Angular Master Class

Jump Start Tutorial

Introduction

Angular since 2.x

Angular since 2.x

- The next major version of the Angular framework

Angular since 2.x

- The next major version of the Angular framework
- Rather a development platform than just a framework

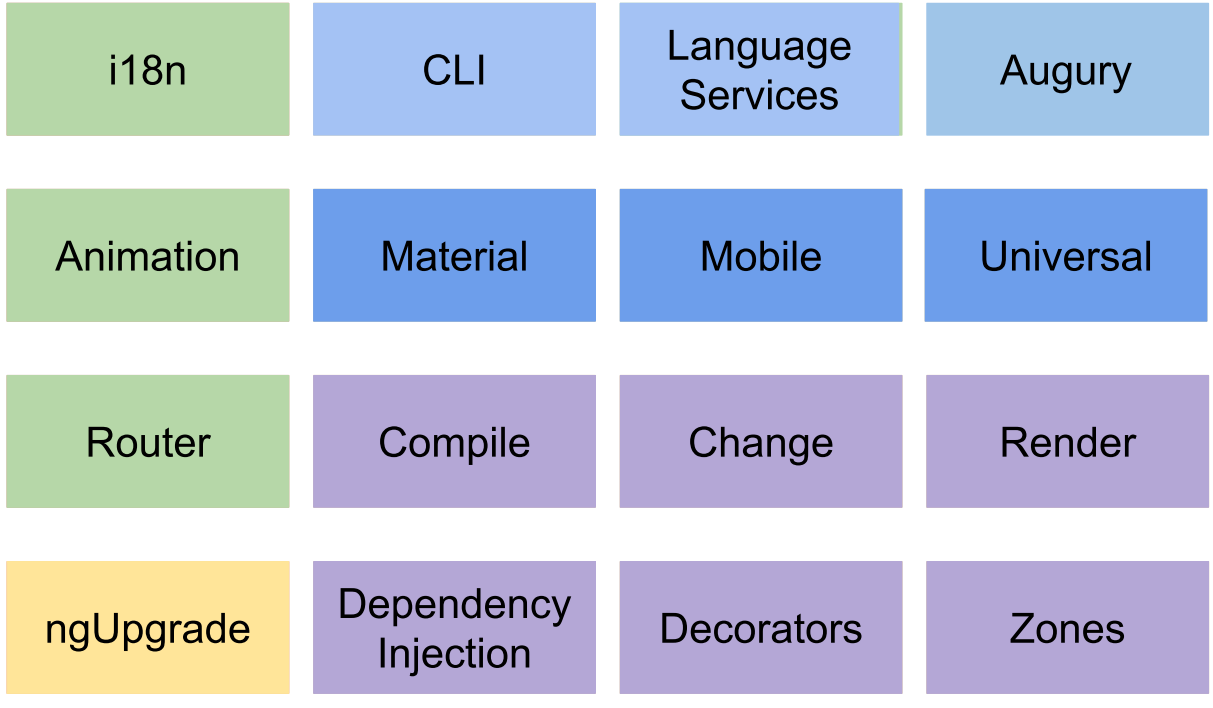
Angular since 2.x

- The next major version of the Angular framework
- Rather a development platform than just a framework
- Mobile First, future ready, powered by open source community

Angular since 2.x

- The next major version of the Angular framework
- Rather a development platform than just a framework
- Mobile First, future ready, powered by open source community
- Faster, better tooling, more flexible, ...

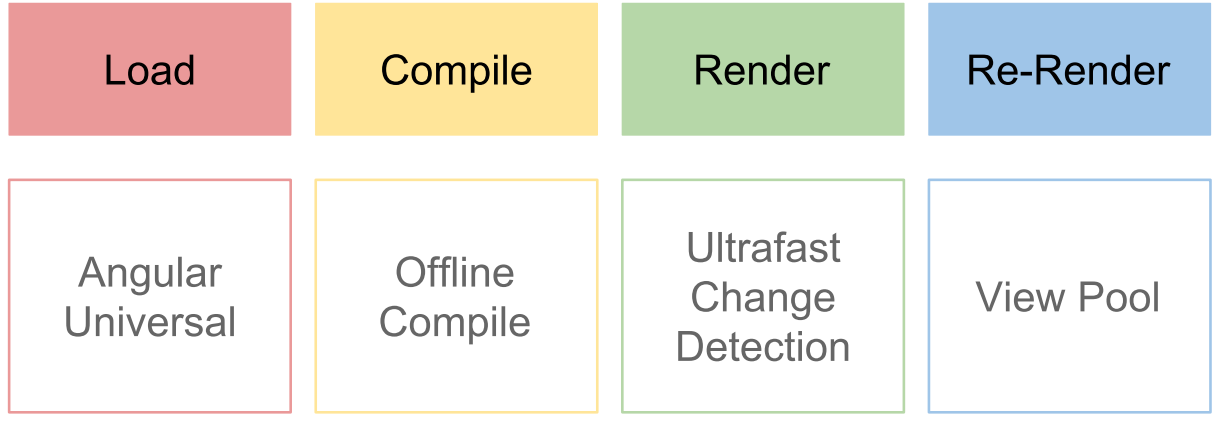
Framework to Platform



Performance

Performance

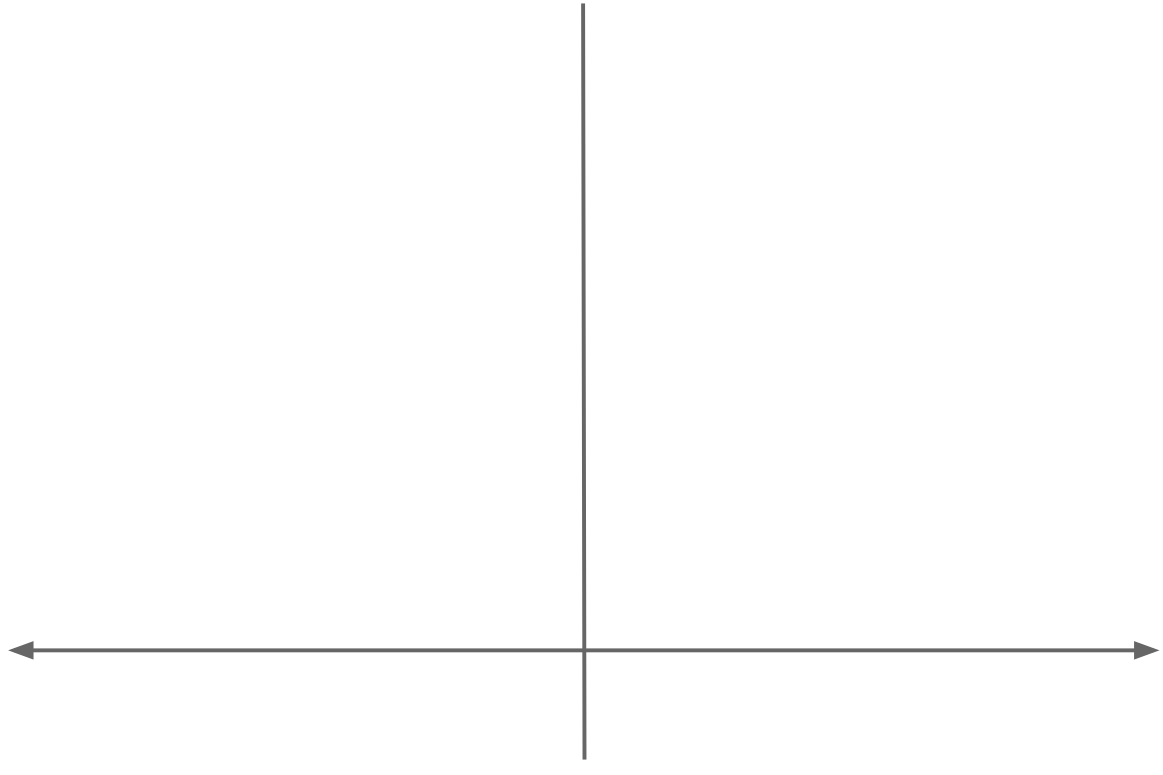
Performance



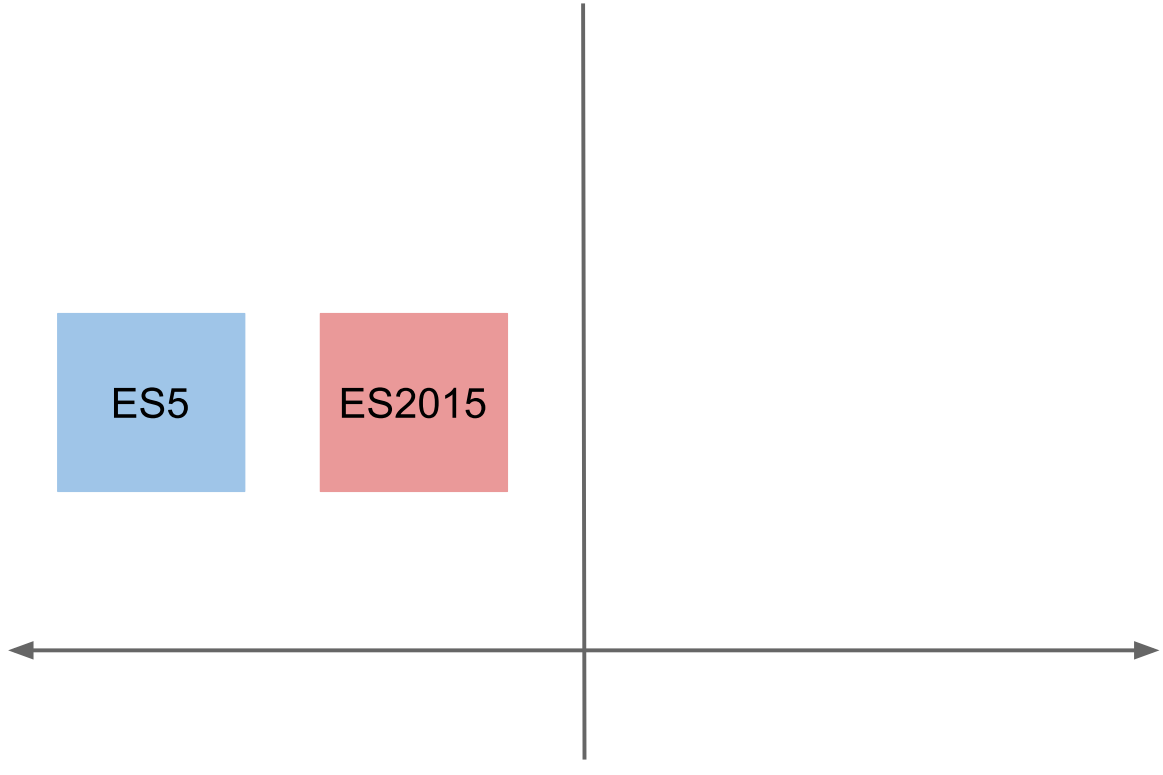
Performance

Load	Compile	Render	Re-Render
Angular Universal	Offline Compile	Ultrafast Change Detection	View Pool
Instant Rendering	3x faster vs ng2	Always 5x faster than ng1	Always 5x faster than ng1

