Mastering Angular Fundamentals: From Project Setup to Displaying Data with Built-in Directives

Building Angular application: creating a new project, adding components, displaying data, using built-in directives

AI Based - By Eduardo L Silveira

Dear reader,

I am thrilled that you have chosen to learn about building Angular applications with my ebook. Angular is a powerful and popular framework for building modern web applications, and I am confident that you will find the knowledge and skills you gain from this book to be invaluable.

In this ebook, you will learn how to create a new Angular project, add components to your project, display data in your application, and use built-in directives to add additional functionality. These skills will form the foundation of your Angular development journey and will enable you to create stunning and highly functional web applications.

My name is Eduardo Luiz da Silveira, and I am a seasoned developer with a passion for teaching others. I hope that my guidance and expertise will help you to succeed as you work through this ebook and begin building your own Angular applications. If you have any questions or feedback along the way, please don't hesitate to reach out to me.

I wish you all the best in your Angular development journey, and I am excited to see what you will create.

Sincerely,
Eduardo Luiz da Silveira

Topics

- **Setting up development environment:** Installing necessary tools and libraries like Node.js and Angular CLI.

- **Creating a new Angular project:** Using the Angular CLI to generate a new project.

- **Understanding project structure:** Different files and folders serve different purposes.

- **Adding components:** Reusable pieces of code that make up an Angular app. Use the component decorator to specify behavior.

- **Displaying data in templates:** Use interpolation or property binding to show data.

- **Responding to user actions:** Use event binding to specify behavior for specific events.

- **Using built-in directives**: ngFor, ngIf, and ngClass add functionality to app.

- **Creating custom directives:** Reuse code and add custom functionality.

- **Sharing data and functionality with services and dependency injection:** Services can be shared between components. Dependency injection allows for managing dependencies in a single place.

Setting up Development Environment

Setting up a development environment for Angular involves installing the necessary tools and libraries to get started with Angular development. This includes installing Node.js and the Angular CLI.

Node.js is a runtime environment for JavaScript that allows you to run JavaScript on the server side. It is required for Angular development, as it is used to run the Angular CLI and build your Angular application. You can download and install the latest stable version of Node.js from the official website (**https://nodejs.org/**) or using a package manager like Homebrew (for macOS) or Chocolatey (for Windows).

To check if Node.js is installed correctly, you can open a terminal window and run the following command:

```
node -v
```

This should display the version number of Node.js that you have installed.

Once you have installed Node.js, you can install the Angular CLI by opening a terminal window and running the following command:

```
npm install -g @angular/cli
```

This will install the Angular CLI globally on your system, allowing you to use it from any directory.

To check if the Angular CLI is installed correctly, you can run the following command:

```
ng version
```

This should display the version number of the Angular CLI as well as the versions of Angular and other libraries that it uses.

The Angular CLI is a command line tool that makes it easy to create, build, and deploy Angular applications. It provides a set of commands that you can use to generate new Angular projects, add components and services, and perform a variety of other tasks.

For example, you can use the following command to generate a new Angular project:

```
ng new my-project
```

This will create a new directory called "my-project" with a basic Angular application set up inside. The generated project will include a root component, a root module, and some configuration files.

You can then navigate into the project directory and start the development server by running the following commands:

```
cd my-project
ng serve
```

The development server will start up and you can view your application by opening a web browser and navigating to **http://localhost:4200**.

Setting up a development environment for Angular is an essential step in the process of building an Angular application. It involves installing the necessary tools and libraries, such as Node.js and the Angular CLI, to get started with Angular development. A properly set up development environment will provide you with the necessary tools and libraries to build, test, and deploy your Angular application.

Failing to set up a development environment correctly can lead to a variety of problems. For example, if you do not have Node.js installed, you will not be able to run the Angular CLI or build your Angular application. This can make it difficult to develop and test your application, and may even prevent you from deploying it.

In addition to installing the necessary tools and libraries, it is also important to set up your development environment in a way that is consistent with best practices. This may include using a code editor like Visual Studio Code, setting up a version control system like Git, and using a linter to enforce coding standards.

By setting up a development environment that is well-organized and follows best practices, you will be able to work more efficiently and produce higher quality code. This can save you time and effort in the long run, as you will be able to avoid common pitfalls and mistakes that can occur when developing an Angular application.

Overall, it is essential to set up a development environment for Angular correctly in order to build, test, and deploy your application effectively. By following the steps outlined in this article and setting up your development environment in a way that is consistent with best practices, you will be well on your way to building a successful Angular application.

Advanced tip: In addition to installing the necessary tools and libraries, you may also want to consider using a code editor like Visual Studio Code for your Angular development. A code editor can provide helpful features such as syntax highlighting, code completion, and debugging tools that can make your development experience more efficient and enjoyable.

Creating a new Angular Project

Creating a new Angular project is an important step in the process of building an Angular application. It involves using the **Angular CLI** to generate a basic project structure with the necessary configuration files. The **Angular CLI** is a command line tool that makes it easy to create, build, and deploy Angular applications. It provides a set of commands that you can use to generate new Angular projects, add components and services, and perform a variety of other tasks.

To create a new Angular project, you will need to have the Angular CLI installed. You can install the **Angular CLI** by running the following command:

```
npm install -g @angular/cli
```

This will install the Angular CLI globally on your system, allowing you to use it from any directory.

Once you have installed the **Angular CLI,** you can create a new Angular project by running the following command:

```
ng new my-project
```

The root component is the top-level component of your Angular application. It controls the overall layout and appearance of your app, and it is typically the component that is loaded first when your app starts. The root component is defined in a file called "**app.component.ts**" and is decorated with the **@Component** decorator. The **@Component** decorator is a function that provides the component with metadata that defines its behavior.

The root module is a module that provides the root component with the services and dependencies that it needs to function. It is defined in a file called "**app.module.ts**" and is decorated with the **@NgModule** decorator. The **@NgModule** decorator is a function that provides the module with metadata that defines its behavior.

The configuration files include a file called "**angular.json**" that contains the configuration options for your Angular project. This file includes options such as the default stylesheet format (CSS, SCSS, etc.), the default browser configuration, and the default test configuration.

Once you have generated a new Angular project, you can navigate into the project directory and start the development server by running the following commands:

```
cd my-project
ng serve
```

This will start the development server and you can view your application by opening a web browser and navigating to **http://localhost:4200**. The development server will automatically reload the application whenever you make changes to the code, making it easier to develop and test your application.

In addition to creating a new Angular project, you can use the Angular CLI to perform a variety of other tasks as well. For example, you can use it to generate new components, services, and modules by running the following commands:

```
ng generate component my-component
ng generate service my-service
ng generate module my-module
```

You can also use the Angular CLI to build and deploy your application by running the following commands:

```
ng build
ng deploy
```

The "ng build" command will build your Angular application and generate a production-ready version in the "dist" directory. The "ng deploy" command will deploy your application to a specified hosting platform, such as Firebase or GitHub Pages.

