Angular Basics

A Pathway to Modern Web Applications

Scott Tremaine
Software Developer and Educator

Breakpoint Coding Tutorials

Contents

What is Angular?

Angular is a robust platform and framework that is engineered specifically for building dynamic client-side applications. Its core purpose is to simplify both the development and the testing of such applications by providing a framework to construct the application structure. Born out of the need to extend HTML's capabilities for dynamic content, Angular incorporates HTML as a template language and then expands its syntax to express your application's components clearly and succinctly.

Built in TypeScript

At its heart, Angular is built in TypeScript, which ensures higher security with strong typing and object-oriented features that make the code more manageable and less prone to runtime errors. This pairing with TypeScript not only enhances application performance but also improves developer productivity through advanced editing and debugging capabilities.

Component Based Architecture

Angular's architecture is designed around components—a fundamental concept in Angular. Each component is essentially a self-contained segment of user interface, with its own functionality and typically its own view. This modular approach allows you to build large-scale applications composed of small, manageable, reusable pieces that can be developed in parallel by different teams. This component-based architecture is what makes Angular particularly well-suited for enterprise-level applications where scalability and maintainability are key concerns.

Active Support Channels

Moreover, Angular's ecosystem includes a range of tools and libraries that support routing, form management, client-server communication, and more, providing everything you need to build sophisticated, modern web applications. Angular is continuously updated by Google and a community of developers, ensuring it remains cutting-edge and robust for modern development needs.
This framework not only assists developers in creating efficient applications but also ensures they are future-proof, adapting smoothly to new web technologies as they emerge. For those looking to dive into web development with a tool that scales with your needs, Angular offers a compelling choice.

History and Versions

Angular's journey began with AngularJS, first released in 2010 by Google. AngularJS marked a significant shift in the way developers build web applications, introducing the concept of single-page applications (SPAs) where much of the work is done on the client side, making websites feel faster and more responsive. It used JavaScript and presented a new way of adding interactivity through directives, which extended HTML attributes with custom functionality.

Angular 2

However, as web applications grew in complexity and scale, AngularJS began to show its limitations in performance and scalability. Recognizing these challenges, the Angular team at Google decided to undertake a complete rewrite of the framework. This led to the release of Angular 2 in 2016, which was not just an update but an entirely new framework, built from the ground up. This version introduced TypeScript as the standard language, replacing JavaScript, to provide better tooling and scalability options. Angular 2's release marked the beginning of a new naming convention as well—dropping the "JS" from its name to simply "Angular," symbolizing its complete overhaul and broader capabilities beyond just JavaScript.

Continually Improving

From Angular 2 onwards, the development team adopted semantic versioning and a scheduled release cycle, ensuring more predictable and manageable updates. Angular has since seen multiple versions, with each new release bringing enhancements in performance, additional features, and more refined practices. Notable versions include Angular 4 (2017), which introduced smaller generated code bundles, and Angular 5 (2017), which improved application speed and a focus on material design compatibility.

The most recent versions have continued to build on this foundation, focusing on improving the developer experience and streamlining the application development process. Each version aims to be backward compatible with the last, with deprecations and breaking changes handled with care to ensure a smooth transition for existing projects.

Angular's history reflects its evolution from a pioneering yet somewhat experimental framework into a mature, enterprise-ready platform that powers applications for some of the world's largest companies. Each iteration has been a step forward in addressing the needs of modern web developers, making it one of the most popular and reliable frameworks in the industry today.

Angular vs. Other Frameworks (React, Vue)

Angular, React, and Vue are among the most popular frameworks for developing modern web applications. Each of these frameworks has its strengths and is best suited for different types of projects and developer preferences. Here's a comparative analysis highlighting the key aspects of Angular versus React and Vue:

Angular

- Angular is a full-fledged framework that includes everything you need to build a complex application. It offers a robust set of features out of the box, such as dependency injection, routing, animations, and form validation.
- Uses TypeScript, which provides strong typing and object-oriented features. This can lead to more reliable code that is easier to refactor and maintain.

