AngularJS Days 2015

Exploring Angular 2

Welcome!

About us

Christoph Burgdorf

@cburgdorf

Pascal Precht

@PascalPrecht

Blog | thoughtram - Extend

blog.thoughtram.io

blog.thoughtram.io

*Only interested in **AngularJS**, **Angular 2** or **Git**?*

Exploring Angular 2 by thou

blog.thoughtram.io/exploring-angular-2/

Todays plans

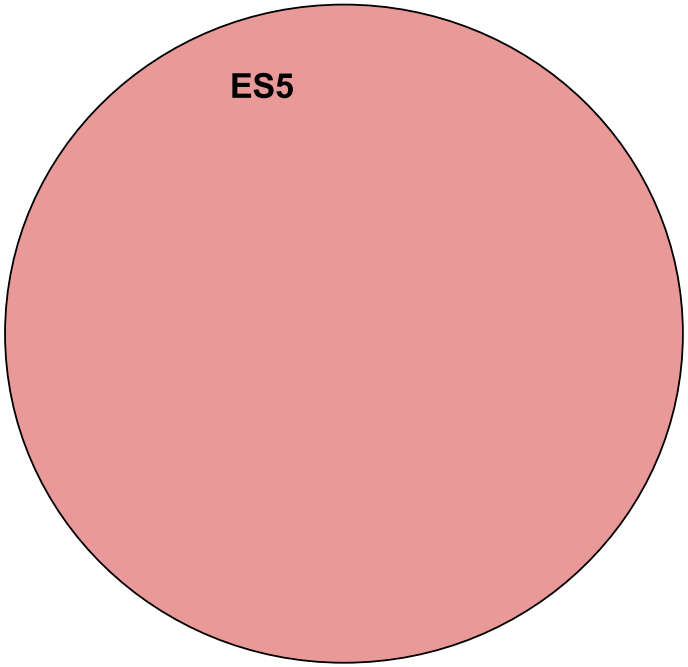
Todays Plans

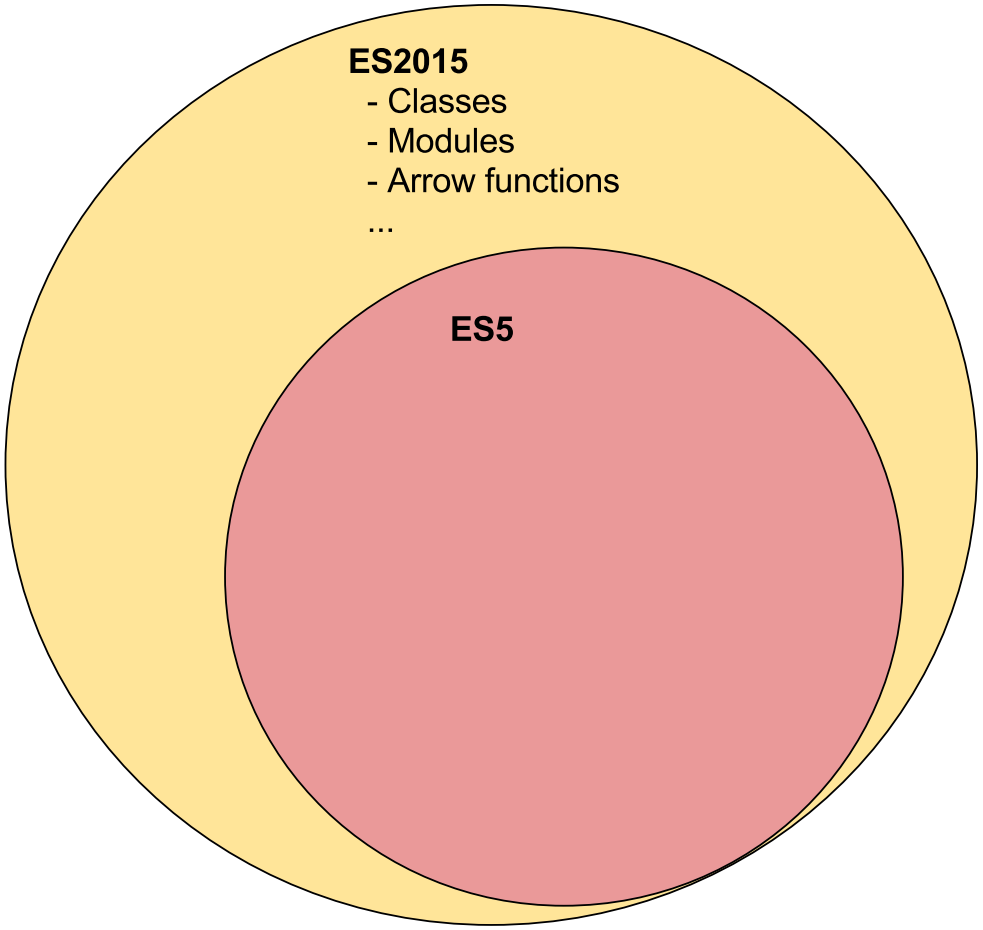
1. **ES2015/TypeScript Basics** - Classes, Decorators, Modules...
2. **Bootstrapping** - We bootstrap our first application
3. **Services** - Implementing a simple service
4. **Dependency Injection** - Injector Bindings and Trees
5. **Directives** - A brief usage of built-in directives
6. **Template Syntax** - Binding types
7. **Routing** - Simple component routing
8. **Forms** - View and Model driven

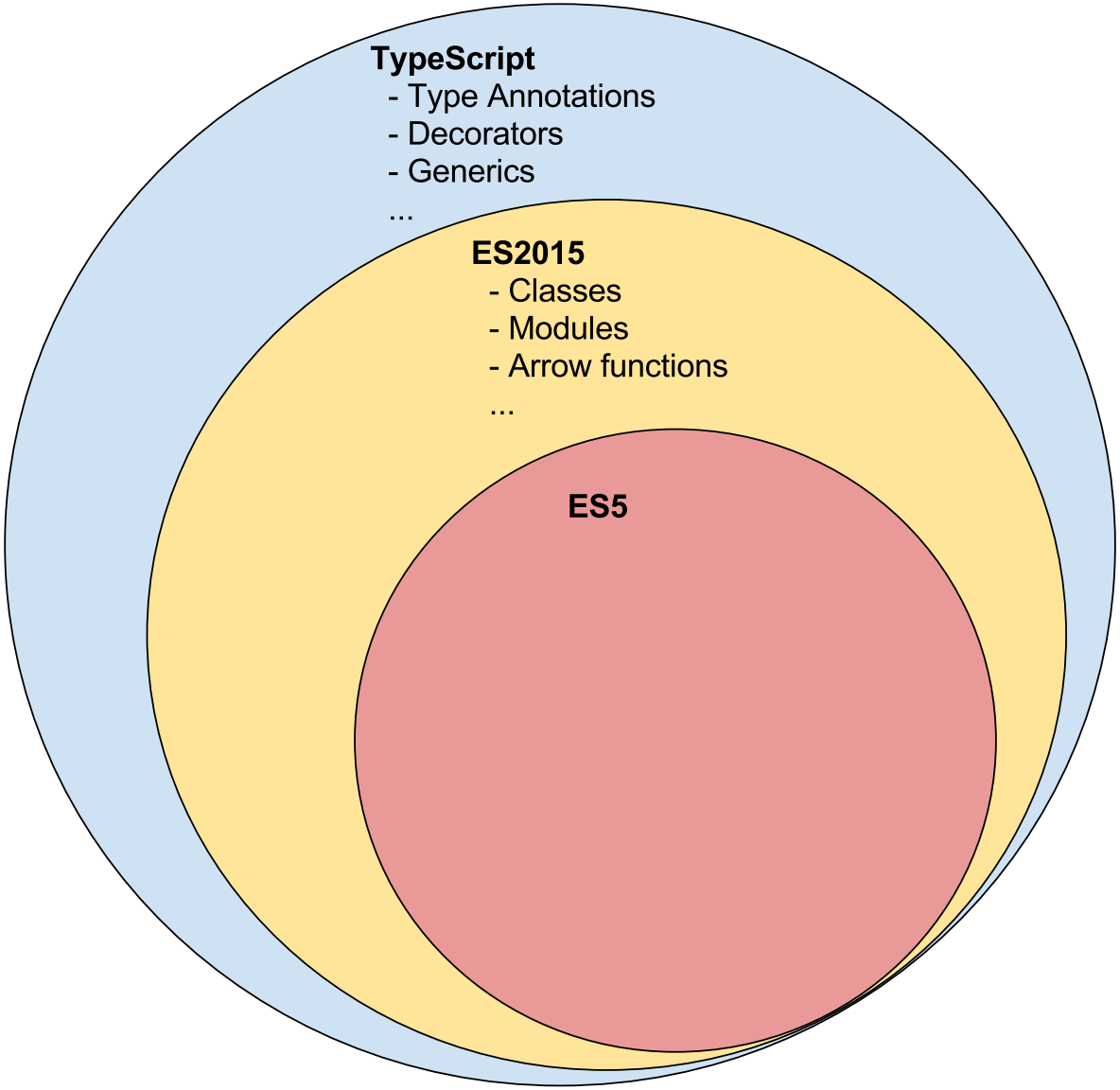
Ask questions whenever you like!

