

[Technologies ▾](#)[References & Guides ▾](#)[Feedback ▾](#)[Sign in](#) 

Third party APIs

[← Previous](#)[↑ Overview: Client-side web APIs](#)[Next →](#)

The APIs we've covered so far are built into the browser, but not all APIs are. Many large websites and services such as Google Maps, Twitter, Facebook, PayPal, etc. provide APIs allowing developers to make use of their data (e.g. displaying your twitter stream on your blog) or services (e.g. displaying custom Google Maps on your site, or using Facebook login to log in your users). This article looks at the difference between browser APIs and 3rd party APIs and shows some typical uses of the latter.

Prerequisites:	JavaScript basics (see first steps , building blocks , JavaScript objects), the basics of Client-side APIs
Objective:	To learn how third-party APIs work, and how to use them to enhance your websites.

What are third party APIs?

Third party APIs are APIs provided by third parties — generally companies such as Facebook, Twitter, or Google — to allow you to access their functionality via JavaScript and use it on your own site. As we showed in our [introductory APIs article](#), one of the most obvious examples is using the [Google Maps APIs](#) to display custom maps on your pages.

Let's look at our map example again (see the [source code on GitHub](#); [see it live also](#)), and use it to illustrate how third-party APIs differ from browser APIs.

Note: You might want to just [get all our code examples](#) at once, in which case you can then just search the repo for the example files you need in each section.

They are found on third-party servers

Browser APIs are built in to the browser — you can access them from JavaScript immediately. For example, the [Geolocation API](#) used in our example is accessed using the `geolocation` property of the `Navigator` object, which returns a `Geolocation` object. This example uses the `getCurrentPosition()` method of this object to request the device's current position:

```
1 | navigator.geolocation.getCurrentPosition(function(position) { ... });
```

Third party APIs, on the other hand, are located on third party servers. To access them from JavaScript you first need to connect to the API functionality and make it available on your page. This typically involves first linking to a JavaScript library available on the server via a `<script>` element, as seen in our example:

```
1 | <script type="text/javascript" src="https://maps.google.com/maps/api/js?ke
```

You can then start using the objects available in that library. For example:

```
1 | var latlng = new google.maps.LatLng(position.coords.latitude,position.coor
2 | var myOptions = {
3 |   zoom: 8,
4 |   center: latlng,
5 |   mapTypeId: google.maps.MapTypeId.TERRAIN,
6 |   disableDefaultUI: true
```

```
7 | }  
8 |  
9 | var map = new google.maps.Map(document.getElementById("map_canvas"), myOpt
```

Here we are creating a new `LatLng` object using the `google.maps.LatLng()` constructor, which contains the latitude and longitude of the location we want to show, as returned from the Geolocation API. Next, we create an options object (`myOptions`) containing this, and other information related to displaying the map. Finally, we actually create the map using the `google.maps.Map()` constructor, which takes as its parameters the element that we want to draw the map onto, and the options object.

This is all the information the Google Maps API needs to plot a simple map. The server you are connecting to handles all the complex stuff, like displaying the correct map tiles for the area being displayed, etc.

Note: Some APIs handle access to their functionality slightly differently, requiring the developer to make an HTTP request (see [Fetching data from the server](#)) to a specific URL pattern to retrieve specific data. These are called RESTful APIs, and we'll show an example of this later in the article.

Permissions are handled differently

Security for browser APIs tends to be handled by permission prompts, as discussed in our [first article](#). The purpose of these is so that the user knows what is going on in the websites they visit and is less likely to fall victim to someone using an API in a malicious way.

Third party APIs have a slightly different permissions system — they tend to use key codes to allow developers access to the API functionality. Look again at the URL of the Google Maps API library we linked to:

```
1 | https://maps.google.com/maps/api/js?key=AIzaSyDDuGt0E5IEGkcE6ZfrKfUtE9Ko_d
```

The URL parameter provided at the end of the URL is a developer key — the developer of the application must apply to get a key, and then include it in their code in a specific way to be allowed access to the API's functionality. In the case of Google Maps (and other Google APIs), you apply for a key at the [Google Cloud Platform](#).


Other APIs may require that you include the key in a slightly different way, but the pattern is fairly similar for most of them.

