ArcGIS API for JavaScript 101

A Training course for GIS developer, partner and customer

29 June 2020

Prepared by GIS Co., Ltd.

ArcGIS API for JavaScript 101

A Training course for GIS developer, partner and customer

29 June 2020

Prepared by GIS Co., Ltd.


Course Outlines

To learn the **ArcGIS API for JavaScript 101** training course, you should be familiar with the following:

- JavaScript
- HTML
- CSS
- Basic concept Angular 10+

Cost: 1.5 man-days

1. GIS Concept (15 min)
2. Layers (20 min)
3. Features and Geometries (20 min)
4. Coordinate Systems (25 min)
5. Core Concepts (160 min)
6. Create a starter app (40 min)
7. Set basemap and add layers (60 min)
8. Style feature layers (40 min)
9. Configure pop-ups (30 min)
10. Query a feature layer (40 min)
11. Get map coordinates (25 min)
12. Display point, line, and polygon graphics (20 min)
13. Draw graphics (20 min)
14. Add, edit, and remove features (120 min)
15. Get a route and directions (90 min)

Please bring your own Laptop with minimum spec.:

- CPU 2.0 GHz
- 64-bit architecture
- 4 or more CPUs/cores
- At least 16 GB of memory/RAM
- At least 25 GB of disk space
- Windows 10, macOS 10.13 or Ubuntu 20.04

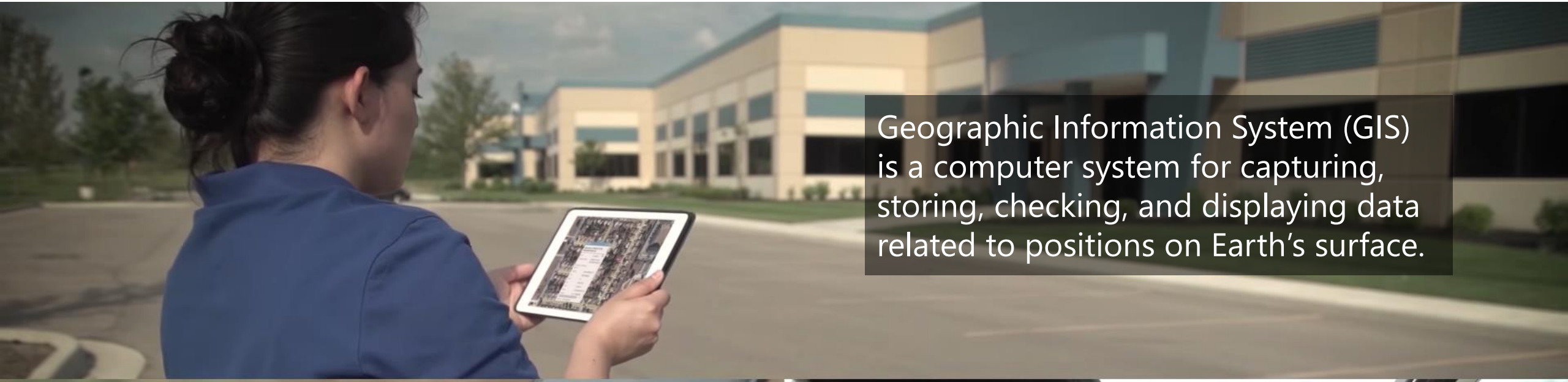
Install programs to your laptop with the following:

- Visual Studio Code
- Node.js (Requires a current, active LTS, or maintenance LTS)
- Angular CLI

GIS Concept

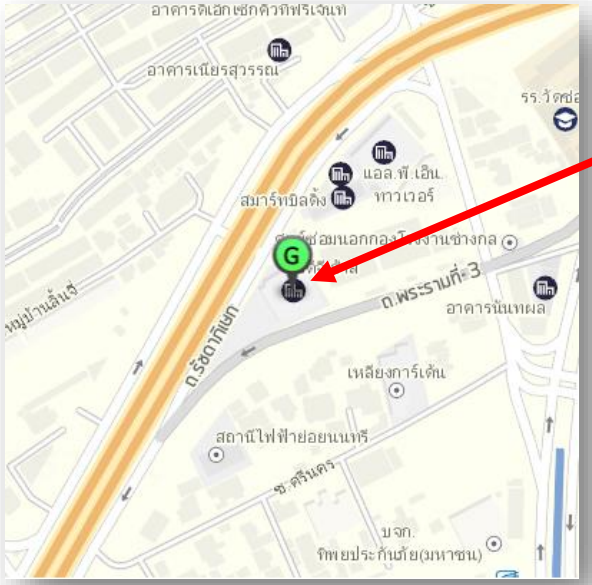
15 minutes

GIS concept



Geographic Information System (GIS) is a computer system for capturing, storing, checking, and displaying data related to positions on Earth's surface.

- Something about Maps
- About Computer and Database system
- About Internet, Could, and Online
- And also about Mobile



Spatial Location

Attribute

บริษัท จีไอเอส จำกัด

เลขที่ 202 อาคาร ซีดีจี เฮ้าส์
ถนนนางลิ้นจี่ แขวงช่องนนทรี
เขตยานนาวา กรุงเทพมหานคร 10120

gis.contact@cdg.co.th

โทรศัพท์: 0 2678 0707

โทรสาร: 0 2678 0321-3

Hotline: 0 2678 0033

http://www.giscompany.co.th

Buildings data

202 CDG House
Nanglinchi Rd.,
Chongnonsee,
Yannawa,
Bangkok

→ Attribute

13° 42′ 10.77′′ N
100° 32′ 28.33″ E

→ Geometry

Feature

Buildings Table (MIS)

Address	Sub-district	District	Province
202 CDG House Nanglinchi Rd.	Chongnonsee	Yannawa	Bangkok

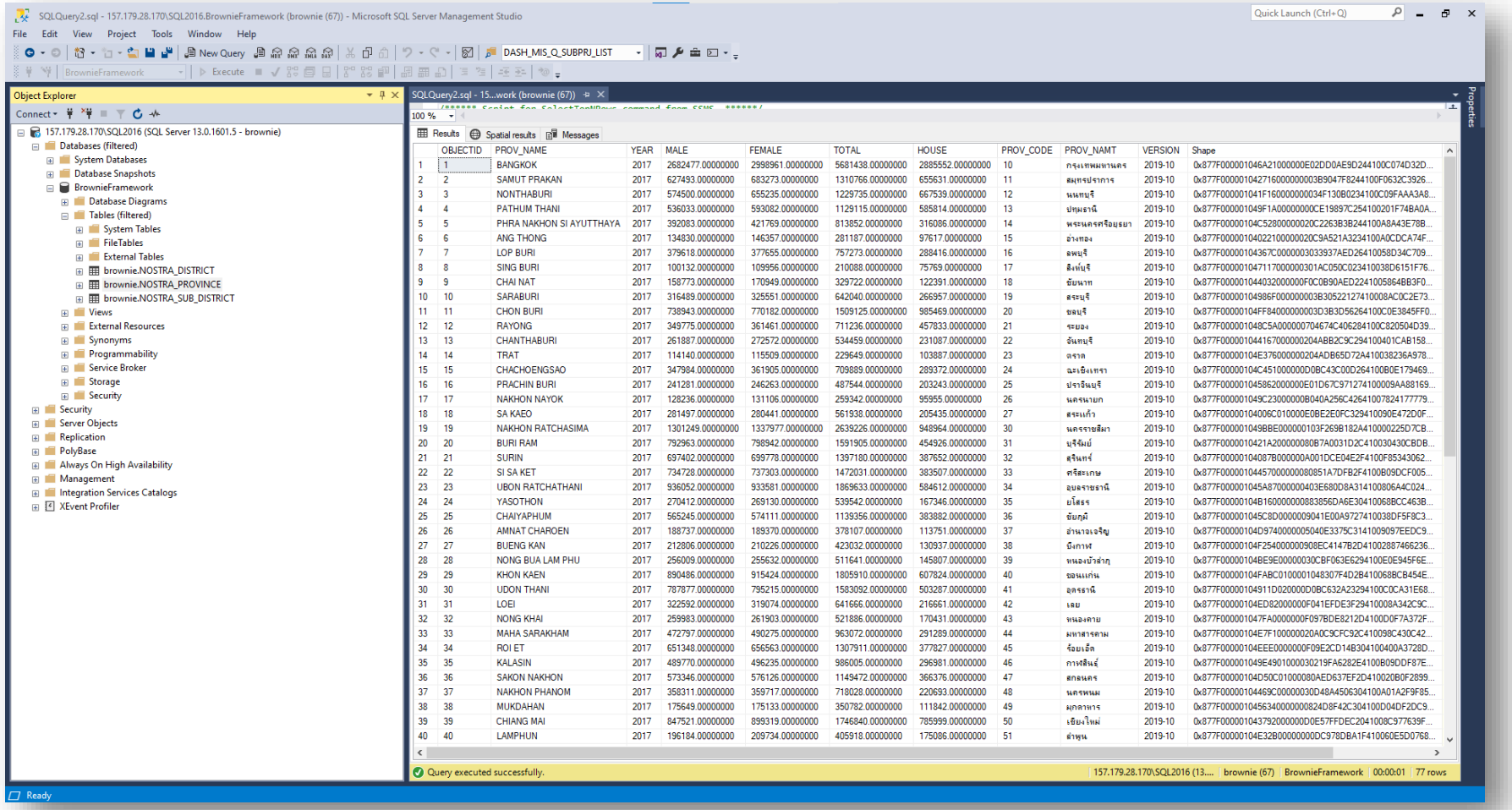
Buildings FeatureClass (GIS)

Address	Sub-district	District	Province	Shape
202 CDG House Nanglinchi Rd.	Chongnonsee	Yannawa	Bangkok	0xE6100000010C E46BEE39C7225 940A0B1FCC10C 682B40

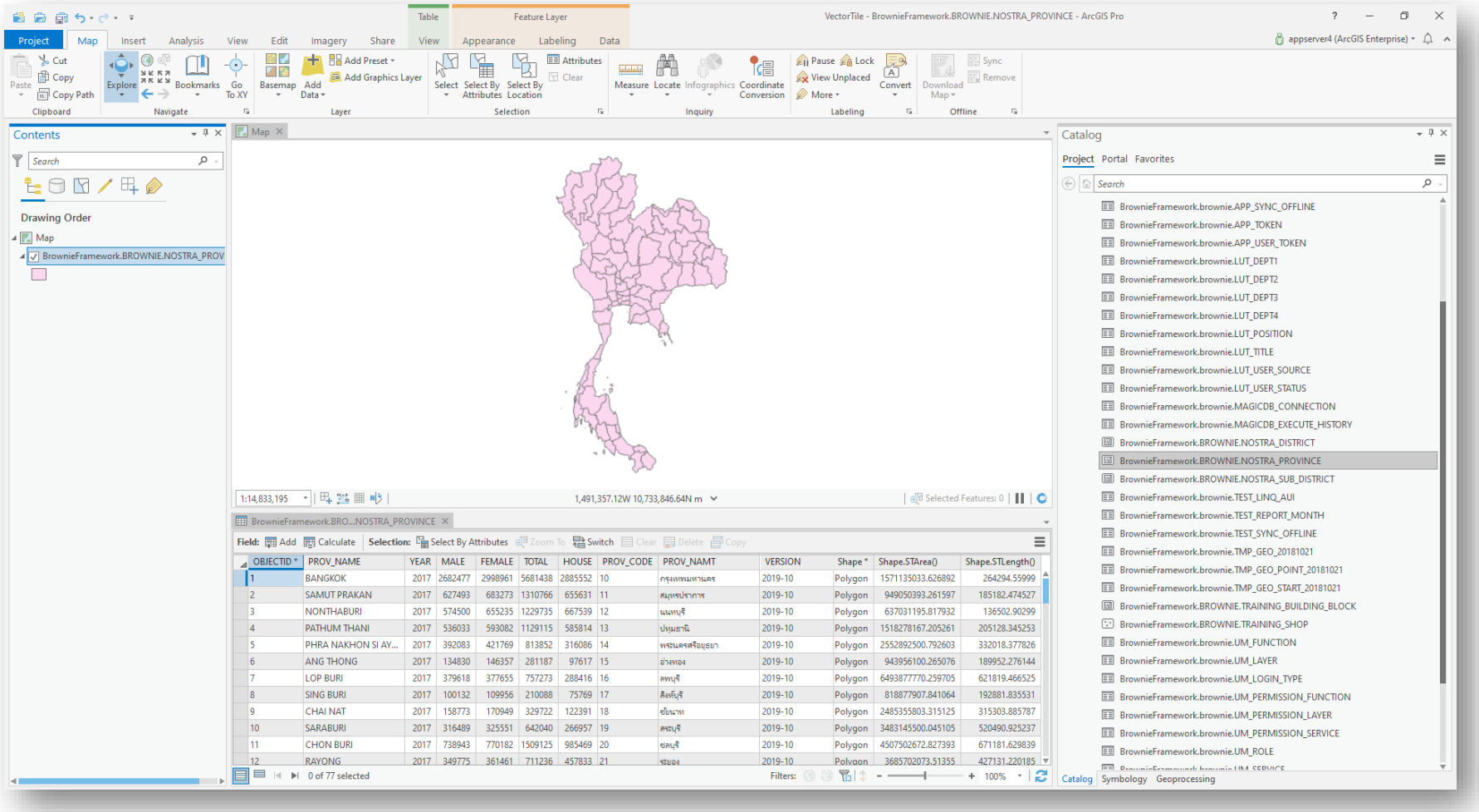
Attributes

Geometry
(Large number)

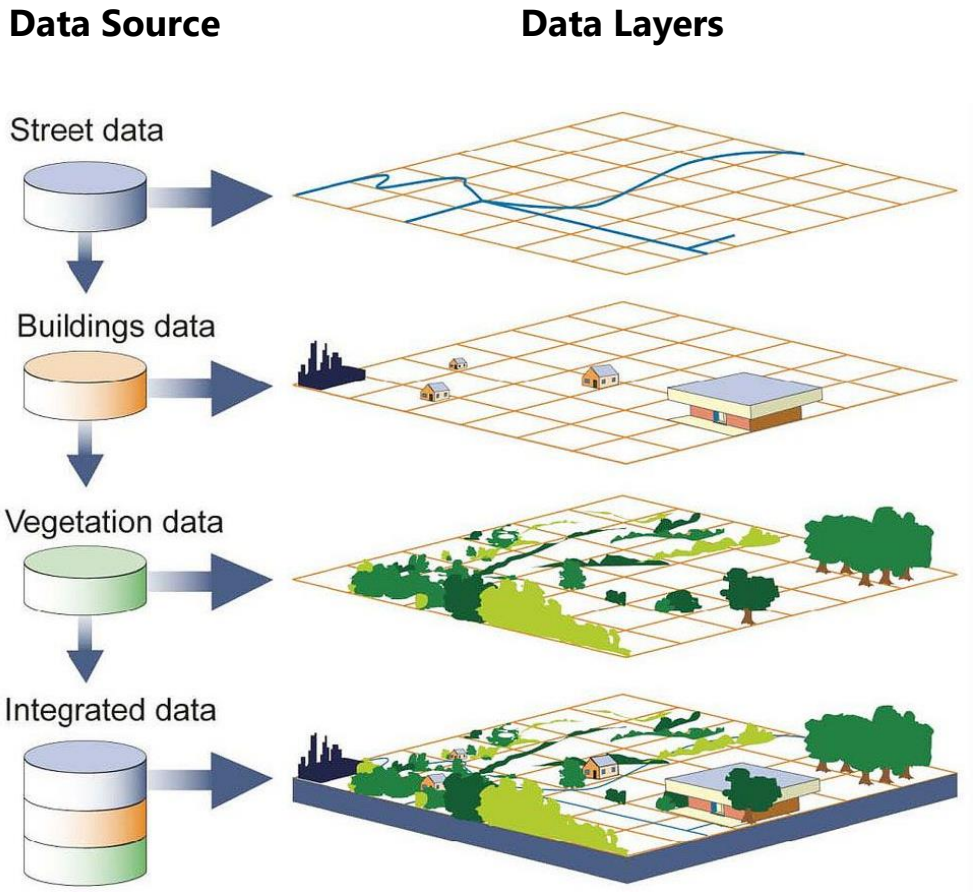
Show provincial boundaries in Thailand with Microsoft SQL Server Management Studio.



In the same data source, show provincial boundaries in Thailand with ArcGIS Pro.



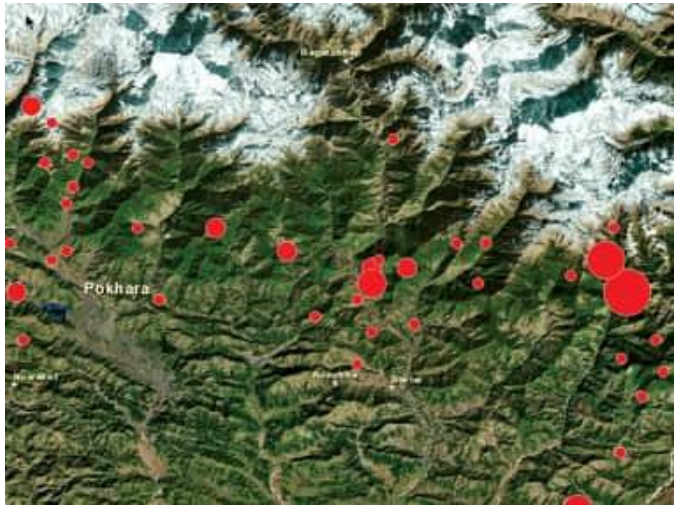
GIS concept – Stores data as layers



GIS can show many different kinds of data on one map, such as streets, buildings, and vegetation.

This enables people to more easily see, analyze, and understand patterns and relationships.

GIS concept – Stores data as layers



Feature point data from in-ground data sensors.

Cell-based raster using historical predictive data.

Feature polygon data from cadastral surveys.

1. From the following picture, categorize the types of data as appropriate.

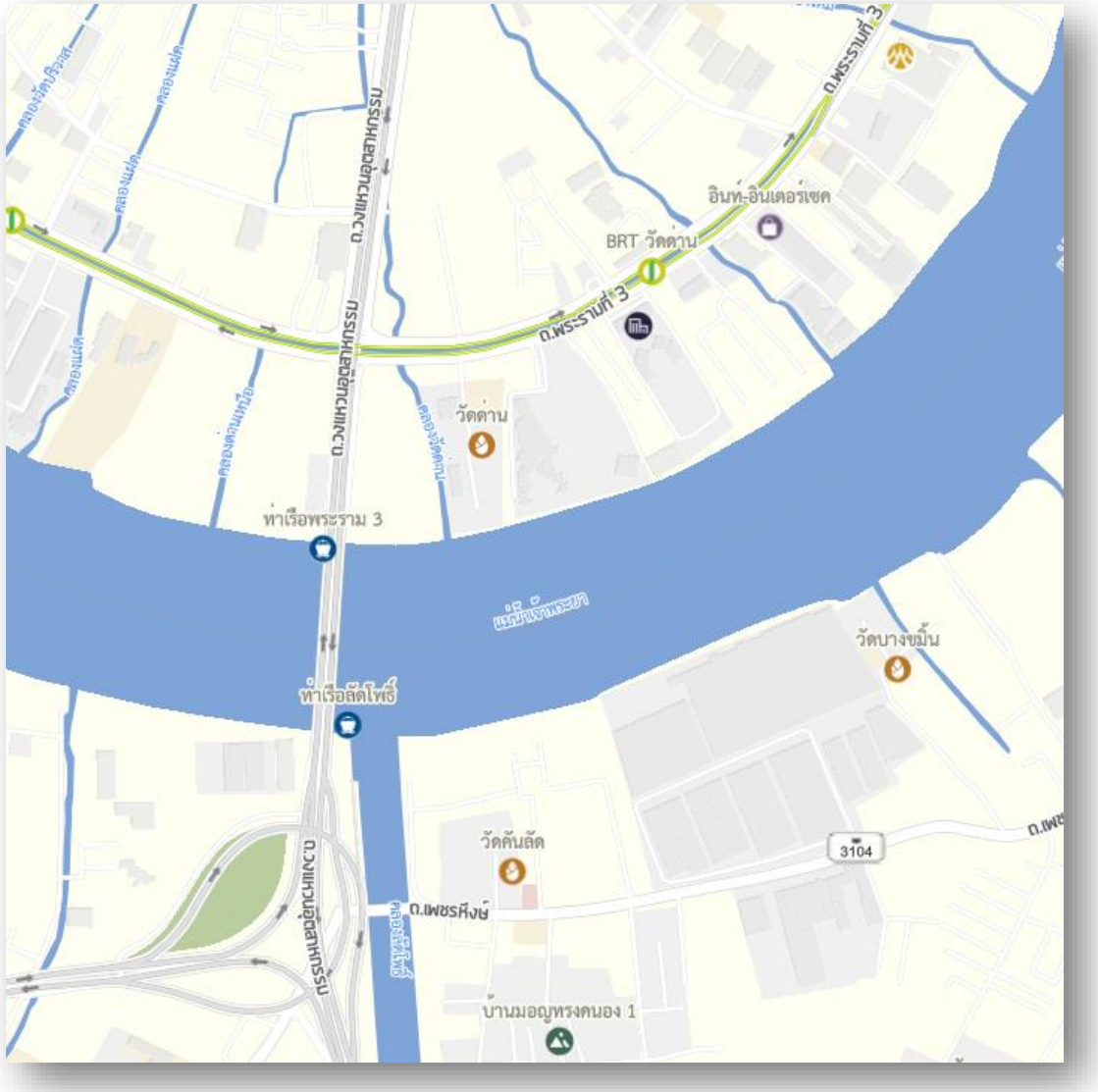
Sample answer

Parks: *สวนสุขภาพลัดโพธิ์*

_____:

_____:

_____:



Layers

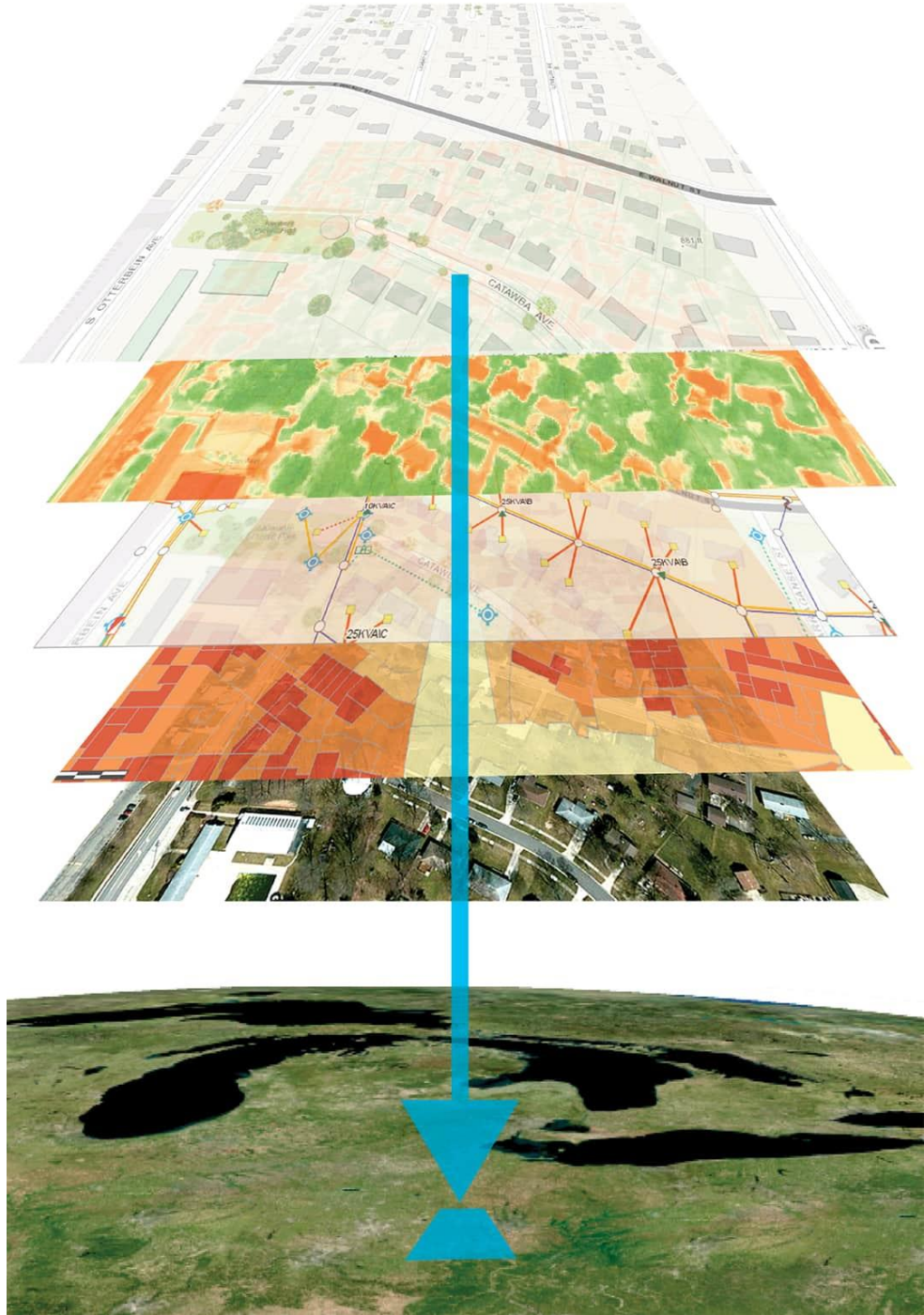
20 minutes

Layers line up on Earth

Georeferenced layers of information are the key characteristic of GIS that enabled disparate types of data to be displayed, combined, and analyzed in common geographic space.

Things that map layers can represent

- Buildings
- Roads
- Parks
- Trees
- Vegetation Health
- Utility Networks
- Demographic Data
- Satellite Imagery



Layers are logical collections of geographic data that can be used to create maps and applications, and can be broadly categorized as either feature or tile layers:

- Feature layers can store geographic features, edit or update attributes, and synchronize with offline databases.

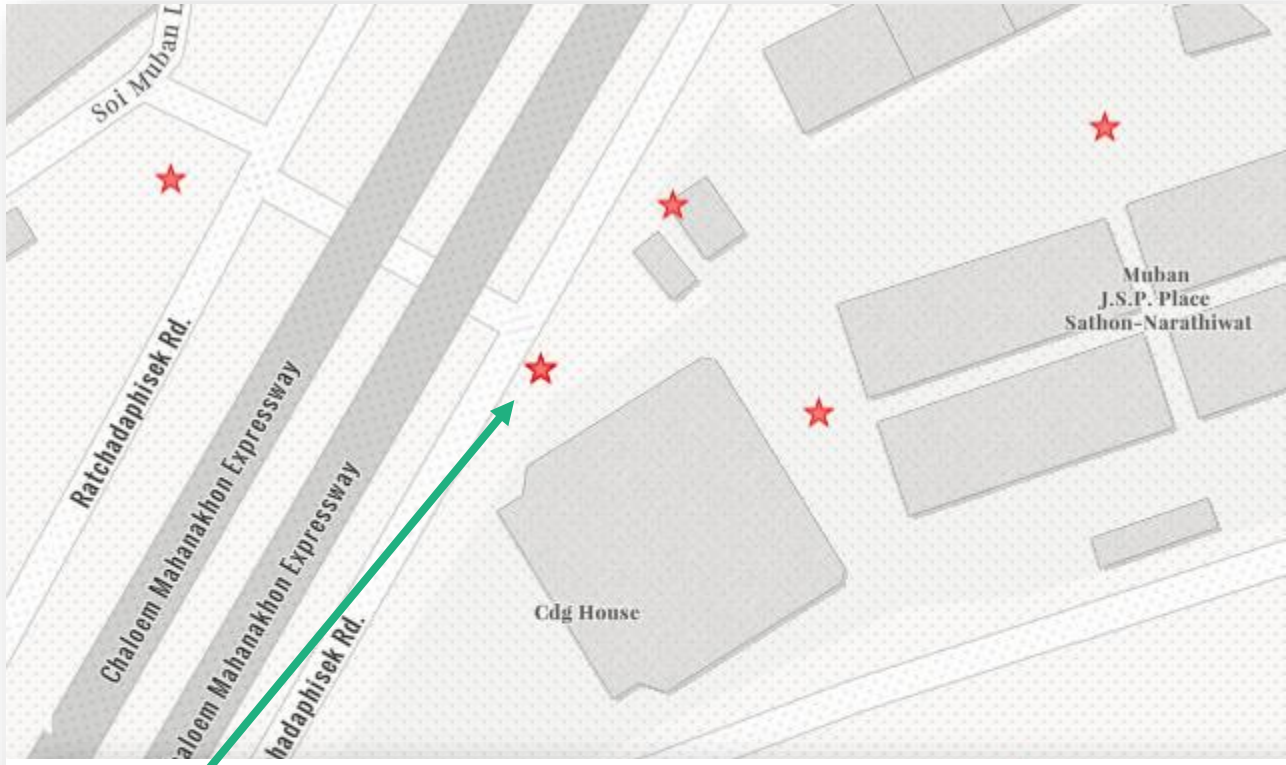
- Tile layers are pre-generated and the tiles are cached on a server. Vector tile basemaps, for example, are tile layers. Scene Layers can be thought of as tile layers that cache features, including 3D objects like buildings, for rapid display in 3D applications.

Feature Layer

REST API Service
Feature Service

Cached
No

Use Cases
Querying, rendering, and editing vector geographic information.



{objectIdFieldName: "OBJECTID", globalIdFieldName: "", geometryType: "esriGeometryPoint"
▼features: [{attributes: {OBJECTID: 2823, SHOP_NAME: "AUI 1"},…},…]
▶0: {attributes: {OBJECTID: 2823, SHOP_NAME: "AUI 1"},…}
▶1: {attributes: {OBJECTID: 2825, SHOP_NAME: "bell"},…}
▼2: {attributes: {OBJECTID: 5227, SHOP_NAME: "test shop 101"},…}
▼attributes: {OBJECTID: 5227, SHOP_NAME: "test shop 101"}
OBJECTID: 5227
SHOP_NAME: "test shop 101"
▶geometry: {x: 11192414.244465057, y: 1540261.3890377367}
▶fields: [{name: "OBJECTID", alias: "OBJECTID", type: "esriFieldTypeOID"},…]
geometryType: "esriGeometryPoint"

Tile Layer

REST API Service
Map Service

Cached
Yes

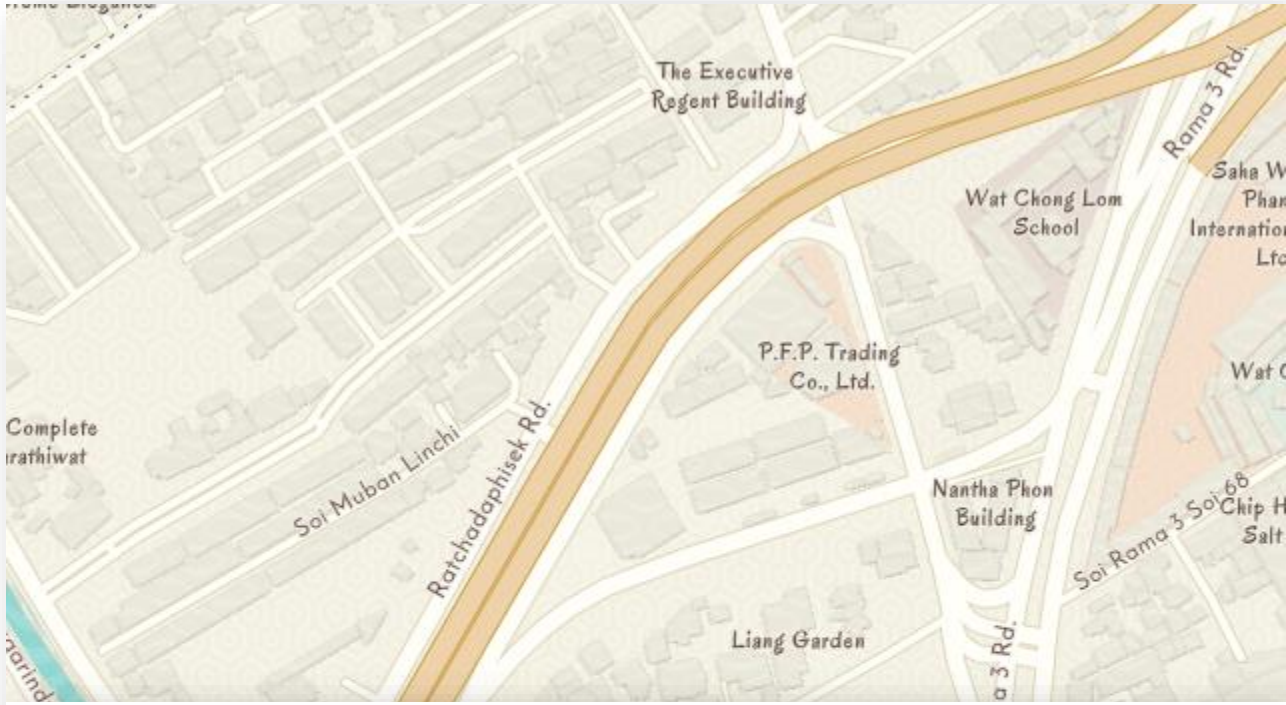
Use Cases
Basemaps and other complex datasets that change infrequently. Tiles can be kept in sync with feature layers for operational data.

