ANGULAR 2 in TypeScript

ANGULAR 2 in TypeScript

Overview

Module Component { } · Module Service { } · Module Value 3.1415 · Module Fn λ · Metadata · Template < > · Directive { } · Property Binding · Metadata · Event Binding · Component · Injector · Service

Angular 2 is a framework to help us build client applications for the Web and mobile.

Core Features:

- Speed and Performance
  Angular 2 is dramatically faster than Angular 1 with support for fast initial loads through server-side pre-rendering, offline compile for fast start-up, and ultrafast change detection and view caching for smooth virtual scrolling and snappy view transitions.
- Simple and Expressive
  Make your intention clear using natural, easy-to-write syntax. Reduce complexity for your team: new, structure-rich templates are readable and easy to understand at a glance.
- Seamless Upgrade from Angular 1
  Upgrade your Angular 1 application at your own pace by mixing in Angular 2 components, directives, pipes, services and more by using the ngUpgrade APIs.
- Flexible Development
  The choice of language is up to you. In addition to full support for ES5, TypeScript, and Dart Angular 2 works equally well with ES2015 and other languages that compile to JavaScript.

We write applications by composing HTML templates with Angularized markup, writing component classes to manage those templates, adding application logic in services and handing the top root component to Angular's bootstrapper.

The Module

Module Component { }

Angular apps are modular. In general we assemble our application from many modules. A typical module is a cohesive block of code dedicated to a single purpose. A module exports something of value in that code, typically one thing such as a class.

Most applications have an AppComponent. By convention it's called app.component.ts. When we look inside such a file we'll see an export statement like this one:

app/app.component.ts (excerpt)

```
export class AppComponent() { }
```

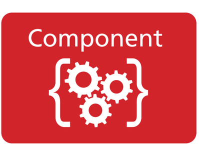
When we need a reference to AppComponent, we import it like this:

app/main.ts (excerpt)

```
import {AppComponent} from './app/component';
```

The import statement tells the system it can get an AppComponent from a module named app.component located in a neighbouring file. The module name is often the same as the filename without its extension. Angular uses an external module loader like SystemJS to bind all the exports and imports together. SystemJS is not the only module loader that will work with Angular 2. Other module loaders, such as WebPack, can be swapped in instead. Which one you take is up to you, but a module loader is crucial for every Angular application.

The Component



A component controls a patch of screen real estate that we could call a view. The shell at the application root with navigation links, the list of Todos, the Todo-editor, they're all views controlled by components. We define a component's application logic – what it does to support a view – inside a class. The class interacts with the view through an API of properties and methods.

app/todo-list.component.ts

```
export class TodoListComponent implements OnInit {
  constructor(private service: TodoService) { }

  todos: Todo[];
  selectedTodo: Todo;

  ngOnInit() {
    this.todos = this.service.getTodos();
  }

  selectTodo(todo: Todo) {
    this.selectedTodo = todo;
  }
}
```

Angular creates, updates and destroys components as the user moves through the application. The developer can take action at each moment in this lifecycle through optional Lifecycle Hooks like *OnInit* (shown above), *AfterContentInit*, *AfterViewInit* or *OnDestroy*.

Library Modules

Library Module: Component { } · Directive { } · Service { } · Value 3.1415 · Fn λ

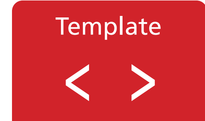
Some modules are libraries of other modules. Angular itself ships a collection of library modules called barrels. Each Angular library is actually a public façade over several logically related private modules.

The angular2/core library is the primary Angular library module from which we get most of what we need. Other important library modules are angular2/common, angular2/http, and angular2/routing. We import what we need from an Angular library module in much the same way as our own modules:

```
import {Component} from 'angular2/core';
```

Unlike our own components, when importing from an Angular library module, the import statement refers to the bare module name, *without a path prefix*.

The Template



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

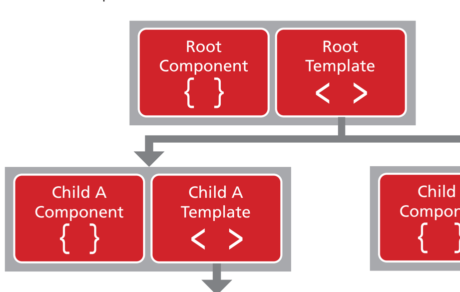
Here's a template for our TodoListComponent, extracted in its own file:

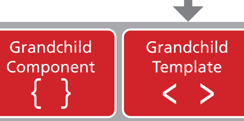
app/todo-list.component.html

```
<h2>Todo List</h2>

<p><i>Pick a Todo from the list</i></p>
<div *ngFor="#todo of todos" (click)="selectTodo(todo)">
  {{todo.name}}
</div>

<todo-detail *ngIf="selectedTodo" [todo]="selectedTodo"></todo-detail>
```

If the template isn't too big, we can embed it directly into the @Component metadata. Please note that such embedded markup has to be placed between two back ticks, especially when it spans more than one line.

