Angular

Agenda

1. Client-side frameworks
2. Angular architecture
3. Angular Setup
4. Components and Templates
5. Lifecycle
6. Component Interaction
7. Forms
8. Observables / RxJS
9. Modules
10. Routing

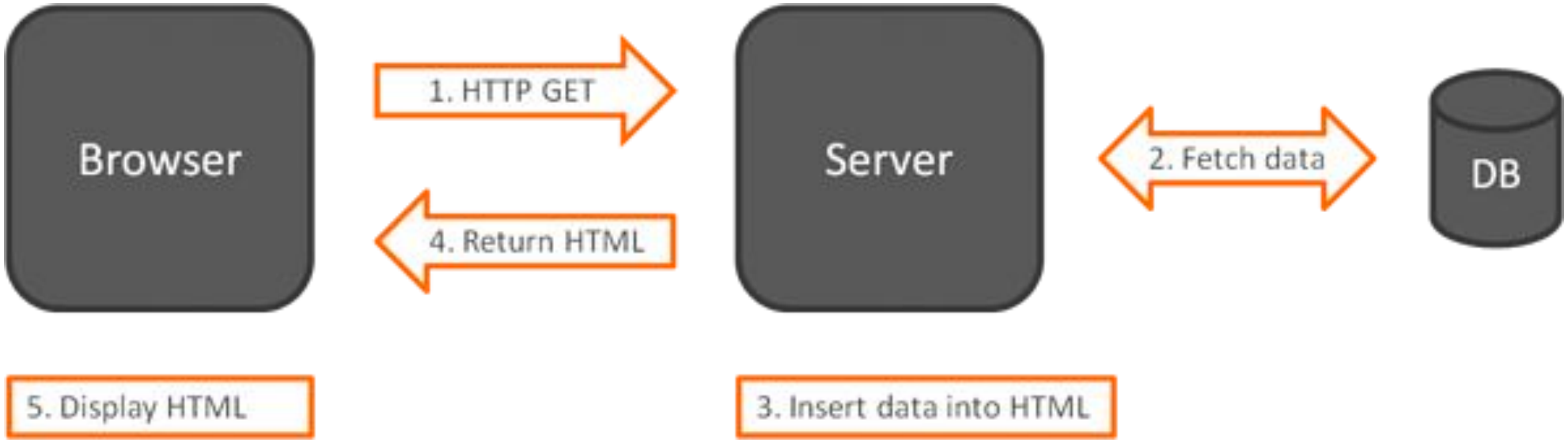
Client-side frameworks

A brief history of JS

- In the 2000s Javascript was just a scripting language – used to create interactions on web pages, commonly using jQuery as only library.
- In 2010 first framework was created – Backbone.js.
- Following the development, language started to adapt – in 2015, ES6 JS version was released.
- Later, full-scale MVC frameworks (such as AngularJS) appeared.

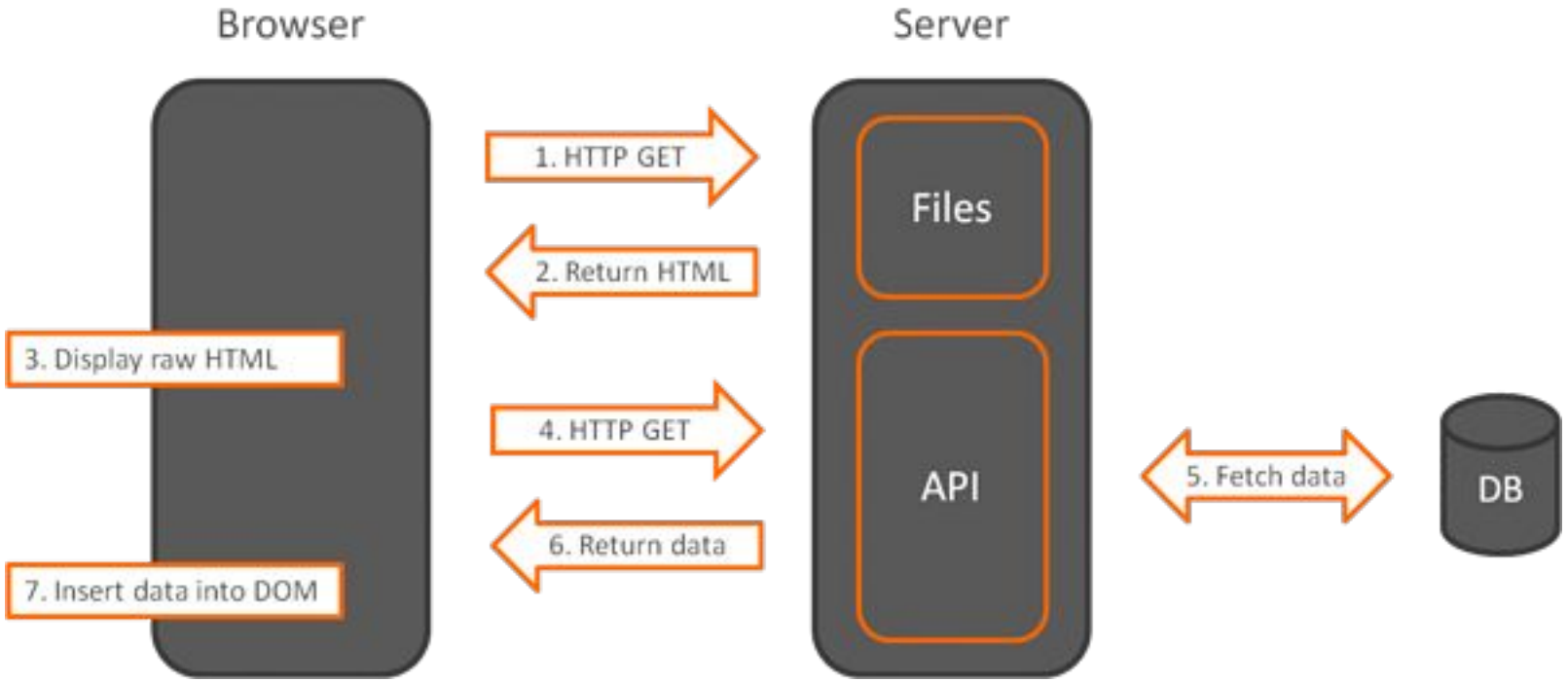
How do traditional web frameworks work?