Vector Tile Layer

REST API Service
Vector Tile Service

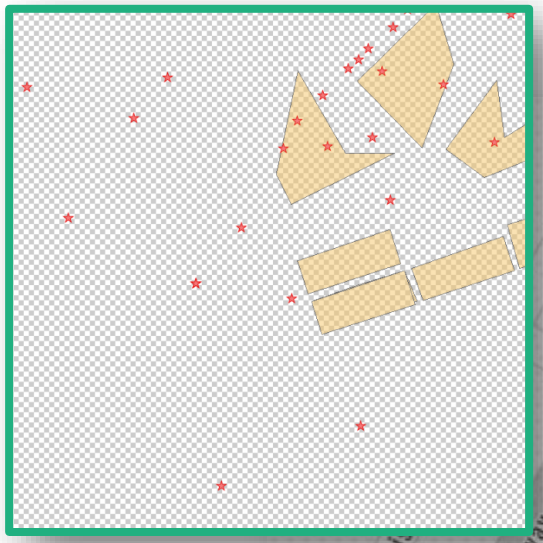
Cached
Yes

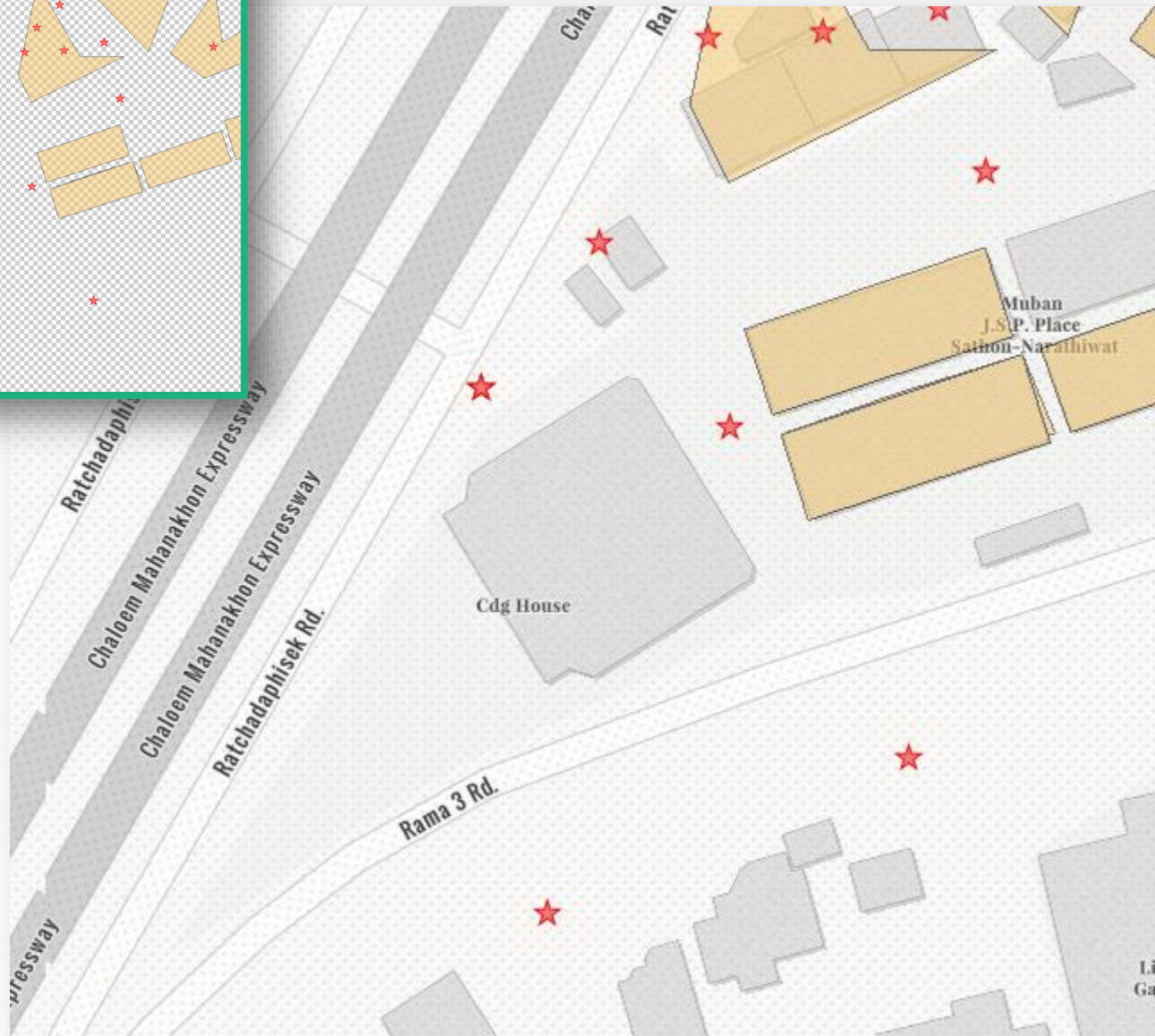
Use Cases
Basemaps and other complex datasets that change infrequently. Tiles can be kept in sync with feature layers for operational data.



PBF file
(application/octet-stream)

Dynamic Map Layer





REST API Service
Map Service

Cached
No

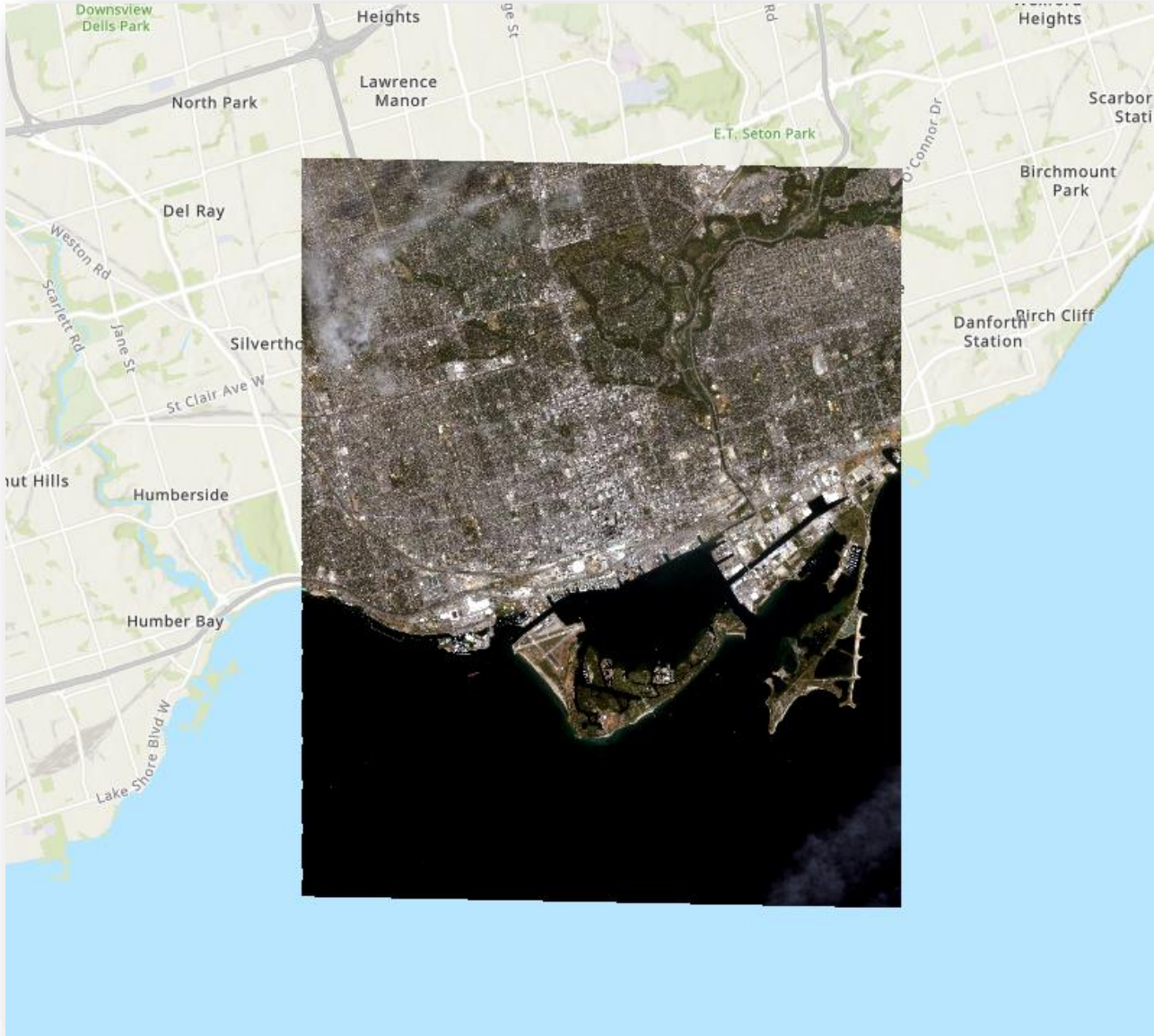
Use Cases
Complex data sets that change frequently or need complex rendering requirements.

Image Layer

REST API Service
Image Service

Cached
No

Use Cases
Rendering and analyzing satellite or other imagery data.

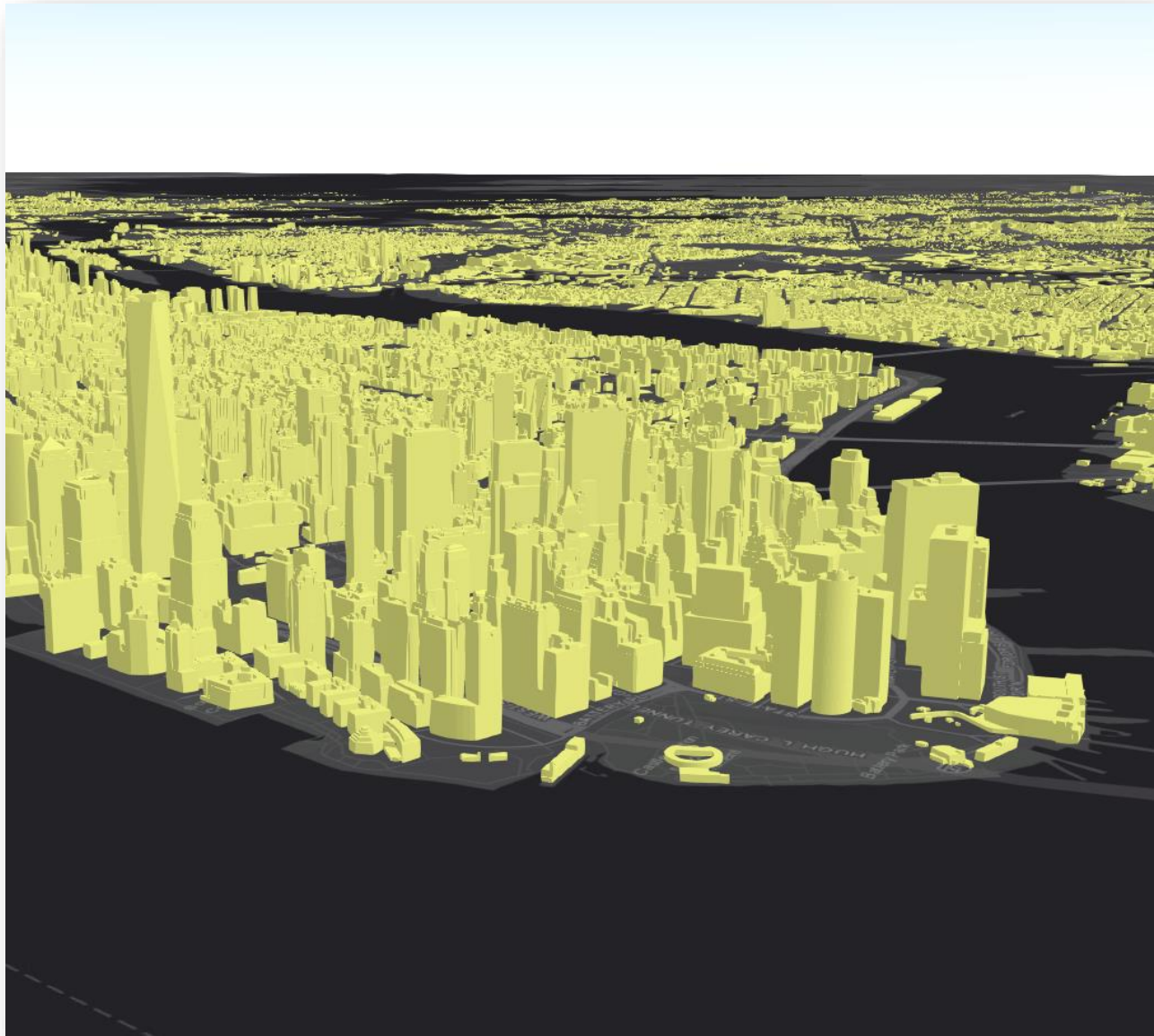


Scene Layer

REST API Service
Scene Service

Cached
Yes

Use Cases
Displaying and rendering 3D datasets.



1. You would like to provide the Layer that includes aerial photographs that you fly, survey and photograph twice a year. What kind of layers do you need to share?

2. You have Point of Interest (POI) data source and would like to share it so that partner can edit the POI. What kind of layers do you need to share?

Features and Geometries

20 minutes

ArcGIS APIs and SDKs make use of a dedicated JSON specification to define geographic data for use on the map.

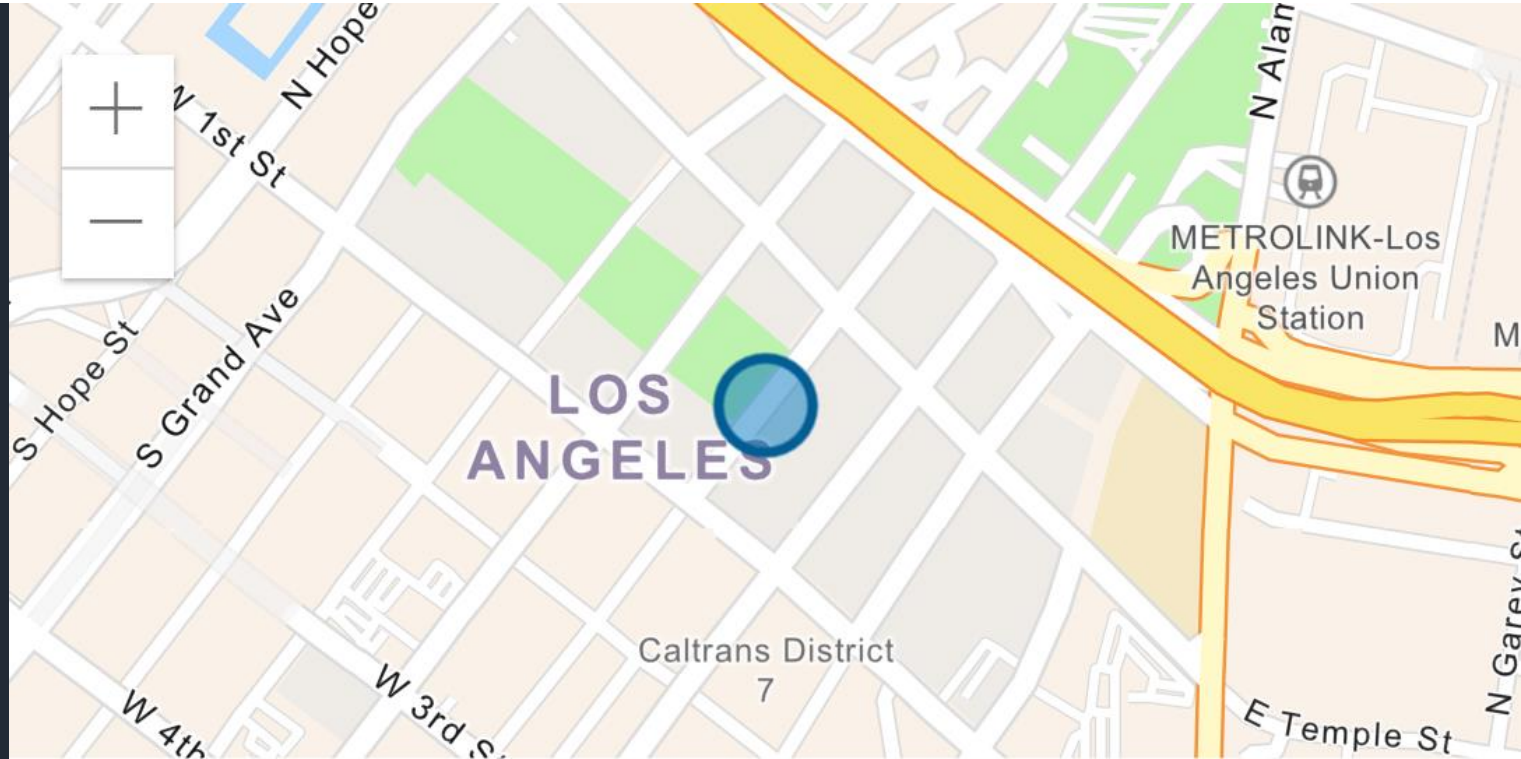
These JSON objects include:

Points	Define a single location on the map, such as a user's location.
Polylines	Define a series of points representing a line, such as a road or a river.
Polygons	Define closed shapes, such as the outline of a lake.
Extents	Define closed rectangular shapes, such as the outline of a building.
Features	Consist of a geometry (point, line or polygon) and additional attributes (like a name) and a symbol that represents how the feature is rendered on the map.

Points are used to represent a single specific location, such as an address, users location, or asset. Points are required to have at least three properties:

- *x* – The location of the point along the x axis.
- *y* – The location of the point along the y axis.
- *spatialReference* – Defines the measurement system used to locate the polygon on a model of the Earth.

```
{
    "x": -118.24354,
    "y": 34.05389,
    "spatialReference": {
        "wkid": 4326
    }
}
```



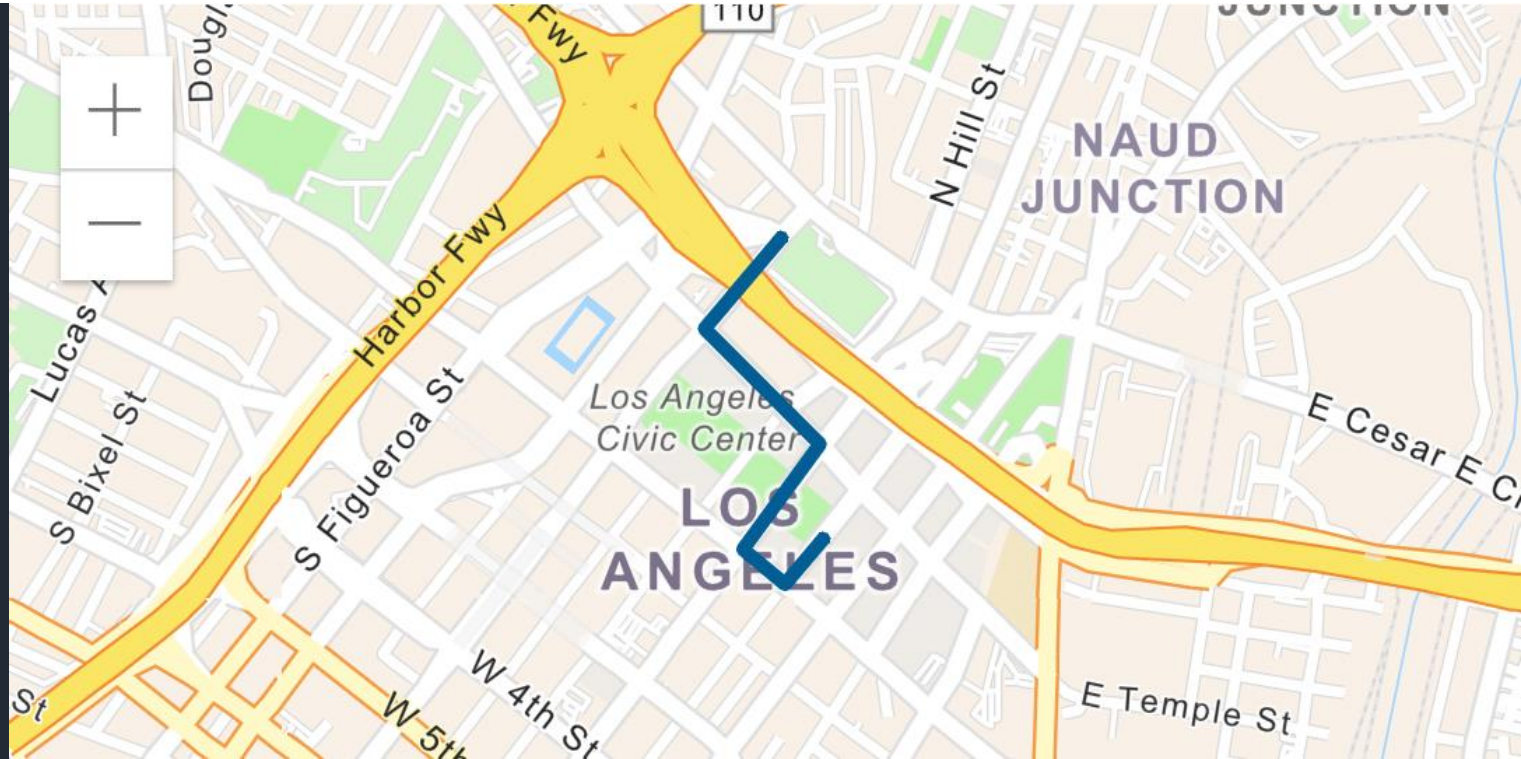
Points may also optionally contain *z* and *m* properties. z refers to elevation while m represents an arbitrary linear measurement that is independent of point's location.

Note: A *wkid* (or Well Known ID) identifies the spatial reference used to articulate the point's location. *4326* is the standard system commonly used by GPS. The `x` axis represents longitude and the y axis represents latitude. **This may be the reverse of what you are accustomed to.**

Polylines are used to represent unclosed lines between two or more points. A line might represent a wall or barrier, a trail or road or a route between two locations. Polylines can include individual segments that do not touch and have two key top level properties:

- *paths* – An array of paths, each path is an array of coordinates pairs in *[x, y]* format.
- *spatialReference* – Defines the measurement system used to locate the polygon on a model of the Earth.

```
{
    "paths": [
        [
            [ -118.24354, 34.05389 ],
            [ -118.24446, 34.05294 ],
            [ -118.24554, 34.05364 ],
            ...
            [ -118.24454, 34.06001 ]
        ]
    ],
    "spatialReference": {
        "wkid": 4326
    }
}
```



Polygons are used to represent closed and filled shapes such as state or country boundaries, parks or building footprints. They can include both holes and non-overlapping geometries. Polygons have two key top level properties:

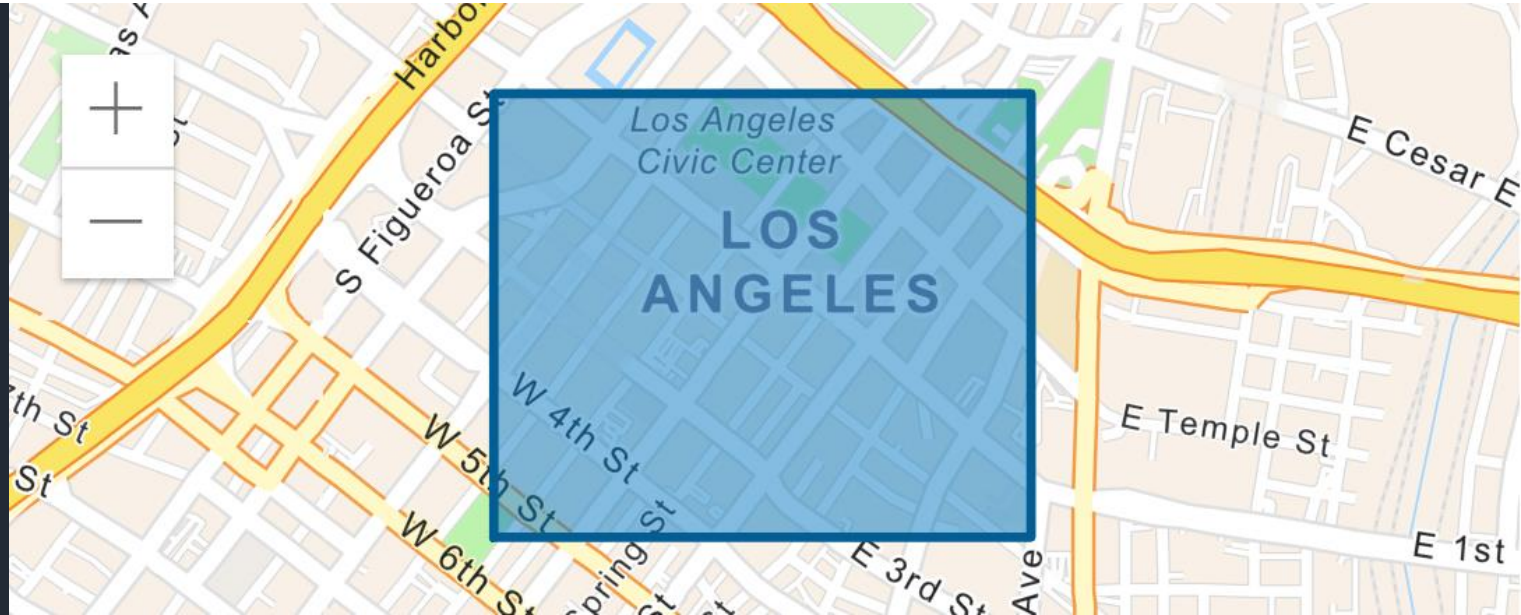
- *rings* – An array of rings, clockwise rings are filled and counterclockwise rings are considered holes. Each ring is an array of *[x, y]* coordinate pairs.
- *spatialReference* – Defines the measurement system used to locate the polygon on a model of the Earth.

```
{
    "rings": [
        [
            [ -118.38516, 34.01270 ], ...
        ],
        [
            [ -118.38661, 34.01486 ], ...
        ]
    ],
    "spatialReference": {
        "wkid": 4326
    }
}
```

Extents represent rectangular shapes that are commonly used to center the map on a particular area of interest. Extents are also commonly referred to as envelopes or bounding boxes and have five key top level properties:

- *xmin* – The lowest value of the extent along the x-axis.
- *ymin* – The lowest value of the extent along the y-axis.
- *xmax* – The highest value of the extent along the x-axis.
- *ymax* – The highest value of the extent along the y-axis.
- *spatialReference* – Defines the measurement system used to locate the extent on a model of the Earth.

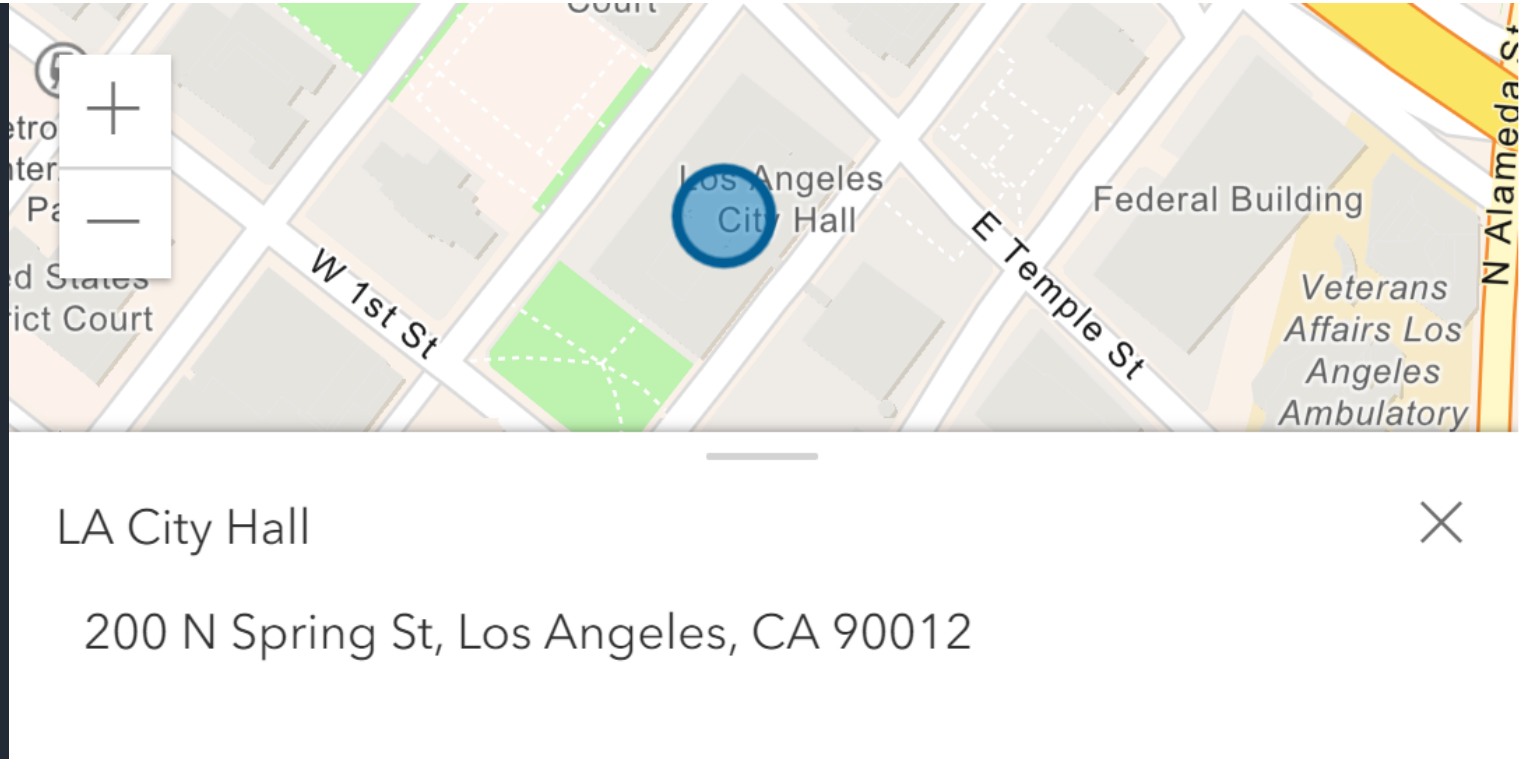
```
{
    "xmin": -118.252655,
    "ymin": 34.048244,
    "xmax": -118.239434,
    "ymax": 34.057265,
    "spatialReference": {
        "wkid": 4326
    }
}
```



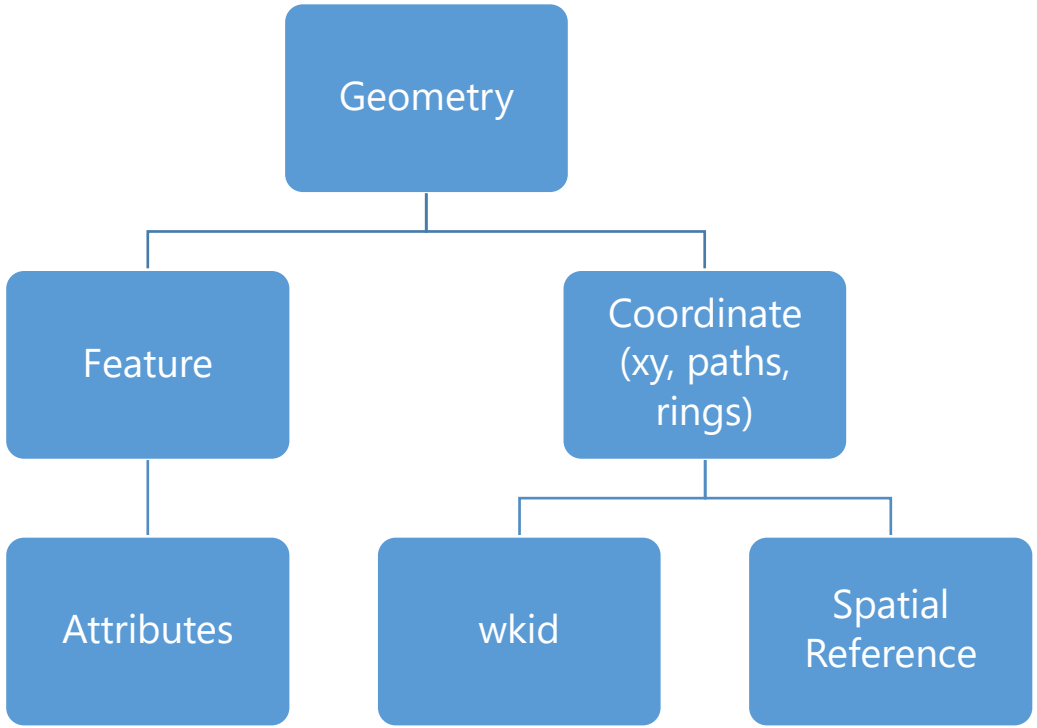
Features are used to associate tabular data with geographic information. This additional information can used to affect the rendering of the feature, provide additional metadata to services, be displayed in popups, or used to filter features on the map. Features have two important top level properties:

- *geometry* – A point, polyline, polygon or extent object.
- *attributes* – An object of key/value pairs in JSON format to associate with the geometry.


```
{
    "geometry": {
        "x": -118.24274,
        "y": 34.05369,
        "spatialReference": {
            "wkid": 4326
        }
    },
    "attributes": {
        "name": "LA City Hall",
        "address": "200 N Spring St, Los An
geles, CA 90012"
    }
}
```

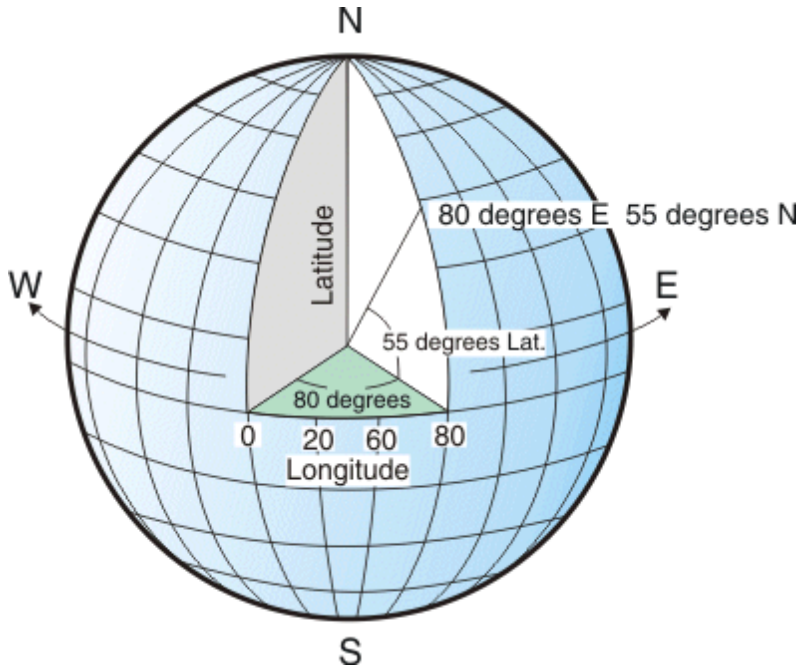
1. Rearrange the relationship correctly.



