

Angular 2/4

Bhavana Desai

Instruction

- Internet Connection Mandatory
- Visual Studio Tool – 2015 / Visual Studio Code
- Type Carefully – Code
 - Case sensitive
 - Careful about the brackets
- Browser – Google Chrome

Angular2

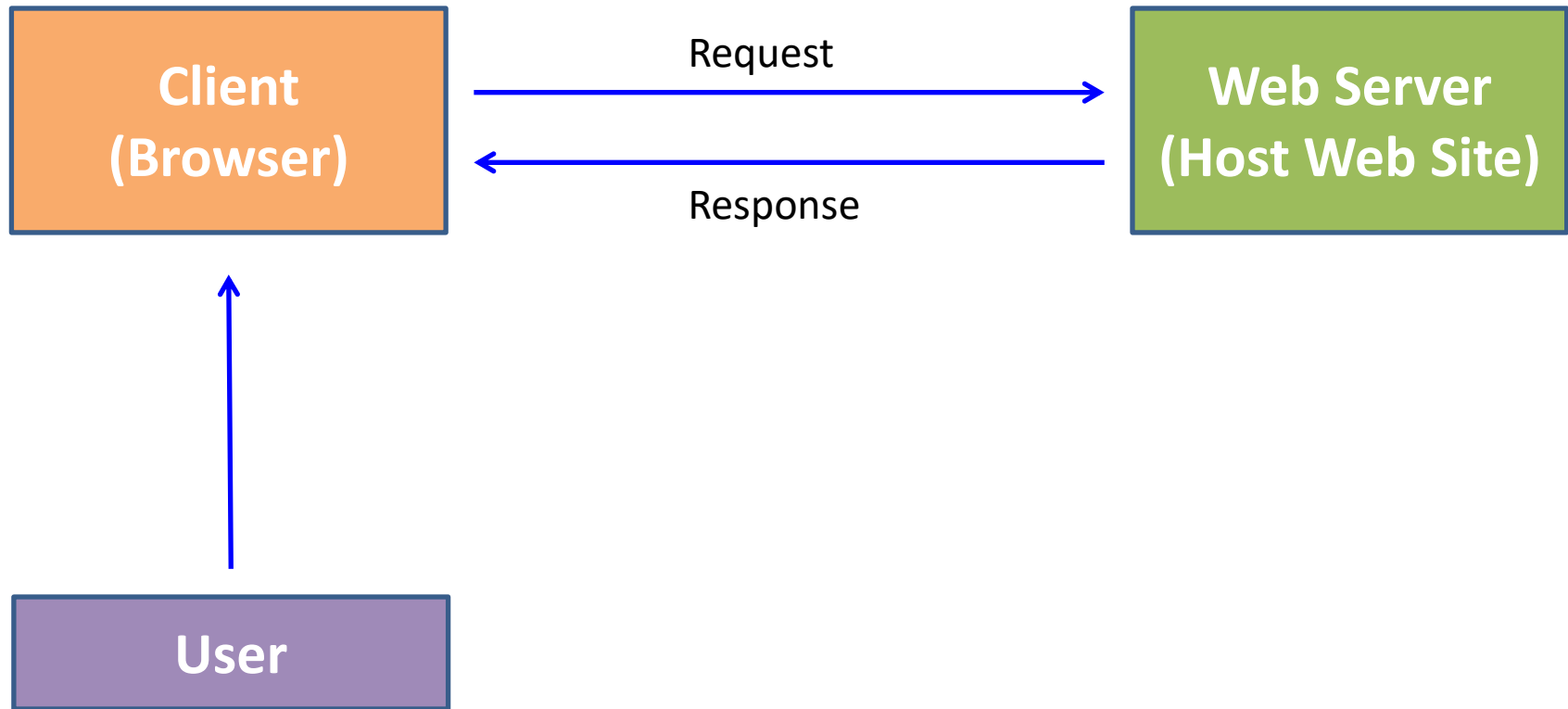
- AngularJS initial release : 20 October 2010
- Angular2 stable release : 14 September 2016
- Angular4 release : 23 March 2017

- Client side Framework
- MVW Framework
- Two way Data Binding / Binding Framework
- Single Page Application (SPA)
- One framework for Mobile & Desktop

Compare to AngularJS 1

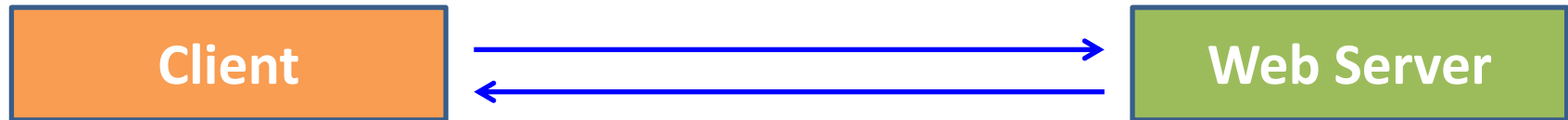
- Angular2/4 use TypeScript for coding
- Angular 2/4 is entirely component based.
- Controllers and \$scope are no longer used
- They have been replaced by components and directives.
- The digest cycle from Angular 1.X has been replaced by another internal mechanism known as “Change Detection”
- No migration

Traditional Web Request

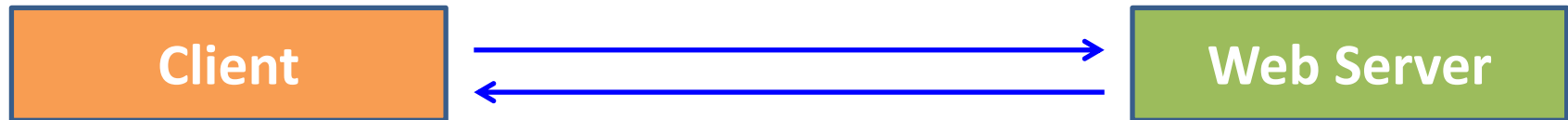


Traditional Request Processing

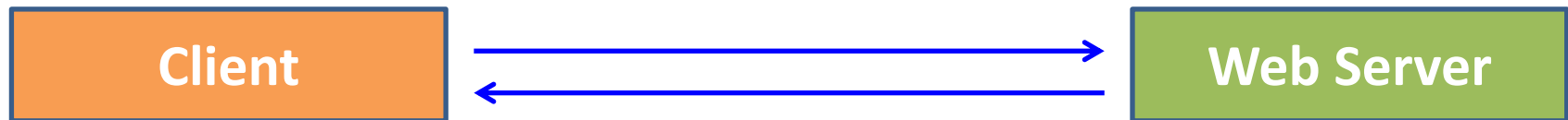
http://Demo.com/



http://Demo.com/About



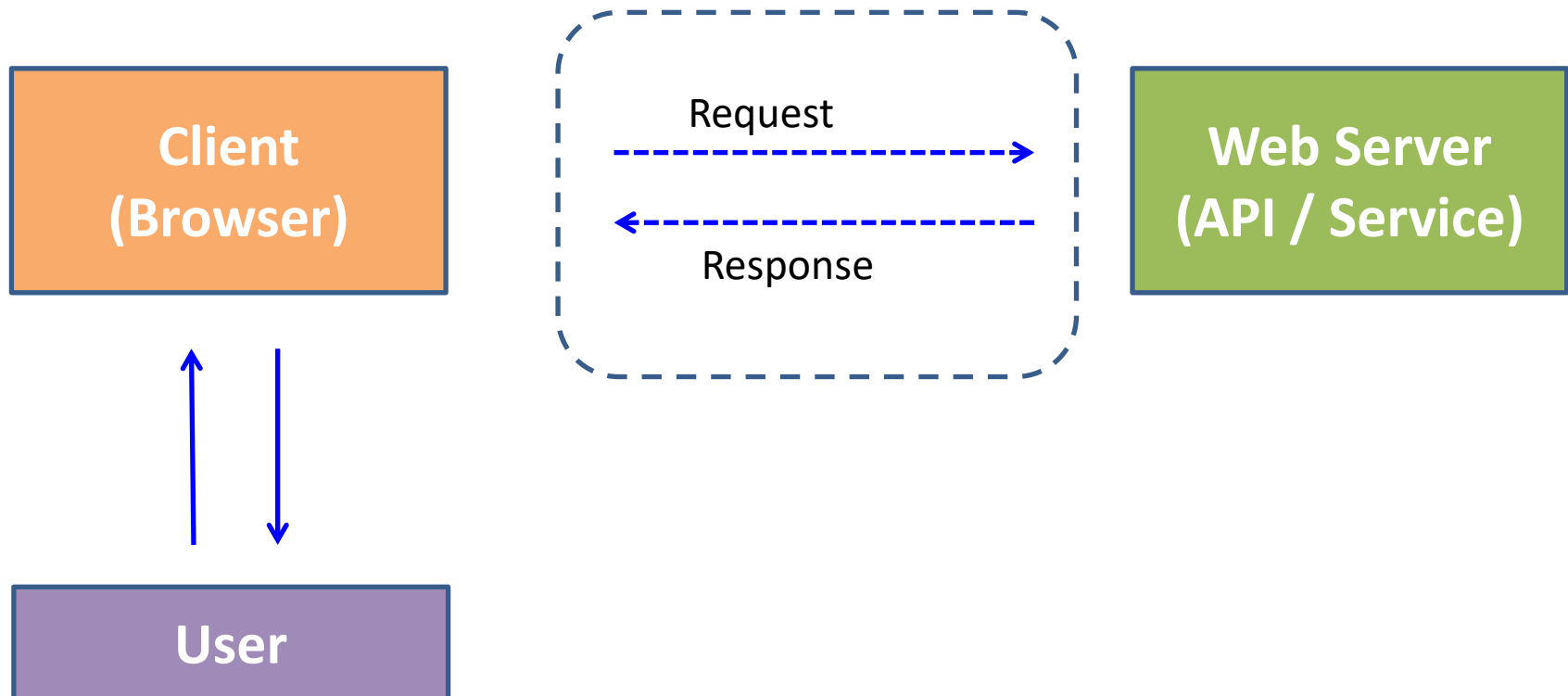
http://Demo.com/Account



Request to server : 3 times

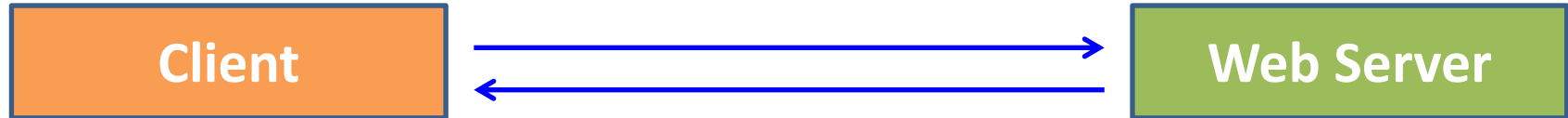
Single Page Application (SPA)

- Request server to get initial page
- Server side communication for database operations
- Most of the server request is asynchronous

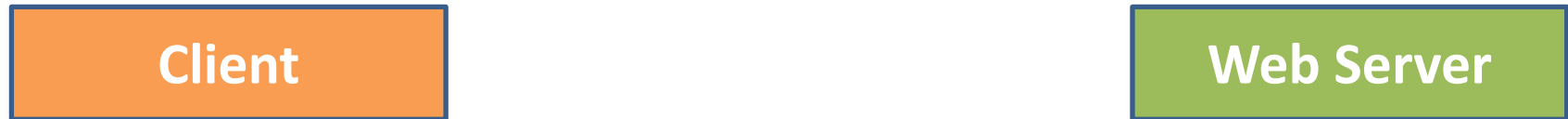


SPA Request Processing

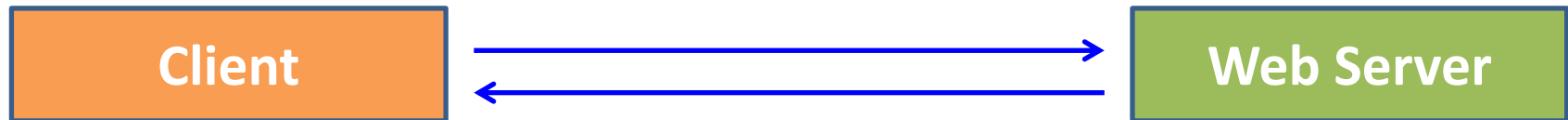
http://Demo.com/



http://Demo.com/About

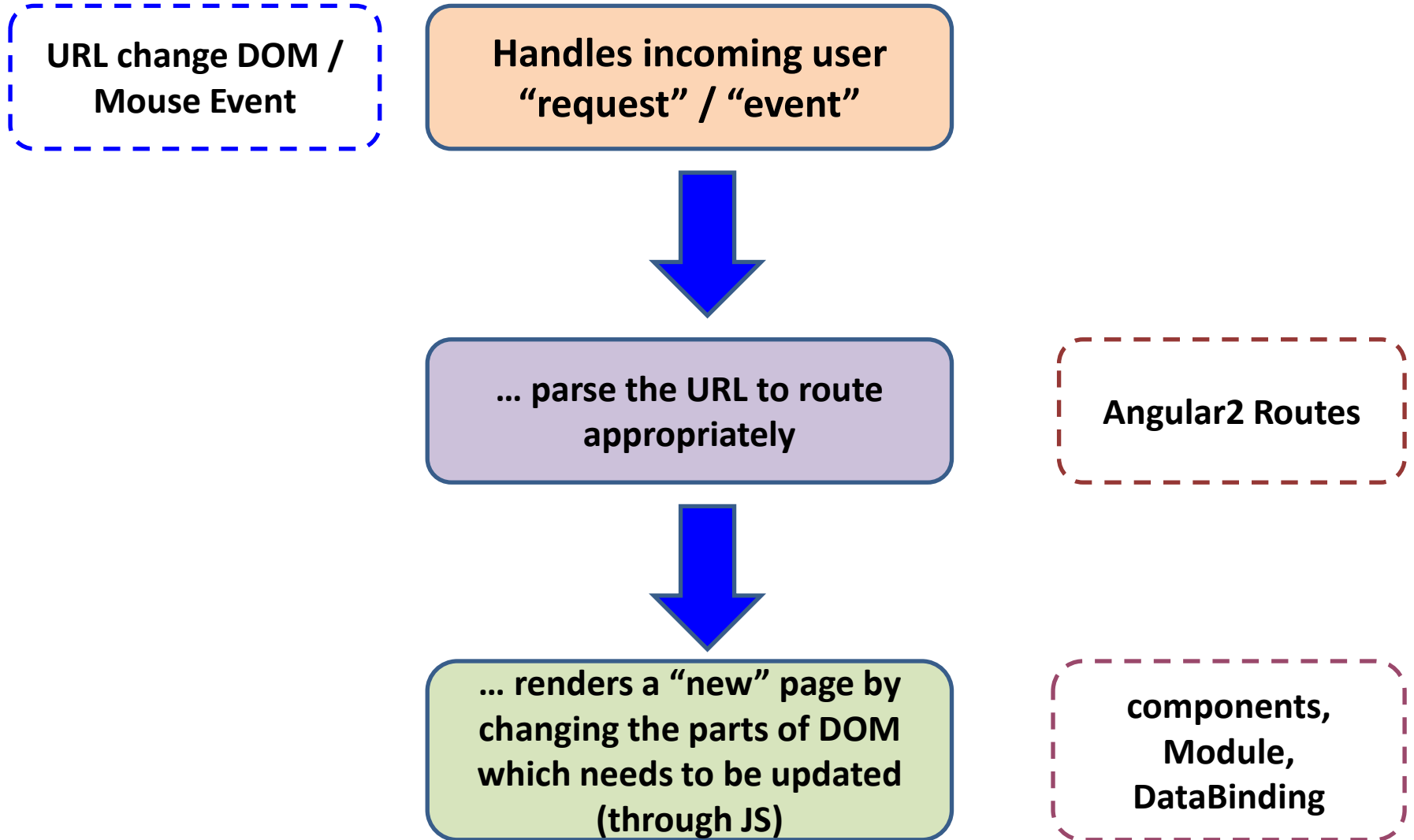


http://Demo.com/Account



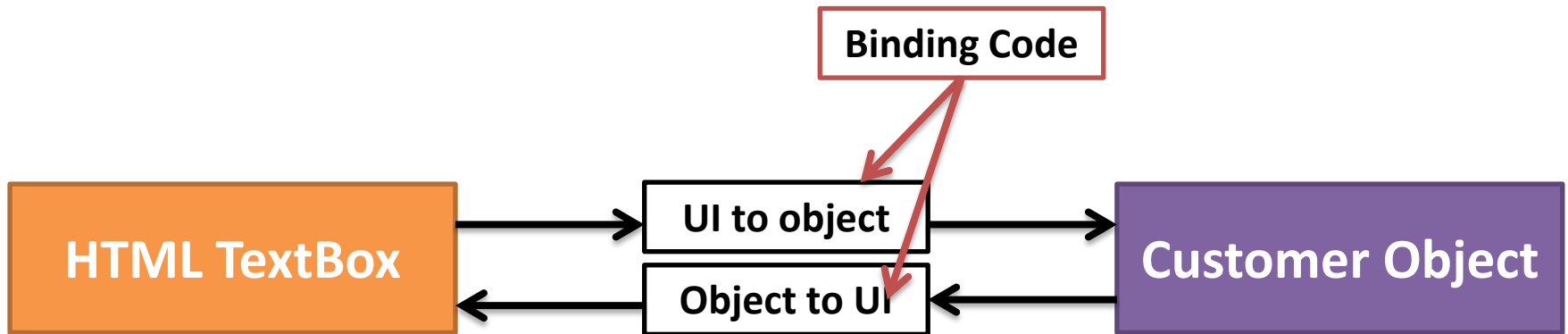
Request to server : 2 times

Angular2



Binding Framework

- Angular binds HTML UI with JavaScript Object



Base Requirement

- Angular2 use lots of other open source JavaScript Framework
- Knowledge of which will be helpful
 - NodeJS for npm
 - TypeScript - Transpiler
 - SystemJS - Loader
 - core-js - Shim and Polyfills
 - Zone.JS - Execution context
 - Reflect-Metadata – Decorator
 - WebPack - Bundler

Visual Studio Code

- VS Code is a lightweight open source editor from Microsoft
- Available for Windows, macOS and Linux
- It comes with built-in support for JavaScript, TypeScript and Node.js
- And has a rich ecosystem of extensions for other languages (such as C++, C#, Python, PHP, Go) and runtimes (such as .NET and Unity).
- Download and install from following web site

<https://code.visualstudio.com/download>

Start New Web Project

- In Windows Explorer create a folder in which we want to save all out Angular project files
- Start VS Code Editor
- From “File” menu select “Open Folder” option and select the folder which we created to store project files

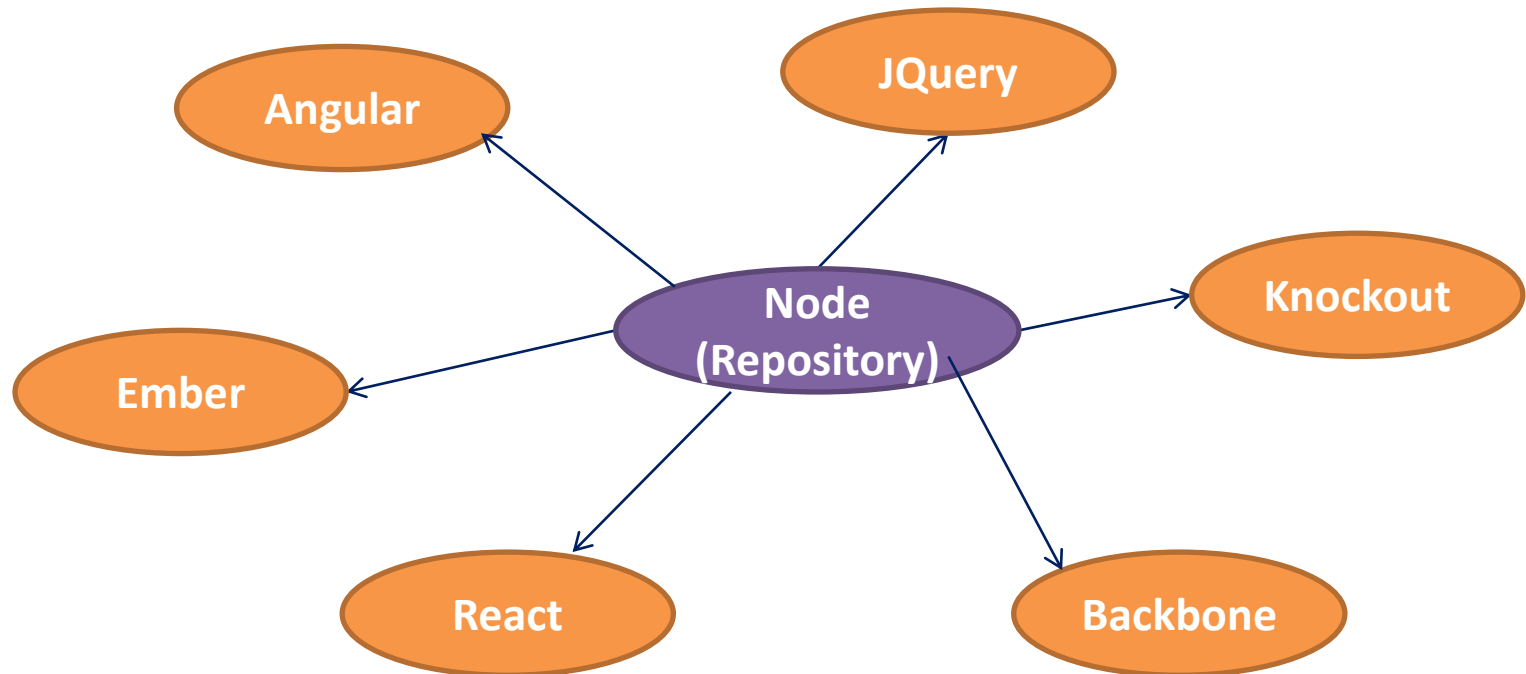
Node.JS

Node.js

- Go to <https://nodejs.org/en/> website and download stable version
- This will gives you node command window
- Two uses
 - Node Package Manager (npm)
 - Runs JavaScript outside browser

Node Package Manager

- Central repository for downloading JavaScript packages like Angular, Knockout, JQuery etc.



Npm Command to install JS file

- JavaScript runtime engine
- Goto node command window and install JQuery

npm install jquery

- By default it will store inside profile folder
- Ex: C:\Users\Admin\node_modules\Jquery

Npm command to Run JavaScript

- Npm command use for running JavaScript outside the browser
- In profile folder create JavaScriptCode.js file

```
for (var x=0;x<5;x++) {  
    console.log(x);  
}
```

- Goto node prompt and type
node JavaScriptcode.js

Package.json

- In a real time project we need multiple js files like JQuery, Angular, TypeScript etc.
- Now passing individual command is tedious task
- So you can create JSON file.
- Name of the file is “package.json” only
- Visual Studio 2015 provides template for npm config file
- Create a folder and save this package.json file inside the folder where you want all JS file to download

Package.json

```
{  
  "version": "1.0.0",  
  "name": "asp.net",  
  "devDependencies": {  
    "angular": "^1.5.8",  
    "jquery": "^3.1.0"  
  }  
}
```

- Now in node prompt go to the folder path and pass the command
npm install
- This will create node_module folder inside your folder and download all necessary JavaScript files

Add other JS file

- Now you want to install knockout JS
- Also want to save the entry in package.json file
- Go to Node command prompt → folder where your package.json file saved
- Type the command

npm install knockout --save

- --save will save this entry in package.json file

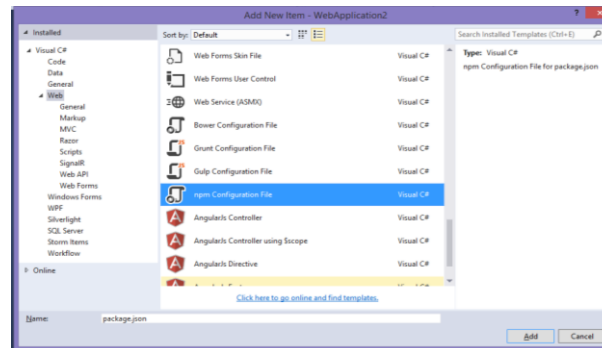
```
{  
  "version": "1.0.0",  
  "name": "asp.net",  
  "devDependencies": {  
    "angular": "^1.5.8",  
    "jquery": "^3.1.0"  
  },  
  "dependencies": {  
    "knockout": "^3.4.2"  
  }  
}
```

npm "dependencies" vs "devDependencies"

- If you are publishing to npm, then it is important that you use the correct flag for the correct modules.
- If it is something that your npm module needs to function, then use the "-save" flag to save the module as a dependency.
- If it is something that your module doesn't need to function but it is needed for testing, then use the "--save-dev" flag
- *# For dependent modules*
- npm install dependent-module --save
- *# For dev-dependent modules*
- npm install development-module --save-dev
- **If you aren't publishing to npm, it technically doesn't matter which flag you use**

Important Points for Node

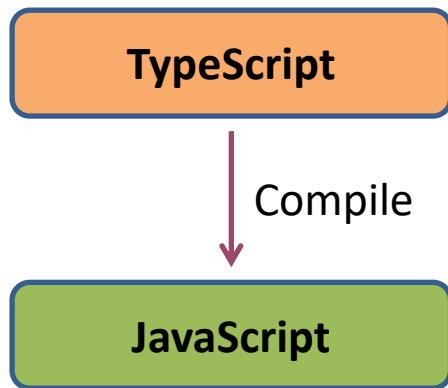
- Node Package Manager (NPM) helps to download JavaScript package
- Node also helps to run JavaScript outside the browser
- package.json config file helps us to get multiple JavaScript files in ONE go
- Visual Studio 2015 provides template for npm config.json



TypeScript

TypeScript

- TypeScript is from Microsoft
- TypeScript is a superset of JavaScript that compiles to plain JavaScript
- Open Source run on any browser, any host, any OS
- TypeScript is a transpiler(Convert one language to another language)



Need of TypeScript

- Writing Object Oriented code in JavaScript is very complex and lengthy
- Syntax of object oriented is different from typical programming language like, C++, Java, C# when we use keywords like class, object, inheritance, polymorphism

Install TypeScript

- Simple way to install TypeScript is through node command prompt
npm install -g typescript
- “g” for global means you can use it from any folder
- Since this is globally installed you can check in
“C:\Users\Admin\AppData\Roaming\npm\node_modules”
- If you type
npm install typescript --save
- Then you will see this entry in your local folder and in
Package.json file

Demo

- Add TypeScript file in your project
- Start Notepad and save file as Customer.ts
- TypeScript file will have .ts as extension
- Compile → go to node command prompt and type
tsc Customer.ts
- After compilation it will generate .js

```
class Customer {  
}
```

```
var Customer = (function () {  
    function Customer() {  
    }  
    return Customer;  
})();
```

TypeScript Code - class

```
class Customer {  
    //PropertyName : datatype = "default value";  
    CustomerId: number;  
    CustomerName: string;  
    greet() {  
        alert("Hello");  
    }  
}
```

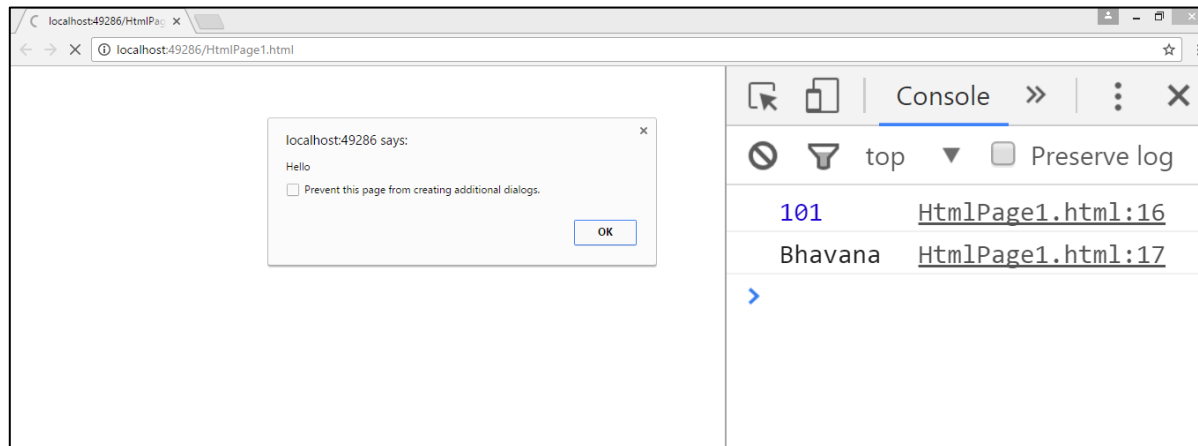
-----JS Code-----

```
var Customer = (function () {  
    function Customer() {  
    }  
    Customer.prototype.greet = function () {  
        alert("Hello");  
    };  
    return Customer;  
})();
```

HTML Page

```
<script src="Customer.js"></script>
<script>
    var custObj = new Customer();
    custObj.CustomerId = 101;
    custObj.CustomerName = "Bhavana";

    console.log(custObj.CustomerId);
    console.log(custObj.CustomerName);
    custObj.greet();
</script>
```



Inheritance

```
class Customer {  
    greet(name: string) {  
        alert("Hello " + name);  
    }  
}
```

```
class CustomerChild extends Customer {  
    greet(name: string) {  
        alert("CustomerChild wish Hi " + name);  
    }  
}
```

```
var custObj = new Customer();  
custObj.greet("bhavana");
```

```
var childObj = new CustomerChild();  
childObj.greet("test");
```

Properties

```
class Customer {  
    private _custName: string = "";  
    public get CustomerName() {  
        return this._custName;  
    }  
  
    public set CustomerName(value: string) {  
        this._custName = value;  
    }  
  
    greet(name: string) {  
        alert("Hello " + name);  
    }  
}
```

```
var custObj = new Customer();  
custObj.CustomerName = "bhavana";
```


If Error for ES5

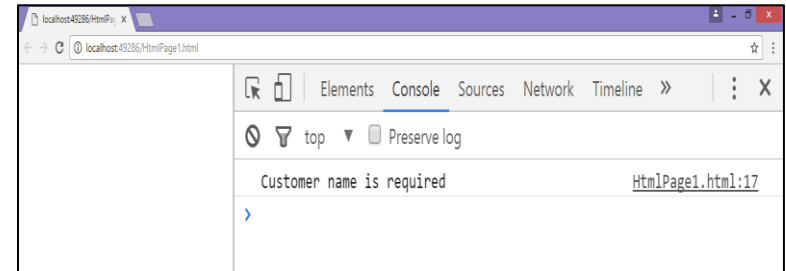
- At the compile time if you get the following error

Customer.ts(3,16): error TS1056: Accessors are only available when targeting ECMAScript 5 and higher.

