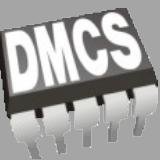




# Interactive Web Applications



1. HTML
2. CSS
3. Angular (history, main concept and architecture)
4. Angular Component
  - Introduction
  - Structure
  - Template
  - Data binding
5. Angular NgModule
6. Building new app and custom components

1

HTML

## What is HTML?

HTML is the standard markup language for creating Web pages.

- HTML stands for Hyper Text Markup Language
- HTML describes the structure of Web pages using markup
- HTML elements are the building blocks of HTML pages
- HTML elements are represented by tags
- HTML tags label pieces of content such as "heading", "paragraph", "table", and so on
- Browsers do not display the HTML tags, but use them to render the content of the page

**HTML**



## HTML Tags

HTML tags are element names surrounded by angle brackets:

- `<tagname>content goes here...</tagname>`
- HTML tags normally come in pairs like `<p>` and `</p>`
- The first tag in a pair is the start tag, the second tag is the end tag
- The end tag is written like the start tag, but with a forward slash inserted before the tag name

**HTML**



## HTML Versions

Since the early days of the web, there have been many versions of HTML:

- HTML - 1991
- HTML 2.0 - 1995
- HTML 3.2 - 1997
- HTML 4.01 - 1999
- XHTML - 2000
- HTML5 - 2014



## HTML Page Structure

```
<html>
```

```
<head>
```

```
<title>Page title</title>
```

```
</head>
```

```
<body>
```

```
<h1>This is a heading</h1>
```

```
<p>This is a paragraph.</p>
```

```
<p>This is another paragraph.</p>
```

```
</body>
```

```
</html>
```

**HTML**

## HTML example

```
<!DOCTYPE html>
<html>
<head>
<title>Page Title</title>
</head>
<body>

<h1>My First Heading</h1>
<p>My first paragraph.</p>

</body>
</html>
```

```
<div style="background-color:black;color:white;padding:20px;">
  <h2>London</h2>
  <p>London is the capital city of England. It is the most populous city in the United Kingdom, with a metropolitan area of over 13 million inhabitants.</p>
</div>
```

## HTML





2

CSS

## What is CSS?

- CSS stands for Cascading Style Sheets
- CSS describes how HTML elements are to be displayed on screen, paper, or in other media
- CSS saves a lot of work. It can control the layout of multiple web pages all at once
- External stylesheets are stored in CSS files



## Why Use CSS?

CSS is used to define styles for your web pages, including the design, layout and variations in display for different devices and screen sizes.

### CSS Saves a Lot of Work!

The style definitions are normally saved in external .css files. With an external stylesheet file, you can change the look of an entire website by changing just one file!



## CSS Solved a Big Problem

HTML was NEVER intended to contain tags for formatting a web page! HTML was created to describe the content of a web page, like:

```
<h1>This is a heading</h1>
```

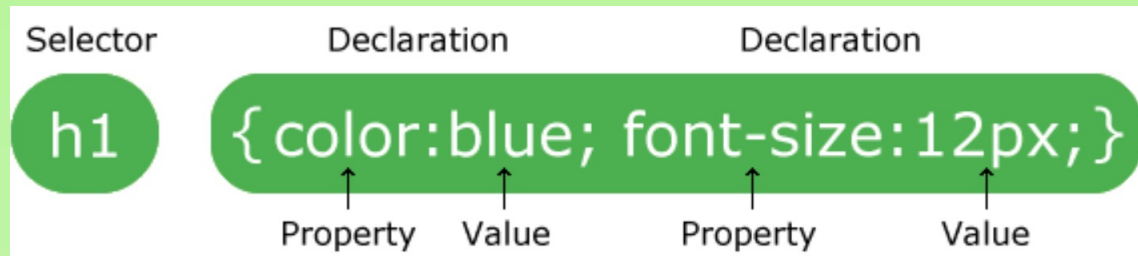
```
<p>This is a paragraph.</p>
```

When tags like `<font>`, and colour attributes were added to the HTML 3.2 specification, it started a nightmare for web developers. Development of large websites, where fonts and color information were added to every single page, became a long and expensive process. CSS removed the style formatting from the HTML page!



## CSS Syntax

A CSS rule-set consists of a selector and a declaration block:



The selector points to the HTML element you want to style. The declaration block contains one or more declarations separated by semicolons. Each declaration includes a CSS property name and a value, separated by a colon. A CSS declaration always ends with a semicolon, and declaration blocks are surrounded by curly braces.

**CSS**



## CSS example

```
h1 {  
  text-align: center;  
  color: red;  
}
```

```
h2 {  
  text-align: center;  
  color: red;  
}
```

```
p {  
  text-align: center;  
  color: red;  
}
```

```
h1, h2, p {  
  text-align: center;  
  color: red;  
}
```

```
<h1 style="color:Tomato;">Hello World</h1>  
<p style="color:DodgerBlue;">Lorem ipsum...</p>  
<p style="color:MediumSeaGreen;">Ut wisi enim...</p>
```

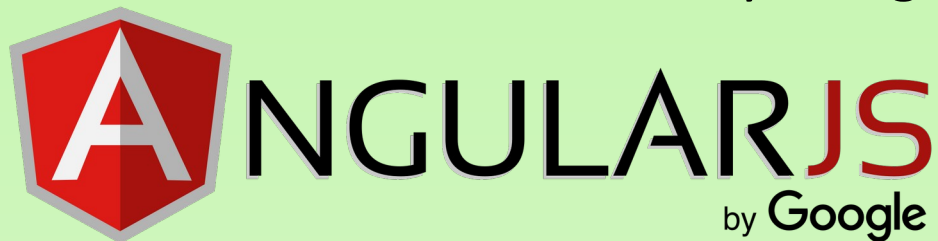
```
p {  
  color: red;  
  /* This is a single-line comment */  
  text-align: center;  
}  
  
/* This is  
a multi-line  
comment */
```

**CSS**

3

Angular

- Everything began in 2008
  - Misko Hevery (a developer at Google)
  - Adam Abrons – friend of Misko
  - A part time project for web designers
- Google internal tool (Google Feedback Tool)
  - Three developers – 6 months – 17000 lines of code (GWT)
  - A simple bet: 2 weeks by himself with Angular
  - Result: One developer – 3 weeks – 1500 lines of code
- AngularJs stable release API 1.0 backed by Google May 2011



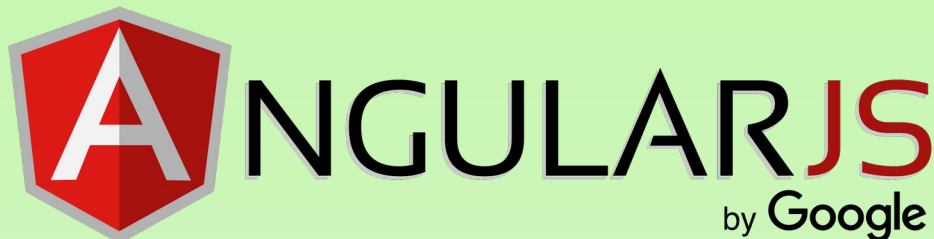


## AngularJs:

- JavaScript
- based on the Model View Controller
- uses terms of scope and controller