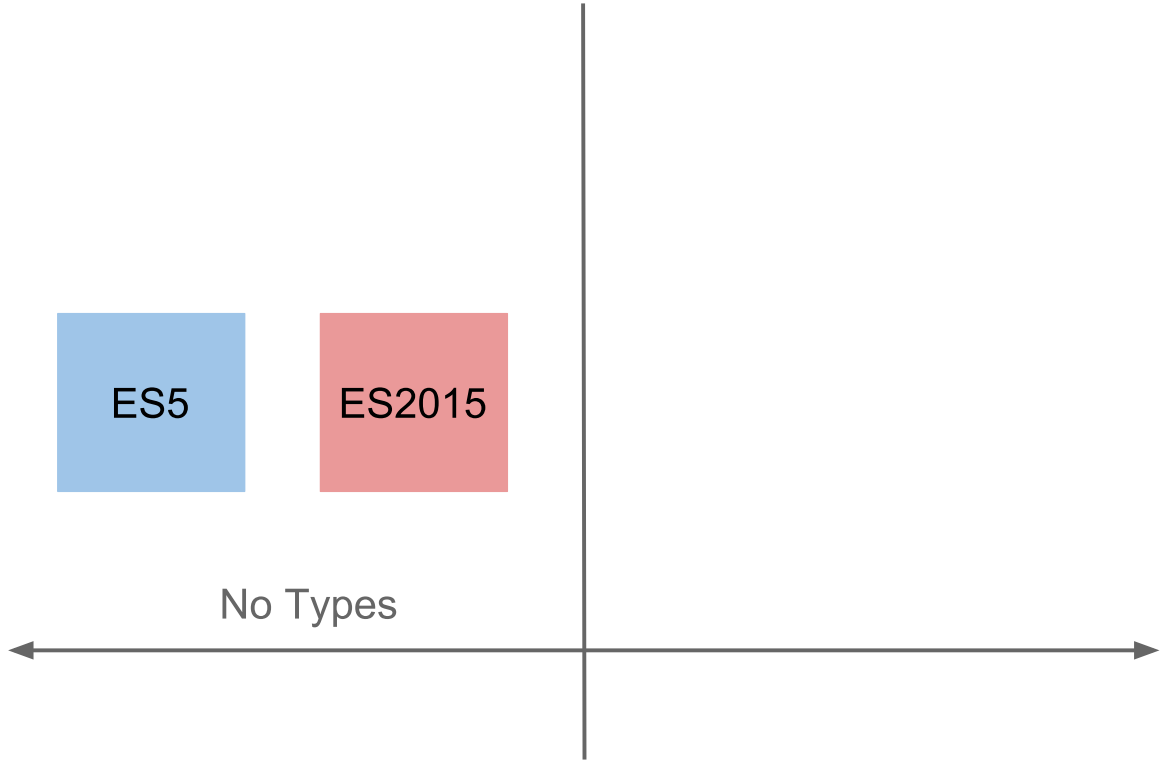
Languages



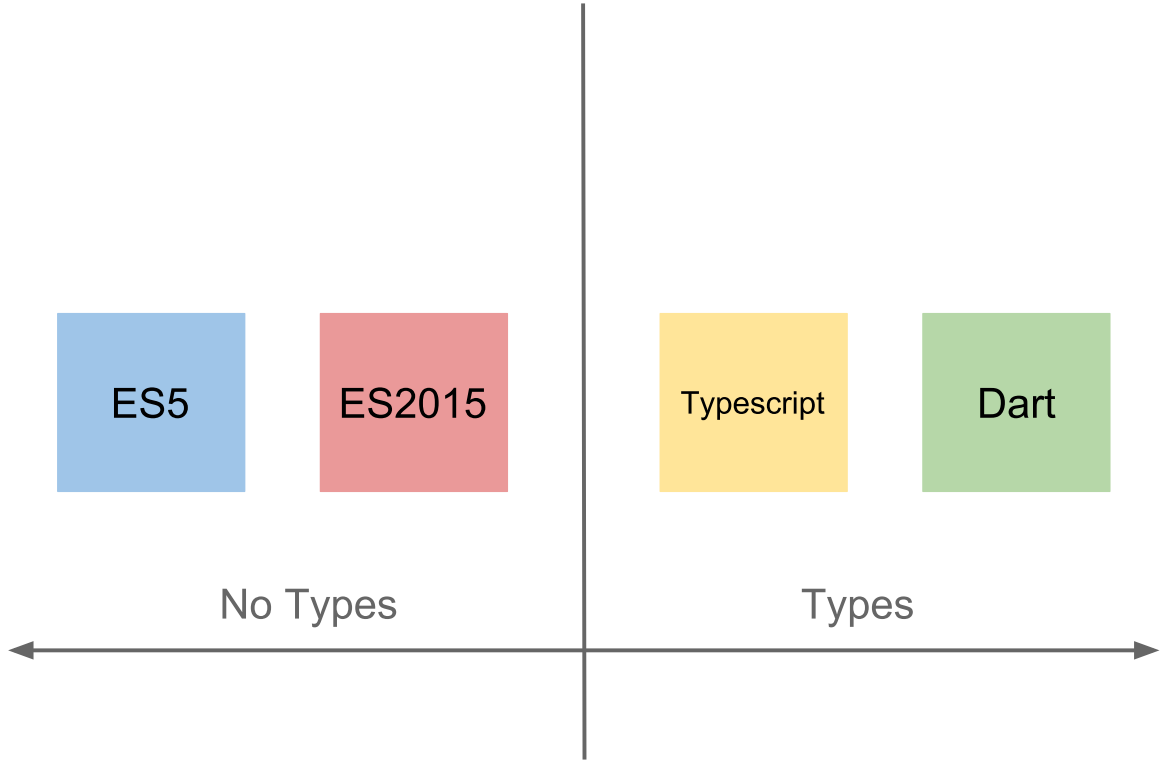
Languages



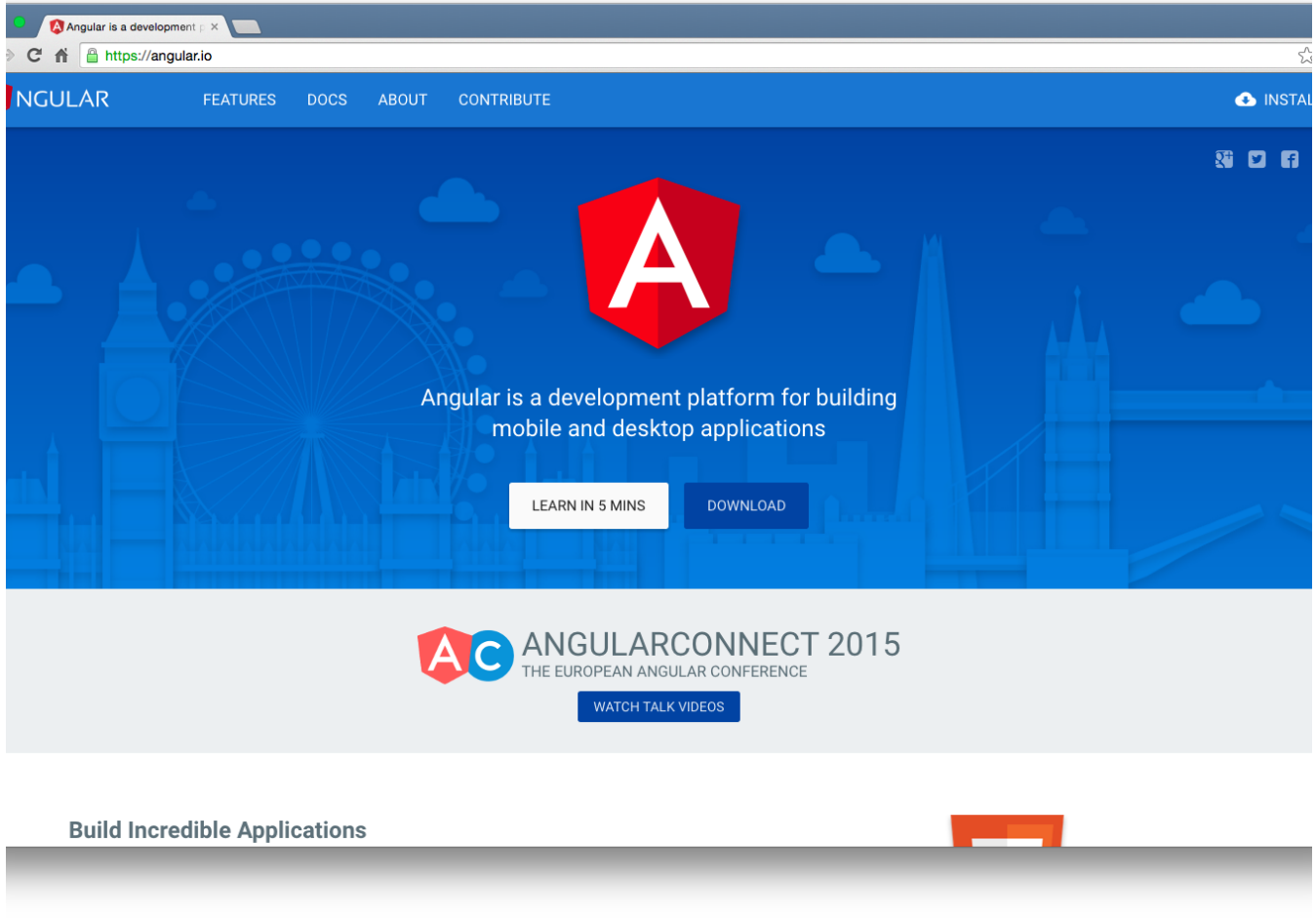
Languages



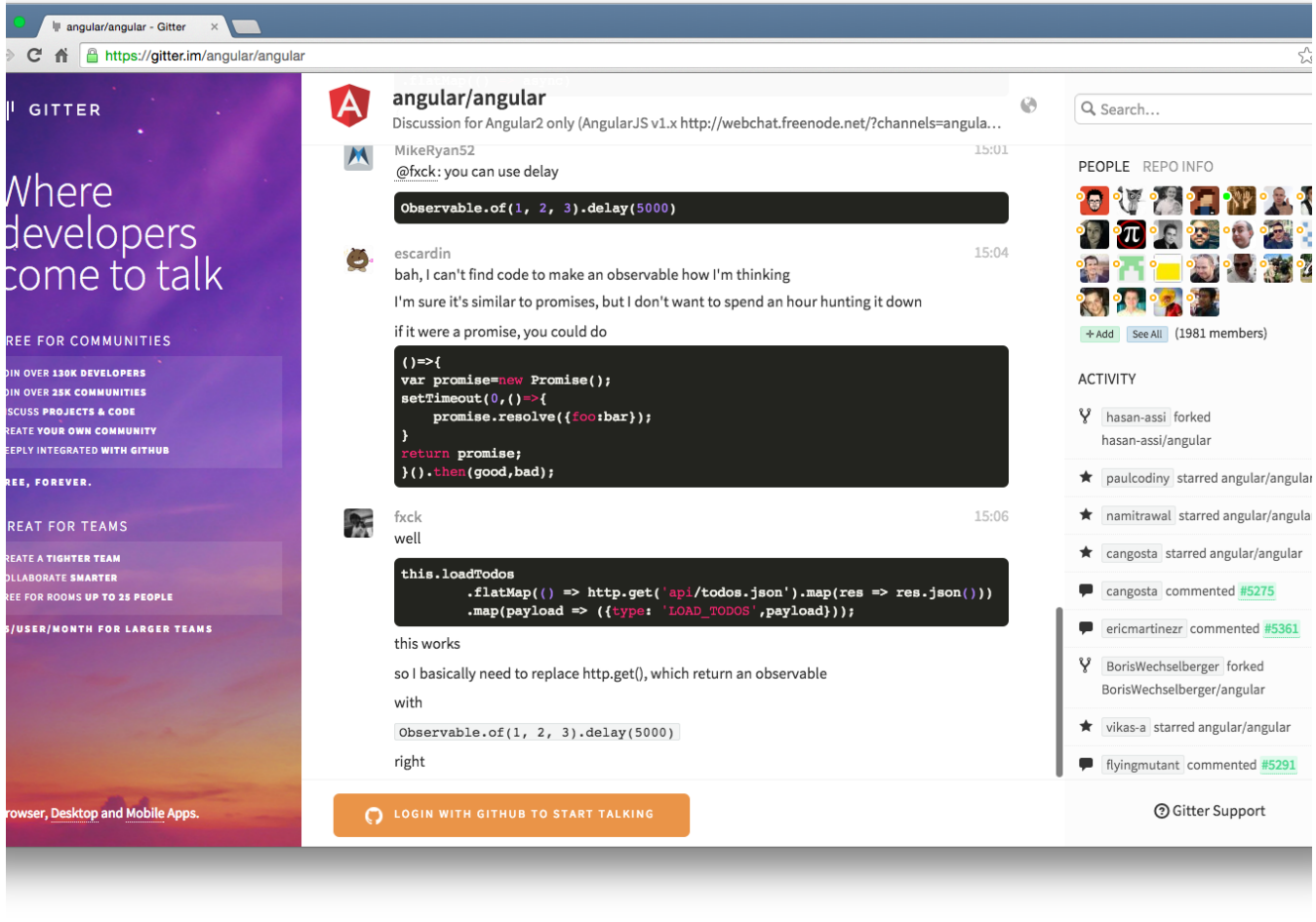
Languages



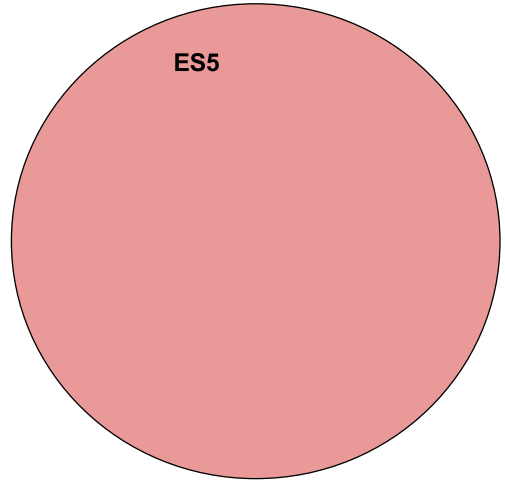
Useful Resources

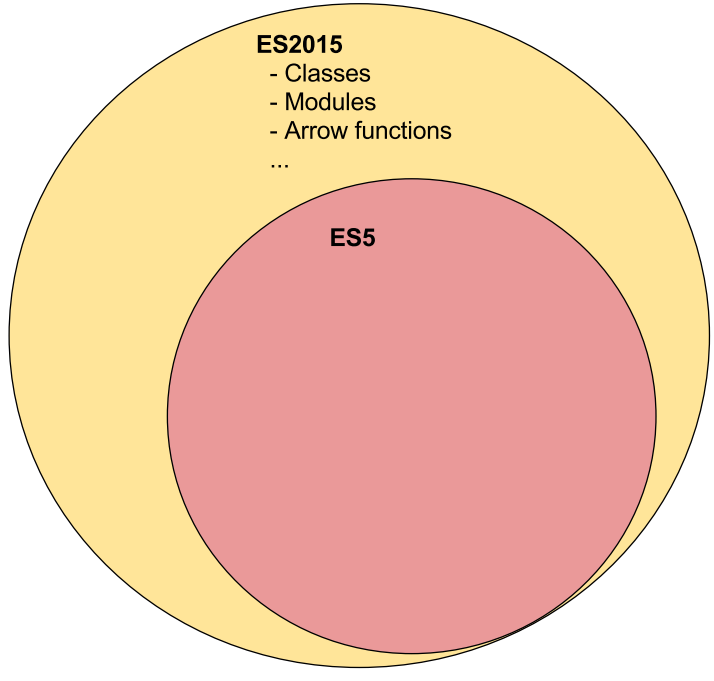

Angular is a development
https://angular.io
NGULAR
FEATURES
DOCS
ABOUT
CONTRIBUTE
INSTAL
Angular is a development platform for building
mobile and desktop applications
LEARN IN 5 MINS
DOWNLOAD
ANGULARCONNECT 2015
THE EUROPEAN ANGULAR CONFERENCE
WATCH TALK VIDEOS
Build Incredible Applications

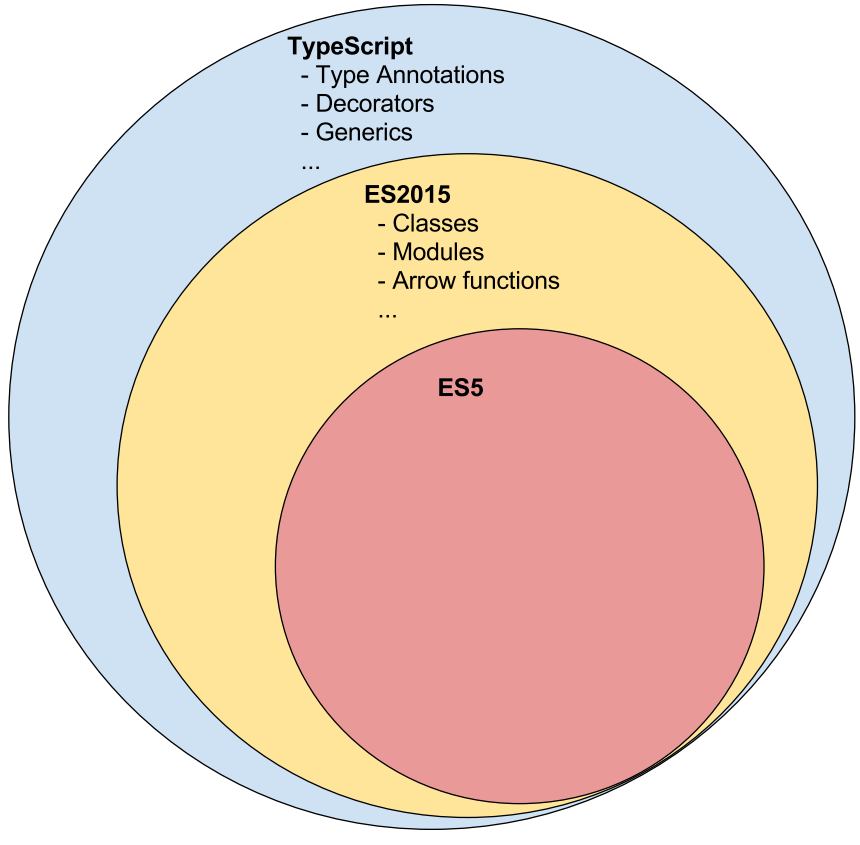

THOUGHTRAM
TRAINING
CODE REVIEW
BLOG
TRAINING WITH PASSION
LEARN ANGULAR AND GIT THE RIGHT WAY.


angular/angular - Gitter
https://gitter.im/angular/angular
GITTER
Where developers come to talk
REE FOR COMMUNITIES
IN OVER 130K DEVELOPERS
IN OVER 25K COMMUNITIES
SCUSS PROJECTS & CODE
REATE YOUR OWN COMMUNITY
EEPLY INTEGRATED WITH GITHUB
REE, FOREVER.
REAT FOR TEAMS
REATE A TIGHTER TEAM
OLLABORATE SMARTER
REE FOR ROOMS UP TO 25 PEOPLE
/USER/MONTH FOR LARGER TEAMS
rowser, Desktop and Mobile Apps.
angular/angular
Discussion for Angular2 only (AngularJS v1.x http://webchat.freenode.net/?channels=angula...
Search...
MikeRyan52 15:01
@fxck: you can use delay
Observable.of(1, 2, 3).delay(5000)
escardin 15:04
bah, I can't find code to make an observable how I'm thinking
I'm sure it's similar to promises, but I don't want to spend an hour hunting it down
if it were a promise, you could do
()=>{
var promise=new Promise();
setTimeout(0,()=>{
promise.resolve({foo:bar});
}
return promise;
}().then(good,bad);
fxck 15:06
well
this.loadTodos
.flatMap(() => http.get('api/todos.json').map(res => res.json()))
.map(payload => ({type: 'LOAD_TODOS',payload}));
this works
so I basically need to replace http.get(), which return an observable
with
Observable.of(1, 2, 3).delay(5000)
right
LOGIN WITH GITHUB TO START TALKING
PEOPLE REPO INFO
+Add See All (1981 members)
ACTIVITY
hasan-assi forked hasan-assi/angular
paulcodiny starred angular/angular