Traditional web frameworks use a straightforward approach to display the data. When the browser requests the data, the server returns raw HTML, and it is displayed in the browser.



How do client-side frameworks work?

Client-side frameworks operate on the browser level and display the data retrieved from the server. Usually, server side just needs to provide a REST API in order for the application to work. So the server doesn't return the HTML all the time – it mostly returns data in JSON format that the client-side frameworks will operate on.



Why separate client and server?

- Some of the apps require include calculations and processes that can be done on client side, thus reducing the amount of work that the server should handle.
- Separation of logic makes it easier for developers to maintain cleaner codebase.
- Thus client-side frameworks come into attention.

List of popular client-side frameworks

- Angular / Angular.js
- React.JS
- Vue
- Backbone.js
- Ember.js
- Mithril
- Polymer
- Aurelia
- And the list goes on... 😊

Angular architecture

What is Angular?

- Angular is a client-side framework written in **TypeScript**.
- Angular works on the browser level, helps in organizing the view layer and handles the basic routing.
- Angular provides tools to connect to the backend layer which can be written in any language. Java being one of them. 😊

What does Angular consist of?

- Basic building blocks of Angular are **NgModules**.
- A typical app consists of several logical **NgModules**. Usual practice being a root module and several feature modules around it.
- **NgModules** consist of **Components**.
- **Components** define **Views** (HTML with data injections).
- **Components** use **Services** (classes that handle logic and can be injected as dependencies).
- Class separation and binding is done using metadata that defines the associations.

Angular setup

Setup using provided CLI tools

To install Angular, you will need to have **NPM** installed on your machine. **Node.JS** of version 10.9+ is required as well. Once both are installed, perform these steps:

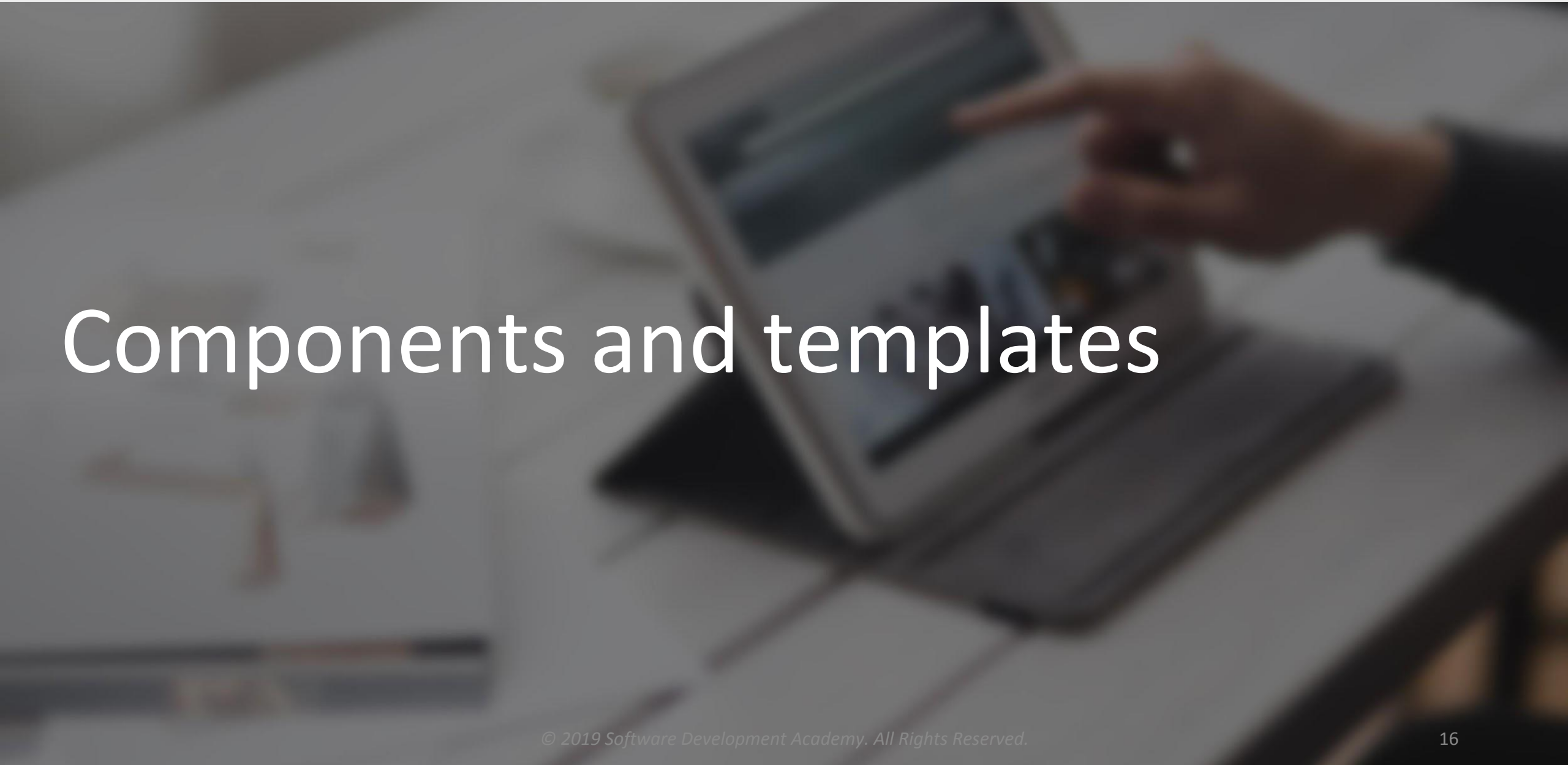
1. Run "**npm install -g @angular/cli**" in Command Prompt or Terminal.
2. Go into your projects folder and run "**ng new my-app**".
3. For more information, visit the Angular project repo on: https://gitlab.com/sda-international/program/java/angular.

