



# MODERN WEB DEVELOPMENT WITH TYPESCRIPT & ANGULAR

## Alain Chautard (or just AI)



1.

Google Developer Expert in  
Web technologies / Angular

2.

Java developer since 2006

3.

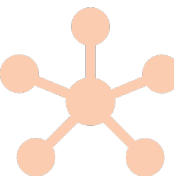
Angular JS addict since 2011

4.

Web consultant (60%) / trainer  
(40% of the time)

5.

Organizer of the Sacramento  
Angular Meetup group



# ? Quick Poll

- How many of you are Java developers? C#, .Net?
- How many of you are developers? Full-stack? Back-end?
- Any experience with Javascript? TypeScript? Node.JS?
- jQuery?
- Any other Javascript framework?

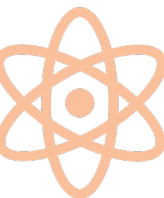


# How we're going to work

- Your questions are welcome, anytime!
- Being a web developer requires constant learning
- My goal is to give you the tools to work efficiently with web technologies - We're going to practice a lot!
- As a result, we will be going through some online documentation when needed



- Repository for all labs code + solutions:  
<https://github.com/alcfeoh/ng2-training>
- Link to these slides:  
<http://bit.ly/web-TS-NG>




# Today we're going to build...

[License Plate Store](#) [Home](#) [My cart](#)

Welcome to our store

Browse our collection of License Plates below

2008 Georgia license plate




Ad occaecat ex nisi reprehenderit dolore esse. Excepteur laborum fugiat sint tempor et in magna labore quis exercitation consequat nulla tempor occaecat. Sit cillum deserunt eiusmod proident labore mollit. Cupidatat do ullamco ipsum id nisi mollit pariatur nulla dolor sunt et nostrud qui.

\$8

Add to cart >

2015 New Jersey license plate




A beautiful license plate from the Garden State. Year is 2015.

\$11

Add to cart >

2013 California My Tahoe license plate



Sunt irure nisi excepteur in ea consequat ad aliqua. Lorem duis in duis nisi sit. Cillum eiusmod ipsum mollit veniam consectetur ex eiusmod nisi laborum amet anim mollit non nulla. Lorem ea est exercitation nostrud consectetur officia laborum fugiat sint. Nostrud consequat magna officia minim et aute nostrud.

\$9

Add to cart >





## Introduction to Typescript

Generics and  
Advanced  
types in  
Typescript

### Outline for today

01

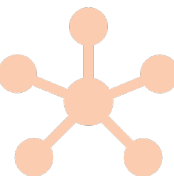
03

02

04

Typescript variables and  
object oriented features

Modules in Typescript  
+ important ES6 features



## Section 1



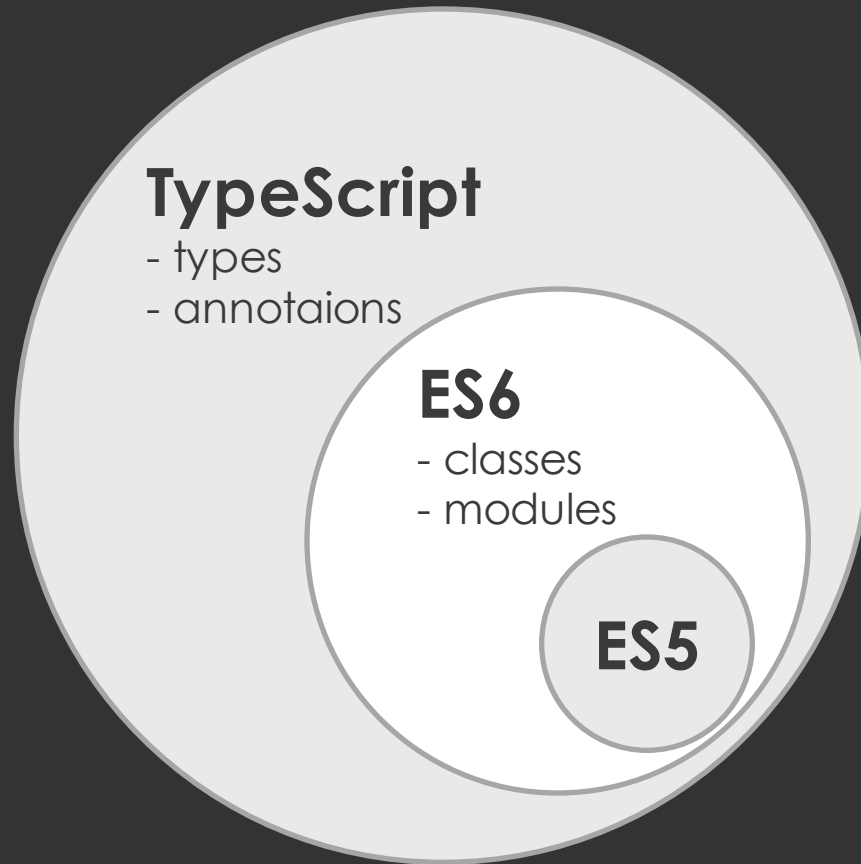
# Introduction to Typescript



30 min

# TypeScript

- TypeScript is a strict superset of ES6 (tomorrow's Javascript) which is the next iteration of ES5 (today's Javascript)
- Any valid Javascript code is also valid Typescript code
- Browser independent because runs ES5 Javascript



# Lab: 5-minute setup

# Get TypeScript

## Node.js

The command-line TypeScript compiler can be installed as a Node.js package.

### INSTALL

```
npm install -g typescript
```

### COMPILE

```
tsc helloworld.ts
```

## Visual Studio



Visual Studio 2015



Visual Studio 2013



Visual Studio Code

## And More...



Sublime Text



Atom



Eclipse



Emacs



WebStorm



Vim



# Lab #0: Hello, World

- Locate the file: **0-greeting.ts** in your lab folder
- Compile it on your computer:  
**tsc 0-greeting.ts**
- Let's take a look at the compiled code
- Now open **index.html** in your favorite browser and see the end result



## Wait, this was just Javascript

- Let's add a type declaration to the parameter of our function  
- now it's not JavaScript anymore:

```
function sayHello(person : string) {  
    return "Hello, " + person;  
}
```

```
var user = "World";
```

```
document.body.innerHTML = sayHello(user);
```





# TypeScript and types

- Typing is optional and can be used anywhere:

```
function sayHello(person : string) : string {  
    return "Hello, " + person;  
}
```

```
var user : string = "World";
```

```
document.body.innerHTML = sayHello(user);
```



# TypeScript and types

- Typing is optional and can be used anywhere:

```
function sayHello(person : string) : string {  
    return "Hello, " + person;  
}
```

```
var user : string = "World";
```

```
document.body.innerHTML = sayHello(user);
```



Lab #1:



# Usefulness of types

- Open **1-greeting.ts**
- Compile it on your computer:  
**tsc 1-greeting.ts**
- Fix the code so that it compiles and try again+



# Basic Types

- boolean, number, string
- Array:

```
var list : number[] = [1,2,3];
```

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

- Tuples:

```
// Declare a tuple type
let x: [string, number];
// Initialize it
x = ["hello", 10]; // OK
// Initialize it incorrectly
x = [10, "hello"]; // Error
```



# Basic Types

- Enums:

```
//A list of typed constants
enum Color {Red, Green, Blue}
let c: Color = Color.Green;
```

Enums start their numbering at 0, but we can change that:

```
// Custom values for each instance
enum Color {Red = 1, Green = 2, Blue = 4}
```

