ArcGIS API for JavaScript:

Programming Patterns and API Fundamentals

Tristan Morrison
René Rubalcava

Why choose the ArcGIS JS API?

Why choose the ArcGIS JS API?

- Seamless interfaces to ArcGIS Platform

Why choose the ArcGIS JS API?

- Seamless interfaces to ArcGIS Platform
- Client-side analysis

Why choose the ArcGIS JS API?

- Seamless interfaces to ArcGIS Platform
- Client-side analysis
- Rich, interactive data visualizations powered by WebGL

Why choose the ArcGIS JS API?

- Seamless interfaces to ArcGIS Platform
- Client-side analysis
- Rich, interactive data visualizations powered by WebGL
- 2D & 3D in one API

Why choose the ArcGIS JS API?

- Seamless interfaces to ArcGIS Platform
- Client-side analysis
- Rich, interactive data visualizations powered by WebGL
- 2D & 3D in one API
- Widgets help rapidly build a professional, well-tested UI

Map and View

Map and View

Map and View

```
const map = new Map({
  basemap: "topo"
});

const mView = new MapView({
  map: map,
  container: "viewDiv"
});
const sView = new SceneView({
  map: map,
  container: "viewDiv"
});
```

Basemaps and Ground

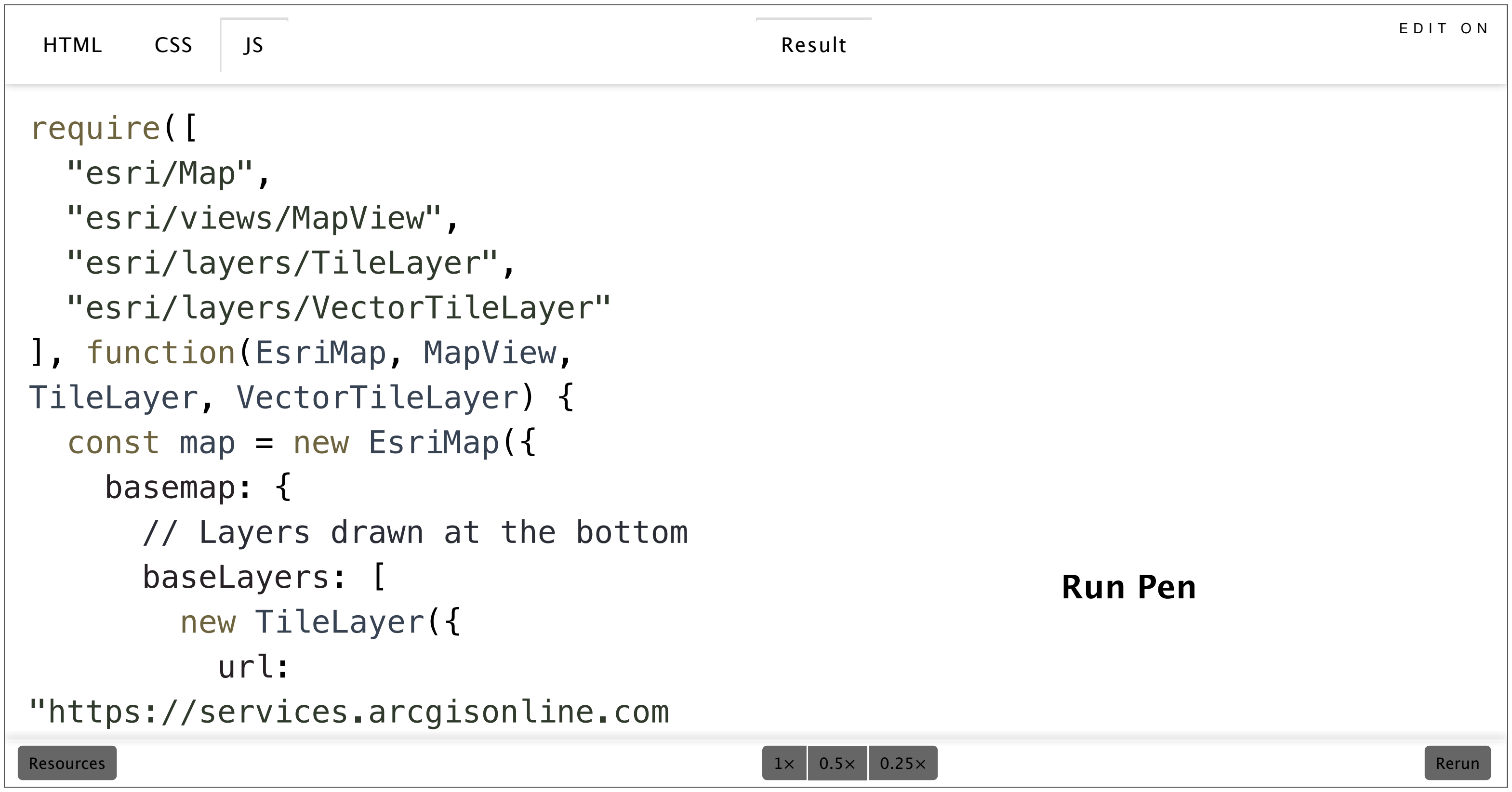
- Convenience Strings

```
const map = new Map({
  /*
   streets, satellite, hybrid, terrain, topo, gray,
   dark-gray, oceans, national-geographic, osm,
   dark-gray-vector, gray-vector, streets-vector, topo-vec
   streets-night-vector, streets-relief-vector, streets-na
   */
  basemap: "streets"

  /*
   world-elevation
   */
  ground: "world-elevation"
});
```

Basemaps and Ground

```
const map = new Map({
  basemap: {
    // Layers drawn at the bottom
    baseLayers: [
      new TileLayer({ url: baselayer })
    ],
    // Layers drawn on top
    referenceLayers: [
      new TileLayer({ url: refUrl })
    ],
  },
  ground: {
    layers: [
      new ElevationLayer({ url: elevationUrl })
    ]
```