- Has a steeper learning curve due to its comprehensive nature and the need to understand various concepts like modules, components, services, and dependency injection.
- Best suited for enterprise-grade applications or projects where a scalable architecture is needed from the start.

React

- React is a library focused on building user interfaces, particularly through the use of components. It is less prescriptive about architecture, offering more flexibility in how to structure an application.
- Uses JavaScript with JSX, which allows HTML to be written within JavaScript code. This approach can be more intuitive for developers familiar with JavaScript.
- Generally considered easier to start with due to its focused approach, but complexity can increase as you need to integrate other libraries for routing, state management, etc.
- Ideal for single-page applications and situations where the developer wishes to integrate with other libraries and frameworks for flexibility.

Vue

- Vue offers a balanced approach with a core library focused on the view layer and accompanying libraries for advanced needs like routing and state management.
- Uses JavaScript and an HTML-based template syntax that can be more approachable for developers new to modern JavaScript frameworks.
- Known for its simplicity and fine documentation, Vue provides a gentle learning curve while still offering powerful features.
- Great for both small projects due to its simplicity and large-scale applications, as it is robust enough to handle complex applications.

In summary, the choice between Angular, React, and Vue can often come down to the specific requirements of the project, team expertise, and personal or organizational preferences in terms of ecosystem and design philosophy. Angular is comprehensive, React is flexible, and Vue strikes a balance between the two, offering a progressive approach to web development.

Setting Up the Development Environment

Before diving into Angular development, it's essential to set up the right tools and software on your machine. The foundation of an Angular development environment starts with Node.js and npm (Node Package Manager). Here's a detailed guide on installing these critical components:

Installing Node.js and npm

Node.js is a runtime environment that allows you to run JavaScript on the server-side, and npm is a package manager that facilitates the installation of JavaScript libraries and tools.

Download Node.js:

- Go to the official Node.js website.
- You'll find options to download Node.js for various platforms (Windows, MacOS, Linux). Select the version recommended for most users, which will include npm automatically.

Install Node.js and npm:

Windows:

- Run the downloaded installer (.msi file). Follow the installation prompts. Ensure that the installer adds Node.js and npm to your PATH so they can be accessed from the command line.

MacOS:

- Open the .pkg file you downloaded and follow the instructions to install Node.js and npm.
- Optionally, you can use Homebrew (a package manager for MacOS). If you have Homebrew installed, simply run `brew install node` in the terminal.

Linux:

- Use a package manager appropriate for your Linux distribution. For example, on Ubuntu, you can run `sudo apt-get install nodejs` and `sudo apt-get install npm` in the terminal.
- Ensure that your Linux distribution is running a suitable version of Node.js and npm by checking their versions. Run `node -v` and `npm -v` in the terminal to see if they are properly installed.

Verify Installation:

Once installation is complete, you can verify it by opening your command line or terminal and typing:

- `node -v` - This should display the current version of Node.js.
- `npm -v` - This will show the current version of npm installed on your system.

Update npm:

It's a good practice to ensure that npm is updated to its latest version. You can update npm using the command:

```
npm install npm@latest -g
```

After setting up Node.js and npm, you have the necessary environment to start working with Angular. This setup will allow you to install the Angular CLI and other necessary libraries using npm, paving the way for you to begin developing applications with Angular.

Installing Angular CLI

The Angular Command Line Interface (CLI) is a powerful tool that simplifies the process of creating, managing, and testing Angular applications. It automates a lot of development tasks, making it an essential part of setting up your Angular development environment. Here's how you can install the Angular CLI:

Open your Command Line or Terminal

Ensure that Node.js and npm are installed by running `node -v` and `npm -v`. If these commands return versions, you're ready to proceed.