- Any and Void:

```
// Any type works!
let notSure: any = 4;

// This function does not return anything
function warnUser(): void {
    alert("This is my warning message");
}
```



# Iterators and loops: for...of vs for...in

- One returns the keys while the other returns the values

```
let list = [4, 5, 6];

for (let i in list) {
  console.log(i); // Displays "0", "1", "2"
}

for (let i of list) {
  console.log(i); // Displays "4", "5", "6"
}
```



Introduction to  
Typescript

Generics and  
Advanced  
types in  
Typescript

**Outline  
for today**

01

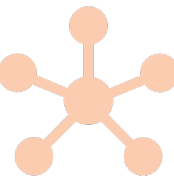
03

02

04

**Typescript variables and  
object oriented features**

Modules in Typescript  
+ important ES6 features





Section 2



# Typescript variables and object oriented features



30 min

# Variable Declarations: let vs var

- var is the default JS way and has a lot of scope specificities
- let has block-scoping and was introduced with ES6

```
{
  var x = 10;
  console.log("x inner scope: " + x); // Displays 10
}
console.log("x outer scope: " + x); // Displays 10

{
  let y = 20;
  console.log("y inner scope: ", y); // Displays 20
}
console.log("y outer scope: ", y); // TypeScript compile error
}
```

TS2304: Cannot find name 'y'.



# Variable Declarations: let and const

- Const is similar to let but only allows one assignment

```
{  
    const z = 20;  
    z = 50; // Error, a constant cannot be reassigned  
}
```

<https://www.typescriptlang.org/docs/handbook/variable-declarations.html>



# Interfaces

- Here is an interface that describes objects that have a `firstName` and a `lastName`, as well as a **`toString`** method:

```
interface Person {  
    firstName : string,  
    lastName : string,  
    toString() : string  
}
```

**`implements`** is not required. Having the right “shape” is enough.

- Optional properties can be defined with “?”

```
interface Person {  
    firstName? : string,  
    lastName : string  
}
```



# Interfaces

- Interfaces can also define the structure of a function:

```
interface SearchFunc {  
    (source: string, subString: string): boolean;  
}
```

Names of the params don't need to match in order to compile



# Interfaces can extend interfaces

- An interface can extend one or more interfaces

```
interface Shape {  
    color: string;  
}  
  
interface PenStroke {  
    penWidth: number;  
}  
  
interface Square extends Shape, PenStroke {  
    sideLength: number;  
}
```



# Interfaces can define indexable types

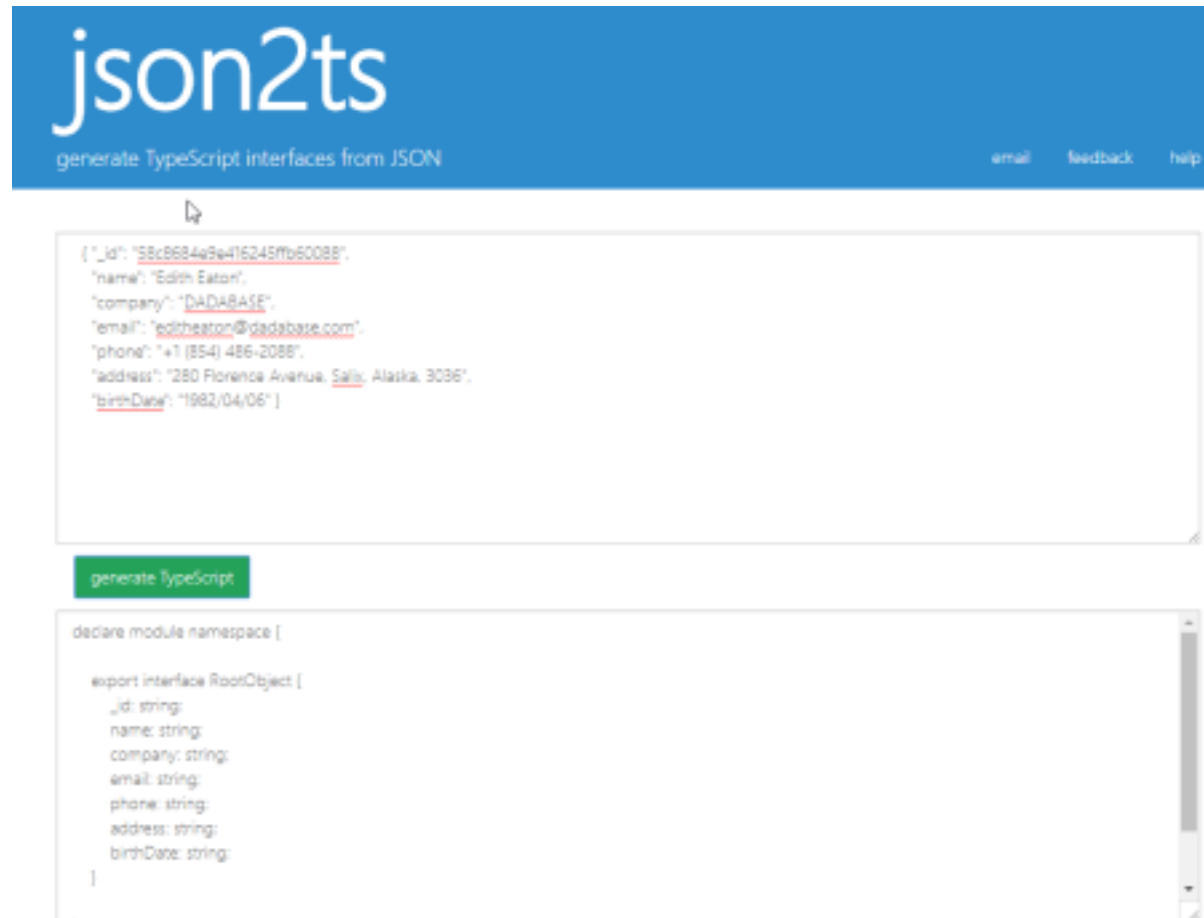
- We can define types that use indexes, like **a[1]** or **ageMap["Dan"]**

```
interface StringArray {  
    [index: number]: string;  
}  
  
let myArray: StringArray;  
myArray = ["Bob", "Fred"];  
  
let myStr: string = myArray[0];
```



# How to generate types from JSON data

- Websites like <http://json2ts.com/> can do that easily:





# Lab #2: Interfaces

- Create a file called **interface.ts**
- Use the same function as in our “Hello world” example, but pass a **Person** as a parameter and make it all work

```
interface Person {  
    firstName: string,  
    lastName: string  
}
```



# Classes

- Classes can be instantiated and as such can have a constructor:

```
class Employee {  
    private name: string;  
    constructor(theName: string) {  
        this.name = theName;  
    }  
}
```



## Classes - Constructor syntax shortcut

- Constructor parameters marked as **public**, **private** or **protected** are automatically turned into class properties:

```
class Employee {  
    constructor(private theName: string) {}  
}
```

- The above code is equivalent to the class declaration in our previous example - only much shorter



# Classes: Visibility of properties

- Default visibility is **public** so using that keyword is redundant:

```
class Animal {  
    public name: string;  
    public constructor(theName: string) { this.name = theName; }  
    public move(distanceInMeters: number) {  
        console.log(`${this.name} moved ${distanceInMeters}m.`);  
    }  
}
```

Private restricts the visibility to that class

Protected restricts the visibility to that class and its subclasses



# Classes: Inheritance

```
class Animal {  
  name: string;  
  constructor(theName: string) { this.name = theName; }  
  move(distanceInMeters: number = 0) {  
    console.log(`${this.name} moved ${distanceInMeters}m.`);  
  }  
}  
  
class Snake extends Animal {  
  constructor(name: string) { super(name); }  
  move(distanceInMeters = 5) {  
    console.log("Slithering...");  
    super.move(distanceInMeters);  
  }  
}
```



