AngularJS Recipes

Fakiolas Marios

AngularJS Recipes

Marios Fakiolas

This book is for sale at http://leanpub.com/angularjs-recipes

This version was published on 2016-05-03

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

To my little princesses Georgia and Xrisi...

Contents

Angular 2 with ES5 . . . . . **1**
Create a simple component 1
Use service's data into component 2
Loop through a service's collection items 3
Use NgFor odd and even indicators to provide different style 4
Validate fast a login form 6

Angular 2 with ES5

Create a simple component

We want to create a custom component named `FirstComponent`. We start by chaining the `Component` and `Class` methods that belong to the global Angular core namespace, `ng.core`. The `Component` method takes a configuration object with two properties. These are `selector` and `template`. So the first one specifies a simple CSS selector for a host HTML element and the second one is the HTML template Angular should render. The `Class` method is where we implement the component itself, giving it properties and methods that bind to the view and further behavior for this specific part of the UI.

```
var FirstComponent = ng.core
    .Component({
        selector: 'first-component',
        template: '<h1>Welcome {{getFullName()}}!</h1>'
    })
    .Class({
        constructor: function() {
            this.firstname = "John";
            this.lastname = "Doe";
        },
        getFullName: function() {
            return this.firstname + ' ' + this.lastname;
        }
    });
```

So we did specify a selector and a template for our component. Then in the `Class` method we declared our demo user's data in the constructor and finally we did concatenate user's first and last name using a custom method. This method is used into our component's template to output user's full name. Since we are done with our custom component we should bootstrap our application by adding this below of our component:

```
document.addEventListener('DOMContentLoaded', function() {
    ng.platform.browser.bootstrap(FirstComponent);
});
```

So what happens there? Angular calls the bootstrap function, reads our component's metadata, finds its selector, locates an element tag named `first-component` and loads our application between those tags. How nice! We should be careful because the order here really matters and the bootstrap code should be placed always after our component. Time to call into action our custom component:

```
<html>
<body>
        <first-component>Loading...</first-component>
</body>
</html>
```

Use service's data into component

We want to create a custom component named `UsersComponent` which is going to manipulate data injected by a custom service. Let's create our service first:

```
function UsersService() {
    this.all = [{
        name: 'John Doe'
    }, {
        name: 'Jane Doe'
    }, {
        name: 'Jack Doe'
    }];
}
```

A simple users service is created so its about time to create our component. We shall use `Component` method `providers` property to pass our service as a dependency and in our `Class` constructor we are going to assign our services data to the component's `users` variable using famous array syntax for dependency injection:

```
var UsersComponent = ng.core
    .Component({
        selector: 'users',
        template: '<h1>Great! It seems we have {{users.length}} users!</h1>',
        providers: [UsersService]
    })
    .Class({
        constructor: [UsersService, function(UsersService) {
            this.users = UsersService.all;
        }]
    });
```

Since we are done with our custom component we should bootstrap our application by adding this below of our component:

```
document.addEventListener('DOMContentLoaded', function() {
    ng.platform.browser.bootstrap(UsersComponent);
});
```

And we are set to go. Time to call into action our custom component:

```
<html>
<body>
        <users>Loading...</users>
</body>
</html>
```

Loop through a service's collection items

If we want to loop through a collection of items passed by a service inti our component we have to use NgFor directive. Let's create our service first:

```
function UsersService() {
    this.all = [{
        name: 'John Doe'
    }, {
        name: 'Jane Doe'
    }, {
        name: 'Jack Doe'
    }];
}
```

A simple users service is created so its about time to create our component. We shall use `Component` method `providers` property to pass our service as a dependency. Also in our `Class` constructor we are going to assign our services data to the component's `users` variable using famous array syntax for dependency injection:

```
var UsersComponent = ng.core
    .Component({
        selector: 'users',
        templateUrl: 'users.html',
        providers: [UsersService]
    })
    .Class({
        constructor: [UsersService, function(UsersService) {
            this.users = UsersService.all;
        }]
    });
```

In order to keep our code clean we are going to create a separate partial named `users.html` which is going to host all required code for the loop:

```
<h4>Our Users ({{users.length}})</h4>
<hr>
<ul>
    <li *ngFor="#user of users; #i = index">{{i + 1}}. {{user.name }}</li>
</ul>
```

