

ANGULAR

Introduzione

Cos'è Angular?

Angular è un framework per lo sviluppo di applicazioni web, open source, sviluppato principalmente da Google e da una comunità di sviluppatori individuali e aziendali, per affrontare le sfide incontrate nello sviluppo di applicazioni singola pagina. Angular è scritto in TypeScript. È possibile utilizzarlo per creare applicazioni web, applicazioni per dispositivi mobili o desktop.

Perché Angular?

Angular è un framework completo per lo sviluppo di applicazioni. È possibile utilizzare Angular per creare applicazioni web, applicazioni per dispositivi mobili o desktop. Angular fornisce funzionalità per la gestione di diversi aspetti di un'applicazione, come la gestione del routing, la gestione delle richieste HTTP, la gestione dello stato, la gestione dell'autenticazione, ecc. Angular è basato su componenti. Un componente è un'unità logica di un'applicazione. Un'applicazione Angular è un albero di componenti Angular. Angular è basato su TypeScript. TypeScript è un linguaggio di programmazione orientato agli oggetti sviluppato e mantenuto da Microsoft. È un super set di JavaScript. Angular è un framework open source. È possibile utilizzare Angular gratuitamente. È possibile utilizzare Angular per creare applicazioni web, applicazioni per dispositivi mobili o desktop. Angular

Componenti

Cos'è un componente?

Un componente è un'unità logica di un'applicazione. Un'applicazione Angular è un albero di componenti Angular. Un componente è costituito da tre parti:

- **Template:** Il template è la parte visiva di un componente. Il template contiene gli elementi HTML, le direttive e le espressioni di associazione dei dati.
- **Class:** La classe contiene la logica di un componente. La classe contiene le proprietà e i metodi di un componente.
- **Metadata:**

- Il metadata è un oggetto JavaScript che contiene le informazioni necessarie per un componente. Il metadata è decorato con un decoratore `@Component`. Il decoratore `@Component` è una funzione che accetta un oggetto JavaScript come argomento. Il metadata contiene le informazioni seguenti:
 - `selector`: Il selettore è un selettore CSS che seleziona un elemento HTML nel DOM. Il selettore è preceduto da un carattere hash (`#`). Il selettore è unico per ogni componente. Il selettore viene utilizzato per identificare un componente in un template.
 - `templateUrl`: Il `templateUrl` è il percorso del file HTML che contiene il template di un componente.
 - `styleUrls`: Lo `styleUrls` è un array di percorsi di file CSS che contengono gli stili di un componente.
 - `providers`: Il `providers` è un array di servizi che sono necessari per un componente.
 - `directives`: Il `directives` è un array di direttive che sono necessarie per un componente.

esempio:

```
import { Component } from '@angular/core'

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  providers: [MyService],
  directives: [MyDirective],
})
export class AppComponent {
  title = 'app'
}
```

Creare un componente

Per creare un componente, è necessario eseguire i passaggi seguenti:

- Creare un file TypeScript con estensione `.ts`.
- Importare il decoratore `@Component`.
- Creare una classe e applicare il decoratore `@Component` alla classe.
- Esportare la classe.
- Importare il componente in un modulo.
- Aggiungere il componente al metadata `declarations` del modulo.
- Aggiungere il selettore del componente al template di un altro componente.

- Aggiungere il tag del componente al template di un altro componente.
- Avviare l'applicazione.

Creare un componente con Angular CLI

Per creare un componente con Angular CLI, è necessario eseguire i passaggi seguenti:

- Aprire il terminale.
- Passare alla directory del progetto.
- Eseguire il comando `ng generate component component-name`.
- Il comando precedente crea una directory con il nome del componente nella directory `src/app`.
- Il comando precedente crea quattro file nella directory del componente.
 - `component-name.component.ts`: Il file contiene la classe del componente.
 - `component-name.component.html`: Il file contiene il template del componente.
 - `component-name.component.css`: Il file contiene gli stili del componente.
 - `component-name.component.spec.ts`: Il file contiene le specifiche del componente.

Direttive

Cos'è una direttiva?

Una direttiva è un'istruzione che dice a Angular di aggiungere un comportamento a un elemento HTML esistente. Angular ha tre tipi di direttive:

- **Componenti**: I componenti sono direttive con un template.
- **Direttive strutturali**: Le direttive strutturali sono direttive che modificano la struttura del DOM.
 - **ngIf**: : `ngIf` è una direttiva strutturale che aggiunge o rimuove un elemento HTML dal DOM in base a un'espressione.
 - **ngFor**: : `ngFor` è una direttiva strutturale che ripete un elemento HTML in base a un'espressione.
 - **ngSwitch**: : `ngSwitch` è una direttiva strutturale che mostra un elemento HTML in base a un'espressione.
 - **ngClass**: : `ngClass` è una direttiva strutturale che aggiunge o rimuove una classe CSS da un elemento HTML in base a un'espressione.
 - **ngStyle**: : `ngStyle` è una direttiva strutturale che aggiunge o rimuove uno stile CSS da un elemento HTML in base a un'espressione.
 - **ngTemplateOutlet**: : `ngTemplateOutlet` è una direttiva strutturale che inserisce un template in un altro template.

- **ngComponentOutlet**: : ngComponentOutlet è una direttiva strutturale che inserisce un componente in un template.
- **ngPlural**: : ngPlural è una direttiva strutturale che mostra un elemento HTML in base a un'espressione.
- **ngPluralCase**: : ngPluralCase è una direttiva strutturale che mostra un elemento HTML in base a un'espressione.
- **Direttive attributo**: Le direttive attributo sono direttive che modificano l'aspetto o il comportamento di un elemento, un componente o un'altra direttiva.

Creare una direttiva

Per creare una direttiva, è necessario eseguire i passaggi seguenti:

- Creare un file TypeScript con estensione .ts.
- Importare il decoratore @Directive.
- Creare una classe e applicare il decoratore @Directive alla classe.
- Esportare la classe.
- Importare la direttiva in un modulo.
- Aggiungere la direttiva al metadato declarations del modulo.
- Aggiungere il selettore della direttiva al template di un altro componente.
- Aggiungere il tag della direttiva al template di un altro componente.
- Avviare l'applicazione.

Creare una direttiva con Angular CLI

Per creare una direttiva con Angular CLI, è necessario eseguire i passaggi seguenti:

- Aprire il terminale.
- Passare alla directory del progetto.
- Eseguire il comando ng generate directive directive-name.
- Il comando precedente crea una directory con il nome della direttiva nella directory src/app.
- Il comando precedente crea due file nella directory della direttiva.
 - directive-name.directive.ts: Il file contiene la classe della direttiva.
 - directive-name.directive.spec.ts: Il file contiene le specifiche della direttiva.

Servizi

Cos'è un servizio?

