Angular Basic Training - Gofore

- Open slides: angular-basic-training.herokuapp.com
- Open IDE and start local dev server as instructed in prerequisites
- If using IntelliJ IDEA, open "Settings" and go to *TypeScript > Code Style > TypeScript* and set:
  - Spaces: Within > "Object literal braces" & "ES6 import/export braces"
  - Punctuation: "Use Single quotes in always"

Agenda - Day 1

Morning

- Introduction to SPAs
- TypeScript & Tooling
- Angular Fundamentals

Afternoon

- Angular Fundamentals (continued)
- Angular Advanced Topics

Agenda - Day 2

Morning

- Reactive Programming with Angular

Afternoon

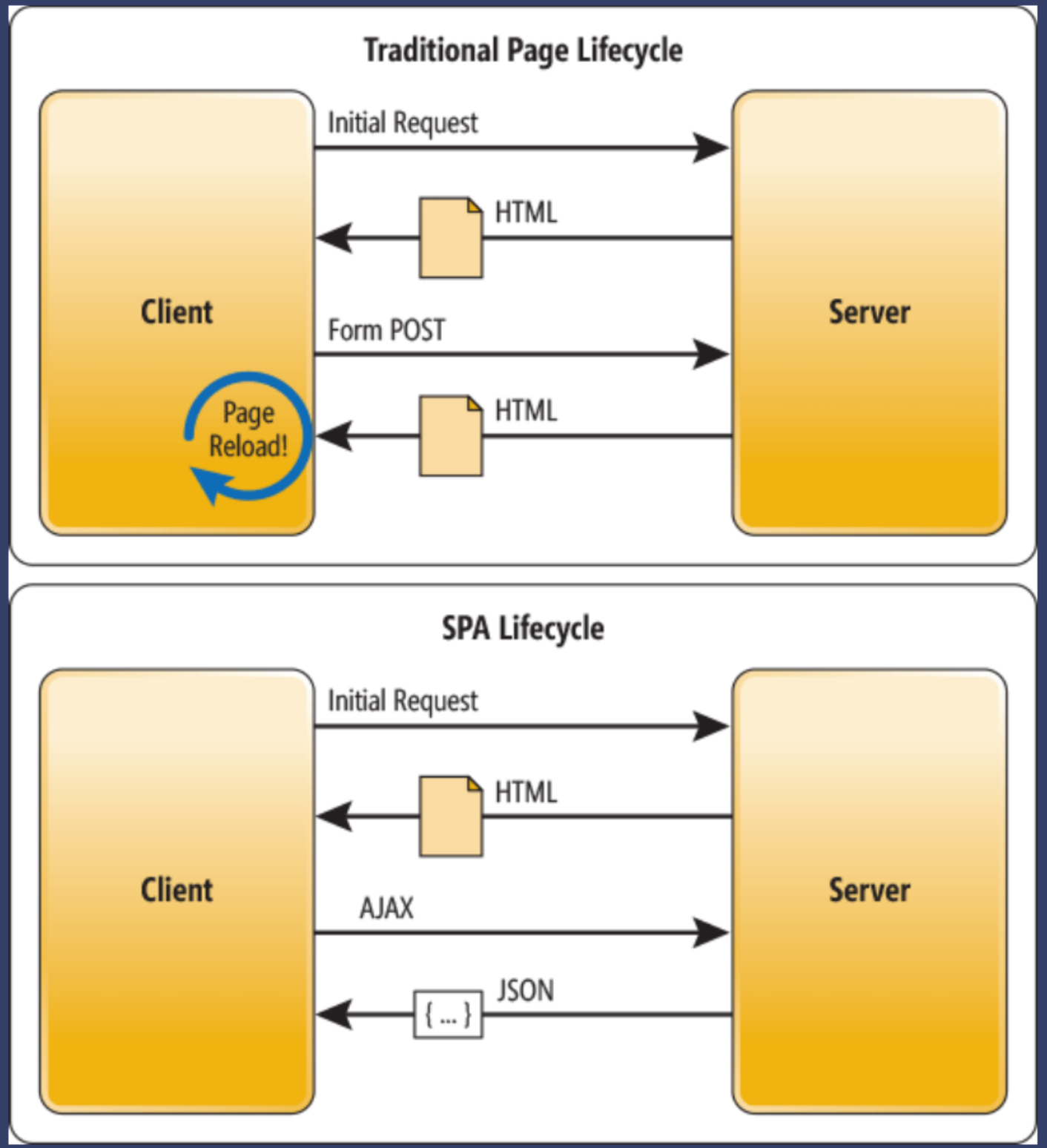
- Testing

Introduction to SPAs

- What is a SPA?
- Real-life examples
- Technical overview

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


Traditional Page Lifecycle
Client
Server
Initial Request
HTML
Form POST
HTML
Page Reload!
SPA Lifecycle
Client
Server
Initial Request
HTML
AJAX
JSON
{ ... }

Real-life Examples

- Google search
- Facebook
- Twitter

SPA Frameworks/Libraries

- Backbone.js
- Ember
- Meteor
- AngularJS
- Aurelia
- React
- Vue.js
- Angular
- And so many more...

AngularJS

- Published 2010
- MVC (Model-View-Controller) framework with dependency injection
- Revolutionary on its own time
- Two-way data binding
- Emphasis on testability (decouples DOM manipulation from app logic)

Angular

- Built by around 20 Google developers & lots of open source devs
- Built with TypeScript (ES2015 and Dart versions available)
- Complete rewrite of AngularJS
- Not just another web framework, complete platform
- Also for desktop and mobile development
- Documentation

Angular Release

- Major version every 6 months (April and October)
- Two version deprecation policy
- Even numbers (4, 6, ..) offer LTS (Long-Term Support)
- Releases:
  - 2.0.0-beta.0 1/2016
  - 2.0.0-rc.0 5/2016
  - 2.0.0 9/2016
  - 4.0.0 3/2017
  - 5.0.0 11/2017
  - 6.0.0 4/2018
- More info in Angular GitHub

Tooling

- Traditionally web pages have been just static HTML, CSS and maybe some simple JS for dropdowns etc.
- Nowadays massive SPAs require something more advanced and thus the need for tooling
- Some basic needs for tooling:
  - Compiling ES2015/TypeScript -> ES5 and LESS/SASS -> CSS
  - Combining multiple source files into single bundle file for faster loading
  - Running test suites
  - Optimizations (minification, Dead code elimination, tree shaking)