# Classes and interfaces

- A class can implement one or more interfaces

```
interface ClockInterface {  
    currentTime: Date;  
    setTime(d: Date);  
}  
  
class Clock implements ClockInterface {  
    currentTime: Date;  
    setTime(d: Date) {  
        this.currentTime = d;  
    }  
    constructor(h: number, m: number) { }  
}
```



# Classes and interfaces

```
interface ClockInterface {  
    currentTime: Date;  
    setTime(d: Date);  
}  
  
class Clock implements ClockInterface {  
    currentTime: Date;  
    setTime(d: Date) {  
        this.currentTime = d;  
    }  
    constructor(h: number, m: number) { }  
}
```

A class can implement one or more interfaces

```
class Control {  
    private state: any;  
}  
  
interface SelectableControl extends Control {  
    select(): void;  
}
```

An interface can extend a class as well:





# Lab #3: Classes



# Classes: Static properties

- A static property is a member of the class itself and is the same for all instances of that class:

```
class Grid {  
  static origin = {x: 0, y: 0};  
  calculateDistanceFromOrigin(point: {x: number; y: number;}) {  
    let xDist = (point.x - Grid.origin.x);  
    let yDist = (point.y - Grid.origin.y);  
    return Math.sqrt(xDist * xDist + yDist * yDist) / this.scale;  
  }  
  constructor (public scale: number) { }  
}
```

As a result, we don't use "this" but the class name to access it



# Classes: Abstract classes

- Abstract classes provide implementation details:

```
abstract class Animal {  
    abstract makeSound(): void;  
    move(): void {  
        console.log("roaming the earth...");  
    }  
}
```

They are meant to be extended.



# Classes: Getters and setters

```
class Employee {  
    private _fullName: string;  
  
    get fullName(): string {  
        return this._fullName;  
    }  
  
    set fullName(newName: string) {  
        if (passcode && passcode == "secret passcode") {  
            this._fullName = newName;  
        }  
        else {  
            console.log("Error: Unauthorized update of employee!");  
        }  
    }  
}  
  
let employee = new Employee();  
employee.fullName = "Bob Smith";  
if (employee.fullName) {  
    console.log(employee.fullName);  
}
```



Introduction  
to Typescript

**Generics and  
Advanced  
types in  
Typescript**

**Outline  
for today**

01

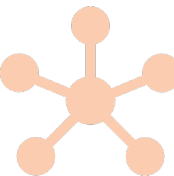
03

02

04

Typescript variables and  
object oriented features

Modules in Typescript  
+ important ES6 features



## Section 3



# Generics and Advanced types in Typescript



30 min

# Generics

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

```
class Greeter<T> {  
    greeting: T;  
    constructor(message: T) {  
        this.greeting = message;  
    }  
    sayHello(): T {  
        return this.greeting;  
    }  
}
```

```
let greet = new Greeter<string>("Hello " + user.getFullName());
```





# Advanced Types

- Union Types - Here a string OR a number:

```
/**
 * Takes a string and adds "padding" to the left.
 * If 'padding' is a string, then 'padding' is appended to the left side.
 * If 'padding' is a number, then that number of spaces is added to the left side.
 */
function padLeft(value: string, padding: string | number) {
    // ...
}

let indentedString = padLeft("Hello world", true); // errors during compilation
```



# Advanced Types: Typeof

- Returns the type of an object as a string:

```
if (typeof padding === "number") {  
    return Array(padding + 1).join(" ") + value;  
}  
if (typeof padding === "string") {  
    return padding + value;  
}
```



# Advanced Types: Instance of

- This is actually plain old Javascript:

```
// Type is 'SpaceRepeatingPadder | StringPadder'  
let padder: Padder = getRandomPadder();  
  
if (padder instanceof SpaceRepeatingPadder) {  
    padder; // type narrowed to 'SpaceRepeatingPadder'  
}  
  
if (padder instanceof StringPadder) {  
    padder; // type narrowed to 'StringPadder'  
}
```



# Advanced Types: Guards and assertions

<https://www.typescriptlang.org/docs/handbook/advanced-types.html>

```
function isFish(pet: Fish | Bird): pet is Fish {  
    return (<Fish>pet).swim !== undefined;  
}
```

```
// Both calls to 'swim' and 'fly' are now okay.
```

```
if (isFish(pet)) {  
    pet.swim();  
}  
else {  
    pet.fly();  
}
```



# Lab #4: Generics

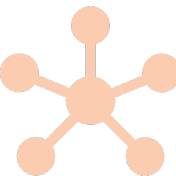
Introduction  
to Typescript

Generics and  
Advanced  
types in  
Typescript

## Outline for today

Typescript variables and  
object oriented features

**Modules in Typescript  
+ important ES6 features**



## Section 4



# Modules in Typescript + important ES6 features



30 min

# Modules

- Any file containing a top-level import or export is considered a module
- Exporting a definition makes it available outside of the current file

```
export const numberRegex = /^[0-9]+$/;

export class ZipCodeValidator implements StringValidator {
  isAcceptable(s: string) {
    return s.length === 5 && numberRegex.test(s);
  }
}
```





# Modules - Export

- You can also define a class first, then export it - and even export it under a different name:

```
class ZipCodeValidator implements StringValidator {  
    isAcceptable(s: string) {  
        return s.length === 5 && numberRegexp.test(s);  
    }  
}  
export { ZipCodeValidator };  
export { ZipCodeValidator as mainValidator };
```



# Modules: Import

- Exported modules can then be imported in other modules:

```
import { ZipCodeValidator } from "../ZipCodeValidator";  
  
let myValidator = new ZipCodeValidator();
```

- Imports can rename a module for local use:

```
import { ZipCodeValidator as ZCV } from "../ZipCodeValidator";  
let myValidator = new ZCV();
```



# ! Modules

## Step 1

TypeScript compiles down to JavaScript so actual dependency management happens at runtime in JavaScript

## Step 2

Well-known module loaders used in JavaScript are the CommonJS module loader for Node.js and require.js or System JS for Web applications. Webpack is also increasingly popular.

## Step 3

As a result, the TypeScript compiler can be given a specific module-loader to specify how to generate JavaScript code.



# ES6 magic in TS: Strings

- Backticks ` allow for multi-line strings
- Template expressions replace JS concatenation

```
var a = 5;
var b = 10;

// ES5 javascript
console.log('Fifteen is ' + (a + b) + ' and\nnot ' + (2 * a + b) + '.');

// ES6 javascript and TypeScript
console.log(`Fifteen is ${a + b} and
not ${2 * a + b}.`);

// Both display:
// "Fifteen is 15 and
// not 20."
```



# ES6 magic in TS: Arrow functions

- Short syntax for functions
- Arrow function used as a one-liner automatically return the value on the right-hand side of the arrow

*// This function definition*

```
function test(a,b) {  
    return a + b;  
}
```

*// Is perfectly equivalent to the one below*

```
let test2 = (a,b) => a+b;
```

*// And the one one below*

```
let test3 = (a,b) => {  
    return a+b;  
};
```

```
test(1,2); // 3  
test2(1,2); // 3  
test3(1,2); // 3
```