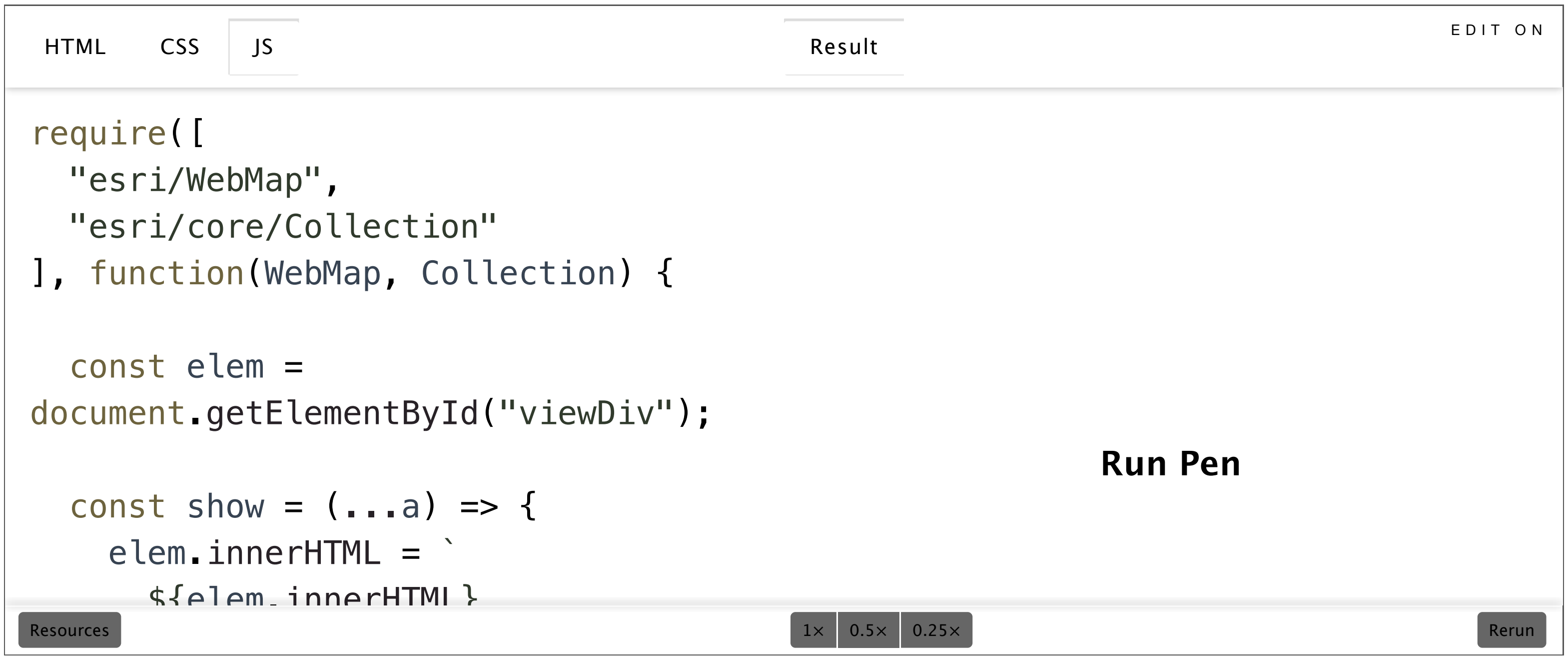
Basemap and Ground



```
require([
  "esri/Map",
  "esri/views/MapView",
  "esri/layers/TileLayer",
  "esri/layers/VectorTileLayer"
], function(EsriMap, MapView,
TileLayer, VectorTileLayer) {
  const map = new EsriMap({
    basemap: {
      // Layers drawn at the bottom
      baseLayers: [
        new TileLayer({
          url:
"https://services.arcgisonline.com
```

Collections

- `esri/core/Collection`



```
require([
  "esri/WebMap",
  "esri/core/Collection"
], function(WebMap, Collection) {

  const elem =
document.getElementById("viewDiv");

  const show = (...a) => {
    elem.innerHTML = `
      ${elem.innerHTML}
```

Working with Accessor

- Objects are have properties that can be:
  - read and set
  - or read-only
  - constructor arguments
  - watchable

Accessor - property access

```
layer.opacity = 0.5;
layer.title = "My test layer";

// setting multiple values
layer.set({
  opacity: 0.5,
  title: "My test layer"
});

// accessing the value of a deep property
view.get("map.basemap.title");
view.set("map.basemap.title", "new title");
```

Accessor - property watching

```
mapView.watch("scale", (newValue, oldValue, property, targ
  console.log(`scale changed: ${newValue}`);
});


mapView.watch("map.basemap.title", (newValue, oldValue, pr
  console.log(`new basemap title: ${newValue}`);
});


mapView.watch("ready, stationary", (newValue, oldValue, pr
  console.log(`property ${property}: ${newValue}`);
});

watchUtils.whenTrue(view, "stationary", () => {
```

watchUtils

Accessor - autocasting and single constructor

```
{
  type: "simple-marker",
  style: 'square',
  color: 'red',
  size: 10,
  outline: {
    color: 'rgba(255, 255, 255, 0.5)'
    width: 4
  }
});
```

Promises

Promises

- All asynchronous methods return a native promise
- The basic pattern looks like this:

```
layer.queryFeatures(query)
    .then(handleResult).catch(handleError);
```

Promises with async/await

```
const doQuery = async (query) => {
  const results = await layer.queryFeatures(query);
  const transformedResults = results.map(transformData);
  return transformedResults;
}
```

Dynamic Imports

```
async function loadMap(id) {
  const { default: WebMap } =
    await import('@arcgis/core/WebMap');
  return new WebMap({
    portalItem: { id }
  });
}
```

Promises

- Load resources
- Asynchronously initialized `Layer`, `WebMap`, `WebScene`, `View`

```
const map = new Map({...})

view = new SceneView({
  map: map,
  //...
});

view.when(() => {
  // the view is ready to go
});
```

