Angular

Banu Prakash C

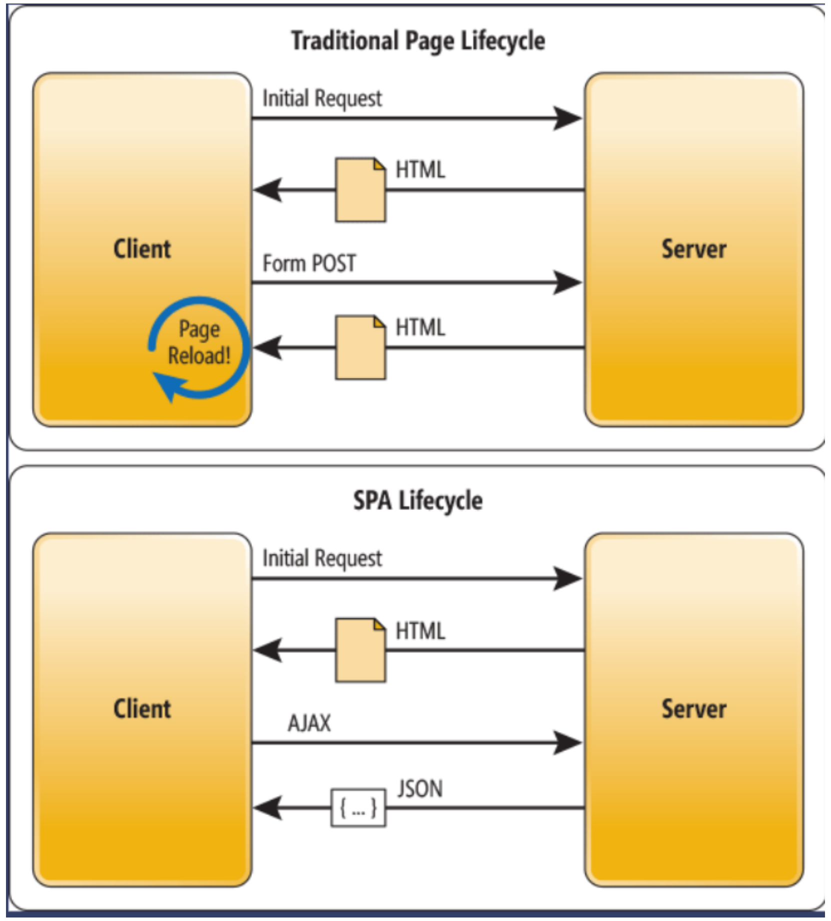
banuprakashc@yahoo.co.in

banu@advantech-global.com

banu@lucidatechnologies.com

What Is a SPA?

- "A single-page application (SPA) is a web application or web site that fits on a single web page with the goal of providing a more fluid user experience similar to a desktop application." - Wikipedia
- Browser fetches executable code that makes asynchronous calls for actual data to be shown
- Data is visualized and/or manipulated and stored back on server asynchronously



Angular

- Built by 20 core developers working for Google & lots of open source devs
- Built with TypeScript (ES6 and Dart versions available)
- First beta 1/2016, first RC 5/2016, 2.0.0 9/2016, *4.0.0 3/2017*
- Complete rewrite of AngularJS
- Not just another web framework, complete platform
- Not just for SPAs, but also for desktop and mobile applications

Angular - Cool Parts

- Ahead-of-Time compilation - Boost build times by compiling offline
- Lazy loading - Only load parts of code when needed
- Universal apps - render A2 app as HTML on server
- Progressive Web Apps (PWA) - Offline web application & push notifications
- Web workers support - Boost up performance with threading
- Migration from 1 - Easy to upgrade from A1 to A2
- Any rendering target (browser, mobile, desktop) - Build apps to all platforms
  - Electron for desktop
  - Ionic 2 (works on top of Apache Cordova) for hybrid mobile apps
  - NativeScript for native UI mobile apps

Angular Fundamentals

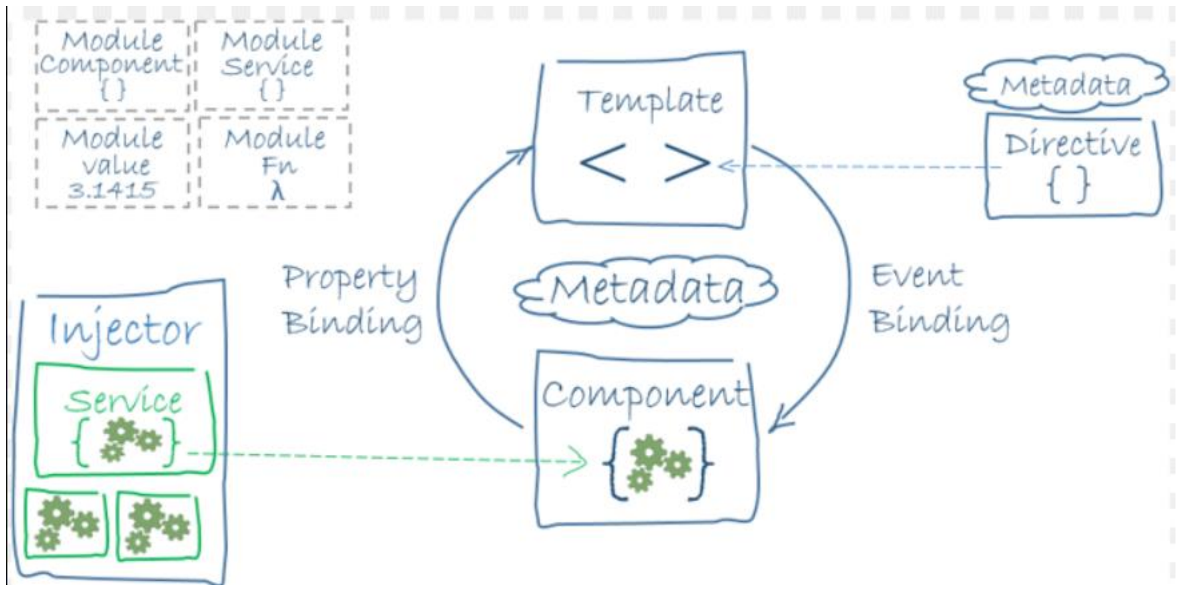
- npm Modules
  - Framework code is distributed as npm modules:
    - @angular/common: Common utilities (pipes, structural directives etc.)
    - @angular/compiler: Ahead-of-Time compiler
    - @angular/core: Core functionality (always needed)
    - @angular/forms: Form handling
    - @angular/http: HTTP requests
    - @angular/platform-*: Platform-specific modules (platforms: browser, server, webworker)
    - @angular/router: Routing functionality
    - @angular/upgrade: NgUpgrade to upgrade from A1 -> A2

