Advanced JavaScript for Web Sites and Web Applications

Geolocation and Google Maps

Geolocation API

- The geolocation API allows our code to retrieve information about the user’s geographic location.
- It does so by asking the user’s permission to retrieve their position.
- If they choose to share, their browser will populate the *geolocation object*
  - How it does this will vary from device to device

The Geolocation object

- The Geolocation API exposes the geolocation object via the *navigator* interface
  - The *navigator* interface represents the *state* of the user-agent (the browser)

```
navigator.geolocation
```

Getting the user’s location

- You can get the current location of a user by calling the object’s `getCurrentPosition()` method:

```
navigator.geolocation.getCurrentPosition(
    success,
    error
);
```

`getCurrentPosition()`

- `getCurrentPosition` accepts 2 arguments, both of which are *callback* functions:
- `success` : is executed when position is retrieved without errors (required argument)
- `error` : is executed when the browser encounters an error retrieving the position

Callback arguments

- The `success` callback function will receive a *position object* as an argument.
  - The contents of this object will vary from device to device
- The `error` callback function will receive an *error object* as an argument
  - Contains information about the error that occurred

getCurrentPosition() and callbacks

```
// A success callback
function geoYes(pos) {
    console.log(pos);
}
// An error callback
function geoNo(err) {
    console.log(err);
}

navigator.geolocation.getCurrentPosition(
    geoYes,
    geoNo
);
```

The *position* object

- The *position object* received by the success callback will contain at least 2 other objects
  - `coords` : object representing the current location
  - `timestamp` : Unix timestamp representing the time the location was retrieved
- Within the `coords` object, we can find various pieces of data related to the user’s location
- Of most interest to us are the properties: `latitude` and `longitude`

The *position* object: latitude and longitude

- Retrieving the latitude and longitude values:

```
function geoYes(pos) {
    // latitude
    console.log(pos.coords.latitude);
    // longitude
    console.log(pos.coords.longitude);
}
```

The *error* object

- The *error* object passed to our error callback has 2 useful properties:
- `message` : A human-readable message describing the error
- `code` : A numeric code which can be `1` (permission denied), `2` (position unavailable) or `3` (timeout)
- NB: The *message* is for debugging purposes, and is not intended to be displayed to the end user

Debugging with the *error* object

- Retrieving error information:

```
function geoNo(err) {
    console.log(err.code);
    console.log(err.message);
}
```

Geolocation - the user is in control!

- When we attempt to retrieve the user’s location, the browser will ask the user if they want to share or not
- If the user chooses not to share their data, what happens next will vary from browser to browser and can also be influenced by the environment…

Geolocation - when the user doesn't share

- In some scenarios, a geolocation error will be triggered, so our *error* callback will run.
- In other scenarios, nothing will happen!
  - neither the *success* nor *error* callbacks will run
- This may cause a problem, as our page will *hang* while we wait for a response from geolocation!

Geolocation - handling user behaviour

- One solution to this is to use a `timeout` function to monitor the user’s response (or lack of response)
- If no response has been received after a specified amount of time, we can execute a fallback routine

Geolocation - handling user behaviour (1)

```
// Set a "flag" variable
var flag = '';
function geoYes(position) {
    // Update flag
    flag = 'georeturned';
    // + other code
}
function geoNo(err) {
    // Update flag
    flag = 'georeturned';
    // + other code
}
```

Geolocation - handling user behaviour (2)

```
// Ask for location
navigator.geolocation.getCurrentPosition(
    geoYes,
    geoNo
);
// Assess value of "flag" after 5 secs
setTimeout(function () {
    if (flag == ''){
        console.log("No callback fired after 5 sec
    } else {
        console.log("A callback has fired");
    }
}, 5000);
```

Google Maps

The Google Map API

- To embed a Google map in our page, we first have to load the Google Maps API:

```
<script
    src="https://maps.googleapis.com/maps/api/js">
</script>
```

Creating a map object

- Once the API is loaded, we can initialise a new Google Map object and store it in a variable: `gMap`

```
var gMap = new google.maps.Map(
        mapElement,
        mapOptions
    );
```

