❤️

Angular

Tags	

Topics

1. Components: Angular applications are built using reusable components, which are self-contained units of code that have their own view and logic.
2. Templates: Components in Angular are defined using templates, which are written in HTML and define the layout and structure of the component's view.
3. Directives: Angular provides a number of built-in directives that allow you to manipulate the DOM and bind data to your templates.

4. Services: Services in Angular are classes that provide a way to share data and logic across the application.
5. Routing: Angular provides a powerful routing system that allows you to define the different routes in your application and how they should be navigated.
6. Forms: Angular provides a robust system for building forms, including support for data validation and handling of user input.
7. HTTP client: Angular provides a built-in HTTP client that makes it easy to send HTTP requests and receive responses from a server.
8. Dependency injection: Angular uses dependency injection to provide instances of services and other objects to components and other parts of the application.
9. Pipes: Angular provides a number of pipes that can be used to transform data in templates, such as formatting dates or converting numbers to strings.
10. Testing: Angular provides a number of tools and techniques for testing components, services, and other parts of the application.

Basic Angular app

Here is a basic Angular application that demonstrates some of the key concepts of the framework:

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

@Component({
  selector: 'app-root',
  template: `
    <h1>Welcome to my Angular app!</h1>
    <p>{{ message }}</p>
  `
})
export class AppComponent {
  message = 'Hello, world!';
}
```

This application defines a single component, called `AppComponent`, which is the root component of the application. The component is defined using the `@Component` decorator, which provides metadata about the component.

The `template` property of the `@Component` decorator specifies the HTML template that defines the view for the component. In this case, the template includes a heading and a paragraph element that displays a message.

The `AppComponent` class includes a `message` property, which is bound to the template using interpolation (denoted by the double curly braces). This allows the value of the `message` property to be displayed in the template.

To run this application, you'll need to configure an Angular module that imports the `AppComponent` and bootstraps it to run in the browser. You'll also need to include the Angular library in your project, either by installing it via npm or by including it from a CDN.

Separate Files

Here is an example of how you might separate the HTML template and CSS styles for an Angular component into separate files:

component.html

```
<h1>{{ title }}</h1>

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

<button (click)="addItem()">Add Item</button>
```

component.css

```
h1 {
  color: blue;
  font-size: 20px;
}

button {
  background-color: green;
  color: white;
  border: none;
  padding: 8px 16px;
  font-size: 16px;
  cursor: pointer;
}
```

To use these files in your Angular component, you can specify the `templateUrl` and `styleUrls` properties in the `@Component` decorator:

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

@Component({
  selector: 'app-my-component',
  templateUrl: './my-component.component.html',
  styleUrls: ['./my-component.component.css']
})
export class MyComponentComponent {
  title = 'My Component';
  items = ['Item 1', 'Item 2', 'Item 3'];

  addItem() {
    // Add an item to the list
  }
}
```

In this example, the `templateUrl` property specifies the path to the HTML template file, and the `styleUrls` property specifies an array of paths to the CSS style files.

Templates in Angular

In Angular, templates are written in HTML and define the layout and structure of a component's view. They are a declarative way of specifying what the component's user interface (UI) should look like and how it should behave.

Here is an example of a template for an Angular component:

```
<h1>{{ title }}</h1>

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

<button (click)="addItem()">Add Item</button>
```

This template includes a heading element that displays the value of the `title` property of the component, a list of items that is generated using the `*ngFor` directive, and a button that calls the `addItem()` method of the component when clicked.

The template uses interpolation (denoted by the double curly braces) to display the values of the `title` and `item` properties in the template. It also uses the `*ngFor` directive to loop over the `items` array and generate a list of items. Finally, it uses event binding (denoted by the parentheses) to bind the `click` event of the button to the `addItem()` method.

Templates in Angular are a powerful way to define the UI of a component and bind it to the component's data and logic. They allow you to build reusable components that can be easily reused and shared throughout your application.

Directives in Angular

In Angular, directives are classes that add behavior to elements in the DOM. They allow you to manipulate the DOM, bind data to templates, and add custom behavior to your components.

There are two types of directives in Angular: structural and attribute. Structural directives alter the layout of the DOM by adding, removing, or replacing elements. Attribute directives alter the appearance or behavior of an element.