# Object Spread

- Easy way to copy or merge objects using “...”

```
// Creates a copy of the original object  
let copy = { ...original };  
  
// Let's merge object a into object b  
let a = {foo: "value", bar: 2};  
let b = {name: "John", ...a};  
// Now b is {name: "John", foo: "value", bar: 2}  
  
// Even better: Several objects can be merged into one  
let merged = { ...foo, ...bar, ...baz };
```



# Object Rest

- The same “...” can be used to get the “rest” of things:

```
// Used to pass a variable number of parameters  
function test(a, ...args) {  
  // args is then an array of params so we can  
  // read its length  
  if (args.length) {  
    // ....  
  }  
}
```

```
// With objects  
let obj = { x: 1, y: 1, z: 1 };  
let { z, ...obj1 } = obj;  
// obj1 is now {x: 1, y: 1}
```





# Destructuring Arrays and Objects

- Arrays can be destructured easily with the following syntax:

```
var myArray = [1, 2, 3, 4];
```

```
var [first, second] = myArray;
```

```
// first = 1, second = 2
```

```
var [first, second, ...rest] = myArray;
```

```
// first = 1, second = 2, rest = [3, 4]
```



# Lab #5: Strings

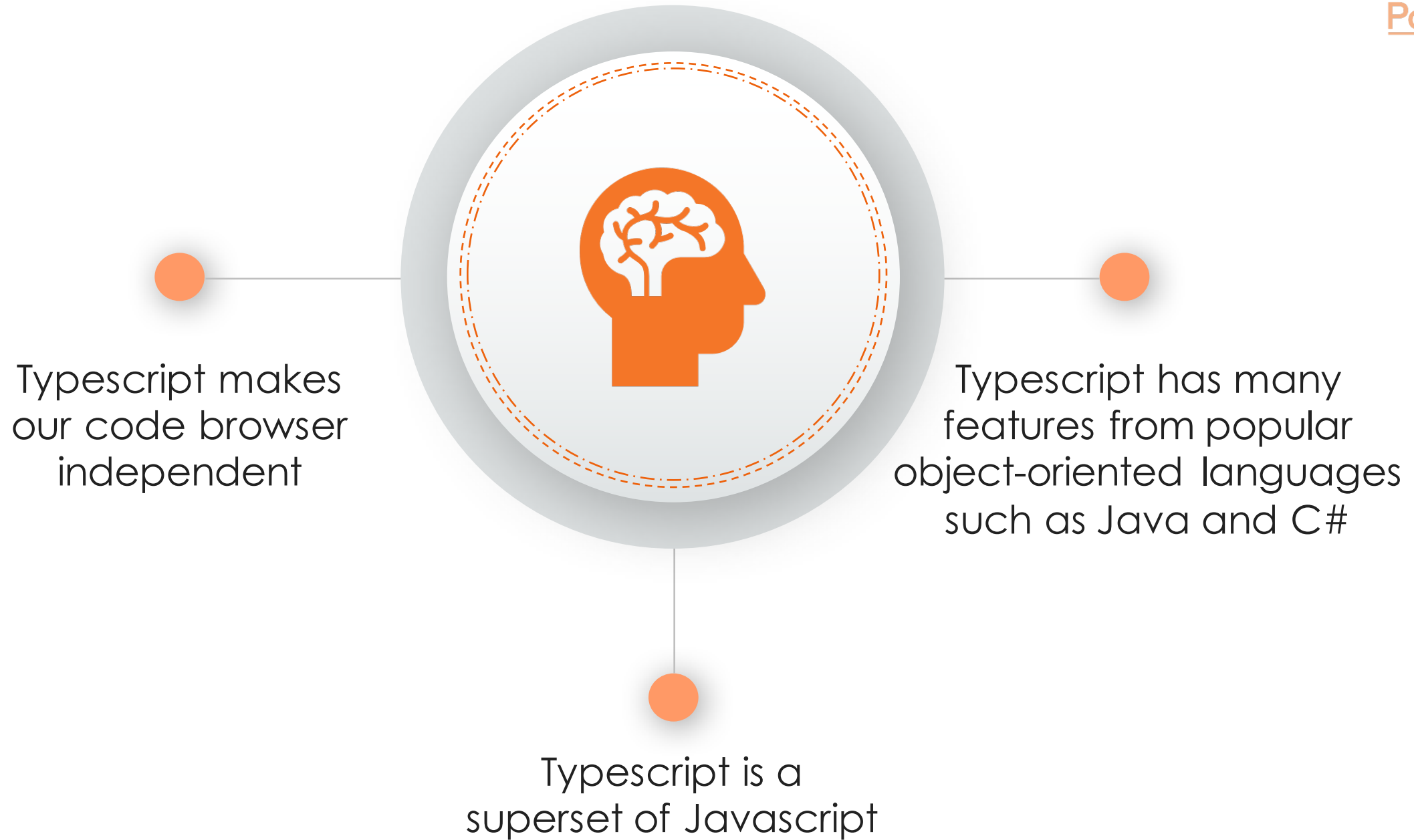


Use backticks and expressions where we used concatenation in the past.

Display our greeting on two lines as follows:  
**Hello**  
**firstName LastName**



# What we just learnt



# End of Day 1



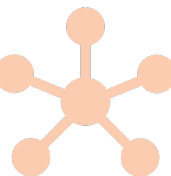
# Day 2

## Introduction to Angular



Using the  
component router  
to emulate  
multiple pages

How to connect Angular  
to the backend?





## Section 5



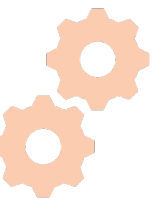
# Introduction to Angular



50 min

# What is Angular?

- ✓ Open-source web application framework, maintained by Google and the community - <http://angular.io>
- ✓ Decouple DOM manipulation from application logic
- ✓ Decouple the client side of an application from the server side
- ✓ Provide structure for the journey of building an application (MVC pattern)
- ✓ Started in 2009 - Version 1.0 in 2012 - 1.5.0 in 2016



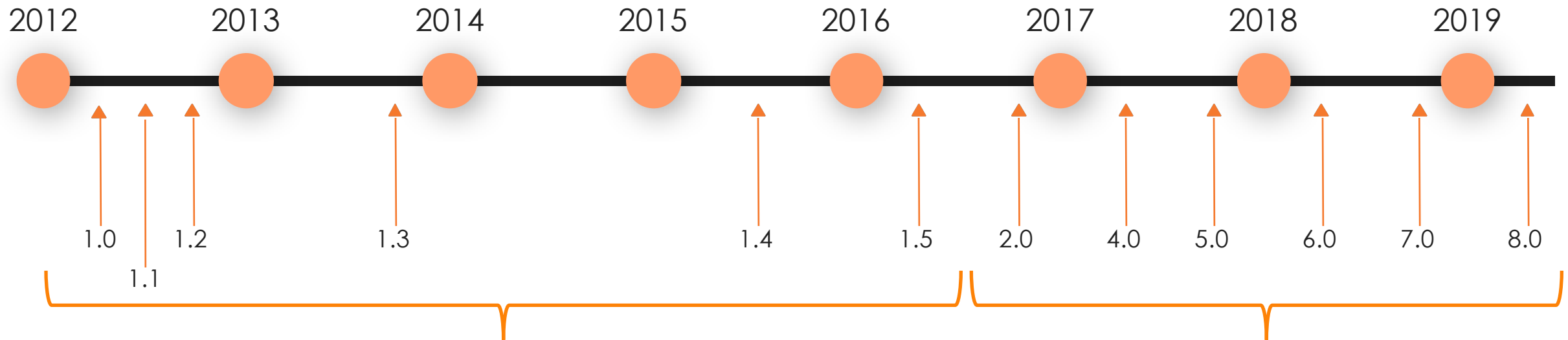
# A bit of Angular history



Angular JS era (1.x)

