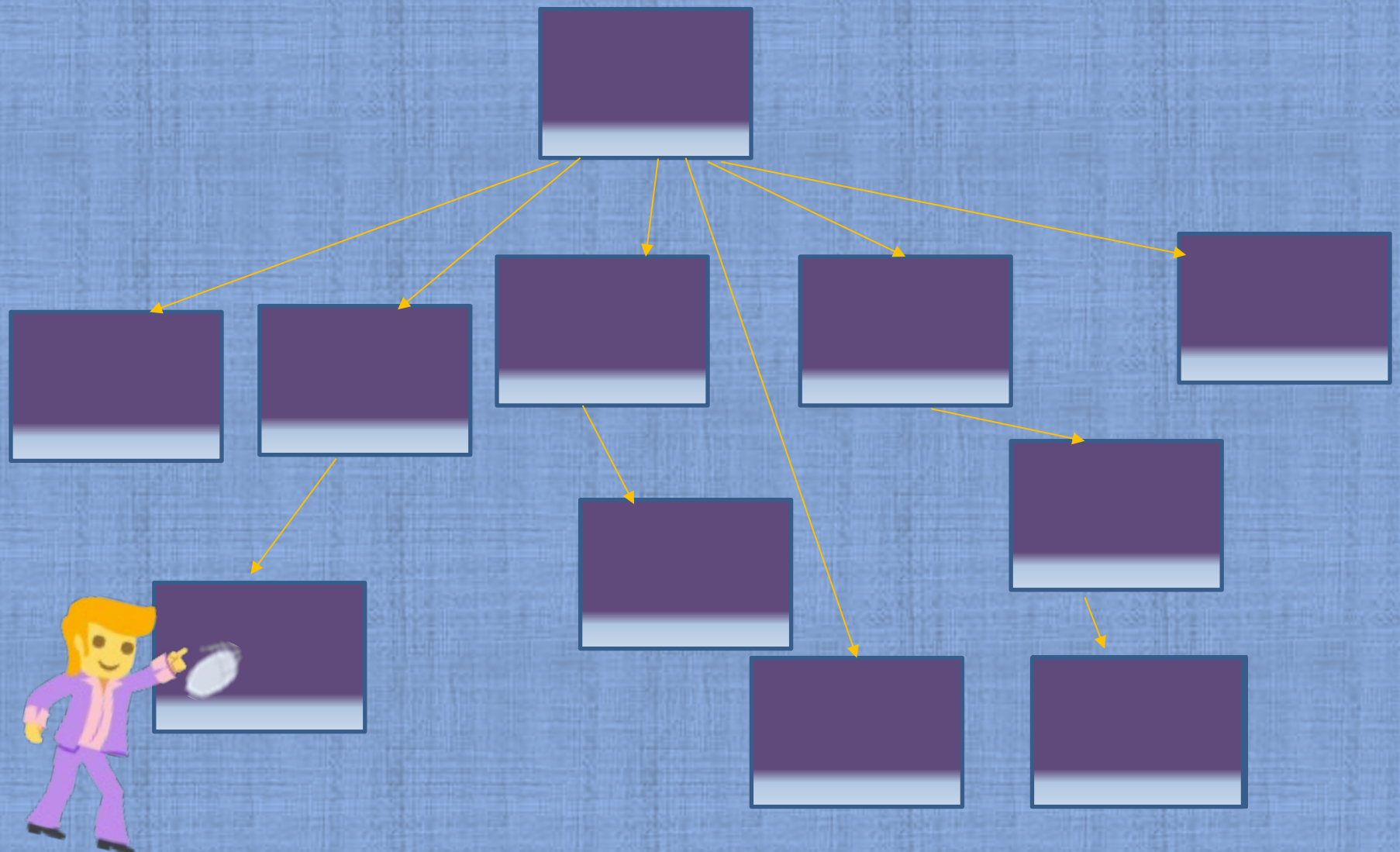
resource()

Anatomy of Angular resource()API

Reactivity
Automatic UI updates when data–model changes



Reactivity

Reactive programming approach is that where the software application

- ***responds to events***
- ***ensures view and data-model are always in synch.***

Signals

- *Reactivity*

  Signals automatically propagate changes to the parts of the application that depend on them, ensuring seamless updates to UI

- *Granular State Tracking*

  They track how and where the state is used, allowing Angular to optimize rendering updates.

Signals

Reactive State Management

- *Signals are **synchronous** by nature,*
- *Work well for managing state that is **immediately available or computed**.*

Leveraging reactivity of Signals

- Signals provide reactivity, however they provide ***synchronous state updates.***

- But, what about adding reactivity to ***Asynchronous state management?***

Async code

Our applications usually fetch data from backend services in an asynchronous way using either

Promise

or

RxJs Observables

Async code

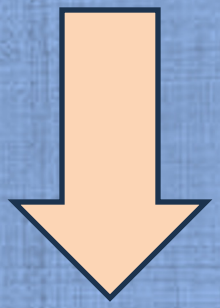
How to make this async code more REACTIVE

Leveraging signals

Can we *integrate signals* with ***async*** operations?

& add more

reactivity to async code !!!