Here is an example of a structural directive in Angular:

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

In this example, the `*ngFor` directive is used to loop over the `items` array and generate a list of items. The `*` prefix to the directive name indicates that it is a structural directive.

Here is an example of an attribute directive in Angular:

```
<input type="text" [ngModel]="name" (ngModelChange)="updateName($event)">
```

In this example, the `ngModel` directive is used to bind the value of the input field to the `name` property of the component and to listen for changes to the input value. The `[ngModel]` attribute binds the value of the `name` property to the value of the input field, and the `(ngModelChange)` event binding listens for changes to the input value and calls the `updateName()` method of the component with the new value.

Directives are a powerful feature of Angular that allow you to extend the functionality of your components and add custom behavior to your templates. They are a key part of the Angular framework and are used extensively in Angular applications.

Routing in Angular

In Angular, routing allows you to define the different routes in your application and how they should be navigated. It is a way of organizing and linking different parts of your application together and making it easy for users to move between different views and components.

Here is an example of how you might define routes in an Angular application:

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

import { HomeComponent } from './home/home.component';
import { AboutComponent } from './about/about.component';
import { ContactComponent } from './contact/contact.component';

const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'contact', component: ContactComponent }
];

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

In this example, the `routes` array defines three routes for the application:

- The first route, with a path of `''`, maps to the `HomeComponent`. This is the default route for the application, and it will be displayed when the user navigates to the root of the application.

- The second route, with a path of `'about'`, maps to the `AboutComponent`. This route will be displayed when the user navigates to `/about`.
- The third route, with a path of `'contact'`, maps to the `ContactComponent`. This route will be displayed when the user navigates to `/contact`.

To use these routes in your application, you'll need to include a `router-outlet` element in the template of your root component:

```
<router-outlet></router-outlet>
```

This element acts as a placeholder for the component that corresponds to the currently active route.

You can also use the `RouterLink` directive to create links between routes in your application:

```
<a routerLink="/about">About</a>
```

This creates a link to the `/about` route. When the user clicks the link, the router will navigate to the `AboutComponent` and display its view in the `router-outlet`.

Routing is a powerful feature of Angular that allows you to organize and link different parts of your application together and make it easy for users to navigate between different views and components. It is an essential part of building modern web applications with Angular.

Add router to root

The `router-outlet` element should be added to the template of the root component of your Angular application. This is the component that is bootstrapped when the application is launched, and it acts as the container for the rest of the application.

Here is an example of how you might add the `router-outlet` element to the root component of your Angular application:

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

@Component({
  selector: 'app-root',
  template: `
    <h1>My Angular App</h1>
    <nav>
      <a routerLink="/home">Home</a>
      <a routerLink="/about">About</a>
      <a routerLink="/contact">Contact</a>
    </nav>
    <router-outlet></router-outlet>
  `
})
export class AppComponent { }
```

In this example, the `router-outlet` element is added to the template of the `AppComponent` component, which is the root component of the application. The `router-outlet` element acts as a placeholder for the component that corresponds to the currently active route.

The template also includes a navigation menu with links to the different routes in the application. The `routerLink` directive is used to create the links, which allows the router to navigate to the corresponding component when the user clicks the link.

Pass data: Input and Output

Input

In Angular, the `@Input` decorator is used to specify that a component property can be bound to a value that is passed in from the component's parent. This allows the parent component to control the behavior and appearance of the child component.

Here is an example of how you might use the `ChildComponent` in a parent component:

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

@Component({
  selector: 'app-parent',
  template: `
    <app-child [name]="parentName"></app-child>
  `
})
export class ParentComponent {
  parentName = 'Parent';
}
```

In this example, the `ParentComponent` uses the `ChildComponent` and binds the value of the `parentName` property to the `name` input of the `ChildComponent` using the `[name]` attribute. This causes the `ChildComponent` to display the value of the `parentName` property in its template.