Here's what we build today

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.

```
class Car {

  manufacturer:string;

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

  drive(miles:number) {}
}

let bmw = new Car();
```

```
class Car {

  manufacturer:string;

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

  drive(miles:number) {}
}

let bmw = new Car();
```

```
class Car {

  manufacturer:string;

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

  drive(miles:number) {}
}

let bmw = new Car();
```

```
class Car {

  manufacturer:string;

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

  drive(miles:number) {}
}

let bmw = new Car();
```

```
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.

```
// Car.js

export class Car { ... }

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

```
// Car.js

export class Car { ... }

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

```
// App.js

import {Car, Convertible} from 'Car';

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

```
// App.js

import {Car, Convertible} from 'Car';

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

```
// 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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Decorators

A decorator is an **expression** that is evaluated after a class has been defined, that can be used to **annotate or modify** the class in some fashion.

```
@someDecoratorExpression()
class Car {

  @propertyDecorator() manufacturer: string;

  constructor(@paramDecorator() manufacturer: string) {

  }

  @methodDecorator()
  drive() {

  }
}
```

```
@someDecoratorExpression()
class Car {

  @propertyDecorator() manufacturer: string;

  constructor(@paramDecorator() manufacturer: string) {

  }

  @methodDecorator()
  drive() {

  }
}
```

Angular 2

Warning!

Everything in this workshop is based on alpha developer previews. Things might change in the future.

(Do not try this in production!)

Our first component

A component in Angular 2 is a **class** with a `@Component` and `@View` decorator.

```
class ContactsApp {

}
```

```
import {Component, View} from 'angular2/core';



@Component({
  selector: 'contacts-app'
})
@View({
  template: 'Hello World!'
})
class ContactsApp {

}
```

How to instantiate that component?

```
import {Component, View} from 'angular2/core';



@Component({
  selector: 'contacts-app'
})
@View({
  template: 'Hello World!'
})
class ContactsApp {

}
```

```
import {Component, View} from 'angular2/core';
import {bootstrap} from 'angular2/core';

@Component({
  selector: 'contacts-app'
})
@View({
  template: 'Hello World!'
})
class ContactsApp {

}
bootstrap(ContactsApp);
```

```
import {Component, View} from 'angular2/core';
import {bootstrap} from 'angular2/core';

@Component({
  selector: 'contacts-app'
})
@View({
  template: 'Hello World!'
})
class ContactsApp {

}
bootstrap(ContactsApp);
```

```html
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>My first Angular 2 App!</title>
  </head>
  <body>




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

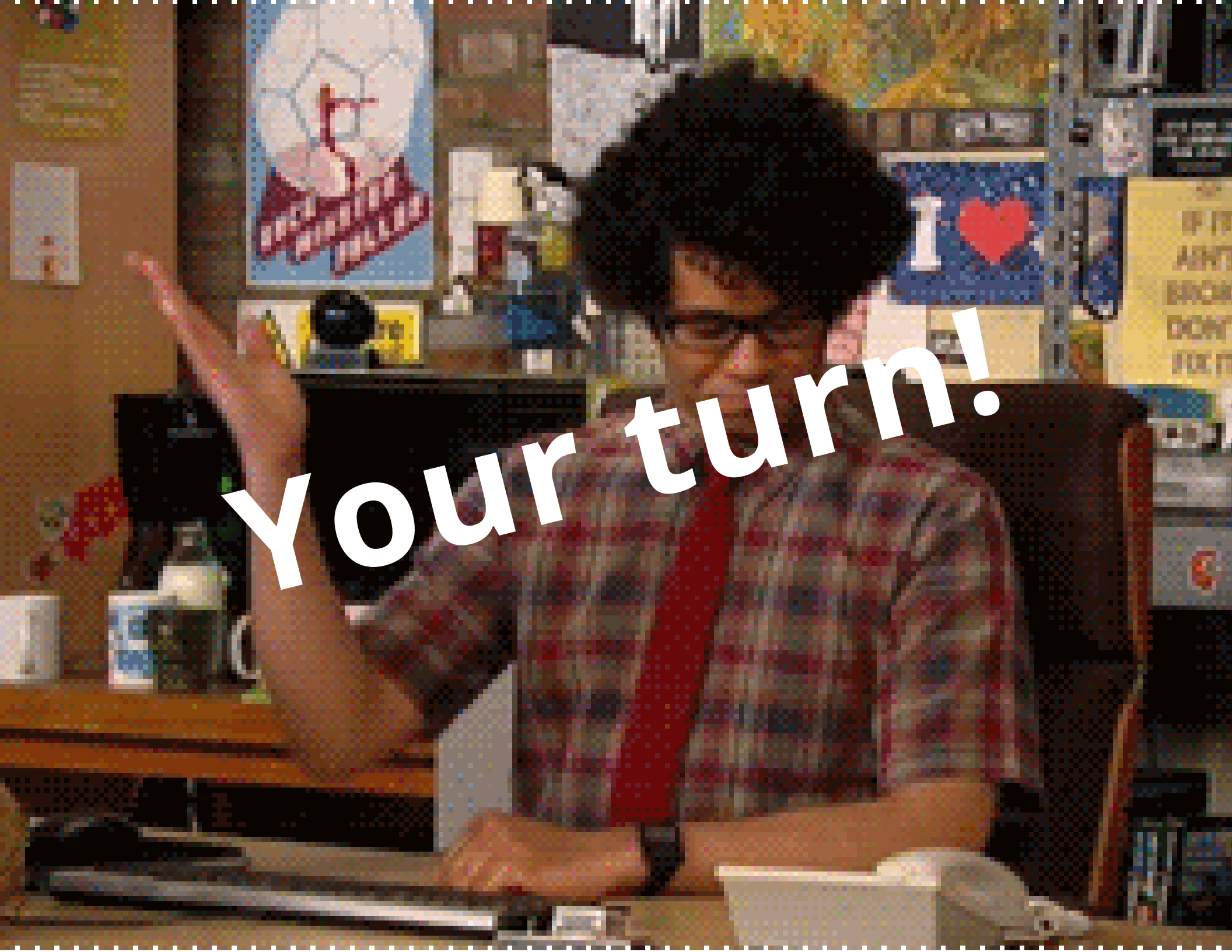
```html
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>My first Angular 2 App!</title>
  </head>
  <body>

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

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

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 →


Your turn!

