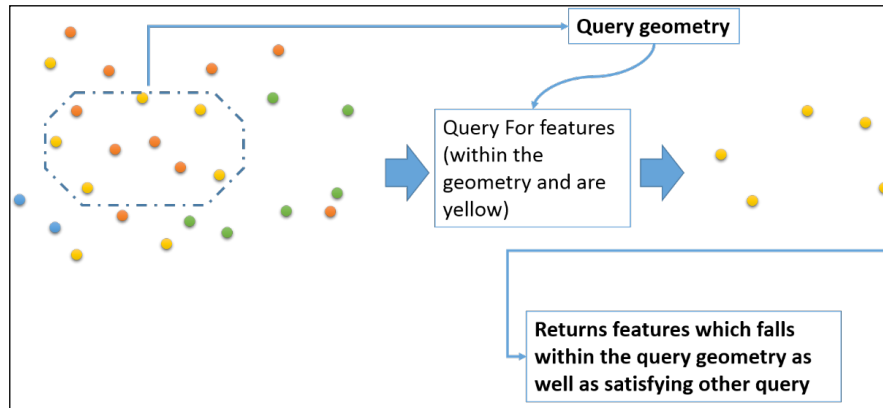


We have also introduced the **Get Features** button in the UI to retrieve the actual feature records that satisfy the query condition and display them in an HTML table. We will be executing the `execute()` method on the `queryTask` object to do this.

Query for Features

This method provides the maximum information about the features being queried.

A figure illustrating the Query for Features operation is shown here:



The `execute()` method in the `QueryTask` object is used to Query for Features. This method returns a `Deferred` object. This success event handler returns a `FeatureSet` object. The `FeatureSet` object returns an array of features along with other ancillary information regarding the geometry type and the spatial reference of the features.

The feature set contains the following:

- **features:** The graphic array. Each item in the graphic array has the following properties:
 - **attributes:** Name value pairs of fields and field values associated with the graphic
 - **geometry:** The geometry that defines the graphic
- **geometryType:** The geometry type of the features.
- **spatialReference:** The Spatial Reference of the features.

In our application, we will try to call the `execute` method on the click of a button, and we will construct an HTML string that will use the result called `FeatureSet` to display it as an HTML table. The following screenshot demonstrates how to iterate through the result feature set and create the HTML table string:

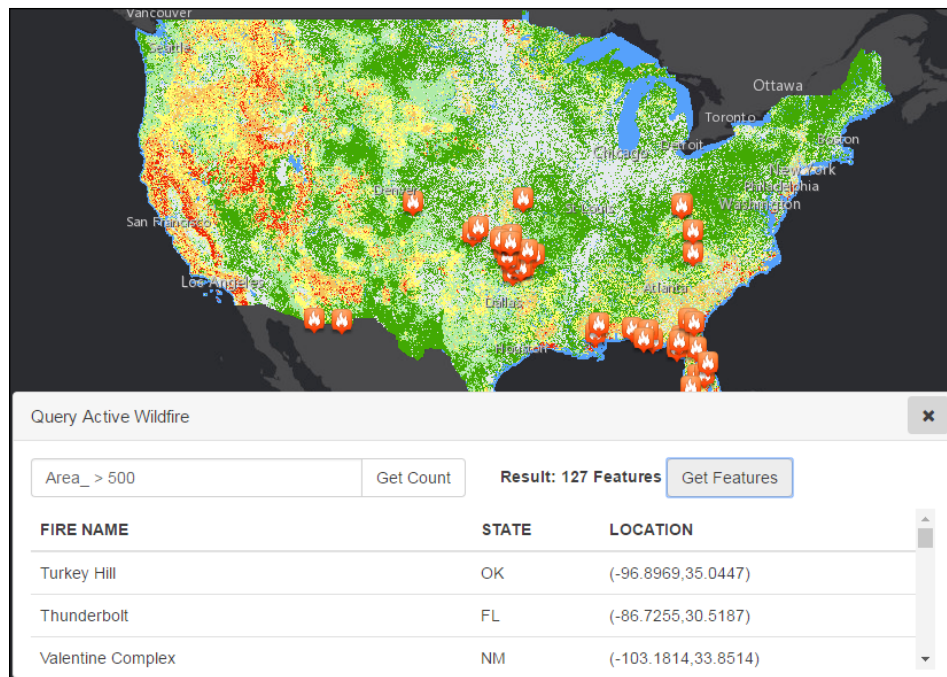
```
on(dom.byId("execQueryBtn"), "click", function () {
    var queryDeferred = queryTask.execute(query);
    queryDeferred.then(function (result) {
        var tblString = '<table class="table table-striped table-hover">';
        tblString += '<thead><tr><th>FIRE NAME</th>';
        tblString += '<th>STATE</th>';
        tblString += '<th>LOCATION</th>';
        array.forEach(result.features, function (feature) {
            tblString += '<tr><td>' + feature.attributes.FIRE_NAME + '</td>';
            tblString += '<td>' + feature.attributes.STATE + '</td>';
            tblString += '<td> (' + feature.attributes.LONGITUDE + ', ' +
                feature.attributes.LATITUDE + ')</td> </tr>';
        });
        tblString += '</tbody> </table >';
        dom.byId("QueryTbl").innerHTML = tblString;
    }, function (err) {
        dom.byId("QueryTbl").innerHTML = err;
    });
});
```

Constructing the HTML Table by iterating through each feature in the result featureset

On clicking on the **Get Features** button, the Query object that was used to get the count of the features is used to execute this query operation too. So ideally, every time we change the query text or the map extent, the **Get Features** button and the HTML query results will be hidden, and we need to click on the **Get Count** button before clicking on the **Get Features** button. We have written a function that hides the div that shows the feature count as well as clear the HTML table. The code is shown as follows:

```
function clearQueryTbl() {
    dom.byId("FeatCountDiv").style.display = "none";
    dom.byId("QueryTbl").innerHTML = '';
}
```

The following screenshot illustrates our code in action on the map:

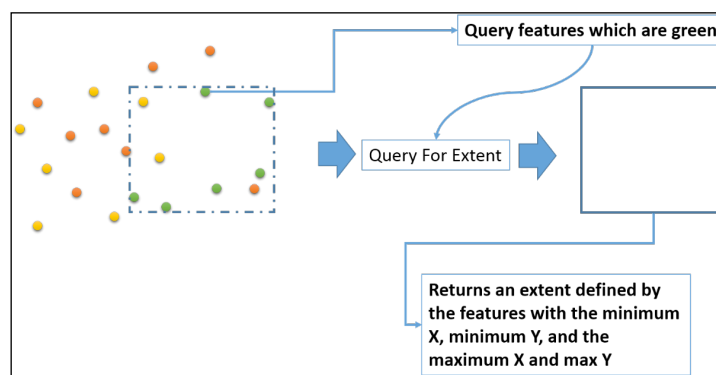


Query for Extent

When we would like to know the extent of the features satisfying a query, we can use this method. This will help us in many ways:

- We can get an idea about the spatial extent of the phenomena
- We can zoom the map to the extent of the features without actually receiving the features

The following diagram illustrates the Query for the Extent operation:



Building and executing IdentifyTask

`IdentifyTask` can operate on multiple layers in a map service and fetch information from all the features intersecting with a given geometry. We will use `IdentifyTask` to click on the map and get the value of the wildfire potential at the clicked location.

To execute `IdentifyTask`, we need to follow three steps:

1. Instantiate `IdentifyTask`.
2. Construct the `Identify` parameters.
3. Execute `IdentifyTask`.

Instantiating IdentifyTask

Instantiating `IdentifyTask` involves loading the required module and instantiating it with a map service URL. The modules required for executing `IdentifyTask` are the following:

- `esri/tasks/IdentifyTask`
- `esri/tasks/IdentifyParameters`

We will be operating `IdentifyTask` on the Wildfire Potential Map service. The map service contains a single raster layer and pixel values representing wildfire potential levels. The following snippet shows how `IdentifyTask` is instantiated:

```
varwildfirePotentialURL = "http://maps7.arcgisonline.com/arcgis/rest/  
services/USDA_USFS_2014_Wildfire_Hazard_Potential/MapServer";  
varidentifyTask = new IdentifyTask(wildfirePotentialURL);
```

Constructing the identify parameters object

`Identify` parameters provides a lot of properties to define the identify operation being performed. While dealing with multiple layers, we can restrict the layers upon which identify can be performed by using the `layerIds` property. The `geometry` property lets us set the geometry that is used to select features in the map service upon which identify operates. In our application, we are using the map `click` point as the input geometry for the `IdentifyParameter`. When using a point geometry, we also need to define the value for the tolerance property in the `IdentifyParameters`. The tolerance value refers to the number of pixels around the input point geometry that can be considered as part of the input geometry.

In the following screenshot, we construct an identify parameter object, which is wrapped around by the map `click` event handler. The `mapPoint` property of the map `click` event handler provides the input geometry for the identify operation:

```
identifyHandle = map.on("click", function (evt) {  
    var identifyParams = new IdentifyParameters();  
    identifyParams.geometry = evt.mapPoint;  
    identifyParams.tolerance = 1;  
    identifyParams.mapExtent = map.extent;  
});
```

Executing IdentifyTask

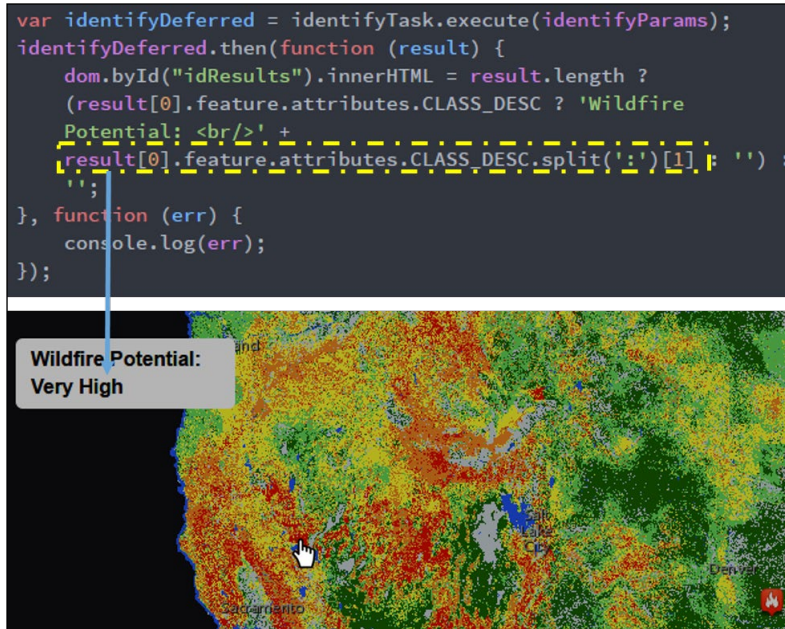
The `execute()` method of `IdentifyTask` can be used to execute the task. The `execute()` method returns the `Deferred` object, and the success callback of the `Deferred` object returns the `IdentifyResult` array object.

An identify result represents a single identified feature from one of the layers in the map service. The object has the following properties:

- `displayFieldName`: This is the name of the layer's primary display field
- `feature`: A feature object contains an array object and a geometry object
- `layerId`: This is the unique ID of the layer that contains the feature
- `layerName`: This is the name of the layer

Since the identify result is an array object, and we are only showing one value, we will be taking only the first value from the identify result object (`result[0]`), as shown in the following screenshot. The value that we need to show is in an attribute field named `CLASS_DESC`. Since this value is prefixed by a class code separated from the class description by a colon (`:`) (for example, `5: Very High`), we will be separating the string based on the colon and use the description part alone.

The following screenshot shows the code that is used to perform the identify operation as well as showing the identify result as a label for the map click location, which is represented by a pointer cursor:



Building and executing a Find task

A Find task is pretty much an attribute-based search on all the fields in a map service. Find task results are identical to IdentifyTask results with an extra value for `foundFieldName`, which indicates the field name in which the search text was found. Similar to Query task and IdentifyTask, the three steps to execute Find task are as follows:

1. Instantiate a Find task.
2. Build Find parameters.
3. Executing a Find task.

Let's discuss these three steps one by one.

Instantiating a Find task

To perform a Find task, the following modules need to be loaded:

- esri/tasks/FindTask
- esri/tasks/FindParameters

We need to provide the URL of a map service to instantiate a Find task. The following snippet shows how we will do this in our application:

```
var find = new FindTask("http://livefeeds.arcgis.com/arcgis/rest/services/LiveFeeds/Wildfire_Activity/MapServer");
```

Building the Find parameters

To construct the Find parameters, we need to use the `esri/task/FindParameters` module. The Find parameter module has properties such as `searchText`, `layerIds`, and `searchFields`, which let us define the Find task. The `searchText` property is the text that needs to be searched. This needs to come from a UI textbox. `layerIds` lets us define the `layerIds` upon which the Find task should operate. We can also restrict the fields upon which the search is performed. The following screenshot shows how we built the UI for the Find task and constructed the Find parameter object:

```
<div class="panel panel-default panelCon">
  <div class="panel-heading">
    <span class="panel-titletext">Find Active Wildfire</span>
    <span class="pull-right clickable" data-effect="slideUp"><i class="fa fa-times"></i></span>
  </div>
  <div class="panel-body" style="height:200px;">
    <div class="input-group">
      <input type="text" id="findTxt" class="form-control" placeholder="WY" />
      <span class="input-group-btn">
        <button class="btn btn-default" id="findBtn" type="button">Find
          Features</button>
      </span>
    </div>
    <!-- /input-group -->
    <div id="FindTbl" style="margin-top:10px;height:140px;overflow-y:scroll;">
    </div>
  </div>
</div>
```

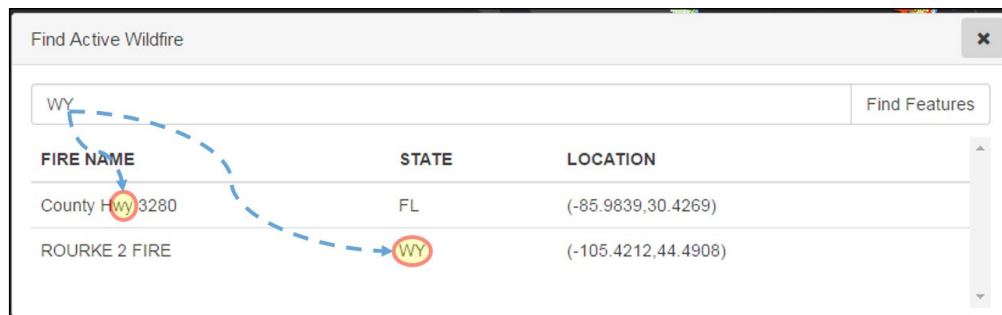
```
on(dom.byId("findBtn"), "click", function () {
  findParams.layerIds = [0];
  findParams.searchFields = ["STATE", "Fire Name"];
  findParams.searchText = dom.byId("findTxt").value;
});
```

Executing a Find task

The `execute()` method of `Find` task can be used to execute it. Calling this method will return a `Deferred` object, which will return a `Find results` object in its success callback function. We will try to build an HTML table, as we did for the `Query` task result, and display it in `FindTbl` div. The following lines of code were used to accomplish this:

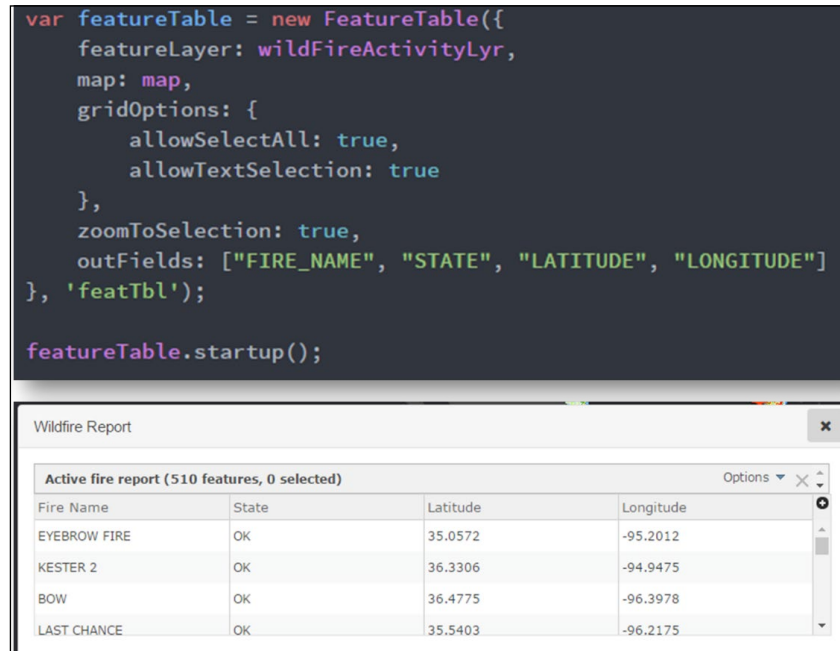
```
var findTaskDeferred = find.execute(findParams);
findTaskDeferred.then(function (result) {
  var tblString = '<table class="table table-striped
  table-hover">';
  tblString += '<thead><tr><th>FIRE NAME</th>';
  tblString += '<th>STATE</th>';
  tblString += '<th>LOCATION</th>';
  array.forEach(result, function (searchitem) {
    tblString += '<tr><td>' + searchitem.feature.attributes["Fire
    Name"] + '</td>';
    tblString += '<td>' + searchitem.feature.attributes["State"] +
    '</td>';
    tblString += '<td> (' +
    searchitem.feature.attributes["Longitude"] + ', ' +
    searchitem.feature.attributes["Latitude"] + ')</td></tr>';
  });
  tblString += '</tbody></table >';
  dom.byId("FindTbl").innerHTML = tblString;
}, function (err) {
  console.log(err);
});
```

In the following screenshot, we can see that the search text has fetched that from two different fields, **Fire Name** and **State** when we inserted the search text w:



Building a feature table

A feature table constructs a table, displays all the information of a given feature layer, and places it in a given dom element. A feature table is an Esri widget can be used by loading the `esri/dijit/FeatureTable` module. The module lets us choose the fields to display. The following screenshot shows how a feature table should be constructed and how it appears in the application:



Building popups

When users of your web application click on a feature of interest, they should be shown a bundle of useful information about the feature that they clicked on. Popups are the medium through which context-specific attribute information is shown to users. Popups complement the map's spatial information.

The simplest popups just show all or selected attribute values. More advanced and intuitive popups make use of charts and images in the pop-up window.

The modules that help to create popups are `esri/InfoTemplate`, `esri/dijit/PopupTemplate`, `esri/dijit/InfoWindow`, and `esri/dijit/Popup`.

`esri/dijit/PopupTemplate` extends `esri/InfoTemplate`, and `esri/dijit/Popup` extends `esri/dijit/InfoWindow`. So, let's deal with `InfoTemplate` briefly and move on to the `Popup` templates.

Building InfoTemplates

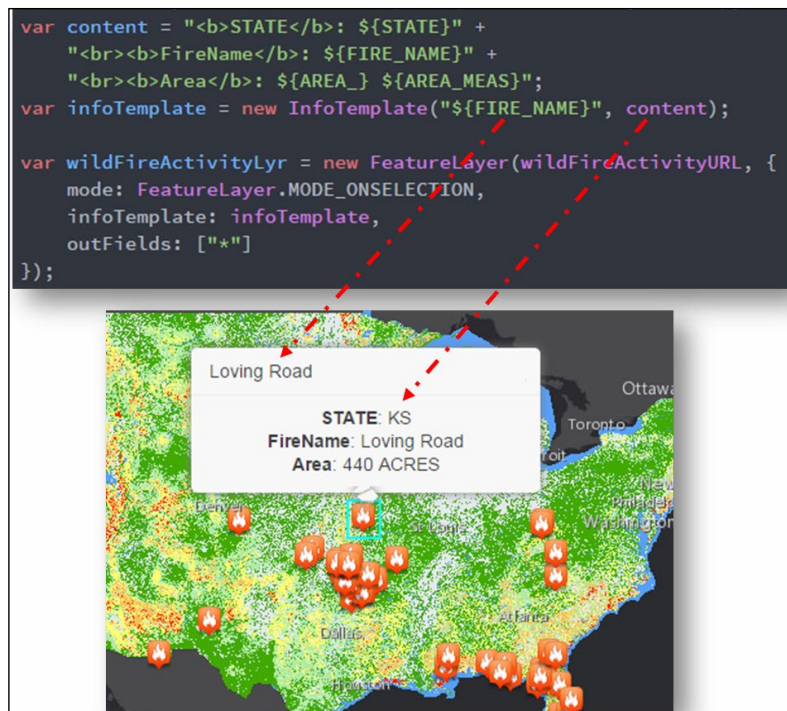
An `InfoTemplate` object can be created using placeholders. A placeholder is usually the attribute field name, starting off with a dollar (\$) sign and surrounded by curly braces ({}), for example, `${Fieldname}`.

When we need to retrieve all the fields provided by the feature of interest, the fieldname can be substituted by `*`, for example, `${*}`.

The feature layers and `Graphics` object have the `InfoTemplate` property. `infotemplate` created could be set to these layers. The `InfoTemplate` constructor takes two arguments, title and content:

Modules	Values
Module name	<code>esri/InfoTemplate</code>
Parent object	Feature layer, Graphic object, dynamic layer, and the Info window of the map
Constructor	<code>new InfoTemplate (title, content)</code>

The following screenshot creates `infotemplate` for the `Active wildfire` feature layer, and it displays fields such as state name, fire name, and the areal extent of the wildfire feature being clicked in a popup. The title of `Infotemplate` is also created by the placeholders:



The code listings for this chapter can be found in the code folder named B04959_03_CODE.

Summary

This chapter explained the different methods for searching and querying data. We built an application that could perform a Query task, a Find task, as well as Identify task. We also discovered the utility of the feature table called `dijit` along with `InfoTemplates`. In the next chapter, we will see how to organize all the code into modularized widgets and use it in our application. We will also be discussing how to construct spatial queries that involve using the draw toolbar, and we will create input geometries that are defined by the user of the app.

4

Building Custom Widgets

The main objective of this chapter is to develop a custom widget that can perform a spatial query and display the results in a simple HTML table. In the process of building the custom widget, you will learn the following topics:

- How to create a simple class using dojo
- How to configure dojo globally
- What is the lifecycle of a dojo widget
- How to create a template widget
- How to provide support for internationalization
- How to organize the dojo code
- How the draw toolbar works
- How to build the custom widget using all the features discussed in the chapter

Creating a simple class

Dojo classes provide a way to inherit and extend other modules to use templating as well as create widgets. Classes in dojo reside within a module and the module returns the class declaration. To declare classes within a module, we need to load a module named `dojo/_base/declare`, which provides support for declaring classes.

The following screenshot shows a simple dojo class declaration:

```
1 ▾ define([
2   //class
3   "dojo/_base/declare"
4 ▾ ], function (declare) {
5   return declare(null, {
6     prop1: 1,
7     prop2: "sample",
8   ▾ constructor: function(name){
9     console.log(name);
10  },
11 ▾ myMethod: function () {
12    return 1;
13  }
14  });
15 });
```

The screenshot shows a code editor with a JavaScript snippet for a dojo class declaration. The code is as follows:

```
1 ▾ define([
2   //class
3   "dojo/_base/declare"
4 ▾ ], function (declare) {
5   return declare(null, {
6     prop1: 1,
7     prop2: "sample",
8   ▾ constructor: function(name){
9     console.log(name);
10  },
11 ▾ myMethod: function () {
12    return 1;
13  }
14  });
15 });
```

Four callout boxes are present on the right side of the code, each with a line pointing to a specific part of the code:

- Callout 1: "Module required to define class" points to the string "dojo/_base/declare" on line 3.
- Callout 2: "Module shall return the class declaration" points to the "return declare" call on line 5.
- Callout 3: "Class is a super class, it doesn't inherit from any other" points to the "declare(null, {" call on line 5.
- Callout 4: "Class properties, methods and constructor" points to the constructor function on line 8.

In this screenshot, `declare` is the callback function decoration for the `dojo/_base/declare` module. The class declaration accepts three arguments: *classname*, *superclass*, and *properties*.

The classname argument is optional. When a classname string is provided, the declaration is called a **named class**. When it is omitted, as in our case, it is called an **anonymous class**. We will stick with using anonymous classes for a while as named classes must be used only under particular conditions.


The superclass is the module or an array of modules that we would like to extend. If the superclass argument is null (as in our snippet), it means that our class declaration itself is a superclass.

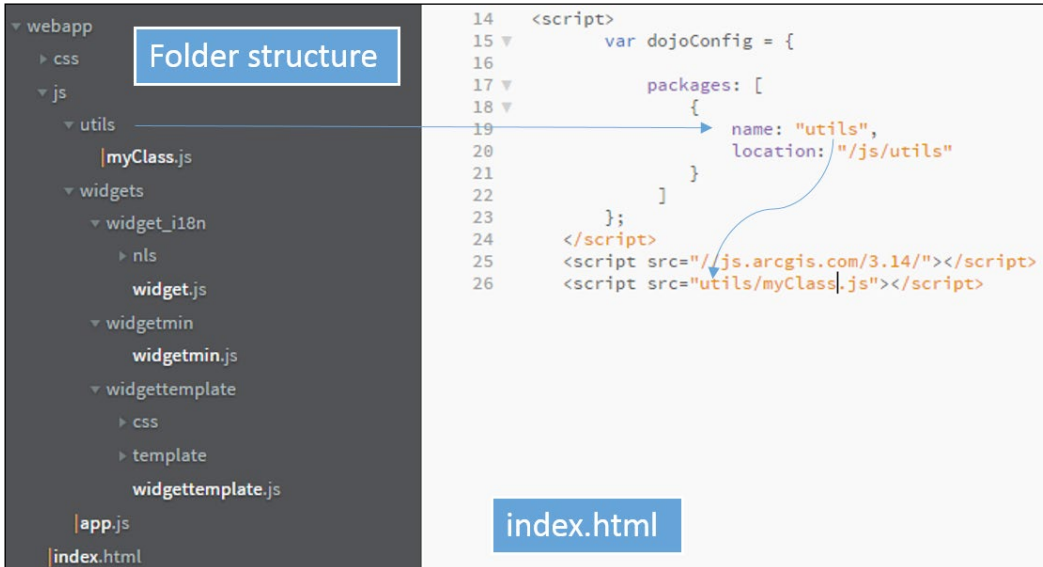
The third argument in the class declaration is the class properties. We can define the class constructor, other class properties, and class methods here.

Configuring dojo

Dojo has a global object named `dojoConfig` which holds all the configuration parameters. We can modify the `dojoConfig` object to configure options and default behavior for the various aspects of the dojo toolkit.

The `dojoConfig` object lets us define the location for the custom modules defined in our web application and tags it with a package name. So, when we need to load these custom modules, we can refer to the folder location using the package name instead.

 The `dojoConfig` object must be declared before referencing the Esri JS API.



The screenshot displays a file explorer on the left and a code editor on the right. The file explorer shows a folder structure for a web application, with a blue box labeled "Folder structure" pointing to the `utils` folder. The code editor shows the `index.html` file, with a blue box labeled "index.html" pointing to the code. The code defines the `dojoConfig` object and loads the dojo toolkit and a custom module.

```

14 <script>
15   var dojoConfig = {
16
17     packages: [
18       {
19         name: "utils",
20         location: "/js/utils"
21       }
22     ]
23   };
24 </script>
25 <script src="//js.arcgis.com/3.14/"></script>
26 <script src="utils/myClass.js"></script>

```

There are other configuration options such as `async`, `parseOnLoad`, `waitSeconds`, and `cacheBust`. For detailed information on the `dojoConfig` topic, refer to the dojo toolkit documentation at https://dojotoolkit.org/documentation/tutorials/1.10/dojo_config/.

- The `async` option defines whether the dojo core should be loaded asynchronously. The recommended value is `true`.
- The `locale` option lets us override the default language provided to dojo by the browser. This will help us develop the app for a different target locale and test our widgets for the internationalization support using dojo's `i18n` module.
- The `cacheBust` option is a very useful option, which when configured to `true`, appends the time string to each URL from the module, thus avoiding module caching.

Let's see how these options work out for us:

```
<script>
  var dojoConfig = {
    has: {
      "dojo-debug-messages": true
    },
    parseOnLoad: false,
    locale: 'en-us',
    async: true,
    cacheBust: true,
    packages: [
      {
        name: "widgets",
        location: "/js/widgets"
      },
      {
        name: "utils",
        location: "/js/utils"
      }
    ]
  };
</script>
<script src="//js.arcgis.com/3.14/"></script>
<script src="js/app.js"></script>
```

Name	Method	Status	Type	Initiator
<input type="checkbox"/> 3.14/	GET	302		(index):39
<input type="checkbox"/> app.js	GET	200	script	(index):40
<input type="checkbox"/> style.css	GET	200	stylesheet	(index):38
<input type="checkbox"/> init.js	GET	200	script	http://js.arcgis.com/3.14/
<input type="checkbox"/> jsapi_en-us.js?1445808976371	GET	200	script	init.js:30
<input type="checkbox"/> myClass.js?1445808976371	GET	200	script	init.js:30
<input type="checkbox"/> svg.js?1445808976371	GET	200	script	init.js:30
<input type="checkbox"/> blank.gif?1445808976371	GET	200	gif	init.js:757
<input type="checkbox"/> content.min.css	GET	200	xhr	content.min.js:1

Effect of configuring cacheBust to True in the dojoConfig object

Developing a standalone widget

Developing standalone widgets is the main purpose of writing classes in dojo. Dojo exclusively provides us a module for supporting the development of widgets: `dijit/_WidgetBase`. We also need other ancillary modules such as `dijit` templating modules, the `dojo` parse, and `dojo` internationalization modules to develop a full-fledged widget in a web application.

The key aspect associated with the `WidgetBase` module is the concept of the life cycle of a widget. The widget life cycle gives us methods to work with during the different stages of the widget, that is, from the initialization of the widget, to the stage when its `dom` nodes are fully loaded and utilizable by the application, until the destruction of the widget.

This module should be passed in as a superclass array in the class declaration. Here is the snippet for a basic widget:

```
define([
    //class
    "dojo/_base/declare",

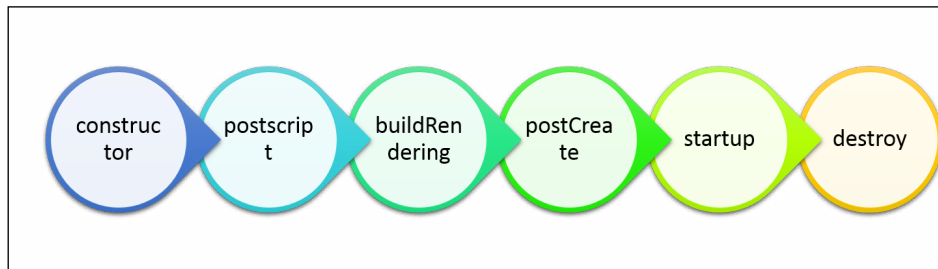
    //widget class
    "dijit/_WidgetBase",

    "dojo/domReady!"
], function (
    declare,
    _WidgetBase
) {
    return declare(_WidgetBase, {
        /*Class declaration inherits "dijit/_WidgetBase" module*/

        constructor: function () {}
    });
});
```

The dijit life cycle

The `_widgetBase` option provides several methods that the program flow will execute in a particular sequence. Some of the most important methods executed in order are shown in the following infographic:



The widget lifecycle infographic

The preceding diagram can be described as follows:

- **constructor:** This is the first method called when the widget is instantiated. The `constructor` function can be used as a special property named `domNode`. This can contain the value of the reference to `domNode` where the widget will be placed. The first argument for the `constructor` function will be an `options` object to which we can send any object value we would like to send to the widget:

```
constructor: function (options, srcRefNode) {  
    this.domNode = srcRefNode;  
}
```

- **postCreate:** This method is executed just after all the properties of the widget are executed. All the event handlers for the widget will be defined here. A particular line of code should be added in the `postCreate()` method so that all the definitions made in `WidgetBase` will be inherited properly. In the following code snippet, the particular line of code has been highlighted:

```
postCreate: function() {  
    this.inherited(arguments);  
}
```

- **postCreate():** This method is also the right place to host the special `this.own()` method. The event handler defined within this method will release event handles when the instance of the widget is destroyed.
- **Startup:** This method is fired after the `dom` nodes are constructed. So, any modification to the `dom` node will be done here. This is the method through which the widget will be called externally for execution.

Creating templated widgets

Templated widgets are ones that allow the developer to load an HTML file as a template string at runtime. All the dom nodes specific to the widget should be defined in this HTML template. Dojo provides two more modules to make our experience of using templates easier and more efficient. These modules are named `dijit/_TemplatedMixin` and `dijit/_WidgetsInTemplateMixin`. Apart from these two modules, we also need to load a dojo plugin named `dojo/text!`, which actually loads the HTML page as a template string. The way the plugin works is that the HTML file path should be appended after the exclamation (!) in `dojo/text!`:

```
"dojo/text!app_widgets/widgettemplate/template/_widget.html"
```

The class properties should include a specific property named `templateString`. The value of the `this` property will be the callback function decoration used to represent the `dojo/text!<filename.html>` plugin.

Let's see a basic code snippet that covers all the topics discussed previously and tries to develop a template widget:

```
define([
  //Modules for Class declaration
  "dojo/_base/declare",
  "dojo/_base/lang",

  //widget class
  "dijit/_WidgetBase",

  //Module for loading templated widget
  "dijit/_TemplatedMixin",

  //Plugin to load HTML Template file
  "dojo/text!app_widgets/widgettemplate/template/_widget.html",

  "dojo/domReady!"
], function (
  declare,
  lang,
  _WidgetBase,
  _TemplatedMixin,
  dijitTemplate
) {
  return declare([_WidgetBase, _TemplatedMixin], {
    //assigning html template to template string
    templateString: dijitTemplate,
    constructor: function (options, srcRefNode) {
      this.domNode = srcRefNode;
    },
    postCreate: function () {
      this.inherited(arguments);
    },
    startup: function () {

    },
    destroy: function () {

    }
  });
});
```

Our template file is very innocuous, containing a simple h1 header tag. It is this HTML string that the `templateString` property holds.

