Mastering ArcGIS Server Development with JavaScript

ESRI and its ArcGIS products have been industry leaders in digital map production and publication. ArcGIS Server lets you design, configure, and publish maps that can be viewed and edited through the Internet.

The book introduces you to ArcGIS Server and the ArcGIS JavaScript API. You'll create your first web map and improve it with ArcGIS modules. You will design custom Dojo AMD widgets for your map. You'll also learn how to edit geographic data in the browser.

Furthermore, you will learn different techniques such as creating charts and graphs with data from geographic features. You will see how to use the ArcGIS JavaScript API with other JavaScript libraries and CSS styling methods to make your map stand out. By the end of the book, you will discover how to make your application compatible with different devices and platforms and test it using the testing libraries.

Who this book is written for
This book is intended for intermediate developers who want to design web mapping applications. You should have some experience with geographic information systems, especially with ArcGIS products such as ArcGIS Server. It also helps to have some experience with HTML, CSS, and JavaScript.

What you will learn from this book
- Create single-page mapping applications, lining up data from different sources
- Search for and display geographic and tabular information based on locations and attributes
- Customize maps and widgets to deliver the best user experience
- Present location data intuitively using charts and graphs
- Integrate mapping applications with your favorite JavaScript frameworks
- Test the working of your webmap application and take advantage of cloud services such as ArcGIS Online
- Create modern-looking webmaps through styling tips and tricks
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 5 'Editing Map Data'
- A synopsis of the book’s content
- More information on Mastering ArcGIS Server Development with JavaScript
About the Author

Ken Doman has worked in the geographic information system field since 2007. In his first GIS job, he was given a stack of maps, some AutoCAD data, and a shoe box full of address notes and was told to make a web-accessible GIS application out of them. Since then, he has strived to make geographic data available and easily accessible over the Internet. He is an ESRI certified web application developer and has created many public web applications using ArcGIS software.

Currently, Ken works for Bruce Harris & Associates, Inc., a mapping and custom development company that specializes in land records and ESRI software solutions. There, he creates custom web, desktop, and mobile applications using JavaScript, .NET, and ArcGIS software.

He has also been a book reviewer for Building Web and Mobile ArcGIS Server Applications with JavaScript, Packt Publishing by Eric Pimpler and ArcGIS for Desktop Cookbook, Packt Publishing by Daniela Cristiana Docan. You can learn more about Ken at http://raykendo.com.
Preface

In this digital age, we expect maps to be available wherever we are. We search for driving directions on our phone. We look for business locations and nearby restaurants using apps. Maintenance personnel use digital maps to locate assets several feet underground. Government officials and C-level executives need up-to-date location data to make important decisions that affect many lives.

ESRI has been an industry leader for 30 years in the realm of digital mapping. Through products such as ArcGIS Server, they empower companies, government agencies, and digital cartographers to publish maps and geographic data on the Web. Websites and applications written with the ArcGIS API for JavaScript make those maps available on both desktop and mobile browsers. The API also provides the building blocks needed to create powerful applications for data presentation, collection, and analysis.

While there are plenty of examples, blog posts, and books to get you started in developing apps with the ArcGIS API for JavaScript, they don't often go that deep. They don't discuss the pitfalls, limitations, and best practices for developing applications with their tools. That next step is what this book tries to accomplish.

What this book covers

Chapter 1, Your First Mapping Application, introduces you to ArcGIS Server and the ArcGIS API for JavaScript. In this chapter, you will learn the basics of creating a map and adding layers.

Chapter 2, Digging into the API, gives an overview of the many API components, widgets, and features that are available within the ArcGIS API for JavaScript. This chapter offers explanations of how these components can be used.
Preface

Chapter 3, *The Dojo Widget System*, explores the Dojo framework, the comprehensive JavaScript framework the ArcGIS API for JavaScript is built in. This chapter explores the inner workings of Asynchronous Module Design (AMD) and how to build custom widgets for our mapping applications.

Chapter 4, *Finding Peace in REST*, examines the ArcGIS REST API, which defines how ArcGIS Server communicates with the application in the browser.

Chapter 5, *Editing Map Data*, covers how to edit geographic data stored in ArcGIS Server. This chapter examines the modules, widgets, and user controls provided by the ArcGIS JavaScript API.

Chapter 6, *Charting Your Progress*, looks at how you can convey map feature information through charts and graphs. The chapter not only discusses charts and graphs made with the Dojo Framework, but also looks at how to integrate other charting libraries, such as D3.js and HighCharts.js.

Chapter 7, *Plays Well with Others*, discusses ways to integrate other popular libraries into an application written with the ArcGIS API for JavaScript. This chapter also explores combining the framework with jQuery, Backbone.js, Angular.js, and Knockout.js.

Chapter 8, *Styling Your Map*, covers the use of CSS styling in the mapping application. This chapter examines how the Dojo framework lays out elements, and explores adding the Bootstrap framework to style the mapping application.

Chapter 9, *Mobile Development*, looks at the need to develop with mobile in mind and discusses the pitfalls of attempting to make your app "mobile-friendly". This chapter implements the dojox/mobile framework, provided by the Dojo framework, as a way to style elements for mobile use.

Chapter 10, *Testing*, discusses the need for test-driven and behavior-driven development. This chapter discusses how to test applications using the Intern and Jasmine testing frameworks.