JSON

- JavaScript Object Notation (JSON) is a lightweight data-interchange format
- Meant to be easy for both, humans and machines
- Key-value pairs, where values can be any JavaScript primitives (except functions)

```
{
  "name": "John Doe",
  "email": "john.doe@example.com",
  "friends": [
    {"name": "Jane Doe"},
    {"name": "John Doe Jr."}
  ]
}
```

Node.js & npm

- Node.js is JavaScript interpreter built on top of Chrome's V8 JavaScript engine
- Used for running development server, to run tests, to build production-optimized bundle, etc.
- npm (node package manager) is the package manager for Node
  - More packages than on any other package manager for any other language: over 270k (May 2016) (modulecounts.com)

package.json - project configuration

- Declares dependencies, development-time dependencies and commands available
- Can be generated with `npm init`

```
{
  "name": "Angular basic training",
  "author": "Gofore",
  "scripts": {
    "build": "my-build.sh",
    "start": "my-webserver.sh"
  },
  "dependencies": {
    "@angular/core": "2.0.0",
    "@angular/forms": "2.0.0",
    "@angular/http": "2.0.0"
  },
  "devDependencies": {
    "typescript": "^2.0.0",
    "jasmine": "^2.5.0"
  }
}
```

Angular CLI

- Command-line interface for Angular development
- Recommended by the core team
- One of the core modules: `@angular/cli`
- Follows Angular core versioning as of *6.0.0*
- Abstracts away the bundling
- Uses Webpack internally

Angular CLI Features

- Generate the project initially
- Run dev server
- Generate modules, components, services, tests, directives
- Generate production build
- Tests
- Update your dependencies
- Supports CSS preprocessors (SASS and LESS)
- Allows third-party generators for Angular CLI projects

Angular CLI Usage

- Generating an app

```
ng new PROJECT_NAME
```

- Run development server

```
ng serve # Available in localhost:4200
```

- Run tests

```
ng test
```

- Generate a component

```
ng generate component todos
```

ES2015 (ES6)

- EcmaScript (ES) is the standard for JavaScript
- One of the newer EcmaScript standards
- Published 2015
- Also known as ES6 because last version was ES5
- Provides a lot of improvements for writing JavaScript in scale
- Not supported by older browsers (namely IE11)

ES2015 - Key Features

- `let` and `const` to replace `var`
- Arrow functions
- Multiline strings
- Modules
- Enhancements on basic types such as `includes()` for string and `find()` for array

Let and const

- `const` declares constant **reference** (not constant value)
- `let` declares variable (eg. `let myVar = 'asd';`)
- **Rule of thumb: Always use `const` if possible, `let` otherwise.**

```
const input = [0, 1, 2, 3, 4];
input = []; // Uncaught TypeError: Assignment to constant variable.
input.push(5); // Works, as input is just the reference
```

For immutable objects & arrays there are libraries such as *Immutable.js*.

Arrow functions

- Lexical `this` and more concise syntax

```
// Traditional function
const increase = function (value) {
  return value + 1;
};

// Arrow function
const increase = (value) => {
  return value + 1;
};

// Parenthesis omitted
const increase = value => {
  return value + 1;
};

// Return value without curly braces
const increase = value => value + 1;
```

String Literals

ES5 string:

```
const str = firstName + ' ' + secondName.charAt(0) + '. ' + lastName;
```

ES2015 multiline string with backticks:

```
const str = `${firstName} ${secondName.charAt(0)}. ${lastName}`;
```

Modules

Allows `importing` and `exporting` code between files (modules)

lib.js

```
export function square(x) {
   return x * x;
}
export function squareSum(x, y) {
   return Math.sqrt(square(x) + square(y));
}
```

main.js

```
import { square, squareSum } from './lib';
console.log(square(11)); // 121
console.log(squareSum(4, 3)); // 5
```

- Importing from npm modules

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

Array Functions

- `map`, `filter` and `reduce` operate over the array
- `map` creates a new array with the results of calling a provided function on every element in the calling array.

```
const arr = [0, 1, 2, 3];
const result = arr.map(item => item * 2);
// result is [0, 2, 4, 6]
```

- `filter` creates a new array with all elements that pass the test implemented by the provided function.

```
const arr = [0, 1, 2, 3];
const result = arr.filter(item => item < 2);
// result is [0, 1]
```

- `reduce` applies a function against an accumulator and each element in the array (from left to right) to reduce it to a single value.

```
const arr = [0, 1, 2, 3];
const result = arr.reduce((acc, item) => acc + item , 0);
// result is 6
```

TypeScript

- General-purpose programming language
- Built by Microsoft to build JavaScript in scale
- Initial release in October 2012
- Typed superset of JavaScript -> Any valid JS is valid TypeScript
- Advantages:
  - Static type system on top of JavaScript to catch errors already on compile-time
  - Angular & RxJS are written in TypeScript

Typing

- Provides the same types as in JavaScript: `number`, `string`, `boolean`, `null`, `undefined` and `object`
- Arrays like `number[]`, `string[]` and `any[]`
- Also some "extra" types such as `any`, `void` and `enum`
- `any` is basically anything like `number`, `string` or `any[]`
- Types are marked after the name
- Type declaration can be omitted when assigning
- Examples:

```
let luckyNumber: number = 50;
luckyNumber = 'nine'; // TypeError: Can't assign string to number

function increase(value: number): number {
    return value + 1;
}
```

Interfaces

- Interfaces to declare the acceptable object structures
- Can have optional properties (declared with ? before :)
- *Structural typing* instead of *Nominal typing*

```
interface Person {
    firstName: string;
    middleName?: string;
    lastName: string;
}

const greeter = (person: Person): string => "Hello, " + person.firstName + " " + p

greeter({ firstName: "John", lastName: "Doe" }); // Hello, John Doe
```

Classes

- Like in many other programming languages
- Member fields can be declared in constructor with visibility modifier (`private`, `protected`, `public`)

```
class Student {
    fullName: string;
    constructor(private firstName: string, private lastName: string) {
        this.fullName = this.firstName + " " + this.lastName;
    }

    getFirstName() {
        return this.firstName;
    }
}

const student = new Student("John", "Doe");
console.log(student.getFirstName());
```

Annotations (Decorators)

- Used to "decorate" classes and properties with additional functionality
- Like Java annotations
- Apply to next entity (class, field, method) after them

```
@Component({
  template: 'my template'
})
class MyClass {
  @Input() myProperty: string;
}
```

Angular Fundamentals

- npm Modules
- File Structure
- Architecture
- NgModules
- Components
- Templates
- Component Lifecycle Hooks
- Two-way Data Binding
- Services
- Asynchronous and Server-side Communication