Resource API()

Async state management

- *Promises*

- *RxJs Observables*

Angular 19, introduces on experimental basis

- *Resource API that leverages signals*

Async code writing before resource API

1. Make API calls to fetch data from server
2. Subscribe to observables in component constructor
3. Create signal property to hold the data fetched from the server.
4. Show loading indicator by creating a boolean property that will toggle

Async code before resource api

5. Handle the situation when multiple requests are made simultaneously, then either cancel the previous request or ignore the new request (write imperative code for this)

6. Write a method to reload the data on button click

7. End the subscription when component is destroyed

Thus, We often end up writing similar repetitive code steps.

New resource API

- It addresses all those issues in an elegant way
- Integrates Signals with async code.
- Enables writing more reactive code.

Handling of async data

With Promise	With Resource api
State :Pending	
<div *ngIf="*isLoading*"> <p> Loading, please wait..<p> class MyComp{ *isLoading* : boolean true; …….. …….	***isLoading()***

Handling of asynch data

With Promise	With Resource api
State : Fulfilled	

<div *ngIf="*!isLoading*">
  <p> {{ data }}.<p>
 	
Class MyComp {
.....
  *promise.**then**((result) => {*
  this.data = result…
  })
……
	value()

Handling of asynch data

With Promise	With Resource api
state : rejected / error	

.promise.then.(..)
.catch((error) =>
{
console.error(error); // Logs "Operation failed!" if rejected
})
	error()

Handling of async data reactively

With Promise	With Resource api
More boilerplate	More reactive
	

Key Benefits of Their Integration

- *Signals and the Resource API work together to provide a consistent reactive programming model for both sync and async data.*
- *Simplified Code: The Resource API reduces **boilerplate** code for handling async states, making it easier to manage complex data flows.*
- *Improved Performance: By leveraging signals, Angular ensures efficient updates to the UI, avoiding unnecessary re-renders.*
- *the Resource API bridges the gap between Angular's signal-based reactivity and the need to handle asynchronous data*

resource() API

***Resource api** automatically tracks*

- *the state of the async operation*

&

- *updates the UI reactively when the data is fetched or changed.*

What is Resource?

As per angular documentation,

A Resource is an asynchronous dependency (*for example, the results of an API call*) , that is managed and delivered through signals.

[It] projects a reactive request to an asynchronous operation defined by a loader function, which exposes the result of the loading operation via signals.

Resource

```
import { resource } from '@angular/core';
// ...

myResource = resource({
  loader: () => /* load data */
});
```

loader must return *a Promise*

Resource api key components loader()

Loader function fetches the data asynchronously and updates resource's value.

For ex.

```
loader: async () => {
        const res = await fetch(this.apiUrl);
        const data = await res.json();
        this.weatherData.set(data);
        return this.weatherData();
}
```

loader function

```
export class HelloWorld {

title = signal('Hello World Component');
simpleRef=resource({
        loader: ()=> {
       return new Promise((resolve)=>{
      setTimeout(()=>{
      resolve("Hello world");
       },2000)
    })
   },

});
```

loader function

```
export class User {

data={'id':'','name':'' }

  userRef =  resource({

            loader : ()=>{
            return fetch('https://jsonplaceholder.typicode.com/users/1’)
            .then((res) => res.json())
            .then((user) => { this.data = user; console.log('user', this.data); })
            .catch((err) => { console.log(err); });
}
})
```

template

```
<ng-container *ngIf="userRef.hasValue(); else
                                    loadingOrError">
    <h3>{{ data.name }}</h3>
  </ng-container>


<ng-template #loadingOrError>
 <ng-container *ngIf="userRef.isLoading(); else errorBlock">
<h3>User loading...</h3>
</ng-container>
</ng-template>
```

template

```
<ng-template #errorBlock>
<ng-container *ngIf="userRef.error()">
<h3>Error loading user: {{ userRef.error() }}</h3>
</ng-container>
</ng-template>
```

Resource key components
params()

Reactive Parameters :

The params function dynamically computes parameters whenever dependent signals change. For example:

```
protected userId = signal(1);
params: ()=>({ id : userId(})
```

Resource key components
params()

```
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),
});
```

Resource key components status tracking signals

The resource 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 encountered, if any,

Demo

```
export class AppComponent {
  title = 'angular-examples';

  // Signal for managing post ID state
  postId = signal(1);

  // Resource API for fetching comments
dynamically
```

Demo

```
// Resource API for fetching comments dynamically
comments = resource({
  request: () => ({ postId: this.postId() }),
  loader: ({ request }) =>

fetch(`https://jsonplaceholder.typicode.com/comments?postId=${request.postId}`)
      .then((response) => {
        if (!response.ok) throw new Error('Failed to fetch comments');
        return response.json() as Promise<Comment[]>;
      }),
});
```

rxResource

The applications using observables for async data management , Angular provides a counter-part of resource : rxResource()

The loader function returns data as observables

It seamlessly connects signals to observables, enabling a more reactive approach to data fetching.

httpResource

- It is a ***reactive*** wrapper over HttpClient.
- It provides request status and response as signals
- The data is parsed as JSON