Install Angular CLI

Run the following command to install the Angular CLI globally on your machine:

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

Installing it globally allows you to run `ng` commands from anywhere on your system.

Verify Installation

After the installation process is complete, verify that the Angular CLI is correctly installed by checking its version:

```
ng version
```

This command will display the version of Angular CLI along with some additional environment information. It's useful for confirming that the CLI is ready to use and to troubleshoot any issues with the installation.

Create a New Project

Initiate the creation of a new project by running:

```
ng new my-first-angular-app
```

Replace `my-first-angular-app` with the desired name of your project.

Respond to Prompts

After initiating the project creation, the CLI will ask you a few questions to configure your project.

Stylesheet Format Prompt

Prompt: "Which stylesheet format would you like to use?"

Options: CSS, SCSS, Sass, Less, Stylus

Recommended Selection: CSS (as it's the most straightforward and widely used format)

Explanation: This choice determines the styling format for your application. CSS is the standard and most familiar styling language, making it a great choice for those new to Angular or web development.

Server-Side Rendering Prompt

Prompt: "Do you want to enable server-side rendering?"

Options: Yes, No

Recommended Selection: No (for simplicity in your first application)

Explanation: Server-side rendering (SSR) can improve the performance and SEO of your application by rendering components on the server before sending them to the browser. However, for learning purposes or a basic application, it's simpler to start without SSR.

Project Creation Process

Once you've answered the prompts, the CLI will proceed to set up your new project. This process can take a few minutes as it involves generating the necessary files and downloading dependencies.

Navigate into Your Project

Change into your project directory:

```
cd my-first-angular-app
```

Serve Your Application

Start the development server and launch your application by running:

```
ng serve
```

This command compiles the application and hosts it on a local web server. By default, the Angular application will be available at `http://localhost:4200/`.

View Your Application

Open a web browser and go to `http://localhost:4200/` to see your new Angular application in action. You should see a welcome message and some default content provided by Angular.

Understanding the Angular Project Structure

Angular applications follow a modular structure that helps in organizing the code in a scalable and maintainable way. Each part of this structure serves a specific purpose and plays a critical role in the application development lifecycle.

Project Root Directory

At the root level of an Angular project, you'll find several configuration files along with the source folder. Each file and folder has a specific purpose:

- **/node_modules/**: Contains all packages that your project depends on. These packages are installed via npm (Node Package Manager) and are defined in your package.json file.
- **/src/**: The source folder where you develop your application. This is where your application's components, templates, styles, images, and anything else your application needs are stored.
- **/.angular/**: Stores configuration files specific to the Angular CLI which help in managing builds, deployments, and other CLI processes.
- **.editorconfig**: Helps maintain consistent coding styles for developers working on your project across various editors and IDEs.
- **.gitignore**: Specifies intentionally untracked files that Git should ignore. Files like system dependencies, build outputs, etc., are typically listed here.
- **angular.json**: Angular CLI configuration file for the entire project. It specifies the schematics of projects, build options, server options, and more.
- **package.json**: Lists the dependencies and versions that your project relies on and may define scripts for running and building the application.
- **package-lock.json** or **yarn.lock**: Automatically generated files which ensure that the same version of each dependency will be used whenever you install them.
- **README.md**: A markdown file where you can write about your project, installation instructions, and other important details.
- **tsconfig.*.json**: Configuration files for TypeScript. They define options for the TypeScript compiler.

/src/ Folder

This is the core directory where most of your Angular's source code resides:

- **/app/**: Contains the logic and data of Angular components, services, directives, pipes, and more:
  - **app.component.***: Defines the root component with its HTML template, CSS, and spec (test) files.

  - **app.module.ts**: Defines the root module that tells Angular how to assemble the application.