Angular npm Modules

- Framework code is distributed as npm modules:
  - `@angular/animations`: Advanced animations functionality
  - `@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/language-service`: Language service for Angular templates for better IDE support
  - `@angular/platform-*`: Platform-specific modules (platforms: browser, server, webworker)
  - `@angular/router`: Routing functionality
  - `@angular/service-worker`: Service worker functionality
  - `@angular/upgrade`: NgUpgrade to upgrade from AngularJS -> Angular
  - `@angular/http`: Deprecated HTTP client

Coding Style

- Angular style guide declares set of rules
- Codelyzer (TSLint plugin) checks for
- File naming *name.type.filetype*:
  - *app.module.ts*
  - *todos.component.ts*
  - *todos.component.html*
  - *todos.component.scss*
  - *user.service.ts*
  - *json.pipe.ts*

Architecture

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

NgModules

- Each application has single root *NgModule*
- *NgModule* is a class with `@NgModule` annotation
- Declares collection of related elements (components, services etc.)

app.module.ts

```
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 - Details

- `declarations` contains list of application build blocks, such as components, pipes and directives, with certain selector
- `imports` allows importing of other *NgModules*
  - For example `BrowserModule` imports browser-specific renderers and core directives such as `ngFor` and `ngIf`
- `exports` allows declaring what is exported from this module (covered later)
- `providers` contains list of services for dependency injection
- `bootstrap` contains root element(s) for the application (usually named `AppComponent`)

Booting the application

- We need to tell Angular to start our application
- This is done by providing root module `bootstrapModule`
- This will go through the `bootstrap` array and checks for those selectors in HTML

main.ts

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

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

index.html

```
<html>
    <body>
        <app-root></app-root>
    </body>
</html>
```

Modules in Large Applications

- Modules are meant to offer possibility to divide the functionality in logical modules
- For example one could have `CustomerModule`, `AdminModule` and `BillingModule`
- To access `exports` of another module, it needs to be `imported` to the module:

app.module.ts

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

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

Exporting

- Exporting allows declaring the public API provided by the module
- Consider company-specific UI module where `ListComponent` utilizes `ListRowComponent`:

ui.module.ts

```
import { NgModule } from '@angular/core';
import { ListComponent } from './list.component';
import { ListRowComponent } from './list-row.component';

@NgModule({
  declarations: [ListComponent, ListRowComponent],
  imports: [],
  exports: [ListComponent],
  providers: []
})
export class UiModule { }
```

Modules as Exports

- Directives `ngIf` and `ngFor` are defined actually in `CommonModule`
- But that is not imported anywhere?
- Actually, `BrowserModule` is exporting `CommonModule` as seen in source
- -> Modules can be exported as part of module
- If `ListModule` and `TableModule` were separate modules:

ui.module.ts

```
import { NgModule } from '@angular/core';
import { ListModule } from './list.module';
import { TableModule } from './table.module';

@NgModule({
  declarations: [],
  imports: [],
  exports: [ListModule, TableModule],
  providers: []
})
export class UiModule { }
```

Components

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

todos.component.ts

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

@Component({})
class TodosComponent { }
```

Components

- Two parameters are mandatory for `@Component` annotation:
  - template with `template` (inline template) or `templateUrl` (separate file)
  - selector (should always start with application specific prefix like the default: `app`)
- Components need to be declared in NgModule's declarations to be available in templates

todos.component.ts

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

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

app.module.ts

```
@NgModule({
  declarations: [AppComponent, `TodosComponent`]
  ...
})
export class AppModule { }
```

Generating a Component

Browse to root of the project and run:

```
ng generate component todos
```

or abbreviated one:

```
ng g c todos
```

This will:

- Create a folder called `todos` with the component, template, styles and test file
- Add the component to `declarations` of your `AppModule`

Templates

- Plain HTML with few Angular specific additions*

todos.component.html

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

- Selectors can be used to add other components

app.component.html

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

Tree Structure

- Including components in templates causes tree-like structure

Angular 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 />
```

Data Binding

Bind property from the component to the template

app.component.ts

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

@Component({
  selector: 'app-component',
  templateUrl: 'app.component.html'
})
class AppComponent {
  `title: string = 'app works!';`
}
```

app.component.html

```
<h1>`{{title}}`</h1>
```

Structural Directives

- Modify the *structure* of the template
- Two most used are `*ngIf` and `*ngFor`

todos.component.ts

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

@Component({
  selector: 'app-todos',
  templateUrl: 'todos.component.html'
})
class TodosComponent {
  `todos: any[] = [{name: 'Do the laundry'}, {name: 'Clean my room'}];`
}
```

todos.component.html

```
<div `*ngFor="let todo of todos"`>
  {{todo.name}}
</div>
```

Attribute Binding

Bind value from component into HTML attribute

app.component.ts

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

@Component({
  selector: 'app-component',
  templateUrl: 'app.component.html'
})
class AppComponent {
  `isDisabled: boolean = true;`
}
```

app.component.html

```html
<input [disabled]="isDisabled" />
```

Special Attribute Bindings

- Classes and styles have special syntax available:

```
<div [class.active]="isActive"></div>
<div [style.display]="isShown ? 'block' : 'none'"></div>
```

Attribute Binding for Components

- Attribute binding only works for native properties of HTML elements by default
- Same concept can also be used to pass data from parent component to child component

parent.component.html

```
  <app-child `[foo]="todo"`></app-child>
```

child.component.ts

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

@Component({
  selector: 'app-child'
})
class ChildComponent {
  @Input()
  foo: string;
}
```

Event Binding

- Register handler code for events
- The actual event can be referenced with `$event`
- The events are basic DOM events (like `click`, `mouseover`, `change` and `input`)

todos.component.html

```
<input type="text" `(change)="handleChanged($event.target.value)"`></input>
```

todos.component.ts

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

@Component({..})
class TodosComponent {
  handleChanged(value: string) {
    // Do something with value
  }
}
```

Event Binding for Components

- Event binding only works for native events of HTML elements by default
- Same concept can also be used to pass data from child component to parent component

parent.component.html

```
<app-child `(change)="todo"`></app-child>
```

child.component.ts

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

@Component({
  selector: 'app-child'
})
class ChildComponent {
  @Output()
  change = new EventEmitter<string>();
}
```

Two-way Data Binding

- Two-way data binding with `ngModel` inside *banana-box syntax*: `[(ngModel)]`
- Data flow is still unidirectional, though:
  - value from component is updated to input on change
  - when user modifies the value, it is updated to component

```
Name: <input type="text" `[(ngModel)]="name"` />
```

which is just sugar for