The point of requiring a key is so that not just anybody can use API functionality without any kind of accountability. When the developer has registered for a key, they are then known to the API provider, and action can be taken if they start to do anything malicious with the API (such as tracking people's location or trying to spam the API with loads of requests to stop it working, for example). The easiest action would be to just revoke their API privileges.

Extending the Google Maps example

Now we've examined the Google Maps API example and looked at how it works, let's add some more functionality to show how to use some other features of the API.

1. To start off this section, make yourself a copy of the [Google Maps starting file](#), in a new directory. If you've already cloned the [examples repository](#), you'll already have a copy of this file, which you can find in the `javascript/apis/third-party-apis/google-maps` directory.
2. Next, get your own developer key using the following steps:
 1. Go to the [Google Cloud Platform API Manager dashboard](#).
 2. Create a new project if you've not already got one.
 3. Click the *Enable API* button.
 4. Choose *Google Maps JavaScript API*.
 5. Click the *Enable* button.
 6. Click *Create credentials*, then choose *API key*.
 7. Copy your API key and replace the existing key in the example's first `<script>` element with your own (the bit between `?key=` and the attribute's closing quote mark `"`).

 **Note:** Getting Google-related API keys can be a bit difficult — the Google Cloud Platform API Manager has a lot of different screens, and the workflow can differ slightly depending on things like whether you already have an account set up. If you have trouble with this step, we will be glad to help — [Contact us](#).

3. Open up your Google Maps starting file, find the string `INSERT-YOUR-API-KEY-HERE`, and replace it with the actual API key you got from the Google Cloud Platform API Manager dashboard.

Adding a custom marker

Adding a marker (icon) at a certain point on the map is easy — you just need to create a new marker using the `google.maps.Marker()` constructor, passing it an options object containing the position to display the marker at (as a `LatLng` object), and the `Map` object to display it on.

1. Add the following just below the `var map ...` line:

```
1 | var marker = new google.maps.Marker({  
2 |   position: latlng,  
3 |   map: map  
4 | });
```

Now if you refresh your page, you'll see a nice little marker pop up in the center of the map. This is cool, but it is not exactly a custom marker — it is using the default marker icon.

2. To use a custom icon, we need to specify it when we create the marker, using its URL. First of all, add the following line above the previous block you added:

```
1 | var iconBase = 'https://maps.google.com/mapfiles/kml/shapes/';
```


This defines the base URL where all the official Google Maps icons are stored (you could also specify your own icon location if you wished).

3. The icon location is specified in the `icon` property of the options object. Update the constructor like so:

```
1 | var marker = new google.maps.Marker({  
2 |   position: latlng,  
3 |   icon: iconBase + 'flag_maps.png',  
4 |   map: map  
5 | });
```

Here we specify the icon property value as the `iconBase` plus the icon filename, to create the complete URL. Now try reloading your example and you'll see a custom marker displayed on your map!

 **Note:** See [Customizing a Google Map: Custom Markers](#) for more information.

 **Note:** See [Map marker](#) or [Icon names](#) to find out what other icons are available, and see what their reference names are. Their file name will be the icon name they display when you click on them, with ".png" added on the end.

Displaying a popup when the marker is clicked

Another common use case for Google Maps is displaying more information about a place when its name or marker is clicked (popups are called **info windows** in the Google Maps API). This is also very simple to achieve, so let's have a look at it.

1. First of all, you need to specify a JavaScript string containing HTML that will define the content of the popup. This will be injected into the popup by the API, and can contain just about any content you want. Add the following line below the `google.maps.Marker()` constructor definition:

```
1 | var contentString = '<div id="content"><h2 id="firstHeading" clas
```

2. Next, you need to create a new info window object using the `google.maps.InfoWindow()` constructor. Add the following below your previous line:

```
1 | var infowindow = new google.maps.InfoWindow({  
2 |   content: contentString  
3 | });
```

There are other properties available (see [Info Windows](#)), but here we are just specifying the `content` property in the options object, which points to the source of the content.