Running the application

1. Run command “**cd my-app**” to go to the folder of your new application.
2. Run command “**ng serve --open**”.
3. If everything went as planned, your browser should open a new window!

Your first Angular app

1. The page you see is the **Application Shell**. It is controlled by Angular component named **AppComponent**.
2. As mentioned before, **components** are building blocks of the Angular applications. They display data on screen, listen for your input and do basically everything a user can see.
3. The AppComponent that you already have in your app consists of 3 elements:
   a. **app.component.ts** – component logic, written in TypeScript,
   b. **app.component.html** – component template in HTML,
   c. **app.component.css** – CSS styles of your component.



Components and templates

What is a component?

In your **app.component.ts**, you can see something like the code on the right.

@Component is the annotation that tells Angular that the next defined class is a component. Its parameters in this case are: a HTML selector (in which the component is rendered), the template and the style files.

`export class AppComponent` – this part looks a bit like Java, right? 😊
The **title** is a scoped variable that we outright defined now and that will be available in the template.

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Our first app!';
}
```

What is a component?

In src/index.html, you can see something like this:

<body> <app-root></app-root> </body>

This new **<app-root>** tag is what tells the Angular app to render our newly created component. The **@Component** selector attribute is what connects our component to it! Try changing the selector parameter in your new Angular application.

Angular has a built-in shell for generating new components.
Try the following command:

”**ng generate component dog**”

As you can see a new component was created in your working directory! We’ll return to it a bit later. 😊

What is inside the template?

<h1>Title: {{ title }}</h1>

- Your **app.component.html** might look like the example above.
- First of all – it basically parses clean HTML! 😊
- Interpolation – note the {{ }} annotation. Try changing the **title** variable in the previous file and see what happens!
- Additional other tags are Angular-specific and are covered in the next slides.

Conditionals

- ***ngIf** attribute works as a conditional parameter. Used for example to define if an element should be displayed.
- Setting an **isActive** parameter in the component would allow us to display an element like: **<div *ngIf="isActive">Some Content **.

Loops

- ***ngFor** attribute works as a loop parameter. Used for example to define how a list of elements should be displayed.
- If we have an array of data (let's say a collection of names) in our component, we can access it in the template as **<div *ngFor="let item of items">{{item.name}} **.

Event handlers

- Users need to interact with our page. That is where **event handlers** come with help.
- Let’s assume we have a function named **onClick** in our component file. It would be called from the template like: **<button (click)="onClick($event)">Click me!</button>**.

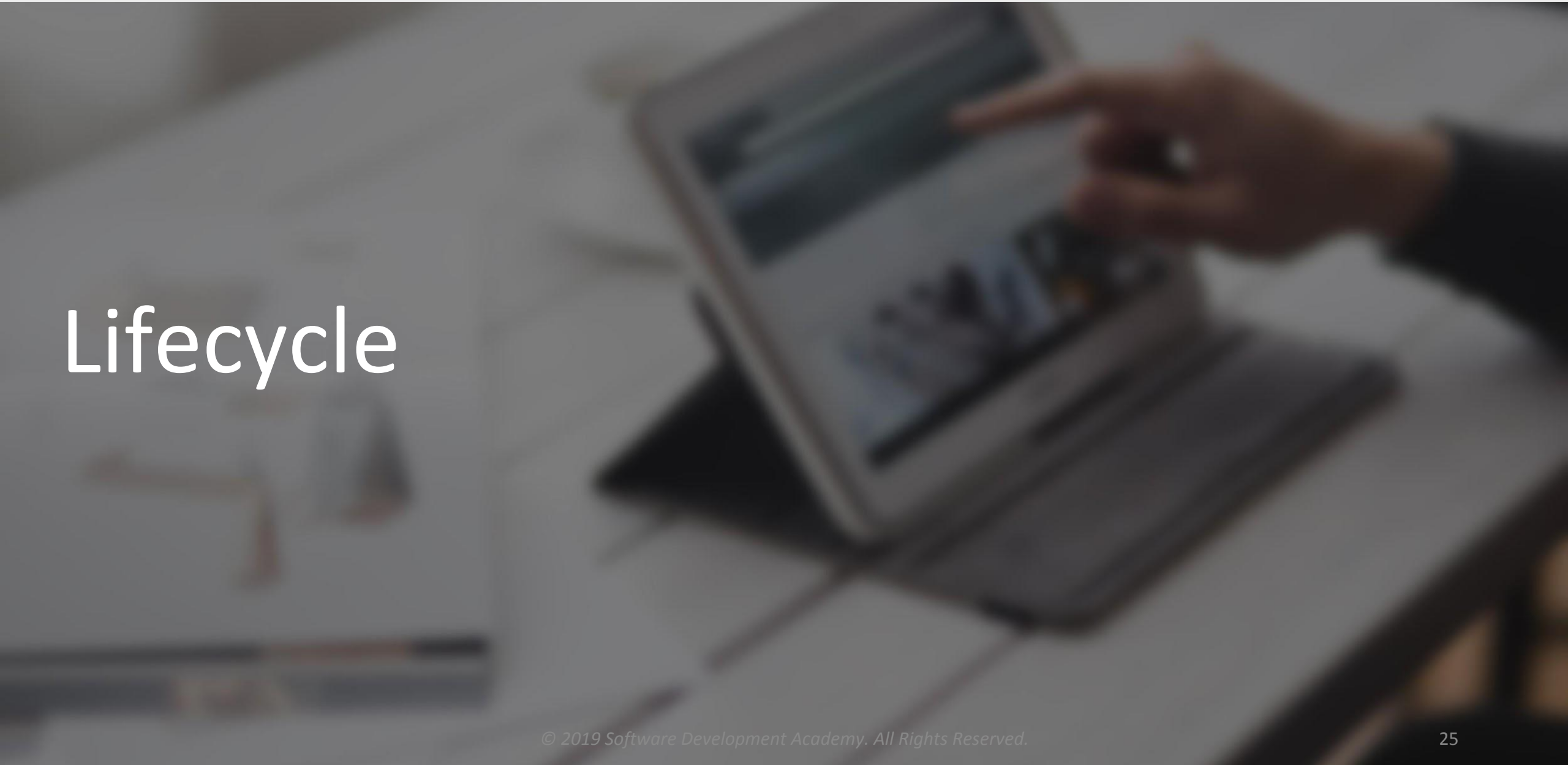
Exercise time!

1. Create an array variable named **doggies** in your **app.component.ts** file and fill it with similar JS objects that each represent a good doggo.
2. Display the names of doggies in the template using the ***ngFor** directive.
3. If there are less than 2 objects in the array variable, display the line “Who let the dogs out?” on the screen using ***ngIf** directive.
4. Add a button to the page that triggers a simple alert and print out some woofs!
5. If you are up for the challenge, add another button that would add a dog (with any generic good name) to the list.

Handling simple user interactions

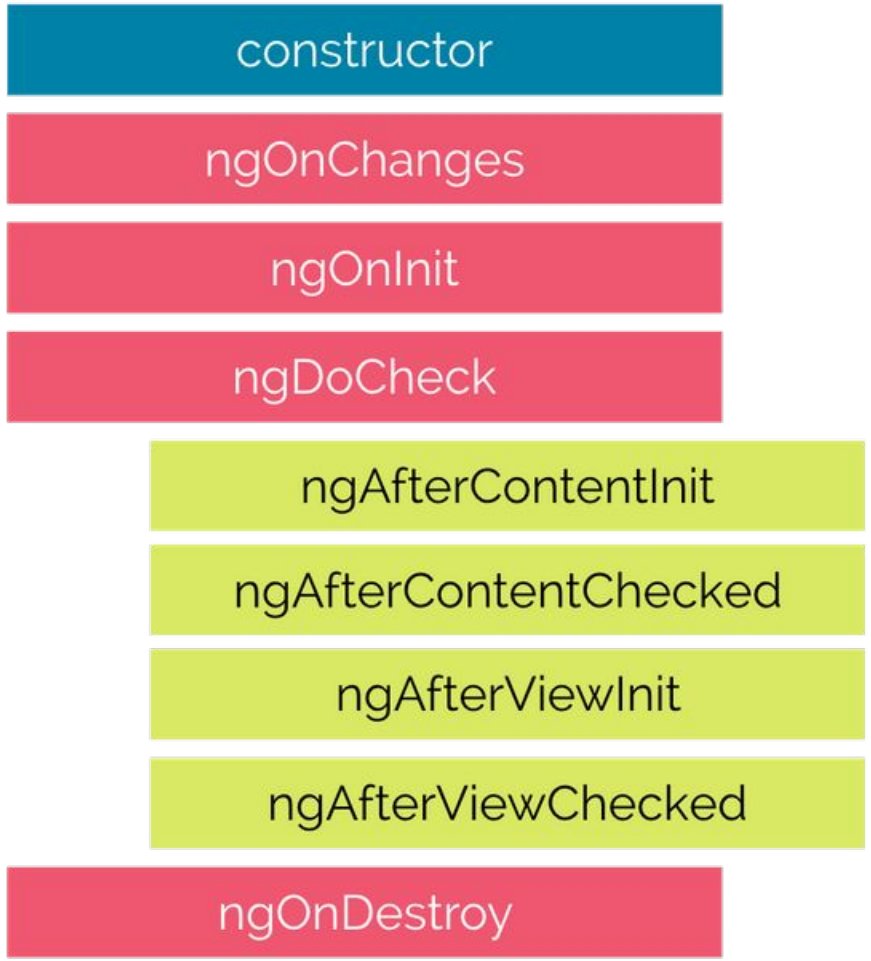
The code below showcases how a simple interaction can be handled – an onClick function triggered by template will add a new doggo to the list. Now, how to adjust this code to fetch a random name from the array?

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  doggos = [ { name: 'Alfred' } ];
  goodNames = [ 'Alfred', 'Joe', 'Friend' ];
  onClick() {
   this.doggos.push(this.goodNames[0]);
  }
 }
```



