Angular Building Blocks

Downloads: http://bit.ly/centric-ng2

Agenda

Datum	Onderwerp
1-2-2017	TypeScript Introduction
15-2-2017	Advanced TypeScript
1-3-2017	Angular Introduction
15-3-2017	Angular Building Blocks
29-3-2017	Components
12-4-2017	Ajax
26-4-2017	Data Entry
10-5-2017	Single Page Applications

Downloads: http://bit.ly/centric-ng2

What are we going to cover?

The Angular building blocks

- Zones
- Modules
- Directives
- Renderer
- Pipes
- Services
- Dependency injection

NB: Components and templates will be covered in a separate module.

Zones

A zone is an execution context that track asynchronous behavior.

- Monkey patches methods/classes like setTimeout(), addEventListener() and XMLHttpRequest.

Use by Angular to track when application code has finished running.

- Triggers a change detection cycle

Angular provides the injectable NgZone class.

- Use runOutsideAngular() to prevent triggering a change detection cycle

Modules

Each Angular application consists of at least one root module.

- This is the module that is bootstrapped.

Different functional parts of the application should be split into different feature modules.

- Helps with testing and reuse.
- Allows for lazy loading of code using the Angular-Router.

Create a shared module with common features.

Generate using the CLI.

- ng generate module <name>

Module properties

Declarations

- List of types that belong to this module

Providers

- List of types that can be used with dependency injection

Imports

- List of modules being imported

Exports

- List types that will be available where this module is imported

Main modules versus feature modules

Main module	Feature module
Imports BrowserModule	Imports CommonModule
Normally imports other modules	May import other modules
Doesn’t export	Exports modules, components, directives and pipes
Provides services for the entire application	Provides services for the entire application unless the module is lazily loaded
Contains bootstrap components	No bootstrap components

Using Web Components with Angular

Web Components can conflict with Angular template validation.

- Suppress errors by adding CUSTOM_ELEMENTS_SCHEMA to the modules schemas

Main module

Allows for Web Components

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule, CUSTOM_ELEMENTS_SCHEMA } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpModule } from '@angular/http';

import { MoviesModule } from './movies/movies.module';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, MoviesModule
  ],
  providers: [],
  bootstrap: [AppComponent],
  schemas:[CUSTOM_ELEMENTS_SCHEMA]
})
export class AppModule { }
```

Feature module

Exports the MovieComponent

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { MoviesComponent }
  from './movies/movies.component';

@NgModule({
  imports: [
    CommonModule
  ],
  declarations: [MoviesComponent],
  exports: [MoviesComponent]
})
export class MoviesModule { }
```

Directives

Directives can be used to manipulate a DOM element.

There are three categories:

- Attribute directives
- Structural directives
- Components

All directives can use the lifecycle hooks when needed.

Attribute directives

An attribute directive adds behavior to a single DOM element.

The constructor is passed an ElementRef which has a nativeElement property.

- The nativeElement can be null when rendering on the server or a web worker.

Angular ships with a number of standard attribute directives:

- ngClass
- ngStyle

Generate using the CLI.

- ng generate directive <name>

Defining the attribute directive

Note: This is suboptimal code.

```
import {Directive, ElementRef} from '@angular/core';

@Directive({selector: '[appShrink]'})
export class ShrinkDirective {
  constructor(private element : ElementRef) {
    element.nativeElement.addEventListener('click', this.onClick);
  }
  onClick = () => {
    const {style} = this.element.nativeElement;
    style.maxHeight = '100px';
    style.overflow = 'hidden';
  }
  ngOnDestroy() {
    this.element.nativeElement
      .removeEventListener('click', this.onClick);
  }
}
```

Using the attribute directive

```
<div appShrink>
  <p>
    Lorem ipsum dolor sit amet, consectetur adipisicing elit.
    Impedit suscipit, mollitia maiores porro illum nam libero.
    Dolorem quam, minima facilis non. Accusamus, recusandae, nam.
    Id dolorem iusto suscipit officia, cumque?
  </p>
</div>
```

Structural directives

Structural directives add or remove DOM elements.

- Use the * template binding syntax

View elements are defined as a template or a component.

- When using a component add it to the modules entryComponents array

Angular ships with a number of standard structural directives:

- ngIf
- ngFor
- ngSwitch

Defining the structural directive

```
import {Directive, TemplateRef, ViewContainerRef, Input}
  from '@angular/core';

@Directive({selector: '[appIf]'})
export class IfDirective {
  constructor(
    private viewContainer: ViewContainerRef,
    private template: TemplateRef<any>) {}

  @Input() set appIf(show) {
    if (show && !this.viewContainer.length) {
      this.viewContainer.createEmbeddedView(this.template);
    } else if (!show && this.viewContainer.length) {
      this.viewContainer.clear();
    }
  }
}
```

Using the structural directive

```
<label>
  <input type="checkbox" [(ngModel)]="showText">
  Show text
</label>

<div *appIf="showText">
  <p>
    Lorem ipsum dolor sit amet, consectetur adipisicing elit.
    Impedit suscipit, mollitia maiores porro illum nam libero.
    Dolorem quam, minima facilis non. Accusamus, recusandae.
    Id dolorem iusto suscipit officia, cumque?
  </p>
</div>
```

The Renderer

The renderer is an abstraction layer between the directives and the platform.

- Different platforms like DOM or Web Worker have different implementations

Inject a Renderer and use it to manipulate a native element.