Coordinate Systems

25 minutes

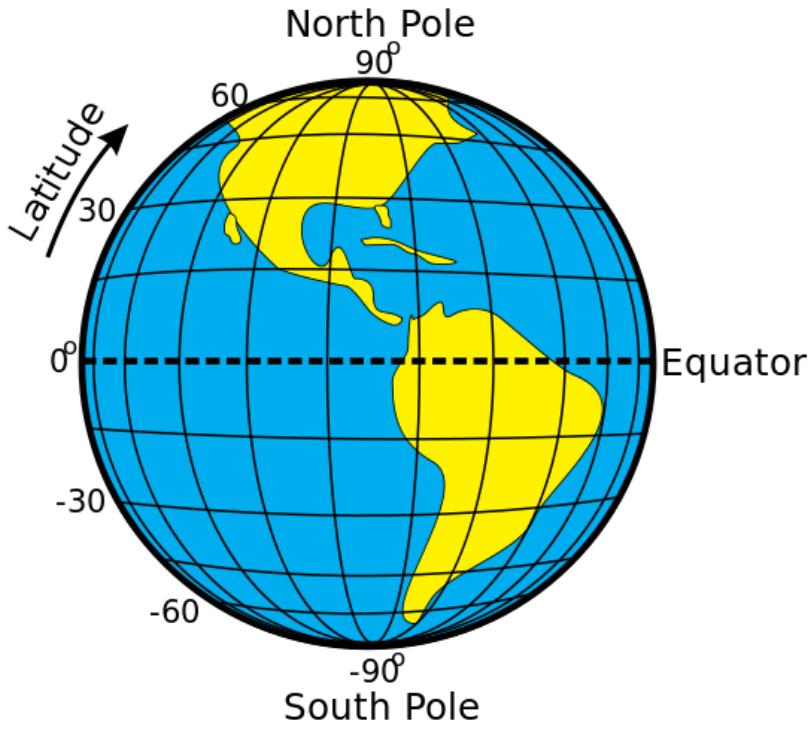
In a geographic coordinate system (GCS), we can reference any point on Earth by its longitude and latitude coordinates. Because a GCS uses a sphere to define locations on the Earth, we use angles measured in degrees from the earth's center to any point on the surface.

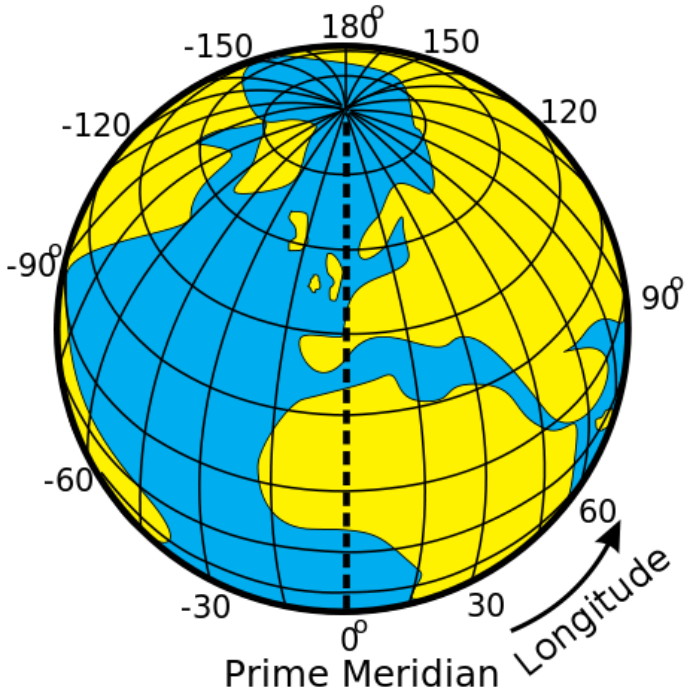


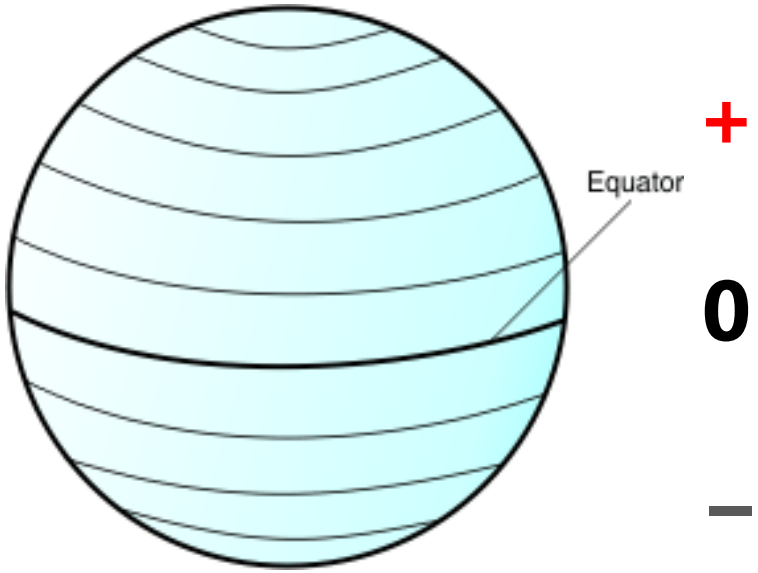
Latitude lines run *east-west* and are parallel to each other. If you go north, latitude values increase. Finally, *latitude values (Y-values) range between -90 and +90 degrees.*

Longitude lines run *north-south*. They converge at the poles. And its *X-coordinates are between -180 and +180 degrees.*

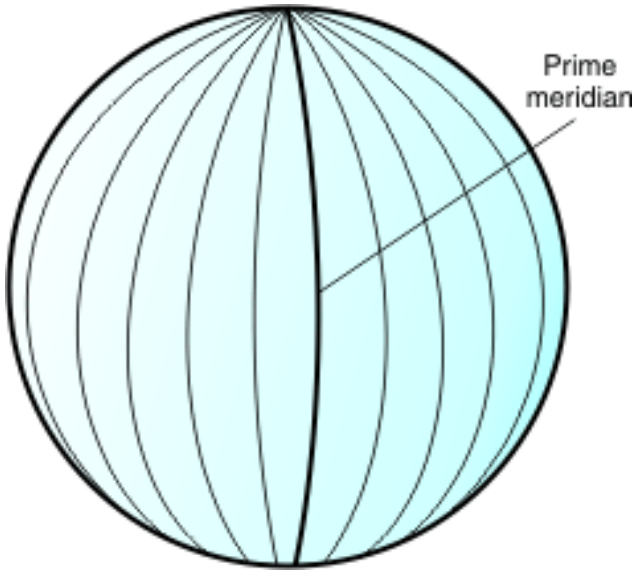
Coordinate systems – Geographic Coordinate Systems







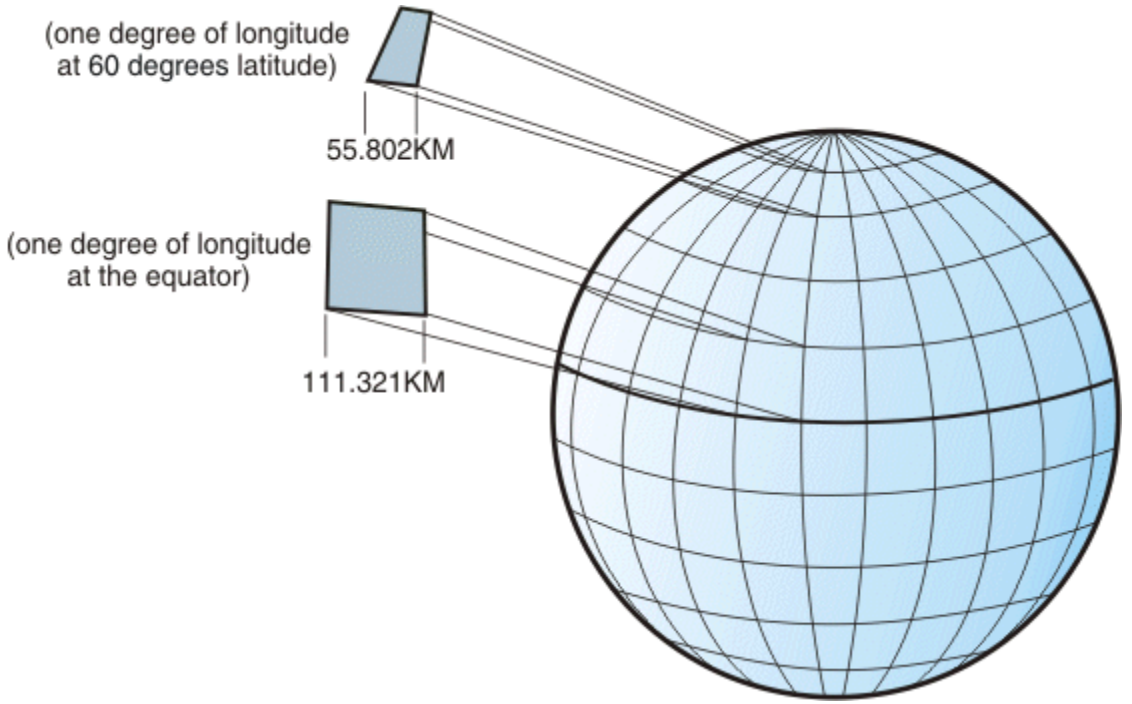
The equator is 0° latitude where we measure north and south. This means that everything north of the equator has positive latitude values. Whereas, everything south of the equator has negative latitude values.



The **Greenwich Meridian (or prime meridian)** is a zero line of longitude from which we measure east and west. In fact, the zero line passes through the **Royal Observatory in Greenwich, England**, which is why we call it what it is today. In a geographical coordinate system, the prime meridian is the line that has 0° longitude.

At the equator, one degree of longitude is approximately 111.321 kilometers, while at 60 degrees of latitude, one degree of longitude is only 55.802 km.

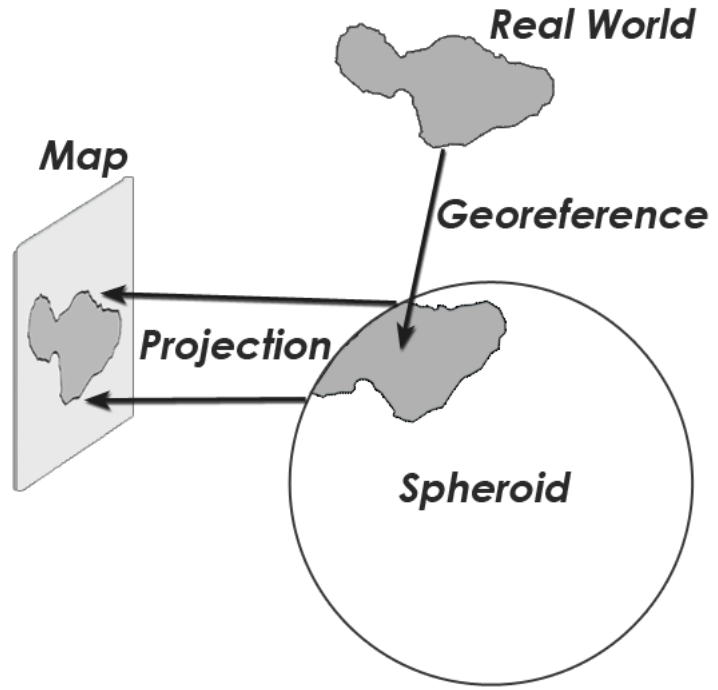
Therefore, because there is no uniform length of degrees of latitude and longitude, the distance between points cannot be measured accurately by using angular units of measure.



"Peel an Orange and Flatten the Peels"

When you look at it any direction, you won't be able to see all sides of it. But when you peel the orange, flatten and stretch it out, you can begin to see everything.

Similarly, a map projection is a method by which cartographers translates a sphere or globe into a two-dimensional representation. In other words, *a map projection systematically renders a 3D ellipsoid (or spheroid) of Earth to a 2D map surface.*

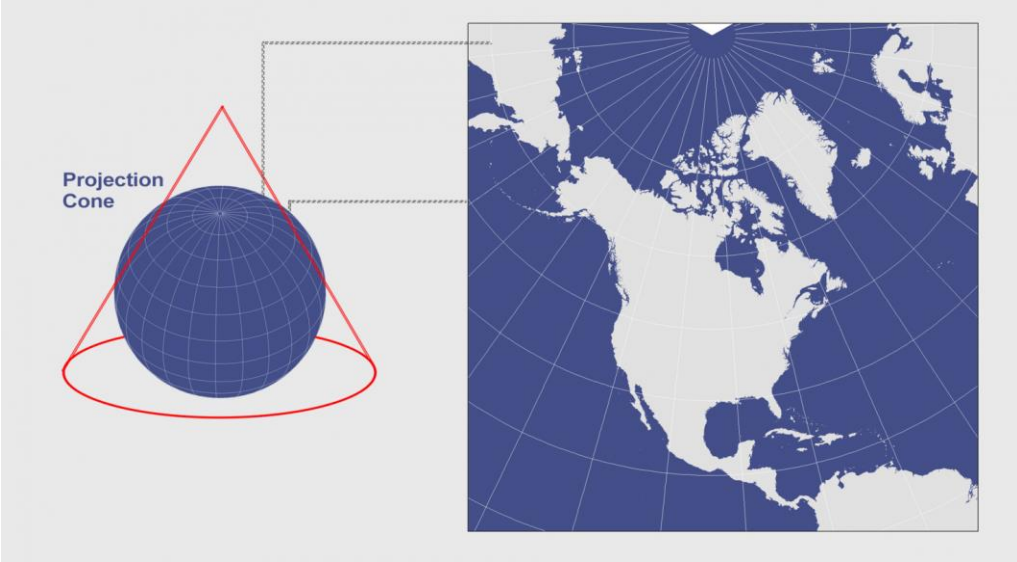


Every projection has strengths and weaknesses. All in all, it is up to the cartographer to determine what projection is most favorable for its purpose.

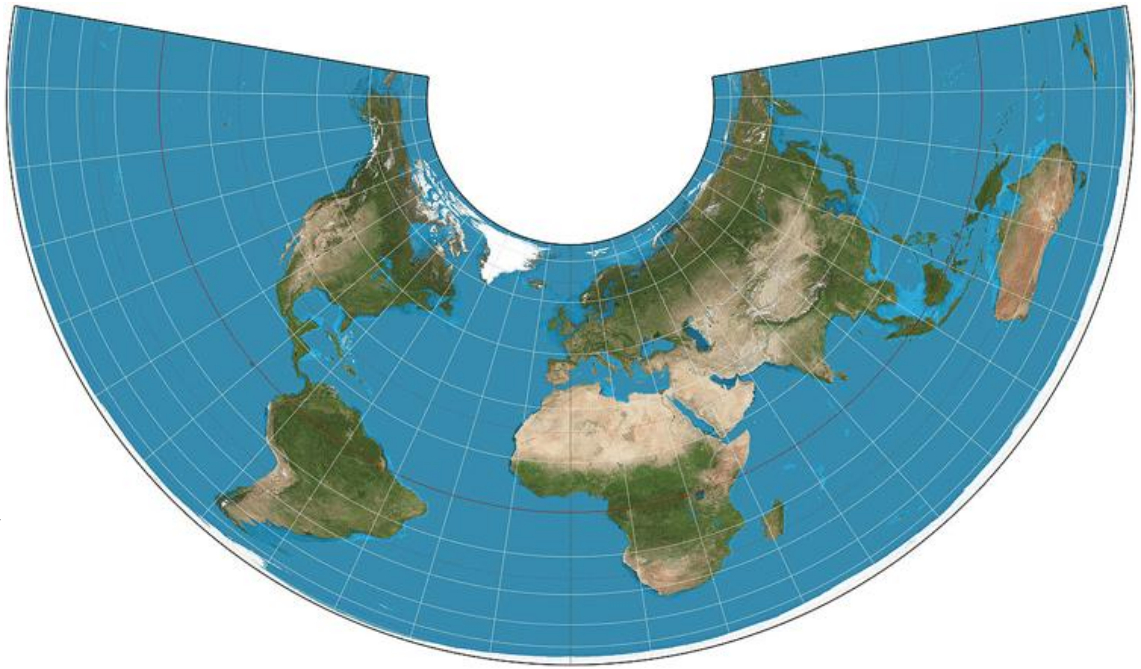
CONIC PROJECTIONS

When you place a cone on the Earth and unwrap it, this results in a conic projection. For example, Albers Equal Area Conic and the Lambert Conformal Conic projections are conic projections.

Both of these map projections are well-suited for mapping long east-west regions because distortion is constant along common parallels.



CONIC PROJECTIONS

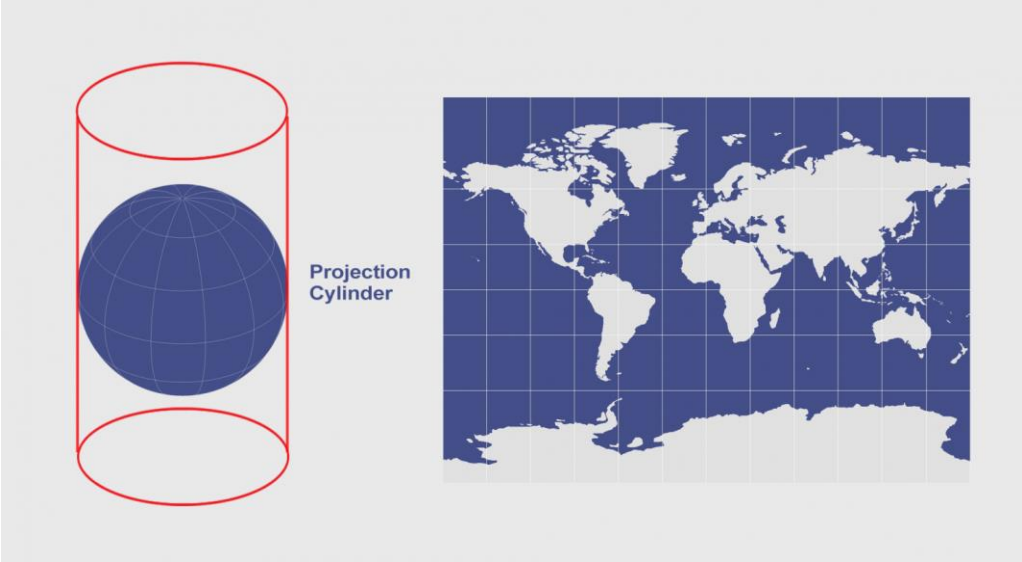


But they struggle at projecting the whole planet. While area is distorted, scale is mostly preserved. For conic map projections, distance at the bottom of the image suffers with the most distortion.

CYLINDRICAL PROJECTIONS

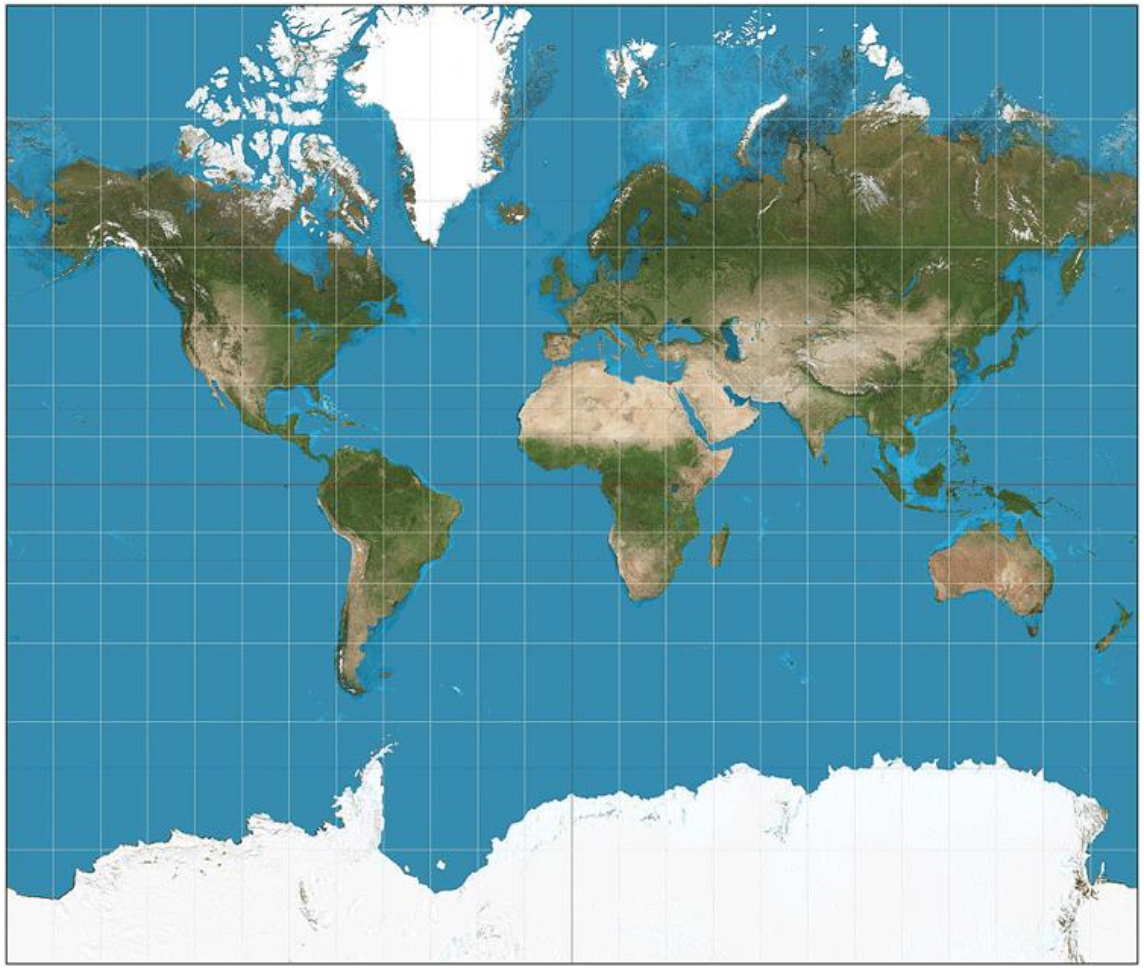
When you place a cylinder around a globe and unravel it, you get the cylindrical projection.

Strangely enough, you see cylindrical map projections like the Mercator and Miller for wall maps even though it inflates the Arctic.



CYLINDRICAL PROJECTIONS

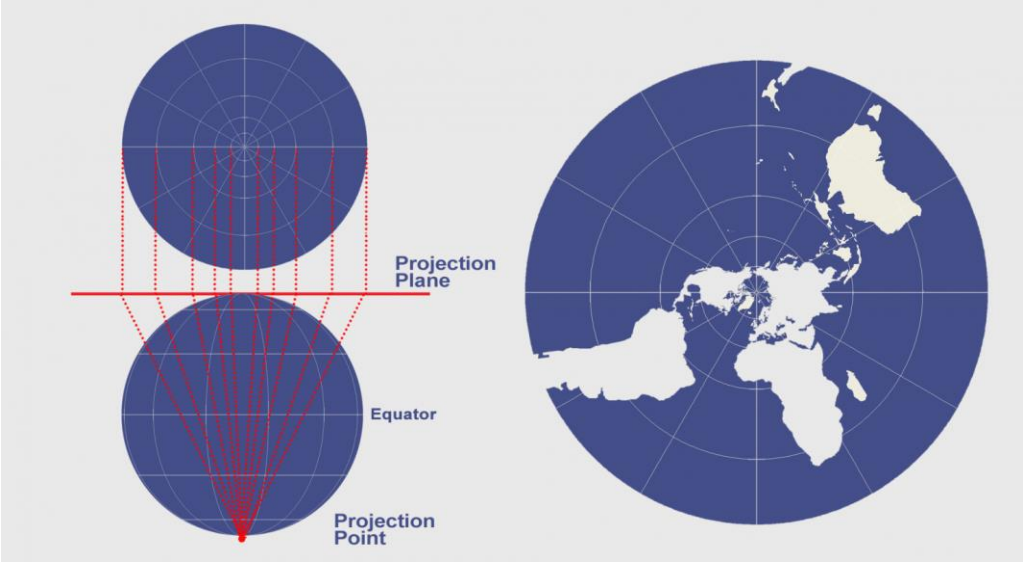
But it makes sense why navigators and even Google Maps use the Mercator projections – it's all because of the unique properties of cylinders and north always facing up.



AZIMUTHAL PROJECTIONS

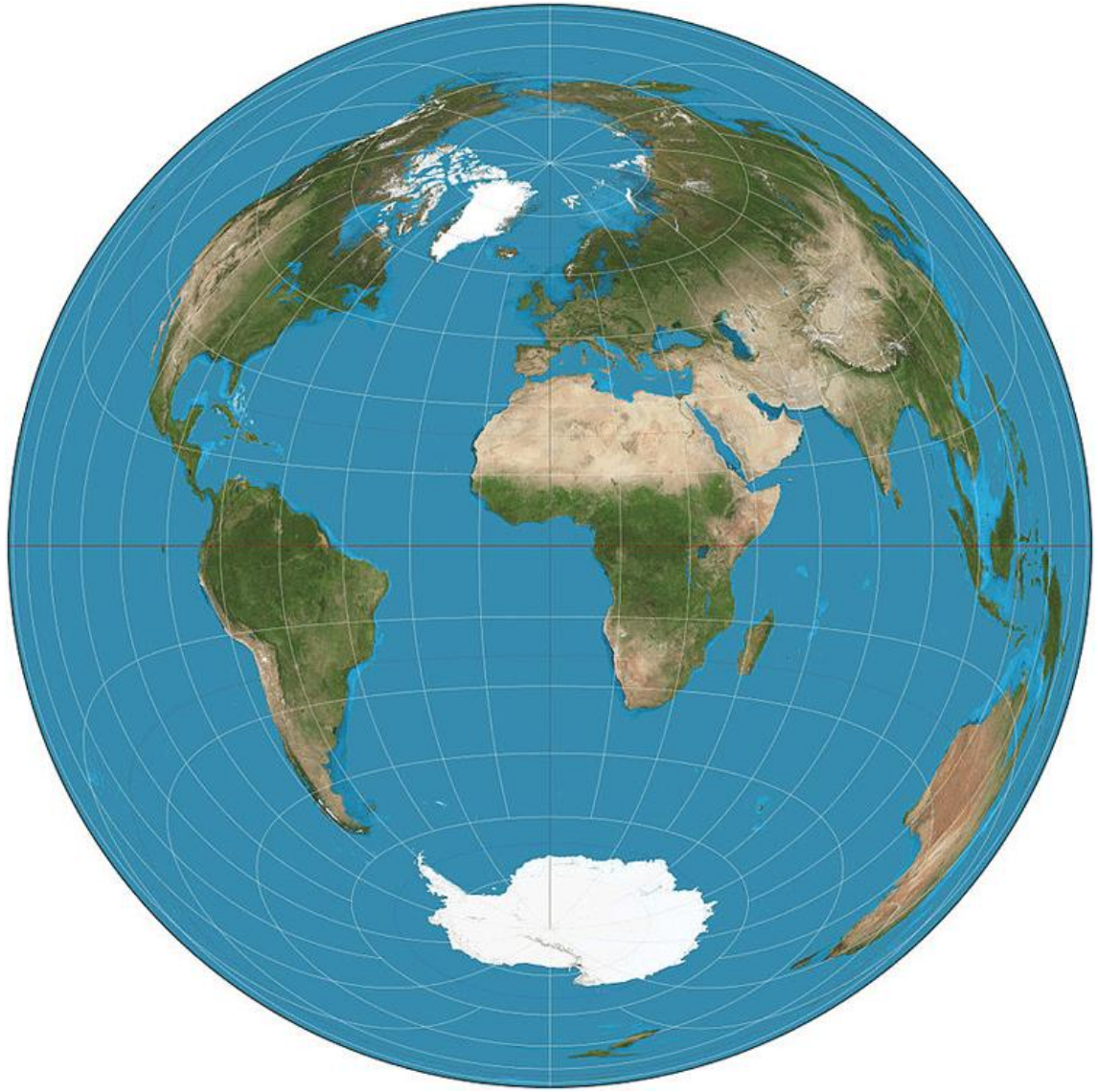
Plots the surface of Earth using a flat plane. Similar to light rays radiating from a source following straight lines, those light rays intercept the globe onto a plane at various angles.

