**Creating a New Project**

We'll start working on the next project to learn about components.

In the last section, we tunnel-visioned on a few files and features. It’s time to expand a little more. We’ll continue to take the approach of slowly peeling away the features in Angular.

The main focus of this section is **components**. Before we get started, let’s create a new project to work in. If you’re using Educative to run the code examples, you can skip this lesson.

In the command line, run the following command:

ng new components

After running the command, we’ll be asked about routing and our choice of CSS. We’ll select the default settings for everything.

# Understanding Components

In this lesson, we'll discuss what components are.

**We'll cover the following**

* [The problem](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5236861905141760#The-problem)
* [The solution](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5236861905141760#The-solution)
* [The App component](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5236861905141760#The-App-component)
* [Dissecting the App component](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5236861905141760#Dissecting-the-App-component)
* [Summary](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5236861905141760#Summary)

Let’s discuss some theory before jumping into our project code. One of the major features of Angular is components. Components are **custom** reusable elements. They were introduced to break an application into smaller pieces.

The browser has a limited set of markup tags that it can understand. There are some tags that have functionality defined by the browser, such as the <select>, <video>, and <audio> tags. These are powerful tags, but what if we want to teach the browser additional tags? Maybe we want a tag that displays a credit card number input or a tag that outputs pagination links.

This is where components come into play. Components are a way to **teach** the browser new tags.

## The problem

Let’s look at an example of why components are necessary. In the example below, we have a list of posts.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

<div class="post">

  <img src="image1.png">

  <div class="author">John</div>

  <div class="title">Some Title</div>

</div>

<div class="post">

  <img src="image2.png">

  <div class="author">Jane</div>

  <div class="title">Another Title</div>

</div>

<div class="post">

  <img src="image3.png">

  <div class="author">Luis</div>

  <div class="title">A Third Title</div>

</div>

The HTML structure is the same for each post. There’s an image, author, and title. The difference between each post is the content.

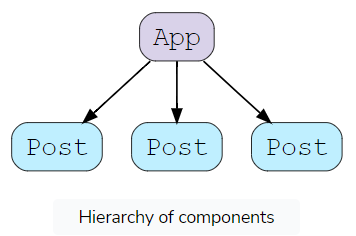
This approach is not scalable. If we want to add more posts, we’d have to duplicate the same HTML structure and replace the content. What if we want to change the structure of a post? We’d have to update each post’s markup. This would take a lot of time if there were hundreds of posts.

## The solution

Components easily solve this issue. We can create a component to store a template of the post. Components are reusable, which allows us to create as many posts as we’d like. If we make a change to the component, the changes are reflected in all uses of the component. A general rule of thumb for defining components is that they should handle one feature in your application. This keeps components manageable.

We’ll dive into the code in a moment. There are some additional things about the components that I want to discuss.

## The App component



Angular apps are made up of components. We’ve been using one from the very beginning. Every application comes with an App component.

In the diagram above, we have a hierarchy of components. The App component is typically the root/parent component of all components. Usually, the job of the app component is to load other components.

Components are allowed to load additional components. The relationship between them becomes a parent-child relationship. The App component is considered the parent component because it’s loading the Post component. The Post components are considered children.

If we’re developing a small app, then it’s perfectly fine to keep everything to one component. In the last section, we didn’t create multiple components. We put everything into the App component.

## Dissecting the App component

A component is made up of 4 files. In the src/app directory, we’ll find the app.component.css, app.component.html, app.component.spec.ts, and app.component.ts files. There’s also the app.module.ts file, but we’ll be ignoring it for now. Modules deserve a lesson of their own.

Here’s an overview of each file:

* app.component.css - The styles for the component
* app.component.html - The template for the component. Also called the “view”
* app.component.spec.ts - The test file for the component
* app.component.ts - The logic for the component (also used to configure the component)

Let’s dive into the app.component.ts file, where we can configure the component.

import { Component } from '@angular/core';  
  
@Component({  
  selector: 'app-root',  
  templateUrl: './app.component.html',  
  styleUrls: ['./app.component.css']  
})  
export class AppComponent {  
  title = 'components';  
}

The very first thing we do is import a decorator called Component. The Component decorator can be used to decorate a class. The Component decorator tells Angular that **the class should be treated as a component.**

We can pass in an object to the @Component decorator to configure the component. If we look inside this example, three options are being set.