Un servizio è una classe che contiene la logica di un'applicazione che non è legata a una vista specifica. Un servizio è un'istanza di una classe che può essere iniettata in un componente, una direttiva, un filtro o un altro servizio.

Creare un servizio

Per creare un servizio, è necessario eseguire i passaggi seguenti:

- Creare un file TypeScript con estensione .ts.
- Creare una classe.
- Esportare la classe.
- Importare il servizio in un modulo.
- Aggiungere il servizio al metadato providers del modulo.
- Iniettare il servizio in un componente, una direttiva, un filtro o un altro servizio.
- Avviare l'applicazione.

Creare un servizio con Angular CLI

Per creare un servizio con Angular CLI, è necessario eseguire i passaggi seguenti:

- Aprire il terminale.
- Passare alla directory del progetto.
- Eseguire il comando `ng generate service service-name`.
- Il comando precedente crea una directory con il nome del servizio nella directory `src/app`.
- Il comando precedente crea due file nella directory del servizio.
 - `service-name.service.ts`: Il file contiene la classe del servizio.
 - `service-name.service.spec.ts`: Il file contiene le specifiche del servizio.

Routing

Cos'è il routing?

Il routing è il meccanismo per navigare tra le pagine di un'applicazione.

Creare un'applicazione con routing

Per creare un'applicazione con routing, è necessario eseguire i passaggi seguenti:

- Creare un file TypeScript con estensione .ts.
- Importare il modulo RouterModule.
- Importare il modulo AppRoutingModuleModule.
- Importare il componente HomeComponent.
- Importare il componente AboutComponent.
- Importare il componente ContactComponent.
- Importare il componente NotFoundComponent.
- Creare un array di oggetti di rotta.
- Creare un oggetto di rotta per ogni componente.
- Creare un oggetto di rotta per la pagina 404.

Creazione progetto

- 1- Installare Node.js
- 2- Installare Angular CLI: `npm install -g @angular/cli`
- 3- Creare un progetto: `ng new nome-progetto`
- 4- Avviare il server: `ng serve --open`

Struttura progetto

- e2e: Contiene i test end-to-end.
- node_modules: Contiene i moduli Node.js.
- src: Contiene il codice sorgente dell'applicazione.
 - app: Contiene il codice sorgente dell'applicazione.
 - app.component.css: Contiene gli stili del componente radice.
 - app.component.html: Contiene il template del componente radice.
 - app.component.spec.ts: Contiene le specifiche del componente radice.
 - app.component.ts: Contiene la classe del componente radice.
 - app.module.ts: Contiene la classe del modulo radice.
 - app-routing.module.ts: Contiene la classe del modulo di routing.
 - assets: Contiene i file statici.
 - environments: Contiene i file di configurazione per gli ambienti di sviluppo e di produzione.
 - browserslist: Contiene la lista dei browser supportati.
 - favicon.ico: Contiene l'icona dell'applicazione.

- index.html: Contiene il file HTML principale.
- karma.conf.js: Contiene la configurazione per Karma.
- main.ts: Contiene il file TypeScript principale.
- polyfills.ts: Contiene i polyfills.
- styles.css: Contiene gli stili globali.
- test.ts: Contiene la configurazione per i test.
- tsconfig.app.json: Contiene la configurazione per il compilatore TypeScript.
- tsconfig.json: Contiene la configurazione per il compilatore TypeScript.
- tsconfig.spec.json: Contiene la configurazione per il compilatore TypeScript.
- tslint.json: Contiene la configurazione per TSLint.

event from CHILD to PARENT

GitHub Copilot: In Angular, you can use `@Output` and `EventEmitter` to emit events from a child component to a parent component.

Here's a step-by-step guide:

1. In your **child** component, import `Output` and `EventEmitter` from `@angular/core`, and declare a new event with `@Output()` decorator:

```
import { Component, Output, EventEmitter } from '@angular/core'

@Component({
  selector: 'app-child',
  template: `<button (click)="emitEvent()">Click me</button>`,
})
export class ChildComponent {
  @Output() myEvent = new EventEmitter<string>()

  emitEvent() {
    this.myEvent.emit('Hello from child component!')
  }
}
```

2. In your **parent** component, handle the event in your HTML:

```
<app-child (myEvent)="handleEvent($event)"></app-child>
```

3. And in your **parent** component TypeScript file, define the `handleEvent` method:

```
import { Component } from '@angular/core'

@Component({
  selector: 'app-parent',
  templateUrl: './parent.component.html',
})
export class ParentComponent {
  handleEvent(event: string) {
    console.log(event) // logs 'Hello from child component!'
  }
}
```

This will emit an event from the child component to the parent component when the button is clicked.

event from PARENT to CHILD

GitHub Copilot: In Angular, you can use `@Input` decorator to send data from a parent component to a child component.

Here's a step-by-step guide:

1. In your child component, import `Input` from `@angular/core`, and declare a new input with `@Input()` decorator:

```
import { Component, Input } from '@angular/core'

@Component({
  selector: 'app-prodotto',
  templateUrl: './prodotto.component.html',
  styleUrls: ['./prodotto.component.css'],
})
export class ProdottoComponent {
  @Input() p: Prodotto
}
```

2. In your parent component, pass the data to the child component in your HTML:

```
<app-prodotto [p]="myProduct"></app-prodotto>
```

3. And in your parent component TypeScript file, define the `myProduct` :


```

import { Component } from '@angular/core'
import { Prodotto } from 'src/app/models/prodotto.module'

@Component({
  selector: 'app-parent',
  templateUrl: './parent.component.html',
})
export class ParentComponent {
  myProduct: Prodotto = {
    id: 4,
    urlImmagine: 'gaming.png',
    categoria: 'Desktop',
    prezzo: 950,
    nome: 'Desktop Gaming',
    varianti: ['Rosso', 'Blu'],
    quantitàProdottiVenduti: 155,
    numeroReclami: 54,
  }
}

```

This will pass the `myProduct` from the parent component to the child component.

exchange data between distant components

GitHub Copilot: To send data to a distant component in Angular, you can use a shared service. Here's a step-by-step guide:

1. First, create a service. You can use Angular CLI to generate a service:

```
ng generate service shared
```

2. In your `shared.service.ts`, create a private `BehaviorSubject` and a public `Observable`. The `BehaviorSubject` holds the current value and the `Observable` allows other components to subscribe to the value changes:

```

import { Injectable } from '@angular/core'
import { BehaviorSubject, Observable } from 'rxjs'

@Injectable({
  providedIn: 'root',
})
export class SharedService {
  private data = new BehaviorSubject<any>(null)
  public data$: Observable<any> = this.data.asObservable()

  updateData(value: any) {
    this.data.next(value)
  }
}

```

3. In your sender component, inject the `SharedService` and use the `updateData` method to update the data:

```

import { SharedService } from '../shared.service';

// ...

constructor(private sharedService: SharedService) { }

// ...

sendData() {
  this.sharedService.updateData(this.data);
}

```

4. In your receiver component, inject the `SharedService` and subscribe to the `data$` Observable to get the updated data:

```
import { SharedService } from '../shared.service';

// ...

constructor(private sharedService: SharedService) {
  this.sharedService.data$.subscribe(data => {
    this.data = data;
  });
}
```

This will allow you to send data from one component to another, no matter how distant they are.

