

ברוכים הבאים !

TypeScript & Angular 7

Shimon Dahan & Kobi Hari

shimond@any-techs.co.il

hari@applicolors.com





Jose Aguinaga

Follow

Web Engineer. Previously @numbrs, @plaidhq, currently @getflynt. Javascript, #people, startups, fintech, pr...

Oct 3, 2016 · 12 min read

How it feels to learn JavaScript in 2016



<https://hackernoon.com/how-it-feels-to-learn-javascript-in-2016-d3a717dd577f>



The road to Angular or evolution of web development





The first browsers

- NCSA Mosaic released on 1993 – first web browser to display images inline with documents text.
- Netscape came along in 1994
- Internet Explorer in 1995





Early phases – around documents

- URL's mostly revolved around static html files
 - <http://www.computer.com/root/folder/document.html>
- The Browser's job was to request a document, and translate HTML to visual presentation
- The server's job was to fetch the document and send it to the client

Mostly Static Content



The next phase – Dynamic content

<http://www.computer.com/root/folder/document.cgi>

<http://www.computer.com/servlet/doc>

- Hard to maintain – code that generates code
- Final document structure not clear from the code

```
package org.test;

import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
import java.io.*;

public class GreetingServlet extends HttpServlet {
    protected void doPost(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        String name = req.getParameter("name");
        String email = req.getParameter("email");
        String message = null;
        GregorianCalendar calendar = new GregorianCalendar();
        if (calendar.get(Calendar.AM_PM) == Calendar.AM)
            message = "Good Morning";
        else
            message = "Good Afternoon";
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        out.println("<html>");
        out.println("<body>");
        out.println("<p>" + message + ", " + name + "</p>");
        out.println("<p> Thanks for registering your email(" + email
            + ") with us.</p>");
        out.println("<p>- The Pro Java Team.</p>");
        out.println("</html>");
        out.println("</body>");
        out.close();
    }
}
```



The next step – Server Templates

also known as active server pages

- A page in HTML with combined parts that run on the server

```
3 </php
4 if ($this->pictures == null) {
5     ?>
6     <div class="warning">No pictures were found</div>
7     <?
8 } else {
9     if ($this->cover != null) {
10        ?>
11        Main picture:
12        
13        <?
14    }
15    foreach ($this->pictures as $picture) {
16        ?>
17        
18        <?
19    }
20 }
21 ?>
```




Asp.NET – in C#

JSP – in Java

<http://www.computer.com/root/folder/document.aspx>

<http://www.computer.com/root/folder/document.jsp>

The screenshot shows the Visual Studio IDE with the 'Client Objects & Events' window open. The code is an ASP.NET page with a login form. The rendered page below the code shows the form with the title 'User Account Tutorials'. A red box highlights the 'asp:TextBox ID="UserName"' in the code, and another red box highlights the 'Login' button. A third red box highlights the 'InvalidCredentialsMessage' label.

```
<?xml version='1.0' encoding="UTF-8"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP
<html>
<head><title>JSP Example</title></head>
<jsp:useBean id="exampleBean" class="com.sty
<jsp:include page="header.jsp"/>
<% String username = session.getAttribute("
    if (username!=null) {
        out.println("Hello" + username);
```

“Smart tags” or “Server Tags”:

- Tags that the server translates into another set of tags and server logic
- Finally allowing us to do some re-use



Along the way – the client gets smarter

- Javascript combined inside the pages
- Has access to the live DOM
- Allows to perform some ui operations without calling the server
- But when new data is needed, the server is called
- The server creates a new HTML page
- The new page replaces the old one completely



Time of Chaos

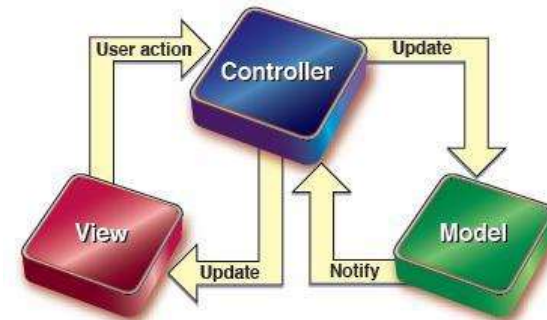
- Server code and client code, written in the same files
- Not always so clear which code runs where...
- View state passed to the server to help with continuation
- Template files with mixed logic
 - DB access
 - Authentication
 - Data manipulation
 - Business Logic
 - UI structure
- No clear Design Pattern

```
<script type="text/javascript">
    $(function () {
        $('#fileupload').fileupload({
            replaceFileInput: false,
            dataType: 'json',
            url: '<%= ResolveUrl("AjaxFileHandler.ashx") %>',
            done: function (e, data) {
                $.each(data.result, function (index, file) {
                    $('<p/>').text(file).appendTo('body');
                });
            }
        });
    });
</script>
<form id="form1" runat="server">
<input id="fileupload" type="file" name="file" multiple="multiple" />
</form>
```



The rise of Web MVC

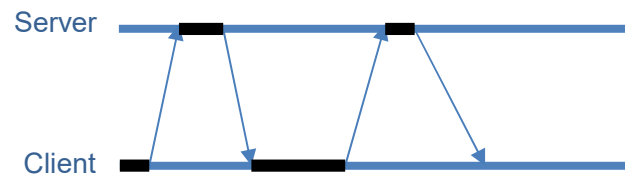
- URLs refer to Controller actions instead of direct page
- The controller performs the logic
- The controller **routes** to the view
- Only then, the view engine renders the template
- Still – this is all done on the server.
- If you want client side generated UI – you lose this





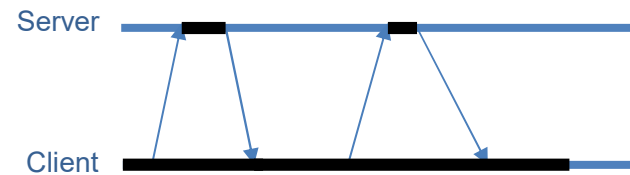
Along came AJAX

- Before Ajax



- Either the server is working, or the client,
- But not both.
- Every time a response comes back – the page is redrawn

- Ajax



- The javascript is making the calls to the server, and while it waits, the client remains active
- When results come – the javascript embeds them in the DOM – but the page is not refreshed



Rise of “Web Applications”

- Not just “pages” or “documents” anymore
- HTML is now used to draw UI – not documents
- Javascript is now used to run complete application logics, not just minor dynamics
- Lots of code that modifies the DOM – servlet style...