## Angular:

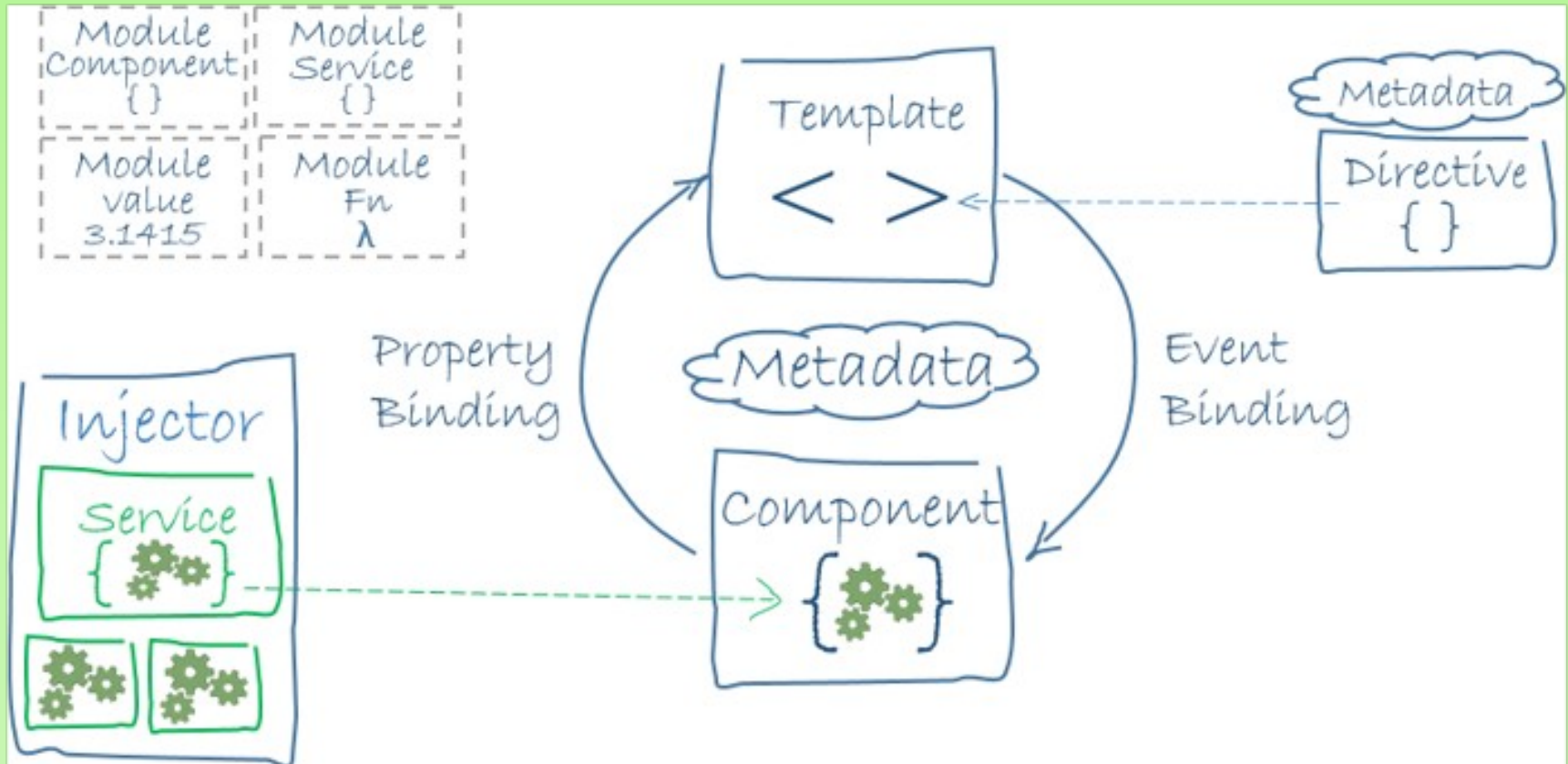
- TypeScript
- Based on the Model View View Model
- uses a hierarchy of components



Angular is a platform and framework for building client applications in HTML and TypeScript. Angular is itself written in TypeScript. It implements core and optional functionality as a set of TypeScript libraries that you import into your apps.



# Angular – main concept and architecture



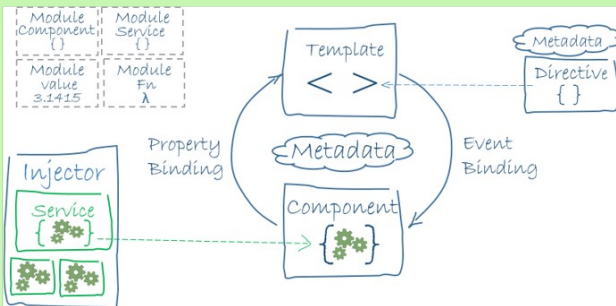
- Modules
- Components (and standalone components)
- Templates and data binding
- Services and dependency injection
- Routing



## Modules

Angular defines the **NgModule**, which declares a compilation context for a set of components that is dedicated to an application domain, a workflow, or a closely related set of capabilities. An **NgModule** can associate its components with related code, such as services, to form functional units.

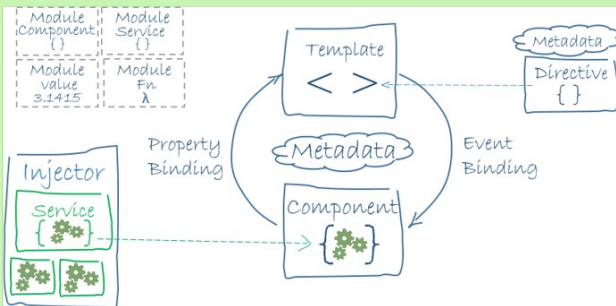
Every Angular app has a **root module**, conventionally named **AppModule**, which provides the bootstrap mechanism that launches the application. An app typically contains many functional modules.



## Modules

Like JavaScript modules, **NgModules** can **import functionality** from other **NgModules**, and allow their own functionality to be exported and used by other **NgModules**.

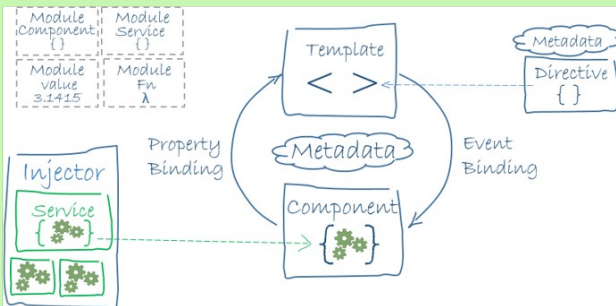
Organizing your code into distinct functional modules helps in managing development of complex applications, and in designing for **reusability**. In addition, this technique lets you take advantage of **lazy-loading**—that is, loading modules on demand—in order to minimize the amount of code that needs to be loaded at startup.



## Components

**Components** define **views**, which are sets of screen elements that Angular can choose among and modify according to your program logic and data. Every app has at least a **root component**.

**Components** use **services**, which provide specific functionality not directly related to views. Service providers can be injected into components as **dependencies**, making your code **modular**, **reusable**, and **efficient**.

