Components

Downloads: http://bit.ly/centric-ng3

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	RxJS and Ajax
26-4-2017	Data Entry
10-5-2017	Single Page Applications

Downloads: http://bit.ly/centric-ng3

What are we going to cover?

Angular Components

- The Component decorator
- Templates
  - Template expressions
  - Content projection
  - Structural directives
- Data binding
  - Change detection
- Lifecyle hooks
- Encapsulation and styles
- Dependency injection
- Unit testing

Angular Components

Components are the basic building block.

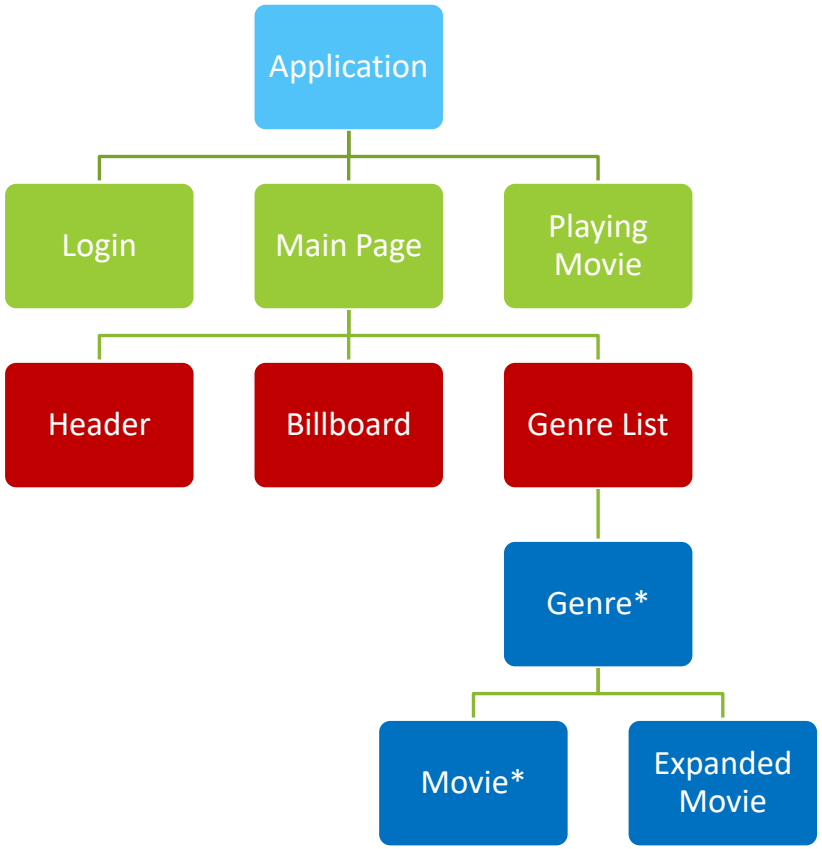
- There is always a root component
- Components form a hierarchical tree

A component is justa class.

- With a template and some metadata to describe the component

Components are a directive on steroids.

- A single component can be associated with an HTML element



The Component decorator

Each Angular component must be decorated with **@Component()**.

- Either the **template** or **templateUrl** property must be provided
- Most components also require a **selector**
- The other properties are optional

Component decorator properties

Metadata Properties:

- **animations** - list of animations of this component
- **changeDetection** - change detection strategy used by this component
- **encapsulation** - style encapsulation strategy used by this component
- **entryComponents** - list of components that are dynamically inserted into the view of this component
- **exportAs** - name under which the component instance is exported in a template
- **host** - map of class property to host element bindings for events, properties and attributes
- **inputs** - list of class property names to data-bind as component inputs
- **interpolation** - custom interpolation markers used in this component's template
- **moduleId** - ES/CommonJS module id of the file in which this component is defined
- **outputs** - list of class property names that expose output events that others can subscribe to
- **providers** - list of providers available to this component and its children
- **queries** - configure queries that can be injected into the component
- **selector** - css selector that identifies this component in a template
- **styleUrls** - list of urls to stylesheets to be applied to this component's view
- **styles** - inline-defined styles to be applied to this component's view
- **template** - inline-defined template for the view
- **templateUrl** - url to an external file containing a template for the view
- **viewProviders** - list of providers available to this component and its view children

Component selector

The CSS selector that identifies a component.