Like the <todo-detail> tag from the example above, which represents the view of another component, we can arrange our components and templates in a hierarchical manner to build out our richly featured application.





Angular Metadata

Metadata tells Angular how to process a class. The TodoListComponent is just a class. It's not a component until we tell Angular about it. This is done by attaching metadata to the class. Here's some metadata for TodoListComponent:

app/todo-list.component.ts (metadata)

```
@Component({
  selector: 'todo-list',
  templateUrl: 'app/todo-list.component.html',
  directives: [TodoDetailComponent],
  providers: [TodoService]
})
export class TodoListComponent { ... }
```

The @Component decorator identifies the class immediately below it as a component class. Here we see some of the most commonly used @Component configuration options:

- selector – a css selector that tells Angular to create an instance of this component where it finds a <todo-list> tag in the parent HTML. If the template of the application shell contains

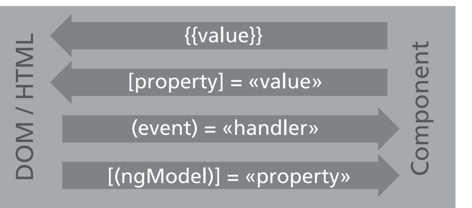
```
<todo-list></todo-list>
```

Angular inserts an instance of the TodoListComponent view between those tags.

- templateURL – the address of this component's template
- template – an inline HTML view directly written within the @Component decorator
- directives – an array of components that this template requires. In our sample it's TodoDetailComponent which defines the view <todo-detail>
- pipes – an array of pipes that this template requires
- providers – an array of dependency injection providers for services this component requires. Classes in this array are typically injected into the component's constructor

We apply other metadata decorators in a similar fashion to guide Angular behaviour. The @Injectable, @Input, @Output, @RouterConfig are few of the more popular decorators.

Data Binding



Angular supports data binding, a mechanism for coordinating parts of a template with parts of a component. We add binding markup to the template HTML to tell Angular how to connect both sides. There are four forms of data binding syntax. Each form has a direction – to the DOM, from the DOM or in both directions – as indicated by the arrows in the diagram. Let's take a variation of our TodoListComponent again to explain the various data bindings:

app/todo-list.component.html (variation)

```
<div>{{todo.name}}</div>
<todo-detail [todo]="selectedTodo"></todo-detail>
<div (click)="selectTodo(todo)"></div>
```

- The "interpolation" displays the component's todo.name property value within the <div> tags.
- The [todo] property binding passes the selectedTodo from the parent TodoListComponent to the todo property of the child TodoDetailComponent.
- The (click) event binding calls the component's selectTodo method when the user clicks on a Todo's name.

Two-way data binding is an important fourth form that combines property and event binding in a single notation using the ngModel directive. Here's an example from the TodoDetailComponent:

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

In two-way data binding, a data property 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.

The Directive

Metadata · Directive { }

Our Angular templates are dynamic. When Angular renders them, it transforms the DOM according to the instructions given by a directive. A directive is a class with directive metadata. We already met one form of directive: the @Component. A component is actually an @Directive with template-oriented features. There are two other kinds of directives as well that we call "structural" and "attribute" directives.

Structural directives alter the layout by adding, removing or replacing elements in the DOM:

```
<div *ngFor="#todo of todos" (click)="selectTodo(todo)">
<todo-detail *ngIf="selectedTodo"></todo-detail>
```

- *ngFor tells Angular to stamp out one <div> per todo in the todos list.
- *ngIf includes the TodoDetail component only if a selected Todo exists.

Attribute directives alter the appearance or behaviour of an existing element. In templates they look like regular HTML attributes, hence the name. The ngModel directive, which implements two-way data binding, is an example of an attribute directive:

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

Angular ships with a few other directives that either alter the layout structure (e.g. ngSwitch) or modify aspects of DOM elements and components (e.g. ngStyle and ngClass).

The Service

"Service" is a broad category encompassing any value, function or feature that our application needs. Almost anything can be a service. A service is typically a class with a narrow, well-defined purpose. It should do something specific and do it well.

