Frontend Development with Angular

Marco Martorana, Senior Software Developer at **ELCA**

Personal: https://marcomartorana.it

LinkedIn: https://www.linkedin.com/in/marcomartorana

Marco Martorana

Professional Experience:

- Senior Software Engineer & Frontend Lead **@ELCA**
- Development on frontend side with Angular
- Development on backend side with Spring/Java
- ….

Teaching & Training

- Organizer and facilitator of training courses
- Conducted: 3 Angular Courses
- Conducted: 2 OOP Course
- …

Reach out for any questions!

ELCA
We make it work.

This talk is a retrospective of my experience as frontend lead with Angular framework a sort of Back to the Future trip.

I'm going to share my personal journey and some recommendation and lesson learnt.

The Angular framework

An open source **framework** for creating advanced web application (SPA)
Developed and maintained by Google
Ecosystem consists of **1.7 million** developers, library authors and content creators

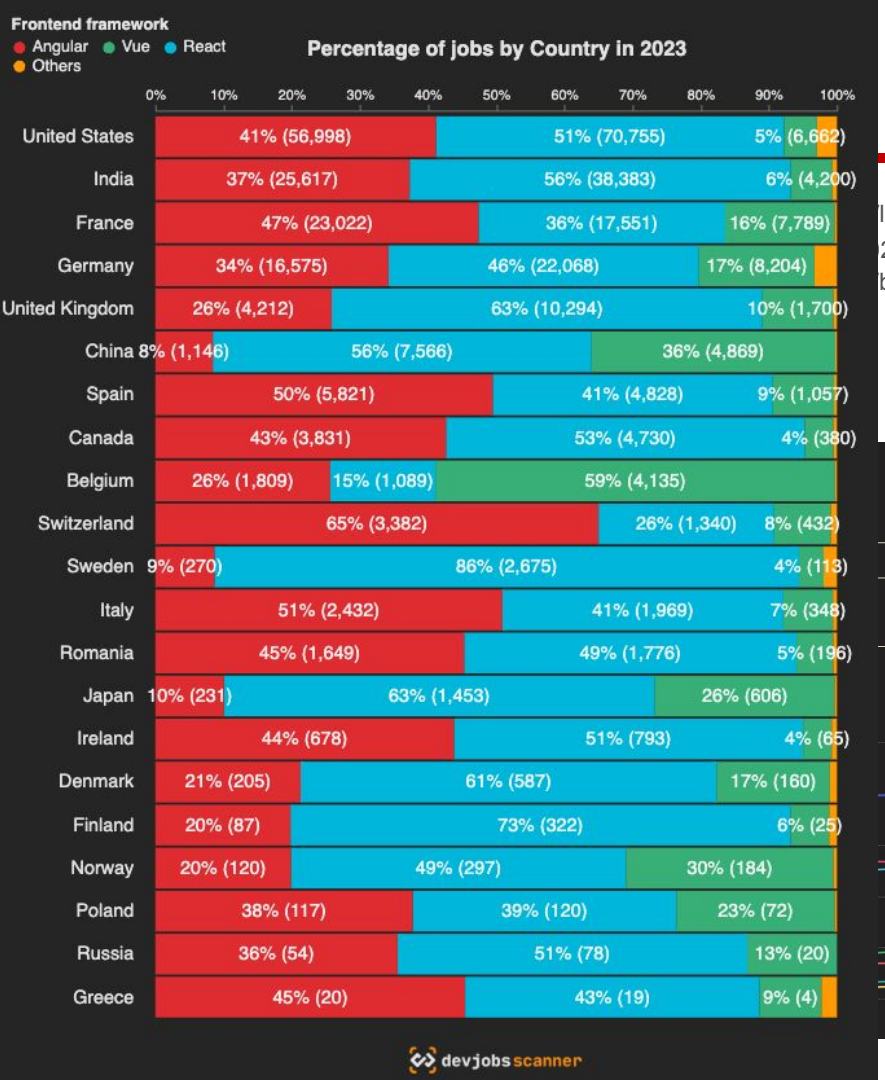
Angular JS	Angular 2	Angular 4	Angular 5	Angular 6	Angular 7	Angular 8	Angular 9	Angular 10	Angular 11	Angular 12	Angular 13	Angular 14	Angular 15	Angular 16	Angular 17	Angular 18	Angular 19
Oct 2010	Sep 2016	Mar 2017	Nov 2017	May 2018	Oct 2018	May 2019	Feb 2020	Jun 2020	Nov 2020	May 2021	Nov 2021	Jun 2022	Nov 2022	May 2023	Nov 2023	May 2024	Nov 2024

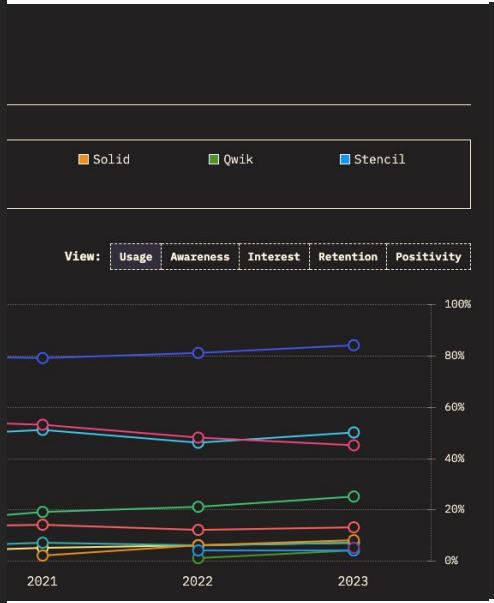
Databinding: How can update automatically my UI ?
Type Checking TS: How can improve code quality ?
Framework: How can improve maintainability of my code ?

