# ANGULAR 2+ MVW WEB FRAMEWORK

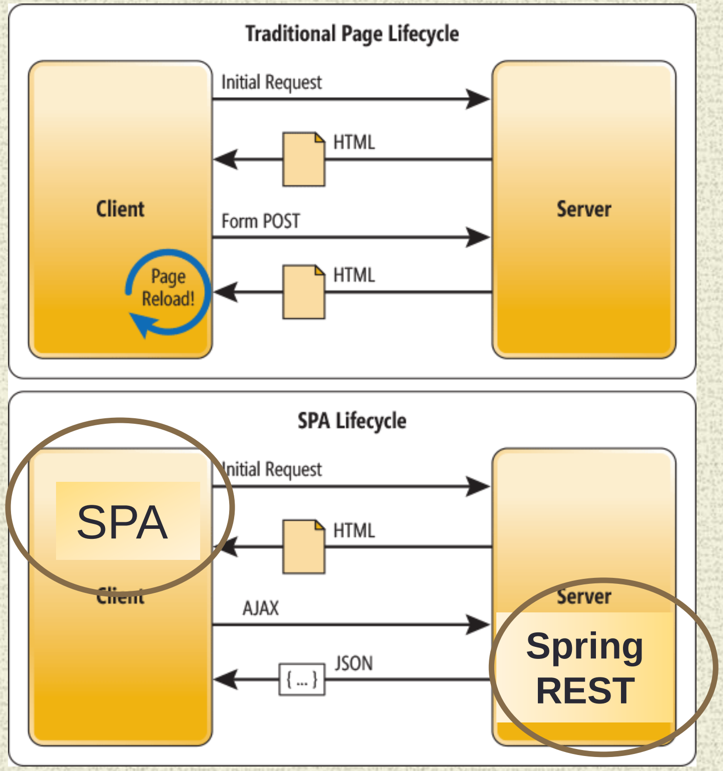
# Spring MVC Rest Controller & SPA

**SPA – Single Page Application**

Web application that fits on a single  to provide a more fluid user experience
All the "heavy lifting" presentation-wise is done on the client side, in JavaScript.
The client-side JavaScript handles all the  page routing.

Server interaction with a SPA involves
dynamic communication with the web
server behind the scenes – primarily data access.

Spring RESTful Controller is a perfect fit for
Server side support of SPA.



## Two Applications Instead of One

**PARADIGM SHIFT**

**Client-Side JavaScript Application**

We will use Angular 2+:

“Superheroic Web MVW Framework” [Google]

(where the “W” stands for “Whatever”)