namitrawal starred angular/angular
cangosta starred angular/angular
cangosta commented #5275
ericmartinezr commented #5361
BorisWechselberger forked BorisWechselberger/angular
vikas-a starred angular/angular
flyingmutant commented #5291
Gitter Support

ES2015/TypeScript Basics


ES5


ES2015
- Classes
- Modules
- Arrow functions
...
ES5


TypeScript
- Type Annotations
- Decorators
- Generics
...
ES2015
- Classes
- Modules
- Arrow functions
...
ES5

Classes

Syntactic sugar for JavaScript prototypes introduced in ES2015.

Classes

Syntactic sugar for JavaScript prototypes introduced in ES2015.

```
class Car {

  manufacturer:string;

  constructor(manufacturer:string = 'BMW') {
    this.manufacturer = manufacturer;
  }

  drive(miles:number) {}
}

let bmw = new Car();
```

Classes

Syntactic sugar for JavaScript prototypes introduced in ES2015.

```
class Car {

  manufacturer:string;

  constructor(manufacturer:string = 'BMW') {
    this.manufacturer = manufacturer;
  }

  drive(miles:number) {}
}

let bmw = new Car();
```

Classes

Syntactic sugar for JavaScript prototypes introduced in ES2015.

```
class Car {

  manufacturer:string;

  constructor(manufacturer:string = 'BMW') {
    this.manufacturer = manufacturer;
  }

  drive(miles:number) {}
}

let bmw = new Car();
```

Classes

Syntactic sugar for JavaScript prototypes introduced in ES2015.

```
class Car {

  manufacturer:string;

  constructor(manufacturer:string = 'BMW') {
    this.manufacturer = manufacturer;
  }

  drive(miles:number) {}
}

let bmw = new Car();
```

Classes

Syntactic sugar for JavaScript prototypes introduced in ES2015.

```
class Car {

  manufacturer:string;

  constructor(manufacturer:string = 'BMW') {
    this.manufacturer = manufacturer;
  }

  drive(miles:number) {}
}

let bmw = new Car();
```

```
class Car { ... }

class Convertible extends Car {

}

let cabrio = new Convertible();
```

```
class Car { ... }

class Convertible extends Car {

}

let cabrio = new Convertible();
```

Modules

ES2015 brings a module system to the table that enables us to write modular code.