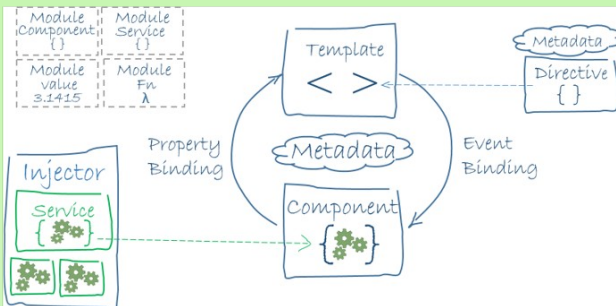


## Components

Each **component** defines a **class** that contains application data and logic, and is associated with an **HTML template** that defines a **view** to be displayed in a target environment.

The **@Component** decorator identifies the class immediately below it as a component, and provides the template and related component-specific metadata.

Both **components** and **services** are simply classes, with decorators that mark their type and provide metadata that tells Angular how to use them.



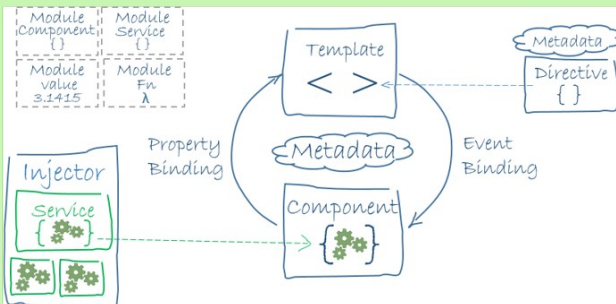


## Templates and data binding

A template combines HTML with Angular markup that can modify the HTML elements before they are displayed. Template binding markup connects your application data and the document object model (DOM).

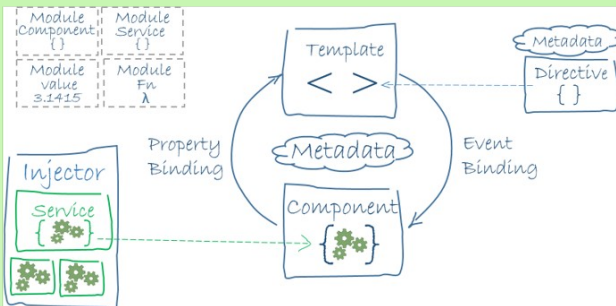
Event binding lets your app respond to user input in the target environment by updating your application data.

Property binding lets you interpolate values that are computed from your application data into the HTML.



## Templates and data binding

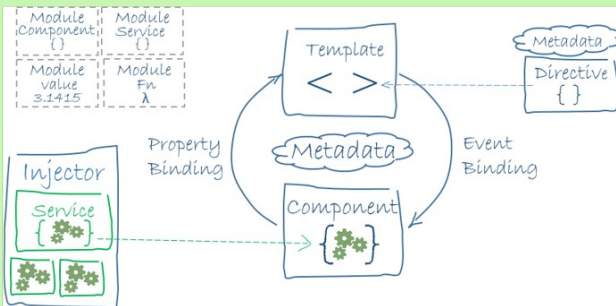
Before a view is displayed, Angular resolves the binding syntax in the template to modify the HTML elements and the DOM, according to your program data and logic. Angular supports two-way data binding, meaning that changes in the DOM, such as user choices, can also be reflected back into your program data.



## Services and dependency injection

For data or logic that is not associated with a specific view, and that you want to share across components, you create a **service class**. A service class definition is immediately preceded by the **@Injectable** decorator. The decorator provides the metadata that allows your service to be injected into client components as a **dependency**.

**Dependency injection (or DI)** lets you keep your **component** classes **lean** and **efficient**. They don't fetch data from the server, validate user input, or log directly to the console; they delegate such tasks to services.

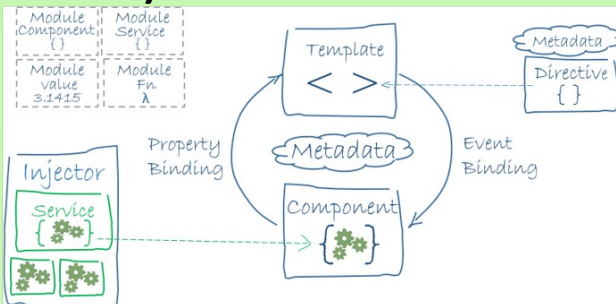


## Routing

The Angular **Router NgModule** provides a service that lets you define a **navigation path** among the different application states and view hierarchies in your app.

It is modeled on the familiar browser **navigation conventions**:

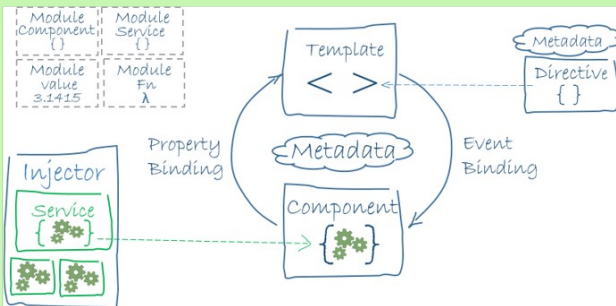
- Enter a **URL** in the address bar and the browser navigates to a corresponding page.
- Click **links** on the page and the browser navigates to a new page.
- Click the browser's **back and forward buttons** and the browser navigates backward and forward through the history of pages you've seen.



## Routing

The **router** maps **URL-like paths** to **views** instead of pages. When a user performs an action, such as clicking a link, that would load a new page in the browser, the router intercepts the browser's behaviour, and shows or hides **view hierarchies**.

If the router determines that the current application state requires particular functionality, and the module that defines it has not been loaded, the router can **lazy-load** the module on demand.



4

## Angular Component

**Components** are the fundamental building blocks of Angular applications. The “application” itself is just the top-level **Component**. Then we break our application into smaller child **Components**.

A **component** controls a patch of screen called a **view**. You define a component's application **logic**—what it does to support the view—inside a **class**. The **class** interacts with the **view** through an API of **properties** and **methods**.



In our **Angular** apps, we write **HTML markup** that becomes our **interactive application**, but the browser only understands a limited set of markup tags; Built-ins like **<select>** or **<form>** or **<video>** all have functionality defined by our browser creator.

What if we want to **teach the browser new tags**? What if we want to have a **<weather>** tag that shows the weather? Or what if we want to create a **<login>** tag that shows a login panel? This is **the fundamental idea behind components**: we will teach the browser **new tags** that have custom functionality attached to them.





```
import { Component, OnInit } from '@angular/core';
import { Student } from './student.model';

@Component({
  selector: 'app-student-list',
  templateUrl: './student-list.component.html',
  styleUrls: ['./student-list.component.css']
})

export class StudentListComponent implements OnInit {
  students: Student[];
  className: string;

  constructor() {
    this.students =
      [
        new Student('RafKot', 123456),
        new Student('JanKow', 987654),
      ];
    this.className = 'IWA';
  }

  ngOnInit() {
  }
}
```



## Importing dependencies

```
import { Component, OnInit } from '@angular/core';  
import { Student } from './student.model';  
import { StudentService } from './student.service';
```

The **import** statement defines the **modules** we want to use to write our code. Here we are importing four things: **Component**, **OnInit**, **Student** and **StudentService**. In this case, we are telling the compiler that **'@angular/core'** defines and exports two JavaScript/TypeScript objects called **Component** and **OnInit**. **OnInit** is used to run code when we initialize the **component**. We also import class **Student** from **'./student.model'** and **StudentService** from **'./student.service'**.