```
Name: <input type="text" `[ngModel]="name" (ngModelChange)="name = $event"` />
```

CSS Encapsulation

- Component can have styles applied:
  - Inline in annotation (field `styles` in `@Component`)
  - In external files (field `styleUrls`)
- These styles are scoped for component -> No other component can get affected by them

todos.component.html

```
<div class="todos"></div>
```

todos.component.css

```
.todos {
    background-color: red;
}
```

Does not affect styles here:

clients.component.html

```
<div class="todos"></div>
```

Components - Inline Styles

todos.component.ts

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

@Component({
  selector: 'app-todos',
  templateUrl: 'todos.component.html',
  `styles: ['.active-todo { background-color: yellow; }']`
})
class TodosComponent {
}
```

Components - Styles From File

todos.component.ts

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

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

Component Lifecycle Hooks

- For hooking into certain lifecycle events
- Interface for each hook
- Example hooks: `ngOnInit` (interface `OnInit`), `ngOnChanges` (interface `OnChanges`) and `ngOnDestroy` (interface `OnDestroy`)

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

export class MyComponent `implements OnInit` {
  `ngOnInit() { ... }`
}
```

- `ngOnInit` should be used for initialization of data for testability

Services

- Module-wide singletons
- Used to store state, communicate with backends etc.
- Examples: `UserService`, `BackendService`
- Declaring:

user.service.ts

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

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

- Need to be registered for NgModule

app.module.ts

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

Generating a Service

Browse to root of the project and run:

```
ng generate service todo
```

or abbreviated one:

```
ng g s todo
```

This will:

- Create a file called `todos.service.ts` in the root of `app/` folder along with the test stub
- *not* add it as provider in `AppModule` so you must do it yourself!

DI with Constructor Parameters

- Angular implements concept called `Dependency Injection`
- In DI you can just ask for the dependencies as constructor parameters as follows:

todos.component.ts

```
import { Component } from '@angular/core';
import { BackendService } from './backend.service';

@Component({
  selector: 'app-todos',
  templateUrl: 'todos.component.html'
})
class TodosComponent {
  constructor(private backendService: BackendService) {
  }

  initializeData() {
    this.backendService.makeRequest();
  }
}
```

Angular will then pass the singleton instance of `BackendService` to the component when creating it

Asynchronous and Server-side Communication

- Asynchronous things are modeled as Observables (covered later) in Angular
  - For now, we only need to know that there is `subscribe` method
- For AJAX requests, use `HttpClient` service found in `@angular/common`

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

@Component({...})
export class MyComponent {
  filteredData: any[];
  constructor(private httpClient: HttpClient) {
  }

  getData() {
      this.httpClient.get('https://example.com/mydata').subscribe(data => {
       // Do stuff with data
       this.filteredData = data.filter(item => item.id > 100);
    })
  }
}
```

Angular Advanced Topics

- Router
- Forms
- Pipes
- Directives

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 Basics

Route declarations

app.module.ts

```
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 Basics

`<router-outlet></router-outlet>` declares the placeholder for routed component tree

app.component.html

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

Router - More advanced routes

app.module.ts

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

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

Redirects

- By default matching of URLs is done with *startsWith* algorithm (and everything starts with path `''`)
- `pathMatch: 'full'` can be used to set the algorithm to full matching
- Redirects can be done by using `redirectTo` instead of specifying `component`

app.module.ts

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

Accessing Route Parameters

- Route parameters can be accessed via `params` field of `ActivatedRoute` as observable
- Once parameters change new event will be sent.
- Makes it possible to reuse the same component instead of instantiating a new one

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

Fragments & Query Parameters

- Fragments (`example.com#key=value`) and query parameters (`example.com?key=value`) are shared by all routes
- Can be accessed like path parameters:

```
@Component({})
export class MyComponent {
  constructor(route: ActivatedRoute) {
    this.route.queryParams.subscribe(params => { // or .fragment
      this.key = params.key;
    });
  }
```

Router - Navigating

Two ways to navigate between states:

- Imperatively inside a component:

```
this.router.navigateByUrl('todos/1')
```

- Declaratively inside a template:

```
<a [routerLink]="todos/1"></a>
```

Navigation syntax

- Parts of paths can be composed easily with the array syntax
- Array syntax concatenates the parts with /
- So the below are the same:

```
this.router.navigateByUrl('todos/1/tasks');
this.router.navigate(['todos', 1, 'tasks']);
```

```
<a [routerLink]="'todos/1/tasks'"></a>
<a [routerLink]="['todos', 1, 'tasks']"></a>
```

Absolute vs. Relative Navigation

- If the path starts with a slash (/), it is an absolute navigation.
- Example:

```
// Assuming we are now in example.com/todos/1
this.router.navigateByUrl('tasks'); // example.com/todos/1/tasks
this.router.navigateByUrl('/tasks'); // example.com/tasks
```

- ../ can be used to go one level up

```
// Assuming we are now in example.com/todos/1/tasks
this.router.navigateByUrl('../'); // example.com/todos/1
```

Router Guards

- Guards allow changing the behaviour of routing for certain routes
- Guards can be registered for navigation into and out of route
- Guards can cancel the routing and instead redirect user to somewhere else
- Example use cases:
  - Authentication
  - Preloading data for component
  - Logging
  - "You have unsaved changes. Are you sure you want to leave?"

Guard Types

- 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

Guard Example

- Usage:

app.module.ts

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

auth-guard.ts

```
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);
  }
```

Router URL Strategies

- Two strategies for URL formation:
  - `PathLocationStrategy`: HTML5 *pushState* style (`example.com/todos/1`) (default)
  - `HashLocationStrategy`: Hash URLs (`example.com/#/todos/1`)
- Setting the strategy:

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

Forms

- Angular provides highly advanced form management functionality:
  - Two-way data binding
  - Change tracking
  - Validation
  - Error handling

Template-driven vs. Reactive Forms

- Angular forms can be either template-driven or reactive
- Both have their own `NgModule`: `FormsModule` and `ReactiveFormsModule`
- Both modules are in `@angular/forms` npm package
- The underlying principles are the same but the usage is different
- Both can be used within a single application, even on single component
- Only template-driven forms are covered in this training, as they are enough for most use cases

Reactive Forms

- As the name suggests, based on reactive programming (covered tomorrow)
- Form controls defined in component instead of template
- Require more boilerplate code
- For complex use cases
- More easy to test

Reactive Forms Example