Angular Fundamentals

- Codelyzer (tslint plugin) lets you lint against them
- File naming ***name.type.filetype***:
  - *my.module.ts*
  - *todo.component.ts*
  - *user.service.ts*
  - *json.pipe.ts*
  - *yellow-background.directive.ts*

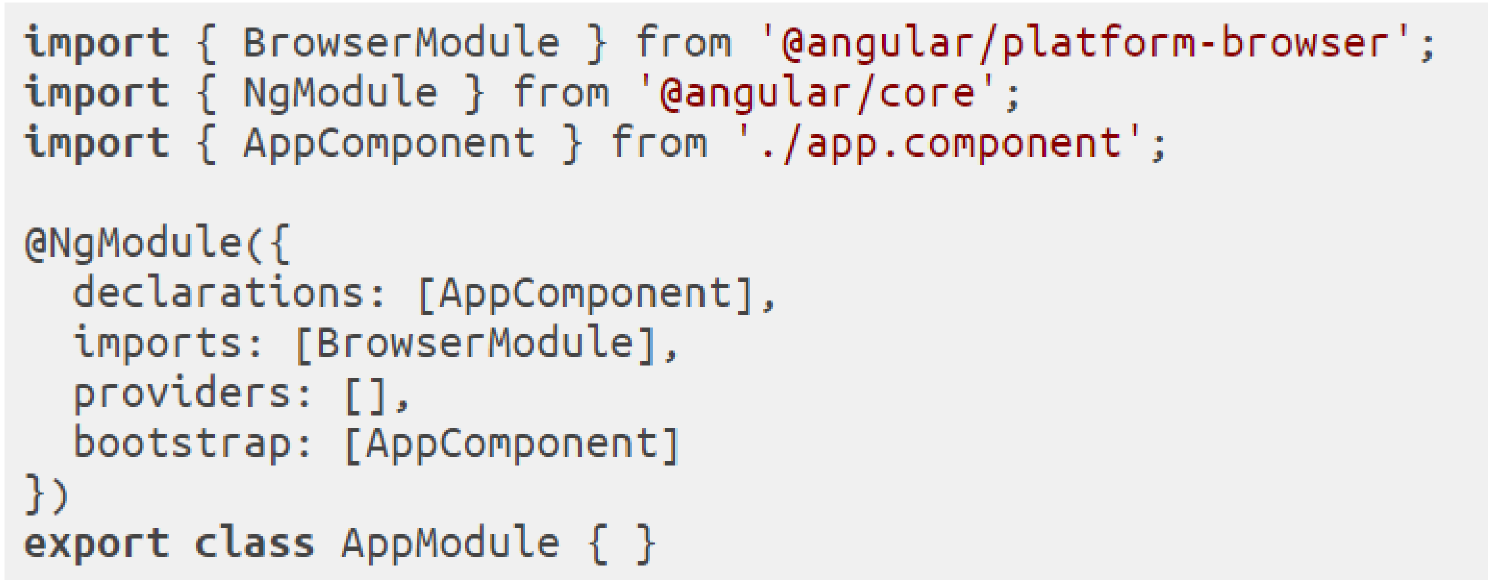
Angular Fundamentals

- Architecture
  - App needs to have at least one **module**
  - Module has one root **component**
  - Component can have child components



NgModules

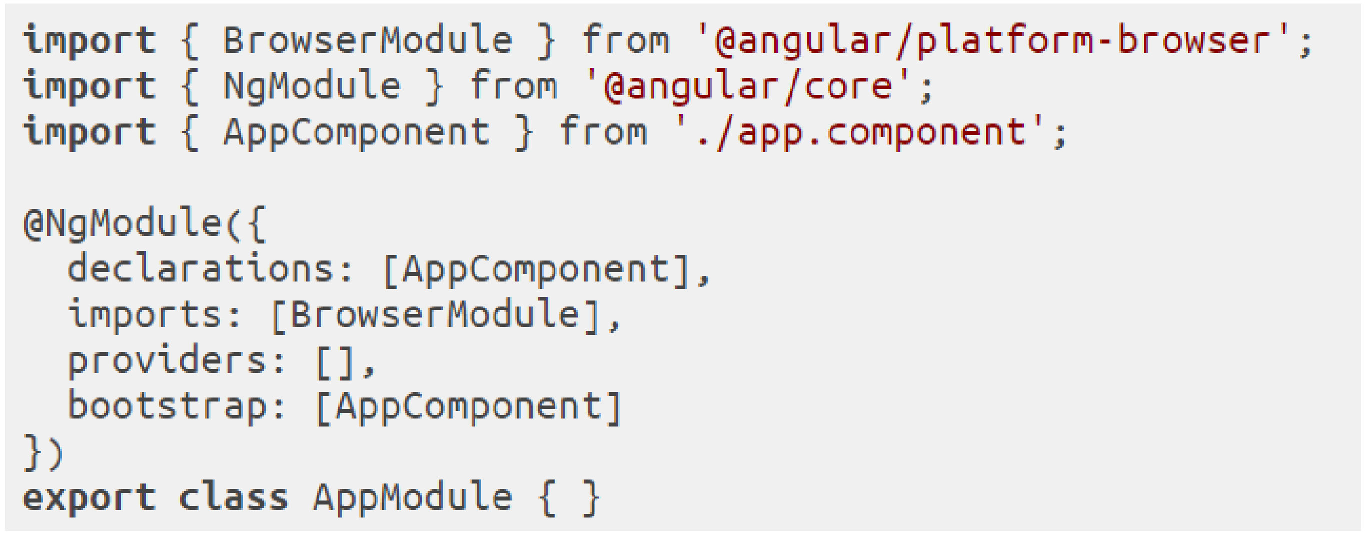
- Declared with @NgModule annotation
- Declares single unit of things relating to same thing
- Each app has single root NgModule



```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';

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

NgModules



```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';

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

- declarations contains list of application build blocks, such as components, pipes and directives, with certain selector
- imports allows importing of other NgModules
- providers contains list of services for dependency injection
- bootstrap contains root element for the application

Booting the application

- Browser executes (compiled) main.ts
- main.ts is responsible for setting up our application
- Root module provided for Angular bootstrapModule