Promises

```
view.when(() => {
  return view.whenLayerView(map.findLayerById("awesomeLaye
})
.then(layerView => {
  return watchUtils.whenFalseOnce(layerView, "updating");
})
.then(result => {
  const layerView = result.target;
  return layerView.queryFeatures();
})
.then(doSomethingWithFeatures)
.catch(errorHandler);
```

API sample

async/await

```
const init = async (doSomethingWithFeatures) => {
  await view.when();
  const layerView =
    await view.whenLayerView(map.findLayerById("awesomeLay
  await watchUtils.whenFalseOnce(layerView, "updating");
  const features = await layerView.queryFeatures();
  doSomethingWithFeatures(features);
};

try {
  init();
}
catch(error) {
  errorHandler(error);
}
```

Patterns

Interactivity with view events

- Use view events to interact with the view
- List of events
- You can stop the propagation of the event to prevent the default behavior

```
view.on("drag", event => {
  // user won't be able to drag
  event.stopPropagation();
})
```

Interactivity with view events

- Access the features on click

```
view.on("click", ({ x, y }) => {
  const screenPoint = {x, y};
  view.hitTest(screenPoint)
    .then(response => {
       // do something with the result graphic
       const graphic = response.results[0].graphic;
    });
});
```

- API Sample

goTo() with View

- Sets the view to a given target.
  - Navigate to a geometry/feature/location
- API Sample

Loadables

- brings better control, and scheduling of loading resources.
- the views automatically loads the map and its layers

Loadables

- `WebMap` / `WebScene` need to load:
  - the portal item
  - the layer module
  - the layer's item
- `MapView` / `SceneView` need to load:
  - the map
  - the layers

```
// In a single page application, get a feature from a Fe
// from a WebMap without displaying it, ASAP!
const webmap = new WebMap({
  portalItem: {
    id: 'affa021c51944b5694132b2d61fe1057'
  }
});

webmap.load()
  .then(() => {
    return webmap.findLayerById('myFeatureLayerId').loa
  })
  .then(featureLayer => {
    return featureLayer.queryFeatures({
      where: 'OBJECTID = 1'
```

Zoom or Scale

```
const view = new MapView({
  container: "viewDiv",
  map: map,
  center: [-116.5, 33.80],
  zoom: 14 // what does that really mean?
});
```

- Zoom = LOD (Level of Details)
- Not all LODs are created equal

Zoom is not Scale

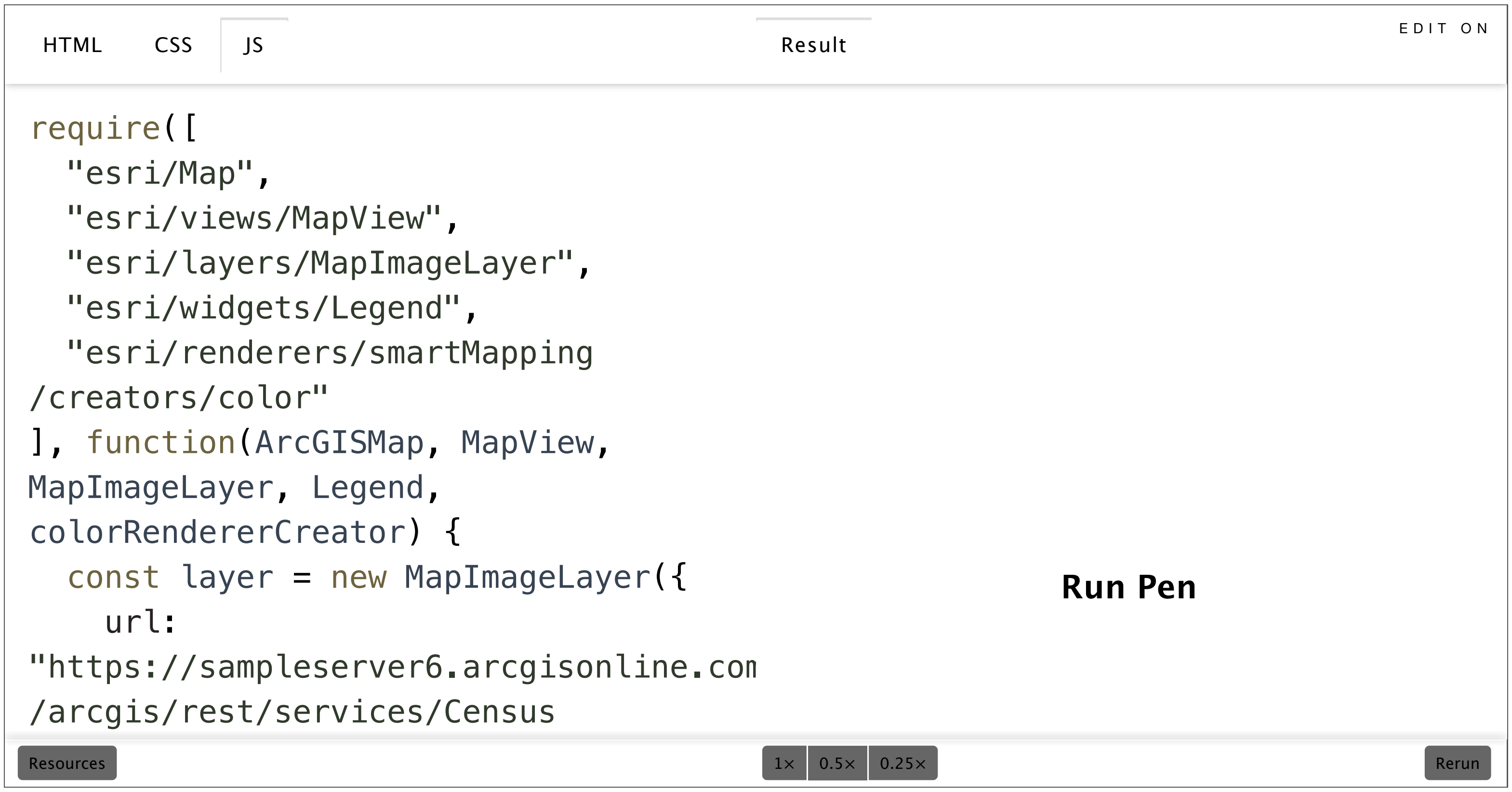
```
const view = new MapView({
  container: "viewDiv",
  map: map,
  center: [-116.5, 33.80],
  scale: 50000 // I know what that means!
});
```