```
import { Component } from '@angular/core';
import { FormGroup, FormBuilder, Validators } from '@angular/forms';

export class TodosComponent {
    todosForm: FormGroup;

    constructor(private fb: FormBuilder) {
        this.todosForm = this.fb.group({
          name: ['', Validators.required],
          done: [false],
        });
    }
}
```

```
<form [formGroup]="todosForm">
    <input class="form-control" formControlName="name">
    <input type="checkbox" formControlName="done">
</form>
```

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>
```

Using Template-local Variables

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

```
<form>
  <input [(ngModel)]="name" `name="name"` `#nameModel="ngModel"` required />
  <input [(ngModel)]="email" `name="email"` `#emailModel="ngModel"` required />
  <div *ngIf="`nameModel.invalid || emailModel.invalid`">
    Either name or email is invalid
  </div>
<form>
```

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`

```
.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>
```

Forms - Accessing Form Inside Component

- Template-driven forms can be accessed from component with `@ViewChild('myForm')` annotation:

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

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

Pipes

- Simple display-value transformations
- Similar concept as *filters* in AngularJS
- Angular provides few commonly needed pipes, e.g.: `uppercase`, `lowecase` and `date`

```
  <!-- user.name is initially "john doe" -->
<div>{{user.name` | uppercase`}}</div> <!-- JOHN DOE -->
```

Pipe arguments

- Pipes can take arguments that modify its behavior:

```
<div>{{user.birthDay | date`:'fullDate'`}}</div> <!-- Friday, April 15, 1988 -
<div>{{user.birthDay | date`:'yyyy-MM-dd HH:mm a z':'+0900'`}}</div> <!-- 1988
```

Multiple pipes

- Pipes can be piped (like in UNIX)

```
<div>{{user.birthDay` | date:'fullDate' | uppercase`}}</div>
  <!-- FRIDAY, APRIL 15, 1988 -->
```

Custom Pipes

- Declared with `@Pipe` annotation
- `PipeTransform` interface with `transform` method
- `transform` takes the value as first argument and the optional arguments after it
- 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: number): number {
    return Math.pow(value, exponent);
  }
}
```

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

Generating a Pipe

Browse to root of the project and run:

```
ng generate pipe capitalize
```

or abbreviated one:

```
ng g p capitalize
```

This will:

- Create a file called `capitalize.pipe.ts` in the root of `app/` folder along with the test stub
- add it as declaration in `AppModule` so it is available in the templates

Directives

- There are three kinds of directives in Angular:
  - Components, which are basically selectors with templates and inputs and outputs
  - Structural directives, `ngFor` and `ngIf` are examples of these
  - Attribute directives
- Here we talk about attribute directives which can be used to decorate the elements

Directives - Example

- A directive for setting background color to yellow

```
import { Directive, ElementRef, Input } from '@angular/core';
@Directive({
    selector: '[myHighlight]'
})
export class HighlightDirective {
    constructor(el: ElementRef) {
       el.nativeElement.style.backgroundColor = 'yellow';
    }
}
```

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

Directives - Events

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

```
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>
```

Reactive Programming With Angular

Background

- Reactive programming is programming with asynchronous data streams
- JavaScript is asynchronous by design
  - HTTP requests
  - Timeouts
  - UI events (clicks, key presses, etc.)
  - *How to handle all this?*

Callbacks

- Traditionally solved by registering *callback* functions to be executed upon completion of task
- E.g. `setTimeout` that calls callback function once given amount of milliseconds has passed

```
window.setTimeout(() => {
    // Executed after one second
}, 1000);
```

Problem: Messy Code

- Using callbacks quickly leads to messy code with multiple nested functions that is hard to follow and rationalize

```
getData((x) => {
    getMoreData(x, (y) => {
        getMoreData(y, (z) => {
            ...
        });
    });
});
```

- For more information google for *callback hell*

Solution: Promises

- *Promise* is a promise of providing a value later
- Promise constructor takes single argument that is a function with two parameters:
  - `resolve`: function to be called when we want to indicate **success**
  - `reject`: function to be called when we want to indicate **failure**

```
new Promise((resolve, reject) => {
  if (...) resolve(x);
  else if (...) resolve(y);
  else reject();
});
```

- Both functions allow arguments that are provided for promise consumer

Promises are Resolved or Rejected

- Promises are consumed by calling `then` on them. `then` takes two arguments: success and failure handler

```
somethingReturningPromise().then(
  (value) => { // Resolved
    // Handle success case
  },
  (value) => { // Rejected
    // Handle reject case, e.g. show an error note
  });
```

Promise Chaining

- Promises can be "chained" by calling then multiple times in a row
- Each `.then()` will change the value of the promise by returning a new value
- If the value returned it is a promise, it will be waited for

```
fetch('/users') // Make the HTTP request
  .then(response => response.json()) // .json() will return a promise
  .then(json => json.users) // Map the result to contain only the "users" field
  .then(users => alert('Found ' + users.length + ' users')); // Show an alert with
```

Problem: Stream Handling and Disposability

- Promises don't work for streams, they are just to subscribe for **single events**
- Promises can't be **cancelled**

Solution: Observables

- Generalization of promises for streams
- A way for representing **asynchronous event streams**
  - e.g. mouse clicks, WebSocket streams
- Can also be used for single events e.g. HTTP requests

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**

Idea:

"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));
```

Streams can be manipulated with traditional array conversion functions such as *map* and *filter*

```
observable
  .filter(node => node.children.length > 2)
  .map(node => node.name);
```

You can *merge*, *concat* and do other operations on streams to **produce new streams** from the existing ones

```
const resultStream = stream1.merge(stream2);
```

ReactiveX operators

Observables can also be created from e.g. **objects**, **maps** and **arrays**

```
Rx.Observable.of(42);
Rx.Observable.from([1,2,3,4]);
Rx.Observable.range(1,10);
```

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
  - E.g. observable that produces events that are saved into the memory

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

  - Sequence will not stop until unsubscribed

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

Demo

Catching Errors

- Observables die on errors
- The way to survive from errors is by catching them and returning a new observable sequence

```
observable.catch((error) => {
    console.log(error);
    return Rx.Observable.of([1, 2, 3]);
};
```

Hot vs. Cold Observables

- Cold observables start running **upon subscription**
  - E.g. http request
- Hot observables are already producing values **before the subscription** is active
  - E.g. mouse move events

Observables in Angular

- Observables used exclusively instead of promises
  - E.g. HTTP requests only result in single event (one response) but they are modeled as observables

```
this.httpClient.get('url/restapi/resource') // Returns observable
  .subscribe(
      data => { this.data = data}, // Success
      err => console.error(err), // Failure
      () => console.log('done') // Done
  );
```