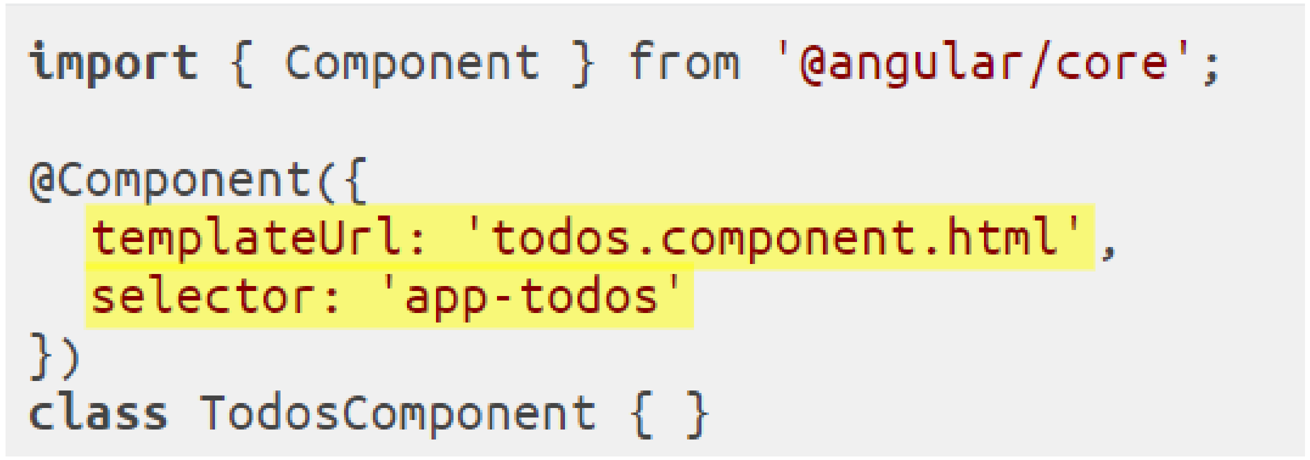
```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { enableProdMode } from '@angular/core';
import { environment } from './environments/environment';
import { AppModule } from './app/app.module';

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

Components

- Build blocks of application UI
- Has a template that it binds data to
- Can contain other components
- Class with @Component annotation



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

@Component({
  templateUrl: 'todos.component.html',
  selector: 'app-todos'
})
class TodosComponent { }
```

Templates

- *app.component.html*

```
<h2>Todos</h2>
<app-todos></app-todos>
```

- *todos.component.html*

```
<h1>My todo application</h1>
```

Template Syntax

- **Data binding** with *property name* inside *double curly braces* {{property}}
- **Structural directives** with *asterisk (*) followed by directive name*
  - <**div** *ngIf="showItem">Item</**div**>
- **Attribute binding** with *attribute name* inside *square brackets ([])*
  - <input [disabled]="property" />
- **Event binding** with *event name* inside *parenthesis*
  - <div (click)="clickHandler()"></div>
- **Template local variables** with *hash (#) followed by name*
  - <input #nameInput />
- Two way binding
  - Data flow is bi-directional:value from component is updated to input
  - when user modifies the value, it is updated to component
  - [(ngModel)]="name"

Services

- Module-wide singletons
- Used to store state, communicate with backends etc.

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

@Injectable()
export class UserService {
}
```

- Need to be registered for NgModule

```
@NgModule(
  ...
  providers: [UserService]
)
export class AppModule() {}
```

Asynchronous and Server-side Communication

Router

- Core responsibilities:
  - Map URL into app state
  - Provide transitions from one URL to another (state to another)
- Supports lazy loading of certain paths

Router – using components

<router-outlet></router-outlet>

declares the placeholder for routed component tree

```
<h1>App works!</h1>
<router-outlet></router-outlet>
```

```
import { NgModule } from '@angular/core';
import { RouterModule } from '@angular/router';

const routeConfig = [
   {
     path: 'todos',
     component: TodosComponent
   }
];

@NgModule({
  declarations: [
    AppComponent, TodosComponent, TodoItemComponent
  ],
  imports: [
    BrowserModule,
    RouterModule.forRoot(routeConfig),
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Router – using components

- declares routes todos/ and todos/:index
- :index is named placeholder for path parameter that can be accessed within the component.

```
@Component({})
export class EditTodoItemComponent {
  constructor(route: ActivatedRoute) {
    route.params.subscribe((params) => {
      this.index = +params['index']; //
    });
  }
```

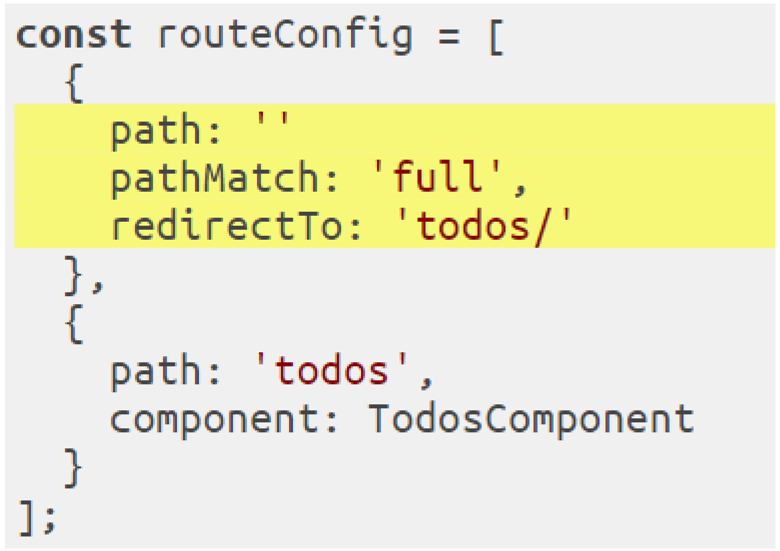
```
const routeConfig = [
  {
    path: 'todos',
    children: [
      {
        path: '',
        component: TodosComponent
      },
      {
        path: ':index',
        component: EditTodoItemComponent
      }
    ]
  }
];
```

Router - Navigating

- Two ways to navigate between states:
  - Imperatively from within the components code:
    - this.router.navigateByUrl('todos/1’)
    - this.router.navigate(['todos', 1])
  - Declaratively from within the template: <**a** [routerLink]="['todos’, 1]">

Router - Redirects

- By default matching of URLs is done with startsWith algorithm
- pathMatch: 'full' can be used to set the algorithm to full matching
- Redirects can be done with redirectTo



```
const routeConfig = [
  {
    path: ''
    pathMatch: 'full',
    redirectTo: 'todos/'
  },
  {
    path: 'todos',
    component: TodosComponent
  }
];
```

Router using child modules

```
@NgModule({
  imports: [
    RouterModule.forRoot([
        {path: '', redirectTo: '/orders', pathMatch: 'full'},
        {path: 'customers', loadChildren: 'app/customers/customers.module#CustomersModule'},
        {path: 'orders', loadChildren: 'app/orders/orders.module#OrdersModule'}
    ])
 ],
exports: [
    RouterModule
]
})
export class AppRoutingModule { }
```

Router - Guards

- Multiple guards available for each route:
- CanActivate to mediate navigation to a route
- CanActivateChild to mediate navigation to a child route
- CanDeactivate to mediate navigation away from the current route
- Resolve to perform route data retrieval before route activation
- CanLoad to mediate navigation to a feature module loaded asynchronously
- The Can* guards can return either boolean or promise (resolving to a boolean value) to allow or prevent navigation

