Introduction

About Lund&Bendsen

- Our core competencies is sharing knowledge on technologies related to IT application and system development.
  - Training, consulting, reviews, mentoring, development – primarily in Java and related languages and frameworks.
- Often for larger corporations with internal IT development
  - DR, Nordea, PFA Pension, Nets, CS Topdanmark, TDC, ATP, DSB, BEC, SDC, Systematic (and many more)

Website: www.lundogbendsen.dk

Contact: info@lundogbendsen.dk

Lund&Bendsen
Børsen gazelle 2008 2009
Email
Adgangskode
Log ind
Forside | Undervisning | Rådgivning | Udvikling | Om Lund&Bendsen A/S

Eksperter i undervisning og rådgivning indenfor Java-baseret systemudvikling

OPLEV AT FORMIDLING & FAGLIGHED MØDES

Vores speciale er undervisning og formidling af teoretisk og praktisk knowhow om udvikling med Java EE og relaterede frameworks.

Læs mere om vores undervisning

FÅ MERE VÆRDI - OGSÅ EFTER UNDERVISNINGEN

Adgang til undervisningsportalen klæder dig på til at skabe værdi ude i din virksomhed - også efter undervisningen er slut.

Log ind på my.lundogbendsen.dk

OPNÅ UDSYN & INDSIGT I ARKITEKTUR & DESIGN

Tilmeldt vores faglige nyhedsbrev modtager du tekniske artikler og vores betragtninger om relevante nyheder og faglige udfordringer.

Eksempler på vores faglige artikler

RÅDGIVNING TIL SYST EJER & PROJEKTLEDE

Få innovativ sparring e opinion på tekniske mu faldgruber når jeres IT designes eller opdatere

Læs mere om vores rå

About the instructor

- Flemming Bregnvig
  - Owner of loopme
    - specializing in TypeScript development
  - Instructor at Lund&Bendsen
  - Instructor at Prosa
- Previous & current:
  - nes tech
  - letpension
  - NOTA
  - SDC
  - etc.
- Primary techologies: TypeScript, JavaScript, CSS
- Github: github.com/bregnvig
- E-mail: flemming@loopme.dk

Practical information

- The course starts at 9:00
- Lunch at 12:00 - 12:45
- Ends at 16:00

If this suits you?

- Contakt:
  - +45 33 861 861 (Lund&Bendsen)
  - info@lundogbendsen.dk

Prerequisites

We assume a certain knowledge of HTML 4 or 5, JavaScript and/or TypeScript.

Rationale behind the course design

- Syntax
  - the TypeScript & Angular syntax
- Semantics
  - the meaning of what you write
- Pragmatics
  - the idea behind writing in a certain way
- Practical exercises and examples
  - learn by doing

Tell us about yourself...

- What do you expect to gain from this course?
  – why are you participating?
  – what do you hope to learn?

- How much do you know already about:
  - JavaScript
  - TypeScript
  - HTML – 4 and 5

Training material

- Book containing
  - Copy of all the slides
  - Exercises (Labs)
- USB Memory containing:
  - Software necessary for the course
  - Assorted documentation
  - Sample programs
  - Code used in the exercises
  - Suggested solutions to the exercises

Online material

- www.kursusportal.it
  - Overview of all your courses at Lund&Bendsen
  - Access to view course slides
  - Access to material published by the instructor during the course
  - Access to forum for the participants
  - Access to course evaluation
  - Download of course certificate
  - Possibility to express interest in other courses

LAB INTR-1

Install the software

Introduction to Angular

What is Angular?

What?:

Angular is a platform that helps you create advanced web applications in HTML using TypeScript, JavaScript or ~~Dart~~

Why?

Big web applications can easily get messy. A framework helps you structure your code, separate your concerns and remove trivial code.

Angular Alternatives

Some well-known alternatives to Angular:

- AngularJS
- React
- vue.js
- Knockout
- jQuery + jQuery UI or jQuery Mobile

- ... and others...

Why Angular ?

Why should you choose Angular?

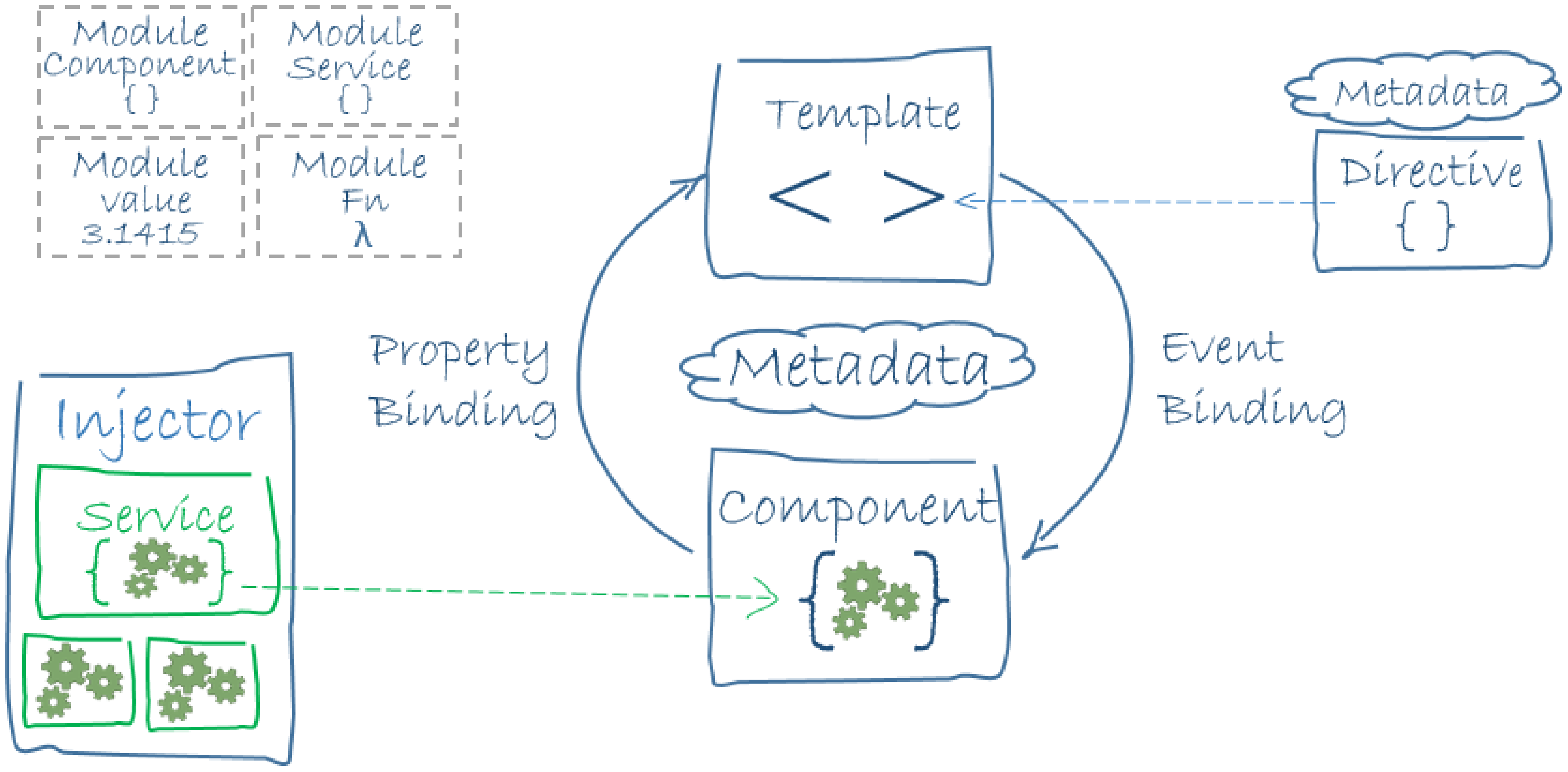
- Angular contains a lot of features – for HTML generation, data binding, dependency injection etc.
- Backed by Google
- Active community
- Well documented
- Performance
- Multiple platforms (Web, Electron and NativeScript)

Why Not Angular ?

Why should you **not** choose Angular?

- If you dislike JavaScript/TypeScript
- If you already have React skills
- If you already have vue.js skills
- If you have a well structured AngularJS application

Angular Architecture



Components

- A component controls a patch of real estate, also known as the view
- A component is implemented as a class
- The components class interacts with the view through an API of methods and properties
- A component will be created, updated and destroyed by Angular. (Lifecycle)

Componts

```
export class SidebarComponent implements OnInit {

  public playgrounds: IPlayground[];

  public constructor(private service: PlaygroundService) {
  }

  public ngOnInit() {
       this.playgrounds = this.service.getPlaygrounds();
  }
}
```

Template

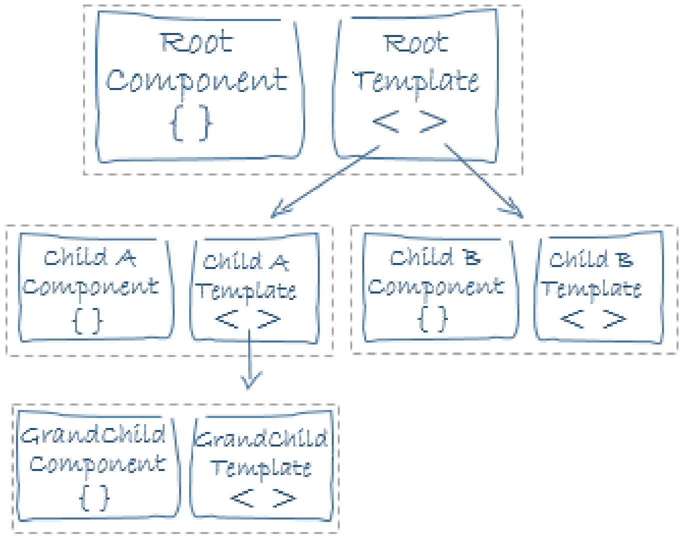
- A template is firmly attached to a component
- A template tells Angular how to render the component
- A template looks like regular HTML much of the time, and then it gets a bit strange!

