# ANGULAR 2

by Google

# Acknowledgement

- Some slides are based on

‘Angular 2 Development with TypeScript’, Yakov Fain and Anton Moiseev, ISBN 9781617293122

https://www.manning.com/books/angular-2-development-with-typescript

# Outline

- What is Angular and why should you care!
- Single Page Application (SPA)
- Angular Architecture
- Angular Key Components
  - Components
  - Directives
  - Two way binding
  - Routing and views
  - Ajax with $http

# What is ANGULAR$^2$ by Google ?

- SPA framework for **efficiently** creating dynamic views in a web browser (using “HTML” and JavaScript)
  - It is a **client-side View engine** (**template engine**) that generates HTML views from an html template containing place holders that will be replaced by dynamic content
- Some highlights/focuses:
  - **Complete application framework** for building Single Page Application (SPA)
  - Open Source, Comprehensive and Forward Thinking
  - Popularity, Google and a large community behind it
    - **Google is paying developers to actively develop Angular**

# Angular #1?

- Angular appears to be winning the JavaScript framework battle

(and for good reason)

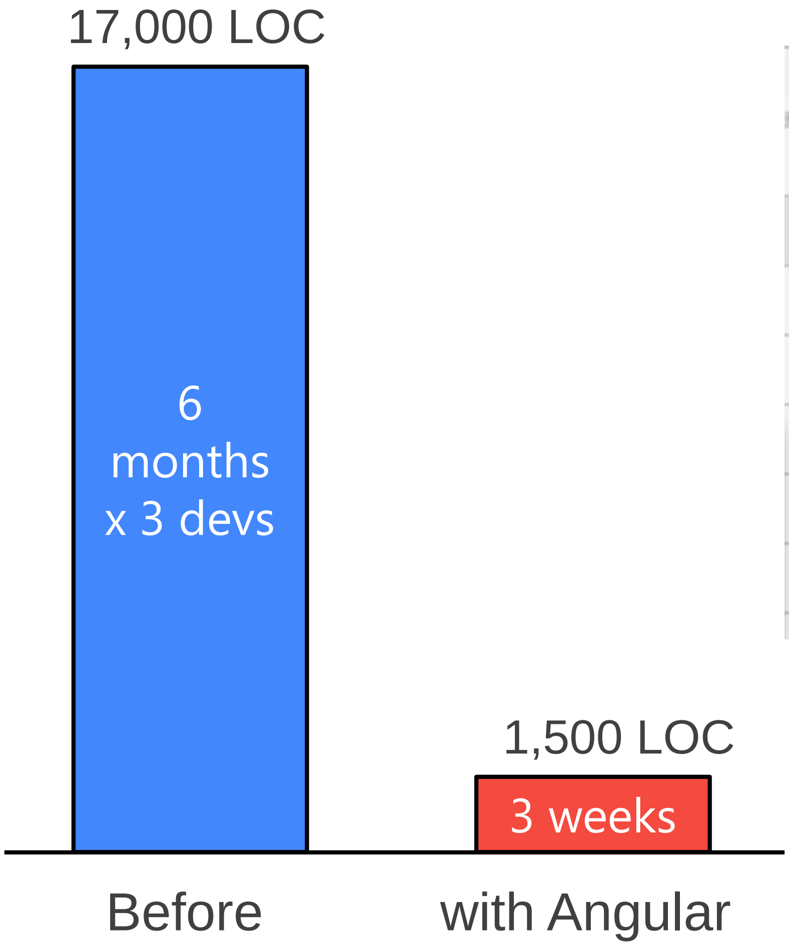
- You will see yourself…!

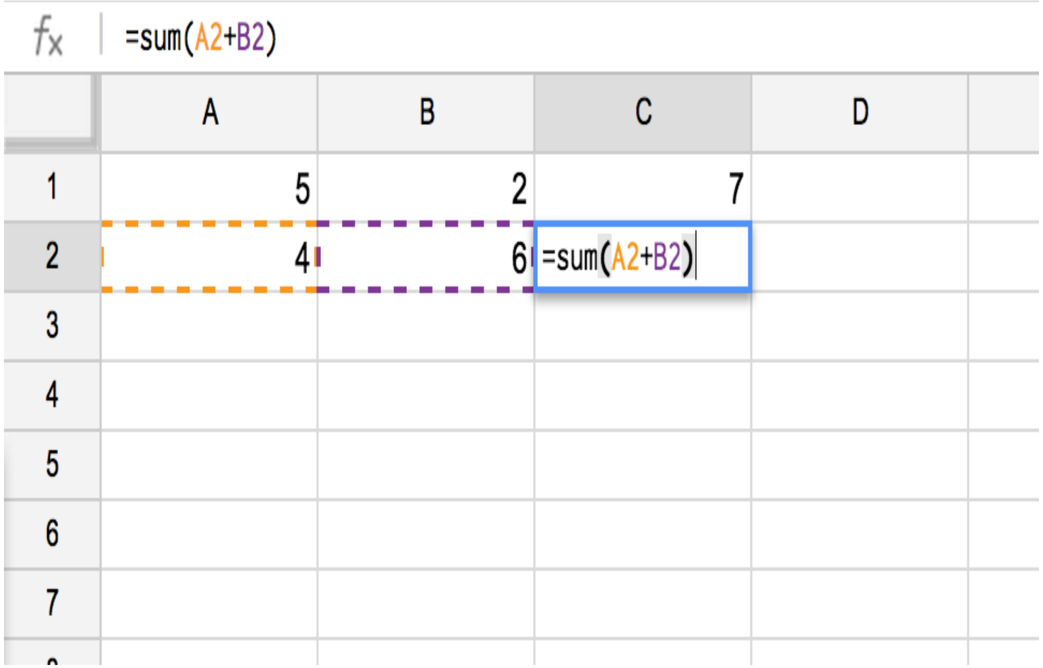
~~Direct DOM Manipulation~~

~~<form> </form> To Submit to server~~

{ ALL JSON }

# Google feedback




fx =sum(A2+B2)

|   | A | B | C | D |
|---|---|---|---|---|
| 1 | 5 | 2 | 7 |  |
| 2 | 4 | 6 | =sum(A2+B2) |  |
| 3 |  |  |  |  |
| 4 |  |  |  |  |
| 5 |  |  |  |  |
| 6 |  |  |  |  |
| 7 |  |  |  |  |

# jQuery

- Allows for DOM Manipulation
- Common API across multiple browsers
- Does not provide structure to your code
- Does not allow for two way binding

# Hello JQuery

```
<p id="greeting2"></p>
 <script>
$(function(){
  $('#greeting2').text('Hello World!');
});
</script>
```

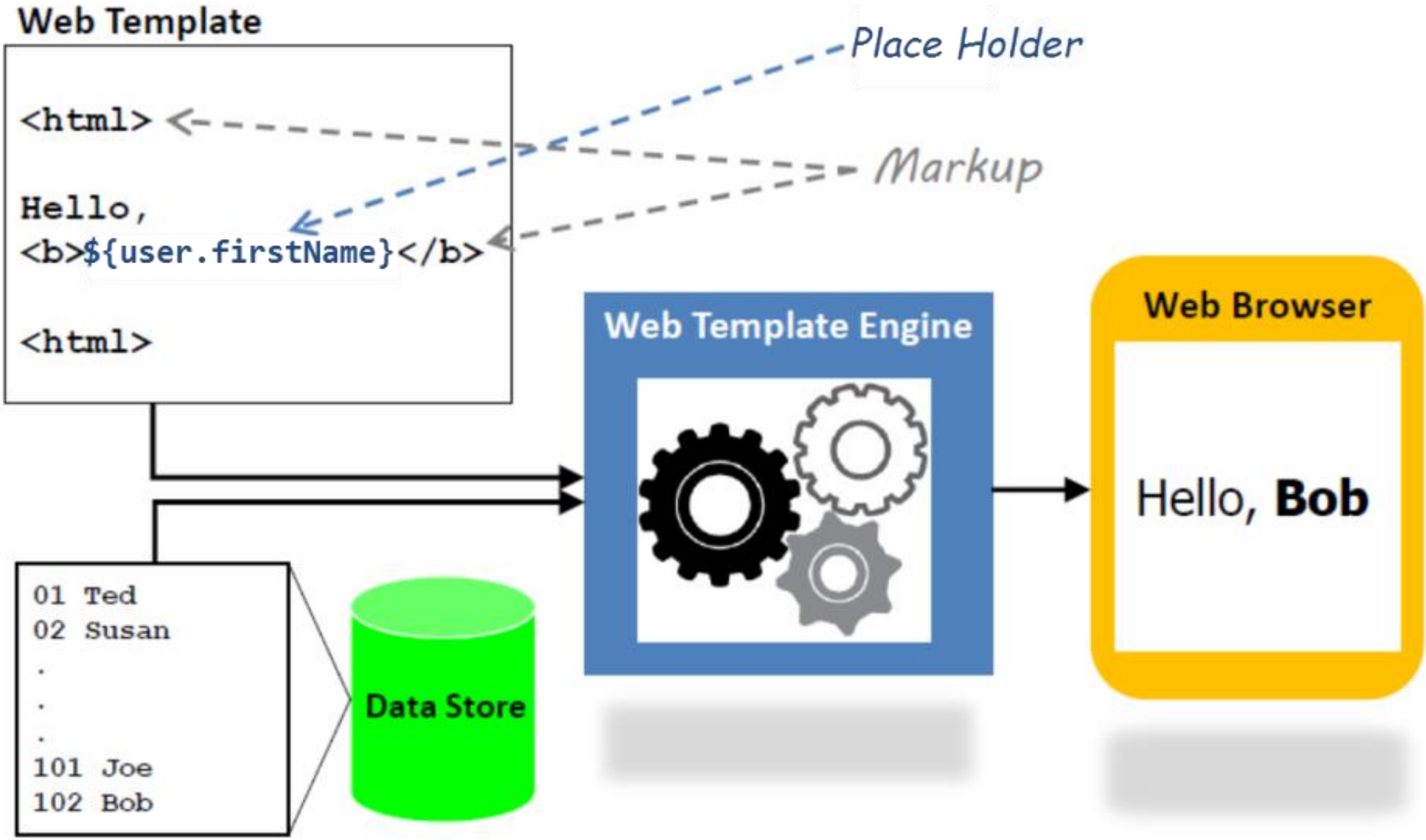
# Hello Angular

```
<p>{{greeting}}</p>
```

# Single Page Application (SPA)

Back

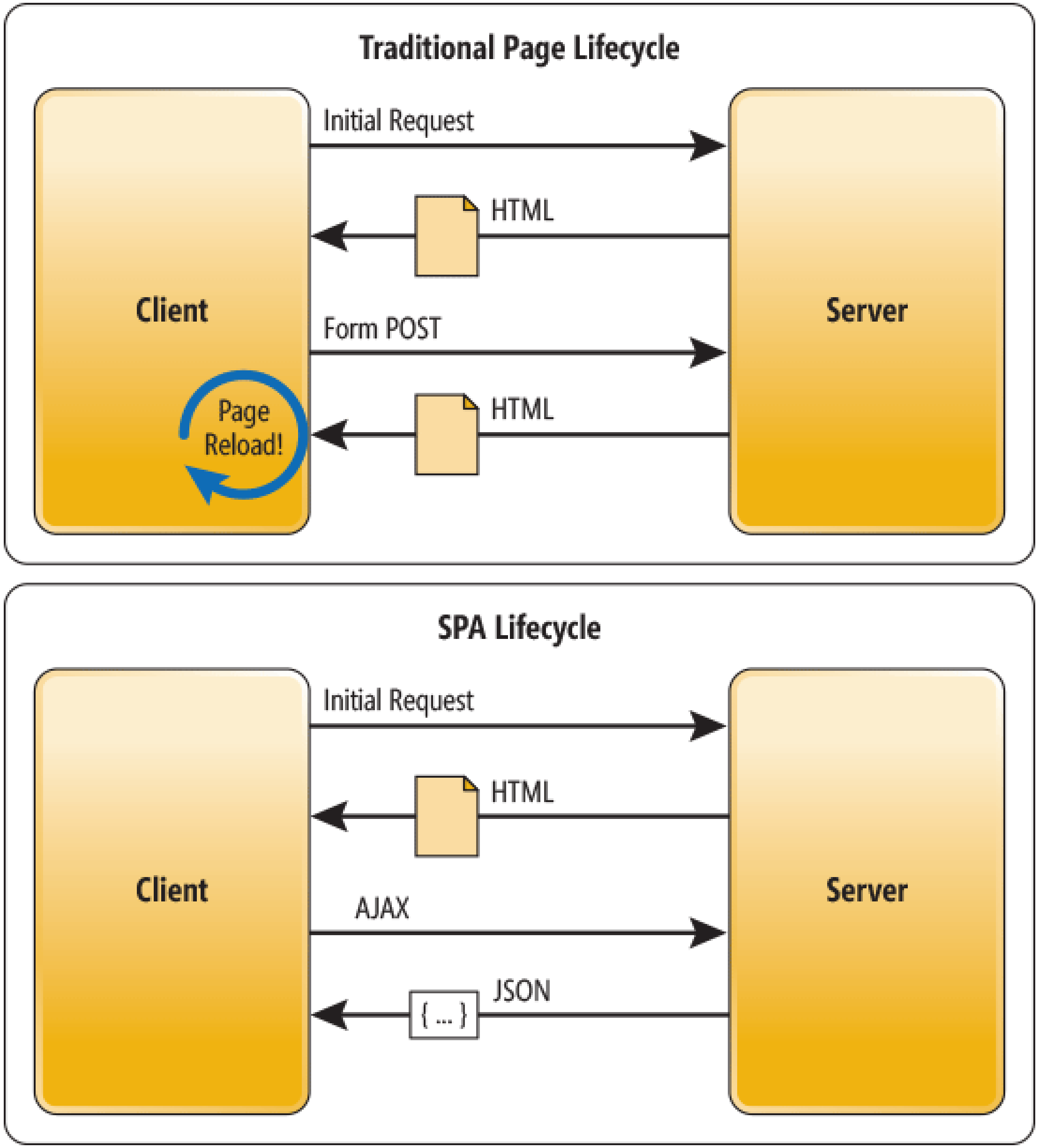
# Traditional Architecture



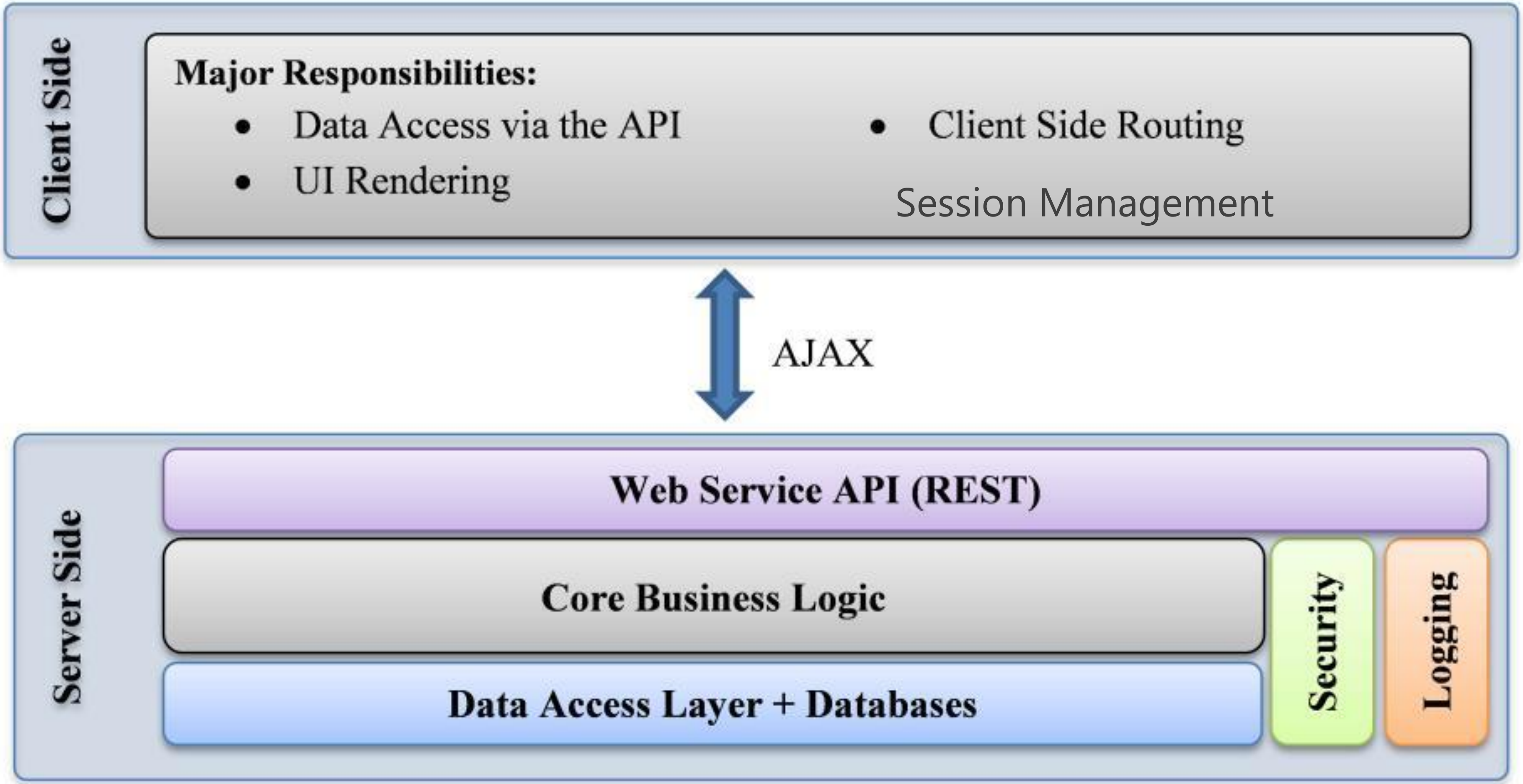
```
<html>

Hello,
<b>${user.firstName}</b>

<html>
```

```
01 Ted
02 Susan
.
.
.
101 Joe
102 Bob
```