Javascript steps up

- New JS packages emerge every week – to help with DOM manipulation and Ajax calls
- Some try to allow for client side MVC or MVVM
- Some even try to allow Templates...
- Slowly – more and more UI related code moves to the client

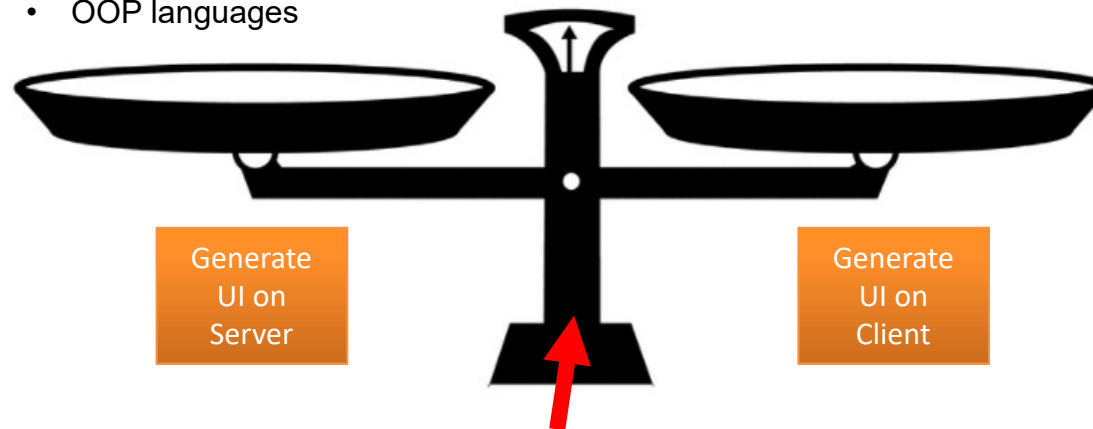




The UI Coding Dilemma

- Better Programming Languages
- Better Tooling
- Templates
- MVC supporting languages
- OOP languages

- Faster and More fluent
- Easier Session maintenance
- More natural Pattern



Neither option is perfect – but the worse is in the middle...
Some UI on the server and some on the client



The Move to Single page applications

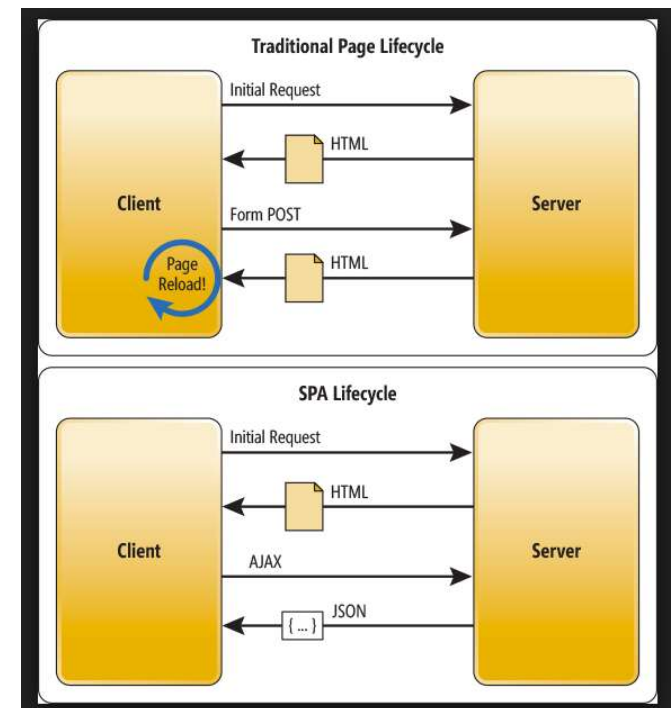
- Eventually the challenges involved in generating UI on the server and the fluid experiences drove towards client generated code.
- And since it's better to have one place to maintain (even if it is hard to maintain) than 2 places...
- The natural result was to try and do on the client, what we once did on the server.





Single page application means:

- The entire application is in one HTML page
- Usually the server provides an empty HTML page – and the Javascript draws the entire UI
- The server is still accessed for data and business logic operations
- The server returns JSON instead of HTML

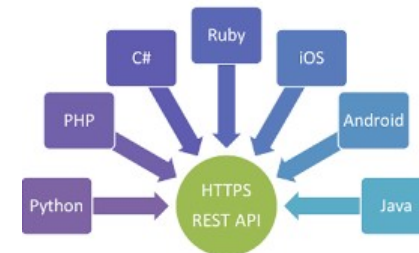




RESTful servers - Representational state transfer

RESTful API
GET PUT POST DELETE

- Deal with data state – not UI
- Over the years – conventions emerged for REST through HTTP
- Usually stateless – so the UI maintains the session
- Conventions define a uniform interface using HTTP verbs and URL formats
- Allowed for various UI platform to use the same data server





Angular + Typescript



- Aimed at SPA
- Revolves around Componentization of HTML
- Components have Templates
- “Smart Tags” are evaluated in JS on the browser
- Logic is written in Typescript – Modern Strongly typed OOP
- Strongly driven by best practices:
 - MVC and MVVM
 - Dependency Injection
 - TDD
 - Separation of Concerns
- An unusual Cooperation between Google and Microsoft