The `Map` method

- The `Map` method accepts two parameters:
- `mapElement` : the html element on the page we want to insert the map in (normally an empty div)
- `mapOptions` : an object that will hold config options for the map

The Map options object

- The *options* object we pass to the `Map` method can hold many different properties, which define the way the map will behave
- There are 2 required properties that *must* be included:
  - `center` : An object containing the *latitude* and *longitude* coordinates of the map's center
  - `zoom` : A number specifying the *zoom* level (0-19 for default map type)

About latitude and longitude

- When using the API, we often need to specify latitude and longitude coordinates
  - e.g. the `center` property we pass to the `Map` method
- There are 2 ways of specifying these coordinates:
  - as a Google Maps `LatLng` object
  - as a Google Maps `LatLngLiteral` object

`LatLng` VS `LatLngLiteral`

- Assuming our coordinates are stored in variables:

```
myLat = 51.527278532168275;
myLng = -0.10360836982727051;
```

`LatLng` VS `LatLngLiteral`

- The `LatLngLiteral` object is a simple object with 2 properties:

```
myCoords = {
    lat: myLat,
    lng: myLng
};
```

`LatLng` vs `LatLngLiteral`

- The `LatLng` object is created with a constructor function:

```
myCoords = new google.maps.LatLng(
    myLat,
    myLng
);
```

`LatLng` VS `LatLngLiteral`

- While the *Literal* object will work in most scenarios, we will use the *LatLng* object as it works in *all* scenarios

The Map options object : example

```
var myLat = 51.527278532168275,
    myLng = -0.10360836982727051,
    centerObj = new google.maps.LatLng(
        myLat,
        myLng
    ),
    mapOptions = {
        center: centerObj,
        zoom: 8
    }
```

Loading the map

- Before we can create our map, we need to be sure that the DOM has been loaded by the browser.
- We can do this via the `addDomListener` method of Google Maps *event* interface, which allows us to attach a function to the window’s *load* event

Adding the *load* event

- `addDomListener` is very similar to the native `addEventListener`
- We pass it the *element* to attach the listener to, the *event* to listen for and the *function* to run when the event fires

```
google.maps.event.addDomListener(
    window,
    "load",
    initMap
);
```

The *load* event handler

- The event handler function can then safely create a new Map object:

```
function initMap() {
    gMap = new google.maps.Map(
        mapElement,
        mapOptions
    );
}
```

A working example (1):

```
// (1) set up the variables:
var gMap,
    mapEl = document.getElementById('my-map');
    myLat = 51.527278532168275,
    myLng = -0.1036083698272705l,
    centerObj = new google.maps.LatLng(
        myLat,
        myLng
    ),
    mapOpts = {
        center: centerObj,
        zoom: 8
    };
```

A working example (2 & 3):

```
// (2) Define the load event handler:
function initMap() {
    gMap = new google.maps.Map(mapEl, mapOpts);
}

// (3) Attach handler to load event:
google.maps.event.addDomListener(
    window,
    "load",
    initMap
);
```

Manipulating the map

- Once the new Google Map has been created, we can change some of its properties by calling it’s various methods

Manipulating the map - setting the center

- `setCenter(coords)` will change where the map is centered. We pass it a `LatLng` object holding the new coordinates:

```
newLat = 51.527278532168275;
newLng = -0.1036083698272705l;
loc = new google.maps.LatLng(newLat, newLng);
// Set the center:
gMap.setCenter(loc);
```

Manipulating the map - setting the zoom level

- `setZoom(number)` will change the zoom level of the map
- For the default map type, `number` can be between 0 and 19

```
gMap.setZoom(16);
```

Adding markers to the map

- We can place a marker on a map by creating a `Marker` object
- We pass an *options* object with 2 properties to it's constructor:
  - `position` : a LatLng object holding the coordinates for the marker
  - `map` : the map object we want to add the marker to

Adding markers to the map

```
// Adding a marker to the "gMap" map
var options = {
    position: new google.maps.LatLng(40.7, -74),
    map: gMap
};
var myMarker = new google.maps.Marker(options);
```

Exercise

- Now do *Exercise 2* from the session exercises document.