By using the Angular CLI, you can streamline your development process and focus on building the features and functionality of your application. It is an essential tool for any Angular developer and can save you time and effort in the long run.

Creating a new Angular project is an essential step in the process of building an Angular application. It involves using the Angular CLI to generate a basic project structure with the necessary configuration files. By creating a new Angular project in the right way, you will have a solid foundation for your application and be able to focus on building the features and functionality that you need.

One of the benefits of using the Angular CLI to create a new project is that it provides a standard project structure that is consistent with best practices. This makes it easier to understand the organization of your project and navigate your codebase. It also ensures that you have all of the necessary configuration files in place, such as the "angular.json" file that contains the configuration options for your project.

Another benefit of using the Angular CLI to create a new project is that it saves you time and effort. Instead of manually creating the necessary files and directories, you can use the Angular CLI to generate a new project with a single command. This can be especially helpful when you are starting a new project from scratch or working with a team of developers.

In addition to creating a new project, the Angular CLI can also be used to perform a variety of other tasks, such as generating new components, services, and modules. This can save you even more time and effort, as you can use the Angular CLI to create the basic boilerplate code for these parts of your application.

Overall, it is important to create a new Angular project in the right way in order to have a solid foundation for your application. By using the Angular CLI to create a new project and follow best practices, you will be able to focus on building the features and functionality of your application and avoid common pitfalls and mistakes.

Understanding Project Structure

Understanding the structure of an Angular project is an important step in the process of building an Angular application. Angular projects have a standard structure that is based on best practices and designed to make it easier to understand the organization of your code and navigate your project.

An Angular project consists of a set of files and directories that are organized into a specific structure. At the **root** of the project, you will find the configuration files and the top-level directories for your application. These directories include:

"**e2e**": This directory contains the end-to-end (**e2e**) tests for your application.
"**node_modules**": This directory contains the npm packages that your application depends on.
"**src**": This directory contains the source code for your application.
"**tmp**": This directory is used by the **Angular CLI** for temporary storage during builds.
Inside the "**src**" directory, you will find the main code for your application. This includes the root component, the root module, and any other components, services, and modules that you create. The "**src**" directory is organized into a set of subdirectories that correspond to the different parts of your application. These directories include:

"**app**": This directory contains the main code for your application, including the root component and any other components, services, and modules that you create.
"**assets**": This directory contains the static assets for your application, such as images and fonts.
"**environments**": This directory contains environment-specific configuration files for your application.
"**styles**": This directory contains the global styles for your application.
The Angular CLI uses the "**angular.json**" file at the root of the project to store the configuration options for your project. This file includes options such as the default stylesheet format (CSS, SCSS, etc.), the default browser configuration, and the default test configuration.

By understanding the structure of an Angular project, you will be able to navigate your codebase more effectively and understand the organization of your application. It is an important step in the process of building an Angular application and can save you time and effort in the long run.

Here is an example of a simple Angular project structure:

```
my-project
├── e2e
│   ├── src
│   └── tsconfig.json
├── node_modules
├── src
│   ├── app
│   │   ├── app.component.css
│   │   ├── app.component.html
│   │   ├── app.component.spec.ts
│   │   ├── app.component.ts
│   │   ├── app.module.ts
│   ├── assets
│   ├── environments
│   ├── styles.css
│   ├── index.html
│   ├── main.ts
│   ├── polyfills.ts
│   ├── test.ts
│   ├── tsconfig.app.json
│   ├── tsconfig.spec.json
├── .editorconfig
├── .gitignore
├── angular.json
├── package.json
├── README.md
├── tsconfig.json
└── tslint.json
```

This project includes a root component called "**app.component**" and a root module called "**app.module**". The root component is defined in the "**app.component.ts**" file and is decorated with the **@Component** decorator. The root module is defined in the "**app.module.ts**" file and is decorated with the **@NgModule** decorator.

The "**app**" directory also includes the template for the root component in the "app.component.html" file, the styles for the root component in the "**app.component.css**" file, and the unit tests for the root component in the "**app.component.spec.ts**" file.

The "**assets**" directory contains the static assets for the application, such as images and fonts. The "**environments**" directory contains environment-specific configuration files for the application. The "**styles.css**" file contains the global styles for the application.

The "**index.html**" file is the main HTML file for the application and it loads the root component and the required dependencies. The "**main.ts**" file is the main entry point for the application and it bootstraps the root module. The "**polyfills.ts**" file includes the polyfills that are required for the application to run in older browsers. The "**test.ts**" file is the entry point for the unit tests.

The "**tsconfig.app.json**" and "**tsconfig.spec.json**" files are the TypeScript configuration files for the application and the unit tests, respectively.

The "**angular.json**" file at the root of the project contains the configuration options for the project. This file includes options such as the default stylesheet format (CSS, SCSS, etc.), the default browser configuration, and the default test configuration.

Here is an example of a more advanced Angular project structure:

```
my-project
├── e2e
│   ├── src
│   └── tsconfig.json
├── node_modules
├── src
│   ├── app
│   │   ├── components
│   │   │   ├── my-component
│   │   │   │   ├── my-component.component.css
│   │
│ │ │ │ ├── my-component.component.html
│ │ │ │ ├── my-component.component.spec.ts
│ │ │ │ ├── my-component.component.ts
│ │ ├── services
│ │ │ ├── my-service.service.ts
│ │ ├── app.component.css
│ │ ├── app.component.html
│ │ ├── app.component.spec.ts
│ │ ├── app.component.ts
│ │ ├── app.module.ts
│ ├── assets
│ ├── environments
│ ├── styles
│ │ ├── _variables.scss
│ │ ├── styles.scss
│ ├── index.html
│ ├── main.ts
│ ├── polyfills.ts
│ ├── test.ts
│ ├── tsconfig.app.json
│ ├── tsconfig.spec.json
├── .editorconfig
├── .gitignore
├── angular.json
├── package.json
├── README.md
├── tsconfig.json
└── tslint.json
```

In this example, the "**app**" directory is organized into subdirectories for components and services. The "**components**" directory contains the individual components for the application, such as "**my-component**". Each component has its own directory with the necessary files for the component, including the component class, template, styles, and unit tests. The "**services**" directory contains the services for the application, such as "**my-service**".

The "**styles**" directory contains the global styles for the application, organized into SCSS partial files. The "**styles.scss**" file imports the partial files and includes the global styles for the application.

Overall, understanding the structure of an Angular project is an important step in the process of building an Angular application. It can help you to understand the organization of your code and navigate your project more effectively. By following best practices and using the standard project structure provided by the Angular CLI, you can save time and effort in the long run.