Chapter 11, *The Future of ArcGIS Development*, looks at new mapping and application services such as ArcGIS Online and Web AppBuilder. You will learn how to develop an application around an ArcGIS Online webmap.
Data has to come from somewhere. In the multi-billion dollar geospatial industry, collecting data is expensive. Features visible from aerial photography need to be plotted, and features not so visible on a workstation need their GPS coordinates collected in the field. Your time is valuable, and data collection won't happen on its own.

But what if you could get others to do the work for you? What if you could create a website that let other people collect the information? Trained workers could document utility lines, or concerned citizens could report problem locations in town. By using volunteer data collection, you can quickly collect the data you need.

ArcGIS Server provides not only data visualization on a map, but editing capabilities as well. Services can be created, and applications can be built around them, which allow users to add items to a map, change their shape, edit their attributes, and even delete them. ArcGIS Server also gives the creator of the services control over which of those data changes are allowed.

In this chapter, we're going to do the following:

• Look at the data editing capabilities ArcGIS Server provides with the ArcGIS JavaScript API
• Learn about the editing controls in the API, and how they create a helpful editing experience
• Create an editing application that uses the tools in the ArcGIS JavaScript API
The use cases for webmap editing

A GIS professional doesn't have to edit all the map data on his own. Trained staff and eager volunteers can assist with data collection and map editing projects that interest them. As the developer, it's up to you to give them the tools they need to collect and update the data. The following are examples of applications which use web map editing that you can create:

- Field crews updating utility data
- Public service requests and incident reports
- Parcel classification reassignments after analysis
- Volunteer geographic information data collection

Map editing requirements

Editing geographic data using the ArcGIS JavaScript API requires some setup on ArcGIS Server. An editable feature service must be published on ArcGIS Server, which requires an ArcSDE geodatabase. File geodatabases, personal geodatabases, and shapefiles cannot be used to store editable data. ArcGIS Online allows you to upload editable data to ESRI's cloud service, but the data upload and editing process has requirements which are covered in Chapter 11, The Future of ArcGIS Development.

There are a few requirements for setting up an editable map application using ArcGIS Server and its JavaScript API. The geodatabase storing the data should be versioned, if you want to review the data before committing it to your default database. Versioned data also supports undo and redo operations. You may want to publish a read-only map service along with your editable feature service. Finally, some editing operations require a geometry service to handle geometry changes, as you cut, merge, and trim features.

Feature services

A feature service provides a web interface between data stored on the server and an application on the browser created to use it. They can be accessed through URL endpoints similar to map services. However, they produce very different results. They can be loaded on a map and queried much like dynamic or tiled services, but there is more. Feature services return graphics instead of tiles. These graphics can be queried, and even edited, if the service allows.
Feature templates

With ArcGIS Server 10.1, feature services can be published with the added functionality of **feature templates**. Feature templates give the user preconfigured features to add to the map. Feature templates are created in ArcMap, and define the symbology and predefined attributes. These templates make it easier to edit service data.

One example of Feature templates can be found on an animal sighting map. The points on the map designate where animal sightings take place. Feature templates could be created to show pictures of each major type of animal (cat, dog, bird, rabbit, deer, and so on). Values in some of the fields could be defined ahead of time. For instance, you could say that all cats are warm-blooded.

How do you, as the developer, take advantage of feature templates? Apart from demonstrating what each symbol means, there are template pickers in the ArcGIS JavaScript API's tools that not only show the feature templates, but also let you click on them and add them to your map.

Feature layer

The feature layer provides access to graphics within a feature class. The user can thus both query and edit the shapes and attributes of the graphics. We reviewed their REST service profile in Chapter 4, *Finding Peace in REST*. We load feature layers in much the same way we load dynamic and tiled services. However, their options often require more parameters, due to the editable nature of the content.

Feature service modes

When initializing a feature layer from a feature service, you have a choice as to how the data is loaded. Do you want to load it all at once? Do you want to load all the features that you can see? Do you only want to load the one you’ve selected, and not show the rest? In the next sections, we’ll review the three feature service modes used to download data to the client browser.

Snapshot mode

Sometimes, if there is not a lot of data, it's better to download it all at once. That's what snapshot mode does. Snapshot mode downloads feature data based on time definitions and definition expressions, but it is limited by the maximum download limit. The visibility of the data is then determined by time extent.
Snapshot mode is helpful if there is not a lot of data to download, or if connectivity may be an issue during use. The user can download all the feature data at once, work with it, and then save their changes when connections become favorable again.

**On demand mode**

Sometimes, you're only interested in downloading the data in front of you. In that case, on demand mode is the best option. On demand mode only downloads features within the map extent. They too are affected by time definitions and definition expressions. Unlike snapshot mode, data requests are made every time the map extent changes. On demand mode is the default mode for any FeatureLayer.

On demand mode is typically used when there is a lot of data in the feature layer, but the user is only expected to view a small portion of it. It's very good for focussed editing tasks. It's not as good for mobile applications with lots of map navigation and connectivity issues, since some graphics will fail to load.

**Selection mode**

Loading features by selection is more constraining because it only shows those features that have been selected. Feature selection is handled using the feature layer's selectFeatures() method, in a manner similar to querying from a map service layer. In this case, the graphics returned are considered "selected". Selection methods include clicking on the map and sending a query with specific parameters. This method is very helpful if there are lots of features, and you only want to download specific ones, whether it's by area or attributes.