Modules

ES2015 brings a module system to the table that enables us to write modular code.

```
// Car.js

export class Car { ... }

export class Convertible extends Car {
  ...
}
```

Modules

ES2015 brings a module system to the table that enables us to write modular code.

```
// Car.js

export class Car { ... }

export class Convertible extends Car {
  ...
}
```

Modules

ES2015 brings a module system to the table that enables us to write modular code.

```
// App.js

import {Car, Convertible} from 'Car';

let bmw = new Car();
let cabrio = new Convertible();
```

Modules

ES2015 brings a module system to the table that enables us to write modular code.

```
// App.js

import {Car, Convertible} from 'Car';

let bmw = new Car();
let cabrio = new Convertible();
```

Modules

ES2015 brings a module system to the table that enables us to write modular code.

```
// App.js

import {Car, Convertible} from 'Car';

let bmw = new Car();
let cabrio = new Convertible();
```

Type Annotations

Type annotations provide optional static typing. Applied using `: T` syntax

Type Annotations

Type annotations provide optional static typing. Applied using `: T` syntax

```
let height:number = 6;
let isDone:boolean = true;
let name:string = 'thoughtram';

let list:number[] = [1, 2, 3];
let list:Array<number> = [1, 2, 3];

function add(x: number, y: number): number {
  return x+y;
}
```

Type Annotations

Type annotations provide optional static typing. Applied using `: T` syntax

```
let height:number = 6;
let isDone:boolean = true;
let name:string = 'thoughtram';

let list:number[] = [1, 2, 3];
let list:Array<number> = [1, 2, 3];

function add(x: number, y: number): number {
  return x+y;
}
```

Type Annotations

Type annotations provide optional static typing. Applied using `: T` syntax

```
let height:number = 6;
let isDone:boolean = true;
let name:string = 'thoughtram';

let list:number[] = [1, 2, 3];
let list:Array<number> = [1, 2, 3];

function add(x: number, y: number): number {
  return x+y;
}
```

Type Annotations

Type annotations provide optional static typing. Applied using `: T` syntax

```
let height:number = 6;
let isDone:boolean = true;
let name:string = 'thoughtram';

let list:number[] = [1, 2, 3];
let list:Array<number> = [1, 2, 3];

function add(x: number, y: number): number {
  return x+y;
}
```

Type Annotations

Type annotations provide optional static typing. Applied using `: T` syntax

```
let height:number = 6;
let isDone:boolean = true;
let name:string = 'thoughtram';

let list:number[] = [1, 2, 3];
let list:Array<number> = [1, 2, 3];

function add(x: number, y: number): number {
  return x+y;
}
```

Type Annotations

Type annotations provide optional static typing. Applied using `: T` syntax

```
let height:number = 6;
let isDone:boolean = true;
let name:string = 'thoughtram';

let list:number[] = [1, 2, 3];
let list:Array<number> = [1, 2, 3];

function add(x: number, y: number): number {
  return x+y;
}
```

Bootstrapping an app

Creating an Angular View Component...

```
export class ContactsAppComponent {}
```

1) Use the @Component() Decorator

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

@Component({
  selector: 'trm-contacts-app',
  template: 'Hello World!',
  styleUrls: ['./contacts.component.scss']
})
export class ContactsAppComponent {}
```

Every component is part of an **NgModule**

Every component is part of an **NgModule**

`ES2015 Module != **NgModule**`

NgModule Usages

NgModule Usages

1. Root Module

NgModule Usages

1. Root Module
2. Feature Module

NgModule Usages

1. Root Module
2. Feature Module
3. Routing Modules

NgModule Usages

1. Root Module
2. Feature Module
3. Routing Modules
4. Lazy-Loading Modules

NgModule Usages

1. Root Module
2. Feature Module
3. Routing Modules
4. Lazy-Loading Modules
5. Shared Modules

NgModule Usages

1. Root Module
2. Feature Module
3. Routing Modules
4. Lazy-Loading Modules
5. Shared Modules
6. Core Modules

NgModule Usages

1. **Root** Module
2. Feature Module
3. **Routing** Modules
4. **Lazy-Loading** Modules
5. Shared Modules
6. Core Modules

2) Root Module

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



@NgModule({



})
export class ContactsModule {}
```

2) Root Module

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

import { ContactsAppComponent } from './contacts.component';

@NgModule({

  declarations: [ContactsAppComponent],
  bootstrap: [ContactsAppComponent]
})
export class ContactsModule {}
```

2) Root Module

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

import { ContactsAppComponent } from './contacts.component';

@NgModule({

  declarations: [ContactsAppComponent],
  bootstrap: [ContactsAppComponent]
})
export class ContactsModule {}
```

2) Root Module

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

@NgModule({
  imports: [BrowserModule],
  declarations: [ContactsAppComponent],
  bootstrap: [ContactsAppComponent]
})
export class ContactsModule {}
```

2) Root Module

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

@NgModule({
  imports: [BrowserModule],
  declarations: [ContactsAppComponent],
  bootstrap: [ContactsAppComponent]
})
export class ContactsModule {}
```

How to bootstrap this module?

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

platformBrowserDynamic().bootstrapModule(ContactsModule);
```

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

platformBrowserDynamic().bootstrapModule(ContactsModule);
```

```
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Contacts</title>
  </head>
  <body>



    <script src="..."></script>
  </body>
</html>
```

```
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Contacts</title>
  </head>
  <body>

    <trm-contacts-app>Loading...</trm-contacts-app>

    <script src="..."></script>
  </body>
</html>
```

Bootstrapping Tasks

Angular performs the following tasks to bootstrap an application (simplified):

Bootstrapping Tasks

Angular performs the following tasks to bootstrap an application (simplified):

1. Upgrades located DOM element into Angular component

Bootstrapping Tasks

Angular performs the following tasks to bootstrap an application (simplified):

1. Upgrades located DOM element into Angular component
2. Creates injector for the application

Bootstrapping Tasks

Angular performs the following tasks to bootstrap an application (simplified):

1. Upgrades located DOM element into Angular component
2. Creates injector for the application
3. Creates (emulated) Shadow DOM on component's host element

Bootstrapping Tasks

Angular performs the following tasks to bootstrap an application (simplified):

1. Upgrades located DOM element into Angular component
2. Creates injector for the application
3. Creates (emulated) Shadow DOM on component's host element
4. Instantiates specified component

Bootstrapping Tasks

Angular performs the following tasks to bootstrap an application (simplified):

1. Upgrades located DOM element into Angular component
2. Creates injector for the application
3. Creates (emulated) Shadow DOM on component's host element
4. Instantiates specified component
5. Performs change detection

Demo: Bootstrapping an app

Using (3rd party) directives

Directive Dependencies

Directives/Components are added to an **NgModule**'s `declarations` property.

```
@NgModule({
  imports: [BrowserModule],
  declarations: [ContactsAppComponent],
  bootstrap: [ContactsAppComponent]
})
export class ContactsModule {}
```

Directive Dependencies

Directives/Components are added to an **NgModule**'s `declarations` property.

```
import { ContactsHeaderComponent } from './contacts-header';

@NgModule({
  imports: [BrowserModule],
  declarations: [ContactsAppComponent, ContactsHeaderComponent],
  bootstrap: [ContactsAppComponent]
})
export class ContactsModule {}
```

Directive Dependencies

Directives/Components are added to an **NgModule**'s `declarations` property.

```
import { ContactsHeaderComponent } from './contacts-header';

@NgModule({
  imports: [BrowserModule],
  declarations: [ContactsAppComponent, ContactsHeaderComponent],
  bootstrap: [ContactsAppComponent]
})
export class ContactsModule {}
```

3rd Party Directives

Directives/Components can be imported from another **NgModule**.

```
@NgModule({
  imports: [BrowserModule],
  declarations: [ContactsAppComponent],
  bootstrap: [ContactsAppComponent]
})
export class ContactsModule {}
```

3rd Party Directives

Directives/Components can be imported form another **NgModule**.

```
import { ContactsMaterialModule } from '...';

@NgModule({
  imports: [BrowserModule, ContactsMaterialModule],
  declarations: [ContactsAppComponent],
  bootstrap: [ContactsAppComponent]
})
export class ContactsModule {}
```

3rd Party Directives

Directives/Components can be imported from another **NgModule**.

```
import { ContactsMaterialModule } from '...';

@NgModule({
  imports: [BrowserModule, ContactsMaterialModule],
  declarations: [ContactsAppComponent],
  bootstrap: [ContactsAppComponent]
})
export class ContactsModule {}
```

Using Directives

Directives/Components can then be used in components throughout our module.

```
@Component({
  selector: 'trm-contacts-app',
  template: 'Hello World!',
  styleUrls: ['./contacts.component.scss']
})
export class ContactsAppComponent {}
```

Using Directives

Directives/Components can then be used in components throughout our module.

```
@Component({
  selector: 'trm-contacts-app',
  template: '<mat-toolbar>Contacts</mat-toolbar>',
  styleUrls: ['./contacts.component.scss']
})
export class ContactsAppComponent {}
```

Exercise 1: Using Directives

t forget to git commit your solution

Displaying Data

We can bind data to elements in HTML templates

Angular automatically updates the UI as data changes.

Let's display our first contact!

Defining a Contact

To enable type safety during compilation, we can define interfaces using the `interface` key word

Defining a Contact

To enable type safety during compilation, we can define interfaces using the `interface` key word

```
interface Contact {
  id: number;
  name: string;
  email: string;
  phone: string;
  ...
}
```

```
import { Contact } from './models/contact';

@Component(...)
export class ContactsAppComponent {









}
```

```
import { Contact } from './models/contact';

@Component(...)
export class ContactsAppComponent {
  contact: Contact = {
    id: 1,
    name: 'Christoph',
    email: 'chris@burgdorf.io',
    image: 'path/to/image',
    ...
  }
}
```

```
@Component({
  selector: 'trm-contacts-app',
  template: `
    <mat-list>
      <mat-list-item>
        <img [src]="contact.image">
        <h3 mat-line>{{contact.name}}</h3>
      </mat-list-item>
    </mat-list>
  `
})
export class ContactsAppComponent {
  ...
}
```

```
@Component({
  selector: 'trm-contacts-app',
  template: `
    <mat-list>
      <mat-list-item>
        <img [src]="contact.image">
        <h3 mat-line>{{contact.name}}</h3>
      </mat-list-item>
    </mat-list>
  `
})
export class ContactsAppComponent {
  ...
}
```

Exercise 2: Display first contact

t forget to git commit your solution

Property Binding Explained

Props and Attrs

Attributes are **always** strings and only set up the initial value

```
<input value="Christoph">
<img src="path/to/img.png">
```

Props and Attrs

Properties can be **any** value and changed imperatively at run-time

```
var input = document.querySelector('input');
input.value; // 'Christoph'

var img = document.querySelector('img');
img.src; // 'path/to/img.png'
```

Props and Attrs

Properties and attributes values aren't always reflected automatically

```
var input = document.querySelector('input');
input.value; // 'Christoph'

input.value = 'Pascal';
input.getAttribute('value'); // 'Christoph'
```

Props and Attrs

Properties and attributes values aren't always reflected automatically

```
var input = document.querySelector('input');
input.value; // 'Christoph'

input.value = 'Pascal';
input.getAttribute('value'); // 'Christoph'
```

Property Binding

Angular binds to element properties instead of attributes in order to work with **any** element

```
<div title="thoughtram"></div>
<div [title]="'thoughtram'"></div>
<div [title]="user.firstname"></div>
```

Property Binding

Angular binds to element properties instead of attributes in order to work with **any** element

```
<div title="thoughtram"></div>
<div [title]="'thoughtram'"></div>
<div [title]="user.firstname"></div>
```

[] - syntax is the declarative way of writing to a property

Property Binding

Angular binds to element properties instead of attributes in order to work with **any** element

```
<div title="thoughtram"></div>
<div [title]="'thoughtram'"></div>
<div [title]="user.firstname"></div>
```