Understanding the structure of an Angular project is an essential part of building an Angular application. By following the standard project structure and patterns provided by the Angular CLI, you can ensure that your code is organized in a way that is easy to understand and maintain.

One of the main benefits of respecting the project structure and following patterns is that it helps to make your code more readable and understandable. When your code is organized in a consistent and logical way, it is easier for other developers (and even yourself) to understand how it works and make changes to it as needed. This can save time and effort in the long run, as it reduces the need for debugging and makes it easier to add new features and functionality to your application.

Another benefit of respecting the project structure and following patterns is that it helps to ensure that your application is scalable and maintainable. By following best practices and using proven patterns, you can build an application that is easy to extend and modify as your needs change. This can save you time and effort in the long run, as you won't need to spend as much time refactoring your code or dealing with technical debt.

In addition to these benefits, respecting the project structure and following patterns can also help to make your application more efficient and performant. By following best practices and using proven patterns, you can build an application that is optimized for performance and uses resources effectively. This can improve the user experience and make your application more competitive in the marketplace.

Overall, respecting the project structure and following patterns is an important part of building an Angular application. By doing so, you can ensure that your code is organized, readable, maintainable, scalable, and performant. It is an essential part of building a successful Angular application and can save you time and effort in the long run.

One way to respect the project structure and follow patterns is to use the Angular CLI to create your project and generate components, services, and modules. The Angular CLI provides a standard project structure that is based on best practices and follows proven patterns. By using the Angular CLI, you can ensure that your project is set up correctly from the beginning and can focus on building the features and functionality of your application.

Another way to respect the project structure and follow patterns is to follow the Angular style guide and use the recommended coding practices. The Angular style guide provides guidelines for writing clean, maintainable, and scalable code, and includes recommendations for naming conventions, component organization, and other important aspects of Angular development. By following the Angular style guide, you can ensure that your code is consistent and follows best practices.

In addition to these tips, it is also a good idea to stay up to date with the latest Angular best practices and patterns. The Angular ecosystem is constantly evolving, and new patterns and practices are being developed all the time. By staying up to date, you can ensure that you are using the most effective and efficient techniques for building your Angular application.

Overall, respecting the project structure and following patterns is an essential part of building a successful Angular application. By doing so, you can ensure that your code is organized, readable, maintainable, scalable, and performant, and can save you time and effort in the long run.

Adding Components

In an Angular project, components are reusable pieces of code that make up an Angular application. They are responsible for the view logic of an Angular application and control the rendering of the HTML templates.

To create a new component in an Angular project, you can use the Angular CLI to generate a new component with a single command. For example, to create a new component called "**my-component**", you can use the following command:

```
ng generate component my-component
```

This command will create a new directory for the component in the "**src/app**" directory and generate the necessary files for the component, including the component class, template, styles, and unit tests.

The component class is the main code for the component and is defined in the "**.component.ts**" file. It is decorated with the **@Component** decorator, which specifies the metadata for the component, such as the selector, template, and styles.

Here is an example of a simple component class:

```
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';
}
```

In this example, the component has a selector of "app-my-component" and a template located at "**./my-component.component.html**". It also has a style sheet located at "**./my-component.component.css**". The component also has a property called "**title**" with a value of "**My Component**".

"**.component.html**" file. It is a standard HTML file that defines the layout and content for the component.

Here is an example of a simple component template:

```
<h1>{{ title }}</h1>
<p>Welcome to my component!</p>
```

In this example, the template includes an h1 element with the component's "**title**" property and a paragraph element with some static content.

The component styles are the CSS styles for the component and are defined in the "**.component.css**" file. They are standard CSS styles that apply to the component and its template.

Here is an example of a simple component styles file:

```
h1 {
  color: red;
}

p {
  font-size: 18px;
}
```

In this example, the styles file includes rules for styling the **h1** and p elements in the component template.

Components are a powerful and important aspect of an Angular application, as they allow you to create reusable pieces of code that can be easily shared and reused throughout your application. By using the component decorator and following best practices, you can create components that are well-organized, maintainable, and scalable.

Components can be as simple or as complex as needed, depending on the requirements of your application. Here are some additional examples and advanced concepts to consider when working with components in an Angular project:

Input and **Output**

Components can communicate with each other and with their parent components using input and output properties. Input properties allow a component to receive data from its parent component, while output properties allow a component to emit data to its parent component.

Here is an example of a component with **input** and **output** properties:

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

@Component({
  selector: 'app-my-component',
  templateUrl: './my-component.component.html',
  styleUrls: ['./my-component.component.css'],
  inputs: ['title'],
  outputs: ['onTitleChanged']
})
export class MyComponentComponent {
  title = 'My Component';
  onTitleChanged = new EventEmitter<string>();

  changeTitle() {
    this.title = 'New Title';
    this.onTitleChanged.emit(this.title);
  }
}
```

In this example, the component has an input called "title" and an output called "onTitleChanged". The input allows the parent component to pass a value for the "title" property into the component, while the output allows the component to emit an event when the "title" property changes. The component also has a method called "changeTitle" that changes the value of the "**title**" property and emits the "onTitleChanged" event.

You can also specify the providers for a component using the "**providers**" property of the component decorator. The "**providers**" property allows you to specify the services that the component will use, and can be used to inject dependencies into the component.

Here is an example of a component with providers:

```
import { Component, Injectable } from '@angular/core';
import { MyService } from './my-service.service';