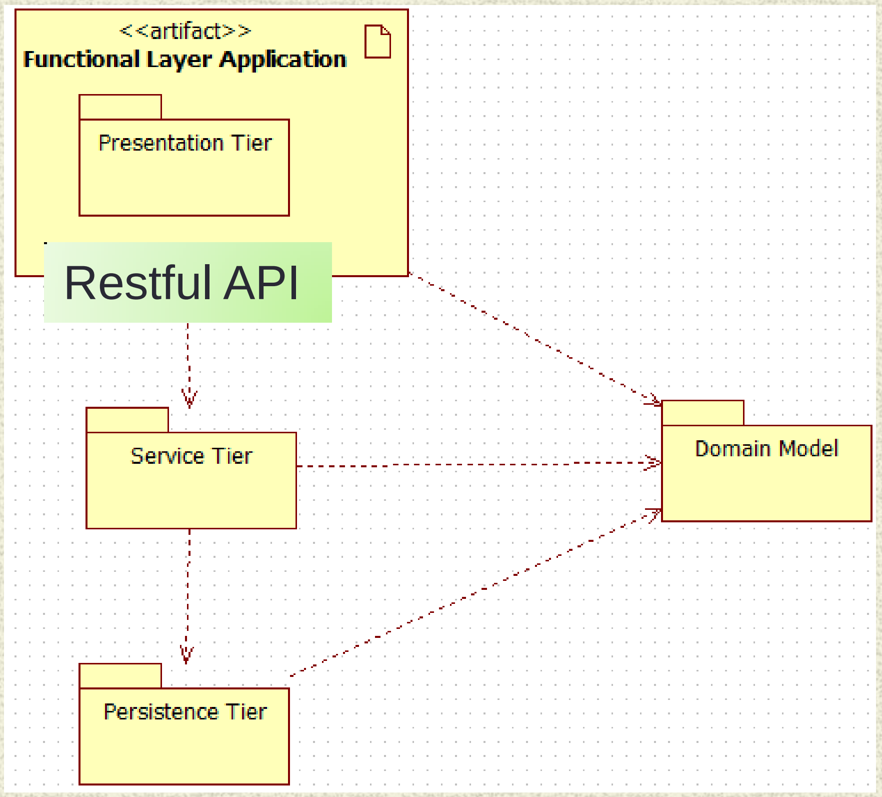
**Server-Side Application**

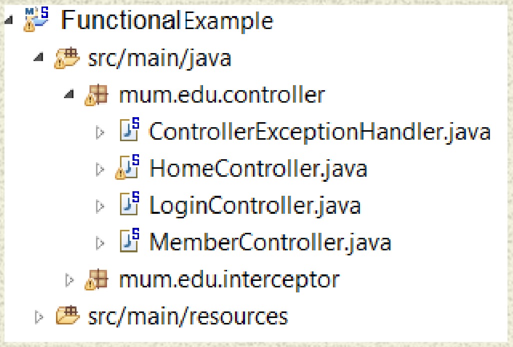
We will use the Spring N-Tier architecture

[Functional Separation]

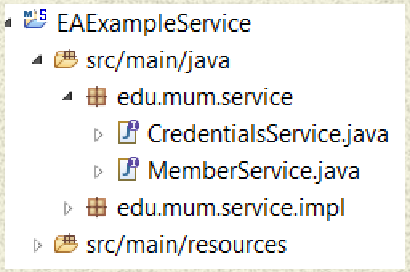
with Spring MVC Restful Web Services

# Functional N-Tier

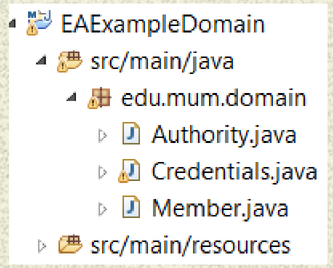




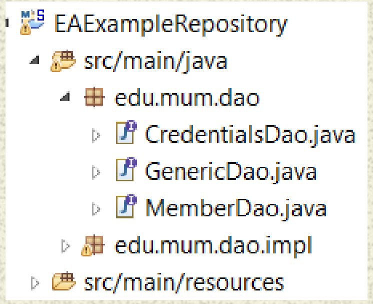
- FunctionalExample
  - src/main/java
    - mum.edu.controller
      - ControllerExceptionHandler.java
      - HomeController.java
      - LoginController.java
      - MemberController.java
    - mum.edu.interceptor
  - src/main/resources



- EAExampleService
  - src/main/java
    - edu.mum.service
      - CredentialsService.java
      - MemberService.java
    - edu.mum.service.impl
  - src/main/resources



- EAExampleDomain
  - src/main/java
    - edu.mum.domain
      - Authority.java
      - Credentials.java
      - Member.java
  - src/main/resources