Router - Guards

```
import { CanActivate, Router,
  ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/router';

@Injectable()
export class AuthGuard implements CanActivate {
  constructor(private router: Router, private authService: AuthService) {}

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean
    return this.authService.checkLogin(state.url);
  }
```

```
const routeConfig = [
  {
    path: 'todos',
    canActivate: [AuthGuard],
    component: TodosComponent
  }
];
```

Pipes

- Simple display-value transformations
- Angular 2 provides few common ones, e.g.: UpperCasePipe, LowerCasePipe and DatePipe

```
import {UpperCasePipe} from '@angular/common';

@Component({
  pipes: [UpperCasePipe]
})
class MyComponent {}
```

```
<div>{{user.name | uppercase}}</div>
```

```
<div>{{user.birthDay | date:'fullDate'}}</div>
```

Custom Pipes

- Declared with @Pipe annotation
- PipeTransform interface with transform method
- transform takes the value as first argument and the optional arguments as rest of parameters
- Must be declared in NgModule declaration to be available in templates

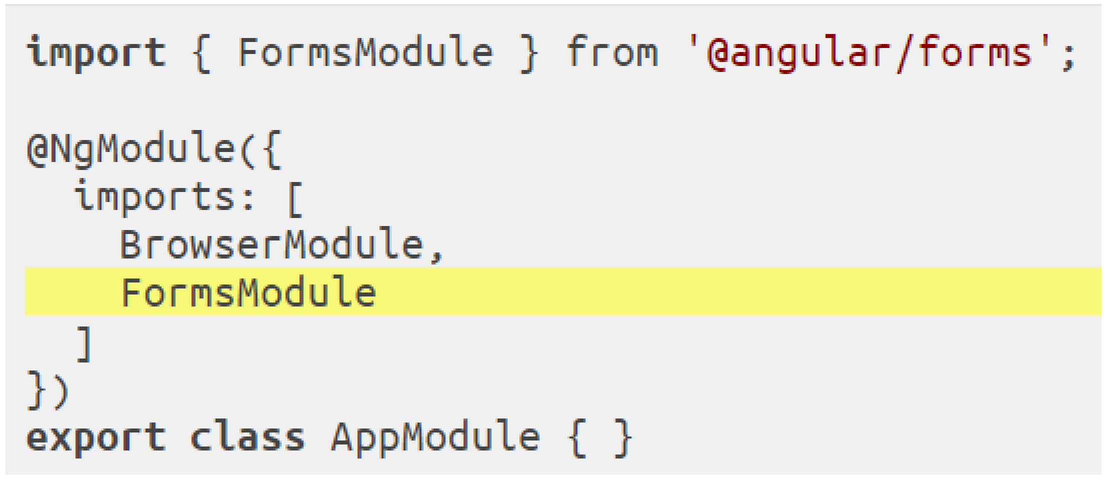
```
import {Pipe, PipeTransform} from '@angular/core';

@Pipe({name: 'exponential'})
export class ExponentialPipe implements PipeTransform {
   transform(value: number, exponent: string): number {
    let exp = parseFloat(exponent);
    return Math.pow(value, isNaN(exp) ? 1 : exp);
  }
}
```

```
<div>{{10 | exponential:3}}</div> <!-- 1000 -->
```

Forms

- Angular 2 provides common form functionalities:
  - Two-way data binding
  - Change tracking
  - Validation
  - Error handling
- Angular 2 forms can be either model- or template-driven
  - Only template-driven forms are covered here, as they are enough for most use cases



```
import { FormsModule } from '@angular/forms';

@NgModule({
  imports: [
    BrowserModule,
    FormsModule
  ]
})
export class AppModule { }
```

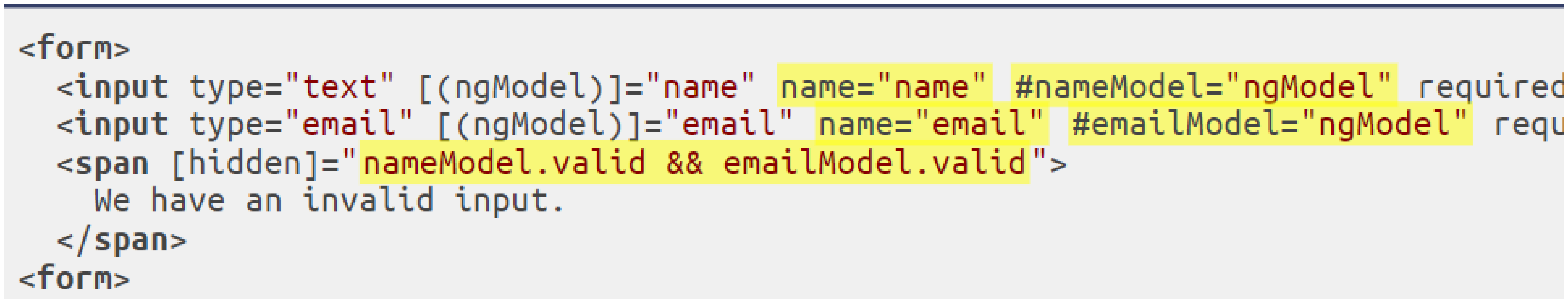
Forms : Template-driven Forms

- Forms declared in templates rather than in component
- Each input inside form element is attached by default
- Each input needs to have (unique) name attribute set

```
<form>
  <input type="text" name="name" [(ngModel)]="name" />
  <input type="email" name="email" [(ngModel)]="email" />
<form>
```

Forms :Using Template-local Variables

- Forms export FormControl as ngModel for each input to be bound to template-local variable