@Component({
        selector: 'app-my-component',
        templateUrl: './my-component.component.html',
        styleUrls: ['./my-component.component.css'],
        providers: [MyService]
})
export class MyComponentComponent {
        title = 'My Component';
        data: any;
        constructor(private myService: MyService) { }
        ngOnInit() {
                this.data = this.myService.getData();
        }
}
```

In this example, the component has a provider for the "**MyService**" service. The service is injected into the component using the constructor, and the component uses the service to retrieve data in the "**ngOnInit**" lifecycle hook.

Components are a powerful and flexible part of an Angular application, and the component decorator provides a range of options for customizing the behavior and functionality of a component. By using the component decorator and following best practices, you can create components that are well-organized, maintainable, and scalable.

Adding components to an Angular application is an important part of building a well-organized, maintainable, and scalable application. Components are reusable pieces of code that make up an Angular app and are responsible for the view logic of an application. They allow you to create modular and reusable code that can be easily shared and reused throughout your application.

One of the main benefits of using components is that they allow you to create reusable pieces of code that can be easily shared and reused throughout your application. This can save you time and effort in the long run, as you won't need to write the same code multiple times or maintain multiple copies of the same code. Components also make it easier to test your application, as you can test each component individually and be confident that it is working correctly.

Another benefit of using components is that they allow you to organize your code in a logical and consistent way. By following best practices and using the component decorator to specify the behavior and functionality of a component, you can create components that are well-organized, maintainable, and scalable. This can make it easier to understand your code and make changes to it as needed.

To get the most out of components, it is important to reuse them whenever possible. By reusing components, you can ensure that your code is modular and maintainable, and can save you time and effort in the long run. However, it is also important to consider when it is appropriate to reuse a component and when it is better to create a new component. For example, if a component is specific to a particular feature or part of your application, it may be better to create a new component rather than trying to reuse an existing component.

Overall, using components is an important part of building a successful Angular application. By creating reusable and modular code with components and using the component decorator to specify behavior, you can create an application that is well-organized, maintainable, and scalable. It is an essential part of building a successful Angular application and can save you time and effort in the long run.

Displaying data in templates

Displaying data in an Angular template is an important part of building an Angular application. There are several ways to display data in an Angular template, including interpolation, property binding, and **async** pipe.

Interpolation is a simple way to display data in an Angular template. It allows you to insert the value of a component's property into the template using double curly braces.

Here is an example of interpolation in an Angular template:

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

In this example, the template includes an **h1** element that displays the value of the component's "**title**" property using interpolation. The value of the "**title**" property is inserted into the template between the double curly braces.

Interpolation is a simple and straightforward way to display data in an Angular template, but it has some limitations. It can only display the value of a property and cannot be used to bind to the property or update the property's value.

Property binding is a more powerful way to display data in an Angular template. It allows you to bind the value of a component's property to an element in the template using the square brackets.

Here is an example of property binding in an Angular template:

```
<h1 [innerHTML]="title"></h1>
```

In this example, the template includes an **h1** element that binds to the component's "**title**" property using property binding. The value of the "**title**" property is bound to the element's "**innerHTML**" property using the square brackets.

The async pipe is a useful tool for displaying asynchronous data in an Angular template. It allows you to bind to an **Observable** or a **Promise** in the template and will automatically update the template when the data changes.

Here is an example of using the async pipe in an Angular template:

```
<h1>{{ asyncData | async }}</h1>
```

In this example, the template includes an **h1** element that binds to an asyncData property using the **async** pipe. The async pipe will automatically subscribe to the **asyncData** Observable or **Promise** and update the template with the latest value when it changes.

Here is an example of a component that uses an **Observable** to retrieve data from an **HTTP** request and displays the data in the template using the **async** pipe:

```
import { Component, OnInit } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

@Component({
        selector: 'app-my-component',
        templateUrl: './my-component.component.html',
        styleUrls: ['./my-component.component.css']
})
export class MyComponentComponent implements OnInit {
        data$: Observable<any>;
        constructor(private http: HttpClient) { }
        ngOnInit() {
                this.data$ = this.http.get('/api/data');
  }
}
```

```
<h1>{{ data$ | async }}</h1>
```

In this example, the component uses the HttpClient service to make an HTTP request and retrieve data from an API. The data is stored in an Observable called "**data$**", which is then bound to the template using the async pipe. The async pipe will automatically subscribe to the "**data$**" Observable and update the template with the latest data when it changes.

One more example of displaying async data in an Angular template using the async pipe is as follows:

```
import { Component, OnInit } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

@Component({
        selector: 'app-my-component',
        templateUrl: './my-component.component.html',
        styleUrls: ['./my-component.component.css']
})
export class MyComponentComponent implements OnInit {
        data$: Observable<any>;
        loading = true;
        error = false;

        constructor(private http: HttpClient) { }

        ngOnInit() {
                this.data$ = this.http.get('/api/data');
                this.data$.subscribe(
                        data => {
                                this.loading = false;
                                this.error = false;
                         },
                        error => {
                                 this.loading = false;
                                this.error = true;
                        }
                );
        }
}
```

```
<ng-container *ngIf="loading; else dataBlock">
    Loading data...
</ng-container>
<ng-template #dataBlock>
    <div *ngIf="error; else data">
        An error occurred while loading the data.
    </div>
    <ng-template #data>
        <h1>{{ data$ | async }}</h1>
    </ng-template>
</ng-template>
```

In this example, the component uses the **HttpClient** service to make an HTTP request and retrieve data from an **API**. The data is stored in an Observable called "**data$**", which is then bound to the template using the async pipe. The **async** pipe will automatically subscribe to the "**data$**" Observable and update the template with the latest data.

Displaying data in an Angular template is an essential part of building an Angular application. It allows you to present data to the user and create a dynamic and interactive experience. There are several ways to display data in an Angular template, including interpolation, property binding, and async pipe, and each has its own benefits and use cases.

Interpolation is a simple and straightforward way to display data in an Angular template. It allows you to insert the value of a component's property into the template using double curly braces and is a useful tool for displaying static or simple data.

Property binding is a more powerful way to display data in an Angular template. It allows you to bind the value of a component's property to an element in the template using the square brackets and is often used in conjunction with event binding to handle user input and update the component's data.

The async pipe is a useful tool for displaying asynchronous data in an Angular template. It allows you to bind to an Observable or a Promise in the template and will automatically update the template when the data changes. This is a useful tool for displaying data that is loaded from an API or other external source.

Overall, it is important to choose the right tool for displaying data in an Angular template based on your needs and use case. By using interpolation, property binding, or async pipe appropriately, you can create a dynamic and interactive application that effectively presents data to the user.

Responding to User Actions

Event binding is an important feature of Angular that allows you to specify behavior for specific events in an Angular template. It allows you to bind to an element's event in the template and specify a component's method to be called when the event occurs.

Here is an example of event binding in an Angular template:

```
<button (click)="onClick()">Click me</button>
```

In this example, the template includes a button element that binds to the "**click**" event using event binding. When the button is clicked, the component's "**onClick()**" method will be called.

You can use event binding to handle a wide range of events, such as mouse clicks, keyboard events, and form submissions. It is a powerful tool for creating interactive and dynamic Angular applications.

Here is an example of a component that uses event binding to handle form submissions:

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

@Component({
	selector: 'app-my-form',
	templateUrl: './my-form.component.html',
	styleUrls: ['./my-form.component.css']
})
export class MyFormComponent {
	name: string;
	email: string;

	onSubmit() {
		console.log(this.name, this.email);
	}
}
```

```
<form (submit)="onSubmit()">
        <label>
                Name:
                <input type="text" [(ngModel)]="name">
        </label>
  <br>
        <label>
                Email:
                <input type="email" [(ngModel)]="email">
        </label>
  <br>
        <button type="submit">Submit</button>
</form>
```

In this example, the component has "**name**" and "**email**" properties that are bound to the template using two-way binding. The form element binds to the "**submit**" event using event binding and calls the component's "**onSubmit()**" method when the form is submitted. The "**onSubmit()**" method logs the values of the "**name**" and "**email**" properties to the console.