- **/assets/**: Used for static files like images, icons, PDFs, etc., which are not part of Angular's compilation process.
- **/environments/**: Contains configuration files for different destination environments, like production or development.
- **favicon.ico**: The small icon displayed in the browser tab.
- **index.html**: The main HTML entry point of your application. It's mostly bare as Angular dynamically inserts views depending on the current router state.
- **main.ts**: The main entry point for your application's module, bootstrapping the AppModule defined in app.module.ts.
- **polyfills.ts**: Different browsers have different levels of support for web standards. Polyfills help normalize those differences.
- **styles.css** or **styles.scss**: Global stylesheets for your application.

Components and Modules

In Angular, the architecture of an application is primarily built using components and modules. These elements are foundational to understanding how Angular applications are structured and function.

Components

Components are the fundamental building blocks of Angular applications. They control a portion of the screen called a view through their associated template and class. A component consists of three main parts:

- **Class:** Written in TypeScript, it handles data and functionality. The class defines properties and methods that are used by the view.
- **Template:** An HTML view that declares and visualizes the data provided by the component's class. Templates use Angular's template syntax, allowing you to enhance the HTML with features like loops, conditionals, and local variables.
- **Decorator:** A feature of TypeScript that provides metadata about the component class, which Angular uses to define how the component should be processed, instantiated, and used at runtime. The most common decorator in Angular is `@Component`.

Each component is designed to be self-contained and reusable, with its own functionality and typically its own view, making components an effective tool for a modular and maintainable codebase.

Example of a Simple Component

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

@Component({
  selector: 'app-hello-world',
  templateUrl: './hello-world.component.html',
  styleUrls: ['./hello-world.component.css']
})
export class HelloWorldComponent {
  title = 'Hello, world!';
}
```

In this example, `HelloWorldComponent` is defined with a template and style URLs pointing to its HTML and CSS files, respectively. It manages a property `title`, which is displayed in its view.

Modules

Angular modules (or NgModules) organize components and other blocks of code into cohesive blocks, each focused on a specific application domain, workflow, or closely related set of capabilities. An NgModule can include components, service providers, and other code files whose scope is defined by the containing NgModule.

NgModule Decorator

- **@NgModule Decorator:** Configures the module instance, describing how to compile a component's template and how to create an injector at runtime. It identifies the module's own components, directives, and pipes, making some of them public so they can be used by the components of other modules.

Example of an Angular Module

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HelloWorldComponent } from './hello-world.component';

@NgModule({
  declarations: [HelloWorldComponent],
  imports: [BrowserModule],
  providers: [],
  bootstrap: [HelloWorldComponent]
})
export class AppModule { }
```

In the example above, `AppModule` is the root module that bootstraps the `HelloWorldComponent`. It declares which components belong to it (declarations), other modules it depends on (imports), and which component should be instantiated on the application load (bootstrap).

Understanding how components and modules interact within an Angular application provides the foundation for effectively organizing and scaling Angular projects. They

help encapsulate and manage the complexity of large applications by modularizing the functionality, making the development process more manageable and more maintainable.

Templates and Data Binding

In Angular, templates are used to define the views of your application, and data binding is a technique to link your application's data with the view, allowing dynamic updates to the user interface.

Templates

Templates in Angular are written in HTML and include Angular-specific template syntax, which allows you to add dynamic behavior to HTML elements. This syntax includes data binding, structural directives, event bindings, and more, which help you create interactive applications efficiently.

Data Binding

Data binding in Angular provides various ways to manage the connection between the DOM (Document Object Model) and your component's data. The main types of data binding are:

- **Interpolation:** {{ `value` }} - Injects a value from the component into the HTML.
- **Property Binding:** `[property]="value"` - Sets a property of a view element to the value of a template expression.
- **Event Binding:** `(event)="handler()"` - Calls a component's method when a specific DOM event occurs.
- **Two-Way Data Binding:** `[(ngModel)]="property"` - Combines property and event binding to create a two-way data flow between a form input and the component's property.