Observable

GitHub Copilot: In Angular, you can use `Observable` to handle asynchronous data. They are part of the 'rxjs' library, embraced by Angular to deal with asynchronous data. Here's a step-by-step guide:

1. First, import `Observable` from `rxjs` :

```
import { Observable } from 'rxjs'
```

2. Then, create an `Observable` :

```
const observable = new Observable((observer) => {
  observer.next('Hello')
  observer.next('World')
  observer.complete()
})
```

3. Finally, subscribe to the `Observable` to get the data:

```
observable.subscribe({
  next: (value) => console.log(value),
  complete: () => console.log('Complete!'),
})
```

This will log `Hello` and `World` to the console.

Subject

It's a special type of Observable that allows values to be multicasted to many Observers. While plain Observables are unicast (each subscribed Observer owns an independent execution of the Observable), Subjects are multicast.

A Subject is like an Observable, but can multicast to many Observers. Subjects are like EventEmitters: they maintain a registry of many listeners.

Every Subject is an Observable and an Observer. You can subscribe to a Subject, and you can call next to feed values as well as error and complete. (<https://rxjs.dev/guide/subject>)

example:

```
import { Subject } from 'rxjs'

const subject = new Subject<number>()

subject.subscribe({
  next: (v) => console.log(`observerA: ${v}`),
})
subject.subscribe({
  next: (v) => console.log(`observerB: ${v}`),
})

subject.next(1)
subject.next(2)

// Logs:
// observerA: 1
// observerB: 1
// observerA: 2
// observerB: 2
```

BehaviorSubject

It's a type of Subject that requires an initial value and emits its current value whenever it is subscribed to.

example:

```
import { BehaviorSubject } from 'rxjs'

const subject = new BehaviorSubject(0) // 0 is the initial value

subject.subscribe({
  next: (v) => console.log(`observerA: ${v}`),
  error: (err) => console.log(`observerA: ${err}`),
})

subject.next(1)
subject.next(2)

subject.subscribe({
  next: (v) => console.log(`observerB: ${v}`),
  error: (err) => console.log(`observerB: ${err}`),
})

subject.next(3)
```

OPERATORI (rxjs: <https://rxjs.dev/guide/operators>)

They are functions that are used to create Observables. They are used to create Observables from things such as events, timers, promises, etc.

In order to understand how they work, see this website: <https://rxmarbles.com/>

GitHub Copilot: # RxJS Operators in Angular

Here are the operators you asked for:

1. Map

The `map` operator applies a given function to each item emitted by the source Observable and emits the resulting values as an Observable.

```
import { from } from 'rxjs'
import { map } from 'rxjs/operators'

const numbers = from([1, 2, 3])
const squareNumbers = numbers.pipe(map((val) => val * val))
squareNumbers.subscribe((val) => console.log(val)) // Outputs: 1, 4, 9
```

2. Filter

The `filter` operator is used to filter items emitted by the source Observable, only emitting those that satisfy a specified condition.

```
import { from } from 'rxjs'
import { filter } from 'rxjs/operators'

const numbers = from([1, 2, 3, 4, 5])
const evenNumbers = numbers.pipe(filter((val) => val % 2 === 0))
evenNumbers.subscribe((val) => console.log(val)) // Outputs: 2, 4
```

3. SwitchMap

The `switchMap` operator maps each value to an Observable, then it flattens all of these inner Observables using `switch`.

```
import { from, of } from 'rxjs'
import { switchMap } from 'rxjs/operators'

const letters = from(['a', 'b', 'c'])
const result = letters.pipe(
  switchMap((x) => of(1, 2, 3).pipe(map((i) => x + i)))
)
result.subscribe((val) => console.log(val)) // Outputs: a1, a2, a3, b1, b2, b3, c1, c2, c3
```

4. ConcatMap

The `concatMap` operator projects each source value to an Observable which is merged in the output Observable, in a serialized fashion waiting for each one to complete before merging the next.

```
import { of } from 'rxjs'
import { concatMap, delay } from 'rxjs/operators'

const source = of(1, 2, 3)
const example = source.pipe(concatMap((val) => of(val).pipe(delay(2000))))
example.subscribe((val) => console.log(val)) // Outputs: 1, 2, 3 with a delay of 2 seconds between
```

5. CombineLatest

The `combineLatest` operator combines multiple Observables to create an Observable whose values are calculated from the latest values of each of its input Observables.

```
import { combineLatest, of } from 'rxjs'

const first = of(1, 2, 3)
const second = of(4, 5, 6)
const combined = combineLatest([first, second])
combined.subscribe((val) => console.log(val)) // Outputs: [3, 4], [3, 5], [3, 6]
```

6. StartWith

The `startWith` operator returns an Observable that emits the items you specify as arguments before it begins to emit items emitted by the source Observable.

```
import { of } from 'rxjs'
import { startWith } from 'rxjs/operators'

const numbers = of(1, 2, 3)
const startWithZero = numbers.pipe(startWith(0))
startWithZero.subscribe((val) => console.log(val)) // Outputs: 0, 1, 2, 3
```

7. DistinctUntilChanged

The `distinctUntilChanged` operator only emits when the current value is different than the last.

```
import { of } from 'rxjs'
import { distinctUntilChanged } from 'rxjs/operators'

const numbers = of(1, 1, 2, 2, 3, 3)
const distinctNumbers = numbers.pipe(distinctUntilChanged())
distinctNumbers.subscribe((val) => console.log(val)) // Outputs: 1, 2, 3
```

8. DebounceTime

The `debounceTime` operator only emits a value from the source Observable after a particular time span has passed without another source emission.

```
import { of } from 'rxjs'
import { debounceTime } from 'rxjs/operators'

const numbers = of(1, 2, 3)
const debouncedNumbers = numbers.pipe(debounceTime(1000))
debouncedNumbers.subscribe((val) => console.log(val)) // Outputs: 3 after 1 second
```

9. CatchError

The `catchError` operator catches errors on the source Observable by returning a new Observable or throwing an error.

```
import { of, throwError } from 'rxjs'
import { catchError } from 'rxjs/operators'

const source = throwError('This is an error!')
const example = source.pipe(catchError((val) => of(`I caught: ${val}`)))
example.subscribe((val) => console.log(val)) // Outputs: I caught: This is an error!
```

10. Tap

The `tap` operator is used for side effect logging and does not alter the main data stream. It can be used for actions like logging to console, logging to a file, or making API calls that do not affect the main data stream.

```
import { of } from 'rxjs'
import { tap } from 'rxjs/operators'

const numbers = of(1, 2, 3)
const tappedNumbers = numbers.pipe(
  tap((val) => console.log(`Logged Value: ${val}`))
)
tappedNumbers.subscribe((val) => console.log(`Subscribed Value: ${val}`))
// Outputs: Logged Value: 1, Subscribed Value: 1, Logged Value: 2, Subscribed Value: 2, Logged \
```

In this example, the `tap` operator is used to log the value from the source Observable without affecting the data that is emitted to the subscriber.