Event binding is an essential tool for creating interactive and dynamic Angular applications. It allows you to respond to user actions and specify behavior for specific events in the template. Here are three more examples of event binding in Angular, ranging from initial to advanced:

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

@Component({
        selector: 'app-my-component',
        template: `
                <button (click)="counter = counter + 1">Click me</button>
                <p>You have clicked the button {{ counter }} times.</p>
        `,
        styleUrls: ['./my-component.component.css']
})
export class MyComponentComponent {
        counter = 0;
}
```

In this example, the template includes a button element that binds to the "**click**" event using event binding. When the button is clicked, the component's "**counter**" property is incremented by

1. The template also includes a paragraph element that displays the value of the "**counter**" property using interpolation.

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

@Component({
	selector: 'app-my-component',
	template: `
		<input type="text" (input)="onInput($event)">
		<p>You have typed: {{ value }}</p>
	`,
	styleUrls: ['./my-component.component.css']
})
export class MyComponentComponent {
	value = '';

	onInput(event: Event) {
	this.value = (event.target as HTMLInputElement).value;
	}
}
```

In this example, the template includes an input element that binds to the "**input**" event using event binding. When the user types in the input, the component's "**onInput()**" method is called with the event as an argument. The "**onInput()**" method updates the component's "**value**" property with the value of the input element. The template also includes a paragraph element that displays the value of the "**value**" property using interpolation.

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


@Component({
	selector: 'app-my-component',
	 template: `
		<input type="text" (input)="onInput($event)" (blur)="onBlur()">
		<p *ngIf="value">You have typed: {{ value }}</p>
	`,
	styleUrls: ['./my-component.component.css']
})
```

```
export class MyComponentComponent {
    value = '';
    onInput(event: Event) {
        this.value = (event.target as HTMLInputElement).value;
    }
    onBlur() {
        this.value = this.value.trim();
    }
}
```

In this example, the template includes an input element that binds to both the "**input**" and "**blur**" events using event binding. When the user types in the input, the component's "**onInput()**" method is called with the event as an argument, and the component's "**value**" property is updated with the value of the input element. When the input element loses focus, the component's "**onBlur()**" method is called, and the value of the "**value**" property is trimmed of leading and trailing whitespace. The template also includes a paragraph element that is only displayed if the "**value**" property has a value, and it displays the value of the "**value**" property using interpolation.

my-component.component.ts:

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

@Component({
    selector: 'app-my-component',
    templateUrl: './my-component.component.html',
    styleUrls: ['./my-component.component.css']
})
export class MyComponentComponent {
    // Property to store the selected value
    selectedValue: string;

    // Method to be called when the select element changes
    onChange(event: Event) {
    // Update the selectedValue property with the selected option's value
    this.selectedValue = (event.target as HTMLSelectElement).value;
    }
}
```

my-component.component.html:

```
<!-- Select element with event binding -->
<select (change)="onChange($event)">
        <!-- Options with values -->
        <option value="option1">Option 1</option>
        <option value="option2">Option 2</option>
        <option value="option3">Option 3</option>
</select>

<!-- Display the selected value -->
<p *ngIf="selectedValue">You have selected: {{ selectedValue }}</p>
```

In this example, the component has a "**selectedValue**" property that is used to store the selected value of the select element. The select element has event binding to the "**change**" event, and it calls the component's "**onChange()**" method when the selected option changes. The "**onChange()**" method updates the "**selectedValue**" property with the value of the selected option. The template also includes a paragraph element that is only displayed if the "selectedValue" property has a value, and it displays the value of the "**selectedValue**" property using interpolation.

Event binding is an important feature of Angular that allows you to specify behavior for specific events in an Angular template. It is a powerful tool for creating interactive and dynamic Angular applications, and it is essential for responding to user actions in your app.

There are a few best practices to keep in mind when using event binding in your Angular app:

Keep your template clean and easy to read by using event binding sparingly. Only bind to events that are absolutely necessary for your app's functionality.

Use descriptive names for your event binding methods. This will help you understand the purpose of the method at a glance, and it will make it easier to maintain your code in the future.

Avoid using inline templates for event binding methods. Instead, define your event binding methods in the component's TypeScript code, and call them from the template using event binding. This will make your code more maintainable and easier to debug.

Consider using event bubbling to your advantage. In some cases, it may be more convenient to bind to a parent element's event rather than the event of a specific child element. This can help you avoid clutter in your template, and it can make it easier to handle events that occur on multiple elements.

Always consider the performance implications of your event binding methods. If your method has a lot of logic or performs a lot of processing, it could slow down your app. Consider optimizing your methods or using more efficient techniques to keep your app running smoothly.

By following these best practices, you can effectively use event binding to create interactive and dynamic Angular applications, while keeping your code clean and maintainable.

Using built-in directives

Built-in directives are a powerful feature of Angular that allow you to extend the behavior of your templates and add additional functionality to your application. There are several built-in directives that are commonly used in Angular, including:

ngFor: This directive is used to repeat an element or template for each item in an iterable. It is commonly used to generate lists or tables in an Angular template.
Example:

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

In this example, the **ngFor** directive is used to generate a list item for each item in the "**items**" array. The "**item**" template variable is defined using the "**let**" keyword, and it is used to access the current item in the loop.

ngIf: This directive is used to conditionally include or exclude an element or template from the DOM based on a boolean expression. It is commonly used to show or hide elements based on certain conditions.
Example:

```
<p *ngIf="isLoggedIn">Welcome, {{ user.name }}!</p>
<p *ngIf="!isLoggedIn">Please log in to continue.</p>
```

In this example, the ngIf directive is used to show the first paragraph if the "**isLoggedIn**" property is true, and to show the second paragraph if it is false. The "**user**" object is assumed to contain a "**name**" property, which is interpolated in the first paragraph.

ngClass: This directive is used to bind a set of CSS classes to an element based on a boolean expression. It is commonly used to apply dynamic styles based on certain conditions.
Example:

```
<div [ngClass]="{ 'selected': isSelected, 'disabled': isDisabled }">
  Some content
</div>
```

In this example, the **ngClass** directive is used to bind the "**selected**" and "**disabled**" CSS classes to the div element based on the values of the "**isSelected**" and "**isDisabled**" properties. If the "**isSelected**" property is true, the "**selected**" class will be applied to the div. If the "**isDisabled**" property is true, the "**disabled**" class will be applied to the div.