- Compile with following command
- `tsc --target ES5 yourfilename.ts`

Exception Handling

```
class Customer {  
    private _custName: string = "";  
    public set CustomerName(value: string) {  
        if (value.length == 0) {  
            throw "Customer name is required";  
        }  
        this._custName = value;  
    }  
}  
  
try {  
    var custObj = new Customer();  
    custObj.CustomerName = "";  
    alert(custObj.CustomerName);  
} catch (ex) {  
    console.log(ex);  
}
```

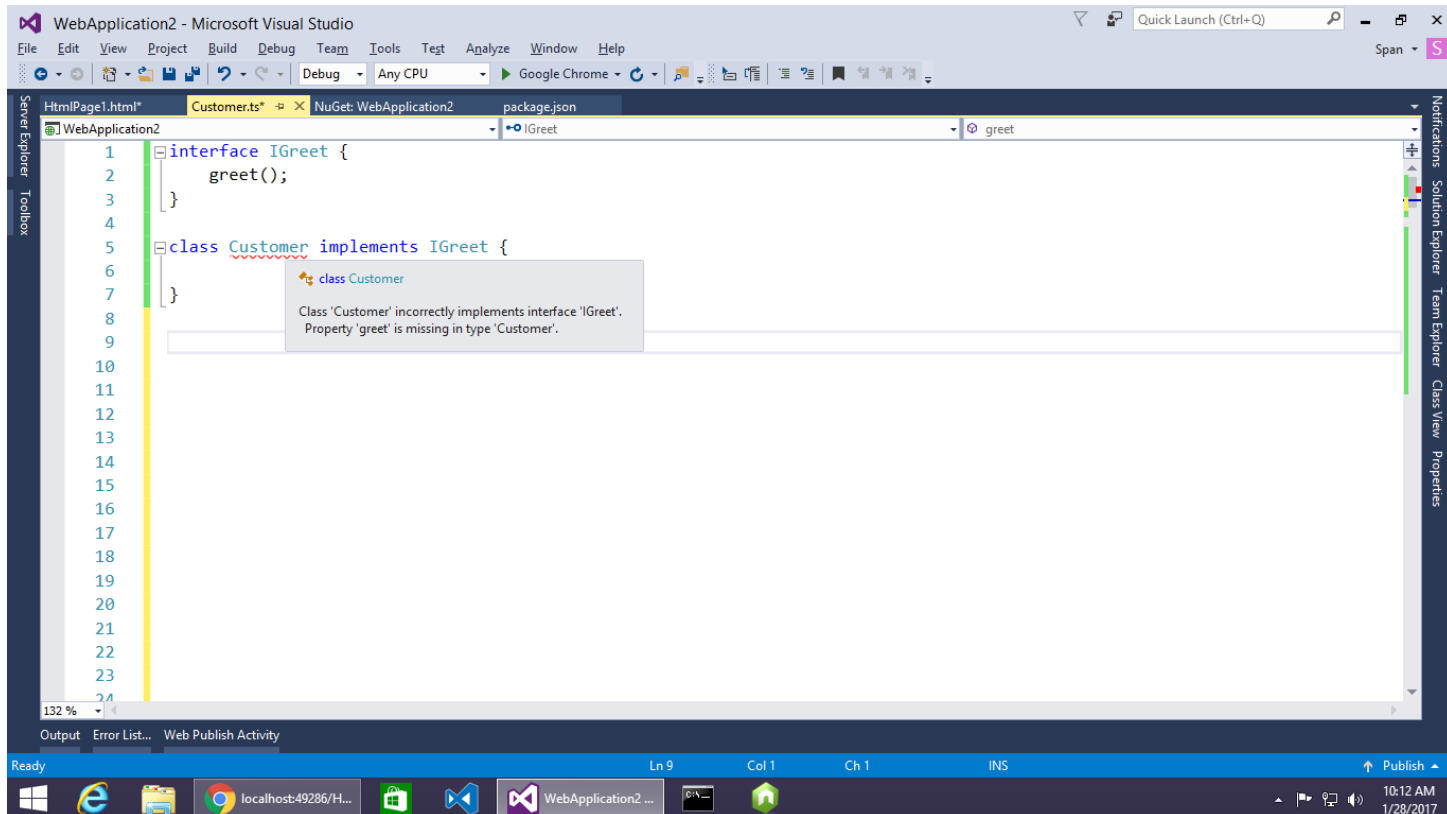


Interface

- TypeScript is a transpiler
- It gives you the feeling of Object Oriented programming
- JavaScript is not Object Oriented
- In JavaScript we do not have interface
- If we create interface in TypeScript, it works in TypeScript, but no equivalent code gets generated in JavaScript
- What you see in TypeScript is not exactly what you get in JavaScript

Interface

- In TypeScript interface will work, but no code generated in JavaScript



Private & Protected Member (Bad Code)

- In JavaScript no equivalent of protected
- So protected members of TypeScript become public in JavaScript
- Even private members are also accessible from outside

```
class Customer {  
    private _custName: string = "";  
}
```

```
<script>  
    var custObj = new Customer();  
    custObj._custName = "test";  
    console.log(custObj._custName);  
</script>
```

Module

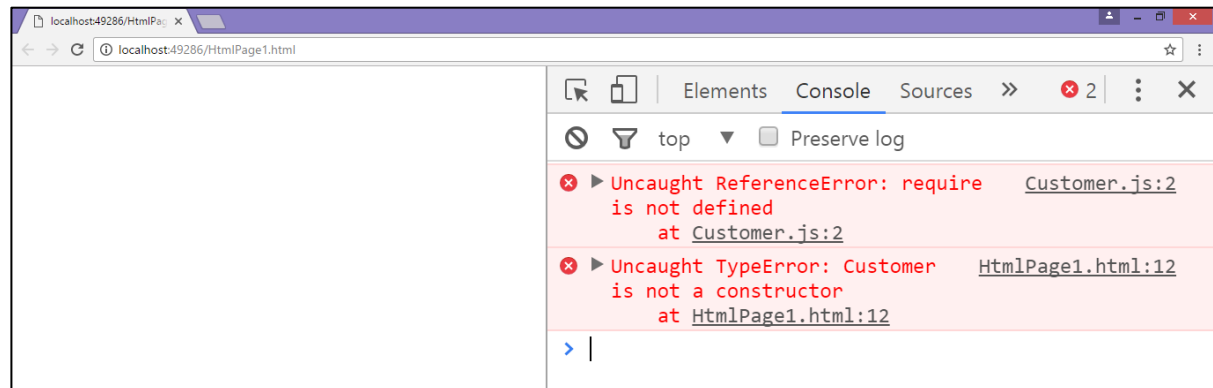
- Create separate class file and then reference them
- Create each class in separate .ts file
- The class which you want to reuse will be marked as *export*
- When you refer class which defined in other file you will use *import*

Code

```
export class Address {  
  area: string;  
  city: string;  
}  
import {Address} from "../Address";
```

```
class Customer {  
  custName: string;  
  add: Address;  
}
```

Error:



Export and require Key Words

- Class which you want to call / use from other class mark as ***export***
- The class which going to consume other class will use ***require*** key word

```
var Address = (function () {  
    ...  
})();
```

```
exports.Address = Address;
```

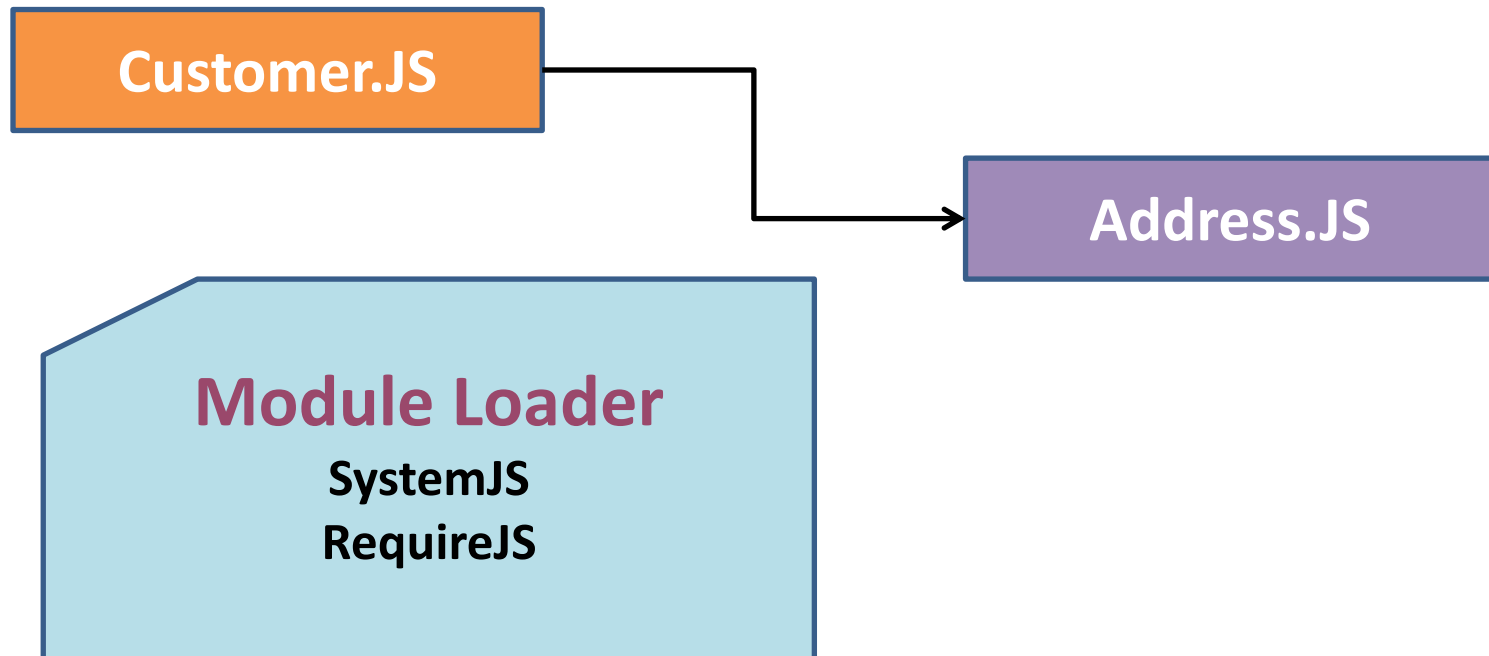
```
var Address_js_1 = require('./Address.js');
```

```
var Customer = (function () {  
    ...  
})();
```

```
exports.Customer = Customer;
```


Module Loader

- TypeScript works on “on demand loading” (Export pattern)
- So TypeScript needs *loader* like RequireJS, SystemJS
- So Address module require to be loaded when needed

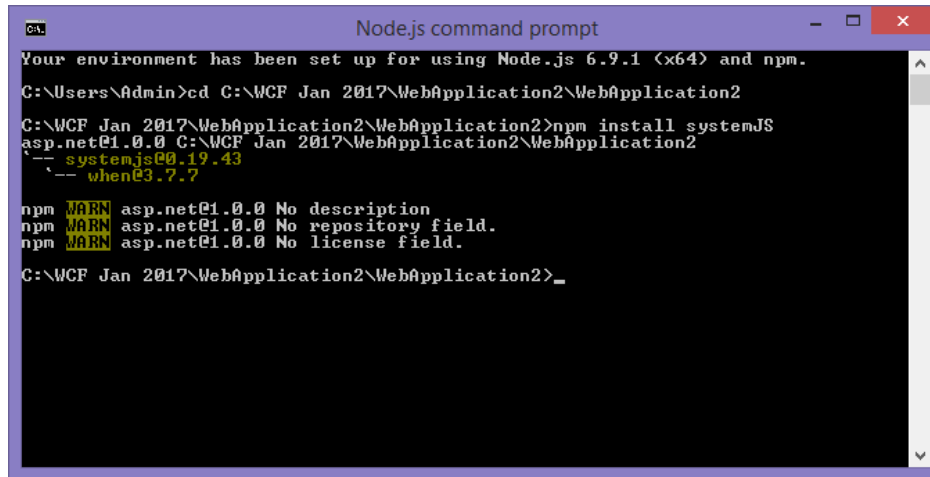


SystemJS

Install SystemJS

- Go to node command prompt
- Go to your folder and install systemJS (small letters)

npm install systemjs --save

A screenshot of a Windows command prompt window titled "Node.js command prompt". The window shows the following text: "Your environment has been set up for using Node.js 6.9.1 (x64) and npm." followed by the command "C:\Users\Admin>cd C:\WCF Jan 2017\WebApplication2\WebApplication2". The next line shows the command "C:\WCF Jan 2017\WebApplication2\WebApplication2>npm install systemJS". The output shows the package being installed: "asp.net@1.0.0 C:\WCF Jan 2017\WebApplication2\WebApplication2" followed by a tree view of the package structure: "├── systemjs@0.19.43" and "└── when@3.7.7". Below this, there are three warning messages: "npm WARN asp.net@1.0.0 No description", "npm WARN asp.net@1.0.0 No repository field.", and "npm WARN asp.net@1.0.0 No license field.". The prompt ends with "C:\WCF Jan 2017\WebApplication2\WebApplication2>_".

```
C:\Users\Admin>cd C:\WCF Jan 2017\WebApplication2\WebApplication2
C:\WCF Jan 2017\WebApplication2\WebApplication2>npm install systemJS
asp.net@1.0.0 C:\WCF Jan 2017\WebApplication2\WebApplication2
├── systemjs@0.19.43
└── when@3.7.7

npm WARN asp.net@1.0.0 No description
npm WARN asp.net@1.0.0 No repository field.
npm WARN asp.net@1.0.0 No license field.
C:\WCF Jan 2017\WebApplication2\WebApplication2>_
```

- This will create a folder “node_module” in your application folder
- Inside this folder you will get systemJS folder

Use of SystemJS

- So now we will use SystemJS to load out module
- Remove direct reference of Customer module

```
<script src="node_modules/systemjs/dist/system.js"></script>
<script>
    System.config({
        "defaultJSExtensions" : true
    });

    System.import("Customer").then(function (exports) {
        var cust = new exports.Customer();
    });
</script>
```

- Run in debug mode and check on demand loading

core-js

Shims, Polyfills, Core-js

- Core-js is a JavaScript polyfill framework
- It is an open source
- Download from Git using npm command
- *“PolyFill is an element to fix cracks on the wall”*
- So polyfill helps to fill up gaps in our JavaScript code
- JavaScript version ECMAScript 5, ECMAScript 6
- Polyfill frameworks “Ensures the new syntax of JavaScript will run on an old browser which are not supporting this new features”
- Core-js is used by Angular2 for polyfill

Usage of core-js

- Add html page and add following code

```
<script>
```

```
    var myarray = Array.from(new Set([1, 2, 3]));  
    alert(myarray);
```

```
</script>
```

- If you run this code in Google chrome this will work
- But if you run in IE this code will give error
- So now down load core-js
- npm install core-js
- Copy in your visual studio project
- Add reference on your html page and run same code in IE
- `<script src="node_modules/core-js/client/core.min.js"> </script>`

Zone.JS

Zone.js

- *Zone is an execution context that persist across async task*
- It creates single execution context for async task
- So when we execute multiple async task, its create single context for executing all task

Demo

- We want to run two JavaScript function and want to check time taken by this two function for execution

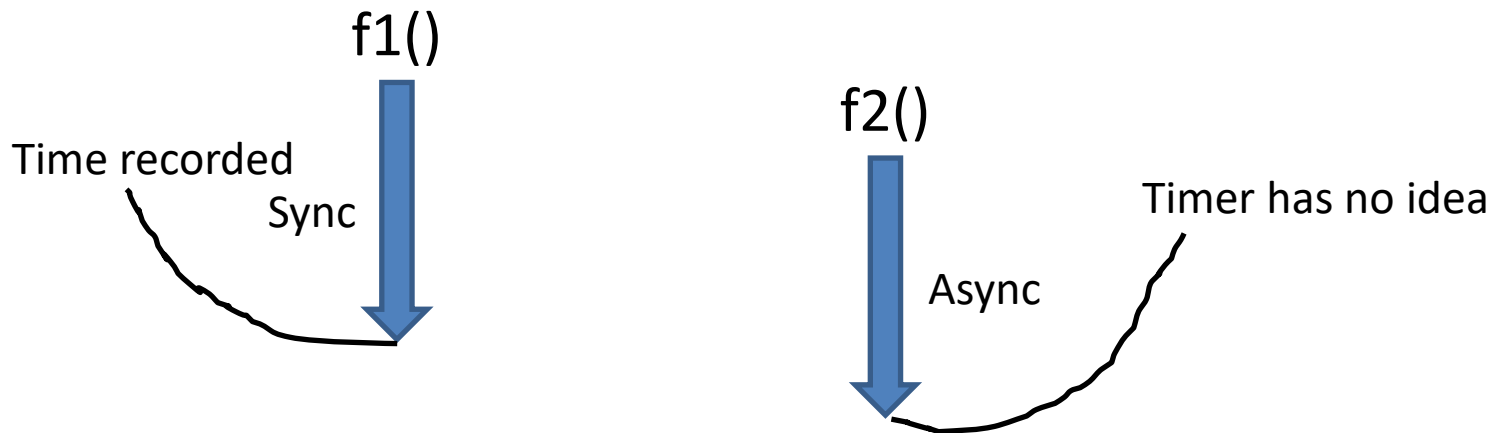
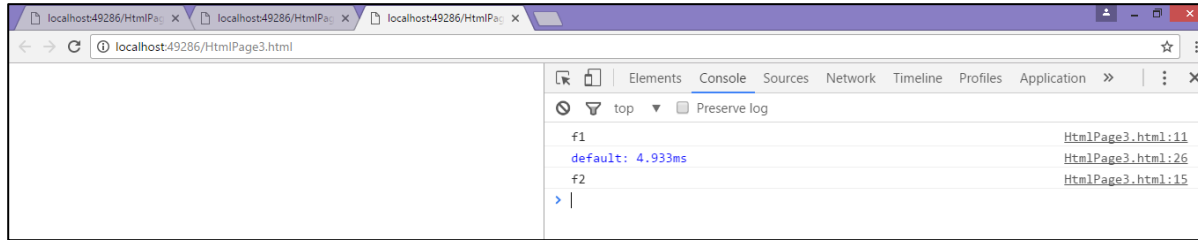
```
<script>
function f1() {
    console.log("f1");
}

function f2() {
    console.log("f2");
}

function OneUnit() {
    f1();    //sync call
    //setTimeout is async call
    setTimeout(f2, 5000); //async call. execute f2() after 5 seconds
}
console.time(); //start watch
OneUnit();
console.timeEnd(); //stop watch
</script>
```

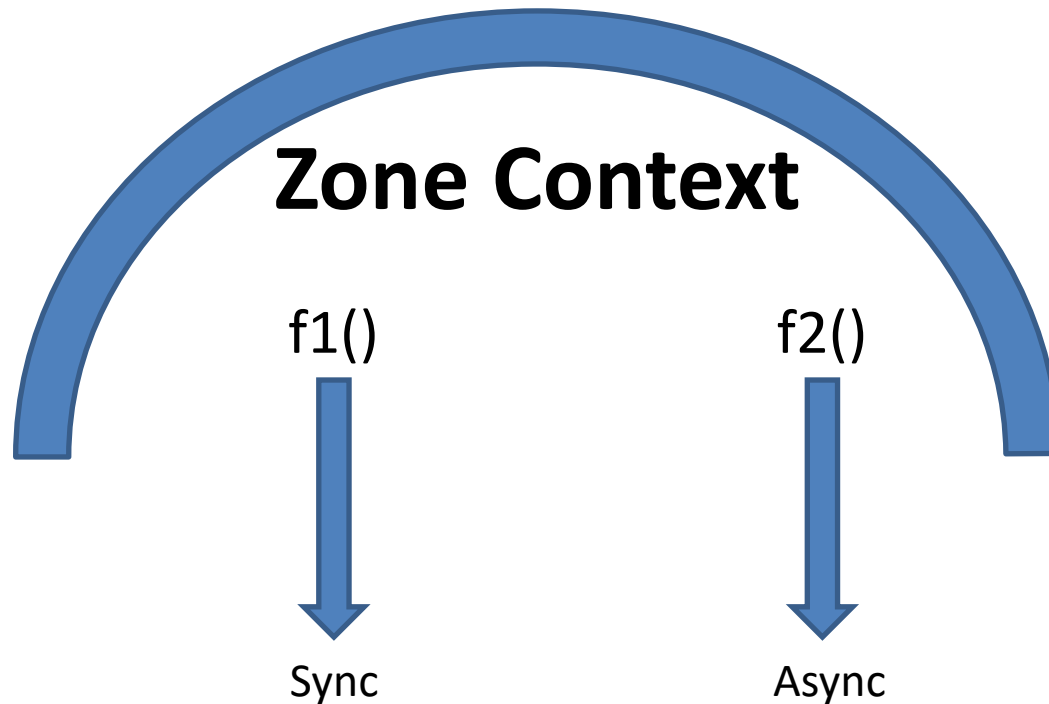
Output

- If you see the output in Google chrome developer command prompt you will get time immediate after first function call because second function is running asynchronously



Solution

- We want to create this as one context so we can measure both the functions call together
- So zone will help us to create them as one unit



Install & Add Zone Reference

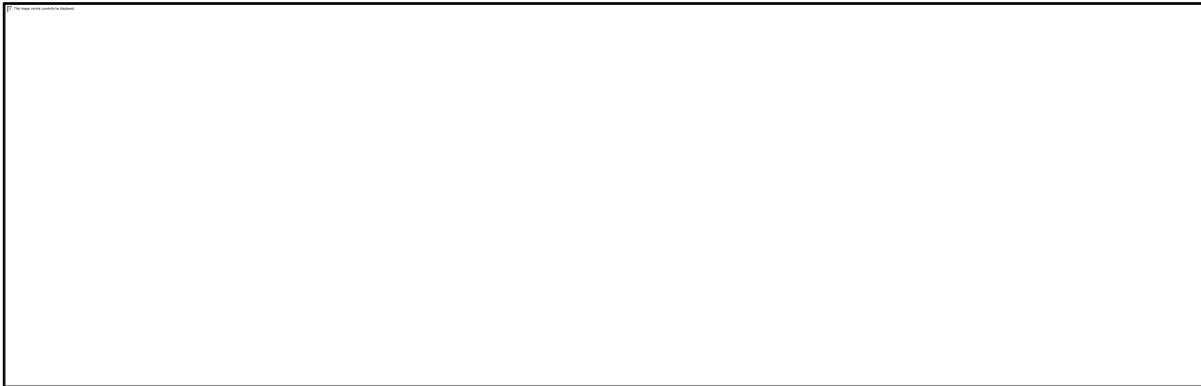
- Install Zone

npm install zone.js

- Add reference in your html page
- `<script src="node_modules/zone.js/dist/zone.js"></script>`
- Add reference

Code

```
var z = Zone.current.fork({  
  onHasTask: function (parent, current, target, hasTask) {  
    console.timeEnd();  
  }  
});  
  
console.time();  
z.run(function () {  
  OneUnit();  
});
```



Reflect-Metadata

Reflect-Metadata

- Decorators
- Decorators can be attached to class or method
- In C# / Java called as attribute
- Angular 2 is heavily based on metadata

```
export class Customer {  
    @Reflect.metadata("obsolete", "true")  
    add() {  
        console.log("Inside add()");  
    }  
}
```