As we see `#user` and `#i` are used for each users collection item and index accordingly. Quite straightforward right? Since we are done with our custom component we should bootstrap our application by adding this below of our component:

```
document.addEventListener('DOMContentLoaded', function() {
    ng.platform.browser.bootstrap(UsersComponent);
});
```

And we are set to go. Time to call into action our custom component:

```
<html>
<body>
        <users>Loading...</users>
</body>
</html>
```

Use NgFor odd and even indicators to provide different style

If we want to loop through a collection of items passed by a service into our component we have to use NgFor directive. Let's create our service first then inject it into our component:

```
function UsersService() {
    this.all = [{
        name: 'John Doe'
    }, {
        name: 'Jane Doe'
    }, {
        name: 'Jack Doe'
    }];
}

var UsersComponent = ng.core
    .Component({
        selector: 'users',
        templateUrl: 'users.html',
        providers: [UsersService]
    })
    .Class({
        constructor: [UsersService, function(UsersService) {
            this.users = UsersService.all;
        }]
    });

document.addEventListener('DOMContentLoaded', function() {
    ng.platform.browser.bootstrap(UsersComponent);
});
```

So we created a simple service, we instantiated our component and passed service's data and finally we bootstrapped our application. Next task is to create the `users.html` template of our component:

```
<table class="table table-bordered">
    <thead>
        <tr>
            <th>#</th>
            <th>Name</th>
        </tr>
    </thead>
    <tbody>
        <tr *ngFor="#user of users; #i = index, #even = even, #odd = odd" [ngCla\
ss]="{even: even, odd: odd}">
            <td>{{i + 1}}</td>
            <td>{{user.name }}</td>
        </tr>
```

```
    </tbody>
</table>
```

Angular `NgFor` directive offers some very useful exported values such as `index`, `odd` and `even`. In order to use them we have to assign those to specific variables and then use them in a `NgClass` directive accordingly. Remember it is very important to assign first exported values to specific referrers while declaring `*ngFor` so `NgClass` can have access to them right after. Simple right? Time to call into action our custom component:

```
<html>
<body>
        <users>Loading...</users>
</body>
</html>
```

Validate fast a login form

If we want to validate a simple form in Angular 2 things are quite straightforward. Let's start by creating our component:

```
var UsersComponent = ng.core
    .Component({
        selector: 'user-form',
        templateUrl: 'form.html'
    })
    .Class({
        constructor: function() {
            this.user = {
                username: '',
                password: ''
            };
            this.submitted = false;
        },
        onSubmit: function(valid) {
            this.submitted = true;

            if (valid) {
                console.log('valid', this.user);
            } else {
                console.log('invalid');
            }
```

```
        }
    });

document.addEventListener('DOMContentLoaded', function() {
    ng.platform.browser.bootstrap(UsersComponent);
});
```

So we created a very simple component in ES5, where we registered in its `contructor` a user entity with some very common credentials for sign in such as username and password and a flag variable named `submitted` which becomes `true` after our form is submitted. Apart from those we declared a method named `onSubmit()` to handle our form's submission. In the end we bootstrapped our component as we normally do. As we understand in our js code things are quite simple and this is its final format.

Let's move to our html `form.html` template where we have to be a little more cautious. We are going to use `novalidate` attribute inside opening form tag so we can handle all input fields validation on our own and prevent HTML5 default validation. Let's create first a simple html form for the occasion with some Bootstrap 3.3.x flavor:

```
<form novalidate>
    <div class="form-group">
        <label for="username">Username</label>
        <input type="text" name="username" class="form-control">
    </div>
    <div class="form-group">
        <label for="password">Pasword</label>
        <input type="password" name="password" class="form-control">
    </div>
    <div class="form-group">
        <button type="submit" class="btn btn-success">Submit</button>
    </div>
</form>
```

This is a basic HTML5 form with some fundamental Bootstrap 3.3.x classes for decoration purposes only, without any Angular 2 directive. So, in order to validate our form we should watch for the `onsubmit` event and this event will be triggered either if we hit enter while in an input field either if we press the submit button. To do this we have to use `NgForm` directive Angular 2 offers.