Lifecycle

Component lifecycle



Angular manages each component's lifecycle and offers various methods to hook into these events. Each component must implement an interface in order to hook into an event, like so:

export class MyClass implements OnInit

Hook actions are fired automatically – for instance, each time the component would render, an **ngOnInit()** function would be called.

When to use lifecycle actions?

- Lifecycle actions are used mainly for fetching or clearing the data in your components.
- For example, if we would like to fetch a list of objects from an API and make the component display them afterwards, we would add the call into **ngOnInit** function of the component.

Adding an onInit hook to your component

With following example a (non-existent yet) function **myInitFunction()** would be called each time the component is about to render.

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit  {
  ngOnInit() { this.myInitFunction(); }
}
```

Adding a call for data in our application

1. Add an **onInit** hook in your AppComponent.
2. On component init, a function named **fetchDoggos()** should be called.
3. The function should simply provide us a list of objects and set its value to **this.doggos**. Later, we will replace it with an API call. 😊
4. Don’t forget to still initialize **this.doggos** as an empty array in the beginning of your component!
5. Exercise time!

Component interactions

Splitting your code into components

Currently, our **AppComponent** handles all the logic – data fetching, displaying, etc. In order to keep it easily maintainable, it should be split into logical parts. Let's use the Angular input binding to solve this!

Parent template code:

```
<child-element *ngFor="let element of parentArray"
[attribute1]="element"
[attribute2]="anyOtherAttribute">
```

In the given example, Parent component will be used as a **container component** – the one that handles all the logic. Child component will be used as a **presentational component** – the one that displays the data. In this case, 2 properties are passed from the parent component.

Splitting your code into components

Child component code:

```
@Component({
    selector: 'child-component',
    template: `<h2>{{attribute1}}</h2><p>{{attribute2}}</p>`
})
export class HeroChildComponent {
@Input() attribute1: Attribute1Type;
@Input() attribute2: string;
}
```

Note the **@Input()** attribute – it is the binding required for the child component to parse the data from the parent. Also note that the properties have their own types - that is a part of validation from TypeScript!

Adding attribute type

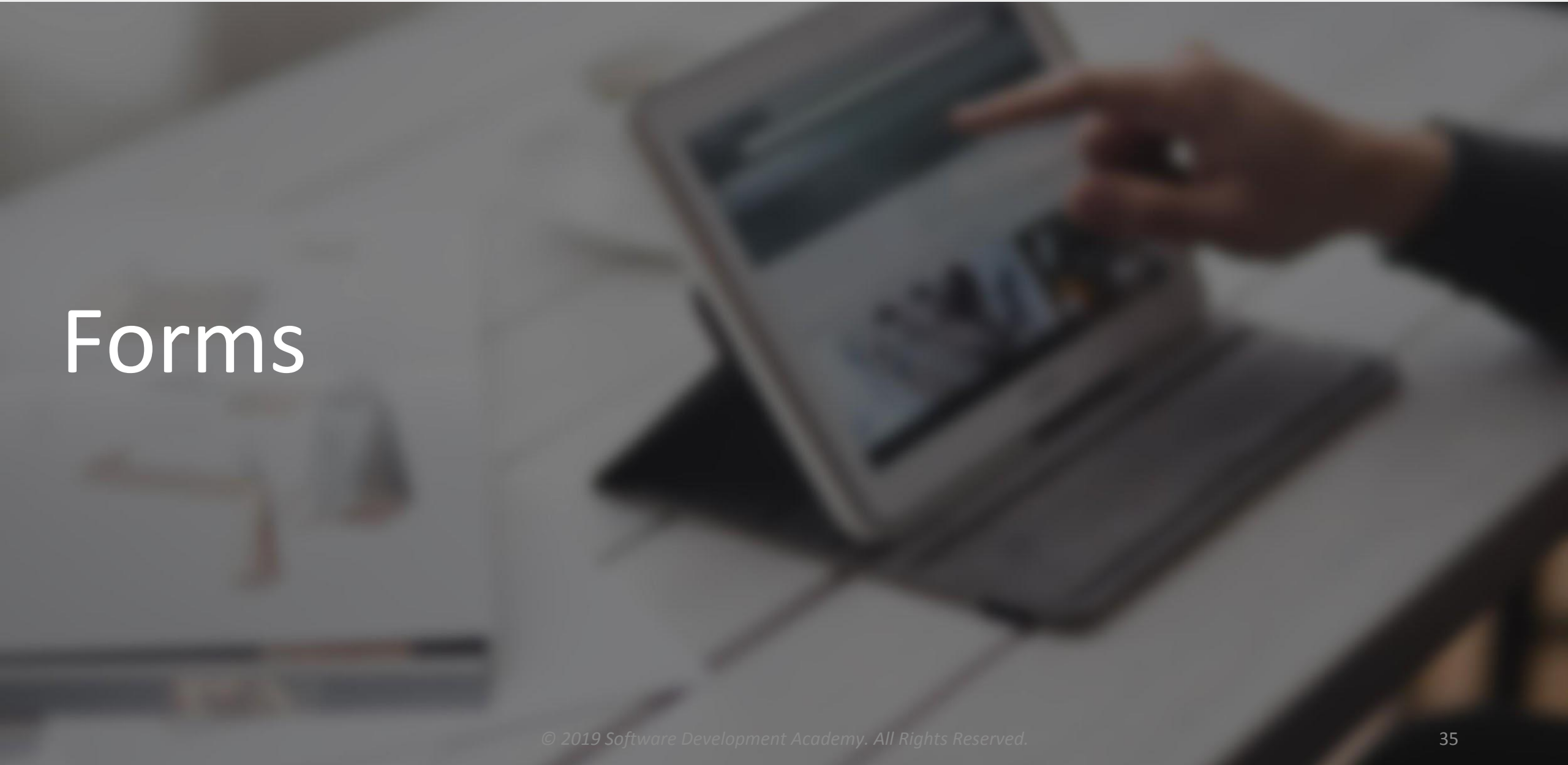
Since Angular is written in TypeScript, it assumes that type validation should be present when any data manipulation is done. For that reason, separate classes representing types should be created for your custom types of data. Of course, generic types like string, integer and array are supported by default.