Observables in Angular

- Changes in route parameters are propagated through an observable sequence

```
constructor(route: ActivatedRoute) {
  route.params.subscribe(params => this.index = +params['index']);
}
```

Exercises

Open exercise instructions

RxJS 5.5

- RxJS 5.5.0 introduced major change to RxJS called *pipeable operators*
- Importing is a mess with 5.5 but RxJS 6 will fix it
- Read more

Pipeable Operators Example

Pre RxJS 5.5

```
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/observable/range';
import 'rxjs/add/operator/filter';
import 'rxjs/add/operator/map';

const source$ = Observable.range(0, 10);
source$
  .filter(x => x % 2 === 0)
  .map(x => x + x)
  .subscribe(x => console.log(x))
```

RxJS 5.5

```
import { range } from 'rxjs/observable/range';
import { map } from 'rxjs/operators/map';
import { filter } from 'rxjs/operators/filter';

const source$ = range(0, 10);
source$.pipe(
  filter(x => x % 2 === 0),
  map(x => x + x)
).subscribe(x => console.log(x))
```

RxJS 5.5 Renaming

Some of the operators are reserved words in JavaScript:

- `do -> tap`
- `catch -> catchError`
- `switch -> switchAll`
- `finally -> finalize`

RxJS 5.5 Pros

- No more "prototype patching" -> Tree-shaking possible -> Smaller bundle sizes
- Custom operators are easier to make
- Better tooling support by linters and compilers

RxJS 6

- Released 05/2018
- Major changes:
  - Simpler imports (`import { map, filter } from 'rxjs/operators'`)
  - Errors thrown asynchronously
  - Deprecations
  - New operator (`throwIfEmpty`)
- Provides compatibility library (`rxjs-compat`) to support the migration from 5 to 6
- See Ben Lesh's (RxJS 5 author) presentation in ngConf 2018 for more details (slides, video)

Testing

Unit testing

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

Jasmine

- 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

First Jasmine Test

- Class under test

```
export class Person {
  constructor(private name: string) {
  }

  getName() {
    return this.name;
  }
}
```

First Jasmine Test

- Test case

```
describe('Person', () => {
  let person;

  beforeEach(() => {
    person = new Person('John');
  });

  it('should return name', () => {
    expect(person.getName()).toBe('John');
  });
});
```

Spies

- Test double functions AKA spies let you stub any function and track calls to it

```
it('tracks that the spy was called', () => {
  spyOn(person, 'getName');

  person.getName();
  expect(person.getName).toHaveBeenCalled();
});
```

Spies

```
it('stubs the function return value', () => {
  spyOn(person, 'getName').and.returnValue('Jane');

  expect(person.getName()).toBe('Jane');
});
```

Angular Testing Platform (ATP)

- TestBed for wiring angular components for testing
- Inject mock dependencies
- Testing components with async behavior

Component Under Test

- Suppose we had the following component:

```
@Component({
  selector: 'videos',
  templateUrl: 'videos.component.html'
})
class VideosComponent {
  videos: Video[];

  constructor(videoService: VideoService) {
    this.videos = videoService.getList();
  }
}
```

Wiring Up

- We can setup the test environment with mocks using TestBed:

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

Access to Tested Component

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

comp = fixture.componentInstance;

debugElement = fixture.debugElement;

nativeElement = fixture.nativeElement;
```

Async with ATP

- Let's change the implementation of the VideoService to return a promise:

```
ngOnInit() {
  this.videoService.getList().then(videos => this.videos = videos);
}
```

- This means that we'll have to deal with asynchronous behavior

async()

- We can run test code within asynchronous zone

```
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);
  });
}));
```

fakeAsync()

- Or within fake asynchronous zone

```
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:

```
npm run test
```

E2E testing

- End-to-End (E2E) tests test flow of the application
- Ensures that the components of the application function together as expected
- E2E tests often define use cases of the application

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:

```
npm run e2e
```

Example spec

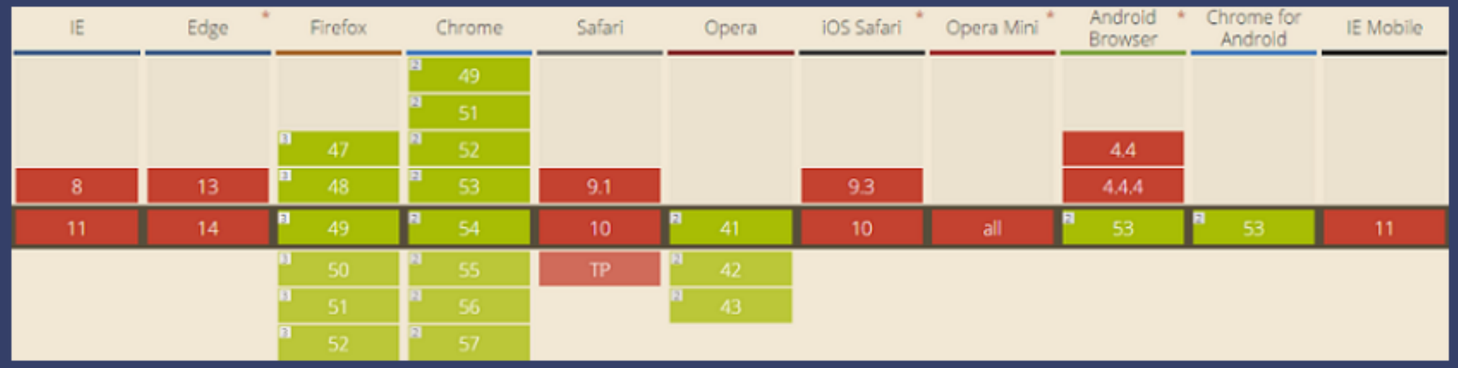
```
describe('Protractor Demo App', () => {
  it('should have a title', () => {
    browser.get('http://demo.com/protractor-demo/');

    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

Browser Support



Polyfill for older browsers

Adding Polyfill to Angular CLI

```
npm install --save web-animations-js
```

angular-cli.json

```
"scripts": [
  "../node_modules/web-animations-js/web-animations.min.js"
],
```

Animations - Supported Properties

- List available in the standard
- Basically every color (background, border, etc.) and dimension (width, height, size, etc.)
  - `background-color`
  - `border-width`
  - `max-height`

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

Trigger is bound to the certain element

my.component.ts

```
@Component({
  animations: [
    trigger('myTrigger', [
      ...
    ])
  ]
})
export class MyComponent {
  state = 'state1';
}
```

my.component.html