Here is an example of how you might use the `@Input` decorator in an Angular component:

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

@Component({
  selector: 'app-child',
  template: `
    <p>Hello, {{ name }}!</p>
  `
})
export class ChildComponent {
  @Input() name: string;
}
```

The `ChildComponent` has an `@Input` decorated property called `name`. This property can be bound to a value that is passed in from the parent component.

Output

The `@Output` decorator, on the other hand, is used to specify that a component property is an event that can be emitted to the component's parent. This allows the child component to communicate with the parent component and trigger actions or behaviors in the parent.

Here is an example of how you might use the `@Output` decorator in an Angular component:

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

@Component({
  selector: 'app-child',
  template: `
    <button (click)="onClick()">Click me</button>
  `
})
export class ChildComponent {
  @Output() click = new EventEmitter<void>();
  onClick() {
    this.click.emit();
  }
}
```

In this example, the `ChildComponent` has an `@Output` decorated property called `click`, which is an event that is emitted when the button in the template is clicked.

Here is an example of how you might use the `ChildComponent` in a parent component:

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

@Component({
  selector: 'app-parent',
  template: `
    <app-child (click)="onClick()"></app-child>
  `
})
export class ParentComponent {
  onClick() {
    console.log('Child component clicked');
  }
}
```

In this example, the `ParentComponent` uses the `ChildComponent` and listens for the `click` event using the `(click)` event binding. When the `click` event is emitted by the `ChildComponent`, the `onClick()` method of the `ParentComponent` is called.

Build simple forms

Sure! Here is a simple example of a form in an Angular app that allows a user to enter their name and email address:

```
<form #form="ngForm" (ngSubmit)="submitForm(form)">
  <label for="name">Name:</label>
  <input type="text" id="name" name="name" ngModel required>
  <br>
  <label for="email">Email:</label>
  <input type="email" id="email" name="email" ngModel required>
  <br>
  <button type="submit">Submit</button>
</form>
```

This form has two input fields, one for the name and one for the email, and a submit button. The `ngModel` directive is used to bind the values of the input fields to the component's data model, and the `required` attribute ensures that the user must enter a value in the field before the form can be submitted.

The `#form` template reference variable is used to access the form object in the component's class. The `(ngSubmit)` directive is used to specify a function to be called when the form is submitted.

In the component's class, you can define the `submitForm()` function to handle the form submission and access the form data:

```
export class AppComponent {
  submitForm(form: NgForm) {
    console.log(form.value); // {name: '', email: ''}
  }
}
```

To validate the form, you can use the `required` attribute on the input fields, as well as the `pattern` attribute to specify a regular expression that the input value must match. You can also use Angular's built-in validators, such as `minLength` and `maxLength`, to specify validation rules.

To display error messages to the user, you can use the `ngModel` directive's `invalid` and `touched` properties to show the error messages only when the input is invalid and has been modified by the user.

Here is an example of a reactive form in an Angular app:

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

export class AppComponent {
  form = new FormGroup({
    name: new FormControl('', [Validators.required]),
    email: new FormControl('', [Validators.required, Validators.email]),
  });

  get name() { return this.form.get('name'); }
  get email() { return this.form.get('email'); }

  submitForm() {
    console.log(this.form.value); // {name: '', email: ''}
  }
}
```

Dependency injection Angular

In Angular, dependency injection (DI) is a design pattern that allows a component to depend on a service and have that service provided to it at runtime, rather than having to create the service itself. This makes it easier to manage dependencies and isolate the component from the implementation details of the service.

A service is a class that provides a set of related functions or methods that can be used by multiple components in an Angular app. Services are typically used to share data and logic across components, and to abstract away complex or reusable functionality.

To use dependency injection in an Angular app, you first need to define a service class and annotate it with the `@Injectable` decorator. For example:

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

@Injectable({
  providedIn: 'root'
})
export class UserService {
  getUsers() {
    // Return a list of users
  }
}
```

Then, in a component that needs to use the service, you can inject the service into the component's constructor using the `constructor` injection pattern. For example:

```
import { Component } from '@angular/core';
import { UserService } from './user.service';