[] - syntax is the declarative way of writing to a property

Longer version: **bind-[property]="expression"**

Creating a list

```
export const CONTACTS: Contact[] = [
  { id: 1, firstname: 'Christoph', ...},
  { id: 2, firstname: 'Pascal', ...},
  { id: 3, firstname: 'Julie', ...},
  { id: 4, firstname: 'Igor', ...},
  ...
];
```

```
import { CONTACTS } from './data/contact-data';

@Component({
  selector: 'trm-contacts-app',
  template: `
    <mat-list>
      <mat-list-item>
        <!-- each contact goes here -->
      </mat-list-item>
    </mat-list>
  `
})
export class ContactsAppComponent {
  contacts: Contact[] = CONTACTS;
}
```

```
import { CONTACTS } from './data/contact-data';

@Component({
  selector: 'trm-contacts-app',
  template: `
    <mat-list>
      <mat-list-item>
        <!-- each contact goes here -->
      </mat-list-item>
    </mat-list>
  `
})
export class ContactsAppComponent {
  contacts: Contact[] = CONTACTS;
}
```

Lists with `ngFor`

The `NgFor` directive instantiates a template once per item from an iterable.

```
<mat-list-item *ngFor="let item of items">
  ...
</mat-list-item>
```

Lists with `ngFor`

The `NgFor` directive instantiates a template once per item from an iterable.

```
<mat-list-item *ngFor="let item of items">
  ...
</mat-list-item>
```

- `*` - Indicates template directive

Lists with `ngFor`

The `NgFor` directive instantiates a template once per item from an iterable.

```
<mat-list-item *ngFor="let item of items">
  ...
</mat-list-item>
```

- `*` - Indicates template directive
- `items` - Expression evaluating to collection

Lists with `ngFor`

The `NgFor` directive instantiates a template once per item from an iterable.

```
<mat-list-item *ngFor="let item of items">
  ...
</mat-list-item>
```

- `*` - Indicates template directive
- `items` - Expression evaluating to collection
- `let item` - Local variable for each iterator

```
@Component({
  selector: 'trm-contacts-app',
  template: `
    <mat-list>
      <mat-list-item>
        <!-- each contact goes here -->

      </mat-list-item>
    </mat-list>
  `
})
export class ContactsAppComponent {
  ...
}
```

```
@Component({
  selector: 'trm-contacts-app',
  template: `
    <mat-list>
      <mat-list-item *ngFor="let contact of contacts">
        <img mat-list-avatar [src]="contact.image">
        <h3 mat-line>{{contact.name}}</h3>
      </mat-list-item>
    </mat-list>
  `
})
export class ContactsAppComponent {
  ...
}
```

Exercise 3: Display list of contacts

t forget to git commit your solution

Services and DI

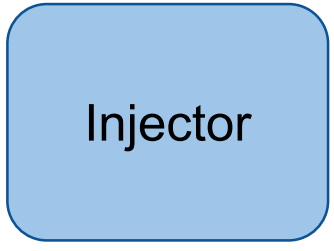
Services

We can create services in Angular using **ES2015 classes**.

Services

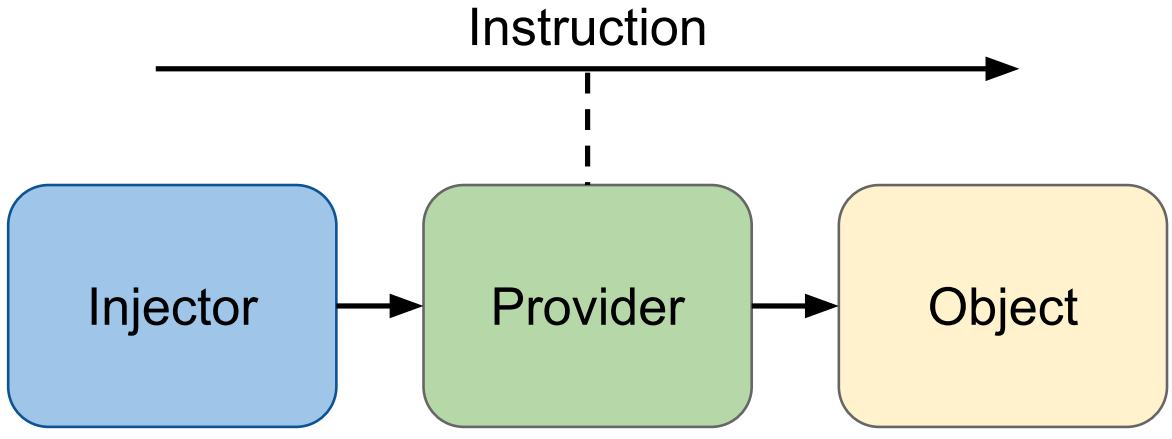
We can create services in Angular using **ES2015 classes**.

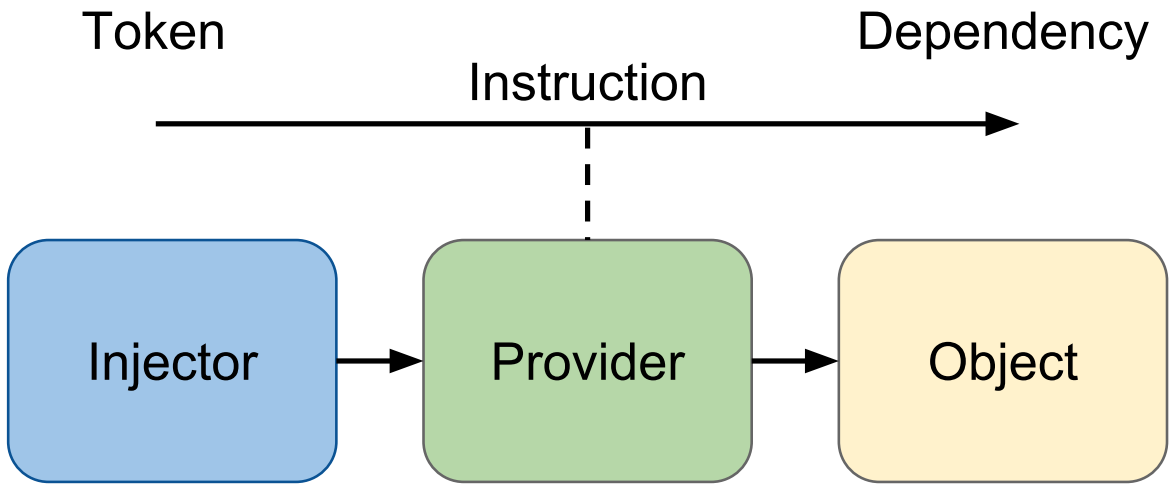
```
import { Injectable } from '@angular/core';

@Injectable()
export class ContactsService {

  private contacts: Contact[] = CONTACTS;

  getContacts() {
    return this.contacts;
  }
}
```


Injector


Injector
Object


Injector
Provider
Object


Instruction
Injector
Provider
Object


Token
Dependency
Instruction
Injector
Provider
Object

Configuring the Injector

An NgModule's **providers** property accepts a list of providers to configure the injector.

```
@NgModule({
  ...
  providers: [
    { provide: ContactsService, useClass: ContactsService }
  ]
})
export class ContactsModule {}
```

Configuring the Injector

An NgModule's **providers** property accepts a list of providers to configure the injector.

```
@NgModule({
  ...
  providers: [
    { provide: ContactsService, useClass: ContactsService }
  ]
})
export class ContactsModule {}
```

Configuring the Injector

An NgModule's **providers** property accepts a list of providers to configure the injector.

```
@NgModule({
  ...
  providers: [
    ContactsService
  ]
})
export class ContactsModule {}
```

Injecting Dependencies

Dependencies can be injected using TypeScript type annotations.

```
@Component(...)
export class ContactsAppComponent {

  contacts: Contact[];

  constructor(contactsService: ContactsService) {
    this.contacts = contactsService.getContacts();
  }
}
```

Injecting Dependencies

Dependencies can be injected using TypeScript type annotations.

```
@Component(...)
export class ContactsAppComponent {

  contacts: Contact[];

  constructor(contactsService: ContactsService) {
    this.contacts = contactsService.getContacts();
  }
}
```

Lifecycle Hooks

Directives and Components have lifecycle hooks. Some of them are:

Lifecycle Hooks

Directives and Components have lifecycle hooks. Some of them are:

- **ngOnInit** - Initializes directive

Lifecycle Hooks

Directives and Components have lifecycle hooks. Some of them are:

- **ngOnInit** - Initializes directive
- **ngOnChanges** - Respond after Angular sets data-bound property

Lifecycle Hooks

Directives and Components have lifecycle hooks. Some of them are:

- **ngOnInit** - Initializes directive
- **ngOnChanges** - Respond after Angular sets data-bound property
- **ngDoCheck** - Custom change detection

Lifecycle Hooks

Directives and Components have lifecycle hooks. Some of them are:

- **ngOnInit** - Initializes directive
- **ngOnChanges** - Respond after Angular sets data-bound property
- **ngDoCheck** - Custom change detection
- **ngOnDestroy** - Cleanup before Angular destroys directive

OnInit Lifecycle

We use **ngOnInit** to do initialization work in the component.

```
import { OnInit } from '@angular/core';
...
export class ContactsAppComponent implements OnInit {

  contacts: Contact[];

  constructor(private contactsService: ContactsService) {}

  ngOnInit() {
    this.contacts = this.contactsService.getContacts();
  }
}
```

OnInit Lifecycle

We use **ngOnInit** to do initialization work in the component.

```
import { OnInit } from '@angular/core';
...
export class ContactsAppComponent implements OnInit {

  contacts: Contact[];

  constructor(private contactsService: ContactsService) {}

  ngOnInit() {
    this.contacts = this.contactsService.getContacts();
  }
}
```