The main features of azimuthal map projections are straight meridian lines, radiating out from a central point, parallels that are circular around the central point, and equidistant parallel spacing.

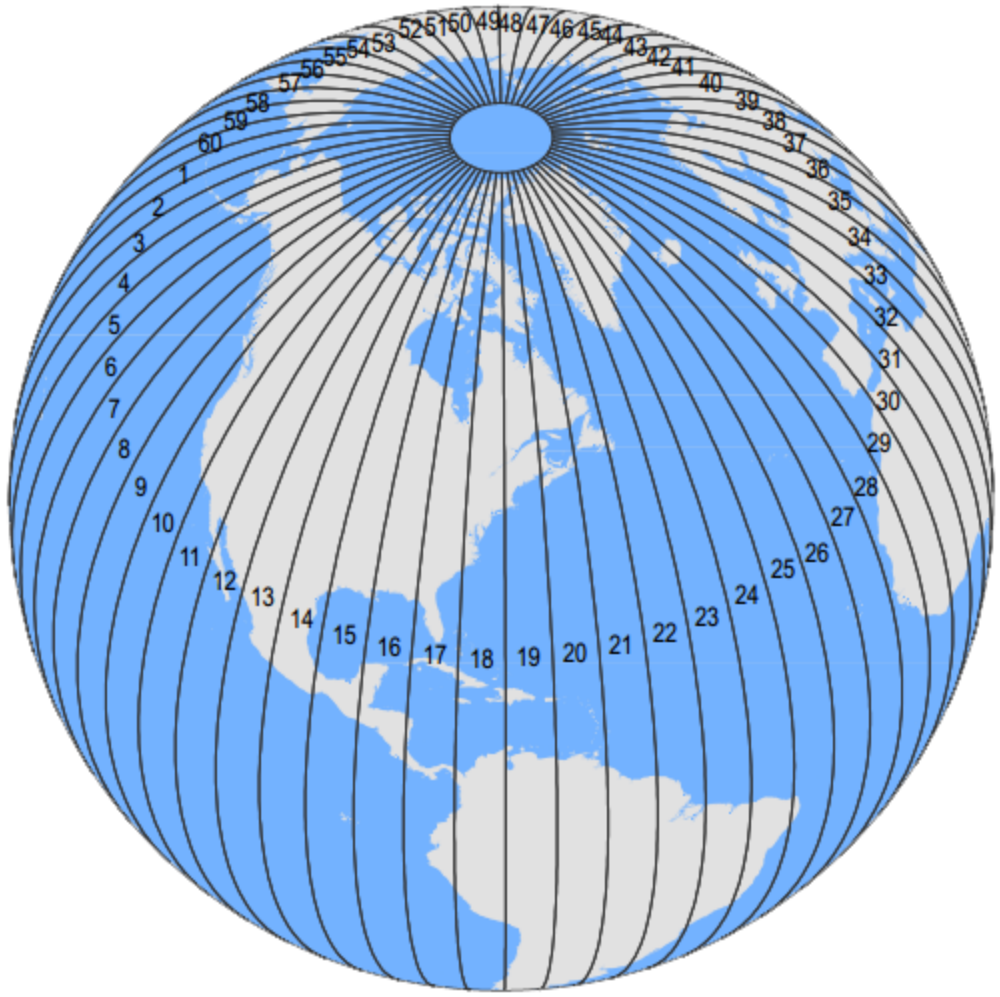


AZIMUTHAL PROJECTIONS

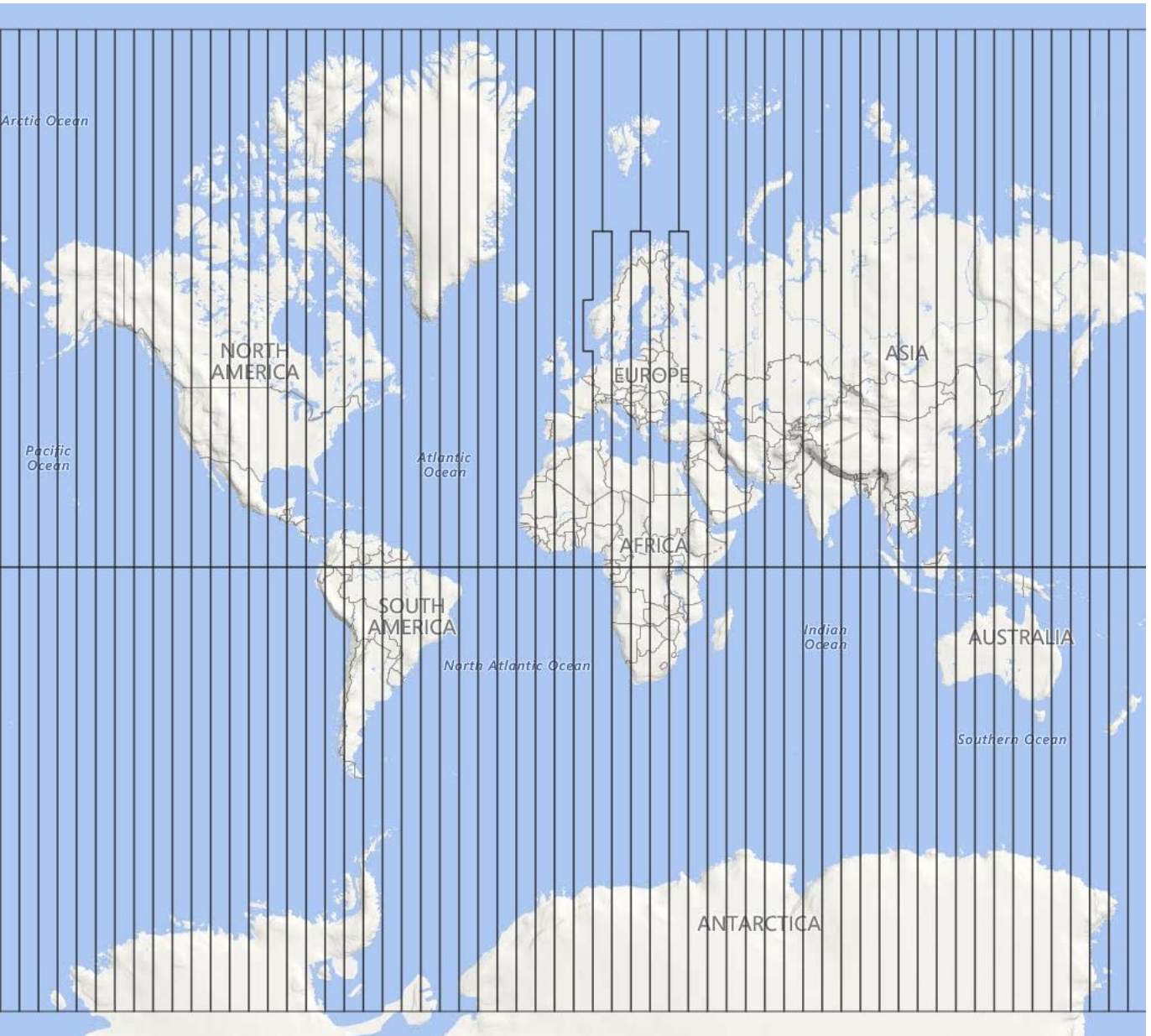
Azimuthal maps are beneficial for finding direction from any point on the Earth using the central point as a reference.



A UTM zone is a 6° segment of the Earth. Because a circle has 360°, this means that there are 60 UTM zones on Earth. (360 ÷ 6 = 60), right?

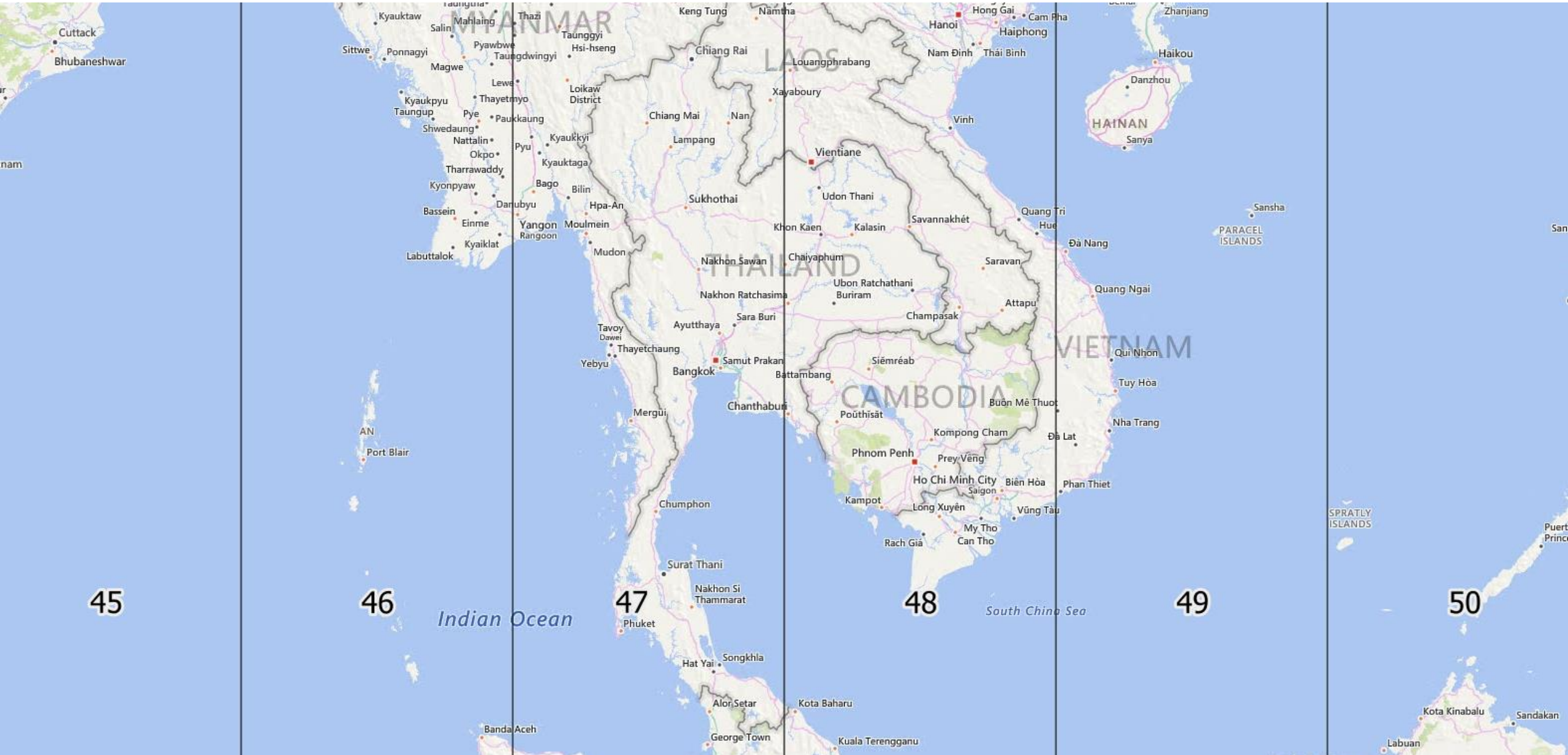


Coordinate systems – Universal Transverse Mercator (UTM)

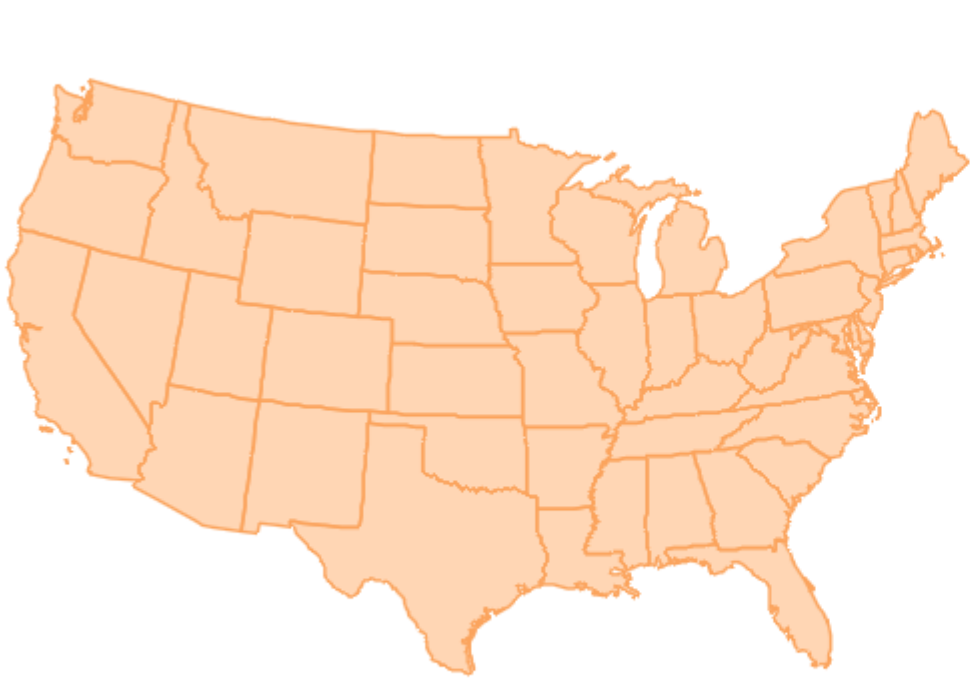


Instead of using latitude and longitude coordinates, each 6° wide UTM zone has a central meridian of 500,000 meters.

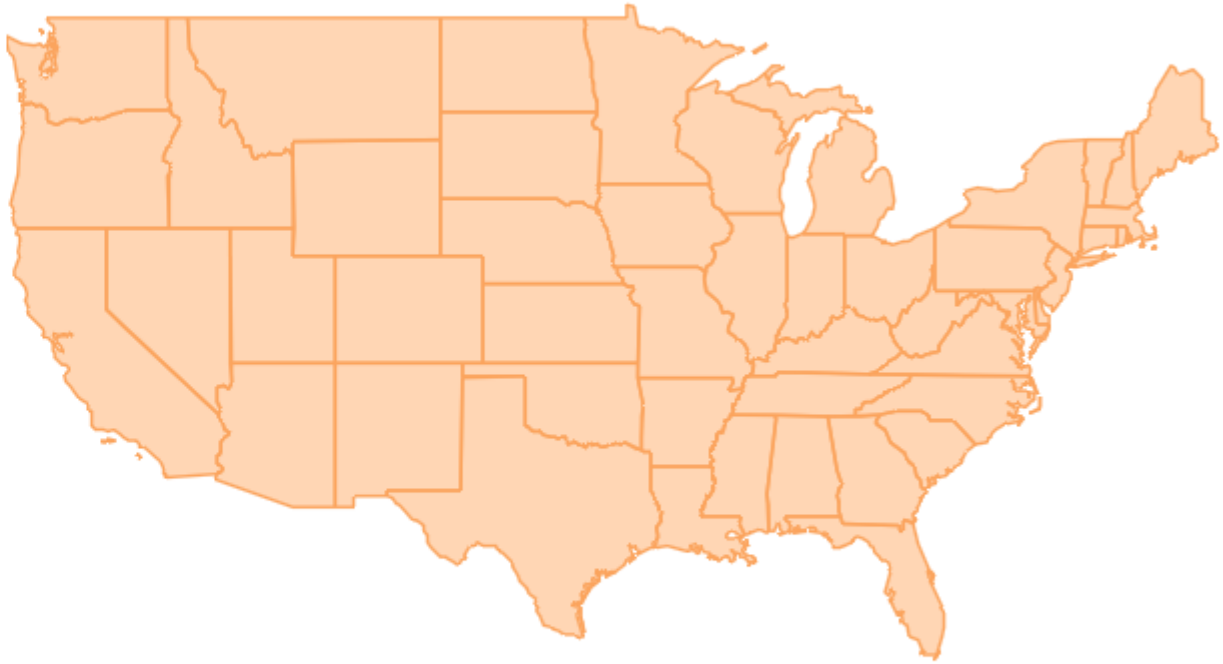
Coordinate systems – Universal Transverse Mercator (UTM)



Spatial reference systems are a key component of writing spatial applications. A spatial reference helps describe where features are located in the real world. Different spatial reference systems are used for different purposes.



102003 - Albers Equal Area



3857 - Web Mercator

Core Concepts

160 minutes

Maps are containers used to manage references to layers and basemaps.

Views are used to display the map layers and handle user interactions, popups, widgets, and the map location.

Working with maps

Maps are created from the Map class. Map objects are always passed to a View object. There are two View classes used to display maps: the MapView class for 2D maps and the SceneView class for 3D maps.

Create a new map

One way to create a map is to make a new instance of the Map class while specifying a basemap and optionally a collection of layers.

```
const myMap = new Map({                  // Create a Map object
  basemap: 'streets-vector',
  layers: additionalLayers               // Optionally, add additional layers collection
});

const mapView = new MapView({            // The View for the Map object
  map: myMap,
  container: 'mapDiv'
});
```

Working with Views

The primary role of the view is to display layers, popups, and UI widgets, handle user interactions, and to specify which portion of the world the map should be focused on (i.e. the "extent" of the map).

Create a view

There are separate classes for creating views for maps and scenes: a MapView and SceneView class. A MapView displays a 2D view, and a SceneView displays a 3D view, of a Map object.

For a map to be visible, a view object requires a Map object and a String identifying the id attribute of a div element or a div element reference.

```
const mapView = new MapView({          // Create MapView object
  map: myMap,
  container: 'mapViewDiv'
});
```

```
const sceneView = new SceneView({      // Create SceneView object
  map: myMap,
  container: "sceneViewDiv"
});
```

gis

Set the visible portion of the map

The initial position for the MapView and SceneView can be set by setting the center and the zoom or scale properties when the view is created.

```
const view = new MapView({
  center: [-112, 38],           // The center of the map as lon/lat
  zoom: 13                      // Sets the zoom level of detail (LOD) to 13
});
```

When using the SceneView (3D), the position of the observer can by set by defining the properties of the camera.

```
var view = new SceneView({
  camera: {
    position: [
      -122,       // lon
      38,         // lat
      50000       // elevation in meters
    ],
    heading: 95,  // direction the camera is looking
    tilt: 65      // tilt of the camera relative to the ground
  }
});
```

Animate the view to a new position

The goTo method of MapView also changes the location of the view but provides the additional option to transition smoothly. This technique is often used to "fly" from one location to another on the surface or to zoom to results of a search.

The goTo method can accept a Geometry, Graphic, or Viewpoint object. Additional options can control the animation.

```
view.goTo({                              // go to point with a custom animation duration
  target: {
    center: [-114, 39]
  }, {
  duration: 5000
});
```

Interacting with the view

The view is also responsible for handling user interaction and displaying popups. The view provides multiple event handlers for user interactions such as mouse clicks, keyboard input, touch screen interactions, joysticks, and other input devices.

When a user clicks on the map, the default behavior is to show any popups that have been pre-configured in your layers. This behavior can also be approximated manually with the following code by listening for the click event and using the hitTest() method to find features where the user clicked.

```
view.popup.autoOpenEnabled = false;  // Disable the default popup behavior

view.on("click", function (event) { // Listen for the click event
  view.hitTest(event).then(function (hitTestResults) { // Search for features where the user clicked
    if (hitTestResults.results) {
      view.popup.open({ // open a popup to show some of the results
        location: event.mapPoint,
        title: "Hit Test Results",
        content: hitTestResults.results.length + "Features Found"
      });
    }
  })
});
```

Adding widgets and UI components to the view

The view is also a container for overlaying widgets and HTML Elements. The view.ui provides a DefaultUI container that is used to display the default widgets for the view. Additional widgets and HTML Elements can also be added to the view by using the view.ui.add method. The code snippet below demonstrates adding widgets that allows users to search for an address or place.

```
var searchWidget = new Search({
  view: view
});

// Add the search widget to the top right corner of the view
view.ui.add(searchWidget, {
  position: "top-right"
});
```

Layers are collections of data that can be used in a Map. Layer data can be created on the client, hosted by ArcGIS Online and ArcGIS Enterprise, or hosted by external servers.

Data as collections of features

Layers are often used to manage and display large collections of features. Features are records of geographical locations or entities. Every feature contains spatial coordinates defined for a type of geometry (point, polyline, or polygon) and attribute fields that store other information. These collections of features can be thought of as:

- Structured if every feature has the same geometry type and the same attributes keys
- Unstructured if any features have different geometry types or different attributes keys

When working with a collection of features the general rule of thumb is:

- If the data is structured use FeatureLayer to display the data
- If unstructured use GraphicsLayer to display the data

Core layer types

The ArcGIS API for JavaScript has a number of layer classes that can be used to access and display layer data. All classes inherit from Layer. The class used depends on the format of the data and where the data is stored. Each layer type also exposes a different set of capabilities.

Below is a list of the most common layer classes.

Class	Data Storage	Capabilities
FeatureLayer	Geographic data stored in ArcGIS Online or ArcGIS Enterprise.	Displaying, querying, filtering and editing large amounts of geographic features.
GraphicsLayer	Geographic data stored temporarily in memory.	Displaying individual geographic features as graphics, visual aids or text on the map.
CSVLayer/KMLLayer/GeoJSONLayer	Geographic or tabular data stored in an external file accessed over a network.	Displaying data stored in an external file format as a layer.
TileLayer/VectorTileLayer	Datasets stored in a tile scheme for fast rendering.	Displaying basemaps and other tiled datasets for geographic context.
MapImageLayer	Geographic data stored in ArcGIS Enterprise and rendered as an image.	Displaying layers dynamically rendered by an ArcGIS Server service.
ImageryLayer	Georeferenced imagery stored in ArcGIS Enterprise.	Displaying satellite or other imagery data.

gis

Displaying data sources with a FeatureLayer

A FeatureLayer is a layer that references a collection of geographic features. All features in the collection must have the same geometry type and attribute keys.

Feature layer data sources can be either in memory from data loaded by the application or the layer will request data from a REST API service hosted on ArcGIS Online or ArcGIS Enterprise. Hosting your data in ArcGIS Online or ArcGIS Enterprise is the preferred method, especially for accessing and displaying large amounts of geographic data. Feature layers are highly optimized on both the client and the server for fast display and support a variety of other features including

- User interaction and popups
- Client side filtering, querying and analysis
- Editing
- Data driven visualization

Client-side data sources

Typically layer data is loaded from a REST API service hosted on ArcGIS Online or ArcGIS Enterprise however, it is also possible to create a feature layer directly from a collection of features in memory.

For example, a collection of features in Los Angeles, California, is given below in JSON format. This data can be transformed into a format acceptable for displaying in a FeatureLayer.


```
{
    "places": [
        {
            "id": 1,
            "address": "200 N Spring St, Los Angeles, CA 90012",
            "longitude": -118.24354,
            "latitude": 34.05389
        },
        {
            "id": 2,
            "address": "419 N Fairfax Ave, Los Angeles, CA 90036",
            "longitude": -118.31966,
            "latitude": 34.13375
        }
    ]
}
```

The first step to create a feature layer from the above JSON data is to transform each place into a Graphic object with the attributes and geometry properties.

Property	Type	Description
attributes	Object	Key-value pairs used to store geographical information about the feature
geometry	Geometry	Provides the location for a feature relative to a coordinate system. Possible values are Point, Polygon, and Polyline objects

The following code sample transforms the array of places into an array of Graphic objects.

```
const graphics = places.map(function (place) {
  return new Graphic({
    attributes: {
      ObjectId: place.id,
      address: place.address
    },
    geometry: {
      longitude: place.longitude,
      latitude: place.latitude
    }
  });
});
```

The second step in creating a FeatureLayer is to actually create a FeatureLayer object, and specify at least the objectIdField, fields, renderer, and source properties described in the following table.

Property	Type	Description
source	Collection<Graphic>	The collection of Graphic objects used to create the feature layer
renderer	Renderer	The renderer used to display a symbol at the feature's location
objectIdField	String	The name of the field identifying the feature
fields	Object[]	An array of JavaScript objects with field names and values
popupTemplate	PopupTemplate	The popup template for the feature

The following code sample creates a new FeatureLayer and explicitly sets the source property to graphics. Autocasting is used to set the renderer, popup, and fields properties.

```
const renderer = {
  type: 'simple',                    // autocasts as new SimpleRenderer()
  symbol: {                          // autocasts as new SimpleMarkerSymbol()
    type: 'simple-marker',
    color: '#102A44',
    outline: {                       // autocasts as new SimpleLineSymbol()
      color: '#598DD8',
      width: 2
    }
  }
};

const popupTemplate = {               // autocasts as new PopupTemplate()
  title: 'Places in Los Angeles',
  content: [{
    type: 'fields',
    fieldInfos: [
      {
        fieldName: 'address',
        label: 'Address',
        visible: true
      }
    ]
  }]
};
```

The following code sample creates a new FeatureLayer and explicitly sets the source property to graphics. Autocasting is used to set the renderer, popup, and fields properties.

```
const fields = [
  {
    name: 'ObjectID',
    alias: 'ObjectID',
    type: 'oid'
  },
  {
    name: 'address',
    alias: 'address',
    type: 'string'
  }
];

const featureLayer = new FeatureLayer({
  source: graphics,
  renderer: renderer
  popupTemplate: popupTemplate,
  objectIdField: 'ObjectID',  // This must be defined when creating a layer from `Graphic` objects
  fields: fields
});

map.layers.add(featureLayer);
```

Server-side data sources

FeatureLayer also supports collections of features returned from a a REST API service specified with by the url property. This is the most effective way to access and display large datasets. The feature layer will work with the feature service to retrieve features as efficiently as possible and, if enabled, will provide access to additional capabilities such as editing.

```
const layer = new FeatureLayer({
  url: "https://services3.arcgis.com/GVgbJbqm8hXASVYi/arcgis/rest/services/Trailheads/FeatureServer/0"
});

map.layers.add(layer);
```

In addition to URLs you can also reference layer items stored in ArcGIS Online or ArcGIS Enterprise. These items reference a REST API service which stores the layer's dat as well as additional configuration options.

```
const layer = new FeatureLayer({
  portalItem: {
    id: "883cedb8c9fe4524b64d47666ed234a7",
    portal: "https://www.arcgis.com"                // Default: The ArcGIS Online Portal
  }
});

map.layers.add(layer);
```

Displaying graphics with a GraphicsLayer

Graphics are typically used for adding text, shapes, and images with different geometries to a map. The simplest way to create a graphics layer is to create Graphic objects as an array, and pass this array to the graphics property of a new GraphicsLayer object.

Every Graphic class includes the following properties:

Property	Type	Description
attributes	Object	Key-value pairs used to store geographical information about features
geometry	Geometry	Provides the location for a feature relative to a coordinate system Possible values are Point, Polygon, and Polyline objects
popupTemplate	PopupTemplate	The popup template for the graphic
symbol	Symbol	Defines how the graphic will be rendered in the layer

The below code sample creates a new Graphic object with a Point geometry type, a popup, and a symbol. It then creates a new GraphicsLayer by passing an array of graphics to the graphics property.

```
const attributes = {
  name: 'LA City Hall',
  address: '200 N Spring St, Los Angeles, CA 90012'
};

const point = {
  type: 'point',                    // autocasts as new Point()
  longitude: -118.24354,
  latitude: 34.05389
};

const symbol = {
  type: 'simple-marker',            // autocasts as new SimpleMarkerSymbol()
  color: [226, 119, 40],
  outline: {                        // autocasts as SimpleLineSymbol()
    color: [255, 255, 255],
    width: 2
  }
};
```

Core Concepts – Layers and data

```
const popupTemplate = {                // autocasts as new PopupTemplate()
  title: 'Places in Los Angeles',
  content: [{
    type: 'fields',
    fieldInfos: [
      {
        fieldName: 'name',
        label: 'Name',
        visible: true
      },
      {
        fieldName: 'address',
        label: 'Address',
        visible: true
      }
    ]
  }]
};
```

```
const pointGraphic = new Graphic({
  attributes: attributes,
  geometry: point,
  symbol: symbol,
  popupTemplate: popupTemplate
});

var graphicsLayer = new GraphicsLayer({
  graphics: [pointGraphic]
});

map.layers.add(graphicsLayer);
```

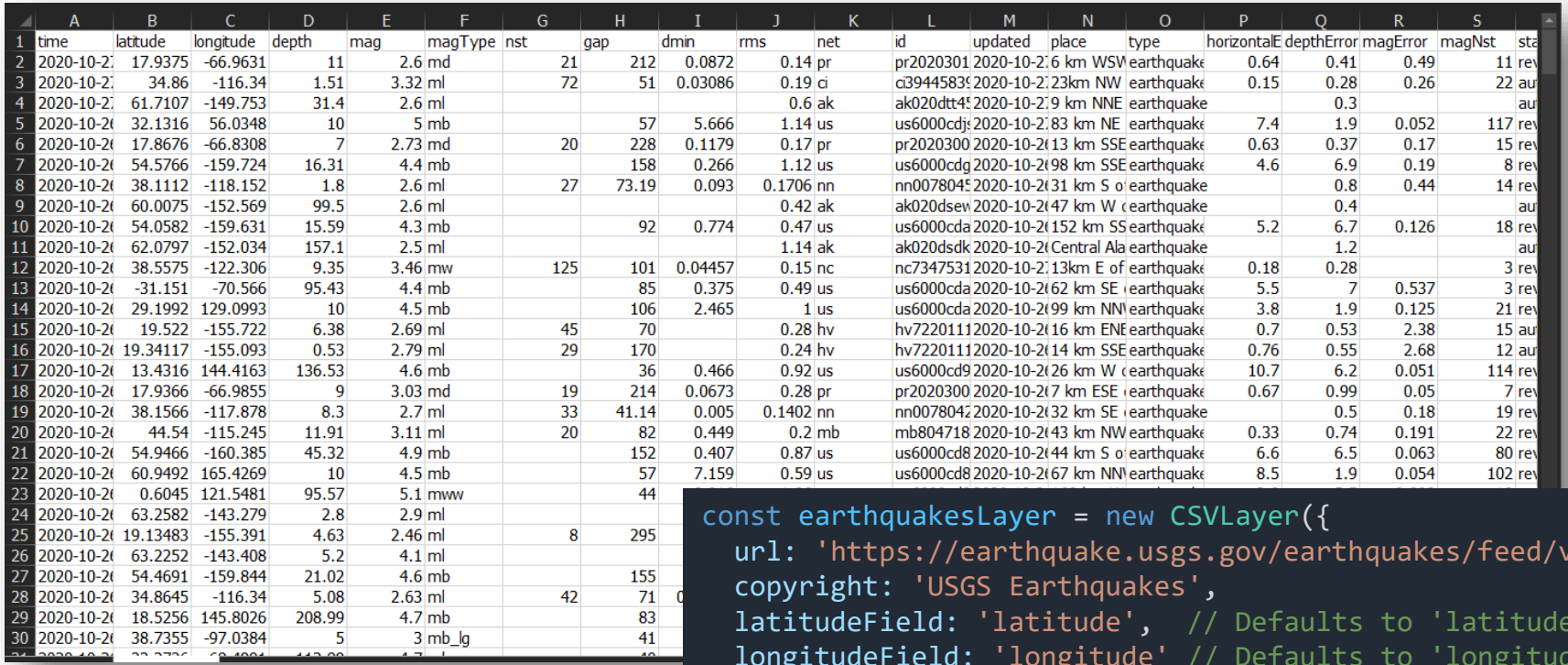
Working with external data sources

Additional types of data and files are directly supported by specific subclasses of Layer. These include specific types of layers for working with external files like CSV or GeoJSON files or integrating external services such as Bing Maps.