# Traditional vs. SPA Lifecycle



# Role of Client and Server in SPA



Client Side

**Major Responsibilities:**

- Data Access via the API
- UI Rendering
- Client Side Routing

Session Management

AJAX

Server Side

**Web Service API (REST)**

**Core Business Logic**

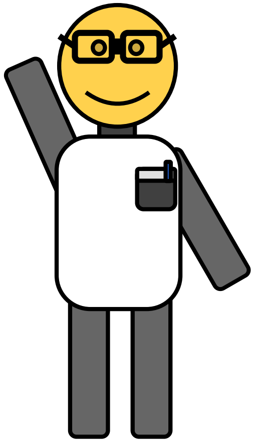
**Data Access Layer + Databases**

**Security**

**Logging**

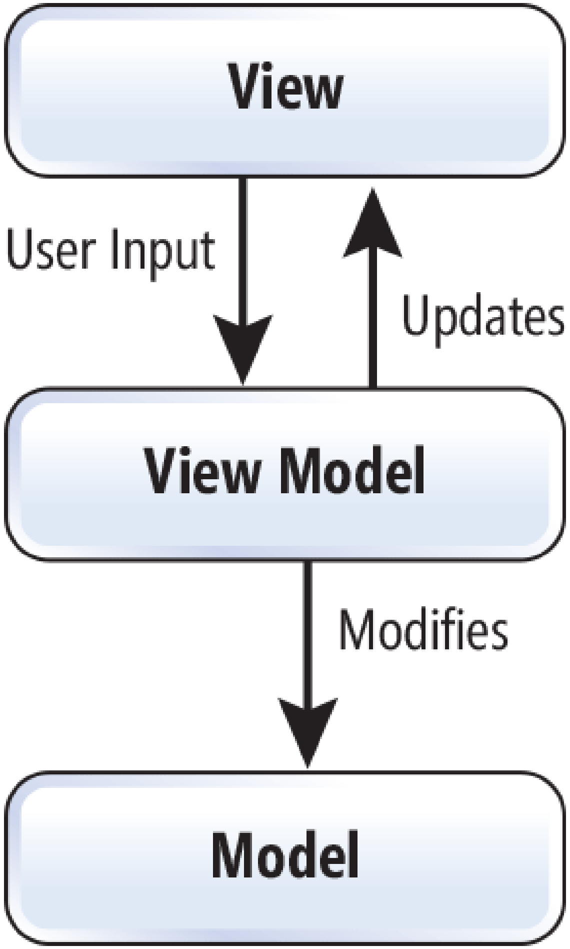
# Benefits of a Single Page App

- **State maintained on client + offline support**
  - Can use HTML5 JavaScript APIs to store state in the browser's localStorage
- **Better User experience**
- More interactive and responsive
- Less network activity and waiting
- Developer experience
  - Better (if you use a framework!)
  - No constant DOM refresh
  - Rely on a 'thick' client for caching etc.

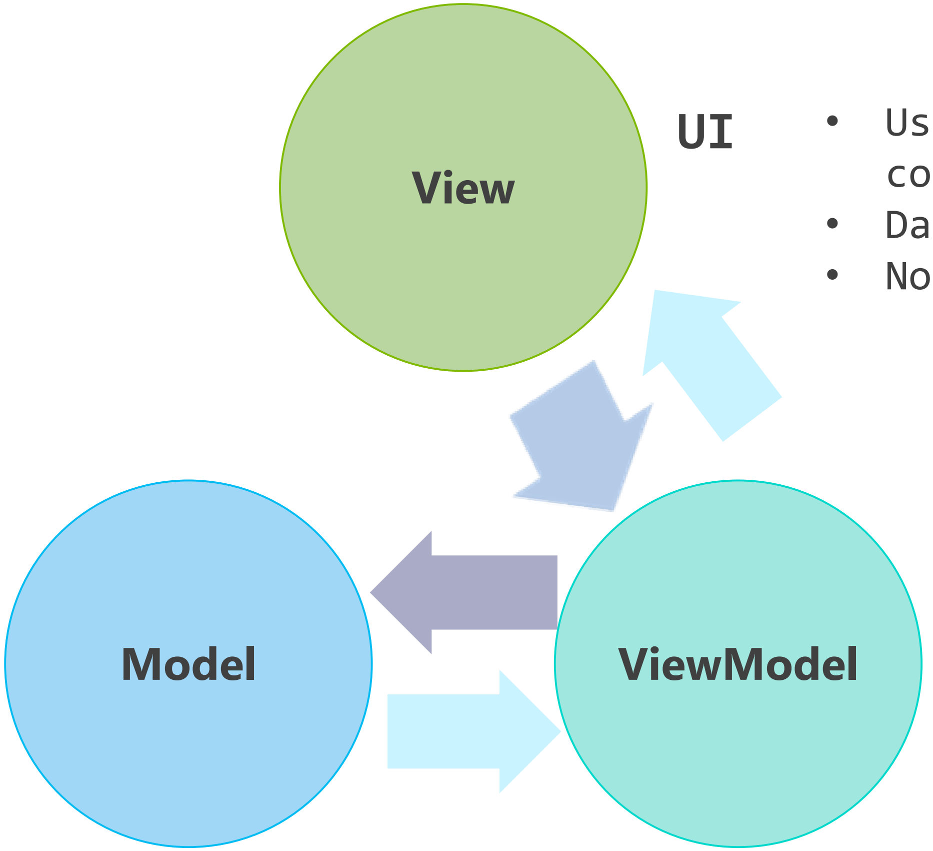


# SPA uses MVVM

- A more recent variant of MVC is the MVVM pattern:
  - The model still represents the domain data
  - The View Model is an abstract representation of the view
  - The View displays the View Model and sends user input to the View Model
  - View Model reads/modifies the server-side Model



# MVVM: Model View ViewModel



UI

- User actions, commands
- Data binding
- Notifications

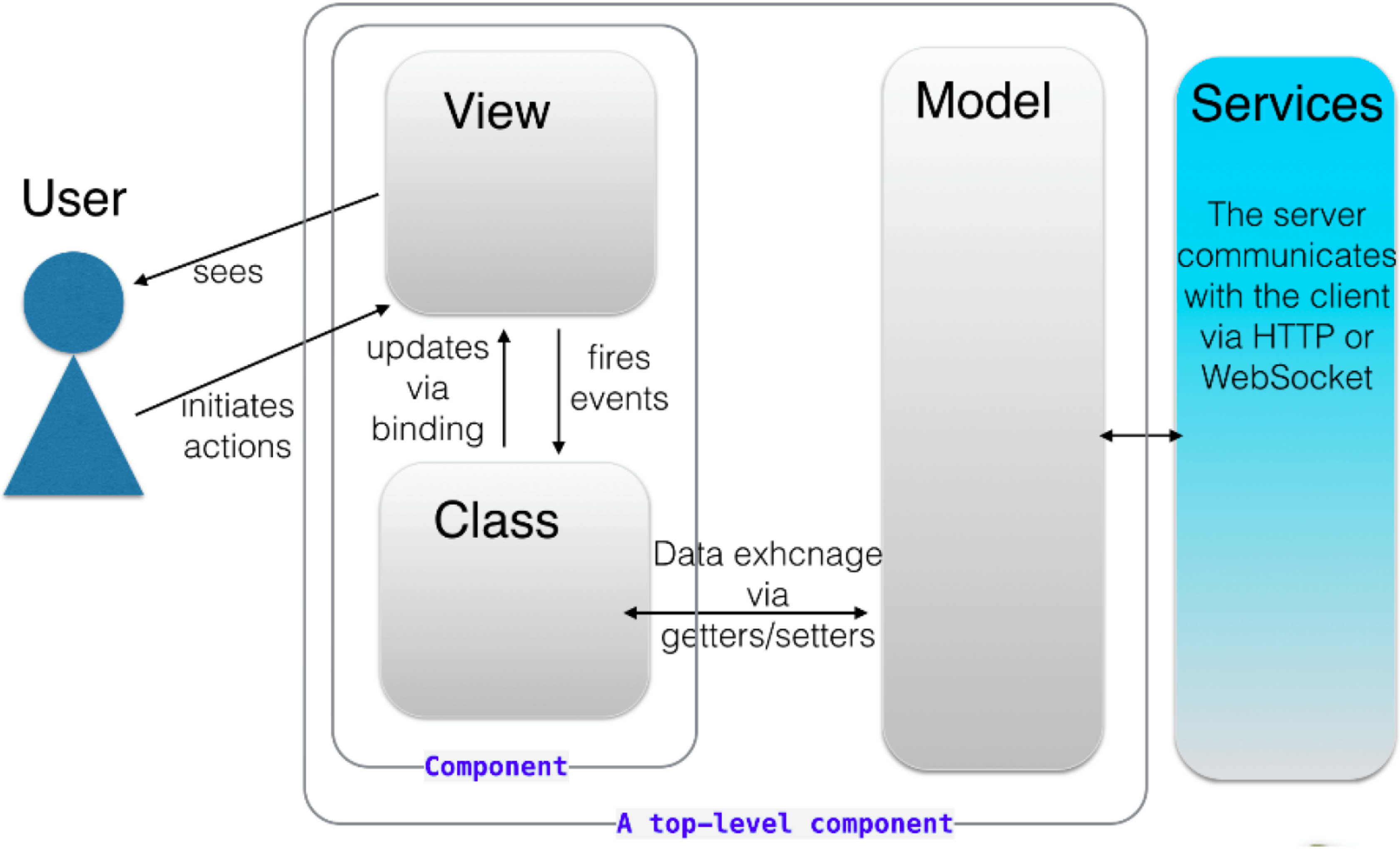
Business Logic and Data

- Data Access
- Update ViewModel about change

Presentation Logic

# Angular App Architecture

# Architecture of an Angular app



Angular App is composed of **components**

# Angular Architecture Highlights

- An Angular component is a centerpiece of the new architecture.
- A Component is a TypeScript class annotated with **@Component** annotation, it specifies:
  - a **selector** declaring the name of the custom tag to be used to load to component in HTML document
  - the **template** (=an HTML fragment with data binding expressions to render by the view) or **templateURL**
  - **directives** property specifies in required any other components the view depends
- Event handlers are implemented as methods of the class

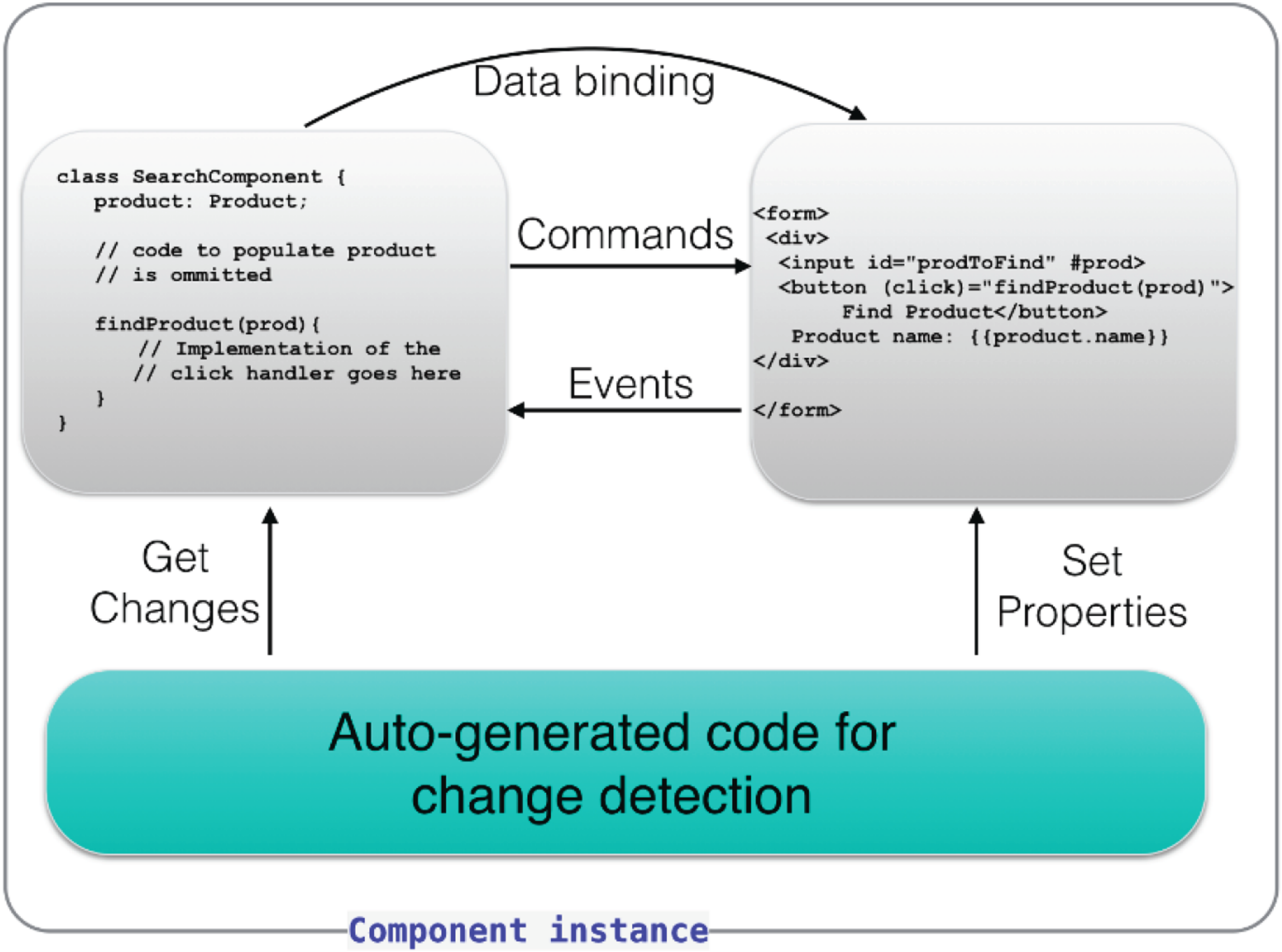
## Component in Example

```
@Component({
 selector: 'display'
 template: '<p>My name: {{ myName }}</p>'
})
class DisplayComponent {
 myName: string;

 constructor() {
  this.myName = "Ali";
 }
}
```

# Angular Components

- Paired with a View
- Contains the code behind the view (presentation logic)
- Makes **data** (i.e., model objects) and **functions** accessible to the View
- A parent component sends **commands** to its child components, and children send **events** to their parent