Here are three more examples of using built-in directives in Angular, ranging from initial to advanced:

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

@Component({
	selector: 'app-my-component',
	template: `
		<ul>
			<li *ngFor="let item of items; index as i">{{ i + 1 }} - {{ item }}</li>
		</ul>
	`,
	styleUrls: ['./my-component.component.css']
})
export class MyComponentComponent {
	items = ['Item 1', 'Item 2', 'Item 3'];
}
```

In this example, the **ngFor** directive is used to generate a list of items in an unordered list. The "**items**" array contains three strings, and the **ngFor** directive is used to generate a list item for each of them. The "**index as i**" syntax is used to create a template variable "**i**" that represents the current index of the loop. The value of "**i**" is then used in the template to display the index of each item, along with the item itself.

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

@Component({
	selector: 'app-my-component',
	 template: `
		<div *ngIf="loading; else loaded">
			Loading...
		</div>
		<ng-template #loaded>
			<p>{{ message }}</p>
			<button (click)="refresh()">Refresh</button>
		</ng-template>
	`,
	styleUrls: ['./my-component.component.css']
})
export class MyComponentComponent {
	loading = false;
	message = 'Some content';

	refresh() {
		this.loading = true;
		// Simulate an HTTP request
		setTimeout(() => {
		this.loading = false;
		this.message = 'Updated content';
	}, 1000);
	 }
}
```

In this example, the **ngIf** directive is used to show a loading message while a request is being made, and to show the updated content once the request has completed. The "**loading**" property is a boolean that determines whether the loading message or the content should be shown. The "**loaded**" template reference variable is used to define an "**else**" block for the ngIf directive, which is used to show the content once the request has completed. The "**refresh**" method is bound to the click event of the refresh button using event binding, and it is used to simulate an HTTP request and update the content of the component.

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

```
@Component({
	selector: 'app-my-component',
	template: `
		<div [ngClass]="getClasses()">
			Some content
		</div>
	`,
	styleUrls: ['./my-component.component.css']
})
export class MyComponentComponent {
	isSelected = false;
	isDisabled = false;

	getClasses() {
		return {
			'selected': this.isSelected,
			'disabled': this.isDisabled
		};
	 }
}
```

In this example, the **ngClass** directive is used to bind a set of CSS classes to a div element based on the values of the "**isSelected**" and "**isDisabled**" properties. The "**getClasses**" method is called by the **ngClass** directive, and it returns an object that maps CSS class names to boolean values. If the value of a property is true, the corresponding CSS class will be applied to the element. In this case, the "**selected**" and "**disabled**" classes will be applied to the div element based on the values of the "**isSelected**" and "**isDisabled**" properties.

Using built-in directives is an important aspect of building Angular applications, as they allow you to extend the behavior of your templates and add additional functionality to your app. In this ebook, we have covered three of the most commonly used built-in directives: **ngFor**, **ngIf**, and **ngClass**.

ngFor is a powerful directive that allows you to repeat an element or template for each item in an iterable. It is commonly used to generate lists or tables in an Angular template. When using

ngFor, it is important to understand the syntax and the different options available, such as the "index as i" syntax that allows you to access the current index of the loop.

ngIf is another useful directive that allows you to conditionally include or exclude an element or template from the DOM based on a boolean expression. It is commonly used to show or hide elements based on certain conditions. One advanced use of **ngIf** is to use an "else" block with a template reference variable, which allows you to define a template that will be used if the condition is false.

ngClass is a directive that allows you to bind a set of CSS classes to an element based on a boolean expression. It is commonly used to apply dynamic styles based on certain conditions. When using **ngClass**, it is important to understand how to bind to an object that maps CSS class names to boolean values, as this allows you to apply multiple classes to an element based on different conditions.

In conclusion, using built-in directives is an essential part of building Angular applications. They allow you to extend the behavior of your templates and add additional functionality to your app, and they can be used in a variety of ways to suit the needs of your application. It is important to understand the syntax and the options available for each directive, as well as the best practices for using them in your templates.

Creating custom directives

Custom directives are a powerful feature of Angular that allow you to create reusable components that encapsulate **DOM** manipulation and other logic. They can be used to extend the behavior of your templates and add custom functionality to your app.

There are two types of directives in Angular: structural and attribute. Structural directives alter the structure of the **DOM** by adding, removing, or replacing elements. They are identified by a leading asterisk (*) in the template, such as ***ngIf** and ***ngFor**. Attribute directives alter the appearance or behavior of an element, component, or directive. They are identified by an attribute selector, such as [**ngClass**] and [**ngModel**].

To create a custom directive in Angular, you need to use the **@Directive** decorator and define the selector that will be used to apply the directive to an element. The directive class should implement the **OnInit** interface and define the **ngOnInit** lifecycle hook, which will be called when the directive is initialized. You can also define other lifecycle hooks, such as **ngOnChanges**, **ngDoCheck**, and **ngOnDestroy**, which will be called at different points in the directive's lifecycle.

In addition to the lifecycle hooks, you can define input and output properties in your directive class, which allow you to pass data into and out of the directive. Input properties are decorated with the **@Input** decorator, and they allow you to pass data from the parent component to the directive. Output properties are decorated with the **@Output** decorator and an instance of the **EventEmitter** class, and they allow you to emit events from the directive that can be listened to by the parent component.

Here is an example of a custom directive that adds a red border to an element when the mouse enters it, and removes the border when the mouse leaves:

```
import { Directive, ElementRef, HostListener, Input, Output, EventEmitter } from
'@angular/core';

@Directive({
        selector: '[appHighlight]'
})
```

```
export class HighlightDirective implements OnInit {
	@Input() highlightColor: string;
	@Output() highlightChange = new EventEmitter<string>();

	constructor(private el: ElementRef) { }

	ngOnInit() {
		this.highlight(this.highlightColor || 'red');
	}

	@HostListener('mouseenter') onMouseEnter() {
		this.highlight(this.highlightColor || 'red');
	}

	@HostListener('mouseleave') onMouseLeave() {
		this.highlight(null);
	}

	private highlight(color: string) {
		this.el.nativeElement.style.borderColor = color;
		this.highlightChange.emit(color);
	}
}
```

In this example, the `HighlightDirective` class is decorated with the `**@Directive**` decorator and defines a selector of `[**appHighlight**]`. This means that the directive can be applied to an element by using the `**appHighlight**` attribute. The directive class defines two host listeners, `**mouseenter**` and `**mouseleave**`, which are used to bind to the corresponding events of the element. The `**onMouseEnter**` and `**onMouseLeave**` methods are called when the mouse enters or leaves the element, and they use the `**highlight**` method to set the border color of the element and emit an event with the current highlight color.

The directive also defines an input property called `**highlightColor**` and an output property called `**highlightChange**`. The `highlightColor` property allows you to specify the color that should be used to highlight the element, and the `**highlightChange**` property is an `**EventEmitter**` that emits an event with the current highlight color whenever it changes.

To use the directive, you can simply add the `**appHighlight**` attribute to an element in your template, and bind to the `**highlightColor**` input property and the `**highlightChange**` output property:

```
<div appHighlight [highlightColor]="'yellow'" (highlightChange)="onHighlightChange($event)">
        Hover me to highlight
</div>
```

In this example, the **appHighlight** directive will be applied to the div element, and the **highlightColor** input property will be set to "**yellow**". When the mouse enters or leaves the element, the **highlightChange** output event will be emitted, and the **onHighlightChange** method in the parent component will be called with the current highlight color.