There is nothing specifically Angular about services. Angular itself has no definition of a service. There is no ServiceBase class. Yet services are fundamental to any Angular application. Here's a TodoService that fetches Todo items and returns them in a resolved promise. The TodoService depends on a LoggerService and another BackendService that handles the server communication.

app/todo.service.ts

```
export class TodoService {
  constructor(
    private backend: BackendService,
    private logger: Logger) { }

  private todos:Todo[] = [];

  getTodos() {
    this.backend.getAll(Todo).then( (todos:Todo[]) => {
      this.logger.log(`Fetched ${todos.length} todos.`);
      this.todos.push(...todos); // fill cache
    });
    return this.todos;
  }
}
```

Our components are big consumers of services. They depend upon services to handle most chores. They don't fetch data from the server, they don't validate user input, they don't log directly to the console. They delegate such tasks to services. A component's job is to enable the user experience and nothing more. It mediates between the view (rendered by the template) and the application logic (which often includes some notion of a "model"). A good component presents properties and methods for data binding. It delegates everything non-trivial to services.

Upgrading from Angular 1

Angular provides a library named ngUpgrade to help you run both versions of Angular in parallel. This allows you to migrate your application step by step. In particular it allows you to use Angular 1 directives in Angular 2 components (upgrade) and use Angular 2 components in Angular 1 directives (downgrade). You'll find further information about upgrading your existing Web application in the official Angular 2 docs [2] and on the thoughtram blog [6].

Preparation · Migration · Angular 1 · Angular 2

Dependency Injection

"Dependency Injection" is a way to supply a new instance of a class with the fully formed dependencies it requires. Most dependencies are services. Angular uses dependency injection to provide new components with the services they need.

In TypeScript Angular can tell which service a component needs by looking at the types of its constructor parameters. For example, the constructor of our TodoListComponent needs the TodoService:

app/todo-list.component.ts (constructor)

```
constructor(private service: TodoService) { }
```

When Angular creates a component, it first asks an injector for the services that the component requires. An injector maintains a container of service instances that it has previously created. If a requested service instance is not in the container, the injector makes one and adds it to the container before returning the service to Angular. When all requested services have been resolved and returned, Angular can call the component's constructor with those services as arguments. This is what we mean by dependency injection. Although we can register classes at the root level while bootstrapping the application, the preferred way is to register at component level within the @Component metadata, in which case we get a new instance of the service with each new instance of that component and its child components.

app/todo-list.component.ts (excerpt)

```
@Component({
  providers: [TodoService]
})
export class TodoListComponent { ... }
```

Note: Unlike all component classes we have to mark all injectable classes with the @Injectable decorator. This is also true for our TodoService:

app/todo-service.ts (excerpt)

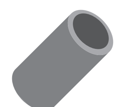
```
@Injectable
export class TodoService { ... }
```

An Angular application needs to be bootstrapped to make dependency injection work and hook up the application lifecycle. This is done in our main class:

app/main.ts (excerpt)

```
import {bootstrap}    from 'angular2/platform/browser';
import {AppComponent} from './app.component';

bootstrap(AppComponent);
```

Pipes



Every application starts out with what seems like a simple task: get data, transform it and show it to users. Getting data could be as simple as creating a local variable or as complex as streaming data over a WebSocket. However, when it comes to transformation, we soon discover that we desire many of the same transformations repeatedly, for example when we need to format dates.

A pipe takes in data as input and transforms it to a desired output. We'll illustrate by transforming a component's due date property into a human-friendly date:

```
<p>Due date of this ToDo item is {{ dueDate | date:"MM/dd/yy" }}</p>
```

Inside the interpolation expression we flow the component's due date value through the pipe separator (|) to the date pipe function on the right. All pipes work this way. You may also provide optional pipe parameters like the exact date format as shown in the example above. Angular comes with a stock set of pipes such as DatePipe, UpperCasePipe, LowerCasePipe, CurrencyPipe and PercentPipe. They are all immediately available for use in any template. Of course, you can write your own custom pipes:

app/exponential-strength.pipe.ts

```
import {Pipe, PipeTransform} from 'angular2/core';

@Pipe({name: 'exponentialStrength'})
export class ExponentialStrengthPipe implements PipeTransform {
  transform(value:number, args:string[]) : any {
    return Math.pow(value, parseInt(args[0] || '1', 10));
  }
}
```

Now we need a component to demonstrate our pipe:

app/exponential-strength.component.ts

```
import {Component} from 'angular2/core';
import {ExponentialStrengthPipe} from './exponential-strength.pipe';

@Component({
  selector: 'power-booster',
  template: '<p>Super power boost: {{2 | exponentialStrength: 10}}</p>',
  pipes: [ExponentialStrengthPipe]
})
export class PowerBooster { }
```

Note that we have to include our pipe in the pipes configuration option of the @Component.