# Component internals

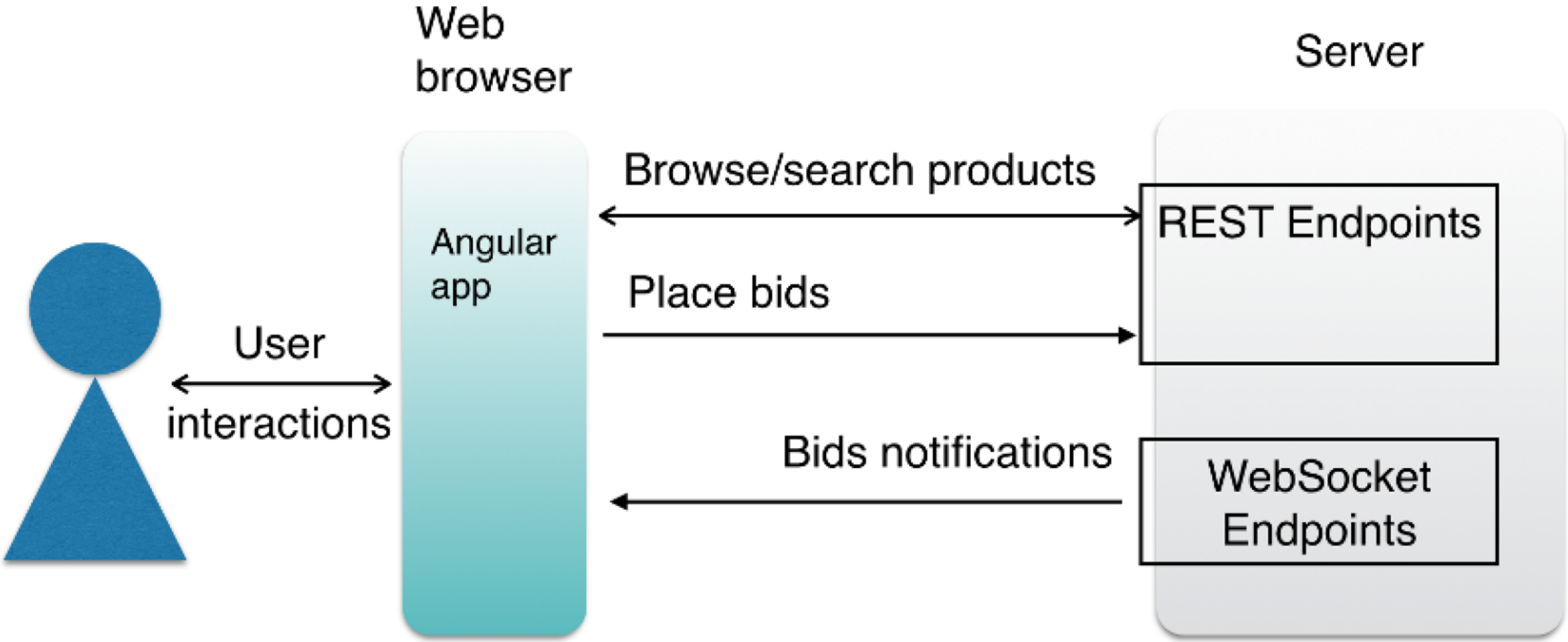


```
class SearchComponent {
   product: Product;

   // code to populate product
   // is ommitted

   findProduct(prod){
       // Implementation of the
      // click handler goes here
   }
}
```

```
<form>
 <div>
  <input id="prodToFind" #prod>
  <button (click)="findProduct(prod)">
       Find Product</button>
   Product name: {{product.name}}
</div>

</form>
```

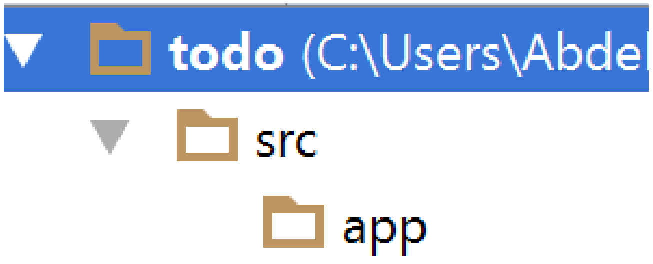
Component is a unit encapsulating functionality of a view, controller, and auto-generated change detector

# The Online Auction workflow
# See Posted Examples



# Setup Angular Project

- Download gitbash https://git-for-windows.github.io/
- Create todo folder. Right click then 'Git Bash Here'
- mkdir src then mkdir src/app
- Open todo folder in WebStrom
- In these in GitBash



todo (C:\Users\Abdel
src
app

```
npm init -y
npm i angular2 systemjs --save --save-exact
npm i typescript live-server --save-dev
```

# package.json

- Replace the ***scripts*** section with:

```
"scripts": {
  "tsc": "tsc -p src -w",
  "start": "live-server --open=src"
},
```

# TypeScript Compiler Config

- Under ***src***

create ***tsconfig.json***

```
{
  "compilerOptions": {
    "target": "ES5",
    "module": "commonjs",
    "sourceMap": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "removeComments": false,
    "noImplicitAny": false
  }
}
```

- Run **T**ype**S**cript **C**ompiler (TSC) in the root folder of the application to watch to **src** folder and auto-compile ts files

**npm run tsc**

- Launch **Live-Server** : little development server with live reload capability when the app changes

**npm start**

# NPM
# Node Package Management

< Back

# Package Management: NPM

- Node.js Package Management (NPM)
  - Install Nodejs packages or client libraries
  - `$ npm init` : Initializes an empty Node.js project with package.json file

```
$ npm init
//enter package details
name: "NPM demos"
version: 0.0.1
description: "Demos for the NPM package management"
entry point: main.js
test command: test
git repository: http://github.com/user/repository-name
keywords: npm, package management
author: ae@qu.edu.qa
license: MIT
```

# Package Management: NPM (2)

- Installing modules
  - `$ npm install package-name [--save-dev]`
    - Installs a package to the Node.js project
    - `--save-dev` suffix adds dependency in package.json

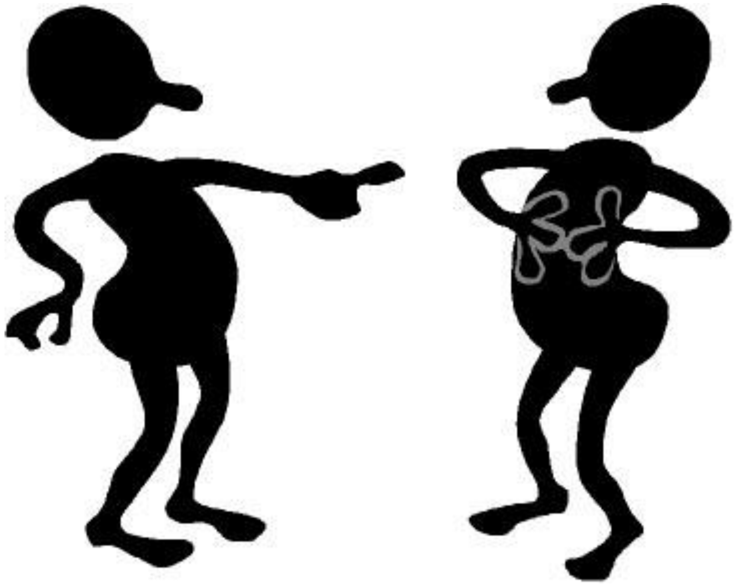
```
npm i angular2 systemjs --save --save-exact
npm i typescript live-server --save-dev
```

- Before running the project

```
$ npm install
```

- Installs all missing packages from package.json

# Directives



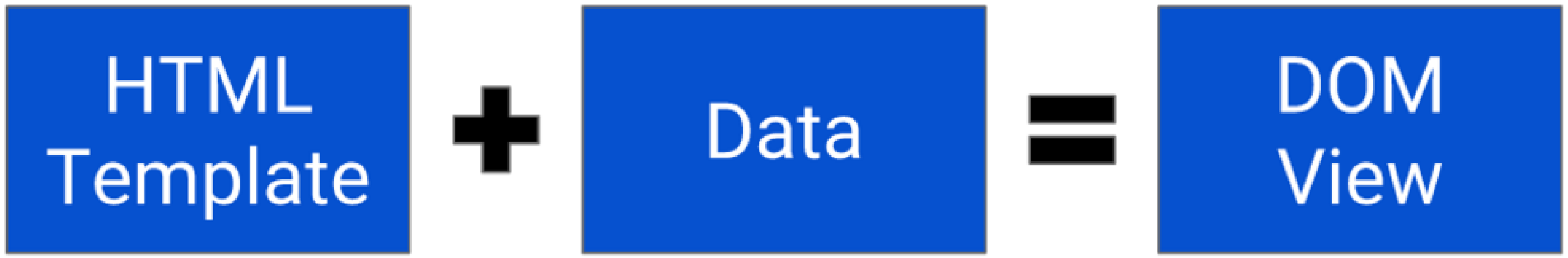
Back

# Directives

- Directives are used to create **client-side HTML templates**
  - Extends HTML functionality = Teaches HTML new tricks!
  - Adds additional markup to the view (e.g., dynamic content place holders)
  - A directive is just a function which executes when Angular 'compiler' encounters it in the DOM
  - Built-in directives start with ***ng** and they cover the core needs

# HTML Template

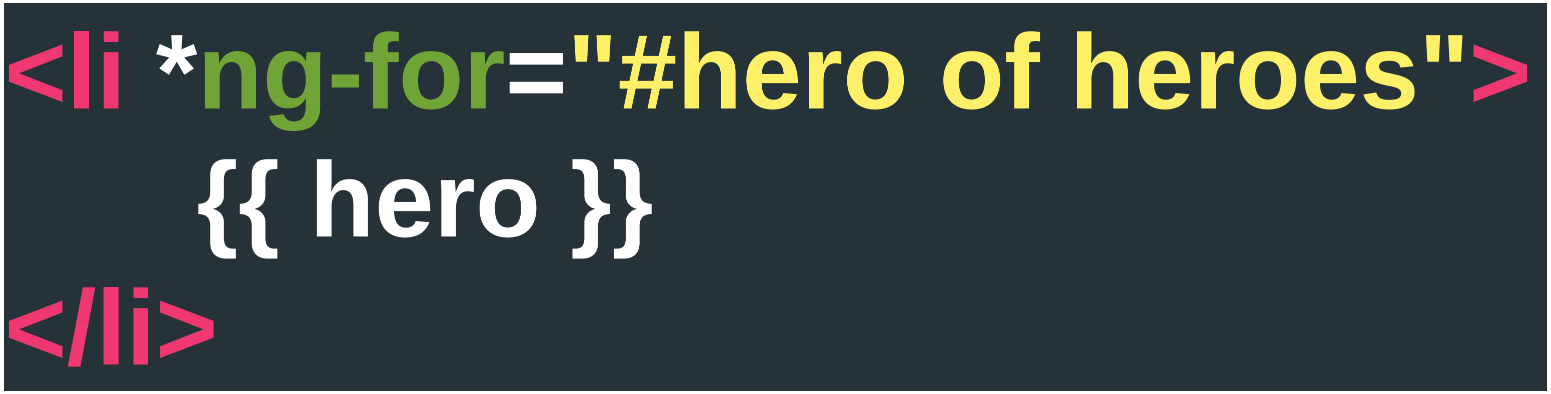
- Template is:
  - Partial HTML file that contains only part of a web page
  - Contains HTML augmented with Angular Directives
  - Rendered in a "parent" view



HTML Template + Data = DOM View

# Common Built-in Directives : ng-for

- **ng-for**: **repeater** directive. It marks <li> element (and its children) as the "repeater template"



```
<li *ng-for="#hero of heroes">
      {{ hero }}
</li>
```

- The **#hero** declares a local variable named hero
- Needs

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

`directives: [NgFor]` in @Component decorator

Or `CORE_DIRECTIVES` to include common directives

# Common Built-in Directives : ng-if

- **ng-if**: conditional display of a portion of a view only if certain condition is true

```
<p *ng-if="heroes.length > 3">There are many heroes!</p>
```

- This element will be displayed only if *heroes.length > 3*
- Needs

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

`directives: [NgIf]` in @Component decorator

Or `CORE_DIRECTIVES` to include common directives

# Expressions

```
<body>
      1+2={{1+2}}
</body>
```

Back

# Expressions

- You can write expressions
  - Use curly brackets
  - You can evaluate some JS expressions
  - You can create arrays

```
{{ expression }}
{{ name }}
{{ amount * 100 + 3 }}
{{ number in [1, 2, 3, 4, 5, 6, 7, 8, 9] }}
```

These double curly braces will display (and automatically update) the name in the view.

# Pipes

- Declarative way to
  - format displayed data
  - filter and sort data arrays
- Pipes:
  - Can modify the output
  - Can format output
  - Can sort data
  - Can filter data

# Piles

- Using pipes

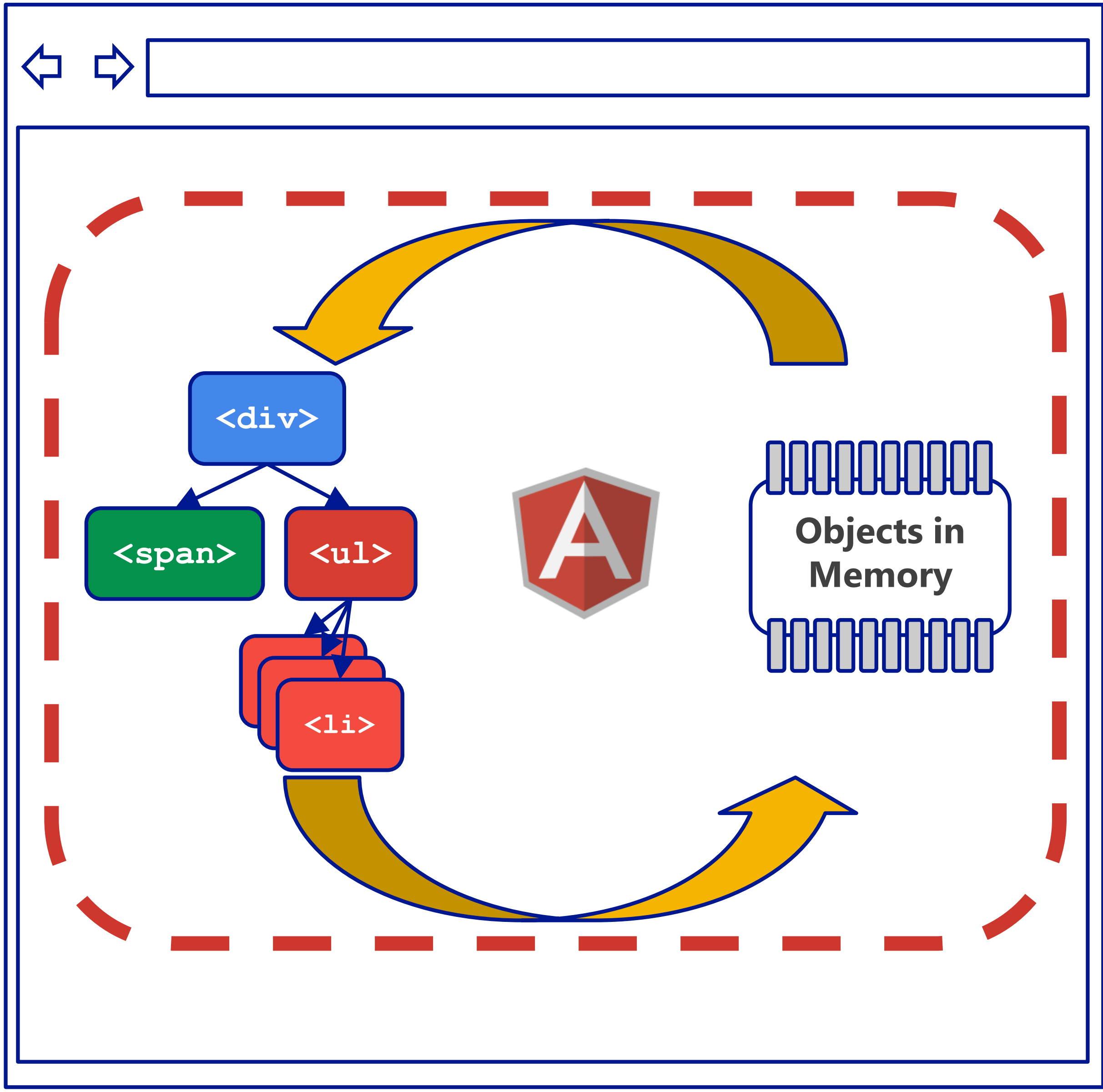
```
{{ expression | pipe }}
```

- Built-in pipes
    - uppercase, lowercase
    - number
    - currency
    - json
    - orderBy, limitTo, filter

```
<span>
  Today’s date is {{today | date}}
</span>
```