Keep in mind

Keep in mind

- We use the token (or type) to ask for a dependency **instance**

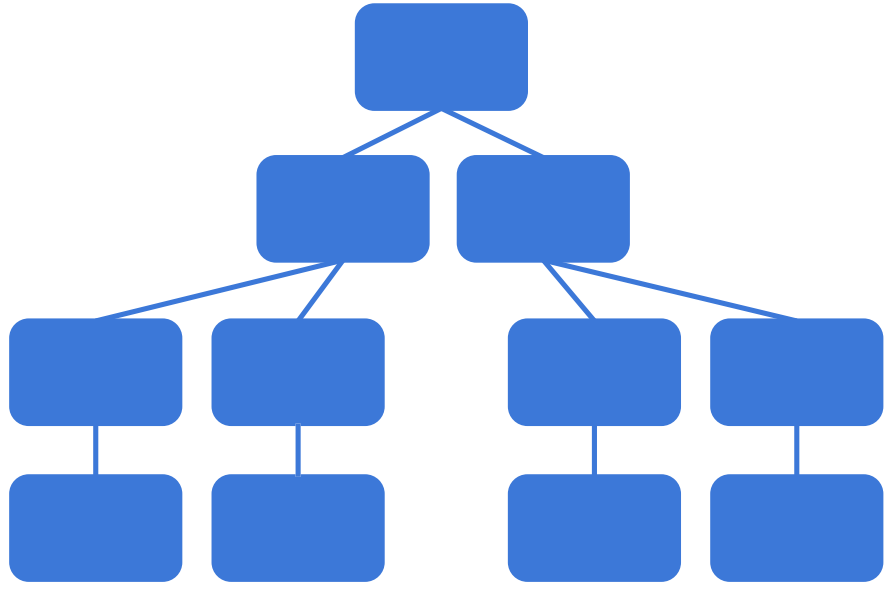
Keep in mind

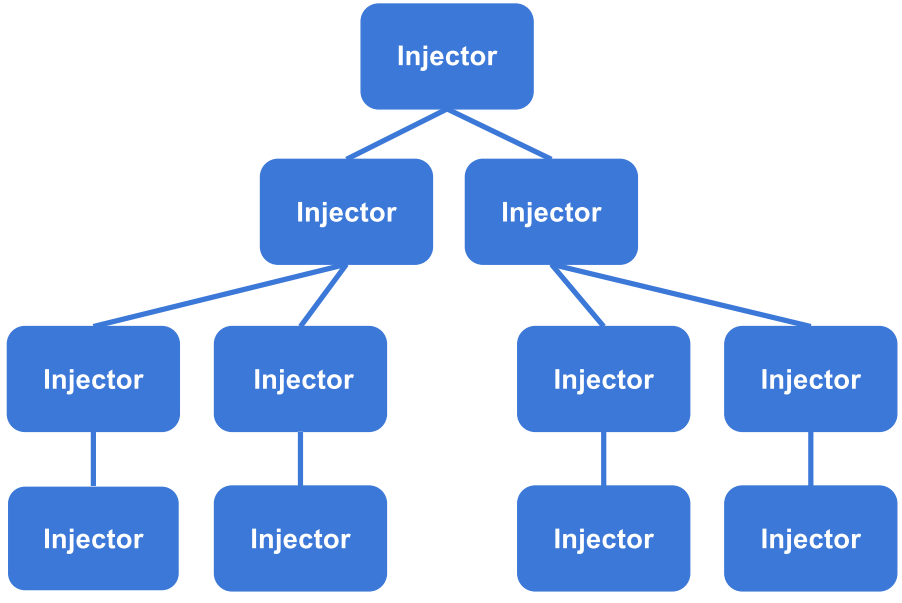
- We use the token (or type) to ask for a dependency **instance**
- But a provider defines **what** and **how** an object is being created for that token (or type)

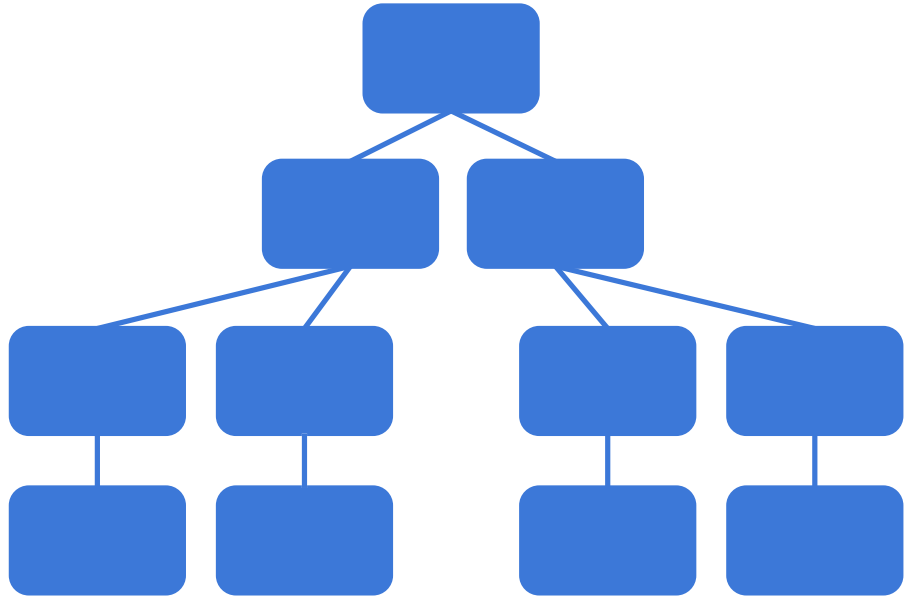
Exercise 4: Services and DI

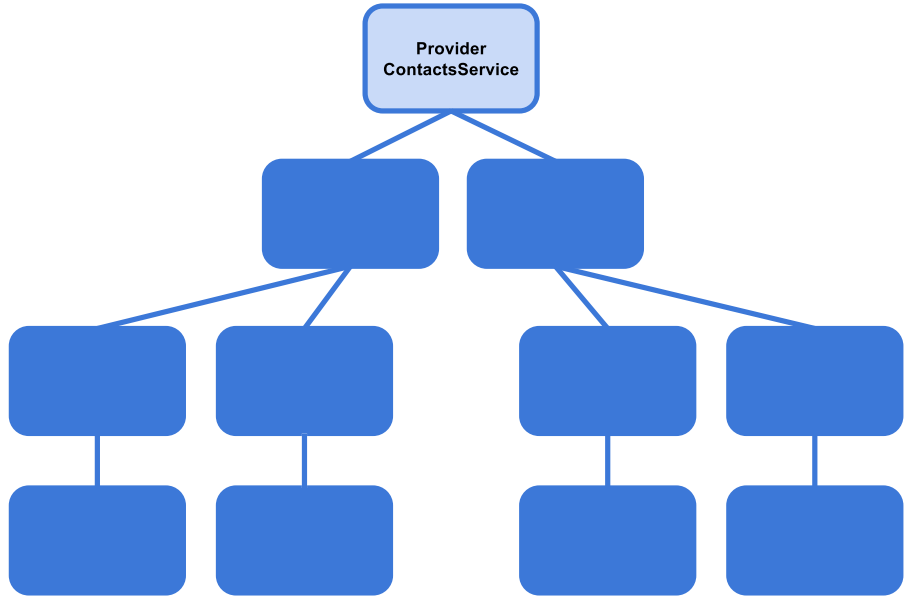
t forget to git commit your solution

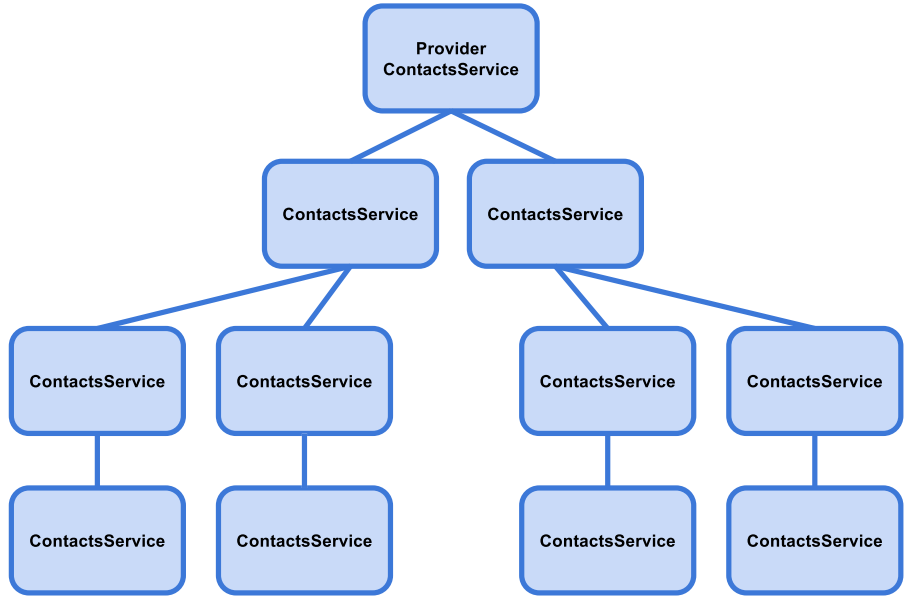
Understanding the injector tree

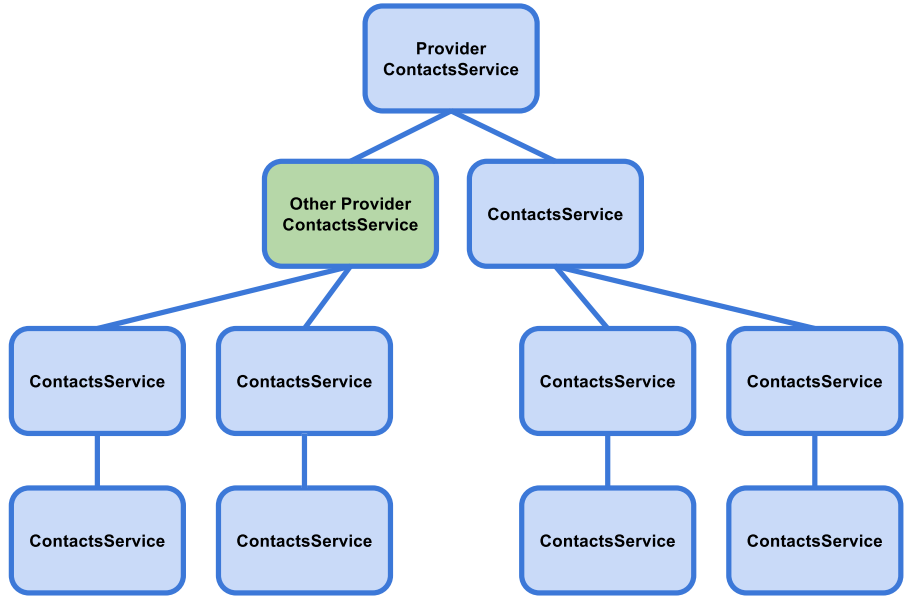


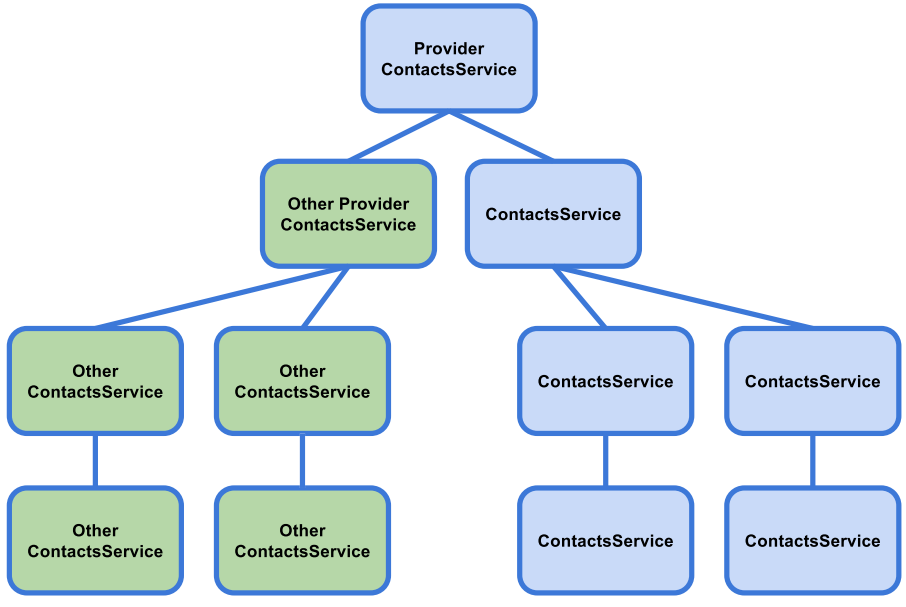

Injector
Injector
Injector
Injector
Injector
Injector
Injector
Injector
Injector
Injector