The contents of the `app_widgets/widgettemplate/template/_widget.html` file are as follows:

```
<h1>This is Templated widget</h1>
```

Now, let's see how to instantiate this widget. As mentioned earlier, we need to call the `startup` method in the widget to execute this widget. We will call this from another JavaScript file, which will pass a reference to the dom node where our widget will be placed:

```
require([
  "app_widgets/widgettemplate/widgettemplate",
  "dojo/domReady!"
],
function (widgettemplate) {
  var templateWidget = new widgettemplate({}, /* Pass an empty object */
    'templatedWidgetDiv' /*Reference to the dom element where the widget shall be placed */
  );
  templateWidget.startup();
});
```

Contents of `/js/widgets/app.js`

This file will be called from the `index.html` file, which has the dom element named `templatedWidgetDiv`:

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
  <title></title>
  <meta name="description" content="">
  <meta name="viewport" content="width=device-width">
  <link rel="stylesheet" type="text/css" href="css/style.css">
</head>

<body>
  <h1>Using Dojo Classes</h1>
  <div id="templatedWidgetDiv"/>
  <script>
    /* dojo config */
```

```
var dojoConfig = {
  has: {
    "dojo-debug-messages": true
  },
  parseOnLoad: false,
  locale: 'en-us',
  async: true,
  cacheBust: true,
  packages: [
    {
      name: "app_widgets",
      location: "/js/widgets"
    },
    {
      name: "utils",
      location: "/js/utils"
    }
  ]
};
</script>
<!--Call the esri JS API library-->
<script src="//js.arcgis.com/3.14/"></script>
<!--Call the /js/utils/app.js file-->
<script src="js/app.js"></script>
</body>

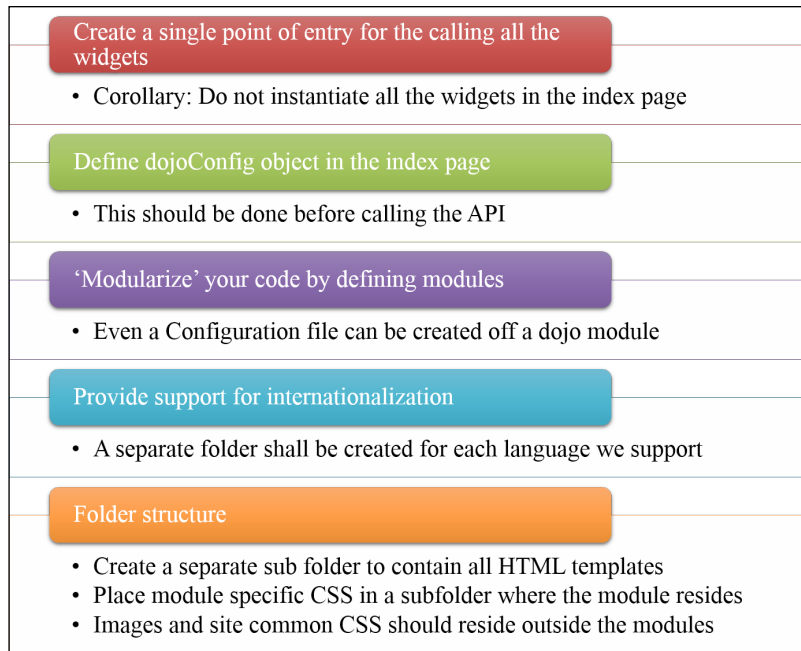
</html>
```

Widget folder structure

It is now time to discuss the widget folder structure. When developing large projects, the folder structure is an important part of the project-building exercise that we need to define at the initial stage of the project itself. We will be providing general guidelines on how to decide upon the folder structure. This can be modified to suit your preferences and project needs.

Guidelines for creating project folders

The guideline for creating project folders are describe in the following diagram:



Let's discuss each of these in a detailed manner.

Creating a single point of entry

We need not crowd the index page with all widget instantiation. It's better if we can define a module that can serve as the single point of entry to instantiate all the widgets we need.

Our HTML page will only contain a reference to the JS API and this single point of entry JavaScript module:

```
<!--Call the esri JS API library-->  
<script src="//js.arcgis.com/3.15/"></script>  
<!--Call the javaScript file which serves as the single point of  
entry-->  
<script src="js/app.js"></script>
```

The contents of the file serving as the single-point-of-entry are as follows:

```
require(["dojo/dom",
        "app_widgets/widgettemplate/widgettemplate",
        "utils/myClass",
        "app_widgets/widget_il8n/widget_il8n",
        "dojo/domReady!"], function (dom, WidgetTemplate, MyClass, Widget_il8n) {
    var msgDiv = dom.byId("msgDiv");
    /* Instantiate MyClass module */
    var myClass = new MyClass('const param');
    msgDiv.innerHTML += '<br>Property 1: ' + myClass.prop1;
    msgDiv.innerHTML += '<br>Property 2: ' + myClass.prop2;
    msgDiv.innerHTML += '<br>Class Method: ' + myClass.myMethod();

    /* Instantiate WidgetTemplate widget */
    var templateWidget = new WidgetTemplate({}, 'templatedWidgetDiv');
    templateWidget.startup();

    /* Instantiate Widget_il8n */
    var s = new Widget_il8n({}, 'widgetlocal');
    s.startup();
});
```

Defining dojoConfig

We discussed this earlier. The `dojoConfig` object will be declared in the index page itself. This is a global object whose values can be accessed anywhere in the program by loading the module named `dojo/_base/config`.

Modularizing the code

This is the core tenet of the AMD pattern of coding. The concept of modularizing means that we should decouple any code that is functionally different. As we know, the dojo modules return a publicly accessible object. This object can be a class declaration, as we saw earlier. A different use of the module is that this can be used as a configuration file for the application.

A sample `config` file that has been created off a dojo module has been provided for your reference:

```
define(function () {
    /* Private variables*/
    var baseUrl = "http://maps.ngdc.noaa.gov/arcgis/rest/services/";
    var web_mercator_etopo1_hillshade_MapServer = "http://maps.ngdc.noaa.gov/arcgis/rest/services/web_mercator_etopo1_hillshade_MapServer";
    var NOAA_MapService = "http://maps.ngdc.noaa.gov/arcgis/rest/services/web_mercator_hazards_MapServer";
    var earthquakeLayerId = 5;
    var volcanoLayerId = 6;
});
```

```
    /*publicly accessible object returned by the COnfig module */
    return {
        app: {
            currentVersion: "1.0.0"
        },

        // valid themes: "claro", "nihilo", "soria", "tundra",
        "bootstrap", "metro"
        theme: "bootstrap",

        // url to your proxy page, must be on same machine hosting you
        app. See proxy folder for readme.
        proxy: {
            url: "proxy/proxy.ashx",
            alwaysUseProxy: false,
            proxyRuleUrls: [NOAAMapService]
        },

        map: {

            // basemap: valid options: "streets", "satellite",
            "hybrid", "topo", "gray", "oceans", "national-geographic", "osm"
            defaultBasemap: "streets",
            visibleLayerId: [this.earthquakeLayerId, this.
            volcanoLayerId];

            earthquakeLayerURL: this.NOAAMapService + "/" + this.
            earthquakeLayerId,
            volcanoLayerURL: this.NOAAMapService + "/" + this.
            volcanoLayerId
        }
    }
});
```

Providing support for internationalization

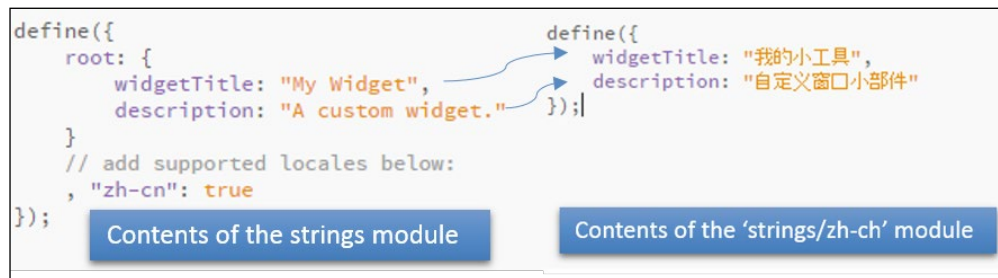
Customizing the text that is shown in the app according to the locale of the user is known as **internationalization**. Dojo provides a plugin named `dojo/i18n!` to provide this support. When we mention plugin, it means that it expects a file path as an argument after the exclamation mark (!). The file path refers to a JavaScript module, which mentions an object named `root` and lists all the supported locales.

For example, `dojo/i18n!app_widgets/widget_i18n/nls/strings` refers to the `strings` module defined within the `app_widgets/widget_i18n/nls` folder (remember that `app_widgets` is the name package referring to the `/js/widgets` location).

The current locale is determined by the user's browser. A `locale` in `dojo` is usually a five-letter string; the first two characters represent the language, the third character is a hyphen, and the last two characters represent the country.

For example, take a look at the following:

- The `en-us` value represents English as the language and the US as the country
- The `ja-jp` value represents Japanese as the language and Japan as the country
- The `zh-cn` value represents Simplified Chinese as the language and China as the country
- The `zh-tw` value represents Simplified Chinese as the language and Taiwan as the country



Steps to provide internationalization support

The steps to provide internationalization support are as follows:

1. Create a folder named `nls` in the folder where the widget resides.
2. Define a module that has an object named `root` and lists all the locales supported below the `root` object. For example, take a look at the following:


```

"zh-cn" : true,
"de-at" : true

```
3. The `root` object will contain all the string variables for which language support is provided, for example, `widgetTitle` and `description`.

4. Create a folder for each locale defined, such as zh-cn and de-at.
5. Create a module with the same name as that of the `root` module in each of the language folders.
6. The new modules will contain all the properties of the `root` object. The value of the properties will contain the language-specific translation of the corresponding values.
7. Load the module named `dojo/i18n!` appended with the path to the root module.
8. In the `declare` constructor, assign the `callback` function declaration of the `i18n` module to the property named `this.nls`:

```
define([
    //class
    "dojo/_base/declare",
    "dojo/_base/lang",

    //widget class
    "dijit/_WidgetBase",

    //templated widget
    "dijit/_TemplatedMixin",

    // localization
    "dojo/i18n!app_widgets/widget_i18n/nls/strings",

    //loading template file
    "dojo/text!app_widgets/widget_i18n/template/_widget.html",

    "dojo/domReady!"
], function (
    declare, lang,
    _WidgetBase,
    _TemplatedMixin,
    nls,
    dijitTemplate
) {
    return declare([_WidgetBase, _TemplatedMixin], {
        //assigning html template to template string
```

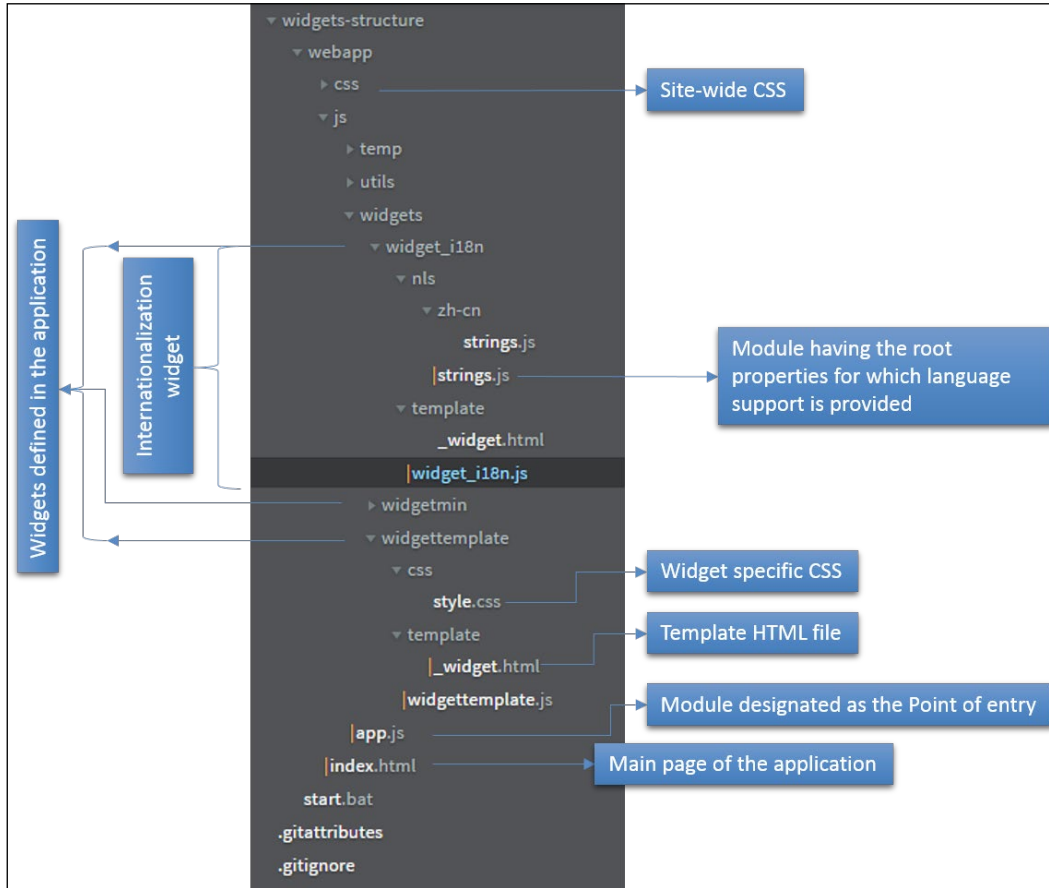
```
    templateString: dijitTemplate,
    constructor: function (options, srcRefNode) {
        console.log('constructor called');
        // widget node
        this.domNode = srcRefNode;
        this.nls = nls;
    },
    // start widget. called by user
    startup: function () {
        console.log('startup called');
    }
    });
});
```

An overview of the widget folder structure

Let's review the widget folder structure once again so that we can use this as a template before starting any project:

1. We need a main file (say `index.html`). The main file should have the `dojoConfig` object, a reference to all the CSS used in the app, and the Esri CSS. It should also have a reference to the API and a reference to the module, which serves as the point of entry (`app.js`).
2. All widgets go into the `js` folder.
3. All the site-wide CSS and images go into the `css` and `image` folders in the application root directory, respectively.
4. All the widgets will be placed within the `widgets` folder inside the `js` folder. Each widget can be placed within a separate folder within the `widgets` folder too.
5. Templates will be placed inside the `template` folder within the widget folder.

6. Place the resources needed for internationalization within a folder named `nls`:



Building a custom widget

We will extend the app that we developed in the last chapter with advanced functionalities and modularized code refactoring. Let's create a custom widget in the app that does the following:

- Allows the user to draw a polygon on the map. The polygon will be symbolized by a semi-transparent red fill and a dashed yellow outline.
- The polygon should fetch all the major wildfire events within the boundary of the polygon.
- This shall be shown as a graphic and the data should be in a grid.
- Internationalization support must be provided.

Modules required for the widget

Let's list the modules required to define classes and their corresponding intended callback function decoration.

Modules for the class declaration and OOPS

Modules	Values
dojo/_base/declare	declare
dijit/_WidgetBase	_WidgetBase
dojo/_base/lang	lang

Modules for using HTML templates

Modules	Values
dijit/_TemplatedMixin	_TemplatedMixin
dojo/text!	dijitTemplate

Module for using event

Modules	Values
dojo/on	on
dijit/allclick	allclick

Modules for manipulating dom elements and their styles

Modules	Values
dojo/dom-style	domStyle
dojo/dom-class	domClass
dojo/domReady!	

Modules for using the draw toolbar and displaying graphics

Modules	Values
esri/toolbars/draw	Draw
esri/symbols/SimpleFillSymbol	SimpleFillSymbol
esri/symbols/SimpleLineSymbol	SimpleLineSymbol
esri/graphic	Graphic
dojo/_base/Color	Color

Modules for querying data

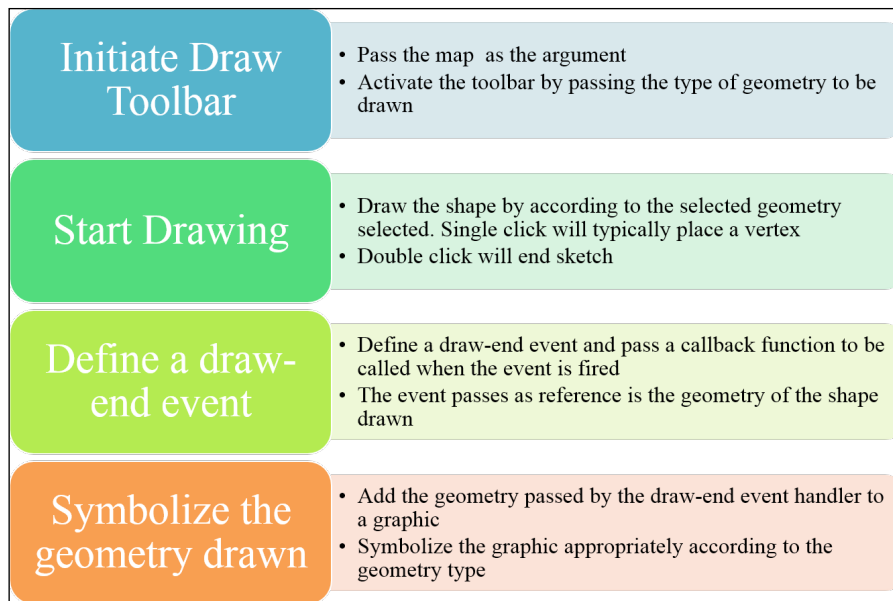
Modules	Values
esri/tasks/query	Query
esri/tasks/QueryTask	QueryTask

Modules for internationalization support

Modules	Values
dojo/i18n!	nls

Using the draw toolbar

The draw toolbar enables us to draw graphics on the map. This toolbar has events associated with it. When a draw operation is completed, it returns the object drawn on the map as geometry. Follow these steps to create a graphic using the draw toolbar:



Initiating the draw toolbar

The draw toolbar is provided by the module called `esri/toolbars/draw`. The draw toolbar accepts the map object as an argument. Instantiate the draw toolbar within the `postCreate` function. The draw toolbar also accepts an additional optional argument named `options`. One of the properties in the `options` object is named `showTooltips`. This can be set to `true` so that we can see a tooltip associated while drawing. The text in the tooltip can be customized. Otherwise, a default tooltip associated with draw geometry is displayed:

```
return declare([_WidgetBase, _TemplatedMixin], {
  //assigning html template to template string
  templateString: dijitTemplate,
  isDrawActive: false,
  map: null,
  tbDraw: null,
  constructor: function (options, srcRefNode) {
    this.map = options.map;
  },
  startup: function () {},
  postCreate: function () {
    this.inherited(arguments);
    this.tbDraw = new Draw(this.map, {showTooltips : true});
  }
  ...
});
```

The draw toolbar can be activated on the `click` or `touch` event (in case of smart phones or tablets) of a button, which is intended to indicate the start of a draw event. Dojo provides a module that takes care of `touch` as well as `click` events. The module is named `dijit/allclick`.

To activate the draw toolbar, we need to provide the type of symbol to draw. The draw toolbar provides a list of constants, which corresponds to the type of the draw symbol. These constants are `POINT`, `POLYGON`, `LINE`, `POLYLINE`, `FREEHAND_POLYGON`, `FREEHAND_POLYLINE`, `MULTI_POINT`, `RECTANGLE`, `TRIANGLE`, `CIRCLE`, `ELLIPSE`, `ARROW`, `UP_ARROW`, `DOWN_ARROW`, `LEFT_ARROW`, and `RIGHT_ARROW`.

While activating the draw toolbar these constants must be used to define the type of the draw operation required. Our objective is to draw a polygon at the click of a draw button. The code is shown in the following screenshot:

```
postCreate: function () {
    this.inherited(arguments);
    // events
    this.own(
        /* setup an event handler (automatically remove() when destroyed) */
        on(this.btndrawpoly, 'click', lang.hitch(this, this.toggleDraw)),
        on(this.btnclear, 'click', lang.hitch(this, function(){
            this.map.graphics.clear();
        })))
    );
    this.tbDraw = new Draw(this.map);
    this.tbDraw.on("draw-end", lang.hitch(this, this.querybyGeometry));
},
toggleDraw: function () {
    domClass.toggle(this.btndrawpoly, "btn-danger");
    if (!this.isDrawActive) {
        this.tbDraw.activate(Draw.POLYGON);
        this.isDrawActive = true;
    } else {
        this.tbDraw.deactivate();
        this.isDrawActive = false;
    }
},
},
```

Draw.POLYGON is a constant which is passed as an argument to the DrawToolbar

The draw operation

Once the draw toolbar is activated, the draw operation begins. For point geometry, the draw operation is just a single click. For a polyline and a polygon, the single click adds a vertex to the polyline, and a double-click ends the sketch. For a freehand polyline or polygon, the `click` and `drag` operation draw the geometry and a `mouse-up` operation ends the drawing.

The draw-end event handler

When the draw operation is complete, we need an event handler to do something with the shape that was drawn by the draw toolbar. The API provides a `draw-end` event, which is fired once the draw operation is complete. This event handler must be connected to the draw toolbar. This event handler will be defined within the `this.own()` function inside the `postCreate()` method of the widget. The event result can be passed to a named or anonymous function:

```
postCreate: function () {
  ...
  this.tbDraw.on("draw-end", lang.hitch(this,
    this.querybyGeometry));
},
...
querybyGeometry: function (evt) {
  this.isBusy(true);
  //Get the Drawn geometry
  var geometryInput = evt.geometry;
  ...
}
```

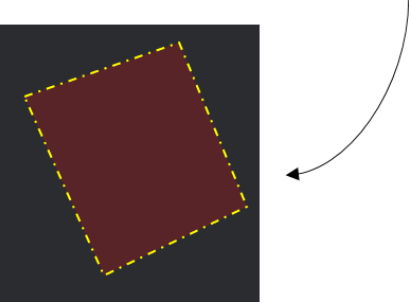
Symbolizing the drawn shape

In the `draw-end` event callback function, we will get the geometry of the drawn shape as the result object. To add this geometry back to the map, we need to symbolize it. A symbol is associated with the geometry that it symbolizes. Also, the styling of the symbol is defined by the colors or pictures used to fill up the symbol and its size. Just to symbolize a polygon, we need to use the `SimpleFillSymbol` and `SimpleLineSymbol` modules. We may also need the `esri/color` module to define the fill colors.

Let's review a snippet to understand this better. This is a simple snippet used to construct a symbol for a polygon with a semi-transparent solid red color fill and a yellow dash-dot line:

```
1  var tbDrawSymbol =
2      new SimpleFillSymbol(
3          SimpleFillSymbol.STYLE_SOLID,
4          new SimpleLineSymbol(
5              SimpleLineSymbol.STYLE_DASHDOT,
6              new Color([255, 255, 0]),
7              2
8          ),
9          new Color([255, 0, 0, 0.2])
10 );
```

Symbol type used to style a polygon geometry
The Fill type constant
Symbol type for the outline of the polygon
The Line type constant for the outline
Color (yellow) and width of the outline
Red Color fill for the polygon along with transparency




In the preceding screenshot, `SimpleFillSymbol.STYLE_SOLID` and `SimpleLineSymbol.STYLE_DASHDOT` are the constants provided by the `SimpleFillSymbol` and `SimpleLineSymbol` modules respectively. These constants are used to style the polygon and the line.

Two colors are defined in the construction of the symbol: one for filling up the polygon and the other for coloring the outline. A color can be defined by four components. They are as follows:

- Red
- Green
- Blue
- Opacity

The red, green, and blue components takes values from 0 to 255 and the opacity takes a value from 0 to 1. A combination of red, green, and blue components can be used to produce any color according to the RGB color theory. So, to create a yellow color, we use the maximum of red component (255) and the maximum of green component (255); we don't want the blue component to contribute to our color, so we use 0. An opacity value of 0 means 100% transparency, and an opacity value of 1 means 100% opaqueness. We have used 0.2 for the fill color. This means that we need our polygon to be 20% opaque, or 80% transparent. The default value for this component is 1.

A symbol is just a generic object. This means any polygon geometry can use the symbol to render itself. Now, we need a container object to display the drawn geometry with the previously defined symbol on the map. A graphic object provided by the `esri/Graphic` module acts as a container object, which can accept a geometry and symbol. The graphic object can be added to the map's graphic layer.

 A graphic layer is always present in the map object, which can be accessed by using the `graphics` property of the map (`this.map.graphics`).

```
querybyGeometry: function (evt) {
    this.tbDraw.deactivate();
    this.toggleDraw();
    this.isDrawActive = false;
    this.isBusy(true);

    var geometryInput = evt.geometry;
    var tbDrawSymbol = new SimpleFillSymbol(SimpleFillSymbol.STYLE_SOLID, new
    SimpleLineSymbol(SimpleLineSymbol.STYLE_DASHDOT, new Color([255, 255, 0]), 2), new
    Color([255, 255, 0, 0.2]));
    this.map.graphics.clear();
    var graphicPolygon = new Graphic(geometryInput, tbDrawSymbol);
    this.map.graphics.add(graphicPolygon);
}
```

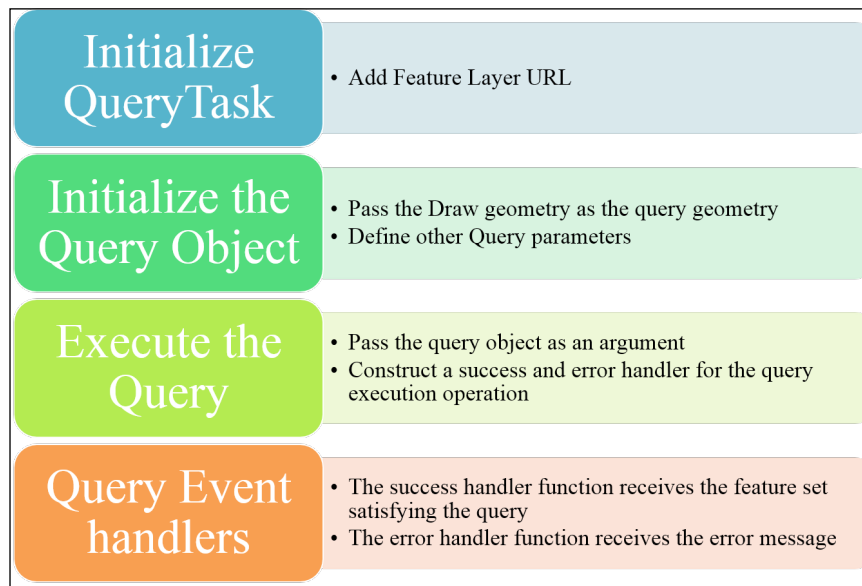
Draw Symbol defined

Graphic object is defined with Draw geometry and the Symbol

Graphic object is added to the map's graphic layer

Executing the query

The widget's main function is to define and execute a query based on the user's draw input. The following image will provide a general way for us to construct a `querytask` and handle the execution:



Initializing the QueryTask and Query object

We will be working with Active Wildfire feature layers that we used in the previous chapter. When providing the input geometry, we will be using the geometry that we got from the `draw-end` event instead of using the map's current extent geometry, like we did in the previous chapter. We will be fetching all the features within the draw geometry, hence we will be using the truthy expression (`1=1`) as the `where` clause. The following lines of code explain how the `query` object is constructed and how the `queryTask` is executed and stored as a deferred variable:

```
var queryTask = new QueryTask(this.wildFireActivityURL);
var query = new Query();
query.where = "1=1";
query.geometry = geometryInput;
query.returnGeometry = true;
query.outFields = ["FIRE_NAME", "AREA_", "AREA_MEAS"];
var queryDeferred = queryTask.execute(query);
```

Query event handlers

The `execute` method on the `QueryTask` object returns a deferred variable. This means that we should use the `.then()` operation to elicit the task execution result. The success handler returns a `featureset`. A `featureset` is an array of features. A feature contains a graphic as well as some attributes.

Now, there are two operations that we need to perform to display the query results:

1. Highlight the query result by symbolizing it appropriately and adding it as appropriate graphics on the map.
2. Show the details of Active Wildfires satisfying the query conditions in a simple HTML table. The HTML table should come from an HTML template file.

Defining the HTML template

We need an HTML template to render the widget. This widget will have the following components:

- A button whose `click` event will toggle the draw event
- A button to clear the draw graphic, as well as the result graphic and the HTML table
- A `dom` element to hold onto the HTML table being constructed

The following screenshot explains how the HTML template is constructed:

```
<div>
  <button data-dojo-attach-point="btndrawpoly" style="float:right;" class="btn btn-
  success" data-toggle="tooltip" data-placement="right" title="Click on map to draw
  polygon"><span class="glyphicon glyphicon-pencil"></span></button>

  <button data-dojo-attach-point="btnclear" style="float:right;" class="btn btn-
  success" data-toggle="tooltip" data-placement="right" title="Clear"><span
  class="glyphicon glyphicon glyphicon-refresh"></span></button>

  <span data-dojo-attach-point="loadingdiv" class="loading" style="display:none;">
  </span>
  <div style="height:145px;overflow-y:auto;">
    <table class="table">
      <thead>
        <tr>
          <th style="width:10px;">#</th>
          <th style="width:180px;">Fire Name</th>
          <th style="width:20px;">Area</th>
          <th style="width:50px;">Units</th>
        </tr>
      </thead>
      <tbody data-dojo-attach-point="tbcontent">
      </tbody>
    </table>
  </div>
</div>
```

The screenshot shows a web application window titled "Spatial Query - Active Wildfire". The window contains a table with the following structure:

#	Fire Name	State	Area
---	-----------	-------	------

Annotations in the image:

- A blue box highlights the `<tbody data-dojo-attach-point="tbcontent">` tag in the code, with an arrow pointing to the table in the UI. A label below it reads "DOM element to build the HTML Table".
- A blue box points to the `btndrawpoly` button in the code, with an arrow pointing to a green button with a refresh icon in the UI. A label next to it reads "Draw Toggle Button".
- A blue box points to the `btnclear` button in the code, with an arrow pointing to a green button with a pencil icon in the UI. A label next to it reads "Clear Graphics button".

This HTML file should be loaded as a plugin using the `dojo/text!` plugin. Once this is done, all the dom elements referred by `dojo-attach-point` can be accessed in the code using this notation. Also, functions to handle the click events for the `toggleDraw` button and the `clear` button should be implemented. The following screenshot shows a barebones implementation of this:

```

define([
  "dojo/text!appWidgets/SpatialQuery/template/_spatialquery.html",
  "dojo/domReady!"
], function () {
  Our template HTML should be inherited as templateString property inside our app
  dijitTemplate({
    return declare("wildfireEventwidget", [_WidgetBase, _TemplatedMixin], {
      templateString: dijitTemplate,
      isDrawActive: false,
      map: null,
      tbDraw: null,
      constructor: function (options, srcRefNode) {
        this.map = options.map;
      },
      postCreate: function () {
        this.inherited(arguments);
        this.own(
          on(this.btndrawpoly, 'click', lang.hitch(this, this.toggleDraw)),
          on(this.btnclear, 'click', lang.hitch(this, function () {
            this.map.graphics.clear();
            this.tbcontent.innerHTML = '';
          })))
        );
        this.tbDraw = new Draw(this.map);
        this.tbDraw.on("draw-end", lang.hitch(this, this.querybyGeometry));
      },
      toggleDraw: function () {
        Implement Toggle Draw function on click of the corresponding button
        domClass.toggle(this.btndrawpoly, "btn-danger");
        ...
      },
      querybyGeometry: function (evt) {
        ...
      }
    });
  });
});

```

Symbolizing query results

The features returned by the query are wildfire locations, all of which have a point geometry. We can use `SimpleMarkerSymbol` or `PictureMarkerSymbol` to symbolize features returned by the query. The `PictureMarker` symbol accepts the following properties:

- angle
- xoffset
- yoffset
- type
- url
- contentType
- width
- height

We will use a PNG resource, which is part of the application to define `PictureMarkerSymbol`:

```
var symbolSelected = new PictureMarkerSymbol({
  "angle": 0,
  "xoffset": 0,
  "yoffset": 0,
  "type": "esriPMS",
  "url": "images/fire_sel.png",
  "contentType": "image/png",
  "width": 24,
  "height": 24
});
```

Adding the graphics to the map

All the query result features should be converted into a graphic with the `PictureMarkerSymbol` that we just defined. Additionally, we will also be adding an `infotemplate` to each graphic. The `infotemplate` content will be taken from query result attributes. The HTML table can also be constructed by iterating through the features returned by the query result object. The following screenshot illustrates the entire process clearly:

.then() handles the deferred returned by the Query execution

```

queryDeferred.then(lang.hitch(this,
function (result) {
    this.map.graphics.clear();
    var str = '';
    for (var i = 0; i < result.features.length; i++) {
        var featAttr = result.features[i].attributes;
        var featGeom = result.features[i].geometry;
        var infoTemplate = new InfoTemplate(featAttr.FIRE_NAME,
"Area:" + featAttr.AREA_);
        var selectionGraphic = new Graphic(featGeom, symbolSelected,
null, infoTemplate);
        this.map.graphics.add(selectionGraphic);
        str = str + '<tr><th scope="row">' + (i + 1) + '</th><td>' +
featAttr.FIRE_NAME + '</td><td>' + featAttr.STATE + '</td>' +
<td>' + featAttr.AREA_ + " " + featAttr.AREA_MEAS + '</td>' +
</tr>';
    }
    this.map.infoWindow.show();
    this.tbcontent.innerHTML = str;
    this.isBusy(false);
}),
function (err) {
    /*Error handler*/
    console.log(err);
    this.isBusy(false);
});

```

Success handler

Result object contains a feature set (array of features)

Iterate through the array of features

Construct the HTML from the attribute info

Assign the HTML to the DOM element

Error handler



Spatial Query - Active Wildfire

#	Fire Name	State	Area
1	Rumuda	NM	619 ACRES
2	Jester	NM	2250 ACRES
3	La Joya 1	NM	298 ACRES

The complete code listing can be found in the folder called B049549_04_CODE02.

Summary

In this chapter, you learned how to create classes and custom widgets in dojo, and you also learned about the life cycle of a dojo widget. Then, we walked through the guidelines for creating a folder structure for any dojo-related project. We also looked at how we can provide support for different languages using the internationalization feature provide by the dojo module. Finally, we created a custom widget that uses a draw tool to accept a user-drawn polygon and uses it to query a feature layer. We showed the results in an HTML table as well as on the map. In the following chapters, we will be dealing with how to symbolize the graphics better and intuitively using a technique known as rendering. Rendering is a great visualization technique that lets us define rules to symbolize features differently, based on the value of a particular attribute in the feature. In further chapters, we will be extending the visualization techniques to cover non-spatial representations of data such as charts and graphs.

5

Working with Renderers

Renderers provide us with a medium to visualize data intuitively using different symbols and colors. More than a data visualization technique, renderers are increasingly considered as a data analytic tool. The correct use of renderers will help us see spatial patterns in the data and display the geographic distribution of various phenomena. An understanding of basic cartography, color theory, and even statistics will help us create better renderers and eventually better insights into the available data. The following topics will be covered in this chapter:

- Learning about different symbols and colors provided by the API
- Learning how to create a `SimpleRenderer` method
- Learning how to create a `UniqueValueRenderer` method efficiently
- Learning when to use `ClassBreakRenderer` and `HeatmapRenderers`
- Discussing scenarios where `ScaleDependantRenderers` can be useful
- An introduction to smart mapping

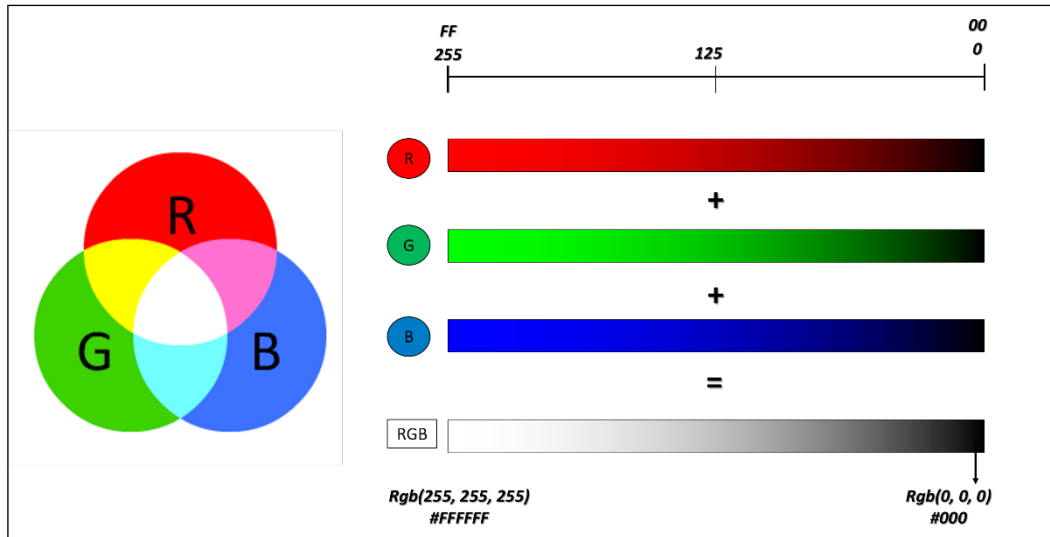
Working with colors

The Esri module dealing with colors is called `esri/Color`. Before dealing with the color module, let's have a fundamental understanding of colors.

The RGB color model

Any color in the visible spectrum (the range of colors between Violet to Red), can be represented using a combination of Red (R), Green (G) or Blue (B) colors. This is known as the **RGB color model**. There are other color models too, but let's stick with the RGB color model for now. And each color R, G, or B can be expressed in a scale from 0 to 255.

The following picture shows the relationship between the three primary colors (R, G, and B) and their additive effect:



When the three colors (R, G, and B) are mixed in equal proportion, the resultant color always lies somewhere in the grey scale. The following points are worth noting:

- For example, if the level of $R = 0, G = 0,$ and $B = 0,$ the mixture produces black.
- If $R = 255, G = 255,$ and $B = 255,$ the mixture produces white.
- Any other number values, when mixed equally, produce a shade of grey. For example, if $R = 125, G = 125,$ and $B = 125,$ it will be grey.
- The color model also shows that when Red and Green are mixed together ($R = 255, G = 255,$ and $B = 0$), we get yellow.
- When Red and Blue alone are mixed ($R=255, G= 0,$ and $B=255$), we get Magenta.
- When Green and Blue are mixed, we get Cyan ($R=0, G=255,$ and $B= 255$).

The Esri color module

To define a color using the RGB color model, the following format can be used:

```
var r = g = b = 125;  
var color = new Color([r, g, b]);
```

In the preceding snippet, `color` is an instance of the `esri/Color` module and `r`, `g`, and `b` are values for Red, Green, and Blue respectively. The colors should always be ordered as (`r`, `g`, and `b`) and added as an array object. As expected, the `color` variable stores a grey color. If we need to add transparency to the color, we can define the transparency value known as `alpha`, which is an integer between 0 and 1.0, where 0 represents full transparency and 1.0 represents no transparency. The `alpha` value will be added as the fourth value in the array:

```
define(["esri/Color"], function(Color) {
  var r = g = b = 100;
  var alpha = 0.5; // 50 % transparency
  var color2 = new Color ([r, g, b, alpha]);
})
```

The RGB values can be represented as a hexadecimal number. For example, [255, 0, 0] can be represented as `#FF0000`. The API also allows us to represent the color by its English named string, for example, `blue`:

```
define(["esri/Color"], function(Color) {
  var colorString = "red";
  var colorHex = "#FF0000";
  var color1 = new Color(colorString);
  var color2 = new Color(colorHex);
})
```

Working with symbols

Symbols are based on the geometry that they try to symbolize. Thus, the symbols used to represent a point, line, and polygon are different from each other. Apart from the geometry, the three important parameters required to define a symbol are the following:

- Style
- Color
- Dimension (or size)

The style is usually provided as a module constant. For example, `SimpleLineSymbol.STYLE_DASHDOT`, `SimpleFillSymbol.STYLE_SOLID`, and `SimpleMarkerSymbol.STYLE_CIRCLE` where `SimpleLineSymbol`, `SimpleFillSymbol`, and `SimpleMarkerSymbol` are the modules used to symbolize the line, polygon, and point features respectively:

- The colors of these symbols can be defined by the color modules that we discussed in earlier sections.

- The dimension or size means different things based on the geometry type. For example, for a line symbol, we use the parameter known as `width` to refer to the line thickness, whereas for a point, we use the parameter named `size` to define its dimension.

Let's discuss about the three geometry-based symbols first, and then we will deal with the non-geometry-based and special symbols.

The geometry-based symbols are as follows:

- `SimpleLineSymbol`: This is used to symbolize the line geometry
- `SimpleMarkerSymbol`: This is used to symbolize the point geometry
- `SimpleFillSymbol`: This is used to symbolize the polygon geometry

SimpleLineSymbol

The line symbol constructor is the simplest, because it can be defined with just three parameters namely style, color, and width.

Name	Value
Module name	<code>esri/symbols/SimpleLineSymbol</code>
Constructor	<code>new SimpleLineSymbol(style, color, and width)</code>

The `style` is a module constant. The following styles are provided by the module:

- `STYLE_DASH` (to create lines made of dashes)
- `STYLE_DASHDOT` (to create lines made of a dash-dot pattern)
- `STYLE_DOT` (to create lines made of dots)

The module provides other style constants such as `STYLE_LONGDASH`, `STYLE_LONGDASHDOT`, `STYLE_NULL`, `STYLE_SHORTDASH`, `STYLE_SHORTDASHDOT`, `STYLE_SHORTDASHDOTDOT`, `STYLE_SHORTDOT`, and `STYLE_SOLID`.

`STYLE_SOLID` is the default style, which provides an uninterrupted solid line.

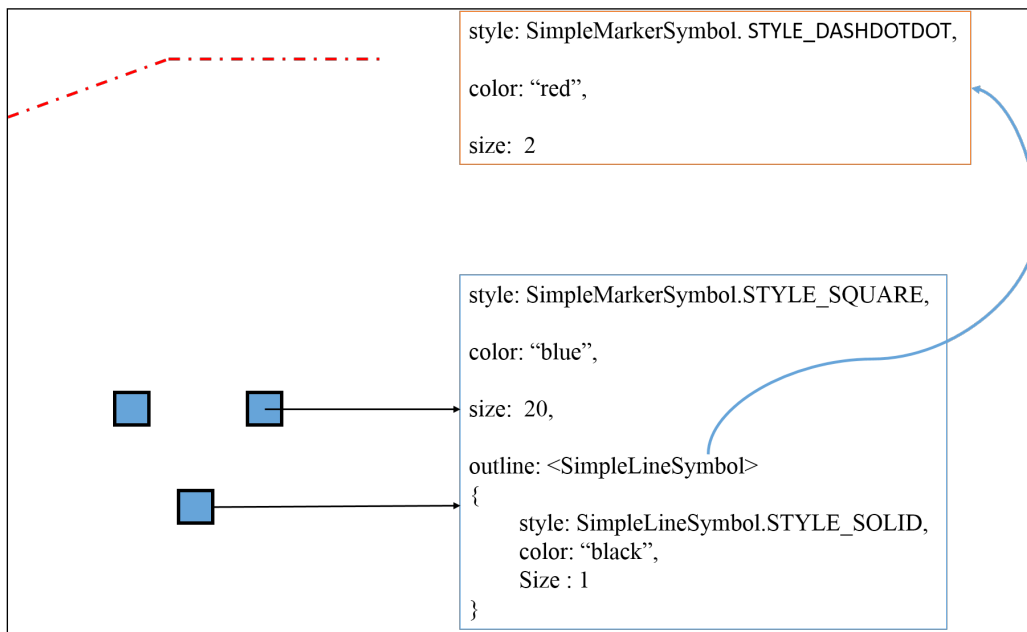
We can set the color of the line using the `simpleLineSymbol.setColor(color)` method; here, `color` is Esri Color object, and `simpleLineSymbol` is an instance of `SimpleLineSymbol` object. The style constant can be set using the `setStyle(style)` method. `SimpleLineSymbol.toJson()` is an important method that converts a `SimpleLineSymbol` to an ArcGIS Server JSON representation.

The following code snippet will create a solid red line:

```
var simpleLineSymbol = new SimpleLineSymbol();
var color = new Color("red");
simpleLineSymbol.setColor(color);
simpleLineSymbol.setWidth(2);
```

SimpleMarkerSymbol

The `SimpleMarkerSymbol` method is used to symbolize a point. Symbolizing a point geometry has an extra layer of complexity than symbolizing a line in that it accepts an outline parameter which in itself is a `SimpleLineSymbol` object.



Name	Value
Module name	esri/symbols/SimpleMarkerSymbol
Constructor:	<code>new SimpleMarkerSymbol(style, size, outline, color)</code>

The following style constants are provided by the module:

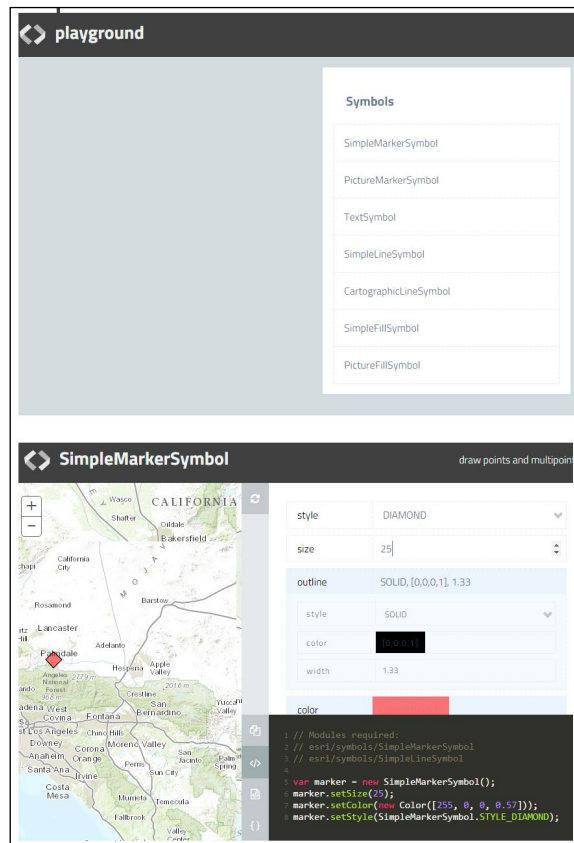
- `STYLE_CIRCLE`
- `STYLE_DIAMOND`
- `STYLE_SQUARE`

The `setAngle (angle)` method rotates the symbol clockwise around its center by a specified angle. The `setColor (color)` method sets the symbol color. `setOffset (x and y)` sets the x and y offsets of a marker in screen units. `setOutline (outline)` sets the outline of the marker symbol. `setSize (size)` lets us set the size of a marker in pixels. `setStyle (style)` sets the marker symbol style. `toJson ()` converts objects into their ArcGIS Server JSON representation.

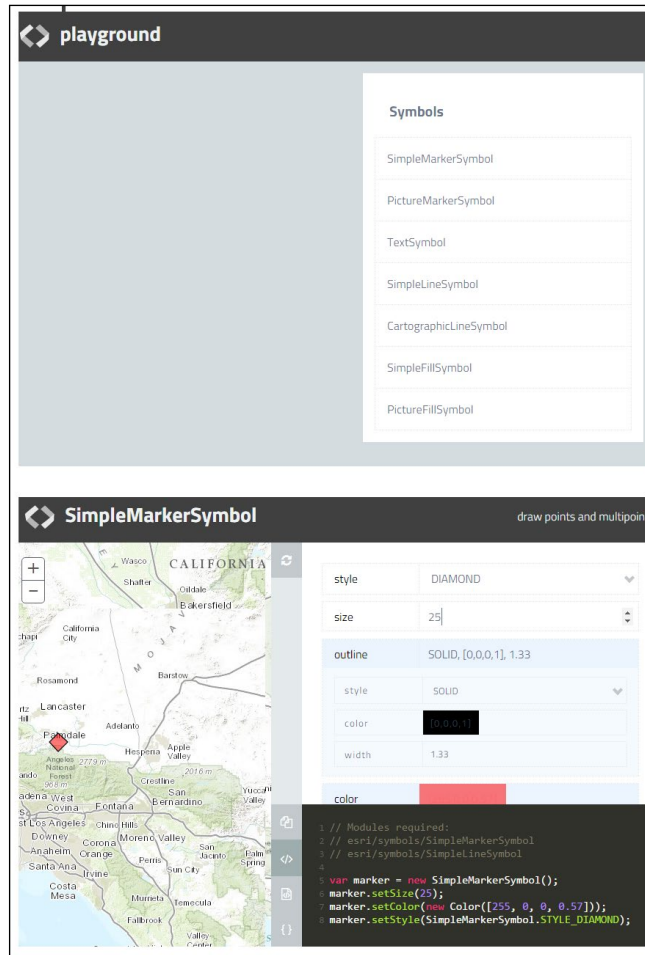
ArcGIS symbol playground

If selecting the appropriate color and style and other properties for a symbol seemed like a difficult choice, the following web page tries to help you out by providing a sandbox to generate any type of symbol and the code required to define a similar symbol in your code. The webpage is at <http://developers.arcgis.com/javascript/samples/playground/index.html>.

Navigating to this URL will land you in a page similar to the following screenshot. We can select almost any type of symbol:



Selecting one of them will navigate you to another page where you can select the properties and generate the symbology code.



Well, we easily generated the code required to generate a semi-transparent, red-colored, diamond-shaped SimpleMarkerSymbol (with no outline):

```
// Modules required:
// esri/symbols/SimpleMarkerSymbol
// esri/symbols/SimpleLineSymbol

var marker = new SimpleMarkerSymbol();
marker.setStyle(SimpleMarkerSymbol.STYLE_DIAMOND);
marker.setColor(new Color([255, 0, 0, 0.5]));
marker.setSize(25);
```

SimpleFillSymbol

The SimpleFillSymbol module helps us generate symbology for polygons.

- Module name: `esri/symbols/SimpleFillSymbol`
- `new SimpleFillSymbol(style, outline, color)`

Some of the module constants for the STYLE parameter are given here:

- `STYLE_BACKWARD_DIAGONAL`
- `STYLE_CROSS`
- `STYLE_NULL`

`SimpleFillSymbol.STYLE_SOLID` is the default styling.

PictureMarkerSymbol

When we need to picture an icon to symbolize a point geometry, we can use this module. Instead of providing the color information as a parameter, we need an image URL to display a picture as a marker symbol.

Name	Value
Module	<code>esri/symbols/PictureMarkerSymbol</code>
Constructor	<code>new PictureMarkerSymbol(url, width, height)</code>

Searching for the appropriate PictureMarkerSymbol is aided by a web page found at http://developers.arcgis.com/javascript/samples/portal_symbols/index.html.

Navigating to this URL will open a page as shown next. When a picture icon is selected, a code is generated below. This code can be reused to recreate PictureMarkerSymbology as the one selected in the web page.

The generated code is a JSON representation of PictureMarkerSymbol. The JSON object provides the following properties:

- `angle`
- `xoffset`
- `yoffset`
- `type`
- `url`
- `contentType`

- width
- height
- imageData

Among these, `imageData` and `url` are redundant, so we can avoid the `imageData` property, if we can use the URL property. The `imageData` property is just the Base64 representation of the image. To avoid this, we can uncheck a box at the top-right corner of the web page, which reads something like **Enable Base64 encoding**.

Also, if the values for `angle`, `xoffset`, and `yoffset` are 0, we can omit these too.

Select a marker symbol then copy the JSON to use in your application.

Category: Basic Enable Base64 encoding

```
var symbol = new
Symbol.PictureMarkerSymbol({"angle":0,"xoffset":0,"yoffset":12,"type":"esriPMS",
"url":"http://static.arcgis.com/images/Symbols/Basic/RedStickpin.png", "cont
entType":"image/png", "width":24, "height":24});
```

Using the URL of the icon provided by this web page and in ArcGIS Symbol Playground will enable us to further customize `PictureMarkerSymbol`.

The screenshot displays the ArcGIS Symbol Playground interface for the `PictureMarkerSymbol`. The interface is divided into three main sections: a map, a configuration panel, and a code editor.

- Map:** Shows a map of Southern California with a red stickpin marker placed over the Los Angeles area. The map includes various geographical features and city names.
- Configuration Panel:** Contains several input fields and a slider for customizing the symbol:
 - `url*`: `http://static.arcgis.com/images/Symbols/Ba`
 - `width`: `64`
 - `height`: `64`
 - `angle`: `0` (with a slider)
 - `xoffset`: `0`
 - `yoffset`: `0`
- Code Editor:** Shows the following JavaScript code:

```
1 // Modules required:
2 // esri/symbols/PictureMarkerSymbol
3
4 var marker = new PictureMarkerSymbol();
5 marker.setHeight(64);
6 marker.setWidth(64);
7 marker.setUrl("http://static.arcgis.com/images/Symbols/Basic/RedStickpin.png");
```

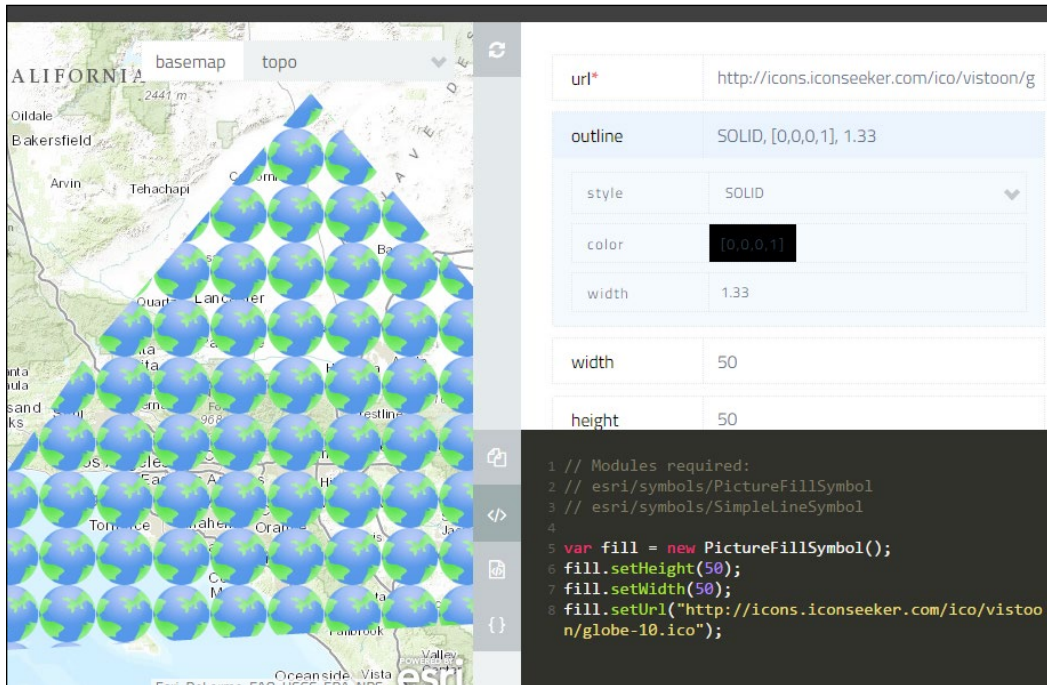
To customize `PictureMakerSymbol` use the following:

```
// Modules required:
// esri/symbols/PictureMarkerSymbol

var marker = new PictureMarkerSymbol();
marker.setHeight(64);
marker.setWidth(64);
marker.setUrl("http://static.arcgis.com/images/Symbols/Basic/RedStickpin.png");
```

PictureFillSymbol

`PictureFillSymbol` goes a step further and lets us fill a polygon geometry with an image.



url*	http://icons.iconseeker.com/ico/vistoon/g
outline	SOLID, [0,0,0,1], 1.33
style	SOLID
color	[0,0,0,1]
width	1.33
width	50
height	50

```

1 // Modules required:
2 // esri/symbols/PictureFillSymbol
3 // esri/symbols/SimpleLineSymbol
4
5 var fill = new PictureFillSymbol();
6 fill.setHeight(50);
7 fill.setWidth(50);
8 fill.setUrl("http://icons.iconseeker.com/ico/vistoon/globe-10.ico");

```

TextSymbol

Text symbols can be generated in lieu of labels. Text symbols lack geometry, so it needs to be attached to geometry.



The following snippet generated from ArcGIS Symbol Playground demonstrates the components of generating `TextSymbol`:

```
// Modules required:
// esri/symbols/TextSymbol
// esri/symbols/Font

var font = new Font();
font.setWeight(Font.WEIGHT_BOLD);
font.setSize(65);
var textSym = new TextSymbol();
textSym.setFont(font);
textSym.setColor(new Color([255, 0, 0, 1]));
textSym.setText("Sample Text");
```

Working with renderers

When an application uses layers that are referenced from a web map or a GIS service, the web map or service itself provides default drawing properties that determine how the layer will be drawn. A developer can choose to override this behavior by working with colors, symbols, and renderers to change and enhance how the features are displayed.

You can use the `setSymbol()` method to apply a symbol to a single graphic. When you want to apply symbology to all the graphics in a dynamic, feature, or a graphics layer, you can use a renderer.

Renderers make it easy to symbolize many features quickly, using either a single symbol or multiple symbols based on attribute values.

Several of the renderers available in the ArcGIS API for JavaScript are as follows:

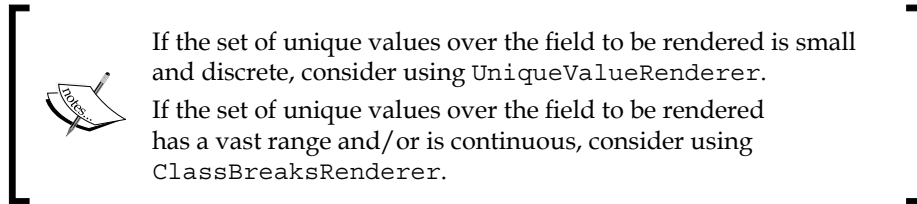
- `SimpleRenderer`: This applies the same symbol to all the graphics in a layer
- `UniqueValueRenderer`: This applies specific symbols based on the unique attribute values of each graphic
- `ClassBreaksRenderer`: This applies the symbols of different sizes or colors based on the ranges of attribute values
- `DotDensityRenderer`: This shows the variation in the spatial density of a discrete spatial phenomenon
- `HeatmapRenderer`: This converts point data into a raster display that shows the high density or weighted areas of concentration using a blur radius and the intensity value

- `TemporalRenderer`: This visualizes real-time or historic observations in the current extent of the map, factoring in relative feature aging and tracks along which observed incidents occur, such as a hurricane
- `ScaleDependentRenderer`: This applies different renderers to the same layer based on the current scale of the map

Choosing a renderer for a scenario

The symbols and renderers guide in the API documentation provides a great guide on using symbols and renderers. The documentation can be accessed at https://developers.arcgis.com/javascript/jshelp/inside_renderers.html.

`UniqueValueRenderer` and `ClassBreaksRenderer` are attribute-based renderers. This means that attribute values determine how the features are symbolized. To determine whether to use `UniqueValueRenderer` or `ClassBreaksRenderer` in a given situation, consider the nature of the field values upon which the categorization needs to be performed.



`UniqueValueRenderer` and `ClassBreaksRenderer` have the `defaultSymbol` property that gets used when a value or break cannot be matched. During development, you can use a default symbol with a high-contrast color to quickly verify whether any feature has failed to match the renderer's criteria.

Developing a Stream Gauge application

We will be developing a Stream Gauge app to demonstrate how to use the following renderers:

- Simple renderer
- Unique value renderer
- Class breaks renderer
- Heatmap renderer

The data source

The Stream Gauge data is provided by Esri as part of their Living Atlas of the World portal. This means that we need to have an ArcGIS Developer login to access the content. The URL to the MapService for the Stream Gauge data is `http://livefeeds.arcgis.com/arcgis/rest/services/LiveFeeds/StreamGauge/MapServer/`.

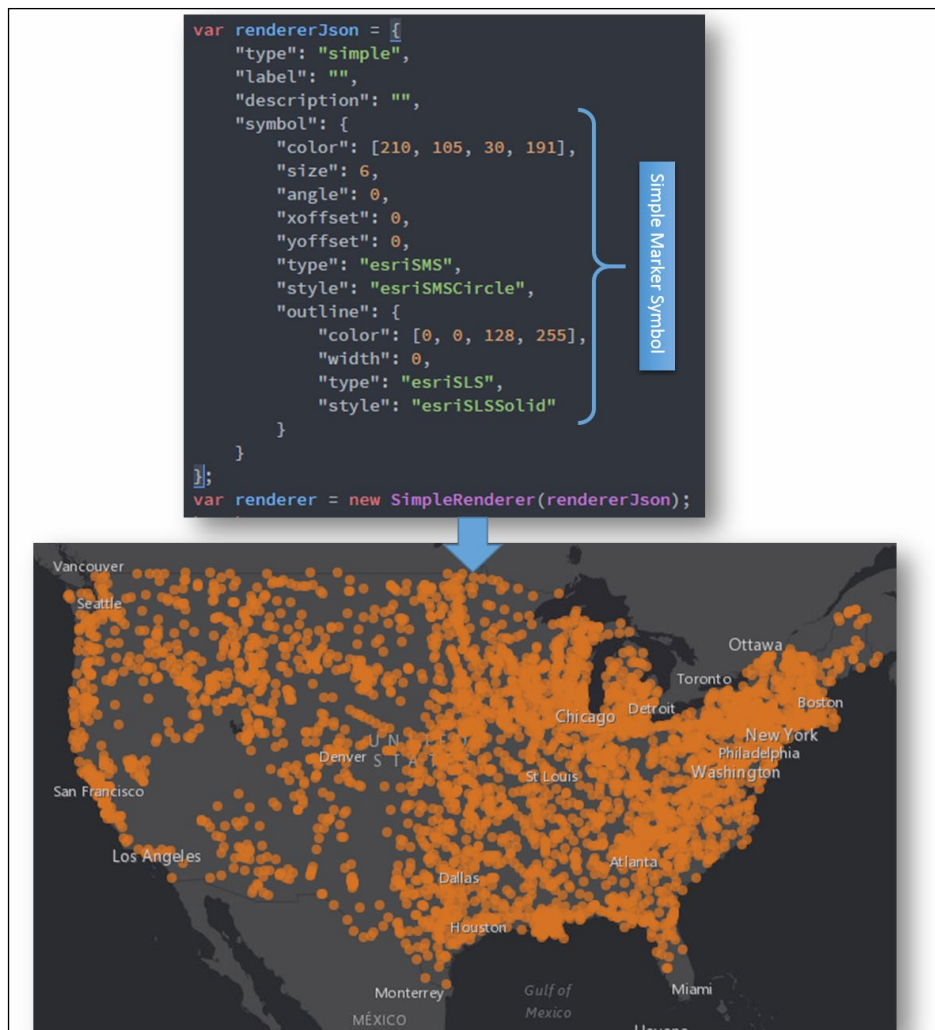
The map service provides the readings of Stream Gauges around the United States, which depict the current water level in the measured areas. The application we are trying to develop endeavors to demonstrate different rendering techniques over the Gauge data. The upcoming snapshot in the next section provides a rough rendition of our final application that we will have developed by the end of this chapter.

If you do not have an ArcGIS Developer account, refer to *Chapter 3, Writing Queries*, for instructions on how to register an account and use the credentials in the application proxy.

Simple renderer

Simple renderer is provided by the `esri/renderers/SimpleRenderer` module, and its constructor accepts any appropriate symbol or a JSON. Since all the Gauge locations are point locations, we will use `SimpleMarkerSymbol` to symbolize them.

Since we have already discussed how to construct a `PictureMarkerSymbol` from its corresponding module, we will see how to use the JSON form of the symbol. Using the JSON representation of the symbol means that we no longer need to load the modules separately for each symbol and color. The following snapshot shows how JSON is formed and used in the `SimpleRenderer` constructor:



In the preceding code, after the renderer is assigned with `SimpleRenderer`, the renderer object must be set to the feature layer by using the `setRenderer()` method. Also, the legend should be refreshed once the rendering is applied to the feature layer:

```

streamLyr.setRenderer(renderer);
streamLyr.redraw();
legend.refresh();

```

Applying unique value renderer

Unique value renderer is provided by the `esri/renderers/UniqueValueRenderer` module. Unique value renderer lets us define different symbols for a set of unique values in the data. Up to three attribute fields can be provided to determine the uniqueness of the data. Unique value renderer expects the `uniqueValueInfos` object. This object is basically a mapping between the unique value and the symbol that is used to represent the value. Therefore, all the features with a specific value will be rendered by the corresponding mapped symbol. We can provide the `defaultSymbol` object to the renderer that will be used to symbolize any value that is not defined in the `uniqueValueInfos` object. The following is a JSON representation of a unique value renderer object symbolizing unique values of flood stages. The unique values for flood stages that we symbolize are as follows:

- major
- moderate
- minor
- action

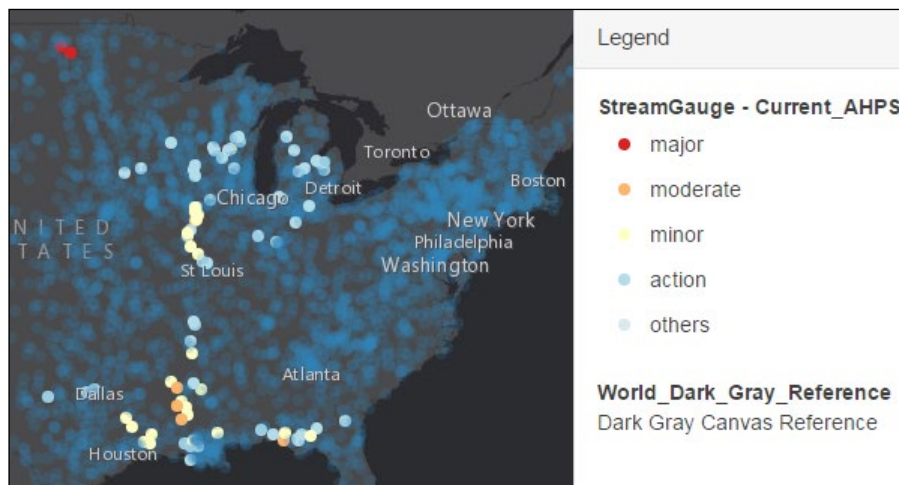
```
var rendererJson = {
  "type": "uniqueValue",
  "field1": "STAGE",
  "defaultSymbol": {},
  "uniqueValueInfos": [{
    "value": "major",
    "symbol": {
      "color": [163, 193, 163],
      "size": 6,
      "type": "esriSMS",
      "style": "esriSMSCircle"
    }
  }, {
    "value": "moderate",
    "symbol": {
      "color": [253, 237, 178],
      "size": 6,
      "type": "esriSMS",
      "style": "esriSMSCircle"
    }
  }, {
    "value": "minor",
    "symbol": {
      "color": [242, 226, 206],
      "size": 6,
```

```

        "type": "esriSMS",
        "style": "esriMSCircle"
    }
    }, {
    "value": "action",
    "symbol": {
        "color": [210, 105, 30],
        "size": 6,
        "type": "esriSMS",
        "style": "esriMSCircle"
    }
    }
    ]
};
var renderer = new UniqueValueRenderer(rendererJson);

```

The preceding code renders in the app as follows:



The following properties can be used with feature layers to render it based on multiple visual properties, such as color, rotation, size, and opacity:

Renderer method	Purpose
setColorInfo()	This shows an array of continuous values using a color ramp
setRotationInfo()	This rotates a symbol to indicate variance in direction (for example, a traveling vehicle or a hurricane event)

Renderer method	Purpose
setSizeInfo()	This changes the symbol size or width based on a range of data values
setOpacityInfo	This changes the alpha values used to display a layer

Class breaks renderer

When the field is classified and visually differentiated it is spread over a range of values, we can use `ClassBreaksRenderer`. `ClassBreaksRenderer` can be used by loading the `esri/renderers/ClassBreaksRenderer` module.

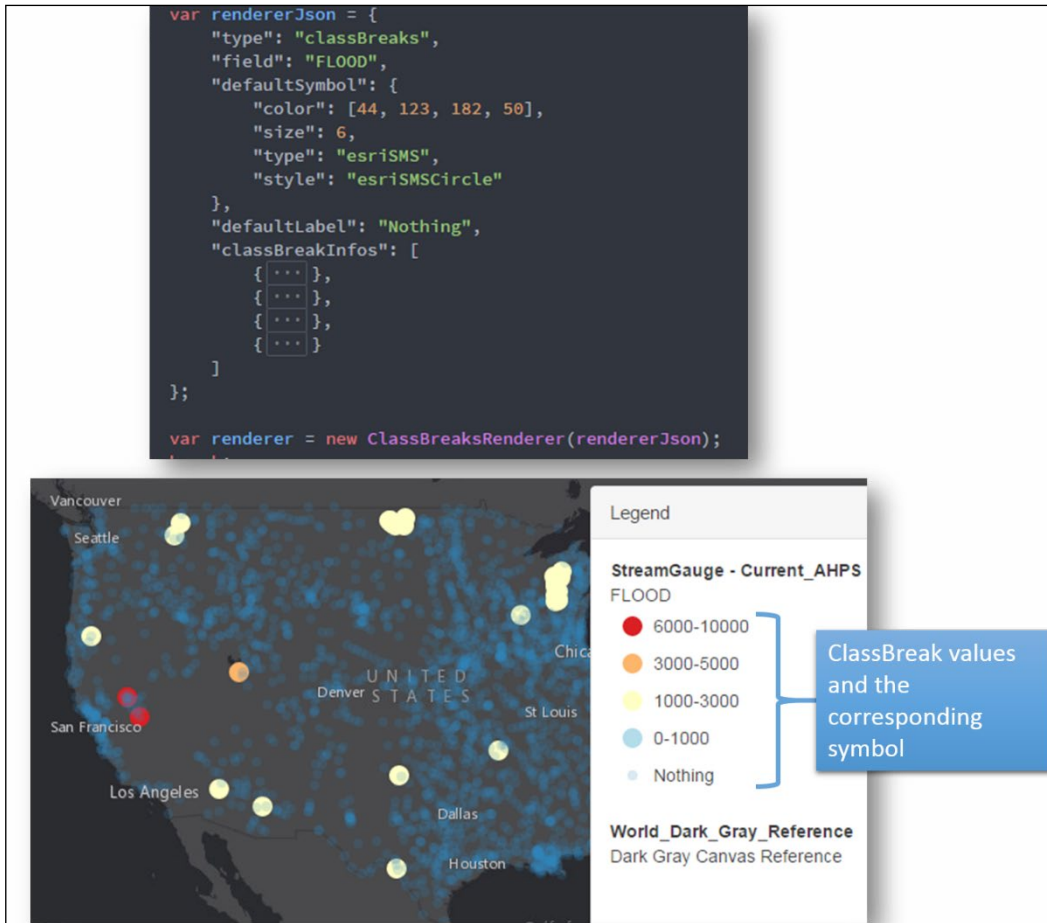
Class break renderer is very similar to unique value renderer in that the constructor for Class break renderer expects a `classBreakInfos` object, which is similar to the `uniqueValueInfos` object.

`classBreakInfos` is an array of `classBreakInfo` objects, which maps between a class range and a symbol. A class range is defined by the class' minimum (`classMinValue`) and the class' maximum (`classMaxValue`).

```
"classBreakInfos":  
[  
  {  
    "classMinValue": 0,  
    "classMaxValue": 1000,  
    "label": "0-1000",  
    "symbol": {  
      "color": [171, 217, 233],  
      "size": 12,  
      "type": "esriSMS",  
      "style": "esriSMSCircle"  
    }  
  },  
  {},  
  {},  
  ...  
]
```

Each ClassBreakInfo object has a mapping b/w a symbol and a class range (maximum - minimum)

The following snapshot shows how the `ClassBreakRenderer` JSON object is constructed with the `classBreakInfo` array and rendered on the map:

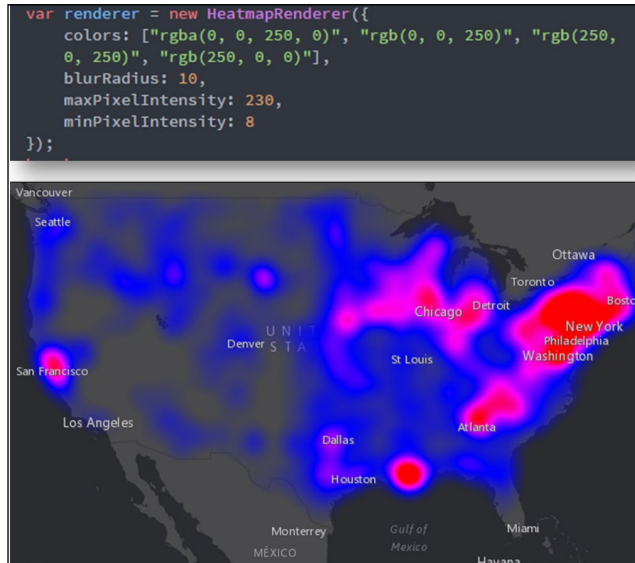


HeatmapRenderer

`HeatmapRenderer` renders point data into a raster visualization that emphasizes areas of higher density or weighted values. This renderer normal distribution curve to spread value out in vertical and horizontal directions.

This averaging function is applied horizontally and vertically to produce a blurred area of influence instead of a single specific point.

A `HeatmapRenderer` module constructor accepts an array of colors. The first color is used to represent areas with *least influence*, and the last color in the array is used to represent pixels with the highest influence. We can also define other parameters for the `HeatmapRenderer` constructor such as `blurRadius`, the maximum pixel intensity, and the minimum pixel intensity. The following snapshot of the code is used to generate a `HeatmapRenderer`:



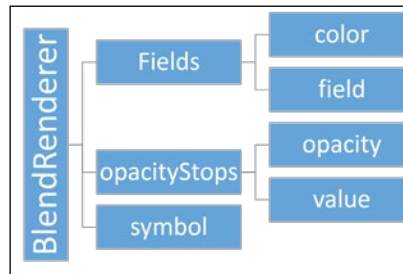
DotDensityRenderer

`DotDensityRenderer` provides the ability to create dot density visualizations of the data. A dot density map can be used to visualize the variation in the spatial density of a discrete spatial phenomenon. We can use multiple fields to visualize multiple variables on one map with different colors. For example, we can use different colors to show the distribution of various ethnic groups. The density on the map always changes as the user zooms in or out. Use `ScaleDependentRenderer` to set a unique-dot density renderer for each scale or zoom range, so `dotValue` and `dotSize` can vary across multiple scale ranges.

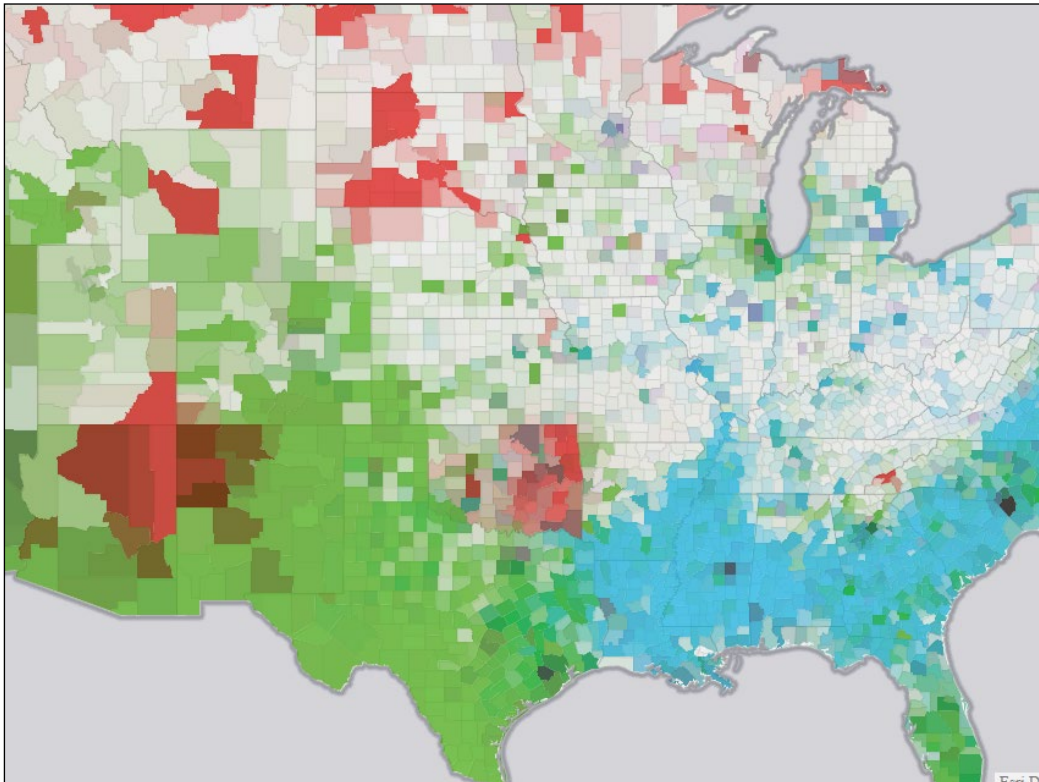
BlendRenderer

The problem with `ClassBreakRenderer` or `UniqueValueRenderer` is that you have to assign a specific color to any given value. When assigning discrete colors based on clear boundary values isn't desirable, we can use `BlendRenderer`.

`BlendRenderer` lets you do a fuzzy classification of data. It lets you assign different colors for values from different fields and use some opacity to represent the magnitude of the value. The final rendering will be a blend of these colors since we are using opacity for each field. This diagram shows how color and opacity variables can be blended to provide a rendering:

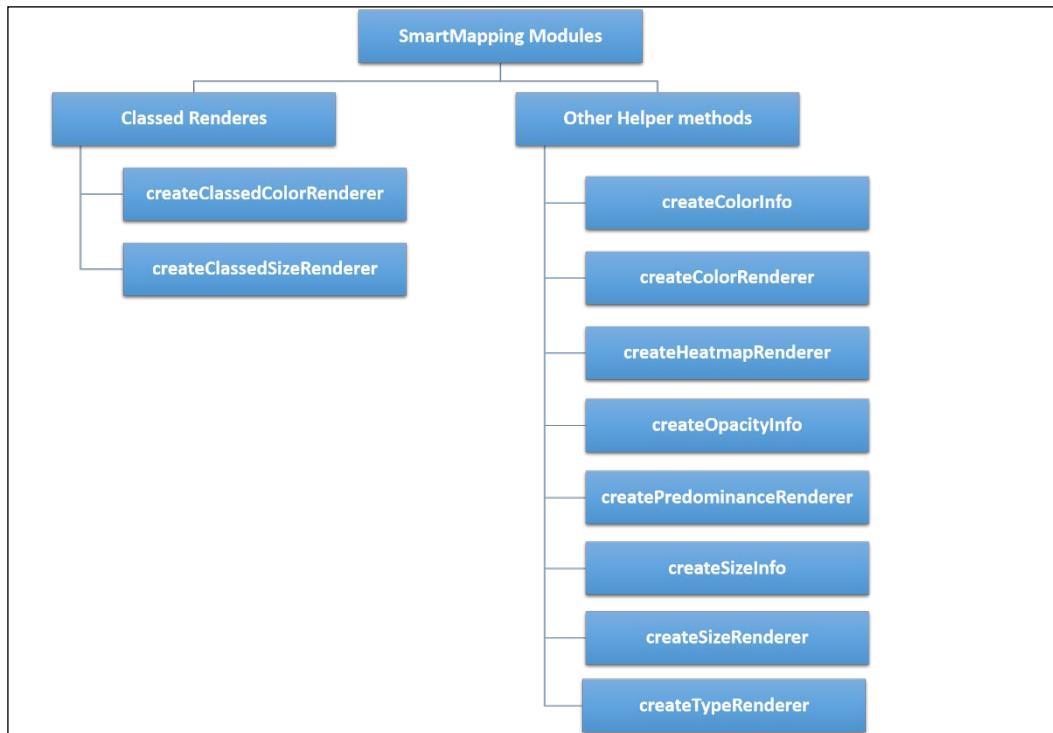



The following map shows a map of predominant minority groups throughout the United States. Such illustrations give a sense of predominant features while not completely suppressing other details:



SmartMapping

The `SmartMapping` module provides a lot of helper methods that help us choose the best rendering method. The following illustration shows a list of methods available with the `SmartMapping` module:



[ Smart Mapping Module: `esri/renderers/smartMapping`]

A classification method for classed renderers

The classed renderer helper methods, such as `createClassedColorRenderer()` and `createClassedSizeRenderer()`, need `classificationMethod` as a parameter. Choosing this value is quite important if we need to understand the significance of each.

The following classification methods are available:

- equal-interval
- natural-breaks
- quantile
- standard-deviation

The default method is equal-interval.

Equal interval classification divides the data equally into a predefined number of classes. Such a classification might not necessarily reflect the skewness in the data. For example, if the data range is from 0-1 million, and the majority of the data is concentrated between 300,000-500,000, then instead of classifying the data between 0-250,000, 250,000-500,000, 500,000-750,000, and 750,000-1,000,000, it would be a better classification scheme if there were a greater number of classification ranges between 300,000-500,000.

Classification methods such as natural-breaks, quintile, and standard deviation help to segregate the data better; hence, our data visualization technique would be statistically much accurate. This topic will be discussed in greater detail in *Chapter 7, Map Analytics and Visualization Techniques*.

Summary

This chapter gives an in-depth treatment on the topic of colors, symbols, renderers, and the situations where each can be used effectively. This chapter also dealt with the nuances of data visualization techniques along with tips and tricks to create symbols and picture marker symbols easily. We demonstrated the utility of three basic renderers: simple renderer, unique value renderer, and class breaks renderer by developing a Stream Gauge app. In the following chapters, we will be dealing with advanced visualization techniques to visually classify data on spatial as well as temporal scales.

6

Working with Real-Time Data

Data that are updated constantly presents us with a significant challenge in retrieving and rendering them. In this chapter, we will deal with two basic methods to work with real-time data by developing an application that is meant to track hurricanes. You will learn about the following topics in this chapter:

- Understand about the nature of real-time data such as hurricane data
- Use the in-built options given by ArcGIS to visualize the data
- Methods to get the latest data
- Methods to set the refresh interval for a layer

Background about the application

We are going to deal with hurricane data provided by the National Hurricane Center (NHC). The NHC provides a map service that describes the path and forecast of tropical hurricane activity. The live feeds provided as a map service by the NHC can be found at http://livefeeds.arcgis.com/arcgis/rest/services/LiveFeeds/Hurricane_Active/MapServer.

The map service provides data about the following:

- **Forecast Position**
- **Observed Position**
- **Forecast Track**
- **Observed Track**
- **The Cone of Uncertainty**
- **Watches and Warnings**
- **Tropical Storm Force**

The forecast and observed positions represent the center of the cyclone, whereas the track represents forecast and observed positions connected to give a sense of the movement of the hurricane.

[Home](#) > [services](#) > [LiveFeeds](#) > [Hurricane Active \(MapServer\)](#)

[JSON](#) | [SOAP](#)

LiveFeeds/Hurricane_Active (MapServer)

View In: [ArcGIS JavaScript](#) [ArcGIS.com Map](#) [Google Earth](#) [ArcMap](#) [ArcGIS Explorer](#)

View Footprint In: [ArcGIS.com Map](#)

Service Description:

Map Name: Hurricane Active

[Legend](#)

[All Layers and Tables](#)

[Dynamic Legend](#)

[Dynamic All Layers](#)

Layers:

- [Forecast Position](#) (0)
- [Observed Position](#) (1)
- [Forecast Track](#) (2)
- [Observed Track](#) (3)
- [Forecast Error Cone](#) (4)
- [Watches and Warnings](#) (5)
- [5-Day Wind Force Probability](#) (6)
 - [Tropical Storm Force \(34kts\)](#) (7)
 - [Strong Tropical Storm \(50kts\)](#) (8)
 - [Hurricane Force \(64kts+\)](#) (9)
 - [Raw 1/10th Degree Data \(All\)](#) (10)
- [Observed Wind Swath](#) (11)

In the **Service Catalog** heading, click **ArcGIS.com Map** to get a holistic perspective of the data in the map service.

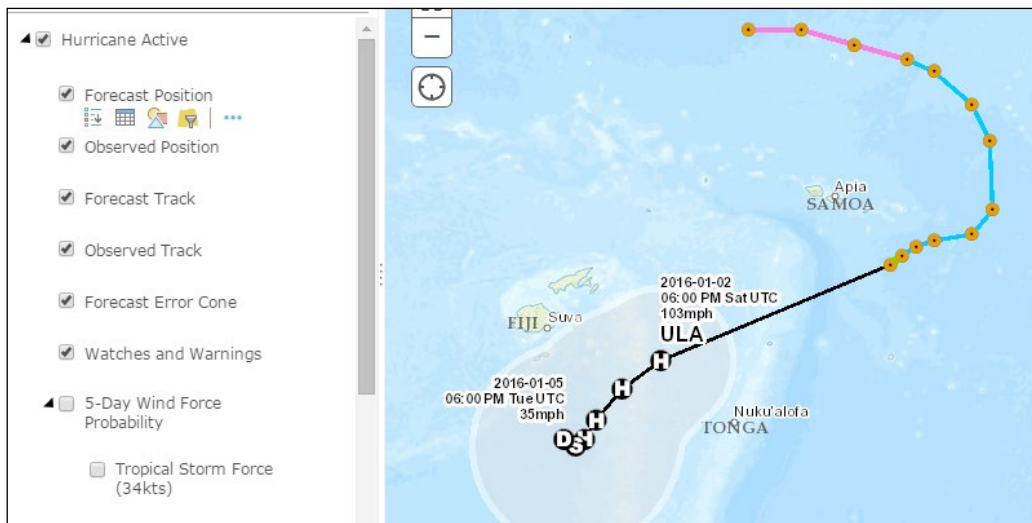
Visualizing map data

ArcGIS Online is an effective medium to visualize and play with data hosted on ArcGIS Server. When opening a map service in ArcGIS Online, the default symbology is displayed and we can get a sense of the extent of the data we would be using in our application.

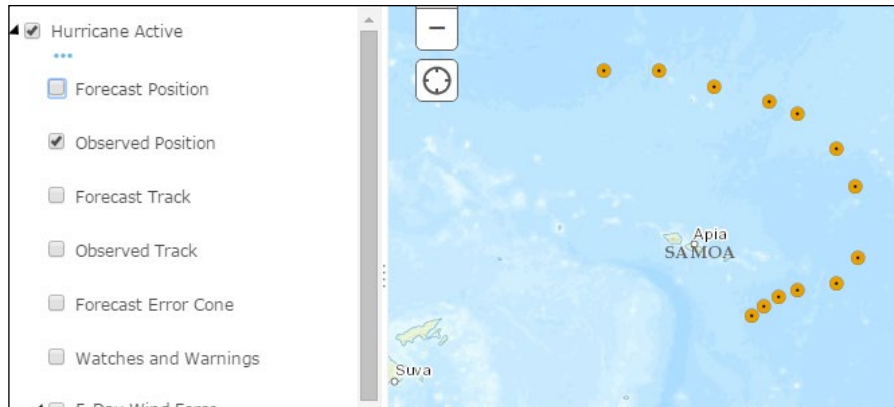
In the following screenshot, we can see the **Forecast Position** feature layer and its default symbology. The symbology being used is `PictureMarkerSymbol` and it gives a sense of what is the intensity of the hurricane over the past three days (72 hours).



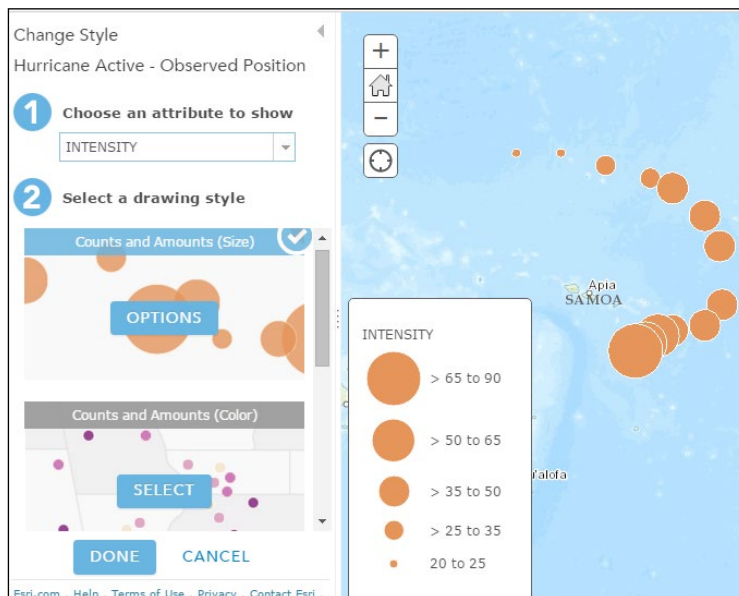
The following screenshot gives a holistic picture of the entire data in the map service including forecast locations and track, as well as observed locations:



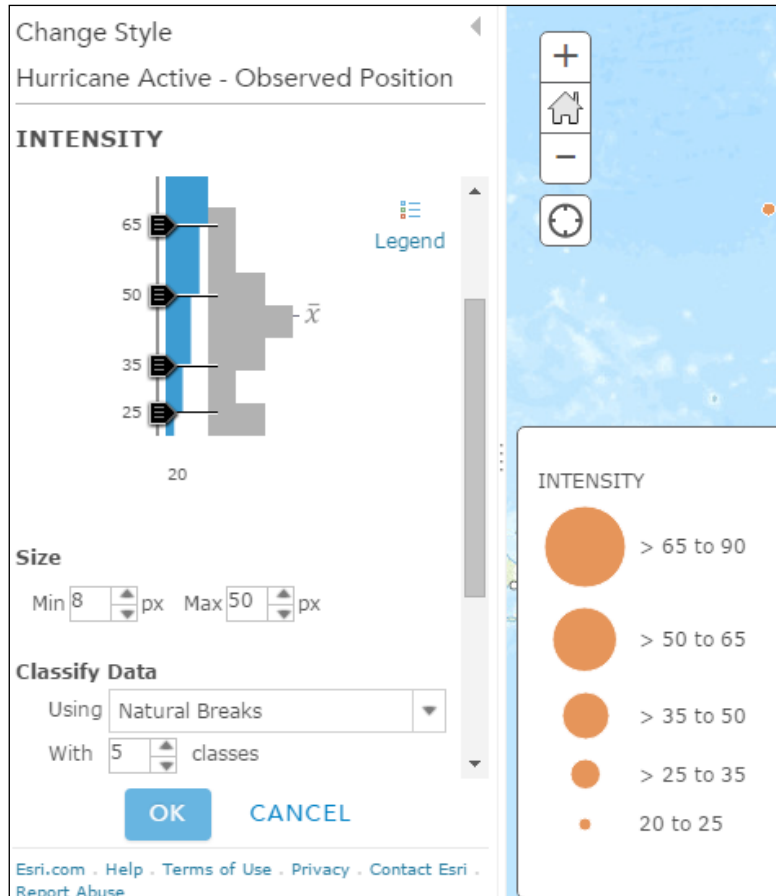
Turn off all the layers in the Table of Contents (TOC) and turn on just the **Observed Position** layer. The **Observed Position** layer is just rendered by just a simple Renderer. The symbology doesn't vary in size according to any field value. It just shows the locations where storm activity was measured in the past 72 hours.



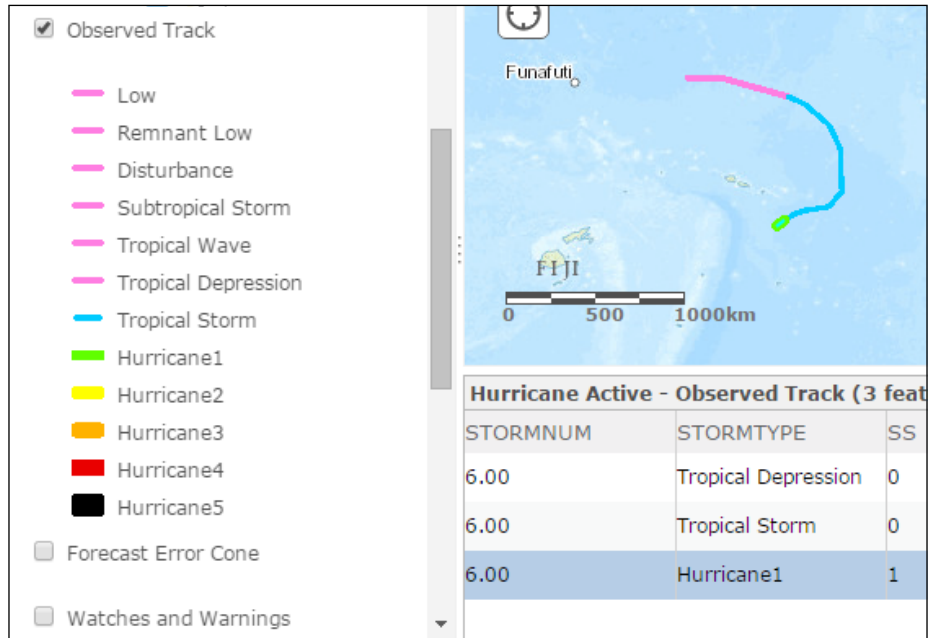
Now ArcGIS Online gives us options to set its symbology in various ways. When we click the layer's name in the TOC, the following screen opens up. It shows various styles based on which symbology can be changed. In the following screenshot, **INTENSITY** of the storm is chosen as the field of display, and the size of the symbol is based on the quantity of the **INTENSITY** value:



The data can be classified into groups according to various classification techniques such as **Equal Breaks**, **Quantile**, **Natural Breaks**, and so on.



Finally, the **Observed Track** actually shows the track taken by the hurricane over the past 72 hours and uses a unique value renderer to render the data.



Building a hurricane tracking app

Now that we have gained an understanding of our data using the ArcGIS Online service, we can use the map service URL to build a web mapping application of our own. In our application, we intend to incorporate the following:

- Add layers to the map that displays the past and present hurricane locations
- Add global wind data
- Add a gauge widget to display the wind speed
- Add a current weather widget, which displays the current weather information at the user's browser location
- Add a **Current Hurricane List** widget, which shows the updated list of current hurricanes and the details of each hurricane when selected

Symbolizing active hurricane layers

We have more than one feature layer to deal with. Let's try to build a layer dictionary. In the following code snippet, we will try to create an array of objects where each object has properties such as a URL and title. The URL refers to the URL of a feature layer and the title property refers to the title by which we would like to refer to the feature layer:

```
var windDataURL = "http://livefeeds.arcgis.com/arcgis/rest/services/  
LiveFeeds/NOAA_METAR_current_wind_speed_direction/MapServer";  
  
var activeHurricaneURL = "http://livefeeds.arcgis.com/arcgis/rest/  
services/LiveFeeds/Hurricane_Active/MapServer";  
  
var layerDict = [  
    {  
        title: "Forecast Error Cone",  
        URL: activeHurricaneURL + "/4"  
    },  
    {  
        title: "Forecast Tracks",  
        URL: activeHurricaneURL + "/2"  
    },  
    {  
        title: "Observed Track",  
        URL: activeHurricaneURL + "/3"  
    },  
    {  
        title: "Watches and Warnings",  
        URL: activeHurricaneURL + "/5"  
    },  
    {  
        title: "Forecast Positions",  
        URL: activeHurricaneURL + "/0"  
    },  
    {  
        title: "Past Positions",  
        URL: activeHurricaneURL + "/1"  
    },  
    {  
        title: "Wind Data",  
        URL: windDataURL + "/0"  
    }  
];
```

This helps us retrieve the feature layer using the layer name or title property. Let's use the `array.map()` method provided by the `dojo/_base/array` module to add the corresponding feature layer for each object into the `layerDict` array. The `array.map()` method, if you can recollect from *Chapter 1, Foundation for the API*, actually iterates through the elements in the array and will return an array. Then, each item being iterated can be modified. In our case we are trying to do the following for each item:

1. Create a feature layer from the URL in each item.
2. Add the feature layer to the map.
3. Add an additional layer property to each item object in the `layerDict` array.

The following code snippet explains the process:

```
var layerDict = array.map(layerDict, function (item) {
    var featLayer = new FeatureLayer(item.URL, {
        mode: FeatureLayer.MODE_ONDEMAND,
        outFields: ["*"]
        //infoTemplate: infoTemplate
    });
    map.addLayer(featLayer);
    item.layer = featLayer;
    return item;
});
```

Now each object in the `layerDict` array will have an additional layer property, which holds the feature layer referred by the URL.

To retrieve a feature layer, we can use the layer name in the `array.filter()` method provided by the `dojo/_base/array` module. The `filter` method() iterates through each object item and returns a filtered array based on our predicate condition.

The following line of code returns the feature layer with the title "Forecast Error Cone" and saves it in the variable named `foreCastErrorConeFeatureLayer`:

```
var foreCastErrorConeFeatureLayer = array.filter(layerDict, function
(item)
{
    return item.title == "Forecast Error Cone";
})[0].layer;
```

We are trying to symbolize the features in some of the feature layers. We will start off with the past positions. The past positions feature a layer, which by default is represented by a circle with a dot in the center. We will try to use a red flag to symbolize it. The following approach shall be taken to symbolize it:

1. Import the `esri/symbols/PictureMarkerSymbol` module.
2. Find the URL for a PNG representing a red flag and use it to create a `PictureMarkerSymbol`.
3. Import the `esri/renderers/SimpleRenderer` module and create a `SimpleRenderer` assigning the symbol for the renderer with the `PictureMarkerSymbol` we just created.
4. Set the renderer for the feature layer with the simple renderer we just created.

The following lines of code explain this process clearly:

```
var pastPositionLayer = array.filter(layerDict, function (item) {
    return item.title == "Past Positions";
})[0].layer;

var pastPositionSymbol = new PictureMarkerSymbol({
    "angle": 0,
    "type": "esriPMS",
    "url":
    http://static.arcgis.com/images/Symbols/Basic/RedFlag.png",
    "contentType": "image/png",
    "width": 18,
    "height": 18
});

var pastPositionRenderer = new SimpleRenderer(pastPositionSymbol);
pastPositionLayer.setRenderer(pastPositionRenderer);
```

Now, we can try and render the forecast error cones layer. Forecast error cones are polygon feature layers that represent the uncertainty in the forecast predictions. Two polygon features are present for each hurricane type. One of the polygon represents a 72-hour forecast error polygon and the other represents a 120-hour forecast error polygon. This information is available in the `FCSTPRD` field in the feature layer.

Let's create a unique value renderer and symbolize each of these types of polygon differently based on the value of the FCSTPRD field name. To create a unique value renderer, we need to take the following approach:

1. Import the `esri/renderers/UniqueValueRenderer`, `esri/symbols/SimpleLineSymbol` and `esri/symbols/SimpleFillSymbol` modules.
2. Create a default symbol for the renderer. Since we know that with all our Forecast Error polygons the FCSTPRD field value will be either 72 or 120, we will create a `SimpleFillSymbol` with empty symbology and also set its outline as null line symbol.
3. Create a `UniqueValueRenderer` object from the `esri/renderers/UniqueValueRenderer` module. Assign it the default symbology we just created as well as the FCSTPRD as the fieldname upon which the rendering is based.
4. Add values to the renderer using the `addValue()` method. The `addValue()` method accepts the unique value (72 /120) and the corresponding symbol for each unique value.
5. Set the renderer to the Forecast Error Cone Feature layer.

```
//Get the Forecast Error Cone feature layer
var forecastErrorConeFeatureLayer = array.filter(layerDict,
function (item) {
    return item.title == "Forecast Error Cone";
})[0].layer;

//Create a Null SimpleFillSymbol
var defaultSymbol = new SimpleFillSymbol().
setStyle(SimpleFillSymbol.STYLE_NULL);

//With a null Line Symbol as its outline
defaultSymbol.outline.setStyle(SimpleLineSymbol.STYLE_NULL);

var renderer = new UniqueValueRenderer(defaultSymbol, "FCSTPRD");

//add symbol for each possible value
renderer.addValue('72', new SimpleFillSymbol().setColor(new
Color([255, 0, 0, 0.5])));
renderer.addValue('120', new SimpleFillSymbol().setColor(new
Color([255, 255, 0, 0.5])));

//Set Renderer
forecastErrorConeFeatureLayer.setRenderer(renderer);
```

We have tried symbolizing a feature layer with `PictureMarkerSymbol` and render it using `SimpleRenderer`. For another feature layer, we used a unique value renderer to render features having different values for a particular field differently. Now let's try a special kind of symbology known as `CartographicLineSymbol`.

The `CartographicLineSymbol` provides additional properties such as `cap` and `join`, which defines how the end cap of the lines and the edge joins are rendered. To know more about these two properties, visit the API page at <https://developers.arcgis.com/javascript/jsapi/cartographiclinesymbol-amd.html>.

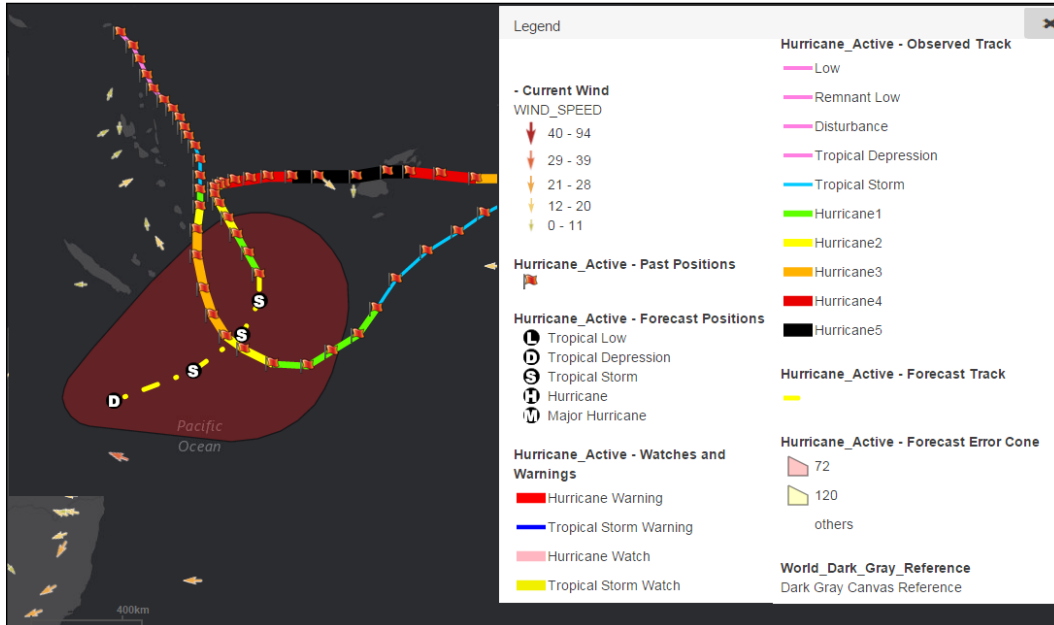
We would like to use the `CartographicLineSymbol` to symbolize the forecast track feature layer. The following shows us how to use the symbol and to render the particular feature layer:

1. Import the `esri/symbols/CartographicLineSymbol` module.
2. Use `STYLE_DASHDOT` for the style parameter, yellow for the color parameter, 5 as the width in pixels, `CAP_ROUND` as the cap type, and `JOIN_MITER` as the join type.
3. Use the symbol for a `SimpleRenderer`.
4. Set the renderer to the forecast track feature layer.

The following snippet codifies the previous approach:

```
var lineSymbol = new CartographicLineSymbol(  
    CartographicLineSymbol.STYLE_DASHDOT,  
    new Color([255, 255, 0]), 5,  
    CartographicLineSymbol.CAP_ROUND,  
    CartographicLineSymbol.JOIN_MITER, 5  
);  
var CartoLineRenderer = new SimpleRenderer(lineSymbol);  
  
forecastTrackLayer.setRenderer(CartoLineRenderer);
```

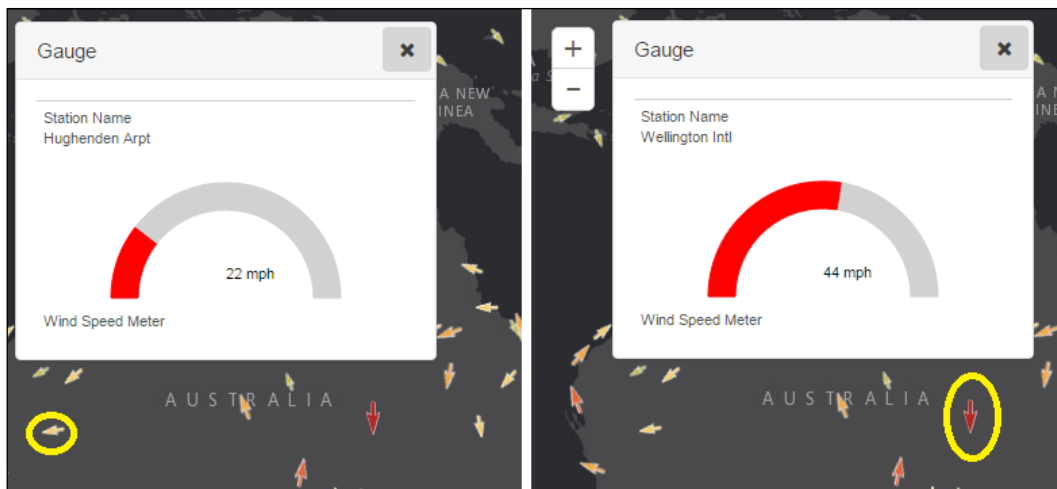

Our map looks like the following when the previous renderers are applied to the past positions layer, **Forecast Track**, and the **Forecast Error Cone** layers:



Adding a global wind data gauge

Global wind data is also a map service provided by the ArcGIS livefeeds, providing global-level wind data at various locations. Our objective is to incorporate a gauge widget that changes its gauge reading based on the wind location being hovered upon. The wind data has been appropriately symbolized by default.

The following screenshot shows a gauge widget based on our global wind data. The arrows in the map are wind feature locations, the direction of the arrow represents the direction of the wind, and the color and size of the arrow represents the speed of the wind. The gauge reading in the two instances represents the feature being hovered upon (which is highlighted by a thick yellow circle).



The URL for the wind data has been provided in one of our earlier snippets and has been added to the `layerDict` array:

```
var activeHurricaneURL = "http://livefeeds.arcgis.com/arcgis/rest/
services/LiveFeeds/Hurricane_Active/MapServer";
```

Since this URL has been added to the `layerDict` array already, we can go ahead and create a feature layer representing the wind data from its title "Wind Data":

```
var windFeatureLayer = array.filter(layerDict, function (item) {
    return item.title == "Wind Data";
})[0].layer;
```

Let's now add a gauge widget that can harness the data from this layer. The gauge is provided by an Esri dijit (dojo widget) named `esri/dijit/Gauge`. The gauge constructor is very simple. It accepts a `GaugeParameter` object and the container dom ID.

The `GaugeParameter` object needs to be constructed by us. Keep the following in mind before creating the `GaugeParameter` object:

1. The `layer` property accepts the reference to the feature layer it represents.
2. The `dataField` property indicates which field shall be used to get the gauge reading.
3. The `dataFormat` property accepts two values — `value` or `percent`. When `percent` is chosen, the maximum value of the gauge is automatically calculated and the gauge reading is shown as a percentage of the maximum value. When the `dataFormat` value is chosen as `value`, the actual value of the feature being hovered upon is shown as the gauge reading.

4. The `dataLabelField` property can be used to represent the station name or any other ancillary property about the feature being hovered upon, which can identify the feature. This shall be clubbed with the `title` property, which represents what the `dataLabelField` property represents.
5. The `color` property lets us set the color of the gauge reading.
6. If `value` is chosen as the `dataFormat` value, we need to provide a value for the `maxDataValue` property as well.

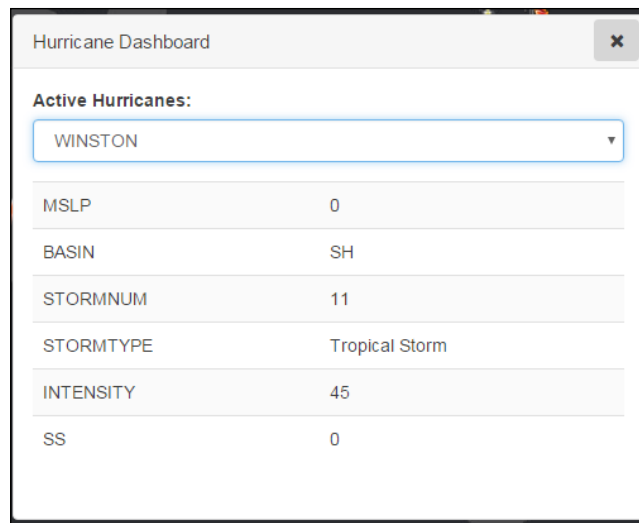
The following code is the one we used to create the wind gauge widget you saw in the previous screenshot:

```
var windGaugeParams = {
    caption: "Wind Speed Meter",
    dataFormat: "value",
    dataField: 'WIND_SPEED',
    dataLabelField: "STATION_NAME",
    layer: windFeatureLayer,
    color: "#F00",
    maxDataValue: 80,
    title: 'Station Name',
    unitLabel: " mph"
};
var windGauge = new Gauge(windGaugeParams, "gauge");
windGauge.startup();
```

Tracking the latest active hurricanes

Let's create a widget to track the latest active hurricanes. We already have all the layers representing the active hurricanes positions. Our objective is to get all the latest positions of active hurricanes and display it in a widget.

The following screenshot shows how our widget would look after development:



The dropdown in the widget lists the names of all the prevalent active hurricanes. The following grid displays the details of the selected hurricane.

The following thought process has been incorporated into the development of this widget:

1. Use a cache-bust query to get the unique list of storm names and fill the dropdown with this list.
2. On selection change of the dropdown, get the latest feature for the selected storm.
3. Populate the details of the selected storm in the widget.
4. Get the updated details for every 30 seconds.

Getting a unique list of storms

To get the unique values in our data, the Query object has a property known as `returnDistinctValues`, the value for which should be a Boolean `true`.

The following snippet explains the usage of the property:

```
query.returnDistinctValues = true;
```

Also the `outFields` property of the Query object should only list those fields for which the unique values are required. In our case, the fieldname is `STORMNAME`. Refer to the following snippet to understand this:

```
query.outFields = ["STORMNAME"];
```

To get updated results every time, we need to avoid cached query results. So instead of using a truthy expression such as `1=1`, we may need to use a pattern that reads something like:

```
"random_number = random_number".
```

This will help us get non-cached query results. Non-cached query results ensure that we are viewing the latest data within a set time period. Let's write a function that can create such a query string:

```
var _bust_cache_query_string: function () {  
    var num = Math.random();  
    return num + "=" + num;  
}
```

We can now use this function every time we need to assign a value for the `where` property of the Query object:

```
query.where = this._bust_cache_query_string();
```

When using the `returnDistinctValues` property in the Query object, we need to set the `returnGeometry` property to a Boolean `false`. The following line of code explains how to form the Query task and Query object, and how to use the result from the query to populate the drop-down box. At the end of the code we would call a `._update_hutticane_details()` method. This method fetches the latest details about the selected StormName:

```
events: function () {  
    //initialize query task  
    var queryTask = new QueryTask("http://livefeeds.arcgis.com/arcgis/  
rest/services/LiveFeeds/Hurricane_Active/MapServer/1");  
  
    //initialize query  
    var query = new Query();  
    query.returnGeometry = false;  
    query.where = "1=1 AND " + this._bust_cache_query_string();  
    query.outFields = ["STORMNAME"];  
    query.returnDistinctValues = true;  
    var that = this;  
  
    queryTask.execute(query, function (result) {  
        console.log(result);  
  
        var i;  
        //Remove all existing items
```

```

    for (i = that.cbxactiveHurricane.options.length - 1; i >= 0;
        i--) {
        that.cbxactiveHurricane.remove(i);
    }
    //Fill n the new values
    array.forEach(result.features, function (feature) {
        console.debug(feature.attributes.STORMNAME);
        that.cbxactiveHurricane.options[that.cbxactiveHurricane.options.
            length] = new Option(feature.attributes.STORMNAME, feature.attributes.
                STORMNAME);
    });
    that._update_hutticane_details();
});

this.updateTimmer = setInterval(lang.hitch(this, this._update_
    hutticane_details), 30000);
}

```

In the previous lines of code, observe the last three lines. We are using a timer function that calls the `_update_hutticane_details()` every 30 seconds. This is the function that fetches the latest details about the hurricane.

Fetching the latest data and displaying on the grid

When we tried to construct the Query object in the previous snippet, we used the `returnDistinctValues` property to get the distinct values based on field names. Now we will use the `orderByFields` property of the Query object to order the features based on a field name. To get the latest features first, the fieldname should represent a time field. In our case the field name is `DTG`. To ensure that we get the latest time as the first feature of our query result, we can use the following line of code while constructing the query object. The `orderByField` accepts a string array, each item mentioning the field name to be ordered upon and whether the ordering should be ascending (`ASC`) or descending (`DESC`). The default order is ascending:

```
query.orderByFields = ["DTG DESC"];
```

The following lines of code demonstrate how the required Query object is constructed and how the result is used to populate information about the latest storm in the widget:

```

_update_hutticane_details: function () {
    var selected_hurricane = this.cbxactiveHurricane.value;

```

```
var queryTask = new QueryTask("http://livefeeds.arcgis.com/arcgis/
rest/services/LiveFeeds/Hurricane_Active/MapServer/1");
var query = new Query();
query.returnGeometry = true;
query.where = "STORMNAME='"+ selected_hurricane +"' AND " +
this._bust_cache_query_string();
query.outFields = ["*"];
query.orderByFields = ["DTG DESC"];
var that = this;
queryTask.execute(query, function (result) {
  console.log(result);
  if (result.features.length>0){
    that._mslp.innerHTML = result.features[0].attributes.MSLP;
    that._basin.innerHTML = result.features[0].attributes.BASIN;
    that._stormnum.innerHTML =
result.features[0].attributes.STORMNUM;
    that._stormtype.innerHTML =
result.features[0].attributes.STORMTYPE;
    that._intensity.innerHTML =
result.features[0].attributes.INTENSITY;
    that._ss.innerHTML = result.features[0].attributes.SS;
  }
});
}
```

Notice the where clause in the previous chunk of code. We are selecting only the details of StormName that we have selected from the drop-down box, as well as using the cache-busting function to get the latest data:

```
query.where = "STORMNAME='"+ selected_hurricane +"' AND " + this._
bust_cache_query_string();
```

Refreshing feature layer

The feature layer displaying time data may need to refresh at various intervals. We can use feature layers to refresh an interval property to set this:

```
featureLayer.refreshInterval = 5; // in minutes
```

This is in addition to the cache-busting techniques we dealt with earlier.

Creating a weather widget

We will try to create a weather widget in our application, which displays the current weather conditions at the user's location. The user's location actually means the browser's location as recognized by the Geolocation API in modern browsers. When the browser is unable to find the user's location, we will try to find the weather data for the center of the map. Creating a weather widget presents us with the following opportunities as well as challenges:

- Weather data is continuously updated in real time and is a spatio-temporal phenomenon, meaning something that changes with place and time
- It presents us with an opportunity to use an external weather API, which is a non-ArcGIS based data
- It presents us with an opportunity to explore the client-side geometric operations such as buffer and converting between Geographic and Web Mercator coordinates

The open weather API

We need to find a data source to fetch the latest weather data. Fortunately, the open weather API is a simple and free option to fetch weather data in different formats. Paid plans provide greater usage levels. For our purposes the free version works splendidly.

The API provides REST endpoints, which provides access to the following kinds of data:

- Current weather data
- 5-day/3-hour forecast
- 5-day/3-hour forecast
- Historical data
- UV Index
- Weather map layers
- Weather stations

We will be using the current weather data endpoint to fetch the weather details for a given location.

To access the API you need to sign up for an API key. The following URL explains how to get an appid and use it in the REST queries: <http://openweathermap.org/appid#get>.

The base URL we would be using would be this:

```
var url = "http://api.openweathermap.org/data/2.5/weathers";
```

We will be providing the latitude and longitude values to issue the request to the open weather API. We have tried to make the HTTP GET request using the `esriRequest` object for which the `esri/request` module needs to be imported. The following snippet explains how the `esriRequest` object was constructed:

```
var request = esriRequest({
  // Location of the data
  url: this.url + '?lat=' + this.lat + '&lon=' + this.lon +
    '&appid=' + this.apikey,

  handleAs: "json"
});
```

If you observe the URL being constructed, it required three parameters, namely `lat`, `lon`, and `appid`.

The `appid` parameter accepts the application key that we generated earlier. There are two methods we are going to follow to get the latitude and longitude values:

1. If the Geolocation API is supported by the browser, get the latitude and longitude values from the browser's location.
2. If the Geolocation API is not supported by the browser, the map extent's centroid shall be projected to geographic coordinates and used to fetch the weather data for that location.

Using the Geolocation API

Using the Geolocation API is as simple as a call to the navigator object's `geolocation.getCurrentPosition()` method. The method returns a callback object, which contains the location of the browser. The following lines of code show how to call the geolocation API to get the current position of the browser:

```
getLocation: function () {
  if (navigator.geolocation) {
    navigator.geolocation.getCurrentPosition(lang.hitch(this,
      this.showPosition));
  } else {
    console.log("Geolocation is not supported by this browser.");
  }
}
```

In the previous code, the call object is a function by the name of `showPosition()`. The `showPosition()` function gets the position as the callback object. The coordinates of the position can be accessed by using the property `coords`.

Using geometry engine on input data

The `coords` object gives three properties, namely:

- `latitude`
- `longitude`
- `accuracy`

We clearly understand what the latitude and longitude are, but what is accuracy? Accuracy is the numeric quantity representing a possible error in meters with the coordinates being provided by the API. Or, in other words, the location is accurate within a circle of error. When we mention that it's a circle of error, wouldn't it be nice to visualize it on our map, so that we know the approximate location of our browser and maybe corroborate the results. We tried it; it seems pretty accurate. To create a circle of error, we took the following approach:

1. Using the latitude and longitude values to create a point geometry.
2. Use the `webMercatorUtils` provided by the API to convert the point from geographic coordinates to the Web Mercator coordinates.
3. Using the `geometryEngine` module provided by the API, create a buffer around the projected point with the buffer radius equal to the accuracy of the location.
4. Symbolize the buffer geometry using a `SimpleFillSymbol`.

The following lines of code explain the previous process clearly:

```
showPosition: function (position) {
    console.log(position);
    this.accuracy = position.coords.accuracy;
    this.lat = position.coords.latitude;
    this.lon = position.coords.longitude;

    //error circle
    var location_geom = new Point(this.lon, this.lat, new
    SpatialReference({ wkid: 4326 }));
    var loc_geom_proj =
    webMercatorUtils.geographicToWebMercator(location_geom);
    var location_buffer =
    geometryEngine.geodesicBuffer(loc_geom_proj, this.accuracy,
    "meters", false);
```

```
    console.log(location_buffer);
    var symbol = new SimpleFillSymbol().setColor(new Color([255, 0,
    0, 0.5]));
    this.map.graphics.add(new Graphic(location_buffer, symbol));
    //this.map.setExtent(location_buffer.getExtent());
    this.getWeatherData();
  }
}
```

We would be using the latitude and longitude obtained from the `showPosition()` method to fetch the weather data for that location.

Displaying the weather data in the widget

We earlier visited how we used the `esriRequest` module to issue an HTTP GET request to the weather API and request for the current weather data at the latitude and longitude provided by the browser. The request is a promise and we will use a `then` method upon the promise to resolve it.

The following chunk of code demonstrates how the `esriRequest` promise is resolved and how it is used to display the current weather data:

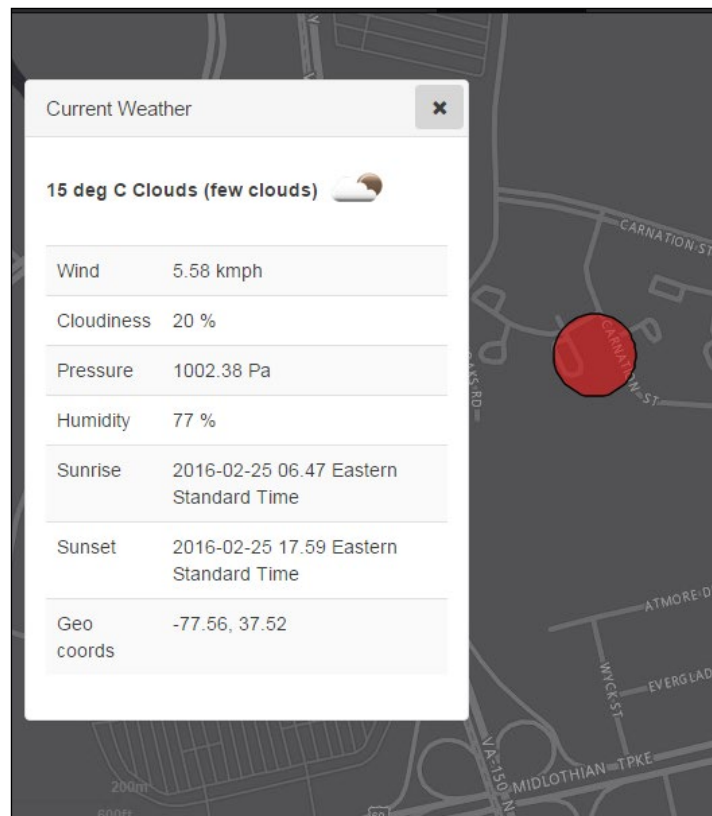
```
request.then(function (data) {
  console.log("Data: ", data);
  that.weather.innerHTML = Math.round(data.main.temp - 270) + "
  deg C " +
  data.weather[0].main + ' (' + data.weather[0].description + ')';
  var imagePath = "http://openweathermap.org/img/w/" +
  data.weather[0].icon + ".png";
  // Set the image 'src' attribute
  domAttr.set(that.weatherIcon, "src", imagePath);
  that.windSpeed.innerHTML = data.wind.speed + ' kmph';
  that.cloudiness.innerHTML = data.clouds.all + ' %';
  that.pressure.innerHTML = data.main.pressure;
  that.humidity.innerHTML = data.main.humidity + ' %';
  that.pressure.innerHTML = data.main.pressure + ' Pa';
  that.sunrise.innerHTML = that._processDate(data.sys.sunrise);
  that.sunset.innerHTML = that._processDate(data.sys.sunset);
  that.coords.innerHTML = data.coord.lon + ', ' + data.coord.lat;
}
```

In the previous code, the temperature is always returned in kelvin. So to convert it to centigrade, we need to subtract it by 270. The time conversions are being applied using the function named `_processDate()`. The time issued by the open weather API is Unix base time in UTC.

The `_processDate()` function we wrote looks like this:

```
_processDate: function (dateStr) {
  if (dateStr == null) {
    return "";
  }
  var a = new Date(dateStr * 1000);
  return dateLocale.format(a, {
    selector: "date",
    datePattern: "yyyy-MM-dd HH.mm v"
  });
}
```

The `dateLocale` object used in the previous function is a dojo module (`dojo/date/locale`), which provides localized time versions of the `date` object being dealt with. The widget looks like the following screenshot shown. The red circle is the circle of error we were talking about. We were also able to create a small weather icon, which summarizes the weather condition.



If you're curious what the HTML template for the previous widget would look like, we have one thing to say – have we disappointed you? Here it is:

```
<div>
  <form role="form">
    <div class="form-group">
      <label dojoAttachPoint="weather"></label>
      <img dojoAttachPoint="weatherIcon"/>
    </div>
  </form>
  <table class="table table-striped">
    <tbody>
      <tr>
        <td>Wind</td>
        <td dojoAttachPoint="windSpeed"></td>
      </tr>
      <tr>
        <td>Cloudiness</td>
        <td dojoAttachPoint="cloudiness"></td>
      </tr>
      <tr>
        <td>Pressure</td>
        <td dojoAttachPoint="pressure"></td>
      </tr>
      <tr>
        <td>Humidity</td>
        <td dojoAttachPoint="humidity"></td>
      </tr>
      <tr>
        <td>Sunrise</td>
        <td dojoAttachPoint="sunrise"></td>
      </tr>
      <tr>
        <td>Sunset</td>
        <td dojoAttachPoint="sunset"></td>
      </tr>
      <tr>
        <td>Geo coords</td>
        <td dojoAttachPoint="coords"></td>
      </tr>
    </tbody>
  </table>
</div>
```

The innocuous HTML template was all we needed to develop the weather widget, which we used to display the current weather data at our location.

Summary

In this chapter, we have covered in detail what constitutes real-time data and how to visualize and get the latest features. We will be dealing with how to deal with time-aware layers and how to visualize spatio-temporal layers in later chapters. Thus, we will be able to build effective web applications that are refreshed continuously. In the following chapters, we will be dealing with advanced visualization techniques using the statistical capabilities of the feature layer, and learning about charting libraries.

7

Map Analytics and Visualization Techniques

Performing analytics on map data will reveal a lot of spatial patterns that would otherwise stay hidden. The API provides a lot of methods to elicit such information using advanced statistical queries on the data. Combine this with the intuitive data visualization methods provided by the API and you're one step nearer to becoming a map data scientist. We will be building a demographics analytic portal in this chapter by first trying to understand a few basic statistical concepts, and then by practically applying those in the code with the aid of the analytic and rendering modules provided by the API. Specifically we will be covering the following topics:

- Introduction to the demographics analytic portal we're going to develop
- Introduction to basic statistical measures
- Modules provided by the API to calculate the feature statistics
- A brief introduction to the classification methods
- Code-backed explanation for developing renderers with visual variables
- Performing multivariate mapping
- Performing automatic mapping using smart mapping

Building a demographics analytic portal

We are going to build a demographic analytic portal to demonstrate the advanced analytics capabilities of the API. Demographics refers to the classification of the population living in an area based on various socio-economic factors such as age, educational attainment, nationality, median household income, race, gender, and so on. The demographic data is mostly based on Census data and other reliable sources.

The demographics can be used to perform various analytics and is equally useful to government to make policy decisions and businesses to make marketing decisions. The power of demographic data lies in performing appropriate analytics such that we can extract useful information about the population living in an area in comparison to the ones surrounding it. Let's consider this URL, which provides detailed statistics on the median household income at block level—http://demographics5.arcgis.com/arcgis/rest/services/USA_Demographics_and_Boundaries_2015/MapServer.

This map service shows the most updated 2015 demographic data for the USA. Among the hundreds of demographic parameters provided, we are interested in the median household income in the United States in 2015. Income amounts are expressed in current dollars, including an adjustment for inflation or cost-of-living increases. The median is the value that divides the distribution of household income into two equal parts. For more information on this map, including the terms of use, visit this URL: <http://doc.arcgis.com/en/living-atlas/item/?itemId=6db428407492470b8db45edaa0de44c1&subType=demographics>

These data are provided as a part of the Living Atlas endeavor of Esri. To use this data, you will require an ArcGIS Online organizational subscription or an ArcGIS Developer account. To access this item, you'll need to do one of the following:

- Sign in with an account that is a member of an organizational subscription
- Sign in with a developer account
- If you don't have an account, you can sign up for a free trial of ArcGIS or a free ArcGIS Developer account at this link: <https://developers.arcgis.com/en/sign-up/>

Basic statistical measures

Let's discuss some basic statistics so that we can utilize some of the statistical functionality provided by the API to the fullest extent. The five basic statistical parameters we may need to understand clearly before proceeding further are:

- Minimum
- Maximum
- Average
- Standard deviation
- Standardization

Minimum

As the name suggests, this implies the least value in a dataset. In our case of the block-level household income, the `minimum` statistic indicates the block with the least median household income.

Maximum

Similar to the `minimum`, the `maximum` statistic defines the maximum median household income value among all the blocks considered.

Sum

`Sum` is a simple yet effective statistic that gives us the total value of all the data being considered.

Average

An `Average` statistic defines the arithmetic mean value of all the values. An average is derived by dividing the `Sum` statistic by the count of the data values taken for the calculation.

$$\text{Average} = \text{Sum} / \text{Count}$$

Standard deviation

Standard deviation is perhaps the most important statistic that one can derive from any given data. Standard deviation is a measure of how spread out the data are or how much the data deviates from the mean or average. When we know the standard deviation, we can normally observe that:

- 68% of values are within plus or minus one times the standard deviation from the mean
- 95% of values are within plus or minus two times standard deviation from the mean
- 99.7% of values are within three times the standard deviation from the mean

This is based on the fact that most data follows the normal distribution curve. When we order the data and plot the values, the histogram looks like a bell curve.

Standardization

Knowing the concept of standard deviation and mean, we can normalize our data. This process is known as **standardization** and the statistical measure derived from the process is known as the **standard score** (*z-score*). When we have datasets with large values, standardization is an effective way to summarize the data and quantify it.

So to convert any value to a standard score (*z-score*), we need to first subtract the value from the mean, then divide by the standard deviation.

$$z\text{-score} = (\text{Value} - \text{Mean}) / \text{Standard_Deviation}$$

Statistical functionality provided by the API

Let's investigate what the API has to offer us in terms of these basic statistical measures. Later we will use these statistical measures in our application to provide better insight into the data. We will also use these techniques in our visualization techniques.

StatisticDefinition module

The API provides a module called the `StatisticDefinition` module, which can be used in conjunction with the `Query` task and `Query` modules to extract the basic statistical measures we just discussed.

Module name: `esri/tasks/StatisticDefinition`

The following are the properties used to define a statistic definition object:

- `onStatisticField`: Used to define the field on which statistics will be calculated
- `outStatisticFieldName`: The name of the output field
- `statisticType`: Used to define the type of statistic. The accepted statistic types are:
 - `min`: to get the minimum statistic
 - `max`: to get the maximum statistic
 - `sum`: the get the sum statistic
 - `avg`: to derive the average value statistic
- `stddev`: to derive the standard deviation statistic

Let's try to use these and derive these statistical measures on the demographics layer URL we just provided at the beginning of the chapter.

The following screenshot shows a code snippet and explains how these statistics are derived for the county layer in the demographics map service:

```

var queryTask = new QueryTask(BlockLevelFeatureLayerURL);
var query = new Query();
var maxStatDef = new StatisticDefinition();
maxStatDef.onStatisticField = 'MEDHINC_CY';
maxStatDef.outStatisticFieldName = 'MAX_MEDHINC_CY';
maxStatDef.statisticType = 'max';

var minStatDef = new StatisticDefinition();
minStatDef.onStatisticField = 'MEDHINC_CY';
minStatDef.outStatisticFieldName = 'MIN_MEDHINC_CY';
minStatDef.statisticType = 'min';

var stdDevStatDef = new StatisticDefinition();
stdDevStatDef.onStatisticField = 'MEDHINC_CY';
stdDevStatDef.outStatisticFieldName = 'STDDEV_MEDHINC_CY';
stdDevStatDef.statisticType = 'stddev';

var avgDevStatDef = new StatisticDefinition();
avgDevStatDef.onStatisticField = 'MEDHINC_CY';
avgDevStatDef.outStatisticFieldName = 'AVG_MEDHINC_CY';
avgDevStatDef.statisticType = 'avg';

console.log([maxStatDef, minStatDef, stdDevStatDef]);
query.returnGeometry = false;
query.where = "1=1";
query.outStatistics = [maxStatDef, minStatDef, stdDevStatDef, avgDevStatDef];
queryTask.execute(query, handleQueryResult, errorHandler);

function handleQueryResult(results) {
  if (!results.hasOwnProperty("features") || results.features.length === 0) {
    console.log('No features, something went wrong!');
    return;
  }
  var statsObj = results.features[0].attributes;
  console.log(statsObj);
  stats.Plus1StdDev = statsObj.AVG_MEDHINC_CY + 1 * statsObj.STDDEV_MEDHINC_CY;
  stats.Plus2StdDev = statsObj.AVG_MEDHINC_CY + 2 * statsObj.STDDEV_MEDHINC_CY;
  stats.Plus3StdDev = statsObj.AVG_MEDHINC_CY + 3 * statsObj.STDDEV_MEDHINC_CY;
  stats.Minus1StdDev = statsObj.AVG_MEDHINC_CY - 1 * statsObj.STDDEV_MEDHINC_CY;
  stats.Minus2StdDev = statsObj.AVG_MEDHINC_CY - 2 * statsObj.STDDEV_MEDHINC_CY;
  stats.Minus3StdDev = statsObj.AVG_MEDHINC_CY - 3 * statsObj.STDDEV_MEDHINC_CY;
  console.log(stats);
}

```

Maximum, Minimum, Average & Standard Deviation Statistic Definition

Pass it as an array for outStatistics property of the Query

This object contains all the defined statistics

Calculate +1, +2, +3 Std.Deviation as well as -1, -2 & -3 Std.Dev

The required statistics can be extracted using this simple code snippet.

The console screen should look like this after the code has been executed:

```
Object {MAX_MEDHINC_CY: 113282, MIN_MEDHINC_CY: 18549, STDDEV_MEDHINC_CY: 10960.43202775655, AVG_MEDHINC_CY: 42115.877187400576}
```

```
Object {Plus1StdDev: 53076.309215157125, Plus2StdDev: 64036.741242913675, Plus3StdDev: 74997.17327067023, Minus1StdDev: 31155.445159644027, Minus2StdDev: 20195.013131887477...}
```

The derived statistics such as Plus1StdDev, Plus2StdDev, Plus3StdDev, Minus1StdDev, Minus2StdDev, and Minus3StdDev shall be used later to render the data better.

Classification methods

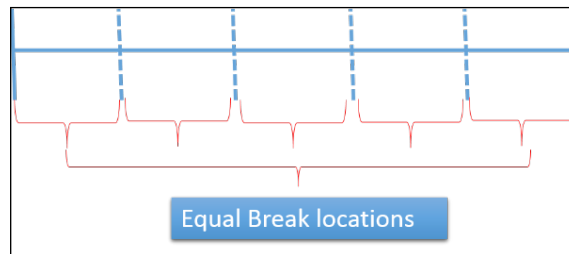
When we have a large quantity of data, we use rendering methods to classify it. We need to identify an appropriate classification method to create the class breaks. The following classification methods are supported by the API:

- Equal interval
- Natural breaks
- Quantile
- Standard deviation

Let's discuss very briefly the implications of using each classification method.

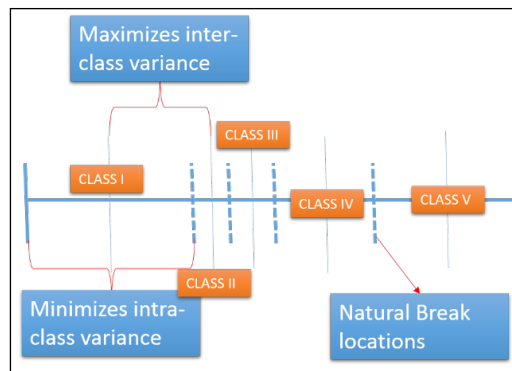
Equal interval

This classification method breaks the data into equal parts. We need to know the data range to use this classification method. This method should be used when the data is dispersed and well distributed.



Natural breaks

Natural breaks is a classification method based on the Jenks Break Algorithm. Basically, this algorithm creates more numbers of breaks at locations where data is more clustered. This is done by seeking to minimize each class's average deviation from the class mean, while maximizing each class's deviation from the means of the other groups. In other words, the method seeks to reduce the variance within classes and maximize the variance between classes.

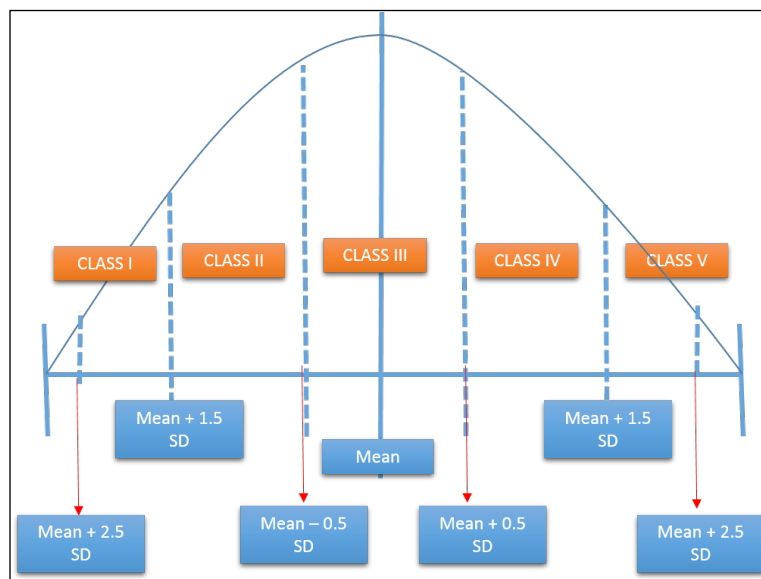


Quantile

This method classifies data such that there are an equal number of data points in each group.

Standard deviation

As discussed previously, standard deviation is a measure of how much the data deviates from the mean. Using this classification methodology, we can find how much and also where the data is within beyond three standard deviations (outliers' cases), between two and three standard deviations (higher and lower end values), and within one standard deviation from the mean.



Concept of normalization

Normalizing a data value is useful for computing a lot of things. Consider the following scenarios:

- **Case 1:** We need to symbolize how densely populated each state is. Symbolizing based on the population field would give a wrong measure or convey wrong information. All we might have to do is to divide each state's population by its geographic area to get a measure of the population density. Similarly, if we need to convey the percentage of youth population (age < 35) against the total population, we need to divide the field having the youth population by the field displaying the total population.
- **Case 2:** When trying to symbolize the income distribution of the entire world, we may encounter a large range of values. If we were to use a color or opacity renderer, some countries would be on the higher end of the spectrum, while some would be at the bottom with many in between, with lots of color information not really used up. In such scenarios, it would be more useful to display the income distribution using a logarithmic scale.
- **Case 3:** When we need to calculate the value as a percentage of totals such as crime data or number of participants from each state in a marathon, we need to divide the value by the total.

Many renderers have a `normalizationField` and `normalizationType` property to implement such normalization.

`normalizationField` lets us define the field that is used for normalization. For example, for *Case 1*, the `Area` field and `Total Population` field is the `normalizationField`.

`normalizationType` is the type of normalizing that needs to be performed on the value. The three possible values for `normalizationType` are `field`, `log`, and `percent-of-total`. For example, for *Case 1*, we need to use the `normalizationType` as `field`. For *Case 2*, we need to use `log`, and for *Case 3*, we need to use `percent-of-total` as `normalizationType`.

Feature layer statistics

In version 3.13 of the API, this plugin was introduced that could prove handy for calculating feature layer statistics. Using the feature layer statistics plugin, we can calculate the statistics for the following:

- The basic statistics on a field for a feature layer
- Class break statistics
- Unique values in a field
- Suggested scale range for viewing a layer
- Getting a sample feature
- Calculating a histogram

The plugin can be added to the feature layer using the following code snippet:

```
var featureLayerStats = new FeatureLayerStatistics({
    layer: CountyDemogrphicsLayer
});
```

In the previous snippet, `CountyDemogrphicsLayer` is the name of the feature layer to which the `FeatureLayerStatistics` plugin is being added.

The usual parameters expected by the methods used in the plugin are `field` and `classificationMethod`. The `field` plugin refers to the name of the attribute field based on which the statistic is computed. The `classificationMethod` refers to one of the classification methods discussed previously based on which statistics are computed:

```
var featureLayerStatsParams = {
    field: "MEDHINC_CY",
    classificationMethod : 'natural-breaks'
};
```

The methods on the plugins always return a promise. The following snippet calculates the basic statistical values on the field as defined in `featureLayerStatsParams`:

```
featureLayerStats.getFieldStatistics(featureLayerStatsParams).
then(function (result) {
    console.log("Successfully calculated %s for field %s, %o",
"field statistics", featureLayerStatsParams.field, result);
}).otherwise(function (error) {
    console.log("An error occurred while calculating %s, Error:
%o", "field statistics", error);
});
```


The results look like this in the browser console:

```
Successfully calculated field statistics for field MEDHINC_CY,  
Object {  
  source:"service-query",  
  min:20566,  
  max:130615,  
  avg:46193.26694241171,  
  stddev:12564.308382029049,  
  count:3143,  
  sum:145185438,  
  variance:157861845.1187254  
}
```

The previous result provided about the same or more information as that derived from the statistical definition module that we used earlier.

The following snippet calculates the class break values on the field as defined in `featureLayerStatsParams`:

```
featureLayerStats.getClassBreaks(featureLayerStatsParams).  
then(function (result) {  
    console.log("Successfully calculated %s for field %s,  
%o", "class breaks", featureLayerStatsParams["field"], JSON.  
stringify(result));  
}).otherwise(function (error) {  
    console.log("An error occurred while calculating %s, Error:  
%o", "class breaks", error);  
});
```

The beautified result looks like this:

```
{  
  "minValue": 20566,  
  "maxValue": 130615,  
  "classBreakInfos": [  
    {  
      "minValue": 20566,  
      "maxValue": 27349.802772469,  
      "label": " < -1.5 Std. Dev.",  
      "minStdDev": null,  
      "maxStdDev": -1.5  
    },  
    {
```

```
        "minValue": 27349.802772469,  
        "maxValue": 39912.112219098,  
        "label": "-1.5 - -0.50 Std. Dev.",  
        "minStdDev": -1.5,  
        "maxStdDev": -0.5  
    },  
    {  
        "minValue": 39912.112219098,  
        "maxValue": 52474.421665726,  
        "label": "-0.50 - 0.50 Std. Dev.",  
        "minStdDev": -0.5,  
        "maxStdDev": 0.5,  
        "hasAvg": true  
    },  
    {  
        "minValue": 52474.421665726,  
        "maxValue": 65036.731112354,  
        "label": "0.50 - 1.5 Std. Dev.",  
        "minStdDev": 0.5,  
        "maxStdDev": 1.5  
    },  
    {  
        "minValue": 65036.731112354,  
        "maxValue": 77599.040558982,  
        "label": "1.5 - 2.5 Std. Dev.",  
        "minStdDev": 1.5,  
        "maxStdDev": 2.5  
    },  
    {  
        "minValue": 77599.040558982,  
        "maxValue": 130615,  
        "label": "> 2.5 Std. Dev.",  
        "minStdDev": 2.5,  
        "maxStdDev": null  
    }  
],  
    "source": "service-generate-renderer"  
}
```

Working with continuous and break renderers

Continuous renderers refers to renderers that symbolize features on a continuous spectrum of values unlike unique value renderers. We need to define several `stops` or `breakpoints` for such renderers. These `stops` define a class and the renderer checks which class each value falls into. Based on the class, the data is visualized with the aid of visualization variables such as color, size, opacity, or even rotation.

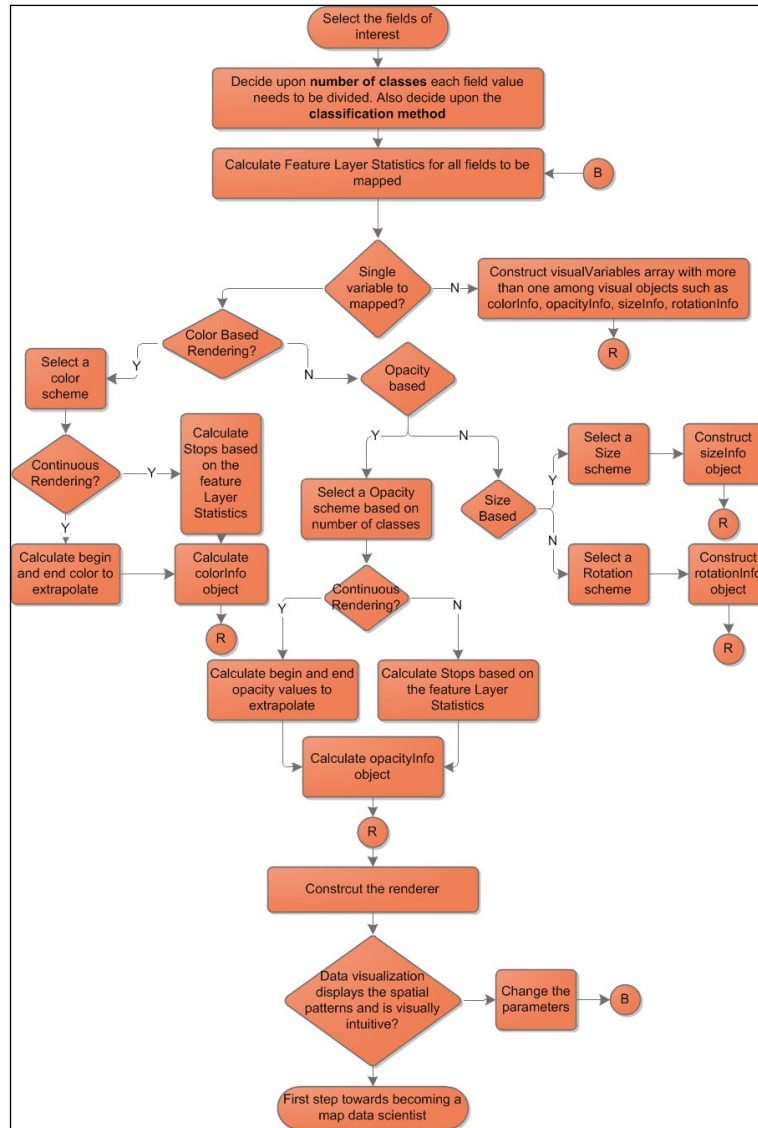
Using the statistics available, we can use the `ClassBreaksRenderer` provided by the API to create classed and continuous renderers easily. `ClassBreaksRenderer` symbolizes each graphic based on the value of some numeric attribute with different visualization.

Module name: `esri/renderers/ClassBreaksRenderer`

The setting of color, size, or opacity is enabled on this module with the aid of properties such as `colorInfo`, `opacityInfo`, and `sizeInfo`. The following methods are available on the `ClassBreaksRenderer`:

- `setColorInfo(colorInfo)`: Sets the `colorInfo` property
- `setOpacityInfo(opacityInfo)`: Sets opacity info for the renderer as defined by the `info` parameter
- `setRotationInfo(rotationInfo)`: Modifies rotation info for the renderer
- `setSizeInfo(sizeInfo)`: Sets size info of the renderer to modify the symbol size based on the data value

Let's discuss more about these in detail. The following diagram provides a brief guide to developing a renderer:



ColorInfo

ColorInfo is an object used to define the color ramp to render the layer. We only need to provide discrete sets of color values at the stops or sometimes just the color values in the ramp:

A simple ColorInfo object example is as follows:

```
renderer.setColorInfo({
  field: "MEDHINC_CY",
  minDataValue: featureLayerStats.min,
  maxDataValue: featureLayerStats.max,
  colors: [
    new Color([255, 255, 255]),
    new Color([127, 127, 0])
  ]
});
```

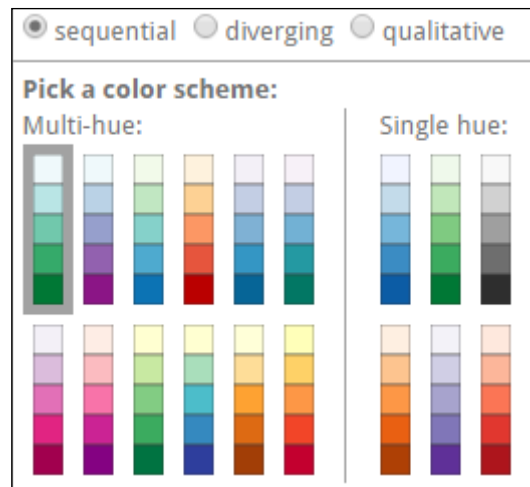
To create a classed color renderer, we need define a stops object to define discrete colors instead of continuous colors. A stops object will contain the color at each stop. When defining stops, we need *not* define the minDataValue or maxDataValue. Let's discuss a bit about where we can get an appropriate color scheme for our renderer.

Selecting a color scheme

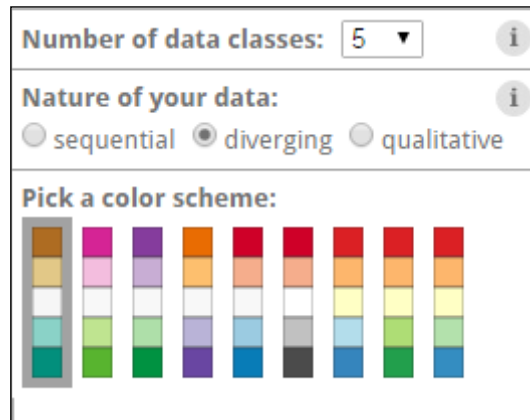
The following website provides us with an easy way to choose a color scheme that can be used for constructing a colorInfo object or color ramps: <http://colorbrewer2.org/>

In this website you can do the following:

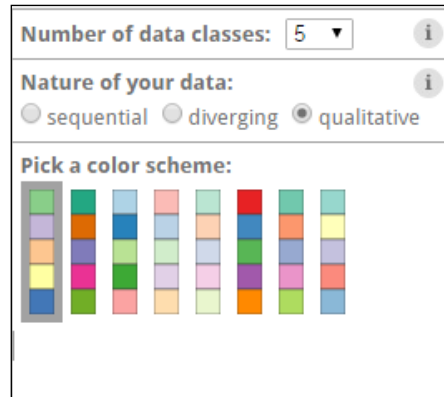
1. Select the number of data classes – the default is 3. The API's default number of classes is 5. So change the drop down value to 5 classes.
2. Select the nature of your data:
 - **sequential:** Use this to show an incremental quantity such as population or population density.



- **diverging:** Use this to emphasize the difference in values, especially at the extreme ends. For example, when mapping median income, the lower end of the income scale may be displayed in red and the higher end in blue.



- **qualitative:** This color scheme is used when we need to differentiate distinct values or classes using different colors.



3. Pick as multihued or a single-hue color scheme.
4. Constraint the color hues based on:
 - Purpose:
 - Color-blind friendly
 - Print friendly
 - Photocopy safe
 - Context:
 - Roads
 - Cities
 - Borders
5. Export the color scheme as:
 - JavaScript Array object – this is the handiest function
 - Adobe PDF

Number of data classes: 5

Nature of your data:
 sequential diverging qualitative

Pick a color scheme:

Only show:
 colorblind safe
 print friendly
 photocopy safe

Context:
 roads
 cities
 borders

Background:
 solid color
 terrain

color transparency

5-class RdYlBu

Export your selected color scheme:

Permalink
 Have a direct link to this color scheme.
<http://colorbrewer2.org/?type=diverging&>

Adobe
 Download an Adobe Swatch Exchange (ASE) file of this scheme.

GIMP and Inkscape
 GIMP color palette for this scheme.

JavaScript
 Colors for this scheme as a JS array
`["#d7191c", "#fdae61", "#ffffbf", "#abd9e9", "#2c7bb6"]`

CSS
 CSS classes for this scheme
`.RdYlBu .q0-5(fill:rgb(215,25,28)) .RdYlB`

Use this color scheme if you want to emphasize the differences in the extreme data values

This color scheme is not photocopy friendly, but is color-blind friendly and should look good on LCD screens

JavaScript array for the hex values of the selected color scheme

Creating a classed color renderer

As we discussed earlier, to create a classed color renderer, we need to define a `stops` object to define discrete colors instead of continuous colors. A `stops` object will contain the color at each stop. A `stops` object is an array object that is assigned to the renderer object. A `stops` array object contains objects with the following properties:

- value
- color
- label

A stops object mostly looks like this:

```
var stops =
[
  {
    "value": 27349.802772469,
    "color": { "b": 226, "g": 235, "r": 254, "a": 1 },
    "label": " < -1.5 Std. Dev."
  },
  {
    "value": 39912.112219098,
    "color": { "b": 185, "g": 180, "r": 251, "a": 1 },
    "label": "-1.5 - -0.50 Std. Dev."
  },
  {
    "value": 52474.421665726,
    "color": { "b": 161, "g": 104, "r": 247, "a": 1 },
    "label": "-0.50 - 0.50 Std. Dev."
  },
  {
    "value": 65036.731112354,
    "color": { "b": 138, "g": 27, "r": 197, "a": 1 },
    "label": "0.50 - 1.5 Std. Dev."
  },
  {
    "value": 77599.040558982,
    "color": { "b": 119, "g": 1, "r": 122, "a": 1 },
    "label": "1.5 - 2.5 Std. Dev."
  }
]
]
```

Now let's find a way to automatically populate the stops object. Remember we can get an array of colors based on a color scheme we select from the `colorbrewer2.org` website. The `color` array can be used to fill the `color` property of each object in the stops object. The `value` property of each object in the stops object can be derived from the return object of the `featureLayerStatistics` computation. The `featureLayerStatistics` computation provides the minimum, maximum, and label values for each class. We can assign the maximum value of each class to the `value` property for each object in the stops object:

```
//Create a params object for use getClassBreaks method in
// FeatureLayerStatistics module
//Define the field upon which Stats is computed,
//The classification method which should be one among the following:
//standard-deviation, equal-interval, natural-breaks, quantile
//Number of classes the data should be classified. Default is 5
var featureLayerStatsParams_color = {
    field: "MEDHINC_CY",
    classificationMethod: selectedClassificationMethod,
    numClasses: 5
};

//Compute the Class Break Statistics. This returns a promise

var color_stats_promise = featureLayerStats.getClassBreaks(featureLayerStatsParams_color);
color_stats_promise.then(function (color_stat_result) {

//The classBreakInfos property of the color_stat_result has all the
//class break values

var colorStops = [];

//Color JavaScript array exported from colorbrewer2.org
var colors = ['#feebe2', '#fbb4b9', '#f768a1', '#c51b8a', '#7a0177'];

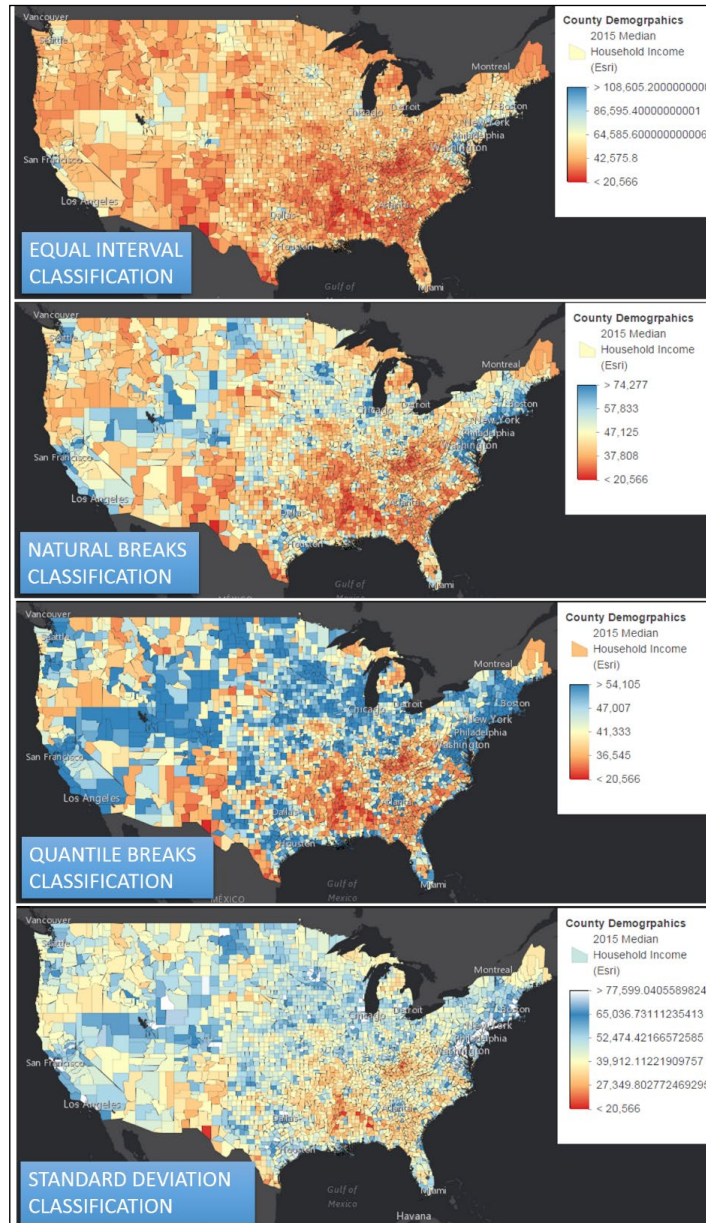
//Loop through each Break info provided by the Feature Layer Stats
    array.forEach(color_stat_result.classBreakInfos,
function (classBreakInfo, i) {
        colorStops.push({
//Get value property from the Break value's maximum value
            value: classBreakInfo.maxValue,
//Get color from the color Array
            color: new Color(colors[i]),
//Get label value from the label value provided by the Feature Layer
//Stats
            label: classBreakInfo.label
        });
    });

//Define Default renderer symbol
var symbol = new SimpleFillSymbol();
symbol.setColor(new Color([255, 0, 0]));
symbol.setOutline(new SimpleLineSymbol(SimpleLineSymbol.STYLE_SOLID,
new Color([0, 0, 0]), 0.5));

var colorBreakRenderer = new ClassBreaksRenderer(symbol);

//Set the color stops to the stops property to setColorInfo method of
```

```
//the renderer
colorBreakRenderer.setColorInfo({
    field:"MEDHINC_CY",
    stops: colorStops
});
});
```



opacityInfo

`opacityInfo` is an object that defines how a feature's opacity is calculated.

The `opacityInfo` object can be used to set the opacity levels for the classes in a `ClassBreaksRenderer`. The `opacityInfo` object can also be used to set a continuous opacity renderer.

Similar to the `colorInfo` object, you can either specify the opacity values as an array along with the minimum and maximum data value, or you can define the `stops` object within which you can define the opacity value.

Using `opacityInfo` to create a continuous renderer:

```
var minOpacity = 0.2;
var maxOpacity = 1;

var opacityInfo = {
  field: "DIVINDX_CY",
  minDataValue: 0,
  maxDataValue: 100,
  opacityValues: [minOpacity, maxOpacity]
};
```

Using opacityInfo to create a classes opacity renderer

Let's use the `opacityInfo` to render another field representing the diversity Index of each county. The diversity index measures diversity on a scale from 0 to 100.

The diversity index, an Esri proprietary index, is defined as the likelihood that two persons, selected at random from the same area, would belong to a different race or ethnic group. The diversity index measures only the degree of diversity in an area, not its racial composition.

Our objective is to display the counties with a higher diversity index with higher opacity values, and counties with a lesser diversity index with lesser opacity values. The opacity values can be broken between a minimum and maximum value by using the following snippet:

```
var opacity = minOpacity + i * maxOpacity / (opacity_stat_result.
classBreakInfos.length - 1);
```

In the previous snippet, `opacity_stat_result` is the promise result of the `getClassBreaks()` method of the `FeatureLayerStatistics` module:

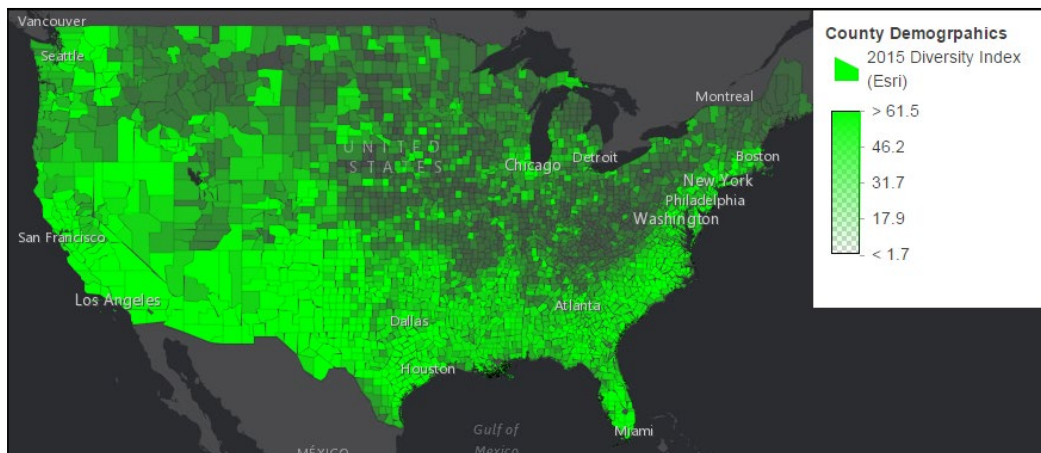
```
var featureLayerStatsParams_opacity = {
  field: "DIVINDX_CY",
  classificationMethod: selectedClassificationMethod, //standard-
  deviation, equal-interval, natural-breaks, quantile and standard-
  deviation
  numClasses: 5
};

var opacity_stats_promise = featureLayerStats.getClassBreaks(featureLa
yerStatsParams_opacity);
opacity_stats_promise.then(function (opacity_stat_result) {

  var opacityStops = [];
  array.forEach(opacity_stat_result.classBreakInfos, function
(classBreakInfo, i) {
    var minOpacity = 0;
    var maxOpacity = 1;
    //Calculate opacity by dividing between
    var opacity = minOpacity + i * maxOpacity / (opacity_stat_result.
classBreakInfos.length - 1);
    opacityStops.push({
      value: classBreakInfo.maxValue,
      opacity: opacity
    });
  });

var symbol = new SimpleFillSymbol();
symbol.setColor(new Color([255, 0, 0]));
var opacityBreakRenderer = new ClassBreaksRenderer(symbol);
opacityBreakRenderer.setOpacityInfo({
  field:"MEDHINC_CY",
  stops: stops
});