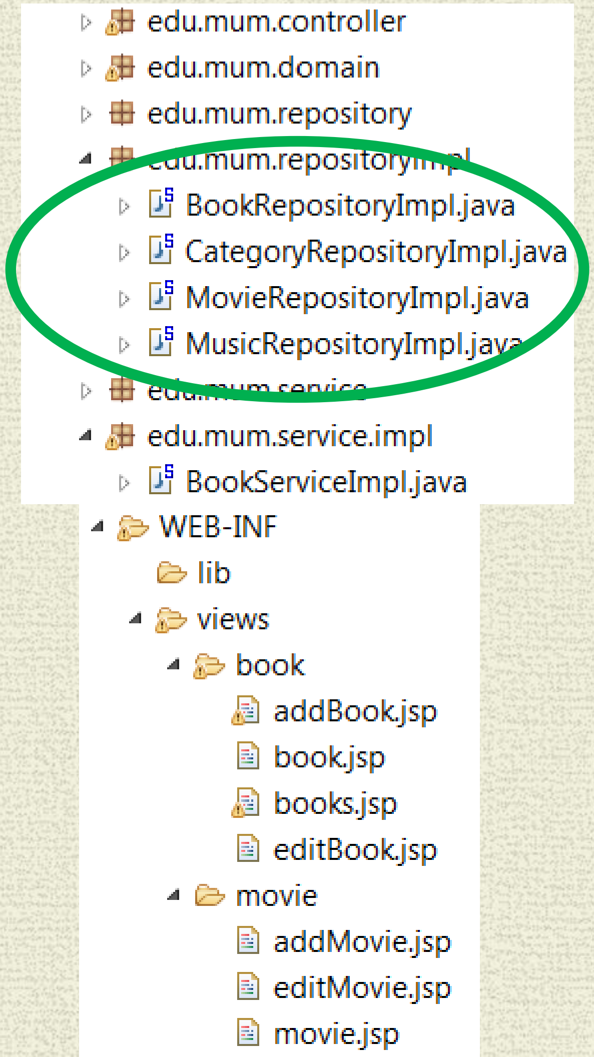


- EAExampleRepository
  - src/main/java
    - edu.mum.dao
      - CredentialsDao.java
      - GenericDao.java
      - MemberDao.java
    - edu.mum.dao.impl
  - src/main/resources