3. Finally, to get the popup to display when the marker is clicked, we use a simple click event handler. Add the following below the `google.maps.InfoWindow()` constructor:

```
1 | marker.addListener('click', function() {  
2 |     infowindow.open(map, marker);  
3 | });
```

Inside the function we simply invoke the `infowindow.open()` function, which takes as parameters the map you want to display it on, and the marker you want it to appear next to.

4. Now try reloading the example, and clicking on the marker!

Controlling what map controls are displayed

Inside the original `google.maps.Map()` constructor, you'll see the property `disableDefaultUI: true` specified. This disables all the standard UI controls you usually get on Google Maps.

1. Try setting its value to `false` (or just removing the line altogether) then reloading your example, and you'll see the map zoom buttons, scale indicator, etc.
2. Now undo your last change.
3. You can show or hide the controls in a more granular fashion by using other properties that specify single UI features. Try adding the following underneath the `disableDefaultUI: true` (remember to put a comma after `disableDefaultUI: true`, otherwise you'll get an error):

```
1 | zoomControl: true,  
2 | mapTypeControl: true,  
3 | scaleControl: true,
```

4. Now try reloading the example to see the effect these properties have. You can find more options to experiment with at the [MapOptions](#) object reference page.

That's it for now — have a look around the [Google Maps APIs documentation](#), and have some more fun playing!

A RESTful API — NYTimes

Now let's look at another API example — the [New York Times API](#). This API allows you to retrieve New York Times news story information and display it on your site. This type of API is known as a **RESTful API** — instead of getting data using the features of a JavaScript library like we did with Google Maps, we get data by making HTTP requests to specific URLs, with data like search terms and other properties encoded in the URL (often as URL parameters). This is a common pattern you'll encounter with APIs.

An approach for using third party APIs

Below we'll take you through an exercise to show you how to use the NYTimes API, which also provides a more general set of steps to follow that you can use as an approach for working with new APIs.

Find the documentation

When you want to use a third party API, it is essential to find out where the documentation is, so you can find out what features the API has, how you use them, etc. The New York Times API documentation is at <https://developer.nytimes.com/>.

Get a developer key

Most APIs require you to use some kind of developer key, for reasons of security and accountability. To sign up for an NYTimes API key, you need to go to <https://developer.nytimes.com/signup>.

1. Let's request a key for the "Article Search API" — fill in the form, selecting this as the API you want to use.
2. Next, wait a few minutes, then get the key from your email.
3. Now, to start the example off, make copies of [nytimes_start.html](#) and [nytimes.css](#) in a new directory on your computer. If you've already [cloned the examples repository](#), you'll already have a copy of these files, which you can find in the `javascript/apis/third-party-apis/nytimes` directory. Initially the `<script>` element contains a number of variables needed for the setup of the example; below we'll fill in the required functionality.

The app will end up allowing you to type in a search term and optional start and end dates, which it will then use to query the Article Search API and display the search results.

NY Times video search


Enter a SINGLE search term (required):

Enter a start date (format YYYYMMDD):

Enter an end date (format YYYYMMDD):

Quarterback Josh McCown Signs One-Year Deal With Jets


McCown, who will turn 38 this summer, will most likely serve as much as a mentor as he will a placeholder at his position.



Keywords:

A Busy Day for the Redskins, but No Certainty at Quarterback

Washington signed wide receiver Terrelle Pryor, but there's no guarantee Kirk Cousins will be throwing to him even though Cousins signed a contract for the franchise.



Connect the API to your app

First, you'll need to make a connection between the API, and your app. This is usually done either by connecting to the API's JavaScript (as we did in the Google Maps API), or by making requests to the correct URL(s).

In the case of this API, you need to include the API key as a get parameter every time you request data from it.

1. Find the following line:

```
1 | var key = 'INSERT-YOUR-API-KEY-HERE';
```

Replace `INSERT-YOUR-API-KEY-HERE` with the actual API key you got in the previous section.