```
export class DogType {
    name: string;
}
```

For now, our Dogs should probably just have names – let's assume they are of string type, and thus let's create a separate file in our working directory named types.ts and put this code in there.

Displaying a list of components in our app

1. Let's use the **Dog** component that we created before to render the names of our dog as a list element (<li>).
2. Pass the "dog" property to each **Dog** component for each object in your **doggos** array.
3. Validate the type of objects passed to each component using **Input()** attribute.



Forms

Handling user input with forms

- To handle some data from the user we need HTML forms.
- There are two ways of form generation in Angular, **Reactive** and **Template** driven.
- Both allow us to create the visuals for the form, and validate its inputs.
- For the given training, template driven forms will be used, as it is the most straightforward and quickest way of generating a form.
- For more information on Reactive forms, Angular official documentation can be found here: https://angular.io/guide/reactive-forms.
- Now, let's create a new component named **DogForm** in our application!

Adding a form to our application

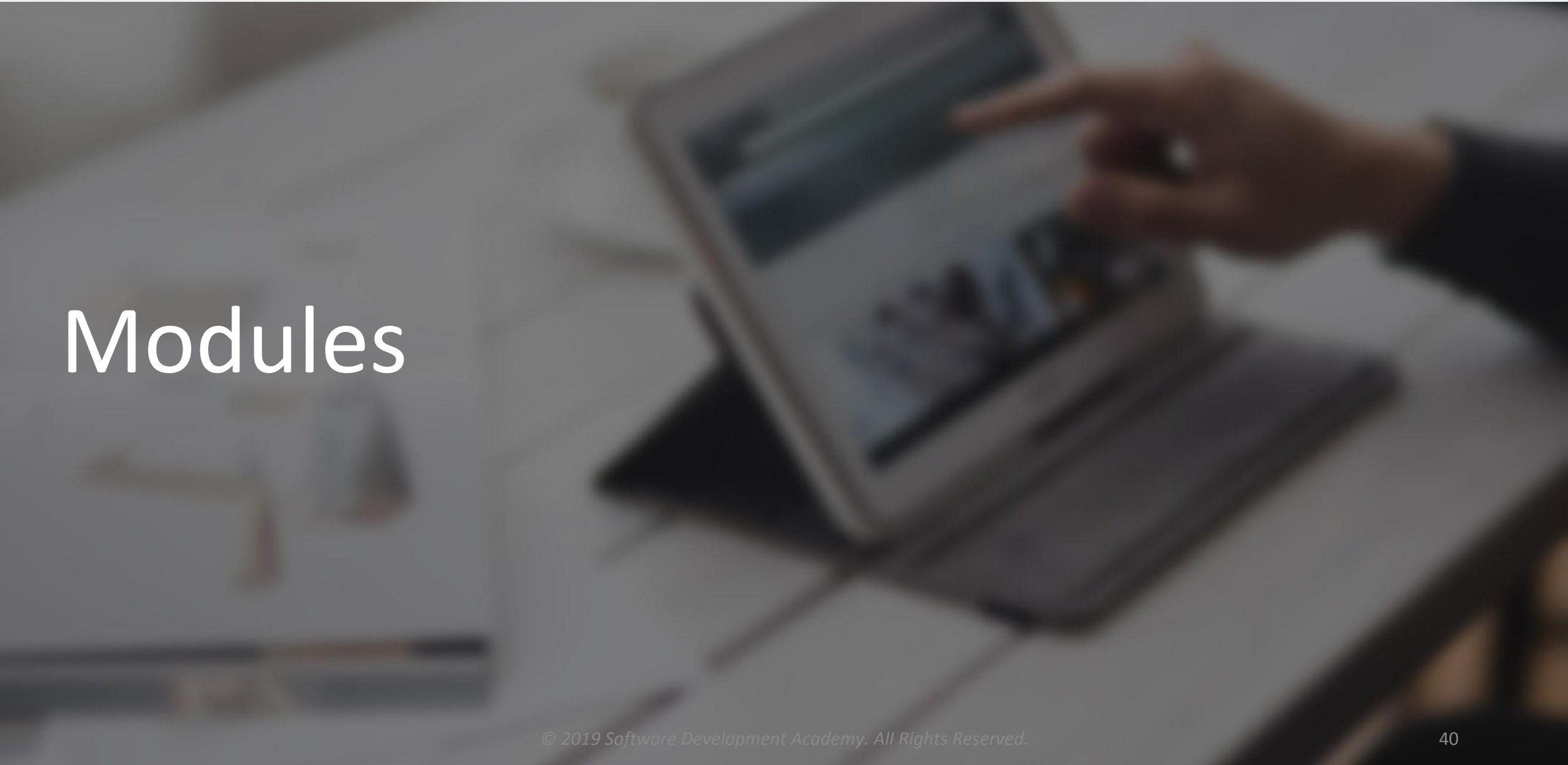
- Create a new component using shell command “**ng generate component HeroForm**”.
- Display the component in the main view and pass a function property **onSubmit** to it.
- **onSubmit** function should accept the event from the form and parse it in effect getting the form values.
- Once the name value is present, a new **Dog** object with the new name should be pushed to the **doggos** array.
- The form itself should have a form tag accepting the submit event, like in the example below.
- Exercise time! 😊

```
<form (ngSubmit)="onSubmit()">
   <label for=”name”>Name</label>
   <input name=“name” id=“name” />
</form>
```

Observables/RxJS

What are observables?

- Sometimes we want to implement some event based scenarios in our application.
- For this purposes **Observable** instances are created. They can subscribe to certain events and perform the actions that you define within them.
- **RxJS** is a standalone library that provides us a defined **Observable** interface.



Modules

What are modules?

- Our application can consist of different modules that can be split to store their own set of components and logic.
- Modules are a great way to organize the application, for example Angular's own libraries are organized as modules. 😊
- Every Angular app has at least one module – the **AppModule**. As the app grows, it might make sense to split it into feature modules.

Splitting application into modules

- Module generation is possible using Angular CLI. For example, we can create a new module using command **“ng generate module NewDashboard**”.
- We can also generate new components specifically for our new module using command **“ng generate module new-dashboard/NewDashboard”**.
- It creates a new folder structure and a new component within our application.
- Now, to use it within our root module, we need to include it in **app.module.ts** within **imports** array**.**