Here are a few examples of where you can use custom directives and how to create them:

In a component template: You can use custom directives to add or remove elements, change element styles or behavior, or bind to element events. To create a custom directive, you can use the **@Directive** decorator and define the directive's selector, input properties, and methods. For example:

```
import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';

@Directive({
        selector: '[appUnless]'
})
export class UnlessDirective {
        @Input() set appUnless(condition: boolean) {
        if (!condition) {
                this.vcRef.createEmbeddedView(this.templateRef);
        } else {
                this.vcRef.clear();
        }
  }
  constructor(private templateRef: TemplateRef<any>, private vcRef: ViewContainerRef) { }
}
```

This structural directive adds or removes an element from the DOM based on a boolean condition. The directive defines an input property called **appUnless** and a constructor that injects the **TemplateRef** and **ViewContainerRef** dependencies. The **TemplateRef** represents the template that will be added or removed, and the **ViewContainerRef** represents the container in which the template will be added or removed.

To use the directive, you can bind to the appUnless input property in your template:

```
<div *appUnless="showElement">
        This element will be added or removed based on the value of "showElement"
</div>
```

In a component class: You can use custom directives to inject dependencies or access the component's template or DOM elements. To create a custom directive, you can use the **@Directive** decorator and define the directive's selector, input properties, and methods. For example:

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

@Directive({
        selector: '[appClickCounter]'
})
export class ClickCounterDirective {
        @Input() clickCount: number;

        constructor(private el: ElementRef) { }

        @HostListener('click', ['$event']) onClick(event: MouseEvent) {
                this.clickCount++;
        this.el.nativeElement.innerHTML = `Click count: ${this.clickCount}`;
         }
}
```

This attribute directive increments a click counter and updates the inner HTML of the element it is applied to when the element is clicked. The directive defines an input property called **clickCount** and a constructor that injects the **ElementRef** dependency. The **ElementRef** represents the element that the directive is applied to, and the **nativeElement** property of the **ElementRef** allows the directive to access the element's properties and methods.

The directive also defines a **@HostListener** decorator that listens for the click event on the host element and increments the clickCount property when the event is emitted. The **@HostListener** decorator takes two arguments: the event to listen for and an optional array of arguments to pass to the event handler.

To use the directive, you can bind to the **appClickCounter** attribute in your template and specify the **clickCount** input property:

```
<div appClickCounter [clickCount]="0">
        Click me to increment the click count
</div>
```

In a service or provider: You can use custom directives to specify providers or dependencies that should be injected into a service or provider. To create a custom directive, you can use the **@Directive** decorator and define the directive's selector, input properties, and methods. For example:

```
import { Directive, InjectionToken } from '@angular/core';

export const API_URL = new InjectionToken<string>('API_URL');

@Directive({
        selector: '[appApiUrl]',
        providers: [{ provide: API_URL, useValue: 'http://example.com' }]
})
export class ApiUrlDirective { }
```

uses the **@Directive** decorator to define a selector and a providers array that contains the provider definition. The provider definition uses the provide property to specify the injection token and the **useValue** property to specify the value that should be injected.

To use the directive, you can apply the **appApiUrl** attribute to an element in your template:

```
<div appApiUrl>
	This element will provide the "API_URL" injection token with the value
"http://example.com"
</div>
```

In a route configuration: You can use custom directives to specify route guards or resolvers that should be executed before or after a route is activated. To create a custom directive, you can use the **@Directive** decorator and define the directive's selector, input properties, and methods. For example:

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

@Directive({
  selector: '[appAuthGuard]'
})
export class AuthGuardDirective implements CanActivate {
  constructor(private authService: AuthService, private router: Router) { }

  canActivate() {
	if (!this.authService.isAuthenticated()) {
		this.router.navigate(['/login']);
		return false;
	}
	return true;
	 }
}
```

This attribute directive implements the **CanActivate** route guard and prevents unauthenticated users from accessing protected routes. The directive defines a constructor that injects the **AuthService** and **Router** dependencies, and implements the **canActivate()** method of the **CanActivate** interface. The **canActivate()** method checks if the user is authenticated and navigates to the login page if the user is not authenticated.

To use the directive, you can apply the **appAuthGuard** attribute to a route configuration in your module:

```
const routes: Routes = [
	{ path: 'protected', component: ProtectedComponent, canActivate: [AuthGuardDirective]
}];
```

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

Custom directives are a powerful feature of Angular that allow you to extend the functionality of your application and create reusable code. When used correctly, custom directives can help you write cleaner, more maintainable, and more flexible code that can be easily shared and maintained across your application.

One of the main benefits of custom directives is that they allow you to encapsulate complex logic and behavior into a reusable component. This can help you avoid duplicating code and reduce the risk of introducing bugs or inconsistencies. By creating a custom directive, you can define a clear interface and contract for the directive's behavior, making it easier for other developers to understand and use the directive.

Another benefit of custom directives is that they can improve the modularity of your code. By separating concerns into distinct directives, you can create more modular and testable code that is easier to reason about. This can help you build larger and more complex applications that are easier to maintain and evolve over time.

However, it's important to use custom directives with care. Overusing directives can lead to code that is difficult to understand and maintain. It's also important to follow best practices when creating custom directives. For example, you should avoid manipulating the DOM directly in your directives and use the Angular API instead. You should also avoid using directives to implement complex business logic and consider creating a service or provider instead.

It's also important to choose the right type of directive for your needs. Angular provides three types of directives: components, attribute directives, and structural directives. Components are the most feature-rich and flexible type of directive, but they are also the most heavy-weight. Attribute directives are simpler and more focused, but they can only modify the appearance or behavior of an element. Structural directives are used to manipulate the DOM and add or remove elements from the template. Each type of directive has its own use cases and trade-offs, so it's important to choose the right one for your needs.

Overall, custom directives can be a powerful tool for extending the functionality of your Angular application. By following best practices and using them wisely, you can create reusable, modular, and maintainable code that helps you build better applications.

}

Sharing data and functionality with services and dependency injection

Services are a key concept in Angular that allow you to share data and functionality between components. A service is a class that defines a specific set of functionality, such as fetching data from a server, validating user input, or logging events. By creating a service, you can encapsulate this functionality and make it available to any component in your application that needs it.

One of the main benefits of services is that they allow you to centralize code that is used in multiple places. This can help you avoid duplicating code and make it easier to maintain your application over time. Services can also improve the testability of your code, as you can write isolated unit tests for the service and mock its dependencies.