Displaying Data

We can bind data to elements in HTML templates and Angular automatically updates the UI as data changes.

```
@Component({...})
@View({
  template: 'Hello {{name}}'
})
class ContactsApp {
  name:string = 'AngularJS Days';
}
```

```
@Component({...})
@View({
  template: 'Hello {{name}}'
})
class ContactsApp {
  name:string = 'AngularJS Days';
}
```

Which is the equivalent of...

```
@Component({...})
@View({
  template: 'Hello {{name}}'
})
class ContactsApp {
  name:string;
  constructor() {
    this.name = 'AngularJS Days';
  }
}
```

```
@Component({...})
@View({
  template: 'Hello {{name}}'
})
class ContactsApp {
  name:string;
  constructor() {
    this.name = 'AngularJS Days';
  }
}
```

Demo →

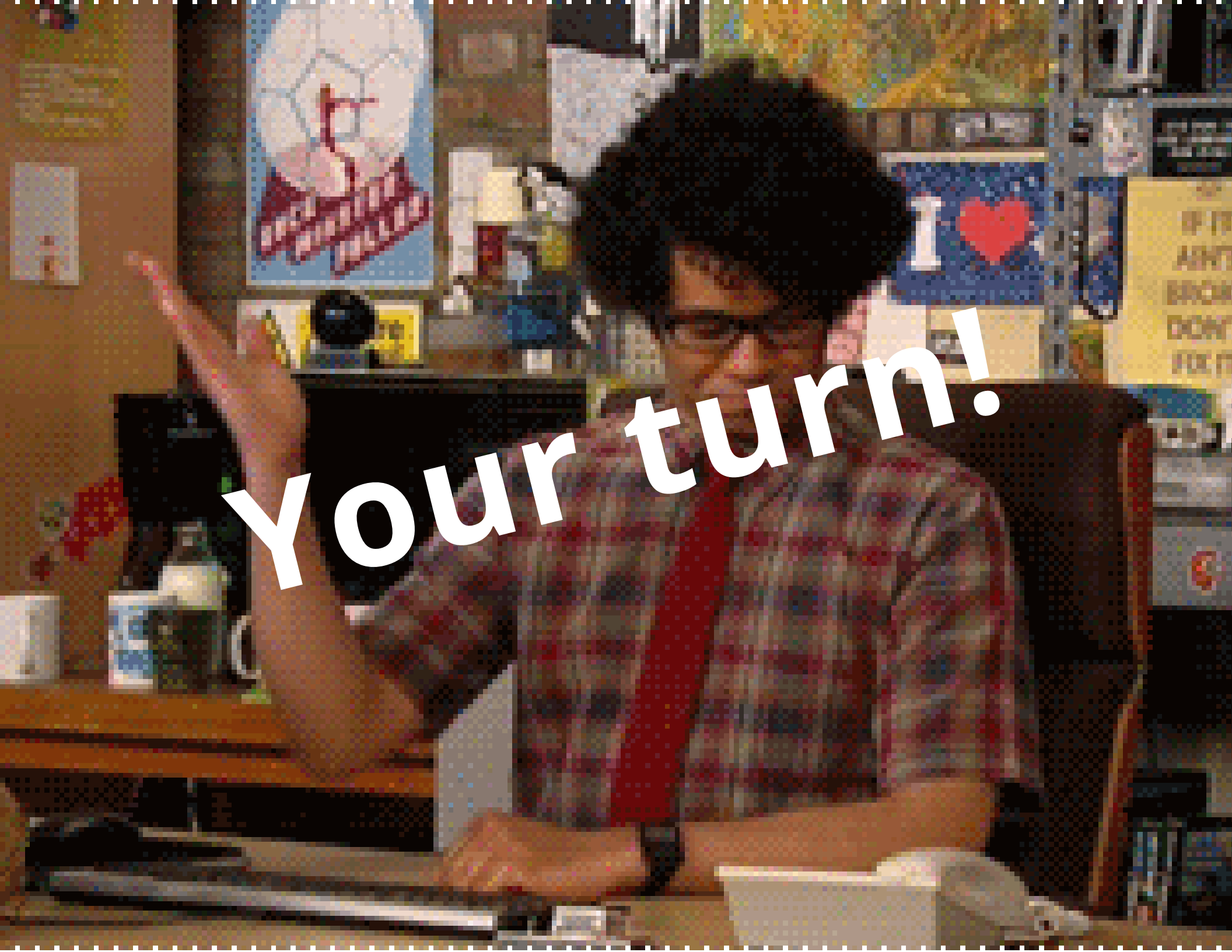
Let's take it one step further!

```
interface Contact {
  id: Number;
  firstname?: string;
  lastname?: string;
  street?: string;
  zip?: string;
  city?: string;
  image?: string;
}
```

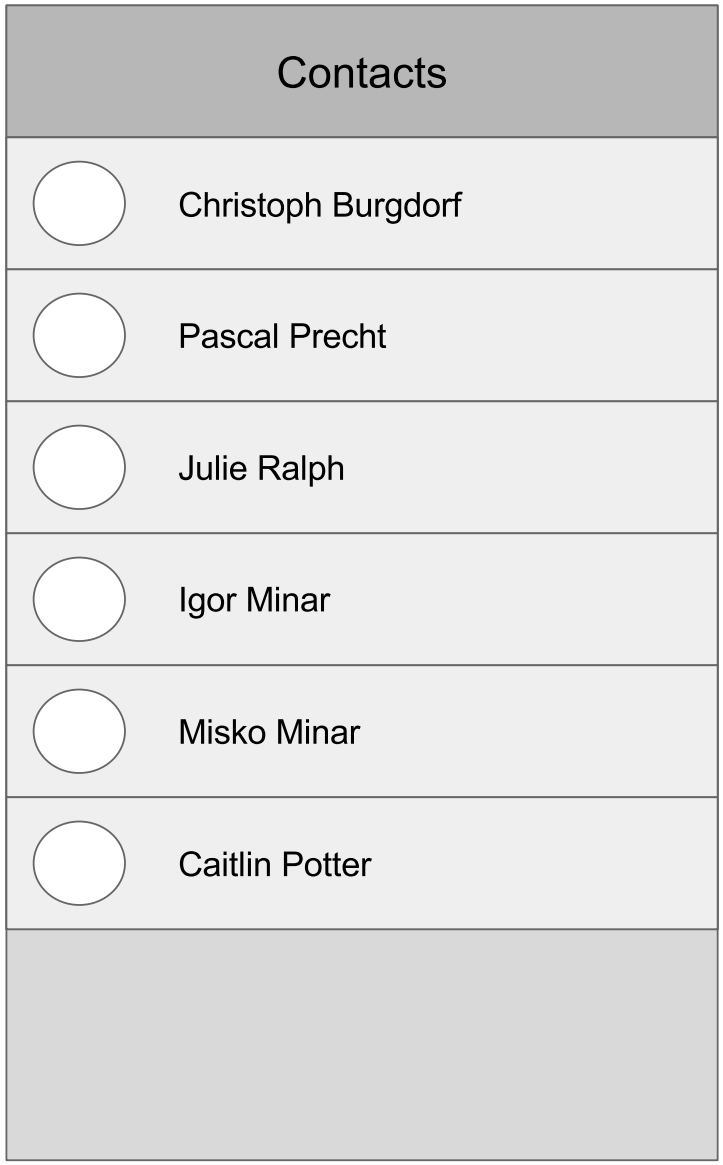
```
@Component()
@View()
class ContactsApp {
  contact: Contact = {
    id: 1,
    firstname: 'Christoph',
    lastname: 'Burgdorf',
    street: 'thoughtroad 2',
    zip: '30149',
    city: 'thoughtworld',
    image: 'path/to/image'
  }
}
```

```
@Component()
@View()
class ContactsApp {
  contact: Contact = {
    id: 1,
    firstname: 'Christoph',
    lastname: 'Burgdorf',
    street: 'thoughtroad 2',
    zip: '30149',
    city: 'thoughtworld',
    image: 'path/to/image'
  }
}
```

```
<div>
  <img [src]="contact.image">
  <span>
    {{contact.firstname}}
    {{contact.lastname}}
  </span>
</div>
```


Your turn!

Using Directives


Contacts
Christoph Burgdorf
Pascal Precht
Julie Ralph
Igor Minar
Misko Minar
Caitlin Potter

Let's make it a list!