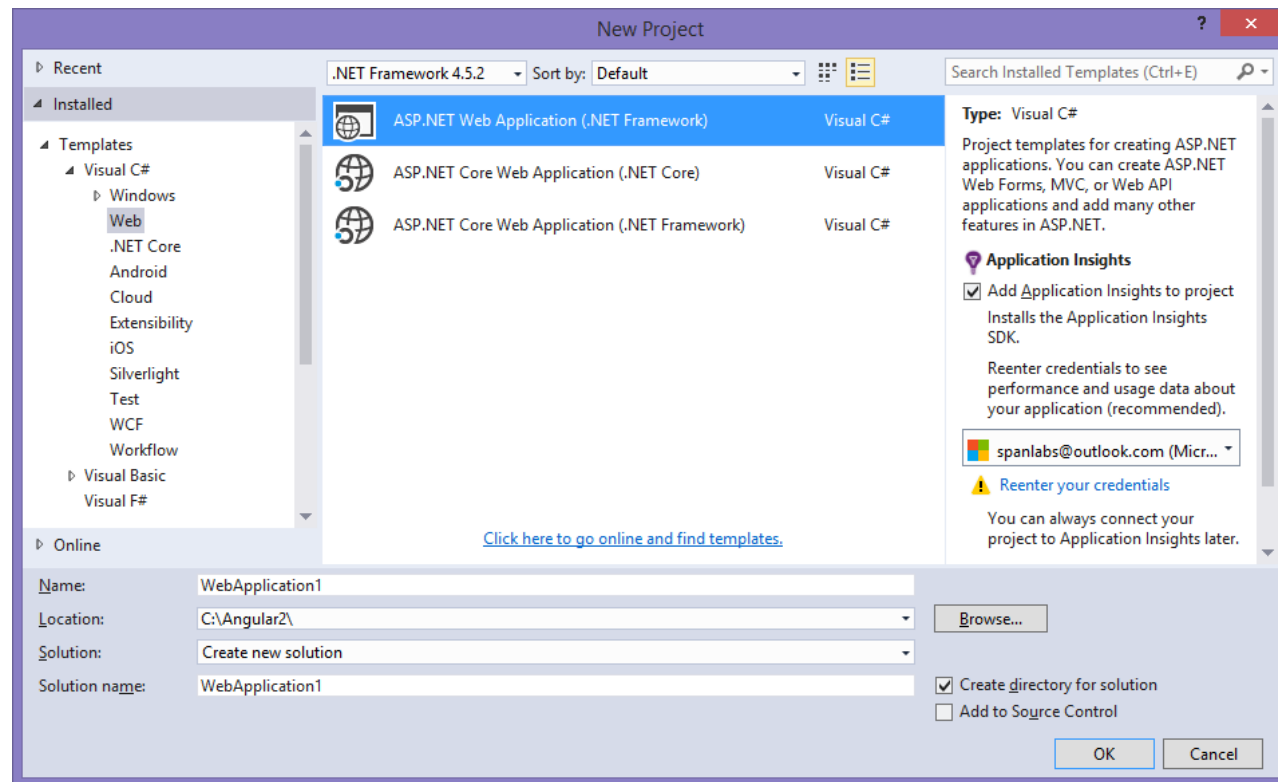

Angular 2

Requirement

- Angular CLI
 - Command line interface for Angular Apps
- Editor
 - Atom
 - Webstrom
 - Visual Source code
- IDE
 - Eclipse
 - **Visual Studio (2015)**
- Node Command Prompt
 - node_module

First Project

- Start new ASP.NET Web Application
- Select template as Empty



Configure Project

- Add following config file in the project
 - package.json
 - All the necessary JavaScript Framework we need in our project
 - tsconfig.json (TypeScript configuration)
 - typings.json
 - Systemjs.config.js

Configuration File

- package.json
 - Add new item and select “npm configuration file”
 - Name it as package.json
- tsconfig.json
 - Add new item and select “TypeScript JSON configuration File”
 - Name it as tsconfig.json
- typings.json
 - Add new item and select “TypeScript JSON configuration File”
 - Name it as typings.json
- SystemJS.config.js
 - Add new item and select “JavaScript file”
 - Name it as SystemJS.config.js

Install Node Modules

- Add node_module manually
- Close visual studio
- Start node command prompt
- Go to Application folder
- Execute the command
- `npm install`
- This will download all necessary JavaScript files specified in `package.json` file and store inside `node_modules` folder
- After downloading all JS file, Open your project in Visual Studio and now exclude `package.json`, `typings.json` and `node_modules`

Start Coding

- Create three folders
 - Models
 - UI
 - Binder (Code behind / ViewModel / Controller / Presenter)

HTML Page

- In Views folder add HTML Page and name it as “Customer.HTML”
- Add three Textboxes

```
<div>
```

```
Customer Id : <input type="text" /> <br/>
```

```
Customer Name : <input type="text" /> <br />
```

```
Customer Amount: <input type="text" /> <br />
```

```
</div>
```

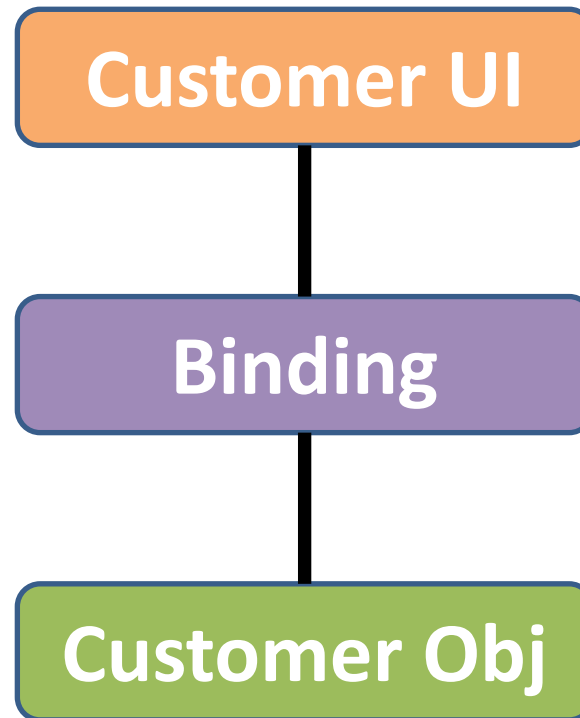

Model Class

- In Models folder add TypeScript file to define model class
- Name the file as Customer.ts
- Add three properties

```
export class Customer {  
    CustomerId: string = "";  
    CustomerName: string = "";  
    CustomerAmount: number = 0;  
}
```

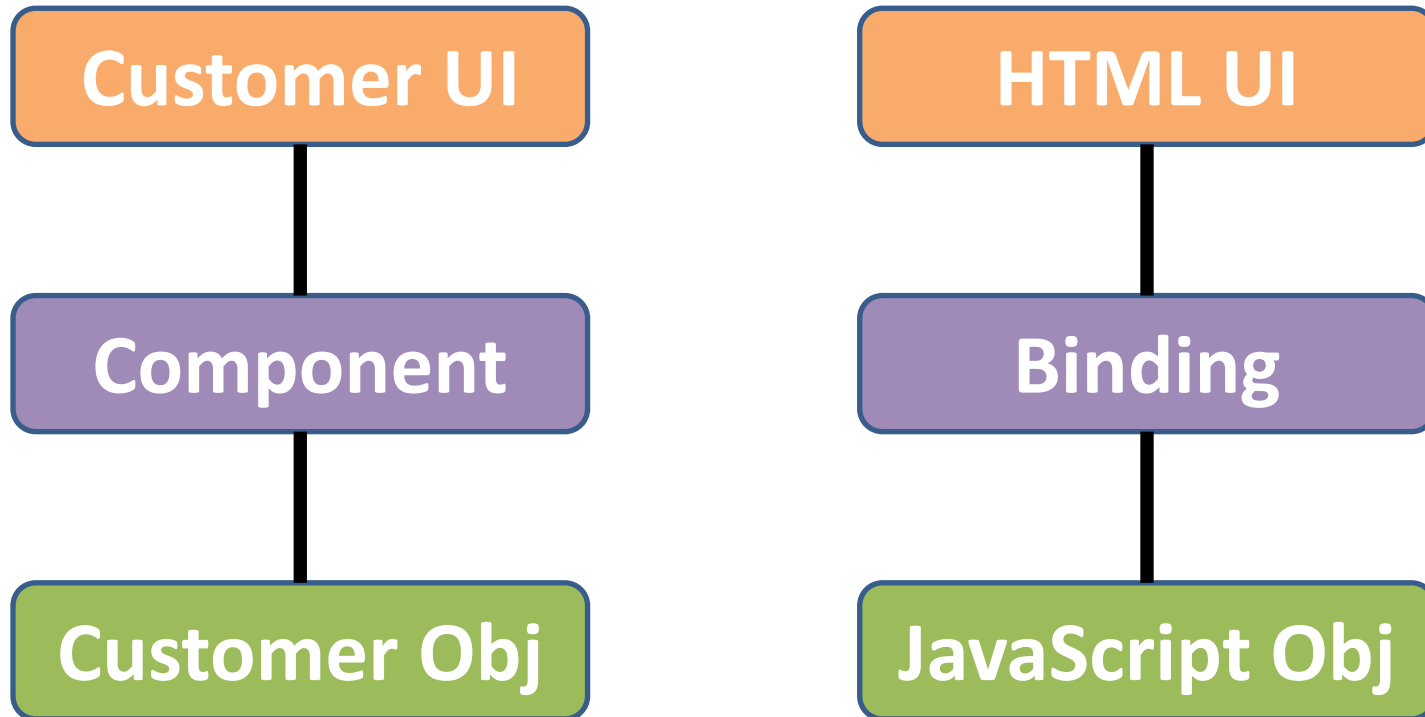
Binding

- We want to Bind Customer object to the view
- Angular is a “Binding Framework”
- Angular will help us to bind model to the view using Binder code



Component

- In angular 2 UI and object gets bind through the *component*
- Component is a heart of angular



Binder

- In Binder folder add two sub folder
 - Component
 - Module
- Add TypeScript file in component folder
- Name it as “CustomerComponent.ts”

```
import {Component} from "@angular/core";  
import {Customer} from "../../Model/Customer";
```

```
@Component({  
  selector : "customer-ui",  
  templateUrl: "../../UI/Customer.html"  
})  
export class CustomerComponent {  
  customerObj: Customer = new Customer();  
}
```

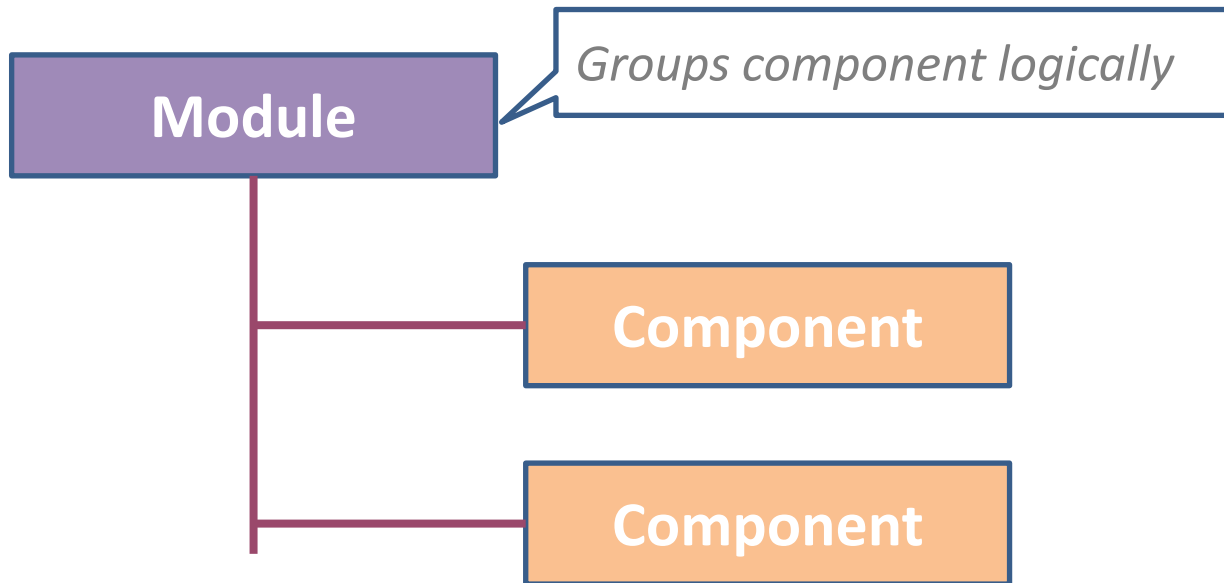
Decorator

- Help us to load component into UI

```
@Component({  
  selector : "customer-ui",  
  templateUrl: "../UI/Customer.html"  
})
```

Module

- Module is a collection of components
- Module groups component logically (like namespace in C#)



Module code

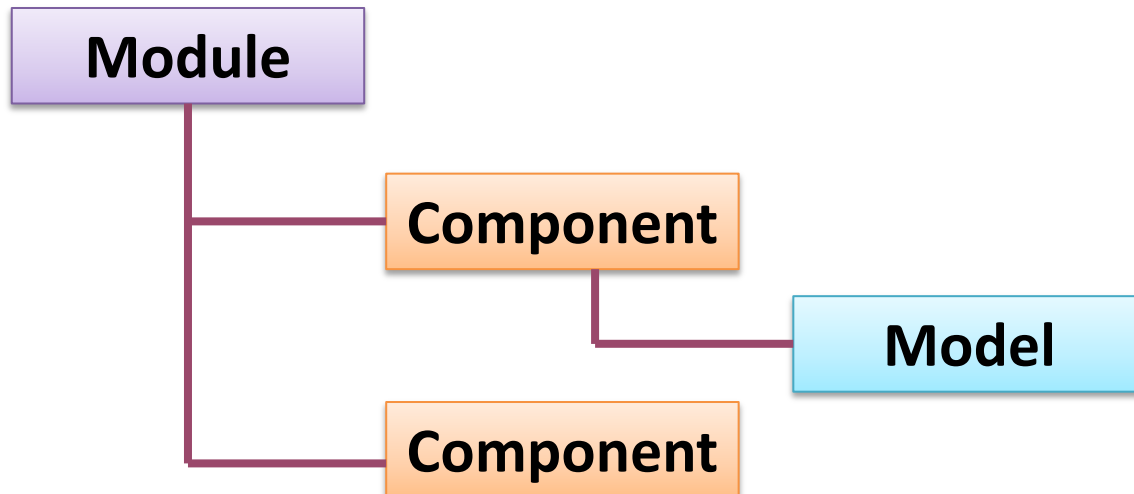
```
import {NgModule} from "@angular/core";
import {FormsModule} from "@angular/forms";
import {BrowserModule} from "@angular/platform-browser";
import {CustomerComponent} from
"../Component/CustomerComponent";
```

```
@NgModule({
  imports: [BrowserModule, FormsModule],
  declarations: [CustomerComponent],
  bootstrap: [CustomerComponent]
})
export class MainModule {

}
```

Hierarchy & Start-up

- We have Module
- Module has components
- Component refer lots of Model
- Now to load this complex hierarchy, angular has given start-up file
- Inside the start-up file you can load the module, and internally it will load all components and models



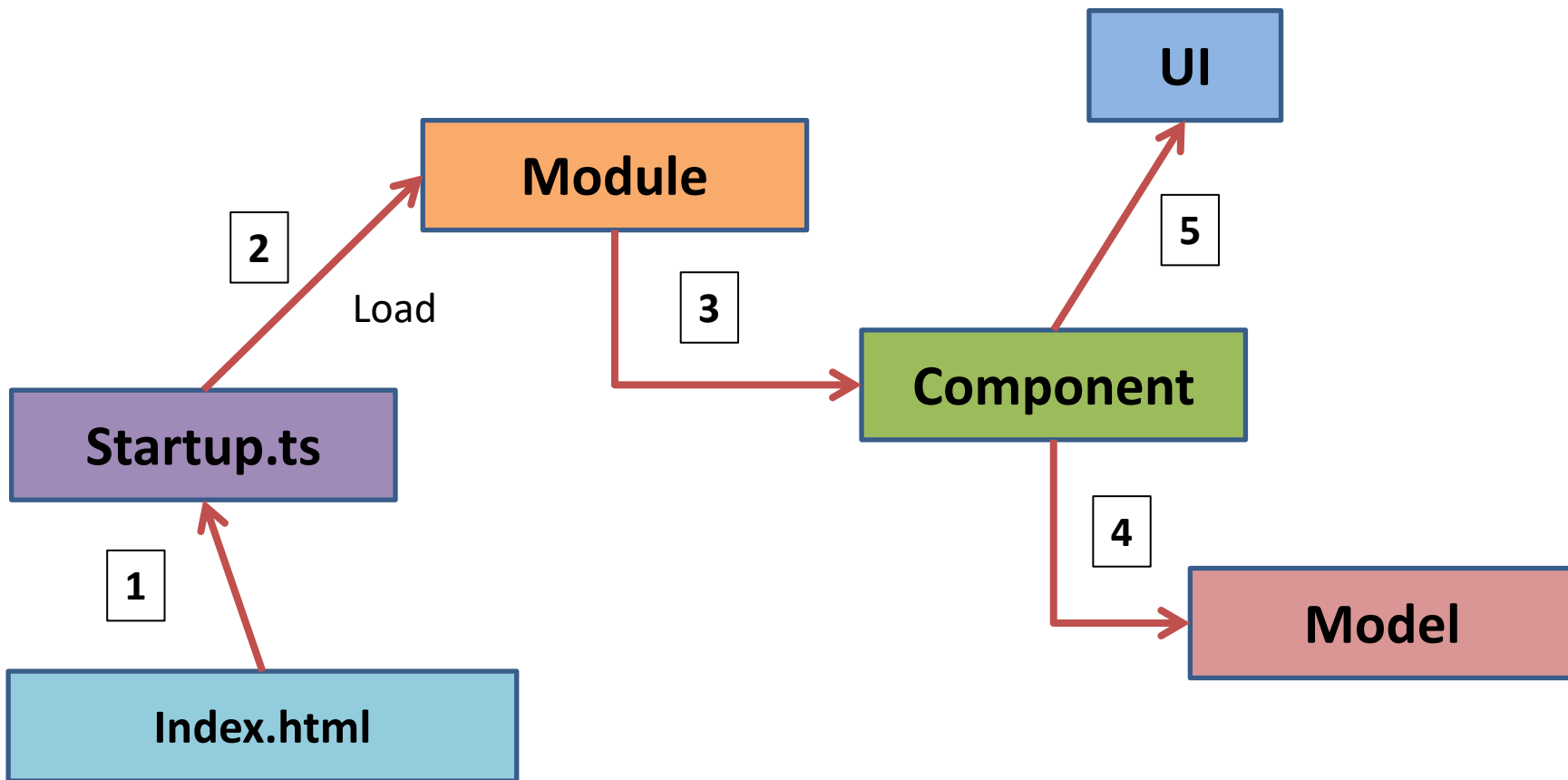
Bootstrap Module

- Add new folder as Startup
- Add TypeScript file as Startup.ts

```
import {platformBrowserDynamic} from "@angular/platform-  
browser-dynamic";  
import {CustomerModule} from "../Binder/Module/MainModule";  
  
const platform = platformBrowserDynamic();  
platform.bootstrapModule(MainModule);
```

Bootstrap Angular

- Add html file in UI folder as StartAngular.html
- This file will start angular
- Mark this file as startup file
- Compile code and run HTML file



Visual Source Code Editor

- VS Code is a lightweight open source editor from Microsoft
- Available for Windows, macOS and Linux
- It comes with built-in support for JavaScript, TypeScript and Node.js
- And has a rich ecosystem of extensions for other languages (such as C++, C#, Python, PHP, Go) and runtimes (such as .NET and Unity).

Start Project in VS Code

- In Windows Explorer create a folder in which we want to save all out Angular project files
- Start VS Code Editor
- From “File” menu select “Open Folder” option and select the folder which we created to store project files
- Add 4 base file i.e. Package.json, tsconfig.json, typings.json, systemjs.config.js
- Save all

Download JavaScript Frameworks

- Download node.js from Node website and use node command prompt or Visual Studio Code editor will provide command prompt from there execute command
- Go to node command prompt or VS Code Editor command prompt
- Go to folder path
- Type command as “npm install”
- This will generate node_module folder with JS framework files

Compile TypeScript Code

- Write code as we discuss earlier
- To compile the code we will use TypeScript compiler “tsc”
- Go to “View” menu and select option “Integrated Terminal”
- Type the command “tsc”
- On successful compilation for every .ts file will generate .js and .map file

Web Server : http-server

- http-server is a simple, zero-configuration command-line http server
- It is powerful enough for production usage, but it's simple and hackable enough to be used for testing, local development, and learning
- First install the server globally

npm -g http-server

- Usage

http-server [path]

Run Web Site

- Start Browser
- Type the address

<http://localhost:8080/ui/index.html>

- Will display your Page

Hide JavaScript Files in Visual Studio Code

- Select option from menu File -> Preferences -> Workspace Settings
- To hide any files (not just JavaScript files) you add the following code

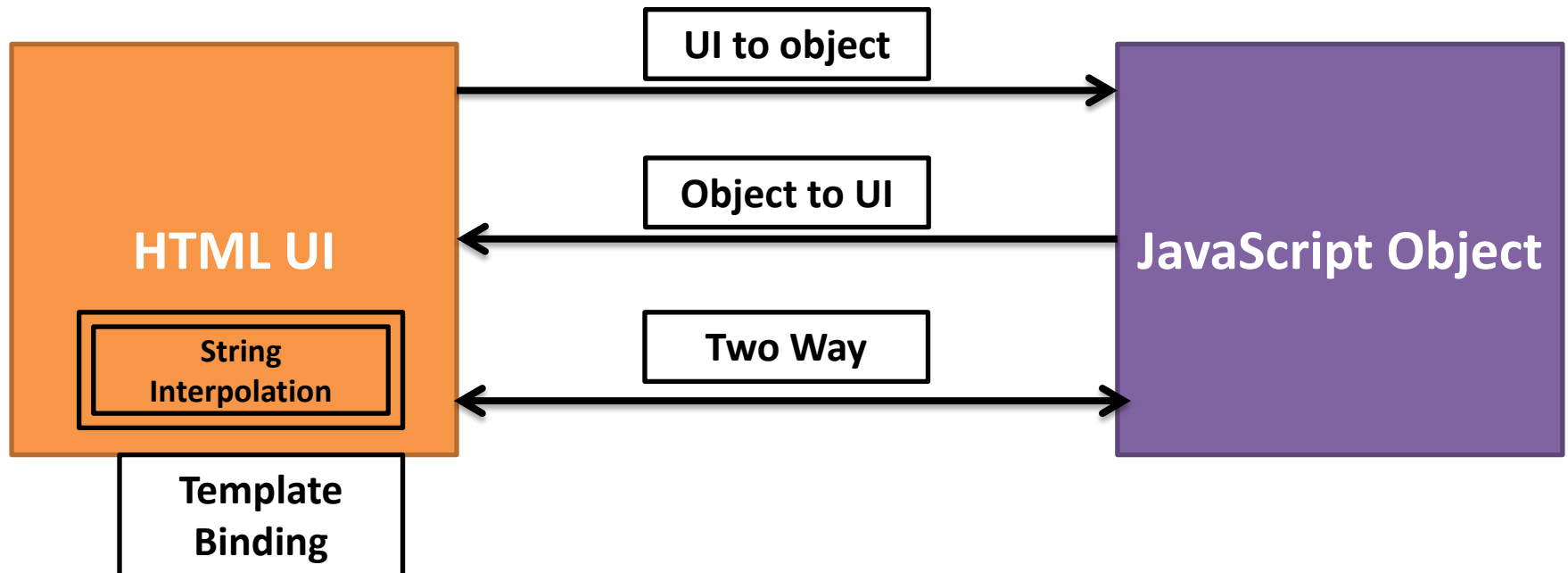
```
{  
  "files.exclude": {  
    "**/.git": true,  
    "**/*.js": true,  
    "**/*.js.map": true  
  }  
}
```

- True means hide the file with that extension and false means show the file with that extension
- Save the changes. This will generate settings.json file inside .vscode folder

Data Binding

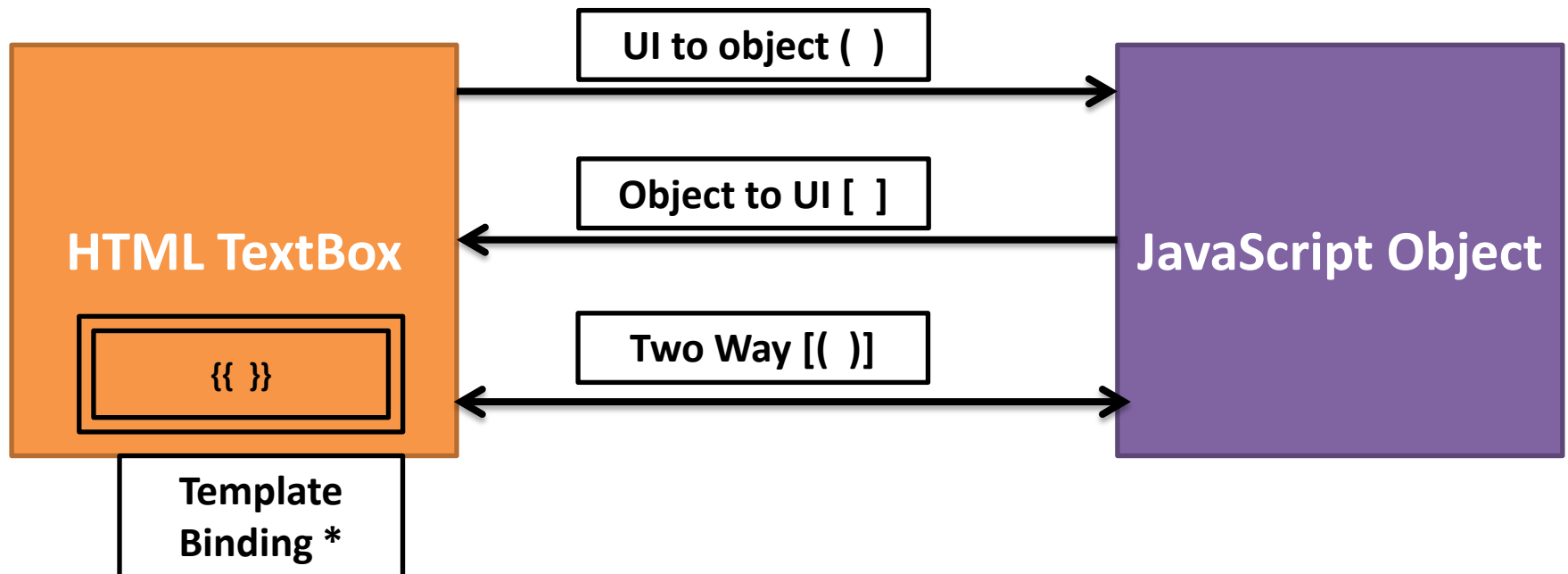
Databinding

- Databinding means communication
- Interaction with data
- Interaction with template and business logic