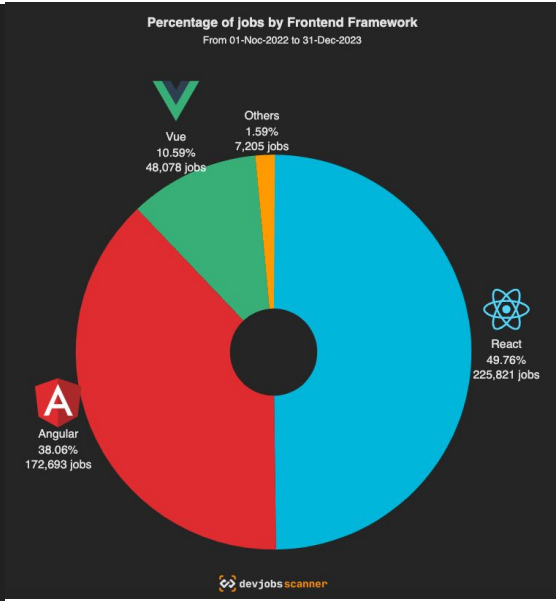
Used for frontend development

Library Popularity

libraries/front-end-frameworks
23/#most-popular-technologies-webframe-prof
blog/the-most-demanded-frontend-frameworks







Back to the Angular

The Frontend Ecosystem @2006/7

Areas for Improvement

- Manual Synchronization
- Boilerplate Code
- Code Unreadability
- Slower Development
- Tight Coupling

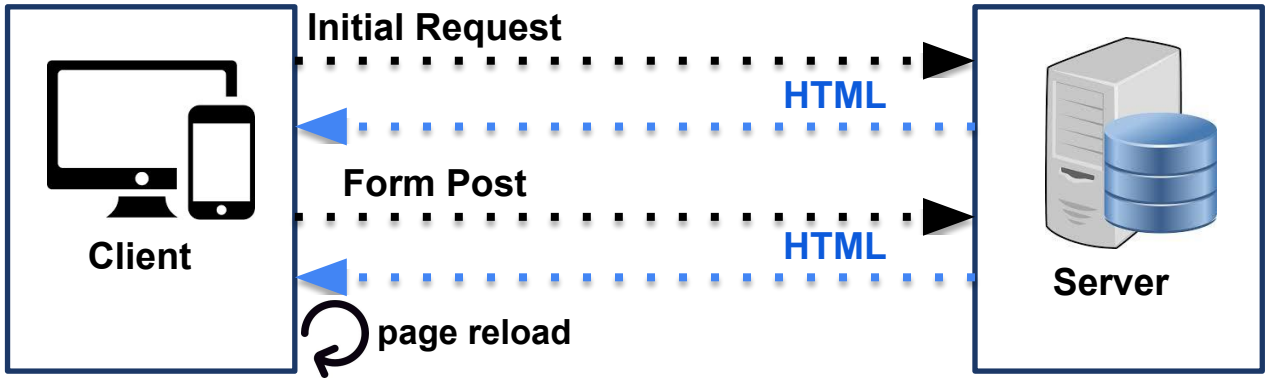
1995

2006

Server Rendering

Traditional Web Application

UI generated by server ex. with PHP, JSP, JSF ...