- Scale is portable
- Scale has meaning
- We still snap to closest LOD/zoom

Sublayer to FeatureLayer

- You can extract a FeatureLayer from MapImageLayer Sublayer
- `sublayer.createFeatureLayer()`
- Can use capabilities not normally available with Sublayer

Sublayer to FeatureLayer

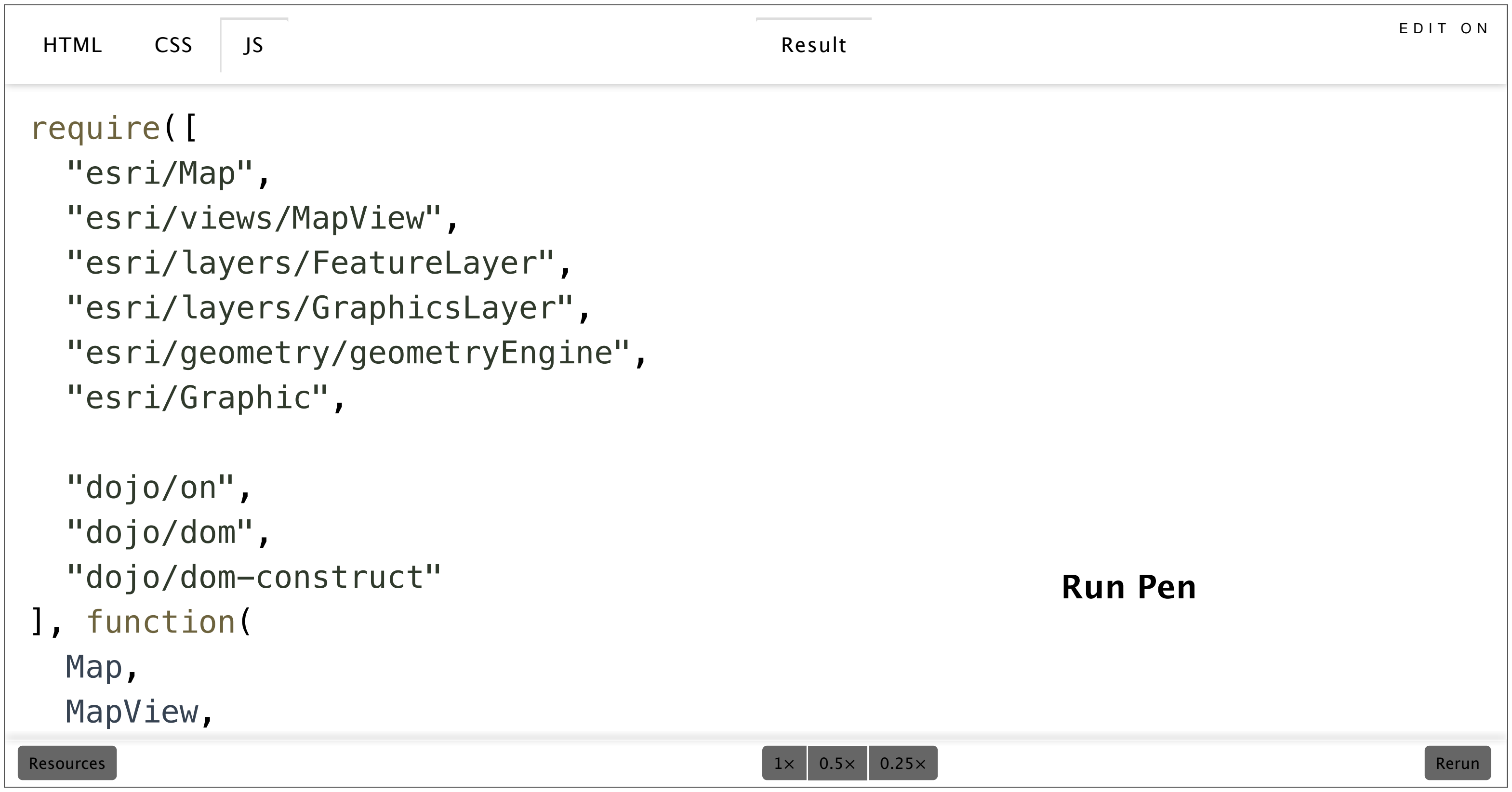


```
require([
  "esri/Map",
  "esri/views/MapView",
  "esri/layers/MapImageLayer",
  "esri/widgets/Legend",
  "esri/renderers/smartMapping
/creators/color"
], function(ArcGISMap, MapView,
MapImageLayer, Legend,
colorRendererCreator) {
  const layer = new MapImageLayer({
    url:
"https://sampleserver6.arcgisonline.com
/arcgis/rest/services/Census
```

createQuery

- When you can do `layer.createQuery()`
  - `query` object will already have the layers filters and layer definitions
  - more consistent
- Use `new Query()` when you don't want predefined filters to be applied