## Component Decorators

```
@Component ({  
  selector: 'app-student-list',  
  templateUrl: './student-list.component.html',  
  styleUrls: ['./student-list.component.css'],  
  providers: [StudentService]  
})
```

Decorators are functions that modify JavaScript classes. Angular defines a number of such decorators that attach specific kinds of metadata to classes, so that it knows what those classes mean and how they should work. The **@Component** decorator identifies the class immediately as a **component** class, and specifies its **metadata**. The **metadata** for a component tells Angular where to get the major building blocks it needs to create and present the **component** and its **view**. In addition to containing or pointing to the template, the **@Component** metadata configures, for example, how the component can be referenced in HTML and what services it requires.



## Component Decorators

```
@Component ({  
  selector: 'app-student-list',  
  templateUrl: './student-list.component.html',  
  styleUrls: ['./student-list.component.css'],  
  providers: [StudentService]  
})
```

### selector

With **selector** we are defining a new tag that we can use in our markup. Wherever Angular finds corresponding **tag** (selector) in **template** HTML an instance of this component has to be created and inserted. For example, if an app's HTML contains **<app-student-list></app-student-list>**, then Angular inserts an instance of the **StudentListComponent** view between those tags. By convention, Angular component selectors begin with the prefix **app-**, followed by the component name.



## Component Decorators

```
@Component ({  
  selector: 'app-student-list',  
  templateUrl: './student-list.component.html',  
  styleUrls: ['./student-list.component.css'],  
  providers: [StudentService]  
})
```

### Adding template with **templateUrl**

In this component we are specifying a **templateUrl** of **./student-list.component.html**. This means that we will load our **template** from the file **student-list.component.html** in the same **directory** as our component. In other words it is the module-relative address of this **component's HTML template**.

```
<p>  
  student-list works!  
</p>
```



## Component Decorators

```
@Component ({
  selector: 'app-student-list',
  template: `
    <p>
      student-list works!
    </p>
  `,
  styleUrls: ['./student-list.component.css'],
  providers: [StudentService]
})
```

### Adding a **template**

Alternatively, we can provide the **HTML template inline**, as the value of the **template property**. This **template** defines the component's host view. Notice that we're defining our **template** string between **backticks** (`` ... ``). This allows us to do multiline strings which makes it easy to put templates inside your code files.



## Component Decorators

```
@Component ({  
  selector: 'app-student-list',  
  templateUrl: './student-list.component.html',  
  styleUrls: ['./student-list.component.css'],  
  providers: [StudentService]  
})
```

### styleUrls

This property allows us to use the **CSS** in the file **student-list.component.css** as the styles for this component. Angular uses a concept called “**style-encapsulation**” which means that **styles** specified for a **particular component** only apply to that component.



## Component Decorators

```
@Component ({  
  selector: 'app-student-list',  
  templateUrl: './student-list.component.html',  
  styleUrls: ['./student-list.component.css'],  
  providers: [StudentService]  
})
```

### providers

An array of **dependency injection** providers for **services** that the component requires.





## Component Class

```
export class StudentListComponent implements OnInit {  
  students: Student[];  
  className: string;  
  
  constructor() {  
    this.students =  
      [  
        new Student('RafKot', 123456),  
        new Student('JanKow', 987654),  
      ];  
    this.className = 'IWA';  
  }  
  
  ngOnInit() {  
  }  
}
```

Instead, let Angular call **ngOnInit** at an appropriate time after constructing a **StudentListComponent** instance.

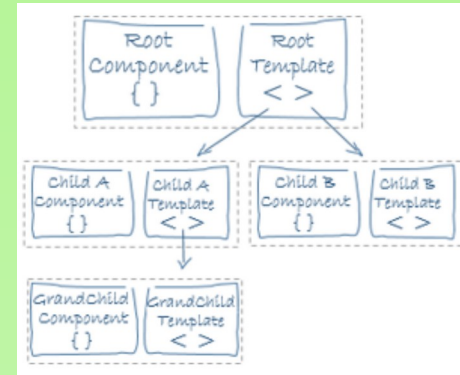
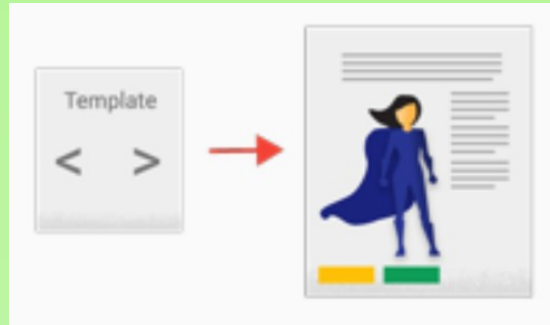
## Properties definition

### Constructor

It is a function that is called when we create new instances of this class. Reserve the **constructor** for simple **initialization** such as wiring **constructor parameters** to **properties**. The **constructor** shouldn't do anything else. It certainly shouldn't call a function that makes HTTP requests to a remote server as a real data **service** would.



## Templates and views



You define a component's view with its companion **template**. A **template** is a form of **HTML** that tells Angular **how to render the component**.

Views are typically arranged **hierarchically**, allowing you to modify or show and hide entire UI sections or pages as a unit. The **template** associated with a component defines that component's **host view**.

The component can also define a **view hierarchy**, which contains **embedded views, hosted by other components**. A view hierarchy can include views from components **in the same NgModule**, but it also can include views from components that are defined **in different NgModules**.



## Template syntax

```
student-list.component.html x
1  <h1> This is a student-list component! </h1>
2
3  <p>{{ student.name }}</p>
4
5  <app-student-detail *ngFor="let student of students" [student]="student">
6  </app-student-detail>
7
8  <button (click)="addPoints()"></button>
9
10 <input [(ngModel)]="student.name">
```

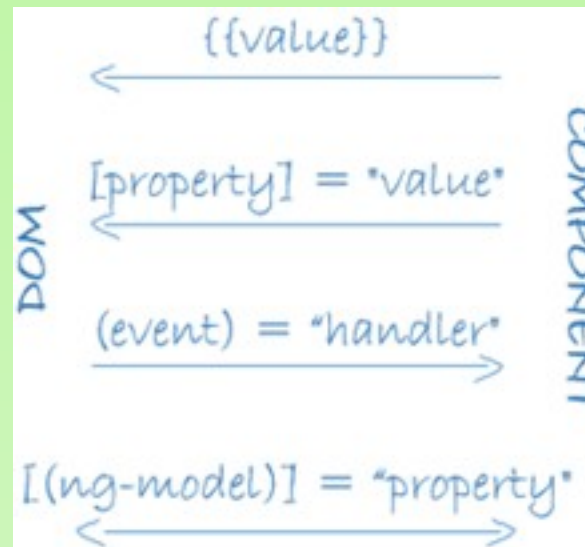
A **template** looks like **regular HTML**, except that it also contains **Angular template syntax**, which **alters** the HTML based on **app's logic** and the **state of app** and **DOM data**. Your template can use data binding, pipes to transform data before it is displayed, and directives to apply app logic.