The I Revolution of Angular @2010

AngularJS

jQuery

JS

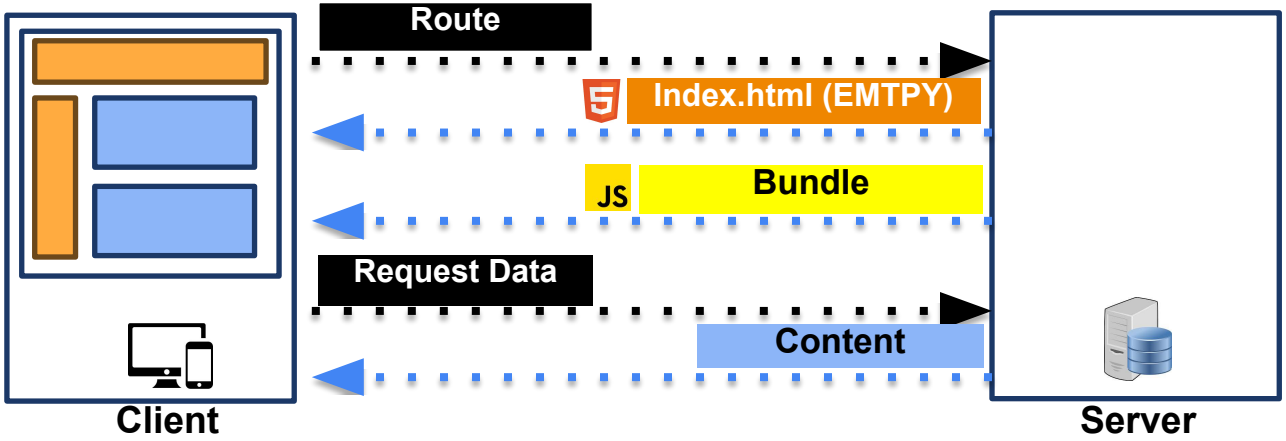
Binding

1995 2006 2010

ClientSide Rendering

Single Page Application

UI updated by client ex. Ajax, Angular



The Frontend Ecosystem @2011/12

RxJS

TypeScript

AngularJS

jQuery

JS

1995 2006 2010 2011 2012

The II Revolution of Angular @2016

- Rewritten in **Typescript**
- Integrate Reactive Programming **RxJS**
- …..

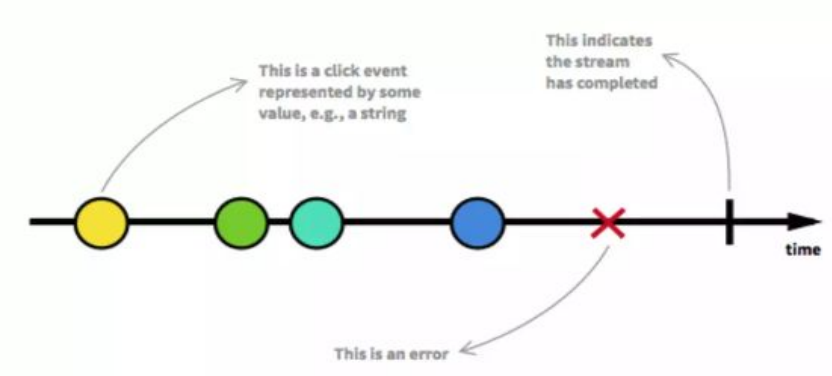
Reactive Programming

- Imperative Programming
- Functional Programming
- Reactive Programming

Reactive programming is an *asynchronous programming paradigm* concerned with **data streams** or event streams and the *propagation of change*.

Languages

- Java: RxJava
- JavaScript: RxJS
- C#: Rx.NET
- C#(Unity): UniRx
- Scala: RxScala
- Clojure: RxClojure
- C++: RxCpp
- Lua: RxLua
- Ruby: Rx.rb
- Python: RxPY
- Go: RxGo
- Groovy: RxGroovy
- JRuby: RxJRuby
- Kotlin: RxKotlin
- Swift: RxSwift
- PHP: RxPHP
- Elixir: reaxive
- Dart: RxDart



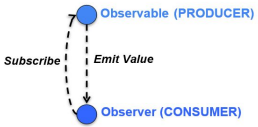
Scenario one of **Data Manipulation**

```
constructor(private http: HttpClient) { }

ngOnInit(): void {
  this.cityInput.valueChanges
    .pipe(
      filter(text => text.length > 2),
      debounceTime(1000),
      distinctUntilChanged(),
      switchMap(
        text => this.http.get(`http://api.openweathermap.org/data/2.5/weather?q=${text}&units=metric&APPID=XXX`)
          .pipe(
            map((meteo: any) => {
              return {
                temperature: meteo.main.temp,
                icon: 'http://openweathermap.org/img/w/' + meteo.weather[0].icon + '.png',
                map: 'https://maps.googleapis.com/maps/api/staticmap?center=' + text + '&zoom=7&size=200x100&key=XXX',
                error: false
              };
            }),
            catchError(() => of({ error: true}))
          )
      ),
    )
    .subscribe(result => {
      console.log(result);
      this.meteo = result;
    });

  this.cityInput.setValue('palermo');
```