Layer Subclass	Data Source	Data Types	Features	Limitations
CSVLayer	CSV files	- Vector graphics downloaded as points	Client-side processing, popup templates, renderers with 2D and 3D symbols	May require large download depending on the number of features
GeoRSSLayer	GeoRSS feed	Vector graphics as points, polylines, and polygons	- Graphics storage - Popup templates	- No 3D support No support for renderers
GeoJSONLayer	GeoJSON file	Vector graphics as points, polylines, and polygons	Create layer from GeoJSON data	- Each GeoJSON Layer accepts a single geometry type - Data must comply with RFC 7946 specification
KMLLayer	KML data source	N/A	N/A	N/A

Layer Subclass	Data Source	Data Types	Features	Limitations
OGCFeatureLayer	-OGC API - Features	Points, polylines, polygons	Renderers, labels, popups	Data must comply with the RFC 7946 specification which states that the coordinates are in SpatialReference WGS84
WMSLayer	- WMS Service Portal Item	Raster data exported as a single image	OGC specification	N/A
WMTSLayer	- WMTS tile services - Portal Item	Image tiles	OGC specification	N/A
OpenStreetMapLayer	OpenStreetMap tile services	Image tiles	Displays OpenStreetMap tiled content	N/A
BingMapsLayer	Bing Spatial Data Service data	N/A	N/A	N/A

Each of these layers requires different properties depending on how they are initialized. Refer to each layer type for more details. An example of creating a CSVLayer layer is shown below.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	
1	time	latitude	longitude	depth	mag	magType	nst	gap	dmin	rms	net	id	updated	place	type	horizontalE	depthError	magError	magNst	sta
2	2020-10-2	17.9375	-66.9631	11	2.6	md	21	212	0.0872	0.14	pr	pr2020301	2020-10-2	6 km WSW	earthquake	0.64	0.41	0.49	11	rev
3	2020-10-2	34.86	-116.34	1.51	3.32	ml	72	51	0.03086	0.19	ci	ci3944583	2020-10-2	23km NW	earthquake	0.15	0.28	0.26	22	aut
4	2020-10-2	61.7107	-149.753	31.4	2.6	ml				0.6	ak	ak020dtt4	2020-10-2	9 km NNE	earthquake		0.3			aut
5	2020-10-2	32.1316	56.0348	10	5	mb		57	5.666	1.14	us	us6000cdj	2020-10-2	83 km NE	earthquake	7.4	1.9	0.052	117	rev
6	2020-10-2	17.8676	-66.8308	7	2.73	md	20	228	0.1179	0.17	pr	pr2020300	2020-10-2	13 km SSE	earthquake	0.63	0.37	0.17	15	rev
7	2020-10-2	54.5766	-159.724	16.31	4.4	mb		158	0.266	1.12	us	us6000cdg	2020-10-2	98 km SSE	earthquake	4.6	6.9	0.19	8	rev
8	2020-10-2	38.1112	-118.152	1.8	2.6	ml	27	73.19	0.093	0.1706	nn	nn0078045	2020-10-2	31 km S o	earthquake		0.8	0.44	14	rev
9	2020-10-2	60.0075	-152.569	99.5	2.6	ml				0.42	ak	ak020dsew	2020-10-2	47 km W o	earthquake		0.4			aut
10	2020-10-2	54.0582	-159.631	15.59	4.3	mb		92	0.774	0.47	us	us6000cda	2020-10-2	152 km SS	earthquake	5.2	6.7	0.126	18	rev
11	2020-10-2	62.0797	-152.034	157.1	2.5	ml				1.14	ak	ak020dsdk	2020-10-2	Central Ala	earthquake		1.2			aut
12	2020-10-2	38.5575	-122.306	9.35	3.46	mw	125	101	0.04457	0.15	nc	nc7347531	2020-10-2	13km E of	earthquake	0.18	0.28		3	rev
13	2020-10-2	-31.151	-70.566	95.43	4.4	mb		85	0.375	0.49	us	us6000cda	2020-10-2	62 km SE o	earthquake	5.5	7	0.537	3	rev
14	2020-10-2	29.1992	129.0993	10	4.5	mb		106	2.465	1	us	us6000cda	2020-10-2	99 km NNW	earthquake	3.8	1.9	0.125	21	rev
15	2020-10-2	19.522	-155.722	6.38	2.69	ml	45	70		0.28	hv	hv7220111	2020-10-2	16 km ENE	earthquake	0.7	0.53	2.38	15	aut
16	2020-10-2	19.34117	-155.093	0.53	2.79	ml	29	170		0.24	hv	hv7220111	2020-10-2	14 km SSE	earthquake	0.76	0.55	2.68	12	aut
17	2020-10-2	13.4316	144.4163	136.53	4.6	mb		36	0.466	0.92	us	us6000cd9	2020-10-2	26 km W o	earthquake	10.7	6.2	0.051	114	rev
18	2020-10-2	17.9366	-66.9855	9	3.03	md	19	214	0.0673	0.28	pr	pr2020300	2020-10-2	7 km ESE o	earthquake	0.67	0.99	0.05	7	rev
19	2020-10-2	38.1566	-117.878	8.3	2.7	ml	33	41.14	0.005	0.1402	nn	nn0078042	2020-10-2	32 km SE o	earthquake		0.5	0.18	19	rev
20	2020-10-2	44.54	-115.245	11.91	3.11	ml	20	82	0.449	0.2	mb	mb804718	2020-10-2	43 km NW	earthquake	0.33	0.74	0.191	22	rev
21	2020-10-2	54.9466	-160.385	45.32	4.9	mb		152	0.407	0.87	us	us6000cd8	2020-10-2	44 km S o	earthquake	6.6	6.5	0.063	80	rev
22	2020-10-2	60.9492	165.4269	10	4.5	mb		57	7.159	0.59	us	us6000cd8	2020-10-2	67 km NNW	earthquake	8.5	1.9	0.054	102	rev
23	2020-10-2	0.6045	121.5481	95.57	5.1	mww		44												
24	2020-10-2	63.2582	-143.279	2.8	2.9	ml														
25	2020-10-2	19.13483	-155.391	4.63	2.46	ml	8	295												
26	2020-10-2	63.2252	-143.408	5.2	4.1	ml														
27	2020-10-2	54.4691	-159.844	21.02	4.6	mb		155												
28	2020-10-2	34.8645	-116.34	5.08	2.63	ml	42	71												
29	2020-10-2	18.5256	145.8026	208.99	4.7	mb		83												
30	2020-10-2	38.7355	-97.0384	5	3	mb_lg		41												

```
const earthquakesLayer = new CSVLayer({
  url: 'https://earthquake.usgs.gov/earthquakes/feed/v1.0/summary/2.5_week.csv',
  copyright: 'USGS Earthquakes',
  latitudeField: 'latitude',  // Defaults to 'latitude'
  longitudeField: 'longitude' // Defaults to 'longitude'
});

map.layers.add(earthquakesLayer)
```

Using basemaps and tile layers

Basemaps are used to provide geographic context to a map by displaying roads, boundaries, buildings and other data. Basemaps are often served as tiles for faster rendering. Raster basemaps request pre-created images. Vector basemaps request data in a compressed binary format and style it on the client. The ArcGIS contains a curated set of basemaps.

Vector basemaps can be customized with the the Vector Tile Style Editor. Custom data can also be published as vector or raster tiles with ArcGIS Online or ArcGIS Enterprise.

The basemap for a specific Map object can be controlled with the basemap property which can be a string identifying a specific basemap or a Basemap object.

```
const map = new Map({
  basemap: 'streets-navigation-vector'
})
```

Working with Map Services from ArcGIS Enterprise

MapImageLayer is used to display data from a Map Service in ArcGIS Enterprise. Map Services often contain multiple sub layers and complex cartography. Map Services render data as a server side image that is dynamically generated and displayed on the client.

Using raster and imagery data

ImageryLayer is used to display imagery or other raster based data stored in an Image Service in ArcGIS Enterprise. ImageryLayer is often used to display and analyze raw imagery data captured from drones or satellites or for displaying scientific data.

This topic provides an overview of the many workflows you can use for querying and filtering data. Query and filter operations can be done against all features available in the service on the server-side or against all features available in the browser (or view) on the client-side.

First, we will review which layers allow you to query and filter subsets of features. In doing so, we must understand the concept of server-side vs client-side layers, and Layer vs LayerView.

Server-side and client-side layers

The ArcGIS API for JavaScript makes it possible for you to add data from many sources. Layers that allow you to query and filter subsets of their features can be grouped into server-side layers and client-side layers.

Server-side layers fetch only required features when they load. Afterwards, layers fetch their features from the server as needed or requested. These layers include: FeatureLayer, SceneLayer and StreamLayer. The server-side layer is created by setting the layer's url property to point to a service.

Client-side layers fetch all of their features at once and store them on the client-side when they load. Once these layers are loaded, there will be no more server-side requests. These layers include: CSVLayer and GeoJSONLayer. They are created by setting the layer's url property to a csv or geojson file. It also includes a FeatureLayer created from an array of client-side graphics by setting its source property.

Layer and LayerView

A LayerView is created when a layer is added to either a MapView or a SceneView. The LayerView is responsible for rendering features in the view. The layer view also provides methods and properties that give developers the ability to query, filter, and highlight graphics in the view on the client-side.

The following table shows a simplified steps that take place when the user adds a layer to a view.

Behavior	Server-side layers	Client-side layers
Layers	FeatureLayer, SceneLayer and StreamLayer	CSVLayer, GeoJSONLayer and client-side FeatureLayer

Initialization	Created by setting its url property to point to a server-side feature, scene, or stream service.
	
	`const layer = new FeatureLayer({`
	`  url: '<service url>'`
	`});`
	`view.map.add(layer);`
		CSVLayer and GeoJSONLayer are created by setting their url property. Client-side FeatureLayer is created by setting its source property.
		
		`const layer = new CSVLayer({`
		`  url: '<csv file url>'`
		`});`
		`view.map.add(layer);`

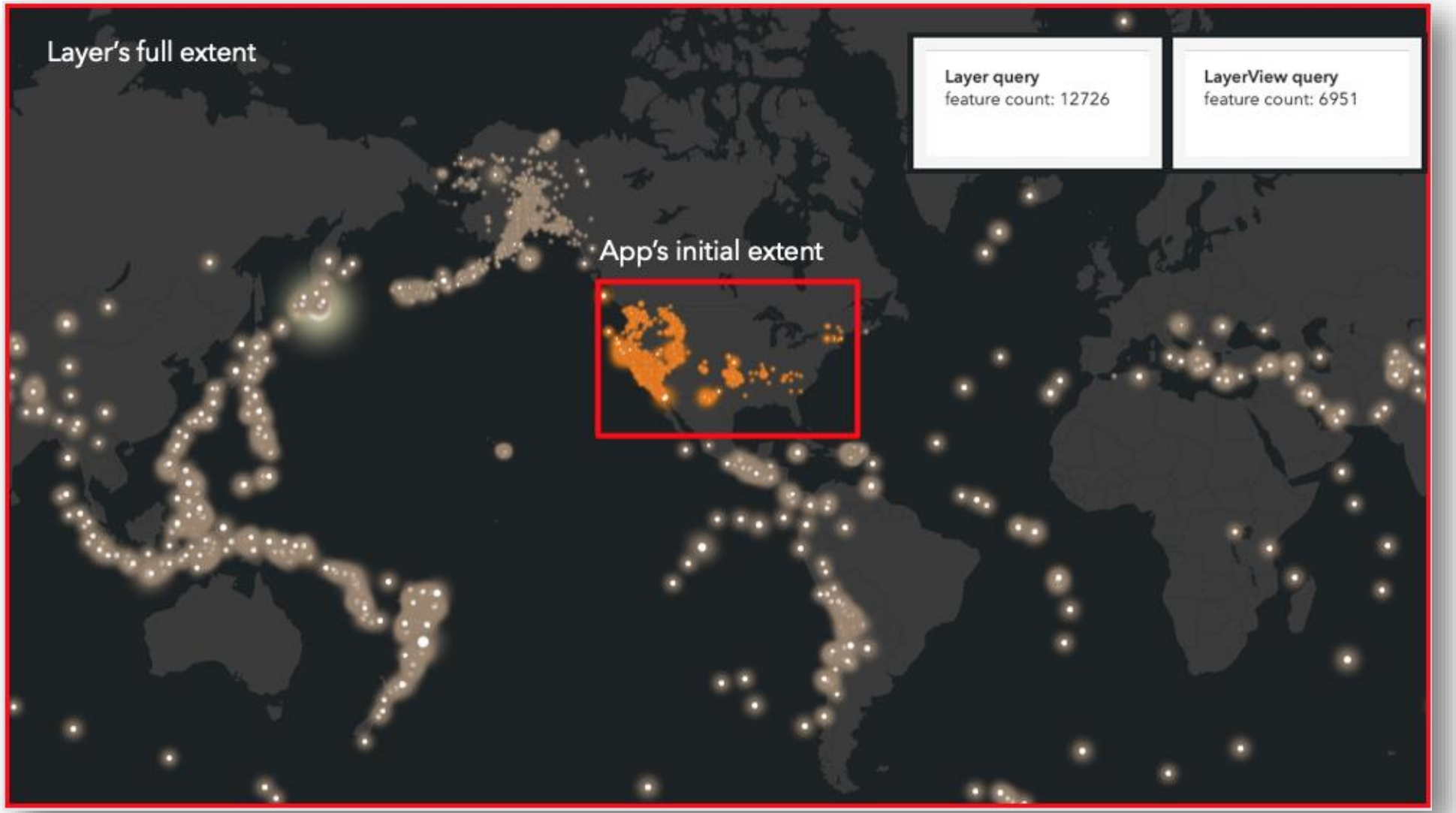
Behavior	Server-side layers	Client-side layers
Initial features fetching	The layer fetches only required features from the server.	The layer fetches all of its features when initialized and stores it on the client.

LayerView initialization	FeatureLayerView, SceneLayerView or StreamLayerView representing FeatureLayer, SceneLayer or StreamLayer is initialized containing features available for drawing.
	
	`view.whenLayerView(layer).then(function (lv) {`
	`  // now we have access to the layerView, an`
	`  // object representing the layer in the view`
	`});`
		CSVLayerView, GeoJSONLayerView, or FeatureLayerView representing CSVLayer, GeoJSONLayer, or FeatureLayer is initialized containing features available for drawing.
		
		`view.whenLayerView(layer).then(function (lv) {`
		`  // now we have access to the layerView, an`
		`  // object representing the layer in the view`
		`});`

Subsequent network requests	**Yes.** Subsequent network requests are made as needed.	**No.** All features is fetched on load.

The following image illustrates the features available for querying from a layer and a layer view. As you can see the layer has features covering much more area than the initial extent of the application. The layer properties and methods provide access to all of these features. When the layer is loaded, the layer view has access to features that are visible within the app's initial extent. Any operation called on the layer view after the app loads provides access to features visible in the view. The image also shows the count of features available on the layer versus on the layer view. The layerView feature count is much less because it returns features within the initial extent of the view while layer count represents all features in the layer.

Core Concepts – Query and filter



Querying

There are three types of queries: attribute, spatial, and statistic. This document provides detailed information on each type of query. Queries can be done on the layer or on its layer view.

Server-side and client-side queries

A **server-side query** is issued when a ***query...*** method is called on a server-side layer. The query is executed against all features available in the service.

```
(Feature | Scene)Layer // queries all features in the service
  .queryFeatures() // queries all features and returns a FeatureSet
  .queryExtent() // queries all features returns extent of features that satisfy query
  .queryFeatureCount() // queries all features and returns count of features
  .queryObjectIds() // queries all features and returns objectIds array of features
```

A **client-side query** is issued when a ***query...*** method is called on a client-side layer or any layer view. The query is executed against all features available in the layer or layer view.

```
(Feature | CSV | GeoJSON)Layer  // queries all features in the layer
(Feature | CSV | GeoJSON | Scene | Stream)LayerView // queries available features in view
  .queryFeatures() // queries features and returns a FeatureSet
  .queryExtent() // queries features returns extent of features that satisfy query
  .queryFeatureCount() // queries features and returns count of features
  .queryObjectIds() // queries features and returns objectIds array of features
```

Should I use client-side query or server-side query?

What matters?	Server side query	Client-side query
Speed and responsiveness	**No** for server-side layers. You are making network requests, so it is slower compared to client-side queries.	**Yes** for client-side layers and layer views.
Geometry precision	**Yes.** Geometry precision is preserved. Use when it is important to get accurate results with a precise geometry.	**Yes** for client-side layers to query user provided geometries. No for layer views as geometries are generalized for drawing. The results can be imprecise and change as the user zooms in the map. Examples include calculating an area for selected geometries, or getting points contained in a polygon.
Must query every feature	**Yes.** Use a server-side query to make sure that the query runs against all features. Paginated queries must be done when you need to get more than max record count.	**Yes** for CSVLayer, GeoJSONLayer and client-side FeatureLayer. No for layerViews as the query will only run against features that are available on the client-side.

Tips when using query methods on LayerViews

- Add fields to a layer's outFields at the time of the layer initialization to ensure that you have access to these fields on the client-side. By default the layer view only fetches the fields that are required for layer rendering, labeling, elevation info.
- If you query a layerView when the app loads, then you **must** wait until layerView's updating property becomes false to make sure that the features are loaded with the layerView. You can use watchUtils.whenFalseOnce if the query needs to run only once at the time of initialization.
- If you query a layerView each time the view extent changes, then you **must** wait until the layerView's updating property becomes false to make sure the layerView finished fetching the features for that extent.
- The client-side attribute values are case sensitive.

Filtering

Filters affect the availability of features in a layer or the visibility of features in a layer view. Features that satisfy the filter requirements will be displayed in the view. Filtering can take place on the server-side or on the client-side.

Server-side and client-side filtering

All layers covered in this guide have a definitionExpression property. Setting a definitionExpression on a server-side layer triggers a network request to fetch features that satisfy the definition expression. Setting a definition expression is useful when the dataset is large and you don't want to bring all features to the client for analysis. If the definition expression is set after the layer has been added to the map, the view will automatically refresh itself to display the features that satisfy the new definition expression.

```
// fetch all features that satisfy requirements from the service
(Feature | Scene | Stream)Layer
  .definitionExpression = "type = 'metal'";
```

A ***definitionExpression*** on a client-side layer will only display features that satisfy the definitionExpression. Setting a definitionExpression happens on the client-side against all features available in the layer.

```
// only display features that satisfy the requirements in the layer
(Feature | CSV | GeoJSON)Layer
  .definitionExpression = 'mag > 5';
```

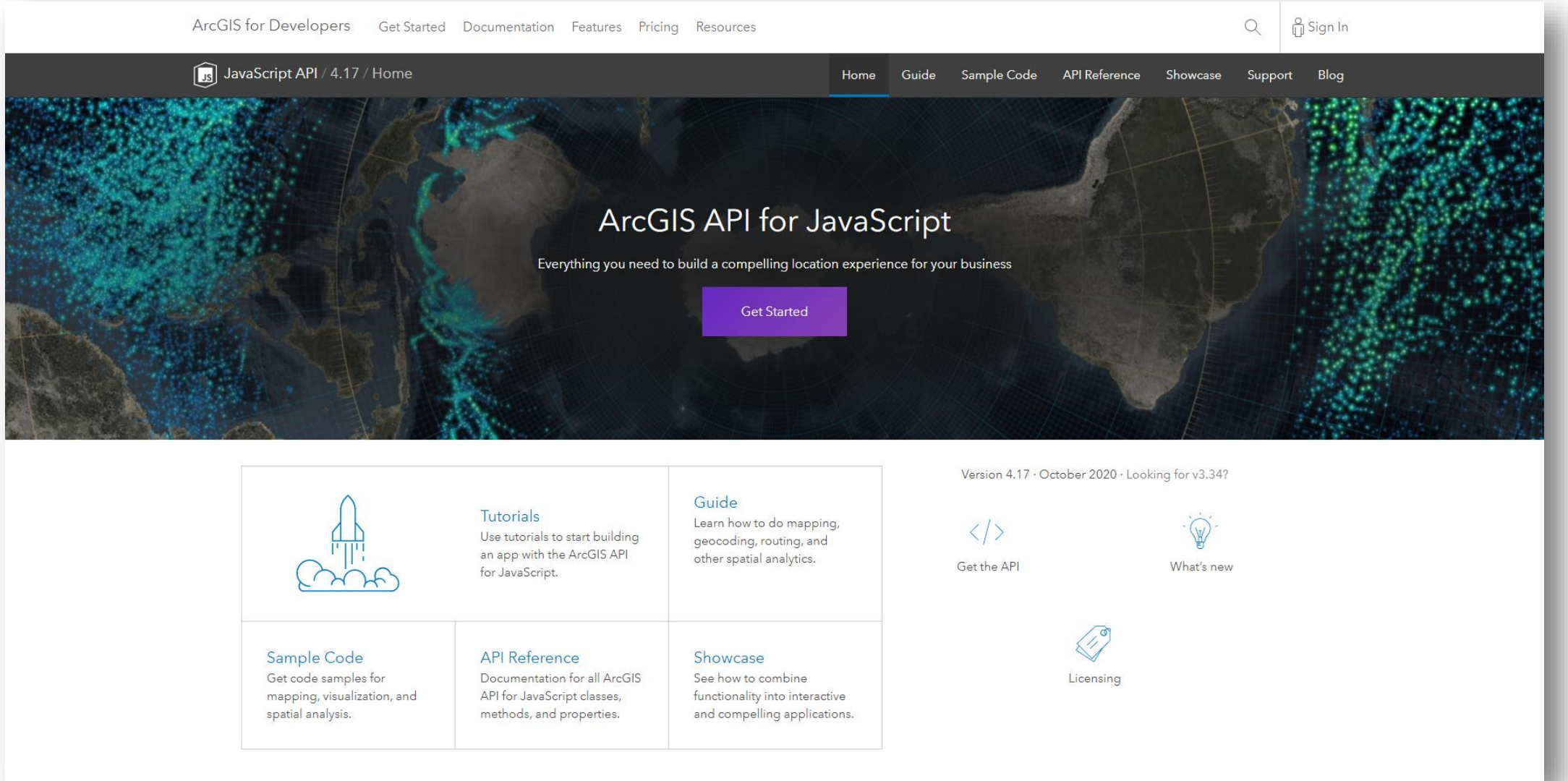
If a layer has a definitionExpression, all layerView queries and filters will honor the definitionExpression. This means only features that meet the layer's definitionExpression will be evaluated by the layer view's query and filter operations.

You can apply filters on features available for drawing by setting a filter on a LayerView. The FeatureFilter allows you to display the features that satisfy the filter requirements in the layer view. Since the filter is applied to a layer view, this happens on the client-side against features that are available for drawing.

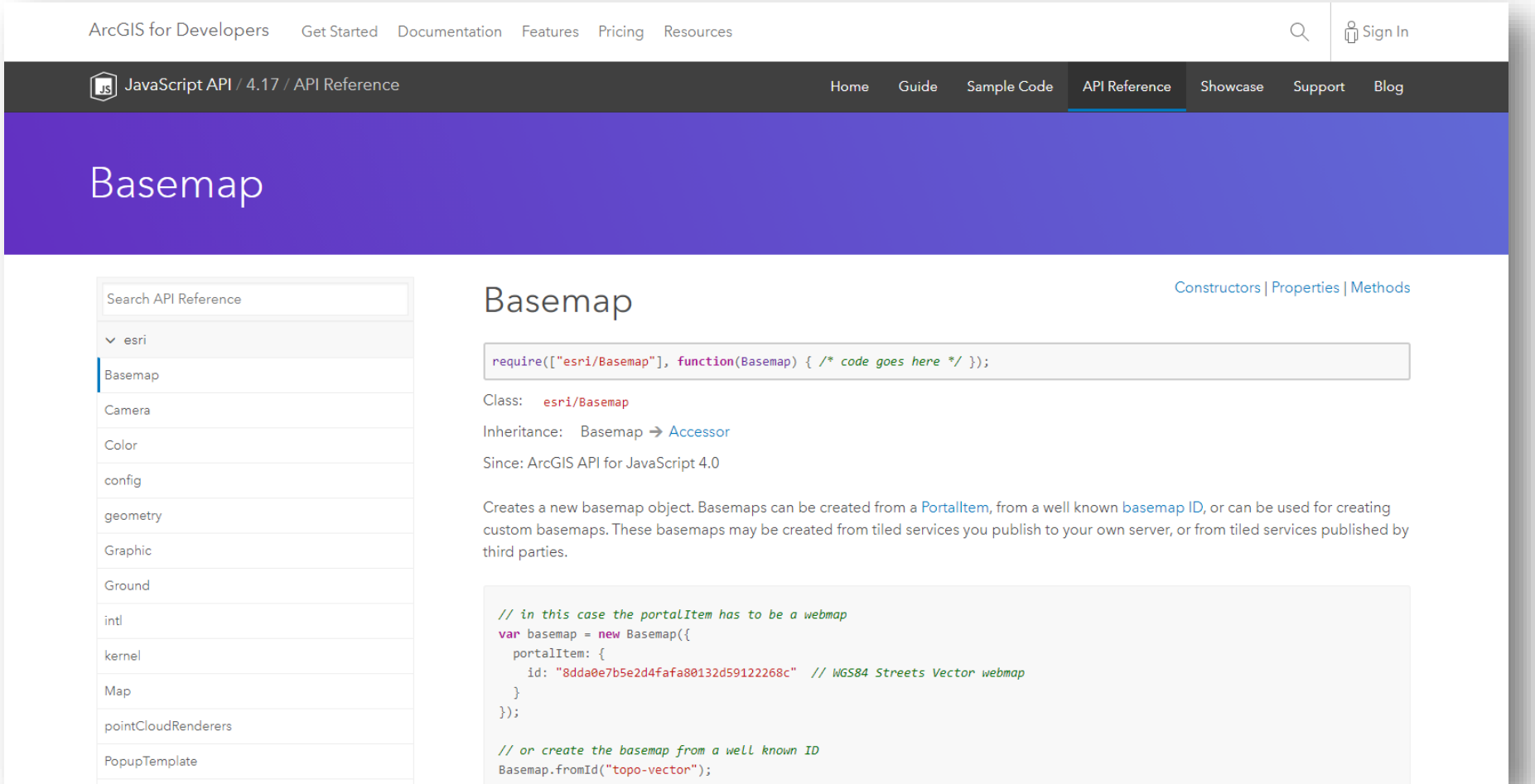
```
(Feature | CSV | GeoJSON | Scene | Stream)LayerView
  // only display features that satisfy the requirements in the layer
  .filter = {
    where: "age > 25";
  }
```

Filters can be applied based on attributes, time, and/or geometry. Search the sample code using the FeatureFilter tag to explore all current samples that demonstrate how you can use the featureFilter to display subset of features that meet requirements. This tutorial walks through querying FeatureLayer and FeatureLayerView.

This section discusses programming patterns and best practices for writing applications with the ArcGIS API for JavaScript.



The ArcGIS API for JavaScript is a lightweight way to embed maps and tasks in web applications. You can get these maps from ArcGIS Online, your own ArcGIS Server or others' servers.



Loading Classes

After getting the ArcGIS API for JavaScript, use require() to asynchronously load ArcGIS API for JavaScript classes into an application.

The method requires two parameters:

- An array of ordered strings as the full namespaces for each imported API class
- Each API class is loaded as the positional arguments in a callback function

For example, to load the Map and MapView class, first go to the documentation and find the full namespace for each class. In this case, Map has the namespace "esri/Map" and the namespace for MapView is "esri/views/MapView". Then pass these strings as an array to require(), and use the local variable names Map and MapView as the positional arguments for the callback function:

```
require(['esri/Map', 'esri/views/MapView'], function (Map, MapView) {
  // The application logic using `Map` and `MapView` goes here
});
```

Not every module needs to be loaded with require(), as many classes can be initialized from within a constructor using autocasting.

Constructors

All classes in the ArcGIS API for JavaScript have a single constructor, and all properties can be set by passing parameters to the constructor.

For example, here is an example of calling the constructor for the Map and MapView classes.

```
const map = new Map({
  basemap: 'topo-vector'
});

const view = new MapView({
  map: map,
  container: 'map-div',
  center: [-122, 38],
  scale: 5
});
```

Properties

The ArcGIS API for JavaScript supports a simple, consistent way of getting, setting, and watching all properties of a class.

Many API classes are subclasses of the Accessor class, which defines the following methods.

Method Name	Return Type	Description
get(propertyName)	Varies	Gets the value of the property with the name propertyName
set(propertyFields)	N/A	For each key/value pair in propertyFields, this method sets the value of the property with the name key to value
watch(propertyName, callback)	WatchHandle	Calls the callback function callback whenever the value of the property with the name propertyName changes

Getters

The get method returns the value of a named property.

This method is a convenience method because, without using *get()*, to return the value nested properties, e.g., to return the title of the property basemap of a Map object, requires an if statement to check whether basemap is either *undefined* or *null*.

```
const basemapTitle = null;
if (map.basemap) {                    // Make sure `map.basemap` exists
  basemapTitle = map.basemap.title;
}
```

The get method removes the need for the *if* statement, and returns the value of *map.basemap.title* if *map.basemap* exists, and *null* otherwise.

```
var basemapTitle = map.get('basemap.title');
```

Setters

The values of properties may be set directly.

```
view.center = [ -100, 40 ];
view.zoom = 6;
map.basemap = 'oceans';
```

When several property values need to be changed, set() can be passed a JavaScript Object with the property names and the new values.

```
const newViewProperties = {
  center: [-100, 40],
  zoom: 6
};

view.set(newViewProperties);
```

Watching for Property Changes

Watching property changes are handled with *watch()*, which takes two parameters:

- A property name as a String, and
- A callback function that is called whenever the property value changes

A WatchHandle instance is returned by *watch()*.

The following code sample watches the *basemap.title* property of a *Map* object, and calls the *titleChangeCallback* function whenever the basemap title value changes.

```
const map = new Map({
  basemap: 'streets-vector'
});

function titleChangeCallback(newValue, oldValue, property, object) {
  console.log('New value: ', newValue,
    '<br>Old value: ', oldValue,
    '<br>Watched property: ', property,
    '<br>Watched object: ', object);
};

const handle = map.watch('basemap.title', titleChangeCallback);
```

For example, if the *basemap* property changes:

```
map.basemap = 'topo-vector';
```

The remove method can be called on a WatchHandle object to stop watching for changes.

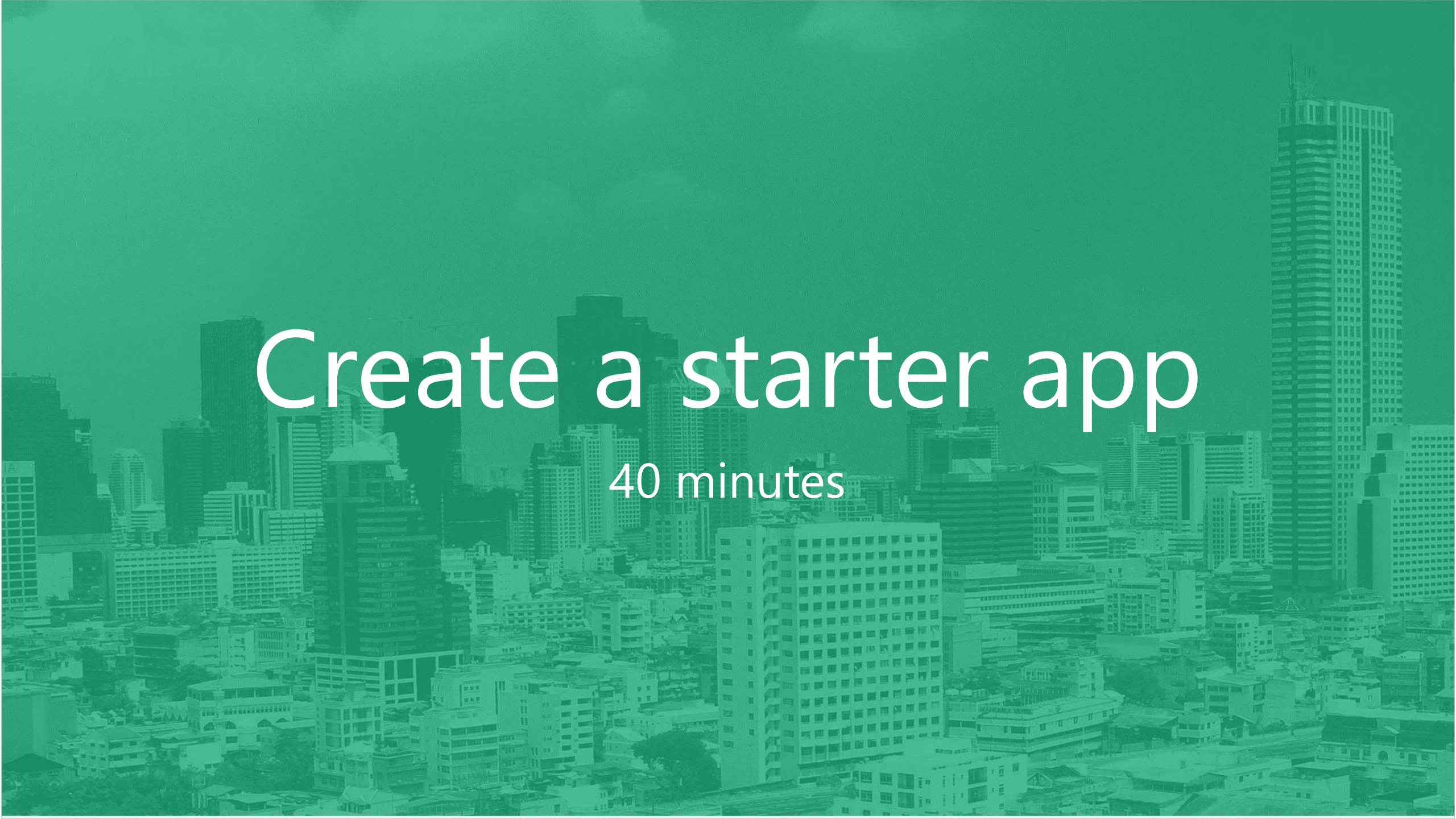
```
handle.remove();
```

Not all properties can be watched, including collections. Register an event handler to be notified of changes to a collection.