Resources

[1] Angular 2 Website
https://angular.io

[2] Angular 2 Developer Guides
https://angular.io/docs/ts/latest/guide/

[3] Pluralsight Course: Angular 2: First Look by John Papa
https://app.pluralsight.com/library/courses/angular-2-first-look

[4] John Papa: Angular 2 Style Guide
https://github.com/johnpapa/angular-styleguide/blob/master/a2/README.md

[5] bbv Blog: Angular 2
http://blog.bbv.ch/category/angular2/

[6] thoughtram Blog: Angular 2
http://blog.thoughtram.io/categories/angular-2/

[7] RxMarbles: Interactive diagrams of RX Observables
http://rxmarbles.com/

Routing and Navigation

In most applications, users navigate from one view to the next as they perform application tasks. When the browser's URL changes, the router looks for a corresponding route definition from which it can determine the component to display. A router has no route definitions until we configure it. The preferred way to simultaneously create a router and add routes is with an @RouteConfig decorator applied to the router's host component.

In this example, we configure the top-level AppComponent with three route definitions:

app/app.component.ts (excerpt)

```
@Component({ ... })
@RouteConfig([
  {path: 'projects', name: 'Projects', component: ProjectComponent},
  {path: 'todos', name: 'ToDos', component: TodoListComponent},
  {path: 'todos/:id', name: 'ToDo', component: TodoDetailComponent},
])
export class AppComponent { }
```

With the routing in place, we can define the navigation in our AppComponent template:

```
template: `
  <nav>
    <a [routerLink]="['Projects']">Projects</a>
    <a [routerLink]="['ToDos']">ToDos</a>
  </nav>
  <router-outlet></router-outlet>
`
```

The <router-outlet> tag defines the place where the component views will be rendered. When working with routing and navigation, don't forget to include in your index.html:

index.html (excerpt)

```
<script src="node_modules/angular2/bundles/router.dev.js"></script>
```

HTTP Client

HTTP is the primary protocol for browser/server communication. We use the Angular HTTP client to communicate via XMLHttpRequest (XHR) with the server. Let's first have a look at our TodoService:

app/todo-service.ts

```
import {Http, Response} from 'angular2/http';
import {Observable}     from 'rxjs/observable';
import {Todo}           from './Todo';

export class TodoService {
  constructor(private http: Http) { }

  private todoUrl = 'app/todos'; // URL to web api

  getTodos() {
    return this.http.get(this.todoUrl)
                    .map(res => <Todo[]> res.json().data)
                    .catch(this.handleError);
  }

  private handleError(error: Response) {
    console.error(error);
    return Observable.throw(error.json().error || 'Server error');
  }
}
```

As we can see, the Json response from the server is mapped into an array of Todos with the .json() function. The result of getTodos() is a cold observable of Todos, which means the original HTTP request won't go out until something subscribes to the observable. That something is our TodoListComponent:

app/todo-list.component.ts (ngOnInit)

```
ngOnInit() {
  this.service.getTodos()
              .subscribe(
                todos => this.todos = todos,
                error => this.errorMessages = <any>error);}
```

Sending data to the server works like this:

app/todo-service.ts (addTodo)

```
addTodo(name: string): Observable<Todo> {
  let body = JSON.stringify({ name });
  let headers = new Headers({ 'Content-Type': 'application/json' });
  let options = new RequestOptions({ headers: headers });

  return this.http.post(this.todoUrl, body, options)
                  .map(res => <Todo>res.json().data)
                  .catch(this.handleError);
}
```

Back in the TodoListComponent, we see that its addTodo method subscribes to the observable returned by the service's addTodo. When the data arrives, it pushes the new todo object into its own todos array for presentation to the user.

app/todo-list.component.ts (addTodo)

```
addTodo(name: string) {
  if (!name) { return; }

  this.service.addTodo(name)
              .subscribe(
                todo => this.todos.push(todo),
                error => this.errorMessages = <any>error);
}
```

When working with the HTTP client, don't forget to include in your index.html:

index.html (excerpt)

```
<script src="node_modules/angular2/bundles/http.dev.js"></script>
<script src="node_modules/rxjs/bundles/Rx.js"></script>
```

Legal Disclaimer: While we have made every attempt to ensure that the information in this publication has been obtained from reliable sources, bbv Software Services (bbv) is not responsible for any errors or omissions, or for the results obtained from the use of this information. All information is provided with no guarantee of completeness or accuracy, and without warranty of any kind. In no event will bbv or its employees therefore be liable to you or anyone else for any decision made or action taken in reliance on the information in this publication. The information in this publication should not be used as a substitute for consultation with professional bbv advisors. Before making any decision or taking any action, you should consult a bbv professional. The names of actual companies and products (e.g. Google, bbv Software Services) mentioned in this publication may be the trademarks of their respective owners. ©2016 bbv Software Services
This poster is a derivative of Angular's documentation page [2] by Google, used under CC BY 4.0. The poster is licensed under CC BY 4.0 by bbv Software Services AG.

www.bbv.ch · info@bbv.ch