Example of Data Binding

Let's demonstrate a simple use case where we bind a form input to a component's title property, allowing it to update dynamically as the user types.

app.component.ts Here's the default AppComponent class generated by Angular CLI:

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

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
```

```
  title = 'my-first-app';
}
```

app.component.html To create a template that binds an input field to the title property, modify the app.component.html like this:

```
<div>
  <h1>Welcome to {{ title }}!</h1>
  <input [(ngModel)]="title" placeholder="Enter title">
</div>
```

In this template:

- **Interpolation** ({{ `title` }}): Displays the value of title within the `<h1>` tag.
- **Two-Way Data Binding** (`[(ngModel)]="title"`): Binds the input field to the title property of the component. This allows the title to update automatically whenever the user types in the input field, and vice versa, updating the input field if the title property changes programmatically.

Importing FormsModule in a Standalone Component

In Angular's modular system each component must explicitly import any external modules it depends on. For `ngModel` to work, you need to import `FormsModule` from Angular's forms package.

```
import { Component } from '@angular/core';
import { FormsModule } from '@angular/forms'; // Import FormsModule
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [
    RouterOutlet,
    FormsModule // Add FormsModule to the imports array
  ],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'my-first-app';
}
```

Directives and Pipes

In Angular, directives and pipes are powerful features that help you manipulate the DOM and format data directly within your templates. Directives come in two main types—structural and attribute—while pipes are used for transforming displayed values.

Directives

Structural Directives

Structural Directives change the DOM layout by adding, removing, and manipulating elements. Common examples include:

- `*ngIf`: Conditionally includes a block of HTML.
- `*ngFor`: Repeats a node for each item in a list.
- `*ngSwitch`: Adds/removes DOM elements by switching over a value.

Attribute Directives

Attribute Directives change the appearance or behavior of an element, component, or another directive. Examples include:

- `ngStyle`: Dynamically changes the style of an element.
- `ngClass`: Adds and removes CSS classes based on an expression.

Pipes

Pipes transform displayed values within template expressions. Angular comes with several built-in pipes like `date`, `uppercase`, and `lowercase`, and you can also define your own custom pipes.

Example: Using Directives and Pipes in Angular

Let's create a simple example in the `app.component` to demonstrate the usage of directives and pipes. This example will include an `*ngIf` directive to conditionally display data, an `*ngFor` directive to list items, and a `date` pipe to format a date.

app.component.ts

Here, we define our component with a list of items and a date value:

```
import { Component } from '@angular/core';
import { FormsModule } from '@angular/forms'; // Import FormsModule
import { RouterOutlet } from '@angular/router';
import { CommonModule } from '@angular/common';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [
    RouterOutlet,
    FormsModule,
    CommonModule
  ],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
```

```
  title = 'my-first-app';
  items = ['Item 1', 'Item 2', 'Item 3'];
  currentDate = new Date();
  showItems = true;
}
```

app.component.html

In the template, we'll use the directives and pipe to interact with these values:

```
<div>
  <h1>Directives and Pipes Example</h1>

  <!-- Toggle visibility -->
  <button (click)="showItems = !showItems">{{ showItems ? 'Hide' : '
   Show' }} Items</button>

  <!-- Structural Directive: *ngIf -->
  <div *ngIf="showItems">
    <h2>Items List</h2>
    <!-- Structural Directive: *ngFor -->
    <ul>
      <li *ngFor="let item of items">{{ item }}</li>
    </ul>
  </div>

  <!-- Pipe: date -->
  <p>Current Date: {{ currentDate | date:'fullDate' }}</p>
</div>
```

Explanation of the Code

- **Button for Toggling Items:** When clicked, this button toggles the `showItems` property, which in turn toggles the visibility of the list of items using `*ngIf`.
- ***ngIf Directive:** This directive checks if `showItems` is true. If true, it renders the `<div>` element and its children.
- ***ngFor Directive:** This directive iterates over the items array and creates a new list item `<li>` for each item.
- **Date Pipe:** The date pipe formats the `currentDate` object into a more readable full date format.