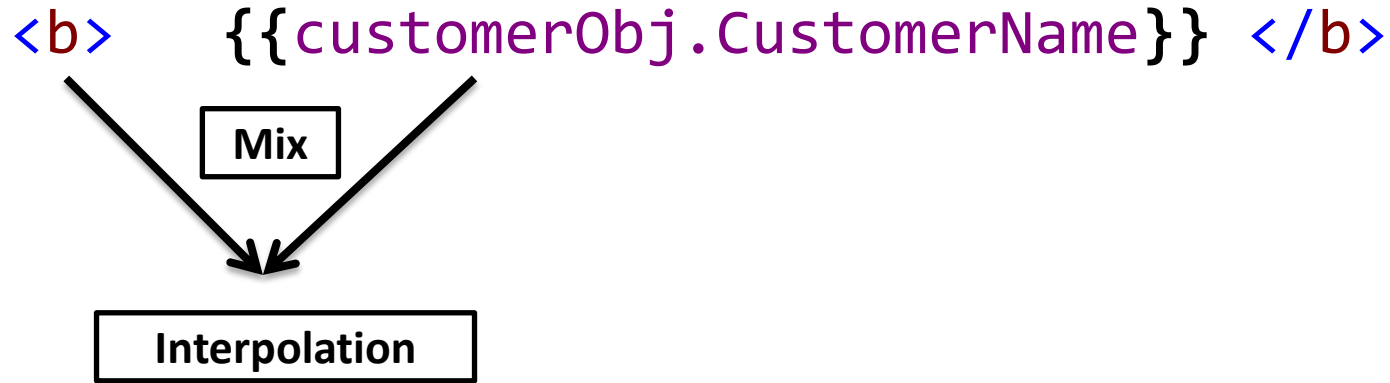
Data Binding Brackets

- [] – Receive data from object to UI – Property Binding
- () – send data from UI to object – Event Binding
- [()] – Two way binding (use both the brackets)



String Interpolation

- `{{ }}` – string interpolation
 - means mix angular value with HTML



Databinding Methods

String Interpolation

`{{Expression resolving to a string }}`

Property Binding

`<button [disabled]="expression" />`

Binding / sending data to properties in your HTML DOM

Event Binding

`<button (click)="expression handling the event" />`

Listen to Event

Two-way Binding

`<input [(ngModel)] = "bound model (e.g object) />`

Sending and listening both

Directive

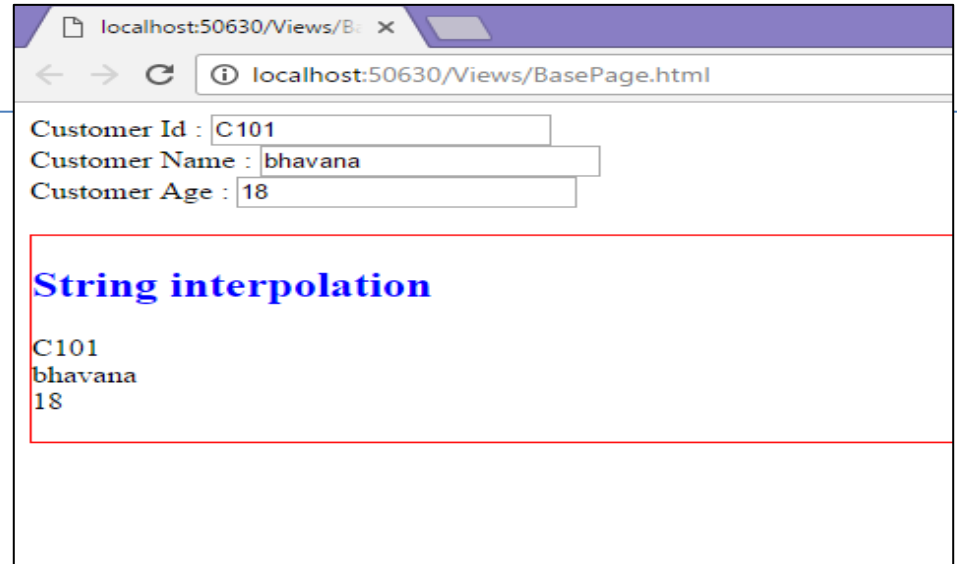
- Three types of directive in Angular
 - **Components** – directive with a template
 - **Structural directive** - change the DOM layout by adding and removing DOM elements. Example *ngIf, *ngFor
 - **Attribute directive** - change the appearance or behaviour of an element, component, or another directive. Example ngStyle, ngClass

Databinding Demo

```
<div>
  CustomerId : <input [(ngModel)]="customerObj.CustomerId" />
<br />
  CustomerName : <input [(ngModel)]="customerObj.CustomerName"
/> <br />
  CustomerAge : <input [(ngModel)]="customerObj.CustomerAge" />
<br />
  <div [ngClass]="{redborder : true}">
    <h2 [ngStyle]="{color : 'blue'}">String interpolation</h2>
    {{customerObj.CustomerId}} <br/>
    {{customerObj.CustomerName}} <br />
    {{customerObj.CustomerAge}} <br /> <br/>
  </div>
  <button (click)="onClicked()">Click Me!</button>
</div>
```

Component Code

- Code for style and event




```
@Component({
  selector : "customer-ui",
  templateUrl: "../Views/Customer.html",
  styles: ['.redborder {border : 1px solid red }']
})
export class CustomerComponent {
  customerObj: Customer = new Customer();

  onClicked() {
    alert("Button clicked");
  }
}
```

Template Binding (*)

- Add a button and on click we want to add customer record in memory collection and bind to grid (Table)



The screenshot shows a web browser with three tabs. The active tab is titled 'localhost:50630/Views/BasePage.html'. The page content includes the AngularJS logo and the text 'Angular 2 Web Site'. Below this, there is a form with three input fields: 'Customer Id :', 'Customer Name :', and 'Customer Age : 0'. A button labeled 'Add Customer' is positioned below the form. Underneath the button, a table displays a list of customers with columns 'CustomerId', 'CustomerName', and 'CustomerAge'.

CustomerId	CustomerName	CustomerAge
101	bhavana	25
102	test	21

Designed by Bhavana Desai

Code for CustomerComponent

```
export class CustomerComponent {  
    currentCustomer: Customer = new Customer();  
    //create new customers collection  
    customers: Array<Customer> = new Array<Customer>();  
  
    //add current customer to collection  
    Add() {  
        this.customers.push(this.currentCustomer);  
        //clear the object so textboxes  
        this.currentCustomer = new Customer();  
    }  
}
```


Select()

- Provide Hyperlink next to each record in table and on click of a link display the record in respective text boxes



localhost:50630/Views/BasePage.html#

 **Angular 2 Web Site**

ANGULARJS

Customer Id :

Customer Name :

Customer Age :

CustomerId	CustomerName	CustomerAge	
101	bhavana	25	Select
102	test	21	Select

Designed by Bhavana Desai

Code for Select

```
Select(selectedCustomer: Customer) {  
    this.currentCustomer = selectedCustomer;  
}
```

```
<tr *ngFor="let item of customers">  
    <td>{{item.CustomerId}}</td>  
    <td>{{item.CustomerName}}</td>  
    <td>{{item.CustomerAge}}</td>  
    <td><a href="#" (click)="Select(item)" >Select</a></td>  
</tr>
```

- Try to manipulate
- You can see the changes because it's a reference binding

Clone

- We do not want our current object to modify when we type in text box
 - So when we select we want clone of the object to bind with text box, so when we change the value in text box, till the time we will not click update button no changes will be updated
 - `Object.assign()` method will create the clone of given object
- ```
this.currentCustomer = Object.assign({}, selectedCustomer);
```

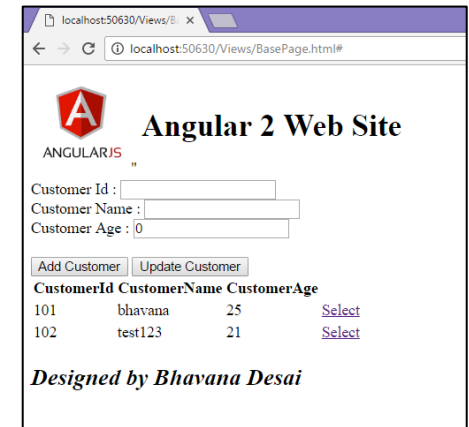




# Update Changes

```
Update() {
 for (let customer of this.customers) {
 if (customer.CustomerId ==
 this.currentCustomer.CustomerId) {
 customer.CustomerName =
 this.currentCustomer.CustomerName;
 customer.CustomerAge =
 this.currentCustomer.CustomerAge;
 }
 }
 this.currentCustomer = new Customer();
}
```

```
<input type="button" value="Update Customer"
(click)="Update()" />
```



# Cancel Current Update (Clear)

---

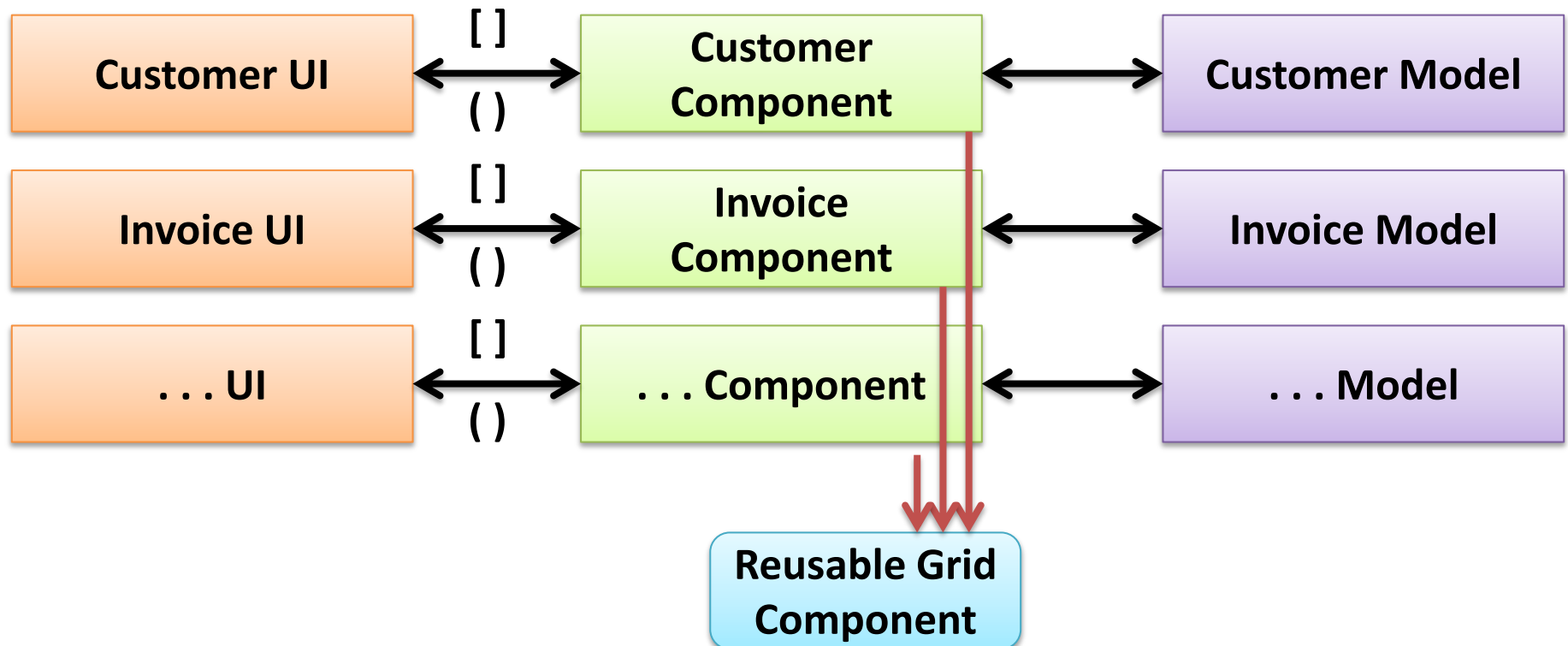
```
Clear() {
 this.currentCustomer = new Customer();
}
```

```
<input type="button" value="Cancel" (click)="Clear()" />
```

**Reusable Component**

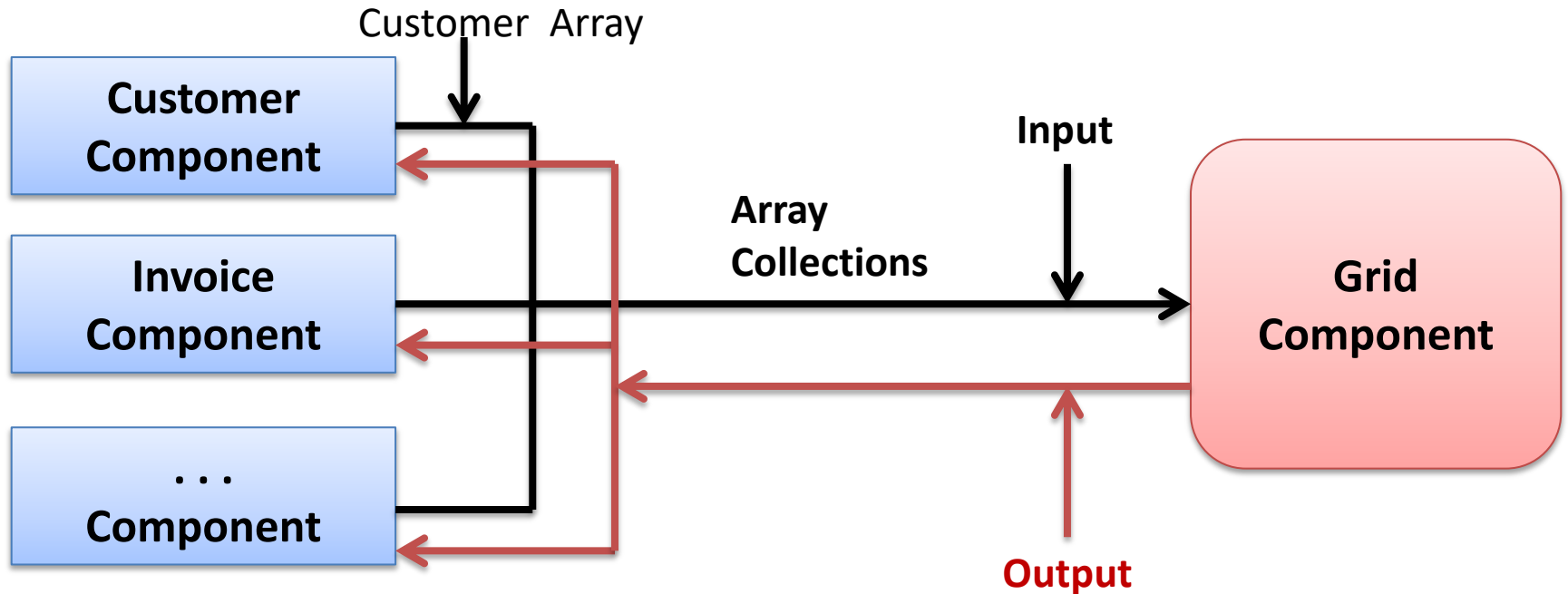
# Reusable Angular Component

- Reusable angular component – Generic Component
  - Reusable component can be consumed inside another angular component
- @Input()
- @Output()



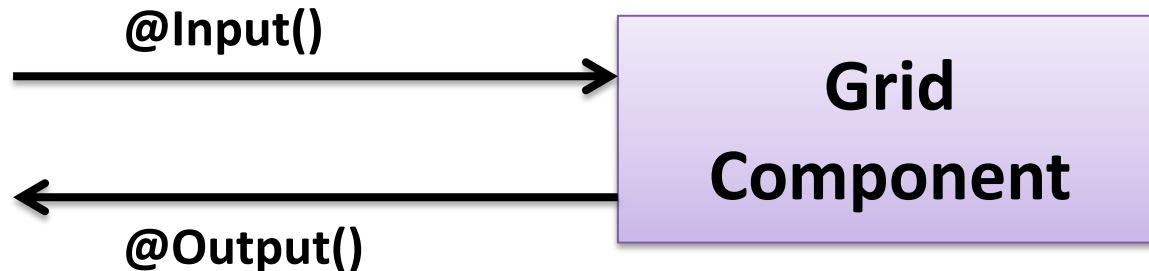
# Generic component

- In our example Grid component should not be aware of with which component it is getting attached



# @Input() & @Output() Decorator

- Pass data into component dynamically
- **@Input()** decorator
  - Defines input for a component
  - In our example grid component will receive the collection
- By binding event component can give output
  - Create custom event by using **@Output()** decorator



# Code : Add New Grid Component

---

- Inside Component folder add new TypeScript file
- Name it as GridComponent

```
import {Component, Input, Output} from "@angular/core";

@Component({
 selector: "grid-ui",
 templateUrl: "../../Views/Grid.html"
})

export class GridComponent {

}
```

- Also add HTML file Grid.HTML in View folder

- 
- Grid component first receive the collection and when you select it will return selected object

- `//declare generic collection for data`

```
gridData: Array<Object> = new Array<Object>();
```

- `//generic column collection`

```
gridColumns: Array<Object> = new Array<Object>();
```

- `//Entity Name`

```
EntityName: string = "";
```



# Pass Grid Data Collection

```
set gridDataSet(_gridData: Array<Object>) {
 if (_gridData.length > 0) {
 if (this.gridColumns.length == 0) { //Add column heading
once
 //fill column names in gridColumns collection
 var columnNames = Object.keys(_gridData[0]); //JavaScript
method to which pass the first object of the collection
 for (var index in columnNames) {
 this.gridColumns.push(columnNames[index]);
 }
 }
 }
 this.gridData = _gridData;
}
```

# Grid.Html

---

Title : Displaying records for {{EntityName}}

```
<table>
 <tr>
 <td *ngFor="let colName of gridColumn">
 {{colName}}
 </td>
 </tr>
 <tr *ngFor="let row of gridData">
 <td *ngFor="let colName of gridColumn">
 {{row[colName]}}
 </td>

 </tr>
</table>
```

# Call Grid Component

---

- Call GridComponent inside Customer Component

```
<div>
```

```
 . . .
```

```


```

```
 <grid-ui [grid-data]="customers" [grid-
entityname]="'Customer' ">
```

```
 </grid-ui>
```

```
</div>
```

# Define Input in Component

---

- In Grid Component define @Input
- @Input("grid-data")
- set gridDataSet(\_gridData: Array<Object>) { }
- @Input("grid-entityname")
- EntityName: string = "";

# Load Grid Component

---

- Go to HomeModule and load GridComponent

. . .

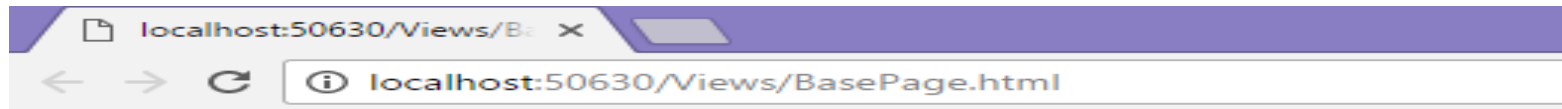
```
import {GridComponent} from "../Component/GridComponent";
```

```
@NgModule({
 imports: [BrowserModule, FormsModule],
 declarations: [CustomerComponent, GridComponent],
 bootstrap: [CustomerComponent]
})

export class CustomerModule {

}
```

# Build & Test



ANGULARJS

..

## Angular 2 Web Site

Customer Id :   
Customer Name :   
Customer Age :

Add Customer

Update Customer

Cancel

**Title : Displaying records for Customer**

*Designed by Bhavana Desai*

# Display Data

---

- When you try to add data nothing is getting added in our grid
- For testing bind EntityName with Customer Name

```
<grid-ui [grid-data]="customers"
```

```
[grid-entityname]="currentCustomer.CustomerName">
```

- Now if you run and test
- You will see that name is getting binded
- Then why grid is not populating with our object

# Change Detection

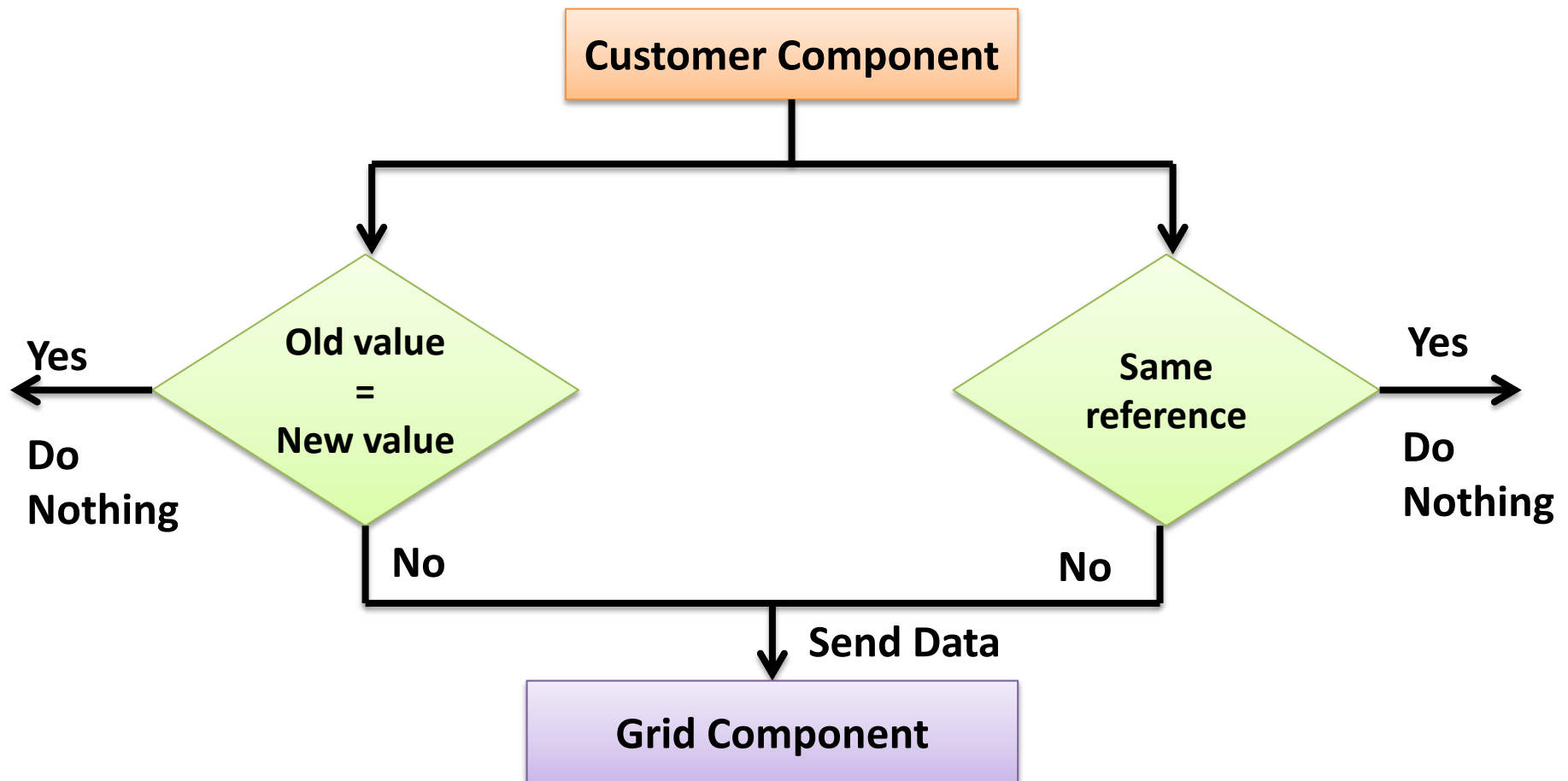
---

- Angular change detection process
- Customer Component only send data to Grid Component when there is some kind of change
- A change for angular is the change of actual value
- In case of text box when we type inside text box it's changing the value
- When we add customer object, we add it to customers collection
- Angular detects change to collection when object reference change
- So here by adding element to a collection , reference is not changing and because of this it's not sending data to Grid Component



# Change Detection

- When we add customer to a collection, we are adding element to a collection, which is not changing the collection reference.
- So it's not sending any change signal to Grid Component



# Refresh Collection

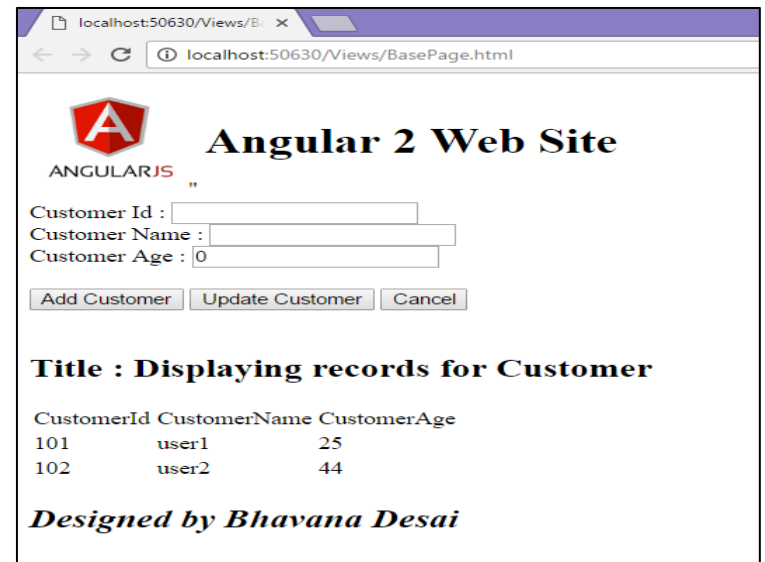
---

- How to refresh collection so angular detect change
- Refresh complete Customer Collection with a new reference
- Create a copy (fresh copy – clone) of a collection
- Slice() method of JavaScript helps, which creates a fresh copy of collection
- So it's a new reference (fresh reference)
- So angular detect the change and send data to Grid Component

# Customer Component Add() Method

```
Add() {
 this.customers.push(this.currentCustomer);
 //new fresh reference
 this.customers = this.customers.slice();
 //clear the object so textboxes
 this.currentCustomer = new Customer();
}
```

- Build, run & test



# @Output()

---

- **Input** is to send data to the component (*Customer Component sending customers collection to Grid Component*)
- **Output** is to send data back to the parent component (*send customer object to Customer Component from Grid Component*)
- Notifying parent components that something has changed, via events
- Create custom event by using @Output() decorator and EventEmitter
- EventEmitter send the event back to the parent component along with data

# Code - GridComponent

---

```
import {Component, Input, Output, EventEmitter} from
"@angular/core";

//selected is an event which emit out an object back to the
parent component
@Output("grid-selected")
selected: EventEmitter<Object> = new EventEmitter<Object>();
//selected is an event which is getting raised from Select()
method
Select(_selected: Object) {
 this.selected.emit(_selected); //emit out selected
object to the parent component
}
```

# HTML code

```
<td>Select</td>
```

```
<!-- Call Select(object) method of GridComponent
```

```
Which internally raise Selected event and emit Selected
object to the parent component Select($event) method via
$event property-->
```

```
-----Customer.HTML
```

```
<grid-ui [grid-data]="customers" [grid-
entityname]="'Customer'" (grid-selected)="Select($event)">
```