2. Add the following line to your JavaScript, below the `// Event listeners to control the functionality` comment. This runs a function called `fetchResults()` when the form is submitted (the button is pressed).

```
1 | searchForm.addEventListener('submit', submitSearch);
```

3. Now add the `submitSearch()` and `fetchResults()` function definitions, below the previous line:

```
1 function submitSearch(e) {  
2   pageNumber = 0;  
3   fetchResults(e);  
4 }  
5  
6 function fetchResults(e) {  
7   // Use preventDefault() to stop the form submitting  
8   e.preventDefault();  
9  
10  // Assemble the full URL  
11  url = baseUrl + '?api-key=' + key + '&page=' + pageNumber + '&q=' +  
12  searchTerm + '&fq=' + fq;  
13  if(startDate.value !== '') {  
14    url += '&begin_date=' + startDate.value;  
15  };  
16  
17  if(endDate.value !== '') {  
18    url += '&end_date=' + endDate.value;  
19  };  
20  
21 }
```

`submitSearch()` sets the page number back to 0 to begin with, then calls `fetchResults()`. This first calls `preventDefault()` on the event object, to stop the form actually submitting (which would break the example). Next, we use some string manipulation to assemble the full URL that we will make the request to. We start off by assembling the parts we deem as mandatory for this demo:

- The base URL (taken from the `baseUrl` variable).
- The API key, which has to be specified in the `api-key` URL parameter (the value is taken from the `key` variable).
- The page number, which has to be specified in the `page` URL parameter (the value is taken from the `pageNumber` variable).
- The search term, which has to be specified in the `q` URL parameter (the value is taken from the value of the `searchTerm` text `<input>`).
- The document type to return results for, as specified in an expression passed in via the `fq` URL parameter. In this case, we just want to return articles.

Next, we use a couple of `if()` statements to check whether the `startDate` and `endDate` `<input>`s have had values filled in on them. If they do, we append their values to the URL,

specified in `begin_date` and `end_date` URL parameters respectively.

So, a complete URL would end up looking something like this:

```
1 | https://api.nytimes.com/svc/search/v2/articlesearch.json?api-key=4f3c267e1
2 | &fq=document_type:("article")&begin_date=20170301&end_date=20170312
```

Note: You can find more details of what URL parameters can be included in the [Article Search API reference](#).

Note: The example has rudimentary form data validation — the search term field has to be filled in before the form can be submitted (achieved using the required attribute), and the date fields have pattern attributes specified, which means they won't submit unless their values consist of 8 numbers (pattern="[0-9]{8}"). See [Form data validation](#) for more details on how these work.

Requesting data from the api

Now we've constructed our URL, let's make a request to it. We'll do this using the `Fetch API`.

Add the following code block inside the `fetchResults()` function, just above the closing curly brace:

```
1 | // Use fetch() to make the request to the API
2 | fetch(url).then(function(result) {
3 |     return result.json();
4 | }).then(function(json) {
5 |     displayResults(json);
6 | });
```

Here we run the request by passing our `url` variable to `fetch()`, convert the response body to JSON using the `json()` function, then pass the resulting JSON to the `displayResults()` function so the data can be displayed in our UI.

Displaying the data

OK, let's look at how we'll display the data. Add the following function below your `fetchResults()` function.

```
1 function displayResults(json) {
2   while (section.firstChild) {
3     section.removeChild(section.firstChild);
4   }
5
6   var articles = json.response.docs;
7
8   if(articles.length === 10) {
9     nav.style.display = 'block';
10  } else {
11    nav.style.display = 'none';
12  }
13
14  if(articles.length === 0) {
15    var para = document.createElement('p');
16    para.textContent = 'No results returned.';
17    section.appendChild(para);
18  } else {
19    for(var i = 0; i < articles.length; i++) {
20      var article = document.createElement('article');
21      var heading = document.createElement('h2');
22      var link = document.createElement('a');
23      var img = document.createElement('img');
24      var para1 = document.createElement('p');
25      var para2 = document.createElement('p');
26      var clearfix = document.createElement('div');
27
28      var current = articles[i];
29      console.log(current);
30
31      link.href = current.web_url;
32      link.textContent = current.headline.main;
33      para1.textContent = current.snippet;
34      para2.textContent = 'Keywords: ';
35      for(var j = 0; j < current.keywords.length; j++) {
36        var span = document.createElement('span');
37        span.textContent += current.keywords[j].value + ' ';
38        para2.appendChild(span);
39      }
```

```
40
41     if(current.multimedia.length > 0) {
42         img.src = 'http://www.nytimes.com/' + current.multimedia[0].url;
43         img.alt = current.headline.main;
44     }
45
46     clearfix.setAttribute('class', 'clearfix');
47
48     article.appendChild(heading);
49     heading.appendChild(link);
50     article.appendChild(img);
51     article.appendChild(para1);
52     article.appendChild(para2);
53     article.appendChild(clearfix);
54     section.appendChild(article);
55 }
56 }
57 };
```