To use a service in an Angular component, you need to inject the service into the component's constructor. Angular uses a technique called dependency injection (**DI**) to manage the service's dependencies and provide them to the component at runtime. **DI** allows you to centralize the management of dependencies in a single place, rather than scattered throughout your code. This can make it easier to understand and maintain your code, as well as improve the testability of your application.

Here's an example of a simple service that fetches data from a server and a component that injects and uses the service:

data.service.ts:

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
@Injectable({
	providedIn: 'root'
})
export class DataService {
	constructor(private http: HttpClient) {}
	fetchData() { return this.http.get('/api/data'); }
}
```

data.component.ts:
```
import { Component } from '@angular/core';
import { DataService } from './data.service';

@Component({
	selector: 'app-data-component',
	template: '{{ data }}'
})
export class DataComponent {
	data: any;
	constructor(private dataService: DataService) {
		this.data = this.dataService.fetchData();
	}
}
```

By injecting the **DataService** into the **DataComponent** and using the **fetchData()** method, the component can display the data fetched from the server.

Here are a few additional examples of how you can use services and dependency injection in an Angular application:

Example 1 - Sharing state between components
Let's say you have a shopping cart feature in your application, and you want to share the contents of the cart between multiple components. One way to do this is to create a **CartService** that stores the cart items and exposes methods for adding and removing items. You can then inject the **CartService** into any component that needs to access the cart:

cart.service.ts:
```
import { Injectable } from '@angular/core';
@Injectable({
	providedIn: 'root'
})
export class CartService {
	private items: string[] = [];
	addItem(item: string) {
		this.items.push(item);
	 }
```

```
        removeItem(item: string) {
                this.items = this.items.filter(i => i !== item);
        }
        getItems() {
                return this.items;
        }
}
```

cart.component.ts:

```
import { Component } from '@angular/core';
import { CartService } from './cart.service';

@Component({
        selector: 'app-cart',
        template: `
                <ul>
                <li *ngFor="let item of items">{{ item }}</li>
                </ul>
        `
})
export class CartComponent {
         items: string[];

        constructor(private cartService: CartService) {
                this.items = this.cartService.getItems();
        }
}
```

By injecting the **CartService** into the **CartComponent**, the component can display the items in the cart and update the cart when items are added or removed.

Example 2 - Handling HTTP requests

Angular provides a built-in **HttpClient** service that you can use to make HTTP requests to a server. You can create a service that wraps the **HttpClient** and exposes methods for common HTTP operations, such as fetching data or submitting a form. Here's an example of a **HttpService** that fetches data from a server and a component that injects and uses the service:

http.service.ts:

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

```
@Injectable({
	providedIn: 'root'
})
export class HttpService {
	constructor(private http: HttpClient) {}

	fetchData() {
		return this.http.get('/api/data');
	}
}
```

cart.component.ts:

```
import { Component } from '@angular/core';
import { HttpService } from './http.service';

@Component({
	selector: 'app-http-component',
	template: '{{ data }}'
})
export class HttpComponent {
	data: any;

	constructor(private httpService: HttpService) {
		this.data = this.httpService.fetchData();
	}
}
```

By injecting the `**HttpService**` into the `**HttpComponent**` and using the `**fetchData()**` method, the component can display the data fetched from the server.

Example 3 - Validating user input

Let's say you have a form in your application that requires users to enter a valid email address. You can create a `**ValidationService**` that defines a method for validating email addresses and inject the service into the form component:

validation.service.ts:

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

@Injectable({
	providedIn: 'root'
})
export class ValidationService {
	validateEmail(email: string) {
		// Validate email address using regular expression or other method
	}
}


cart.component.ts:
import { Component } from '@angular/core';
import { ValidationService } from './validation.service';

@Component({
	selector: 'app-form',
	template:
		<form (submit)="onSubmit()">
			<input type="email" [(ngModel)]="email" name="email" required />
			<button type="submit">Submit</button>
		</form>
})
export class FormComponent {
	email: string;
	constructor(private validationService: ValidationService) {}

	onSubmit() {
		if (this.validationService.validateEmail(this.email)) {
			// Submit form
		} else {
			// Show error message
		}
	}
}
```

By injecting the `**ValidationService**` into the `**FormComponent**` and using the `**validateEmail()**` method, the component can validate the email address entered by the user and submit the form if the email is valid.

These are just a few examples of how you can use services and dependency injection in an Angular application. By leveraging these features, you can create reusable, modular, and maintainable code that can be easily shared and tested across your application.

In Angular, services are a great way to share data and functionality between components. By creating a service, you can encapsulate logic and data that can be used across multiple components in your application. This allows you to keep your components lean and focused on their specific responsibilities, as well as making it easier to test and maintain your code.

One of the key benefits of using services is that they can be injected into any component using dependency injection. This allows you to manage your dependencies in a single place, making it easier to update and maintain your code. Dependency injection also helps to improve the testability of your code, as you can easily mock or stub dependencies when testing your components.

However, it's important to keep in mind that dependency injection should be used in a responsible and disciplined way. Overusing dependency injection can lead to over-reliance on the injector and can make your code more difficult to understand and maintain.

To ensure that you're using dependency injection effectively, it's important to follow best practices such as keeping your components lean and focused, and only injecting services that are truly needed by a component. You should also consider the lifetime of your dependencies, as injecting a service into a component with a longer lifetime than the service itself can lead to unexpected behavior.

Overall, services and dependency injection are powerful tools in Angular that can help you to share data and functionality between components in a scalable and maintainable way. By following best practices and using these features responsibly, you can create robust and scalable Angular applications.

Thank you for purchasing our ebook, "Mastering Angular Fundamentals: From Project Setup to Displaying Data with Built-in Directives". We hope that you have found the content valuable and informative.

In this ebook, we covered the essential aspects of building an Angular application, including setting up a development environment, creating a new project, understanding project structure, adding components, displaying data in templates, responding to user actions, and using built-in directives. We also touched on creating custom directives and shared the importance of using services and dependency injection to share data and functionality between components.

We understand that learning a new framework can be overwhelming, but we hope that this ebook has provided a solid foundation for you to continue your journey in becoming an Angular pro. Remember to keep practicing and experimenting with the concepts covered in this ebook, as it is through hands-on experience that you will truly master these skills.

To continue learning about Angular, we recommend checking out some of the following resources:

- The official Angular documentation is a great resource for learning about Angular and its various features and functionality.

- There are many online tutorials and courses available that can help you learn about Angular, such as those offered by Udemy, Coursera, and Code Academy.

- Online communities and forums, such as Stack Overflow and Reddit, can be a great place to ask questions and get help from other developers.

- Blogs and podcasts dedicated to Angular, such as the official Angular blog and the AngularAir podcast, can be a great source of information and inspiration for learning about Angular.

We hope that you have a great day and continue to excel in your programming endeavors.

Best regards,

Eduardo Luiz da Silveira