Template

```
<aside>
  <form>
    <div>
         <input
           id="filterPlaygrounds"
           placeholder="Filtrer legepladser"
           [(ngModel)]="filter‟
           autocomplete="off">
    </div>
  </form>
  <ul>
    <li *ngFor="let playground of playgrounds"
      (click)="selectPlayground(playground)">
      <playground-info [playground]="playground"></playground-info>
    </li>
  </ul>
</aside>
```

Template

- Strange attributes
  - `[(ngModel)]`
  - `*ngFor`
  - `(click)`
- Strange element
  - `<playground-info>`



Metadata

- Without metadata a component class is just a class, and not really a component
- Metadata is attached using decorators when we are using TypeScript
- The decorator must be placed directly above the component class
- A decorator is a function, and often requires parameters

Metadata

```
@Component({
  selector: `app-sidebar',
  templateUrl: 'sidebar.component.html`
})
export class SidebarComponent implements OnInit, OnDestroy {
        …
}
```

Template example

```
<nav>…</nav>
<app-sidebar></app-sidebar>
```

Without data binding

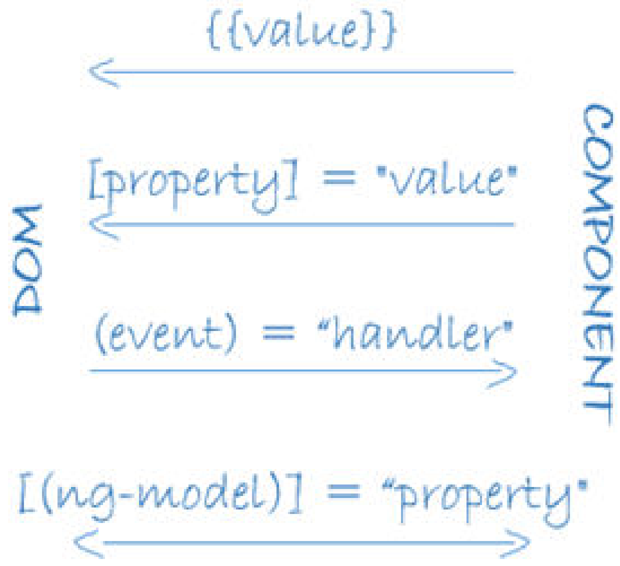
```
jQuery('#myElement').addClass('highlight');
```

Data binding

- Data binding coordinates between the template and the component class
- Data binding is done by adding markup to the HTML template
- Four forms of data binding: to the DOM, from the DOM or in both directions

Data binding

- Interpolation
  - `<li>{{playground.title}}</li>`
- Property binding
  - `<img [src]=”imageSrc">`
- Event binding
  - `<div (click)="selectPlayground(playground)">`
- Two way binding
  - `[(ngModel)]="filter"`



Directives

- A directive is a class with directive metadata
- Three kinds of directives
  - Template (The component)
  - Structural
  - Attribute
- Angular renders the templates according to the instructions given by the directives

Directives

- Structural
  - Adds or removes DOM elements

```
<div *ngFor="let playground of playgrounds“>
```

- Attribute
  - Alters appearance or behavior of an DOM element

```
<img [src]=”imageSrc">
```

Services

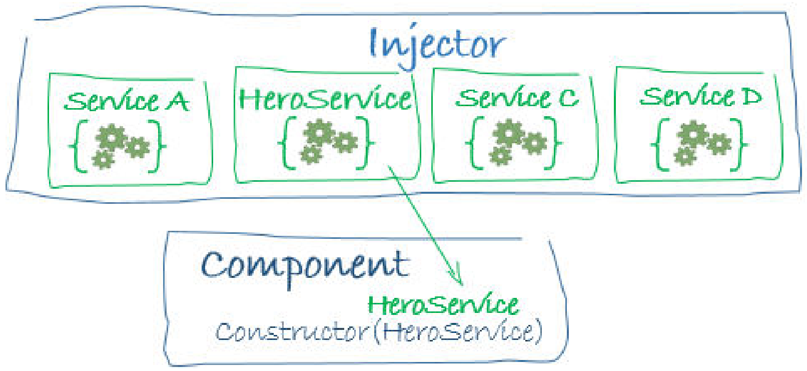
- A service is a concept, it is not an Angular specific thing
- A service is all about business logic
  - Can be a class, often is
  - Can be a function
  - Can be a value
- A service should have a narrow well defined purpose

Dependency injection

Angular uses dependency injection as a way to provide a component, service or pipe with the dependencies it may require.

Angular maintains a container of these dependable services

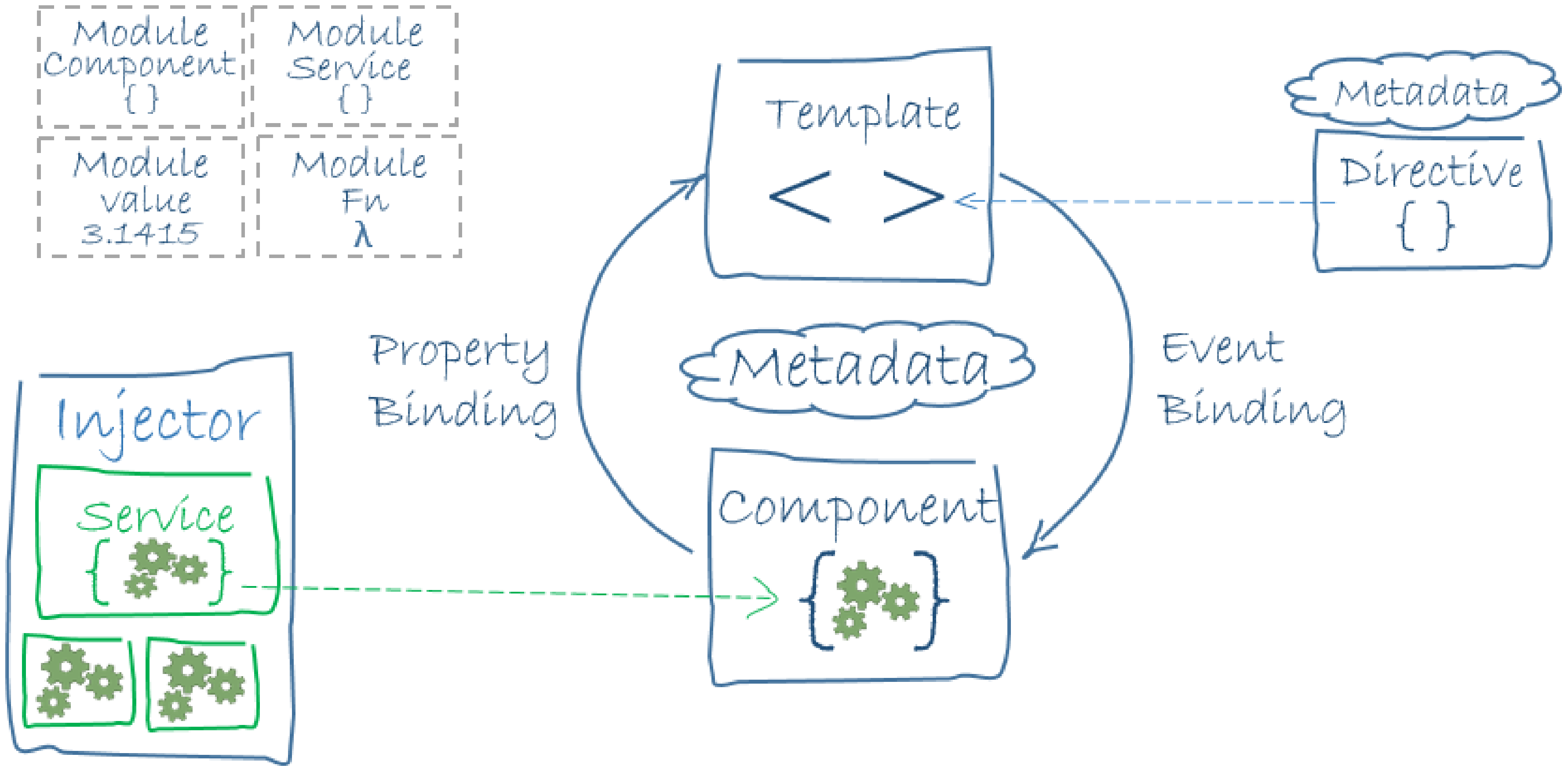
Dependency injection



Angular modules

- A collection of what we talked about
- Help organize an application into cohesive blocks of functionality
- It defines which components, services, pipes etc. that belongs together
- Hides implementation details
- Exports stuff to share

Angular Architecture



TypeScript

JavaScript that scales

A TypeScript primer

TypeScript is a typed superset of JavaScript that compiles to plain JavaScript

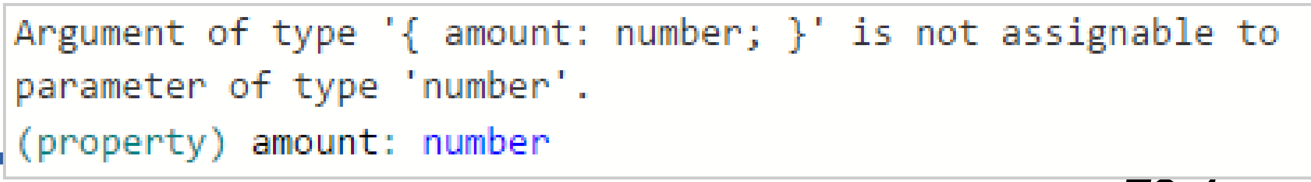
What do we get?