This template uses typical HTML elements like **<h1>** and **<p>**, and also includes **Angular template-syntax** elements: **\*ngFor**, **{{student.name}}**, **(click)**, **[student]**, and **<app-student-detail>**. Those elements tell Angular how to render the HTML to the screen, using program logic and data.



Without a framework, you would be responsible for pushing data values into the HTML controls and turning user responses into actions and value updates. Writing such push/pull logic by hand is tedious, error-prone, and a nightmare to read, as any experienced jQuery programmer can attest.

Angular supports **two-way data binding**, a mechanism for coordinating parts of a **template** with parts of a **component**. Add **binding markup** to the **template HTML** to tell Angular how to connect both sides.



```
<p>{{ student.name }}</p>

<app-student-detail *ngFor="let student of students"
  [student]="student">

</app-student-detail>
```

The `{{ student.name }}` *interpolation* displays the component's `student.name` property value within the `<p>` element.

The `[student]` property binding passes the value of `student` from the parent `StudentListComponent` to the `student` property of the child `StudentDetailComponent`.

```
export class StudentDetailComponent implements OnInit {
  @Input() student: Student;
```

We added to the `student` property a decorator of `@Input`. This syntax allows us to pass in a value from the parent template. In order to use `Input` remember to add it to the list of **constants** in `import`. To pass values to a component we used the bracket `[]` syntax in our template.



```
<button (click)="addPoints()"></button>
```

```
<input [(ngModel)]="student.name">
```

The **(click) event** binding calls the component's **addPoints()** method when the user clicks a student's name.

**Two-way data binding** is an important fourth form that combines property and event binding in a single notation. A **data property value** flows to the input box **from the component** as with **property binding**. The user's changes also flow **back to the component**, resetting the property to the latest value, as with **event binding**.

Signal Model: A modern approach introduced in newer versions, using the `model()` API. Syntax: `[(value)]="mySignal"`

Angular processes all data bindings **once per JavaScript event cycle**, from the root of the application component tree through all child components.

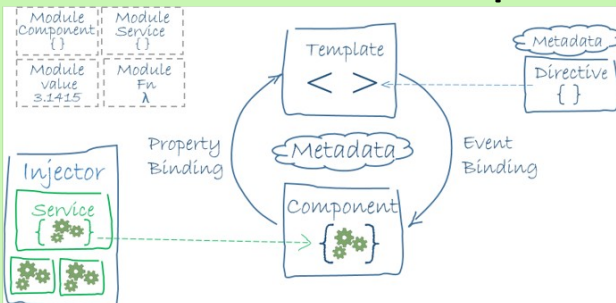


## Standalone components

**Standalone components** are a new organizational pattern that were introduced in Angular v15 and is the default since v17. In contrast to NgModules, it allows developers to organize code and manage dependencies through components rather than feature modules.

Two things that are unique to this “new” pattern are:

- **standalone** - when provided the value true, this tells Angular that the component does not need to be declared in an NgModule;
- **imports** - allows developers to declare what dependencies will be used in the component.



5

## Angular NgModule



Angular apps are **modular** and Angular has its own **modularity system** called **NgModules**. **NgModules** are **containers** for a **cohesive block of code** dedicated to an application **domain**, a **workflow**, or a closely related set of **capabilities**. They can contain **components**, **service providers**, and other **code files** whose scope is defined by the containing **NgModule**. They can **import** functionality that is exported **from other NgModules**, and **export** selected functionality **for use by other NgModules**.

Every Angular app has at least **one NgModule class**, the **root module**, which is conventionally named **AppModule** and resides in a file named **app.module.ts**. You launch your app by bootstrapping the root NgModule.

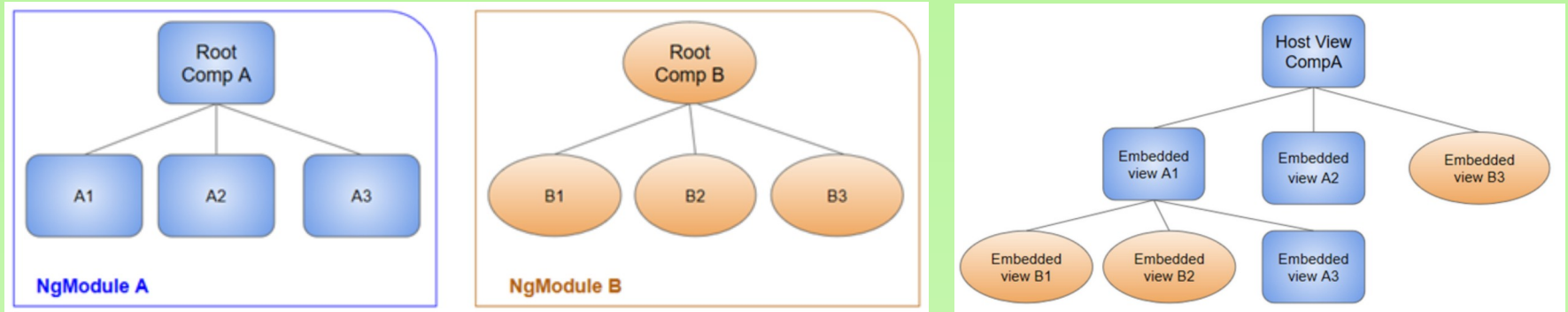


An **NgModule** is defined by a class decorated with **@NgModule()**. The **@NgModule()** decorator is a function that takes a single metadata object, whose properties describe the module.

```
app.module.ts x
Compile TypeScript to JavaScript?
1  import { BrowserModule } from '@angular/platform-browser';
2  import { NgModule } from '@angular/core';
3  import { AppComponent } from './app.component';
4  import { StudentListComponent } from './student-list/student-list.component';
5
6  @NgModule({
7    declarations: [
8      AppComponent,
9      StudentListComponent
10   ],
11   exports: [],
12   imports: [
13     BrowserModule
14   ],
15   providers: [],
16   bootstrap: [AppComponent]
17 })
18 export class AppModule { }
```



**NgModules** provide a **compilation context** for their components. A **root NgModule** always has a **root component** that is created during **bootstrap**, but any NgModule can include any number of additional components.



A **component and its template** together define a **view**. A component can contain a **view hierarchy**, which allows you to define **complex areas of the screen** that can be created, modified, and destroyed as a unit. A view hierarchy can **mix views** defined in components that belong to **different NgModules**.



## Registration of components in app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';
import { StudentListComponent } from './student-list/student-list.component';
@NgModule({
  declarations: [
    AppComponent,
    StudentListComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Angular has a powerful concept of **modules**. When you boot an Angular app, you are not booting a component directly, but instead you create an **NgModule** which points to the component you want to load. You have to declare **components** in an **NgModule** before you can use them in your **templates**. You can think of an **NgModule** a bit like a “**package**” and **declarations** states what components are “owned by” this module.