There's a lot of code here; let's explain it step by step:

- The `while` loop is a common pattern used to delete all of the contents of a DOM element, in this case the `<section>` element. We keep checking to see if the `<section>` has a first child, and if it does, we remove the first child. The loop ends when `<section>` no longer has any children.
- Next, we set the `articles` variable to equal `json.response.docs` — this is the array holding all the objects that represent the articles returned by the search. This is done purely to make the following code a bit simpler.
- The first `if()` block checks to see if 10 articles are returned (the API returns up to 10 articles at a time.) If so, we display the `<nav>` that contains the *Previous 10/Next 10* pagination buttons. If less than 10 articles are returned, they will all fit on one page, so we don't need to show the pagination buttons. We will wire up the pagination functionality in the next section.
- The next `if()` block checks to see if no articles are returned. If so, we don't try to display any — we just create a `<p>` containing the text "No results returned." and insert it into the `<section>`.
- If some articles are returned, we first of all create all the elements that we want to use to display each news story, insert the right contents into each one, and then insert them into the DOM at the appropriate places. To work out which properties in the article objects contained the right data to show, we consulted the [Article Search](#)

API reference. Most of these operations are fairly obvious, but a few are worth calling out:

- We used a `for` loop (`for(var j = 0; j < current.keywords.length; j++) { ... }`) to loop through all the keywords associated with each article, and insert each one inside its own ``, inside a `<p>`. This was done to make it easy to style each one.
- We used an `if()` block (`if(current.multimedia.length > 0) { ... }`) to check whether each article actually has any images associated with it (some stories don't.) We display the first image only if it actually exists (otherwise an error would be thrown).
- We gave our `<div>` element a class of "clearfix", so we can easily apply clearing to it (this technique is needed at the time of writing to stop floated layouts from breaking.)

If you try the example now, it should work, although the pagination buttons won't work yet.

Wiring up the pagination buttons

To make the pagination buttons work, we will increment (or decrement) the value of the `pageNumber` variable, and then re-rerun the fetch request with the new value included in the page URL parameter. This works because the NYTimes API only returns 10 results at a time — if more than 10 results are available, it will return the first 10 (0-9) if the page URL parameter is set to 0 (or not included at all — 0 is the default value), the next 10 (10-19) if it is set to 1, and so on.

This allows us to easily write a simplistic pagination function.

1. Below the existing `addEventListener()` call, add these two new ones, which cause the `nextPage()` and `previousPage()` functions to be invoked when the relevant buttons are clicked:

```
1 | nextBtn.addEventListener('click', nextPage);
2 | previousBtn.addEventListener('click', previousPage);
```

2. Below your previous addition, let's define the two functions — add this code now:

```
1 | function nextPage(e) {
2 |   pageNumber++;
3 |   fetchResults(e);
```

```
4   };  
5  
6   function previousPage(e) {  
7     if(pageNumber > 0) {  
8       pageNumber--;  
9     } else {  
10      return;  
11    }  
12    fetchResults(e);  
13  };
```

The first function is simple — we increment the `pageNumber` variable, then run the `fetchResults()` function again to display the next page's results.