```
<div [@myTrigger]="state">
  My element
</div>
```

States

States are the possible values for triggered value

my.component.ts

```
trigger('myTrigger', [
   state('state1', ...),
   state('state2', ...)
])
export class MyComponent {
  state = 'state1';

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

Styles

States have *styles*

my.component.ts

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

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

Transitions

Transitions between states are animated

my.component.ts

```
trigger('myTrigger', [
  state(...),
  transition('state1 => state2', ...),
  transition('state2 => state1', ...)
])
export class MyComponent {
  state = 'state1';

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

Animations

Animations have duration and timing functions etc.

my.component.ts

```
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)"`

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

Zone.js

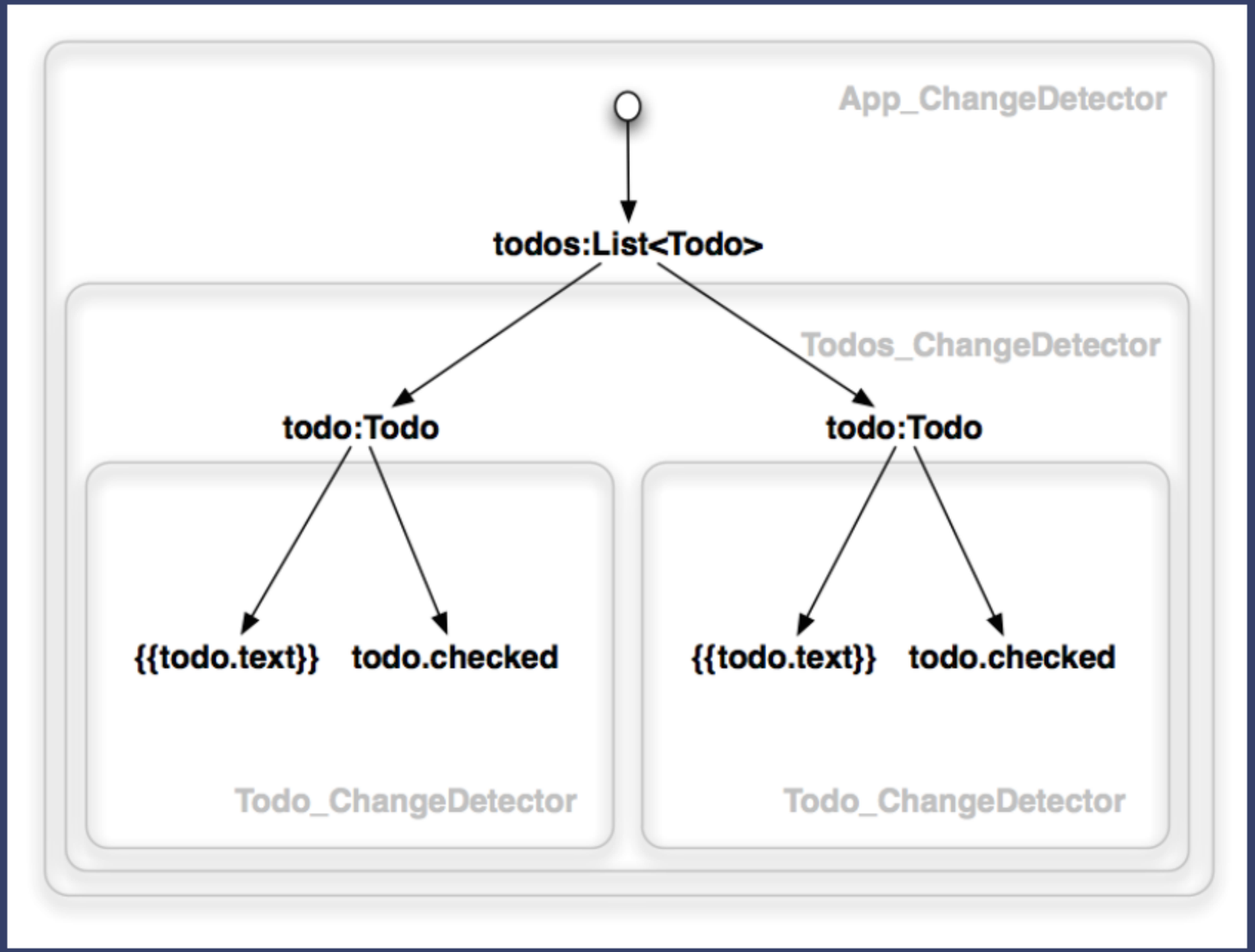
- Implements concept of zones (inspired by Dart) in JavaScript
- "A Zone is an execution context that persists across async tasks"
- In practice makes it possible to track the asyncronous calls made (HTTP, timers and event listeners) within the zone
- Angular uses zones internally to track changes in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

Change Detection - General

- 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

Change Detection - Angular

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


App_ChangeDetector
todos:List<Todo>
Todos_ChangeDetector
todo:Todo
todo:Todo
{{todo.text}}
todo.checked
{{todo.text}}
todo.checked
Todo_ChangeDetector
Todo_ChangeDetector

Change Detection - Simplified Implementation

```
// very simplified version of actual source
class ApplicationRef {
    changeDetectorRefs: ChangeDetectorRef[] = [];

    constructor(private zone: NgZone) {
      this.zone.onTurnDone
        .subscribe(() => this.zone.run(() => this.tick()));
    }

    tick() {
      this.changeDetectorRefs
        .forEach((ref) => ref.detectChanges());
    }
}
```

Change Detection - Strategies

- Change detection strategy describes which strategy will be used the next time change detection is triggered
- Angular has six change detection strategies:
  - **CheckOnce**: After calling *detectChanges* the mode of the change detector will become *Checked*.
  - **Checked**: Change detector should be skipped until its mode changes to *CheckOnce*.
  - **CheckAlways**: After calling *detectChanges* the mode of the change detector will remain *CheckAlways*.
  - **Detached**: Change detector sub tree is not a part of the main tree and should be skipped.
  - **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`})
```

Change Detection - Angular Performance

- Change detection is one of the key functionalities of Angular and thus it is highly optimized
- CDs get VM optimized monomorphic classes generated for them at runtime
- Change detection is always single-pass (stable) because of uni-directional top-to-bottom flow
- More optimizations possible with *immutables* and *observables*

Lazy Loading

Background

- Web applications are used more and more with mobile devices
- Many countries still have slow internet connections
- Load time has a huge impact on user experience

Lazy Loading

- Traditional apps load everything in the beginning
- Lazy loading means only loading parts after initial page load

Default Bundling