Angular provides two ways to work with forms: Template-driven forms and Reactive forms. Both methods allow you to capture user input and validate it, but they differ in how they do it.

FORMS

1. Template-Driven Forms

Template-driven forms are simple to use but less flexible than reactive forms. With template-driven forms, you build your form in the template HTML, and Angular creates the form control objects.

Here's how you can create a template-driven form:

```
<!-- In your component HTML file -->
<form #f="ngForm" (ngSubmit)="onSubmit(f)">
  <input type="text" [(ngModel)]="name" name="name" required />
  <button type="submit">Submit</button>
</form>
```

```
// In your component TypeScript file
import { NgForm } from '@angular/forms'

export class YourComponent {
  name: string

  onSubmit(form: NgForm) {
    console.log(form.value) // { name: 'your name' }
  }
}
```

2. Reactive Forms

Reactive forms are more robust and scalable than template-driven forms. With reactive forms, you create your form in your component class, and Angular creates the template. When angular's gonna find a form tag in the template, it will use the formGroup directive to create a FormGroup instance and bind it to the form tag. The formGroup directive binds the FormGroup instance to the form tag. The FormControlName directive binds each input field to its corresponding FormControl instance.

Here's how you can create a reactive form:

```
// In your component TypeScript file
import { FormGroup, FormControl } from '@angular/forms'

export class YourComponent {
  form: FormGroup

  constructor() {
    this.form = new FormGroup({
      name: new FormControl(''),
    })
  }

  onSubmit() {
    console.log(this.form.value) // { name: 'your name' }
    // You can also access each form control individually
    console.log(this.form.controls.name.value) // 'your name'

    // You can also reset the form
    this.form.reset()

    // you can execute some validation logic
    if (this.form.invalid) {
      // do something
    }

    // you can also check if a specific form control is invalid
    if (this.form.controls.name.invalid) {
      // do something
    }
  }
}
```

```

<!-- In your component HTML file -->
<form [formGroup]="form" (ngSubmit)="onSubmit()">
  <input type="text" formControlName="name" />
  <!-- posso dividere in fieldsets -->
  <fieldset formGroupName="address">
    <input type="text" formControlName="street" />
    <input type="text" formControlName="city" />
    <input type="text" formControlName="state" />
    <input type="text" formControlName="zip" />
  </fieldset>
  <button type="submit">Submit</button>
</form>

```

In both methods, you can use Angular's form directives to handle form validation, such as `required`, `minlength`, and `maxlength`.

Validation can be executed in two ways:

- using Validators
- using async validators

VALIDATION

GitHub Copilot: In Angular, there are several techniques for form validation:

1. **Required Fields:** You can use the `required` attribute in your HTML form field to make it mandatory.
2. **Minimum and Maximum Length:** The `minlength` and `maxlength` attributes can be used to enforce the length of the input.
3. **Pattern Matching:** The `pattern` attribute can be used to enforce a specific pattern using regular expressions.
4. **Built-in Validators:** Angular provides built-in validators for form controls such as `Validators.required`, `Validators.minLength`, `Validators.maxLength`, and `Validators.pattern`.
5. **Custom Validators:** You can create your own custom validation functions to implement complex logic.
6. **Asynchronous Validators:** These are used to perform server-side validation.

Here's an example of using built-in validators in a reactive form:

```
import { FormGroup, FormControl, Validators } from '@angular/forms'

export class YourComponent {
  form: FormGroup

  constructor() {
    this.form = new FormGroup({
      name: new FormControl('', [Validators.required, Validators.minLength(3)]),
      email: new FormControl('', [Validators.required, Validators.email]),
      age: new FormControl('', [Validators.required, Validators.min(18)]),
    })
  }
}
```

In the template, you can use the `formControlName` directive to bind each input field to its corresponding `FormControl` instance. You can then use the `invalid` and `touched` properties to provide visual feedback to the user:

```
<form [formGroup]="form">
  <input type="text" formControlName="name" />
  <div *ngIf="form.controls.name.invalid && form.controls.name.touched">
    Name is required and should be at least 3 characters long.
  </div>
  <!-- Similar code for email and age -->
  <button type="submit" [disabled]="form.invalid">Submit</button>
</form>
```

HTTP CLIENT

GitHub Copilot: # Angular HttpClientModule

The `HttpClientModule` from Angular's `@angular/common/http` package provides a powerful way to communicate with servers using the HTTP protocol.

Importing HttpClientModule

First, you need to import `HttpClientModule` into your `AppModule`.

```
import { HttpClientModule } from '@angular/common/http'

@NgModule({
  imports: [
    HttpClientModule,
    // other imports
  ],
})
export class AppModule {}
```

Making HTTP Requests

You can inject the `HttpClient` service into your components or services. Here's how to perform different types of HTTP requests:

GET Request

Retrieve data from a server using the `get()` method.

```
import { HttpClient } from '@angular/common/http';

constructor(private http: HttpClient) { }

getPosts() {
  return this.http.get('https://jsonplaceholder.typicode.com/posts');
}
```

POST Request

The `post()` method is used to send data to a server.

```
postPost() {
  return this.http.post('https://jsonplaceholder.typicode.com/posts', {
    title: 'foo',
    body: 'bar',
    userId: 1
  });
}
```

PUT Request

The `put()` method is used to update data on a server.

```
updatePost() {  
  return this.http.put('https://jsonplaceholder.typicode.com/posts/1', {  
    id: 1,  
    title: 'foo',  
    body: 'bar',  
    userId: 1  
  });  
}
```

DELETE Request

```
deletePost() {  
  return this.http.delete('https://jsonplaceholder.typicode.com/posts/1');  
}
```

Adding/Editing Headers

You can add or edit HTTP headers by using the `HttpHeaders` class.

```
import { HttpClient, HttpHeaders } from '@angular/common/http';  
  
const httpOptions = {  
  headers: new HttpHeaders({  
    'Content-Type': 'application/json',  
    'Authorization': 'my-auth-token',  
    'CORS-Enabled': 'true', // CORS is enabled by default  
  })  
};  
  
constructor(private http: HttpClient) { }  
  
getPosts() {  
  return this.http.get('https://jsonplaceholder.typicode.com/posts', httpOptions);  
}
```

Handling Errors

You can handle HTTP errors by using the `catchError` operator from RxJS.

```
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { catchError } from 'rxjs/operators';
import { throwError } from 'rxjs';

constructor(private http: HttpClient) { }

getPosts() {
  return this.http.get('https://jsonplaceholder.typicode.com/posts', httpOptions)
    .pipe(
      catchError(error => {
        // handle your errors here
        console.error(error);
        return throwError('Something went wrong!');
      })
    );
}
```

In this example, if the GET request fails, the `catchError` operator catches the error and passes it to the error handling function. The function logs the error to the console and returns an Observable that emits an error message.