6

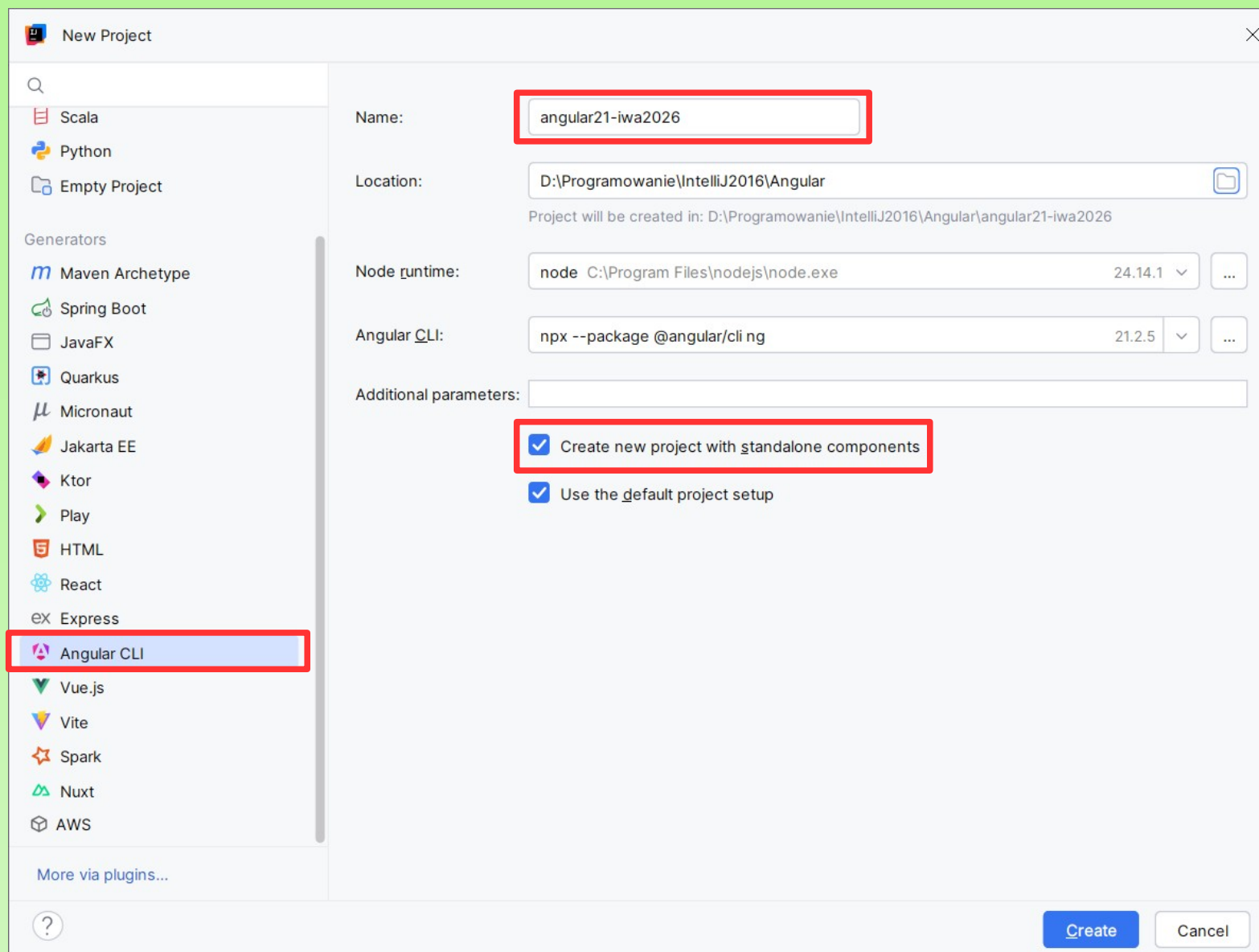
## Building new app and custom components

# Angular – new project in IntelliJ IDEA



54

dr inż. Rafał Kotas, rkotas@dmcs.pl



The image shows the 'New Project' dialog in IntelliJ IDEA, configured for an Angular CLI project. The 'Generators' list on the left has 'Angular CLI' selected. The 'Name' field is 'angular21-iwa2026'. The 'Location' is 'D:\Programowanie\IntelliJ2016\Angular', with a note that the project will be created in 'D:\Programowanie\IntelliJ2016\Angular\angular21-iwa2026'. The 'Node runtime' is 'node C:\Program Files\nodejs\node.exe' (version 24.14.1). The 'Angular CLI' version is '21.2.5'. The 'Additional parameters' field is empty. Two checkboxes are checked: 'Create new project with standalone components' and 'Use the default project setup'. The 'Create' button is at the bottom right.

**New Project**

Search:


Scala  
Python  
Empty Project

Generators

- Maven Archetype
- Spring Boot
- JavaFX
- Quarkus
- Micronaut
- Jakarta EE
- Ktor
- Play
- HTML
- React
- Express
- Angular CLI**
- Vue.js
- Vite
- Spark
- Nuxt
- AWS

More via plugins...

Name:

Location:    
Project will be created in: D:\Programowanie\IntelliJ2016\Angular\angular21-iwa2026

Node runtime:  24.14.1 ...

Angular CLI:  21.2.5 ...

Additional parameters:

☒ Create new project with standalone components

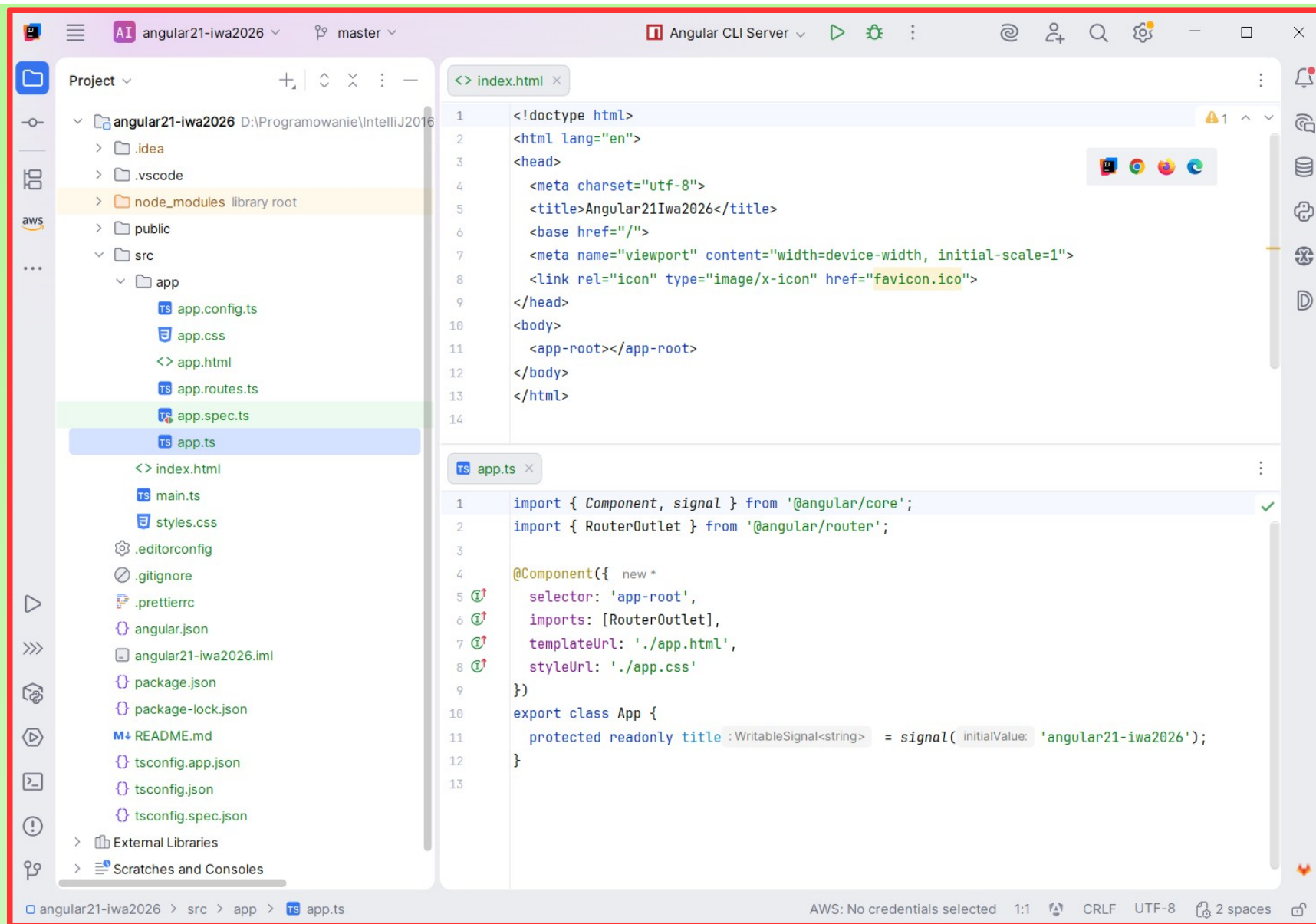
☒ Use the default project setup

# Angular – new project in IntelliJ IDEA



55

dr inż. Rafał Kotas, rkotas@dmcs.pl

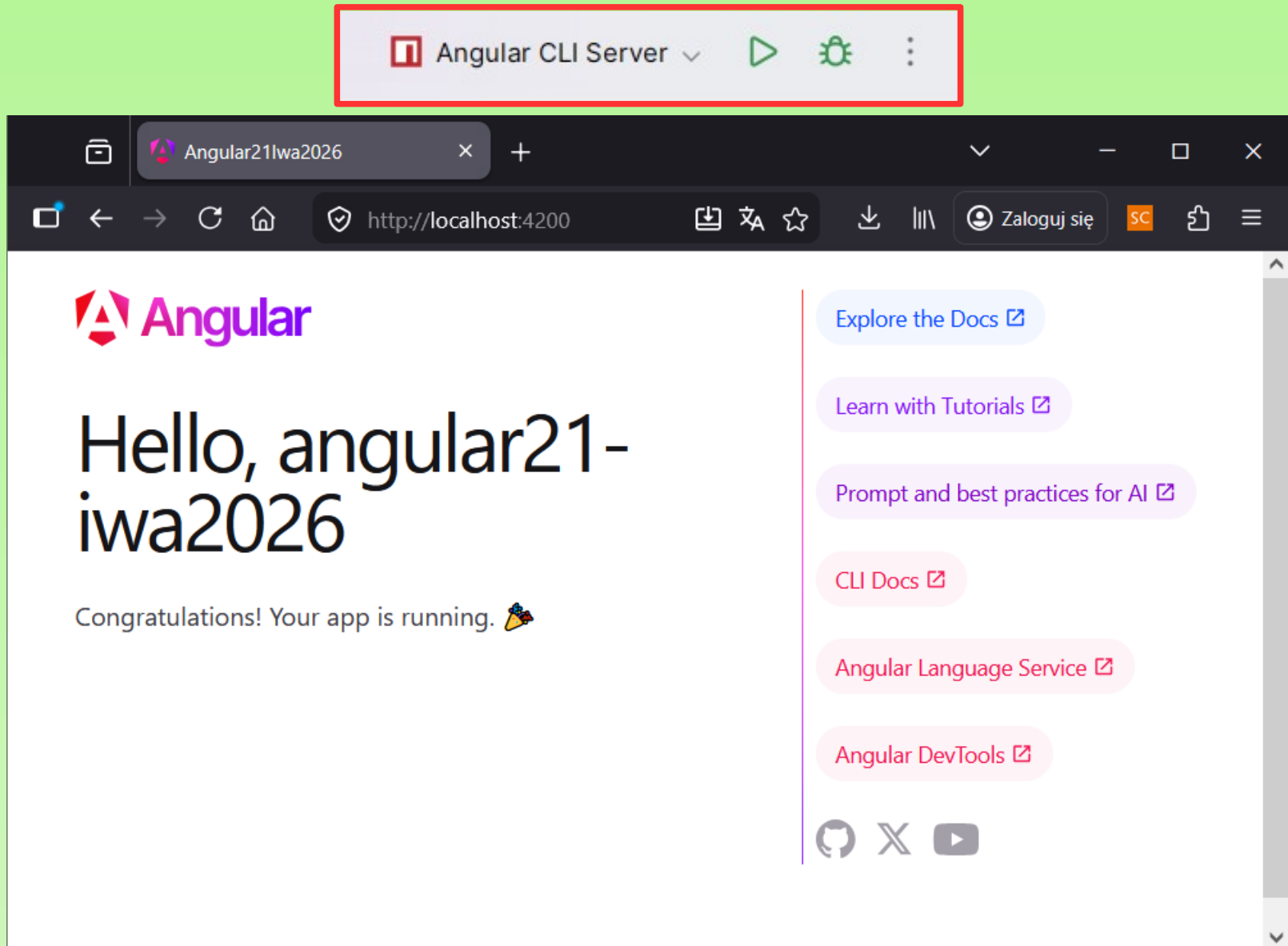


# Angular – new project in IntelliJ IDEA



56

dr inż. Rafał Kotas, rkotas@dmcs.pl



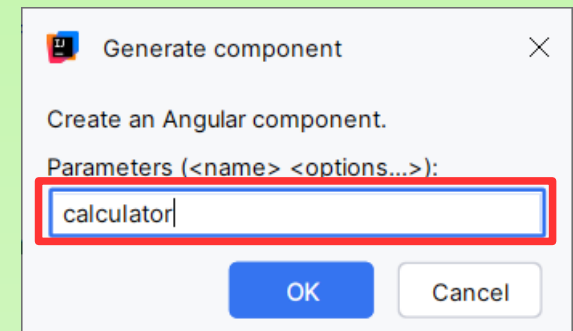
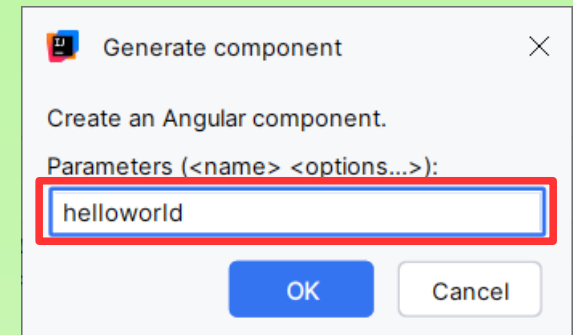
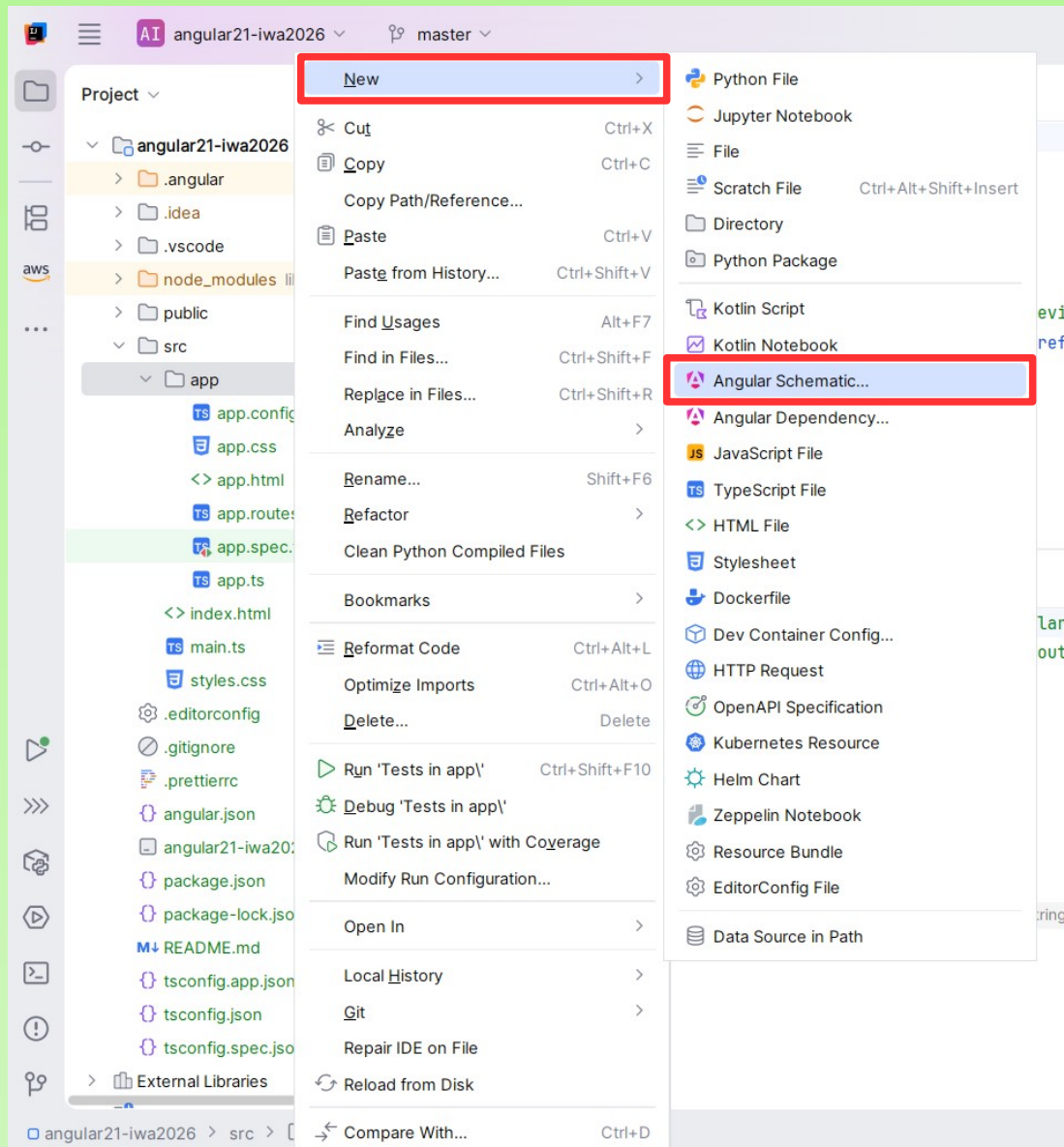


# Angular – new project in IntelliJ IDEA

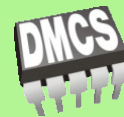


57

dr inż. Rafał Kotas, rkotas@dmcs.pl



# Angular – new project in IntelliJ IDEA



58

dr inż. Rafał Kotas, rkotas@dmcs.pl

```
calculator.ts x
1 import {Component, computed, signal} from '@angular/core';
2 import {form, FormField} from '@angular/forms/signals';
3
4 @Component({
5   selector: 'app-calculator',
6   imports: [FormField],
7   templateUrl: './calculator.html',
8   styleUrls: ['./calculator.css'],
9 })
10 export class Calculator {
11
12   //Angular signals approach
13   operationDataInputsModel : WritableSignal<{ a: number; b: number }> = signal({
14     a: 0,
15     b: 0
16   });
17
18   operationDataInputsForm : FieldTree<{ a: number; b: number }, string> = form(this.operationDataInputsModel);
19
20   protected readonly additionResult : Signal<number> = computed(() : number => {
21     const data : { a: number; b: number } = this.operationDataInputsModel();
22     return data.a + data.b;
23   });
24
25   //RxJS approach
26   result!: number;
27
28   addition (c: number, d: number): void {
29     this.result = c + d;
30     console.log("addition operation: result=" + this.result)
31   }
32
33 }
```

`<> calculator.html x`

```
1  <p>calculator works!</p>
2  Signal Form example
3  <form>
4    <div>
5      <label>a: </label>
6      <input type="number" [formField]="operationDataInputsForm.a" />
7    </div>
8    <div>
9      <label>b: </label>