createQuery

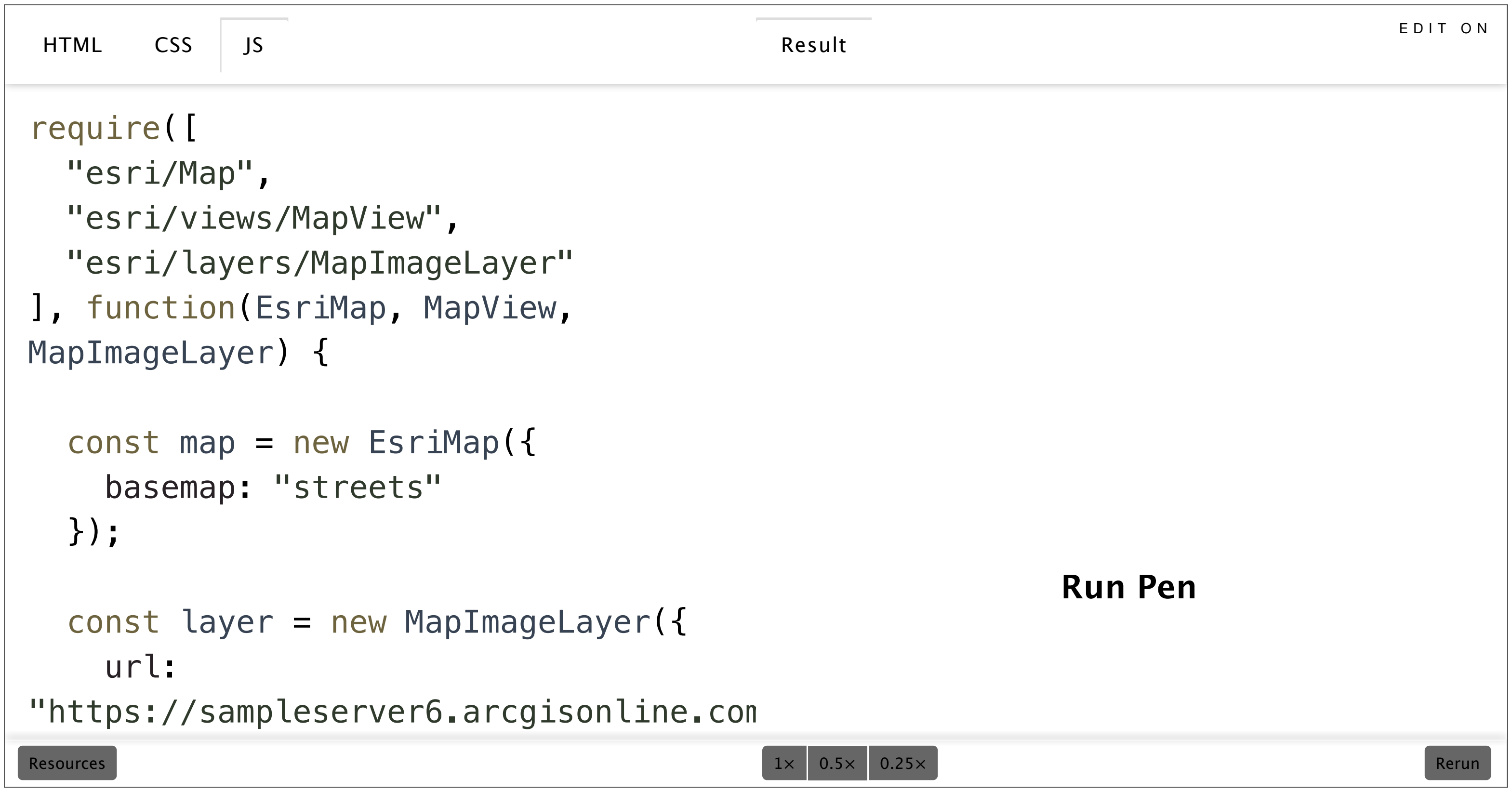


```
require([
  "esri/Map",
  "esri/views/MapView",
  "esri/layers/FeatureLayer",
  "esri/layers/GraphicsLayer",
  "esri/geometry/geometryEngine",
  "esri/Graphic",

  "dojo/on",
  "dojo/dom",
  "dojo/dom-construct"
], function(
  Map,
  MapView,
```

MapImageLayer

- If you want to modify Sublayers, do it after you load the layer
- Defining them upfront overrides the defaults
  - May not be what you want

MapImageLayer

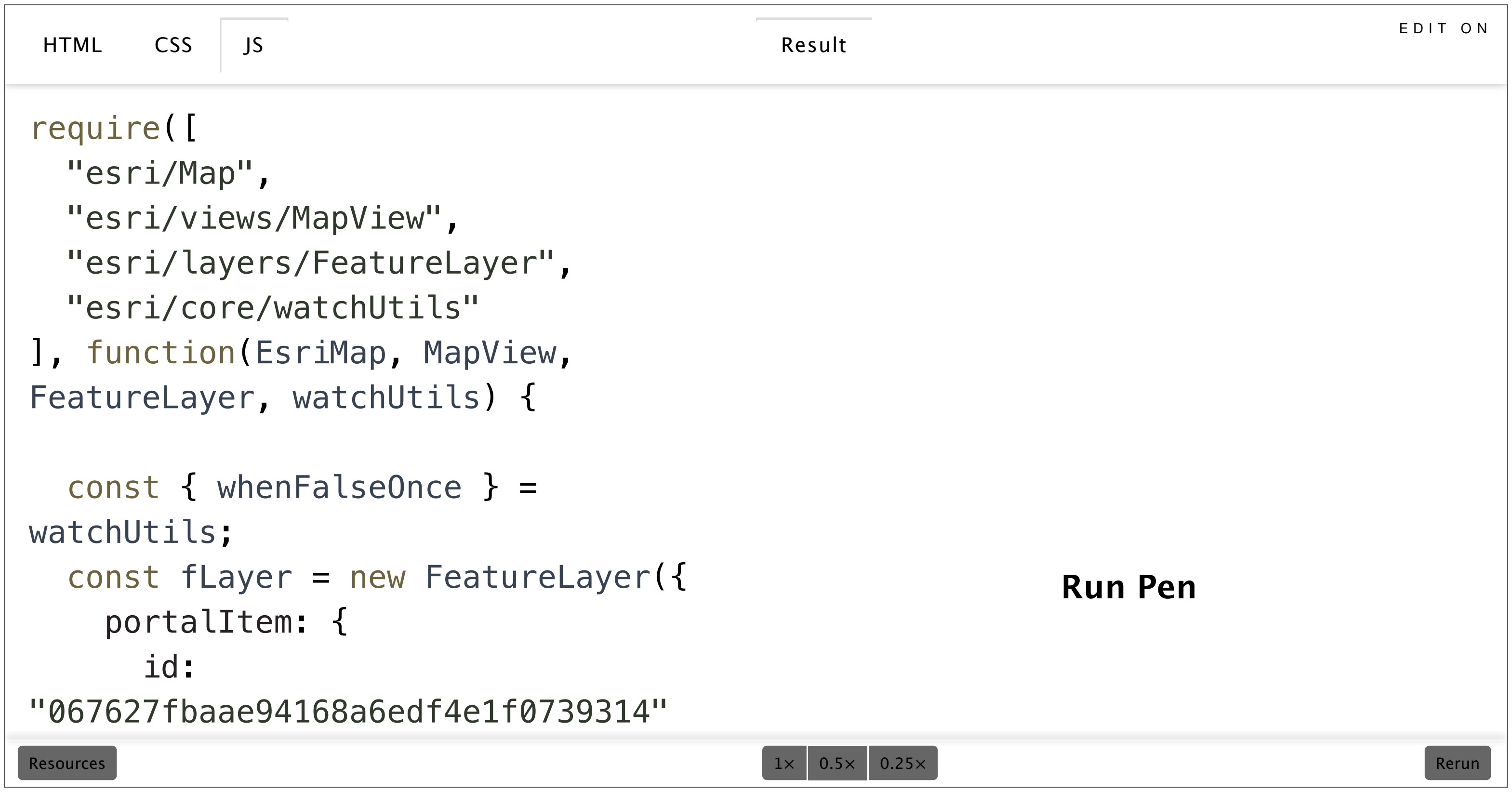


```
require([
  "esri/Map",
  "esri/views/MapView",
  "esri/layers/MapImageLayer"
], function(EsriMap, MapView,
MapImageLayer) {

  const map = new EsriMap({
    basemap: "streets"
  });

  const layer = new MapImageLayer({
    url:
"https://sampleserver6.arcgisonline.com
```