The `NgForm` directive supplements the form element with additional features. It collects `Controls` (elements identified by an `ngControl` directive) and monitors their properties including their validity. It also has its own `valid` property which is true only if every contained control is valid. This is how we are going to evaluate constantly if our form is valid or not. Let's take advantage of what we learned:

```
<form #loginForm="ngForm" (ngSubmit)="onSubmit(loginForm.valid)" novalidate>
        ...
    <div class="form-group">
        <button type="submit" class="btn btn-success" [disabled]="!loginForm.val\
id">Submit</button>
    </div>
</form>
```

First we defined a template local variable named, `#loginForm` and and initialized it with the value, `ngForm`. So `loginForm` references to our login form as a whole and we can track its validity as we discussed before. How we can use this? By disabling the submit button using its `disabled` property with our form's state via `loginForm` reference:

```
<button type="submit" class="btn btn-success" [disabled]="!loginForm.valid">Subm\
it</button>
```

Nice! We wanted a handler for the `onsubmit` event so we used `NgSubmit` directive and bind it to our component's `onSubmit()` method passing our form's state as a parameter.

```
<form #loginForm="ngForm" (ngSubmit)="onSubmit(loginForm.valid)" novalidate>
...
```

Time to deal with our input fields. We will make both of them `required` and we are going to provide error messages accordingly. Let's start with the username field. We shall add the `required` property to make it required. Then we shall use the `ngModel` directive for two-way data binding with our component's `user` credentials. We will create a template local variable named `#username` which is going to become a handle on `ngControl` object for this input box for validation tracking and this is what we will use to toggle the appearance of error messages. The `NgControl` is one of a family of `NgForm` directives that can only be applied to a control within a form tag so that's the way to validate our input fields and provide custom error messages. Let's apply all these:

```
<div class="form-group">
    <label for="username">Username</label>
    <input type="text" name="username" class="form-control" [(ngModel)]="user.us\
ername" ngControl="username" #username="ngForm" required>
        <p *ngIf="username.dirty && username.invalid" class="help-block">
                <span class="glyphicon glyphicon-remove"></span> Your username is required
   </p>
</div>
```

So we created a local variable named `#username` and assigned it to `ngForm` as we did with the parent form. Now `username` is a handle to `NgControl` and we can track its valid state. This is how we are going to show/hide our custom error warning using `NgIf` directive.

```
<p *ngIf="username.dirty && username.invalid" class="help-block">
...
```

Be careful! At the very begining our `username` field is invalid because it is empty. In order to tackle the early appearance of the error message we used the `NgIf` directive by checking the ngControl object's `dirty` and `invalid` property. So far so good. Let's apply same logic to our password field:

```
<form #loginForm="ngForm" (ngSubmit)="onSubmit(loginForm.valid)" novalidate>
    <div class="form-group">
        <label for="username">Username</label>
        <input type="text" name="username" class="form-control" [(ngModel)]="use\
r.username" ngControl="username" #username="ngForm" required>
        <p *ngIf="username.dirty && username.invalid" class="help-block">
            <span class="glyphicon glyphicon-remove"></span> Your username is re\
quired
        </p>
    </div>
    <div class="form-group">
        <label for="password">Pasword</label>
        <input type="password" name="password" class="form-control" [(ngModel)]=\
"user.password" ngControl="password" #password="ngForm" required>
        <p *ngIf="password.dirty && password.invalid" class="help-block">
            <span class="glyphicon glyphicon-remove"></span> Your password is re\
quired
        </p>
    </div>
    <div class="form-group">
        <button type="submit" class="btn btn-success" [disabled]="!loginForm.val\
id">Submit</button>
    </div>
</form>
```

And this is the final format of our login form's template. For demonstration purposes we will toggle the appearance of a success message after the form gets submitted successfully using `NgIf` directive watching our component's `submitted` property:

```
<div *ngIf="submitted" class="alert alert-success">
    <h2>Your Credentials</h2>
    <p>Username: {{user.username}}</p>
    <p>Password: {{user.password}}</p>
</div>
```

Time to call into action our custom component:

```
<html>
<body>
        <user-form>Loading...</user-form>
</body>
</html>
```