Using the Cache API

By Mat Scales
(https://developers.google.com/web/resources/contributors/mscales)
Mat is a contributor to Web**Fundamentals**

The Cache API is a system for storing and retrieving network requests and corresponding responses. These might be regular requests and responses created in the course of running your application, or they could be created solely for the purpose of storing some data in the cache.

The Cache API was created to enable Service Workers to cache network requests so that they can provide appropriate responses even while offline. However, the API can also be used as a general storage mechanism.

Where is it available?

The API is currently available in Chrome, Opera and Firefox. Both Edge and Safari have marked the API as 'In Development'.

The API is exposed via the global `caches` property, so you can test for the presence of the API with a simple feature detection:

```
const cacheAvailable = 'caches' in self;
```

The API can be accessed from a window, iframe, worker, or service worker.

What can be stored

The caches only store pairs of `Request` and `Response` objects, representing HTTP requests and responses, respectively. However, the requests and responses can contain any kind of data that can be transferred over HTTP.

Create the `Request` object using a URL for the thing being stored:

```
const request = new Request('/images/sample1.jpg');
```

The **`Response`** object constructor accepts many types of data, including **`Blob`**s, **`ArrayBuffer`**s, **`FormData`** objects, and strings.

```
const imageBlob = new Blob([data], {type: 'image/jpeg'});
const imageResponse = new Response(imageBlob);

const stringResponse = new Response('Hello world');
```

You can set the MIME type of a **`Response`** by setting the appropriate header.

```
const options = {
  headers: {
    'Content-Type': 'application/json'
  }
}
const jsonResponse = new Response('{}', options);
```

Working with Response objects

If you have retrieved a **`Response`** and wish to access its body, there are several helper methods you can use. Each returns a **`Promise`** that resolves with a value of a different type.

Method	Description
`arrayBuffer`	Returns an **`ArrayBuffer`** containing the body, serialized to bytes.
`blob`	Returns a **`Blob`**. If the **`Response`** was created with a **`Blob`** then this new **`Blob`** has the same type. Otherwise, the **`Content-Type`** of the **`Response`** is used.
`text`	Interprets the bytes of the body as a UTF-8 encoded string.
`json`	Interprets the bytes of the body as a UTF-8 encoded string, then tries to parse it as JSON. Returns the resulting object, or throws a **`TypeError`** if the string cannot be parsed as JSON.
`formData`	Interprets the bytes of the body as an HTML form, encoded either as "multipart/form-data" or "application/x-www-form-urlencoded". Returns a FormData (https://developer.mozilla.org/en-US/docs/Web/API/FormData) object, or throws a **`TypeError`** if the data cannot be parsed.
`body`	Returns a ReadableStream (https://developer.mozilla.org/en-US/docs/Web/API/ReadableStream) for the body data.

For example

```
const response = new Response('Hello world');
response.arrayBuffer().then((buffer) => {
  console.log(new Uint8Array(buffer));
  // Uint8Array(11) [72, 101, 108, 108, 111, 32, 119, 111, 114, 108, 100]
});
```

Creating and opening a cache

To open a cache, use the **`caches.open(name)`** method, passing the name of the cache as the single parameter. If the named cache does not exist it is created. This method returns a **`Promise`** that resolves with the **`Cache`** object.

```
caches.open('my-cache').then((cache) => {
  // do something with cache...
});
```

Retrieving from a cache

To find an item in a cache, you can use the **`match`** method.

```
cache.match(request).then((response) => console.log(request, response));
```

If **`request`** is a string it is first be converted to a **`Request`** by calling **`new Request(request)`**. The function returns a **`Promise`** that resolves to a **`Response`** if a matching entry is found, or **`undefined`** otherwise.

To determine if two **`Requests`** match, more than just the URL is used. Two requests are considered different if they have different query strings, **`Vary`** headers and/or methods (**`GET`**, **`POST`**, **`PUT`**, etc.).

You can ignore some or all of these things by passing an options object as a second parameter.

```
const options = {
  ignoreSearch: true,
  ignoreMethod: true,
  ignoreVary: true
};
```

```
cache.match(request, options).then(...);
```

If more than one cached request matches then the one that was created first is returned.

If you want to retrieve *all* matching responses, you can use **`cache.matchAll`**.

```
const options = {
  ignoreSearch: true,
  ignoreMethod: true,
  ignoreVary: true
};

cache.matchAll(request, options).then((responses) => {
  console.log(`There are ${responses.length} matching responses.`);
});
```

As a shortcut you can search over all caches at once by using **`caches.match()`** instead of calling **`cache.match()`** for each cache.

Searching

The Cache API does not provide a way to search for requests or responses except for matching entries against a **`Response`** object. However, you can implement your own search using filtering or by creating an index.

Filtering

One way to implement your own search is to iterate over all entries and filter down to the ones that you want. Let's say that you want to find all items that have URLs ending with '.png'.

```
async function findImages() {
  // Get a list of all of the caches for this origin
  const cacheNames = await caches.keys();
  const result = [];

  for (const name of cacheNames) {
    // Open the cache
    const cache = await caches.open(name);

    // Get a list of entries. Each item is a Request object
    for (const request of await cache.keys()) {
```

```
      // If the request URL matches, add the response to the result
      if (request.url.endsWith('.png')) {
        result.push(await cache.match(request));
      }
    }
  }

  return result;
}
```

This way you can use any property of the `Request` and `Response` objects to filter the entries. Note that this is slow if you search over large sets of data.

Creating an index

The other way to implement your own search is to maintain a separate index of entries that can be searched, stored in IndexedDB. Since this is the kind of operation that IndexedDB was designed for it has much better performance with large numbers of entries.

If you store the URL of the `Request` alongside the searchable properties then you can easily retrieve the correct cache entry after doing the search.

Adding to a cache

There are three ways to add an item to a cache - `put`, `add` and `addAll`. All three methods return a `Promise`.

`cache.put`

The first is to use `cache.put(request, response)`. `request` is either a `Request` object or a string - if it is a string, then `new Request(request)` is used instead. `response` must be a `Response`. This pair is stored in the cache.

```
cache.put('/test.json', new Response('{"foo": "bar"}'));
```

`cache.add`

The second is to use `cache.add(request)`. `request` is treated the same as for `put`, but the `Response` that is stored in the cache is the result of fetching the request from the network. If the fetch fails, or if the status code of the response is not in the 200 range, then nothing is

stored and the `Promise` rejects. Note that cross-origin requests not in CORS mode have a status of 0, and therefore such requests can only be stored with `put`.

`cache.addAll`

Thirdly, there is `cache.addAll(requests)`, where `requests` is an array of `Requests` or URL strings. This works similarly to calling `cache.add` for each individual request, except that the `Promise` rejects if any single request is not cached.

In each of these cases, a new entry overwrites any matching existing entry. This uses the same matching rules described in the section on retrieving.

Deleting an item

To delete an item from a cache:

```
cache.delete(request);
```

Where request can be a `Request` or a URL string. This method also takes the same options object as `cache.match`, which allows you to delete multiple `Request`/`Response` pairs for the same URL.

```
cache.delete('/example/file.txt', {ignoreVary: true, ignoreSearch: true});
```

Deleting a cache

To delete a cache, call `caches.delete(name)`. This function returns a `Promise` that resolves to `true` if the cache existed and was deleted, or `false` otherwise.

Feedback

Was this page helpful?

Yes

No

Last updated May 29, 2019.

Chromium Blog
The latest news on the Chromium blog.

GitHub
Fork our code samples and other open-source projects.

Twitter
Connect with @ChromiumDev on Twitter.

Videos
Check out our videos.