```
@NgModule({
   declarations: [ AppComponent ],
   imports: [ BrowserModule, FormsModule, HttpClientModule,
NewDashboardModule ],
   providers: [],
   bootstrap: [AppComponent]
})
```

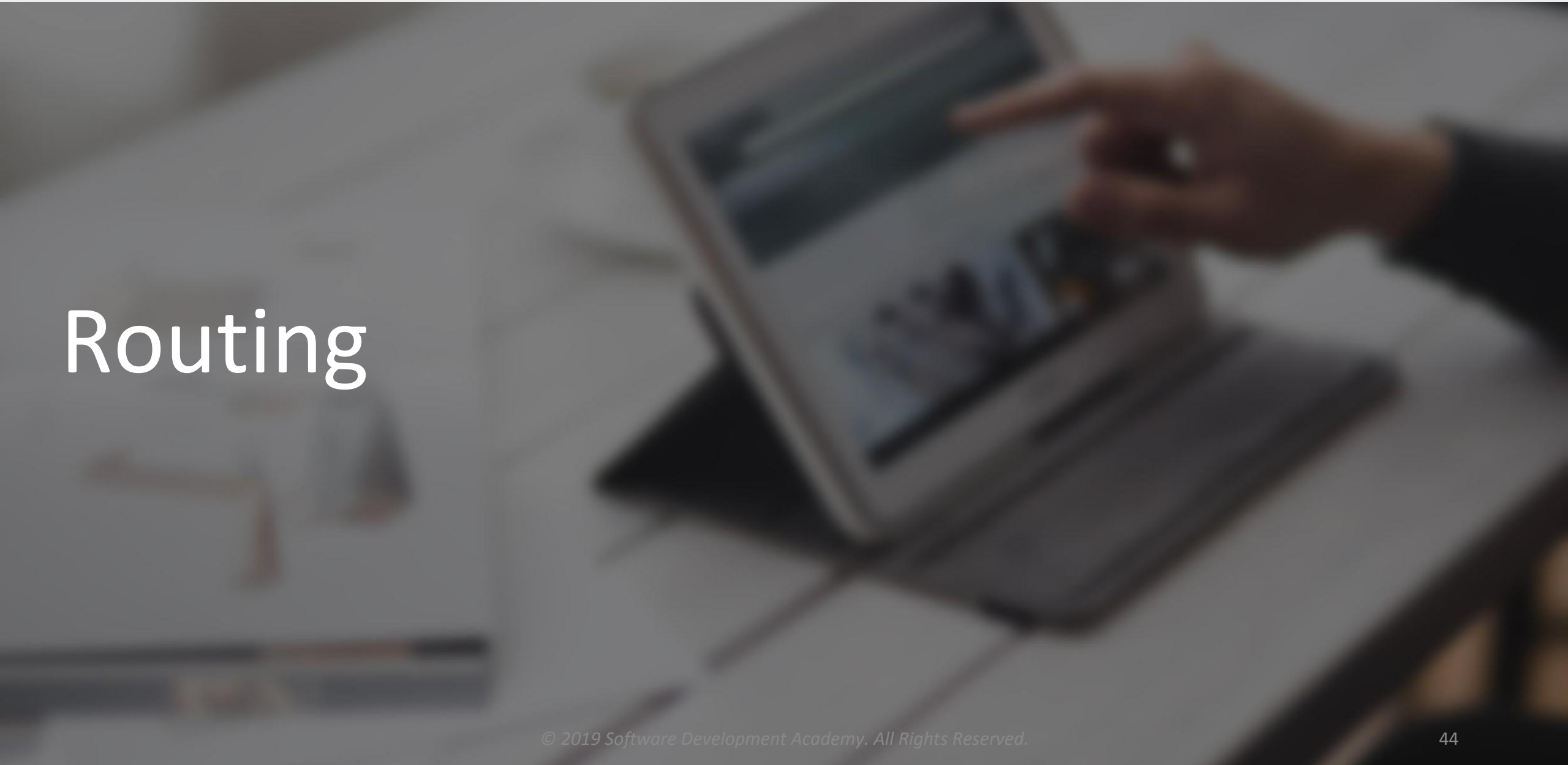
Splitting application into modules

- Of course, our new module should also export some components.
- Export components should be added to the module's own **module.ts** file. In our case it will be **new-dashboard.module.ts**.

  exports: [NewDashboardComponent]

- We are now able to use the new component in our **app.component.html**.

  <new-dashboard></new-dashboard>



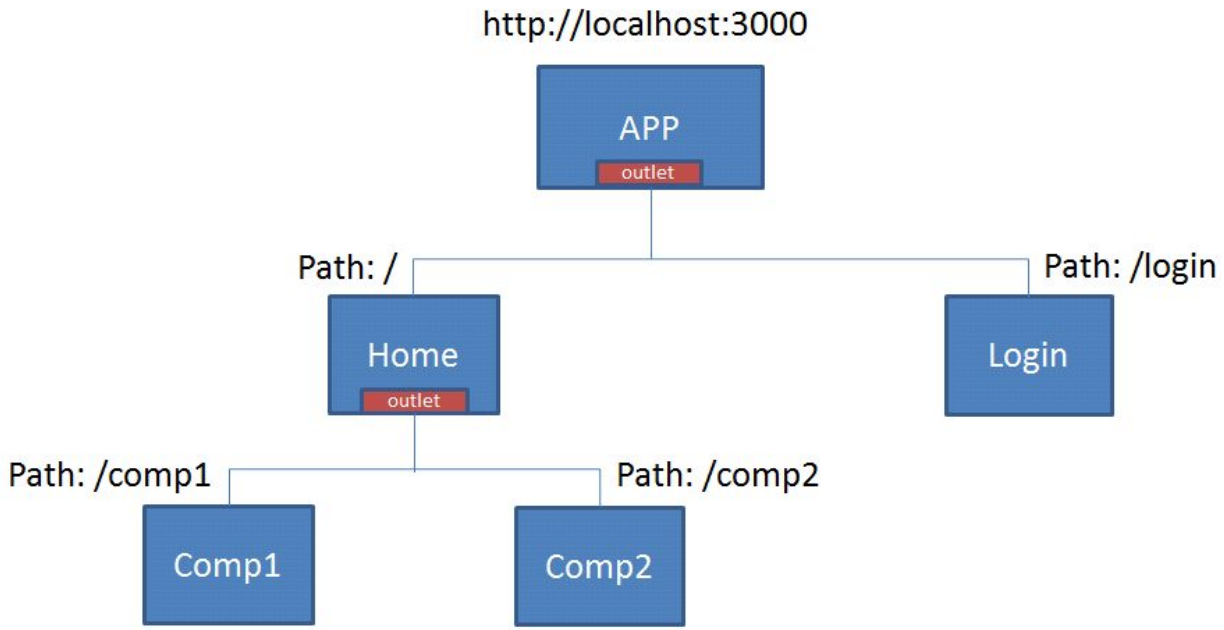
Routing

What about routes?

- Routing helps our application behave like a traditional web page – with URLs and links.
- Angular can accept different routes and handle parameters using its **Routing module**.
- This way, some components will behave like pages of your website.

Routing in Angular

- Components handle routing in Angular.
- When your browser requests a page on your domain, Angular listens to the requests and tries to deliver the required view based on the routes map in the application.



How to add a route?

We need to add our routes in **app.module.ts** and register them.