- Types and inferred types
- Classes
- Access modifiers (public, protected & private)
- Interfaces
- Generics
- Compiler
- Compile time checking
- Refactoring
- Alot of ECMAScript 2015 & 2017 features

Simple types

```
// Plain JavaScript
function calculateVAT(amount) {
    return amount * 1.25;
}
alert(calculateVAT(100));
alert(calculateVAT({amount: 100}));

// Using TypeScript
function calculateVAT(amount: number) {
      return amount * 1.25;
}
alert(calculateVAT(100));
alert(calculateVAT({amount: 100}));
```



```
Argument of type '{ amount: number; }' is not assignable to
parameter of type 'number'.
(property) amount: number
```

Simple class JavaScript (ECMAScript 5)

```
var Vat = (function () {
    function Vat() {
    }
    Vat.prototype.calculate = function (value) {
        return Vat.VAT * value;
    };
    Vat.VAT = 1.25;
    return Vat;
}());
var v = new Vat();
console.log(v.calculate(100));
console.log(v.calculate({amount: 100}));
```

Simple class TypeScript

```
class Vat {
    private static VAT = 1.25;
    public calculate(amount: number) {
        return amount * Vat.VAT;
    }
}

let v = new Vat();
console.log(v.calculate(100));
Vat.VAT = 1.25;
// console.log(v.calculate({amount: 100}));
```

```
error TS2341: Property ‘VAT' is private and only
accessible within class ‘Vat'.
```

Interface

```
interface IVATCalculator {
    calculate(amount: number): number;
}
class Vat implements IVATCalculator {
    private static VAT = 1.25;
    public calculate(amount: number) {
        return amount * Vat.VAT;
    }
}
let v:IVATCalculator = new Vat();
console.log(v.calculate(100));
```

Interface

With read only, required and optional properties

```
interface Playground {
  readonly id: string;
  readonly name: string;
  readonly addressDescription?: string;
  readonly description?: string;
  readonly position: Coordinate;
}
```

Generics

```
interface IDataManager<T> {
    getAll(): T[];
    save(entity: T): T
}

class User {
}

class UserManager implements IDataManager<User> {
    getAll(): User[] {
        // code
        return [];
    }
    save(user: User): User {
        // code
        return user;
    }
}
```

Compiling

- TypeScript compiles into JavaScript by using the TypeScript compiler `tsc`
- One file: `tsc simple-type.ts`
  - Or `tsc -w simple-type.ts` to watch the file for changes
- Using `tsconfig.json` configuration file (The preferred way)

Gotchas

Type, classes, interfaces, access modifiers, generics are all just compile time & tool time help

Everything is gone in the compiled files!

Proof

```
var User = (function () {
    function User() {
    }
    return User;
}());

var UserManager = (function () {
    function UserManager() {
    }
    UserManager.prototype.getAll = function () {
        // code
        return [];
    };
    UserManager.prototype.save = function (user) {
        // code
        return user;
    };
    return UserManager;
}());
```

TypeScript/JavaScript modules

- An application is assembled from many TypeScript modules
- A TypeScript module is a cohesive block of code, dedicated to a single purpose
- A TypeScript module export *something*. E.g class, function or value
- In order to use the exported *something* in another file, we must import the *something* in the other file (Just like in Java or C#)

TypeScript modules

- Two types of modules
  - A module per file. The module id is then the filename without extension
  - A barrel. A re-export of one or more modules from a single public façade. Also know as a library

TypeScript modules

app.component.ts

The exported *something*

```
export class AppComponent { }
```

The imported something

Module id

Other file

```
import { AppComponent } from './app.component';
```

Barrel id

Importing a library/barrel

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

ECMAScript 2015

- Arrow function
  - `(a: number, b: string): void => { … }`
- Default parameter values
  - `(foo: bar, count:number = 1) => { … }`
- Rest
  - `(…numbers: number[]) => { … }`
- `let` & `const`
- Template strings (With back ticks)
  - ``` `Hello ${greet}` = Hello world ```
  - Also works multi line

ECMAScript 2015

- Getter and setter methods

```
class Car {

  private _name: string

  constructor (name: string) {
    this._name = name;
  }

  set name (name: string) {
    this._name = name;
  }

  get name (): string {
    return this._name;
  }
}
```

LAB TS-1

Create a VAT calculator

Angular CLI

A command line interface for Angular

Angular & tooling

- Angular “requires” tooling
- Angular provides a CLI called Angular CLI
- Angular CLI is used for
  - Creating project
  - Serving content in development
  - Adding components, pipes, services, directives, modules, enums & classes
  - Lints code
  - Testing
  - Building for deployment

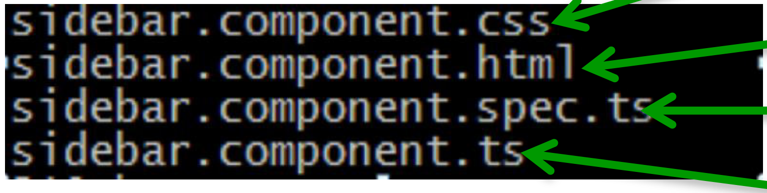
Angular CLI

- Install using NPM
  - `npm install -g @angular/cli`
- Creating new project (There are several more options)
  - `ng new <project-name> --directory <dir>`
- When developing use serve to watch and compile files
  - `ng serve`
- When building for production
  - `ng build --prod`

Angular CLI

- Generating scaffolding for components
  - `ng generate component <component-name>`
  - `ng g c <component-name>`

E.g **ng g c sidebar**



```
sidebar.component.css
sidebar.component.html
sidebar.component.spec.ts
sidebar.component.ts
```

Component specific styling

Component template

Component test file

Component class

Other generate options

```
– ng g module <path>/<module-name>
– ng g service <path>/<service-name>
– ng g class <path>/<class-name>
– ng g interface <path>/<interface-name>
– ng g pipe <path>/<pipe-name>
```

Other commands

- Lint the code
  - `ng lint` (You should do this in the IDE)
- Testing the code
  - `ng test`
- Extract i18n message from source code
  - `ng xi18n`
- Leave the Angular CLI
  - `ng eject`

File structure

- `src/app` – Your source files are placed here
  - `src/app/<component-name>` Your component is place here
- `src/environments` – Environment configurations are placed here
- `src/assets` – Your static assets

Adding additional libraries

- Add additional code and styling using the `angular.json` file
- Styling

```
"styles": [
      "css/app.less",
      "../node_modules/bootstrap-css-only/css/bootstrap.min.css",
      "../node_modules/font-awesome/css/font-awesome.min.css"
],
```

- Scripts

```
"scripts": [
      "../node_modules/moment/min/moment.min.js"
],
```

LAB TOOL-1

Create a project

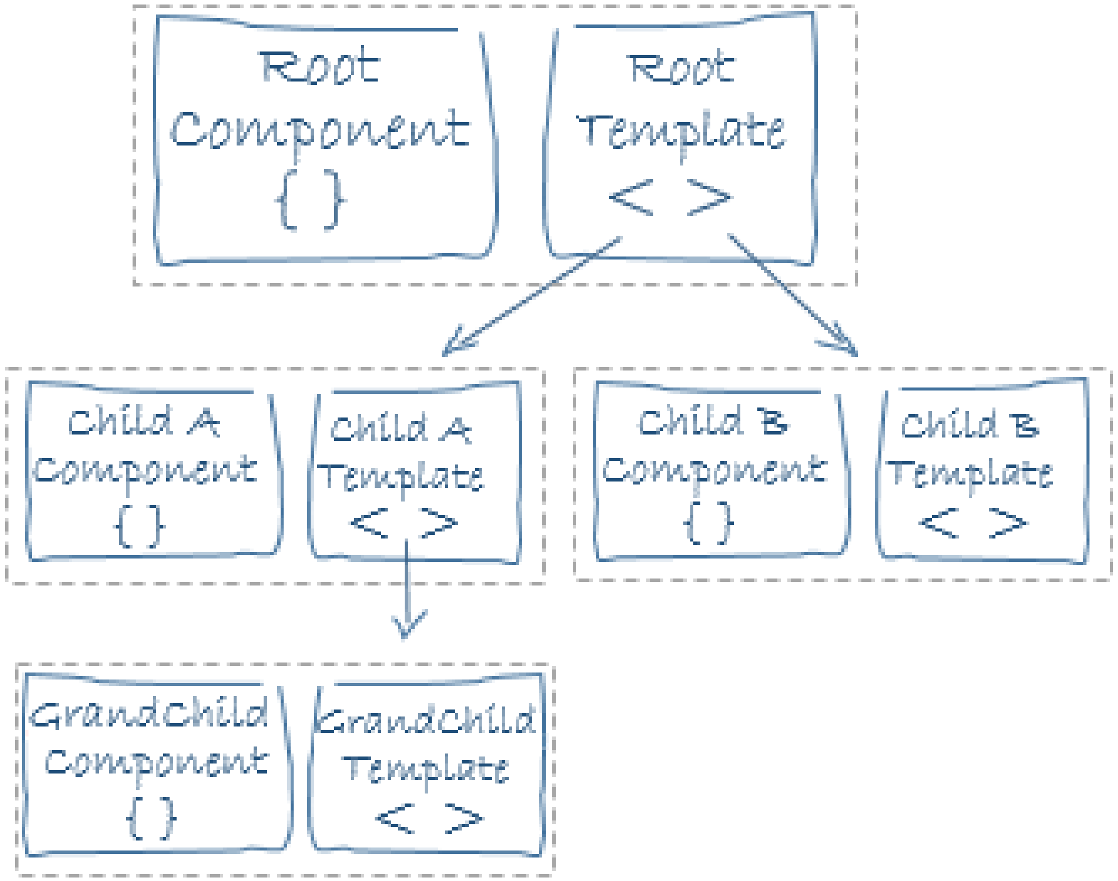
Components

where all Angular applications starts

Components

- All Angular applications has a root component
- An application consists of “many” components
- A component controls a piece of screen real estate
- A component consists of three parts
  - A TypeScript class
  - A decorator
  - A template (In line or as a separate file)
- Angular renders the view based on these three parts

Components


Root Component { }
Root Template < >
Child A Component { }
Child A Template < >
Child B Component { }
Child B Template < >
GrandChild Component { }
GrandChild Template < >

Component class

- Plain old TypeScript object
- Exposes properties and method to be used in the template