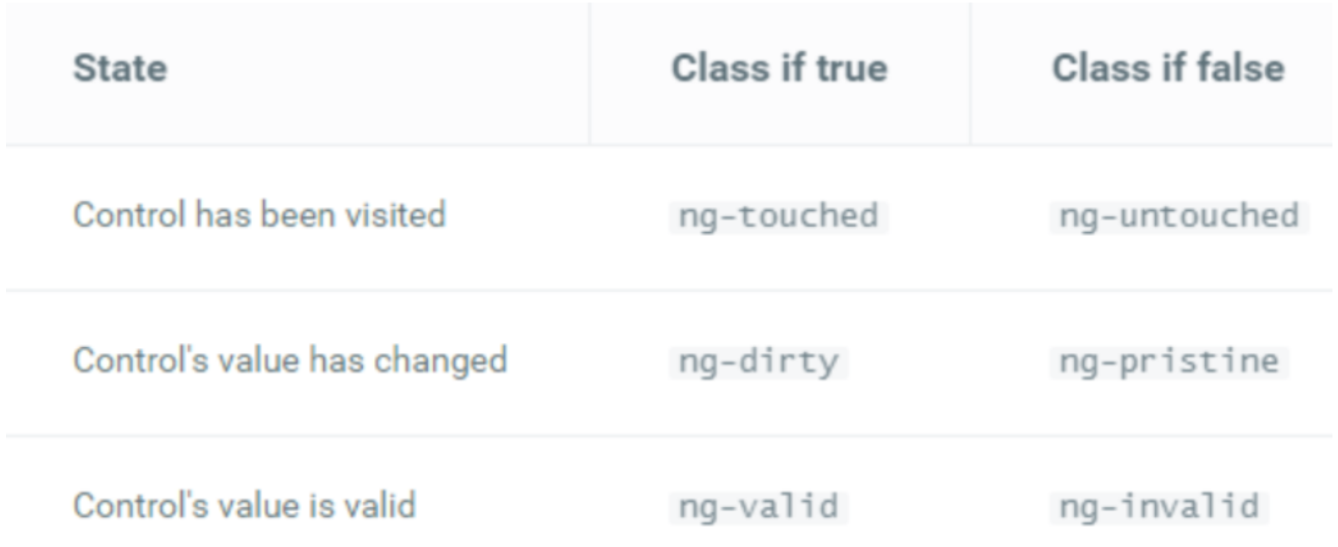


```
<form>
  <input type="text" [(ngModel)]="name" name="name" #nameModel="ngModel" required
  <input type="email" [(ngModel)]="email" name="email" #emailModel="ngModel" requ
  <span [hidden]="nameModel.valid && emailModel.valid">
    We have an invalid input.
  </span>
<form>
```

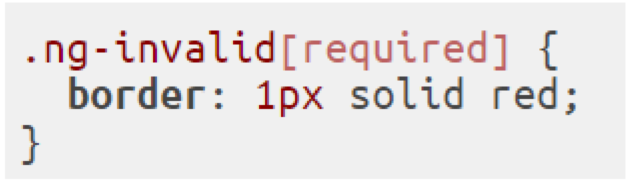
- (#nameModel and #emailModel are called template-local variables)

Forms - CSS Classes

- CSS classes are attached automatically by framework



State	Class if true	Class if false
Control has been visited	`ng-touched`	`ng-untouched`
Control's value has changed	`ng-dirty`	`ng-pristine`
Control's value is valid	`ng-valid`	`ng-invalid`



```css
.ng-invalid[required] {
  border: 1px solid red;
}
```

Forms - NgForm

- Forms exports ngForm which can be bound into template-local variable
- Contains combined information about all the input controls inside the form

```
<form #myForm="ngForm" (submit)="submitForm(myForm)">
  <input type="text" [(ngModel)]="name" name="name" #name="ngModel" />
  <input type="email" [(ngModel)]="email" name="email" #email="ngModel" />

  <button type="submit" [disabled]="myForm.invalid">Submit</button>
<form>
```

```
import { ViewChild } from '@angular/core';
import { FormGroup } from '@angular/forms';

@Component(..)
export class MyComponent {
  @ViewChild('myForm')
  private myForm: FormGroup;
}
```

Directives

- There are three kinds of directives in Angular 2:
  - Components, which are basically selectors with templates and inputs and outputs
  - Structural directives, ngFor and ngIf are examples of these
  - Attribute directives

Directives

- We can use host property to define events with their corresponding handlers

```
<span myHighlight></span>
```

```
import { Directive, ElementRef, Input } from '@angular/core';
@Directive({
    selector: '[myHighlight]',
    host: {
      '(mouseenter)': 'onMouseEnter()',
      '(mouseleave)': 'onMouseLeave()'
    }
})
export class HighlightDirective {
    private el:HTMLElement;
    constructor(el: ElementRef) { this.el = el.nativeElement; }
    onMouseEnter() { this.highlight("yellow"); }
    onMouseLeave() { this.highlight(null); }
    private highlight(color: string) {
      this.el.style.backgroundColor = color;
    }
}
```

Directives - Bind Value from Host Component

- We can bind host components property to our directive by declaring an @Input annotation:
  - @Input('myHighlight') highlightColor: string;
- Now we can pass the property like
  - <span [myHighlight]="color"></span>

Zone.js

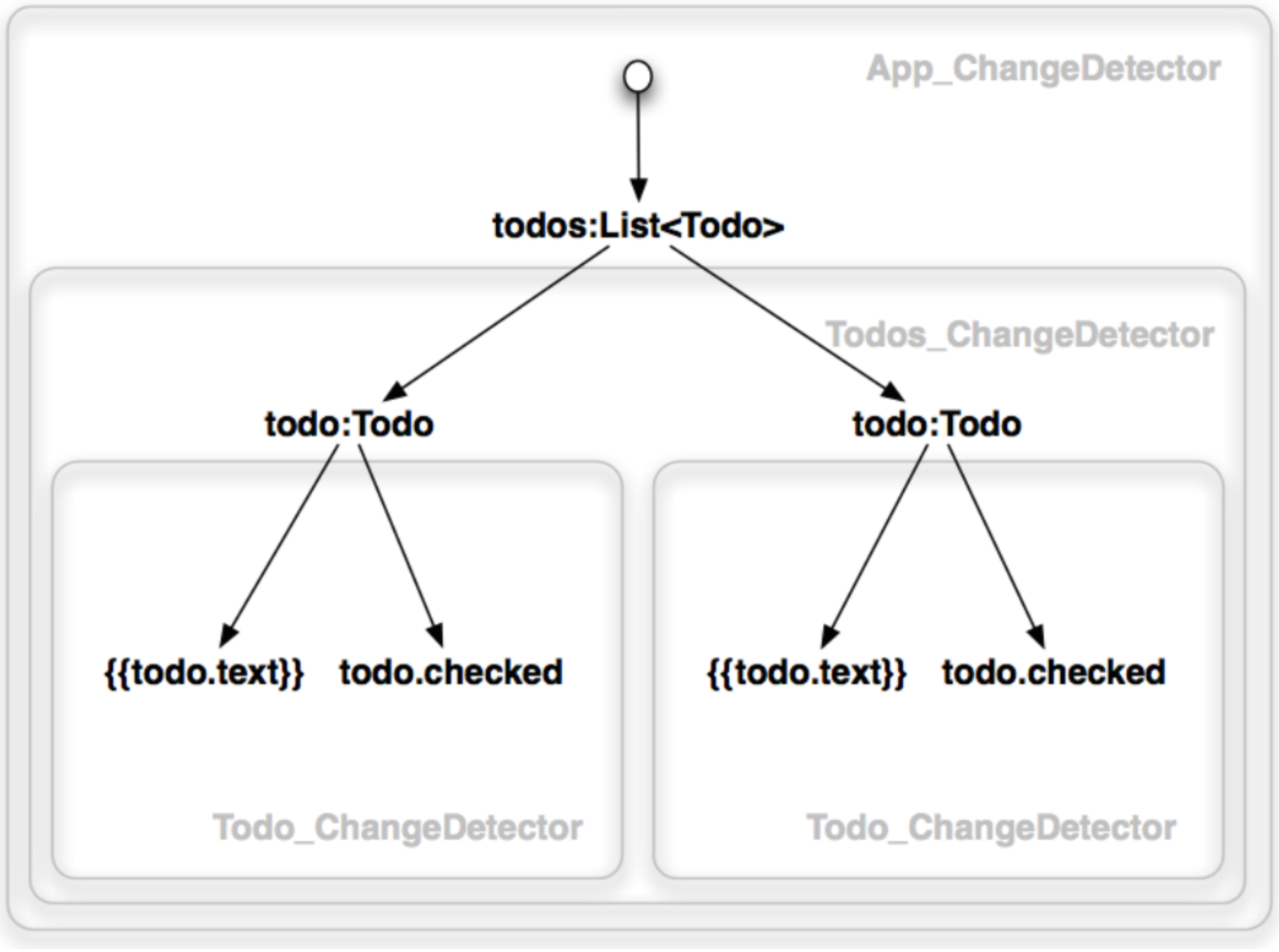
- Implements concept of zones (inspired by Dart) in JavaScript
- "A Zone is an execution context that persists across async tasks"
- In practice makes it possible to track the asyncronous calls made (HTTP, timers and event listeners) within the zone
- Angular 2 uses zones internally to track changes in state