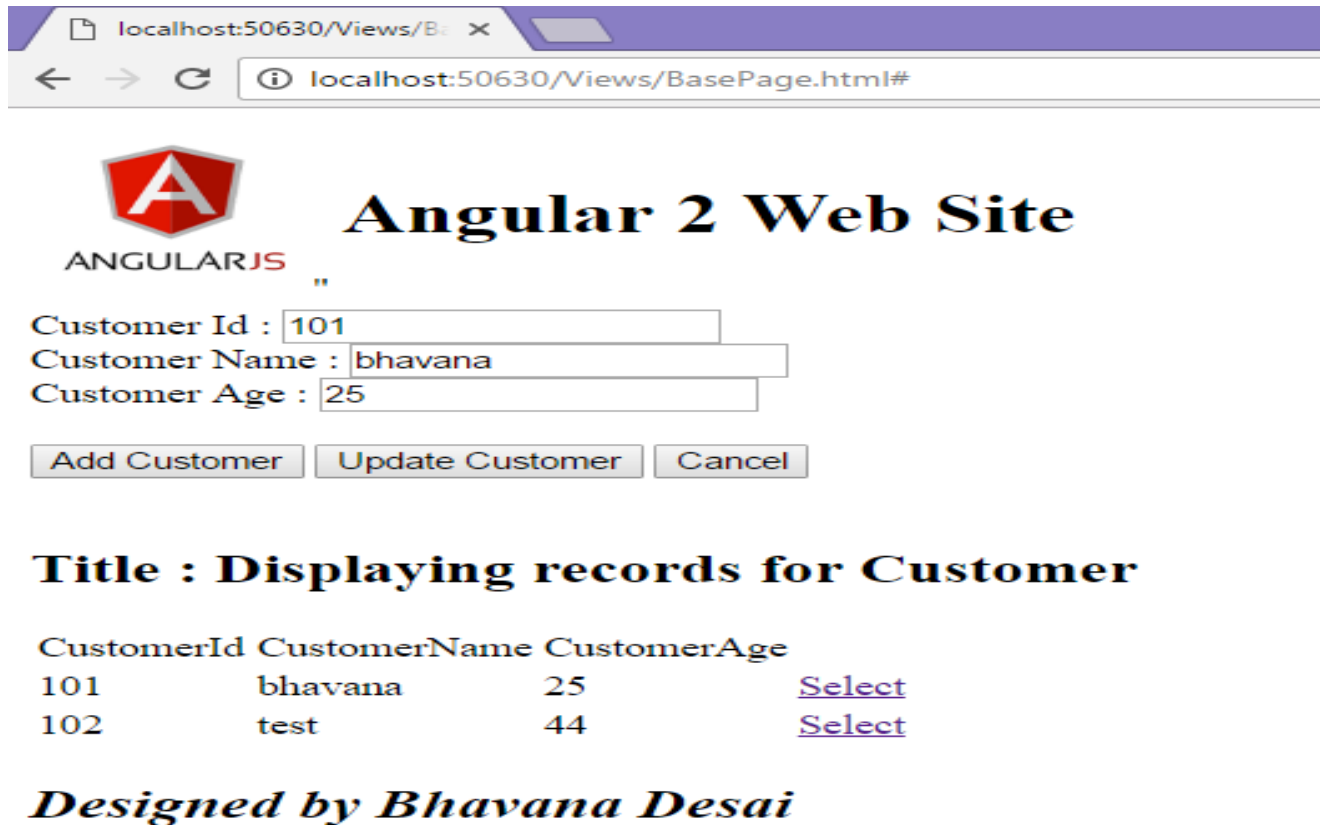
```
<!-- Select($event) is a method of CustomerComponent.
```

```
when we call the (grid-selected) event of GridComponent
it's emit selected object and we pass that object via $event
property to Select () method of the Customer Component -->
```

```
</grid-ui>
```


# Final Output of the Code

- Run and Test



localhost:50630/Views/B: x

localhost:50630/Views/BasePage.html#

 **Angular 2 Web Site**

ANGULARJS

Customer Id : 101

Customer Name : bhavana

Customer Age : 25

Add Customer Update Customer Cancel

**Title : Displaying records for Customer**

CustomerId	CustomerName	CustomerAge	
101	bhavana	25	<a href="#">Select</a>
102	test	44	<a href="#">Select</a>

*Designed by Bhavana Desai*

**Pipes**



# Pipes

---

- AngularJS 1.x we call it Filters
- A pipe takes in data as input and transforms it to a desired output
- Convert text to upper case
- Number formatting with two decimal

**test**      **Format** → **TEST**  
                 **Transform**

**2500.2367**      **Format** → **2,500.24**  
                 **Transform**

# Types of Pipe

---

- Two types of pipes
  - Inbuilt Angular pipe
  - Custom pipe / User defined pipe

# Inbuilt Pipe

---

- Uppercase pipe

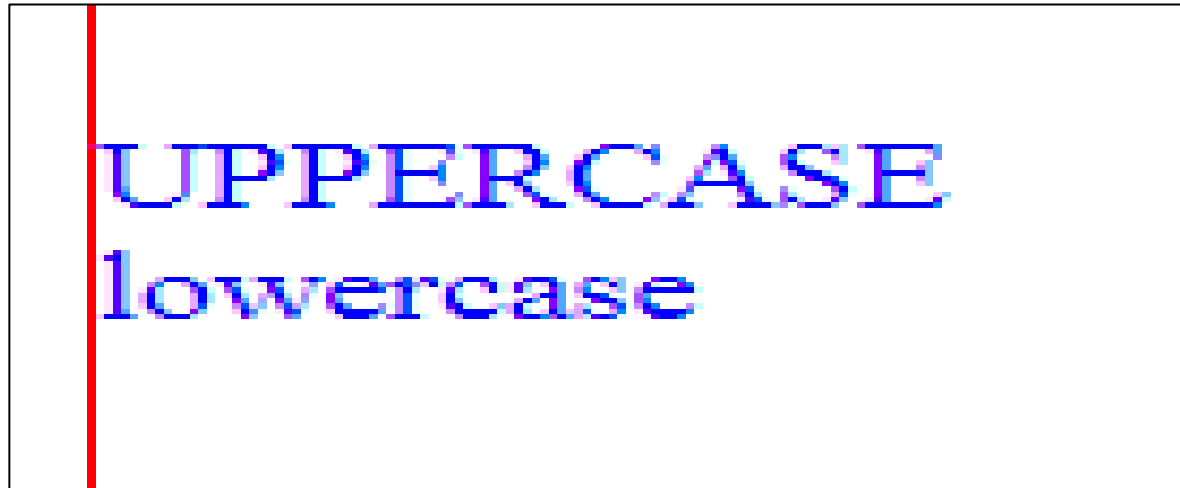
`{{currentCustomer.CustomerName | uppercase}}`

- Lowercase pipe

`{{currentCustomer.CustomerName | lowercase}}`

# Uppercase / lowercase Pipe

- {{"uppercase" | **uppercase**}} <br/>
- {{"LOWERCASE" | **lowercase**}} <br/>



UPPERCASE  
lowercase

# Date Pipe Example

```
printDate = new Date(2017, 4, 15, 15, 18, 25);
```

## Date Pipe Example

**Date format:** My joining Date is May 15, 2017.

**Specific Date Format:** My joining Date is 15/05/2017

**Full Date Format:** My joining Date is Monday, May 15, 2017

**Short Time format:** My joining Date is 3:18 PM

**Medium Date format:** My joining Date is May 15, 2017

<b>Date format:</b> My joining Date is {{printDate | **date**}}.<br/>

<b>Specific Date Format:</b> My joining Date is {{printDate |  
**date:"dd/MM/yyyy"**}}<br/>

<b>Full Date Format:</b> My joining Date is {{printDate |  
**date:"fullDate"**}}<br/>

<b>Short Time format:</b> My joining Date is {{printDate |  
**date:"shortTime"**}}<br/>

<b>Medium Date format:</b> My joining Date is {{printDate |  
**date:"mediumDate"**}}<br/>

# Decimals/Percentages/Number Pipe Example

- Grade : { { 10 | **percent** } } <br/>
- decimal : { { 200 | **number:'1.2-2'** } } <br/>
- currency : { { 500 | **currency : 'USD' : true** } } <br/>
- currency : { { 500 | **currency : 'EUR' : true** } } <br/>
- currency : { { 500 | **currency : 'INR' : true** } } <br/>

## Decimals/Percentages/Number Pipe Example

Grade : 1,000%  
decimal : 2,000.35  
currency : \$500.00  
currency : €500.00  
currency : ₹500.00

# Slice Pipe Example

- `<b>Without Slice:</b> { {"This is slice pipe"}}<br/>`
- `<b>With Slice:</b>{ {"This is slice pipe" | slice:0:8}}... <br/>`

**Slice Pipe Example**

**Without Slice: This is slice pipe**

**With Slice:This is ...**

# Json Pipe Example

- jsonArray = [{"id":"101", "name":"user1"}, {"id":"102", "name":"user2"}];
- { {jsonArray | **json** } }

**JSON Pipe Example**

**[ { "id": "101", "name": "user1" }, { "id": "102", "name": "user2" } ]**



# Custom Pipe / User Defined Pipe

---

- Helps to create custom format
- Write custom pipe which reverse the string
- In a project add folder as “Pipes”
- Add TypeScript file and name it as “ReverseStringPipe.ts”

```
import {Pipe, PipeTransform} from "@angular/core";
@Pipe({name : 'reversestring'})
export class ReverseStringPipe {
 transform(value : string) : string {
 return value.split("").reverse().join("");
 }
}
```

# Load Custom Pipe in Module

- Go to MainModule.ts file and load pipe

...

```
import {ResverseStringPipe} from "../Pipes/ReverseStringPipe";
```

```
@NgModule({
```

```
 imports: [BrowserModule, FormsModule],
```

```
 declarations: [CustomerComponent, GridComponent,
```

```
ResverseStringPipe],
```

```
 bootstrap: [CustomerComponent]
```

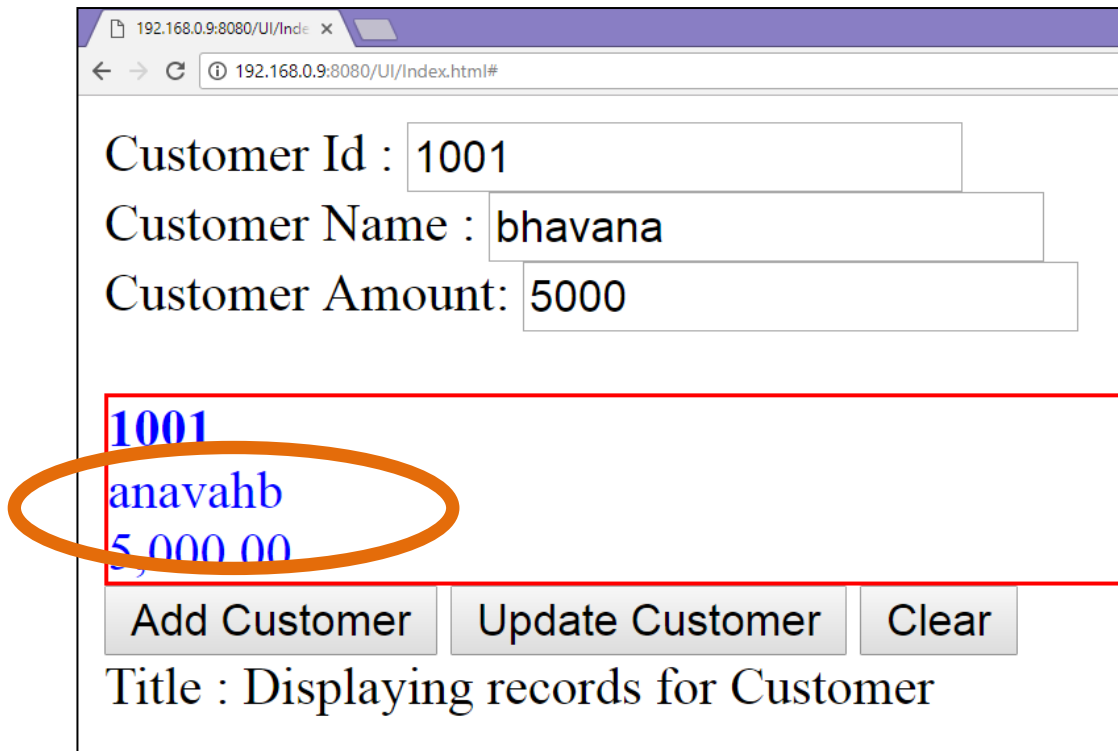
```
})
```

```
export class MainModule {
```

```
}
```

# Custom Pipe Usage in HTML

- `{{currentCustomer.CustomerName | reversestring}}`
- Output



The screenshot shows a web browser window with the address bar displaying `192.168.0.9:8080/UI/Index.html#`. The page contains a form with three input fields: "Customer Id : 1001", "Customer Name : bhavana", and "Customer Amount: 5000". Below the form is a table with a red border. The table has three rows of data: "1001", "anavahb", and "5,000.00". The first row is circled in orange. At the bottom of the page, there are three buttons: "Add Customer", "Update Customer", and "Clear". Below the buttons, the text "Title : Displaying records for Customer" is displayed.

1001
anavahb
5,000.00

Add Customer Update Customer Clear

Title : Displaying records for Customer

# Custom Pipe with Parameter

---

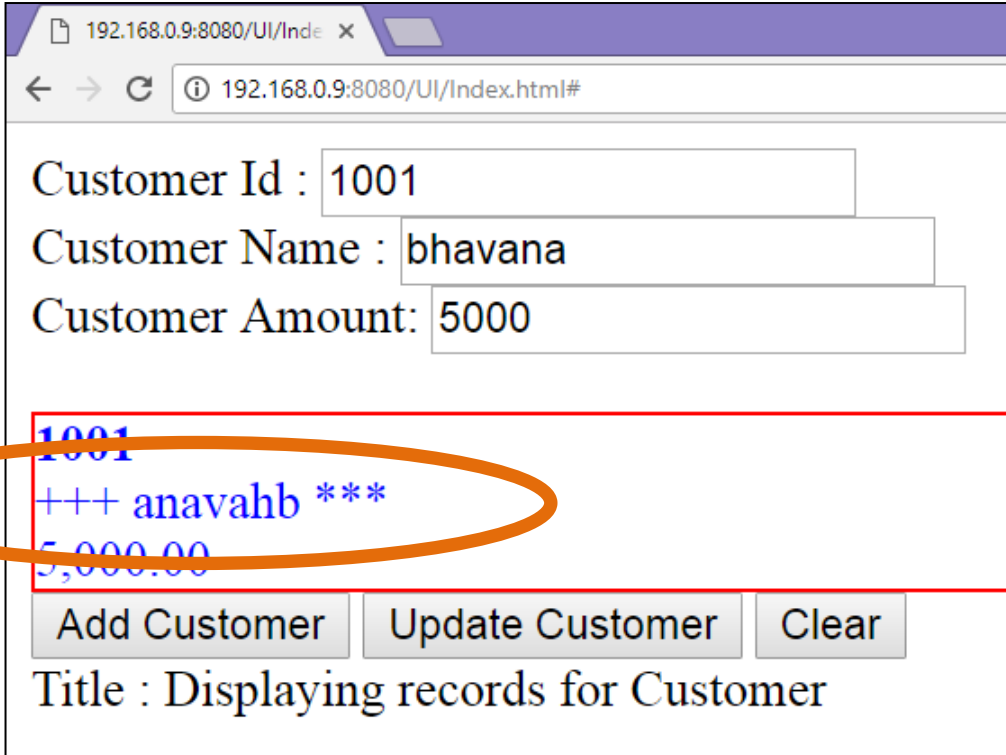
- In our example we want to add prefix and suffix some text
- transform() method can take parameter
- After the first parameter other parameters are input parameter for the pipes

```
@Pipe({name : 'reversestring'})
export class ReverseStringPipe {
 transform(value : string, start : string, end : string) : string {
 return start + " " + value.split("").reverse().join("") + " " + end;
 }
}
```

# Custom Pipe with Parameter Usage

```
{{currentCustomer.CustomerName | reversestring : "+++" : "***"}}}
```

- Output



The screenshot shows a web browser window with the address bar displaying "192.168.0.9:8080/UI/Index.html#". The page contains three input fields for customer information: "Customer Id : 1001", "Customer Name : bhavana", and "Customer Amount: 5000". Below these fields is a table with three rows of data. The first row is highlighted with a red border and an orange oval. The table has three columns: "Id", "Name", and "Amount". The first row contains the values "1001", "+++ anavahb \*\*\*", and "5,000.00". Below the table are three buttons: "Add Customer", "Update Customer", and "Clear". At the bottom of the page, the text "Title : Displaying records for Customer" is displayed.

Id	Name	Amount
1001	+++ anavahb ***	5,000.00

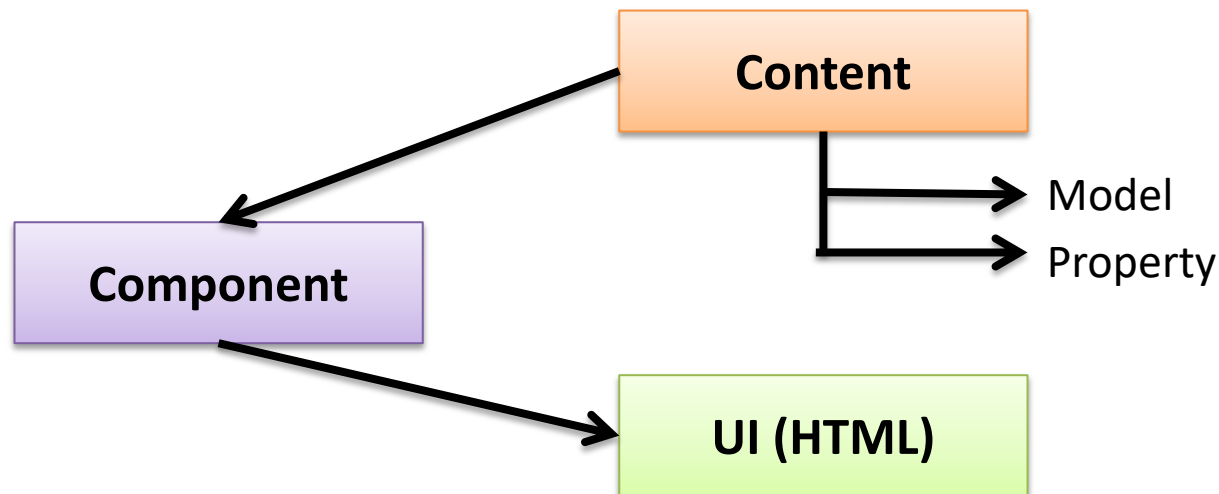
Add Customer Update Customer Clear

Title : Displaying records for Customer

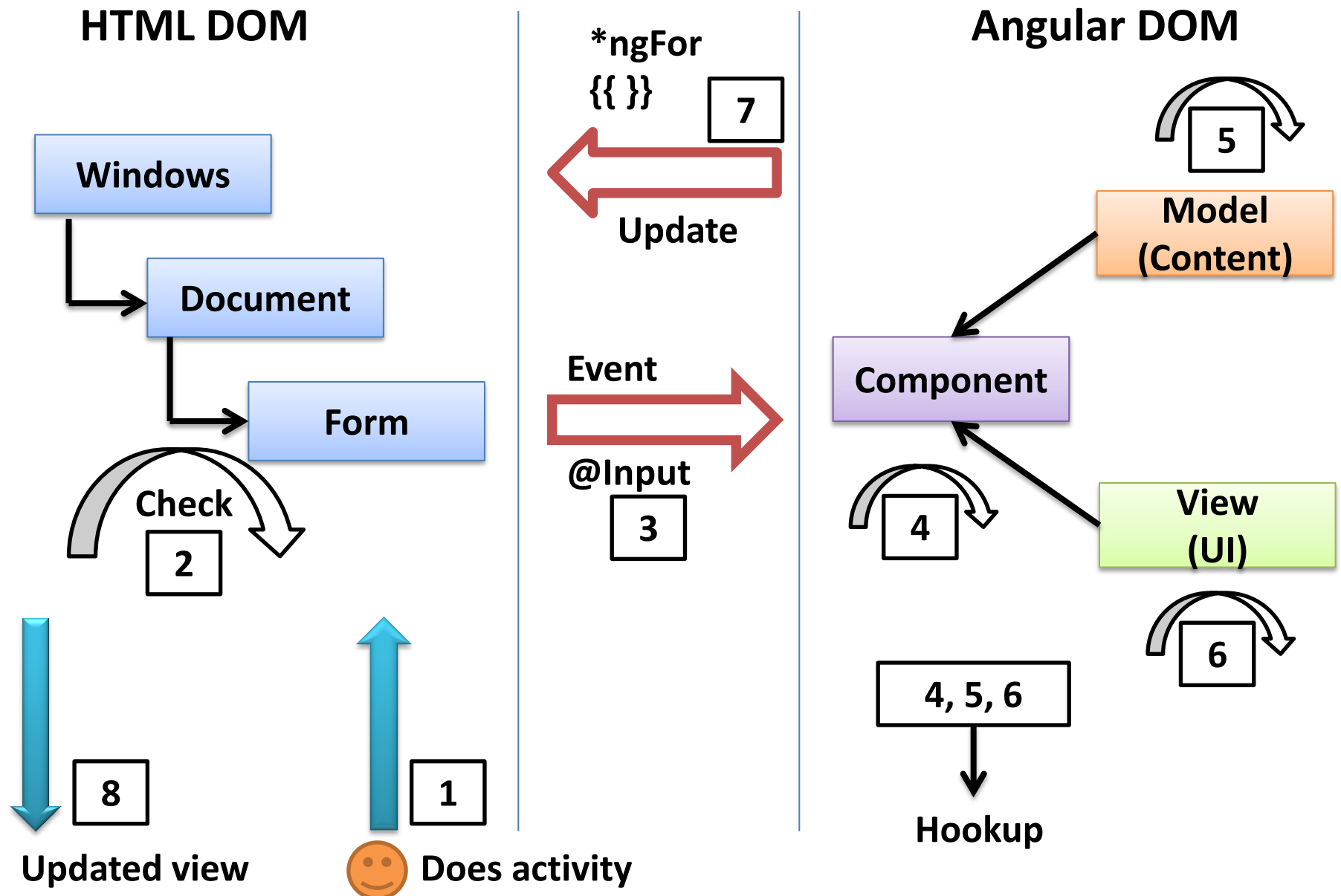
# **Angular Component Life Cycle**

# Angular Component Life Cycle

- Component is a most important part of Angular2
- It binds user interface and model
- Content means property / data
- E.g. `gridData`, `EntityName` they are the content for `GridComponent` or `currentCustomer` in `CustomerComponent`
- View is our HTML code



# Inside Browser



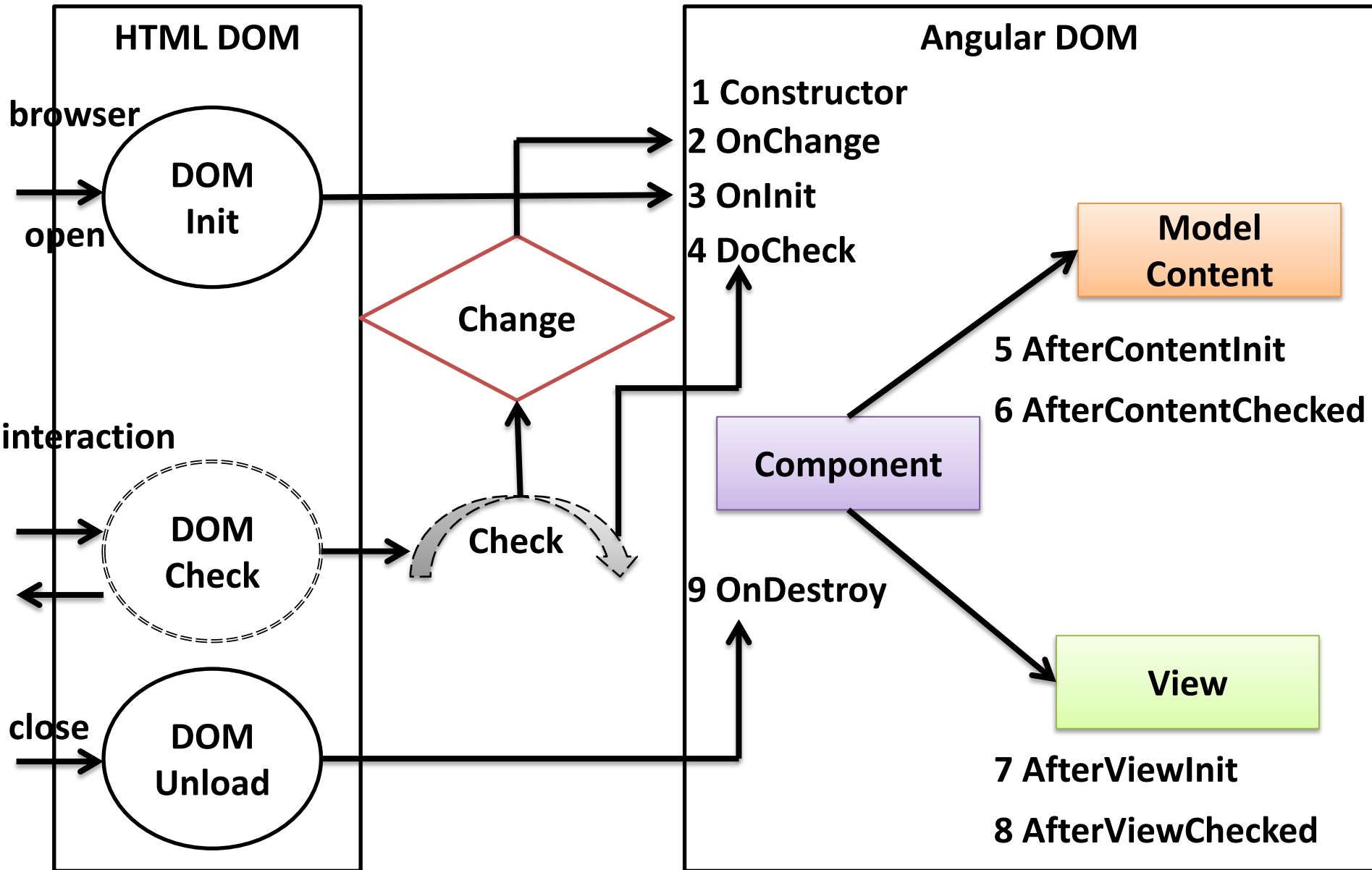


# Steps

---

1. User does some activity. E.g. Type value in text box
2. Check happens inside the HTML DOM e.g. lost focus event, text change event, click event
3. If the events are attached with `@Input()`, those events are then pass to an angular component
4. Some processing is done inside the component
5. Some processing is done inside the Model
6. Some processing is done inside the View
7. All the updates are send back to HTML DOM
8. Finally update HTML is send to end user

# Angular Events



# 9 – Hook Points

---

1. ***Constructor*** – first thing to fire in component class
2. ***ngOnChanges*** – fires when there is a change. E.g. old value & new value, for object reference change
3. ***ngOnInit*** – fires after ngOnChanges and after the component is initialized
4. ***ngDoCheck*** – fires for all activities on DOM
5. ***ngAfterContentInit*** – fires when angular projects content into the view. Called only once
6. ***ngAfterContentChecked*** – fires for every activity on the DOM after angular checks the contents send to the view
7. ***ngAfterViewInit*** – fires after angular initialize the view. Fire once
8. ***ngAfterViewChecked*** – fires after angular checks the views.  
Fired for all activity on DOM
9. ***ngOnDestroy*** – when DOM is unloaded or user navigate to other page.