@Component({
  selector: 'app-users',
  templateUrl: './users.component.html',
  styleUrls: ['./users.component.css']
})
export class UsersComponent {
  constructor(private userService: UserService) { }

  users: any;

  ngOnInit() {
    this.users = this.userService.getUsers();
  }
}
```

In this example, the `UserService` is injected into the `UsersComponent` using the `constructor` injection pattern. The `userService` property is then used to call the `getUsers()` method of the service to retrieve a list of users.

Using dependency injection in this way allows you to easily test the component by mocking the service and injecting the mock service into the component's constructor. It also allows you to easily reuse the service across multiple components in the app.

RxJs in Angular

RxJS (Reactive Extensions for JavaScript) is a library for reactive programming using observables, to make it easier to work with async events and data streams in JavaScript.

1. Observables: Observables are a way to create async data streams in RxJS. You can subscribe to an observable to receive a stream of values, and you can use operators to transform and filter those values.

2. Operators: Operators are functions that you can use to transform and filter the values emitted by an observable. There are many built-in operators, such as `map`, `filter`, and `merge`, as well as ways to create your own custom operators.

3. Subjects: Subjects are a type of observable that can both emit values and also subscribe to other observables. They can be useful for sharing values between different parts of your application.

4. Error handling: RxJS provides ways to handle errors that occur in observables, such as using the `catchError` operator to catch and handle errors, or using the `retry` operator to retry failed requests.

5. Schedulers: Schedulers are a way to control when certain actions are performed in RxJS, such as scheduling work to be done on the next event loop or at a specific time in the future.

6. Async pipes: The async pipe is a built-in Angular pipe that can be used to automatically subscribe to and unsubscribe from observables in your templates.

Observables are a pattern for handling async data that can emit multiple values over time. They are used to represent async data streams, such as user input events or HTTP requests, and provide a way to process and manipulate the data in a reactive manner.

To use RxJS in an Angular app, you first need to install it as a dependency:

```
npm install rxjs
```

Then, you can import the needed operators and functions from the RxJS library and use them in your code.

Here is an example of using the `of` operator to create an observable that emits a sequence of values:

```
import { of } from 'rxjs';

const observable = of(1, 2, 3);

observable.subscribe(val => console.log(val));
// Output: 1, 2, 3
```

In this example, the `of` operator creates an observable that emits the values 1, 2, and 3, and the `subscribe` method is used to subscribe to the observable and process the emitted values.

You can also use operators to transform, filter, and combine observables. For example, the `map` operator can be used to transform the values emitted by an observable, and the `mergeMap` operator can be used to flatten multiple observables into a single observable.

Here is an example of using the `map` and `mergeMap` operators to transform and flatten an observable:

```
import { of, from } from 'rxjs';
import { map, mergeMap } from 'rxjs/operators';

const observable = of([1, 2, 3]);

observable.pipe(
  map(arr => arr.map(val => val * 2)),
  mergeMap(arr => from(arr))
).subscribe(val => console.log(val));
// Output: 2, 4, 6
```

In this example, the `map` operator is used to transform the array emitted by the observable by multiplying each value by 2, and the `mergeMap` operator is used to flatten the transformed array into a single observable.

Firebase in Angular

To integrate Firebase in an Angular app, you can follow these steps:

1. Go to the Firebase console and create a new project.
2. Click on the "Add Firebase to your web app" button to get your Firebase configuration. You will need this information to connect your Angular app to your Firebase project.
3. Install the Firebase npm package by running the following command in your Angular project's root directory:

```
npm install firebase
```

1. In your Angular app, import the Firebase module and the Firebase config that you obtained from the Firebase console:

```
import * as firebase from 'firebase';

const firebaseConfig = {
  apiKey: 'your-api-key',
  authDomain: 'your-auth-domain',
  databaseURL: 'your-database-url',
  projectId: 'your-project-id',
  storageBucket: 'your-storage-bucket',
  messagingSenderId: 'your-messaging-sender-id',
  appId: 'your-app-id'
};

firebase.initializeApp(firebaseConfig);
```

1. You can now use the Firebase API in your Angular app to access Firebase services such as the Realtime Database or Firebase Authentication.

For example, to read data from the Realtime Database, you can use the `ref()` method to get a reference to a specific location in the database and then use the `on()` method to listen for changes to that data:

```
const dbRef = firebase.database().ref('/some/path');
dbRef.on('value', snapshot => {
  console.log(snapshot.val());
});
```

Selection models

SelectionModel is a class in Angular that allows you to manage the selection state of a group of items. Here is an example of how you can use SelectionModel in an Angular component:

```
import { Component } from '@angular/core';
import { SelectionModel } from '@angular/cdk/collections';

@Component({
  selector: 'app-example',
  templateUrl: './example.component.html',
  styleUrls: ['./example.component.css']
})
export class ExampleComponent {
  // Create a new SelectionModel with the initially-selected items
  selection = new SelectionModel<Item>(true, [items[0], items[1]]);

  // The items that will be displayed in the component
  items: Item[] = [
    { name: 'Item 1' },
    { name: 'Item 2' },
    { name: 'Item 3' },
    { name: 'Item 4' }
  ];

  // Toggle the selection state of a single item
  toggleSelection(item: Item) {
    this.selection.toggle(item);
  }

  // Select all items
  selectAll() {
    this.items.forEach(item => this.selection.select(item));
  }

  // Deselect all items
  deselectAll() {
    this.selection.clear();
  }
}
```

In this example, we create a new SelectionModel with the initially-selected items, and then we define a few methods to manipulate the selection state of the items. You can then use the selection model and its methods in your template to display and manipulate the selection state of the items.

HTTP requests

Here is an example of how you can use the HttpClient module in Angular to make HTTP requests:

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

@Component({
  ...
})
export class MyComponent {
  constructor(private http: HttpClient) {}

  getData() {
    this.http.get('/api/data').subscribe(data => {
      console.log(data);
    });
  }

  postData() {
    this.http.post('/api/data', { name: 'John', age: 30 }).subscribe(response => {
      console.log(response);
    });
  }
}
```

In this example, we inject the HttpClient service into our component using the constructor, and then we use the `get()` and `post()` methods to make GET and POST requests, respectively. These methods return observables that we can subscribe to in order to receive the response data.

You can also use the `HttpClient` to make other types of HTTP requests, such as PUT, DELETE, and PATCH, by using the corresponding methods.

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

@Component({
  ...
})
export class MyComponent {
  constructor(private http: HttpClient) {}

  updateData() {
    this.http.put('/api/data/123', { name: 'John', age: 31 }).subscribe(response => {
      console.log(response);
    });
  }

  deleteData() {
    this.http.delete('/api/data/123').subscribe(response => {
      console.log(response);
    });
  }

  patchData() {
    this.http.patch('/api/data/123', { age: 32 }).subscribe(response => {
      console.log(response);
    });
  }
}
```

In these examples, we use the `put()`, `delete()`, and `patch()` methods to send PUT, DELETE, and PATCH requests, respectively. These methods also return observables that we can subscribe to in order to receive the response data.

Simple TODO app

Here is an example of how you could build a simple todo app in Angular:

1. Create a new Angular project using the Angular CLI.
2. Create a `Todo` model to represent a single todo item:

```
export class Todo {
  id: number;
  title: string;
  completed: boolean;
}
```

1. Create a service to handle communication with the backend API:

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';
import { Todo } from './todo';

@Injectable({
  providedIn: 'root'
})
export class TodoService {
  private apiUrl = '/api/todos';

  constructor(private http: HttpClient) { }

  // Get a list of all todos
  getTodos(): Observable<Todo[]> {
    return this.http.get<Todo[]>(this.apiUrl);
  }

  // Add a new todo
  addTodo(todo: Todo): Observable<Todo> {
    return this.http.post<Todo>(this.apiUrl, todo);
  }

  // Toggle the completed status of a todo
  toggleTodo(todo: Todo): Observable<Todo> {
    return this.http.put<Todo>(`${this.apiUrl}/${todo.id}`, todo);
  }

  // Delete a todo
  deleteTodo(todo: Todo): Observable<Todo> {
    return this.http.delete<Todo>(`${this.apiUrl}/${todo.id}`);
  }
}
```

1. Create a component to display and manage the todos:

```
import { Component, OnInit } from '@angular/core';
import { Todo } from './todo';
import { TodoService } from './todo.service';

@Component({
  selector: 'app-todos',
  templateUrl: './todos.component.html',
  styleUrls: ['./todos.component.css']
})
export class TodosComponent implements OnInit {
  todos: Todo[];

  constructor(private todoService: TodoService) { }

  ngOnInit() {
    this.getTodos();
  }

  // Get a list of todos from the service
  getTodos(): void {
    this.todoService.getTodos().subscribe(todos => this.todos = todos);
  }

  // Add a new todo
  addTodo(title: string): void {
    const todo = { title, completed: false } as Todo;
    this.todoService.addTodo(todo).subscribe(todo => this.todos.push(todo));
  }
...
...
...
```