The first option is the selector option. This allows us to set the name of the tag. We can use components in templates like regular HTML elements. For example, the App component is used in the src/index.html file.

<!doctype html>  
<html lang="en">  
<head>  
  <meta charset="utf-8">  
  <title>Components</title>  
  <base href="/">  
  <meta name="viewport" content="width=device-width, initial-scale=1">  
  <link rel="icon" type="image/x-icon" href="favicon.ico">  
</head>  
<body>  
  <app-root></app-root>  
</body>  
</html>

In the <body> element, we can see the App component in action. The name of the tag should match the value we set in the selector option.

Here’s the process Angular will go through when starting the app:

1. It will load the components we’ve created in the project.
2. Next, it will scan the contents of the src/index.html file for uses of any of the components that are registered with our application. It will match the name of the tags to the selector option in each component.
3. The <app-root> tag will match with the App component. Angular will interpret this as us wanting to load the App component.
4. A new instance of the App component is created because the tag name matches the App component’s selector option. The component’s template will be inserted into the <app-root> tags.

The selector option is very important. It’s how Angular is able to identify a component in the DOM.

The next option is templateUrl. This is where we can set the URL to the template file. The App component’s templateUrl option is set to ./app.component.html. This is why the template is able to access the properties and methods in the class. Angular will expose them to the template as long as the templateUrl option is properly configured to the correct template.

Lastly, we have the styleUrls option. This is where we can configure the CSS for the component. By default, the App component is tied to the app.component.css file.

The value for this option is an array. We have the option of loading multiple CSS files. Typically, one will suffice. We are going to look at why components have their own CSS file in another lesson. For now, components can come with their own CSS. We’re not forced to define styles for a component in a generic global file.

## Summary

Components are reusable elements with their own template, logic, and styles. Angular will scan the template for components. If it detects a component, it will create a new instance of the class and insert the component’s template. In the upcoming lessons, we’ll look at how to create components.

**Creating a Component**

Learn how to create a new component.

**We'll cover the following**

* [Designing a component](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/4668084050198528#Designing-a-component)

Components can be generated with the Angular CLI. While not mandatory, creating a component using the CLI can save a lot of time. It will save us the trouble of having to create the necessary files and configure the component. We can create a component with the ng generate component <name> command. We’ll use this command to create a Post component.

ng generate component post

This will generate the four files necessary for a component. In the src/app directory, there will be a new directory, called post, with four files called post.component.ts, post.component.html, post.component.css, and post.component.spec.ts.

As you can see, using the command saved us a lot of time. The command will generate the files, but it won’t change the way the app looks. If we were to run the example, the app would still look the same. This is because the command creates the component but doesn’t insert it into the template. We’ll need to take care of that next.

Before attempting to load the component, it’s essential to open the post.component.ts component class file to check the selector option. By default, Angular will set it to app-post. If you’d like, you can freely modify this to any selector you want. We’ll leave it as is.

In the app.component.html file, we’ll replace the current content with the <app-post> component.

<app-post></app-post>  
<app-post></app-post>  
<app-post></app-post>

In the code snippet above, we’re using the component three times. Components are reusable, which means we are allowed to use it as many times as we’d like. Angular will create three instances of the post component. Even though we’re loading this in the app component’s template, it will still work. We’re allowed to load components in other component’s templates.

Let’s update the post.component.html template file.

<div class="post">  
  <img src="image1.png">  
  <div class="author">John</div>  
  <div class="title">Some Title</div>  
</div>

**Note:** The images will be broken. We’ll fix this in a future lesson.

This will output the same HTML three times. The benefit of this is that we’ve outsourced the template to a single file. We can load the template as many times as we’d like using the <app-post> tag. If we modify the post.component.html file, all instances will reflect those changes.