```
export class HelloComponent implements OnInit {

  public title: string;

  public ngOnInit() {
    this.title = 'Hello world!';
  }
}
```

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

@Component decorator

- Placed right above the component class
- selector – A CSS3 selector
- The template
  - template – An inline template. Use back ticks ``
  - templateUrl – A URL to the template file
- Styling – Specific only to this component!
  - styles – An array of strings containing CSS
  - styleUrls: An array of URLs to the CSS files
- And more…

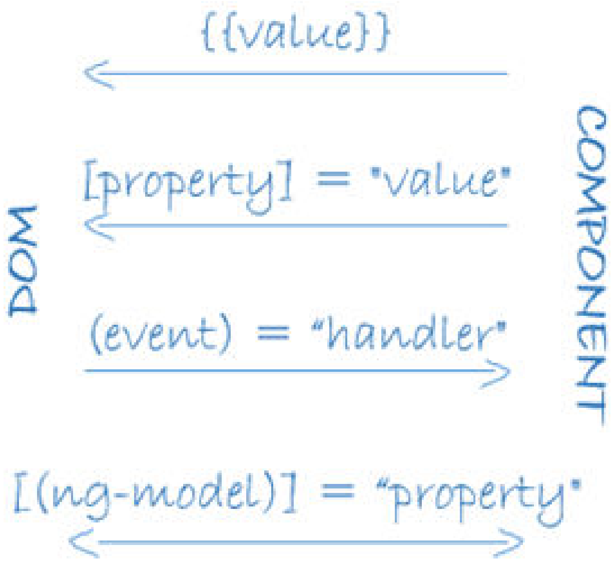
@Component decorator

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

```
<div>
  <p>
    I'm outer
  </p>
  <app-inner></app-inner>
</div>
```

Template

- HTML like syntax with Angular specific attributes and elements
- One way binding
  - `{{}}` - interpolation
  - `[]` - property
  - `()` - event
- Two way binding
  - `[(ngModel)]`



Template - interpolation

- From component to DOM
- All kind of values or object
- Use the elvis operator to allow to nulls
  - object?.property
- All kind of methods
- Expressions

Template - interpolation

```
<ul>
    <li>Property string: {{propertyString}}</li>
    <li>Property number: {{propertyNumber}}</li>
    <li>Property object: {{propertyObject.property}}</li>
    <li>Property null: {{propertyNull}}</li>
    <!--<li>Property null - will fail: {{propertyNull.property}}</li>-->
    <li>Property null - wont fail: {{propertyNull?.property}}</li>
    <li>Simple method: {{simpleMethod()}}</li>
    <li>Parameter method: {{parameterMethod('world')}}</li>
    <li>Simple expressions: {{40+2}}</li>
    <!--<li>New object - will fail {{new Date()}}</li>-->
</ul>
```

Template - property

- From component to DOM
- Use it to set properties in the DOM
- Use it for input into component (More on that later)

```
<ul>
  <li><span [class.red]="showRed">My red</span></li>
  <li><span [hidden]="hide">My hidden</span> </li>
</ul>
```

Template - event

- From DOM to component
- Use it to listen for browser events, even custom events from 3. party libs
- Use it for output from component (More on that later)

```
<ul>
  <li><span (click)="count()">Clicked {{counter}} times</span></li>
  <li><input (keydown)="down()" (keyup)="up()"> {{isDown}}</li>
</ul>
```

Template – two way binding

- Goes in both direction
- Use it to update view model from template and template from view model

```
<h2>{{myTitle}}</h2>
<input [(ngModel)]="myTitle" autofocus>
```

Build-in structural directives

Structural directives changes the DOM

*ngIf

```
export class NgifComponent {
  public show = true;
  public toggle() {
    this.show = !this.show;
  }
}
```

```
<button (click)="toggle()">Toggle</button>
<p *ngIf="show">
  ngif works!
</p>
```

*ngFor

```
export class NgforComponent {
  public items:string[] = ['foo', 'bar'];
  addInput(input: string) {
    this.items.push(input);
  }
}
```

```
<div>
  <input #input (keyup.enter)="addInput(input.value)">
  <ul>
    <li *ngFor="let item of items">{{item}}</li>
  </ul>
</div>
```

LAB COMP-1

Create a sidebar component

Angular modules

define application modules with @NgModule

Angular modules

- **Angular Modules** help organize an application into cohesive blocks of functionality
- Every application has at least one module, known as the root module
- Angular libraries are modules (e.g, `FormsModule`, `HttpClientModule`, `RouterModule`)
- It's a class, decorated with `@NgModule` metadata.

Meta data

- Declare which components, directives and pipes belong together
- Can import other modules
- Can export declared components etc.
- Can re-export modules
- Can provide services

Example of shared module

```
import { NgModule, CommonModule } from '@angular/core';
import { HttpClientModule } from '@angular/common/http';
import { PrivateComponent, SharedComponent }  from './';
import { MyService }  from './';

@NgModule({
  imports:       [ CommonModule, HttpClientModule ],
  declarations: [ PrivateComponent, SharedComponent ],
  providers:     [ MyService ],
  exports:       [ SharedComponent, CommonModule,
                    HttpClientModule]
})
export class SharedModule { }
```

Meta data explained

- imports
  - An array of other modules this module depends on
- declarations
  - An array of components, directives & pipes available to this module
- exports
  - An array of components, directives and pipes to export or modules to re-export
- providers
  - An array of services to be available to this and all other modules! Wait what?

Feature module

- Help us partition the app into areas of specific interest and purpose
- Same `@NgModule` metadata
- Do not import `BrowserModule` in a feature module. Instead import `CommonModule`, or your `SharedModule`
- Is eagerly loaded when the application starts

Route modules

Using routes we can postpone the loading of modules until they are used!

Route modules lives in their own execution context, and therefore have their own injector (We will talk more about injectors later)

LAB MODU-1

Import a feature module

Input & output

Component interaction

Input & output scenarios

- Pass data from parent to child using `@Input`
  - Directly
  - Intercept it using getter and setter
- Parent listens for child event
- Parent interact with child via a template reference variable

Data from parent to child

Decorate a public property of the child component with `@Input()`

The property is now assignable from the parent

Directly – **child**

```
@Component({
  selector: 'app-directly-child',
  template: '<p>{{value1}}</p> <p>{{value2}}</p>'
})
class DirectlyChildComponent {
  @Input() public value1: string;
  @Input() public value2: string;
}
```

Directly – <u>parent</u>

```
@Component({
  selector: 'app-directly',
  templateUrl: 'directly.component.html`
})
export class DirectlyComponent {
  public fromProperty = 'From property';
}
```

```
<p>
  <app-directly-child
        value1="My value"
        [value2]="fromProperty">
  </app-directly-child>
</p>
<p>
  <input [(ngModel)]="fromProperty">
</p>
```

Intercept - **child**

```
@Component({
  selector: 'app-intercept-child',
  template: '<p>{{value}}</p>'
})
class InterceptChildComponent {
  private _value: string;

  @Input()
  public set value(value: string) {
    this._value = value.toUpperCase();
  }

  public get value(): string {
    return this._value;
  }
}
```

Data from child to parent

Decorate a public property of the child component with `@Output()`

This property can now emit events to the parent using the `EventEmitter` from the **@angular/core** library.

Parent ← **child**

```
@Component({
  selector: 'app-io-event-child',
  template: `
    <p>
      <input type="radio" name="newsletter" (click)="update('weekly')"> Weekly
      <input type="radio" name="newsletter" (click)="update('monthly')"> Monthly
    </p>
  `
})
class IOEventChildComponent {
  @Output() public interval = new EventEmitter<string>();
  public update(value: string) {
    this.interval.emit(value);
  }
}
```

Parent ← child

```
@Component({
  selector: 'app-io-event',
  templateUrl: 'event.component.html`
})
export class IOEventComponent {
  public newsletterInterval: string;
}
```

```
<p>
  <app-io-event-child
        (interval)="newsletterInterval = $event">
  </app-io-event-child>
</p>
<p>
  Chosen newsletter {{newsletterInterval}}
</p>
```

Parent interact using template reference variable

- Directly access to the children methods and properties
- Only accessible from the template
- Created using a template reference variable e.g. `#stopwatch`

Parent interact using template reference variable