- It’s recommended to use an HTML tag name
- Can also be a class, property or other selector

A basic component

```
@Component({
  selector: 'app-movie',
  template: `
  <div>
    <h2>{{title}}</h2>
    <p>{{description}}</p>
  </div>
  `
})
export class MovieComponent {
  title: string = 'Jaws';
  description: string = 'A movie about a shark';
}
```

Input properties

Input properties are defined with the **@Input()** decorator.

- Use the **[property]** biding syntax in the parent component template

Output properties

Output properties allow for events to be emitted from a component with the **@Output()** decorator.

- All output properties must be instances of **EventEmitter.**
- Use the **(property)** binding syntax in the parent component template.

A component with in/out properties

```
@Component({
  selector: 'app-movie',
  template: `
  <div>
    <h2>{{movie.title}}</h2>
    <p>{{movie.description}}</p>
    <button (click)="saveMovie.emit(movie)">Save</butto
n>
  </div>
  `
})
export class MovieComponent {
  @Input() movie: Movie;
  @Output() saveMovie = new EventEmitter();
}
```

Using the component

```
<app-movie [movie]="theMovie"
           (saveMovie)="onSaveMovie($event)">
</app-movie>
```

Templates

Each component must have a template.

- Can be inline in the decorator or a relative URL to another file

A template can contain most well formed regular html.

- Script tags are not allowed because of injection attacks

Template expressions

Template data expressions can be embedded with the {{ … }} interpolation syntax.

- The {{ and }} can be changed using the **interpolation** property on the decorator of needed

A template expression should produce a single value.

- Side effects like ++, assignment or the new operator are not allowed
- Calling functions is allowed

Each expression is evaluated using a context.

- Allows for variables to be created and stored
- The component instance is part of that instance
- The global namespace isn’t part of the context
  - No console.log() etc.

Template expressions guidelines

No visible side effects.

- Do not change any state in an expression call

Quick execution.

- Expressions can be called often with bound event like MouseMove
- Use memorization is there is an expensive calculation

Simplicity.

- Keep the expressions simple
- Call a function on the component to hide and test more complex actions

Idempotence and pure.

- The same inputs should always produce the same output
- Evaluating the expression once or multiple times should have the same result

Property bindings

Property bindings can be used to pass a value to an HTML element or Angular component.

- The property name is surrounded by […]
- The property value is a string with a template expression that is evaluated

Event bindings

Event bindings can be used to handle any HTML element's event or Angular component output.

- The event name is surrounded by (…)
- The event value is a string with a template expression, typically a function call, that is evaluated
- The **$event** variable contains the event argument

Template reference variables

A template reference variable is a reference to a DOM element within a template.

- Often a form, input or submit element

Template references

```
@Component({
  selector: 'app-movie',
  template: `
  <form #theForm="ngForm">
    <h2>{{movie.title}}</h2>
    <button (click)="save(theForm)">Save</button>
  </form> `
})
export class MovieComponent {
  @Output() saveMovie = new EventEmitter();
  save(theForm: NgForm) {
    if (theForm.dirty &&  theForm.valid) {
      this.saveMovie.emit(this.movie);
      theForm.form.markAsPristine();
    }
  }
}
```

Content projection

With content projection, you can inject elements children into the components template.

- Otherwise they would be removed

Content projection

```
@Component({
  selector: 'app-blockquote',
  template:
'<blockquote><ng-content></ng-content></blockquote>'
})
export class BlockquoteComponent {
}

// Usage in another component template
<app-blockquote>
  <b>Lorem ipsum</b> dolor sit amet
</app-blockquote>
```

Structural directives

Structural directives can be used to manipulate the DOM.