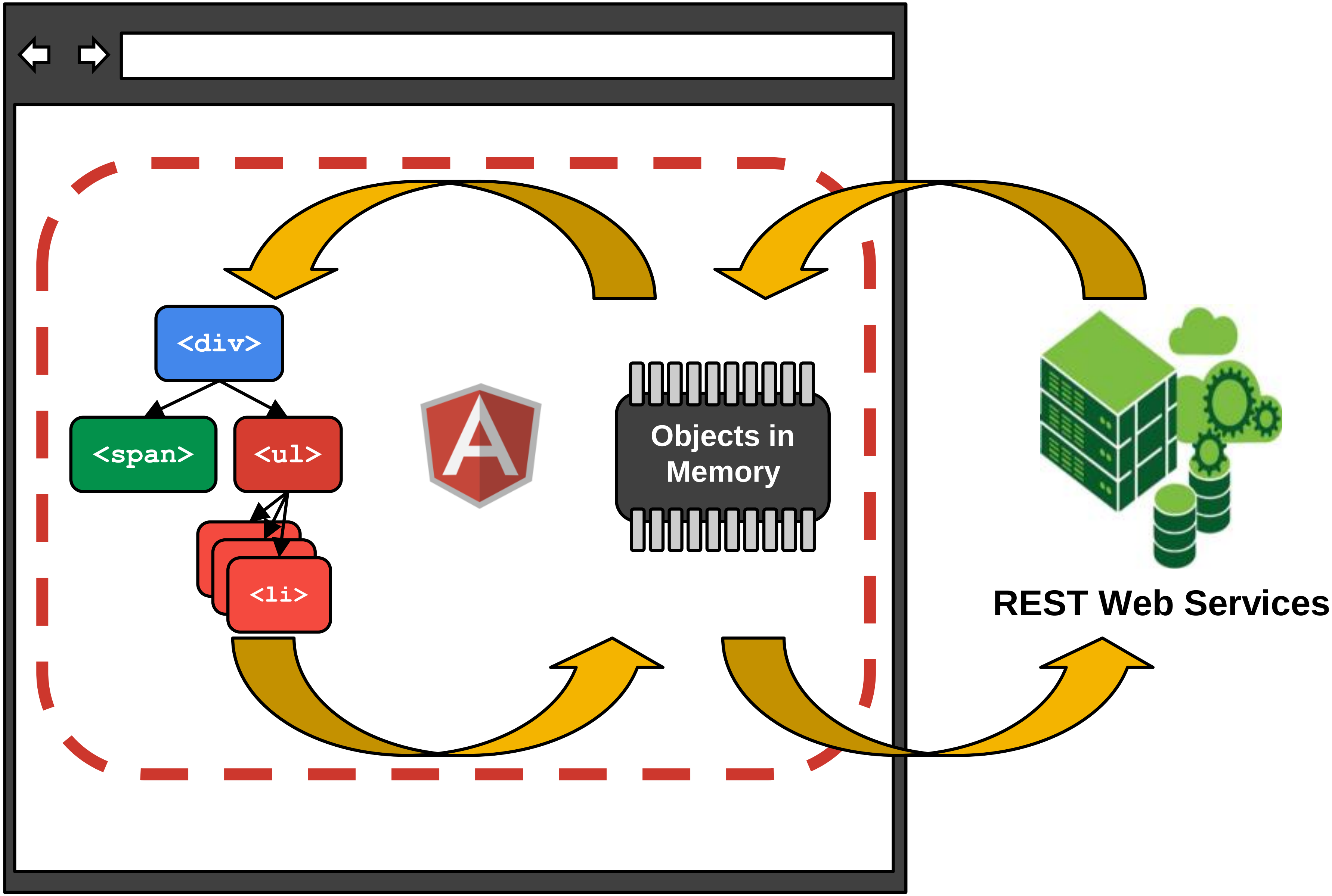
Today’s Date is Nov 4, 2015

# Bindings

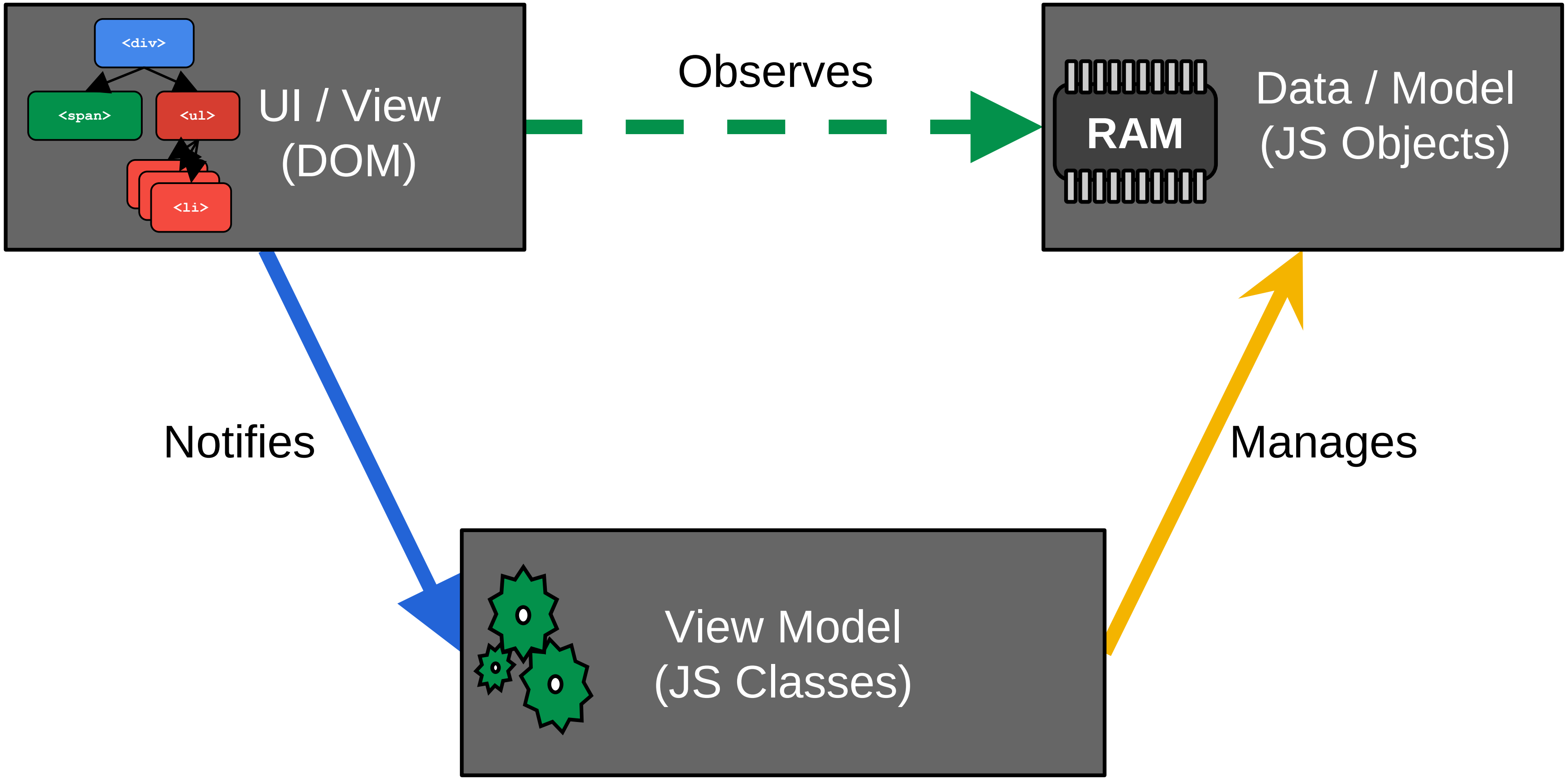


< Back

# Angular big picture



# Structure


<div>
<span>
<ul>
<li>
UI / View
(DOM)
Observes
RAM
Data / Model
(JS Objects)
Notifies
Manages
View Model
(JS Classes)

# Things you can bind to

| Binding | Example |
|---|---|
| **Properties** | <input **[value]**="firstName"> |
| **Events** | <button **(click)**="buy($event)"> |
| **Two-way** | <input **[(ng-model)]**="userName"> |

**Data binding associates the Model with the View**

# Example

```
<button
        [disabled]="!inputIsValid"
        (click)="authenticate()">
    Login
</button>

 <amazing-chart
        [series]="mySeries"
        (drag)="handleDrag()"/>
```

A statement performs an action

```
<div [hidden]="exp">
```

```
<div *ng-for="#guest of guestList">
  <guest-card [guest]="guest">
  </guest-card>
</div>
```

# Angular Event Binding syntax

- **(eventName) = eventHandler**: respond to the click event by calling the component's onBtnClick method

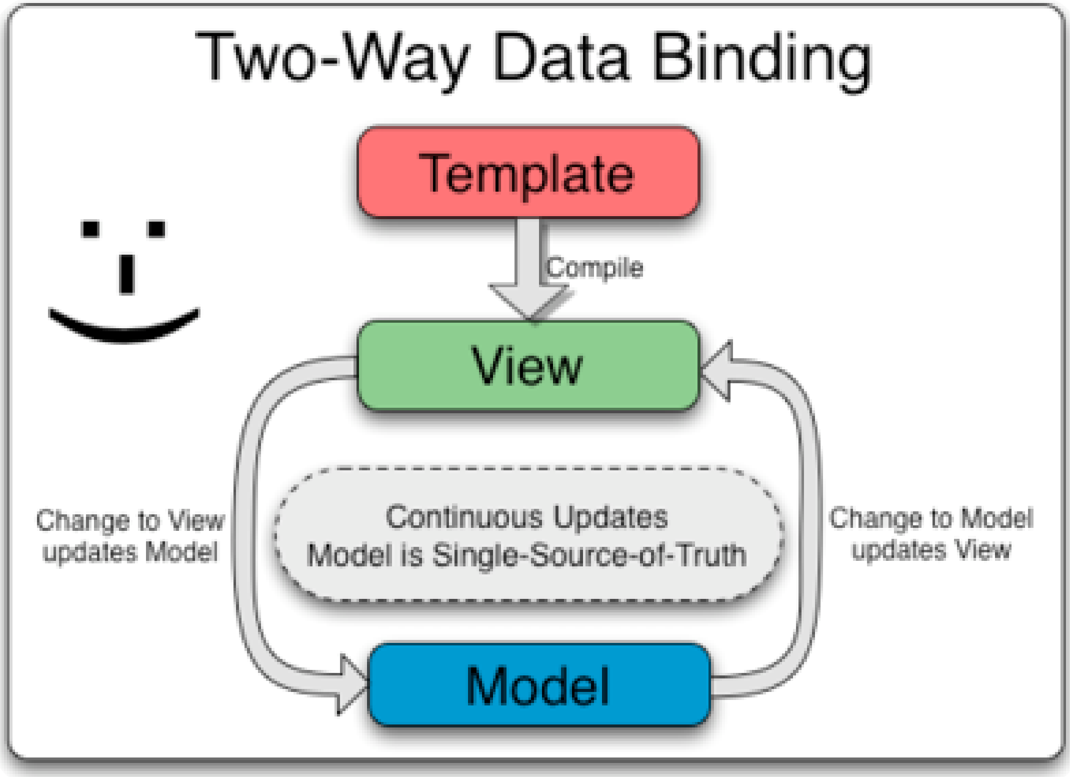
```
<button (click)="onBtnClick()">Click me!</button>
<input (keyup)="onKey($event)">
```

- $event is an optional standard DOM event object. It is value is determined by the source of the event.

# SearchComponent Example

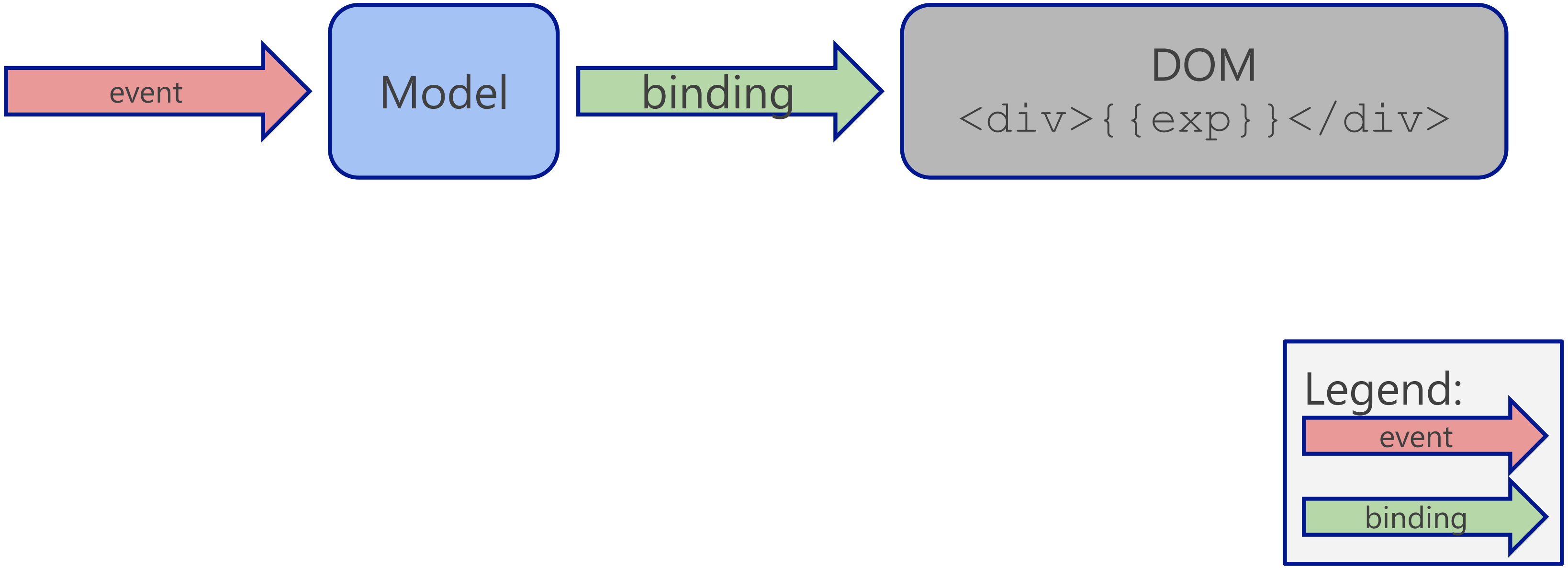
```
@Component({
  selector: 'search-product',
  template:
      `<form>
         <div>
           <input id="prodToFind" #prod>
           <button (click)="findProduct(prod)">Find Product</button>
           Product name: {{product.name}}
         </div>
       </form>
     `

})
class SearchComponent {
   product: Product;

   findProduct(product){
    // Implementation of the click handler goes here
   }
}
```

# Two Way Binding


Two-Way Data Binding
Template
Compile
View
Change to View updates Model
Continuous Updates
Model is Single-Source-of-Truth
Change to Model updates View
Model

< Back

# One-way data binding



event

Model

binding

DOM

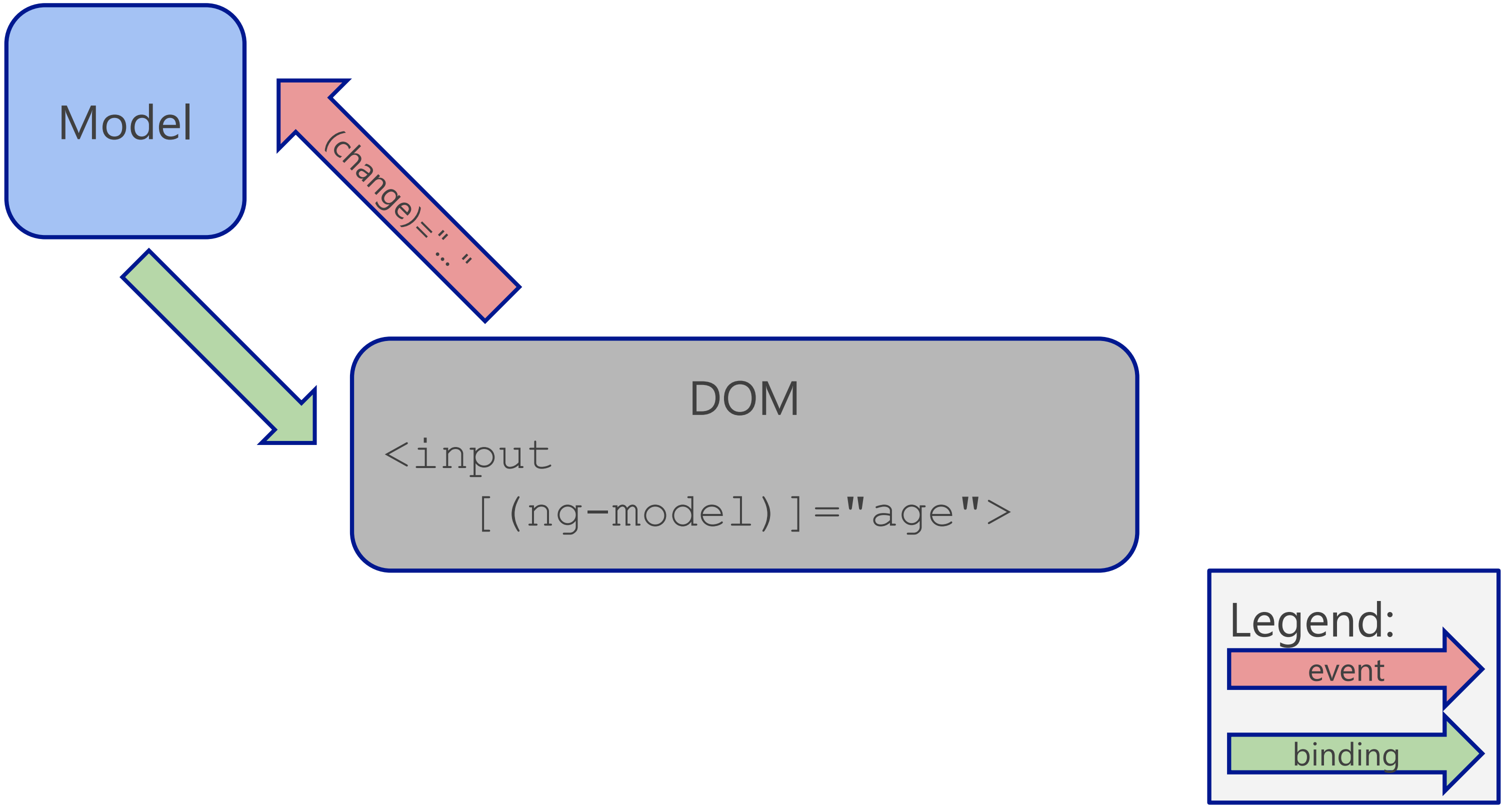
`<div>{{exp}}</div>`

Legend:

event

binding

# two-way data binding



Model

(change)="..."