Local storage

Here is an example of how you could create a simple todo app in Angular that stores the todo data in local storage:

1. Create a service to handle communication with local storage:

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

@Injectable({
  providedIn: 'root'
})
export class TodoService {
  private todoList: Todo[] = [];

  constructor() {
    this.todoList = this.getTodos();
  }

  // Get a list of all todos
  getTodos(): Todo[] {
    const todos = localStorage.getItem('todos');
    return todos ? JSON.parse(todos) : [];
  }
  // Add a new todo
  addTodo(todo: Todo): void {
    this.todoList.push(todo);
    this.updateTodos();
  }

  // Toggle the completed status of a todo
  toggleTodo(todo: Todo): void {
    todo.completed = !todo.completed;
    this.updateTodos();
  }

  // Delete a todo
  deleteTodo(todo: Todo): void {
    const index = this.todoList.indexOf(todo);
    this.todoList.splice(index, 1);
    this.updateTodos();
  }

  // Update the todos in local storage
  private updateTodos(): void {
    localStorage.setItem('todos', JSON.stringify(this.todoList));
  }
}
```

- Create a component to display and manage the todos:

```
import { Component, OnInit } from '@angular/core';
import { Todo } from './todo';
import { TodoService } from './todo.service';

@Component({
  selector: 'app-todos',
  templateUrl: './todos.component.html',
  styleUrls: ['./todos.component.css']
})
export class TodosComponent implements OnInit {
  todos: Todo[];

  constructor(private todoService: TodoService) { }

  ngOnInit() {
    this.getTodos();
  }

  // Get a list of todos from the service
  getTodos(): void {
    this.todos = this.todoService.getTodos();
  }
  // Add a new todo
  addTodo(title: string): void {
    const todo = { title, completed: false } as Todo;
    this.todoService.addTodo(todo);
  }

  // Toggle the completed status of a todo
  toggleTodo(todo: Todo): void {
    this.todoService.toggleTodo(todo);
  }
...
...
...
```

MVC architecture

The Model-View-Controller (MVC) architectural pattern is a way of organizing the code in an application to separate the concerns of the different parts of the application.

In Angular, you can use MVC by dividing your code into three main components:

1. **Model**: This is the data that your application works with. It can be stored in a database, retrieved from an API, or generated locally.
2. **View**: This is the part of the application that the user interacts with. It is responsible for displaying the data from the model to the user and handling user input. In Angular, the view is typically implemented using templates and directives.
3. **Controller**: This is the code that sits between the model and the view. It is responsible for handling logic and data manipulation, as well as coordinating communication between the model and the view. In Angular, the controller is typically implemented using components.

Here is an example of how you could implement MVC in an Angular application:

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

@Component({
  selector: 'app-todos',
  templateUrl: './todos.component.html',
  styleUrls: ['./todos.component.css']
})
export class TodosComponent {
  // Model: the list of todos
  todos: Todo[] = [
    { title: 'Todo 1', completed: false },
    { title: 'Todo 2', completed: true },
    { title: 'Todo 3', completed: false }
  ];

  // Controller: add a new todo
  addTodo(title: string): void {
    this.todos.push({ title, completed: false });
  }
...
...
...
```

Resources

1. The official Angular documentation: **https://angular.io/docs** - This is a comprehensive resource that covers all the basics of Angular, as well as more advanced topics.
2. The Tour of Heroes tutorial: **https://angular.io/tutorial** - This is a step-by-step tutorial that guides you through building a simple Angular app, and it covers many of the core concepts and features of Angular.
3. The Angular University blog: **https://blog.angular-university.io/** - This blog contains a wealth of articles and tutorials on various Angular topics, written by experienced Angular developers.
4. The Angular in Action book: **https://www.manning.com/books/angular-in-action** - This is a comprehensive book that covers all the fundamental concepts of Angular, as well as more advanced topics like reactive programming and performance optimization.