Angular Route Guards

Route guards in Angular are interfaces which can tell the router whether or not it should allow navigation to a requested route. They make this decision by looking for a true or false return value from certain methods. There are five different types of route guards:

1. **CanActivate**: Checks to see if a user can visit a route.
2. **CanActivateChild**: Checks to see if a user can visit a route's children.
3. **CanDeactivate**: Checks to see if a user can exit a route.
4. **Resolve**: Performs route data retrieval before route activation.
5. **CanLoad**: Checks to see if a user can route to a module that lazy loads.

CanActivate

This guard decides if a route can be activated. This can be used to prevent users from accessing a route if they are not logged in.

```
import {
  CanActivate,
  ActivatedRouteSnapshot,
  RouterStateSnapshot,
} from '@angular/router'
import { Observable } from 'rxjs'

export class AuthGuard implements CanActivate {
  canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): Observable<boolean> | Promise<boolean> | boolean {
    return checkIfUserIsLoggedIn() // your logic here
  }
}
```

usecases:

- prevent users from accessing a route if they are not logged in
- prevent users from accessing a route if they are not authorized to access it

CanActivateChild

This guard decides if child routes of a route can be activated.


```
import {
  CanActivateChild,
  ActivatedRouteSnapshot,
  RouterStateSnapshot,
} from '@angular/router'
import { Observable } from 'rxjs'

export class AuthGuard implements CanActivateChild {
  canActivateChild(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): Observable<boolean> | Promise<boolean> | boolean {
    return checkIfUserIsLoggedIn() // your logic here
  }
}
```

usecases:

- prevent users from accessing a route's children if they are not logged in
- prevent users from accessing a route's children if they are not authorized to access it

CanDeactivate

This guard decides if a route can be deactivated. This can be useful for prompting a user to save their work before they leave a page.

```
import { CanDeactivate } from '@angular/router'
import { Observable } from 'rxjs'

export class CanDeactivateGuard
  implements CanDeactivate<ComponentCanDeactivate>
{
  canDeactivate(
    component: ComponentCanDeactivate
  ): Observable<boolean> | Promise<boolean> | boolean {
    return component.canDeactivate ? component.canDeactivate() : true
  }
}
```

uses:

- prompt a user to save their work before they leave a page
- prevent users from leaving a page if they have unsaved changes

Resolve

This guard can be used to fetch data before navigating to a route.

```
import {
  Resolve,
  ActivatedRouteSnapshot,
  RouterStateSnapshot,
} from '@angular/router'
import { Observable } from 'rxjs'

export class DataResolver implements Resolve<Data> {
  resolve(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): Observable<Data> | Promise<Data> | Data {
    return fetchData() // your logic here
  }
}
```

usecases:

- fetch data before navigating to a route
- fetch data before displaying a component
- fetch data before activating a route
- fetch data before activating a child route

CanLoad

This guard decides if a module can be loaded lazily. This can be used to prevent the application from downloading and parsing the code for a module that the user is not authorized to access.

```
import { CanLoad, Route } from '@angular/router'
import { Observable } from 'rxjs'

export class AuthGuard implements CanLoad {
  canLoad(route: Route): Observable<boolean> | Promise<boolean> | boolean {
    return checkIfUserIsLoggedIn() // your logic here
  }
}
```

usecases:

- prevent the application from downloading and parsing the code for a module that the user is not authorized to access
- prevent the application from downloading and parsing the code for a module if the user is not logged in

CanMatch

This guard decides if a route can be matched. This can be used to prevent the application from matching a route if the user is not authorized to access it.

Matching a route means that the router will try to match the URL to a route in your application. If the URL matches a route, the router will navigate to that route.

```
import { CanMatch, UrlSegment } from '@angular/router'
import { Observable } from 'rxjs'

export class AuthGuard implements CanMatch {
  canMatch(
    route: UrlSegment[]
  ): Observable<boolean> | Promise<boolean> | boolean {
    return checkIfUserIsLoggedIn() // your logic here
  }
}
```

usecases:

- prevent the application from matching a route if the user is not authorized to access it
- prevent the application from matching a route if the user is not logged in

To use these guards, you need to provide them in your routing module and use them as a property in your routes configuration.

```
import { NgModule } from '@angular/core'
import { Routes, RouterModule } from '@angular/router'

const routes: Routes = [
  {
    path: 'your-path',
    component: YourComponent,
    canActivate: [AuthGuard],
    resolve: { data: DataResolver },
    // other properties
  },
]

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule],
})
export class AppRoutingModule {}
```

Route guards are a powerful feature in Angular that can help control navigation in your application based on your custom logic.

Angular Route Guards (Functional Style)

CanActivate

This guard decides if a route can be activated. This can be used to prevent users from accessing a route if they are not logged in.

```
import { ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/router'
import { Observable } from 'rxjs'

export function canActivate(
  route: ActivatedRouteSnapshot,
  state: RouterStateSnapshot
): Observable<boolean> | Promise<boolean> | boolean {
  return checkIfUserIsLoggedIn() // your logic here
}
```

CanActivateChild

This guard decides if child routes of a route can be activated.

```
import { ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/router'
import { Observable } from 'rxjs'

export function canActivateChild(
  route: ActivatedRouteSnapshot,
  state: RouterStateSnapshot
): Observable<boolean> | Promise<boolean> | boolean {
  return checkIfUserIsLoggedIn() // your logic here
}
```

CanDeactivate

This guard decides if a route can be deactivated. This can be useful for prompting a user to save their work before they leave a page.

```
import { CanDeactivate } from '@angular/router'
import { Observable } from 'rxjs'

export function canDeactivate(
  component: ComponentCanDeactivate
): Observable<boolean> | Promise<boolean> | boolean {
  return component.canDeactivate ? component.canDeactivate() : true
}
```

Resolve

This guard can be used to fetch data before navigating to a route.

```
import {
  Resolve,
  ActivatedRouteSnapshot,
  RouterStateSnapshot,
} from '@angular/router'
import { Observable } from 'rxjs'

export function resolve(
  route: ActivatedRouteSnapshot,
  state: RouterStateSnapshot
): Observable<Data> | Promise<Data> | Data {
  return fetchData() // your logic here
}
```