Change Detection

- Idea: project the internal state into UI (DOM in web)
- The internal state consists of JavaScript primitives such as objects, arrays, strings etc.
- Change only possible on asynchronous events such as timeouts or HTTP request
- Zone.js makes it possible to track all possible change sources
- Components change detector checks the bindings (like {{name}} and (click)) defined in its template on change
- Bindings are propagated from the root to leaves in the depth first order

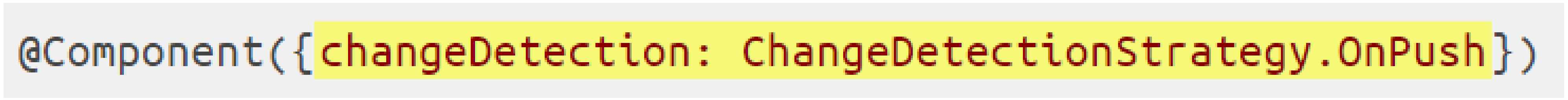
Change Detection

- Change detection graph is directed tree and can't contain cycles -> improved performance, predictability and debugging



Change Detection - Strategies

- Change detection strategy describes which strategy will be used the next time change detection is triggered
  - **OnPush**: Change detector's mode will be set to *CheckOnce* during hydration.
  - **Default**: Change detector's mode will be set to *CheckAlways* during hydration.
- Setting the strategy:



```
@Component({changeDetection: ChangeDetectionStrategy.OnPush})
```

Reactive Programming With Angular

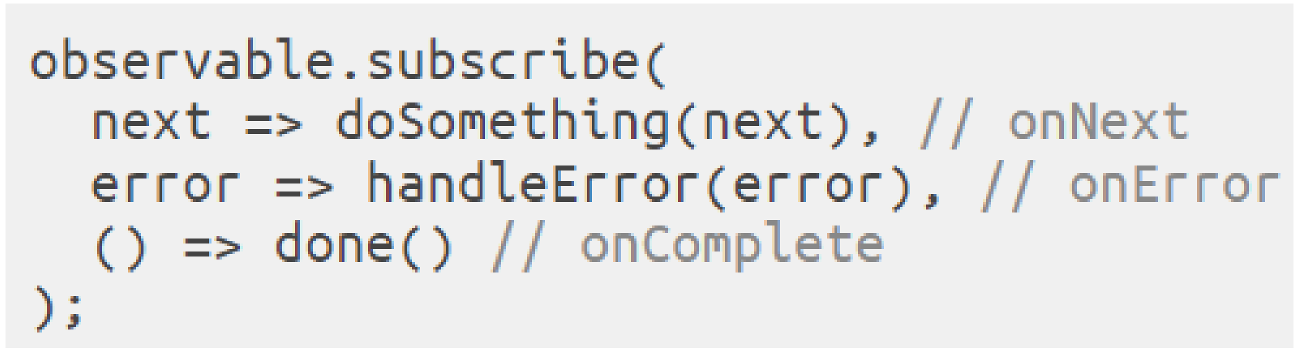
- Reactive programming is programming with asynchronous data streams
- JavaScript is asynchronous by design
  - HTTP requests
  - Timeouts
  - UI events (clicks, key presses, etc.)

RxJS

- *ReactiveX* is a library for representing **asynchronous event streams** with **Observables** and modifying them with various **stream operations**
- *RxJS* is a *ReactiveX* implementation for JavaScript
- Angular integrates with **RxJS 5**
- *"In ReactiveX an observer subscribes to an Observable."*
  - You subscribe to stream of events so that your handler gets invoked every time there is a new item
  - observable.subscribe(item => doSomething(item));

Subscribing

- Subscribe method takes three functions as arguments:
  - **onNext**: called when a new item is emitted
  - **onError**: called if observable sequence fails
  - **onComplete**: called when sequence is complete



```
observable.subscribe(
  next => doSomething(next), // onNext
  error => handleError(error), // onError
  () => done() // onComplete
);
```

Unsubscribing (cancelling)

- Observable sequence subscriptions can be unsubscribed

```
const eventSubscription = eventStream.subscribe(
  event => this.events.push(event)
);
```

- Sequence will not stop until unsubscribed

```
eventSubscription.unsubscribe();
```

Observables in Angular

- Observables used extensively instead of promises
  - HTTP requests can be merely seen as single events (there is only one response) but they are implemented using observable

```
this.http.get('url/restapi/resource') // Returns observable
  .map((res:Response) => res.json()) // Converts response to JSON format
  .subscribe(
      data => { this.data = data}, // Success
      err => console.error(err), // Failure
      () => console.log('done') // Done
  );
```

Unit testing using Jasmine

- Testing of components in isolation from other components
- Guard against changes that break existing code
- Specify and clarify what the code does

- Jasmine is a Unit testing framework for JavaScript
  - Simple basic syntax:
    - describe(string, function) to define suite of test cases
    - it(string, function) to declare single test case
    - expect(a).toBe(b) to make assertions