```
chunk {0} polyfills.b6b2cd0d4c472ac3ac12.bundle.js (polyfills) 59.7 kB [initial] [rendered]
chunk {1} main.37ad1d8b32fb04fc5110.bundle.js (main) 290 kB [initial] [rendered]
chunk {2} styles.ac89bfdd6de82636b768.bundle.css (styles) 0 bytes [initial] [rendered]
chunk {3} inline.318b50c57b4eba3d437b.bundle.js (inline) 796 bytes [entry] [rendered]
```

Chunks are:

- *polyfills.bundle.js*: Polyfills required to run the app (see *polyfills.ts*)
- *main.bundle.js*: Actual application code along with vendor code (Angular)
- *styles.bundle.css*: Styles for the application
- *inline.bundle.js*: Tiny webpack loader to load the app

Custom Chunks

```
chunk {0} 0.e4ae2f1a9d74e23ecb84.chunk.js () 47.6 kB  [rendered]
chunk {1} polyfills.194f3db5c95bb635c66b.bundle.js (polyfills) 59.7 kB [initial] [rendered]
chunk {2} main.472a86e996efec77ab48.bundle.js (main) 246 kB [initial] [rendered]
chunk {3} styles.ac89bfdd6de82636b768.bundle.css (styles) 0 bytes [initial] [rendered]
chunk {4} inline.8ea1e33f9705c60c834d.bundle.js (inline) 1.4 kB [entry] [rendered]
```

Angular Module Loading

- By default, modules are loaded eagerly when the application starts
- Modules can loaded lazily by the router with `loadChildren`

Angular Module Loading

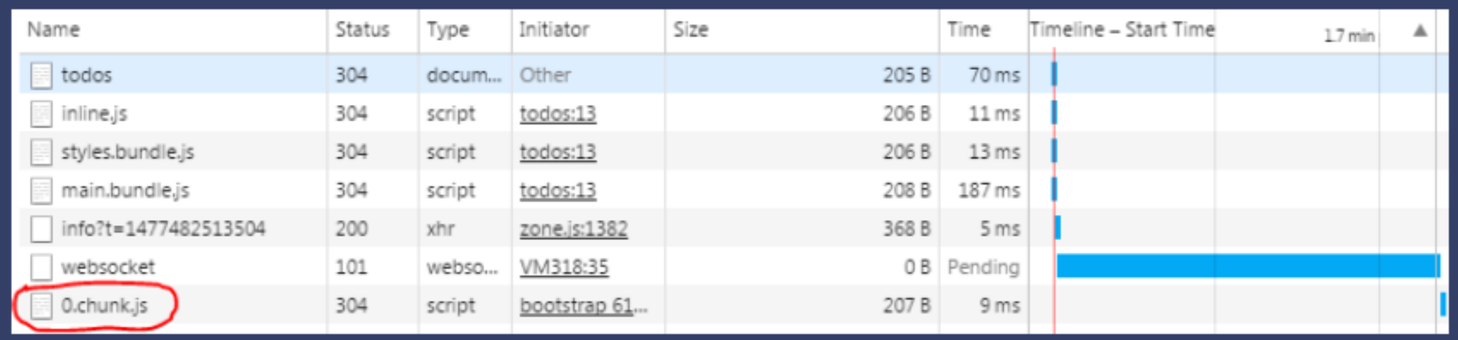
Main bundle is loaded when application starts

Name	Status	Type	Initiator	Size	Time	Timeline – Start Time
todos	304	docum...	Other	205 B	70 ms	
inline.js	304	script	todos:13	206 B	11 ms	
styles.bundle.js	304	script	todos:13	206 B	13 ms	
main.bundle.js	304	script	todos:13	208 B	187 ms	
info?t=1477482513504	200	xhr	zone.js:1382	368 B	5 ms	
websocket	101	webso...	VM318:35	0 B	Pending	

Angular Module Loading

A feature module is loaded when it is routed to the first time

```
<a [routerLink]="['feature']">Click me</a>
```



Name	Status	Type	Initiator	Size	Time	Timeline – Start Time 1.7 min
todos	304	docum...	Other	205 B	70 ms	
inline.js	304	script	todos:13	206 B	11 ms	
styles.bundle.js	304	script	todos:13	206 B	13 ms	
main.bundle.js	304	script	todos:13	208 B	187 ms	
info?t=1477482513504	200	xhr	zone.js:1382	368 B	5 ms	
websocket	101	webso...	VM318:35	0 B	Pending	
0.chunk.js	304	script	bootstrap 61...	207 B	9 ms	

Angular CLI

- Angular CLI has built-in support for lazy loading
- Works for both development and production builds
- If `loadChildren` is detected chunks will be generated -> no extra configuration required

Router Configuration

app.module.ts

```
const routes: Routes = [
  {
    path: '',
    component: AppComponent
  },
  {
    path: 'todos',
    loadChildren: './todos/todos.module#TodosModule'
  }];

@NgModule({
  ...
  imports: [
    ...
    RouterModule.forRoot(routes)
  ],
})
```

Router Configuration

todos.module.ts

```
const routes: Routes = [
  {
    path: '',
    component: TodosComponent
  },
  {
    path: ':id',
    component: EditTodoComponent
  }
];

@NgModule({
  ...
  imports: [
    ...
    RouterModule.forChild(routes)
  ],
})
export class TodosModule {}
```

Demo

https://github.com/RoopeHakulinen/angular-lazy-loading

Analyzing Bundle Sizes

```
{
  "scripts": {
    ...
    "analyze": "ng build --prod --stats-json && webpack-bundle-analyzer dist/stats
  }
}
```

```
npm install webpack-bundle-analyzer --save-dev
npm run analyze
```

Useful Resources

- Thoughtram: Highly advanced articles about Angular, RxJS and other related libraries.
- Style guide: Angular's official style guide. Worth skimming through.
- ngrev: Graphical tool for reverse engineering of Angular projects.
- Ahead-of-Time compilation - Boost build times by compiling offline (not in browser)
- Release schedule - Angular's release schedule and versioning info
- Web workers support - Boost up performance with threading
- Progressive Web Apps (PWA) - Offline-capable, app-like web sites supporting push notification built with Angular
- Lazy loading - Only load essentials parts initially
- Universal apps - render Angular app as HTML on server
- Migration from AngularJS - Migrating from AngularJS to Angular is possible
- Frameworks built on top of Angular for different platforms:
  - Electron for desktop apps
  - Ionic (works on top of Apache Cordova) for hybrid mobile apps
  - NativeScript for native UI mobile apps