CanLoad

This guard decides if a module can be loaded lazily. This can be used to prevent the application from downloading and parsing the code for a module that the user is not authorized to access.

```
import { CanLoad, Route } from '@angular/router'
import { Observable } from 'rxjs'

export function canLoad(
  route: Route
): Observable<boolean> | Promise<boolean> | boolean {
  return checkIfUserIsLoggedIn() // your logic here
}
```

CanMatch

This guard decides if a route can be matched. This can be used to prevent the application from matching a route if the user is not authorized to access it.

```
import { CanMatch, UrlSegment } from '@angular/router'
import { Observable } from 'rxjs'

export function canMatch(
  route: UrlSegment[]
): Observable<boolean> | Promise<boolean> | boolean {
  return checkIfUserIsLoggedIn() // your logic here
}
```

To use these guards, you need to provide them in your routing module and use them as a property in your routes configuration.

```
import { NgModule } from '@angular/core'
import { Routes, RouterModule } from '@angular/router'

const routes: Routes = [
  {
    path: 'your-path',
    component: YourComponent,
    canActivate: [canActivate],
    resolve: { data: resolve },
    // other properties
  },
]

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule],
})
export class AppRoutingModule {}
```

Route guards are a powerful feature in Angular that can help control navigation in your application based on your custom logic.

Authentication and Authorization in Angular

Authentication and authorization are crucial parts of any application. In Angular, you can handle these using services, route guards, and HTTP interceptors.

Authentication Service

You can create an `AuthService` that handles user login, logout, and checks if a user is logged in.

```
import { Injectable } from '@angular/core'

@Injectable({
  providedIn: 'root',
})
export class AuthService {
  private isAuthenticated = false

  login() {
    this.isAuthenticated = true
  }

  logout() {
    this.isAuthenticated = false
  }

  isLoggedIn(): boolean {
    return this.isAuthenticated
  }
}
```

Authorization with Route Guards

You can use the `CanActivate` route guard to protect routes that require authentication.


```

import { Injectable } from '@angular/core'
import { CanActivate, Router } from '@angular/router'
import { AuthService } from './auth.service'

@Injectable({
  providedIn: 'root',
})
export class AuthGuard implements CanActivate {
  constructor(private authService: AuthService, private router: Router) {}

  canActivate(): boolean {
    if (!this.authService.isLoggedIn()) {
      this.router.navigate(['/login'])
      return false
    }
    return true
  }
}

```

In your routing module, you can use this guard to protect your routes.

```

const routes: Routes = [
  {
    path: 'protected',
    component: ProtectedComponent,
    canActivate: [AuthGuard],
  },
  // other routes
]

```

HTTP Interceptors for Authorization

HTTP Interceptors can be used to add authorization tokens to HTTP requests.

```

import { Injectable } from '@angular/core'
import { HttpInterceptor, HttpRequest, HttpHandler } from '@angular/common/http'

@Injectable()
export class AuthInterceptor implements HttpInterceptor {
  intercept(req: HttpRequest<any>, next: HttpHandler) {
    const authToken = this.authService.getAuthToken() // get your auth token from the AuthService
    const authReq = req.clone({
      headers: {
        Authorization: `Bearer ${authToken}`,
      },
    })
    return next.handle(authReq)
  }
}

```

Don't forget to provide this interceptor in your AppModule .

```

@NgModule({
  providers: [
    { provide: HTTP_INTERCEPTORS, useClass: AuthInterceptor, multi: true },
    // other providers
  ],
})
export class AppModule {}

```

This way, you can handle authentication and authorization in your Angular application.

(approfondimento) JWT (JSON Web Token)

JSON Web Tokens (JWT) are an open, industry standard (RFC 7519) method for representing claims securely between two parties. They are often used for authentication and information exchange.

A JWT is composed of three parts: a header, a payload, and a signature.

1. **Header:** The header typically consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA.

2. **Payload:** The payload contains the claims. Claims are statements about an entity (typically, the user) and additional data.
3. **Signature:** To create the signature part you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.

These three parts are separated by dots (.) and together form a complete JWT.

Using JWT in Angular

In Angular, you can use JWT for user authentication. When a user logs in, the server generates a JWT and sends it to the client. The client then stores the JWT (usually in local storage) and includes it in the `Authorization` header in subsequent requests to the server.

Here's an example of how you can set the JWT in the `Authorization` header using an HTTP interceptor:

```
import { Injectable } from '@angular/core'
import { HttpInterceptor, HttpRequest, HttpHandler } from '@angular/common/http'

@Injectable()
export class AuthInterceptor implements HttpInterceptor {
  intercept(req: HttpRequest<any>, next: HttpHandler) {
    const jwt = localStorage.getItem('jwt') // get your JWT from local storage
    if (jwt) {
      const authReq = req.clone({
        setHeaders: {
          Authorization: `Bearer ${jwt}`,
        },
      })
      return next.handle(authReq)
    }
    return next.handle(req)
  }
}
```

Remember to provide this interceptor in your `AppModule` .

```

@NgModule({
  providers: [
    { provide: HTTP_INTERCEPTORS, useClass: AuthInterceptor, multi: true },
    // other providers
  ],
})
export class AppModule {}

```

On the server side, the JWT is validated and if it's valid, the request is processed. If the JWT is not valid (or missing), the server returns an error response.

Lazy Loading Routes in Angular

Lazy loading is a technique that allows you to load modules only when they are needed. This can improve the performance of your application by reducing the initial load time.

Lazy Loading Routes

To lazy load a route, you need to create a separate module for that route. For example, if you have a `DashboardComponent` that you want to lazy load, you need to create a `DashboardModule` and move the `DashboardComponent` to that module.

```

// dashboard.module.ts
import { NgModule } from '@angular/core'
import { CommonModule } from '@angular/common'
import { DashboardComponent } from './dashboard.component'
import { DashboardRoutingModule } from './dashboard-routing.module'

@NgModule({
  declarations: [DashboardComponent],
  imports: [CommonModule, DashboardRoutingModule],
})
export class DashboardModule {}

```

```

// dashboard-routing.module.ts
import { NgModule } from '@angular/core'
import { Routes, RouterModule } from '@angular/router'
import { DashboardComponent } from './dashboard.component'

const routes: Routes = [
  {
    path: '',
    component: DashboardComponent,
  },
]

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule],
})
export class DashboardRoutingModule {}

// app-routing.module.ts
import { NgModule } from '@angular/core'
import { Routes, RouterModule } from '@angular/router'

const routes: Routes = [
  {
    path: 'dashboard',
    loadChildren: () =>
      import('./dashboard/dashboard.module').then((m) => m.DashboardModule),
  },
]

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule],
})
export class AppRoutingModule {}

```

In this example, the `DashboardModule` is lazy loaded when the user navigates to the `/dashboard` route.

Lazy Loading Modules

You can also lazy load modules that are not routes. For example, if you have a `SharedModule` that you want to lazy load, you need to create a `SharedModule` and move the `SharedModule` to that module.