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

```
contacts: Contact[] = [
  { id: 1, firstname: 'Christoph', ...},
  { id: 2, firstname: 'Pascal', ...},
  { id: 3, firstname: 'Misko', ...},
  { id: 4, firstname: 'Igor', ...},
  ...
];
```

Remember our component's template?

```
<div>
  <img [src]="contact.image">
  <span>
    {{contact.firstname}}
    {{contact.lastname}}
  </span>
</div>
```

```html
<ul class="collection">
  <li>
    <!-- each contact goes here -->
  </li>
</ul>
```

Iterating over iterables

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

```
@Component()
@View({

  ...
})
class ContactsApp {
  contacts: Contact[] = [
    { id: 1, firstname: 'Christoph', ...},
    ...
  ];
}
```

```
import {NgFor} from 'angular2/core';

@Component()
@View({
  directives: [NgFor],
  ...
})
class ContactsApp {
  contacts: Contact[] = [
    { id: 1, firstname: 'Christoph', ...},
    ...
  ];
}
```

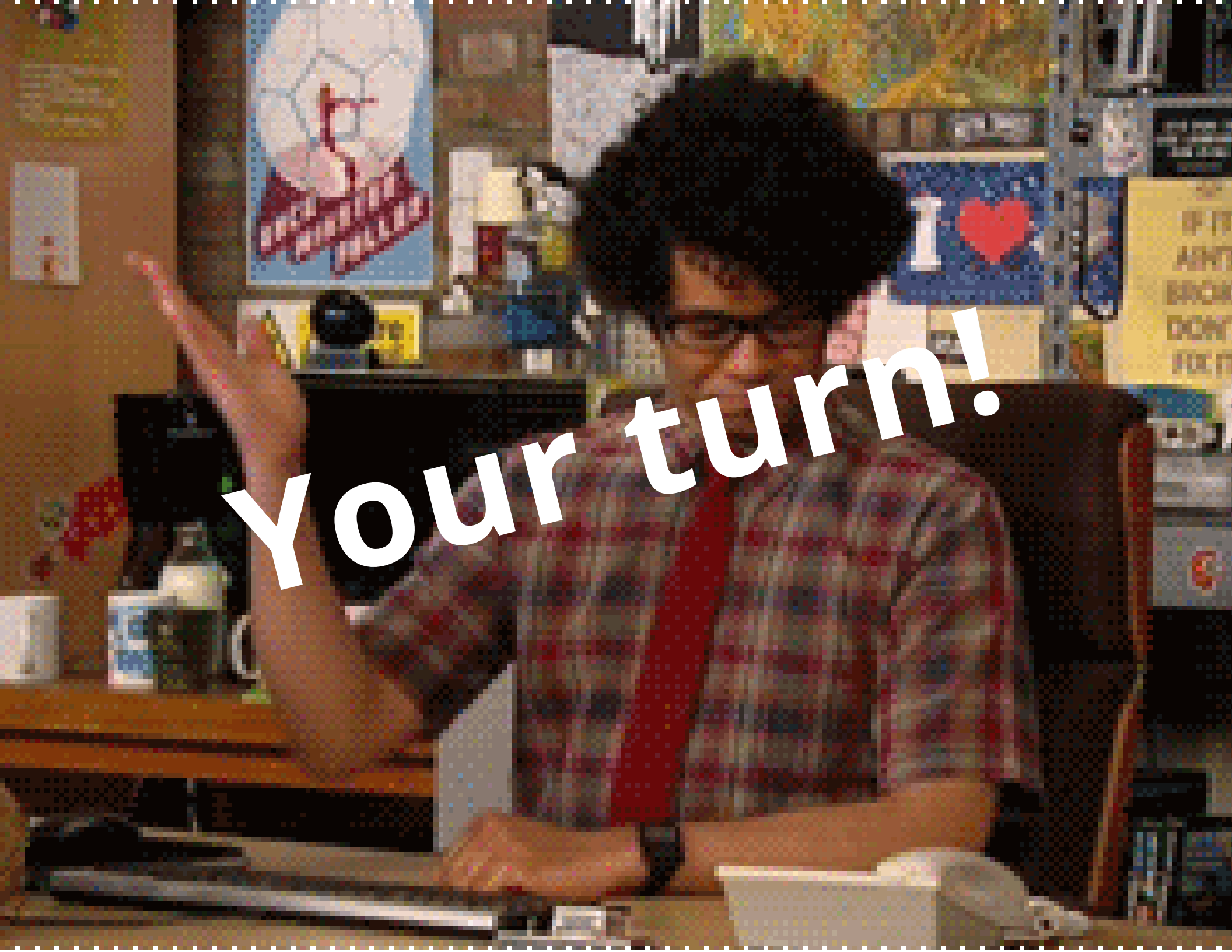
```
import {NgFor} from 'angular2/core';

@Component()
@View({
  directives: [NgFor],
  ...
})
class ContactsApp {
  contacts: Contact[] = [
    { id: 1, firstname: 'Christoph', ...},
    ...
  ];
}
```

```html
<ul class="collection">
  <li>
    <!-- each contact goes here -->
  </li>
</ul>
```

```
<ul class="collection">
  <li *ng-for="#contact in contacts">
    <!-- each contact goes here -->
  </li>
</ul>
```

```
<ul class="collection">
  <li *ng-for="#contact in contacts">
    <!-- each contact goes here -->
  </li>
</ul>
```

Demo →


Your turn!

Services and Dependency Injection

Services

Services in Angular 2 are simpy **ES2015 classes**.

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

class ContactsService {

  private contacts: Contact[] = [
    { id: 1, firstname: 'Christoph', ...},
    { id: 2, firstname: 'Pascal', ...},
    { id: 3, firstname: 'Misko', ...},
  ];

  getContacts() { ... }
}
```

```
@Component()
@View()
class ContactsApp {

  contacts: Contact[];

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

```
@Component()
@View()
class ContactsApp {

  contacts: Contact[];

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

```
@Component()
@View()
class ContactsApp {

  contacts: Contact[];

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

But how do we get there?