```
<p>
  <app-stopwatch #stopwatch></app-stopwatch>
</p>
<p>
  <button (click)="stopwatch.start()">Start</button>
  <button (click)="stopwatch.stop()">Stop</button>
  <button (click)="stopwatch.reset()">Reset</button>
</p>
```

Parent interact using template reference variable

```
@Component({
  selector: 'app-stopwatch',
  template: '{{seconds}}',
})
export class StopwatchComponent {
  public seconds: number = 0;
  private intervalNo: number;
  public start() {
    this.intervalNo = window.setInterval(() => this.seconds += 1, 1000);
  }
  public stop() {
    window.clearInterval(this.intervalNo);
  }
  public reset() {
    this.stop();
    this.seconds = 0;
  }
}
```

Smart vs. presentation

- Make the features “main” component smart
  - It knows how to fetch data
  - It knows what services this features depends on
- Make “internal” components presentation components
  - They know nothing of context
  - They get their data from the smart parent
  - They communicate using EventEmitter

Smart vs. presentation

```
<app-contact-book>
  <app-contacts
    [contacts]="contacts”
    (select)=“selectContact($event)">
  </app-contacts>
</app-contact-book>

<app-department>
  <app-contacts
    [contacts]=”employees”>
  </app-contacts>
</app-department>
```

LAB IO-1

Create a footer

And a bunch more

Services

injectors and providers

Services

Angular documentation says -

- *Service* is a broad category encompassing any value, function, or feature that our application needs.
- Almost anything can be a service. A service is typically a class with a narrow, well-defined purpose. It should do something specific and do it well

Services

- Services should contain the business logic – not the components
- Services are front end services. I.e. they are not shared with others
- A service is typically a TypeScript class, but can just as well be a function or value
- A service must be registered as a *provider*

Services

Plain class – no Angular interference

Well almost…

```
@Injectable()
export class RandomService {

  private _random: number;

  constructor() {
    setInterval(() => this._random = Math.floor(Math.random() * 100), 1000);
  }

  public get random(): number {
    return this._random;
  }
}
```

Using a service

```
@Component({
  selector: 'app-non-provided',
  template: 'Will never work'
})
export class NonProvidedComponent {

  public constructor(private service: NonProvidedService) {

  }
}
```

But this fails!

Why? Dependency injection (DI), injectors and providers!

Injectors & providers

- We must register a service *provider* with the injector, or Angular won't know how to create the service.
- Uses injectors to inject the service
  - This is done by the Angular dependency framework
- Injectors has levels
  - One application wide
  - Zero to many router modules injectors

Injectors & providers

- ~~Application wide providers must be registered in eagerly loaded modules~~ (Changed with Angular 6)
- ~~All modules loaded lazily, works with its own injector~~ (Changed with Angular 6)
- Dependencies are singletons within the scope of an injector

Providers & injectors

Register the RandomService – Pre Angular 6

```
@NgModule({
    imports: [CommonModule],
    providers: [RandomService]
})
```

Register the RandomService – With Angular 6

```
@Injectable({
  providedIn: 'root'
})
export class RandomService {
  …
}
```

Inject the RandomService

```
@Component({…})
export class ProvidedComponent {
  constructor(public service: RandomService) {
}
}
```

Configuring an injector

The convenient way

```
providers: [RandomService]
```

What it expands to

```
providers: [{
        provide: RandomService,
        useClass: RandomService
}]
```

New class but same provider token

```
providers: [{
        provide: RandomService,
        useClass: CryptographicallySecureRandomService
}]
```

Configuring an injector

Use a value

```
providers: [ {
        provide: RandomService,
        useValue: {…}
}]
```

Use a factory

```
providers: [{
    provide: RandomService,
    useFactory: randomServiceFactory
    deps: []
}]
```

`@Injectable()`

- Use `@Injectable()` to enabling injecting of a service into another service
- Use `@Injectable()` to provide a service by using `providedIn`
- The service to be injected still needs to be registered as a provider in an injector, one way or another

@Injectable()

```
@Injectable()
export class RandomLoggerService {

  private _random: number;

  constructor(logger: LoggerService) {
    setInterval(() => {
      this._random = Math.floor(Math.random() * 100);
      logger.log('Generated new random number');
    }, 1000);
  }
  …
}
```

```
@NgModule({
  imports: [CommonModule],
  providers: [
    LoggerService,
    {
      provide: RandomService,
      useClass: RandomLoggerService
    }
  ]
})
```

LAB SERV-1

Create a playground service

Ajax & RxJS

talking to a remote server & reactive extension

Ajax & RxJS

Using Ajax (Asynchronous JavaScript and XML) is the first time we use an asynchronously function in our project

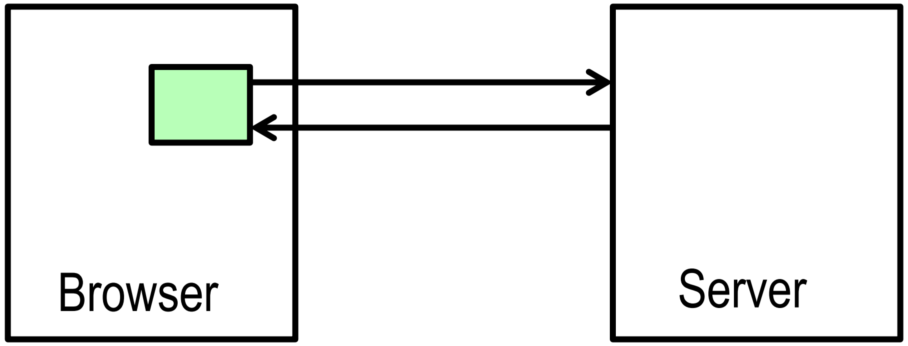
Angular uses RxJS, Reactive Extensions, as default, to work with async operation

RxJS might seem a bit strange to begin with.

But let’s start with Ajax

What is ajax?

- Ajax (or AJAX) is short for Asynchronous JavaScript And XML, but maybe it should be AJAJ for Asynchronous JavaScript And JSON
- Ajax enables an HTML page to communicate with a server in the background.
- Ajax can be used to exchange data and then update parts of the page instead of the whole page.



Angular & Ajax

- Ajax is supported by the provided module `HttpClientModule` located in `@angular/common/http` library

```
import { HttpClientModule } from '@angular/http';
@NgModule({
    declarations: [...],
    imports: [BrowserModule, HttpClientModule],
    providers: [...],
    bootstrap: [AppComponent],
})
export class AppModule { }
```

- Supports XHR and JSONP
  - We will only talk about XHR

Simple example

Service

```
@Injectable()
export class F1SimpleService {
  constructor(private http: HttpClient) { }

  public getDrivers():Observable<any> {
    return this.http.get<any>(`http://ergast.com/api/f1/2018/drivers.json`);
  }
}
```

Component

```
@Component({
  selector: 'app-simple-http-service’,
  templateUrl: './simple-http-service.component.html',
})
export class SimpleHttpServiceComponent implements OnInit {

  public drivers: Driver[];

  public constructor(private service: F1SimpleService) { }

  public ngOnInit() {
    this.service.getDrivers().subscribe(response => this.drivers = response.MRData.DriverTable.Drivers);
  }
}
```

API

```
delete(url: string, options: Options): Observable<any>
get(url: string, options: Options): Observable<any>
head(url: string, options: Options): Observable<any>
options(url: string, options: Options): Observable<any>
patch(url: string, body: any|null, options: Options): Observable<any>
post(url: string, body: any|null, options: Options): Observable<any>
put(url: string, body: any|null, options: Options): Observable<any>
```

```
options: {
  headers?: HttpHeaders | {[header: string]: string | string[]},
  observe?: HttpObserve,
  params?: HttpParams|{[param: string]: string | string[]},
  reportProgress?: boolean,
  responseType?: 'arraybuffer'|'blob'|'json'|'text',
  withCredentials?: boolean,
} = {}
```

Getting a response from Http

- We need to subscribe to the Observable
  - Kind alike `.then` from AngularJS with promises