Angular Pipes

Pipes in Angular are a way to write display-value transformations that you can declare in your HTML. A pipe takes in data as input and transforms it to a desired output.

Built-in Pipes

Angular comes with a stock of pipes such as `DatePipe`, `UpperCasePipe`, `LowerCasePipe`, `CurrencyPipe`, and `PercentPipe`.

Here are some examples:

```
<p>{{ 'hello' | uppercase }}</p>
<!-- Output: "HELLO" -->
<p>{{ 'WORLD' | lowercase }}</p>
<!-- Output: "world" -->
<p>{{ 0.14 | percent }}</p>
<!-- Output: "14%" -->
<p>{{ 123.45 | currency:'USD' }}</p>
<!-- Output: "$123.45" -->
<p>{{ dateObj | date:'short' }}</p>
<!-- Output: "1/1/20, 12:05 PM" -->
```

Custom Pipes

You can also define your own custom pipes. Here's an example of a custom pipe that reverses a string:

```
import { Pipe, PipeTransform } from '@angular/core'

@Pipe({ name: 'reverseString' })
export class ReverseStringPipe implements PipeTransform {
  transform(value: string): string {
    return value.split('').reverse().join('')
  }
}
```

You can use this pipe in your template like this:

```
<p>{{ 'hello' | reverseString }}</p>
<!-- Output: "olleh" -->
```

Remember to declare your custom pipes in your `NgModule` :

```
@NgModule({
  declarations: [
    ReverseStringPipe,
    // other declarations
  ],
})
export class AppModule {}
```

Pipes are a powerful tool in Angular that allow you to easily transform data right within your templates.

Angular Directives

Directives are a unique and powerful feature in Angular. They are classes that add additional behavior to elements in your Angular applications. There are three kinds of directives in Angular:

1. **Component Directives:** These are directives with a template. This is the most common kind of directive and what you think of when you hear "Angular component".
2. **Attribute Directives:** These change the appearance or behavior of an element, component, or another directive.
3. **Structural Directives:** These change the DOM layout by adding and removing DOM elements.

Attribute Directives

Here's an example of a custom attribute directive that changes the background color of an element:

```
import { Directive, ElementRef, Renderer2 } from '@angular/core'

@Directive({
  selector: '[appHighlight]',
})
export class HighlightDirective {
  constructor(el: ElementRef, renderer: Renderer2) {
    renderer.setStyle(el.nativeElement, 'backgroundColor', 'yellow')
  }
}
```

You can use this directive in your template like this:

```
<p appHighlight>Hello, world!</p>
```

Structural Directives

Angular comes with several built-in structural directives like `*ngIf`, `*ngFor`, and `*ngSwitch`.

Here's an example of using `*ngIf` and `*ngFor`:

```
<div *ngIf="isLoading">Loading...</div>

<ul>
  <li *ngFor="let item of items">{{ item }}</li>
</ul>
```

Remember to declare your custom directives in your `NgModule`:


```
@NgModule({
  declarations: [
    HighlightDirective,
    // other declarations
  ],
})
export class AppModule {}
```

Directives provide a way to extend the power of HTML by allowing you to create new behaviors.

Angular Store (NgRx)

NgRx Store is a state management solution for Angular applications that helps us build applications by working around predictable state containers. It's inspired by Redux, a state management library from the React ecosystem.

Core Concepts

1. **State:** The state is a plain JavaScript object that holds your application's state.
2. **Actions:** Actions are dispatched to change the state. They are plain objects that have a `type` and an optional `payload`.
3. **Reducers:** Reducers are pure functions that take the current state and an action, and return a new state.

Basic Usage

First, you need to install NgRx:

```
npm install @ngrx/store
```

Then, you define your state.

The state is a plain JavaScript object that holds your application's state.

```
interface AppState {
  counter: number
}
```

Next, you define your actions.

Actions are one of the main building blocks in NgRx. Actions express unique events that happen throughout your application. From user interaction with the page, external interaction through network requests, and direct interaction through dispatching actions.

```
import { createAction } from '@ngrx/store'

export const increment = createAction('Increment')
export const decrement = createAction('Decrement')
export const reset = createAction('Reset')
```

Then, you define your reducer.

Reducers in NgRx are responsible for handling transitions from one state to the next state in your application. Reducer functions handle these transitions by determining which action to handle based on the type.

```
import { createReducer, on } from '@ngrx/store'
import { increment, decrement, reset } from './counter.actions'

export const counterReducer = createReducer(
  0,
  on(increment, (state) => state + 1),
  on(decrement, (state) => state - 1),
  on(reset, (state) => 0)
)
```

Finally, you register your reducer in your AppModule :

```
import { StoreModule } from '@ngrx/store'
import { counterReducer } from './counter.reducer'

@NgModule({
  imports: [
    StoreModule.forRoot({ counter: counterReducer }),
    // other imports
  ],
})
export class AppModule {}
```

To interact with the store, you can use the `store.select` method to select data, and the `store.dispatch` method to dispatch actions (meaning to trigger a state change):

```
import { Store } from '@ngrx/store'
import { increment, decrement, reset } from './counter.actions'

export class CounterComponent {
  counter: Observable<number>

  constructor(private store: Store<{ counter: number }>) {
    this.counter = store.select('counter')
  }

  increment() {
    this.store.dispatch(increment())
  }

  decrement() {
    this.store.dispatch(decrement())
  }

  reset() {
    this.store.dispatch(reset())
  }
}
```

Lastly, we create an Effects class to handle side effects such as fetching data, long-running tasks that produce multiple events, and other external interactions where you don't want to muddy your components or services with handling these types of tasks.

```

import { Injectable } from '@angular/core'
import { Actions, createEffect, ofType } from '@ngrx/effects'
import { map, mergeMap } from 'rxjs/operators'
import { DataService } from './data.service'

@Injectable()
export class DataEffects {
  loadData$ = createEffect(() =>
    this.actions$.pipe(
      ofType(loadData),
      mergeMap(() =>
        this.dataService.getAll().pipe(map((data) => loadDataSuccess({ data })))
      )
    )
  )

  constructor(private actions$: Actions, private dataService: DataService) {}
}

```

In this example, the `loadData$` effect is listening for the `loadData` action, then it calls the `dataService.getAll()` method and maps the result to a new `loadDataSuccess` action.

NgRx Store provides a single, immutable data state observable, ensuring consistent, predictable behavior in your application.



Alt text

Standalone vs Non-Standalone Angular Projects

Angular projects can be structured in different ways depending on the needs of the project. One such distinction is between standalone and non-standalone projects.

Standalone Angular Projects

A standalone Angular project is a project that is self-contained. It includes all the necessary code and resources within the project itself. This type of project is typically used for small to medium-sized applications where all the code can be managed within a single project.