DOM

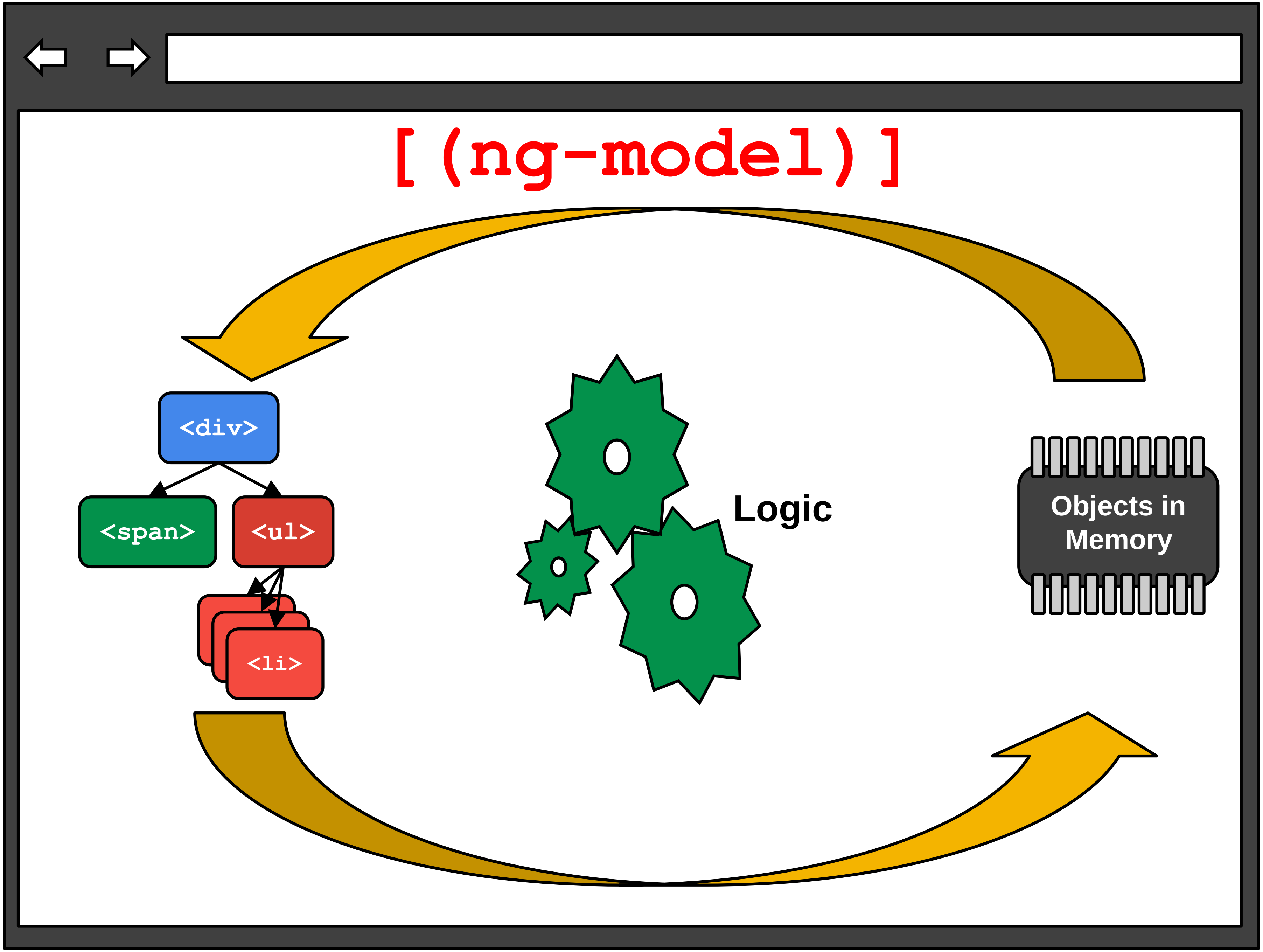
```
<input
    [(ng-model)]="age">
```

Legend:

event

binding

# [(ng-model)]

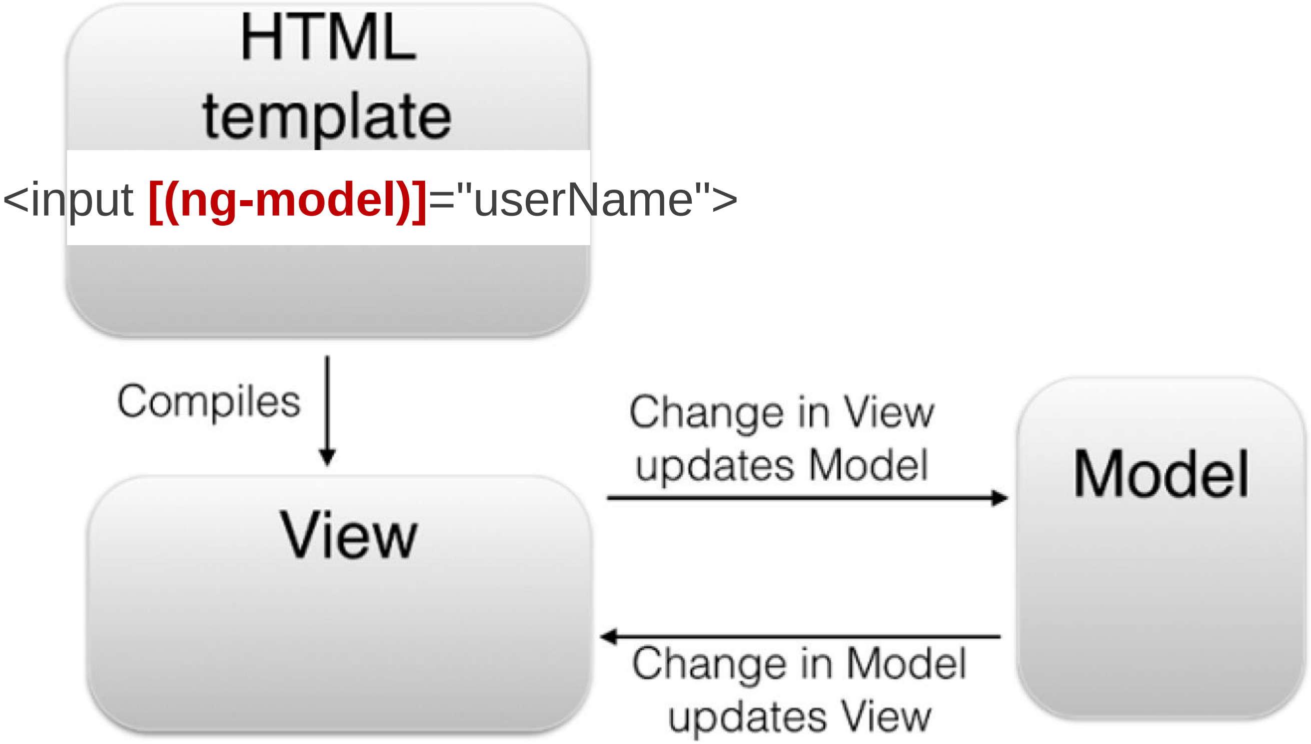


# Two Way Binding

- Automatic propagation of data changes between the model and the view
- Using **ngModel**
  - On input
  - On select
  - On textarea

```
<input type=“text” [(ng-model)]=“object.property” />


<input type=“text” [(ng-model)]=“property” />


<input type=“text” [(ng-model)] =“object.container.property” />
```

# Two-way binding

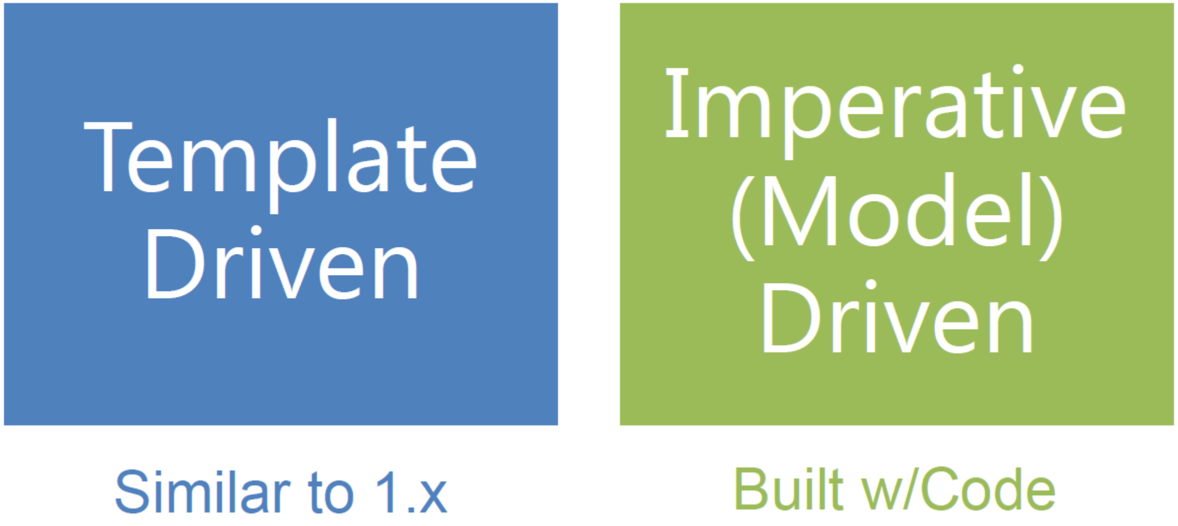

HTML template
<input [(ng-model)]="userName">
Compiles
View
Change in View updates Model
Model
Change in Model updates View

**ng-model** will display the userName in a view and it will automatically update it in case it changes in the model. If the user modifies the userName on the view then the changes are propagated to the model. Such a **two-directional** updates mechanism is called two-way data binding

# Forms

< Back

# Angular 2 Form Techniques



Template Driven

Similar to 1.x

Imperative (Model) Driven

Built w/Code

- Imperative (or Model) Driven: Build Form with Code

# Controller

Template Driven

TypeScript class

Bindable Controller Properties

Controller Methods

```
class CheckOutCtrl {
  model = new CheckoutModel();
  countries = ['US', 'Canada'];

  onSubmit() {
    console.log("Submitting:");
    console.log(this.model);
  }
}
```

# Model

Template Driven

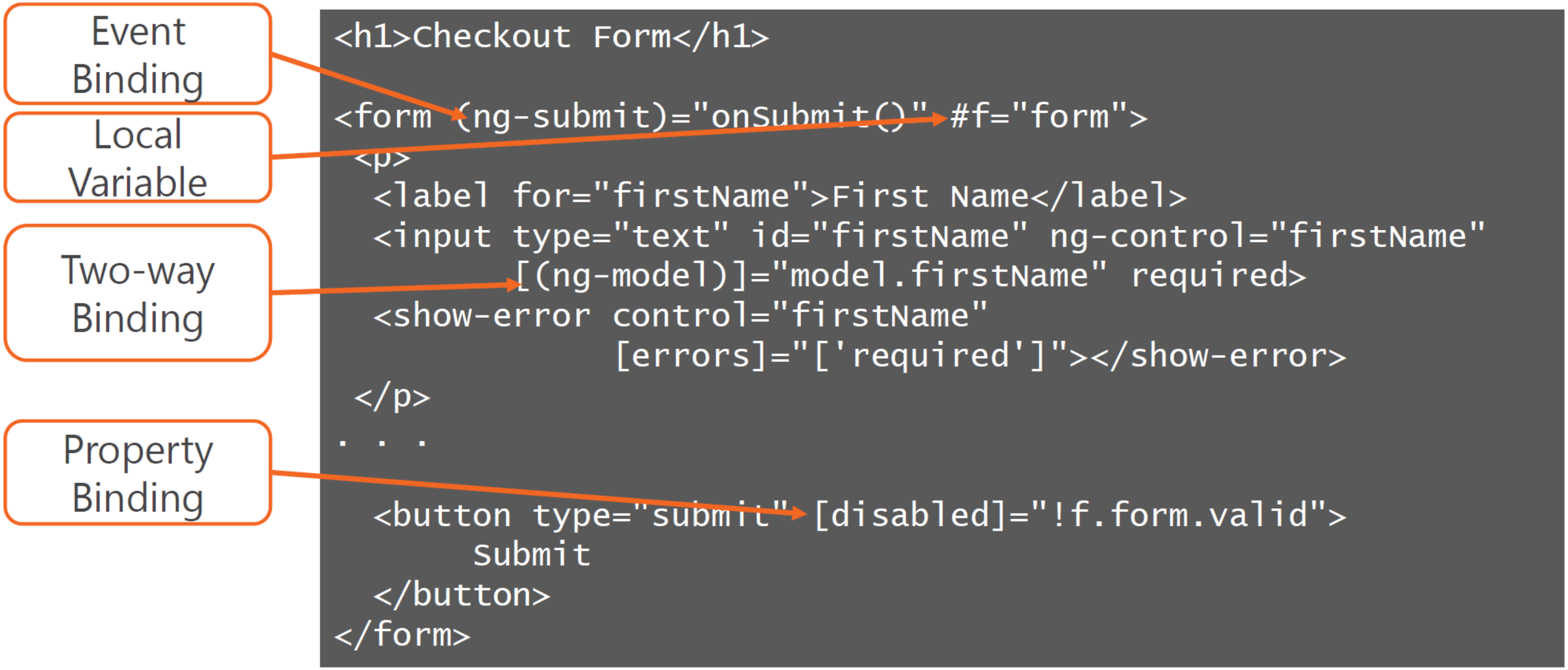
TypeScript class

Strongly Typed Properties

```
class CheckoutModel {
  firstName: string;
  middleName: string;
  lastName: string;
  country: string = "Canada";

  creditCard: string;
  amount: number;
  email: string;
  comments: string;
}
```