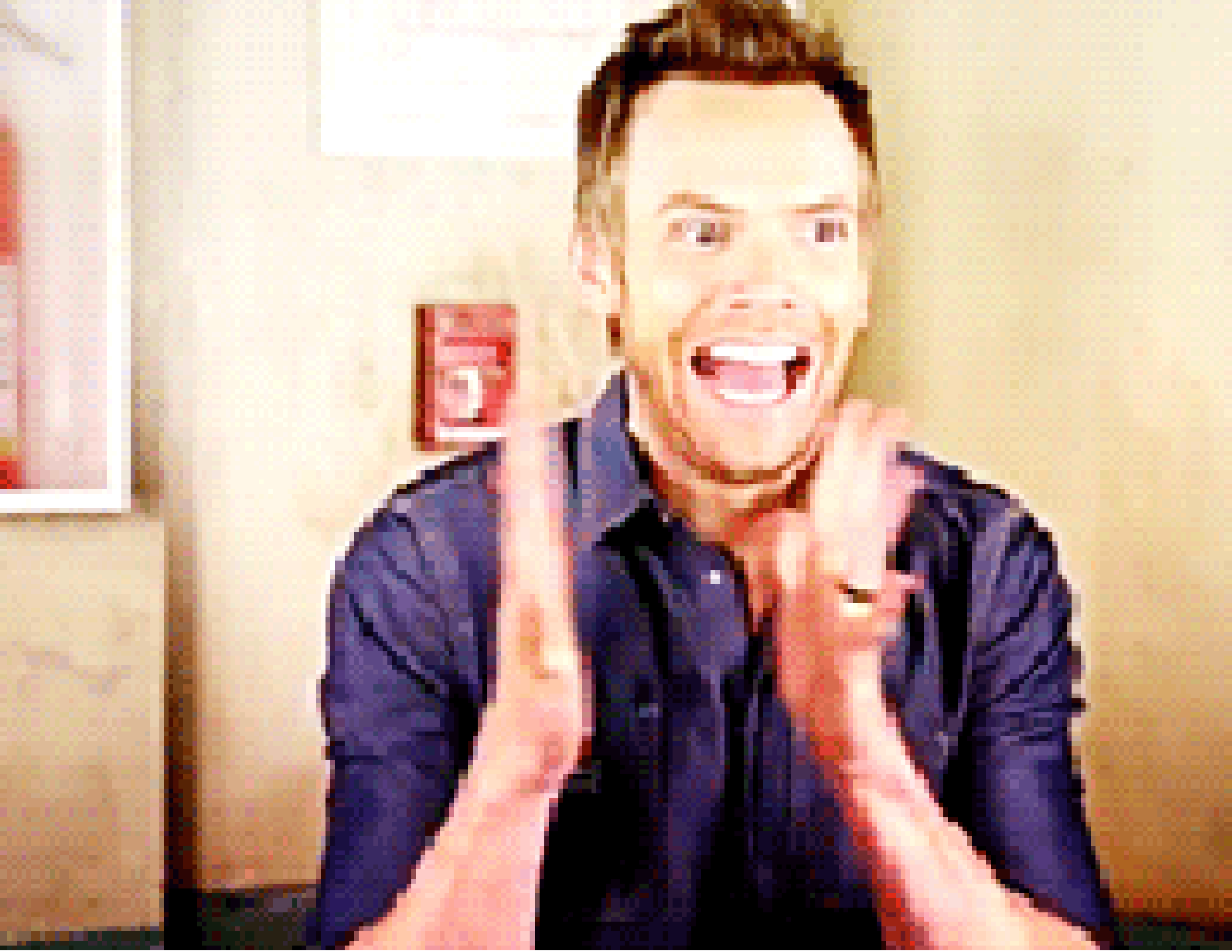
Configuring Injectors

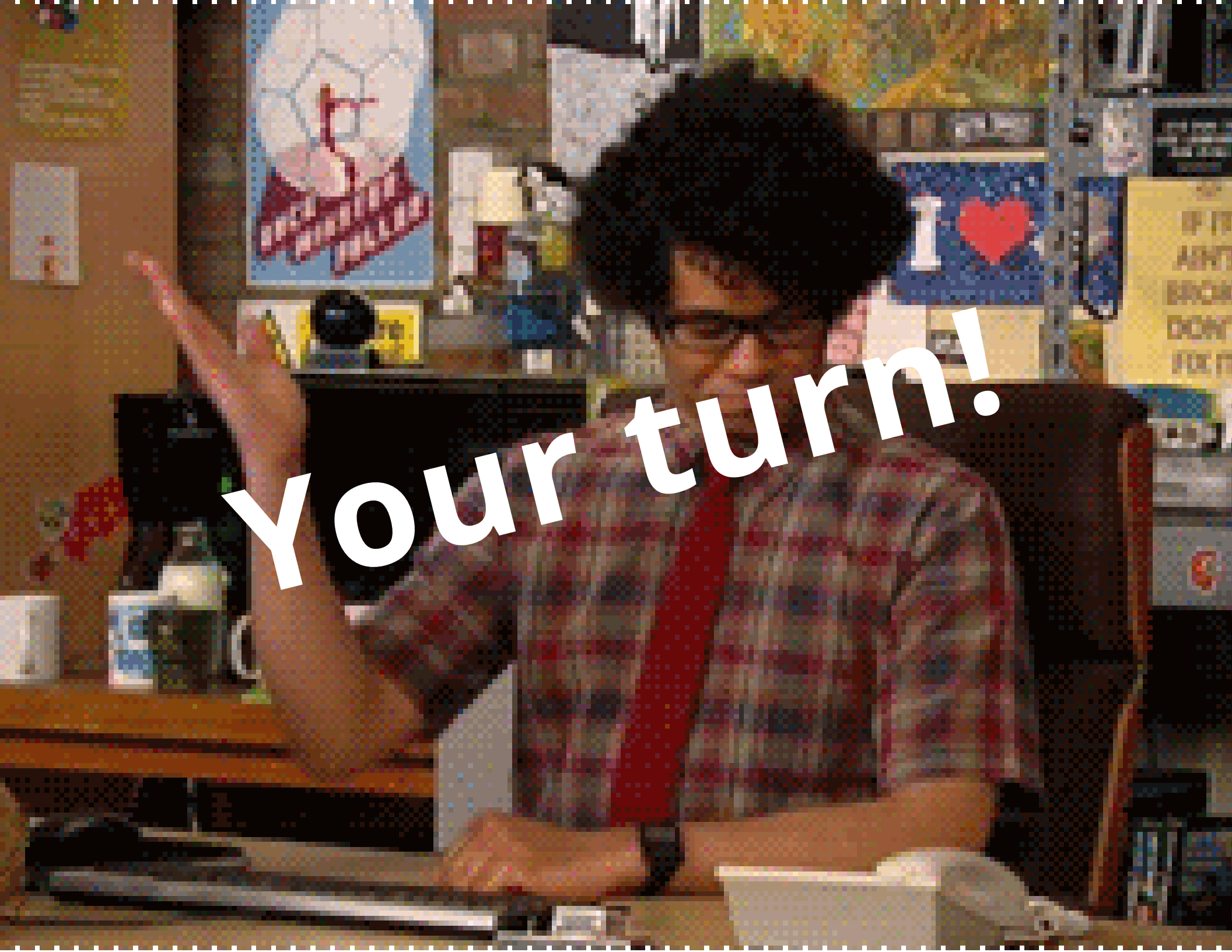
We can configure the root component's injector to make any service available for DI.

```
import {Component, View} from 'angular2/core';
import {bootstrap} from 'angular2/core';
import {ContactsService} from './common/contacts-service';

@Component()
@View()
class ContactsApp {
  ...
}

bootstrap(ContactsApp);
```

```
import {Component, View} from 'angular2/core';
import {bootstrap} from 'angular2/core';
import {ContactsService} from './common/contacts-service';

@Component()
@View()
class ContactsApp {
  ...
}

bootstrap(ContactsApp, [ContactsService]);
```

```
import {Component, View} from 'angular2/core';
import {bootstrap} from 'angular2/core';
import {ContactsService} from './common/contacts-service';

@Component()
@View()
class ContactsApp {
  ...
}

bootstrap(ContactsApp, [ContactsService]);
```

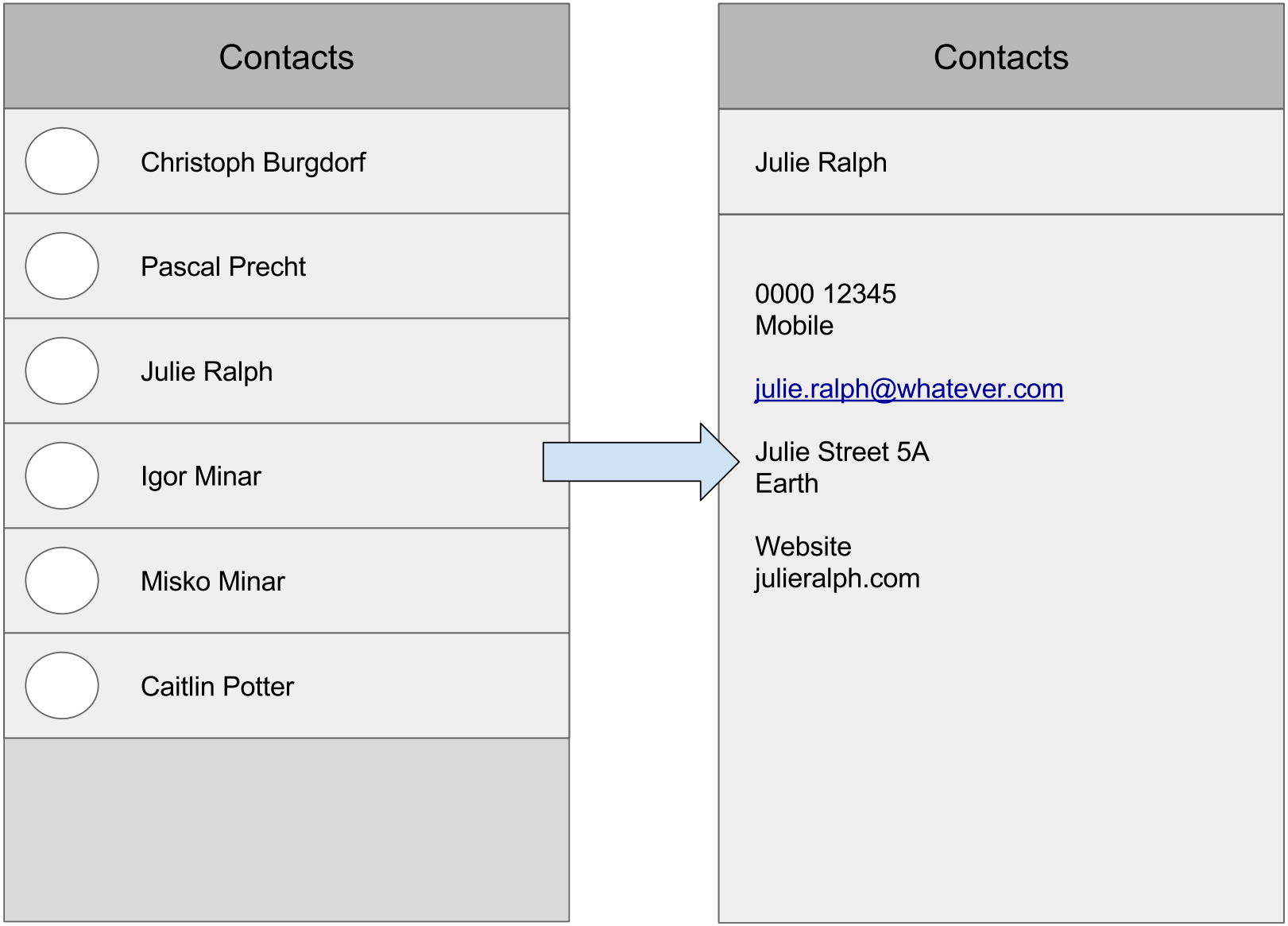



Your turn!