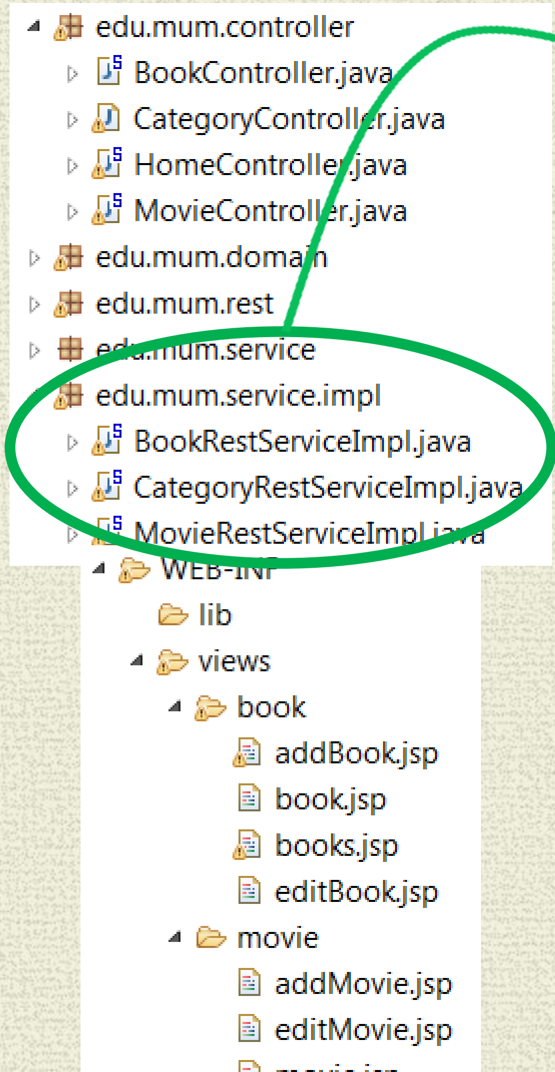
# Spring MVC N-Tier Structure

“normal” Spring MVC
Monolith

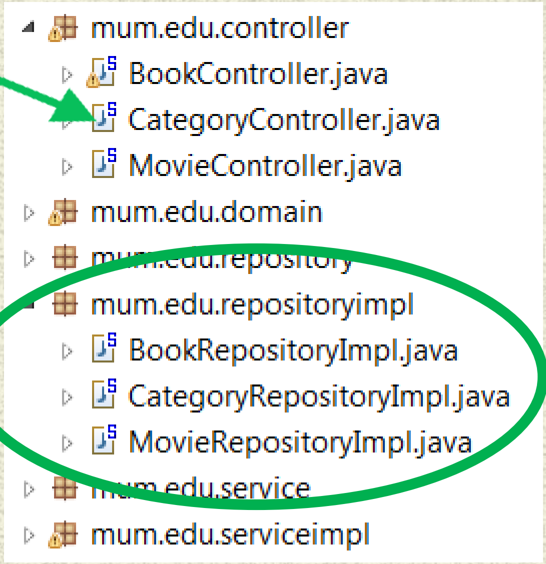
Functional Separation
Spring MVC    REST Services & Persistence



```
▷ edu.mum.controller
▷ edu.mum.domain
▷ edu.mum.repository
◢ edu.mum.repositoryimpl
    ▷ BookRepositoryImpl.java
    ▷ CategoryRepositoryImpl.java
    ▷ MovieRepositoryImpl.java
    ▷ MusicRepositoryImpl.java
▷ edu.mum.service
◢ edu.mum.service.impl
    ▷ BookServiceImpl.java
  ◢ WEB-INF
      lib
    ◢ views
      ◢ book
          addBook.jsp
          book.jsp
          books.jsp
          editBook.jsp
      ◢ movie
          addMovie.jsp
          editMovie.jsp
          movie.jsp
```



```
◢ edu.mum.controller
    ▷ BookController.java
    ▷ CategoryController.java
    ▷ HomeController.java
    ▷ MovieController.java
▷ edu.mum.domain
▷ edu.mum.rest
▷ edu.mum.service
◢ edu.mum.service.impl
    ▷ BookRestServiceImpl.java
    ▷ CategoryRestServiceImpl.java
    ▷ MovieRestServiceImpl.java
  ◢ WEB-INF
      lib
    ◢ views
      ◢ book
          addBook.jsp
          book.jsp
          books.jsp
          editBook.jsp
      ◢ movie
          addMovie.jsp
          editMovie.jsp
```



```
◢ mum.edu.controller
    ▷ BookController.java
    ▷ CategoryController.java
    ▷ MovieController.java
▷ mum.edu.domain
▷ mum.edu.repository
◢ mum.edu.repositoryimpl
    ▷ BookRepositoryImpl.java
    ▷ CategoryRepositoryImpl.java
    ▷ MovieRepositoryImpl.java
▷ mum.edu.service
▷ mum.edu.serviceimpl
```