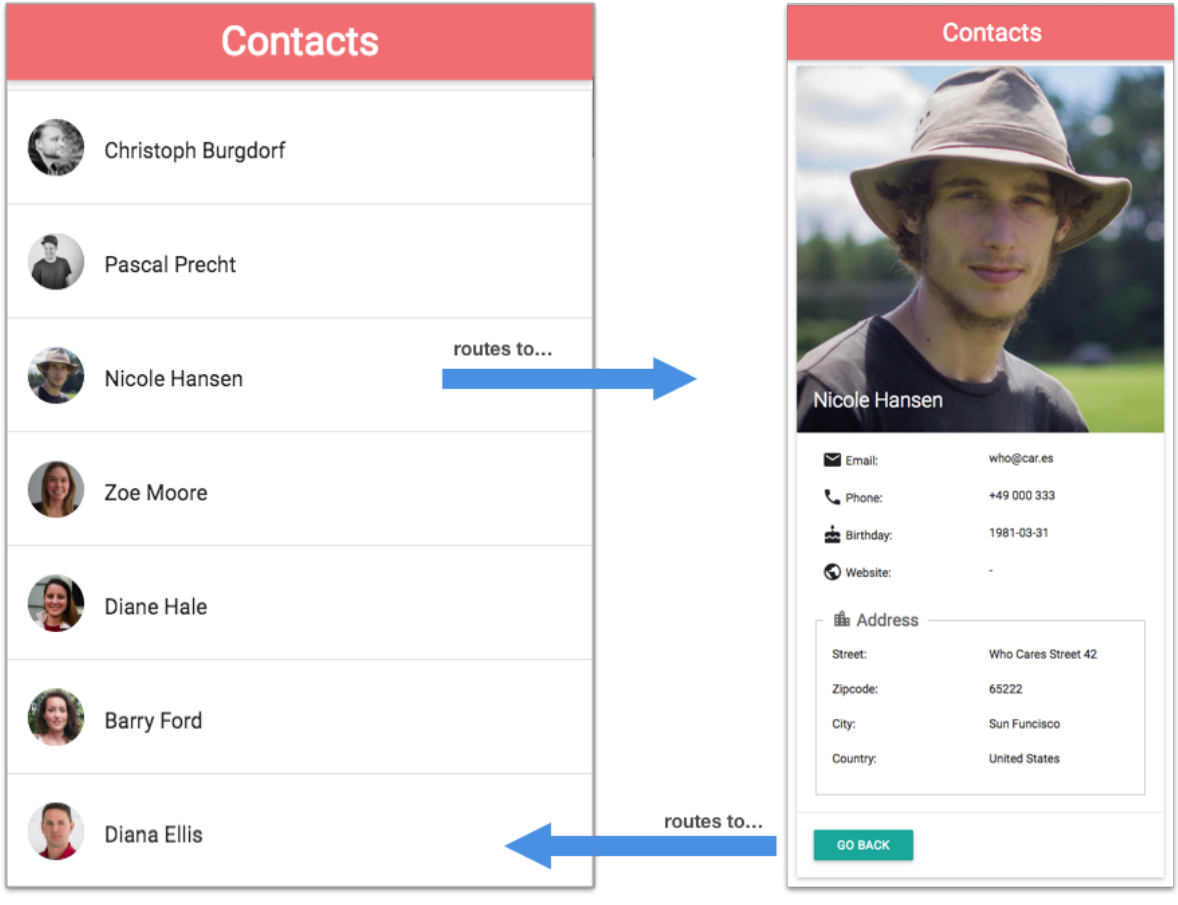

Provider
ContactsService


Provider
ContactsService
ContactsService
ContactsService
ContactsService
ContactsService
ContactsService
ContactsService
ContactsService
ContactsService
ContactsService


Provider
ContactsService
Other Provider
ContactsService
ContactsService
ContactsService
ContactsService
ContactsService
ContactsService
ContactsService
ContactsService
ContactsService


Provider
ContactsService
Other Provider
ContactsService
ContactsService
Other
ContactsService
Other
ContactsService
ContactsService
ContactsService
Other
ContactsService
Other
ContactsService
ContactsService
ContactsService

Component Routing


Contacts
Christoph Burgdorf
Pascal Precht
Nicole Hansen
Zoe Moore
Diane Hale
Barry Ford
Diana Ellis
routes to...
routes to...
Contacts
Nicole Hansen
Email:
who@car.es
Phone:
+49 000 333
Birthday:
1981-03-31
Website:
-
Address
Street:
Who Cares Street 42
Zipcode:
65222
City:
Sun Funcisco
Country:
United States
GO BACK

Components we have

Our application consists of three components:

Components we have

Our application consists of three components:

- **ContactsAppComponent** - The root component that is being bootstrapped

Components we have

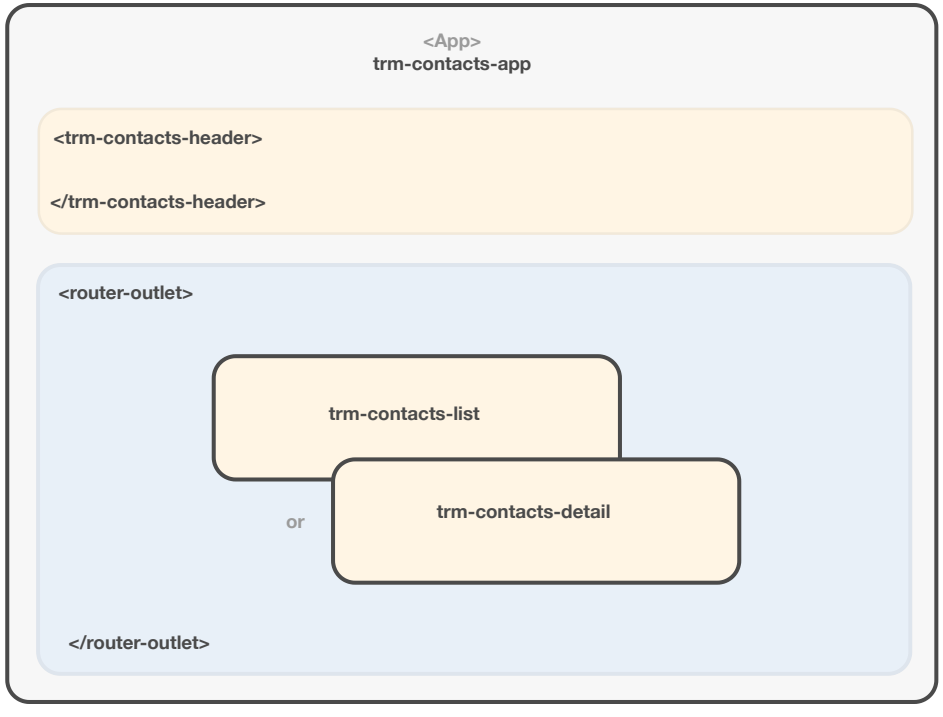
Our application consists of three components:

- **ContactsAppComponent** - The root component that is being bootstrapped
- **ContactsListComponent** - A component to list contacts by provided data

Components we have

Our application consists of three components:

- **ContactsAppComponent** - The root component that is being bootstrapped
- **ContactsListComponent** - A component to list contacts by provided data
- **ContactDetailComponent** - A component to show a contact's details


<App>
trm-contacts-app
<trm-contacts-header>
</trm-contacts-header>
<router-outlet>
trm-contacts-list
or
trm-contacts-detail
</router-outlet>

Defining Routes

We define routes as a collection of **Route** objects that have a **path** and a **component**

```
import { ContactsListComponent } from './contacts-list';

export const APP_ROUTES = [
  { path: '', component: ContactsListComponent }
]
```

Defining Routes

We define routes as a collection of **Route** objects that have a **path** and a **component**

```
import { ContactsListComponent } from './contacts-list';

export const APP_ROUTES = [
  { path: '', component: ContactsListComponent }
]
```

Adding Router Module

RouterModule.forRoot() configures routes for the root module of our app.

Adding Router Module

RouterModule.forRoot() configures routes for the root module of our app.

```
import { RouterModule } from '@angular/router';
import { APP_ROUTES } from './app.routes';

@NgModule({
  imports: [
    ...
    RouterModule.forRoot(APP_ROUTES)
  ]
})
export class ContactsModule {}
```

Adding Router Module

RouterModule.forRoot() configures routes for the root module of our app.

```
import { RouterModule } from '@angular/router';
import { APP_ROUTES } from './app.routes';

@NgModule({
  imports: [
    ...
    RouterModule.forRoot(APP_ROUTES)
  ]
})
export class ContactsModule {}
```

Displaying Components

`<router-outlet>` directive specifies a **viewport** where components should be loaded into.

```
@Component({
  selector: 'trm-contacts-app',
  template: `
    <mat-toolbar>Contacts</mat-toolbar>

  `,
  styleUrls: ['./contacts.component.scss']
})
class ContactsAppComponent {...}
```

Displaying Components

`<router-outlet>` directive specifies a **viewport** where components should be loaded into.

```
@Component({
  selector: 'trm-contacts-app',
  template: `
    <mat-toolbar>Contacts</mat-toolbar>
    <router-outlet></router-outlet>
  `,
  styleUrls: ['./contacts.component.scss']
})
class ContactsAppComponent {...}
```

xercise 5: Route to first component

t forget to git commit your solution

Linking to other components

Links with `RouterLink`

`RouterLink` lets link to a specific part of our app.

Links with `RouterLink`

`RouterLink` lets link to a specific part of our app.

```
<mat-list>
  <mat-list-item
    *ngFor="let contact of contacts">
    <img mat-list-avatar src="{{contact.image}}">
    <h3 mat-line>{{contact.name}}</h3>
  </mat-list-item>
</mat-list>
```

Links with `RouterLink`

`RouterLink` lets link to a specific part of our app.

```
<mat-nav-list>
  <a mat-list-item
    *ngFor="let contact of contacts">
    <img mat-list-avatar src="{{contact.image}}">
    <h3 mat-line>{{contact.name}}</h3>
  </a>
</mat-nav-list>
```

Links with `RouterLink`

`RouterLink` lets link to a specific part of our app.

```
<mat-nav-list>
  <a mat-list-item
    *ngFor="let contact of contacts">
    <img mat-list-avatar src="{{contact.image}}">
    <h3 mat-line>{{contact.name}}</h3>
  </a>
</mat-nav-list>
```

Links with `RouterLink`

`RouterLink` lets link to a specific part of our app.

```
<mat-nav-list>
  <a mat-list-item [routerLink]="['/contact', contact.id]"
    *ngFor="let contact of contacts">
    <img mat-list-avatar src="{{contact.image}}">
    <h3 mat-line>{{contact.name}}</h3>
  </a>
</mat-nav-list>
```

```
import { ContactsListComponent } from './contacts-list';


export const APP_ROUTES: Routes = [
  { path: '', component: ContactsListComponent }

];
```

```
import { ContactsListComponent } from './contacts-list';
import { ContactsDetailComponent } from './contacts-detail';

export const APP_ROUTES: Routes = [
  { path: '', component: ContactsListComponent },
  { path: 'contact/:id', component: ContactsDetailComponent }
];
```

```
import { ContactsListComponent } from './contacts-list';
import { ContactsDetailComponent } from './contacts-detail';

export const APP_ROUTES: Routes = [
  { path: '', component: ContactsListComponent },
  { path: 'contact/:id', component: ContactsDetailComponent }
];
```

Retrieving `parameters`

Retrieving parameters

```
import { ActivatedRoute } from '@angular/router';

@Component(...)
class ContactsDetailComponent implements OnInit {

  constructor(route: ActivatedRoute) {}

  ngOnInit() {
    let id = this.route.snapshot.params['id'];
    this.contact = this.contactsService
                       .getContact(id);
  }
}
```

Retrieving parameters

```
import { ActivatedRoute } from '@angular/router';

@Component(...)
class ContactsDetailComponent implements OnInit {

  constructor(route: ActivatedRoute) {}

  ngOnInit() {
    let id = this.route.snapshot.params['id'];
    this.contact = this.contactsService
                       .getContact(id);
  }
}
```

Exercise 6: Using RouteParams

t forget to git commit your solution

Fetching Data

Http in Angular

Angular (2.x) introduces a completely redesigned Http layer based on Observables.