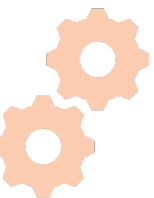


Angular era (2+)



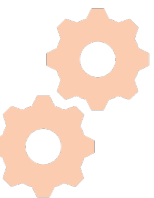
Releases happen from time to time

Clear, predictable schedule

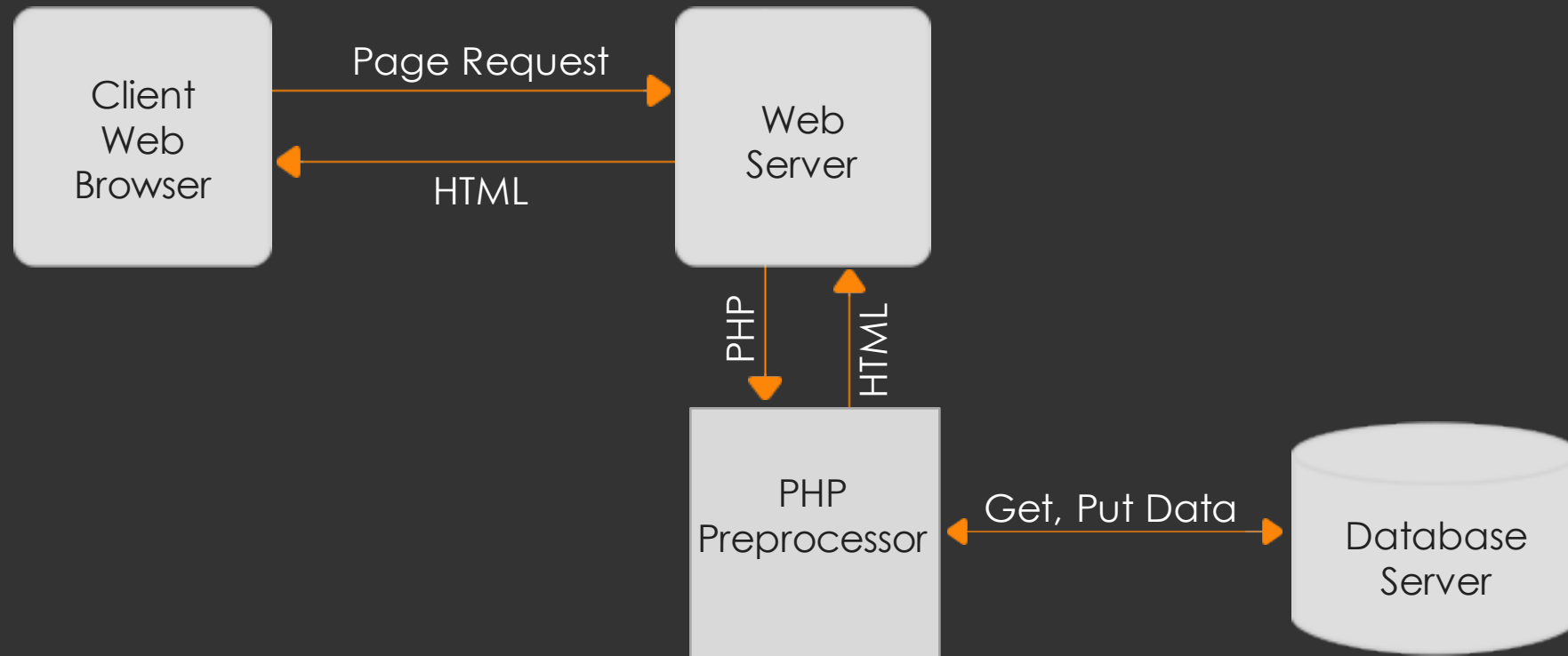


# Semantic versioning and schedule

- ✓ Major versions are released every 6 months
- ✓ All major versions have a 1 year LTS after their initial 6 months lifetime
- ✓ Minor versions happen once month
- ✓ Patch versions happen every week if needed