- Use **ngIf** to show or hide a DOM element
- Use **ngSwitch** to display one of a list of DOM elements
- Use **ngFor** to loop over a collection and repeat a DOM element

Structural directives

* and <template>

```
<!-- With syntactic sugar -->
<p *ngIf="condition">
  condition is true
</p>

<!-- With template element -->
<template [ngIf]="condition">
  <p>
    condition is true
  </p>
</template>
```

“Two” way data binding

Use the ngModel directive to simulate databinding using the [()] syntax.

- Angular doesn’t really do two way data binding

Note: The [()] syntax is called banana in a box.

Two way data binding

```
<!-- With syntactic sugar -->
<input [(ngModel)]="title"/>

<!- Is really -->
<input [ngModel]="title"
       (ngModelChange)="title=$event"/>

<!- Behaves like -->
<input [value]="title"
       (input)="title=$event.target.value"/>
```

Change detection

The change detection cycle is triggered when NgZone detects an asynchronous action.

- A DOM event like input or click
- An AJAX request completing
- A timed event like setTimeout()

By default all data binding expressions are checked.

Optimizing change detection

Change detection is done using a tree of change detectors.

- The **ChangeDetectorRef** service can be injected into a component to manually control this

The default way of detecting changes is quite effective.

- You can optimize this by setting the **changeDetection** property to **ChangeDetectionStrategy.OnPush**
- Can be combined with immutable data, **NgZone** and the **ChangeDetectorRef** service

Local only changes

```
@Component({
  selector: 'app-clock',
  template: `<div>Time: {{now.toLocaleTimeString()}}</div>`
})
export class ClockComponent {
  now: Date = new Date();

  constructor(private ngZone: NgZone, private changeDetector: ChangeDetectorRef) {}

  ngOnInit() {
    this.ngZone.runOutsideAngular(() => {
      setInterval(() => {
        this.now = new Date();
        this.changeDetector.detectChanges()
      }, 1000);
    });
  }
}
```

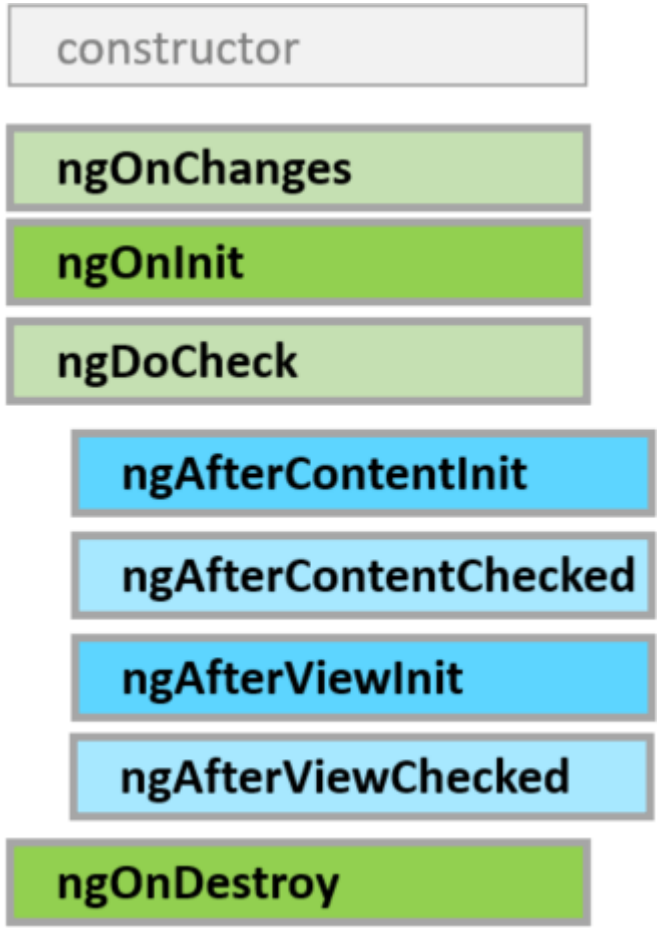
Lifecyle hooks

Angular calls lifecycle hook methods on components when they exist.

There are TypeScript interfaces to assist with type checking.