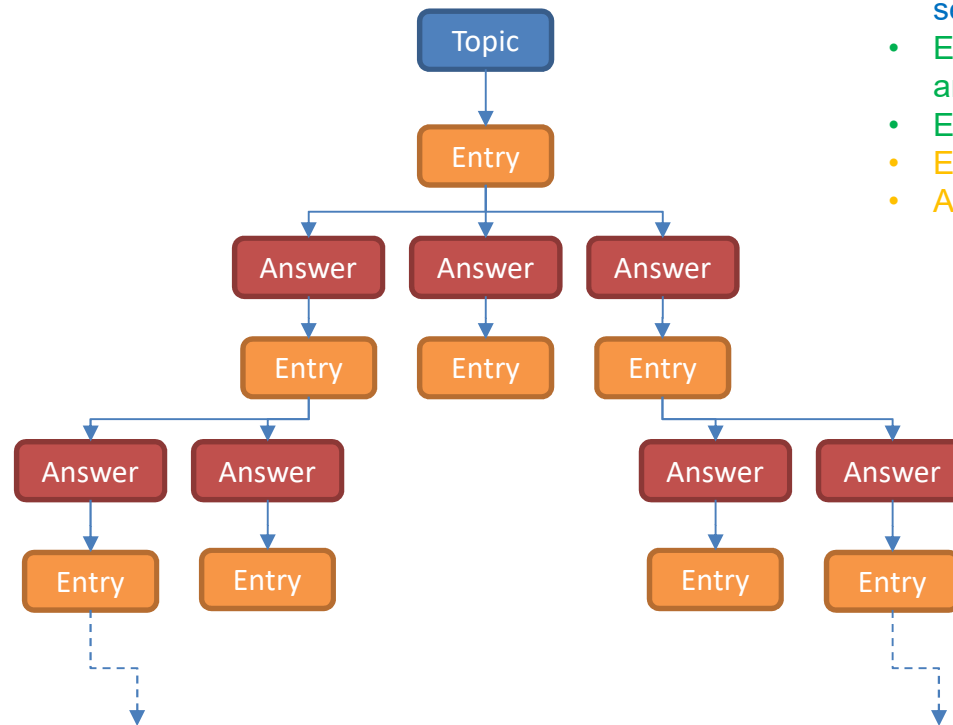


Today's Project: **MAKARA!!!???** or “a simple decision making support app”





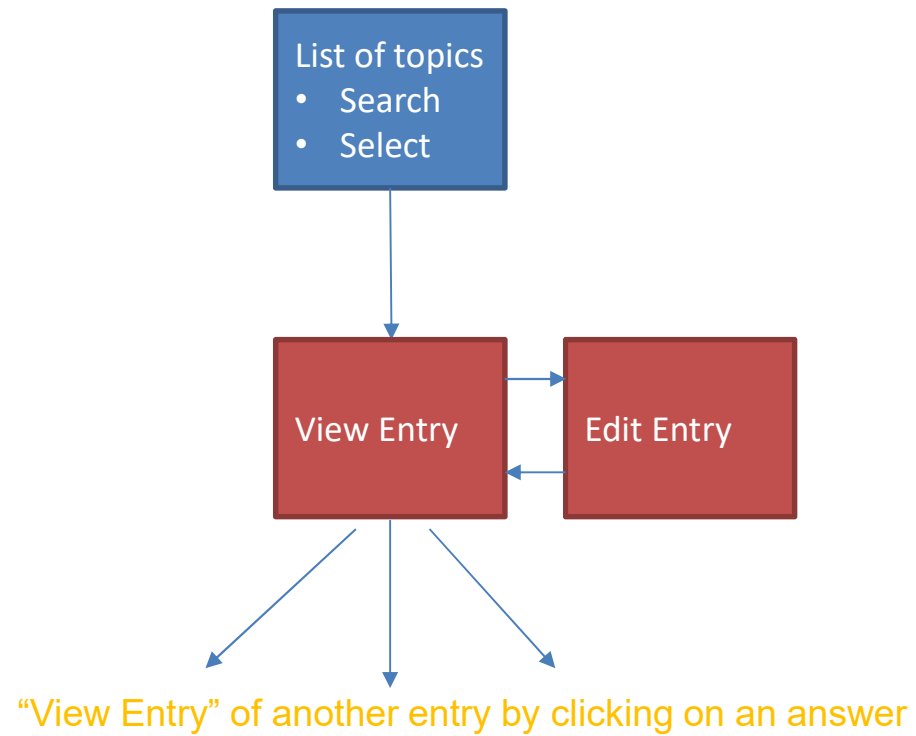
Data Structure



- Each topic leads to a “root” entry – the first separating question
- Entries have a question or title, description, and help
- Entries have 0..many answers
- Each Answer leads to a new Entry
- And so on...



Screens



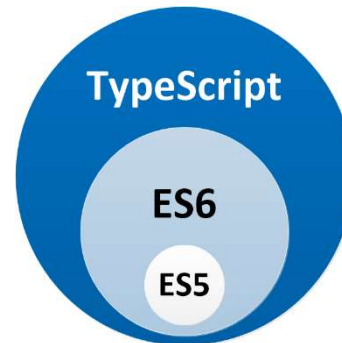


Agenda - Typescript

- What & Why
- Types
- Functions types
- Tooling
- Interfaces
- Generics



- TypeScript is a typed superset of JavaScript that compiles to plain JavaScript.
- Any browser.
- Any host.
- Any OS.
- Open source.





```
1 var Person = function (name, age, id) {
2   this.age = age;
3   this.id = id;
4   this.name = name;
5   this.isSaved = false;
6 }
7
8 //Prototype:
9 Person.prototype.save = function () {
10   var xmlReq = new XMLHttpRequest();
11   //COMBINA:
12   var $this = this;
13   xmlReq.onreadystatechange = function () {
14     // this.isSaved = true; // ERROR
15     $this.isSaved = true;
16   }
17   xmlReq.open("POST", "/api/people/" + this.id);
18   xmlReq.send(JSON.stringify(this));
19 }
20
21 Person.prototype.print = function () {
22   alert(this.name + " " + this.age);
23 }
24
25 var p = new Person("David", 12, 4432);
26 p.print();
27
```

plain Javascript



- Dynamic typing leads to Run Time Errors
- **clean code?!**
- JS is loosely typed.



