**The Resource API**

The Angular **Resource API** is designed to simplify asynchronous resource loading by leveraging signals.

It offers a streamlined approach to managing data requests, tracking loading states, and automatically refreshing data when the signals on which the resource depends change.

**How to use a Resource?**

To use a **resource**, you can use the **resource()** function, where you define:

* A **request function** that returns the parameters for the async request;
* A **loader function** that fetches data based on the **request** parameters.

Here’s a simplified example:

import { resource, signal } from '@angular/core';  
  
const RESOURCE\_URL = 'https://jsonplaceholder.typicode.com/todos/';  
  
private id = signal(1);  
  
private myResource = resource({  
 request: () => ({ id: this.id() }),  
 loader: ({ request }) => fetch(RESOURCE\_URL + request.id),  
});

A **resource** automatically tracks the **request** parameters and, whenever they are updated, fetches new data using the **loader** function.

Most importantly, it monitors the status of the data-fetching operation, tracking the **resource**’s current **value**, **status**, and any **errors**.

For more details about the **Resource API**, including its lifecycle, error handling, and how to consume a resource, check out my previous article:

**[Angular resource( ) and rxResource( ) APIs: what you need to know](https://medium.com/@davidepassafaro/angular-resource-and-rxresource-apis-what-you-need-to-know-aa1c178e43e9?source=post_page-----9abd28b5df70---------------------------------------" \t "_blank)**

[Angular v19 introduced the new Resource API, designed to help us load asynchronous resources by leveraging the power of signals…](https://medium.com/@davidepassafaro/angular-resource-and-rxresource-apis-what-you-need-to-know-aa1c178e43e9?source=post_page-----9abd28b5df70---------------------------------------" \t "_blank)

[medium.com](https://medium.com/@davidepassafaro/angular-resource-and-rxresource-apis-what-you-need-to-know-aa1c178e43e9?source=post_page-----9abd28b5df70---------------------------------------" \t "_blank)

Now that we’ve had a good refresh on the **Resource API**, let’s dive into the new **httpResource**. 💪🏻

**The httpResource**

An **httpResource** is a specialized type of **resource** designed to handle **HTTP requests** and expose all related request information via **signals**.

**httpResponse( ) function**

To create an **httpResource** you can easily use the **httpResource()** function:

import { signal } from '@angular/core';  
import { httpResource, HttpResourceRequest } from '@angular/common/http';  
import { Todo } from './todo.model';  
  
const RESOURCE\_URL = 'https://jsonplaceholder.typicode.com/todos/';  
  
id = signal(1);  
  
staticHttpResource = httpResource<Todo>(  
 RESOURCE\_URL + this.id()  
);  
  
reactiveHttpResource = httpResource<Todo>(  
 () => RESOURCE\_URL + this.id()  
);  
  
configuredHttpResource = httpResource<Todo>(  
 { url: RESOURCE\_URL + this.id() } as HttpResourceRequest  
);  
  
reactiveConfiguredHttpResource = httpResource<Todo>(  
 () => ({ url: RESOURCE\_URL + this.id() } as HttpResourceRequest)  
);

**Note:** similar to a **resource**, an **httpResource** requires an injection context to be created. You can create one as a directive variable, instantiate it inside the constructor, or provide a specific injection context upon creation.

The **httpResource()** function requires only one parameter, which can be:

* A **string**, representing the URL from which to fetch the data;
* A **reactive function that returns a string**, the resource will fetch the data from the new URL whenever it changes due to a signal update;
* An object of type **HttpResourceRequest**, which allows specifying additional details such as headers and request body;
* A **reactive function that returns a HttpResourceRequest object**, the resource will fetch the data with the new request parameters whenever this object changes due to a signal update.

This flexibility allows **httpResource()** to adapt to various application needs, simplifying asynchronous resource consumption in a declarative way.