Http in Angular

Angular (2.x) introduces a completely redesigned Http layer based on Observables.

- Supports XMLHttpRequest API

Http in Angular

Angular (2.x) introduces a completely redesigned Http layer based on Observables.

- Supports XMLHttpRequest API
- Event-driven stream of notifications

Http in Angular

Angular (2.x) introduces a completely redesigned Http layer based on Observables.

- Supports XMLHttpRequest API
- Event-driven stream of notifications
- Enables functional programming structures

Http in Angular

Angular (2.x) introduces a completely redesigned Http layer based on Observables.

- Supports XMLHttpRequest API
- Event-driven stream of notifications
- Enables functional programming structures
- Transfrom Observables to Promises (if needed)

HttpClient in Angular

Angular (4.3) introduces a completely redesigned HttpClient layer based on Observables.

HttpClient in Angular

Angular (4.3) introduces a completely redesigned HttpClient layer based on Observables.

- Simplified API

HttpClient in Angular

Angular (4.3) introduces a completely redesigned HttpClient layer based on Observables.

- Simplified API
- Auto-conversion to JSON objects

HttpClient in Angular

Angular (4.3) introduces a completely redesigned HttpClient layer based on Observables.

- Simplified API
- Auto-conversion to JSON objects
- Strong-typing for request and response objects

HttpClient in Angular

Angular (4.3) introduces a completely redesigned HttpClient layer based on Observables.

- Simplified API
- Auto-conversion to JSON objects
- Strong-typing for request and response objects
- Request and Response interceptors

HttpClient in Angular

Angular (4.3) introduces a completely redesigned HttpClient layer based on Observables.

- Simplified API
- Auto-conversion to JSON objects
- Strong-typing for request and response objects
- Request and Response interceptors
- Improved Error Handling w/ retry()

HttpClient in Angular

Angular (4.3) introduces a completely redesigned HttpClient layer based on Observables.

- Simplified API
- Auto-conversion to JSON objects
- Strong-typing for request and response objects
- Request and Response interceptors
- Improved Error Handling w/ retry()
- Progress Events (upload & download)

Adding HttpClientModule

HttpClientModule configures the injector for http dependencies.

Adding HttpClientModule

HttpClientModule configures the injector for http dependencies.

```
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  imports: [
    ...
    HttpClientModule
  ]
})
export class ContactsModule {...}
```

Adding HttpClientModule

HttpClientModule configures the injector for http dependencies.

```
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  imports: [
    ...
    HttpClientModule
  ]
})
export class ContactsModule {...}
```

Http Service

`HttpClient` is an injectable class with methods to perform http requests.

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { map } from 'rxjs/operators';

@Injectable()
class ContactsService {
  constructor(private http: HttpClient) {

  }
```

Http Service

`HttpClient` is an injectable class with methods to perform http requests.

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { map } from 'rxjs/operators';

@Injectable()
class ContactsService {
  constructor(private http: HttpClient) {

  }
}
```

Http Service

`HttpClient` is an injectable class with methods to perform http requests.

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { map } from 'rxjs/operators';

@Injectable()
class ContactsService {
  constructor(private http: HttpClient) {

  }
}
```

Http + Observables

`HttpClient` returns an Observable which will emit a single JSON response object.

```
import { map } from 'rxjs/operators';

class ContactsService {
  ...
  getContacts() {
    let url = 'http://myapi.com/contacts';
    return this.http.get<any>(url)
      .pipe(map((data) => data.items));
  }
}
```

Http + Observables

`HttpClient` returns an Observable which will emit a single JSON response object.

```
import { map } from 'rxjs/operators';

class ContactsService {
  ...
  getContacts() {
    let url = 'http://myapi.com/contacts';
    return this.http.get<any>(url)
      .pipe(map((data) => data.items));
  }
}
```

Http + Observables

`HttpClient` returns an Observable which will emit a single **untyped** JSON response object.

```
class ContactsService {
  ...
  getContacts() {
    let url = 'http://myapi.com/contacts';
    return this.http.get<any>(url)
      .pipe(map((data) => data.items));
  }
}
```

Http + Observables

`HttpClient` supports STRONGLY-typed response objects using Interfaces.

```
import { Contact } from './models/contact';

  interface ContactResponse  { item  : Contact    }
  interface ContactsResponse { items : Contact[]  }

class ContactsService {
  ...
}
```

Http + Observables

`HttpClient` supports STRONGLY-typed response objects using Interfaces.

```
import { Contact } from './models/contact';

class ContactsService {
  ...
  getContacts() : Observable<Array<Contact>>  {
    let url = 'http://myapi.com/contacts';
    return this.http.get<ContactsResponse>(url)
      .pipe(map(data => data.items));
}
```

Subscribing to Http

We can subscribe to Observables using `.subscribe()`

```
@Component(...)
class ContactsListComponent {
  public contacts : Contact[];

  constructor(contactsService: ContactsService) {
    contactsService.getContacts()
      .subscribe(contacts => {
          this.contacts = contacts;
      });
  }
}
```

Subscribing to Http

We can subscribe to Observables using `.subscribe()`

```
@Component(...)
class ContactsListComponent {
  public contacts : Contact[];

  constructor(contactsService: ContactsService) {
    contactsService.getContacts()
      .subscribe(contacts => {
          this.contacts = contacts;
      });
  }
}
```

Safe Navigation Operator

Angular throws a null reference error when null objects are accessed:

```
The null contact's name is {{nullContact.name}}
```

Safe Navigation Operator

Angular throws a null reference error when null objects are accessed:

```
The null contact's name is {{nullContact.name}}
```

Results in:

```
TypeError: Cannot read property 'name' of null in [null]
```

Safe Navigation Operator

The "safe navigation" operator (`?.`) is a convenient way to guard against nulls in property paths

```
The null contact's name is {{nullContact?.name}}
```

Safe Navigation Operator

The "safe navigation" operator (`?.`) is a convenient way to guard against nulls in property paths

```
The null contact's name is {{nullContact?.name}}
```

Exercise 7: Fetching data using Http

t forget to git commit your solution

Two-way Data Binding

`NgModel` Directive

`NgModel` implements two-way data binding by providing and combining property and event bindings.

```
<input [(ngModel)]="name">
<p>Hello, {{name}}!</p>
```

`NgModel` Directive

`NgModel` implements two-way data binding by providing and combining property and event bindings.

```
<input [(ngModel)]="name">
<p>Hello, {{name}}!</p>
```

Or, the canonical version:

```
<input bindon-ngModel="name">
<p>Hello, {{name}}!</p>
```

Two-way Binding can be implemented like this:

```
<input [value]="name"
  (input)="name=$event.target.value">
```

Two-way Binding can be implemented like this:

```
<input [value]="name"
  (input)="name=$event.target.value">
```

`ngModel` hides repetitive expressions:

```
<input [ngModel]="name"
  (ngModelChange)="name=$event">
```

Two-way Binding can be implemented like this:

```
<input [value]="name"
  (input)="name=$event.target.value">
```

`ngModel` hides repetitive expressions:

```
<input [ngModel]="name"
  (ngModelChange)="name=$event">
```

Shorthand syntax:

```
<input [(ngModel)]="name">
```

Two-way Binding can be implemented like this:

```
<input [value]="name"
  (input)="name=$event.target.value">
```

`ngModel` hides repetitive expressions:

```
<input [ngModel]="name"
  (ngModelChange)="name=$event">
```

Shorthand syntax:

```
<input [(ngModel)]="name">
```

Adding FormsModule

To use **NgModel** we need to add **FormsModule** to our application.

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

@NgModule({
  imports: [
    ...
    FormsModule
  ]
})
export class ContactsModule {}
```

Simple Forms

Using `ngModel` we can already create simple forms.

```
<form>

  <label>Firstname:</label>
  <input [(ngModel)]="contact.firstname">

  <label>Lastname:</label>
  <input [(ngModel)]="contact.lastname">
  ...
</form>
```

Put Requests

`HttpClient` sends data with the `put` request method

```
class ContactsService {
  ...
  updateContact(contact: Contact) {
    let url = `myapi.com/contacts/${contact.id}`;

    return this.http.put(url, contact);
  }
}
```

Put Requests

`HttpClient` sends data with the `put` request method

```
class ContactsService {
  ...
  updateContact(contact: Contact) {
    let url = `myapi.com/contacts/${contact.id}`;

    return this.http.put(url, contact);
  }
}
```

Put Requests

`HttpClient` sends data with the `put` request method

```
class ContactsService {
  ...
  updateContact(contact: Contact) {
    let url = `myapi.com/contacts/${contact.id}`;

    return this.http.put(url, contact);
  }
}
```

Exercise 8: Sending data using Http

t forget to git commit your solution

Using Async Pipe

Pipes

A pipe takes in data as input and transforms it to a desired output.

```
@Component({
  selector: 'trm-contacts-detail',
  template: `
    ...
    <span>Birthday:</span>
    <span>{{contact?.dateOfBirth | date}}</span>
  `
})
class ContactsDetailComponent { }
```

Pipes

A pipe takes in data as input and transforms it to a desired output.

```
@Component({
  selector: 'trm-contacts-detail',
  template: `
    ...
    <span>Birthday:</span>
    <span>{{contact?.dateOfBirth | date}}</span>
  `
})
class ContactsDetailComponent { }
```

AsyncPipe

The Async pipe can receive a Promise or Observable as input and subscribe to the input automatically.

```
import { Observable } from 'rxjs/Observable';

@Component(...)
class ContactsListComponent {

  contacts: Observable<Array<Contact>>;

  constructor(contactsService: ContactsService) {
    this.contacts = contactsService.getContacts();
  }
}
```

AsyncPipe

The Async pipe can receive a Promise or Observable as input and subscribe to the input automatically.

```
import { Observable } from 'rxjs/Observable';

@Component(...)
class ContactsListComponent {

  contacts: Observable<Array<Contact>>;

  constructor(contactsService: ContactsService) {
    this.contacts = contactsService.getContacts();
  }
}
```

AsyncPipe

The Async pipe can receive a Promise or Observable as input and subscribe to the input automatically.

```
<md-nav-list>
  <a md-list-item
     *ngFor="let contact of contacts | async">
    ...
  </a>
</md-nav-list>
```

AsyncPipe

The Async pipe can receive a Promise or Observable as input and subscribe to the input automatically.

```
<md-nav-list>
  <a md-list-item
     *ngFor="let contact of contacts | async">
    ...
  </a>
</md-nav-list>
```

Bonus: AsyncPipe with *ngIf

AsyncPipe + observables can be used with ***ngIf** to conditionally switch between two (2) views.

```
<div *ngif="contact$ | async as contact; else loading">
  Name: {{ contact.name }} <br>
  Twitter: {{ contact.twitter }}
</div>
<ng-template #loading="">
  Loading contact...
</ng-template>
```

3 Things
you did not know about
AsyncPipe

- Subscribing to long-lived Observables
- Keeping track of references
- Marking things for check

thoughtramBlog Article

Exercise 9: Async Pipe

t forget to git commit your solution

- git add .
- git commit -am "(completed) - jump-start"
- git tag classroom/rxjs