- The method will still be called without using the interface

Lifecyle hooks



Initialization logic

Initialization logic should be placed in the ngOnInit().

- It fires once for each component.

Use ngOnDestroy() to clean up where needed.

- For example: Custom event handlers

Note: Use the constructor only to store parameters on the constructor.

- Exceptions in a constructor are very hard to track down and debug

Input properties updates

Every time an input property updates the ngOnChanges() fires.

- The collection of changes is passed as a parameter

Beware: the first time this fires is before the ngOnInit() with previousValue as an empty object.

ngOnChanges

Print changes

```
ngOnChanges(changes: SimpleChanges) {
  for (const propName in changes) {
    const change = changes[propName];
    console.log(
      'ngOnChanges',
      propName,
      change.previousValue,
      change.currentValue)
  }
}
```

Encapsulation and styles

The **styles** or **styleUrls** property on the Component decorator adds CSS styles to the component.

The **encapsulation** controls the styles scope using the **ViewEncapsulation** enum.

- **Native** use the browsers shadow DOM
  - Only supported by Chrome, Opera and Safari
- **Emulated** fakes the shadow DOM by adding a custom attribute as a scoping rule
  - The default encapsulation mode
- Using **None** results in global styles

Using native encapsulation

```
@Component({
  selector: 'app-movie',
  template: `<form #frm="ngForm">
    <h2 class="title">{{movie.title}}</h2>
  </form>`,
  styles: [`
    .title { color:blue;}
  `],
  encapsulation: ViewEncapsulation.Native
})
export class MovieComponent {
  @Input() movie: Movie;
}
```

Dependency injection

Both the **providers** and **viewProviders** properties can be used to add or override types that can be used with DI.

- The **viewProviders** is only used for the current component
- The **providers** property is also available to each child component

Injecting top rated movies

```
@Component({
  selector: ‘top-rated-movies',
  templateUrl: './top-rated-movies.component.html',
  viewProviders: [
    {
      provide: MoviesService,
      useClass: TopRatedMoviesService
    }
  ]

})
export class TopRatedMoviesComponent {
}
```

Unit testing a component

A minimal test is generated by the Angular CLI.

- Only tests if the component can be instantiated

Use the TestBed to dynamically configure a module and instantiate a component.

- Find DOM elements using ComponentFixture<T>. debugElement.query()
- Use dispatchEvent() to dispatch input and other events as needed
- Use ComponentFixture<T>. detectChanges() to trigger a change detection cycle

The component to be tested

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-movie',
  template: `
  <div>
    <h2>{{title}}</h2>
    <p>{{description}}</p>
    <p><input type="text" [(ngModel)]="title" /></p>
  </div>
  `
})
export class MovieComponent {
  title: string = 'Jaws';
  description: string = 'A movie about a shark';
}
```

Unit testing a component update

```
it('can update the title', () => {
  titleInput.value = 'Jaws: The Revenge';
  dispatchEvent(titleInput, 'input');

  fixture.detectChanges();

  expect(titleHeader.textContent)
    .toBe('Jaws: The Revenge');
});
```

Setup part one

```
describe('MovieComponent', () => {
  let component: MovieComponent;
  let fixture: ComponentFixture<MovieComponent>;

  let titleInput;
  let titleHeader;

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [MovieComponent],
      imports: [FormsModule]
    })
      .compileComponents();
  }));
```

Setup part two

```
beforeEach(() => {
  fixture = TestBed.createComponent(MovieComponent);
  component = fixture.componentInstance;

  titleInput = fixture.debugElement
    .query(By.directive(NgModel))
    .nativeElement;
  titleHeader = fixture.debugElement
    .query(By.css('h2'))
    .nativeElement;

  fixture.detectChanges();
});
```

Conclusion

Components are the most important part of an Angular application.

- Directives on steroids.

Use the component lifecycle hooks where appropriate.

- Do not put logic in the constructor!

Templates and template expressions provide a lot of flexibility and power.

- Use [] for property, () for event and [()] for “two way” binding

Change detection if flexible.

- Get full control where you need it