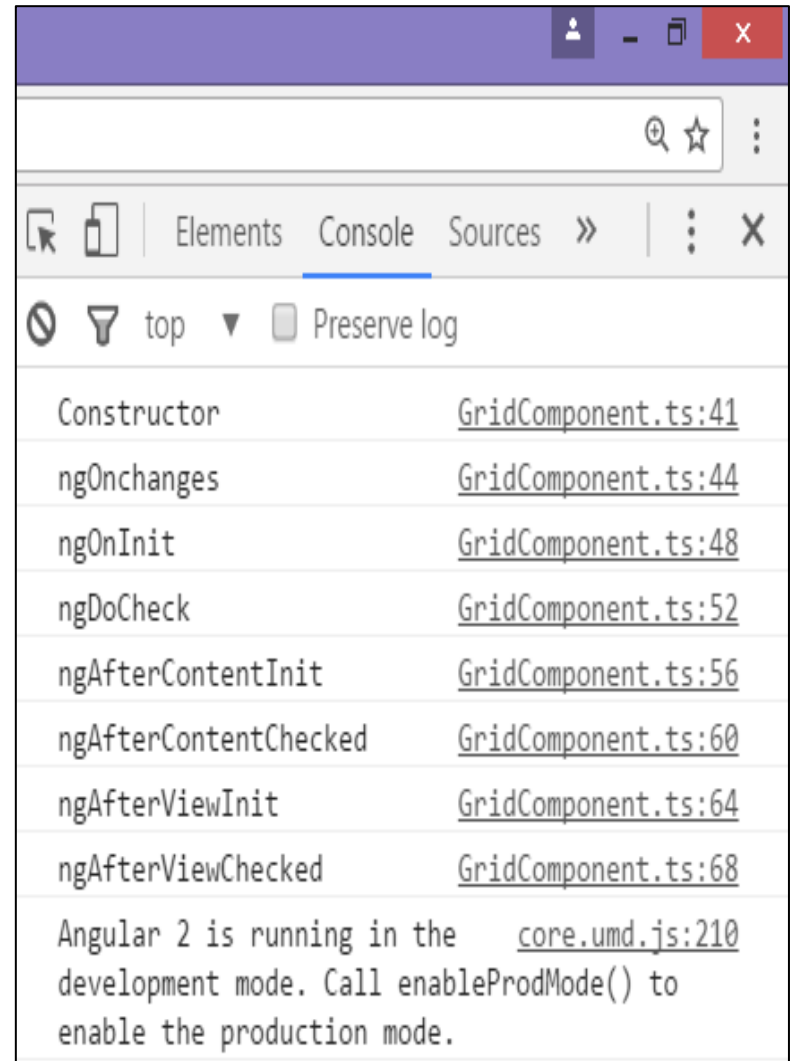
# Code

```
constructor() {
 console.log("Constructor");
}

ngOnChanges() {
 console.log("ngOnchanges");
}

ngOnInit() {
 console.log("ngOnInit");
}

. . .
```



# Life Cycle Hook Interface

---

- Optional to implement
- `import {Component, Input, Output, EventEmitter, OnInit, OnDestroy} from "@angular/core";`
- `export class GridComponent implements OnInit, OnDestroy { . . . }`

**Providers**

# Providers

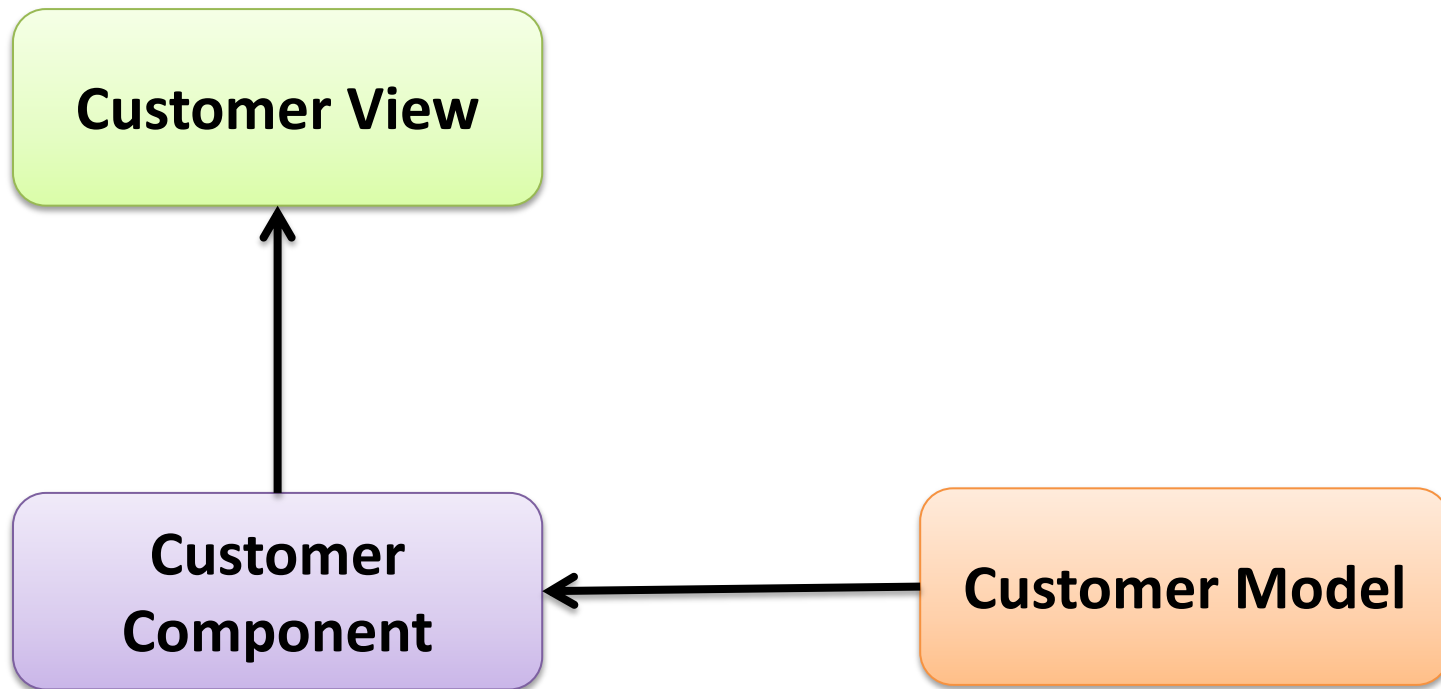
---

- Services
- Providers
- Dependency Injection

# Customer Component

---

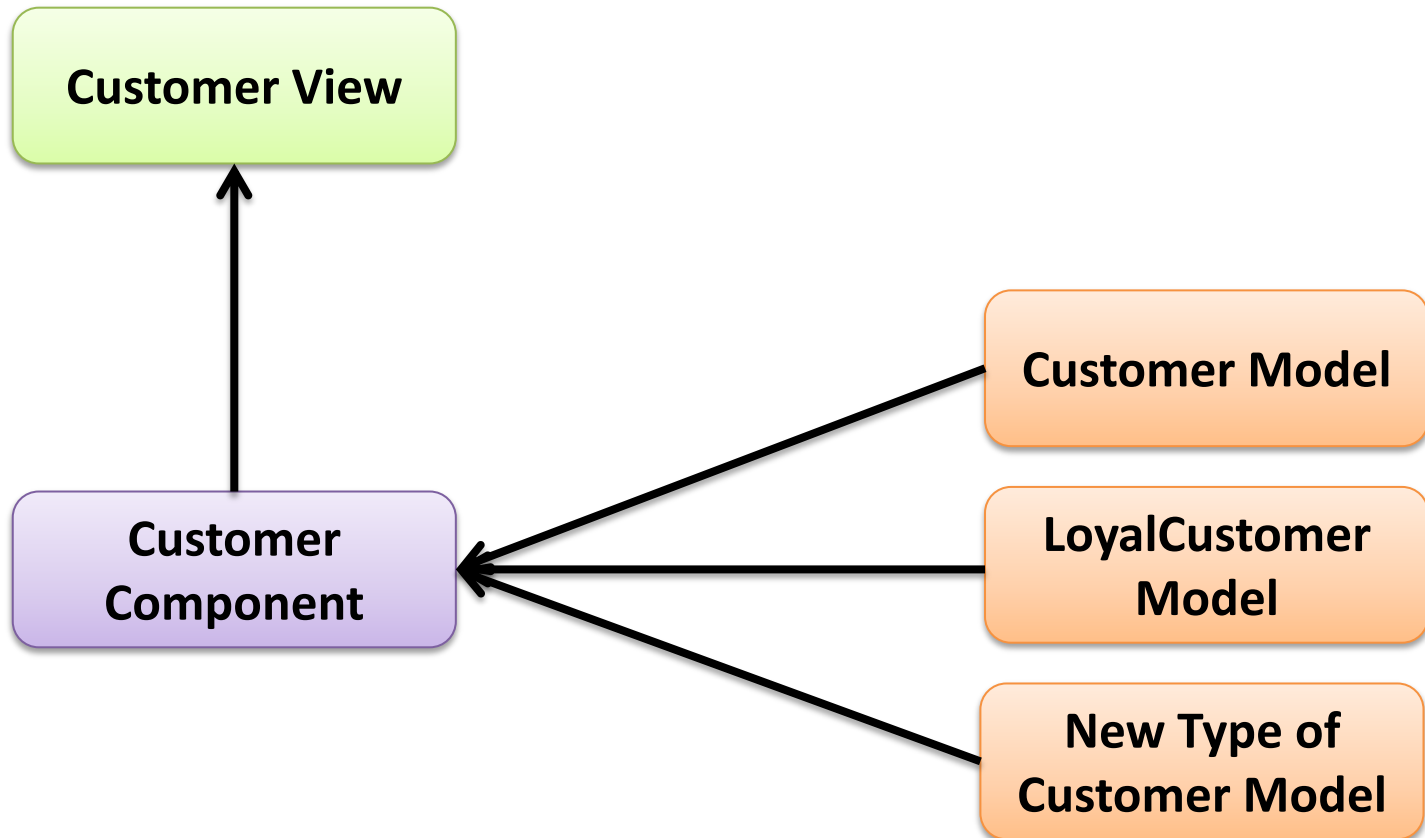
- Current Scenario





# New Requirement

- Different type of Customer
  - Customer
  - LoyalCustomer



# Validity of the Customer

---

- All three fields for Customer are compulsory

```
IsValid(): boolean {
 if (this.CustomerId.length == 0) {
 return false;
 }
 if (this.CustomerName.length == 0) {
 return false;
 }
 if (this.CustomerAmount <= 0) {
 return false;
 }
 return true;
}
```

# Loyal Customer

---

- Amount is not require

```
export class LoyalCustomer extends Customer {
 // Shadowing parent function
 IsValid(): boolean {
 if (this.CustomerId.length == 0) {
 return false;
 }
 if (this.CustomerName.length == 0) {
 return false;
 }
 return true;
 }
}
```

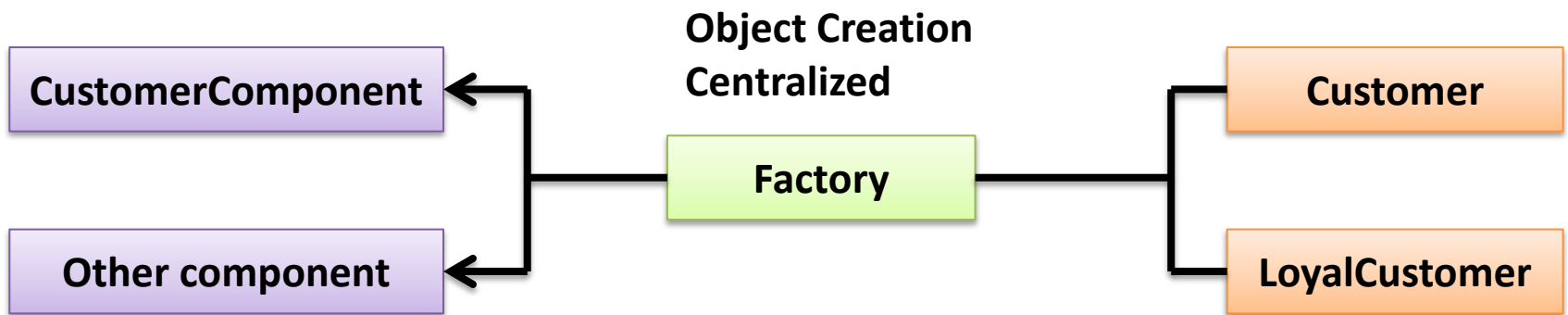
# Consume in CustomerComponent

```
currentCustomer: Customer = null;
constructor(OfTypeCustomer) {
 if (OfTypeCustomer == "Customer") {
 this.currentCustomer = new Customer();
 }
 else {
 this.currentCustomer = new LoyalCustomer();
 }
}
```

- Bad solution
- What if we have more type of Customers
- What if I am accessing this Customer Model from multiple Component like Customercomponent, InvoiceComponent.
- Every place I need to write if condition

# Centralize Object Creation

- Factory class (Factory Pattern)
- Which help us to create object
- We will do object creation in factory class
- Add new folder in your project as “*Factory*”
- Add TypeScript file in Factory folder as “*FactoryCustomer*”



# Factory Code

---

```
export class FactoryCustomer {
 public Create(TypeOfCustomer): Customer {
 if (TypeOfCustomer == "Customer") {
 return new Customer();
 }
 else {
 return new LoyalCustomer();
 }
 }
}
```

- We can also add new customer type in future

# Customer Object Creation

---

- Based on requirement create Customer object

```
import {FactoryCustomer} from "../../service/factorycustomer";

currentCustomer: Customer = null;
factoryCustomer: FactoryCustomer = new FactoryCustomer();

constructor() {
 //depends on the requirement
 this.currentCustomer =
this.factoryCustomer.Create("Customer");
}
```

# Inject Dependency

```
factoryCustomer: FactoryCustomer = new FactoryCustomer();
```



Dependency

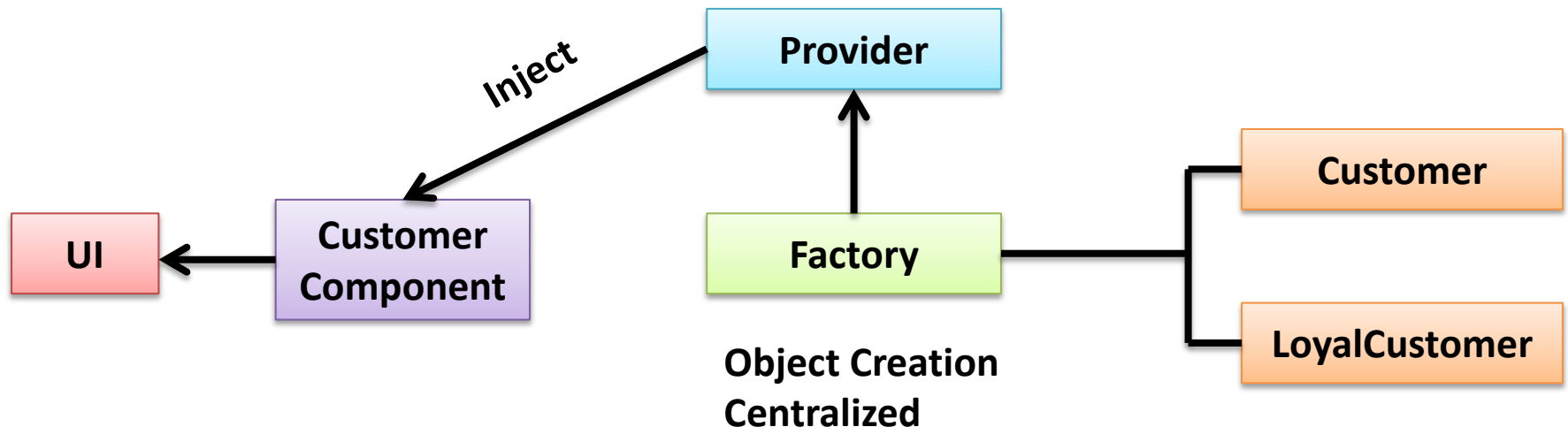
- Now by creating object of FactoryCustomer inside our CustomerComponent, CustomerComponent is creating dependency
- Better coding says that provide / inject this dependency from outside
- So can inject dependency in constructor

```
constructor(_factoryCustomer: FactoryCustomer) {
 this.factoryCustomer = _factoryCustomer;
 //depends on the requirement
 this.currentCustomer =
this.factoryCustomer.Create("Customer");
}
```



# Providers

- Who will inject this dependency
- Providers will help us to inject dependency in Component or Module in angular
- So FactoryCustomer should be *injectable*



# Code : Injectable

---

- Decorate FactoryCustomer with @Injectable()

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

```
@Injectable()
```

```
export class FactoryCustomer {
 public Create(TypeOfCustomer): Customer {
 if (TypeOfCustomer == "Customer") {
 return new Customer();
 }
 else {
 return new LoyalCustomer();
 }
 }
}
```

# Code : Provider

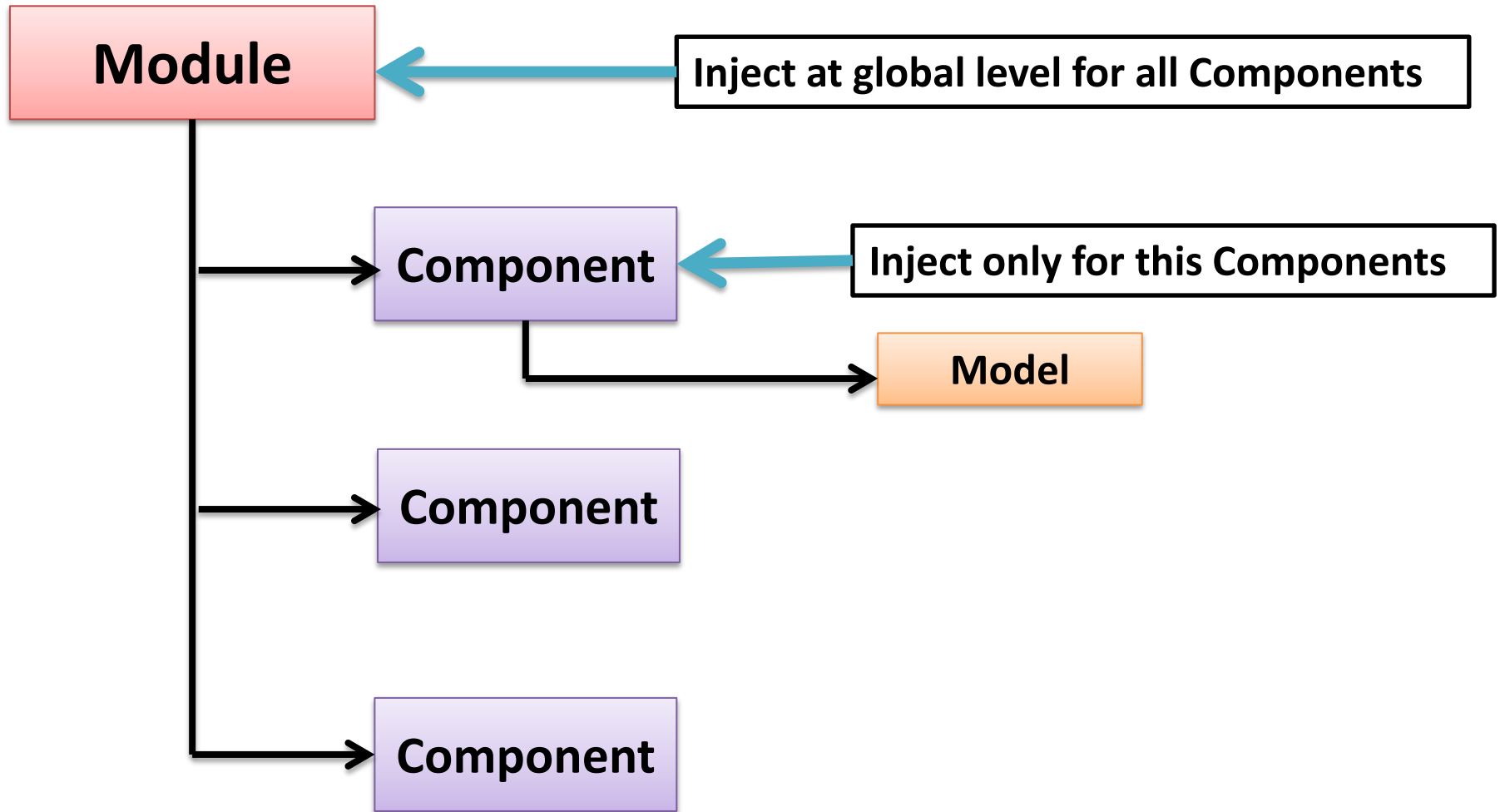
---

- Provide FactoryCustomer to CustomerComponent or Module

```
@Component({
 providers: [FactoryCustomer],
 selector: "customer-ui",
 templateUrl: "../..UI/Customer.html",
 styles: ['.redborder {border : 1px solid red }']
})

export class CustomerComponent { . . . }
```

# Inject Provider



Provider helps you to inject `@injectable()` component  
Component has not to worry how to create these objects

# Dynamic Type

---

- Provide type of customer dynamically through UI
- Give combo box to select type of customer

Customer Type :

```
<select>
 <option value="Customer" >Customer</option>
 <option value="LoyalCustomer">LoyalCustomer</option>
</select>
```

- In CustomerComponent add field

```
customerType: string = "Customer";
```

# Binding

---

- Object to UI – Property Binding

```
<select [ngModel]="customerType" >
```

- UI to Object – Event Binding – On selection of type

- CustomerComponent code

```
onCustomerTypeChange(_typeOfCustomer) {
 this.customerType = _typeOfCustomer
 this.currentCustomer =
 this.factoryCustomer.Create(this.customerType);
}
```

- UI Code

```
<select [ngModel]="customerType"
(ngModelChange)="onCustomerTypeChange($event)">
```

# Change Model Class

---

- Add CustomerType property in Model class also
- If we are adding customer data in database we want to persist the type of customer

```
export class Customer {
 CustomerType: string = "";
 CustomerId: string = "";
 . . .
}
```

# Change Select Method

---

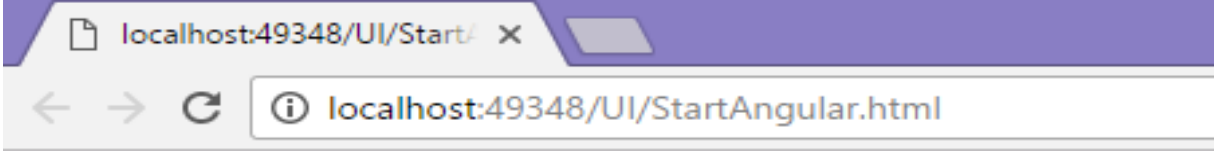
- Use factory to create the object instead of Object.assign()

```
Select(selectedCustomer: Customer) {
 this.currentCustomer =
this.factoryCustomer.Create(this.customerType);
 this.currentCustomer.CustomerId =
selectedCustomer.CustomerId;
 this.currentCustomer.CustomerName =
selectedCustomer.CustomerName;
 this.currentCustomer.CustomerAmount =
selectedCustomer.CustomerAmount;
 this.currentCustomer.CustomerType =
selectedCustomer.CustomerType;
}
```



# Test

- Build, Run & check



The screenshot shows a web browser window with a single tab titled 'localhost:49348/UI/Start'. The address bar displays 'localhost:49348/UI/StartAngular.html'. The page content includes a form with the following elements:

- Customer Tyle :** A dropdown menu with 'Customer' selected.
- Customer Id :** An empty text input field.
- Customer Name :** An empty text input field.
- Customer Amount :** A text input field containing the value '0'.
- Buttons:** Three buttons labeled 'Add Customer', 'Update Customer', and 'Cancel' are positioned horizontally below the input fields.
- Title :** A label 'Displaying records for Customer' is located at the bottom of the form.