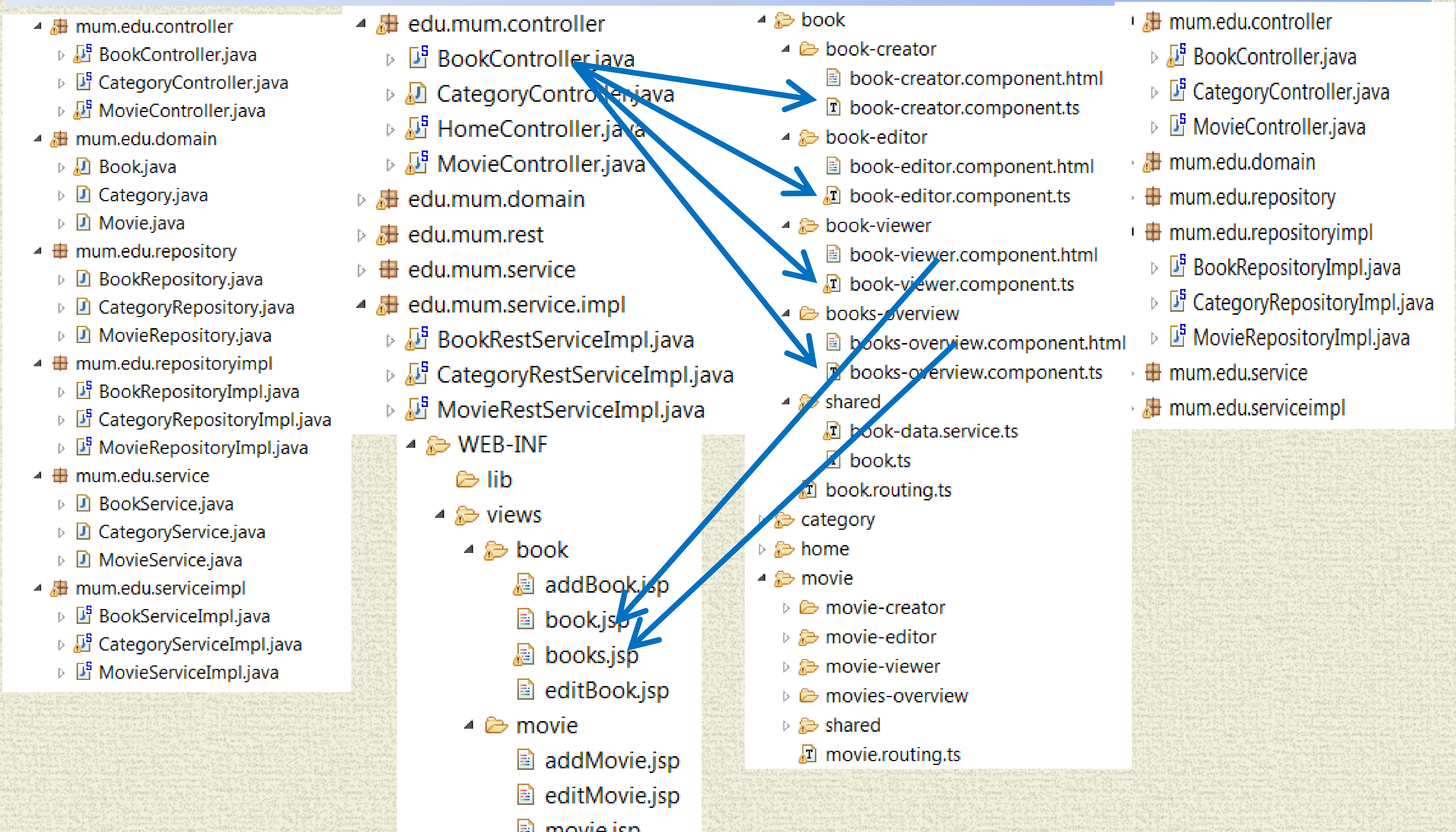
RESTful services implementation
NO UI [NO JSPs]

# Spring MVC SPA Structure

“normal” Spring MVC Monolith

JSPs Views Spring MVC

Functional Separation SPA Angular 2+ RESTful Back End



```
mum.edu.controller
  BookController.java
  CategoryController.java
  MovieController.java
mum.edu.domain
  Book.java
  Category.java
  Movie.java
mum.edu.repository
  BookRepository.java
  CategoryRepository.java
  MovieRepository.java
mum.edu.repositoryimpl
  BookRepositoryImpl.java
  CategoryRepositoryImpl.java
  MovieRepositoryImpl.java
mum.edu.service
  BookService.java
  CategoryService.java
  MovieService.java
mum.edu.serviceimpl
  BookServiceImpl.java
  CategoryServiceImpl.java
  MovieServiceImpl.java
```

```
edu.mum.controller
  BookController.java
  CategoryController.java
  HomeController.java
  MovieController.java
edu.mum.domain
edu.mum.rest
edu.mum.service
edu.mum.service.impl
  BookRestServiceImpl.java
  CategoryRestServiceImpl.java
  MovieRestServiceImpl.java
  WEB-INF
    lib
    views
      book
        addBook.jsp
        book.jsp
        books.jsp
        editBook.jsp
      movie
        addMovie.jsp
        editMovie.jsp
        movie.jsp
```

```
book
  book-creator
    book-creator.component.html
    book-creator.component.ts
  book-editor
    book-editor.component.html
    book-editor.component.ts
  book-viewer
    book-viewer.component.html
    book-viewer.component.ts
  books-overview
    books-overview.component.html
    books-overview.component.ts
  shared
    book-data.service.ts
    book.ts
  book.routing.ts
category
home
movie
  movie-creator
  movie-editor
  movie-viewer
  movies-overview
  shared
  movie.routing.ts
```

```
mum.edu.controller
  BookController.java
  CategoryController.java
  MovieController.java
mum.edu.domain
mum.edu.repository
mum.edu.repositoryimpl
  BookRepositoryImpl.java
  CategoryRepositoryImpl.java
  MovieRepositoryImpl.java
mum.edu.service
mum.edu.serviceimpl
```