```
service.getDrivers().subscribe(response =>
    this.drivers = response.MRData.DriverTable.Drivers;
);
```

The problem

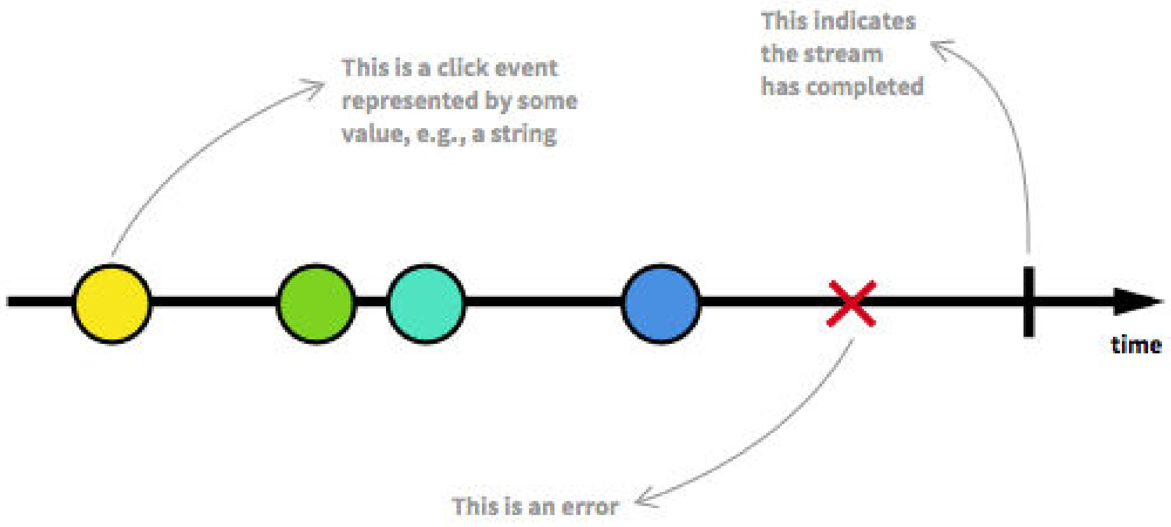
But what about - *A service is typically a class with a narrow, well-defined purpose. It should do something specific and do it well*

In our service we don't expose any types (`Observable<any>`) and we leave error handling, caching etc. to the client.

The solution

What is RxJS

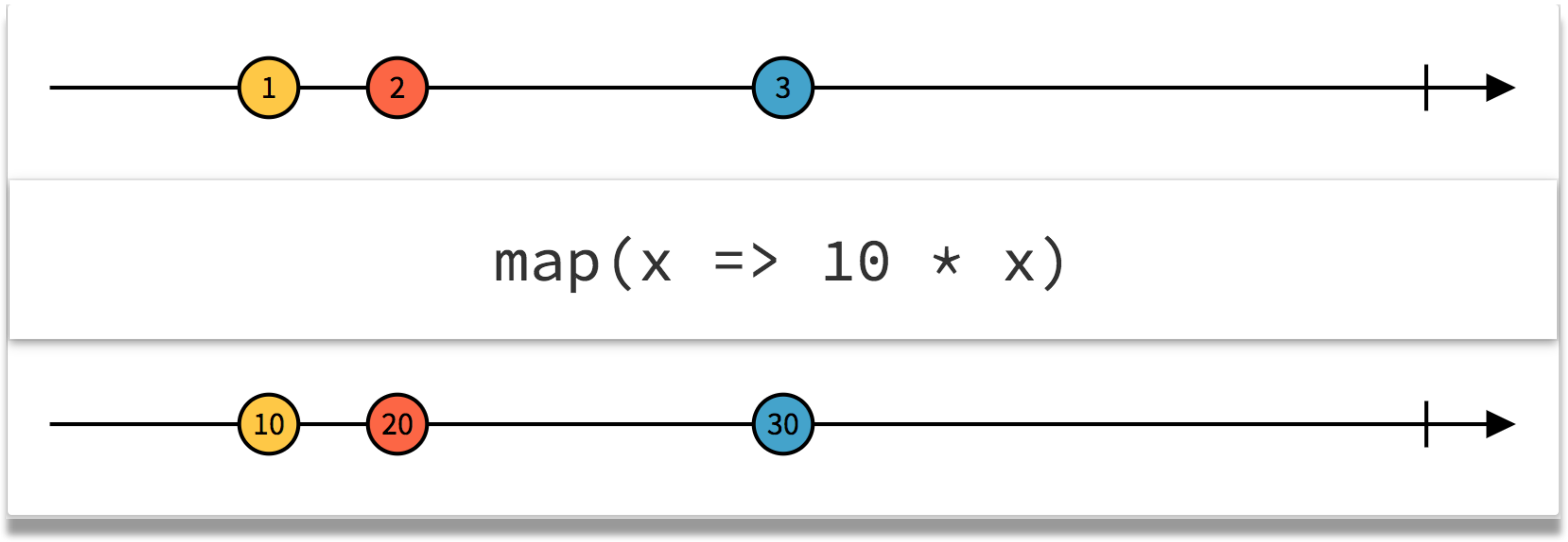
- RxJS (Reactive Extensions) is all about reactive programming
- Reactive programming is basically about data streams and events


This is a click event represented by some value, e.g., a string
This indicates the stream has completed
time
This is an error

RxJS - Observable

- Kinda alike a promise
  - With multiple resolve
- Can be observed using `subscribe(…)`
- Use composition change behavior of an observable
- Immutable

RxJS composition - map



Maps input from one stream into another streams output

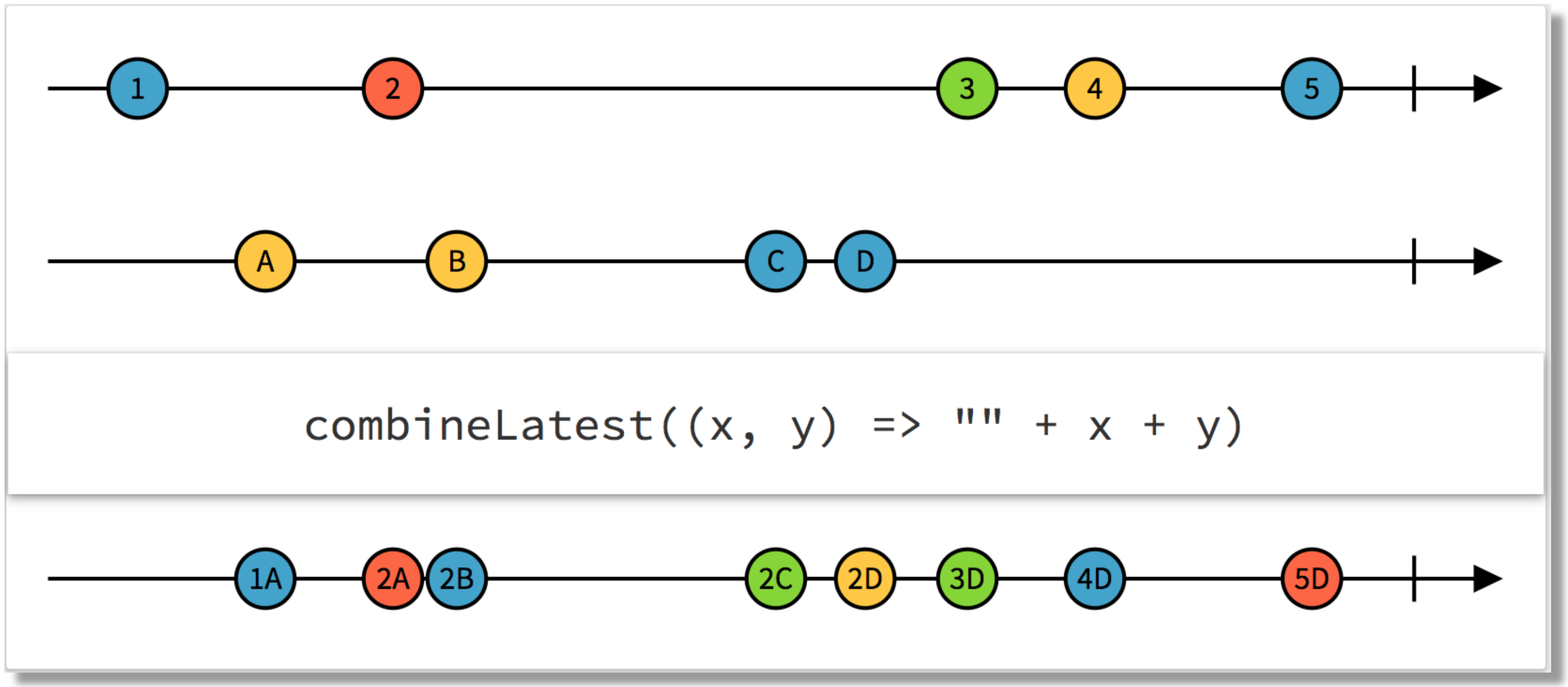
Lets improve the F1Service with this knowledge!

RxJS composition – multicast/sharing

- `publish, publishReplay, publishLast`
  - Turns unicast stream into multicast/shared
- `refCount`
  - Keep the connection alive by tracking reference counting

Lets improve the F1Service with this knowledge!

RxJS composition - combineLatest

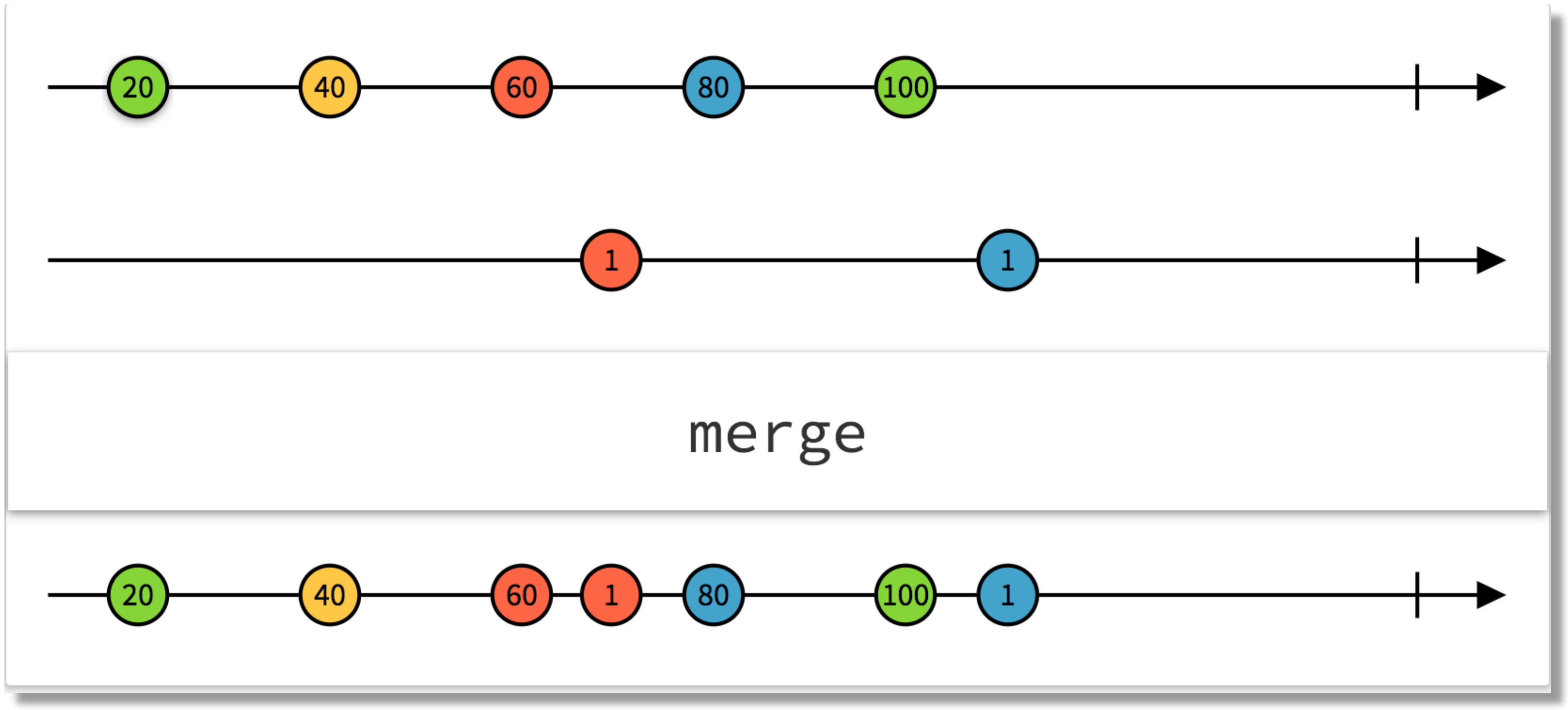