# Enable Validation

---

- Add() button will be enabled or disabled based on the IsValid() method

```
<input type="button" value="Add Customer" (click)="Add()"
[disabled]="!(currentCustomer.IsValid())" />
```

# Display Type of Customer

- To display type of customer assign the value

```
constructor(_factoryCustomer: FactoryCustomer) {
 . . .
 this.currentCustomer =
this.factoryCustomer.Create(this.customerType);
 this.currentCustomer.CustomerType = this.customerType;
}
```

```
onCustomerTypeChange(_typeOfCustomer) {
 this.customerType = _typeOfCustomer
 this.currentCustomer =
this.factoryCustomer.Create(this.customerType);
 this.currentCustomer.CustomerType = this.customerType;
}
```

# Summery

---

- With help of dependency injection and provider we are able to create flexible code
- In future we can add new type of customer without changing in CustmerComponent or UI code

## **Angular Form for Validations**

# Validation

---

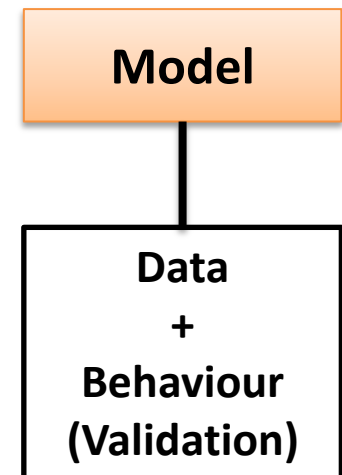
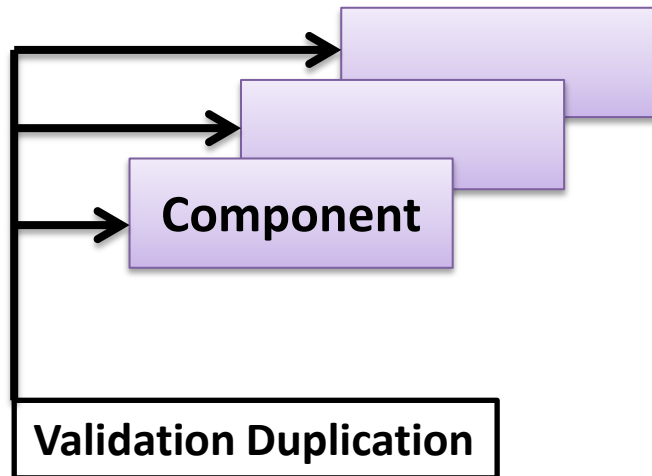
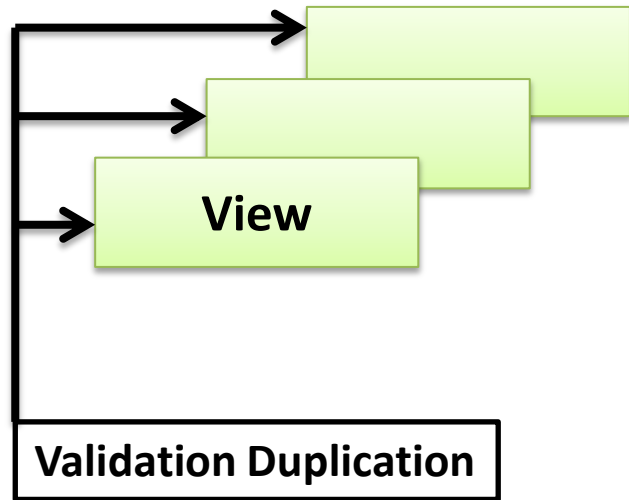
- An Angular form coordinates a set of data-bound user controls, tracks changes, validates input, and presents errors

# Angular 2 Form Validator

---

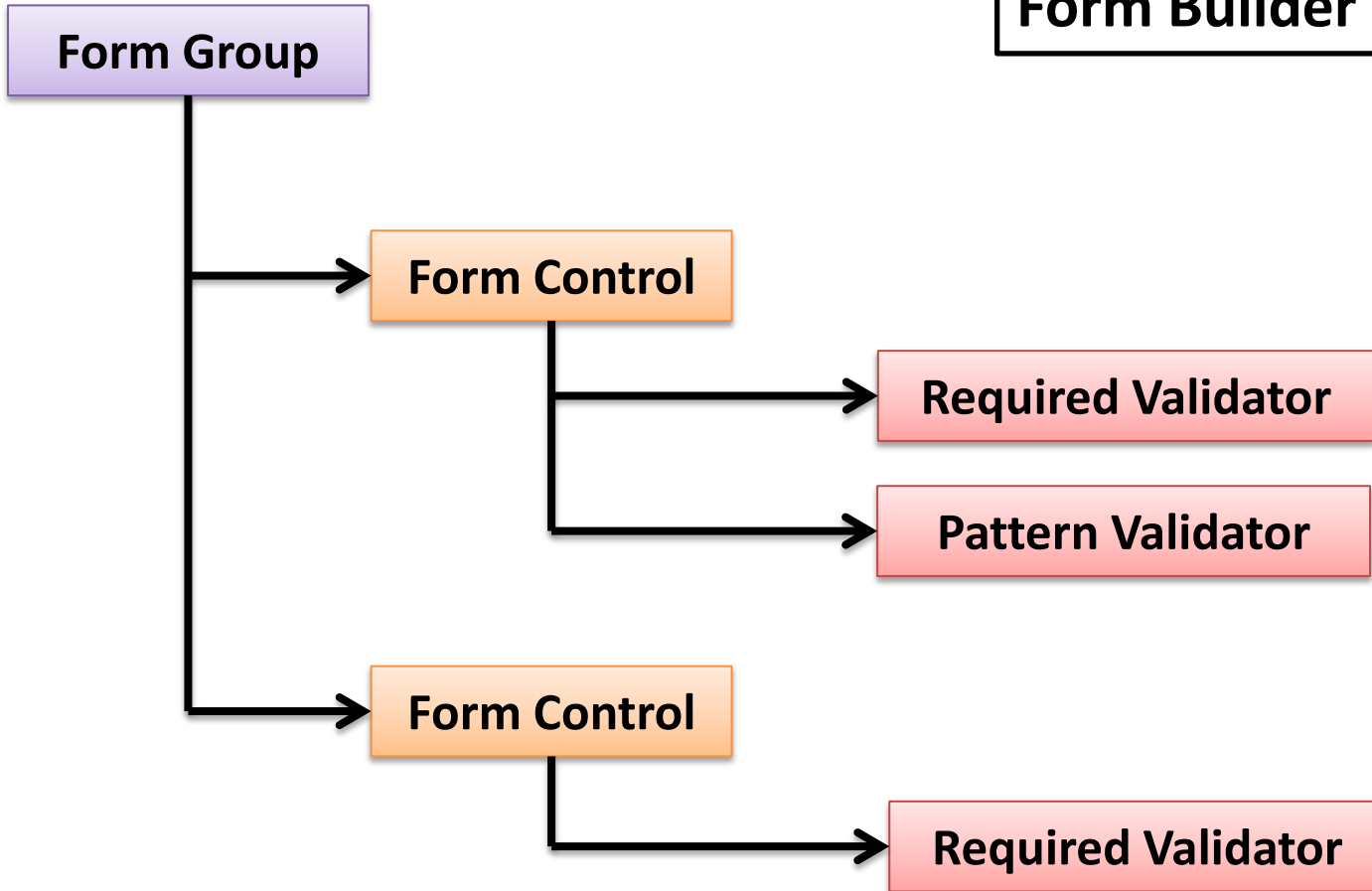
- CusromerCode is required
- CustomerName is required
- CustomerAmount is required
- CustomerCode format should be C001

# Where We Will Put Validation?





## Form Builder



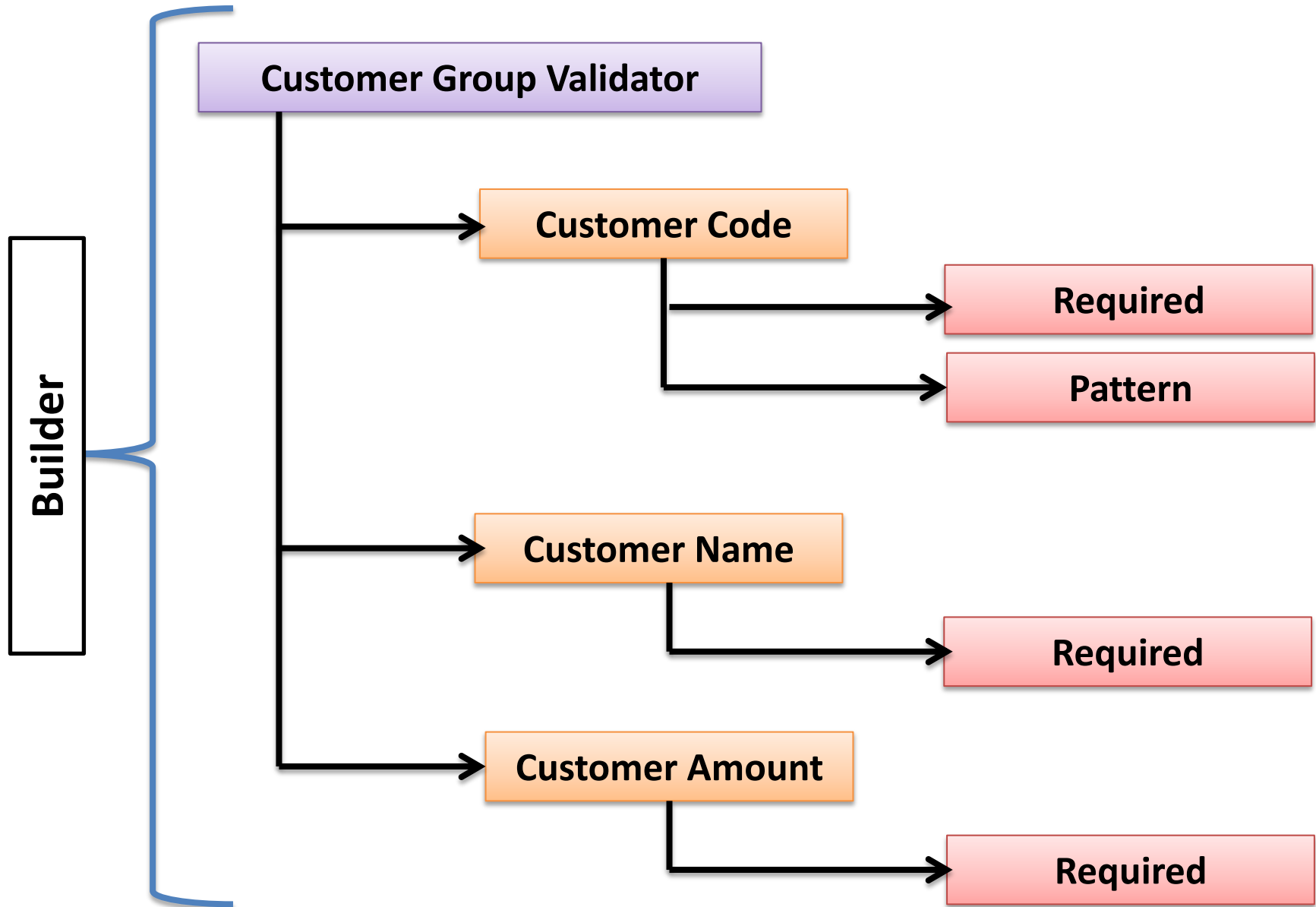
- Import necessary code

```
import {NgForm, FormGroup, FormControl, Validator,
FormBuilder} from "@angular/forms";
```

- FormControl – input control
  - E.g. TextBox is a FormControl on which we put validator.
  - We define TextBox as our FormControl
- Validators
  - Required validator
  - Max length validator, etc
  - One control can have more than one validator
  - Validators are attached with FormControl

- 
- FormGroup is a collection of control
    - E.g one form group consist of controls for Customer – Code, Name, Amount
    - Another Group for Address like – Area, City, Country
  - We have a FormControl and on that control we define the validator
  - All this controls belongs to a FormGroup
  - FormBuilder - ties all this together
    - Helps you to create a control, create a validator and attach to a FormGroup
    - Helps in building the hierarchy

# Customer Validator



# Validation Code

---

```
CustomerGroupValidator: FormGroup = null;
```

```
constructor() {
 var _builder = new FormBuilder();
 this.CustomerGroupValidator = _builder.group(
 {
 'CustomerId': ['', Validators.compose(
[Validators.required,
 Validators.pattern("^[C]{1,1}[0-9]{4,4}$")])],
 'CustomerName': ['', Validators.required],
 'CustomerAmount': ['', Validators.required]
 }
);
}
```

---

```
IsValid(): boolean {
 return this.CustomerGroupValidator.valid;
}
```

- Load validators from MainModule
- Import all the angular2 form component

```
import {ReactiveFormsModule} from "@angular/forms";
```

```
@NgModule({
 imports: [RouterModule.forRoot(ApplicationRoutes),
 BrowserModule, FormsModule, ReactiveFormsModule],
 . . .
})
```

- 
- Validators require HTML <form> tag
  - Wrap your Customer.Html <div> tag inside <form> tag
  - Assign FormGroup to <form> tag

```
<form [formGroup]="currentCustomer.CustomerGroupValidator">
```



# Bind Control with Validator

---

- Bind respective validator to the TextBoxes

Customer Id : `<input type="text"`  
`formControlName="CustomerId"`  
`[(ngModel)]=currentCustomer.CustomerId" /> <br />`

Customer Name : `<input type="text"`  
`formControlName="CustomerName"`  
`[(ngModel)]=currentCustomer.CustomerName" /> <br />`

Customer Amount : `<input type="text"`  
`formControlName="CustomerAmount"`  
`[(ngModel)]=currentCustomer.CustomerAmount" /> <br />`

# Exclude Control from Validation

---

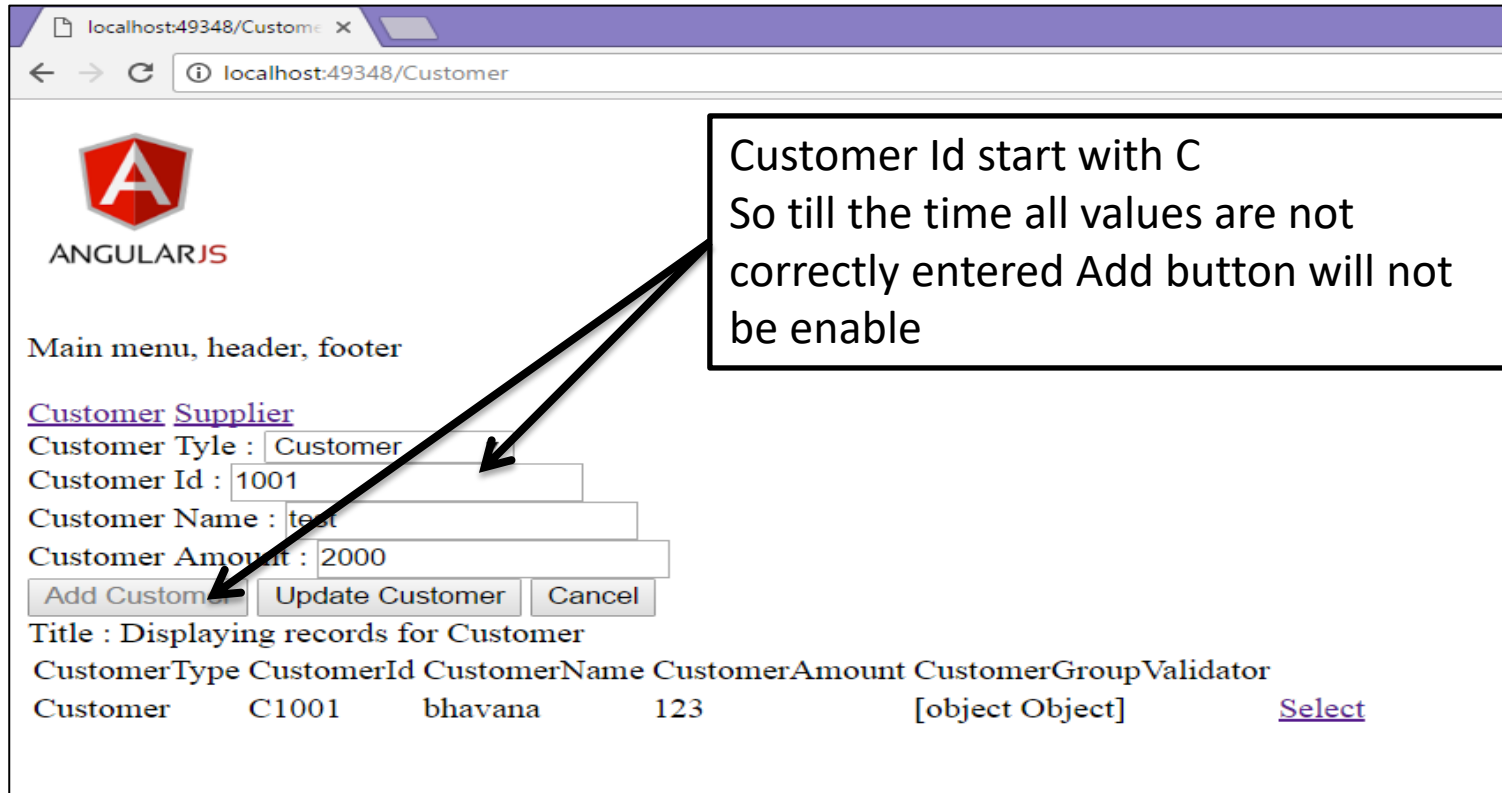
- CustomerType at present do not have validation
- By default when you create a formGroup on form element, it creates FormControlName for every input element
- Internally it has created FormControlName for CustomerType also and registered with formGroup
- We do not want CustomerType to be a part of customer validation

```
<select [ngModelOptions]="{standalone : true}" . . . >
```

- Exclude from validation

# Run & Test

- Add() button will be disabled until all validations are not fulfilled



The screenshot shows a web browser window at localhost:49348/Customer. The page features the AngularJS logo and a main menu. A form for adding a customer is displayed with the following fields: Customer Type (Customer), Customer Id (1001), Customer Name (test), and Customer Amount (2000). Below the form are three buttons: Add Customer, Update Customer, and Cancel. The Add Customer button is disabled. A text box on the right explains that the Add button is disabled because the Customer Id must start with 'C'. Below the form, a table displays the current record: Customer (Customer), CustomerId (C1001), CustomerName (bhavana), CustomerAmount (123), and CustomerGroupValidator ([object Object]). A 'Select' link is also present.

Customer Id start with C  
So till the time all values are not  
correctly entered Add button will not  
be enable

ANGULARJS

Main menu, header, footer

[Customer Supplier](#)

Customer Tyle : Customer

Customer Id : 1001

Customer Name : test

Customer Amount : 2000

Add Customer Update Customer Cancel

Title : Displaying records for Customer

CustomerType	CustomerId	CustomerName	CustomerAmount	CustomerGroupValidator
Customer	C1001	bhavana	123	[object Object]

[Select](#)

# Display Error Messages

```
//check validation for individual control and not in one group
IsValid(controlName, typeofValidator): boolean {
 if (controlName == undefined) {
 return this.CustomerGroupValidator.valid; //return true
 }
 else {
 return

(!this.CustomerGroupValidator.controls[controlName].hasError(typeofValidator));

 //Why negation
 //hasError will return true if element has error
 //so IsValid() should be false if hasError() return true
 //IsValid() should be true if hasError() return false
 }
}
```

# HTML Code

---

- Add error message next to each control

```
<span [ngClass]="{redtext : true}"
```

```
[hidden]="currentCustomer.IsValid('CustomerId', 'required')">C
```

```
ustomer Id is require
```

```
<span [ngClass]="{redtext : true}"
```

```
[hidden]="currentCustomer.IsValid('CustomerId', 'pattern')">Cu
```

```
stomer Id should start with C followed by 4 digits
```

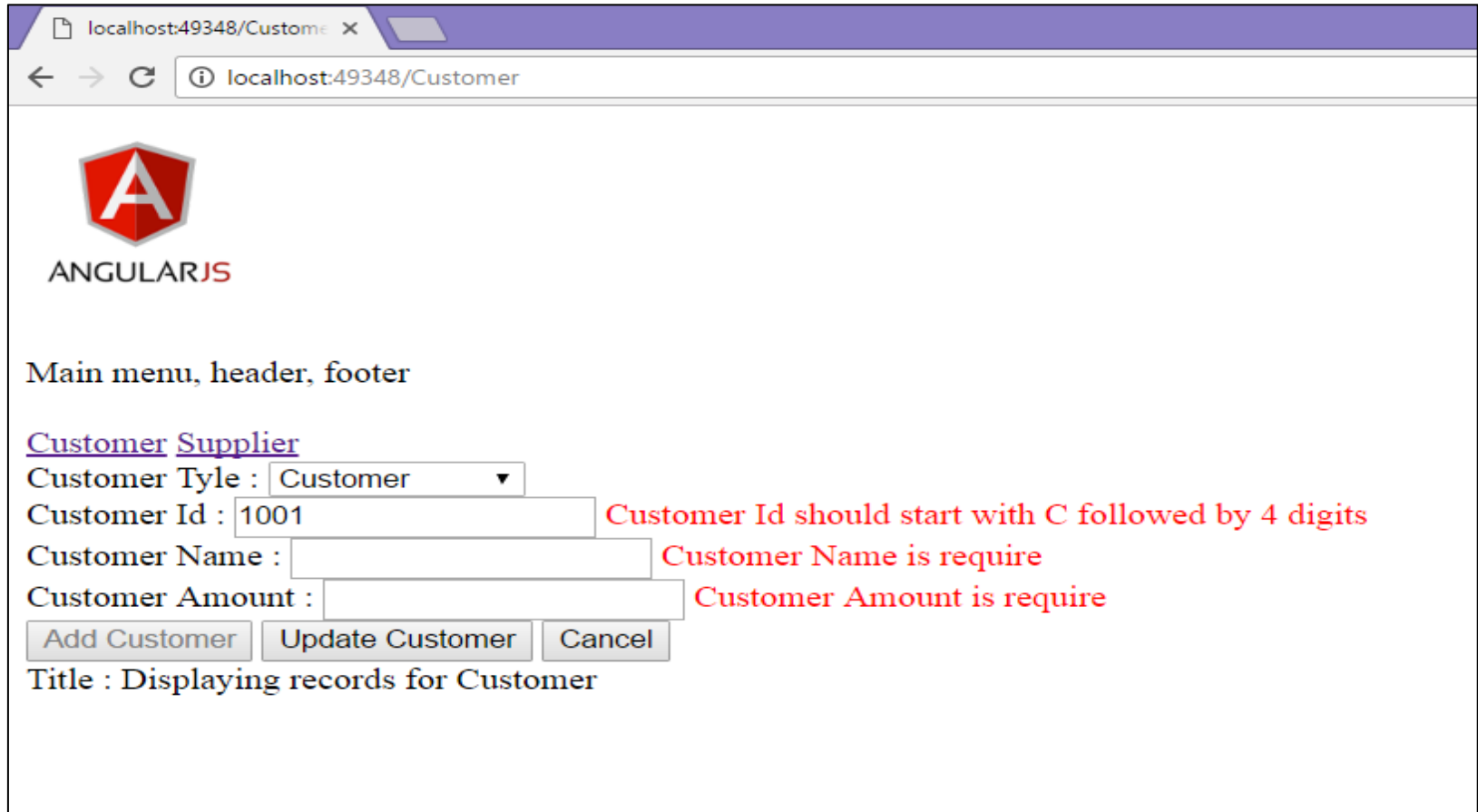
```
<span [ngClass]="{redtext : true}"
```

```
[hidden]="currentCustomer.IsValid('CustomerName', 'required')"
```

```
>Customer Name is require
```


# Execute Code

- Compile, Run & Test



localhost:49348/Customer x

localhost:49348/Customer

  
ANGULARJS

Main menu, header, footer

[Customer](#) [Supplier](#)

Customer Tyle : Customer ▼

Customer Id : 1001 Customer Id should start with C followed by 4 digits

Customer Name : Customer Name is require

Customer Amount : Customer Amount is require

Add Customer Update Customer Cancel

Title : Displaying records for Customer

# Error Message Issue

- Do not want to display error message in beginning
- Until the data is not changed in the textbox, i.e. data is not dirty we do not want to display error message

State	Property If true	Property If false
Control has been visited	Touched	Untouched
Control's value has changed	Dirty	Pristine
Control's value is valid	Valid	Invalid

# IsDirty()

---

- Customer.ts file

```
IsDirty(controlName, typeofValidator): boolean {
 return
 (this.CustomerGroupValidator.controls[controlName].dirty);
 //a control is dirty if user has changed the value in UI
}
```

- Customer.HTML

```
<span *ngIf="(currentCustomer.IsDirty('CustomerId') == true)"
[hidden]="currentCustomer.IsValid('CustomerId','required')">C
ustomer Id is require
```



# Execute

---

- Build, Run & Test

# Pristine and Touched property

---

- Initially when form loads, pristine is true
- The moment you type inside the textbox, form becomes dirty , so pristine is false
- Initially when form load , no control is touched by the user, so initial value of touched is false
- The moment you click inside the textbox and press tab, touched become true

pristine {{currentCustomer.CustomerGroupValidator.pristine}}

touch {{currentCustomer.CustomerGroupValidator.touched}}

**Http Call**

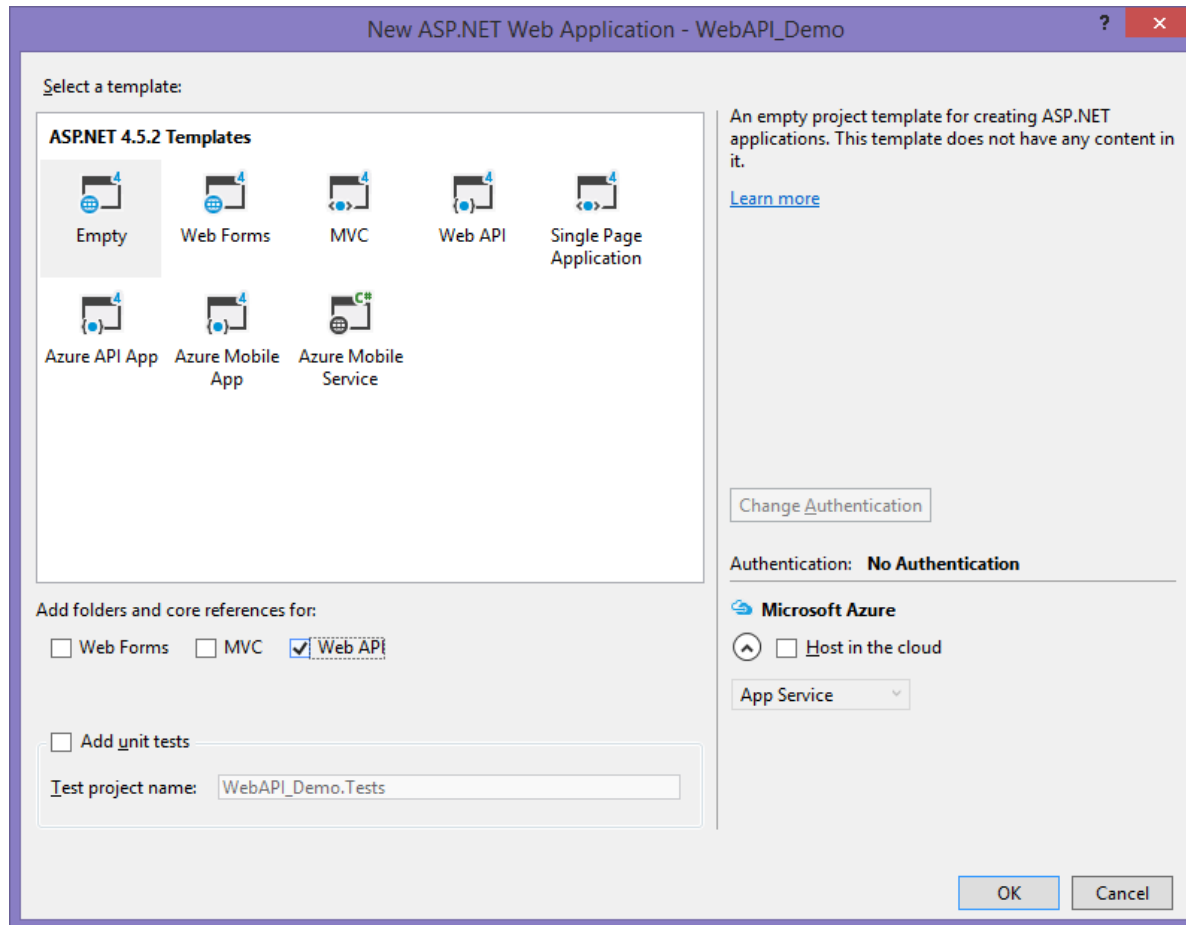
# Http Service

---

- Submit data to server by calling http service of angular2

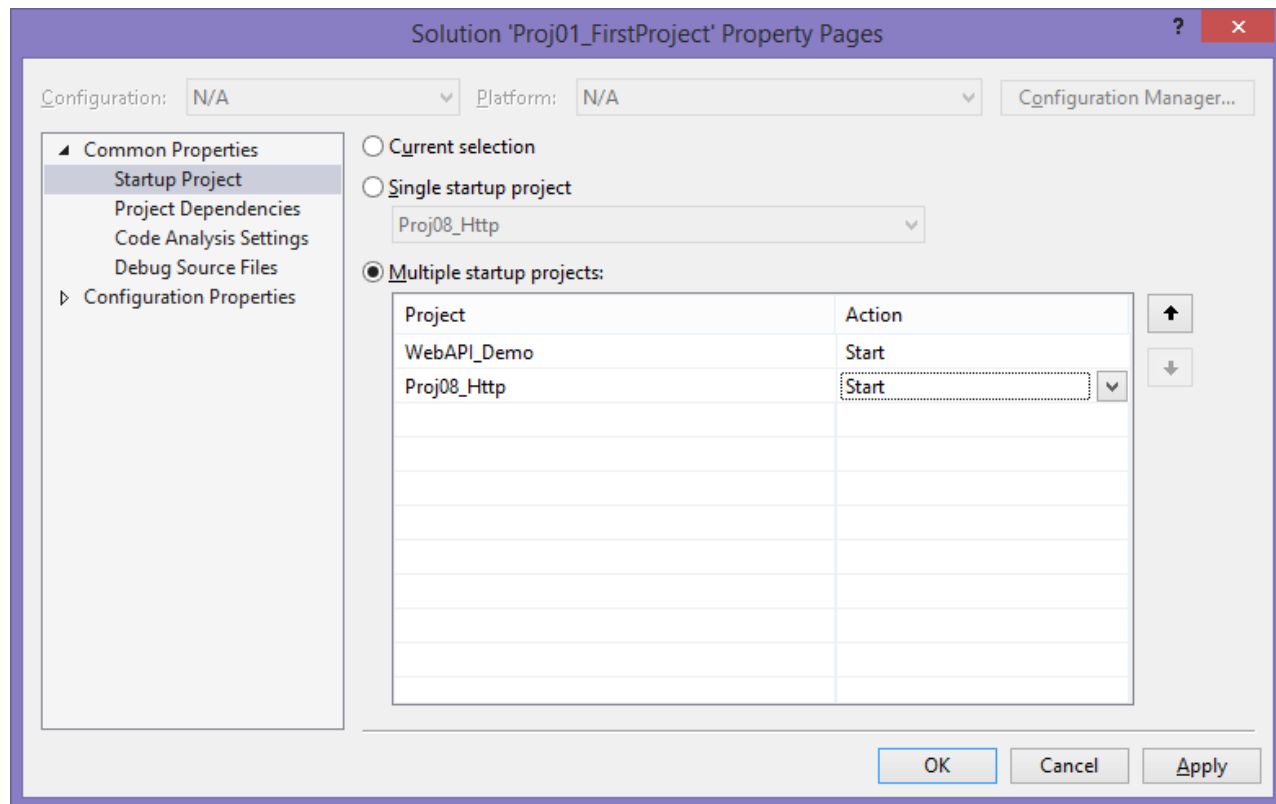
# Service - API

- In solution add another project as Web API



# Start up Project

- Both the program should run
- So mark multiple project start up



# API – Model class

---

- Add class in Models folder which represent Customer Data type .
- In communication we want to pass Customer Data

```
public class Customer
```

```
{
```

```
 public string CustomerType { get; set; }
```

```
 public string CustomerId { get; set; }
```