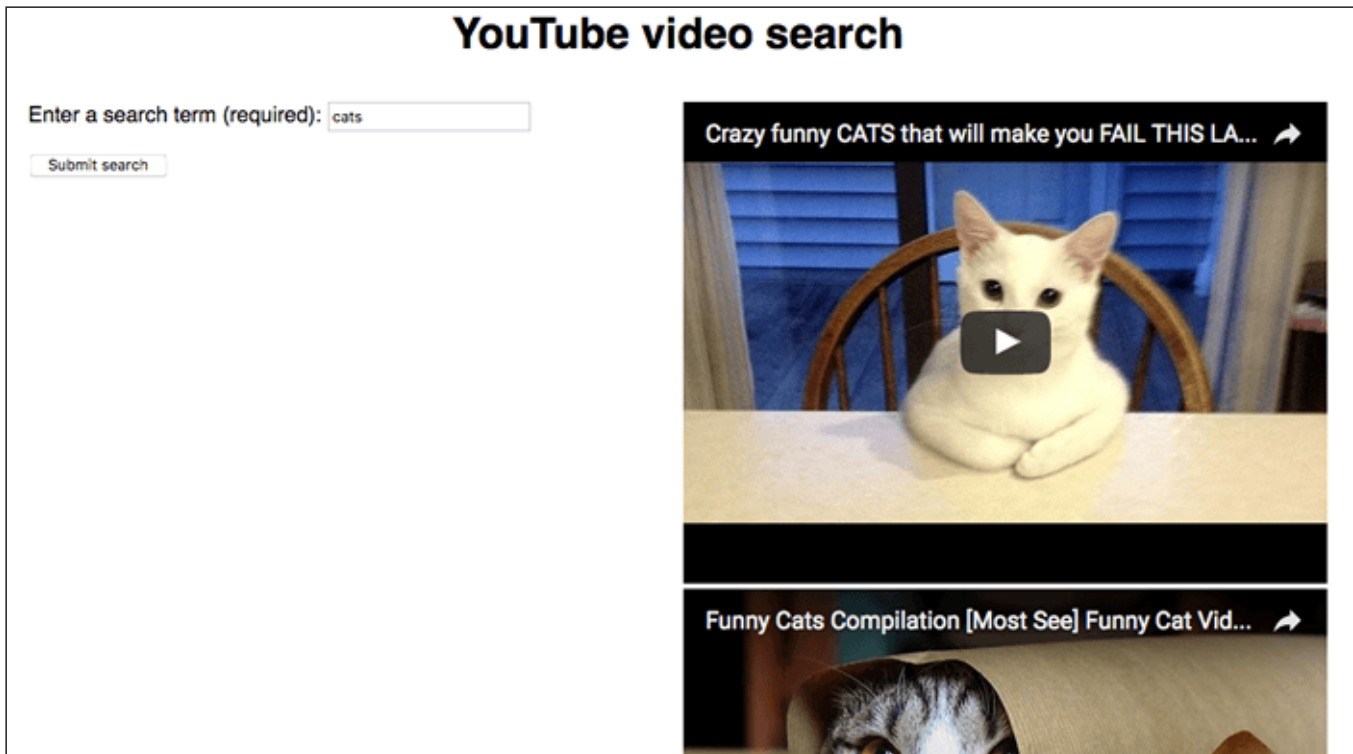
The second function works nearly exactly the same way in reverse, but we also have to take the extra step of checking that `pageNumber` is not already zero before decrementing it — if the fetch request runs with a minus page URL parameter, it could cause errors. If the `pageNumber` is already 0, we simply `return` out of the function, to avoid wasting processing power (If we are already at the first page, we don't need to load the same results again).

YouTube example

We also built another example for you to study and learn from — see our [YouTube video search example](#). This uses two related APIs:

- The [YouTube Data API](#) to search YouTube videos and return results.
- The [YouTube IFrame Player API](#) to display the returned video examples inside IFrame video players so you can watch them.

This example is interesting because it shows two related third-party APIs being used together to build an app. The first one is a RESTful API, while the second one works more like Google Maps (with constructors, etc.). It is worth noting however that both of the APIs require a JavaScript library to be applied to the page. The RESTful API has functions available to handle making the HTTP requests and returning the results, so you don't have to write them out yourself using say `fetch` or `XHR`.



We are not going to say too much more about this example in the article — [the source code](#) has detailed comments inserted inside it to explain how it works.

Summary

This article has given you a useful introduction to using third party APIs to add functionality to your websites.

[← Previous](#)[↑ Overview: Client-side web APIs](#)[Next →](#)

In this module

- Introduction to web APIs
- Manipulating documents
- Fetching data from the server

- Third party APIs
 - Drawing graphics
 - Video and audio APIs
 - Client-side storage
-