# View

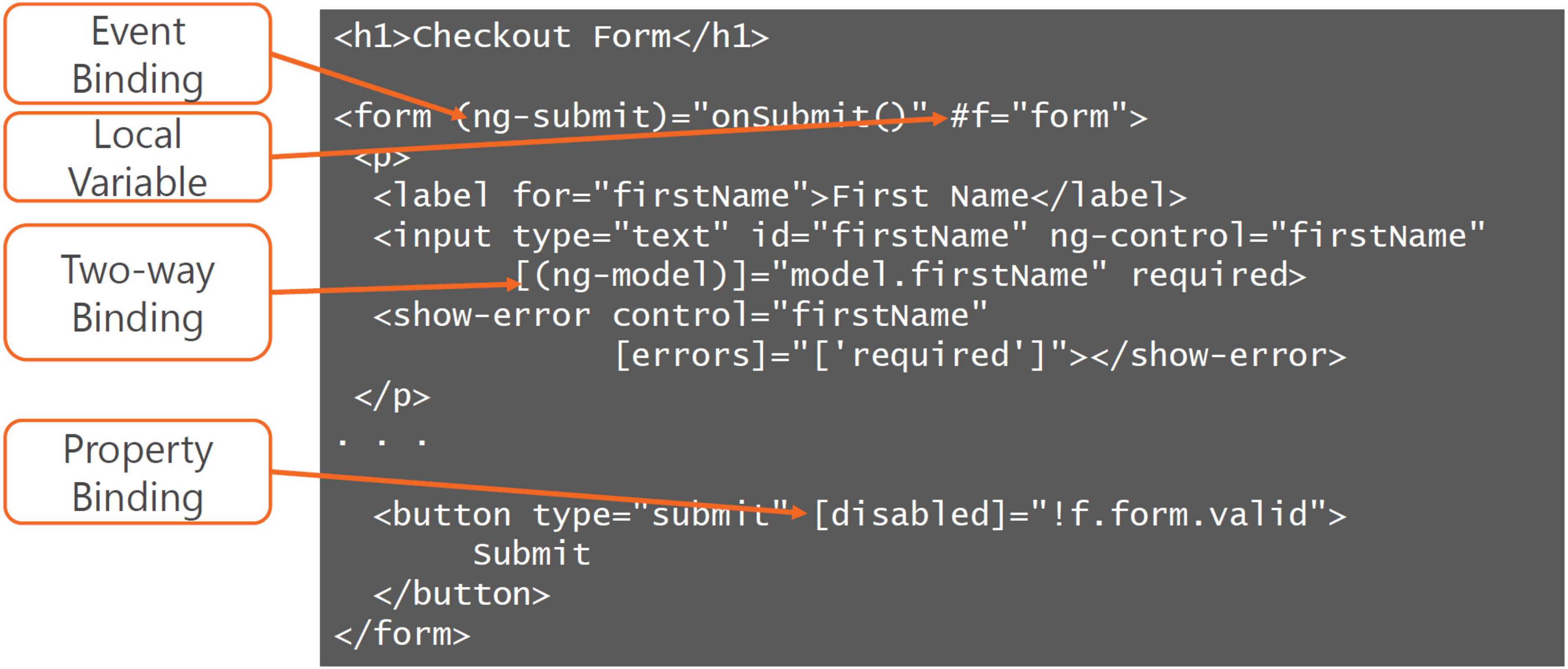
Template Driven



```
<h1>Checkout Form</h1>

<form (ng-submit)="onSubmit()" #f="form">
 <p>
  <label for="firstName">First Name</label>
  <input type="text" id="firstName" ng-control="firstName"
         [(ng-model)]="model.firstName" required>
  <show-error control="firstName"
              [errors]="['required']"></show-error>
 </p>
. . .

  <button type="submit" [disabled]="!f.form.valid">
       Submit
  </button>
</form>
```

# View

Template Driven



Event Binding

Local Variable

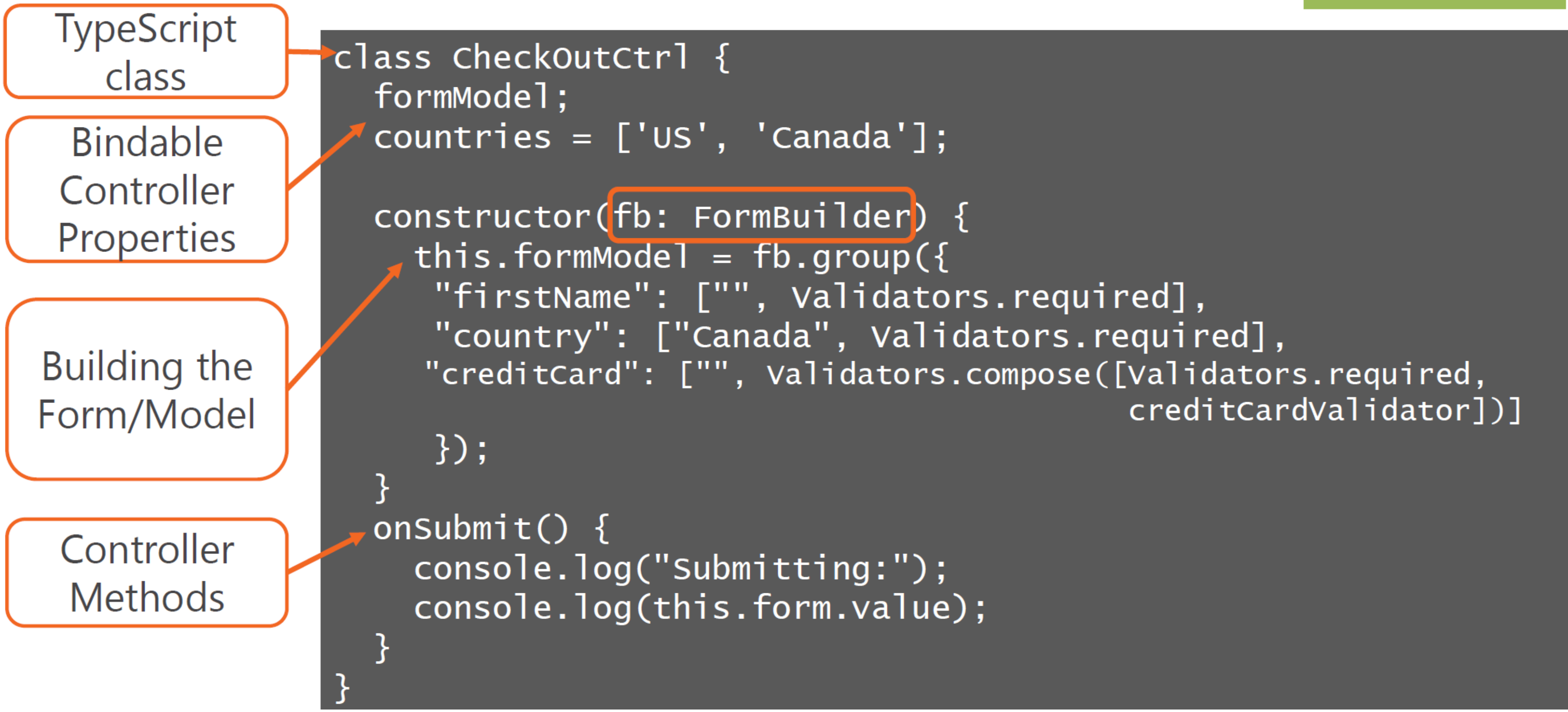
Two-way Binding

Property Binding

```
<h1>Checkout Form</h1>

<form (ng-submit)="onSubmit()" #f="form">
 <p>
  <label for="firstName">First Name</label>
  <input type="text" id="firstName" ng-control="firstName"
         [(ng-model)]="model.firstName" required>
  <show-error control="firstName"
              [errors]="['required']"></show-error>
 </p>
. . .

  <button type="submit" [disabled]="!f.form.valid">
       Submit
  </button>
</form>
```

# Controller

Model Driven



TypeScript class

Bindable Controller Properties

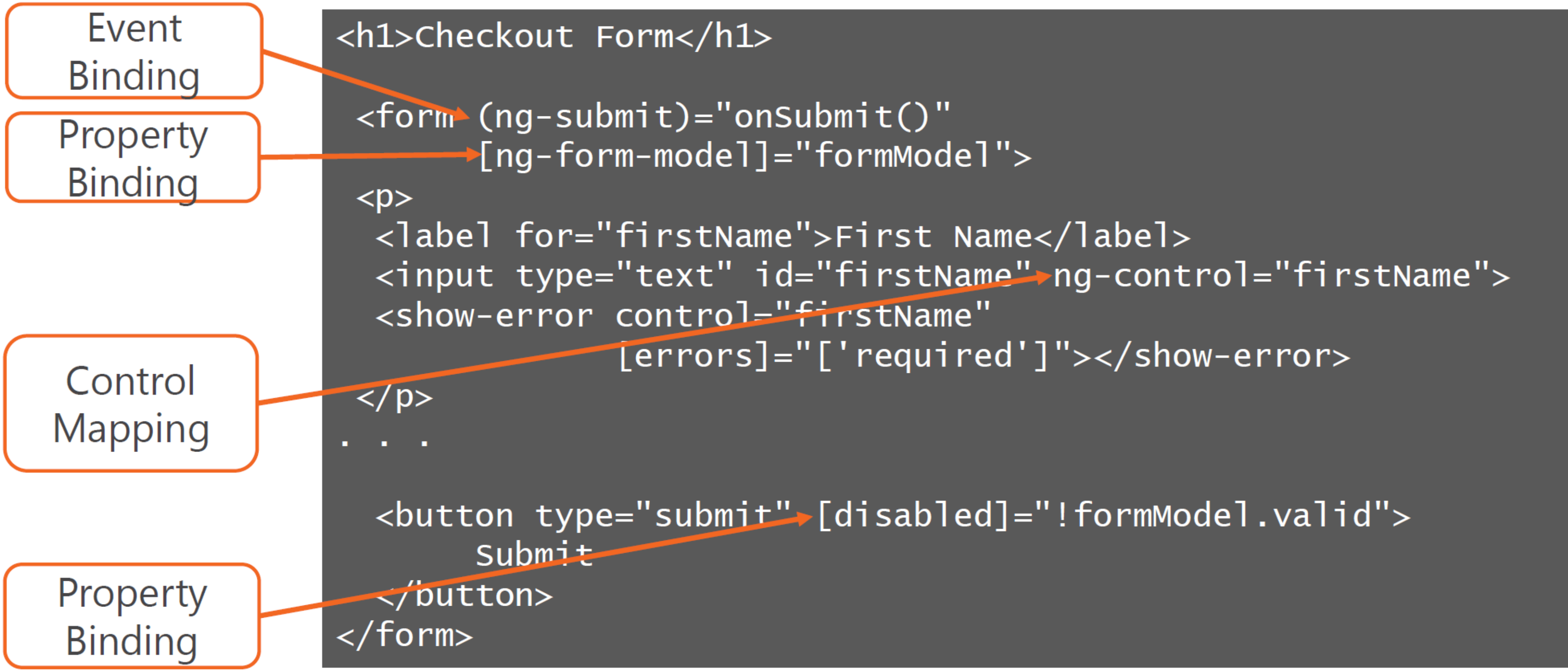
Building the Form/Model

Controller Methods

```
class CheckOutCtrl {
  formModel;
  countries = ['US', 'Canada'];

  constructor(fb: FormBuilder) {
    this.formModel = fb.group({
     "firstName": ["", Validators.required],
     "country": ["Canada", Validators.required],
     "creditCard": ["", Validators.compose([Validators.required,
                                        creditCardValidator])]
     });
  }
  onSubmit() {
    console.log("Submitting:");
    console.log(this.form.value);
  }
}
```

# View

Model Driven



Event Binding

Property Binding

Control Mapping

Property Binding

```
<h1>Checkout Form</h1>

 <form (ng-submit)="onSubmit()"
       [ng-form-model]="formModel">
 <p>
  <label for="firstName">First Name</label>
  <input type="text" id="firstName" ng-control="firstName">
  <show-error control="firstName"
             [errors]="['required']"></show-error>
 </p>
. . .

  <button type="submit" [disabled]="!formModel.valid">
       Submit
  </button>
</form>
```

# Key Differences

## Template-Driven

- Controller exposes **data** model

```
class CheckOutCtrl {
  model = new CheckoutModel();
  countries = ['US', 'Canada'];
...
}
```

## Model-Driven

- Controller exposes **form** model

```
class CheckOutCtrl {
  formModel;
  countries = ['US', 'Canada'];
...
}
```

# Key Differences

## Template-Driven

- Controller exposes data model
- Binding & Validation in **View**

```
<input
 type="text"
 id="firstName"
 ng-control="firstName"
 [(ng-model)]="model.firstName"
 required>
```

## Model-Driven

- Controller exposes form model
- Binding & Validation in **Controller**

```
class CheckoutCtrl {
 formModel;
 countries = ['US', 'Canada'];

 constructor(fb: FormBuilder) {
  this.formModel = fb.group({
  "firstName": ["", Validators.required],
  "LastName": ["", Validators.required],
  . . .
       });
  }
...
}
```

# Key Differences

## Template-Driven

- Controller exposes data model
- Binding & Validation in View
- View contains **data bindings**

```
<input
  type="text"
  id="firstName"
  ng-control="firstName"
  [(ng-model)]="model.firstName"
  required>
```

## Model-Driven

- Controller exposes form model
- Binding & Validation in Controller
- View contains **control mappings**

```
<input
  type="text"
  id="firstName"
  ng-control="firstName">
```

# Benefits to Model-Driven

- Behavior (Binding, validation) is in the code, not the template

  - Easier to reason against

  - More readily unit tested

# Validation

- Default validation
  - Required – makes property required
  - ngPattern – regex pattern
  - Form Properties – valid / invalid
  - CSS Classes – classes can be styled

# Routing and views

< Back

# Routes

- Implement client-side navigation for SPA:
    - Configure routes, map them to the corresponding components in a declarative way.
- In SPA, we want to be able to change a URL fragment that won’t result in full page refresh, but would do a partial page update
- Defines the app navigation in ***@RouteConfig*** *annotation*

  - On URL change => load a particular component

# Angular 2 Router

- **RouterOutlet** – a directive that serves as a placeholder within your Web page where the router should render the component
- **@RouteConfig** – an annotation to map URLs to components to be rendered inside the *<router-outlet></router-outlet>* area
- **RouteParams** – a service for passing parameter to a component rendered by the router
- **RouterLink** – a directive to declare a link to a view and may contain optional parameters to be passed to the component

# Router Programming Steps (1 of 2)

1. Configure the router on the root component level to map the URL fragments to the corresponding named components

- If some of the components expect to receive input values, you can use route **params**

- Needs

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

`directives: [ROUTE_DIRECTIVES]` in @Component decorator

```
@RouteConfig([
  {path: '/', component: AppComponent, as: 'Home'},
  {path: '/hero/:id', component: HeroFormComponent, as: 'Hero'}
])
```

# Router Programming Steps (2 of 2)

2. Add **<router-outlet></router-outlet>** to the view to specify where the router will render the component

3. Add the HTML anchor tags with **[router-link]** attribute, so when the user clicks on the link the router will render the corresponding component.

- Think of **[router-link]** as href attribute of anchor tag

This is the **route name** specified in the **as** attribute of the route defined in @RouteConfig

```
<a [router-link]="['/Home']">Home</a>
<a [router-link]="['/Hero', {id: 1234}]">
Hero</a>
<router-outlet></router-outlet>
```

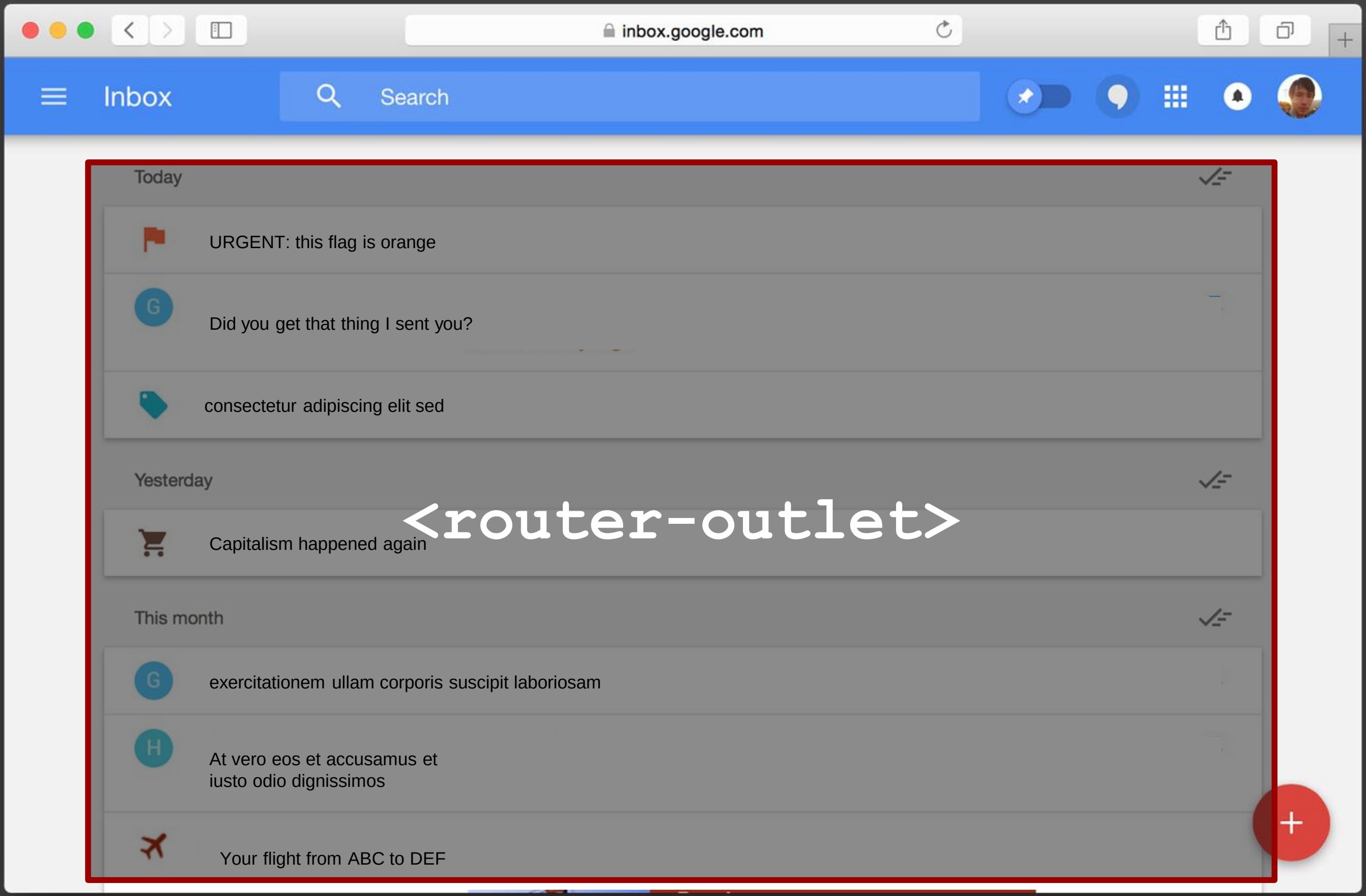
# Route Parameters

- Route parameters start with ":"

```
@RouteConfig([
{path: '/product/:id', component: ProductDetailComponent, as: 'ProductDetail'}
])
```