Why I will never write
JavaScript again*



```
1 class Person {  
2  
3     private isSaved: boolean;  
4  
5     print() : void {  
6         alert(`${this.name} - ${this.age}`)  
7     }  
8  
9     save() : void {  
10        let xmlReq = new XMLHttpRequest();  
11        xmlReq.onreadystatechange = () => {  
12            this.isSaved = true;  
13        }  
14        xmlReq.open("POST", "/api/people/" + this.id);  
15        xmlReq.send(JSON.stringify(this));  
16    }  
17  
18 }  
19  
20 constructor(public name: string, public age: number, public id: number) {  
21  
22 }  
23 }  
24  
25 let p = new Person("David", 12, 4432);  
26 p.print();  
27 p.save();  
28
```

TypeScript



```
1 class Person {
2
3     private isSaved: boolean;
4
5     print(): void {
6         alert(`${this.name} - ${this.age}`)
7     }
8
9     save(): void {
10         if (!this.isSaved)
11         {
12             let xmlReq = new XMLHttpRequest();
13             xmlReq.onreadystatechange = () => {
14                 this.isSaved = true;
15             }
16             xmlReq.open(true, "/api/people/" + this.id);
17             xmlReq.send(JSON.stringify(this));
18         }
19         alert(xmlReq)
20     }
21 }
22
23 constructor(public name: string, public age: number, public id: number) {
24
25 }
26 }
27
28 let p = new Person(12, true, new XMLHttpRequest());
29 p.print();
30 p.save();
31
```

Arguments of type 'true'
is not assignable to
parameter of type string

Can not find parameter
'xmlReq'

Mismatch parameters
type



- Basic Types
 - Boolean
 - String
 - Number
 - Object
 - Array
 - Any
 - void



- Custom types
 - Enum
 - Class
 - Interface



Enum

Enums allow us to define a set of named numeric constants. An enum can be defined using the enum keyword.

```
1 enum Direction {  
2     Up = 1,  
3     Down,  
4     Left,  
5     Right  
6 }
```



```
1 var Direction;  
2 (function (Direction) {  
3     Direction[Direction["Up"] = 1] = "Up";  
4     Direction[Direction["Down"] = 2] = "Down";  
5     Direction[Direction["Left"] = 3] = "Left";  
6     Direction[Direction["Right"] = 4] = "Right";  
7 })(Direction || (Direction = {}));  
8
```



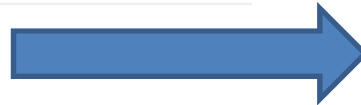

Classes

A class encapsulates data for the object. Typescript gives built in support for this concept called class. JavaScript ES5 or earlier didn't support classes. Typescript gets this feature from ES6.



Classes

```
1 class Product{
2
3   set Name(value: string){
4     this.name = value;
5   }
6
7   get Name() : string {
8     return this.name;
9   }
10
11  set Price(value: number){
12    this.price = value;
13  }
14
15  get Price() : number {
16    return this.price;
17  }
18
19  isExpensive() : boolean {
20    return this.price > 10000;
21  }
22
23  constructor(private name : string, private price : number) {
24
25  }
26
27
28 }
```

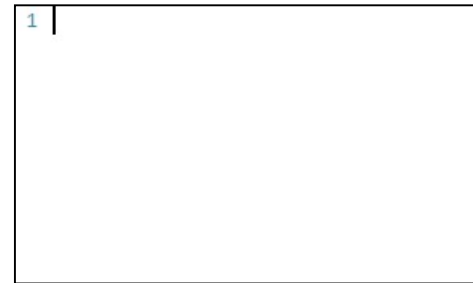


```
1 var Product = (function () {
2   function Product(name, price) {
3     this.name = name;
4     this.price = price;
5   }
6   Object.defineProperty(Product.prototype, "Name", {
7     get: function () {
8       return this.name;
9     },
10    set: function (value) {
11      this.name = value;
12    },
13    enumerable: true,
14    configurable: true
15  });
16   Object.defineProperty(Product.prototype, "Price", {
17     get: function () {
18       return this.price;
19     },
20     set: function (value) {
21       this.price = value;
22     },
23     enumerable: true,
24     configurable: true
25  });
26   Product.prototype.isExpensive = function () {
27     return this.price > 10000;
28   };
29   return Product;
30 }());
31
```



Interfaces

```
1 interface ICustomer{  
2     name: string;  
3     birthdate: Date;  
4     email: string;  
5     print(): void;  
6  
7 }
```





Generics

create a component that can work over a variety of types
rather than a single one

```
1 let arr: Array<string> = new Array<string>();  
2 arr.push  
   push (method) Array<string>.push(...items: string[]..  
   Appends new elements to an array, and returns the new leng...
```

```
1 let arr: Array<number> = new Array<number>();  
2 arr.push  
   push (method) Array<number>.push(...items: number[]..  
   Appends new elements to an array, and returns the new leng...
```



Generics - Custom

```
1 class Point<T> {  
2     x: T;  
3     y: T;  
4  
5     compare(other: Point<T>): boolean {  
6         return this.x == other.x && this.y == other.y;  
7     }  
8  
9 }  
10  
11  
12 |
```

```
class Point<T extends number> {  
    x: T;  
    y: T;  
  
    compare(other: Point<T>): boolean {  
        this.x.  
        return  
    }  
}
```

- toExponential (method) Number.toExponential(fractionalDigits)
- toFixed
- toLocaleString
- toPrecision
- toString
- valueOf



async Await

Allows an asynchronous, non-blocking method call to be performed in a similar way to an ordinary synchronous method call

```
1 function getData(callback) {  
2     window.setTimeout(() => {  
3         callback("{DATA}")  
4     }, 3000);  
5 }  
6  
7 function init() {  
8  
9     getData(function (data) {  
10         alert(data);  
11     });  
12 }  
13
```

```
17 function getDataPromise(): Promise<string> {  
18     return new Promise<string>((resolve, reject) => {  
19         window.setTimeout(x => {  
20             resolve("{DATA}");  
21  
22             }, 3000);  
23  
24     });  
25 }  
26  
27 async function init1() {  
28     let s: string = await getDataPromise();  
29     alert(s);  
30 }  
31
```