Component Routing

Contacts

- Christoph Burgdorf
- Pascal Precht
- Julie Ralph
- Igor Minar
- Misko Minar
- Caitlin Potter


Contacts
Christoph Burgdorf
Pascal Precht
Julie Ralph
Igor Minar
Misko Minar
Caitlin Potter
Contacts
Julie Ralph
0000 12345
Mobile
julie.ralph@whatever.com
Julie Street 5A
Earth
Website
julieralph.com

What components do we have?

It turns out that our application consists of three components:

- **ContactsApp** - The root component that is being bootstrapped
- **ContactsList** - A component to list contacts by provided data
- **ContactDetail** - A component to show a contact's details

Routing

In order to make routing in Angular 2 work, the router module provides the following components:

- `RouteConfig` - Decorator to statically configure routes
- `ROUTER_BINDINGS` - Provider to inject `Router` instance
- `ROUTER_DIRECTIVES` - Provider to make router related directives available

Configuring Routes

We use the `@RouteConfig` decorator to configure routes for our application on the root component.

```
@Component()
@View()



class ContactsApp {
  ...
}
bootstrap(ContactsApp, [ContactsService]);
```

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



@Component()
@View()
@RouteConfig([
  // route definitions go here
])
class ContactsApp {
  ...
}
bootstrap(ContactsApp, [ContactsService]);
```

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



@Component()
@View()
@RouteConfig([
  // route definitions go here
])
class ContactsApp {
  ...
}
bootstrap(ContactsApp, [ContactsService]);
```

Route Definitions

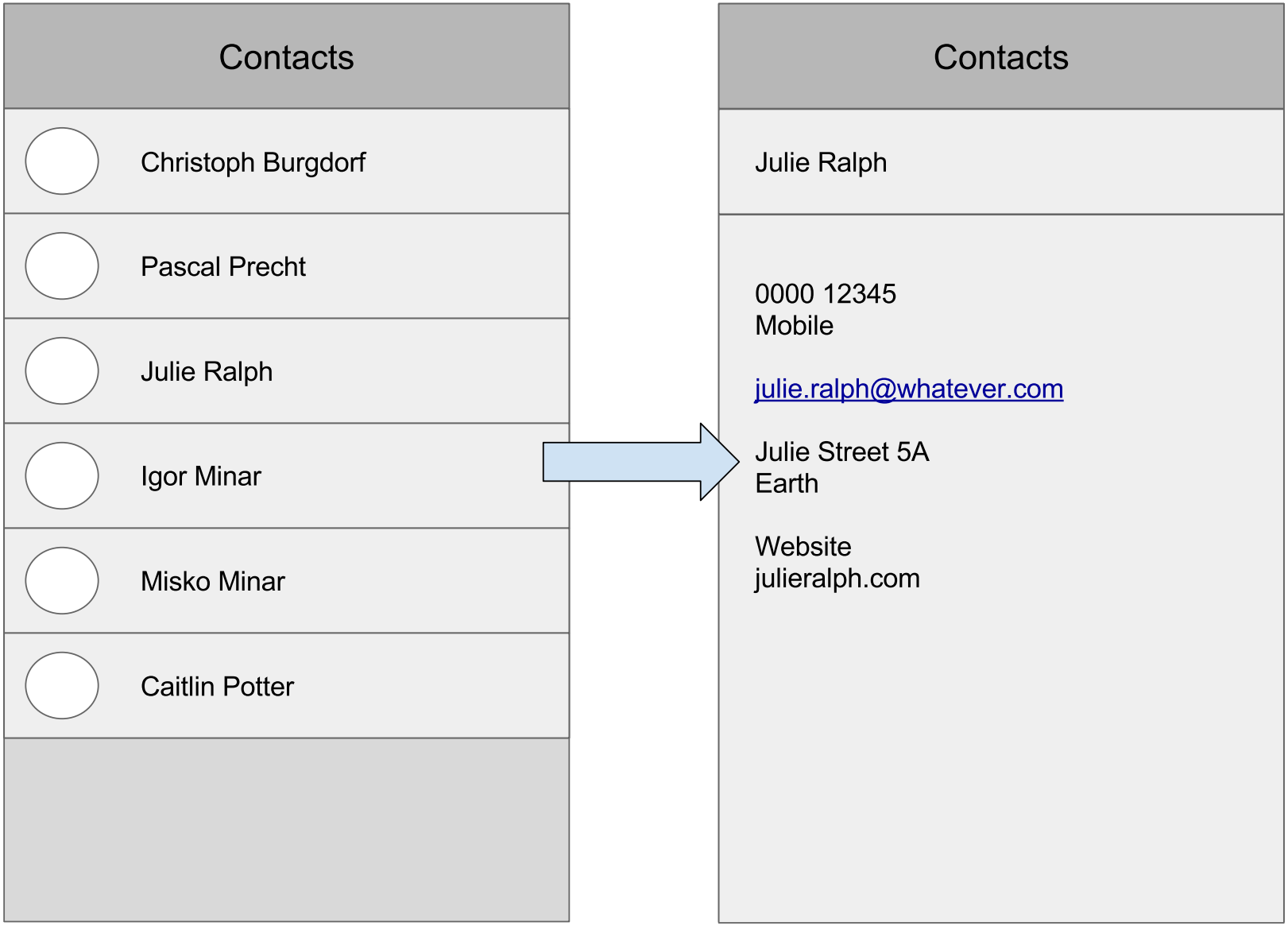
A `RouteDefinition` has a `path`, a `component`, and an optional alias.

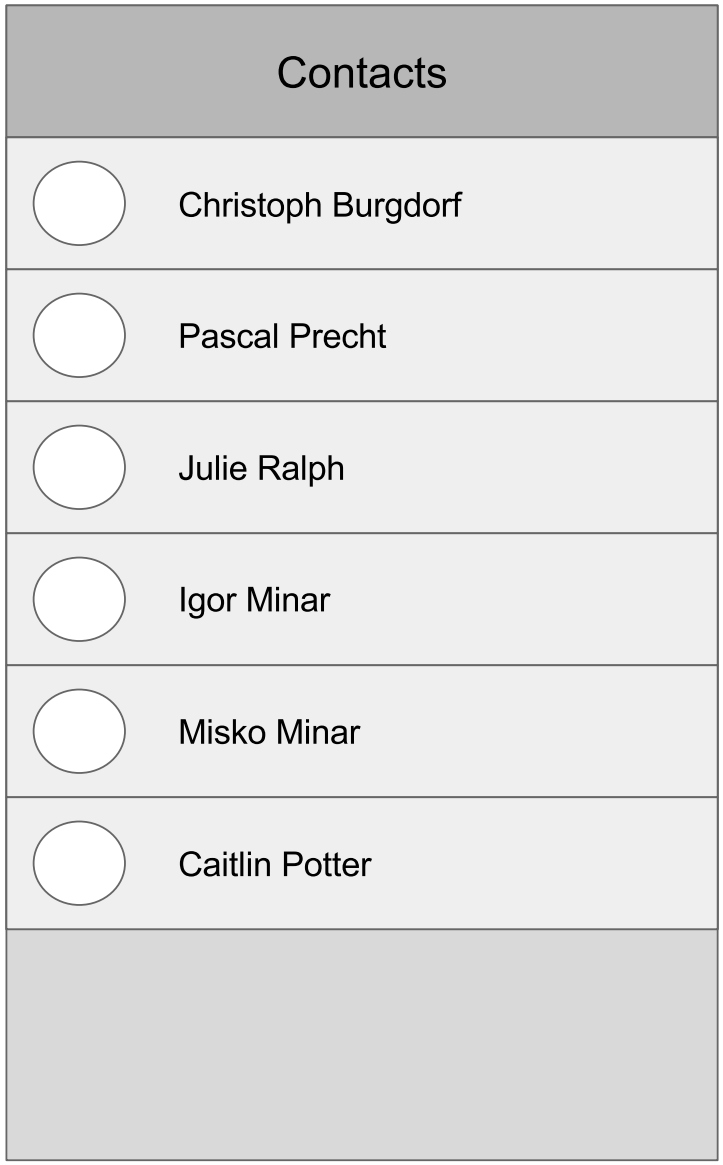
```
import {RouteConfig} from '...';



@Component()
@View()
@RouteConfig([
  // route definitions go here
])
class ContactsApp {
  ...
}
bootstrap(ContactsApp, [ContactsService]);
```