Using the Renderer

```
import {Directive, Renderer, ElementRef} from '@angular/core';

@Directive({selector: '[appShrink]'})
export class ShrinkDirective {
  private removeClickHandler: Function;

  constructor(private element : ElementRef, private renderer: Renderer) { }

  ngOnInit(){
    this.removeClickHandler =  this.renderer
      .listen(this.element.nativeElement, 'click', this.onClick);
  }
  onClick = () => {
    this.renderer.setElementStyle(this.element.nativeElement, 'max-height', '100px')
    this.renderer.setElementStyle(this.element.nativeElement, 'overflow', 'hidden')
  }
  ngOnDestroy() {
    this.removeClickHandler();
  }
}
```

Pipes

Pipes can be used to transform values.

- Using an optional argument
- The argument can be an array

Generate using the CLI.

- ng generate pipe <name>

Pure versus impure pipes

Pure pipes are only executed if the input changes using an === operator.

- Faster and the preferred way
- Will not update when properties of an object have changed

Impure pipes execute every change detection cycle.

- Can become a performance issue
- Set pure to false on the decorator when needed

Defining the pipe

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

@Pipe({
  name: 'time'
})
export class TimePipe implements PipeTransform {
  transform(value: Date, args?: any): any {
    if (args === 'local') {
      return value.toLocaleTimeString();
    } else {
      return value.toTimeString();
    }
  }
}
```

Using the pipe

```
<div>
  {{now | time}}
</div>
<div>
  {{now | time:'local'}}
</div>
```

Services

Services are the collection of remaining classes the application needs.

- A service is just a class that is normally injected using dependency injection

Generate using the CLI.

- ng generate service <name>

Standard services

There are many services available out of the box:

- Location
- Http
- Compiler
- Injector
- Sanitizer
- Renderer
- ComponentFactoryResolver
- …

Custom Services

Create custom services for your own application behavior.

Add the Injectable decorator so you can inject other services into the constructor.

- It’s recommended to always decorate your services with @Injectable() even if there are no dependencies yet

Custom service example

```
import { Injectable } from '@angular/core';
import { Http } from '@angular/http';
import 'rxjs/add/operator/map';

@Injectable()
export class MoviesService {
  constructor(private http: Http) { }

  getMovies() {
    return this.http.get('movies.json')
      .map(resp => resp.json());
  }
}
```

Dependency injection

Dependency injection (DI) is used to decouple different parts of the application.

- The different dependencies are automatically inserted at runtime
- When unit-testing fake dependencies can be used instead

Each component has an injector service.

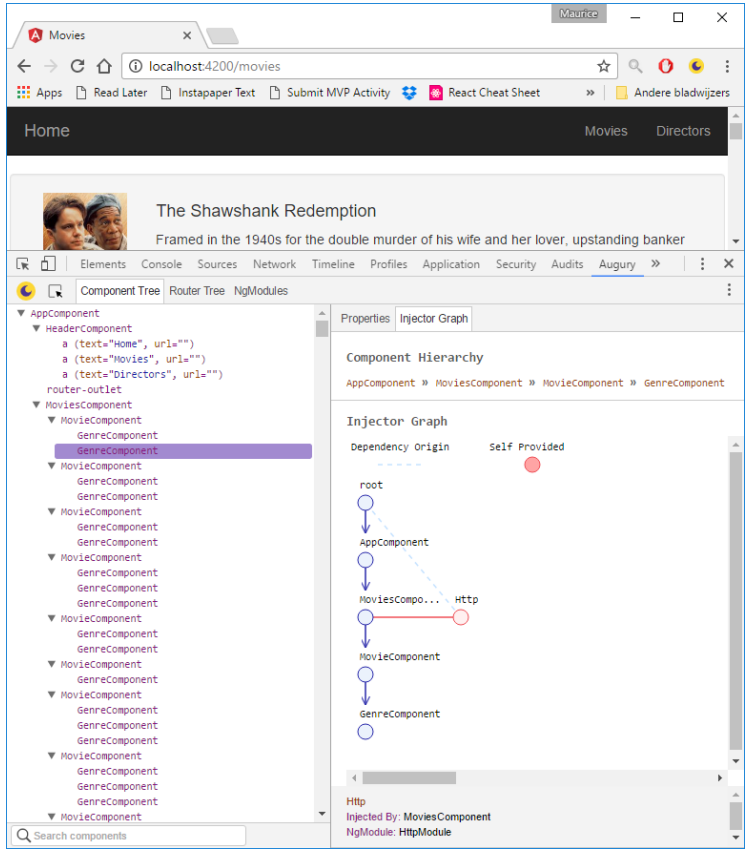
- A tree with links to the parent injector

The main module has it’s own injector.

- As do lazily loaded feature modules

Dependencies can be marked as optional using the @Optional() decorator.

The injector graph



Where to provide services?

Services can be provided in many places:

- Main module
- Feature modules
- Components

Provide in a component for UI specific services.

Provide in a module for other services.

Overriding dependencies

Dependency injection can be overridden anywhere in the injector tree.

Use options like useClass, useValue or useFactory to inject another object.

- Make sure that the original API is supported

Injecting another class

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  providers: [
    {
      provide: ActorsService,
      useClass: YetAnotherActorsService
    }
  ]

})
export class AppComponent {
}
```

Conclusion

Zones are used to detect possible changes to objects.
- Can be manipulated to increase the performance in some cases

Each Angular application consists of one or more modules.
- A module is a reusable unit
- Export the public classes from a feature module
- Services provided my a module a global (except with lazy loading)

Directives are used to interact with the DOM.
- Use the Renderer to isolate the code from different runtime environments

Pipes can be used to transform values when binding to properties.
- Be careful with performance

Dependency injection uses an injector tree.
- Override dependencies as needed