**Editing tools**

The ArcGIS JavaScript API comes with a set of widgets and modules designed specifically for editing. With the editing widgets, the user can add features to the map, change their shape, edit their attributes, and even delete them, if the services allow. Let's look at some of the tools available in the API.

---

Why are the editing tools in your application not working? It may be the CSS. Editing widgets are created with Dojo user controls, or dijits. These controls require the Dojo stylesheets, such as claro.css or nihilo.css. Without them, buttons stop working, and other unexpected behaviors may arise.
The edit toolbar, loaded with the esri/toolbars/edit module, lets the user change the shape, orientation, scale, and position of graphics on a map. We discussed it in Chapter 2, Digging into the API, in relation to the other toolbars. Separate controls are required to save the changes made with the edit toolbar. You can see an image of a triangle selected for the edit toolbar here:

The edit toolbar requires a map in its constructor. The constructor also needs a number of optional parameters to modify its appearance and behavior. Many of the options depend on the geometry type of the data being manipulated. As of API version 3.13, here are some of the available options for the edit toolbar:

- **allowAddVertices** (boolean): If true, you can add vertices to a polyline or polygon.
- **allowDeleteVertices** (boolean): If true, you can remove vertices from a polyline or polygon.
- **ghostLineSymbol** (line symbol): When moving a line or polygon edge, this is the symbol for the line that shows where the new line/edge will go.
- **ghostVertexSymbol** (marker symbol): If you are allowed to add vertices, this is the symbol that shows where to click to insert a vertex.
- **textSymbolEditorHolder** (string or HTML DOMnode): Web page location when you want to add a text symbol editor widget
- **uniformScaling** (boolean): When true, resizing a polyline or polygon keeps the original ratio of width to height.
- **vertexSymbol** (marker symbol): When editing polylines and polygons, this is the symbol of the points at each vertex.
You can see an example of loading the edit toolbar in the following snippet:

```javascript
require([..., "esri/toolbars/edit",
"esri/symbols/SimpleMarkerSymbol",
"esri/symbols/SimpleLineSymbol"],
function (..., EditToolbar, MarkerSymbol, Linesymbol, ...) {

  var editTB = new EditToolbar(map,... {
    allowAddVertices: true,
    allowDeleteVertices: true,
    ghostLineSymbol: new Linesymbol(...),
    ghostMarkerSymbol: new MarkerSymbol(...),
    uniformScaling: false,
    vertexSymbol: new MarkerSymbol(...) 
  });

  });
}

When you want to use the edit toolbar to edit a feature, you call the activate() method. The activate() method requires two arguments, and has the option for a third. Firstly, the method requires a tool, which is made by joining a combination of the edit toolbar constants with the pipe | symbol. The constants include EDIT_TEXT, EDIT_VERTICES, MOVE, ROTATE, and SCALE. Secondly, the activate() method requires a graphic to edit. The final optional argument is an object similar to the one used to create the edit toolbar. In the following code snippet, we have a graphic that is added to the map, and a click event is assigned to it that activates the edit toolbar to edit the graphic when it is double-clicked:

```javascript
var editTB = new EditToolbar(...);

... map.graphics.add(myGraphic);
dojoOn(myGraphic, "dblclick", function () {
  editTB.activate(EditToolbar.EDIT_VERTICES | EditToolbar.MOVE | EditToolbar.ROTATE | EditToolbar.SCALE, myGraphic);
dojoOn.once(myGraphic, "dblclick", function () {
  editTB.deactivate();
});
});
```
Attribute inspector

Sometimes, you don't care where things are, you just care about the content. That's where the attribute inspector comes in. The attachment inspector provides a form with a list of editable fields and the appropriate blanks to edit them. The attachment inspector is bound to a feature layer, and displays the editable values for the selected layer. The fields in the attribute inspector respond to the field types of the attributes. Date fields show a calendar when editing. Fields with coded value domains show a drop-down list instead of a text blank. Below, you can see an example of an attribute inspector loaded in the popup, though it could be added to a separate HTML element.

When initializing an attribute inspector, you need to define how the inspector will handle the different fields within the graphic attributes. The attribute inspector constructor accepts an options object, and either an HTML element or an id string reference to the element. The options object has one parameter called layerInfos, which accepts an array of layerInfo objects. Each layerInfo object contains one or more of the following parameters:

- featureLayer (required): The feature layer to be edited.
- userId (string, optional): The ArcGIS Server user id connected to the service, should the editing require token authentication. This is not needed if you have used the Identity Manager to handle logins.
- showObjectId (Boolean, optional): Whether you want to see the object id of the feature when it is selected. By default, this value is false.
• **showGlobalID (Boolean, optional):** Whether you want to see the global id of the feature when it is selected. By default, this value is false.

• **showDeleteButton (Boolean, optional):** By default, the attribute inspector shows a delete button that lets you delete the selected feature. Setting this to false removes it.

• **showAttachments (Boolean, optional):** When set to true, and if the feature layer has attachments, this displays an attachment editor form in the attribute inspector, which lets you view and upload files attached to the feature.

• **isEditable (Boolean, optional):** Lets you control whether the feature is editable. This doesn't override whether the features are editable server-side. It's just an extra way to block someone without proper credentials from editing data they shouldn't.

• **fieldInfos (Objects [], optional):** Gives the developer granular control over what fields are editable, and how. This does not allow the user to edit fields that aren't allowed to be edited, according to the publishing method of the feature layer. If this value is not set, the attribute inspector lists all editable fields. FieldInfo objects contain the following:
  - **fieldname (string):** The name of the field to be edited
  - **format (object, optional):** An object that lets you edit time when editing dates. When set, add the following object: `{time: true}`
  - **isEditable (Boolean, optional):** When set to false, this disables the user's ability to change the value of that field
  - **stringFieldOption (string, optional):** When set, the user can edit a string value either in a single-line textbox, a text area with multiple lines, or a rich-text field that includes additional formatting
  - **label (string, optional):** When set, this lets you override the name of the field alias from the feature service
  - **tooltip (string, optional):** When set, this shows a text tool tip when the user begins editing the attribute
• You can see an example of an attribute inspector being loaded with a single feature layer here:

```javascript
var layerInfos = [{
  'featureLayer': bananaStandFL,
  'showAttachments': false,
  'isEditable': true,
  'format': {'time': true },
  'fieldInfos': [
    {'fieldName': 'address', 'isEditable':true, 'tooltip': 'Where is it?', 'label':'Address:'},
    {'fieldName': 'time_open', 'isEditable':true, 'tooltip': 'Time the Banana Stand opens.', 'label': 'Open:'},
    {'fieldName': 'time_closed', 'isEditable':true, 'tooltip': 'Time the Banana Stand closes.', 'label': 'Closed:'},
    {'fieldName': 'is_money_here', 'isEditable':false, 'tooltip': 'There\'s money in the Banana Stand.'}
  ]
}];

var attInspector = new AttributeInspector({
  layerInfos: layerInfos
}, "edit-attributes-here");
attInspector.startup();
```

While the attribute inspector allows you to edit the attributes of graphics on a map, it doesn't provide an immediate way to save the edits. It is up to the developer to determine when changes to attributes are saved to the server. The developer could add a save button, or save whenever the feature is no longer selected.
Template picker

The template picker lets the user select from a list of feature templates to add features to the map. It displays a grid of feature templates from connected feature layers. These templates include feature names, geometry types, and preset styles. The user can click on any of the template buttons, and then draw them on the map. You can load more than one feature layer, and switch between them with ease. You can see an example in the following screenshot:

The template picker, like most dijits, requires a parameter object and, either an HTML element or a string reference to the element's id, in order to load. In the options, the template picker accepts an array of `featureLayers`. It also accepts the number of `rows` or `columns` it will create. If you don't use `featureLayers` with their own feature templates, you can define your own using configured items in the `items` array. You can also set the CSS style directly. Finally, you can control whether tooltips show when you hover over the symbols. In the following snippet, you can see an example of a template picker initialized:

```javascript
var widget = new TemplatePicker({
    featureLayers: layers,
    rows: "auto",
    columns: 9,
    showTooltip: true,
    style: "height: 100%; width: 900px;"
}, "templatePickerDiv");
```

The preceding code shows a template picker with nine columns with tooltips to show data about the layers loaded in its `featureLayers` attribute. The size is 900 pixels wide, and as tall as it needs to be.
Attachment editor

There’s an old saying that a picture is worth a thousand words. Sometimes, you need that picture to explain what data you’re submitting. The attachment editor can help. The attachment editor allows the application to upload a file, usually an image, and connect it to the feature on the map. You can view other attachments, and possibly edit them or delete them, if permissions allow. Attachment editors can be loaded as part of the attribute inspector by setting the showAttachments property in the attribute editor options to true, when constructing the editor:

![Attachment editor example](image)

Editor dijit

The editor dijit provides an all-in-one editing tool to create, update, and delete map features. The editor dijit includes the template picker, attribute inspector, and an editing toolbar with numerous tools. It lets you draw new features on a map, edit existing features, and also delete features.

The tools that the editor dijit provides are as follows:

- Feature selection tools, either to add new selections, add to them further, or to remove from existing selections
- Feature drawing tools, including a tool to delete features from the map
- Tools that let you cut, merge, and clip parts of polylines and polygons
- Undo and redo operations (requires versioned map services)
Snapping manager

Here's a common request you might receive when creating an editing tool: "I'd like a tool that lets me edit this feature based on the lines of this other feature". You could try to code your own tool to select a feature and go through each phase of the feature. Or, with a few additional settings, you could implement the map's snapping manager.

The snapping manager imitates ArcMap snapping controls in the browser. As your mouse pointer approaches the corner or edge of a graphic feature, perhaps in a GraphicsLayer or a FeatureLayer, a new pointer moves over the point on the feature. This shows where you would add a point if you clicked on the map. You can click along a set of points, line vertices, or polygon corners to draw something that lines up perfectly with existing features with this tool.

When loading the snapping manager, there are a few important options that need to be set. Every snapping manager requires a map to snap to. It also requires a graphics layer or a feature layer to load, along with information about its snapping behavior. It should know whether to snap to the edge or vertex of a line or polygon, as well as whether to snap to points of a point feature class. All this information is added in a layerInfo array in its constructor options, or can be added later by using the setLayerInfos() method.

There are other optional configurable items in the snapping manager. You can tell the snapping manager to always snap to a graphic, or whether you want to control snapping by holding down a key on the keyboard while clicking. You can also configure which keyboard key is the snapKey, by loading that property with the appropriate dojo/keys constant. Finally, the tolerance of a snapping manager refers to the maximum number of pixels the pointer should be from the feature before it snaps to it.

You can see an example of a snapping manager loaded in a JavaScript API in the following code:

```javascript
require([...], "esri/SnappingManager", "dojo/keys", ...],
function (... , SnappingManager, dojoKeys ...) {

  var propertyLayer = new FeatureLayer({});
  var sm = new SnappingManager({
    alwaysSnap: false, // default: false
    map: map,
    snapKey: dojoKeys.CTRL, // default: dojoKeys.copyKey
    tolerance: 10, // default: 15
    layerInfo: [{
```
layer: propertyLayer, // this is a featureLayer,
snapToEdge: false, // default: true
snapToVertex: true //default: true
}
});
});
...
});

The preceding example shows a snapping manager that turns on when the user holds down the Ctrl key on a PC (the Command key on a Mac). It only snaps to the corners of a line or polygon in the propertyLayer feature layer. The tolerance for snapping was set to 10 pixels.

**Securing editing services**

If you're going to open up your data to be edited by the public, you need to be prepared for trouble. From bad data input to malicious attacks, you, as a developer, need to account for things going wrong. Luckily, ArcGIS Server and the ArcGIS API for JavaScript can help.

**Restricting user input**

I remember a project where we had to let users search for addresses based on a list provided by another system. The other system had no restrictions on what the user could enter. As a result, the address list was anything but normal. On a given street, there could be fifteen different ways the street name could be listed. Some were all caps, while others had "Rd" instead of "Road" Others were misspelled, one m instead of two, and some had too many spaces between the street name and the suffix. Needless to say, the data was poorly constructed and unstandardized.

ArcGIS Server provides some tools to help you restrict user input. Implementing coded value domains and ranges in the geodatabase can help reduce bad input. The attribute inspector honors field properties such as length and data type. You can set default values to limit extra user input in the feature service feature templates,

You can also tie in validation and other controls to make sure the user does not accidently do something like add a phone number to a date column. Dojo comes with user controls such as validation textboxes that limit bad input.
Password protected services

ArcGIS Server also offers a better option when it comes to securing your editing services. If you want to restrict access to editing data, you can demand token-based authentication for map services. A token is an encrypted string that contains a user name, an expiration date, and extra information for verification purposes. You need to request a token from http://myserver/arcgis/tokens, where myServer refers to your ArcGIS Server web endpoint or web adaptor. You submit the necessary user name and password before having the token added as a cookie on your browser. Tokens are only good for a limited time, which can be adjusted through configurable settings in ArcGIS Server.

These token-based security measures work with both map services and editable feature services. Without the token, you are not able to see the protected map services in the browser. With it, you can explore secured services, query them, and even edit data in them.

Identity manager

The identity manager (esri/IdentityManager) is used to handle logins and the security of ArcGIS Server and ArcGIS Online services. The identity manager displays a username and password prompt when you attempt to load token-protected services in the browser. Its user interface uses Dojo UI controls, so loading the appropriate Dojo style sheet is necessary to make the identity manager work properly.

Now that we've reviewed some of the editing capabilities ArcGIS Server offers, let's apply what we've learned to an application.

A new mapping application

So, back to our story and our mapping application. We're still waiting for word from the Y2K society about the Census map, but we have a new application we've been asked to work on. It seems that the city of Hollister, California has asked us to put an app together for them. Let's find out what they want.

The city of Hollister wants to create an application that lets citizens report issues in the city. They want citizens to report things like graffiti, sidewalk, curb, and street issues, damaged property, sewer issues, and tree problems, on a map, and also supply additional information. If possible, they want photos of the problems so crews know what to look for.
The file setup

We don't need to create any custom packages because we're going to use the out-of-the-box ArcGIS JavaScript API editing tools. Instead, we'll create a simple file setup with a css and a js folder. We'll add our custom style.css style sheet in the css folder. We'll add our app.js file in the js folder. We'll also add a folder named proxy to handle our proxy service. The file structure should look something like the following:

![File structure](image)

The front page

Let's start with the HTML document. We'll use our basic cookie-cutter site. This time, we'll add Dojo's claro.css style sheet. We don't need any custom packages, so we can leave those out of the dojoConfig file. We'd like a long column going down the side of the page, and a header part where we'll put the title for the page layout. We'll load the BorderContainer with a sidebar design with the taller side columns. We'll add three ContentPanes for the header, a leading column for the buttons, and a center region for the map.

```html
<!DOCTYPE html>
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
  <meta http-equiv="X-UA-Compatible" content="IE=Edge" />
  <meta name="viewport" content="initial-scale=1,
maximum-scale=1,user-scalable=no" />
  <title>Incident Reporting App</title>
  <meta name="description" content=""/>
  <meta name="author" content="Ken Doman">
  <link rel="stylesheet" href="http://js.arcgis.com/3.13/
dijit/themes/claro/claro.css">
```
Editing Map Data

We'll add some basic styling for the HTML and the body on the style.css page. Let's add the following style:

```css
html, body {
  width: 100%;
  height: 100%;
  border: 0;
  margin: 0;
  padding: 0;
  box-sizing: border-box;
  font-family: Helvetica, Arial, sans-serif;
}

