**Angular v19+ — Understanding the New**resource()**and**rxResource()**APIs 🔥🚀**

[[Giorgio Galassi](https://medium.com/@giorgio.galassi?source=post_page---byline--8a387c7d9351---------------------------------------)](https://medium.com/@giorgio.galassi?source=post_page---byline--8a387c7d9351---------------------------------------)

[Giorgio Galassi](https://medium.com/@giorgio.galassi?source=post_page---byline--8a387c7d9351---------------------------------------)

Follow

5 min read

·

Dec 3, 2024

77

1

**Angular 19 is here, and its renaissance continues, further embracing the power of signals.**  
This release introduces two new primitives that simplify handling asynchronous requests: **resource()** and **rxResource()**. These APIs enable a cleaner and more intuitive way to manage asynchronous data in Angular applications.

While resource() and rxResource() provide powerful tools for handling API calls reactively, **Angular 19.2** introduces yet another step forward: **httpResource()**, a new API that integrates seamlessly with HttpClient, eliminating even more boilerplate.  
If you're interested in taking reactive API calls to the next level, don't miss our deep dive into **httpResource()** and how it compares with the existing solutions.

**Check out the new article on httpResource()**👇🏻

**[Angular v19+ — Understanding the New httpResource() API 🔥🚀](https://medium.com/@giorgio.galassi/angular-v19-understanding-the-new-httpresource-api-837e1dadc990?source=post_page-----8a387c7d9351---------------------------------------" \t "_blank)**

[Angular v19.2 has been live for a few days, and once again, the Angular team hasn’t missed the opportunity to introduce…](https://medium.com/@giorgio.galassi/angular-v19-understanding-the-new-httpresource-api-837e1dadc990?source=post_page-----8a387c7d9351---------------------------------------" \t "_blank)

[medium.com](https://medium.com/@giorgio.galassi/angular-v19-understanding-the-new-httpresource-api-837e1dadc990?source=post_page-----8a387c7d9351---------------------------------------" \t "_blank)

*Note: These APIs are still in****experimental****state!*

**Signals: A Quick Refresher**

Signals represent a new paradigm for Angular application development. Paired with a [**Zoneless**](https://angular.dev/guide/experimental/zoneless) approach, they not only enhance application performance but also improve the Developer Experience (DX).

By using a signal in your view, you can precisely inform Angular which node in the component tree needs to be updated. This fine-grained reactivity results in faster updates and a more efficient rendering pipeline.

Here’s a quick example demonstrating how to create and manipulate a signal in Angular:

import { Component, signal } from "@angular/core";  
  
@Component({  
 selector: "my-component",  
 template: `  
 <p>{{ counter() }}</p>  
  
 <button (click)="add()">Add</button>  
 <button (click)="remove()">Remove</button>  
 <button (click)="reset()">Reset</button>  
 `,  
})  
export default class MyComponent {  
 counter = signal<number>(0);  
  
 add() {  
 this.counter.update((current) => current + 1);  
 }  
  
 remove() {  
 this.counter.update((current) => current - 1);  
 }  
  
 reset() {  
 this.counter.set(0);  
 }  
}

**Which Problem Are They Solving?**

In the early days of Angular signals, several new patterns emerged. One of these patterns tackled how to combine signals with the HttpClient to manage API responses. A common implementation might look like this:

import { Injectable, inject, signal } from '@angular/core';  
import { HttpClient } from '@angular/common/http';  
import { map } from 'rxjs';  
  
import { API\_URL, User } from './user';  
  
@Injectable({ providedIn: 'root' })  
export class ApiService {  
 #http = inject(HttpClient);  
 #response = signal<User[] | null>(null);  
  
 get users() {  
 return this.#response.asReadonly();  
 }  
  
 doApiCall(request: string = '') {  
 this.#http  
 .get<{ users: User[] }>(`${API\_URL}/search?q=${request}`)  
 .pipe(map(({ users }) => users))  
 .subscribe((response) => this.#response.set(response));  
 }  
}

While functional, this approach feels clunky and verbose. It introduces unnecessary boilerplate and requires developers to manually manage loading and error states. Moreover, mixing signals with observables can lead to code that is harder to maintain and reason about, reducing clarity and increasing cognitive overhead.

resource()**: Simplifying Asynchronous API Calls**

The new [resource](https://angular.dev/guide/signals/resource) primitive in Angular provides an elegant way to manage asynchronous API calls in a signal-based application. By leveraging resource, you can seamlessly integrate reactive API calls into your app with better state handling and cleaner code.