Angular - intro

- A **single-page application (SPA)** is a web application or web site that fits on a single web page.
- SPA is essentially an evolution of the MPA (multi-page application) and AJAX design pattern.
- SPA advantages over MPA:
 - Faster page loading times
 - Improved user experience because the data is loading in the background from server
 - No need to write the code to render pages on the server
 - Decoupling of front-end and back-end development
 - Simplified mobile development; you can reuse the same backend for web application and native mobile application

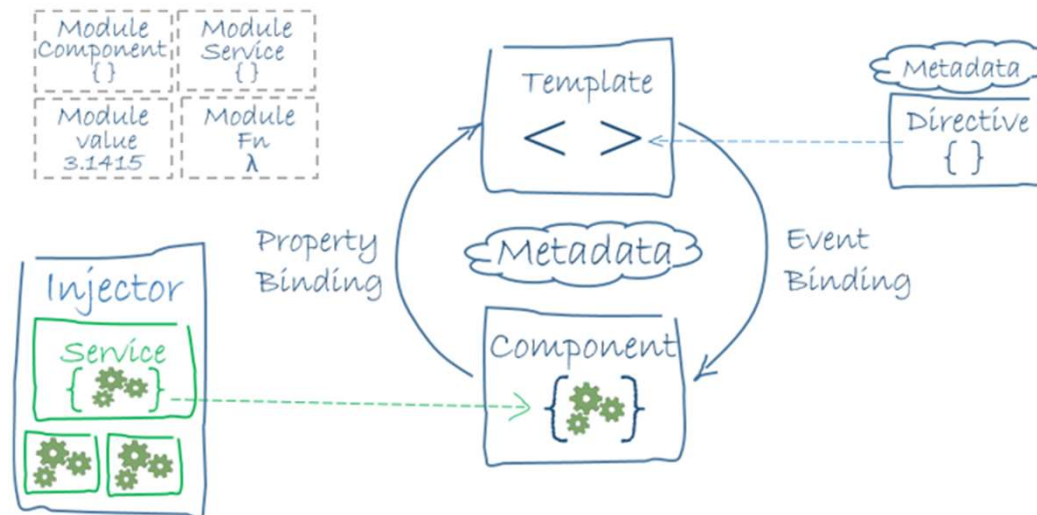


What is Angular?

- AngularJS is a complete JavaScript-based open-source front-end web application framework.
- Angular is the next generation of AngularJS.
- Why Angular?
 - Typescript
 - Angular is more modern, more capable, and easier for new developers to learn than Angular 1.x.
 - Better architecture - Angular 2 modules are far less coupled than Angular 1.x.
 - Better performance - Angular turns your templates into code that's highly optimized.



Angular architecture





Angular 3?!?!?



Angular-cli

Use angular cli –command line interface)

- <https://github.com/angular/angular-cli>
- Automatic change detection, can just change code while you are developing and your browser will automatically be refreshed
- Built in commands to create components, directives, pipes and services
- Run karma tests
- Automates build process
 - Ahead of time compilation
 - Joins all the files together and “tree-shakes” your code to make it as small as possible
 - Webpack



Angular-cli

```
> npm install -g @angular/cli  
> ng new my-dream-app  
> cd my-dream-app  
> ng serve
```

Angular CLI

A command line interface for Angular

GET STARTED



Angular architecture - NgModule

- **Angular Modules** help organize an application into cohesive blocks of functionality.
- NgModule takes a metadata object that tells Angular how to compile and run module code.
- The Angular Modules concept is introduced with Angular 2 RC5.
- Modules helps to organize your application into blocks of functionality.
- Modules consolidate components, directives and pipes into block of functionality.
- Angular Module is mainly a template compilation context but it also helps to define a public API for a subset of functionality as well as help with the dependency injection configuration of our application.



Angular architecture - Components

- Components are the main way we build and specify elements and logic on the page.
- Component is a basic unit in Angular .
- Usually a component will include: name (selector), logic class, html5 and style.

```
1 import { Component, OnInit } from '@angular/core';
2
3 @Component({
4   selector: 'app-my-sample',
5   templateUrl: './my-sample.component.html',
6   styleUrls: ['./my-sample.component.scss']
7 })
8 export class MySampleComponent implements OnInit {
9
10   constructor() { }
11
12   ngOnInit() {
13   }
14
15 }
```



Component Template

Template syntax

One-way, two-way binding

Built in directives

Pipes

Events

Template reference variables

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

@Component({
  selector: 'app-my-sample',
  templateUrl: './my-sample.component.html',
  styleUrls: ['./my-sample.component.scss']
})
export class MySampleComponent implements OnInit {
  userName: string = "Shimon Dahan";