Observable

Operators

Observer



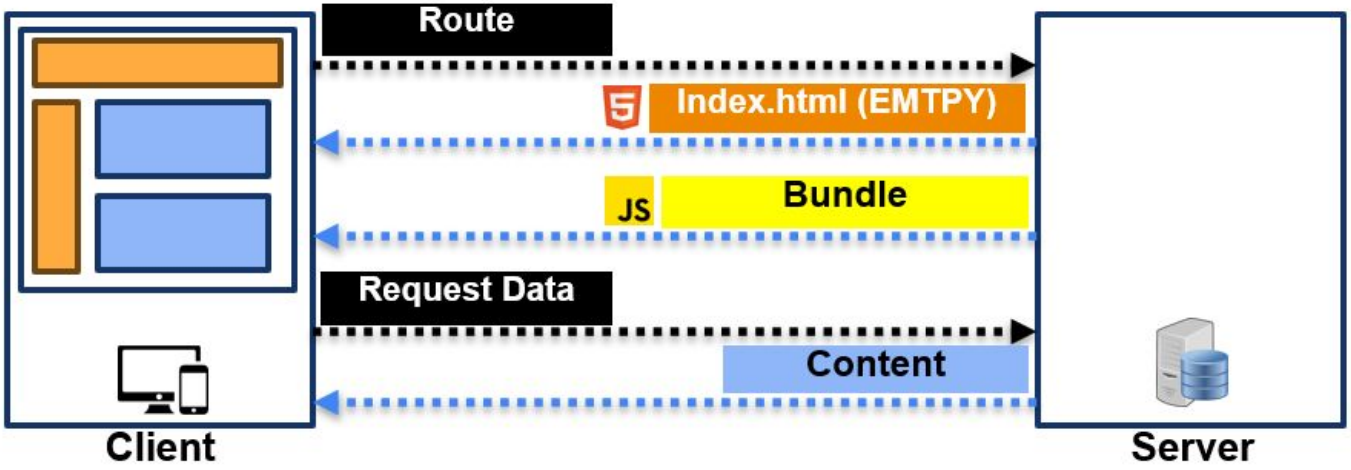
ClientSide Rendering

Single Page Application

UI updated by client ex. Ajax, Angular

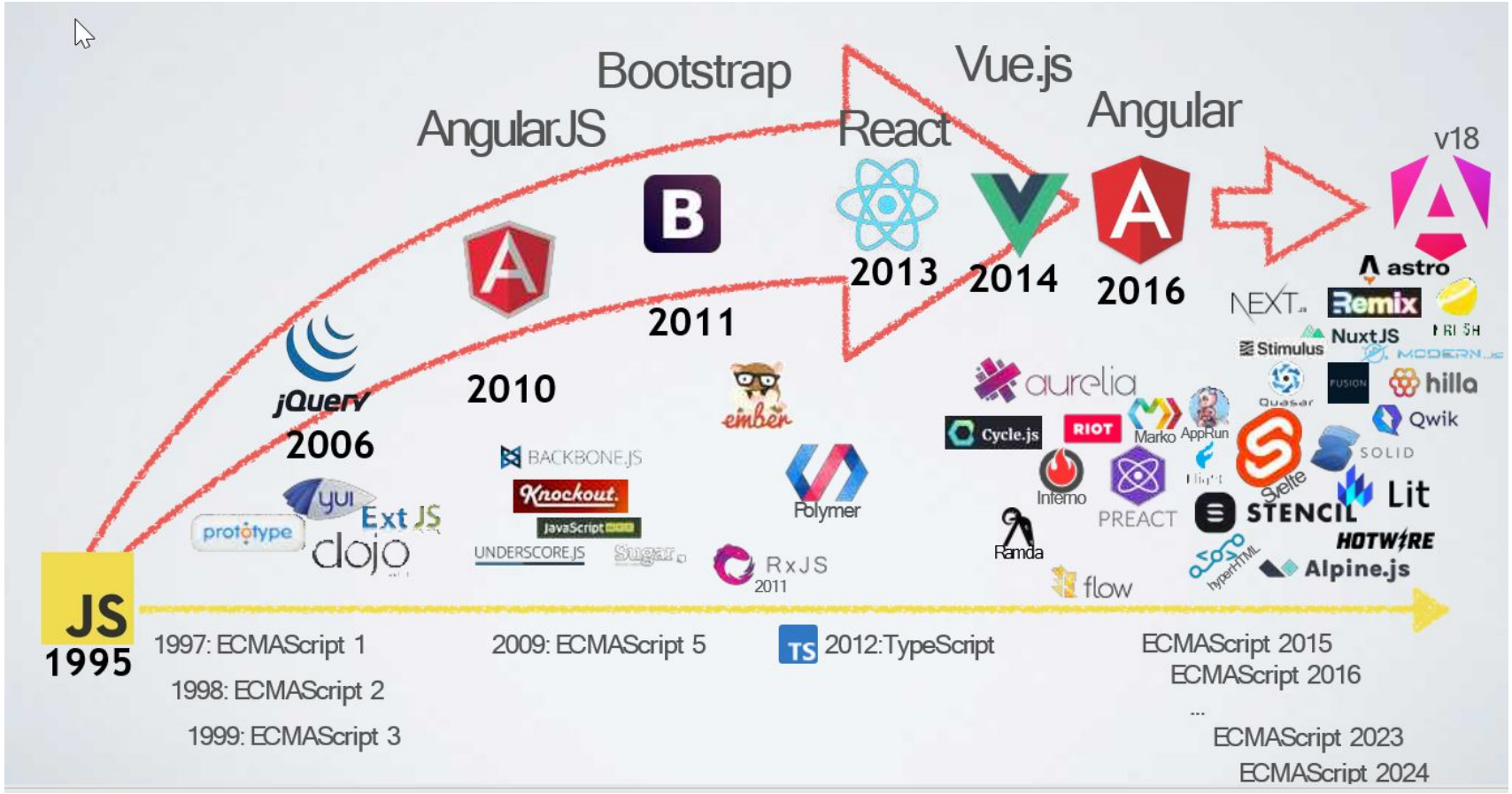
SPA: Single Page Architecture (it is almost used)

.....