*, *:before, *:after { box-sizing: inherit;}
```
We’ve set the width and height of the HTML and body to 100%, with no border, margin, or padding. We’ve also changed the font to a common sans-serif font, such as Helvetica, Arial, or just plain sans-serif. Finally, we set the elements on the page to be sized using border-box box-sizing which makes it easier to work with sizing boxes on the page.

**Loading the map**

We'll begin writing the code for our application with the page setup. We have a focus area, which is the city of Hollister. For the sake of an easy life, we’ll add the city boundaries as an extent:

```javascript
require([  
    "dojo/parser", "esri/map", "esri/graphic", 
    "esri/geometry/Extent", "esri/dijit/editing/Editor", 
    "esri/dijit/editing/TemplatePicker", "esri/tasks/query", 
    "dijit/layout/BorderContainer", "dijit/layout/ContentPane", 
    "dojo/domReady!"  
], function (parser, Map, Graphic, Extent, Editor, TemplatePicker, Query) {

    var maxExtent = new Extent({
        "xmin":-13519092.335425414,
        "ymin":4413224.664902497,
        "xmax":-13507741.43672508,
        "ymax":4421766.502813354,
        "spatialReference":{"wkid":102100}
    }),
    map, selected, updateFeature, attInspector;

    parser.parse();

    map = new Map("map", {
        basemap: "osm",
        extent: maxExtent
    });

});
```

In the preceding code, we've loaded the necessary modules and used the dojo/parser to parse them. We've added a map with the OpenStreetMap based basemap, and we've created a maxExtent to simulate the city boundaries.
Adding the map layers

Now that we have our map, we need to add layers to the map. For the sake of this exercise, we’re going to use the San Francisco 311 Feature Service provided by ESRI. We’re going to load the feature layer in selection mode, so we only affect the features we click on. We’re also going to add the complementary dynamic map service, because we can’t see the features without it. We will also set the feature layer selection symbol using a simple marker symbol to color the features we click on:

```javascript
require([...
    "esri/layers/FeatureLayer",
    "esri/layers/ArcGISDynamicMapServiceLayer",
    "esri/symbols/SimpleMarkerSymbol",
    ...], function (..., FeatureLayer, ArcGISDynamicMapServiceLayer, MarkerSymbol, ...)

    var maxExtent = ...,
    map, incidentLayer, visibleIncidentLayer;

    incidentLayer = new FeatureLayer("http://sampleserver3.arcgisonline.com/ArcGIS/rest/services/SanFrancisco/311Incidents/FeatureServer/0", {
        mode: FeatureLayer.MODE_SELECTION,
        outFields: [
            "req_type", "req_date", "req_time", "address", "district",
            "status"],
        id: "incidentLayer"
    });

    visibleIncidentLayer = new ArcGISDynamicMapServiceLayer("http://sampleserver3.arcgisonline.com/ArcGIS/rest/services/SanFrancisco/311Incidents/MapServer");

    map.addLayers([visibleIncidentLayer, incidentLayer]);
```
When the map layers are added, we can finally interact with them, both as a user and as a developer. We'll add an event listener called `startEditing()` to the map's `layers-add-result` event. We'll set up the editing events for the feature layer there. We'll add a map click event that draws a feature if something has been selected from the menu on the side of the page. Be sure to add this after the layers are defined, but before they are added to the map.

```javascript
var ..., incidentLayer, visibleIncidentLayer, selected;
visibleIncidentLayer = ...;

function startEditing () {
    var incidentLayer = map.getLayer("incidentLayer");
    // add map click event to create the new editable feature
    map.on("click", function(evt) {
        // if a feature template has been selected.
        if (selected) {
            var currentDate = new Date();
            var incidentAttributes = {
                req_type: selected.template.name,
                req_date: (currentDate.getMonth() + 1) + "/" + currentDate.getDate() + "/" + currentDate.getFullYear(),
                req_time: currentDate.toLocaleTimeString(),
                address: ",",
                district: ",",
                status: 1
            };
            var incidentGraphic = new Graphic(evt.mapPoint, 
                selected.symbol, incidentAttributes);
            incidentLayer.applyEdits([incidentGraphic],null,null)
        }
    });

    incidentLayer.setSelectionSymbol(
        new MarkerSymbol({color:[255,0,0]})
    );

    map.infoWindow.on("hide", function() {
        incidentLayer.clearSelection();
    });
}
```
In the preceding code, we've created a callback function called `startEditing()`. This causes the application to add a new graphic to the editable feature layer whenever the map is clicked. Default attributes and a symbol are applied to the editable feature. The editable feature layer clears its selection whenever the popup is hidden. Also, when the edits are complete, the visible layer is refreshed with the new data. The `startEditing()` method is assigned to run when a group of layers are added, which causes the layers to be added to the map.

### Using the proxy

If you try to load the map right now, you may get an error. If you don't get it now, you might get it when you try to save changes on the map. The reason is that these editing operations often require a proxy application to handle data which is too large to fit in the approximately 2,048 character limit of most browser get requests.

Proxies come in three varieties, based on your application environment. ESRI provides proxy services in PHP, Java, and .Net. We'll add a reference to the proxy in our application. This example shows how it's done with a .Net based proxy:

```javascript
require([
    ...
], function (_, esriConfig, ...) {
    ...
    // set up proxy for the featureLayer
    esriConfig.defaults.io.proxyUrl = "./proxy/proxy.ashx";

    incidentLayer = ...
});
```
Finding the user's location

Our client requested that the app provide the user with the ability to find them on the map, should they be using a mobile device or a laptop on Wi-Fi. We can provide that functionality by adding an ArcGIS dijit called LocateButton. We load the module in our application, initialize it when the map is ready, and it's good to go. The code to load it should look something like this:

```javascript
require([..., "esri/dijit/LocateButton", ...], function (... , LocateButton, ...) {
    ...
    function startEditing() {
        // add the Locate button
        var locator = new LocateButton({map: map}, "locatebutton");
    }
    ...
});
```

If we insert a `<div>` with an id of `locatebutton` inside the map ContentPane, and view the page in our browser, we'll see the locate button above the map, pushing the map down. We'd much rather locate it near the other zoom in and out buttons. We'll add the following styling to the `style.css` sheet to achieve that:

```css
.LocateButton {
    position: absolute;
    left: 29px;
    top: 120px;
    z-index: 500;
}
```

The template picker

For our application, we're going to use the ArcGIS JavaScript API's template picker to select incident point types to add to the map. We'll load them in the side pane on the page, and make them one column wide to add features. We'll pass that feature template to the `selected` variable when the feature template is selected. Finally, we'll load all this when both the map and the feature layers have loaded:

```javascript
function generateTemplatePicker(layer) {
    console.log("layer", layer);
    var widget = new TemplatePicker({
        featureLayers: [ layer ],
        rows: layer.types.length,
        columns: 1,
    });
...
```
The attribute inspector

Now that we are able to add new features to the map, we need a way to edit the content of those features. To do that, we'll add the attribute inspector. We're going to initialize the attribute inspector and tie it to the map's infoWindow.

```javascript
function startEditing () {
    var incidentLayer = map.getLayer("incidentLayer");
generateTemplatePicker(incidentLayer);
}
```

```javascript
function generateAttributeInspector(layer) {
    var layerInfos = [{
        featureLayer: layer,
        showAttachments: true,
        isEditable: true,
    }];

    attInspector = new AttributeInspector({
        layerInfos: layerInfos
    }, domConstruct.create("div", null, document.body));
```
attInspector.startup();

// add a save button next to the delete button
var saveButton = new Button({ label: "Save", "class": "saveButton" });
domConstruct.place(saveButton.domNode, attInspector.deleteBtn.domNode, "after");

saveButton.on("click", function () {
    updateFeature.getLayer().applyEdits(null, [updateFeature], null);
});

attInspector.on("attribute-change", function (evt) {
    // store the updates to apply when the save button is clicked
    updateFeature.attributes[evt.fieldName] = evt.fieldValue;
});

attInspector.on("next", function (evt) {
    updateFeature = evt.feature;
    console.log("Next " + updateFeature.attributes.objectid);
});

attInspector.on("delete", function (evt) {
    evt.feature.getLayer().applyEdits(null, null, [updateFeature]);
    map.infoWindow.hide();
});

if (attInspector.domNode) {
    map.infoWindow.setContent(attInspector.domNode);
    map.infoWindow.resize(350, 240);
}

function startEditing () {
    var incidentLayer = map.getLayer("incidentLayer");
generateTemplatePicker(incidentLayer);
generateAttributeInspector(incidentLayer);
    ...
We'll need to add a little positioning style to the save button on the attribute inspector. We'll add this entry to position the save button in the style.css sheet so that it doesn't overlap the delete button.

```css
.saveButton {
  margin: 0 0 0 48px;
}
```

Now that the attribute inspector is loaded, we can incorporate it into the click events for both the map layer and the incident layer. We'll create a `showInspector()` function that accepts a map click event. It will query the `incidentLayer` for any features in that location, and pull up a map `infoWindow` with the attribute inspector inside. It will also assign the selected graphic (if any) to:

```javascript
... 
function showInspector(evt) {
  var selectQuery = new Query(),
      point = evt.mapPoint,
      mapScale = map.getScale();

  selectQuery.geometry = evt.mapPoint;

  incidentLayer.selectFeatures(selectQuery,
    FeatureLayer.SELECTION_NEW, function (features) {
      if (!features.length) {
        map.infoWindow.hide();
        return;
      }

      updateFeature = features[0];

      map.infoWindow.setTitle(updateFeature.getLayer().name);
      map.infoWindow.show(evt.screenPoint,
        map.getInfoWindowAnchor(evt.screenPoint));
    });
  
  ... 
}
... 
function startEditing() {
  ...
  map.on("click", function (evt) {
    ...
    if (selected) {
      ...
      incidentLayer.applyEdits([incidentGraphic],null,null)
        .then(function () {
          ...
```
showInspector(evt);
});

} else {
  showInspector(evt);
}

});

incidentLayer.on("click", showInspector);

The result of the preceding code is shown in the following screenshot:
Securing the application

Now that we have a working incident reporting application, it's time to think about how we can secure the application. It's a public application, so the public should be allowed to submit problems. However, we don't want data that doesn't fit our data schema, or our representative boundaries.

One way we can secure our application from bad input is to restrict the locations where we accept changes. We don't want employee time wasted investigating complaints logged outside the city, state, or even country. We can do this by using the city extent supplied at the beginning of the application. We can test if the clicked point is inside the city extent in the click event, and notify the client if it's not. That should look something like the following:

```javascript
map.on("click", function (evt) {
  // if the clicked point isn't inside the maxExtent
  if (!maxExtent.contains(evt.mapPoint)) {
    alert("Sorry, that point is outside our area of interest.");
    return; // disregard the click
  }
}
```

Speaking of working with extents, we could also lock the selection buttons when navigating outside the city area. This would alert the user that we aren't accepting complaints outside the city proper. Of course, we should notify the user why they are locked out.

We'll start by adding blocking content and notifications in the HTML. We'll add two <div> to the page, a <div> with the id `outsidemessage` in the map, and a div with the id `blockerdiv` next to the editor div. We'll leave the two of them hidden by default, by adding the inline style `display: none`. It should look like the following:

```html
<html>
<head>
</head>
<body>
  ...
  <div id="map" data-dojo-type="dijit/layout/ContentPane"
   data-dojo-props="region:'center'" >
    <div id="locatebutton"></div>
    <div id="outsidemessage" style="display:none;">
      <p>Sorry, but you have navigated outside our city. Click on this message to get back to the city.</p>
    </div>
  </div>
  <div id="editpane" style="width: 130px"
   data-dojo-type="dijit/layout/ContentPane"
```
We'll add the following styling to the `style.css` file to style these items. The outside message will be gray block floating in the lower middle portion of the map, with text big enough to read, and with rounded corners (because lots of people like rounded corners). The blocking div will be positioned directly on top of the template picker buttons. The blocking div will be light gray, semi-transparent, and cover the entire template picker when visible.

```css
#blockerdiv {
    width: 100%;
    background: rgba(188, 188, 188, 0.6);
    position: absolute;
    top: 0;
    left: 0;
    bottom: 0;
    z-index: 500;
}

#outsidemessage {
    position: absolute;
    bottom: 40px;
    left: 50%;
    width: 200px;
    height: auto;
    margin: 0 0 0 -100px;
    background: rgba(255,255,255,0.8);
    padding: 8px;
    z-index: 500;
    border-radius: 8px;
    text-align: center;
    font-weight: bold;
```
We'll add some code to our app.js file to handle the visibility of these two nodes. We can listen for changes in the map's extent. When the map's extent is outside the city extent, and they no longer intersect, both the message div and the blocker div will be made visible (display: block;). If the user can see some of the extent of the viewing area, the div objects will be hidden again (display: none;). It should look like this:

```javascript
require([..., "dojo/dom-style", ...], function (... , domStyle, ...) {

  function onMapExtentChange (response) {
    if (!response.extent.intersects(maxExtent)) {
      // show these blocks if the map extent is outside the city extent
      domStyle.set("blockerdiv", "display", "block");
      domStyle.set("outsidemessage", "display", "block");
    } else {
      // hide these blocks if the max Extent is visible within the view.
      domStyle.set("blockerdiv", "display", "none");
      domStyle.set("outsidemessage", "display", "none");
    }
  }

  map.on("extent-change", onMapExtentChange);

  ...

  We'll also add an event handler to the outside message div that lets the user click to go back to the starting location for the map. We'll load the dojo/on event to handle the click event:

```javascript
require([..., "dojo/dom", "dojo/on", ...], function (... , dojoDom, dojoOn, ...) {

  ... dojoOn(dojoDom.byId("outsidemessage"), "click", function () {
    map.setExtent(maxExtent);
  })

  ...});
```

Now, when we load our application and pan our way outside the city limits, the following message should appear:
Limiting data entry

As well as stopping bad user input from clicks, we should also consider stopping bad user input from text editing. We can pull that off by modifying the `layerInfos` array assigned to the attribute inspector. We’ll start by getting rid of the delete button, since we don’t want citizens deleting everybody else’s complaints. We’ll also modify the `fieldInfos` list and set some of the fields to display only when they are edited in the attribute inspector. In this case, we’ll leave the `req_type`, `address`, and `district` tabs open for editing.

```javascript
var layerInfos = [{
    featureLayer: layer,
    showAttachments: true,
    isEditable: true,
    showDeleteButton: false,
    fieldInfos: [
        {fieldName: 'req_type', isEditable:true, tooltip: 'What\'s wrong?', label:'Status:'},
        {fieldName: 'req_date', isEditable:false, tooltip: 'Date incident was reported.', label:'Date:'},
    ]
}]
```
Editing Map Data

```json
{
    'fieldName': 'req_time',
    'isEditable': false, 'label': 'Time:'},
    {'fieldName': 'address', 'isEditable': true,
    'label': 'Address:'},
    {'fieldName': 'district', 'isEditable': true,
    'label': 'District:'},
    {'fieldName': 'status', 'isEditable': false,
    'label': 'Status:'}
}
```

These are a few of the simple things we can do to help secure our application against unwanted results, and yet still make the application user-friendly to the general public.

Summary

In this chapter, we've examined the tools and processes that ArcGIS Server and the ArcGIS API for JavaScript provide to make web editing possible. We looked at what goes into an editable feature service. We also looked into the various widgets that come with the ArcGIS JavaScript API for adding new features, editing geometries, and editing property attributes. We finished by creating an application that uses the editing tools to create an incident reporting application, with which users can report problems in the city on the map.

In the next chapter, we'll take the existing data and add a graphical twist.
Where to buy this book

You can buy Mastering ArcGIS Server Development with JavaScript from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.