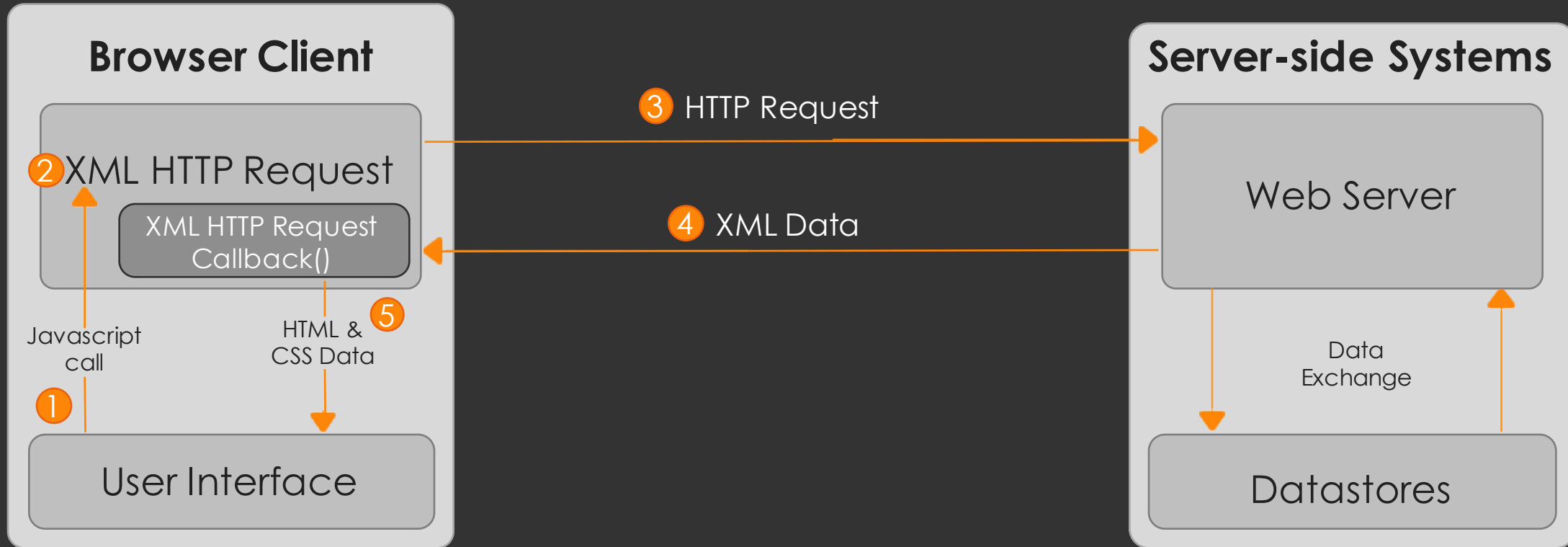


# How is it different from PHP, ASP, JSP?



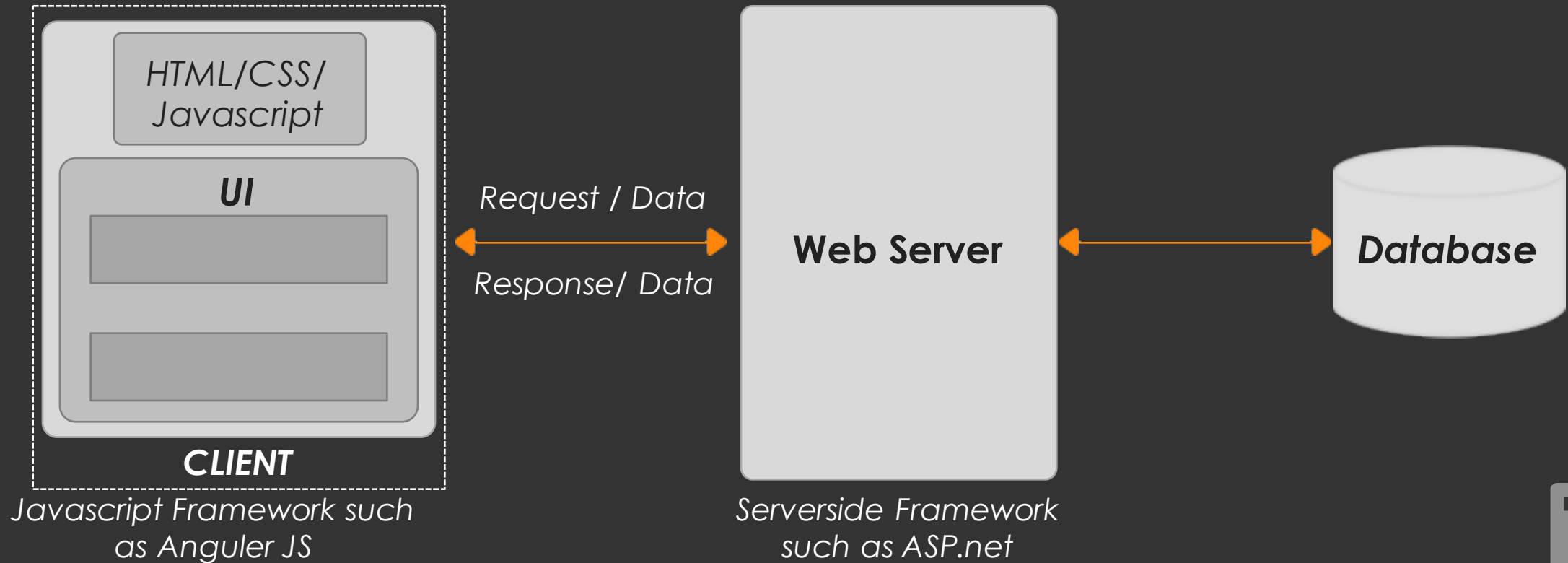
- In the past, all of the front-end code (HTML, JS, CSS) was generated from the back-end.
- User interactions with the webpage often required a full-page refresh.

# How is it different from PHP, ASP, JSP?



- Then came into play AJAX and jQuery
- The main idea was to load content asynchronously in the background to refresh portions of a webpage

# How is it different from PHP, ASP, JSP?



- With Angular, the front-end code is now independent from the back-end
- The web server becomes a web-service that outputs JSON data, not dynamic HTML or CSS

## Angular CLI - <https://cli.angular.io/>

- It's a code generator (with commands such as: `ng generate`)
- It's a set of tools for development (local server, auto-refresh)
- It's a set of build tools (compile, lint, minify...)
- Also very helpful for testing





A module is a package of Typescript objects

## Module

Components

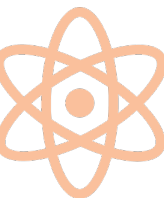
HTML template

Pipes

Your own  
classes

Services

Directives



# Angular Component

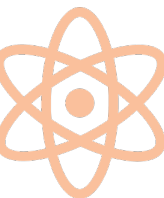
- A component is a reusable piece of HTML (think widget) bundled with its TypeScript controller
- Creating a component is like creating your own HTML element
- 80%+ of the Angular code you're going to write will be components
- An Angular app is a tree of components

```
<my-app>  
  <header title="A  
title"></header>  
  <left-menu  
[items]="items"></left-menu>  
  <data-view  
[hidden]=false></data-view>  
</my-app>
```



# Angular Services, Pipes and Directives

- **Services** are used to handle the business logic of the app (loading data, storing data, etc.)
- **Pipes** are used to format data (dates, internationalization, currencies, etc.)
- **Directives** manipulate the DOM by creating elements, hiding them, etc.
- **Components** are specific directives that have a HTML template



## Example of Angular component

The component class interacts with the view through an API of properties and methods.

```
@Component({
  selector: 'app-hello',
  template: `
    <div>
      <h2>Hello {{name}}</h2>
    </div>
  `
})
export class HelloComponent {

  name : string;

  constructor() {
    this.name = 'Angular';
  }
}
```



# Passing data to a component

Since a component is an HTML element, we can pass data to it using HTML properties:

```
<app-popup-window title="Test pop-up">
```

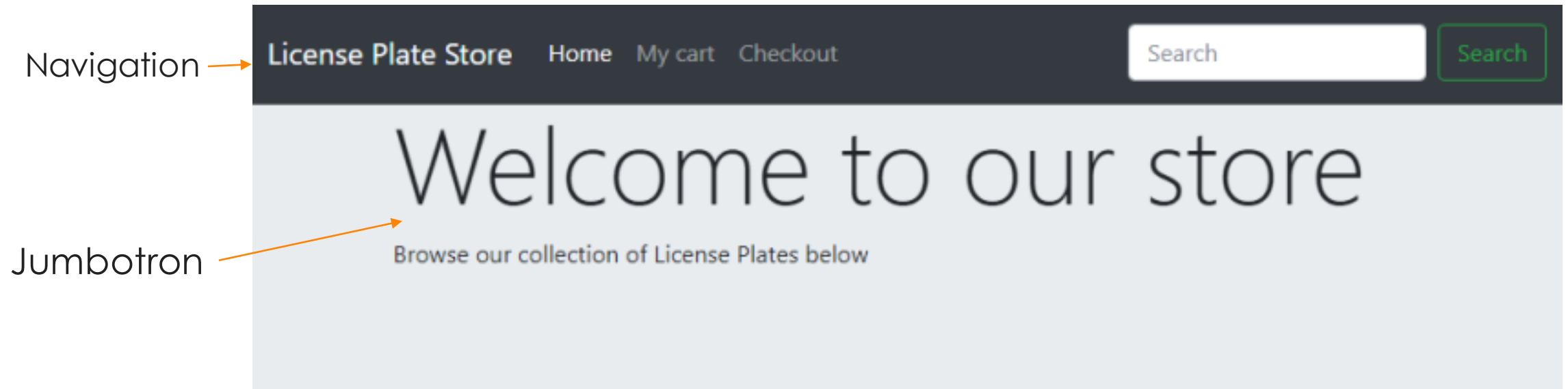
**@Input()** is a decorator that, when applied to a component property, makes the above possible - we can now set the value of title from our HTML code:

```
@Input()  
  
title:  
string;
```



# Lab #6: Creating components

Our goal for this lab is to build two components, navigation, and jumbotron to make our user interface look exactly like this:



- Update **app.component.ts** to make it use **app.component.html** as a template
- Use Angular CLI to generate a **navigation** component
- Change the template for that new component. Use the template provided in: **lab-templates/navigation**
- Use our new **app-navigation** component selector in **app.component.html** where instructed in that file.
- Congrats, you just created your first component!





- Use Angular CLI to generate a **jumbotron** component
- Change the template for that new component. Use the template provided in: **lab-templates/jumbotron**.
- Use our new **app-jumbotron** component selector in **app.component.html** where instructed in that file.
- **jumbotron** will be used in different contexts in our application (cart page, checkout page) so we need to add two properties **title** and **description** that can be set with **@Input** to make that component configurable.