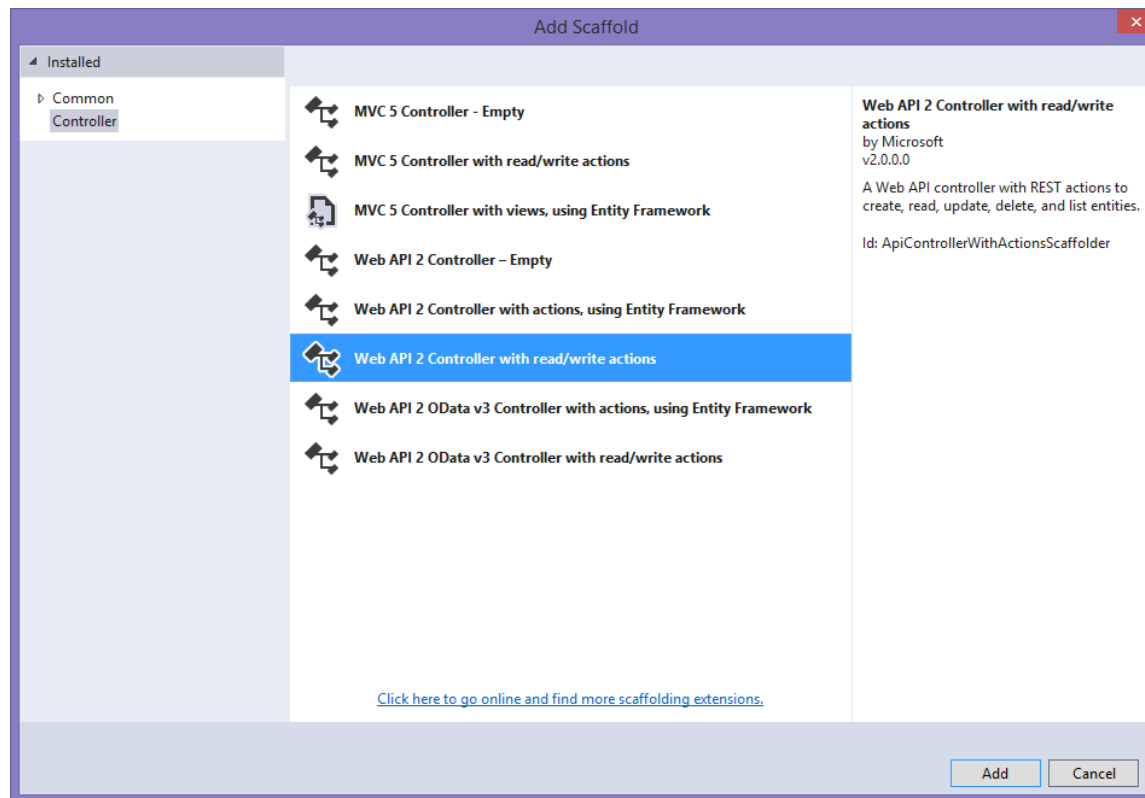
```
 public string CustomerName { get; set; }
```

```
 public double CustomerAmount { get; set; }
```

```
}
```

# API - Controller

- Add Web-API controller as CustomerController





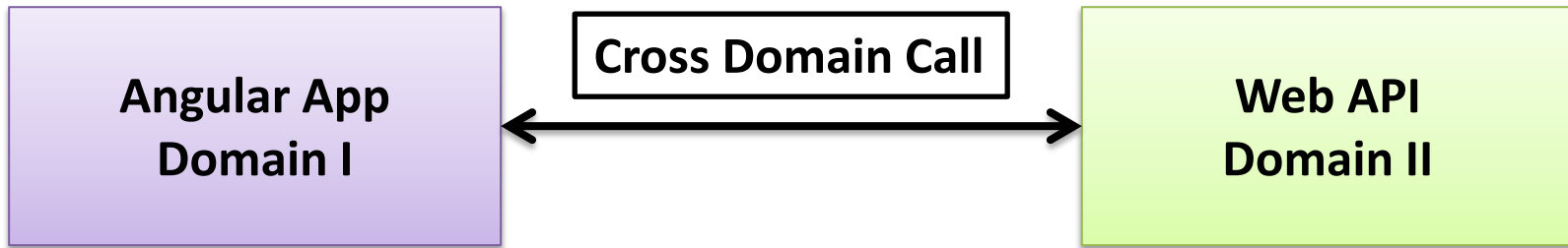
# POST - Method

---

# CORS

---

- Both projects are running on different domain



# Enable Cross Domain - I

---

- First enable in web.config

```
<system.webServer>
```

```
 <httpProtocol>
```

```
 <customHeaders>
```

```
 <add name="Access-Control-Allow-Origin" value="*" />
```

```
 <add name="Access-Control-Allow-Headers" value="*" />
```

```
 <add name="Access-Control-Allow-Methods" value="*" />
```

```
 </customHeaders>
```

```
 </httpProtocol>
```

```
</system.webServer>
```

# Enable Cross Domain - II

- In Global.ASAX file add

```
protected void Application_BeginRequest() {
 /*
 in every request for cross domain, client first request is OPTION
request
 this is to check with server, whether in cross domain server
support Get, Post, Put, Delete
 if answer is negative then client will not initiate the request
 */
 if (Request.Headers.AllKeys.Contains("Origin") &&
Request.HttpMethod == "OPTIONS") {
 //so if the request is Option, just do not do any thing
 Response.Flush();
 }
}
```

# Http angular Module

---

- To use http call in angular we have http Module in angular
- Add in Module

```
import {HttpModule, Jsonp} from "@angular/http";

imports: [RouterModule.forRoot(ApplicationRoutes),
 BrowserModule, FormsModule,
 ReactiveFormsModule, CommonModule, HttpModule]
```

# Http Component

---

- In CustomerComponent add Http Component

```
import {Http, Response, Headers, RequestOptions} from
"@angular/http";
import {Observable} from "rxjs/Rx";
```

Angular will inject Http component in constructor

```
http: Http = null;
constructor(_factoryCustomer: FactoryCustomer, _http: Http) {
 this.http = _http;
 . . .
}
```

# Submit Data to API

---

- `Submit() { . . . }`
- Four steps
  - Prepare data
  - Convert to JSON and prepare headers
  - Post the request
  - Request is asynchronous so handle call-back

# rxjs/RX

---

- rxjs/Rx helps to program asynchronous data stream
- You can give multiple asynchronous call, where rxjs will create data stream for each call
- rxjs has two terms
  - Observable
  - Observer
- Client is an observable which initiate a call to Observer



# Prepare Data - I

---

```
Submit() {
 //Post to http://localhost:49334/api/Customer
 var customerCollection = [];
 //take necessary data from customers array, as we do not
 want to pass validation form group which is the part of
 customers array
 for (let item of this.customers) {
 var custObj: any = {};
 custObj.CustomerType = item.CustomerType;
 custObj.CustomerId = item.CustomerId;
 custObj.CustomerName = item.CustomerName;
 custObj.CustomerAmount = item.CustomerAmount;
 customerCollection.push(custObj);
 }
}
```

# Convert to JSON and prepare headers - II

---

```
//convert data into JSON string
```

```
let data = JSON.stringify(customerCollection);
```

```
//prepare headers
```

```
let headers = new Headers({ 'Content-Type':
'application/x-www-form-urlencoded' });
let options = new RequestOptions({ headers: headers
});
```

# Post the request - III

---

```
var observable =
this.http.post("http://localhost:49334/api/Customer",
data, options); //this call is async call with the help
of rxjs
observable.subscribe(result =>
this.SuccessData(result), err => this.ErrorData(err));
}
```

# Call-back -IV

---

```
SuccessData(result) {
 this.customers = result.JSON();
}
```

```
ErrorData(err) {
 console.log(err);
}
```

# Server API Code

```
public HttpResponseMessage Post(FormDataCollection formData) {
 //apply enumerator on formData and Enumerate
 IEnumerator<KeyValuePair<string, string>> data =
formData.GetEnumerator();
 //move to next position
 data.MoveNext();
 //take the current value
 KeyValuePair<string, string> currentData = data.Current;
 //read data into string variable
 string str = currentData.Key;
 List<Customer> customerList =
Newtonsoft.Json.JsonConvert.DeserializeObject<List<Customer>>(str);

 foreach (var item in customerList) {
 item.CustomerName = item.CustomerName.ToUpper();
 }
 return Request.CreateErrorResponse(HttpStatusCode.OK,
customerList);
}
```

# Register Newtonsoft.JSON

- Use Newtonsoft.JSON to serialize and dehydrate object , which work with Auto implemented properties
- In Global.asax file

```
protected void Application_Start() {
 GlobalConfiguration.Configure(WebApiConfig.Register);
 var formatter =
GlobalConfiguration.Configuration.Formatters.JsonFormatter;
 formatter.SerializerSettings = new
Newtonsoft.Json.JsonSerializerSettings
 {
 Formatting = Newtonsoft.Json.Formatting.Indented,
 TypeNameHandling =
Newtonsoft.Json.TypeNameHandling.Objects
 };
}
```

# Execute

---

- Run & Test
- Add records
- Click on Submit button
- API will return data with CustomerName in capital

# Routing

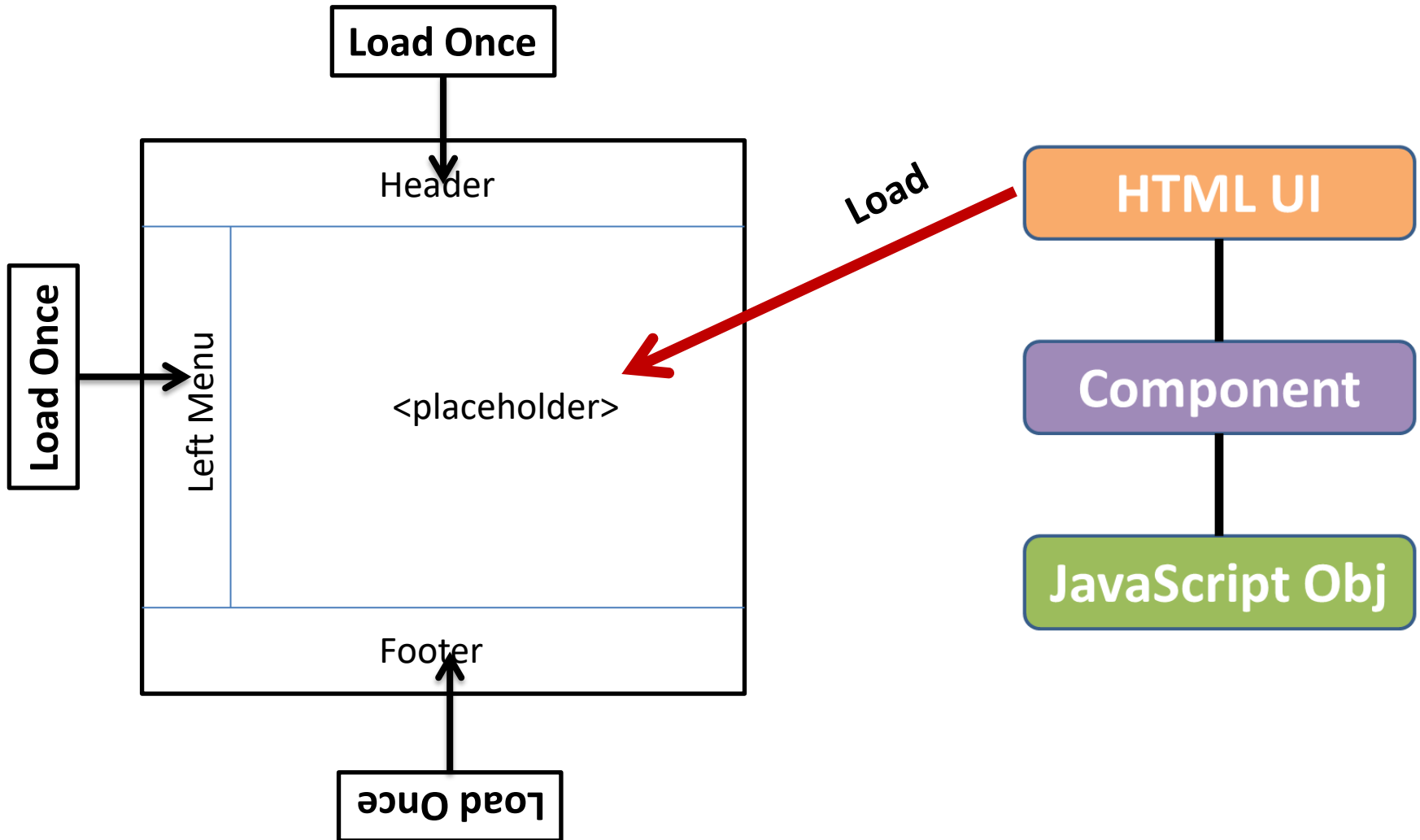


# Routing

---

- With the help of Routing create Single Page Application (SPA)
- Base template which will be loaded first (Header, Footer, Menu, etc.)
- And on demand load require pages / design

# SPA



# MasterPage.HTML

---

- In UI folder add HTML page as MasterPage

# Add Another Component

- Add Supplier Component and UI – supplier.html

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

@Component({
 selector: "supplier-ui",
 templateUrl: "../../UI/supplier.html"
})
export class SupplierComponent {
}
```

- Html code

```
<div>
```

```
 This is supplier page
```

```
</div>
```

# Add MasterPage Component

---

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

@Component({
 selector: "main-ui",
 templateUrl: "../UI/MasterPage.html"
})
export class MasterPageComponent {
}
```

# Add Welcome Component

---

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

```
@Component({
```

```
 template: "<h1>Welcome to Angular2
```

```
Application</h1>"
```

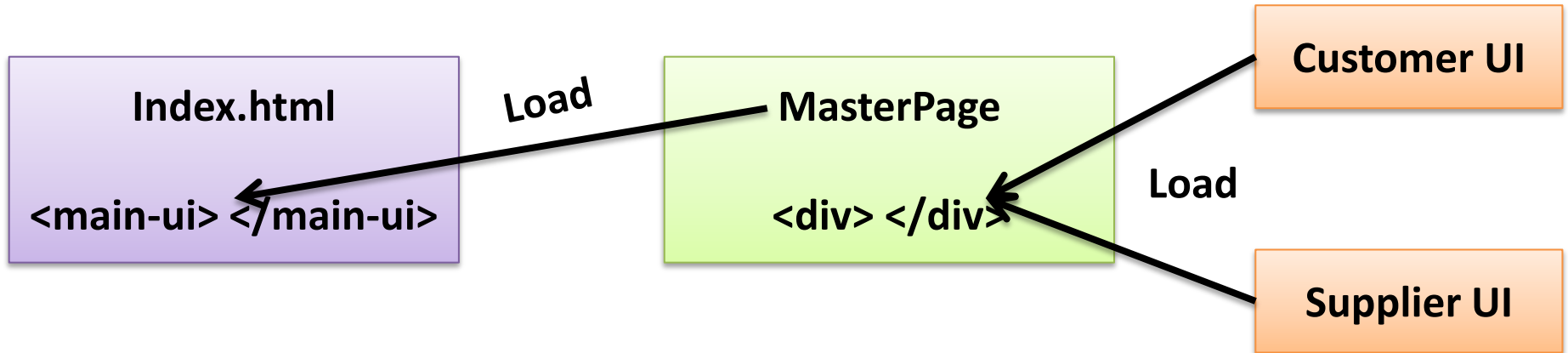
```
})
```

```
export class WelcomeComponent {
```

```
}
```

# Loading

- Now in StartAngular.html page MasterPage.html is load
- And in MasterPage div tag will load Customer.html and Supplier.Html based on user request



- 
- So now for Customer and Supplier we do not require selector
  - Because they are going to be loaded inside MasterPage div tag
  - Now when Startup.ts file is initiated, we will load MainModule
  - In Module folder rename CustomerModule file as MainModule
  - Import MasterPage Component
  - Also import Supplier Component
  - Bootstrap MasterPageComponent from MainModule



# MainModule Code

```
import {NgModule} from "@angular/core";
import {FormsModule} from "@angular/forms";
import {BrowserModule} from "@angular/platform-browser";
import {CustomerComponent} from "../Component/CustomerComponent";
import {GridComponent} from "../Component/GridComponent";
import {MasterPageComponent} from
"../Component/MasterPageComponent";
import {SupplierComponent} from "../Component/SupplierComponent";

@NgModule({
 imports: [BrowserModule, FormsModule],
 declarations: [CustomerComponent, GridComponent,
MasterPageComponent, SupplierComponent],
 bootstrap: [MasterPageComponent]
})
export class MainModule {
}
```

# Startup.ts

---

- Change bootStrap Module in Startup.ts file

```
import {platformBrowserDynamic} from "@angular/platform-browser-dynamic";
```

```
import {MainModule} from "../Binder/Module/MainModule";
```

```
const platform = platformBrowserDynamic();
```

```
platform.bootstrapModule(MainModule);
```

# RouterOutlet

---

- Customer.html and Supplier.html is not getting loaded in main place holder
- They need space to loaded dynamically
- In MasterPage define <router-outlet>
- RouterOutlet
  - Acts as a placeholder that Angular dynamically fills based on the current router state

```
<div>
 <router-outlet></router-outlet>
</div>
```

# Routing

---

- To work as SPA simply anchor tag will not work
- Because by clicking on anchor tag will load page independently
- Solution : define routing
- Routing is a collection where we can define on given segment of url which component we want to load
- Add new folder as – Routing
- Inside Routing folder add TypeScript file, name it as AppRouting.ts
- Routing file will define which part will load which html file

# Routing Code

---

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

```
import {CustomerComponent} from
"../binder/component/CustomerComponent";
```

```
import {SupplierComponent} from
"../binder/component/SupplierComponent";
```

```
export const ApplicationRoutes = [
 { path: '', component: WelcomeComponent },
 { path: 'UI/StartAngular.html', component:
WelcomeComponent },
 { path: 'Customer', component: CustomerComponent },
 { path: 'Supplier', component: SupplierComponent }
];
```

# routerLink

---

- In master page for anchor tag instead of href attribute use routerLink attribute

```
<a [routerLink]="['Customer']">Customer
```

```
<a [routerLink]="['Supplier']">Supplier
```

# RouterModule Component

---

- Responsible for executing routing in angular2
- RouterModule ensures that angular routing is in action
- It loads the routes
- And when we click the link it loads the routes in <router-outlet>
- In MainModule import RouterModule

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

# Add Routes in Module

---

- Specify the route collection to RouterModule
- We have one level of routes

```
import {ApplicationRoutes} from "../../routing/approuting";
```

```
@NgModule({
 imports: [BrowserModule, FormsModule,
 RouterModule.forRoot(ApplicationRoutes)],
 . . .
})
```



# Base URL

---

- When routing start define the base URL
- It's a base URL from where routing starts
- In StartAngular.html file define inside Body tag
- `<base href="" />`

# Run

---

- Run & Test
- If you get 403 forbidden error, this is authentication error in IIS Express
- Solution – in web.config file allow directory browsing

```
<system.webServer>
```

```
 . . .
```

```
 <directoryBrowse enabled="true"/>
```

```
</system.webServer>
```

**Location Strategy**  
**PathLocation & HashLocation**

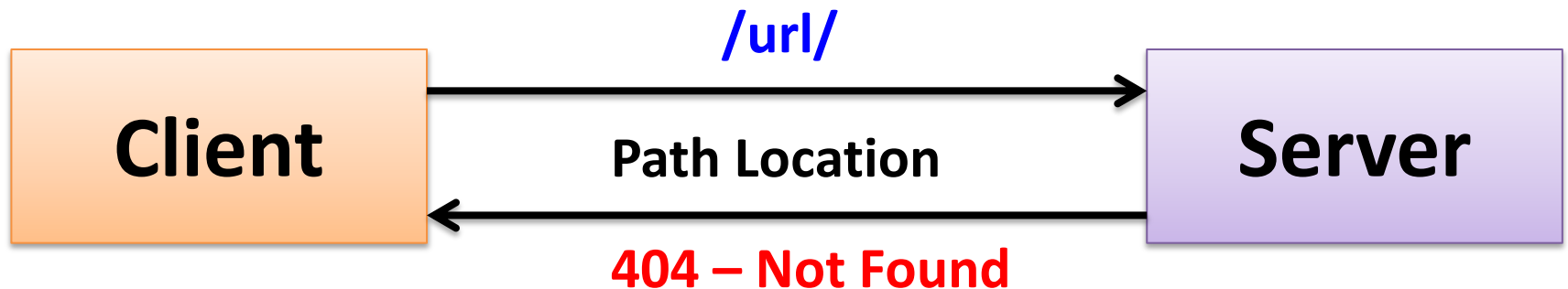
# Application Navigation in Browser

---

- Different ways to navigate in browser
  - Enter a URL in the address bar and the browser navigates to a corresponding page.
  - Click links on the page and the browser navigates to a new page.
  - Click the browser's back and forward buttons and the browser navigates backward and forward through the history of pages you've seen.
  - Click the refresh button to reload the page

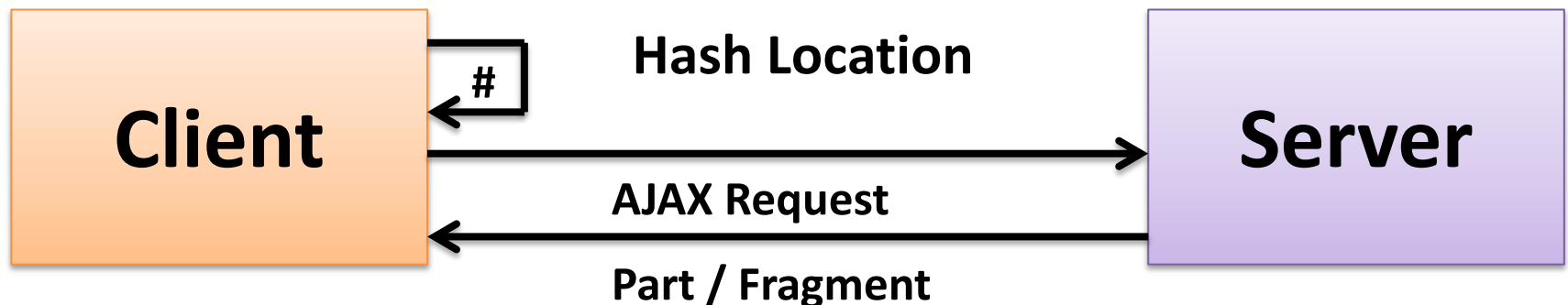
# Why 404 Error (Page Not Found)

- Now in Angular when you use routing and if user do one of the activity described in previous slide, user will get 404 error – Not Found
- Reason
- When we type the url in browser and press enter key, request goes to the server.
- On server no page is available by that name, so throwing 404 error



# Browser Strategy

- Browser URL follows two kinds of strategy
  - Path location strategy
    - Reload the whole page
  - Hash location strategy
    - url has #
    - # indicates that full page will not be loaded
    - Load only part (AJAX request) and not full page
- By default angular 2/4 use path location strategy



# First Step

---

- Import Location Strategy and inject the provider for Hash Location Strategy for all the component route to follow Hash location strategy
- By default route use path location strategy
- MainModule.ts file

```
import {LocationStrategy, HashLocationStrategy} from "@angular/common";

@NgModule({
 providers:[
 {provide : LocationStrategy, useClass : HashLocationStrategy}
],
 ...
})

export class MainModule {
}
```

# Execute

---

- Now when we execute check the url, it is using Hash Location Strategy
- <http://192.168.0.9:8080/UI/Index.html#/>
- <http://192.168.0.9:8080/UI/Index.html#/Customer>
- Even when you refresh button it works properly
- It first take Index.html page then browse to Customer Page
- So it does not hit the server but do AJAX call



# **Appendix**

# Another Component

---

- Add Message Component inside Customer Component
- User can add n numbers of Message Component
- The content for the Message Component we want to keep dynamic

# Example

---

- In Component folder add TypeScript file as MessageComponent.ts

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

@Component({
 selector: "message-ui",
 template: "<ng-content></ng-content>"
})
export class MessageComponent {

}
```

- Inline template instead of separate html file

# Module

---

- Add this component inside CustomerModule, since we want to use this component inside Customer Component

. . .

```
import {MessageComponent} from "../Component/MessageComponent";
```

```
@NgModule({
 imports: [BrowserModule, FormsModule],
 declarations: [CustomerComponent, MessageComponent],
 bootstrap: [CustomerComponent]
})
```

```
export class CustomerModule {

}
```

# Usage

`<div>`

Customer Id : . . .

Customer Name : . . .

Customer Age : . . .

`<message-ui>`

`<h2 style="color:red">First Message</h2>`

`</message-ui>`

`<message-ui>`

`<div style="border:1px solid blue">`

Second Message

`</div>`

`</message-ui>`

`</div>`