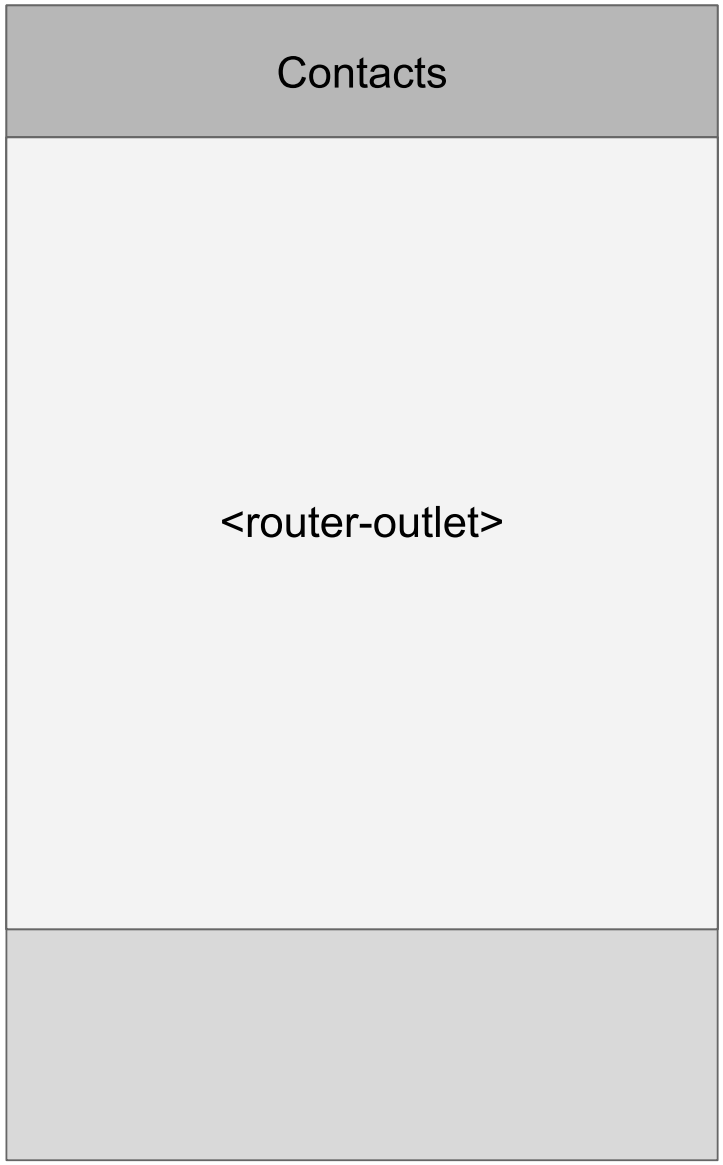
```
import {RouteConfig} from '...';
import {ContactsList} from '...';

@Component()
@View()
@RouteConfig([
  { path: '/', component: ContactsList, as: 'ContactList' }
])
class ContactsApp {
  ...
}
bootstrap(ContactsApp, [ContactsService]);
```

Displaying components

The component router comes with a `<router-outlet>` directive, to specify a **viewport** where components should be loaded into.


Contacts
Christoph Burgdorf
Pascal Precht
Julie Ralph
Igor Minar
Misko Minar
Caitlin Potter
Contacts
Julie Ralph
0000 12345
Mobile
julie.ralph@whatever.com
Julie Street 5A
Earth
Website
julieralph.com


Contacts
Christoph Burgdorf
Pascal Precht
Julie Ralph
Igor Minar
Misko Minar
Caitlin Potter


Contacts
<router-outlet>

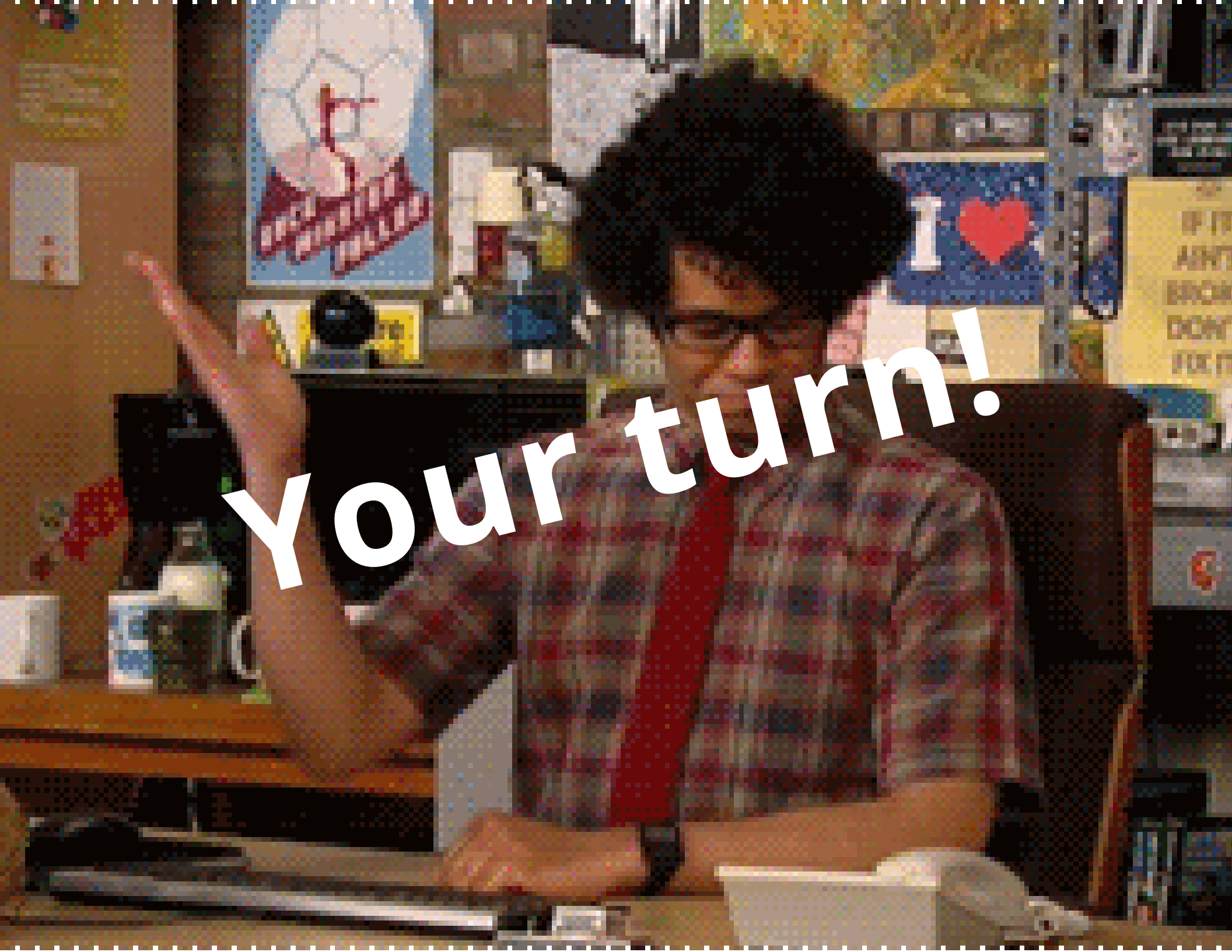
```
@Component()
@View({



})
@RouteConfig()
class ContactsApp {
  ...
}
```

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

@Component()
@View({
  directives: [ROUTER_DIRECTIVES],
  template: '<router-outlet></router-outlet>'
})
@RouteConfig()
class ContactsApp {
  ...
}
```

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

@Component()
@View({
  directives: [ROUTER_DIRECTIVES],
  template: '<router-outlet></router-outlet>'
})
@RouteConfig()
class ContactsApp {
  ...
}
```

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

@Component()
@View({
  directives: [ROUTER_DIRECTIVES],
  template: '<router-outlet></router-outlet>'
})
@RouteConfig()
class ContactsApp {
  ...
}
```

[DEMO]


Your turn!

Linking to other components

The `router-link` directive can be used to declaratively link to a specific part of our application **using a DSL**.