Note: Because both the *FeatureLayer.source* and *GraphicsLayer.graphics* properties are collections, use *on()* instead of *watch()* to be notified of changes to the values of these properties.

Create a feature collection with the following table to show it as a FeatureLayer on the map. Use different symbol between **Yes** and **No** values in the Favorite column.

Id	Attraction	Favorite	Latitude	Longitude
1	Wat Pho	No	13.746577	100.4911166
2	The Grand Palace	No	13.7499533	100.4891229
3	Chatuchak Market	Yes	13.7993922	100.5465864
4	Siam Paragon	No	13.7460939	100.5319794
5	Lumpinee Boxing Stadium	Yes	13.8669948	100.6066211



Create a starter app

40 minutes

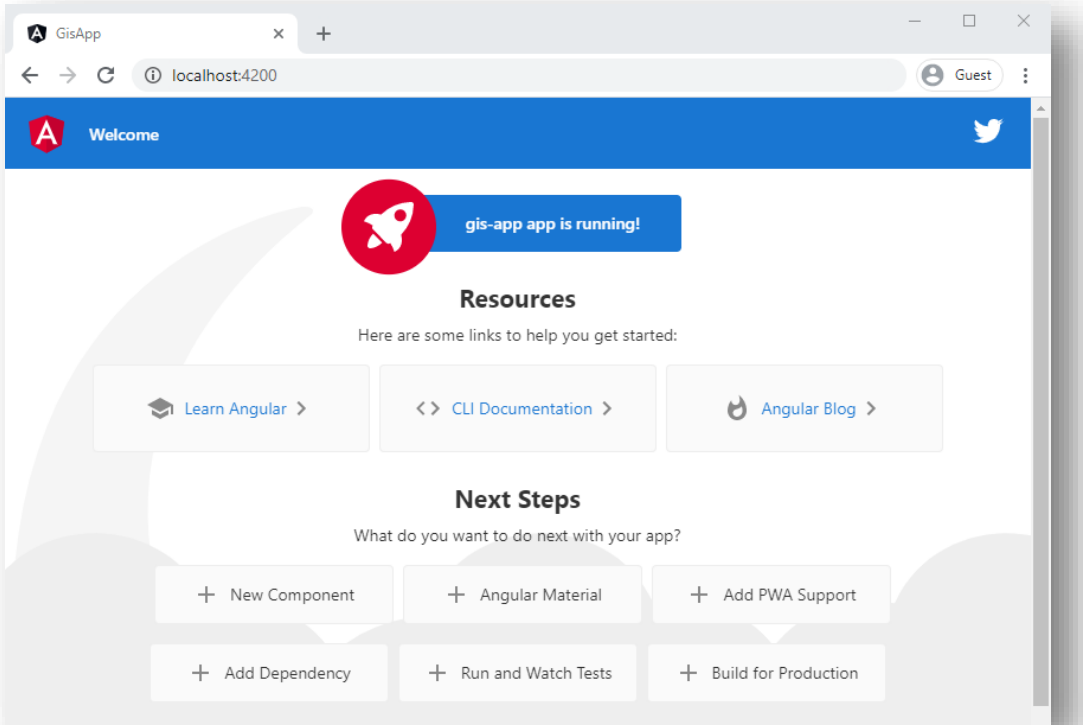
1. Run the CLI command ng new and provide the name gis-app.

```
ng new gis-app
? Would you like to add Angular routing? (y/N) y
? Which stylesheet format would you like to use? SCSS
```

2. Open the workspace folder with Visual Studio Code.
3. In VS Code, go to *Terminal tab* then click *New Terminal* (ctrl + `).
4. Run the following command:

```
ng serve --open
```

If your installation and setup was successful, you should see a page like the following.



1. Install esri-loader package.

```
npm install esri-loader --save
```

The **esri-loader** is a tiny library to help you use the ArcGIS API for JavaScript in applications built with popular JavaScript frameworks and bundlers.

2. At *src/environments* folder, add *arcgis* environment configure both *Development* and *Production* file with following JSON.

```
export const environment = {
  ...

  arcgis: {
    js: 'https://js.arcgis.com/4.15/',
    css: 'https://js.arcgis.com/4.15/esri/themes/light/main.css'
  }
};
```

3. At *src/app* folder, open *app.module.ts* then create *initialApp* function.

```
...
import { setDefaultOptions } from 'esri-loader';

import { environment } from './../environments/environment’;
...

export function initialApp() {
  return () => setDefaultOptions({
    url: environment.arcgis.js,
    css: environment.arcgis.css
  });
}

@NgModule({ ... })
export class AppModule { }
```

4. Import the APP_INITIALIZER DI token from @angular/core and register provider.

```
...
import { NgModule, APP_INITIALIZER } from '@angular/core';

...

export function initialApp() { ... }

@NgModule({
  declarations: [ ... ],
  imports: [ ... ],
  providers: [
    {
      provide: APP_INITIALIZER,
      useFactory:  initialApp,
      multi: true
    }
  ],
  bootstrap: [AppComponent]
})
```

Register provider

1. Create a new terminal. Generate esri-map component with the following command:

```
ng generate component esri-map
```

2. Open *esri-map.component.html* and update file with following code.

```
<div #mapViewRef class="map-view"></div>
```

3. Open *esri-map.component.ts* file. Reference *mapViewRef* property to template(*#mapViewRef*).

```
import { Component, OnInit, ViewChild, ElementRef } from '@angular/core';

@Component({ ... })
export class EsriMapComponent implements OnInit {

  @ViewChild('mapViewRef', { static: true })
  private mapViewRef: ElementRef<HTMLDivElement>;

  ...
}
```

4. Generate gis service with the following command:

```
ng generate service gis
```

5. Open gis.service.ts file then create *map* and *mapView* properties with any data type.

```
export class GisService {

  public map: any;
  public mapView: any;

  ...
```

6. At esri-map.component.ts file, inject GisService class at the constructor function.

```
export class EsriMapComponent implements OnInit, OnDestroy {

  ...

  constructor(private gisService: GisService) { }

}
```

7. Create initializeMap function with the following code and call its at ngOnInit function.

```
ngOnInit() {
  this.initializeMap();
}

async initializeMap() {
  // Load the modules for the ArcGIS API for JavaScript
  const [Map, MapView] = await loadModules(['esri/Map', 'esri/views/MapView']);

  // Configure the Map
  const mapProperties = {
    basemap: 'topo-vector'
  };
  this.gisService.map = new Map(mapProperties);

  // Initialize the MapView
  const mapViewProperties = {
    container: this.mapViewRef.nativeElement,
    center: [100.5428505, 13.702994],
    zoom: 17,
    map: this.gisService.map
  };
  this.gisService.mapView = new MapView(mapViewProperties);
  await this.gisService.mapView.when(); // wait for map to load

  return this.gisService.mapView;
}
```

8. Implement OnDetroy for release memory resource when the component was destroyed.

```
export class EsriMapComponent implements OnInit, OnDestroy {

  ...

  ngOnDestroy() {
    if (this.gisService.mapView) {
      // Destroy the map view.
      this.gisService.mapView.container = null;
    }
}
```

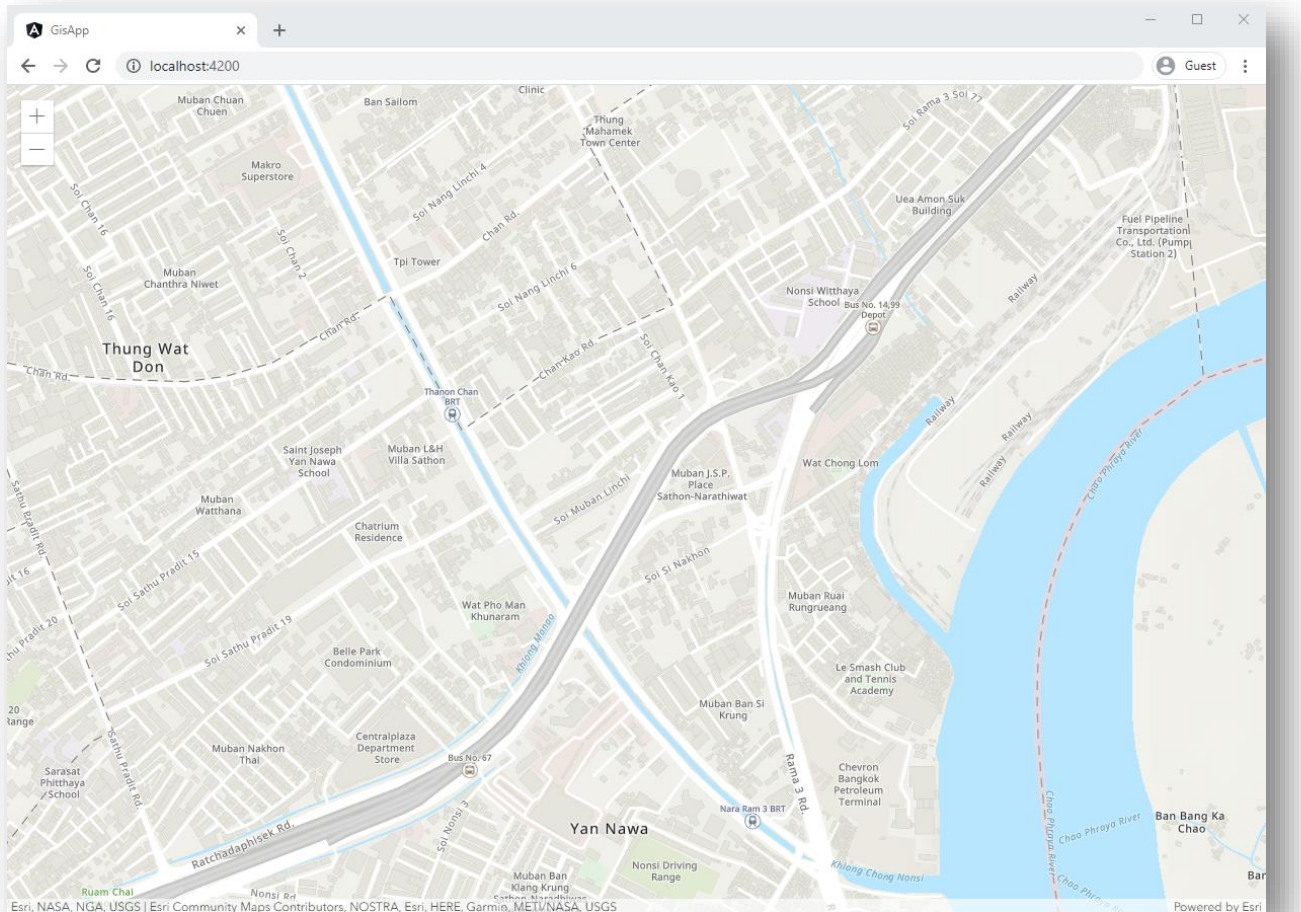
9. Open *esri-map.component.scss* file, Set stylesheet to pinned component to screen and set size of map-view relate with width-height component via following code:

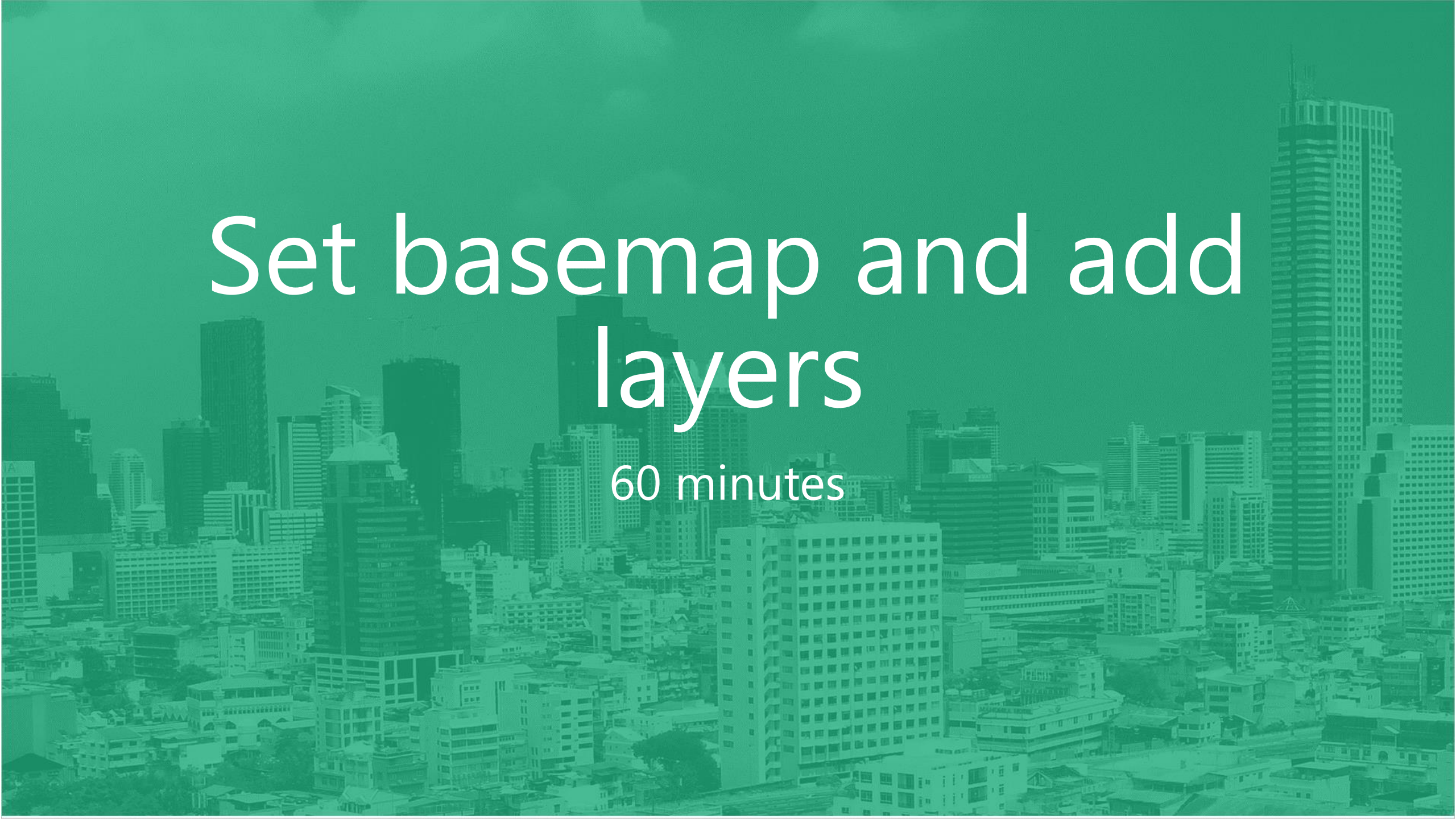
```
:host {
    position: absolute;
    top: 0;
    right: 0;
    bottom: 0;
    left: 0;

    .map-view {
        width: 100%;
        height: 100%;
    }
}
```

10. At src/app folder, open *app.component.html* then remove all text in this file and insert following code:

```
<app-esri-map></app-esri-map>
```

Save all files change and preview.
If you configure and coding was successful, the map should look like the following.





Set basemap and add layers

60 minutes

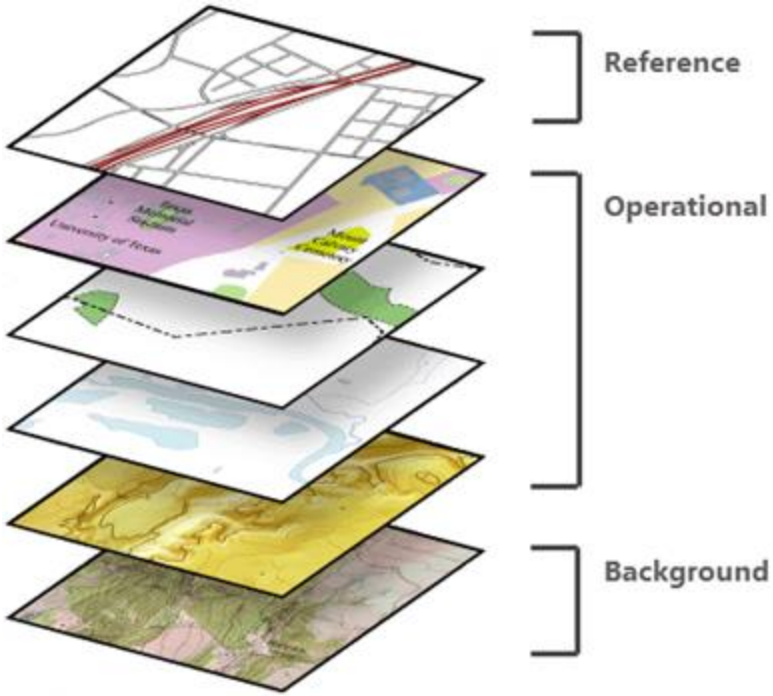
Set basemap and add layers – Basemap

Basemaps serve as a reference map on which you overlay data from layers and visualize geographic information.

An individual basemap can be made of multiple feature, raster, or web layers.

They are the foundation for your maps and provide context for your work.

Set basemap and add layers – Basemap



Authoring a custom basemap allows you to define reference and background layers.

Reference layers draw on top of operational layers, while background layers draw below operational layers.

1. Open gis.service.ts file then create a createTopoBasemap function to generate Topo Basemap object.

```
export class GisService {
  ...

  public async createTopoBasemap() {
    const [Basemap, TileLayer] = await loadModules(['esri/Basemap', 'esri/layers/TileLayer']);

    const topoBasemap = new Basemap({
      baseLayers: [
        new TileLayer({
          url: 'https://services.arcgisonline.com/ArcGIS/rest/services/World_Topo_Map/MapServer',
          title: 'World Topo'
        })
      ],
      title: 'World Topo Basemap',
      id: 'topo_basemap'
    });

    return topoBasemap;
  }

}
```

2. Create a Topo Basemap object and set to basemap property in the mapProperties.

```
async initializeMap() {
  // Load the modules for the ArcGIS API for JavaScript
  const [Map, MapView] = await loadModules(['esri/Map', 'esri/views/MapView']);

  const topoBasemap = await this.gisService.createTopoBasemap();

  // Configure the Map
  const mapProperties = {
    basemap: topoBasemap
  };
  ...

}
```

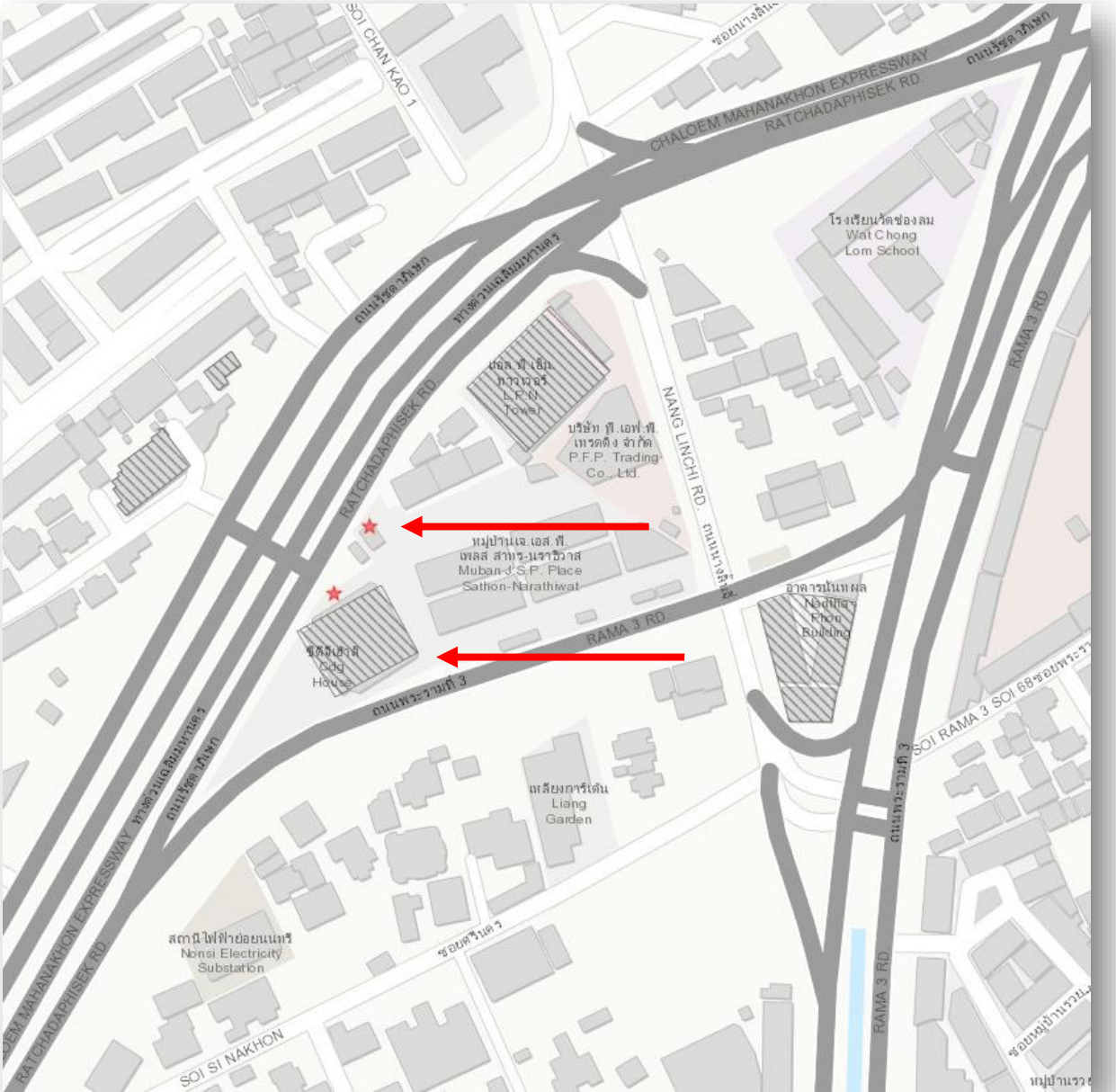
1. At gis.service.ts file, create *buildingBlockLayer* and *shopLayer* properties.
2. Create a createFeatuerLayer function with the following code.

```
public async createFeatuerLayer(options: any) {
  const [FeatureLayer] = await loadModules(['esri/layers/FeatureLayer']);

  return new FeatureLayer(options);
}
```

3. Open esri-map.component.ts file. At before return value in *initializeMap* function, create feature layers and add to map object.

```
  ...

  // Create feature layers and add to map.
  this.gisService.buildingBlockLayer = await this.gisService.createFeatuerLayer({
    url: 'https://appserver2.cdg.co.th/arcgis/rest/services/AtlasX/City/MapServer/1'
  });
  this.gisService.shopLayer = await this.gisService.createFeatuerLayer({
    url: 'https://appserver2.cdg.co.th/arcgis/rest/services/AtlasX/City/MapServer/0'
  });
  this.gisService.map.addMany([
    this.gisService.buildingBlockLayer,
    this.gisService.shopLayer
  ]);

  return this.gisService.mapView;
```

Set basemap and add layers – Add layer to a map



Your map will renderer look like the following.

1. Set NOSTRA Vector Tile to basemap layer with the following URL:

   https://mapportal.nostramap.com/server/rest/services/Hosted/NOSTRA_TEST_20191120_1/VectorTileServer

   Ref:
   https://developers.arcgis.com/javascript/latest/api-reference/esri-layers-VectorTileLayer.html

2. Set layer definition expression to filter out features where building type is "**อื่น ๆ**" in the building blocks.

   Ref:
   https://developers.arcgis.com/javascript/latest/api-reference/esri-layers-FeatureLayer.html#definitionExpression

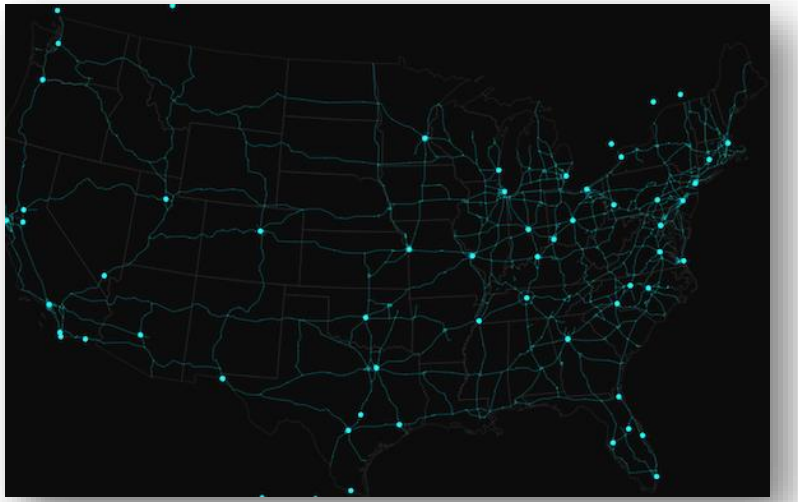
   CodedValueDomain **BUILDING_TYPE**:
   https://appserver2.cdg.co.th/arcgis/rest/services/AtlasX/City/MapServer/1?f=pjson

Style feature layers

40 minutes

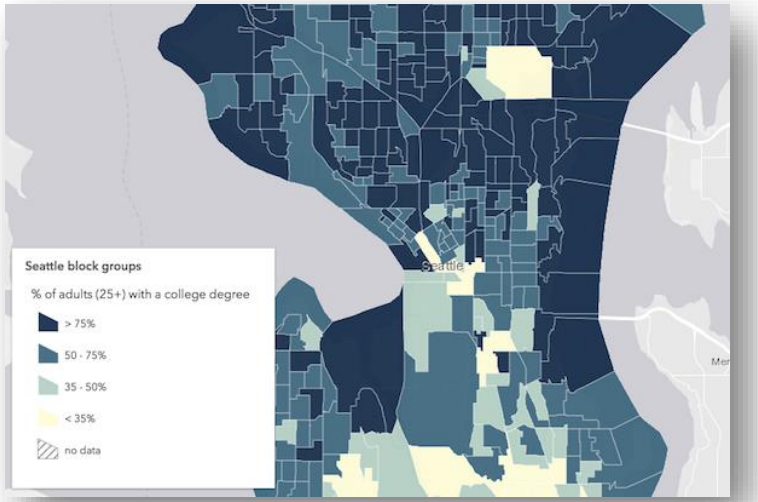
Applications can display feature layer data with different styles to enhance the visualization. The first step is to select the correct type of Renderer.

A SimpleRenderer applies the same symbol to all features, a UniqueValueRenderer applies a different symbol to each unique attribute value, and a ClassBreaksRenderer applies a symbol to a range of numeric values.



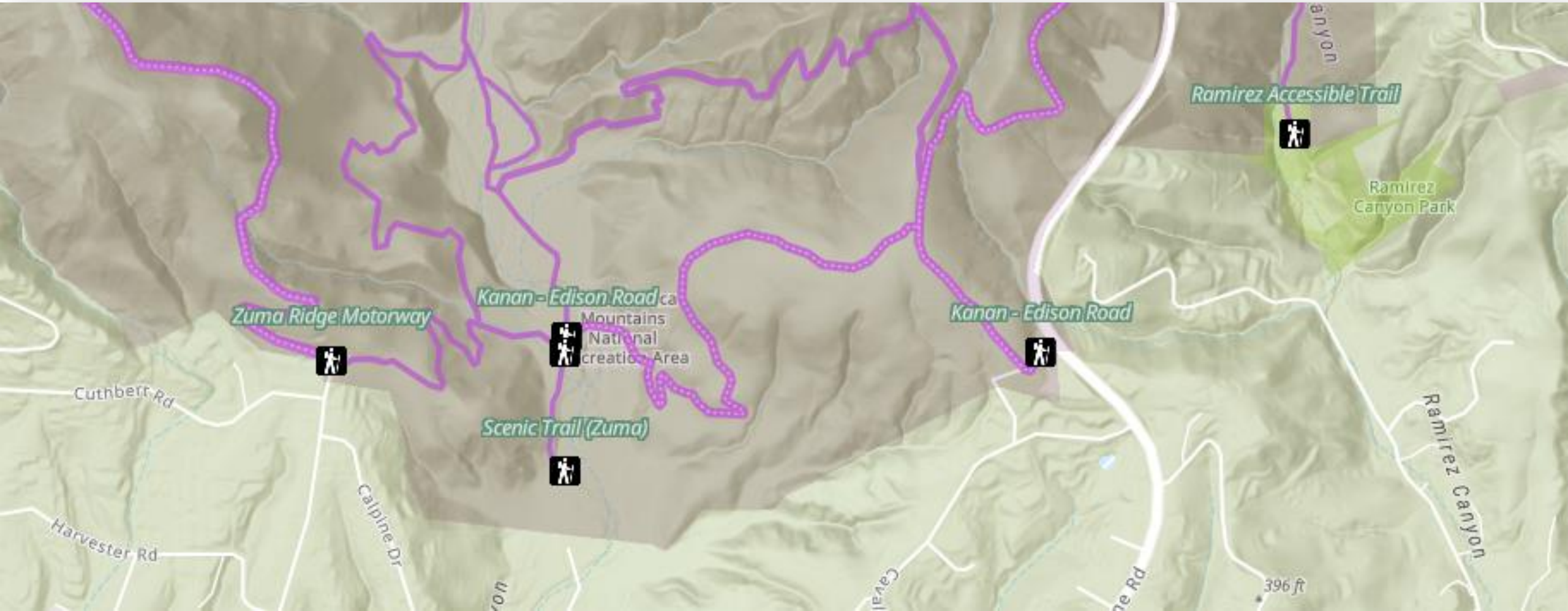
SimpleRenderer

UniqueValueRenderer



ClassBreaksRenderer

Labels can also be displayed to show attribute information for each feature, and visual variables and expressions can also be used to create more complex data-driven visualizations.



1. At the GisService, create a *createShopRenderer()* function and define it as a simple renderer and set the symbol properties to draw a shop image that is a picture-marker, 24px in size for each point. Use the url below for the image.