```
combineLatest((x, y) => "" + x + y)
```

Combine events from two source into pairs

Imagine you need to combine the result from two or more Ajax calls

RxJS composition - merge



Merge events from two or more observables into one

RxJS composition – switchMap

- switchMap is an higher order observable
  - A higher order observable is just a fancy name for an observable that emits observable.
- Is a flatterning operator
  - If a map operator returns an observable use switchMap

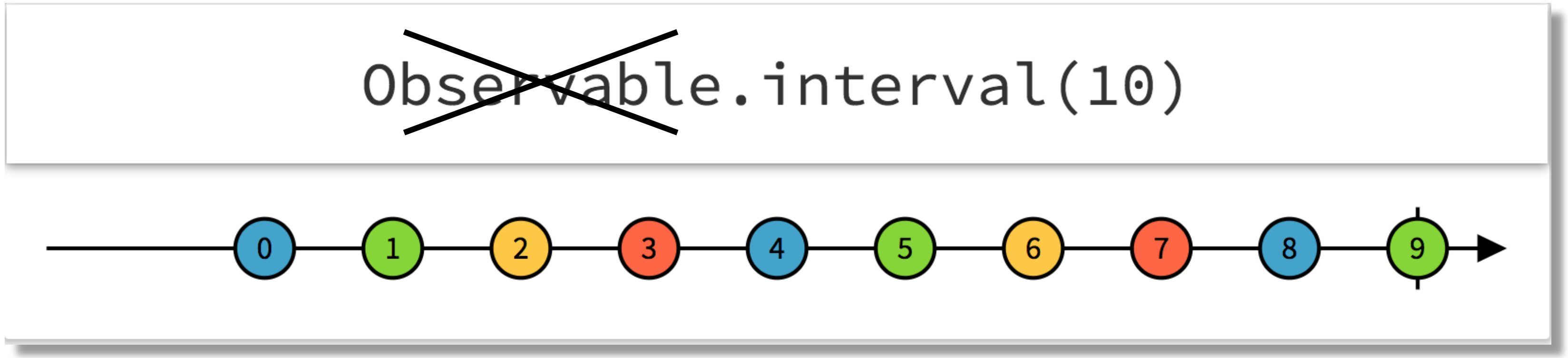
RxJS composition

- And many many many more (143+)
- See more here: http://reactivex.io/rxjs/
- And here http://rxmarbles.com/

These RxJS methods are really important to know in Angular!

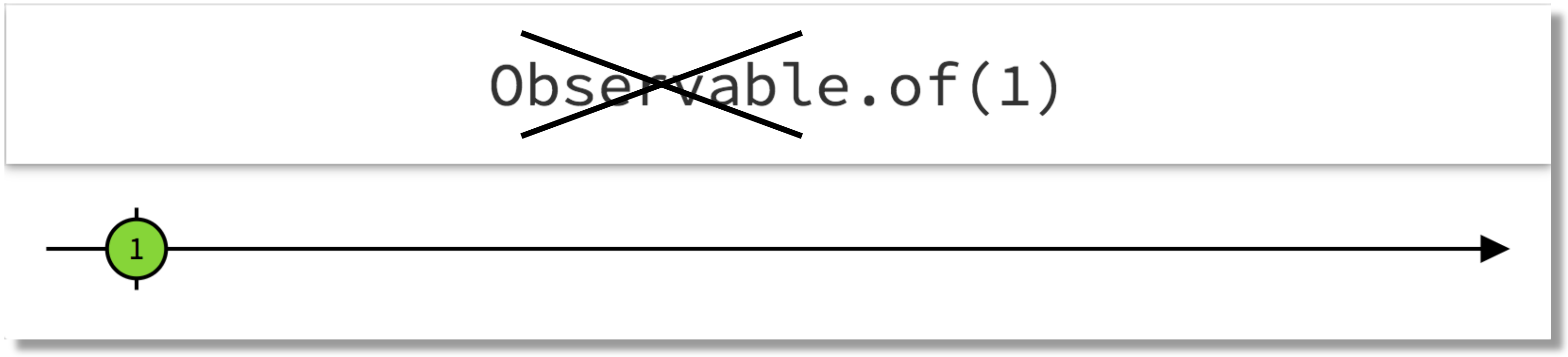
So many things in Angular works with observables, so better get to know them!

RxJS creation - interval



Returns an observable sequence that produces a number value after each period. It will keep on going until stopped.

RxJS creation - of



Converts arguments to an observable sequence

RxJS creation - create

`Observable.create((observer => {…})`

- Creates an observable sequence from a specified observer method implementation
- Good for wrapping existing APIs
- Rarely used by you

And many many more (43+)

Wrapping an existing API

Using the `Observable.create` wrap an existing API (Not really necessary since we have Observable.interval and map)

```
import { Observable } from 'rxjs';
@Injectable()
export class TimerService {
  private timerStream: Observable<Date>;
  constructor() {
    this.timerStream = Observable.create
      (observer => {
        const intervalId = window.setInterval(() =>
                 observer.next(new Date()), 1000);
        return () => {
          console.log('Stopping timer!');
          window.clearInterval(intervalId);
        }
      });
  }
  public get timer(): Observable<Date> {
    return this.timerStream;
  }
}
```

RxJS error handling

- catchError
  - Classic catch. Must return an observable
- throwError
  - To signal an error form within a observable
- retry or retryWhen
  - Retries an operation a certain number of times or after a certain period or something else.
- finalize
  - Classic finally to do some clean up, after the stream has **completed**

RxJS error handling

```
export class ErrorComponent implements OnInit {

  public zipCodes$: Observable<any>
  public elapsed: number;

  constructor(private http: HttpClient) { }

  public ngOnInit() {
    const start: number = Date.now();
    this.zipCodes$ = this.http.get<any>('http://404.com/asd.json’).pipe(
      catchError(() => this.http.get<any> ('http://404.com/another.404’)),
      catchError (() => this.http.get<any> ('https://dawa.aws.dk/postnumre’)),
      catchError (() => throwError(’No data!!’)),
      finalize(() => this.elapsed = Date.now() - start),
    )
  }
}
```

LAB AJAX-1

Add Ajax to the playground service

LAB RXJS-1

Provide the location service

LAB RXJS-2

Include the map

Route & navigation

screen navigation with the Angular Component Router

Routes & navigation

- The browsers navigation model is familiar one – respect it!
- Enter URL or click a deep link and the browser load the requested URL
- Press back and forward button to navigate through history of visited pages.

Routes & navigation

Routing is not part of the core package!

Its part of

`@angular/router`

One route

- A route describes a the relationship between the URL and a component

E.g

```
https://example.com/order
```

```
{
  path: 'order',
  component: OrdersComponent
},
```

Several routes

app-routing.module.ts

```
const routes: Routes = [
    {
        path: 'order',
        component: OrdersComponent
        children: [
                path: ':id',
                component: OrderComponent
        ]
    },
    {
        path: ‘invoice',
        component: InvoicesComponent
    },
    {
        path: '**',
        redirectTo: '/404'
    }
]
```

URL matching

- From top to bottom
- Stops at first match
- Only redirects once

Creating the router module

- We do not import RouterModule in our module, we create our own module

One of these

```
export const AppRoutingModule = RouterModule.forRoot(routes);
```

and maybe several of these

```
export const FeatureRoutingModule = RouterModule.forChild(routes);
```

Import the configured router module

- Import router module into application module

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

- Feature modules can, and should, have routing as well. Use `RouteModule.forChild()`

Navigating

- Navigate programmatically by injecting the `Router`
- Navigate using the `routerLink` attribute directive in the template

Navigating

Navigate programmatically using the router service

```
export class OrdersComponent {
  constructor(private router: Router) { }
  public gotoOrder(orderId: number) {
    this.router.navigate(['/order', orderId]);
  }
}
```

Navigate using routerLink directive

```
<a [routerLink]="['/order', orderId]">Order {{orderId}}</a>
```

Link parameter array

- Its purpose is to add parameters to a URL
- Used with `routerLink` & `router.navigate`
- Primitives will be added as part of the path
- Objects will be added as matrix parameters

Generating URLs

```
<a routerLink="/order">Orders</a>

<a [routerLink]="['/order', order.id]">Order</a>

<a [routerLink]="['/order', {rows: 50, offset: 0, query: ‘Tårn’}]">Search</a>

<a [routerLink]="['/order', {rows: 50, offset: 0, query: ‘Tårn’}, order.id]">
  Search
</a>
```

E.g.