But frontend is still evolving…

Evolution of Frontend ecosystem



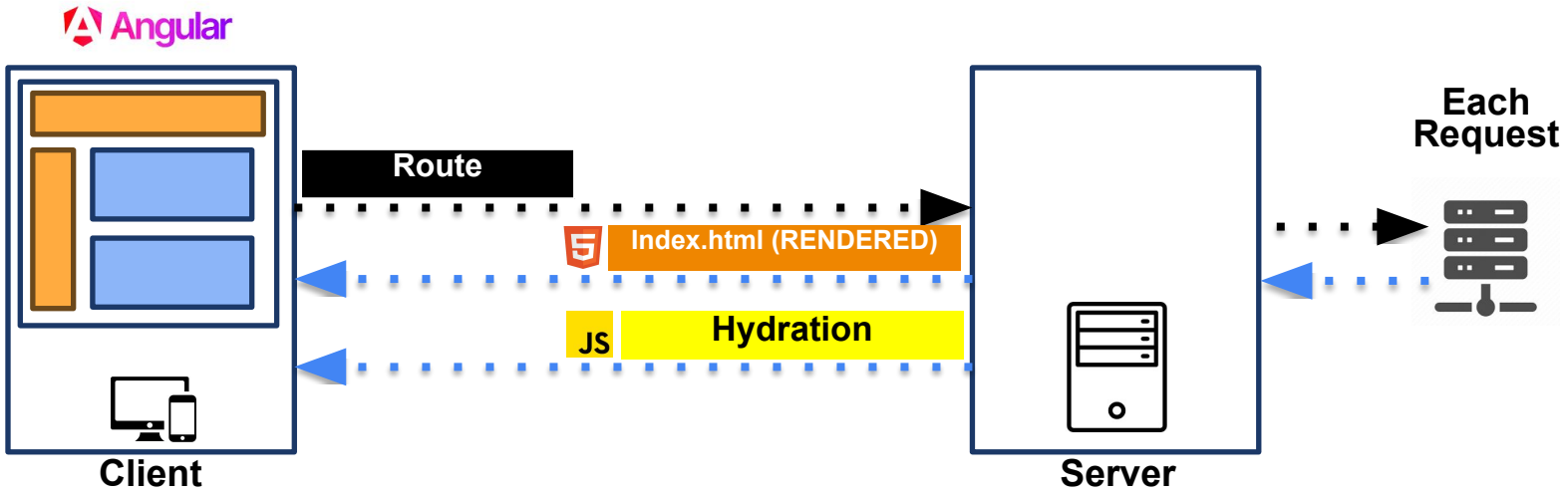
The III Revolution of Angular - Modern Angular

Standalone: Simplification of projects (modules)
Flow Control: Changing of flow control
Hydration: Improving initial page load performance
Server-Side Rendering (SSR): Introduction and enhancement
@defer: Lazy loading
Signals: A new approach to reactivity
…..

Server Side Rendering

SSR - Modern Applications

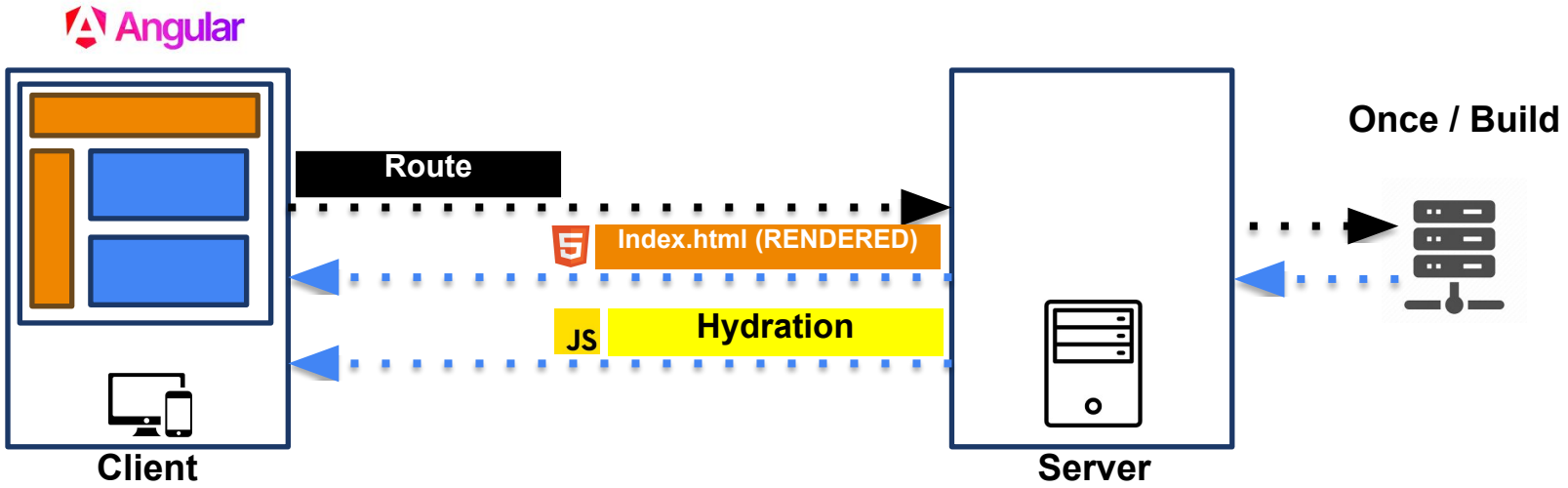
- ✔ Server-side rendering (**SSR**) is the process of rendering web pages on the server and sending the **fully rendered HTML** to the client.
- ✔ The **server generates the HTML**, including any **dynamic data**, and sends it to the client as a complete page **for each request of the user**.