## MVC Review

## MVC Origins Pre-Date the Web

- ***Probably the widest quoted pattern in UI development is Model View Controller (MVC) - it's also the most misquoted***.

  - Martin Fowler

**WITH SPA**

- MVC was created long before the first web application

  Xerox Parc late 70's

- Use case: Rich client ***Desktop*** Graphical User Interfaces
- Fundamental  concept  - use of the ***Observer*** pattern:

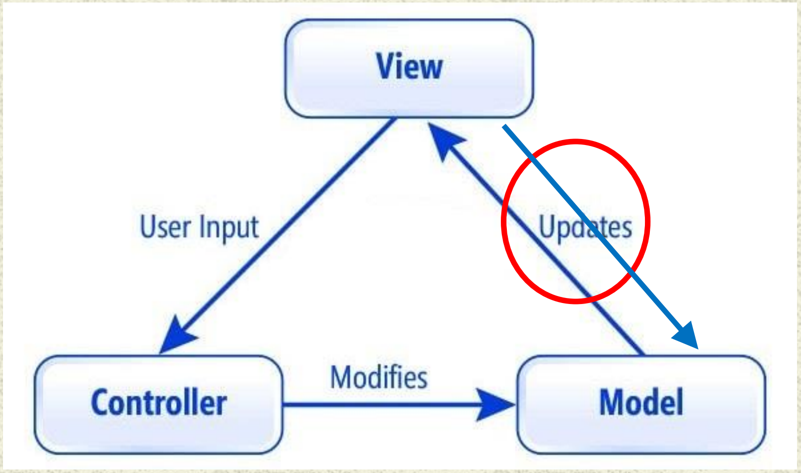
  ***views and controllers can react to model changes***

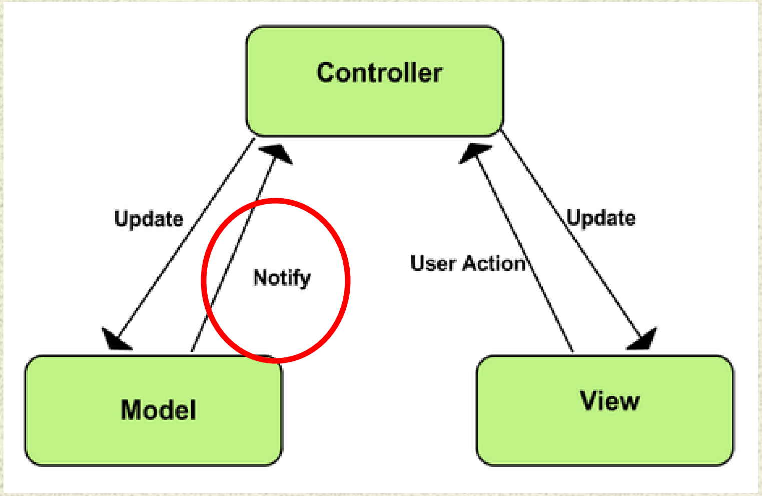
Possible on single machine – no network involved