  constructor() { }

  ngOnInit() {
  }

}

<h1>Your name is {{userName}}</h1>
```



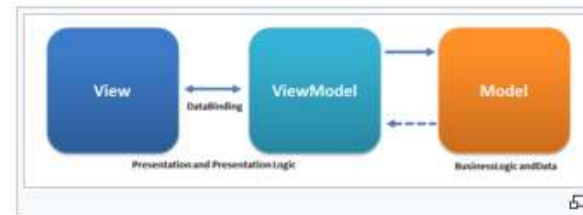
MVVM

Model-view-viewmodel

From Wikipedia, the free encyclopedia

Model-view-viewmodel (MVVM) is a software architectural pattern.

MVVM facilitates a separation of development of the graphical user interface – be it via a markup language or GUI code – from development of the business logic or back-end logic (the *data model*). The *view model* of MVVM is a value converter,^[1] meaning the view model is responsible for exposing (converting) the *data objects* from the model in such a way that objects are easily managed and presented. In this respect, the view model is more model than view, and handles most if not all of the view's display logic.^[1] The view model may implement a *mediator pattern*, organizing access to the back-end logic around the set of *use cases* supported by the view.







MVVM advantages

- Maintainability
- Testability
- Extensibility
- Abstraction and separation

Data Binding

Data direction	Syntax	Binding type
One-way from data source to view target	<pre>{{expression}}</pre> <pre>[target] = "expression"</pre> <pre>bind-target = "expression"</pre> 	Interpolation Property Attribute Class Style
One-way from view target to data source	<pre>(target) = "statement"</pre> <pre>on-target = "statement"</pre> 	Event
Two-way	<pre>[(target)] = "expression"</pre> <pre>bindon-target = "expression"</pre> 	Two-way



Angular architecture - Directives

- There are three kinds of directives in Angular:
 - Components
 - Structural directives - change the DOM layout by adding and removing DOM elements. NgFor and NgIf are two familiar examples.
 - Attribute directives - Attribute directive can change the appearance or behavior of an element. For example: The built-in NgStyle directive.
- A *Component* is really a **directive with a template**. It's the most common of the three directives and we tend to write lots of them as we build applications.



Built in directives

NgIf

NgSwitch

NgFor



NgIf

- We can add an element subtree (an element and its children) to the DOM by binding an NgIf directive to a truthy expression.
- Visibility and NgIf are not the same, Hiding a subtree is quite different from excluding a subtree with NgIf.
- The show/hide technique is probably fine for small element trees. We should be wary when hiding large trees, NgIf may be the safer choice.

```
@Component({
  selector: 'app-my-sample',
  templateUrl: './my-sample.component.html',
  styleUrls: ['./my-sample.component.scss']
})
export class MySampleComponent implements OnInit {
  userName: string = "Shimon Dahan";
  isLoggedIn: boolean = false;
  constructor() {}

  ngOnInit() {
  }
}
```

```
<h1>Your name is {{userName}}</h1>

<div *ngIf="isLoggedIn">
  <a href="/logout">Logout</a>
</div>
```



NgSwitch

- We bind to NgSwitch when we want to display one element tree (an element and its children) from a set of possible element trees, based on some condition. Angular puts only the selected element tree into the DOM.
- Three collaborating directives are at work here:
 - ngSwitch: bound to an expression that returns the switch value.
 - ngSwitchCase: bound to an expression returning a match value.
 - ngSwitchDefault: a marker attribute on the default element.

```
<h1>Your name is {{userName}}</h1>
<div ngSwitch="gender">
  
  
  
</div>
```



NgFor

- NgFor is a repeater directive — a way to customize data display.
- The ngFor directive supports an optional index that increases from 0 to the length of the array:
- NgForTrackBy – The ngFor directive has the potential to perform poorly, especially with large lists. We can improve the performance by using trackby:

```
<div *ngFor="let person of people">{{person.name}}</div>
```



Shai
Dani
Shimon
Ania
Erick



Angular architecture - Pipes(|)

- The result of an expression might require some transformation before we're ready to use it in a binding. For example, we might want to display a number as a currency, force text to uppercase, or filter a list and sort it.
- Angular comes with a stock of pipes such as DatePipe, UpperCasePipe, LowerCasePipe, CurrencyPipe, and PercentPipe. They are all immediately available for use in any template. For example:

```
<h1>Your name is {{userName | uppercase}}</h1>
```




Built-in pipes

```
<div class="pipe">
  <h2>Date Pipe</h2>
  <div>
    {{date | date:'yMMMMEEEEd'}}
  </div>
  <h2>Number Pipe</h2>
  <div>
    {{4.566 | number:'1.2-2'}}
  </div>
  <h2>Currency Pipe</h2>
  <div>
    {{15.99 | currency:'EUR':true:'1.0-0'}}
  </div>
  <h2>Stat }};
  <div>
    {{randomData | async}}
  </div>
</div>
```

```
randomData = new Promise((resolve, reject) => {
  setTimeout(() => resolve('Random data!'), 1000);
});
```

Date Pipe

Tuesday, August 30, 2016

Number Pipe

4.57

Currency Pipe

€16

Stateful Pipe

Random data!



Custom pipes

- We can write our own custom pipes. Here's a custom pipe named CapitalizePipe that capitalize the first letter of each word.

```
import { Pipe, PipeTransform } from '@angular/core';

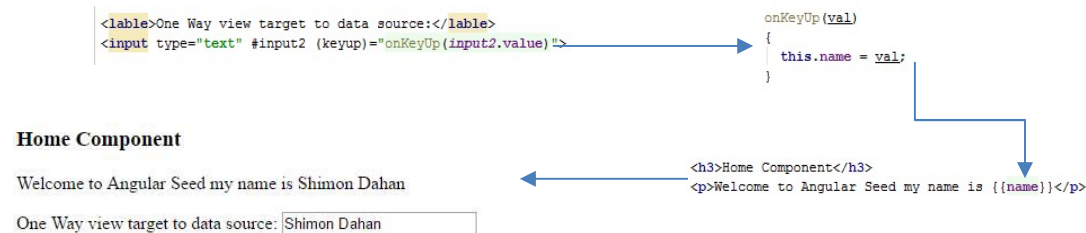
@Pipe({name: 'capitalize'})
export class CapitalizePipe implements PipeTransform {
  transform(value: string, args: string[]): any {
    if (!value) return value;

    return value.replace(/\w\S*/g, function(txt) {
      return txt.charAt(0).toUpperCase() + txt.substr(1).toLowerCase();
    });
  }
}
```



Events

- Users don't just stare at the screen. They enter text into input boxes. They pick items from lists. They click buttons. Such user actions may result in a flow of data in the opposite direction: *from an element to the component*.
- The only way to know about a user action is by listening for events:





Event Emmiter

- Event Emmitters are used by directives and components to emit custom Event.
- A component creates an EventEmitter and exposes it as a property.
- The component calls `EventEmitter.emit(payload)` to fire an event, passing in a message payload that can be anything.

```
@Output()
personSelected: EventEmitter<string>;

onClick(){
    this.personSelected.emit("Person selected.");
}

constructor() {
    this.personSelected = new EventEmitter<string>();
}

ngOnInit() {
}
```



Template reference variables

- A template reference variable is a reference to a DOM element or directive within a template.
- We can reference a template reference variable on the same element, on a sibling element, or on any child elements.