Static Side Generation

SSG - Modern Applications

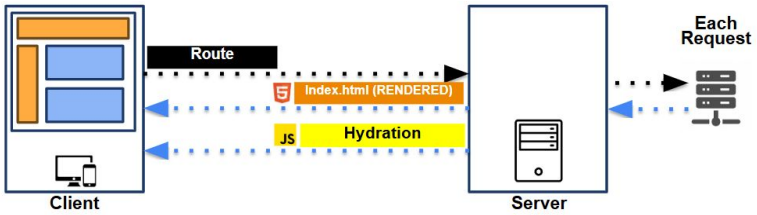
- ✔ **Static Site Generation (SSG)** is a process where web pages are **pre-rendered as static HTML** files during the **build time**, rather than being generated dynamically on the server or client.
- ✔ **Performance Boost**: **SSG** improves **load times** and **user experience** by serving pre-rendered pages through a CDN, enabling near-instant delivery.



SSR vs SSG in Angular

SSR in Angular

Pages are generated on the server **for each user request**.
It's good for dynamic, real-time content.
Supports various backend technologies (**Node**.js, .NET, Java, etc)



Advantages

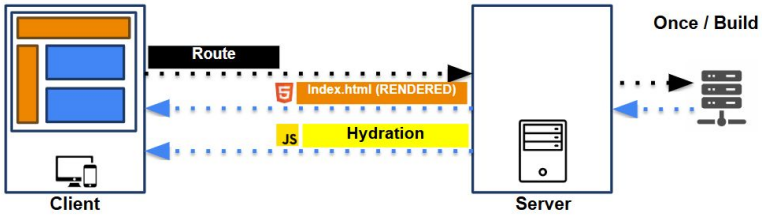
- Seo-friendly
- **Real time update**
- Interactive user experience

Disadvantages

- Higher sever load
- Slower initial load times

SSG in Angular

Pages are generated **once at build time** and served static files.
It's ideal for fast, static content.
Supports various backend technologies (**Node**.js, .NET, Java, etc)