## Designing a component

One last thing to note: technically, we have the freedom to make a component as complex as we’d like. A general rule of thumb is to design a component to implement one feature. In our case, the post component’s job is to output a single post. This keeps things manageable and small.

# Passing Data From Parent to Child Component

Learn how to pass down data from parent to child components.

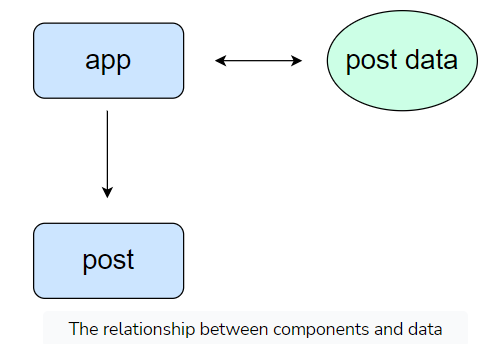
**We'll cover the following**

* [Storing data](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/6074408310931456#Storing-data)
* [Property binding](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/6074408310931456#Property-binding)
* [Input binding](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/6074408310931456#Input-binding)
* [Property alias](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/6074408310931456#Property-alias)

We’ve created our first component. It’s time to make it dynamic by having it load data. First, we must decide where we’re going to define the data.

## Storing data

There are two possible locations in which we can define the data: the post component or the app component. The best approach is to store the data in the parent component. It’s common practice to store data at the highest level possible.



There are a couple of reasons for doing so.

First, it allows the post component to remain reusable. If we stored data in the post component, then we’d have the same data in every instance of the component. We want to be able to output the post component with different data each time. Therefore, it’s better to store the data in the app component and pass it down to the post component. The post component will take the data and output it in the correct locations of the template.

Second, it allows us to share the same data with multiple components. In some cases, you will need to distribute data to numerous components. It’s easier to distribute data when it’s stored in the root component.

Let’s update the app.component.ts file to store the data.

export class AppComponent {  
  posts = [  
    {  
      img: '',  
      author: 'John',  
      title: 'Title #1'  
    },  
    {  
      img: '',  
      author: 'Mike',  
      title: 'Title #2'  
    },  
    {  
      img: '',  
      author: 'Jane',  
      title: 'Title #3'  
    }  
  ]  
}

In the example above, we have an array of posts. Each object is storing the same data. We’re storing the title, author, and image. The img property in each object will be set to an empty string. We’ll update it in a future lesson when we talk about static assets.

## Property binding

The next step is to pass this data down to the post component. We can do so using property binding. Luckily, we already know how property binding works. In the app.component.html file, we’ll update the child components to the following:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

<app-post

··[img]="posts[0].img"

··[author]="posts[0].author"

··[title]="posts[0].title"

></app-post>

<app-post

··[img]="posts[1].img"

··[author]="posts[1].author"

··[title]="posts[1].title"

></app-post>

<app-post

··[img]="posts[2].img"

··[author]="posts[2].author"

··[title]="posts[2].title"

></app-post>

We’re not limited to binding properties to HTML attributes. We can create custom attributes on the child component. After all, the post component is a custom component that we created. We’re allowed to add whatever attributes we want.

The attribute names can be called whatever we like. In this example, we’re using the same names as the property names. While not required, we want our names to accurately and concisely describe the data.

## Input binding

The next thing we need to do is accept the data in the child component. We can do so using what’s called **input binding**. At the top of the post.component.ts file, we’ll import the Input decorator from the @angular/core package.

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

Angular doesn’t know which attributes can be added to our custom element. Therefore, we need to tell it which attributes are valid. The Input decorator helps us with that. It will allow the parent component to pass down data to a child component. In our case, the parent component is the app component, and the child component is the post component.