LayerViews

- Renders the Layer
- When is it done though?
    - When can you actually use it!!

LayerViews



```
require([
  "esri/Map",
  "esri/views/MapView",
  "esri/layers/FeatureLayer",
  "esri/core/watchUtils"
], function(EsriMap, MapView,
FeatureLayer, watchUtils) {

  const { whenFalseOnce } =
watchUtils;
  const fLayer = new FeatureLayer({
    portalItem: {
      id:
"067627fbaae94168a6edf4e1f0739314"
```

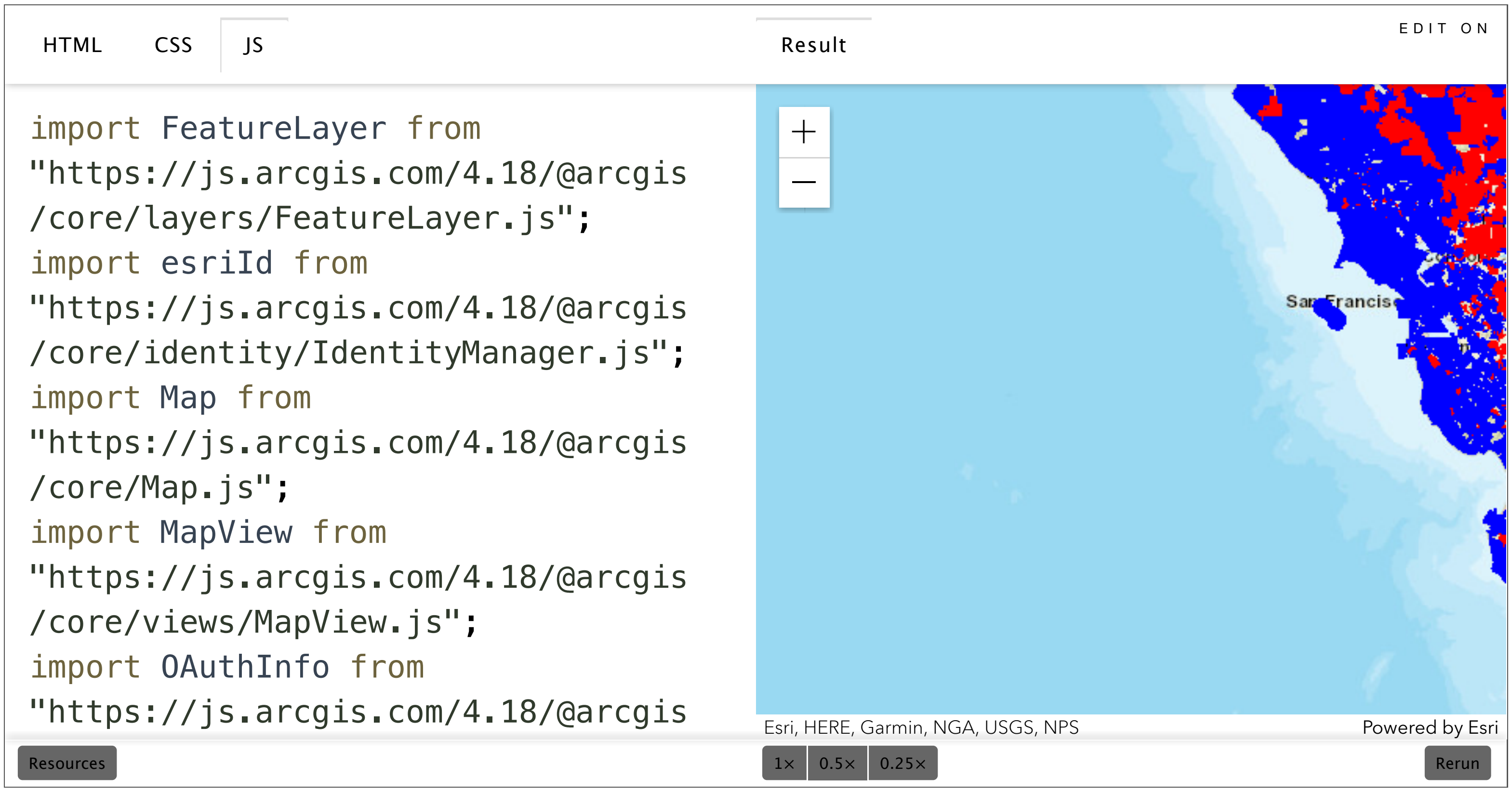
Renderers

- Specifies how a layer is visualized
- Determines what information will be conveyed through the layer's symbology
- *What story do you want to tell with this layer's data?*