Advantages of standalone projects:

1. **Simplicity:** Everything is in one place, making it easier to understand and manage.
2. **Isolation:** The project is self-contained, reducing the risk of conflicts with other projects.
3. **Portability:** The project can be easily moved or deployed because it doesn't depend on external resources.

Non-Standalone Angular Projects

A non-standalone Angular project is a project that relies on external resources or services. This could be a project that is part of a larger application, or a project that uses shared libraries or microservices.

Advantages of non-standalone projects:

1. **Scalability:** The project can be part of a larger system, allowing for better scalability.
2. **Modularity:** The project can use shared libraries or services, promoting code reuse and separation of concerns.
3. **Collaboration:** Multiple teams can work on different parts of the application simultaneously.

In conclusion, the choice between a standalone and non-standalone Angular project depends on the specific needs and constraints of your project. Standalone projects are simpler and more isolated, but non-standalone projects offer better scalability and modularity.

Angular Elements

Angular Elements are Angular components packaged as custom elements, a web standard for defining new HTML elements in a framework-agnostic way.

Key Features

1. **Reusability:** Angular Elements can be used not just in Angular applications, but also in other environments like a React or Vue application, or even with just plain JavaScript.
2. **Framework Agnostic:** Once compiled, Angular Elements are just custom HTML elements and can be added anywhere HTML is used.

Basic Usage

First, you need to install the `@angular/elements` package:

```
npm install @angular/elements
```

Creating a custom Angular element using Angular Elements involves a few steps:

1. **Create an Angular Component:** This is the component that you will convert into a custom element.

```
import { Component, Input } from '@angular/core'

@Component({
  selector: 'app-hello-world',
  template: `<h1>Hello, {{ name }}!</h1>`,
})
export class HelloWorldComponent {
  @Input() name: string
}
```

2. **Convert the Component to a Custom Element:** Use the `createCustomElement` function from the `@angular/elements` package to convert the component into a custom element.

```
import { createCustomElement } from '@angular/elements'
import { HelloWorldComponent } from './hello-world.component'

const helloWorldElement = createCustomElement(HelloWorldComponent, {
  injector: this.injector,
})
customElements.define('app-hello-world', helloWorldElement)
```

3. **Use the Custom Element:** Now you can use your custom element just like any other HTML element.

```
<app-hello-world name="World"></app-hello-world>
```

4. **Add the Component to entryComponents:** In your `NgModule`, add the component to the `entryComponents` array. This is necessary because the component will be bootstrapped

dynamically.

```
@NgModule({  
  declarations: [HelloWorldComponent],  
  entryComponents: [HelloWorldComponent],  
})  
export class AppModule {}
```

That's it! You've created a custom Angular element using Angular Elements.

Angular Elements provide a powerful way to create reusable, framework-agnostic components.

In order to use this Element in a vanilla JS project, you need to add the following polyfills:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/webcomponentsjs/2.2.10/webcomponents-bundle.js"></script>  
<script src="https://cdnjs.cloudflare.com/ajax/libs/webcomponentsjs/2.2.10/custom-elements-es5-adapter.js"></script>
```

Docker: Build, Ship, and Run

Docker is an open-source platform that automates the deployment, scaling, and management of applications. It uses containerization technology to bundle an application and its dependencies into a single object.

Building Docker Images

Docker images are built from Dockerfiles. A Dockerfile is a text file that contains a list of commands that the Docker daemon runs to assemble an image. Here's a simple example of a Dockerfile:

```
# Use an official Node.js runtime as the base image
FROM node:14

# Set the working directory in the container
WORKDIR /usr/src/app

# Copy package.json and package-lock.json
COPY package*.json ./

# Install application dependencies
RUN npm install

# Copy the current directory contents into the container
COPY . .

# Make port 80 available to the world outside this container
EXPOSE 80

# Define environment variable
ENV NAME = World

# Run the app when the container launches
CMD [ "node", "server.js" ]
```

You can build the Docker image using the `docker build` command:

```
docker build -t my-app .
```

We can find a list of all the images on our system using the `docker images` command:

```
docker images
```

We should write a Dockerfile starting from the file which are most often changed, in order to take advantage of the Docker cache, cause Docker will cache the layers of the image, and if we change a file, Docker will rebuild the image from that point on (from bottom to top).

We can find a huge list of usable images on: <https://hub.docker.com/>

Shipping Docker Images

Once you've built a Docker image, you can push it to a Docker registry like Docker Hub:

```
docker push my-app
```

Running Docker Containers

You can run a Docker container from an image using the `docker run` command:

```
docker run -p 4000:80 my-app
```

Use Cases

Docker can be used in a variety of scenarios:

- Simplifying configuration: Docker lets you "write once, run anywhere."
- Code Pipeline Management: Docker can be used to build CI/CD pipelines.
- Scalability: Docker's containerization technology makes it easy to create highly distributed systems.
- Isolation: Docker ensures your applications and resources are isolated and segregated.

Best Tools

- Docker Compose: A tool for defining and running multi-container Docker applications.
- Docker Swarm: A native clustering and scheduling tool for Docker.
- Kubernetes: An open-source platform designed to automate deploying, scaling, and operating application containers.
- Portainer: A lightweight management UI which allows you to easily manage your Docker host or Swarm cluster.
- Prometheus: An open-source monitoring system with a dimensional data model, flexible query language, efficient time series database and modern alerting approach.

Remember, Docker is a powerful tool, but it's not always the best solution for every problem. It's important to understand your requirements and constraints before deciding to use Docker.

Introduction to NGINX and Docker

NGINX is a popular open-source web server and reverse proxy server. It can also be used as a load balancer and HTTP cache. NGINX is known for its high performance, stability, rich feature set, simple configuration, and low resource consumption.

When used in conjunction with Docker, NGINX can provide several benefits:

1. **Containerization:** Docker allows you to package your NGINX server with all its dependencies into a standardized unit. This makes it easy to ship and run on any platform that supports Docker.
2. **Isolation:** Docker ensures that your NGINX server runs in its own isolated environment, reducing the risk of conflicts with other applications.
3. **Scalability:** Docker makes it easy to scale your NGINX servers. You can quickly start up new containers to handle increased load.
4. **Configuration Management:** Docker allows you to version control your NGINX configuration. You can track changes, roll back if necessary, and ensure consistency across multiple environments.

Here's a basic example of a Dockerfile for an NGINX server:

```
# Use the official NGINX image as the base
FROM nginx

# Copy the configuration file
COPY nginx.conf /etc/nginx/nginx.conf

# Copy the website files
COPY www/ /usr/share/nginx/html

# Expose port 80
EXPOSE 80

# Start NGINX
CMD ["nginx", "-g", "daemon off;"]
```

You can build and run this Docker image using the `docker build` and `docker run` commands:

```
docker build -t my-nginx .
docker run -p 80:80 my-nginx
```

In this example, `nginx.conf` is your NGINX configuration file, and `www/` is the directory containing your website files. The NGINX server is configured to listen on port 80.