Here’s how we would use it in the post.component.ts file:

export class PostComponent implements OnInit {  
  @Input() img = '';  
  @Input() author = '';  
  @Input() title = '';  
  
  constructor() { }  
  
  ngOnInit(): void {  
  }  
  
}

In the code snippet above, we’re adding the @Input() decorator before declaring a new property. This will tell Angular that the value for the property can come from the parent. This is called **input binding**. The name of the property must match the name of the attribute we had in the app component.

It’s optional to assign a value to the property. For example, we could write the img property like this:

@Input() img: string;

This works, but if you decide to do so, be sure to set the type of the property because type inference will not work. I prefer to have default values in case the parent component does not send down data.

The last step is to update the post.component.html template with the properties.

<div class="post">  
  <img [src]="img">  
  <div class="author">{{ author }}</div>  
  <div class="title">{{ title }}</div>  
</div>

Passing data from the parent component to a child component is simple in Angular. In the parent component, we used property binding on the child component to supply it with data. In the child component’s class, we can accept data from a parent component with the @Input() decorator.

## Property alias

One final note: it’s possible for the input binding to have a different name. This is what’s known as a **property alias**. You can think of an alias as a nickname for your property.

Let me explain further. The @Input() decorator has one argument—the name of the binding property. If we use this option, we’ll be able to name the property in the child component to anything we’d like.

For example, we have a property called author in the post.component.ts class file. If we want to rename the property to username, we can do the following:

@Input('author') username = '';

This will tell Angular to look for a binding property, called author, from the parent component. It will assign the value from the parent component to a property, called username, in the child component.

If we don’t use this option, the property name and attribute name must match. A property alias can be helpful when you want to avoid conflicting names in a component.

**The ngFor Directive**

Learn how to loop through elements with the ngFor directive.

**We'll cover the following**

* [Capturing the index](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5616772700438528#Capturing-the-index)

We’re able to use the post component to render the same template with different content. As it stands, if we added more objects to the posts array, we’d have to update the template. It would be nice if the template could output the correct number of posts without having to update it as often.

We can make this cleaner using a structural directive called ngFor. This directive will allow us to loop through an array in the template. The element it’s applied to will be output on each iteration.

In the app.component.html file, we can update the template to the following:

1

2

3

4

5

6

<app-post

  \*ngFor="let post of posts"

  [img]="post.img"

  [author]="post.author"

  [title]="post.title"

></app-post>

A couple of things to note: the value for the ngFor directive is let post of posts, and the let post part of the value is the alias for the current item in the loop iteration. We can refer to the item by the variable name.

Afterward, we’re adding the of keyword to tell the directive what array to loop through. In this case, we’re looping through the posts array.

You’ll notice that we’ve updated the property bindings to use the post alias. We can use the item in any of the attributes of the component.

This is much more efficient because we won’t have to update the template whenever a new item is pushed to the posts array. The template will be able to output the correct number of posts.

## Capturing the index

In some cases, you’ll need to get the index of the current loop iteration. You can update the value to include the index, as shown here:

<app-post \*ngFor="let post of posts; let i = index;"></app-post>

In this example, we can add to the value by creating a variable assigned to the index property from the ngFor directive.

**Scoped CSS**

Learn how component-specific CSS works.

**We'll cover the following**

* [Applying styles to the host element](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5904013637189632#Applying-styles-to-the-host-element)
* [Styling the posts](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5904013637189632#Styling-the-posts)

The application has everything it needs. The last thing we need to do is add some styles. We have the option of defining styles in the src/styles.css file. This is what we did in the last project.