Types of Renderers

Visualization Method	**Renderer Class**
Location only	`SimpleRenderer`, `HeatmapRenderer`
Unique values	`UniqueValueRenderer`
Class breaks	`ClassBreaksRenderer`
Continuous color or size	`SimpleRenderer` or `UniqueValueRenderer`, with visual variables
Multivariate	`SimpleRenderer` or `UniqueValueRenderer`, with visual variables

Renderers



```
import FeatureLayer from
"https://js.arcgis.com/4.18/@arcgis
/core/layers/FeatureLayer.js";
import esriId from
"https://js.arcgis.com/4.18/@arcgis
/core/identity/IdentityManager.js";
import Map from
"https://js.arcgis.com/4.18/@arcgis
/core/Map.js";
import MapView from
"https://js.arcgis.com/4.18/@arcgis
/core/views/MapView.js";
import OAuthInfo from
"https://js.arcgis.com/4.18/@arcgis
```

Widgets

Widgets

- Professionally designed and tested building blocks

Widgets

- Professionally designed and tested building blocks
- Responsive, accessible, and localized

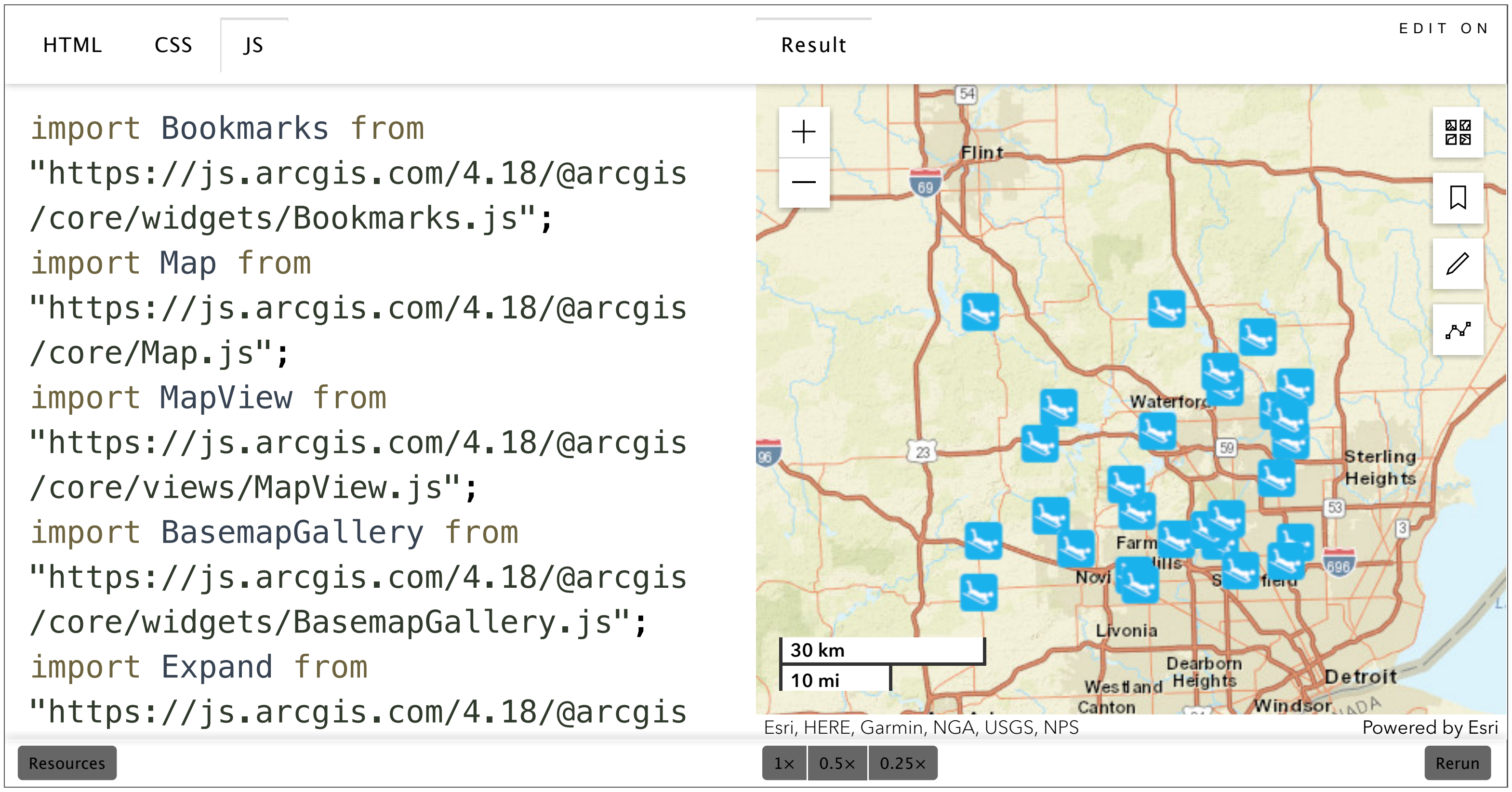
Widgets

- Professionally designed and tested building blocks
- Responsive, accessible, and localized
- 40+ widgets out of the box + everything you need to build your own

Widgets

- Professionally designed and tested building blocks
- Responsive, accessible, and localized
- 40+ widgets out of the box + everything you need to build your own
- Purpose-built for maps-based applications