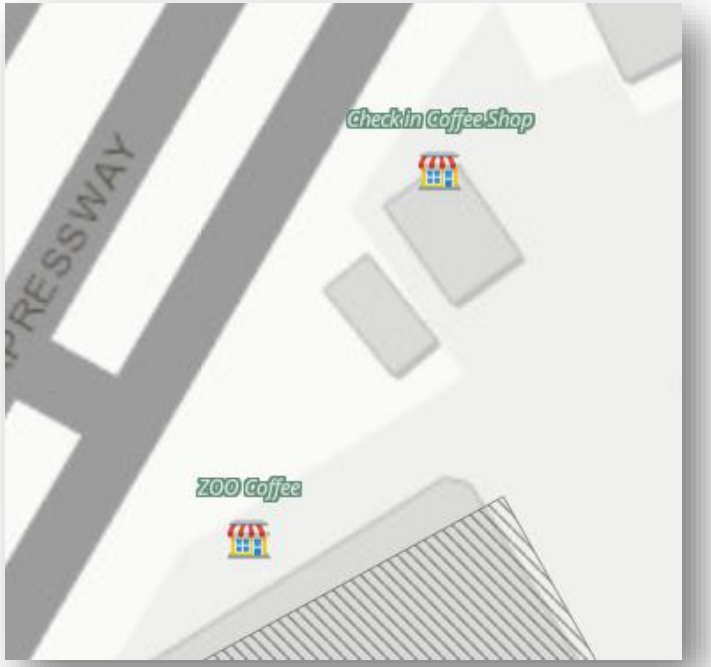
```
public createShopRenderer() {
  return {
    type: 'simple', // autocasts as new SimpleRenderer()
    symbol: {
      type: 'picture-marker', // autocasts as new PictureMarkerSymbol()
      url: 'https://atlasx.cdg.co.th/share/Courses/ArcGIS%20API%20JS/101/shop-icon.png',
      width: '24px',
      height: '24px'
    }
  };
}
```

2. To show shop name labels, create a *createShopLabels()* function to define the labelingInfo. Set the symbol to draw the labels white #FFFFFF with a green #5E8D74 halo, place them centered above the feature, and use a simple expression to reference the SHOP_NAME field (the data to render).

```
public createShopLabels() {
  return {
    symbol: {
      type: 'text',
      color: '#FFFFFF',
      haloColor: '#5E8D74',
      haloSize: '2px',
      font: {
        size: '10px',
        family: 'Noto Sans',
        style: 'italic',
        weight: 'normal'
      }
    },
    labelPlacement: 'above-center',
    labelExpressionInfo: {
      expression: '$feature.SHOP_NAME'
    }
  };
}
```

3. At EsriMapComponent, update a *this.gisService.shopLayer* FeatureLayer by set the renderer and labelingInfo property to the objects created above and add it to the map.

```
this.gisService.shopLayer = await this.gisService.createFeatuerLayer({
  url: 'https://appserver2.cdg.co.th/arcgis/rest/services/AtlasX/City/MapServer/0',
  renderer: this.gisService.createShopRenderer(),
  labelingInfo: [this.gisService.createShopLabels()]
});
```



Open web browser to view the shop images and labels.

Set style to representing building block is rendered with a ClassBreaksRenderer. Features where fewer than 12 meters of the height building are rendered with a deep green color. Features where between 12 and 60 meter of the height building are rendered with a pale green symbol. The other features are similarly rendered based on the value of the attribute of interest.

Ref:

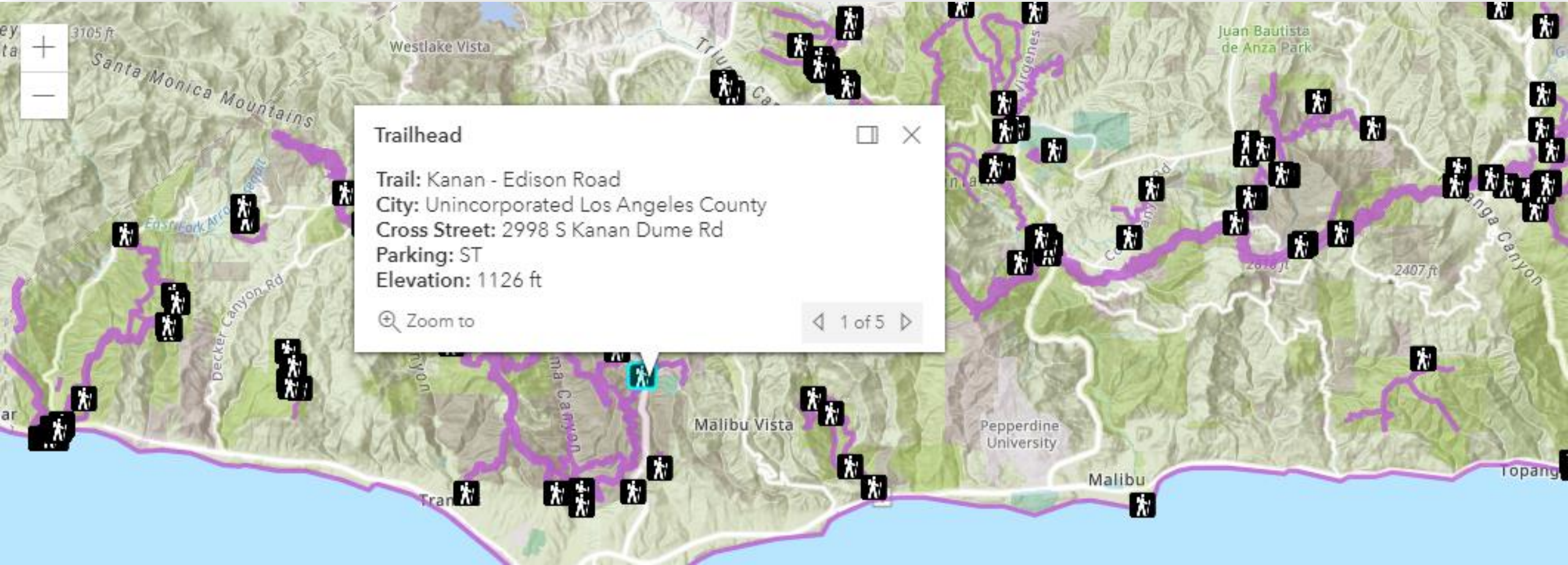
https://developers.arcgis.com/javascript/latest/api-reference/esri-renderers-ClassBreaksRenderer.html

Configure pop-ups

30 minutes

Configure pop-ups

Applications can show pop-ups with attribute information for feature layers and graphics when you click on the map. Pop-ups can be formatted and styled by creating a *PopupTemplate* object. A PopupTemplate allows you to define the title, content and how the media is displayed. You can pass in HTML, a function or a list of fields to display a table view.



1. At the GisService, create a *createBuildingBlockPopupTemplate()* function that sets the title to Shop and the content to a custom HTML string.

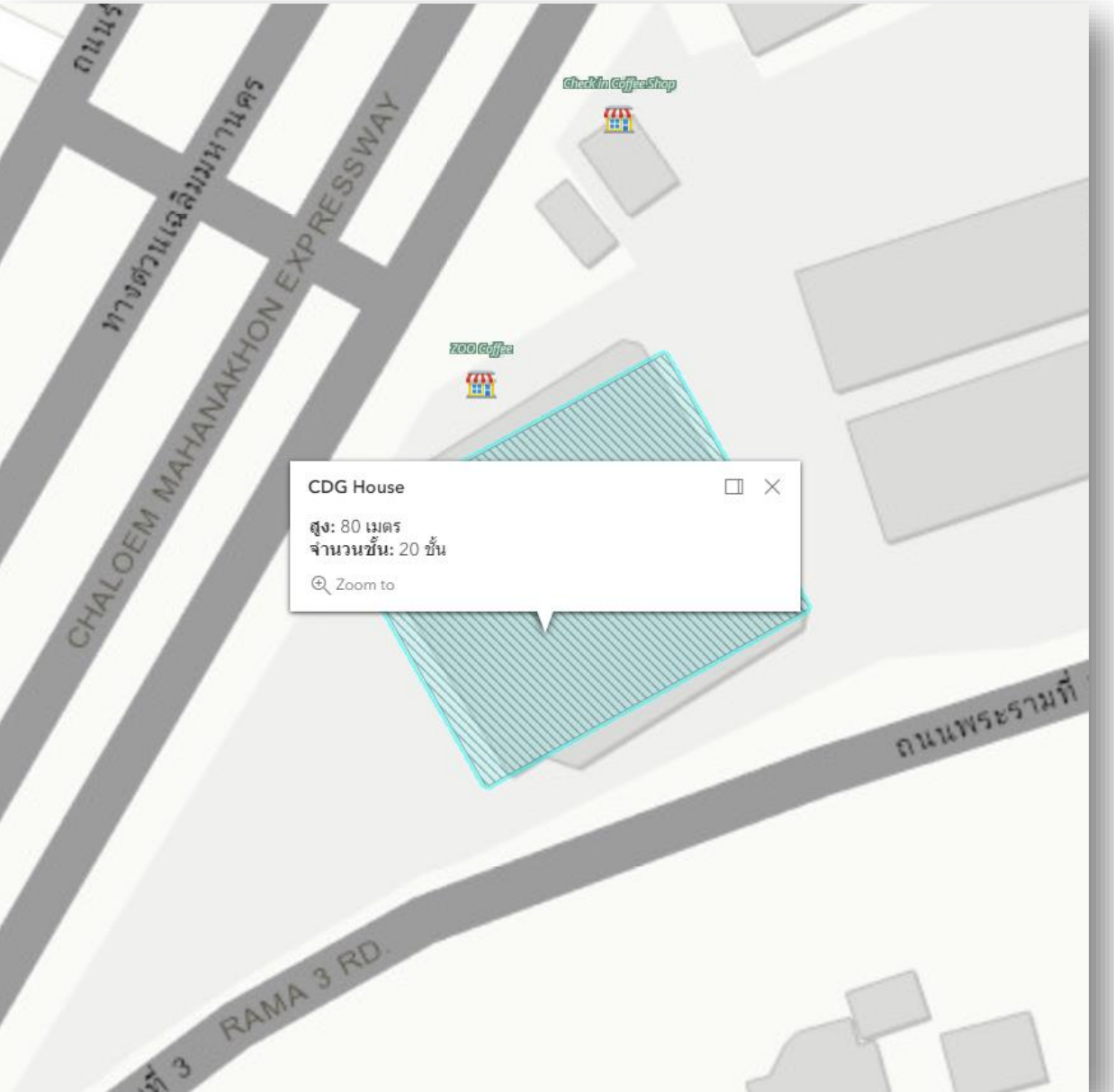
```
public createBuildingBlockPopupTemplate() {
  return {
    title: '{NAME}',
    content: `
      <div><b>สูง:</b> {HEIGHT} เมตร</div>
      <div><b>จำนวนชั้น:</b> {FLOOR} ชั้น</div>
    `
  }
}
```

2. Apply the popupTemplate to the Building block layer.

```
this.gisService.buildingBlockLayer = await this.gisService.createFeatuerLayer({
  url: 'https://appserver2.cdg.co.th/arcgis/rest/services/AtlasX/City/MapServer/1',
  outFields: ['NAME', 'HEIGHT', 'FLOOR'], // use ['*'] to get all field from features.
  popupTemplate: this.gisService.createBuildingBlockPopupTemplate()
});
```

Note: To fetch the values from all fields in the layer, use ['*'] for ***outFields*** property.

Configure pop-ups

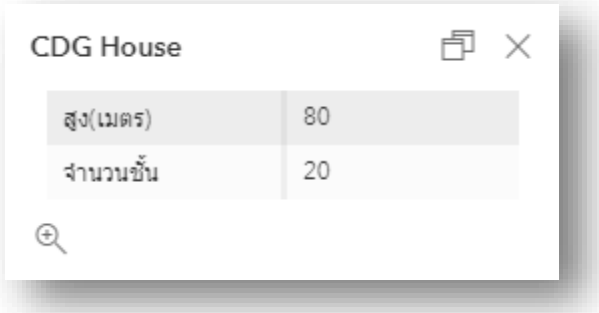


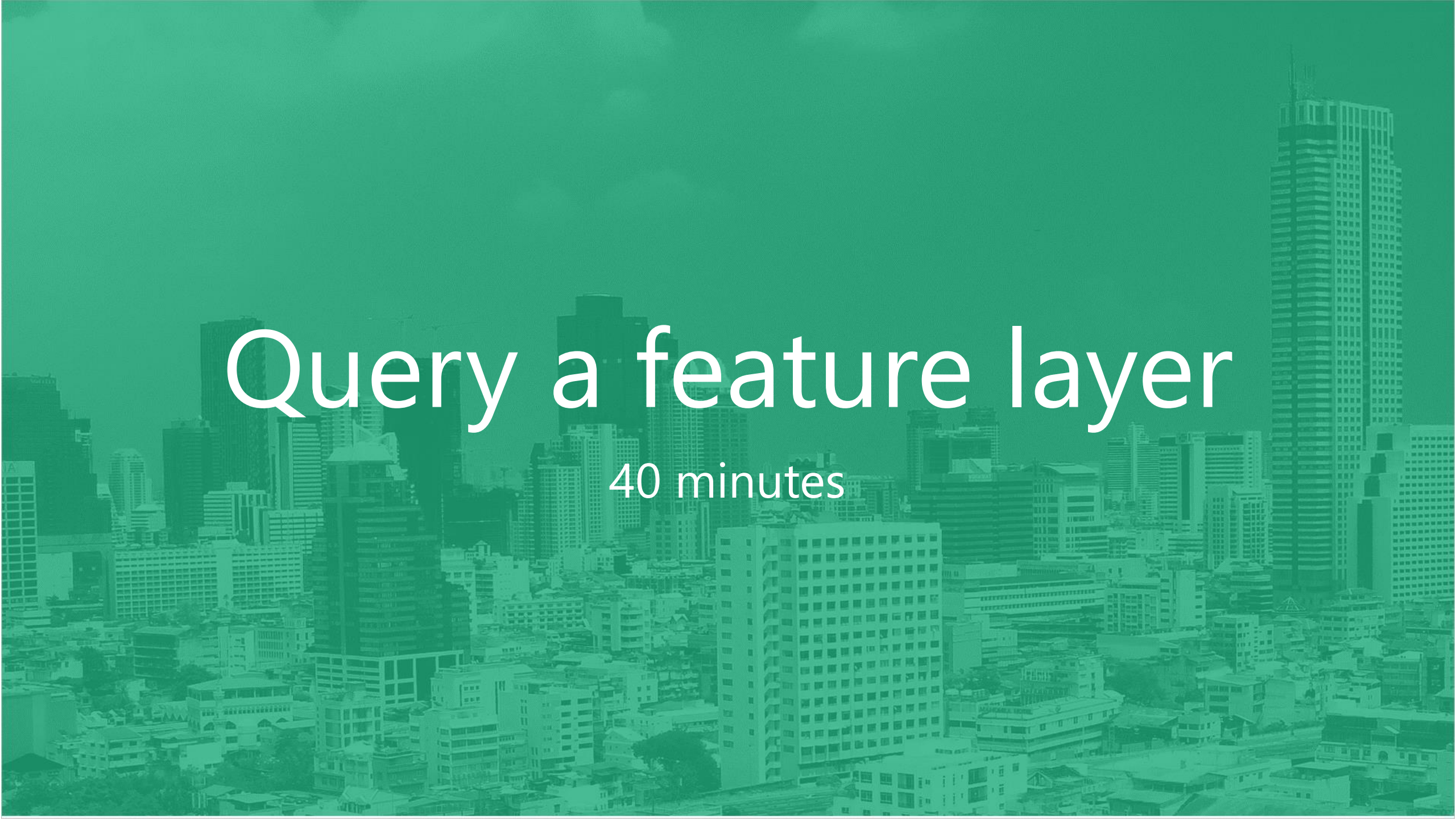
Click on Building blocks to view the pop-up.

Replace your custom HTML string template in content property to *fieldInfos*.

Ref:

https://developers.arcgis.com/javascript/latest/api-reference/esri-PopupTemplate.html





Query a feature layer

40 minutes

Query a feature layer

Applications can perform server-side or client-side SQL, spatial, and statistic queries to access and display data from feature layers. The source data for a feature layer can be hosted on ArcGIS Online or ArcGIS Enterprise or it can be created from an array on the client.

Both client-side and server-side queries can contain a SQL expression and/or a spatial relationship operator. The main difference between client-side and server-side queries is that client-side querying is only possible after the feature layer is added to a map and the attributes are present.

Query a feature layer – Server-side Query

To request a subset of data from the server without adding the feature layer to a map, use the *queryFeatures* method on a *FeatureLayer* object.

Use *createQuery* function creates query parameter object that can be used to fetch features that satisfy the layer's configurations such as *definitionExpression*, *geometry* and *spatialRelationship*.

```
const query = this.gisService.shopLayer.createQuery();
query.where = `SHOP_NAME LIKE '%Coffee%'`;
query.returnGeometry = true;
query.outFields = ['SHOP_NAME'];

const results = await this.gisService.shopLayer.queryFeatures(query);
console.log('results', results);
```

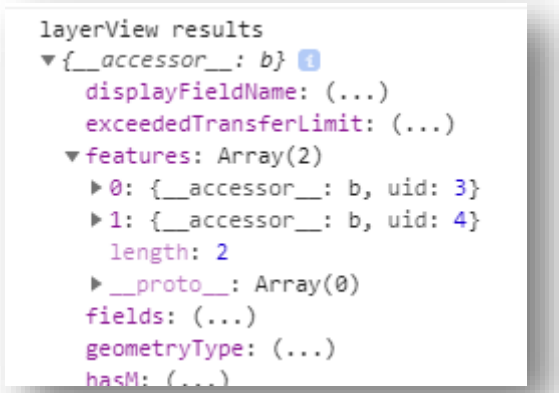
Output:

Query a feature layer – Client-side Query

To access a subset of data on the client, you have to add the feature layer to a map first, and then use the *queryFeatures* method on a *FeatureLayerView* object. Since the data is on the client, client-side queries execute very quickly.

```
const shopLayerView = await this.gisService.mapView.whenLayerView(this.gisService.shopLayer);
shopLayerView.watch('updating', async (val: any) => {
  if (!val) {  // wait for the layer view to finish updating
    const shopResults = await shopLayerView.queryFeatures(query);
    console.log('layerView results', shopResults);
  }
});
```

Output:



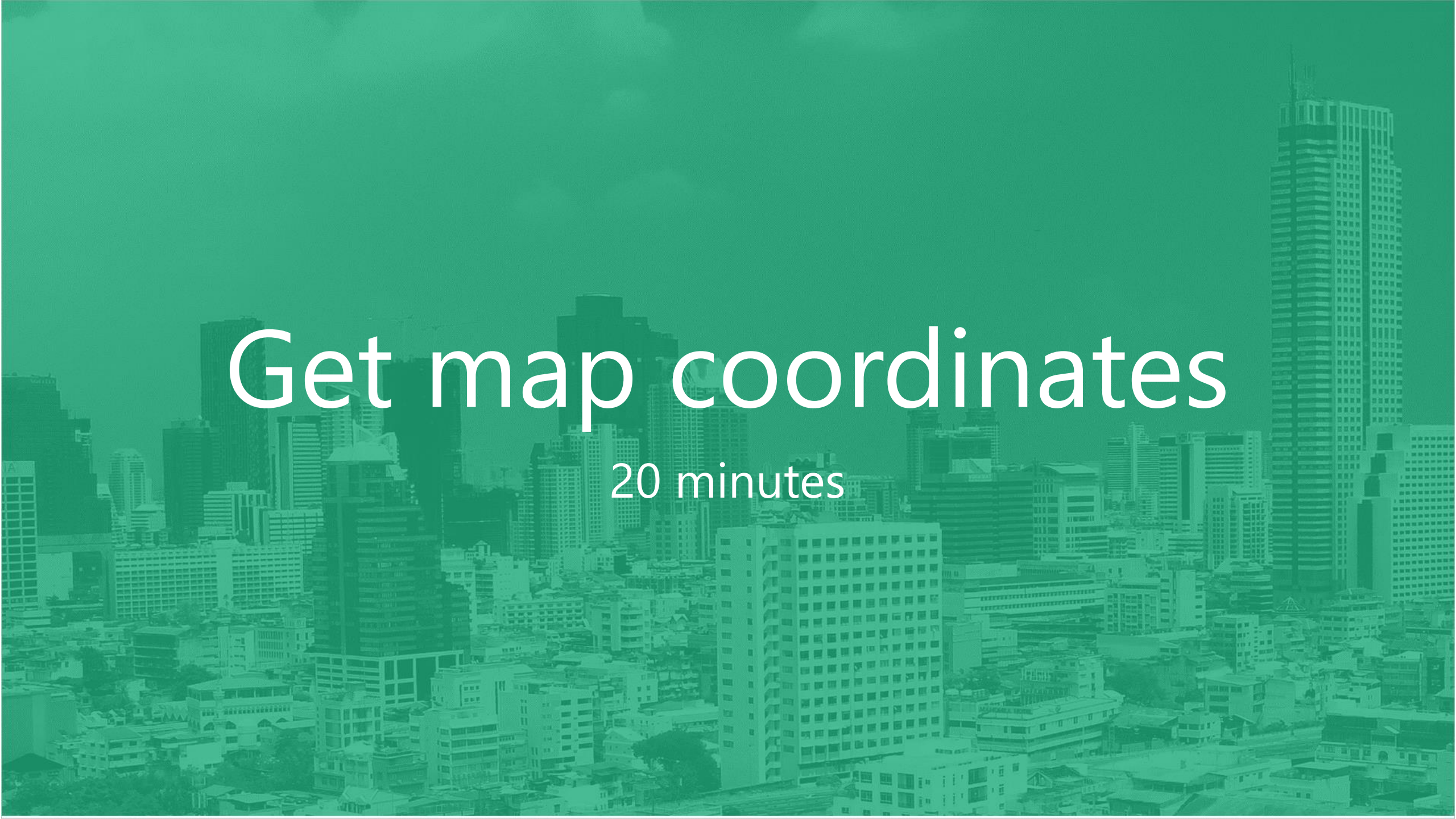
Search for a Shop layer within 50 meters radius by clicking on the map and print out the result to console.

```
this.gisService.mapView.on('click', (event: any) => {
  // event is the event handle returned after the event fires.
  console.log(event.mapPoint);
});
```

Ref:

https://developers.arcgis.com/javascript/latest/api-reference/esri-geometry-geometryEngine.html#buffer

https://developers.arcgis.com/javascript/latest/api-reference/esri-tasks-support-Query.html#geometry



Get map coordinates

20 minutes

The *View* provides a way to interact with the map and to retrieve information about the map location. Using properties and event handlers on the View you can find the current spatial reference information, latitude and longitude, scale, and zoom level for the map or any screen point location.

Once you have this information you can display it on the map, use it to find other locations on the earth, or use it to set the initial extent of your application when it starts.

Get map coordinates

At EsriMapComponent, create a div element and add it to the bottom right corner of the view. Assign it classes and apply some simple styling so the UI looks and behaves like other widgets. All components should use the esri-widget and esri-component css classes.

```
private addCoorsToView() {
  this.gisService.coordsWidget = document.createElement('div');
  this.gisService.coordsWidget.className = 'esri-widget esri-component';
  this.gisService.coordsWidget.style.padding = '7px 15px 5px';

  this.gisService.mapView.ui.add(this.gisService.coordsWidget, 'bottom-right');
}
```

Get map coordinates

At GisService, Create a new function to update the innerHTML of the component and display the current latitude and longitude, scale, and zoom level for the map. The function will take any given point and round the coordinates to a fixed set of decimal places.

```
public showCoordinates(point: any) {
  const coords = `Lat/Lon  ${point.latitude.toFixed(4)} ${point.longitude.toFixed(4)}` +
    ` | Scale 1:${Math.round(this.mapView.scale * 1) / 1}` +
    ` | Zoom ${this.mapView.zoom}`;
  this.coordsWidget.innerHTML = coords;
}
```

Get map coordinates

At the end of the code in addCoorsToView function of EsriMapComponent, Add event and watch handlers to call the *showCoordinates* function when the view is *stationary* and when the *pointer moves*. When the view is stationary, it will show the center location. When the pointer moves, it will display the current pointer location. Use toMap to convert screen coordinates to map coordinates.

```
    this.gisService.mapView.watch('stationary', () => {
      this.gisService.showCoordinates(this.gisService.mapView.center);
    });

    this.gisService.mapView.on('pointer-move', (evt: any) => {
      this.gisService.showCoordinates(this.gisService.mapView.toMap({ x: evt.x, y: evt.y }));
    });
```

Get map coordinates

Run the app and move your cursor around the map. Watch the map coordinates change.

Zoom the map in and out and watch the scale and zoom level change.

Display point, line, and polygon graphics

20 minutes

If you would like to display points, lines, polygons, and text in a map, you can use *graphics* and a *graphic layer*. Adding graphics to a graphics layer is a fast and easy way to display small amounts of temporary geographic data on a map.

If you are working with larger amounts of data (>1000 records), you should consider importing your data as feature layer and adding it to a map.

In the *loadModules* statement, add the *Graphic* and *GraphicsLayer* modules.

```
const [
  Graphic,
  GraphicsLayer
] = await loadModules([
  'esri/Graphic',
  'esri/layers/GraphicsLayer'
]);
```

Create and add a GraphicsLayer to the map.

```
this.gisService.graphicsLayer = new GraphicsLayer();
this.gisService.map.add(this.gisService.graphicsLayer);
```

Define a *point* and *simpleMarkerSymbol* object and use them to create a new point Graphic. The graphic will autocast the objects and create class instances when the graphic is created. After you create the graphic, add it to the graphicsLayer.

```
const point = {
  type: 'point',
  longitude: -118.80657463861,
  latitude: 34.0005930608889
};

const simpleMarkerSymbol = {
  type: 'simple-marker',
  color: [226, 119, 40],  // orange
  outline: {
    color: [255, 255, 255], // white
    width: 1
  }
};

const pointGraphic = new Graphic({
  geometry: point,
  symbol: simpleMarkerSymbol
});

this.gisService.graphicsLayer.add(pointGraphic);
```

Define a *line* and *simpleLineSymbol* object and use them to create a new line Graphic. The graphic will autocast the objects and create class instances when the graphic is created. After you create the graphic, add it to the graphicsLayer.

```
const simpleLineSymbol = {
  type: 'simple-line',
  color: [226, 119, 40], // orange
  width: 2
};

const polyline = {
  type: 'polyline',
  paths: [
    [-118.821527826096, 34.0139576938577],
    [-118.814893761649, 34.0080602407843],
    [-118.808878330345, 34.0016642996246]
  ]
};

const polylineGraphic = new Graphic({
  geometry: polyline,
  symbol: simpleLineSymbol
});

this.gisService.graphicsLayer.add(polylineGraphic);
```

Define a *polygon* and *simpleFillSymbol* object and use them to create a new polygon Graphic. The graphic will autocast the objects and create class instances when the graphic is created. After you create the graphic, add it to the graphicsLayer.

```
const polygon = {
  type: 'polygon',
  rings: [
    [-118.818984489994, 34.0137559967283],
    [-118.806796597377, 34.0215816298725],
    [-118.791432890735, 34.0163883241613],
    [-118.79596686535, 34.008564864635],
    [-118.808558110679, 34.0035027131376]
  ]
};

const simpleFillSymbol = {
  type: 'simple-fill',
  color: [227, 139, 79, 0.8],  // orange, opacity 80%
  outline: {
    color: [255, 255, 255],
    width: 1
  }
};
```

```
const polygonGraphic = new Graphic({
  geometry: polygon,
  symbol: simpleFillSymbol
});

this.gisService.graphicsLayer.add(polygonGraphic);
```

Draw graphics

20 minutes

Applications can provide the ability for users to draw and edit graphics on a map.

Graphics represent geometric shapes such as points, lines, and polygons and are generally stored in memory in a graphics layer.

In the *loadModules* statement, add the *GraphicsLayer* and *SketchViewModel* modules.

```
const [
  GraphicsLayer,
  SketchViewModel,
] = await loadModules([
  'esri/layers/GraphicsLayer',
  'esri/widgets/Sketch/SketchViewModel'
]);
```

Create and add a GraphicsLayer to the map.

```
this.gisService.graphicsLayer = new GraphicsLayer();
this.gisService.map.add(this.gisService.graphicsLayer);
```

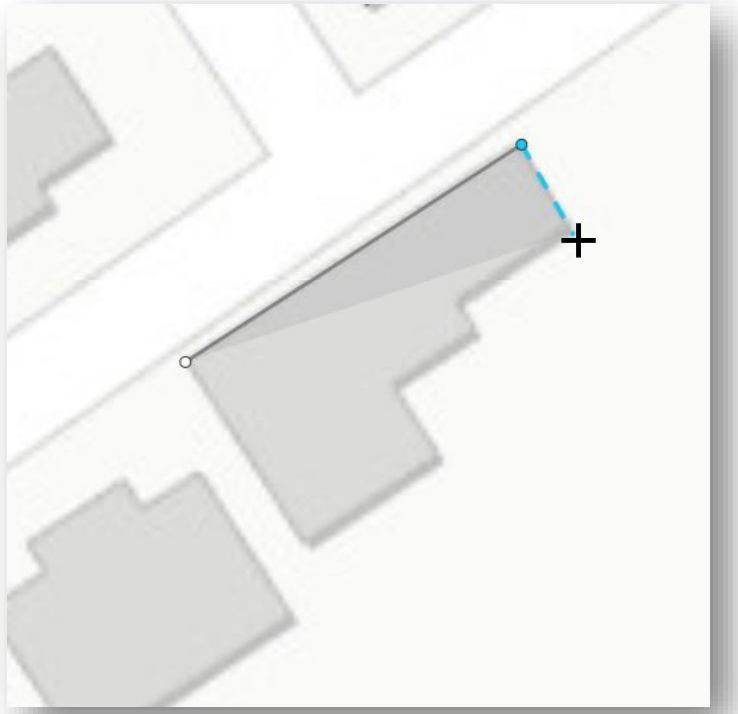
Create a new SketchViewModel and set its required parameters.

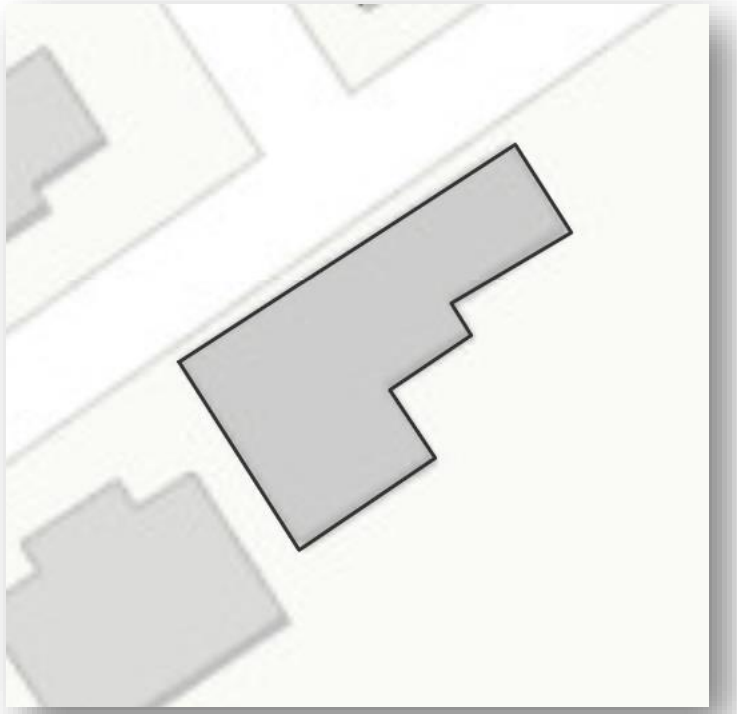
```
const sketchVM = new SketchViewModel({
  layer: this.gisService.graphicsLayer,
  view: this.gisService.mapView
});
```

Call create method to create a polygon with hybrid option. Listen to create event, only respond when event's state changes to complete and print out to console.

```
sketchVM.create('polygon', { mode: 'hybrid' });

sketchVM.on('create', (event: any) => {
  if (event.state === 'complete') {
    console.log(event);
  }
});
```

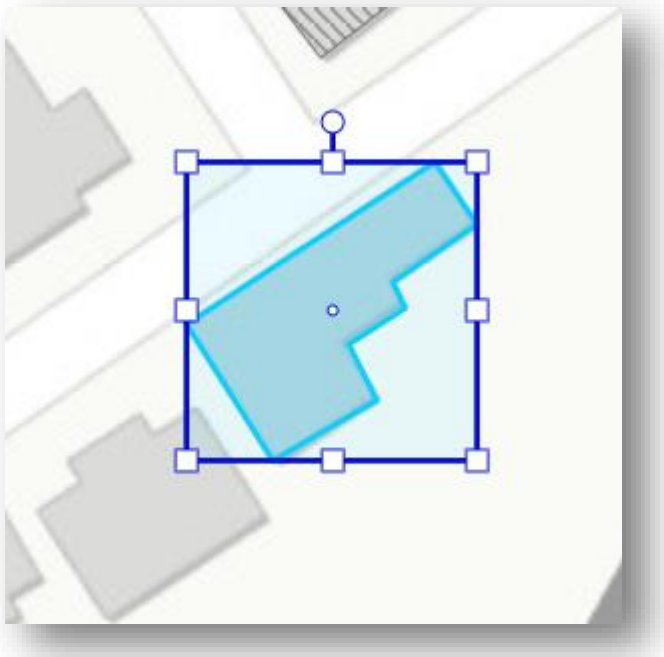
Click on the map to start drawing polygon graphic, and double click its to stop drawing.