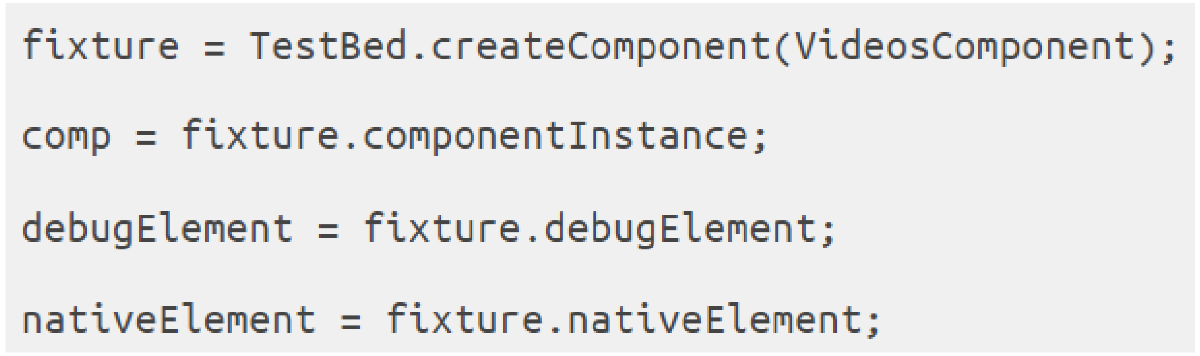
Angular Testing Platform (ATP)

- TestBed for wiring angular components for testing
- Inject mock dependencies
- Testing components with async behavior
- Wiring Up
    - We can setup the test environment with mocks using TestBed:

```
beforeEach(() => {
  TestBed.configureTestingModule({
    declarations: [ VideosComponent ],
    providers: [{provide: VideoService, useClass: FakeVideoService}]
  });
});
```

Angular Testing Platform (ATP)

- Access to Tested Component



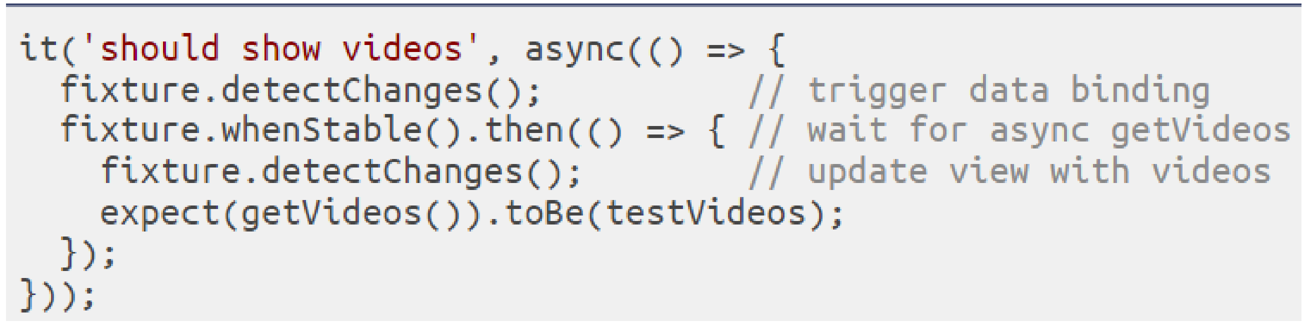
```
fixture = TestBed.createComponent(VideosComponent);

comp = fixture.componentInstance;

debugElement = fixture.debugElement;

nativeElement = fixture.nativeElement;
```

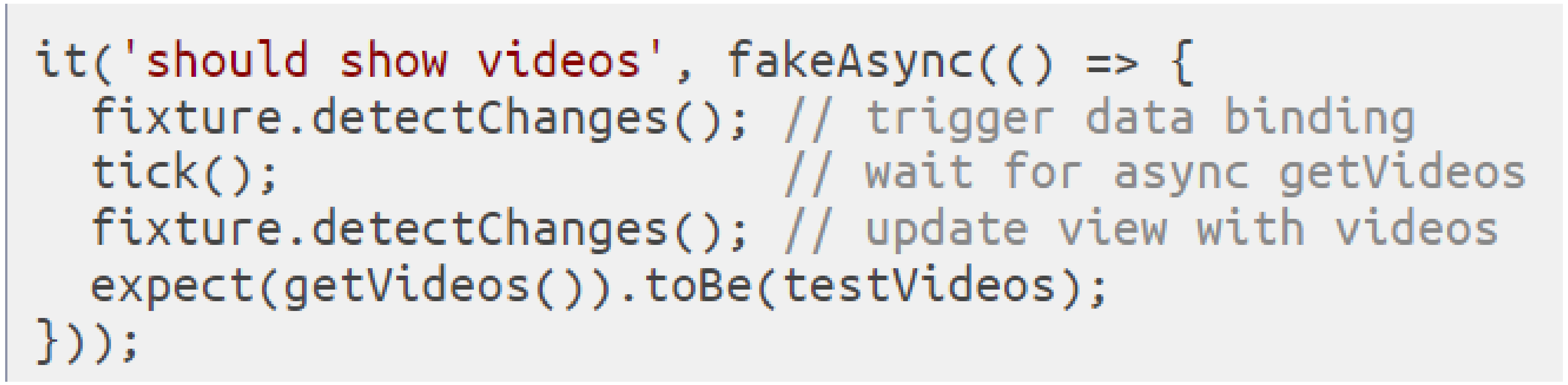
- async()



```
it('should show videos', async(() => {
  fixture.detectChanges();          // trigger data binding
  fixture.whenStable().then(() => { // wait for async getVideos
    fixture.detectChanges();        // update view with videos
    expect(getVideos()).toBe(testVideos);
  });
}));
```

Angular Testing Platform (ATP)

- fakeAsync()



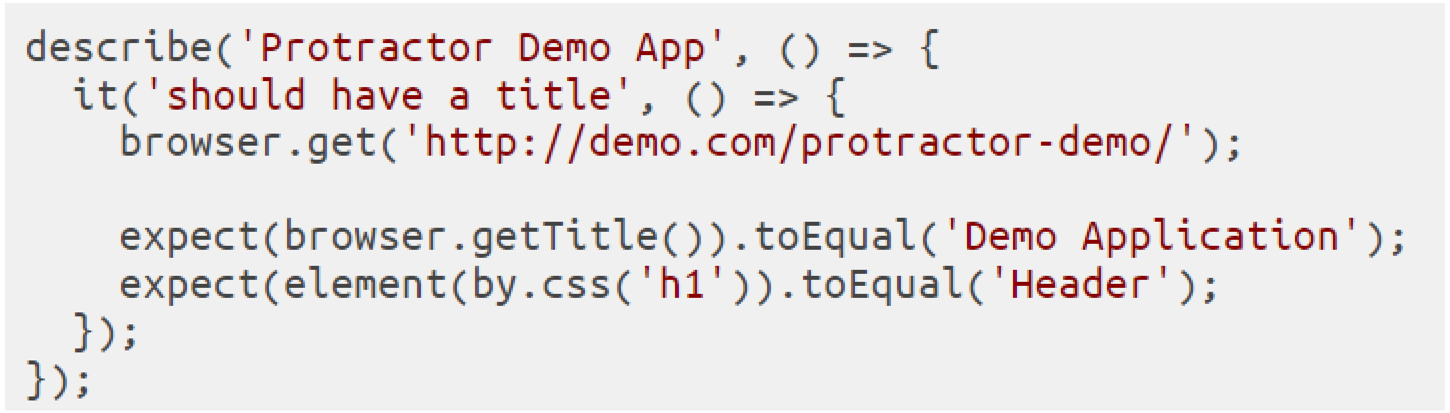
```
it('should show videos', fakeAsync(() => {
  fixture.detectChanges(); // trigger data binding
  tick();                  // wait for async getVideos
  fixture.detectChanges(); // update view with videos
  expect(getVideos()).toBe(testVideos);
}));
```