```
<!-- phone refers to the input element; pass its `value` to an event handler -->  
<input #phone placeholder="phone number">  
<button (click)="callPhone(phone.value)">Call</button>  
  
<!-- fax refers to the input element; pass its `value` to an event handler -->  
<input ref-fax placeholder="fax number">  
<button (click)="callFax(fax.value)">Fax</button>
```



Styles array

- We can add a styles array property to the @Component decorator. Each string in the array defines the CSS.

```
@Component({  
  selector: 'app-my-sample',  
  templateUrl: './my-sample.component.html',  
  styles: ['h1 {color : red}', 'p {text-decoration:underline}']  
})  
|
```



StyleUrls array

- We can load styles from external CSS files by adding a styleUrls attribute into a component's @Component decorator:

```
@Component({  
  selector: 'app-my-sample',  
  templateUrl: './my-sample.component.html',  
  styleUrls: ['./my-sample.component.scss']  
})
```



Directives and Components Life cycle hooks

Hook	Purpose
ngOnInit	Initialize the directive/component after Angular initializes the data-bound input properties.
ngOnChanges	Respond after Angular sets a data-bound input property. The method receives achanges object of current and previous values.
ngDoCheck	Detect and act upon changes that Angular can't or won't detect on its own. Called every change detection run.
ngOnDestroy	Cleanup just before Angular destroys the directive/component. Unsubscribe observables and detach event handlers to avoid memory leaks.



Components only LifeCycle hooks

Hook	Purpose
ngAfterContentInit	After Angular projects external content into its view.
ngAfterContentChecked	After Angular checks the bindings of the external content that it projected into its view.
ngAfterViewInit	After Angular creates the component's view(s).
ngAfterViewChecked	After Angular checks the bindings of the component's view(s).



Lifecycle sequence

Hook	Timing
ngOnChanges	before ngOnInit and when a data-bound input property value changes.
ngOnInit	after the first ngOnChanges.
ngDoCheck	during every Angular change detection cycle.
ngAfterContentInit	after projecting content into the component.
ngAfterContentChecked	after every check of projected component content.
ngAfterViewInit	after initializing the component's views and child views.
ngAfterViewChecked	after every check of the component's views and child views.
ngOnDestroy	just before Angular destroys the directive/component.



Angular architecture - Services

- Every application is composed of a lot of subsystems that do different things like logging, data access, caching, specific application domain knowledge, etc.
- Angular uses the concept of **services**, an Angular service is a *class* that encapsulates some sort of functionality and provides it as a service for the rest of your application.
- Angular takes advantage of dependency injection and inject the service through the component's constructor.



Services

- When multiple components will need access to some data and we don't want to copy and paste the same code over and over again.
- Instead, we'll create a single reusable data service and learn to inject it in the components that need it.
- Refactoring data access to a separate service keeps the component lean and focused on supporting the view. It also makes it easier to unit test the component with a mock service.
- Because data services are invariably asynchronous, we'll explain first the concept of Promises.

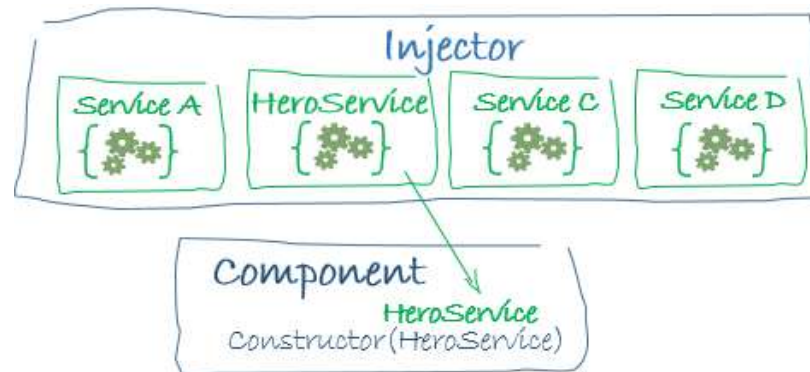


Angular architecture - dependency injection

- The DI in Angular basically consists of three things:
 - **Injector** - The injector object that exposes APIs to us to create instances of dependencies.
 - **Provider** - A provider is like a recipe that tells the injector **how** to create an instance of a dependency. A provider takes a token and maps that to a factory function that creates an object.
 - **Dependency** - A dependency is the **type** of which an object should be created.



Angular architecture - dependency injection





RxJS

- RxJS ("Reactive Extensions") is a 3rd party library, endorsed by Angular, that implements the asynchronous observable pattern.
- All the Developer Guide samples have installed the RxJS npm package because observables are used widely in Angular applications.
- We especially need it when working with the HTTP client. And Angular took a critical extra step to make RxJS observables usable.



Observable vs Promise

- A Promise handles a single callback when an async operation completes or fails.
- An Observable is like a Stream (in many languages) and allows to pass zero or more events where the callback is called for each event.
- Often Observable is preferred over Promise because it provides the features of Promise and more.
- Also: Observables are cancellable.



Routing

- The Angular Component Router enables navigation from one view to the next as users perform application tasks.
- The browser is a familiar model of application navigation. We enter a URL in the address bar and the browser navigates to a corresponding page.
- We click links on the page and the browser navigates to a new page. We click the browser's back and forward buttons and the browser navigates backward and forward through the history of pages we've seen.



Router imports

- The Angular Component Router is an service that presents a particular component view for a given URL.
- It is in its own library package, `@angular/router`. We import what we need from it as we would from any other Angular package.

```
import {RouterModule} from "@angular/router";

@NgModule({
  declarations: [AppComponent, About, RepoBrowser, RepoList, RepoDetail, Home],
  imports      : [BrowserModule, FormsModule, HttpModule, RouterModule.forRoot(rootRouterConfig)],
  providers    : [Github, {provide: LocationStrategy, useClass: HashLocationStrategy}],
  bootstrap    : [AppComponent]
})
```



Router Configuration

The application will have one router. When the browser's URL changes, the router looks for a corresponding Route from which it can determine the component to display.