Under the hood, the provided parameter value is used to generate a function that creates an **HttpRequest**. This function is then passed to the basic **resource** implementation as the **request** function.

By providing reactive functions, **httpResource** can track to signal changes, ensuring the **resource** updates dynamically when dependencies change.

**Note:** you can find a detailed explanation of **HttpResourceRequest** below 👇🏻

**Configure the Resource with HttpResourceOptions**

The **httpResource()** function accepts an optional second parameter of type **HttpResourceOptions**, allowing you to specify the following properties:

* **defaultValue**: the value that the resource will take in **Idle**, **Loading**, or **Error** states. If not specified, the resource will default to **undefined**;
* **parse**: a function that processes the raw response of the http request before assigning it to the **resource** value. If not specified, TypeScript will infer the provided type through a type assertion.  
  It allows validation, transformation, or schema enforcement:

import { httpResource} from '@angular/common/http';  
import { defaultUser, isValidUser, toUser } form './user.utils';  
  
userResource =  
 httpResource<User>(() => `/users/${userId()}`, {  
 defaultValue: defaultUser,  
 parse: (raw) => {  
 if (!isValidUser(raw)) {  
 throw new Error("Invalid user data received");  
 }  
 return toUser(raw);  
 },  
 });  
}

* **injector**: overrides the **Injector** used by the **httpResource** instance to destroy itself when the parent component or service is destroyed;
* **equal**: equality function used to compare the response value.

**Fetching data types other than JSON**

By default, an **httpResource** assumes the response type is JSON.

In order to request a different type of data, you can use specific variants of **httpResource()** function provided by **httpResource**:

* **httpResource.text()**: to handle responses in plain text format;
* **httpResource.blob()**: to handle binary data, such as images or files;
* **httpResource.arrayBuffer()**: to handle raw binary data as **ArrayBuffer**.

import { httpResource } from '@angular/common/http';  
  
// Fetching plain text  
textResource = httpResource.text(() => '/api/endpoint', {  
 defaultValue: ''  
});  
  
// Fetching binary data (e.g., image)  
imageResource = httpResource.blob(() => '/api/image', {  
 defaultValue: new Blob()  
});  
  
// Fetching raw binary data as ArrayBuffer  
arrayBufferResource = httpResource.arrayBuffer(() => '/api/data', {  
 defaultValue: new ArrayBuffer(0)  
});

**HttpClient integration and Interceptors support**

Under the hood, **httpResource** leverages on the **HttpClient**, so to use it, you need to provide it to your component or service, usually via **provideHttpClient** or by importing **HttpClientModule**.

This integration allows you to leverage any **HttpInterceptor** you have in place for request transformation, authentication handling, and error management. This way, you can seamlessly integrate **httpResource** into your existing app while fully benefiting from your existing HTTP setup.

**Note:** the **HttpClient** implementation may be removed in future updates, but it’s likely that an alternative solution will be provided to ensure continued functionality. So, keep an eye on future updates! 🫣

**Real-World Use Cases**

**Theory is great, but** it’s the practical implementation that truly matters.  
Let’s dive into some real-world examples using **httpResource**.

**Forms and Search optimizations**

When building dynamic forms or search features, optimizing data requests is essential. One of the most common strategies is to **reduce unnecessary API calls by controlling when and how requests are triggered**.

This is easily achievable by tracking input changes and applying techniques to ensure requests are only made when the user has stopped typing or when the input value actually changes.

When dealing with **Observable**-based events, you can use operators like **debounceTime** to delay emissions until a specified time has passed without new values, or **distinctUntilChanged** to ensure only unique values are emitted.

However, since signals do not rely on the concept of time like Observables, these strategies cannot be applied directly when using **httpResource**.

In order to work around this limitation, you can combine the power of signals and Observables by leveraging the **toObservable()** and **toSignal()** functions provided by the **@angular/core/rxjs-interop** library.