Karma

- Karma is a test runner with support e.g. for coverage reports and test results exports
- Angular CLI comes with Karma installed
- To run tests, type:
  - ng test

Protractor

- E2E test framework for Angular applications
- Runs tests against your application running in a real browser, interacting with it as a user would
- Angular CLI comes with Protractor installed
  - To run E2E tests, type:
  - **ng** e2e



```
describe('Protractor Demo App', () => {
  it('should have a title', () => {
    browser.get('http://demo.com/protractor-demo/');

    expect(browser.getTitle()).toEqual('Demo Application');
    expect(element(by.css('h1')).toEqual('Header');
  });
});
```

Animations

- Built on top of Web Animations specification
- Part of @angular/core module
- Animations API
  - Part of @angular/core:
    - trigger
    - state
    - style
    - transition
    - animate
  - Declared within the @Component annotation:

```
@Component({
  animations: [
    trigger(...)
  ]
})
```

Animation States

- *Trigger* is bound to the certain element
- *States* are the possible values for triggered value
- States have *styles*
- *Transitions* between states are animated
- *Animations* have duration and timing functions etc.

Triggers, States

- *Trigger* is bound to the certain element
- *States* are the possible values for triggered value

```
trigger('myTrigger', [
  state('state1', style({
    'backgroundColor': '#fff'
  })),
  state('state2', style({
    'backgroundColor': '#000'
  }))
])
export class MyComponent {
  state = 'state1';

  changeState() {
    state = 'state2';
  }
}
```

Transitions and Animations

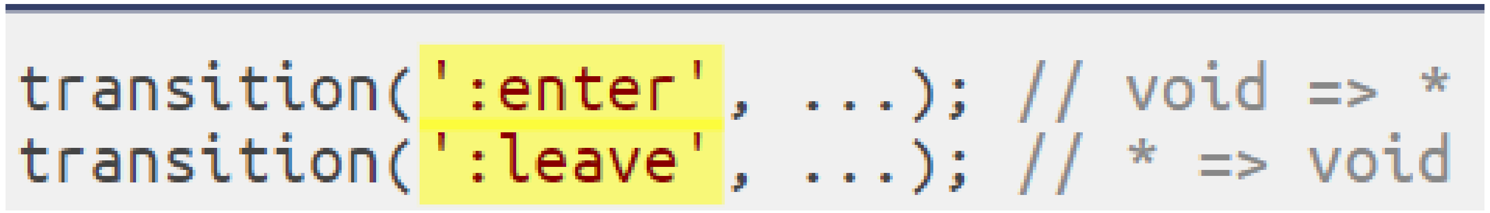
- *Transitions* between states are animated
- *Animations* have duration and timing functions etc.

```
trigger('myTrigger', [
  state(...),
  transition('state1 => state2', animate('100ms ease-in')),
  transition('state2 => state1', animate('100ms ease-out'))
])
export class MyComponent {
  state = 'state1';

  changeState() {
    state = 'state2';
  }
}
```

Special States

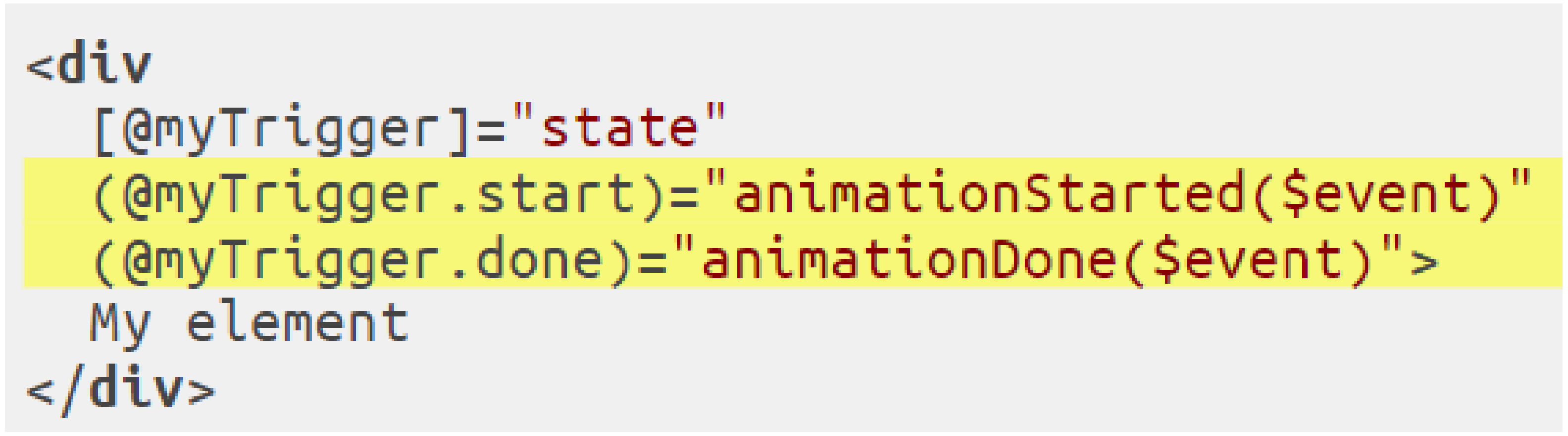
- Two special states:
  - void: element is not attached to a view
  - *: matches any animation state
- Can be used in transitions:
  - transition('void => *', ...): element enters the view
  - transition('* => void', ...): element leaves the view
- Entering and leaving also have aliases:



```
transition(':enter', ...); // void => *
transition(':leave', ...); // * => void
```

Animation Events

- Events to be registered
  - (@myTrigger.start)="animationStarted($event)"
  - (@myTrigger.done)="animationDone($event)"



```html
<div
  [@myTrigger]="state"
  (@myTrigger.start)="animationStarted($event)"
  (@myTrigger.done)="animationDone($event)">
  My element
</div>
```