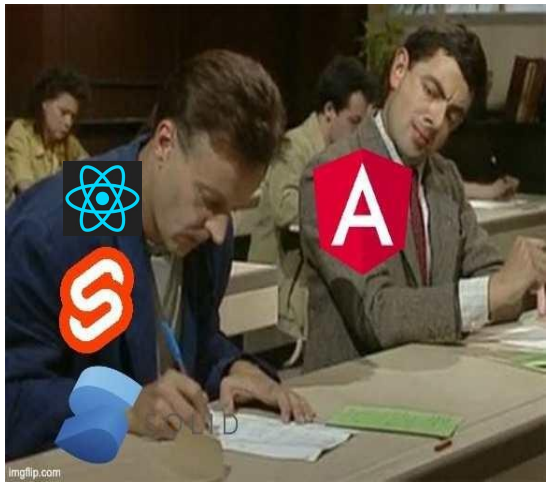
Advantages

- Excellent SEO
- **Extremely fast load times**
- Interactive user experience

Disadvantages

- Less dynamic
- Rebuild required for updates

The Angular Renaissance - modern Angular



The Angular Renaissance

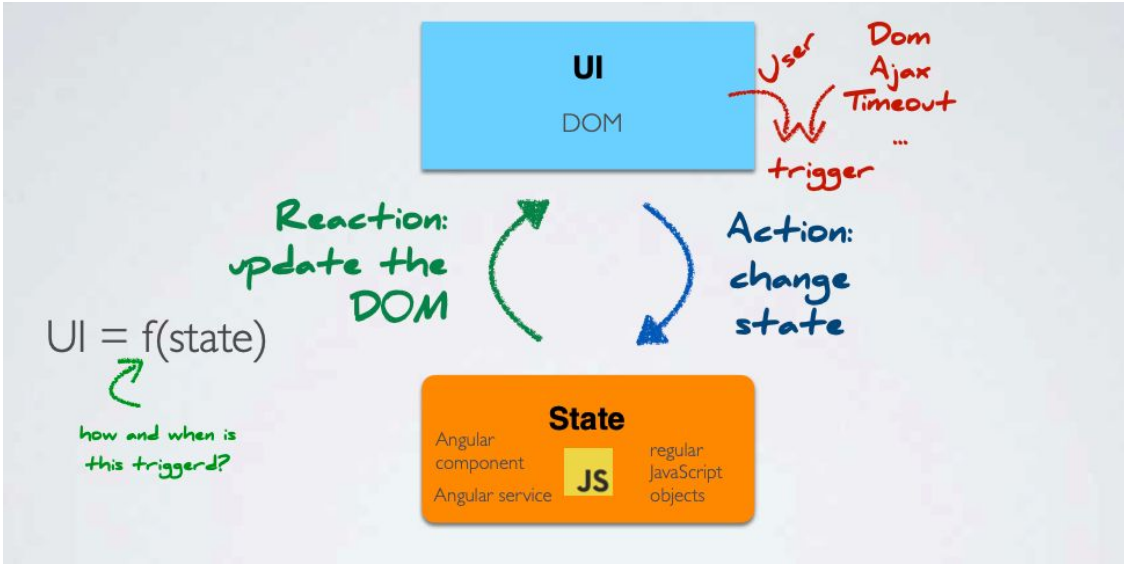
https://angular.dev/

Standalone Components	introduced in v14	Simplifying the structure of Angular projects.	Getting rid of `NgModule`.
`inject()` function	introduced in v14	New mechanism for dependency injection.	No more constructor injection. "Decoupling" DI from class syntax.
Control Flow	introduced in v17	New template syntax: `@if`, `@for` ...	Getting rid of structural directives `*ngIf`, `*ngFor`
Signals	developer preview, introduced in v16	New primitives for modeling reactive state. Eventually enabling "fine- grained" reactivity.	State is explicitely modeled with Angular constructs. It is no longer "just JavaScript".
Making RxJS optional	future	Avoiding the complexity of RxJS for simple scenarios.	Offering alternatives to promise-based APIs.

https://angular.dev/roadmap#explore-modern-angular

Signals in Angular - Introduction

State is Managed in **JavaScript**
Modern frontend frameworks are "**reactive**"



Reactivity: Angular "reacts" on state changes and updates the UI

Signals in Angular - Definition

- ✔ Introduced in Angular 16, now stable
- ✔ Simplifies state management and change detection
- ✔ More intuitive and performant than traditional observables

What is a Signal ?

Signals are wrappers around values that notify consumers when they change.

- **Writable** signals: Can be directly updated
- **Computed** signals: Derive their value from other signals

```
@Component({
   template: `<div>fullName()</div>`
})
export class App {
  firstName = signal('Jane');
  lastName = signal('Doe');

  fullName = computed(() => `${this.firstName()} ${this.lastName()}`);

  constructor() {
    effect(() => console.log('Name changed:', this.fullName()));
  }

...
```

```
signal.set(newVal);
signal.update(val => newVal);
```

```
const signal = signal.asReadOnly();
```

Signals for state

- ❑ It's easy to model state with JavaScript class properties.
- ❑ It is "hands-off" approach to reactivity
- ❑ In "modern Angular" it is recommended to model state explicitly with **Signals**.

This is the new reactivity system of Angular.

Benefits of Signals

- Improved **performance**
- **Simpler syntax** compared to RxJS for basic use cases
- **Reduced boilerplate code**

https://angular.dev/guide/signals

Signals in Angular - API

Signals API

Function	Description	Example

signal<T>()	Creates a signal.	`isLoading = signal(false);`
		`customers = signal<Customer[]>([]);`
		`<!-- Get signal value -->`
		`<div *ngIf="isLoading()">Loading...</div>`

set()	Sets a new value for the signal.	`this.isLoading.set(true);` `this.customers.set(data);`
update()	Update a signal based on its current value.	`count = signal(0);` `this.count.update(val => val + 1);`
mutate()	Mutate an object directly.	`customer = signal<Customer>({first: 'Tim', last: 'Smith'});` `this.customer.mutate(cust => cust.first = 'Tina');`
computed()	Create a new signal that depends on other signals (read-only).	`customer = signal<Customer>({first: 'Tim', last: 'Smith'});` `fullName = computed(() => `${this.customer().first}` ` ${this.customer().last}`;`
effect()	Run side effects as a signal changes	`effect(() => console.log('Customer', this.customer());`

*Component **state** with **RxJS***

An example with RxJS

See live coding….