Press enter or click to view image in full size



This allows you to transform signals into Observables, enabling you to apply operators like **debounceTime** and **distinctUntilChanged**, and then convert the results back into a signal that you can use inside the **httpResource**.

import { signal } from '@angular/core';  
import { httpResource } from '@angular/common/http';  
import { toObservable, toSignal } from '@angular/core/rxjs-interop';  
  
const RESOURCE\_URL = 'https://jsonplaceholder.typicode.com/todos/';  
  
id = signal(1);  
  
// Convert the signal into an observable and apply debounceTime  
idQuery$ = toObservable(this.id).pipe( debounceTime(3000) );  
  
// Convert the observable back to a signal  
query = toSignal(this.idQuery$);  
  
// Use the query signal inside httpResource to fetch data  
myResource = httpResource.text(  
 // This is triggered only after the debounce  
 () => RESOURCE\_URL + this.query()  
);

**Note:** the **distinctUntilChanged** operator is not not necessary here because signals already avoid emitting unless their value actually changes. 🤓

This hybrid approach gives you the flexibility to use time-based strategies while retaining the efficiency and simplicity of **httpResource** and signals.

Great! 🤩

**Disabling the request without destroying the resource**

In some cases, you might want to disable the request without destroying the resource entirely. For instance, when data fetching should only happen after a specific user action, such as clicking a button.

Instead of making the request immediately, you can conditionally enable it based on certain conditions, like so:

import { httpResource } from '@angular/common/http';  
  
const RESOURCE\_URL = 'https://jsonplaceholder.typicode.com/todos/';  
  
id = signal(1);  
shouldFetchData = signal(false);  
  
myResource = httpResource<User[]>(() => {  
 const shouldFetchData = this.shouldFetchData(); // Custom condition  
 return shouldFetchData ? `${RESOURCE\_URL}${this.query()}` : undefined;  
});

In this example, **httpResource** will only trigger the API request if the **shouldFetchData** signal is set to **true**.

If the condition is **false**, the function returns **undefined**, preventing any API call from being made and resetting the values of the **httpResource**.

Returning **undefined** is not the best, but it still gets the job done. 😅

**HttpResourceRequest in detail**

Providing an **HttpResourceRequest** object to the **httpResource()** function allows you to specify additional details for customizing the HTTP request made by the **httpResource**, enabling you to define the following properties:

* **url**: the URL of the request. It should not include query parameters, which can be specified through the dedicated **params** property;
* **method**: the HTTP method for the request. By default, the value is **GET**;
* **body**: the body to include in the request. If no **Content-Type** header is specified, Angular will try to set one based on the **body** type.
* **params**: query parameters to append to the URL, either as **HttpParams** or an object with key-value pairs, where values can be strings, numbers, booleans, or arrays of these types;
* **headers**: headers to include with the request, either as **HttpHeaders** or a plain object with header names as keys and string or array values;
* **context**: dictionary of key-value pairs to store additional context for the request, allowing it to carry custom metadata for handling or logging;
* **reportProgress**: if set to **true**, enables progress events for the request.  
  These events are provided through the **HttpResource.progress** signal;
* **withCredentials**: specifies whether to set the **withCredentials** flag on the outgoing request. When **true**, it allows the browser to send cookies and authentication information with the request;
* **transferCache**: configures the server-side rendering transfer cache for the request. It can be a boolean or an object with an **includeHeaders** array that specifies which headers to include in the transfer cache.

**Designed for fetching data only**

Even though you can define a different HTTP method using **HttpResourceRequest**, the new **httpResource** is primarily designed for fetching data, just like the fundamental **Resource API**.

This flexibility is provided to allow the API to consume data that doesn’t necessarily align with standard HTTP specifications.

However, it’s important to note that it is not safe to use an **httpResource**, or any **resource**, to save/delete data in such a way, as each update could abort previously initiated requests. As a result, operations like **POST**, **PUT**, or **DELETE** may not be guaranteed to complete successfully.