- You can use them later in ViewModel constructor

```
export class ProductDetailComponent {
    productID:string;

    constructor(params : RouteParams) {
        this.productID = params.get('id');
    }
}
```



Today

URGENT: this flag is orange

Did you get that thing I sent you?

consectetur adipiscing elit sed

Yesterday

Capitalism happened again

This month

exercitationem ullam corporis suscipit laboriosam

At vero eos et accusamus et
iusto odio dignissimos

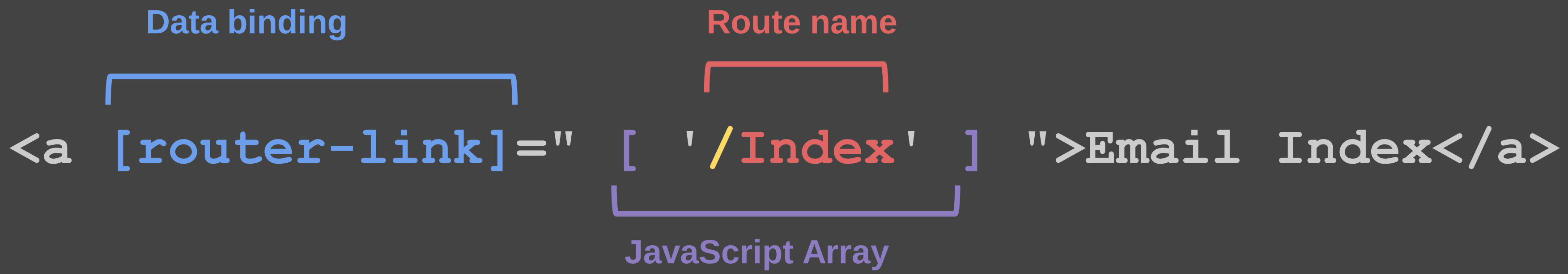
Your flight from ABC to DEF

<router-outlet>

# Creating Links

```
@Component({
  selector: 'app-cmp'
  template: `<a [router-link]="['/Index']">Index</a>
             <a [router-link]="['/Search']">Search</a>
             <router-outlet></router-outlet>`,
  directives: ROUTER_DIRECTIVES
})
@RouteConfig([
  { path: '/',          component: IndexCmp,  as: 'Index' },
  { path: '/email/:id', component: EmailCmp,  as: 'Email' },
  { path: '/search',    component: SearchCmp, as: 'Search' }
])
class AppCmp {}
```

# Link Anatomy



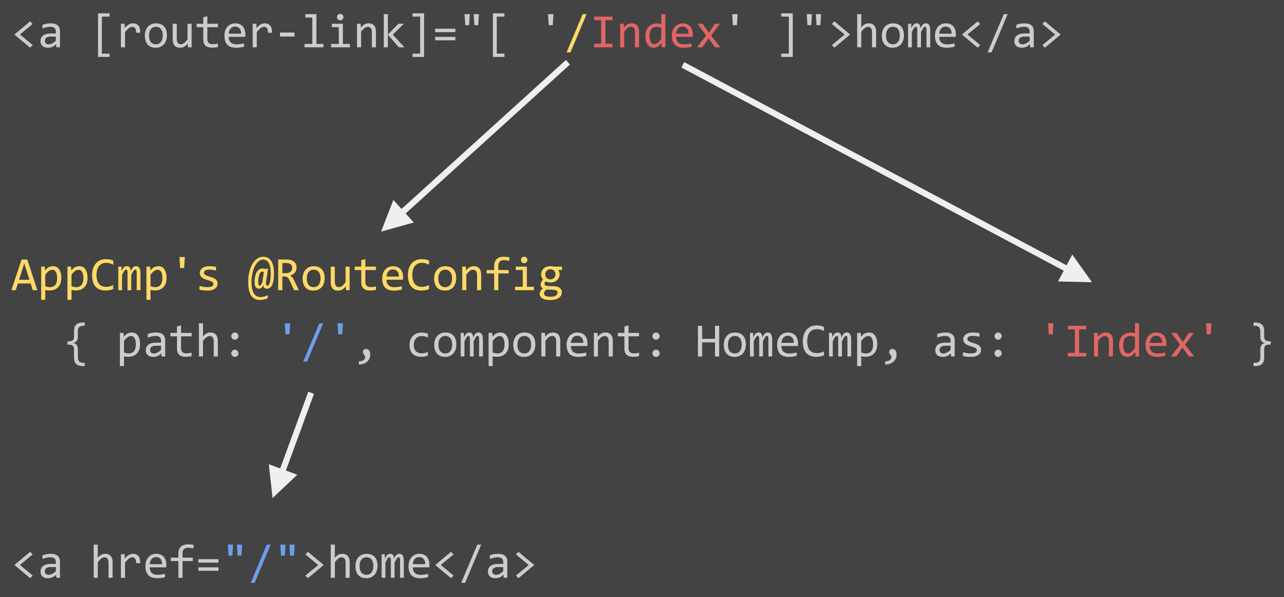
Data binding

Route name

```
<a [router-link]=" [ '/Index' ] ">Email Index</a>
```

JavaScript Array

# Link Anatomy



```
<a [router-link]="[ '/Index' ]">home</a>
```

```
AppCmp's @RouteConfig
  { path: '/', component: HomeCmp, as: 'Index' }
```

```
<a href="/">home</a>
```

Router-link uses route names instead of URL segments
=> Makes it easier to refactor URLs

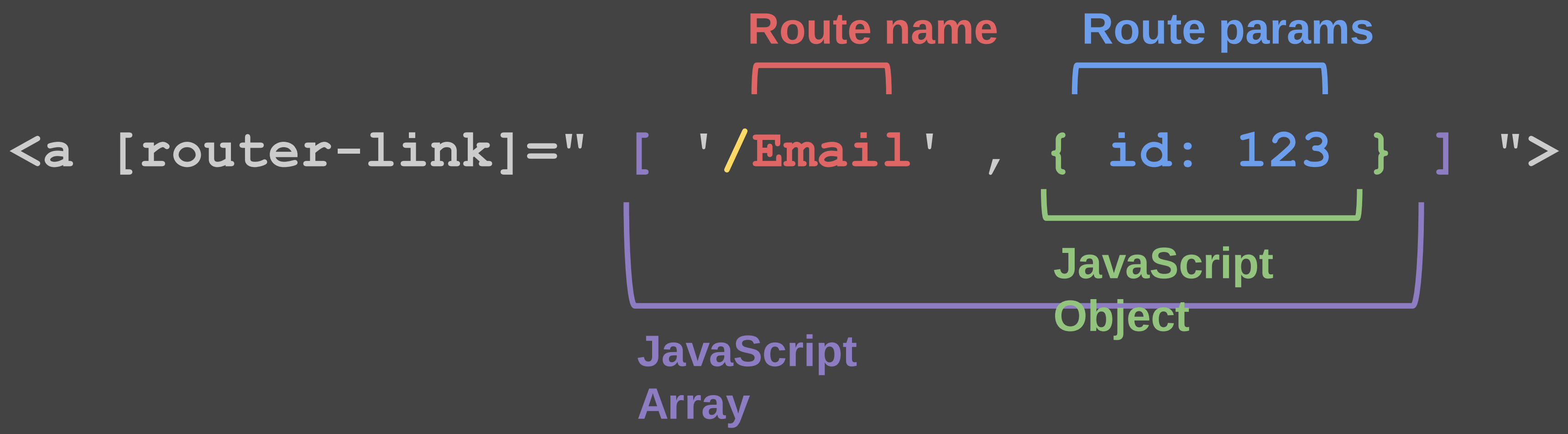
# Route with Params

```
@Component( … )
@RouteConfig([
  { path: '/',          component: IndexCmp,  as: 'Index' },
  { path: '/email/:id', component: EmailCmp,  as: 'Email' },
  { path: '/search',    component: SearchCmp, as: 'Search' }
])
class AppCmp {}
```

# Link with Params

```
@Component({
  selector: 'index-cmp'
  template: `
<ul>
  <li *ng-for="#email of emails">
    <a [router-link]="[ '/Email', { id: email.id } ]">{{ email.subject }}</a>
  </li>
</ul>`,
  directives: ROUTER_DIRECTIVES
})
class IndexCmp {}
```

# Link with Params



Route name

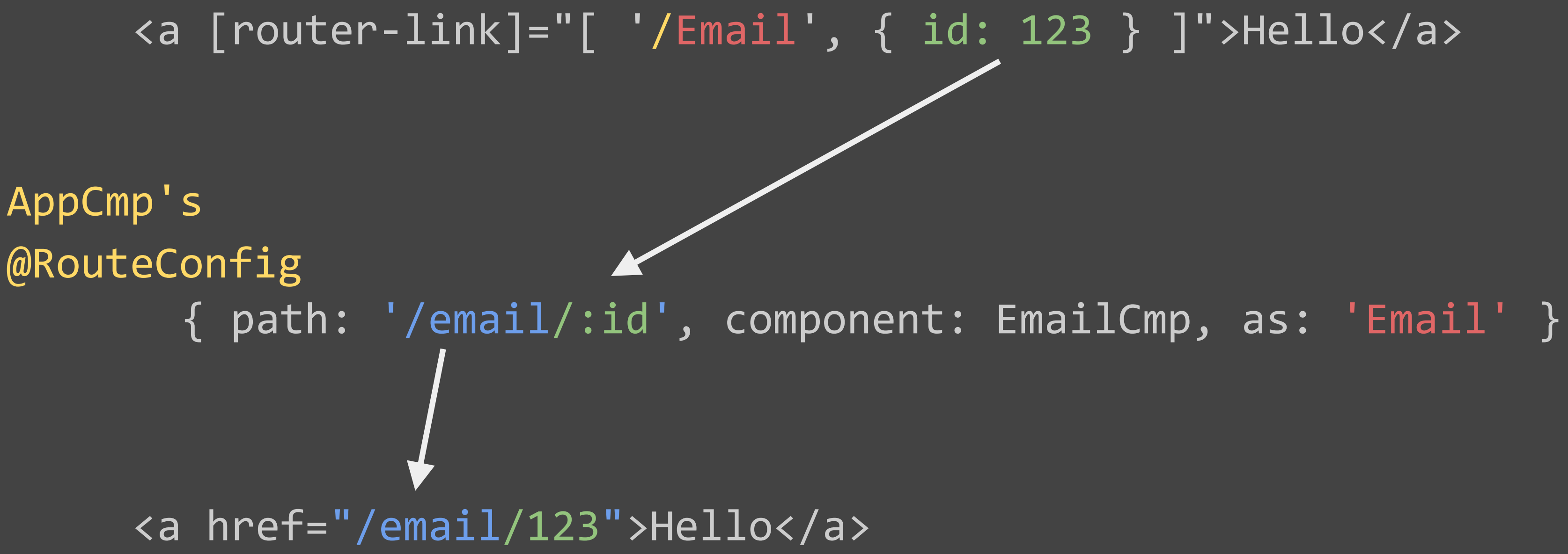
Route params

```
<a [router-link]=" [ '/Email' , { id: 123 } ] ">
```

JavaScript Object

JavaScript Array

# Link with Params



```
     <a [router-link]="[ '/Email', { id: 123 } ]">Hello</a>

AppCmp's
@RouteConfig
       { path: '/email/:id', component: EmailCmp, as: 'Email' }

     <a href="/email/123">Hello</a>
```

# Resources

- Angular 2 Router

https://www.youtube.com/watch?v=z1NB-HG0ZH4

- Angular 2 Router Slides

http://goo.gl/n38EDf

# Client-side Services

< Back

# Client-side Services

- JavaScript class than can be injected and made available to the entire application

```
import {Http} from 'angular2/http';
import {Injectable} from 'angular2/angular2';

@Injectable()
export class PeopleService {
    constructor(http:Http) {
        this.people = http.get('api/people.json')
            .map(response => response.json());
    }
}
```

# Using the Service

- Either add it in the Providers property of the component or when in app bootstrap

```
@Component({
  selector: 'my-app',
  providers: [PeopleService]
})
```

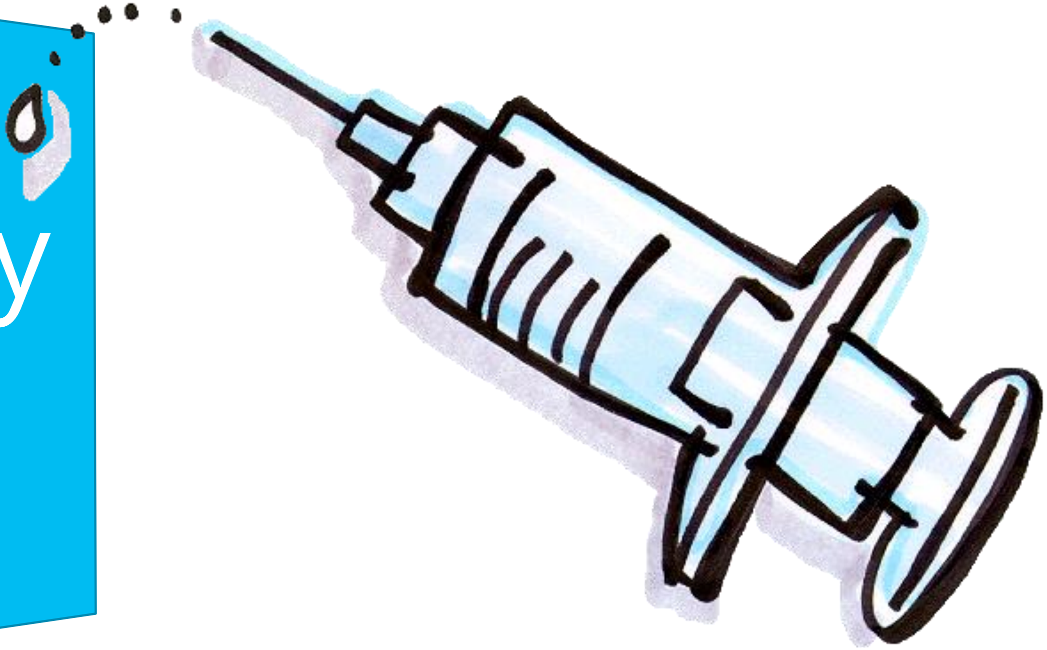
or

```
bootstrap(App, [PeopleService])
```

- Injecting the PeopleService

```
export class PeopleComponent {
    constructor(peopleService:PeopleService) {
```