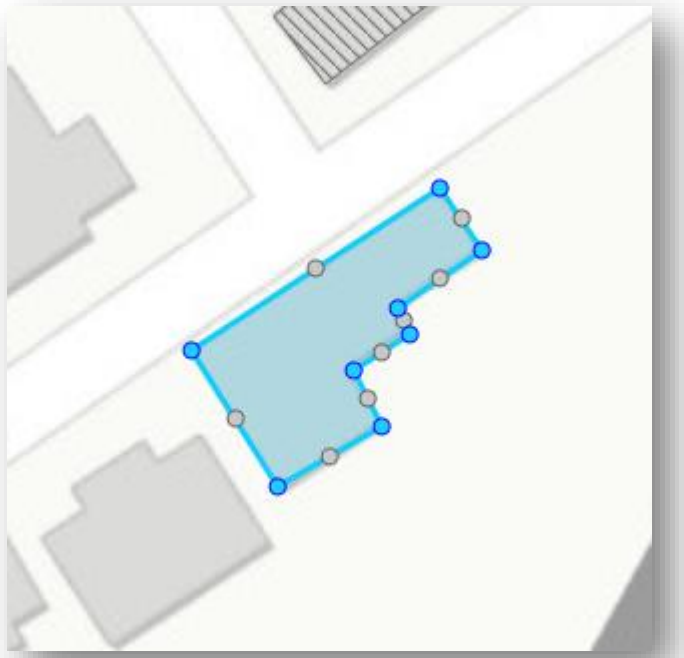


After you added the graphic, you can edit graphic with the following method:

Click at graphic to transform and move



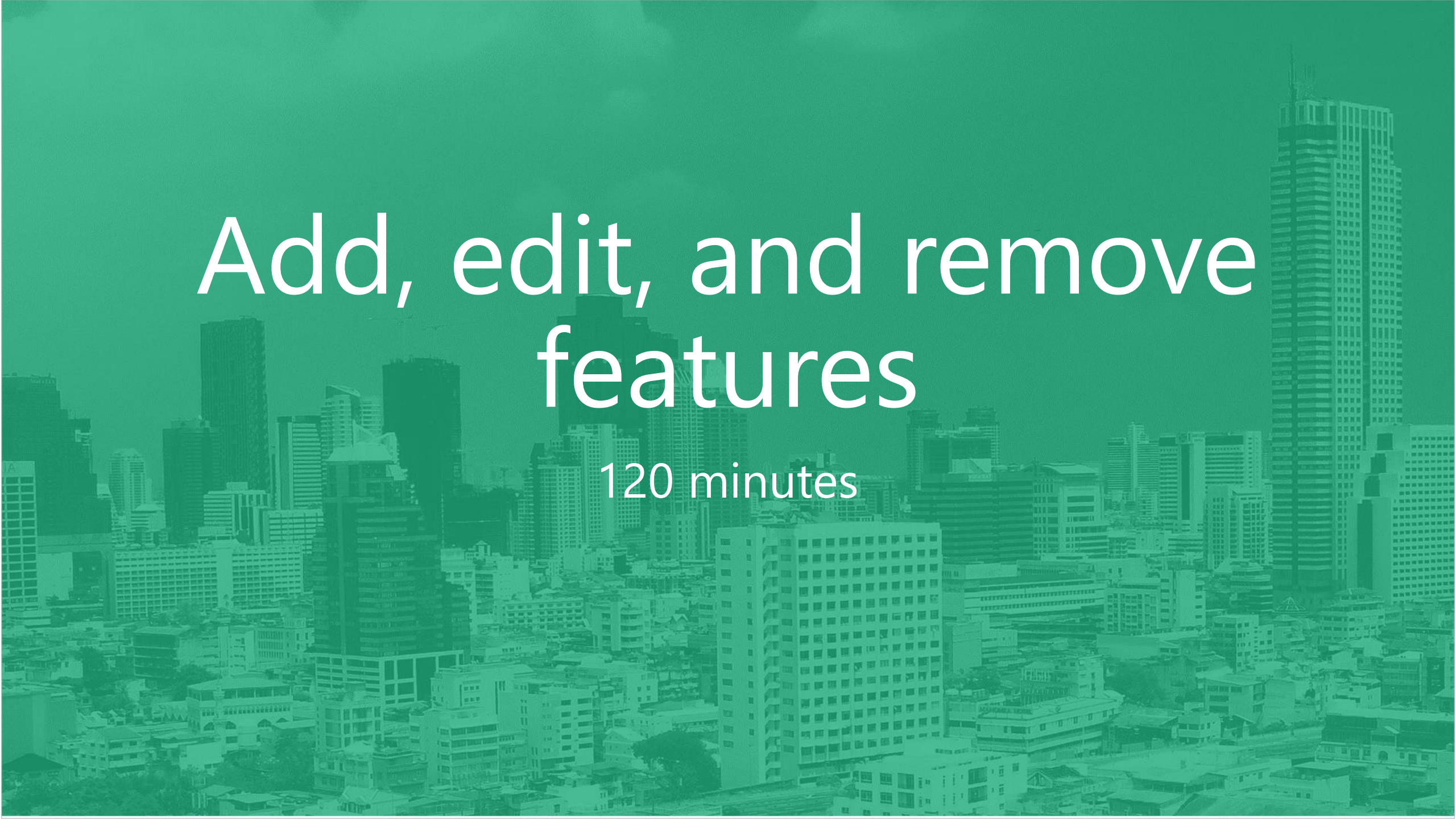
Two click at graphic to reshape and move



Sketch widget provides a simple UI for creating and updating graphics on a *MapView* or a *SceneView*. This significantly minimizes the code required for working with graphics in the view. It is intended to be used with graphics stored in its layer property.

By default, the Sketch widget provides out-of-the-box tools for creating and updating graphics with point, polyline, polygon, rectangle and circle geometries.


```
const sketch = new Sketch({
  view: this.gisService.mapView,
  layer: this.gisService.graphicsLayer
});
```



Add, edit, and remove features

120 minutes

New features can be created, and existing features can be updated or deleted. Feature geometries and/or attributes may be modified. Only applicable to layers in a feature service and client-side features set through the layer's source.

If client-side features are added, removed or updated at runtime using *applyEdits()* then use *queryFeatures()* to return updated features.

An array or a collection of features to be added. Values of non nullable fields must be provided when adding new features. Date fields must have numeric values representing universal time.

Create building block layer with the Feature Service (**FeatuerServer**) and add its to the map.

```
this.gisService.buildingBlockLayer = await this.gisService.createFeatuerLayer({
  url: 'https://appserver2.cdg.co.th/arcgis/rest/services/AtlasX/City/FeatureServer/1'
});
```

Create building block polygon of the Casa Lapin with the following code.

```
const polygonCasaLapin = {
  type: 'polygon',
  rings: [
    [
      [100.54478238021771, 13.704537982001284],
      [100.54480516618976, 13.704560119381359],
      [100.5448427171161, 13.704524940440976],
      [100.54482191091026, 13.704501494090048],
      [100.54478238021771, 13.704537982001284]
    ], [
      [100.54486483905937, 13.704289781176074],
      [100.54480786745414, 13.704480628557532],
      [100.54490105516184, 13.704496276115304],
      [100.54505394113666, 13.70434645219186],
      [100.54486483905937, 13.704289781176074]
    ]
  ]
};
```

Create building block attributes and graphic of the Casa Lapin with the following code.

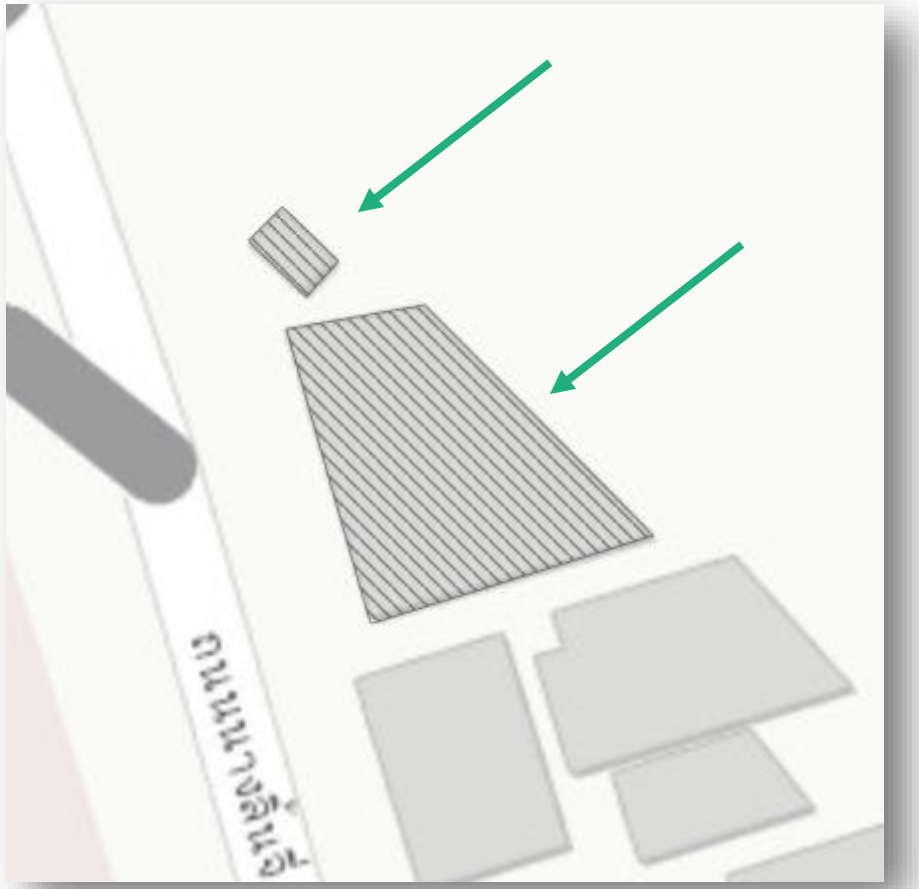
```
const attributesCasaLapin = {
  NAME: 'Casa Lapin Specialty Coffee x Rama 3',
  HEIGHT: 6,
  FLOOR: 1,
  BUILDING_TYPE: 1
};

const graphicCasaLapin = new Graphic({
  geometry: polygonCasaLapin,
  attributes: attributesCasaLapin
});
```

Call the applyEdits function with *addFeatures* parameter to create new feature in Feature Service and print out result in console.

```
const results = await this.gisService.buildingBlockLayer.applyEdits({
  addFeatures: [graphicCasaLapin]
});
console.log('Add Feature', results);
```

After run your app, go to the console to see output.

```
Add Feature                                esri-map.component.ts:310
  {addFeatureResults: Array(1), updateFeatureResult
  s: Array(0), deleteFeatureResults: Array(0), addAt
  tachmentResults: Array(0), updateAttachmentResult
  s: Array(0), …}
  ▶ addAttachmentResults: []
  ▼ addFeatureResults: Array(1)
    ▶ 0: {objectId: 6835, globalId: undefined, error…
      length: 1
    ▶ __proto__: Array(0)
  ▶ deleteAttachmentResults: []
  ▶ deleteFeatureResults: []
  ▶ updateAttachmentResults: []
  ▶ updateFeatureResults: []
  ▶ __proto__: Object
>
```



An array or a collection of features to be updated. Each feature must have valid ***objectId***. Values of non nullable fields must be provided when updating features. Date fields must have numeric values representing universal time.

Update building block polygon of the Casa Lapin with the following code.

```
const polygonCasaLapin = {
  type: 'polygon',
  rings: [
    [100.54486483905937, 13.704289781176074],
    [100.54480786745414, 13.704480628557532],
    [100.54490105516184, 13.704496276115304],
    [100.54505394113666, 13.70434645219186],
    [100.54486483905937, 13.704289781176074]
  ]
};
```

Update building block attributes, set OBJECTID attribute, update height from 6 to 8 meters and create graphic of the Casa Lapin with the following code.

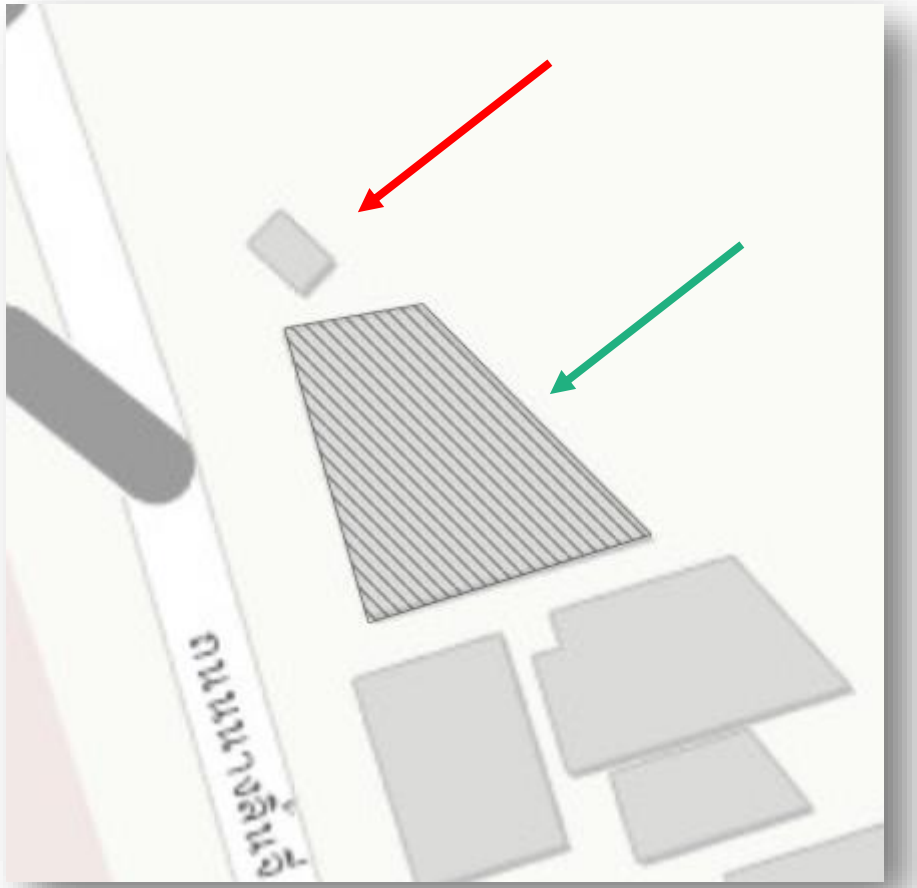
```
const attributesCasaLapin = {
  OBJECTID: 6835, // <-- Added an objectId.
  NAME: 'Casa Lapin Specialty Coffee x Rama 3',
  HEIGHT: 8,
  FLOOR: 1,
  BUILDING_TYPE: 1
};

const graphicCasaLapin = new Graphic({
  geometry: polygonCasaLapin,
  attributes: attributesCasaLapin
});
```

Call the applyEdits function with *updateFeatures* parameter to update the Casa Lapin feature in Feature Service and print out result in console.

```
const results = await this.gisService.buildingBlockLayer.applyEdits({
  updateFeatures: [graphicCasaLapin]
});
console.log('Edit Feature', results);
```

After run your app, go to the console to see output.

```
Edit Feature                    esri-map.component.ts:340
  {addFeatureResults: Array(0), updateFeatureResult
  s: Array(1), deleteFeatureResults: Array(0), addAt
  tachmentResults: Array(0), updateAttachmentResult
  s: Array(0), …}
    addAttachmentResults: []
    addFeatureResults: []
    deleteAttachmentResults: []
    deleteFeatureResults: []
    updateAttachmentResults: []
    updateFeatureResults: Array(1)
      0:
        error: null
        globalId: undefined
        objectId: 6835
        __proto__: Object
      length: 1
      __proto__: Array(0)
    __proto__: Object
>
```



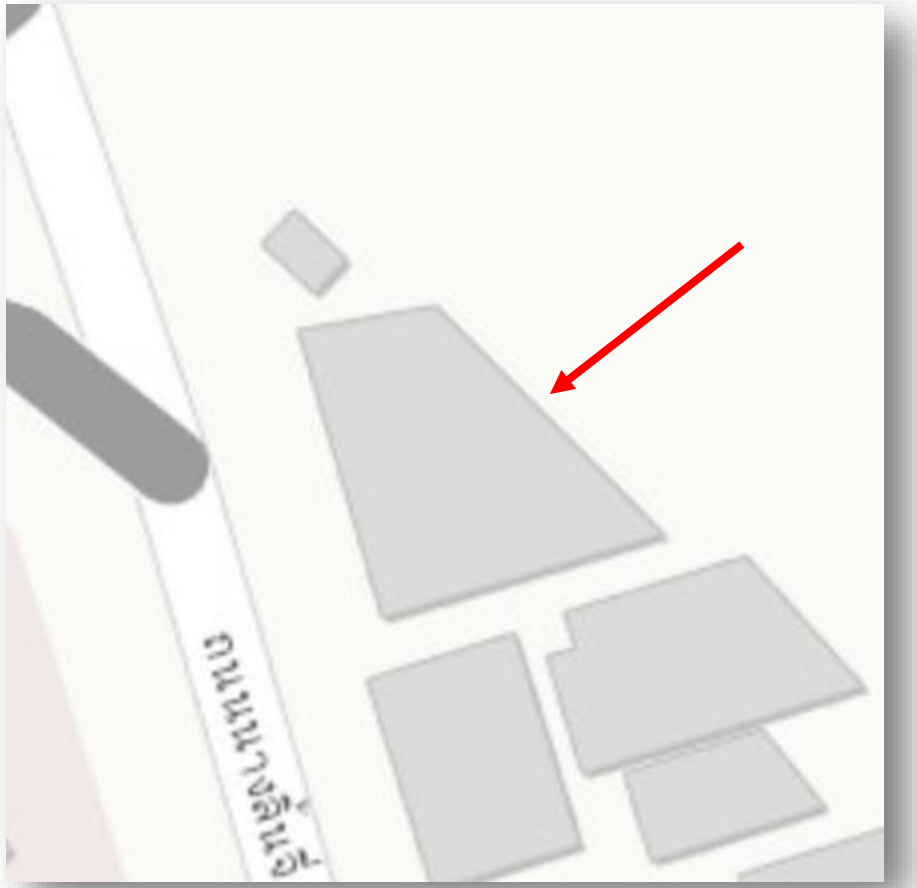
An array or a collection of features, or an array of objects with *objectId* or *globalId* of each feature to be deleted. When an array or collection of features is passed, each feature must have a valid objectId. When an array of objects is used, each object must have a valid value set for objectId or globalId property.

Call the applyEdits function with *deleteFeatures* parameter to remove the Casa Lapin feature in Feature Service and print out result in console.

```
const results = await this.gisService.buildingBlockLayer.applyEdits({
  deleteFeatures: [{ objectId: 6835 }]
});
console.log('Delete Feature', results);
```

After run your app, go to the console to see output.

```
Delete Feature                esri-map.component.ts:348
  {addFeatureResults: Array(0), updateFeatureResult
  s: Array(0), deleteFeatureResults: Array(1), addAt
  tachmentResults: Array(0), updateAttachmentResult
  s: Array(0), …}
  ▶ addAttachmentResults: []
  ▶ addFeatureResults: []
  ▶ deleteAttachmentResults: []
  ▼ deleteFeatureResults: Array(1)
    ▼ 0:
        error: null
        globalId: undefined
        objectId: 6835
      ▶ __proto__: Object
      length: 1
    ▶ __proto__: Array(0)
  ▶ updateAttachmentResults: []
  ▶ updateFeatureResults: []
  ▶ __proto__: Object
>
```



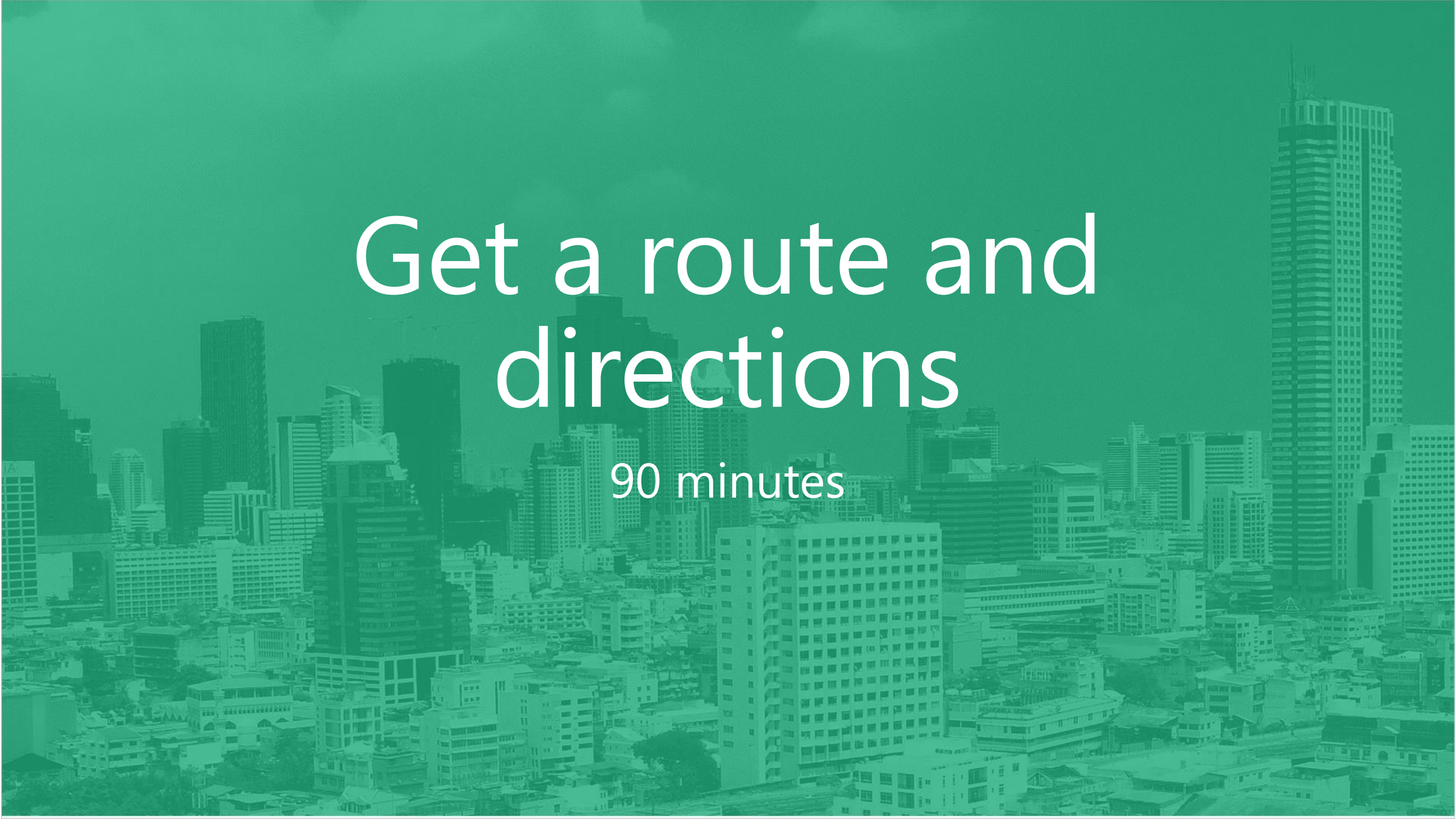
Create a div element with contain a button to create shape of building blocks and add it to the top right corner of the view (see example in the **Get map coordinates** topic).

Only use the SketchVeiwModel to active draw event (see example in the **Draw graphics** topic) when click the button.

When finish drawing, save the feature to building block.

Ref:

Handle click button – https://www.learnrxjs.io/learn-rxjs/operators/creation/fromevent



Get a route and directions

90 minutes

Applications can use the ArcGIS Routing and Network Analytics Services to find routes, get driving directions, calculate drive times, and solve complicated multiple vehicle routing problems.

To create an application that can access the service directly to find driving directions and create an optimized route, you can use ArcGIS World Routing Service and the **RouteTask** class.

To create a route, you provide stop locations, and optionally barriers and the mode of transportation, and the service will return a route with directions. Once you have the results you can add the route to a map, display turn-by-turn directions, or integrate them further into your application.

Before you can execute the route task you need to assemble the input parameters. There are a number of parameters you can provide such as stops, barriers and the preferred order, but at a minimum you need to provide the start and finish location.

In the gis.service.ts file, create a addRouteGraphic function to add start-stop graphics. Create a simple white marker for the start and a black marker for the finish location.

```
public async addRouteGraphic(type: 'start' | 'finish', point: any) {
  const [Graphic] = await loadModules(['esri/Graphic'])

  const graphic = new Graphic({
    symbol: {
      type: 'simple-marker',
      color: (type === 'start') ? 'white' : 'black',
      size: '12px'
    },
    geometry: point
  });
  this.mapView.graphics.add(graphic);
}
```

In the esri-map.component.ts file, create an initRoute function to implement a click handler and add start-stop graphics when the view is clicked. Call the initRoute function at end of function scope in ngOnInit function. After two graphics have been created, call the getRoute function to execute the route task. This will be completed in the next step.

```
initRoute() {
  this.gisService.mapView.on('click', (event: any) => {
    if (this.gisService.mapView.graphics.length === 0) {
      this.gisService.addRouteGraphic('start', event.mapPoint)
    } else if (this.gisService.mapView.graphics.length === 1) {
      this.gisService.addRouteGraphic('finish', event.mapPoint)
    } else {
      this.gisService.mapView.graphics.removeAll()
      this.gisService.addRouteGraphic('start', event.mapPoint)
    }
  });
}
```

Run the app and click on two locations to ensure the graphics are created.

Get a route and directions – Execute the route task

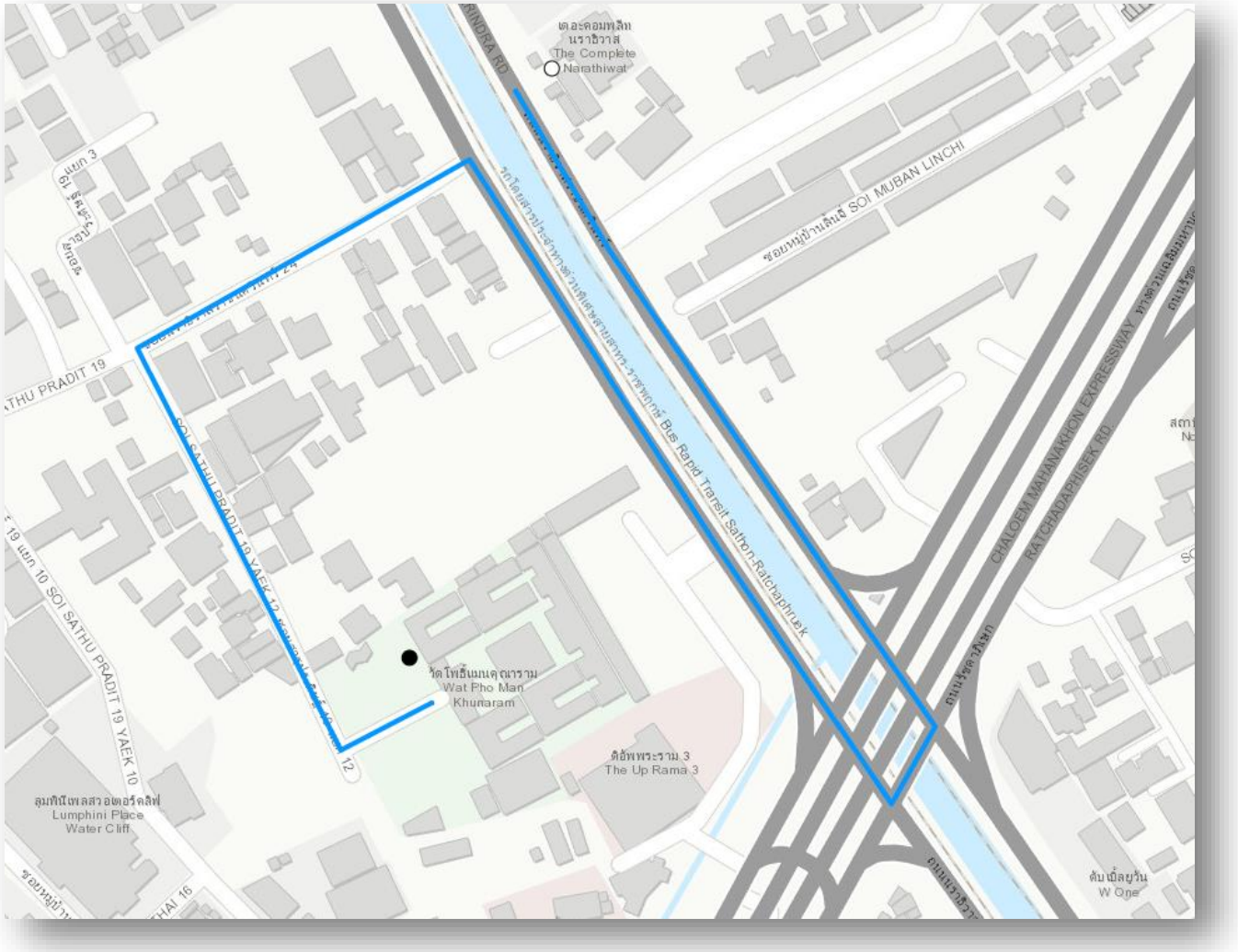
In the gis.service.ts file, write a new getRoute function to create the RouteParameters and pass in the stops graphics collected eariler. Also set returnDirections to true to ensure the text directions are returned as well. Call the solve method and when the task returns, extract the route from the RouteResult and add the route to the map with a blue symbol.

```
public async getRoute() {
    const [RouteTask, RouteParameters, FeatureSet] = await loadModules([
      'esri/tasks/RouteTask',
      'esri/tasks/support/RouteParameters',
      'esri/tasks/support/FeatureSet'
    ])

    const routeTask = new RouteTask({
      url: 'https://utility.arcgis.com/usrsvcs/appservices/ZwgWlSbYuzIavMco/rest/services
/World/Route/NAServer/Route_World/solve'
    })

    // Setup the route parameters
    const routeParams = new RouteParameters({
      stops: new FeatureSet({
        features: this.mapView.graphics.toArray() // Pass the array of graphics
      }),
      returnDirections: true
    });
    // Get the route
    const routeResult = await routeTask.solve(routeParams)
    // Display the route
    routeResult.routeResults.forEach((result: any) => {
      result.route.symbol = {
        type: 'simple-line',
        color: [5, 150, 255],
        width: 3
      }
      this.mapView.graphics.add(result.route)
    })
  }
```

Update the click handler code to call the getRoute function when the second graphic (finish) is passed in.

```
initRoute() {
  this.gisService.mapView.on('click', (event: any) => {
    if (this.gisService.mapView.graphics.length === 0) {
      this.gisService.addRouteGraphic('start', event.mapPoint)
    } else if (this.gisService.mapView.graphics.length === 1) {
      this.gisService.addRouteGraphic('finish', event.mapPoint)
      //*** ADD ***//
      this.gisService.getRoute();
    } else {
      this.gisService.mapView.graphics.removeAll()
      this.gisService.addRouteGraphic('start', event.mapPoint)
    }
  });
}
```

Run the app and click on two locations to generate a route.



Solve route with the **polylineBarriers** (hint: use points from click step 3 and 4 to create a polyline or use SketchViewModel to active *polyline* draw tool), after solve completed show the directions.