Here’s an example of how resource() can be used:

import { Component, signal, resource } from "@angular/core";  
  
@Component({  
 selector: "my-component",  
 template: `  
 <input (input)="search($event)" placeholder="Search user..."/>  
   
 <br />  
 <ul>  
 @let error = users.error();  
 @if (error) {  
 <p>{{ error }}</p>  
 }  
  
 @if (users.isLoading()) {  
 <p>Loading Users...</p>  
 }  
  
 @for (user of users.value(); track user.id) {  
 <li>{{ user.firstName }} {{ user.lastName }}</li>  
 } @empty {  
 <p>No Users!</p>  
 }  
 </ul>  
 `,  
})  
export default class MyComponent {  
 query = signal<string>("");  
  
 // The resource can be typed as follows:  
 // resource<response type, request type>  
 users = resource<User[], string>({  
 request: () => this.query(),  
 loader: async ({ request, abortSignal }) => {  
 const response = await fetch(`${API\_URL}/search?q=${request}`, {  
 signal: abortSignal,  
 });  
  
 if (!response.ok) throw new Error("Unable to load users!");  
 return (await response.json()).users;  
 },  
 });  
  
 search(event: Event) {  
 const { value } = event.target as HTMLInputElement;  
 this.query.set(value);  
 }  
}

**Diving Deeper into the**resource()**API**

The resource() API introduces two essential components to handle asynchronous data:

* **request:**A function that returns a value used for the loader computation. Every time query() changes, the loader function is re-triggered, behaving similarly to a computed signal.
* **loader:**The function where the actual API call is performed. It must return a promise, and errors can be handled like any standard promise.

**Built-In Properties of**resource()

The resource primitive comes with several useful out-of-the-box properties, all of which are signals:

* **value():**Retrieves the current value of the resource's response.
* **isLoading():**Indicates whether the resource is currently loading.
* **error():**Contains the error, if any, encountered during the API call.

**What About**rxResource()**?**

While resource() uses a promise-based loader, rxResource() provides a similar abstraction but leverages Observable streams instead. This makes it a perfect fit for scenarios where your application is already heavily reliant on RxJS or where Observables are the preferred choice for handling asynchronous data.

To illustrate this, let’s look at an example of how rxResource() can be used to perform a backend call while applying RxJS operators like distinctUntilChanged and map:

rxUsers = rxResource<User[], string | undefined>({  
 request: () => this.query(),  
 loader: ({ request }) =>  
 this.#http.get<{ users: User[] }>(`${API\_URL}/search?q=${request}`).pipe(  
 distinctUntilChanged(),  
 map(({ users }) => users),  
 catchError(() => {  
 throw Error('Unable to load!');  
 })  
 ),  
});

**Practical Example and Hands-On Exploration**

The power of resource() and rxResource() becomes evident when managing complex asynchronous workflows in your Angular application. By combining signals and these primitives, you can reduce boilerplate code while maintaining clarity and reactivity.

For example:

* Dynamically update UI components based on isLoading() or error() states.
* Create powerful search interfaces, as shown in the code above, that adapt in real-time as the user interacts.

To see these concepts in action, check out this interactive StackBlitz example:

**[StackBlitz](https://stackblitz.com/~/github.com/giorgiogalassi/resource?source=post_page-----8a387c7d9351---------------------------------------" \t "_blank)**

[Edit description](https://stackblitz.com/~/github.com/giorgiogalassi/resource?source=post_page-----8a387c7d9351---------------------------------------" \t "_blank)

[stackblitz.com](https://stackblitz.com/~/github.com/giorgiogalassi/resource?source=post_page-----8a387c7d9351---------------------------------------" \t "_blank)

In this demo, you’ll find:

* A fully functional search bar integrated with the resource() API.
* Examples of handling loading states, error handling, and real-time updates using signals.

Feel free to experiment by tweaking the code and observing how Angular’s reactivity makes the process seamless.

Thank you for staying with me, and I hope everything was clear. Feel free to explore more of my articles and [**follow me on LinkedIn**](https://www.linkedin.com/in/giorgiogalassi?utm_source=share&utm_campaign=share_via&utm_content=profile&utm_medium=ios_app)**!**

See you in the next one!

Best, G.