Live coding

Component ***state*** *with* ***RxJS***

```ts
import { CommonModule } from '@angular/common';
import { Component } from '@angular/core';
import { BehaviorSubject } from 'rxjs';

@Component({
  selector: 'app-counter',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './counter.component.html',
  styleUrl: './counter.component.scss'
})
export class CounterComponent {
  private readonly countSubject = new BehaviorSubject<number>(0);
  count$ = this.countSubject.asObservable();
  parityMessage: string = 'Il numero è pari';

  increment() {
    const newValue = this.countSubject.value + 1;
    this.countSubject.next(newValue);
    this.updateParityMessage(newValue);
  }

  decrement() {
    const newValue = this.countSubject.value - 1;
    this.countSubject.next(newValue);
    this.updateParityMessage(newValue);
  }

  reset() {
    this.countSubject.next(0);
    this.updateParityMessage(0);
  }

  private updateParityMessage(value: number) {
    this.parityMessage = value % 2 === 0 ? 'Il numero è pari' : 'Il numero è dispari';
  }
}
```


```html
<div>
  <h1>Contatore: {{ count$ | async }}</h1>
  <p>{{ parityMessage }}</p>
  <button (click)="increment()">
      Incrementa
  </button>
  <button (click)="decrement()">
     Decrementa
   </button>
  <button (click)="reset()">Reset</button>
</div>
```

BehaviorSubject

- ✔ countSubject keeps track of the current state of counter
- ✔ .next(newValue) -> update the state
- ✔ .value -> read the current value
- ✔ count$ is an Observable
- ✔ updateParityMessage() method □ updates message

*Component state with **Signals***

An example with Signals

See live coding….

Live coding

Component state with **Signals**

```ts
import { CommonModule } from '@angular/common';
import { Component, signal, computed } from '@angular/core';

@Component({
  selector: 'app-counter-signal',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './signals.component.html',
  styleUrl: './signals.component.scss'
})
export class CounterComponent {
  count = signal<number>(0);

  parityMessage = computed(() =>
    this.count() % 2 === 0 ? 'Il numero è pari' : 'Il numero è dispari'
  );

  increment() {
    this.count.update(currentValue => currentValue + 1);
  }

  decrement() {
    this.count.update(currentValue => currentValue - 1);
  }

  reset() {
    this.count.set(0);
  }
}
```


```html
<div>
  <h1>Contatore: {{ count() }}</h1>
  <p>{{ parityMessage() }}</p>
  <button (click)="increment()">
    Incrementa</button>
  <button (click)="decrement()">
    Decrementa</button>
  <button (click)="reset()">Reset</button>
</div>
```

Signals

- ✔ signal keeps track of the current value
- ✔ .update(currentValue) -> update current value
- ✔ .set(currentValue) -> set current value
- ✔ computed -> performing calculation based on current value of other Signals

Comparison ***Signals VS RxJS***

Feature	Signals	RxJS (BehaviorSubject/Obs)
Simplicity	Simpler and more readable	Requires setup of Subject and Observable
Reactivity	Native	Requires manual updates using .next() or pipelines
Performance	Optimized updates	Depends on correct usage of RxJS
Template	No pipe needed (count())	| async

Why use Signals ?

- **Simplicity**: Eliminates the need for BehaviorSubject or Observable for simple states.
- **Performance**: Optimized DOM updates, as Angular knows exactly what signals to look for.
- **Native Responsiveness**: Signals are responsive by nature without any subscriptions or RxJS operators.

Pro / Cons of Reactive (RxJS)

- **Angular Thinking** Compliant with the framework's architecture RxJS
- **Scalable Data Handling** Reactive programming is well-suited for applications with complex data flows
- **Declarative Syntax** Reactive programming allows for a declarative style of programming
- **Steep Learning Curve** Reactive programming introduces concepts difficult for developers
- **Overhead for Simple Use Cases** Might introduce unnecessary complexity for simple scenarios
- **Error Handling Complexity** Handling errors can be more complex

Best Practices for Modern Angular

- Signals for simple state management
- RxJS for complex asynchronous operations
- Interoperability between signals and observables
- Choose the right tool for the job based on complexity
- Utilize SSR and hydration for improved performance
- Keep up with Angular's documentation

*What **about** next* **Future** of ***Angular*** *?*

Angular continuing to evolve with modern web development trends **Signals**, **hydration**, and **improved SSR** enhance performance and **developer experience**.
RxJS remains a powerful tool for complex asynchronous programming.
Removing dependencies from external libraries like zone.js

https://angular.dev/roadmap#explore-modern-angular

Angular 19: Key Updates and Features
Date: November 2024

Consideration *about* ***Frontend***

My ***consideration*** *about*

frontend *development !*

My Pillars / Consideration & Lesson Learned

- **KIS K**eep **I**t **S**imple may seem trivial, but it is highly relevant!
- **Embrace the platform** and write modern code
- **Tools / Languages Evaluate** and use appropriate tools and languages depending on specific goals
- **Framework / Library** Follow best practices to improve quality and avoid bad practices to prevent mistakes.
- **Quality code**: Testing (component, e2e), Clean code, Team conventions.
- **Knowledge Increase** After acquiring a solid base in OOP, HTML, CSS, JS, Git, and Docker..
- **AI** Leverage AI for trivial tasks but always be aware about generated code.
- ….
- ….

Handle **Social**

My Handle Social:

- **Personal**: https://marcomartorana.it
- Medium: https://medium.com/@marcomatto
- **LinkedIn**: https://www.linkedin.com/in/marcomartorana/
- **Youtube**: https://www.youtube.com/@MarcoMartorana