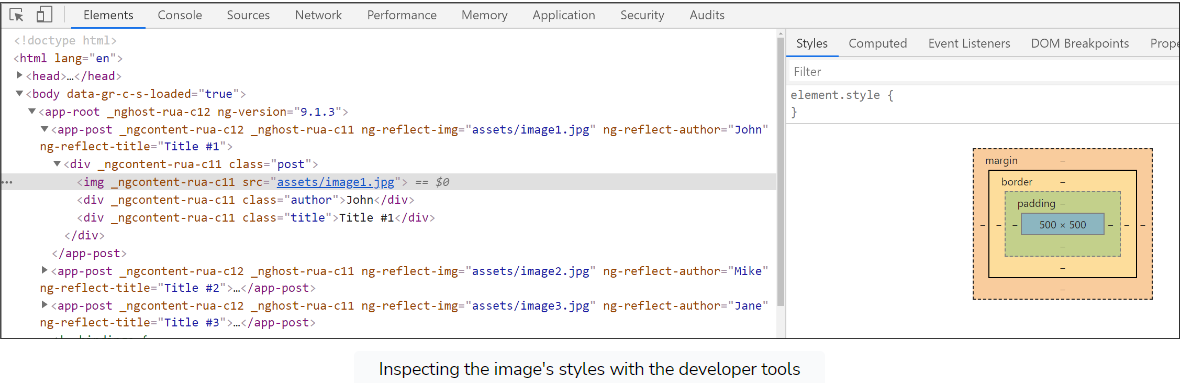
There’s an alternative approach you may want to consider, which uses the component CSS. Every component comes with its CSS files. The app component has the app.component.css file, and the post component has the post.component.css file.

The main benefit of defining styles in the component CSS files is that the styles are scoped. What does this mean? It means that the styles will only affect the template in the component.

For example, in the post.component.html template, we have a <img> tag. If we were to add an img selector in the app.component.css file, the properties wouldn’t affect the post component’s template.

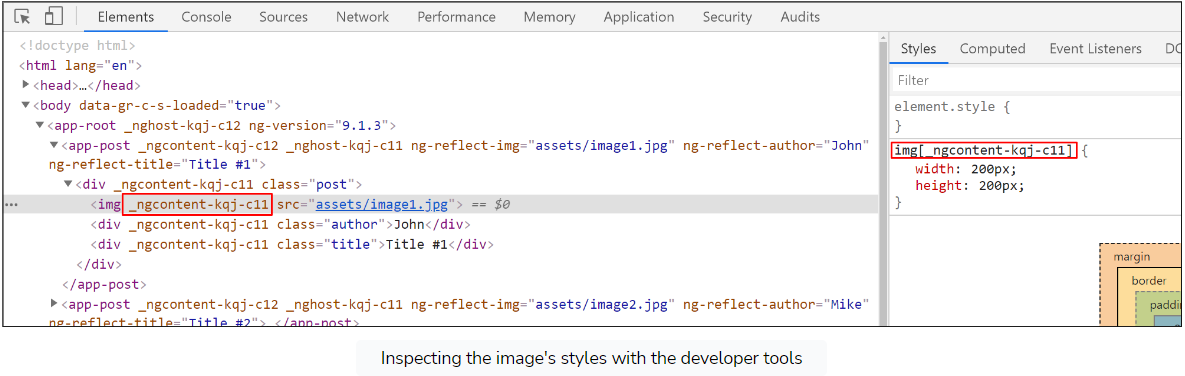
img {  
  width: 200px;  
  height: 200px;  
}

We can use the Chrome Developer Tools to inspect the element’s CSS properties.



The styles we added in the app.component.css file isn’t appearing for the <img> tag in the post component because Angular does some magic behind the scenes. It will limit the scope of the CSS to the component only. We can’t affect external elements like the <body> element or other component’s elements. The styles in the app.component.css file will only apply to the elements in the app.component.html file.

Let’s move the CSS from the app.component.css file to the post.component.css file.



The CSS styles will appear in the developer tools. Angular has modified the selectors to include a custom attribute it assigns to the elements in the template. This is how it scopes CSS.

## Applying styles to the host element

One thing you’ll notice in the developer tools is that every template is wrapped with the component tag. Angular does not replace the component tag with the template. It inserts the template into the tag.

In some cases, we may want to select the tag. For example, we may want to select the <app-post> component tag. If we’re going to apply styles to the <app-post> tag from the post.component.css file, we’ll need to use a special pseudo selector called :host.

:host {  
  font-size: 20px;  
}

We can’t use app-post as the selector because the CSS is scoped to the template. We can’t select the component tag that wraps around the template from the component’s CSS file.