CountyDemogrphicsLayer.setRenderer(opacityBreakRenderer);
CountyDemogrphicsLayer.redraw();
```



SizeInfo

The `SizeInfo` object defines the size of the symbol where feature size is proportional to data value.

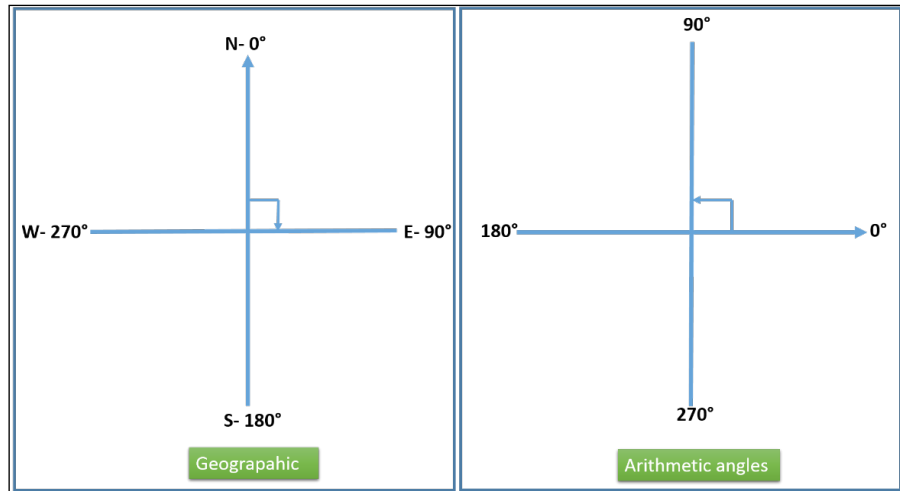
The API help page mentions that the symbol size can represent two different types of data—distance and non-distance. The distance data type refers to the actual distance on the field, and the non-distance data type refers to the cartographic size of the symbols. Representing the tree canopy using the `sizeInfo` based on the actual diameter of the canopy is an example of distance data-type. Representing the size of the roads based on the traffic density or size of the states based on the population density or median income enhances the cartographic presentation of the feature.

RotationInfo

`RotationInfo` can be used to define how marker symbols are rotated. `RotationInfo` can be used to depict wind direction, vehicle heading, and so on. A field specifying the rotation angle must be present to define the `RotationInfo`. There are two types of rotation angle units allowed. They are:

- **geographic:** This represents angles from the geographic north in a clockwise direction. Wind speeds and car directions are normally expressed in geographic angles.
- **arithmetic:** This represents angles measured in an anticlockwise direction.

The following diagram shows the difference between geographic and arithmetic angles:



Multivariate mapping

So far, we have been discussing rendering features using a single field name or variable. And we have also been discussing the various visual variables that can be used to render features such as color, opacity, size, rotation, and so on. What if we could combine the visual variables and render features based on more than one field value?

As an example, when mapping at county level, we may consider using color to represent population density, opacity to indicate median household income, and size to indicate the percentage of federal spending on education, which is normalized by the population field. The number of fields we choose to use is limited to the four visual variables, namely: color, opacity, size, and rotation.

Multivariate mapping is enabled by a property known as `visualVariables` in `ClassBreaksRenderer`. Let's try to use two visual variables, namely `colorInfo` and `opacityInfo`, which we used to demonstrate two different demographic parameters, namely median household income and diversity index. Our current objective would be to represent median house income using color, and at the same time determine the opacity value of the features based on the diversity index:

```
function applySelectedRenderer(selectedClassificationMethod) {  
    var featureLayerStatsParams_color = {  
        field: "MEDHINC_CY",
```

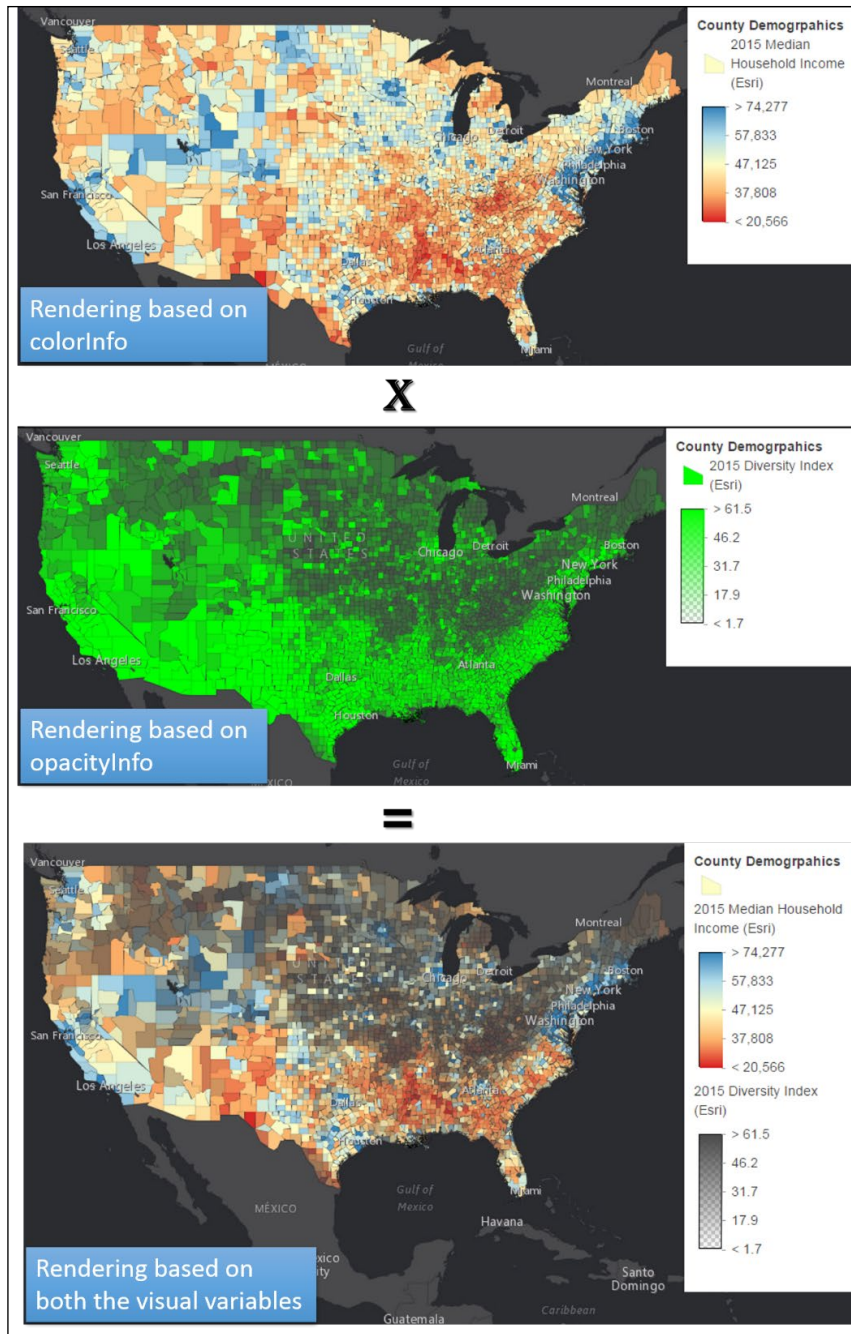
```
        classificationMethod: selectedClassificationMethod, //
standard-deviation, equal-interval, natural-breaks, quantile and
standard-deviation
        numClasses: 5
    };
    var featureLayerStatsParams_opacity = {
        field: "DIVINDX_CY",
        classificationMethod: selectedClassificationMethod, //
standard-deviation, equal-interval, natural-breaks, quantile and
standard-deviation
        numClasses: 5,
        //normalizationField: 'TOTPOP_CY'
    };

    var color_stats_promise = featureLayerStats.getClassBreaks(fea
tureLayerStatsParams_color);
    var opacity_stats_promise = featureLayerStats.getClassBreaks(f
eatureLayerStatsParams_opacity);
    all([color_stats_promise, opacity_stats_promise]).
then(function (results) {
    var color_stat_result = results[0];
    var opacity_stat_result = results[1];

    var colorStops = [];
    var colors = ['#d7191c', '#fdae61', '#ffffbf', '#abd9e9',
'#2c7bb6'];
    array.forEach(color_stat_result.classBreakInfos,
function (classBreakInfo, i) {
        colorStops.push({
            value: classBreakInfo.maxValue,
            color: new Color(colors[i]),
            label: classBreakInfo.label
        });
    });
    var opacityStops = [];
    array.forEach(opacity_stat_result.classBreakInfos,
function (classBreakInfo, i) {
        var minOpacity = 0;
        var maxOpacity = 1;
        var opacity = minOpacity + i * maxOpacity /
(opacity_stat_result.classBreakInfos.length - 1);
        opacityStops.push({
            value: classBreakInfo.maxValue,
            opacity: opacity
        });
    });
```



```
    });  
  
    var visualVariables = [  
      {  
        "type": "colorInfo",  
        "field": "MEDHINC_CY",  
        "stops": colorStops  
      }  
  
      ,  
      {  
        "type": "opacityInfo",  
        "field": "DIVINDX_CY",  
        "stops": opacityStops  
      }  
    ];  
  
    console.log(JSON.stringify(visualVariables));  
    var symbol = new SimpleFillSymbol();  
    symbol.setColor(new Color([0, 255, 0]));  
    symbol.setOutline(new  
    SimpleLineSymbol(SimpleLineSymbol.STYLE_SOLID, new  
    Color([0, 0, 0]), 0.5));  
  
    var colorBreakRenderer = new  
    ClassBreaksRenderer(symbol);  
    colorBreakRenderer.setVisualVariables(visualVariables);  
    CountyDemogrpahicsLayer.setRenderer(colorBreakRenderer);  
    CountyDemogrpahicsLayer.redraw();  
    legend.refresh();  
  });  
}
```



Smart mapping

With the knowledge of all these statistics, it's time to go smart with mapping using the smart mapping module provided by the API. Imagine a module that can automatically call the renderer parameters on its own given a few basic inputs, such as the feature layer on which the renderer needs to be generated and the classification method.

Module name: `esri/renderers/smartMapping`

The smart mapping module provides several methods, each of which produces a renderer. The renderers that the smart mapping module can produce are:

- Color-based classed renderer
- Size-based classed renderer
- Type-based renderer
- Heat map renderer

Smart mapping even takes care of the rendering based on the Basemap. For example, a certain color or opacity renderer works well with a darker-themed Basemap such as satellite, and certain renderers work well with a light-themed Basemap such as street maps.

With three simple steps, you can let the API decide the color scheme and create the classes color renderer for you:

- Construct a schemes object from the Esri styles `choropleth` module (`import esri/styles/choropleth`)
- Construct a classed color parameter object with the following properties:
 - `basemap`
 - `classificationMethod`
 - `layer`
 - `field`
 - `scheme` — choose the `primaryScheme` property from the schemes object constructed earlier
 - `numClasses`
- Assign a classed color parameter object as a parameter to the `createClassedColorRenderer()` method belonging to the smart mapping module

- Assign the renderer property returned by the smart mapping method to the feature layer's `setRenderer()` method as a parameter
- Redraw the feature layer and refresh the legend object

The following code explains how smart mapping can be used to create a classed color renderer:

```
//Call this function with the classification method as input
function applySmartRenderer(selectedClassificationMethod) {

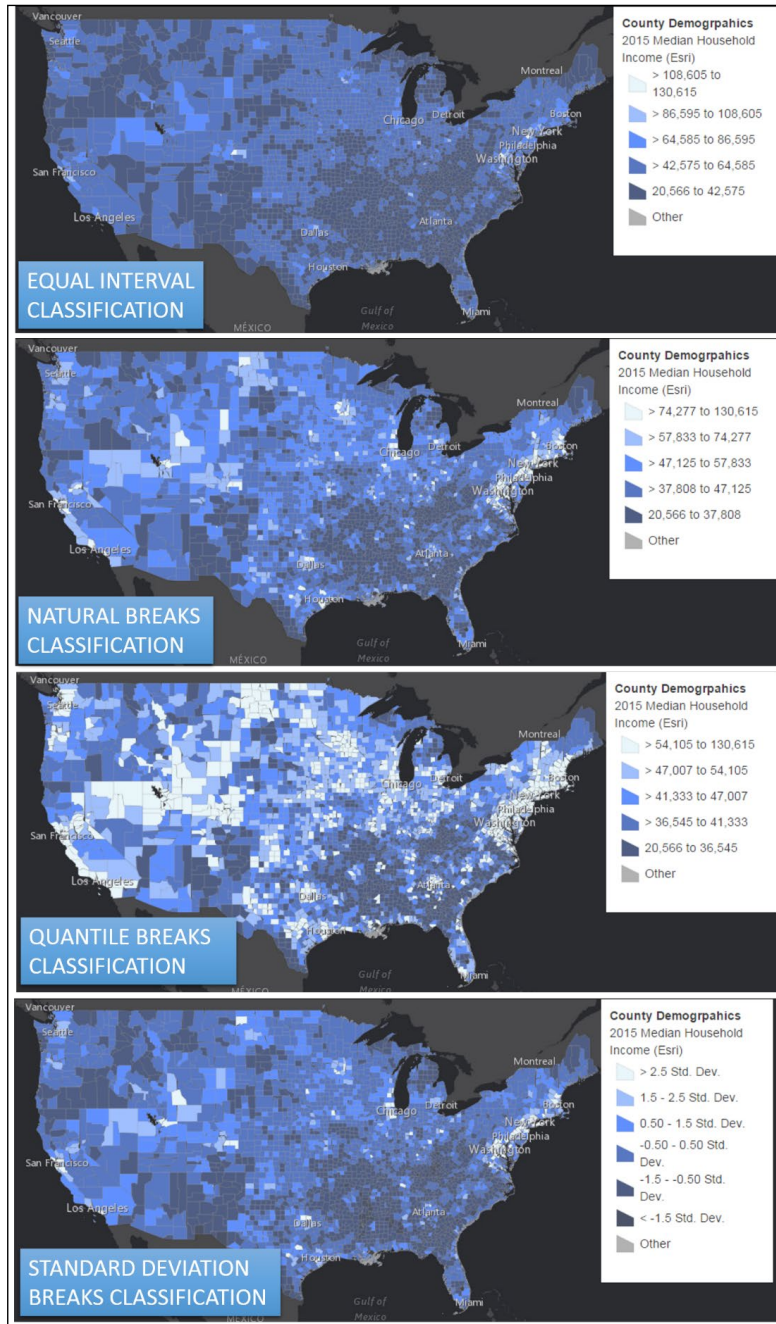
  //Create a scheme object assigning a theme
  var schemes = esriStylesChoropleth.getSchemes({
    //The following options are available for theme:
    // high-to-low, above-and-below, centered-on, or extremes.
    theme: "high-to-low",
    basemap: map.getBasemap(),
    geometryType: "polygon"
  });
  console.log(JSON.stringify(schemes));

  //Create a classed color Render Parameter object
  var classedColorRenderParams = {
    basemap: map.getBasemap(),
    classificationMethod: selectedClassificationMethod,
    field: 'MEDHINC_CY',
    layer: CountyDemogrpahicsLayer,
    scheme: schemes.primaryScheme,
    numClasses: 5
  };

  SmartMapping.createClassedColorRenderer(classedColorRenderParams).
  then(function (result) {
    CountyDemogrpahicsLayer.setRenderer(result.renderer);
    //Redraw the feature layer
    CountyDemogrpahicsLayer.redraw();
    //Update the legend
    legend.refresh();
  }).otherwise(function (error) {
    console.log("An error occurred while performing%s, Error: %o",
      "Smart Mapping", error);
  });
}
```

The following screenshots show the classed color renderer created using the smart mapping module for four different classifications, namely equal interval, natural breaks, quantile, and standard deviation. The user's discretion is used to decide which would serve as the best classification method based on the purpose of mapping the data, and also the audience.

We can manually define the color scheme by editing the scheme object, which is a property in the parameter object for the `createClassedColorRenderer()` method.



Summary

We are one step closer to becoming a map data scientist. We covered a lot of ground in this chapter, starting with a brush up of a few introductory statistics concepts. We then saw the code in action in how a statistics definition and feature layer statistics module can give us invaluable statistic measures that can be used to render the map data meaningfully. We then evaluated how to use the visual variables such as `colorInfo`, `opacityInfo`, `rotationInfo`, and `sizeInfo` effectively in a renderer. We also tried to combine these visual variables and performed a multivariate rendering. And, finally, we tried our hands at automatic rendering using the smart mapping module. In the next chapter, we will be dealing with charts and other advanced visualization techniques to provide analytical information to the users.

8

Advanced Map Visualization and Charting Libraries

Rendering on a map may not be the only way to visualize spatial data. To put the data into perspective, we may have to resort to non-spatial analytics and charting capabilities provided by `dojo` and other popular libraries to complement the spatial visualization capabilities of the map. In this chapter, we are going to extend the Demographics Analytic Portal we started building in the last chapter with the aid of charting libraries and other visualization methods such as data clustering. This chapter deals with the following major topics:

- Charting with `dojo`
- Charting with D3 library
- Charting with Cedar

Charting with `dojo`

The ArcGIS API is well integrated for charting with `dojo`. The charting capabilities are provided by the experimental modules of `dojo`, hence the name `dojox`, for which the `x` refers to the experimental nature of the modules. Yet these modules are stable enough to be integrated into any production environment. The following modules are considered the bare minimum modules for developing charting functionality with `dojo`:

- `dojox/charting`
- `dojox/charting/themes/<themeName>`
- `dojox/charting/Chart2D`
- `dojox/charting/plot2d/Pie`

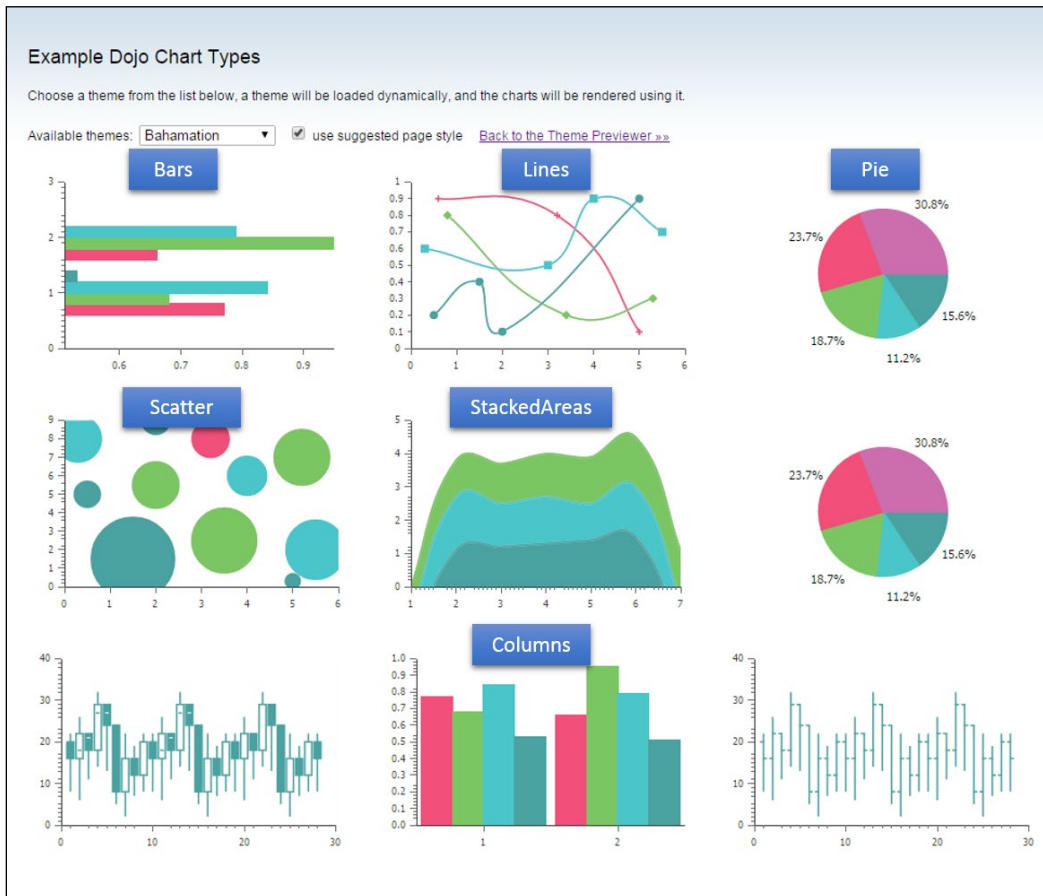
Dojo chart themes

The `dojo` charting library provides a lot of themes, and a theme name must be selected among the list of themes provided by `dojo`. A list of all themes provided by `dojo` is found under the following URL: http://archive.dojotoolkit.org/nightly/dojotoolkit/dojo/charting/tests/theme_preview.html

Themes provided by `dojo` charting library are as follows:

Julie	Desert	CubanShirts
ThreeD	Distinctive	RoyalPurples
Chris	Dollar	SageToLime
Tom	Grasshopper	Shrooms
Claro	Grasslands	Tufte
PrimaryColors	GreySkies	WatersEdge
Electric	Harmony	Wetland
Charged	IndigoNation	PlotKit.blue
Renkoo	Ireland	PlotKit.cyan
Adobebricks	MiamiNice	PlotKit.green
Algae	Midwest	PlotKit.orange
Bahamation	Minty	PlotKit.purple
BlueDusk	PurpleRain	PlotKit.red

The ideal location to test these different chart themes is at http://archive.dojotoolkit.org/nightly/dojotoolkit/dojo/charting/tests/test_themes.html?Julie.



Charting using the popup template

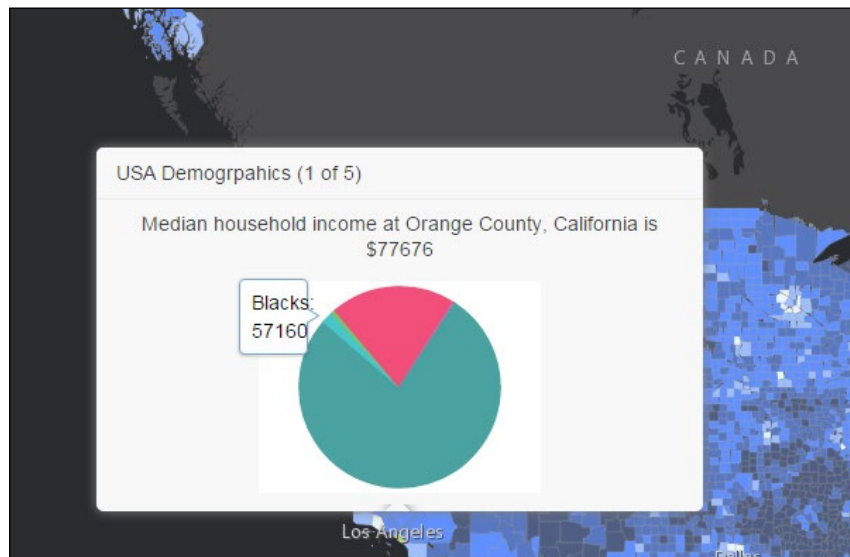
Basic charting capabilities can be displayed in the popup of the feature layer using the `mediaInfos` property of the popup template. We will use the county-level demographics feature layer we used in the last chapter to create this chart. We are interested in the following fields:

Fields	Description
NAME	Name of the county
STATE_NAME	Name of the state
TOTPOP_CY	Total population count for the county
MEDHINC_CY	Median household income of the county
DIVINDX_CY	Diversity index calculated for the county
WHITE_CY	Count of white males and females
BLACK_CY	Count of black males and females
AMERIND_CY	Count of American Indians (male and female)
ASIAN_CY	Count of Asians (male and female)
PACIFIC_CY	Count of Pacific Islanders (male and female)
OTHRACE_CY	Count of other races (male and female)

Creating a `mediaInfos` object involves constructing a `fieldInfos` object if we need to change the field names, or give an alias to them in the chart. The `mediaInfos` object accepts a `theme` property. Mention one of the dojo charting theme names or a custom theme created by you:

```
var template = new PopupTemplate({
  title: "USA Demograpahics",
  description: "Median household income at {NAME}, {STATE_NAME} is
  ${MEDHINC_CY}",
  //define field infos so we can specify an alias in the chart
  fieldInfos: [
    {
      fieldName: "WHITE_CY",
      label: "White Americans"
    },
    {
      fieldName: "BLACK_CY",
      label: "Blacks"
    },
    {
      fieldName: "AMERIND_CY",
      label: "American Indians"
    }
  ]
});
```

```
    },
    {
      fieldName: "ASIAN_CY",
      label: "Asians"
    },
    {
      fieldName: "PACIFIC_CY",
      label: "Pacific Islanders"
    },
    {
      fieldName: "OTHRACE_CY",
      label: "Other Race Count"
    }
  ],
  mediaInfos: [{ //define the bar chart
    caption: "",
    type: "piechart", // image, piechart,
    barchart, columnchart, linechart
    value:
    {
      theme: "Dollar",
      fields: ["WHITE_CY", "BLACK_CY", "AMERIND_CY", "ASIAN_CY",
"PACIFIC_CY", "OTHRACE_CY"]
    }
  }]
});
```



Types of 2D charts provided by dojo modules

We have seen a pie chart in action. Let's discuss some more chart types provided by the `dojo` module and the utility of some of the more popular chart types. Notice the difference between chart types such as Bars and Columns, as well as Scatter and MarkersOnly.

Chart type	Description
Areas	Area under data line(s) will be filled
Bars	Refers to horizontal bars
ClusteredBars	Horizontal bars with grouped data sets
ClusteredColumns	Vertical bars with grouped data sets
Columns	Refers to charts with vertical bars
Grid	For adding a grid layer to your chart
Lines	Basic line chart
Markers	Line chart with data points marked
MarkersOnly	Only data points are shown
Pie	Represents the distribution of data by representing it on a circular dias
Scatter	Used to plot data
Stacked	Data sets charted in relation to the previous data set
StackedAreas	Stacked data sets with filled areas under chart lines
StackedBars	Stacked data sets with horizontal bars
StackedColumns	Stacked data sets with vertical bars
StackedLines	Stacked data sets using lines

Dojo charting methods

The charting module has four important methods that will help us create a chart. They are:

- `addPlot()`: Defines the type of chart and other ancillary properties that define the chart.
- `setTheme()`: Lets us set a dojo theme to the chart. The themes can be customized too.
- `addSeries()`: Defines the data used by the chart.
- `render()`: Renders the chart.

Defining your plot

Using the `addPlot()` method you can define your plot. The plot accepts a name and an argument array:

```
var chart1 = new Chart2D(chartDomNode);
chart1.addPlot("default", plotArguments);
```

Let's see what constitutes the `plotArguments` object. The properties of the `plotArguments` vary based on the type of chart we choose to use. If we choose a chart type that uses a line, area, or data points to define the data, properties such as `lines`, `areas`, or `markers` should be set to a Boolean value. The `lines` option determines whether or not lines are used to connect your data points. If the `areas` type is selected, the area below your data line will be filled. The `markers` option will determine whether markers are placed at your data points.

The `plotArguments` can accept the following properties:

- `type`: The type of chart to be rendered
- `lines`: Boolean to indicate whether the chart data needs to be enclosed by lines
- `areas`: Boolean value to indicate whether the data is enclosed by an area
- `markers`: Boolean value which determines whether markers are placed at data points

For chart types such as `StackedLines` or `StackedAreas`, we can use properties such as `tension` and `shadows` to enhance the visualization of the chart. `Tension` smooths the lines connecting the data points, and the `shadows` property will add shadows to the lines. The `shadow` property itself is an object that accepts three properties named `dx`, `dy`, and `dw`, which define what should be the *x* offset, *y* offset, and width of the shadow line:

```
chart1.addPlot("default", {type: "StackedLines", lines: true, markers:
false, tension : 3, shadows: {dx:2, dy: 2, dw: 2}});
```

When rendering a bar chart, use the `gap` property to represent the number of pixels between bars:

```
chart1.addPlot("default", {type: "Bars", gap: 3});
```

Defining the theme

Using the list of themes mentioned earlier, we can set the theme for our chart using the `setTheme()` method:

```
chart.setTheme(dojoxTheme);
```

Pushing the data

We can push the data into a chart using the `addSeries()` method:

```
chart.addSeries("PopulationSplit", chartSeriesArray);
```

The `addSeries()` method accepts two arguments. The first argument mentions a name for the data and the second argument. The second argument is an array object that holds the actual data. It can be one-dimensional data such as `[10, 20, 30, 40, 50]` or two-dimensional data, in which case the `x` and `y` properties of the data can be mentioned:

```
chart.addSeries("Students", [
  {x: 1, y: 200 },
  {x: 2, y: 185 }
]);
```

The `x` component can be omitted if it is a pie chart.

Chart plugins

There are a few plugins that can be added to the chart module of `dojo` that add value to the charting functions. These plugins provide interactivity to the chart data and most of the plugins reveal extra information about the data item or emphasize the data item being hovered upon. Some provide an overall sense of the data with the aid of visualization elements such as a legend. Some of the functions accomplished by the plugins are:

- Adding a tooltip to the chart
- Moving the pie slice and magnifying it
- Adding a legend
- Highlighting the data item

Plugin modules such as `dojox/charting/widget/Legend` provide support from the Legend widget. The `dojox/charting/action2d/Tooltip` module supports tooltip support on chart data. Including the `dojox/charting/action2d/Magnify` module will magnify the chart data being hovered upon, giving greater interactivity with the chart. The `dojox/charting/action2d/MoveSlice` module treats the chart data as a slice and shifts the locations of the chart data being hovered upon. This, along with the `Magnify` plugin, helps us to effectively give a sense of user interactivity with the chart data. The `dojox/charting/action2d/Highlight` module highlights the data being hovered upon with a different highlight color such as cyan.

Implementing the plugin is very easy too. The following lines of code implements the plugins such as `Highlight`, `Tooltip`, and `MoveSlice` on the dojo chart object:

```
new Highlight(chart, "default");
new Tooltip(chart, "default");
new MoveSlice(chart, "default");
```

Let's create a complete chart in a dynamic div on the `infotemplate` property of the feature layer.

We will use the county-level demographics feature layer for this demonstration too. Our objective is to create a pie chart to display the racial distribution of any county that we click. We would be calling a function to create the `Infowindow` content for each feature dynamically:

```
var template = new InfoTemplate();
template.setTitle("<b>${STATE_NAME}</b>");

//Get the info template content from the getWindowContent function
template.setContent(getWindowContent);

var statesLayer = new FeatureLayer("http://demographics5.arcgis.
com/arcgis/rest/services/USA_Demographics_and_Boundaries_2015/
MapServer/15", {
  mode: FeatureLayer.MODE_ONDEMAND,
  infoTemplate: template,
  outFields: ["NAME", "STATE_NAME", "TOTPOP_CY", "MEDHINC_CY",
"DIVINDX_CY", "WHITE_CY", "BLACK_CY", "AMERIND_CY", "ASIAN_CY",
"PACIFIC_CY", "OTHRACE_CY"]
});
```

In the function that returns the `Infotemplate` content, we will do the following:

1. Create a Tab container that will contain two content panes.
2. The first content will display details about the county being selected and the Median Household Income data.
3. The second content pane will contain the dojo pie chart.
4. Before rendering the pie chart, we shall calculate the percentage of each racial group against the total population.
5. Also, we shall assign a label for each racial group. This label will be used while using the legend.
6. Also, pie chart data objects accept a tooltip property where we will mention the label along with the data value.
7. We will try to use the chart plugins such as `Highlight`, `Tooltip`, and `Moveslice` to highlight the selected sub data item.

Now let's try to implement these steps in the code. We will write a function that constructs the chart and returns the chart content as a dom element. We will use the `setContent()` method of the `infotemplate` to set the dom element returned by the following function:

```
function getWindowContent(graphic) {
    // Make a tab container.
    var tc = new TabContainer({
        style: "width:100%;height:100%;"
    }, domConstruct.create("div"));

    // Make two content panes, one showing Median household income
    //details. And the second showing the pie chart

    var cp1 = new ContentPane({
        title: "Details",
        content: "<a target='_blank' href='http://en.wikipedia.org/wiki/"
+ graphic.attributes.NAME + "'>Wikipedia Entry</a><br/>" + "<br/>
Total Population: " + graphic.attributes.TOTPOP_CY + " <br/> Median
House Income: $" + graphic.attributes.MEDHINC_CY
    });
    // Display a dojo pie chart for the racial distribution in %
    var cp2 = new ContentPane({
        title: "Pie Chart"
    });
    tc.addChild(cp1);
    tc.addChild(cp2);

    // Create the chart that will display in the second tab.
    var c = domConstruct.create("div", {
        id: "demoChart"
    }, domConstruct.create("div"));
    var chart = new Chart2D(c);
    domClass.add(chart, "chart");

    // Apply a color theme to the chart.
    chart.setTheme(dojoxTheme);

    chart.addPlot("default", {
        type: "Pie",
        radius: 70,
        htmlLabels: true
    });
    tc.watch("selectedChildWidget", function (name, oldVal, newVal) {
        if (newVal.title === "Pie Chart") {
            chart.resize(180, 180);
        }
    });
};
```

```
// Calculate percent of each ethnic race
// "WHITE_CY", "BLACK_CY", "AMERIND_CY", "ASIAN_CY", "PACIFIC_CY",
"OTHRACE_CY"
var total = graphic.attributes.TOTPOP_CY;
var white = {
  value: number.round(graphic.attributes.WHITE_CY / total * 100, 2),
  label: "White Americans"
};
var black = {
  value: number.round(graphic.attributes.BLACK_CY / total * 100, 2),
  label: "African Americans"
};
var AmericanIndians = {
  value: number.round(graphic.attributes.AMERIND_CY / total * 100,
2),
  label: "American Indians"
};
var Asians = {
  value: number.round(graphic.attributes.ASIAN_CY / total * 100, 2),
  label: "Asians"
}
var Pacific = {
  value: number.round(graphic.attributes.PACIFIC_CY / total * 100,
2),
  label: "Pacific Islanders"
};
var OtherRace = {
  value: number.round(graphic.attributes.OTHRACE_CY / total * 100,
2),
  label: "Other Race"
};
var chartFields = [white, black, AmericanIndians, Asians, Pacific,
OtherRace];
var chartSeriesArray = [];
array.forEach(chartFields, function (chartField) {
  var chartObject = {
    y: chartField.value,
    tooltip: chartField.label + ' : ' + chartField.value + ' %',
    text: chartField.label
  }
  chartSeriesArray.push(chartObject);
});

chart.addSeries("PopulationSplit", chartSeriesArray);
//highlight the chart and display tooltips when you mouse over a
slice.
new Highlight(chart, "default");
```

```
new Tooltip(chart, "default");
new MoveSlice(chart, "default");

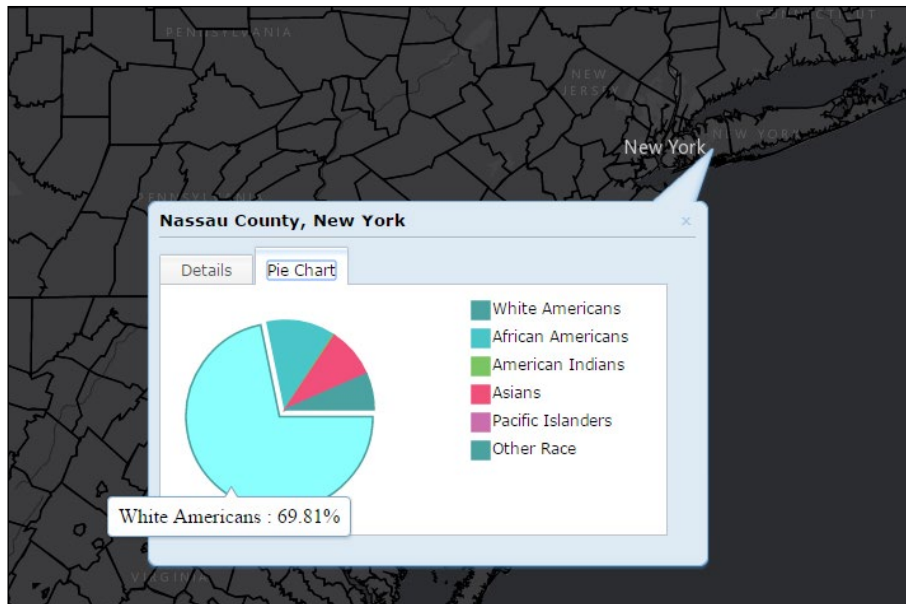
cp2.set("content", chart.node);
return tc.domNode;
}
```

When this code is implemented, we will get a popup after we click on any county. The popup contains two tabs – the first tab gives details about the **Total Population** of the tab and the **Median Household Income** in that county. The title for the entire popup will mention the county name and the state name. The contents of the first tab will have a dynamically generated Wikipedia link to the county and state.

The first tab of the pop-up container is shown in the following screenshot:



The second tab in the popup shows the dojo chart. We have a legend element for the chart. When we hover over any of the data in the pie chart, it is sliced, magnified a bit, and highlighted.



Charting with D3.js

D3.js is a JavaScript library for manipulating documents based on data. D3 stands for data-driven documents and this library provides powerful visualization components and a data-driven approach to DOM manipulation.

To use D3 in our JavaScript application, we can download the library from the D# website found at <http://d3js.org/>.

Or we can use the CDN in our script tag:

```
<script src="//d3js.org/d3.v3.min.js" charset="utf-8"></script>
```

A more dojo-centric approach would be to add this as a package in the `dojoconfig` and use it as a module in the `define` function.

Here is a snippet to add D3 as a package to the `dojoConfig`:

```
var dojoConfig = {
  packages:
  [
    {
      name: "d3",
      location: "http://cdnjs.cloudflare.com/ajax/libs/d3/3.5.5",
    }
  ]
};
```

```
        main: "d3.min"
    }
  ]
};
```

Using d3 library as in the define function:

```
define([
  "dojo/_base/declare",
  "d3",
  "dojo/domReady!"
],
function
(
  declare,
  d3
)
{
  //Keep Calm and use D3 with dojo
});
```

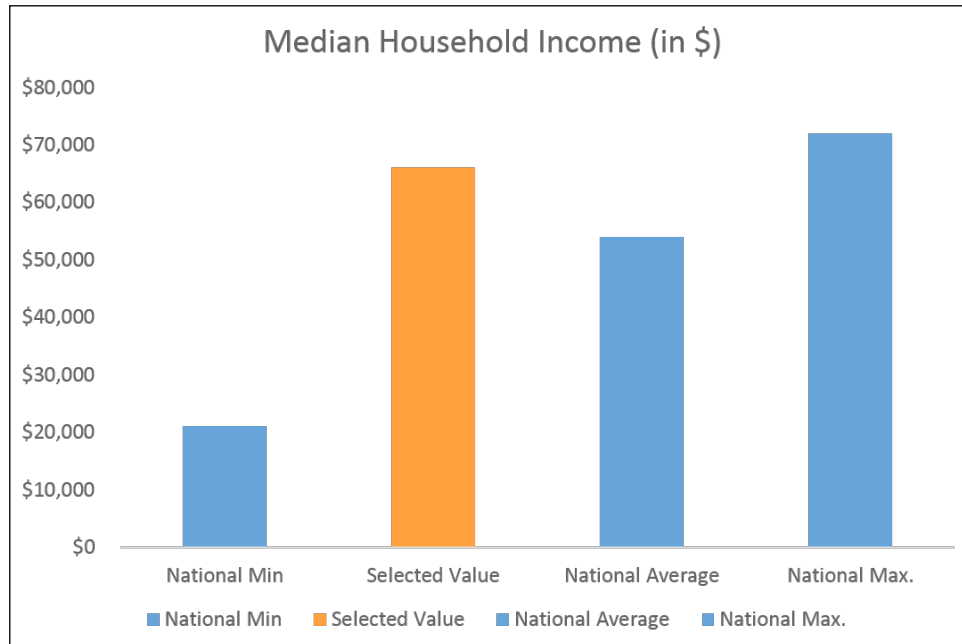
Creating a column chart with D3

Let's create a column chart with D3 using the county-level demographics data. Our objective is to use the column chart to display four measures of Median Household Income centered upon the Median Household Income of the county of interest.

The four measures are:

- The National Minimum or value at 5th percentile (Average – three Standard Deviation)
- The Median Household Income of the county being clicked
- The National Average value for Median Household Income
- The National Maximum or value at the 95th percentile

The following image is a mock-up of how we intend to build our chart:



There are several reasons why we have chosen to demonstrate constructing this chart using D3. D3 is entirely data driven, and hence flexible, especially for data visualization. Many visualization libraries are built on top of D3 and a knowledge of D3 will even help us build intuitive charts and data visualizations.

D3 selections

D3 works on selections. The selections in D3 are very similar to jQuery selections. To select the `body` tag, all you have to do is declare:

```
d3.select("body")
```

To select all `div` tags with a particular style class named `chart`, use the following snippet:

```
d3.select(".chart").selectAll("div")
```

To append an `svg` (scalable vector graphic) tag or any other HTML tag to a `div` or the `body` tag, use the `append` method. The SVG element is used to render most graphic elements:

```
d3.select("body").append("svg")
```

Use it along with an `enter()` method to indicate that the element accepts the input:

```
d3.select("body").enter().append("svg")
```

D3 data

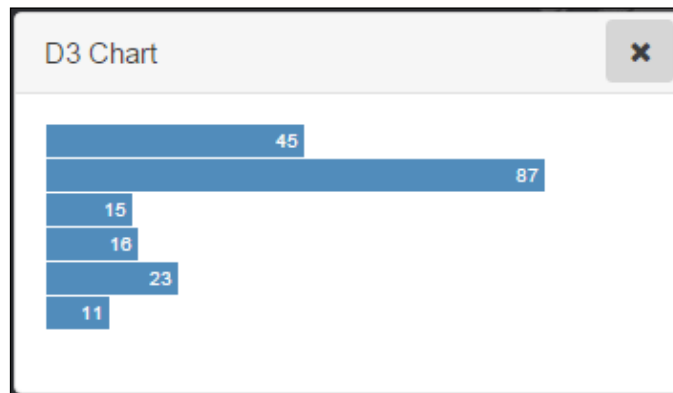
D3 is driven by data as its name suggests. All we need to render a simple chart is to feed data to the D3 selection. Data can be as simple as an array:

```
var data = [45, 87, 15, 16, 23, 11];

var d3Selection = d3.select(".chart").selectAll("div").data(data).
  enter().append("div");

d3Selection.style("width", function (d) {
  return d * 3 + "px";
}).text(function (d) {
  return d;
});
```

All we are doing in the previous snippet is we are setting the width property for the style object of the D3 selection. And we get this:



The width value of each `div` in pixels is taken from the value of each element in the data array multiplied by 20, and the text value within the bar is again taken from the value of the individual data. There's something that needs to be done before, to get this beautiful chart — we need to set the CSS styling for the `div`. Here's a simple CSS snippet we used:

```
.chart div {
  font: 10px sans-serif;
  background-color: steelblue;
```

```
    text-align: right;
    padding: 3px;
    margin: 1px;
    color: white;
}
```

D3 scaling

In the previous snippet to show a simple D3 chart, we used a multiplicand value of 20 with each value of the data to get the pixel value for the `div` width. Since our container `div` was around 400 pixels wide, this multiplicand value was fine. But what multiplicand value should we use for a dynamic data? The rule of thumb is that we should use some kind of scaling mechanism to scale the pixel values so that our maximum-most data value fits inside the chart container `div` comfortably. D3 provides a mechanism to scale our data and calculate the scaling factor, which we use to conveniently scale our data.

D3 provides a `scale.linear()` method to calculate the scaling factor. Additionally, we need to use two more methods, namely `domain()` and `range()`, to actually calculate the scaling factor. The `domain()` method accepts an array with two elements. The first element should mention the minimum-most data value or 0 (whichever is appropriate) and the second element should mention the maximum-most value of the data. We can use the D3 function `d3.max` to find the maximum value of the data:

```
d3.max(data)
```

The `range` function also accepts an array with two elements, which should list the pixel range of the container `div` element:

```
var x = d3.scale.linear()
    .domain([0, d3.max(data)])
    .range([0, 750]);
```

Once we find the scaling factor `x`, we can use this as the multiplicand for the data item value to derive the pixel value:

```
d3.select(".chart").selectAll("div").data(data)
    .enter().append("div").style("width", function (d) {
        return x(d) + "px";
    }).text(function (d) {
        return d;
    });
```

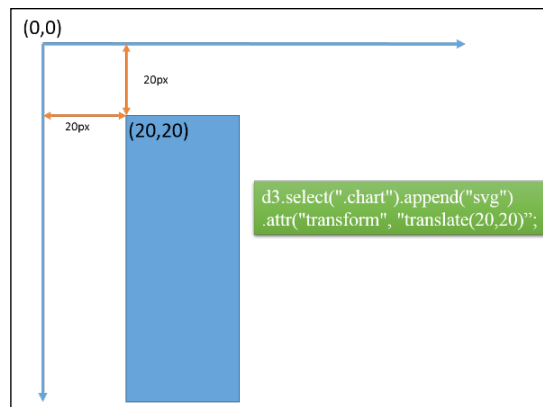

Integrating SVG into D3 charts

SVG, though intimidating in its entirety, offers several advantages while working with data visualizations, and supports a lot of primitive shapes to be rendered in HTML. One key thing to be noted is that the SVG coordinate system starts from the top-left corner and we need to bear this in mind while calculating the desired positions of our elements.

Appending an SVG element is similar to appending a `div` to our chart class:

```
var svg = d3.select(".chart").append("svg")
    .attr("width", 500)
    .attr("height", 500)
    .append("g")
    .attr("transform", "translate(20,20)");
```

In the previous snippet, we can actually set the styling and other attributes such as width and height. `transform` is an important property by which we can move the position of the `svg` element (remember the SVG coordinate system origin is in the top-left corner).



Since we will be constructing a column chart, the first element in the array accepted by the `range()` method while calculating D3 linear scaling should not be the minimum-most value, but rather the maximum height value in pixels. The second element in the array is the minimum pixel value:

```
var y = d3.scale.linear()
    .range([700, 0]);
```

Conversely, the x scaling factor should be based on an ordinal scale (meaning, we don't use numbers to calculate the width and spacing of the bars):

```
var x = d3.scale.ordinal()
    .rangeRoundBands([0, width], .1);
```

From the feature statistics module we have discussed earlier, we should be able to get the mean and standard deviation of a particular field in the feature layer.

From the previous two pieces of information, we know how to calculate the 2.5th percentile (bottom 2.5% income) and 97.5th percentile (top 2.5% income level). We intend to compare the Median Household Income of the selected feature along with these values. The formula to calculate the 2.5th and 97.5th percentile is shown as follows:

<i>1st percentile = mean - 2,33 * SD</i>	<i>99th percentile = mean + 2,33 * SD</i>
<i>2.5th percentile = mean - 1.96 * SD</i>	<i>97.5th percentile = mean + 1.96 * SD</i>
<i>5th percentile = mean - 1.65 * SD</i>	<i>95th percentile = mean + 1.65 * SD</i>

From previous statistic computations, we know the following data:

```
mean = $46193
SD = $12564
```

We need the 2.5th and 97.5th percentile which can be calculated as follows:

```
2.5th percentile value = mean - 1.96 * SD
                        = 46193 - 1.96*(12564)
                        = 21567.56
```

And for the 97.5th:

```
97.5th percentile = mean + 1.96 * SD
                  = 46193 + 1.96*(12564)
                  = 70818.44
```

So, this is going to be the data for our chart:

```
var data = [
  {
    "label": "Top 2.5%ile",
    "Income": 70818
  },
  {
    "label": "Bottom 2.5%ile",
    "Income": 21568
  }
];
```

```
    },
    {
      "label": "National Avg",
      "Income": 46193
    },
    {
      "label": "Selected Value",
      "Income": 0
    }
  ]
};
```

The `Income` value for the `Selected Value` label is set to 0. This value will be updated as we click a feature in the `feature class`. We will also define a `margin` object as well as `width` and `height` variables for use in our chart. The `margin` object we defined looked like this:

```
var margin = {
  top: 20,
  right: 20,
  bottom: 30,
  left: 60
},
width = 400 - margin.left - margin.right,
height = 400 - margin.top - margin.bottom;
```

While constructing the chart, we will be considering the following steps:

1. Determine the x scaling factor and y scaling factor.
2. Define the x and y axes.
3. Clear all the existing contents of the `div` with a `chart class`.
4. Define the x and y domain values based in the `margin` object, as well as `width` and `height` values.
5. Define the `SVG` element that would hold our chart.
6. Add the axes as well as the chart data as `rectangle graphic elements` in the `SVG`.

We will write the functionality in a function, and call the function as needed:

```
function drawChart() {

  // Find X and Y scaling factor

  var x = d3.scale.ordinal()
    .rangeRoundBands([0, width], .1);
```

```
var y = d3.scale.linear()
    .range([height, 0]);

// Define the X & y axes

var xAxis = d3.svg.axis()
    .scale(x)
    .orient("bottom");

var yAxis = d3.svg.axis()
    .scale(y)
    .orient("left")
    .ticks(10);

//clear existing
d3.select(".chart").selectAll("*").remove();
var svg = d3.select(".chart").append("svg")
    .attr("width", width + margin.left + margin.right)
    .attr("height", height + margin.top + margin.bottom)
    .append("g")
    .attr("transform", "translate(" + margin.left + "," +
margin.top + ")");

// Define the X & y domains
x.domain(data.map(function (d) {
    return d.label;
}));
y.domain([0, d3.max(data, function (d) {
    return d.population;
})]);

svg.append("g")
    .attr("class", "x axis")
    .attr("transform", "translate(0," + height + ")")
    .call(xAxis);

svg.append("g")
    .attr("class", "y axis")
    .call(yAxis)
    .append("text")
    .attr("transform", "translate(-60, 150) rotate(-90)")
    .attr("y", 6)
    .attr("dy", ".71em")
```

```
.style("text-anchor", "end")
.text("Population");

svg.selectAll(".bar")
  .data(data)
  .enter().append("rect")
  .attr("class", "bar")
  .style("fill", function (d) {
    if (d.label == "Selected Value")
      return "yellowgreen";
  })
  .attr("x", function (d) {
    return x(d.label);
  })
  .attr("width", x.rangeBand())
  .attr("y", function (d) {
    return y(d.population);
  })
  .attr("height", function (d) {
    return height - y(d.population);
  });
}
```

We can call the previous function on the feature layer `click` event. In our project, the feature `click` event is defined in a separate file, and the D3 chart code is in a separate file. So, we can send the click results through the `dojo` topic:

```
//map.js file

define("dojo/topic",..){
  on(CountyDemogrphicsLayer, "click", function(evt){
    topic.publish("app/feature/selected", evt.graphic);
  });
}
```

The result can be accessed in any other file by using the `subscribe()` method under the `topic` module. In the previous snippet, the result can be accessed by referring to the name called `app/feature/selected`:

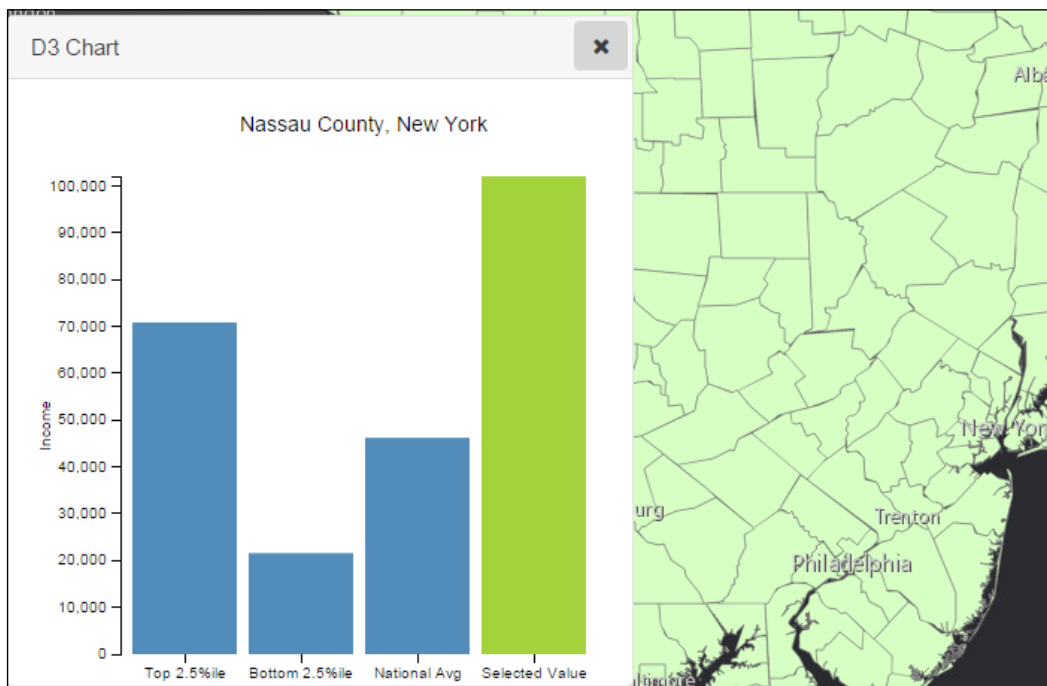
```
//chart_d3.js file

topic.subscribe("app/feature/selected", function () {
  var val = arguments[0].attributes.MEDHINC_CY;
  var title = arguments[0].attributes.NAME + ', ' +
  arguments[0].attributes.STATE_NAME;;
});
```

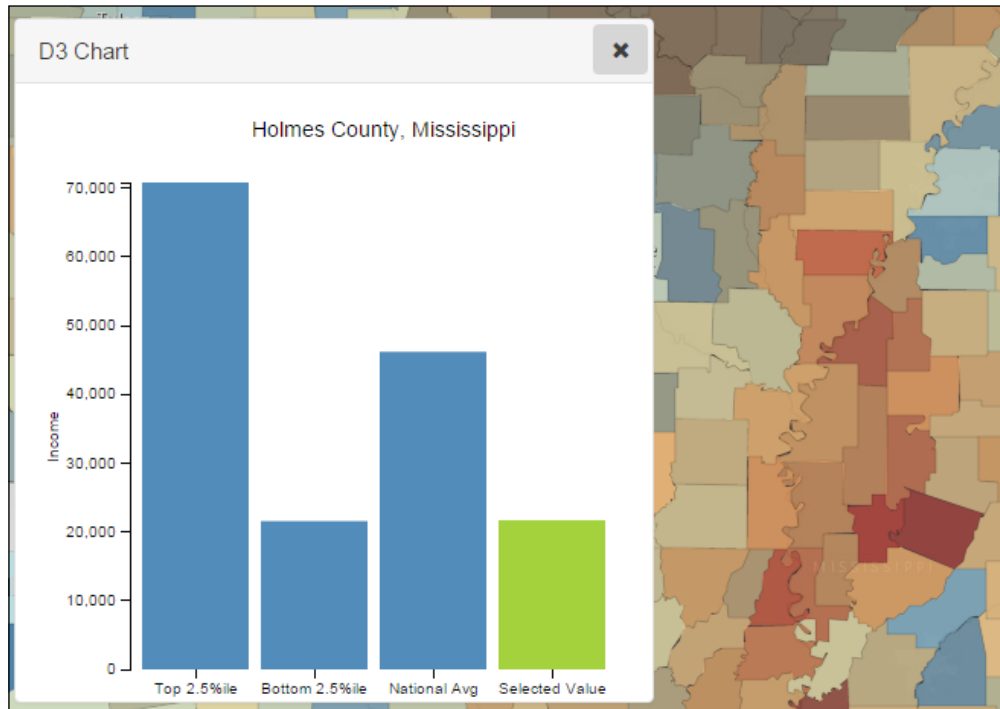
```
array.forEach(data, function (item) {
  if (item.label === "Selected Value") {
    item.Income = val;
  }
});

drawChart(title);
console.log(JSON.stringify(data));
});
```

The following screenshot is a representation of the output of our code. The D3 chart represents a typical column chart with four columns. The first three data values are static as per our code, because we can compute the top and bottom 2.5th percentile as well as the national average from the feature layer data. The last column is the actual value of the selected feature in the feature layer. In the following snapshot we have clicked in Nassau county in New York state and the data value is a bit above \$100,000, which is well above the top 2.5th percentile mark:



In the following screenshot, we have selected one of the counties with the least Median Household Income. Notice how the Y axis re-calibrates itself based on the maximum value of the data.



Charting with D3 using SVG components can be cumbersome, but a basic knowledge of these will go a long way when we need to do high-level customizations.

Charting with Cedar

Cedar is a beta versioned library provided as an open source by Esri to create and share data visualizations based on ArcGIS Server data. It is built upon the D3 and Vega graphics libraries themselves. Cedar lets us create efficient data visualizations and charts using a simple template.

Loading Cedar libraries

We can load Cedar using two methods. We either use the script tags or use the AMD pattern. The latter method is preferred.

Loading using the script tags

Load Cedar and its dependencies by including script tags. This will make the Cedar global available to our application:

```
<script type="text/javascript" src="http://cdnjs.cloudflare.com/ajax/
libs/d3/3.5.5/d3.min.js"></script>
<script type="text/javascript" src="http://vega.github.io/vega/vega.
min.js"></script>
<script type="text/javascript" src="https://rawgit.com/Esri/cedar/
master/src/cedar.js"></script>

<script>
  var chart = new Cedar({"type": "bar"});
  ...
</script>
```

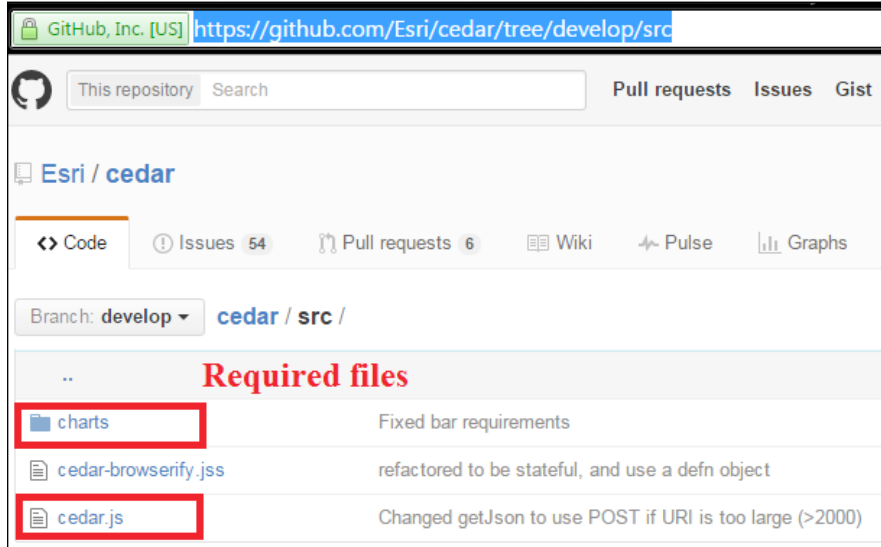
Loading using the AMD pattern

Alternatively, we can use the dojo loader, which is bundled with the ArcGIS API for JavaScript, to load Cedar and its dependencies by declaring them as packages:

```
var package_path = window.location.pathname.substring(0, window.
location.pathname.lastIndexOf('/'));
var dojoConfig = {
  packages: [{
    name: "application",
    location: package_path + '/js/lib'
  },
  {
    name: "d3",
    location: "http://cdnjs.cloudflare.com/ajax/libs/d3/3.5.5",
    main: "d3.min"
  },
  {
    name: 'vega',
    location: 'http://vega.github.io/vega/',
    main: 'vega.min'
  },
  {
    name: 'cedar',
    location: package_path + '/js/cedar',
    main: 'cedar'
  }
  ]
};
```


The dojo packages expect a set of Cedar library files at the `/js/cedar` location. We can download the required files from the following github repository found at <https://github.com/Esri/cedar/tree/develop/src>.

We need all the files found at the previously mentioned URL. Place these files in the `/js/cedar` folder in the application.



We can now load the Cedar module within our own define function as demonstrated in the following snippet:

```
define([
  "cedar",
  "dojo/domReady!"
], function (Cedar)
{
  var chart = new Cedar({
    ...

  });

  chart.show({
    elementId: "#cedarchartdiv",
    width: 900
  });
});
```

To create a simple chart, all we need to define are two properties:

- `type`—defines the type of chart we are trying to construct (bar, bubble, scatter, pie, and so on).
- `dataset`—defines where the data should come from; this can be either from a URL or values (array). The dataset also accepts properties such as query and mappings.
- The mappings property of the dataset defines the objects required to render the map. The specifications for the corresponding type of chart can be found at `/js/cedar/charts/<chart_type>.js`.

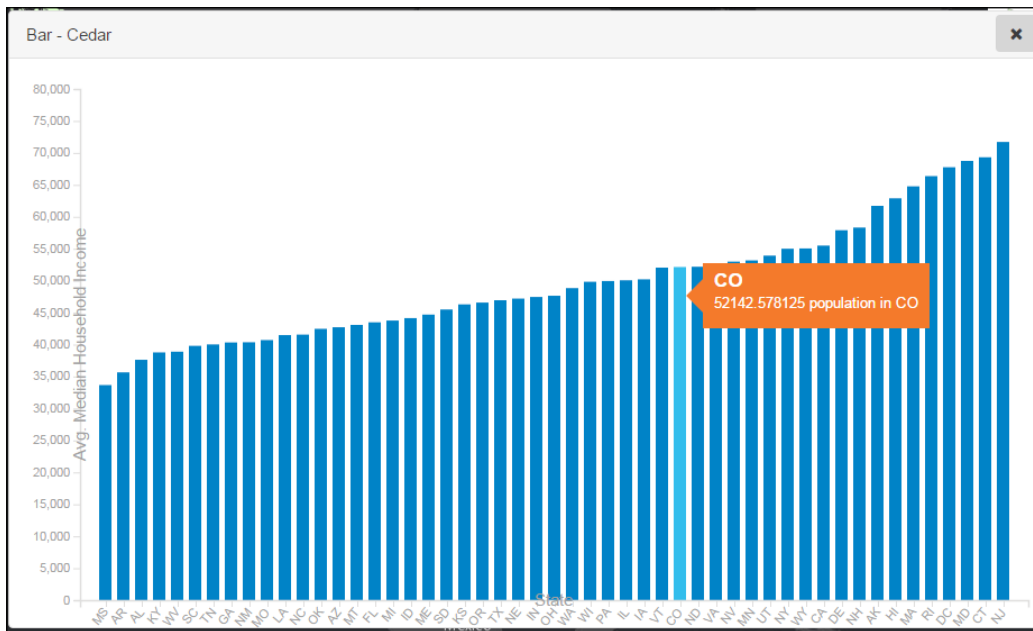
For a bar chart, the mappings property requires two objects, *x* and *y*. Let's try to create a summarization for our county demographics layer. Here we are trying to summarize the average of the Median Household Income of all counties grouped by state. The following simple code does all that and displays a simple bar chart:

```
var chart = new Cedar({
  "type": "bar",
  "dataset": {
    "url": "/proxy/proxy.ashx?http://demographics5.arcgis.com/arcgis/
rest/services/USA_Demographics_and_Boundaries_2015/MapServer/15",
    "query": {
      "groupByFieldsForStatistics": "ST_ABBREV",

//Find the average value of Median Household Income
      "outStatistics": [{
        "statisticType": "avg",
        "onStatisticField": "MEDHINC_CY",
        "outStatisticFieldName": "AVG_MEDHINC_CY"
      }]
    },
    "mappings": {
      "sort": "AVG_MEDHINC_CY",
      "x": {
        "field": "ST_ABBREV",
        "label": "State"
      },
      "y": {
        "field": "AVG_MEDHINC_CY",
        "label": "Avg. Median Household Income"
      }
    }
  }
});
```

```
chart.tooltip = {  
  "title": "{ST_ABBREV}",  
  "content": "{AVG_MEDHINC_CY} population in {ST_ABBREV}"  
}  
  
//show the chart  
chart.show({  
  elementId: "#cedarchartdiv",  
  width: 900  
});
```

The previous lines of code are all that are needed to configure the Cedar library, which provides us with this great visualization of the average income levels of all states and arranged in ascending order.



This kind of chart gives us a holistic picture of the data. Let's get our hands dirty and try to construct a scatter plot, which lets us map more than one variable.

Our objective is to map the income levels of all states along the X axis, and the diversity index along the Y axis, coloring the data points differently according to the state.

The demographics URL for the State-level data is: http://demographics5.arcgis.com/arcgis/rest/services/USA_Demographics_and_Boundaries_2015/MapServer/21

The mapping object should have an extra parameter named color:

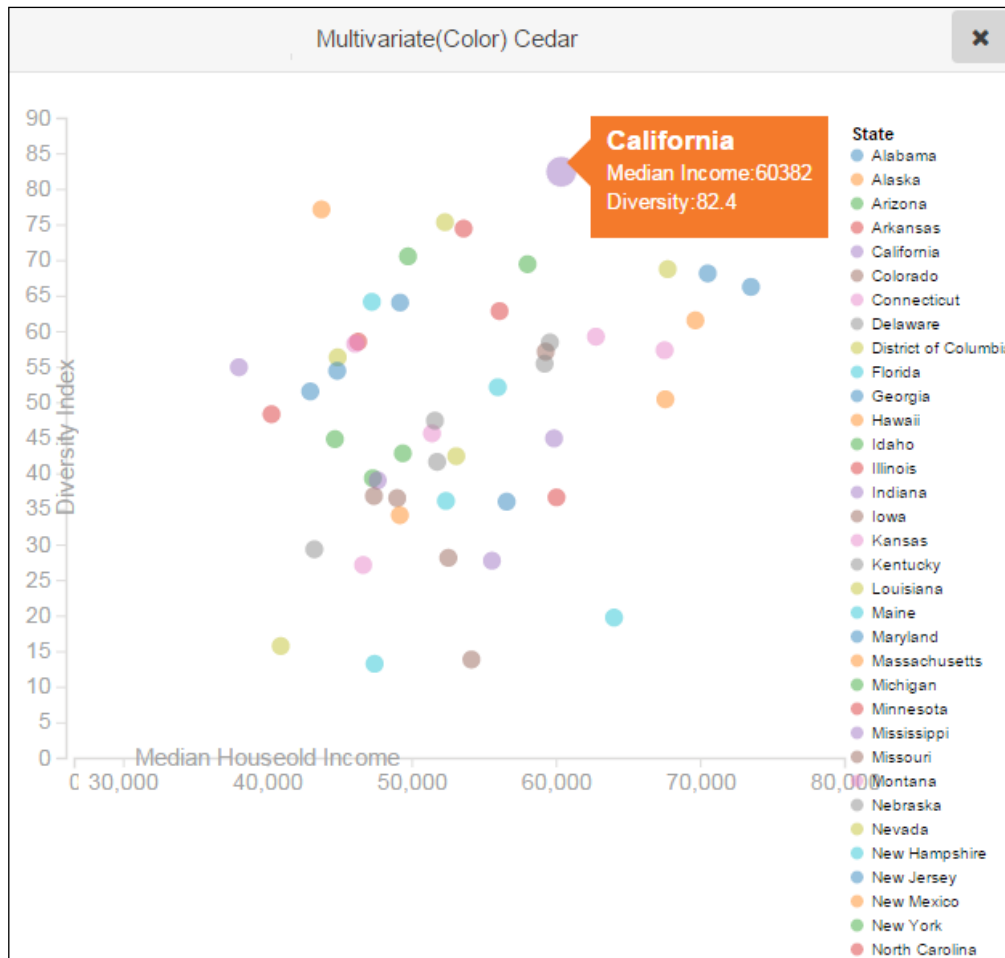
```
//Get data from the Query Task

var query = new Query();
var queryTask = new QueryTask("http://demographics5.arcgis.com/arcgis/
rest/services/USA_Demographics_and_Boundaries_2015/MapServer/21");
query.where = "1 = 1";
query.returnGeometry = false;
query.outFields = ["MEDHINC_CY", "DIVINDX_CY", "NAME", "TOTPOP_CY"];
queryTask.execute(query).then(function (data) {
    /*scatter*/
    var scatter_chart = new Cedar({
        "type": "scatter",
        "dataset": {
            "data": data,
            "mappings": {
                "x": {
                    "field": "MEDHINC_CY",
                    "label": "Median Houseold Income"
                },
                "y": {
                    "field": "DIVINDX_CY",
                    "label": "Diversity Index"
                },
                "color": {
                    "field": "NAME",
                    "label": "State"
                }
            }
        }
    });

    scatter_chart.tooltip = {
        "title": "{NAME}",
        "content": "Median Income:{MEDHINC_CY}<br/>Diversity:{DIVINDX_CY}"
    }

    scatter_chart.show({
        elementId: "#cedarScatterPlotDiv",
        width: 870,
        height: 600
    });
});
```

The following screenshot is the result of the implementation of the code given previously. The chart produces a legend based on the value that is colored differently. In our case, the different states are colored differently. This kind of coloring would be more appropriate if the number of values being colored was small, for example if we were using the colors to represent states categorized into some kind of regions such as North, North East, South, South West, and other cardinal directions.



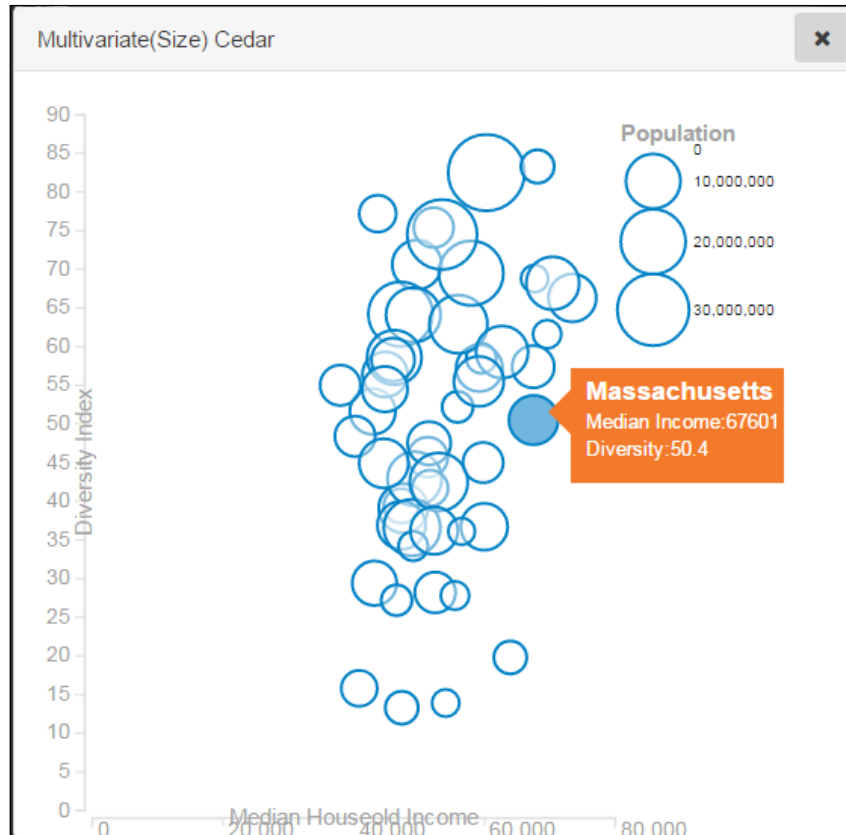
Creating a bubble chart gives an extra handle – representing a third variable using the size of the bubble:

```
var bubble_chart = new Cedar({
  "type": "bubble",
  "dataset": {
    "data": data,
    "mappings": {
      "x": {
        "field": "MEDHINC_CY",
        "label": "Median Household Income"
      },
      "y": {
        "field": "DIVINDX_CY",
        "label": "Diversity Index"
      },
      "size": {
        "field": "TOTPOP_CY",
        "label": "Population"
      }
    }
  }
});

bubble_chart.tooltip = {
  "title": "{NAME}",
  "content": "Median Income:{MEDHINC_CY}<br/>Diversity:{DIVINDX_
CY}"
}

bubble_chart.show({
  elementId: "#cedarBubblePlotDiv"
});
```

The following screenshot shows a bubble chart; the x position of the bubble represents the median household income of the county, the y position of the bubble represents the diversity index of the county, and the radius or the size of the bubble represents the total population of the county:



We began from creating a simple customizable chart in Infotemplate, which can visualize one variable, to a chart that can actually visualize three variables at the same time, thus enhancing our understanding of the data and increasing the value it offers.

Summary

We have covered how we can complement charting techniques along with spatial data to provide a comprehensive insight into the data we have. While working with `Infotemplate` and `dojo chart` is handy, working with `D3` provides greater flexibility and greater control over the graphical elements. `Cedar`, an open source data visualization library provided by Esri, is a great library for creating refreshingly new data visualizations very easily. Once we have mastered these techniques along with the statistical methods, and have learned to look at our data from different perspectives, we can claim ourselves as flag-bearers of map data science. There's one more component missing in the way we visualize our spatial data. That component is time. In the next chapter, we will see how to visualize spatio-temporal data along with the knowledge gained in advanced charting capabilities and the ArcGIS JavaScript API itself.

9

Visualization with Time Aware Layers

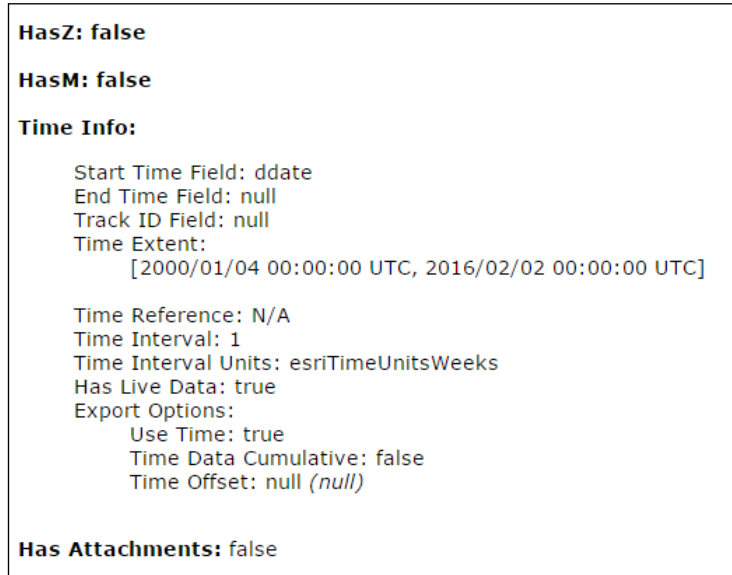
We have dealt with reading and displaying time-based data in our earlier chapters, as well as non-spatial charting methodologies using innovative libraries such as D3 and Cedar. This chapter deals with visualizing space-time data using spatial visualization, as well as other non-spatial visualization aids such as Time Slider and Time Graph. The following topics are discussed in this chapter:

- Time aware layers and the need for them
- Building a drought app using Time Slider
- Querying based on time data using D3
- Advanced spatio-temporal visualization using Cedar charts

Time aware layers

ArcGIS 10 and above includes support for time aware layers. Time aware layers are DynamicMapService or feature layers that have a `TimeInfo` property. The following screenshot shows the Service Catalog of a Time aware feature layer.

Check out the `TimeInfo` property in the image:



Snapshot of `TimeInfo` information in the Service Catalog

Let's discuss the components of the `TimeInfo` object, which is something similar to the one we saw in the previous image. The `TimeInfo` property provides us with the following information:

- Which field in the layer stores the time information (State Time Field, End Time Field).
- The minimum-most and maximum-most time for which data is available (Time Extent).
- The Time Reference property refers to the time zone in which the date time values are stored (if null, UTC time is followed; this shall be discussed in detail).
- The Time Interval units is the time interval at which data is available for each feature.
- *Has Live Data* property refers to a Boolean value, which indicates whether data is continuously updated.
- Export Options provides a list of properties such as Use Time, Time Data Cumulative, and Time Offset. The Time Data Cumulative is a Boolean value referring to whether features retrieved are accumulated with time.

Need for time aware layers

Time aware layers let us understand data in a spatio-temporal context; this means we can see how spatial information changes over a time period. This kind of data has various real-world applications such as:

- Understanding where the crime hotspots are in a city over time
- Tracking a hurricane and displaying its current position
- The spread of flood events in an area over a short span of time
- Displaying the proliferation of oil wells in a state
- How the drought conditions have affected a place over time

Understanding time aware layers

A certain basic understanding of the concept of time in ArcGIS will help us work with time aware layers better. The following points are worth noting:

- Time should always be referred into the **Coordinated Universal Time (UTC)**, which is functionally equivalent to **Greenwich Mean Time (GMT)**.
- Just like the spatial extent of the map, we can define the time extent of the map, which has time aware layers. This will only affect map layers that have the `timeInfo` property. Time extent is provided by the `esri/TimeExtent` module. We can define a `TimeExtent` object with either of the following properties:

- `startTime`
- `endTime`
- `startTime` and `endTime`

```
require(["esri/TimeExtent", ... ],
function(TimeExtent, ... ){
  var timeExtent = new TimeExtent();
  timeExtent.startTime = new Date("1/15/1989 UTC");
  map.setTimeExtent(timeExtent);
});
```

- The **Time Data Cumulative** property of the Export Options object under the Time Info object determines whether data can be cumulated or not.
- When the data in the time aware layer cannot be cumulated in the map display, we should be using just one thumb in the time slider. We will discuss the time slider shortly.

Building the Drought app

Let's build an app displaying the drought conditions of the US to understand the features supporting time aware layers in ArcGIS.

The following URL provides weekly updated values for drought intensity across the United States from 2000 to the present: http://earthobs1.arcgis.com/arcgis/rest/services/US_Drought/MapServer

You may need an ArcGIS Developer's account to access this data.

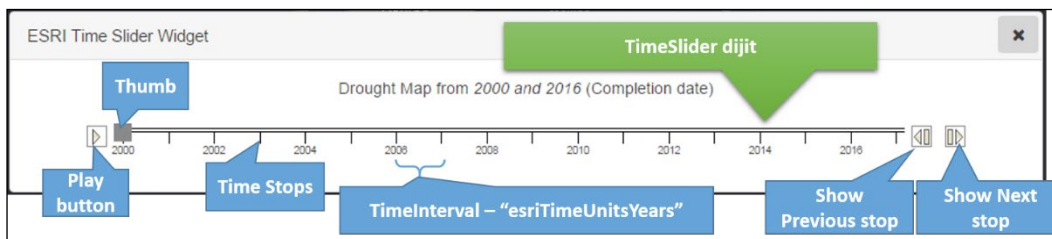
Drought in a region is defined as the imbalance between water supply and water demand over an extended period of time. Since droughts can have direct and indirect environmental, economic, and social consequences, monitoring drought is instrumental in planning, preparedness, and mitigation efforts at all levels of government.

Our application tries to display the current and historical drought values for the entire US. These data have been produced weekly since January 4, 2000 by the US Drought Monitor and the full time series is archived here. A new map is released every Thursday to reflect the conditions of the previous week.

Using the Time Slider

The `TimeSlider` module provided by the API is capable of interacting with the time aware layers. `TimeSlider` is a widget provided by the API, which we can use in our code and query the time aware layers dynamically. It also provides support for animation so that we can see how the spatial features change over time or how the features accumulate between a time interval.

To use `TimeSlider`, we need to load the Esri dijit named `esri/dijit/TimeSlider`. Apart from the `TimeSlider` dijit, we may also need to load the module named `esri/TimeExtent`. The time extent is useful in defining the stops. The following image tries to show the components of a `TimeSlider` dijit and the physical representation of the programming terminology with respect to the `TimeSlider` dijit, such as `stops`, `timeInterval`, `thumb`, and so on:



Steps to create a TimeSlider

Following are the steps to create a Time Slider:

1. On the load event of the DynamicMapService or the feature layer, get the time extent of the layer.
2. Initialize the `TimeSlider` dijit and assign it to a container element such as `div` or content panel. Assign the timeslide to the map too.
3. Set the other properties of the timeslider, such as thumb count, creating time stops given the layer's time extent and time units.
4. Set the moving rate for the thumb.
5. Create the labels for the time slider.
6. Start the time slider animation.

The time extent for the time slider can be obtained from the layer's `timeInfo` property itself:

```
on(droughtcMapServiceLayer, "load", function (evt) {
  var layerTimeExtent = evt.layer.timeInfo.timeExtent;
  _createEsriTimeSlider(layerTimeExtent);
});
```

The following code snippet explains how to set the time slider to the map and start the animation:

```
//Pass the time extent to the function

function _createEsriTimeSlider(layerTimeExtent) {

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

  /* We just need one thumb for our time aware data */

  timeSlider.setThumbCount(1);

  //Though a weekly data is available, let us Create Time stops
  //for Yearly intervals
  //

  timeSlider.createTimeStopsByTimeInterval(layerTimeExtent, 1,
    "esriTimeUnitsYears");
```

```
//Waits at each stop for 2 seconds

    timeSlider.setThumbMovingRate(2000);

//Start the time slider animation

    timeSlider.startup();

//add labels for every other time stop

var labels = array.map(timeSlider.timeStops, function (timeStop,
i) {
    if (i % 2 === 0) {
        return timeStop.getUTCFullYear();
    } else {
        return "";
    }
});

timeSlider.setLabels(labels);

//Wait for the map service to load at each stop

timeSlider.on("time-extent-change", function (evt) {
    //update the text

    var currentValString = evt.endTime.getUTCFullYear();
    dom.byId("daterange").innerHTML = "<i>" + currentValString +
"</i>";
});

on(droughtcMapServiceLayer, "update-start", function (evt) {

//When updating layer, pause the time slider animation

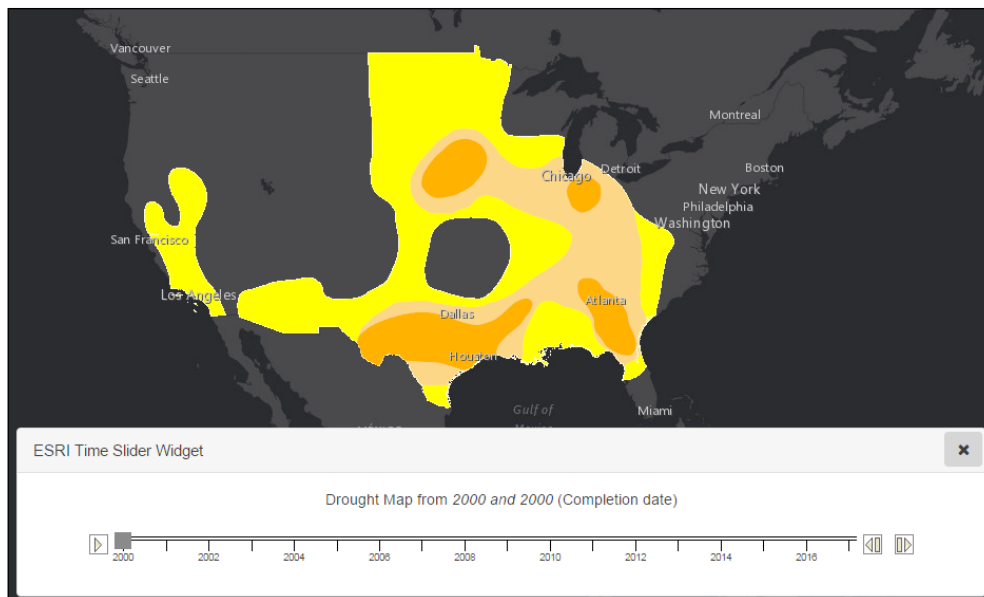
    timeSlider.pause();
});

on(droughtcMapServiceLayer, "update-end", function (evt) {

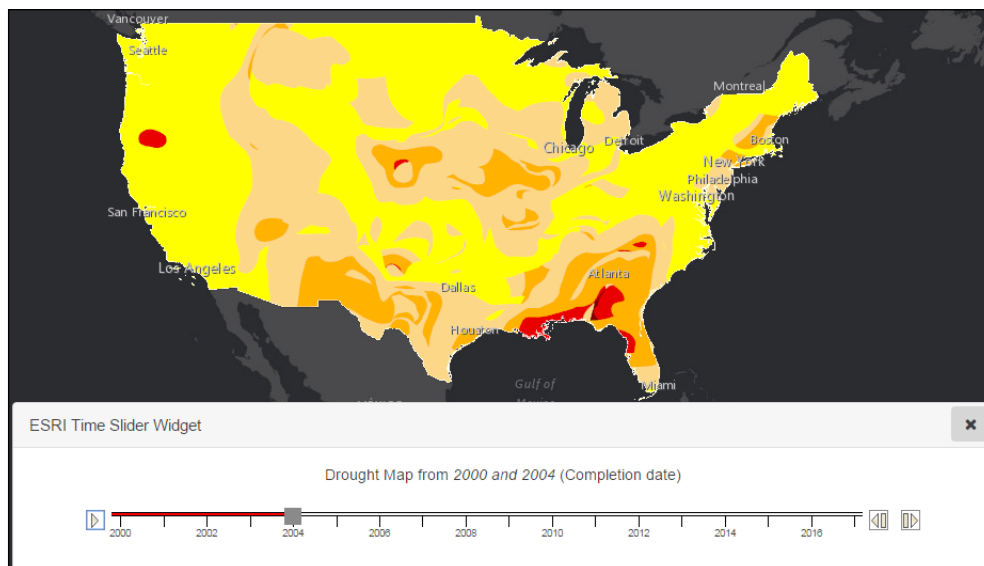
//When update is done, play the time slider animation

    timeSlider.play()
});
}
```

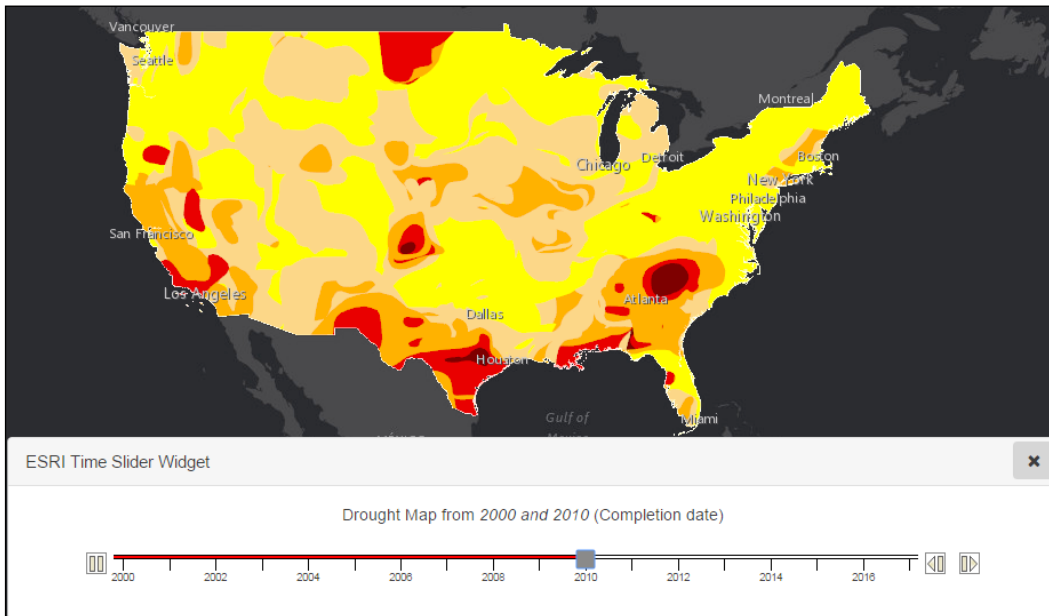
Using the previous chunk of code, we were able to develop a simple app with the `timeSlider` widget; a snapshot of the app during initial time stop can be seen here:



Once the play button is clicked, the thumb starts moving. At each stop, the thumb may pause beyond the stipulated time gap in the play animation. This pause is the time the map service takes to fetch the dynamic map service at the particular time stop. In the following image, the time slider animation stopped for a few seconds more, giving us an opportunity to capture the instance of the map in the year 2004:



If you were observing the networks tab of your browser, you would have noticed an HTTP GET request call being made every time the thumb moves to a stop along the Time Slider. At every stop, a new dynamic map image is being fetched, which corresponds to a map instance at a time. Let's consider this image that shows a snapshot at a time instance, which is the year 2010:



The HTTP GET request issued to generate the dynamic image you've seen previously is this URL with each of its query parameters separated by a new line:

```
http://localhost:9095/proxy/proxy.ashx?  
http://earthobs1.arcgis.com/arcgis/rest/services/US_Drought/MapServer/  
export?  
dpi=96  
&transparent=true  
&format=png8  
&time=946944000000%2C1262563200000  
&bbox=-17599814.30461256%2C1159119.7738912208%2C-  
4234952.783009615%2C6765317.176437697  
&bboxSR=102100  
&imageSR=102100  
&size=1366%2C573  
&f=image
```

The URL gives a lot of information about the kind of image being generated. It should be noted that the request goes through the proxy page. The time parameter represents the year 2010 in ticks.

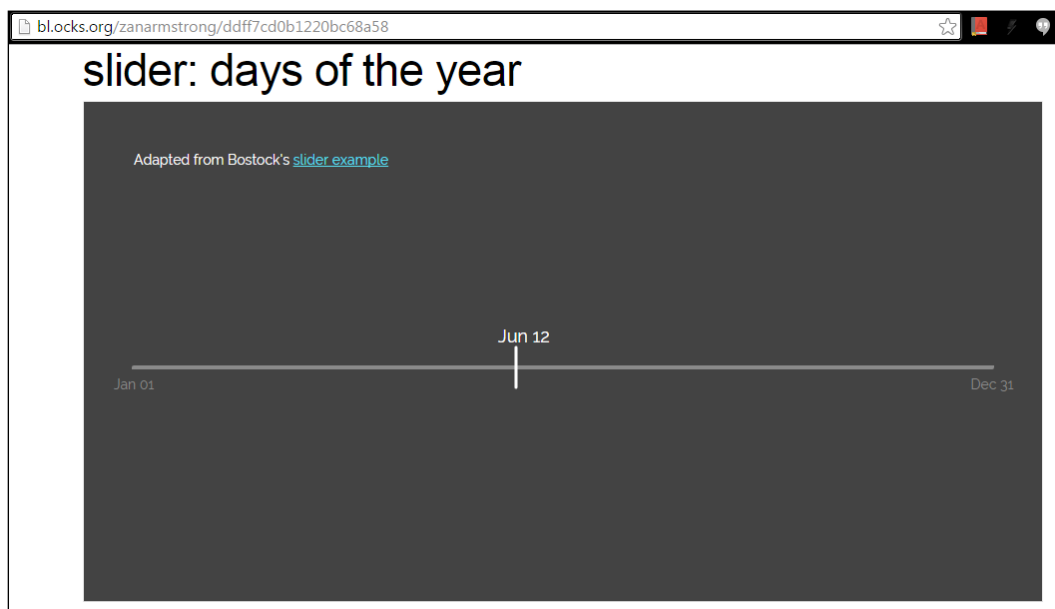
Querying based on time using D3

The previous example was based on querying time aware layers using the in-built `TimeSlider` `digit` provided by the API. We can further the capability of the Time Slider using the rich support for time-based data provided by the D3 library.

Our objective in this section would be to create a D3 Time Slider that can interact with our time aware layer.

The following code is inspired from the code listing given at the <http://bl.ocks.org/zanarmstrong/ddff7cd0b1220bc68a58>.

The webpage explains how to effectively use the D3 object to read and display time data in a time slider.



Here are some of the important concepts we have to understand before implementing the code.

Scaling and formatting time

In our earlier chapter, we dealt with how D3 functions can be used to scale numerical values as well as ordinal values. When we are dealing with time, we need to deal with scaling as applied to time. The following snippet explains how time extents can be scaled to the width of the container:

```
// parameters
var margin = {
  top: 10,
  right: 50,
  bottom: 50,
  left: 50
},
width = 800 - margin.left - margin.right,
height = 150 - margin.bottom - margin.top;

// scale function
var timeScale = d3.time.scale()
  .domain([startDate, endDate])
  .range([0, width])
  .clamp(true);
```

In the previous snippet, we assume that we are able to provide the start and end data values to the D3 time-scaling function from the layer's `timeInfo` property. We also need a proper date format to render the date values we have. The following line of code provides us with the date in the date-month-day format:

```
var formatDate = d3.time.format("%Y-%m-%d");
```

D3 brush

A D3 brush is equivalent to a thumb in a Time Slider `dijit`. Brush is a D3 SVG element object that accepts a time extent. In the following snippet, we have a brush element that accepts a timescale factor on the x axis:

```
// defines brush
var brush = d3.svg.brush()
  .x(timeScale)
  .extent([startingValue, startingValue])
  .on("brush", brushed);
```

Another important aspect we need to understand about the brush is about events such as `mousedown`. When the brush is moved by the user (on `mousemove` following a `mousedown`), the extent will be recomputed by calling `timescale.invert`. This will let us set the new extent for the brush. The following code explains this aspect:

```
svg.on("mousedown", function (data) {
  var value = brush.extent()[0];

  if (d3.event.sourceEvent) { // not a programmatic event
    value = timeScale.invert(d3.mouse(this)[0]);
    brush.extent([value, value]);
  }
  console.log(formatDate(value));
});
```

Apart from the code listing provided in the web page, we need an additional piece of code to fire the query to retrieve the time aware data only when the `mousedown` event on the brush is persistent for at least 500 milliseconds. Else the event will be fired numerous times as we move the brush along the timescale. The following function, which is fired when the brush is moved, will publish a topic titled `application/d3slider/timeChanged`:

```
function brushed() {
  var value = brush.extent()[0];

  if (d3.event.sourceEvent) { // not a programmatic event
    value = timeScale.invert(d3.mouse(this)[0]);
    brush.extent([value, value]);
  }

  handle.attr("transform", "translate(" + timeScale(value) +
    ",0)");
  handle.select('text').text(formatDate(value));
  var reqValue = formatDate(value);

  if (timer) {
    clearTimeout(timer);
  }
  timer = setTimeout(function () {
    //alert(reqValue);
    topic.publish("application/d3slider/timeChanged", value);
  }, 500);
}
```

The code subscribing to the topic will set the map to the time extent that is pointed by the brush:

```
topic.subscribe("application/d3slider/timeChanged", function () {
  console.log("received:", arguments);
  var startDate = arguments[0];
  if (startDate) {
    var timeExtent = new TimeExtent();
    timeExtent.startTime = startDate;
    map.setTimeExtent(timeExtent);
  }
});
```

Find the full code listing to construct the D3 slider:

```
define([
  "dojo/_base/declare",
  "d3",
  "dojo/topic",
  "dojo/_base/array",
  "dojo/domReady!"
], function (
  declare,
  d3,
  topic,
  array) {
  //http://bl.ocks.org/zanarmstrong/ddff7cd0b1220bc68a58

  var isInitilaized = false;

  topic.subscribe("application/d3slider/initialize", function () {
    if (!isInitilaized) {
      console.log("received:", arguments);
      var startDate = arguments[0];
      var endDate = arguments[1];

      var formatDate = d3.time.format("%Y-%m-%d");
      var timer;
      // parameters
      var margin = {
        top: 10,
        right: 50,
        bottom: 50,
        left: 50
      },
```

```
width = 800 - margin.left - margin.right,
height = 150 - margin.bottom - margin.top;

// scale function
var timeScale = d3.time.scale()
  .domain([startDate, endDate])
  .range([0, width])
  .clamp(true);

// initial value
var startValue = timeScale(new Date('2012-03-20'));
startingValue = new Date('2012-03-20');

//////////

// defines brush
var brush = d3.svg.brush()
  .x(timeScale)
  .extent([startingValue, startingValue])
  .on("brush", brushed);

var svg = d3.select("#d3timeSliderDiv").append("svg")
  .attr("width", width + margin.left + margin.right)
  .attr("height", height + margin.top +
margin.bottom)
  .append("g")
  // classic transform to position g
  .attr("transform", "translate(" + margin.left +
  "," + margin.top + ")");

svg.on("mousedown", function (data) {
  var value = brush.extent()[0];

  if (d3.event.sourceEvent) {
    // not a programmatic event
    value = timeScale.invert(d3.mouse(this)[0]);
    brush.extent([value, value]);
  }
  console.log(formatDate(value));
});

svg.append("g")
```

```
.attr("class", "x axis")
// put in middle of screen
.attr("transform", "translate(0," + height / 2 +
)")")
// introduce axis
.call(d3.svg.axis()
  .scale(timeScale)
  .orient("bottom")
  .tickFormat(function (d) {
    return formatDate(d);
  })
  .tickSize(0)
  .tickPadding(12)
  .tickValues([timeScale.domain()[0],
timeScale.domain()[1]]))
.select(".domain")
.select(function () {
  console.log(this);
  return
  this.parentNode.appendChild(this.cloneNode(true));
})
.attr("class", "halo");

var slider = svg.append("g")
  .attr("class", "slider")
  .call(brush);

slider.selectAll(".extent,.resize")
  .remove();

slider.select(".background")
  .attr("height", height);

var handle = slider.append("g")
  .attr("class", "handle");

handle.append("path")
  .attr("transform", "translate(0," + height / 2 +
)")")
  .attr("d", "M 0 -20 V 20");
```

```
        handle.append('text')
            .text(startingValue)
            .attr("transform", "translate(" + (-45) + " ," +
(height / 2 - 25) + ")");

        slider.call(brush.event);

        function brushed() {
            var value = brush.extent()[0];

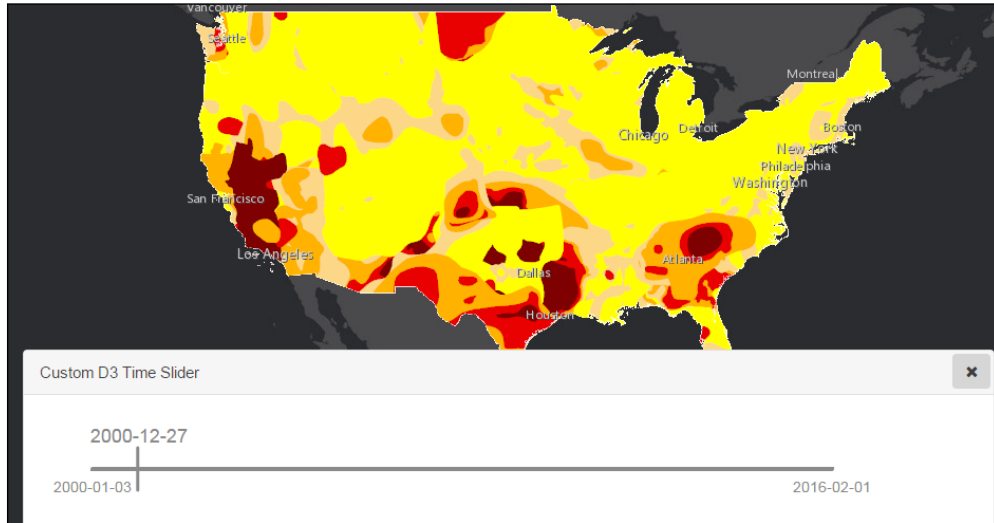
            if (d3.event.sourceEvent) {
                // not a programmatic event
                value = timeScale.invert(d3.mouse(this)[0]);
                brush.extent([value, value]);
            }

            handle.attr("transform", "translate(" +
timeScale(value) + ",0)");
            handle.select('text').text(formatDate(value));
            var reqValue = formatDate(value);

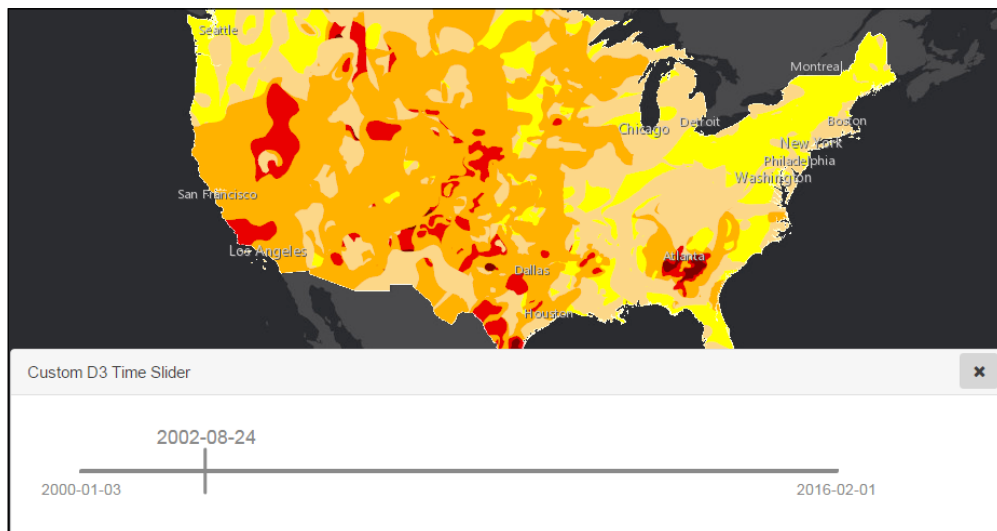
            if (timer) {
                clearTimeout(timer);
            }
            timer = setTimeout(function () {
                //alert(reqValue);
                topic.publish("application/d3slider/timeChanged",
value);
            }, 500);

        }
        isInitilaized = true;
    });
});
```


When we incorporated the previous piece of code in our application, we got this nice D3 Time Slider as seen in the following image, which lets us query through a continuous time spectrum, instead of yearly stops:



The query to fetch the dynamic map image is not fired as we move or even temporarily halt the D3 brush (thumb) to different positions along the time slider. It is only fired when we let the thumb stay at a location for more than 0.5 seconds. This is a safe trade-off between performance and responsiveness. The following screenshot shows the dynamic map image at an instance of time (24th August 2002):



Advanced spatio-temporal visualization with Cedar

A time aware layer provides valuable information about the data – the entire set of values for the features at each time stop for each feature. Until now, we have been visualizing the entire spatial dataset at different time zones using a Time Slider or a similar approach in D3. We have never been able to visualize the values for a particular feature across the entire time extent, or at least across multiple time extents. Our objective in this section would be just that – to visualize the values of a selected feature across the entire time extent.

We will be using the following layer for our visualization purposes, available at http://earthhobs1.arcgis.com/arcgis/rest/services/US_Drought_by_County/FeatureServer/0.

This layer shows the USA Drought intensity from 2000 to the present by county. The temporal range of data is 01/04/2000 to the present and is updated every Thursday to reflect the conditions occurring the previous week.

Our objective is to pull all the data for a selected feature. The following steps can be followed to arrive at our objective:

1. Select a feature and perform an identify task on it to get the feature ID.
2. Use the feature ID to query the previous feature layer.
3. Pass the data to Cedar charts of the type `time`.

The following piece of code explains how identify parameters are formed. The identify task is performed at each map click:

```
function initIdentify () {
    map.on("click", doIdentify);

    identifyTask = new IdentifyTask("http://server.
arcgisonline.com/arcgis/rest/services/Demographics/USA_1990-2000_
Population_Change/MapServer");

    identifyParams = new IdentifyParameters();
    identifyParams.tolerance = 1;
    identifyParams.layerIds = [3];
    identifyParams.returnGeometry = true;
    identifyParams.layerOption =
IdentifyParameters.LAYER_OPTION_ALL;
    identifyParams.width = map.width;
    identifyParams.height = map.height;
}
```

The map click event calls the following function named `doIdentify()`:

```
function doIdentify (event) {
    map.graphics.clear();

    //Use the map click point for the identify task

    identifyParams.geometry = event.mapPoint;
    identifyParams.mapExtent = map.extent;
    identifyTask.execute(identifyParams, function (results) {
        console.log(results[0].feature.attributes);

    });

    //Initiate a Query Task

    var queryTask = new QueryTask("http://earthobs1.arcgis.com/
arcgis/rest/services/US_Drought_by_County/FeatureServer/0");
    var query = new Query();
    query.returnGeometry = true;
    query.outFields = ["*"];

    //Query based on the feature id returned by the identify task

    query.where = "countycategories_admin_fips = '"+results[0].
feature.attributes.ID+"'";
    query.orderByFields = ["countycategories_date"];
    queryTask.execute(query).then(function(qresult) {
        console.log(qresult);

    });

    //Send the query result to the topic "some/topic"

    topic.publish("some/topic", qresult);
});
}
```

The topic that sends the query data shall be subscribed by the function that will construct the Cedar chart. The Cedar chart type required is `time`.

The `time` type Cedar chart expects the following types of fields in the field mapping:

- Esri date time field
- Any numerical value

In our case, we will map the fields, namely `countycategories_date` (date time field) and `countycategories_d0` (numeric field):

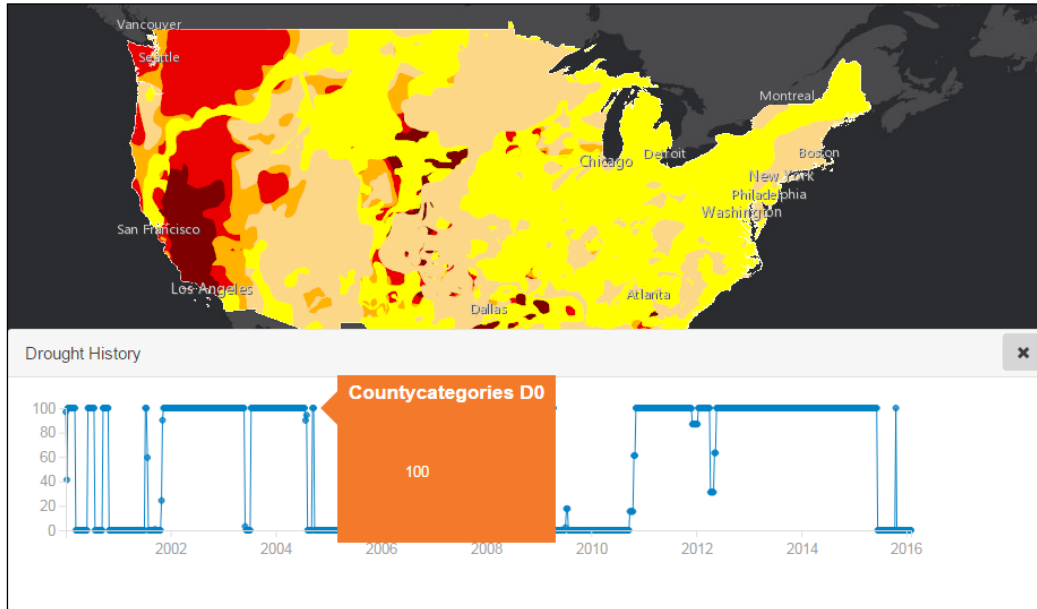
```
topic.subscribe("some/topic", function () {
    var data = arguments[0];
    var chart = new Cedar({
        "type": "time"
    });
    var dataset = {
        "data": data,
        "mappings": {
            "time": {
                "field": "countycategories_date",
                "label": "Date"
            },
            "value": {
                "field": "countycategories_d0",
                "label": "Countycategories D0"
            },
            "sort": "countycategories_date"
        }
    };

    //tool tip field
    chart.tooltip = {
        "title": "Countycategories D0",
        "content": "{countycategories_d0}"
    };

    chart.dataset = dataset;

    //show the chart
    chart.show({
        elementId: "#droughtHistoryMap",
        autolabels: true,
        height:150,
        width:800
    });
    chart.on('click', function(event,data) {
        console.log(event,data);
        topic.publish("application/d3slider/timeChanged", new
        Date(data.countycategories_date));
    });
});
```

Incorporating the previous code into the application, we were able to see the time-based graph of the drought values over a period for a selected county. In the following image, the graph represents the timeline of drought values for the selected feature:



This representation gives a multidimensional perspective in more than one way. One is that we are still seeing the spatial distribution of drought for the entire country at a particular instance of time. At the same time, we are able to use a non-spatial visualization aid, such as a time-graph, to visualize the entire set of drought values throughout a time period for a particular feature at county level.

Summary

We have covered how data can be visualized in a spatio-temporal fashion using three methods, namely Time Slider, D3, and Cedar. While Time Slider is an in-built API provided `dijit`, the D3 solution is much more extensive and flexible. Charting space-time data using the Cedar `time` type chart offers a different perspective of space-time data. We started with the foundations of the API and progressed steadily into the nuances of building a fully-fledged dojo web app with widgets. We dealt with versatile query capabilities provided by the API and have used it throughout the chapters in different forms. Displaying the query results was our later focus. The query results can be displayed as a spatial graphic as well as in tabular form. We later delved into more intuitive ways of rendering our spatial data displayed on the map using rendering techniques.

We then realized that a bit of statistical knowledge would not only help us understand the data better, but also visualize it better so that the user can derive new insights into the data. The last three chapters have been about adding multiple dimensions with the aid of non-spatial components, such as charting techniques and time dimension, to our maps. This chapter culminates at perceiving our maps in all the discussed dimensions, but certainly this is not the limit. Rather, this is the starting point for an enterprising map data scientist like you!

Index

A

- Active Wildfire Data**
 - reference link 61
- advanced spatio-temporal visualization**
 - Cedar, using 251-254
 - reference link 251
- AMD+**
 - about 20
 - define method 21
 - key components 21
 - require method 22
- appid**
 - reference link 161
- ArcGIS Developer**
 - account, reference link 170
 - reference link 61
- ArcGIS DynamicMapService layer 42-46**
- ArcGIS JavaScript API**
 - reference link 6
- ArcGIS Server**
 - about 24-34
 - service, types 25
- ArcGIS Tiledmap service layer 38-40**
- Asynchronous Modular Definition. *See***
 - AMD

B

- BaseMapGallery widget 56**
- Basemap layers 36**
- BlendRenderer 138**
- bootstrap-map-js**
 - reference link 66

- Brackets 1**
- break renderers**
 - working with 180

C

- CartographicLineStyle property**
 - reference link 153
- Cedar**
 - about 224
 - libraries, loading 224
 - used, for advanced spatio-temporal visualization 251, 252
 - used, for charting 224
- Cedar libraries**
 - loading 224
 - loading, AMD pattern used 225-232
 - loading, Script tags used 225
- chart**
 - plugins, adding 208-212
- class breaks renderer 136, 137**
- ClassBreaksRenderer**
 - setColorInfo(colorInfo) method 180
 - setOpacityInfo(opacityInfo) method 180
 - setRotationInfo(rotationInfo) method 180
 - setSizeInfo(sizeInfo) method 180
- classed renderers**
 - classification method 140, 141
- classification methods**
 - about 174
 - equal interval 174
 - natural breaks 174
 - Quantile 175
 - standard deviation 175

- color brewer 2**
 - reference link 182
- ColorInfo object**
 - about 182
 - classed color renderer, creating 185, 186
 - color scheme, selecting 182-184
- colors**
 - Esri color module 120, 121
 - RGB color model 119, 120
 - working with 119
- column chart**
 - creating, with D3 214, 215
- Content Delivery Network (CDN) 6**
- continuous renderers**
 - working with 180
- Coordinated Universal Time (UTC) 237**
- coordinate geometry**
 - about 12, 13
 - current map extent, obtaining 15-19
 - quiz results 14
 - quiz, solving 13
 - spatial reference systems 14
 - template generator, for loading modules 19
- CSV files 34**
- custom widget**
 - building 89
 - dojo, configuring 90, 91
 - draw toolbar, using 106
 - folder structure 97
 - modules 105
 - query, executing 112
 - simple class, creating 89, 90
 - standalone widget, developing 93

D

- D3**
 - data 216
 - scaling 217
 - selections 215
 - used, for creating column chart 214, 215
 - used, for querying based on time 243
- D3 charts**
 - SVG, integrating into 218-224

- D3.js**
 - about 213
 - reference link 213
 - used, for charting 213, 214
- D3, used for querying based on time**
 - about 243
 - D3 brush 244-250
 - time, formatting 244
 - time, scaling 244

data

- pushing, into chart 208
- data sources, supported by API**
 - about 33, 34
 - ArcGIS Server 34
 - flat file formats 34

Deferred object

- displayFieldName property 81
- feature property 81
- layerId property 81
- layerName property 81

demographics analytic portal

- about 169
- building 169, 170
- reference link 170

development environment

- ArcGIS Developer account, setting up 4
- browser 2
- Hello, Map 4, 5
- IDE 2, 4
- jump-start code 6
- setting up 1
- web server 2

dijit life cycle

- constructor 94
- postCreate 94
- startup 94

dojo

- about 21
- charting, popup templates used 204
- chart themes 202
- configuring 90, 91
- modules 22
- reference link 10

- themes, reference link 202
- used, for charting 201
- dojo array module 23**
- dojo charting methods**
 - addPlot() 206
 - addSeries() 206
 - render() 206
 - setTheme() 206
- dojo dom modules**
 - dojo/dom-attr 23
 - dojo/dom-class 23
 - dojo/dom-construct 23
- dojo modules**
 - about 22
 - array module 23
 - dom modules 23
 - event handler module 23
- dojo packages**
 - reference link 226
- dojox modules**
 - 2D charts, types 206
- DotDensityRenderer 138**
- draw toolbar, custom widget**
 - draw-end event handler 109
 - drawn shape, symbolizing 109-111
 - draw operation 109
 - initiating 107, 108
 - using 106
- Drought app**
 - building 238
 - Time Slider, using 238
- drought intensity values**
 - reference link 238

E

- Esri color module 120, 121**
- Esri widgets**
 - BaseMapGallery widget 56
 - Legend widget 57
 - using 56

F

- feature layer**
 - about 47
 - FeatureLayer constructor 48, 50, 51
 - FeatureLayer.MODE_AUTO 49
 - FeatureLayer.MODE_ONDEMAND 48
 - FeatureLayer.MODE_SELECTION 49
 - FeatureLayer.MODE_SNAPSHOT 48
 - Infotemplates 52
 - reference link 69
- Feature Manipulation Engine (FME) 34**
- feature set**
 - features 77
 - geometryType 77
- feature table**
 - building 85
- Find task**
 - building 82
 - executing 82-84
 - instantiating 83
 - parameters, building 83
- flat file formats**
 - about 34
 - CSV files 34
 - Keyhole Markup Language(KML) 34
- functional classification, layers**
 - about 36
 - Basemap layers 36
 - functional layers 37
 - graphics layers 38
- functional layers 37**

G

- Geolocation API**
 - geometry engine, using on input
 - data 163, 164
 - using 162, 163
- geometry-based symbols**
 - SimpelFillSymbol 122
 - SimpleLineSymbol 122
 - SimpleMarkerSymbol 122

global wind data gauge

adding 154, 155

graphics layers

about 38-52

attributes 53

geometry 53

InfoTemplate 53

symbol 53

Greenwich Mean Time (GMT) 237

H

HeatmapRenderer 137, 138

hurricane tracking app

active hurricane layers, symbolizing 149

background 143, 144

building 148

I

IdentifyTask

building 80

executing 80, 81

identify parameters object, constructing 80

instantiating 80

IIS Express

download link 3

installation link 2

initial map extent

coordinate geometry 12

setting up 11

J

jump-start code

about 6

AMD pattern of coding 10

API reference link 6, 9

esri/map module 11

K

Keyhole Markup Language(KML) 34

L

latest active hurricanes

feature layer, refreshing 160

latest data, displaying on grid 159, 160

latest data, fetching 159

tracking 156, 157

unique list of storms, obtaining 157-159

layers

about 33, 35

adding, to map 35, 36

functional classification 36

map and layer events 55, 56

map and layer properties 54

types 38

Legend widget 57

M

map data

visualizing 144-148

map server

reference link 27

multivariate mapping 192

N

NOAA service

reference 39

O

opacityInfo object

about 189

used, for creating classes opacity

renderer 189

P

PictureMarkerSymbol module

about 126, 127

PictureFillSymbol 129

reference link 126

TextSymbol 129, 130

plot
defining 207

popups
building 85, 86
InfoTemplates, building 86, 87

project folders
code, modularizing 99
creating 98
dojoConfig, defining 99
internationalization support, providing 100, 101, 102
single point of entry, creating 98

proxy.config file
reference link 65

Q

query, custom widget
executing 112
query event handlers 113
Query object, initializing 112
QueryTask object, initializing 112

Query endpoint
reference link 28

query event handlers
about 113
graphics, adding to map 116, 117
HTML template, defining 113, 114
query results, symbolizing 115, 116

Query object
building 70, 71
querying, by spatial geometry 72

query operations
about 73
Query for Count operation 73-76
Query for Extent operation 79
Query for Features operation 76-78
reference link 75

Query task
building 68
executing 68
QueryTask object 70

QueryTask constructor
about 68
parameters 69

R

renderer methods
setColorInfo() 135
setOpacityInfo 136
setRotationInfo() 135
setSizeInfo() 136

renderers
about 119
BlendRenderer 138
class breaks renderer 136, 137
ClassBreaksRenderer 130
classification method , for classed
renderers 140, 141
DotDensityRenderer 130
HeatmapRenderer 130-138
reference link 131
ScaleDependentRenderer 131
selecting, for scenario 131
simple renderer 132, 133
SimpleRenderer 130
SmartMapping module 140
Stream Gauge application, developing 131
TemporalRenderer 131
UniqueValueRenderer 130
unique value renderer, applying 134, 135
working with 130, 131

resource-proxy
reference link 64

REST API
about 24
service, types 25

RGB color model 119, 120

RotationInfo object
about 191
arithmetic 191
geographic 191

S

Service Catalog

- about 25
- map server 27
- map server, Query endpoint 28-31
- working with 26

simple class

- anonymous class 90
- creating 89, 90
- named class 90

simple renderer 132, 133

SizeInfo object 191

smart mapping 196, 197

SmartMapping module 140

Spatial Reference

- about 40
- Extent and Scale Info 42
- TileInfo 42
- types 72

standalone widget

- developing 93
- dijit life cycle 94
- templated widgets, creating 95, 96

State-level data

- reference link 229

statistical functionality, API

- about 172
- classification methods 174
- feature layer statistics 176-178
- normalization 176
- statisticDefinition module 172, 173

statistical measures

- about 170
- Average statistic 171
- maximum statistic 171
- minimum statistic 171
- standard deviation 171
- standardization 172
- sum 171

StatisticDefinition module

- onStatisticField property 172
- outStatisticFieldName property 172
- statisticType property 172
- stddev property 172

Stream Gauge application

- data source 132
- developing 131

superclass 90

symbols

- ArcGIS Symbol playground 124, 125
- PictureMarkerSymbol module 126
- SimpleFillSymbol module 126
- SimpleLineSymbol 122, 123
- SimpleMarkerSymbol 123, 124
- working with 121, 122

T

template generator

- reference link 19

theme

- defining 207

TileInfo 42

tiles

- about 25
- reference link 25

time aware layers

- about 235, 236
- considerations 237
- need for 237

Time Data Cumulative property 237

Time Slider

- creating, steps 239-242
- using 238

types, layers

- about 38
- ArcGIS Tiledmap service layer 38-40
- feature layer 47, 48
- graphics layer 52
- Spatial Reference 40
- The ArcGIS DynamicMapService layer 42-46

types, querying operations

- about 67
- find task 68
- identify task 68
- query task 67

U

unique value renderer

applying 134, 135

W

weather widget

creating 161

data, displaying 164-167

Geolocation API, using 162

open weather API 161, 162

Well-known ID (wkid)

about 14

reference link 14

Well-known Text (wkt) 14

widget folder structure

about 97

guideline, for creating project folders 98

guidelines, for creating project folders 98

overview 103

widgets 33

Wildfire application

bootstrapping 66, 67

developing 60, 61

proxy, updating 64, 65

reference link 62

registering, in developer portal 61, 62