```
@Component()
@View()
@RouteConfig([
  { path: '/', component: ContactsList, as: 'ContactList' }





])
class ContactsApp {
  ...
}
```

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

@Component()
@View()
@RouteConfig([
  { path: '/', component: ContactsList, as: 'ContactList' },
  {
    path: '/contact/:id',
    component: ContactDetail,
    as: 'ContactDetail'
  }
])
class ContactsApp {
  ...
}
```

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

@Component()
@View()
@RouteConfig([
  { path: '/', component: ContactsList, as: 'ContactList' },
  {
    path: '/contact/:id',
    component: ContactDetail,
    as: 'ContactDetail'
  }
])
class ContactsApp {
  ...
}
```

```
<ul class="collection">
  <li *ng-for="#contact in contacts">

      <img [src]="contact.image">
      <span>
        {{contact.firstname}}
        {{contact.lastname}}
      </span>

  </li>
</ul>
```

```
<ul class="collection">
  <li *ng-for="#contact in contacts">
    <a [router-link]="['/ContactDetail', { id: contact.id }]">
      <img [src]="contact.image">
      <span>
        {{contact.firstname}}
        {{contact.lastname}}
      </span>
    </a>
  </li>
</ul>
```

```
<ul class="collection">
  <li *ng-for="#contact in contacts">
    <a [router-link]="['/ContactDetail', { id: contact.id }]">
      <img [src]="contact.image">
      <span>
        {{contact.firstname}}
        {{contact.lastname}}
      </span>
    </a>
  </li>
</ul≥
```

```
@Component({selector: 'contact-detail'})
@View({
  templateUrl: 'contact-detail.html'
})
export class ContactDetail {

  contact:Contact;

  constructor() {

  }
}
```

How do we get access to route params?

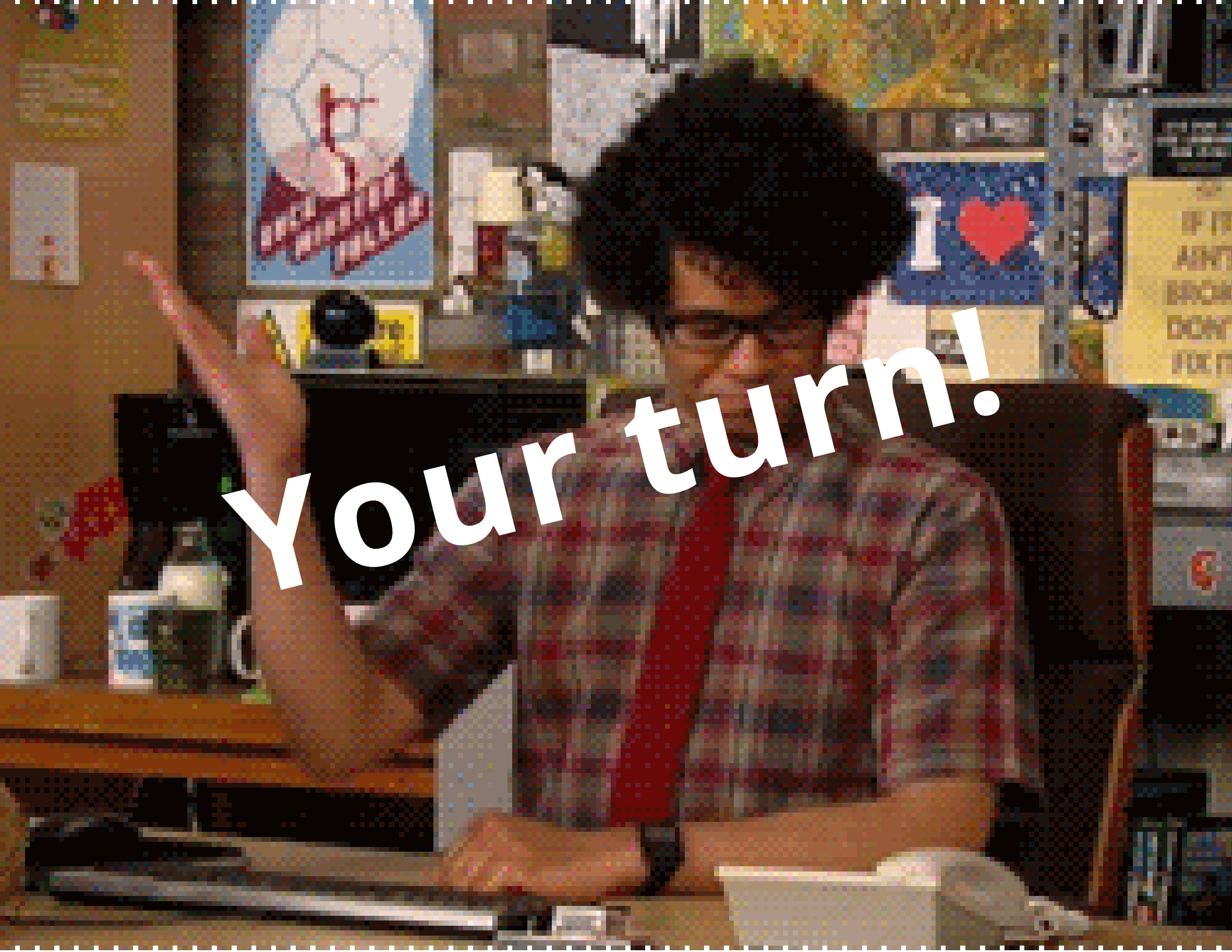
```
@Component({selector: 'contact-detail'})
@View({
  templateUrl: 'contact-detail.html'
})
export class ContactDetail {

  contact:Contact;

  constructor() {

  }
}
```

```
import {RouteParams} from 'angular2/router';

@Component({selector: 'contact-detail'})
@View({
  templateUrl: 'contact-detail.html'
})
export class ContactDetail {

  contact:Contact;

  constructor(params:RouteParams, contactsService:ContactsService) {
    this.contact = contactsService.getContact(params.get('id'));
  }
}
```

```
import {RouteParams} from 'angular2/router';

@Component({selector: 'contact-detail'})
@View({
  templateUrl: 'contact-detail.html'
})
export class ContactDetail {

  contact:Contact;

  constructor(params:RouteParams, contactsService:ContactsService) {
    this.contact = contactsService.getContact(params.get('id'));
  }
}
```

```
import {RouteParams} from 'angular2/router';

@Component({selector: 'contact-detail'})
@View({
  templateUrl: 'contact-detail.html'
})
export class ContactDetail {

  contact:Contact;

  constructor(params:RouteParams, contactsService:ContactsService) {
    this.contact = contactsService.getContact(params.get('id'));
  }
}
```

```
<div>
  <span>{{contact.firstname}} {{contact.lastname}}</span>
  <span>{{contact.street}}</span>
  <span>{{contact.zip}} {{contact.city}}</span>
  <span>{{contact.country}}</span>
</div>
```


Your turn!

Demo App: github.com/thoughtram/ng2-contacts-demo

THOUGHTRAM

EXTEND YOUR MEMORY