```
1. /order
2. /order/1
3. /order;rows=50;offset=0;query=Tårn
4. /order;rows=50;offset=0;query=Tårn/17
```

Reading parameters

- Inject `ActivatedRoute` into component to read parameters
- Read via `params: Observable<Params>`
- Or via `snapshot.params: Params`

Reading parameters

Read parameters using snapshot

```
export class OrderComponent implements OnInit, OnDestroy {
  public orderNo: number;
  constructor(private route: ActivatedRoute) { }
  ngOnInit() {
    this.orderNo = this.route.snapshot.params['id'];
  }
}
```

Read parameters continuously

```
export class OrderComponent implements OnInit {
  public orderNo: number;
  constructor(private route: ActivatedRoute) { }
  ngOnInit() {
    this.route.params
         .subscribe(params => this.orderNo = params['id']);
  }
}
```

Inserting the component

Use <router-outlet> to insert the component identified by the route

E.g
`app.component.html`

```
<nav>
        ...
</nav>
<div>
        <router-outlet></router-outlet>
</div>
```

Inserting the component

Each route that has children, also have a <router-outlet>

E.g

```
<nav>
   ...
</nav>
<div>
   <router-outlet></router-outlet>
   <app-orders>
      <ul>
         ...
      </ul>
      <router-outlet></router-outlet>
      <app-order></app-order>
   </app-orders>
</div>
```

Lazy loading

- Loads the feature module when needed – saves bandwidth

```
{
   path: 'order',
   loadChildren: 'app/order/order.module#OrderModule'
}
```

LAB ROUT-1

Add some routes to the playgrounds!

Forms

two-way data binding, change tracking, validation, and error handling

Forms

- Uses binding for data gathering
- Uses validation, some HTML5 and custom
- Adds CSS classes for easy styling
  - Remember does not include any CSS
- Different ways to handle forms
  - Template driven (FormsModule)
  - Model driven / Reactive (ReactiveFormsModule)

Template driven

- Alot like forms in AngularJS
- Create forms almost entirely in the template
- Import the FormsModule for it to work
- Core directives
  - ngForm
  - ngModel
  - ngModelGroup

ngForm directive

Selector: **form:not([ngNoForm]):not([formGroup])**

```
<form>
   ...
</form>
```

Supercharges the form with properties such as valid, invalid & value
Use template reference variable to reference the form in the template

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

ngForm directive

- Other properties
  - `value` contains the form controls current value. Valid or not.
  - `form` Lets you access
    - invalid / valid
    - dirty / pristine
    - touched / untouched

ngModel

- Add ngModel to include the HTML control in the ngForm.value
- ngModel work in three different modes
  - `[(ngModel)]` two way binding to existing domain model
  - `[ngModel]` one way binding to existing domain model
  - `ngModel` Uses the name attribute from the HTML control as key in ngForm.value
- Use ngModel for validation using template reference variable

ngModel

Use two way binding for continuous synchronization between template and domain model

```
<input type="text" class="form-control"
        required [(ngModel)]="model.firstName" name="firstName">
```

Use one way binding for initialization of the template

```
<input type="text" class="form-control"
        required [ngModel]="model.firstName" name="firstName">
```

Use without expression when no prior domain model exists. The value of the fields are still accessible through ngForm.value

```
<input type="text" class="form-control"
        required ngModel name="firstName">
```

Template reference & error message

Use template refence variable to show and hide an error message

```
<div>
  <label for="firstName">First name</label>
  <input id="firstName" required
         [(ngModel)]="model.firstName"
         name="firstName" #firstName="ngModel">

  <div [hidden]="firstName.valid || firstName.pristine"
         class="alert alert-danger">
    First name is required
  </div>

</div>
```

ngModelGroup

- Adds structure to the `ngForm.value` element
- Enables validation of a related groups of HTML controls

ngModelGroup

```
<fieldset ngModelGroup="name">
    <div>
        <label for="firstName">First name</label>
        <input id="firstName" required minlength="2" ngModel
                  name="firstName">
    </div>
    <div>
        <label for="lastName">Last name</label>
        <input id="lastName‶ required ngModel name="lastName">
    </div>
</fieldset>
```

ngForm.value with ngModelGroup	ngForm.value without ngModelGroup
`{ name: { firstName: 'Flemming', lastName: 'Bregnvig' } }`	`{ firstName: 'Flemming', lastName: 'Bregnvig' }`

Automatic classes

Angular automatically adds classes to the HTML controls, but remember you will need to define the styles yourself

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

Model driven forms

- Actual properties in our components, which makes them easier to test.
- Replicate the form model from the DOM structure in our component
- Angular doesn't magically create the templates for us
- Model driven forms are more like an addition to template-driven forms

Model driven forms

In the controller

```
public fg: FormGroup;
public ngOnInit() {
     this.fg = new FormGroup(
       {
         name: new FormGroup({
           firstName: new FormControl(this.model.firstName, Validators.required),
           lastName: new FormControl(this.model.lastName, Validators.required)
         }),
         favoriteColor: new FormControl(),
         height: new FormControl()
       }
     );
}
```

Model driven forms

In the controller

```
public fg: FormGroup;

constructor(private fb: FormBuilder) { }

ngOnInit() {
  this.fg = this.fb.group({
    name: this.fb.group({
      firstName: [this.model.firstName, Validators.required],
      lastName: [this.model.lastName, Validators.required]
    }),
    favoriteColor: [],
    height: []
  })
}
```

Model driven forms

The template

```
<form [formGroup]="fg" (ngSubmit)="onSubmit()">
    <fieldset formGroupName="name">
        <div>
            <label for="firstName">First name</label>
            <input formControlName="firstName">
        </div>
        <div>
            <label for="lastName">Last name</label>
            <input formControlName="lastName">
        </div>
    </fieldset>
    <div>
        <label for="color">Favorite color</label>
        <select formControlName="favoriteColor">
            <option *ngFor="let color of colors" [value]="color">{{color}}</option>
        </select>
    </div>
    <div>
        <label for="height">Height</label>
        <input type="number" min=100 max=220 formControlName="height">
    </div>
    <button type="submit" [disabled]="fg.invalid">Submit</button>
</form>
```

Model driven form

- The template in model driven resembles that of a template driven
- No more ngModel
- Validators are no longer added in the template, but in the component
- Avoid using both template driven and model driven together

Which should I choose

Template driven will appeal to AngularJS developers

Use model driven for any form that might grew beyond a simple form

Use model driven for ease of unit testing

Reactive

- But why is it called `ReactiveFormsModule`?
- Because!
  - `valueChanges: Observable<any>`
  - `statusChanges: Observable<any>`
- Use these properties to be notified whenever a group or control value and/or status changes

Stand-alone reactive example

Search form

Notice that for stand-alone we use `formControl` – not `formControlName`

```
<input [formControl]="searchControl">
```

Search component

```
  ngOnInit() {
    this.searchControl = new FormControl();
    this.searchControl.valueChanges.pipe(
      debounceTime(200),
      distinctUntilChanged(),
    ).subscribe(param => console.log('Do something with this', param))
  }
```

Unit testing

techniques and practices for unit testing an Angular 2 app

Unit testing

- Angular CLI depends on Jasmine for unit tests and karma for test running
- Jasmine is a unit testing framework for JavaScript that can run in a browser. Headless or not
- Jasmine is widely used to unit test JavaScript frameworks
- Karma is a test runner that simplifies configuration and setup of Jasmine, browser etc.

Unit test and Angular

- The files must be called *.spec.ts to be picked up by Karma
  - E.g. 1st.spec.ts
- Isolated without Angular interference for pipes and services
- Use non-isolated for component testing

Jasmine

```
describe('1st tests', () => {
  it('true is true', () => {
    expect(true).toBe(true));
  }
});
```

This test is an isolated unit test. No dependencies on Angular

Jasmine

- `describe()` is a function that creates a test suite.
- `it()` is a function that creates a single test within a test suite
- `expect()` is a function that creates a matcher which can make assertions about operations executed inside a single test.
- `beforeEach()` is a function that is executed before each test in the suite.
- `afterEach()` is a function that is executed after each test in the suite.

Expect what?

- `toEqual`
- `toBe`
- `toBeNull`
- `toBeTruthy`
- `toBeFalsy`
- `toBeUndefined`
- `toContain`
- `toMatch`

- `toThrow`
- `toThrowError`
- `toBeLessThan`
- `toBeGreaterThan`
- `toBeCloseTo`
- `toHaveBeenCalled`
- `toHaveBeenCalledWith`

And what `.not` to expect

Jasmine example

Lets go to the examples!