Widgets



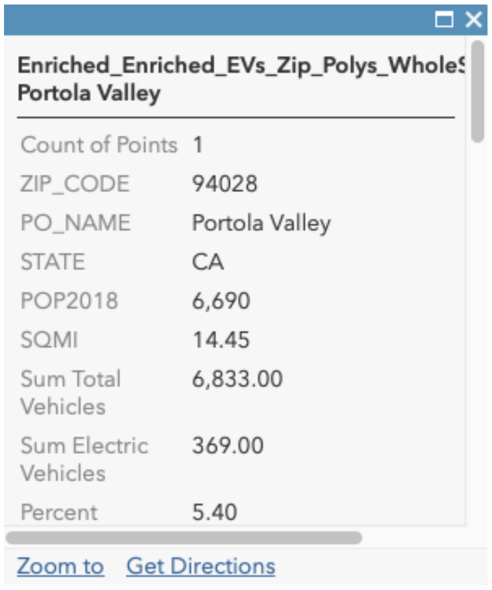
```
import Bookmarks from
"https://js.arcgis.com/4.18/@arcgis
/core/widgets/Bookmarks.js";
import Map from
"https://js.arcgis.com/4.18/@arcgis
/core/Map.js";
import MapView from
"https://js.arcgis.com/4.18/@arcgis
/core/views/MapView.js";
import BasemapGallery from
"https://js.arcgis.com/4.18/@arcgis
/core/widgets/BasemapGallery.js";
import Expand from
"https://js.arcgis.com/4.18/@arcgis
```

Widgets: Popup

- Show users information from feature attributes
- All views have a default popup template

```
const view = new MapView({
 map: myMap,
 defaultPopupTemplateEnabled: true //enable default popup
});
```

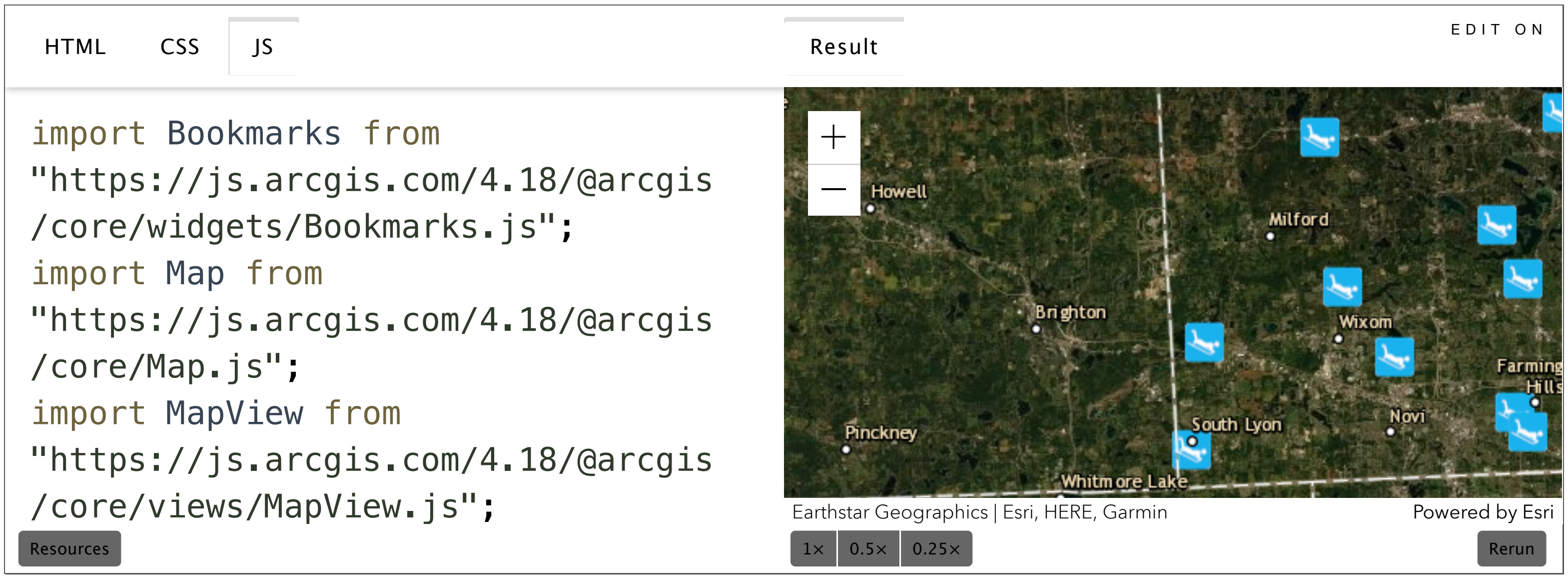
Often, database field names are not very user-friendly...



**Enriched_Enriched_EVs_Zip_Polys_WholeS
Portola Valley**

Count of Points	1
ZIP_CODE	94028
PO_NAME	Portola Valley
STATE	CA
POP2018	6,690
SQMI	14.45
Sum Total Vehicles	6,833.00
Sum Electric Vehicles	369.00
Percent	5.40

Widgets - Popup Template



```
import Bookmarks from
"https://js.arcgis.com/4.18/@arcgis
/core/widgets/Bookmarks.js";
import Map from
"https://js.arcgis.com/4.18/@arcgis
/core/Map.js";
import MapView from
"https://js.arcgis.com/4.18/@arcgis
/core/views/MapView.js";
```

Widget Architecture

Widgets have **two components**

Widget Architecture

Widgets have **two components**

1. **View**: responsible for the user interface and listening for user inputs

Widget Architecture

Widgets have **two components**

1. **View**: responsible for the user interface and listening for user inputs
2. **View Model**: manages the data structures and business logic for the widget

Custom Widgets

Custom Widgets

- Use Sass to style the built-in widgets

Custom Widgets

- Use Sass to style the built-in widgets
- Use the view model class of a built-in widget with a custom view class

Custom Widgets

- Use Sass to style the built-in widgets
- Use the view model class of a built-in widget with a custom view class
- Start from scratch, with comprehensive documentation and tutorials

Plenty to Get You Going

- Nearly 300 code samples

Plenty to Get You Going

- Nearly 300 code samples
- Open source starter apps: Maps App, Nearby Places

Plenty to Get You Going

- Nearly 300 code samples
- Open source starter apps: Maps App, Nearby Places
- Low-code and no-code app builders: Experience Builder, StoryMaps, Web AppBuilder, Configurable Apps

Please provide your feedback for this session by clicking on the session survey link directly below the video.