# Angular Data Bindings - [] and ()

Component class properties can be bound to HTML attributes:

```
<p>  
  <input type="text" [value]="name" />  
  <button (click)="alertName()">Click me</button>  
</p>
```

The above sets the value of the **value** HTML attribute of the **input** element to **this.name** in our component class

It also binds the **onClick** event to the **alertName()** method



# Angular Models - [(ngModel)]

- [(ngModel)] = 2-way data binding
- Changes to the HTML update your TS scope and vice-versa:

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

## [( )] = BANANA IN A BOX

Visualise a banana in a box to remember that the parentheses go inside the brackets.



# Bindings + HTML = Super Powers

Since bindings work on any HTML attribute, you can use them to apply styles conditionally:

```
<p [style.color]="hasError ? 'red' : 'blue'">
```

This also works to show/hide error messages:

```
<div [hidden]="noErrors">  
  The value you entered  
  is not valid  
</div>
```



# Angular \*ngFor

**\*ngFor** is a directive

A loop to iterate through data model items and repeat an element for every single item:

```
<li *ngFor="let person of persons">  
    {{ getDisplayName(person) }}  
</li>
```



# Demo

# Angular \*ngIf

- Directives like Angular JS ng-show, ng-hide are replaced by HTML attribute bindings
- For instance, **[hidden]="myCondition"**
- **\*ngIf** is based on DOM rather than CSS:

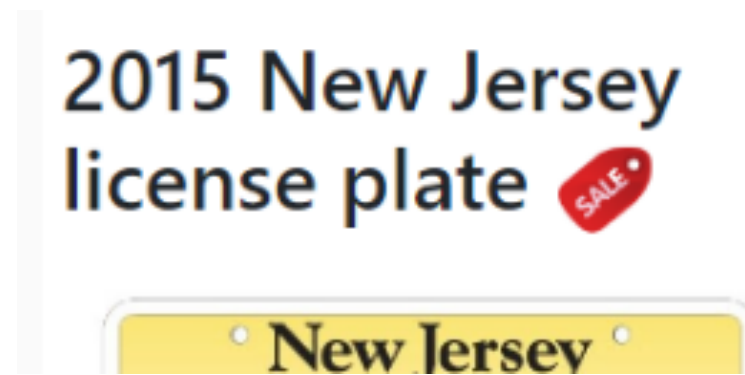
```
<span *ngIf="person.firstName.length > 3">  
  {{ getFullName(person) }}  
</span>
```



# Lab #7: ngFor and ngIf



- Use the \*ngFor directive to render all of the license plates from AppComponent, stored in the licensePlates property.
- Some license plates are on sale (property onSale = true)
- Using the file src/assets/sale.png, update your license plate component so that plates on sale look as follows:



# What we just learnt

Pipes allow us to format data

Angular is a web development framework that allows us to build reusable HTML components

Directives are components without any HTML template

Services are a way to implement the business logic and interact with servers

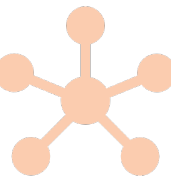


Introduction  
to Angular



Using the  
component router  
to emulate  
multiple pages

**How to connect Angular  
to the backend?**



## Section 6



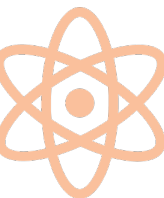
# How to connect Angular to the backend?



50 min

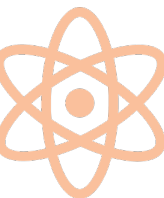
# Angular Services

- ✓ Services are meant to handle application logic (fetch data, update data, etc.)
- ✓ They are classes that can be injected
- ✓ Add **@Injectable()** decorator to make a class injectable
- ✓ Services are singletons: There is only one instance per injector



# Dependency injection

- ✓ Dependency injection is a process where a class delegates the responsibility of providing its dependencies to external code (the injector)
- ✓ The class is not allowed to call the injector code. It is the injecting code that constructs the services and calls the constructor to inject them.



# Dependency injection with Angular

- Angular relies on constructors to achieve dependency injection
- Any class marked as **@Injectable** can be injected into an Angular object as follows:

```
export class ListPostsComponent {  
  constructor(postService : PostsService) {
```





# Observables from RxJs - reactivex.io

- **Observables** implement the observer design pattern.
- The official website is <http://reactivex.io>. It shows that ReactiveX has implementations for multiple languages.
- In our case, we're using RxJS (**R**eactive**X** for **J**ava**S**cript)



# Observables from RxJs - rxivex.io

- In the Angular framework, Observables are used by multiple services to get notified:
  - Whenever the URL changes in the browser
  - When a form input gets updated
  - When a HTTP request completes successfully
- Observables are perfect to listen to a stream of events, and as a result an Observable's lifecycle is usually represented with a timeline such as this one:



# Example of Observable

Let's take a look at **list-posts.component.ts** in **projects/demos/src/app/observable-example/list-posts**

This is how we subscribe to an observable:

```
export class ListPostsComponent {  
  
    posts: Post[] = [];  
  
    constructor(private  
postService: PostsService) {  
  
        this.postService.getPosts()  
            .subscribe(post  
=> this.posts.push(post));  
    }  
}
```



# Observables can be seen as a newspaper subscription



- When we subscribe to a newspaper, we expect to receive several issues over time.
- Those issues are delivered at a specific location: The address we give when we subscribe is equivalent to the callback function we pass to the **subscribe** method.
- That subscription will complete at some point (for instance after a given number of months)



# Angular HttpClient

- Angular service for http requests (AJAX)
- Works seamlessly with JSON data and REST

```
this.http.get("http://localhost:8080/persons.json")  
  .subscribe(data => this.persons = data)
```



# Angular HttpClient

- **HttpClient** is a service that can be injected in the constructor of a class:

```
import {HttpClient} from "@angular/common/http";

@Injectable()
export class PersonService {

    constructor(private http : HttpClient) { }

    getList() : Observable<Person[]> {
        return this.http.get("http://localhost:8080/persons.json")
    }
}
```



Lab #8:



# Connecting Angular with the back-end



- Your mission: Create a LicensePlateService to load a list of License plates from <http://localhost:8000/data>
- Start the back-end server by opening a new terminal and running: `npm run server`
- Make sure that all license plates render properly in the user interface:

2008 Georgia license plate



Ad occaecat ex nisi reprehenderit dolore esse. Excepteur laborum fugiat sint tempor et in magna labore quis exercitation consequat nulla tempor occaecat. Sit cillum deserunt eiusmod proident labore mollit. Cupidatat do ullamco ipsum id nisi mollit pariatur nulla dolor sunt et nostrud qui.

\$8

Add to cart +

2015 New Jersey license plate



A beautiful license plate from the Garden State. Year is 2015.

\$11

Add to cart +

2013 California My Tahoe license plate



Sunt inure nisi excepteur in ea consequat ad aliqua. Lorem duis in duis nisi sit. Cillum eiusmod ipsum mollit veniam consectetur ex eiusmod nisi laborum amet anim mollit non nulla. Lorem ea est exercitation nostrud consectetur officia laborum fugiat sint. Nostrud consequat magna officia minim et aute nostrud.