```
export const rootRouterConfig: Routes = [
  {path: '', redirectTo: 'home', terminal: true},
  {path: 'home', component: Home,
  children: [
    {
      path: '',
      redirectTo: 'products',
    },
    {
      path: 'products',
      component: ProductsModuleComponent,
    },
    {
      path: "productEdit/:id",
      component: ProductEditModuleComponent,
    }
  ]},
  {path: 'about', component: About}
];
```



Router Outlet

Given this configuration, when the browser URL for this application becomes /about, the router matches that URL to the Route path /about and displays the About Component in a the RouterOutlet that we've placed in the host view's HTML.

```
<div class="mainContainer">
  <div>
    <headerControl></headerControl>
  </div>
  <main>
    <router-outlet></router-outlet>
  </main>
  <footer>
    <p>Shai Vashdi @ 2016</p>
  </footer>
</div>
```





Router Links

We add a RouterLink directive to the anchor tag. Since we know our link doesn't contain any dynamic information, we can use a one-time binding to our route path:

```
<nav>
  <a [routerLink]="['/']">
    Home
  </a>
  |
  <a [routerLink]="['/about']">
    About
  </a>
</nav>
```



Router Navigation

We can use the router service in order to navigate dynamically in the code:

```
this.router.navigate(['home/products']);

this.router.navigate(['home/productEdit', this.selectedProduct.id]);

export const rootRouterConfig: Routes = [
  {path: '', redirectTo: 'home', terminal: true},
  {path: 'home', component: Home,
    children: [
      {
        path: '',
        redirectTo: 'products',
      },
      {
        path: 'products',
        component: ProductModuleComponent,
      },
      {
        path: "productEdit/:id",
        component: ProductEditModuleComponent,
      },
    ]},
  {path: 'about', component: About}
];
```



Angular Testing – Unit test

- Jasmine
 - Jasmine is a behavior-driven development framework for testing JavaScript code.
- Karma
 - Karma allows us to run JavaScript code within a browser like Chrome or Firefox,
- TestBed
 - primary api for writing unit tests for Angular applications and libraries.



Angular Testing

- Use “spec” file : “my-sample.component.spec.ts”

```
describe('App: myApp', () => {  
  it('should expect true to be true', () => {  
    expect(true).toBe(true);  
  });  
});
```




Angular Testing – Unit test

```
import { TestBed, async } from '@angular/core/testing';

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

describe('AppComponent', () => {
  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [
        AppComponent
      ],
    }).compileComponents();
  }));

  it('should create the app', async(() => {
    const fixture = TestBed.createComponent(AppComponent);
    const app = fixture.debugElement.componentInstance;
    expect(app).toBeTruthy();
  }));
});
```

Configure
test module

Check if
“app”
creates



Angular Testing – Unit test

```
it(`should have as title 'app works!'`, async(() => {  
  const fixture = TestBed.createComponent(AppComponent);  
  const app = fixture.debugElement.componentInstance;  
  expect(app.title).toEqual('app works!');  
}));  
  
it('should render title in a h1 tag', async(() => {  
  const fixture = TestBed.createComponent(AppComponent);  
  fixture.detectChanges();  
  const compiled = fixture.debugElement.nativeElement;  
  expect(compiled.querySelector('h1').textContent).toContain('app works!');  
}));  
});
```

Check if element
"inner text" ==
"app works!"

Check if title
property ==
"app works!"



Angular Testing - Run unit test

- Running unit tests
 - ng test

The screenshot shows the Karma v1.4.1 interface. At the top, a green bar indicates 'Karma v1.4.1 - connected' with a 'DEBUG' button. Below this, it states 'Chrome 58.0.3029 (Windows 10 0.0.0) is idle'. The Jasmine 2.5.2 section shows 'finished in 0.291s' and '3 specs, 0 failures'. A list of three specs for 'AppComponent' is shown: 'should create the app', 'should have as title 'app works!'', and 'should render title in a h1 tag'. A large green 'app works!' message is displayed. At the bottom, a black terminal window shows the following logs:

```
17 06 2017 23:30:31.089:INFO [karma]: Karma v1.4.1 server started at http://0.0.0.0:9876/
17 06 2017 23:30:31.095:INFO [launcher]: Launching browser Chrome with unlimited concurrency
17 06 2017 23:30:31.200:INFO [launcher]: Starting browser Chrome
17 06 2017 23:30:33.419:INFO [Chrome 58.0.3029 (Windows 10 0.0.0)]: Connected on socket ISeU0PJv8_g6deRVAAAA with id 35741001
Chrome 58.0.3029 (Windows 10 0.0.0): Executed 3 of 3 SUCCESS (0.143 secs / 0.313 secs)
```



Angular Testing – E2E

- Interact with the application like a user would and evaluate if the app seems to work from the “outside”

```
└─ e2e
   ├── app.e2e-spec.ts
   ├── app.po.ts
   └── tsconfig.e2e.json
```

```
import { browser, element, by } from 'protractor';

export class MyProjectPage {
  navigateTo() {
    return browser.get('/');
  }

  getParagraphText() {
    return element(by.css('app-root h1')).getText();
  }
}
```



Angular Testing – E2E

```
import { MyProjectPage } from './app.po';

describe('my-project App', () => {
  let page: MyProjectPage;

  beforeEach(() => {
    page = new MyProjectPage();
  });

  it('should display message saying app works', () => {
    page.navigateTo();
    expect(page.getParagraphText()).toEqual('app works!');
  });
});
```



Angular Testing – E2E

- Run e2e
– ng e2e

```
[23:40:58] I/launcher - Running 1 instances of WebDriver
[23:40:58] I/direct - Using ChromeDriver directly...
Spec started

my-project App
  ✓ should display message saying app works

Executed 1 of 1 spec SUCCESS in 2 secs.
[23:41:05] I/launcher - 0 instance(s) of WebDriver still running
[23:41:05] I/launcher - chrome #01 passed
```

Thanks :)
See you next year