# Dependency Injection

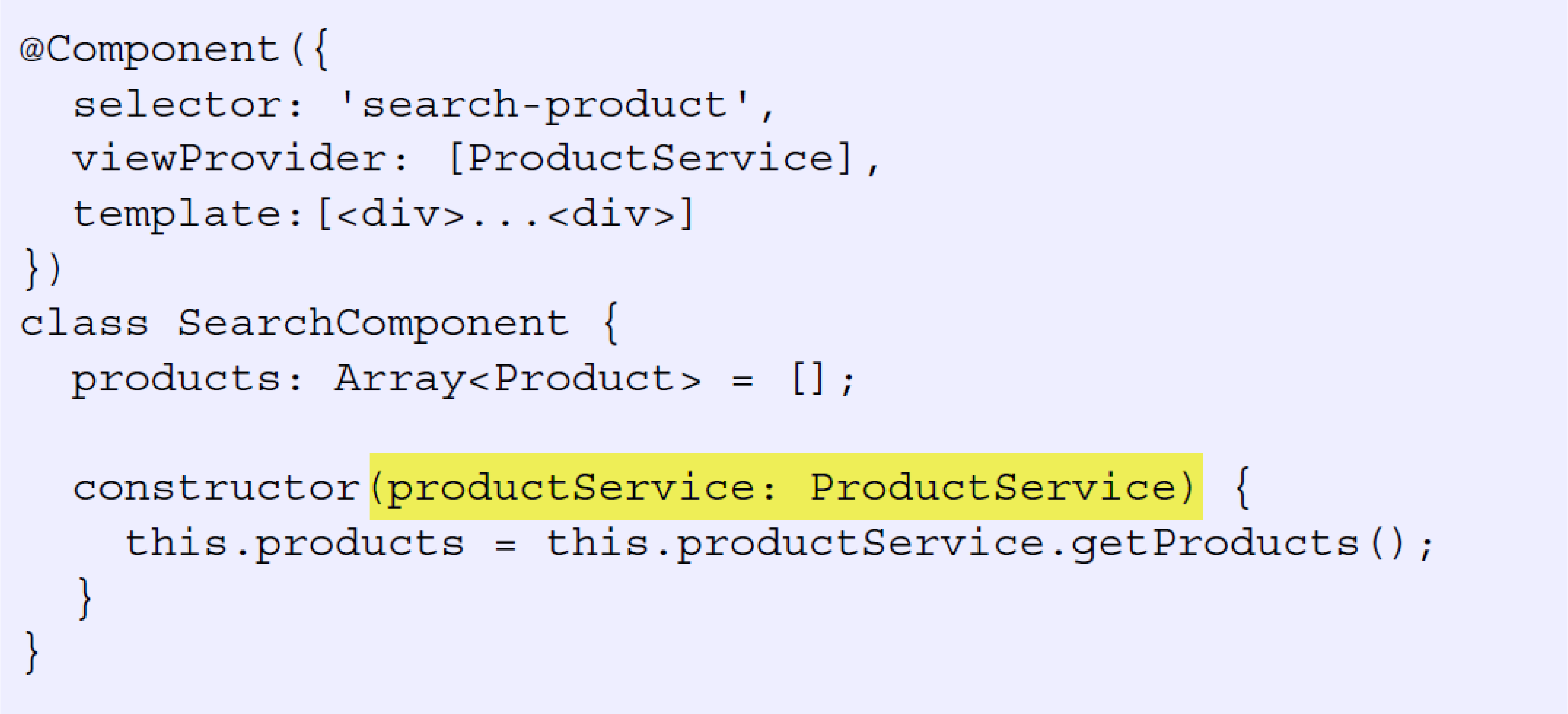


Back

# Dependency Injection

- Dependency Injection is a design pattern that inverts the way of creating objects your code depends on
- Instead of explicitly creating object instances (e.g. with new) the framework will create and inject them into your code
- Angular comes with a dependency injection module
- You can inject dependencies into the component only via its constructor

# Dependency Injection



```
@Component({
  selector: 'search-product',
  viewProvider: [ProductService],
  template:[<div>...<div>]
})
class SearchComponent {
  products: Array<Product> = [];

  constructor(productService: ProductService) {
    this.products = this.productService.getProducts();
  }
}
```

- Inject the **ProductService** object into the **SearchComponent** by declaring it as a constructor argument
- Angular will instantiate the ProductService and provide its reference to the SearchComponent.

# Reactive Programming

Programming paradigm oriented around data flows and the propagation of change

Back

# Push vs. Pull

Iterable vs. Observable:

**Iterable**: sequence the consumer iterate over and pull elements

**Observable**: sequence that notifies when a new value is added and pushes the value to observer (listener)

=> difference in who is the master, and who is the slave

# Reactive Programming is based on Observables

What’s an Observable?

- Like a Promise… of many values
- Like an array… but async
  = Asynchronous Data Streams

# Observable: Like a Promise of Many Values

- **Promise:**

```
httpRequest.then(success, error);
```

- **Oberverable:**

```
socketMessages
   .subscribe(next, error, complete);
```

```
e.g:
socketMessages
   .subscribe(
     (msg) => console.log(msg),
     (err) => console.error(err),
     () => console.log(‘completed!’)
   );
```

# Observable - like an array, but async = Asynchronous Data Streams

```
let socketMessages = [
  {body: ‘hello’},
  {body: ‘goodbye’},
  {body: ‘tak!’}
];
```

# Map/Filter/ConcatAll can be applied to observables similar to the way done of arrays

```
> [1, 2, 3].map(x => x + 1)
> [2, 3, 4]


> [1, 2, 3].filter(x => x > 1)
> [2, 3]


> [ [1], [2, 3], [], [4] ].concatAll()
> [1, 2, 3, 4]
> █
```

# Observable Combinators

```
socketMessages
    .map(msg => msg.body)
    .subscribe(body => {
      console.log(body);
    });
```

# Observable Combinators

```
let taks = socketMessages
    .map(msg => msg.body)
    .filter(body => body === ‘tak!’);

taks.subscribe(msg => console.log(msg));

//later
takSubscriber.unsubscribe();
```

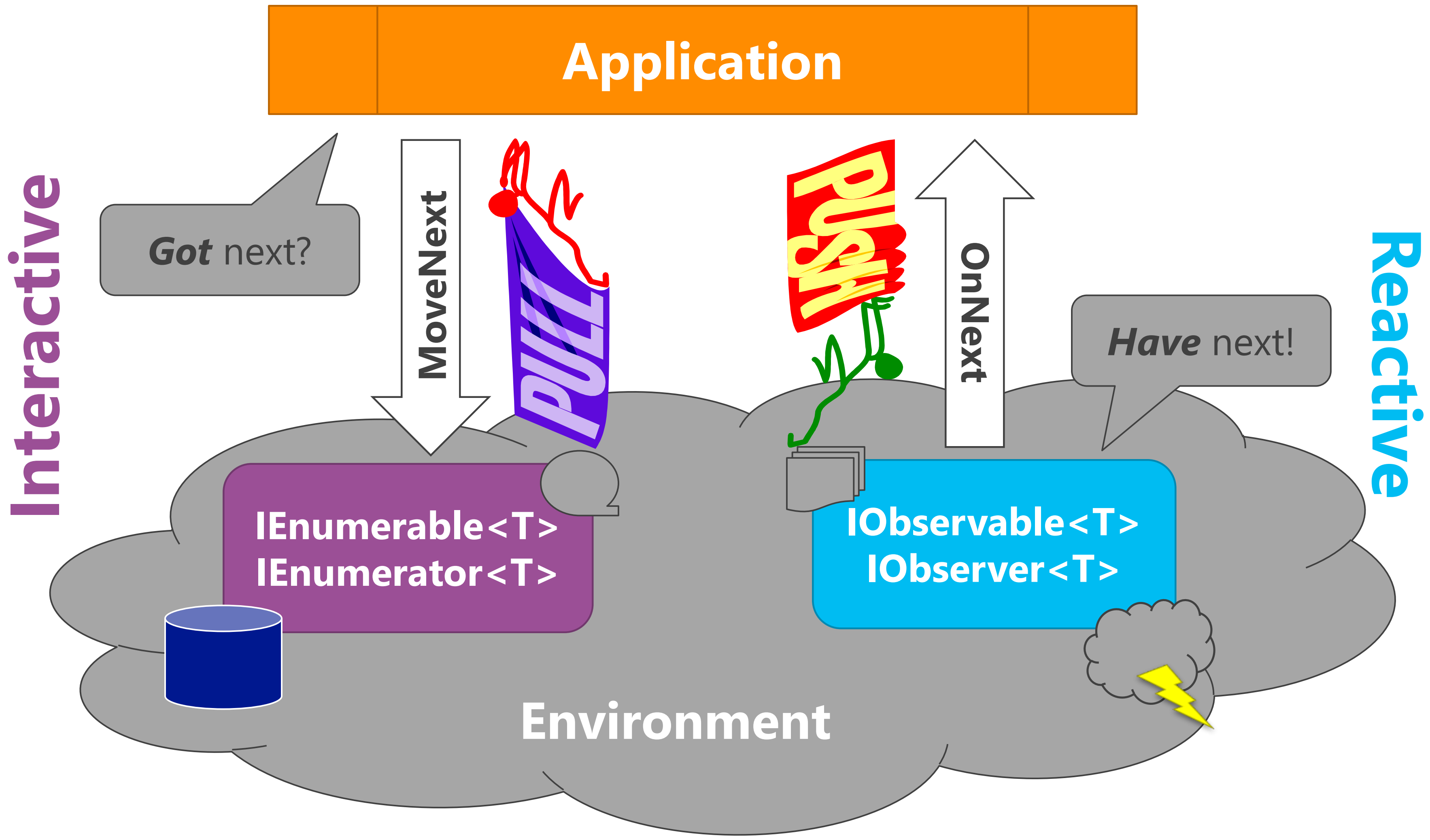
# Essential Interfaces

## Observables – a push-based world

```
interface IObservable<out T>
{
   IDisposable Subscribe(IObserver<T> observer);
}


interface IObserver<in T>
{
   void  OnNext(T value);
   void  OnError(Exception ex);
   void  OnCompleted();
}
```

# Essential Interfaces

## Summary – push versus pull



# Analogy

- Can compare to subscribing to a news paper
  - When calling subscribe:
    - You give the address of your letter box (the observer)
    - You get a way to cancel the subscription in the future
    - Asynchronous nature! You’re not blocked till the next news paper arrives.
  - The observer is the letter box:
    - OnNext happens when the postman drops the newsletter in it
    - OnError happens when someone breaks your letter box or the postman drops dead
    - OnCompleted happens when the newspaper publisher goes bankrupt

# Events as data sources

**In jQuery**

```
 $(document).ready(function () {
      $(document).mousemove(function (event) {
           $("<p/>").text("X: " + event.pageX+" Y: " + event.pageY)
                    .appendTo("#content");
      });
  });
```

**In Rx**

```
 $(document).ready(function () {

$(document).toObservable("mousemove").Subscribe(function(event){
        $("<p/>").text("X: " + event.pageX+" Y: " + event.pageY)
                 .appendTo("#content");
    });
 });
```

Like a Promise and an Array…

… but cancellable

# Future – will be part of ES2016 Standard

github.com/zenparsing/es-observable

# Now - RxJS (Reactive Extensions for JS)

# Angular 2 Form Controls can be watched using Observables

# Substribing to Textbox Value Changes

## Template

```
<input
  type="text"
      #symbol
      [ng-form-control]="searchText"
      placeholder="ticker symbol">
```

## Component

```
export class TypeAhead {
     searchText = new Control();
     constructor() {

   this.searchText.valueChanges
      .subscribe(...);
     }
}
```

```
this.searchText.valueChanges
    .debounceTime(200)
    .subscribe(...);
```

# TickerSearcher usin Angular 2 Http

```
class Searcher {
  constructor(private _http: Http){}
  get(val:string):Observable<any[]> {
      return this._http
        .get(`/stocks?symbol=${val}`)
        .map(res => res.json());
    }
}
```

```
this.searchText.valueChanges
    .debounceTime(200)
  .switchMap(text => searcher.get(val))
    .subscribe(tickers => {
    this.tickers = tickers;
  });
```

```
<li *ng-for="#tick of tickers">
  {{ticker.symbol}}
</li>
```

```
this.tickers =
 this.searchText.valueChanges
     .debounceTime(200)
   .switchMap(text => searcher.get(val))
```

```
<li *ng-for="#tick of tickers | async">
  {{ticker.symbol}}
</li>
```

## Classical Style Typeahead (Component - 26 LOC)

```
export class TypeAhead {
           searchText: string;
           searchTimeout: any;
           currentRequest: any;
           constructor() {}
           doSearch(text){
             var searchText = this.searchText;
             this.currentRequest = null;
             this.currentRequest = fetch(`server?symbol=${this.searchText}`)
             this.currentRequest
               .then(res => res.json())
               .then(tickers => {
                 if (this.searchText === searchText) {
                   this.tickers = tickers;
                 }
               });
           }

           searchChanged(){
             if(typeof this.searchTimeout !== 'number'){
               clearTimeout(this.searchTimeout);
               this.searchTimeout = null;
             }
             this.searchTimeout = setTimeout(() => {
               this.doSearch(this.searchText);
               this.searchTimeout = null;
             }, 500);
           }
         }
```

## Reactive Style Typeahead (Component - 11 LOC)

```
export class TypeAhead {
       ticker = new Control();
       tickers: Observable<any[]>;
       constructor(seacher:TickerSearcher) {
         this.tickers = this.searchText.valueChanges
           .debounceTime(200)
           .switchMap((val:string) =>
                  seacher.get(val));
       }
}
```

# Ajax with angular2/http

New Data Architecture using Reactive Programming and Observables

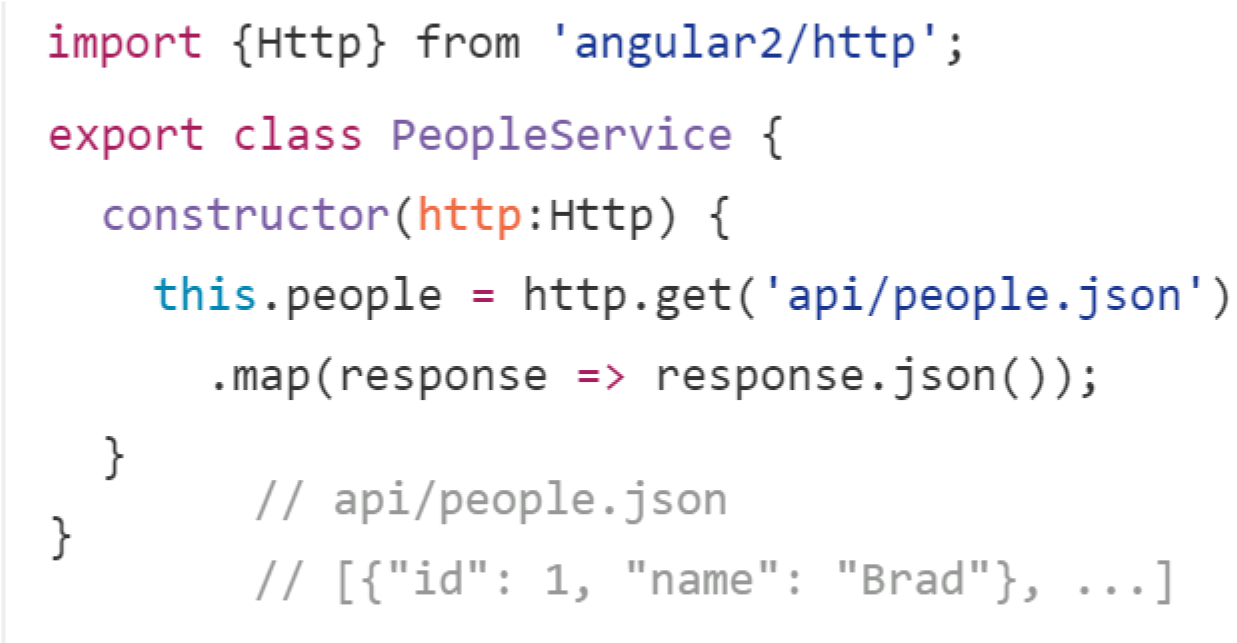
< Back

# Setting up angular2/http

- In order to the Http module we have to import *HTTP_BINDINGS* and include it using the component ***providers*** property or during bootstrap.

```
import {HTTP_BINDINGS} from 'angular2/http';
@Component({
  ...
  providers: [HTTP_BINDINGS]
})
class MyComponent { }          OR

bootstrap(App, [HTTP_BINDINGS]);
```

# Get Request Example



```
import {Http} from 'angular2/http';

export class PeopleService {

  constructor(http:Http) {

    this.people = http.get('api/people.json')

      .map(response => response.json());

  }
        // api/people.json
}
        // [{"id": 1, "name": "Brad"}, ...]
```

- http.get returns an Observable emitting Response objects
- ***.map*** *is used* to parse the result into a JSON object
- The result of ***map*** is also an ***Observable*** that emits a JSON object containing an Array of people.

# Post Example

```
postData(){
   var headers = new Headers();
   headers.append('Content-Type', 'application/json');

   this.http.post('http://www.syntaxsuccess.com/poc-post/',
     JSON.stringify({firstName:'Joe',lastName:'Smith'}),
     {headers:headers})
     .map((res: Response) => res.json())
     .subscribe((res:Person) => this.postResponse = res);
}
```

# Summary

Angular 2 introduces many innovations:

- Performance improvements
- Component Router
- Sharpened Dependency Injection (DI)
- Reactive Programming
- Async templating
- Server rendering (aka Angular Universal)
- Orchestrated animations

All topped with a solid tooling thanks to TypeScript + excellent testing support

# Resources

- Angular Cheat Sheet

https://angular.io/cheatsheet

- Tour of Heroes tutorial

https://angular.io/docs/ts/latest/tutorial/

http://angular.meteorhub.org/tutorials/angular2/

- angular2-education – useful links

https://github.com/cexbrayat/angular2-education

- Book

https://books.ninja-squad.com/angular2

https://ng-book.com/2