\$9

Add to cart +

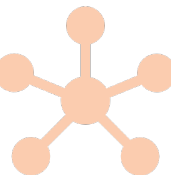


## DEMO (if time allows) - Cart features

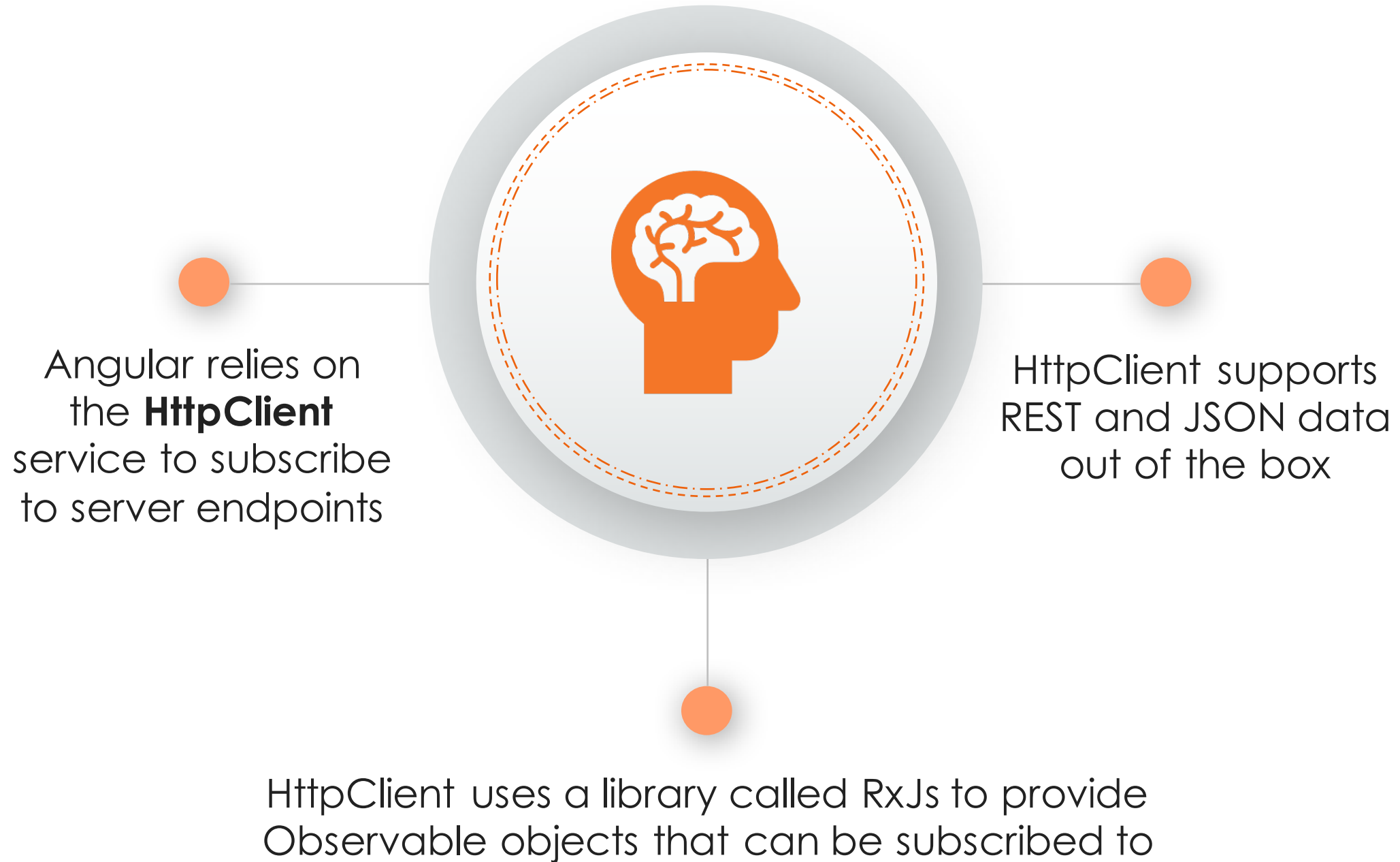


Let's connect the "Add to cart" feature that will be used to add a product to the cart

The let's implement the "Cart page" so that our user interface will be able to display the current cart contents



# What we just learnt

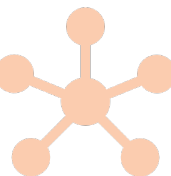


Introduction  
to Angular



**Using the  
component router  
to emulate  
multiple pages**

How to connect Angular  
to the backend?



## Section 7



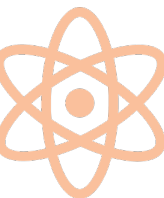
# Using the component router to emulate multiple pages



25 min

# Angular Component Router

- ✓ Angular was designed for Single Page Applications
- ✓ But very often we need more than one page!
- ✓ The Component Router allows us to emulate this by loading components based on URLs
- ✓ For instance, **/cart** would load a Cart component, **/login** for login, etc.



## Example of router implementation

Here the navigation component and the footer are always the same on all pages.

The only dynamic part is the blue section, determined by the **<router-outlet>** directive

Navigation

Dynamic content area where components get loaded based on the URL

**<router-outlet> </router-outlet>**

Footer





# Angular Component Router

Requires a routing table which maps URLs to components  
**(app.routing.ts):**

```
const appRoutes: Routes = [  
  {  
    path: 'if', component: NgifComponent  
  },  
  {  
    path: '', component: HelloComponent  
  },  
  {  
    path: 'http', component: HttpComponent  
  }  
];  
export const routing = RouterModule.forRoot(appRoutes);
```



**<router-outlet>** is the place where routed components are loaded in the HTML:

```
<h1>Component Router</h1>
<ul>
  <li><a routerLink="/" >Hello World example</a></li>
  <li><a routerLink="/if" >ngIf example</a></li>
  <li><a routerLink="/http">HTTP example</a></li>
</ul>

<hr>
<router-outlet></router-outlet>
```



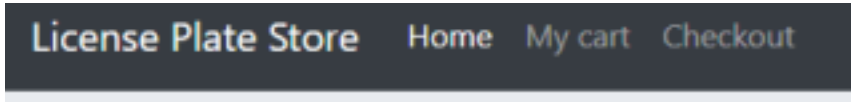
# Lab #9: Component Router

- We're going to have three different views: A store view, a checkout view, and a cart view
- The **StoreViewComponent** is going to be the default one, rendered when the URL is <http://localhost:4200>
- Then **CartViewComponent** is going to be rendered when the URL is <http://localhost:4200/cart>
- Finally **CheckoutViewComponent** is going to be rendered when the URL is <http://localhost:4200/checkout>



Once your routing config is ready, you can test it by typing the different URLs in your browser window.

If everything works, then make the navigation links work properly so we can navigate using those:



License Plate Store Home My cart Checkout

Take a look at the router guide if you need any help:  
<https://angular.io/guide/router>



# Router Service

Angular has a Router service that can be used to navigate programmatically:

```
constructor(private router: Router) {  
  
  this.router.navigateByUrl('login');  
}
```

This service can also be used to change the router config at runtime to dynamically add or remove routes.



# Guards for Authentication

Prevent routes from being accessed using a guard:

```
path: 'if', component: NgifComponent, canActivate: [AuthGuard]
```

A guard implements a canActivate method and simply returns true if the route is allowed, false otherwise:

```
@Injectable()
export class AuthGuard implements CanActivate {

  constructor(){

  }

  canActivate() : Observable<boolean> {
    console.log('AuthGuard#canActivate called');
    return Observable.of(true);
  }
}
```





# Authentication

- Let's use the **AuthGuard** service provided in the router folder to guard the **/checkout** route.
- Then create a new, unguarded route to the **LoginComponent** provided in **router/login**
- Update the **AuthGuard** code to return **false** if the user is not logged in (using the provided **LoginService**)
- Then let's complete the authentication feature with proper redirects just like any production application





 **THANKYOU**  
**WE HOPE YOU ENJOYED THIS COURSE**



• **FINAL Q/A SESSION** •



al@interstate21.com



@AlainChautard



HTTP://BLOG.ANGULARTRAINING.COM