```
const appRoutes: Routes = [{
    path: ‘dogg/:id’,
    component: DoggComponent,
}, {
    path: '',
    redirectTo: ‘/test',
    pathMatch: 'full' }, {
        path: '**',
        component: PageNotFoundComponent
    }
];

NgModule({
    imports: [
        RouterModule.forRoot(
            appRoutes,)
    ],
    exports: [
        RouterModule
    ]
})
```

How to display routes?

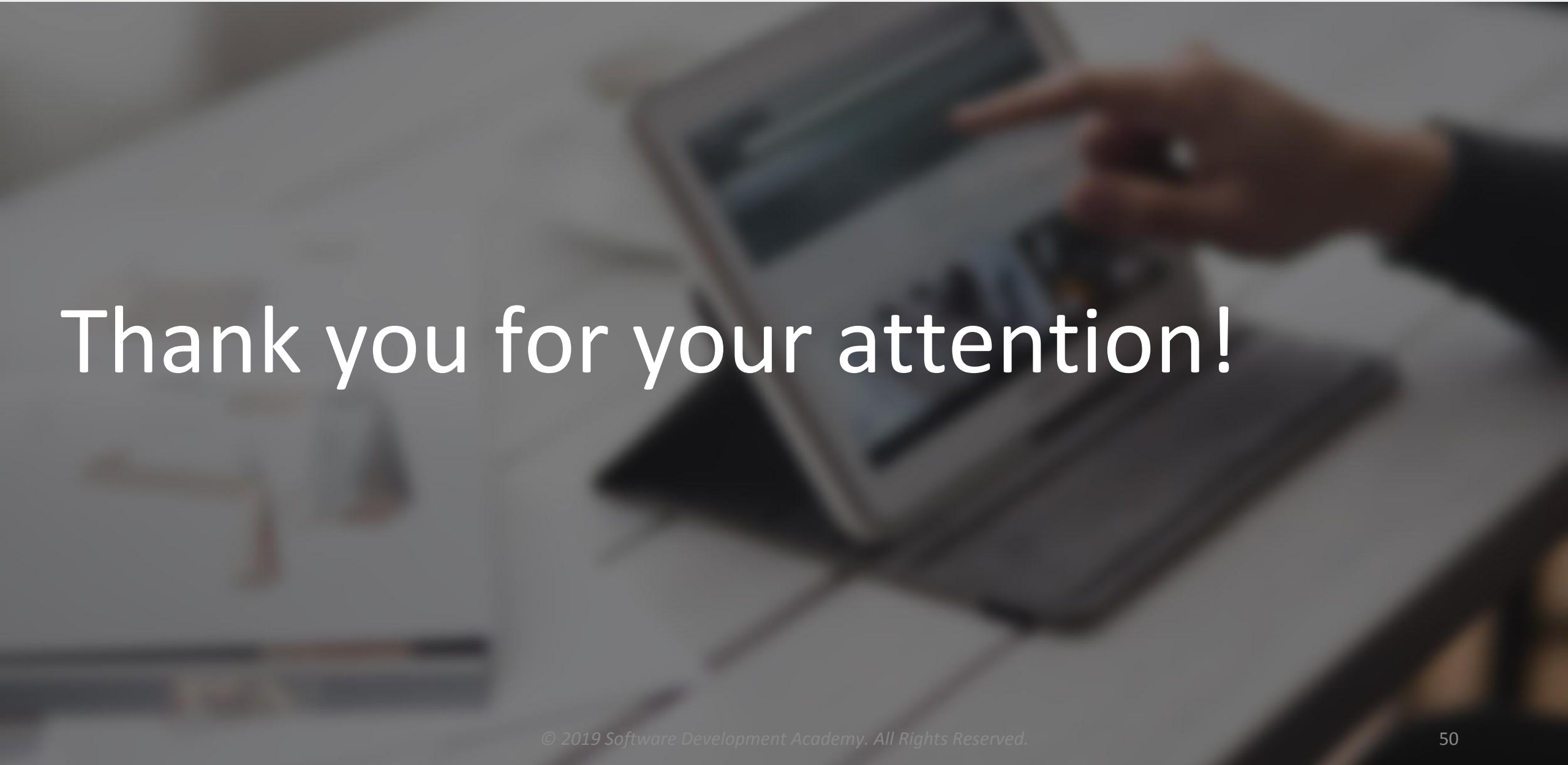
Angular has separate tags designed for displaying the route:

```
<h1>Main Page</h1>
<nav>
  <a routerLink="/test" routerLinkActive="active">Test Page</a>
  <a routerLink="/test2" routerLinkActive="active">Test Page 2</a>
</nav>
<router-outlet>
   { OUR COMPONENT UNDER THE ROUTER WILL BE DISPLAYED HERE }
</router-outlet>
```

Basically, all content that should differ from page to page like page content, details, etc. should go inside router-outlet tag. Content that does not change like navigation, header or footer should stay outside.

Creating a routed web application

1. Add a route of type **/dog/:name** in your **app.module.ts**.
2. When a user clicks on the dog name in your application, a details page for your dog should be shown.
3. At this point you can add additional fields to your Dog objects. How about a link to the picture of the dog? Or perhaps its age or color? Display this data on the details page.
4. If you are up for an additional challenge, add the form to edit your dog details on the Details page.
5. Since this is a final challenge, do not hesitate to also add some shiny CSS to your web page so that your page is looking extra good.
6. Final exercise time! 😊



Thank you for your attention!