10     <input type="number" [formField]="operationDataInputsForm.b" />
11   </div>
12   <h3>Addition result: {{ additionResult() }}</h3>
13 </form>
14
15 <hr/>
16
17 RxJS Form example
18 <form>
19   <input #c :HTMLInputElement type="number" placeholder="c value"/>
20   <input #d :HTMLInputElement type="number" placeholder="d value"/>
21   <button (click)="addition(c.valueAsNumber, d.valueAsNumber)" type="button">Addition</button>
22
23   <h3> Result: {{result}}</h3>
24 </form>
```

# Angular – new project in IntelliJ IDEA



60

dr inż. Rafał Kotas, rkotas@dmcs.pl

```
<> app.html x
187     <main class="main">
188         <div class="content">
189             <div class="left-side">
235                 <h1>Hello, {{ title() }}</h1>
236                 <p>Congratulations! Your app is running. 🚀</p>
237                 <app-helloworld></app-helloworld>
238                 <app-calculator></app-calculator>
239             </div>
```

# Angular – new project in IntelliJ IDEA



61

dr inż. Rafał Kotas, rkotas@dmcs.pl

Angular21Iwa2026

http://localhost:4200

Zaloguj się

Angular

## Hello, angular21-iwa2026

Congratulations! Your app is running. 🎉

helloworld works!

calculator works!

Signal Form example

a: 5

b: 5

**Addition result: 10**

RxJS Form example

4 7 Addition

**Result: 11**

Explore the Docs

Learn with Tutorials

Prompt and best practices for AI

CLI Docs

Angular Language Service

Angular DevTools

GitHub X YouTube

## 1) Install Node.js

<https://nodejs.org/en>

## 2) Install npm

<https://www.npmjs.com/>

## 3) Install Angular

<https://angular.dev/>



1. Implement Angular HelloWorld and Calculator (addition operation) from the above example.
2. Add other mathematical operations for Calculator (subtraction, multiplication, division).
3. Add styling to Calculator (Angular Material).
4. Add „quadratic equation and Fibonacci sequence” components to the project.





# THE END

WIKAMP → IWA\_5.zip