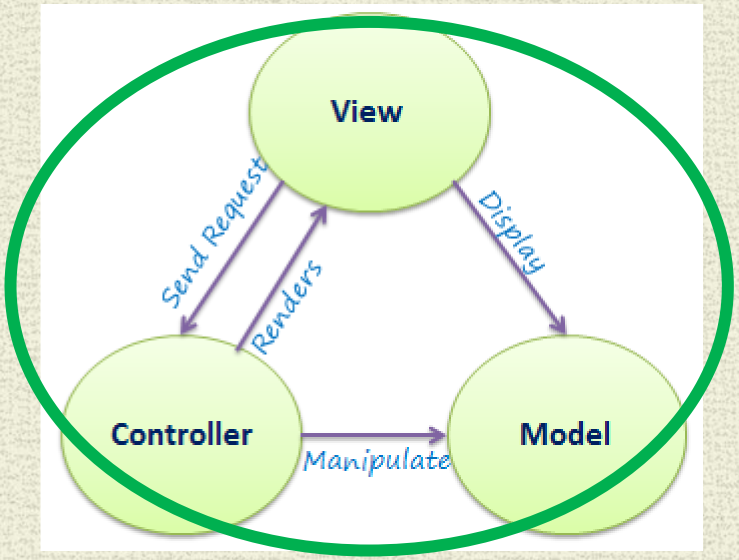
~~***On the Web***~~

~~***Request/response nature of HTTP disables the use of the Observer pattern***~~

# “Generic” MVC Diagrams







# What is a [MVC] Web Framework?

- Designed to simplify development
  - Has already been built, tested, and industry hardened
  - Increases reliability and reduces programming time
  - Adheres to DRY principle
  - Helps enforce best practices and rules
- *Common Features*
  - ***MVC Front Controller Pattern***
  - ***Validation Framework***
  - ***Declarative Routing***
  - ***Data Binding***
  - *Session Management*
  - *Security*
  - *Separation of Concerns*
- NOTE: All Frameworks have: Learning Curves”

## “Common Features” in Angular 2+

Here is what we will investigate in Angular 2+:

- MVC
  - “Front Controller”
  - Separation of Concerns
- Dependency Injection
- Data Binding
- Validation
- Internationalization
- Security

## Angular 2+

# Genuitec Webclipse
# Angular  Eclipse IDE

***A suite of Eclipse add-ons designed to improve the coding experience—especially for the modern web developer***

**Angular IDE** - complex web applications with Angular & TypeScript
**JSjet** JavaScript and TypeScript support with content assist, debugging and more
**Code Live** with **Live Preview** connects the IDE with browser for faster web coding
**Terminal+**, **Slack Integration** and **Navigation Aids** power up web development

Hides the CLI interface details from developer
Gives Angular2+ development a consistent Eclipse Look & Feel

"Free" version – 8 days/month                Webclipse

**See "Download Webclipse"; & "Basic Webclipse Function" documents**

## Angular 2+ “Background” Resources

Angular 2+ -- *bringing true object oriented Javascript web development to the mainstream, in a syntax that is strikingly close to Java 8.*

***Complete rewrite of AngularJS 1***

Depends on a variety of third-party packages, e,g:

Superset of JavaScript. Tooling for increased Productivity [ OO like]…”compiles” to Javascript

TypeScript

RxJS library of reactive programming

RxJS implements the **asynchronous Observable** pattern.

ECMAScript - ES6

***Node.js & Node Package Manager* [npm]** are **ALSO** essential to Angular

Third party packages – are maintained and installed with

***Node Package Manager [ which runs on node.js]***

# Angular Async calls
# Observables – Promises

- Angular 2+ favors

**Observables** over **Promises**

Angular $http based on Observables

Promises are native in ES6

Observables are part of Rx JS [ projected for ES7?, ES8…]

A Promise handles a **single event** when an operation completes or fails.

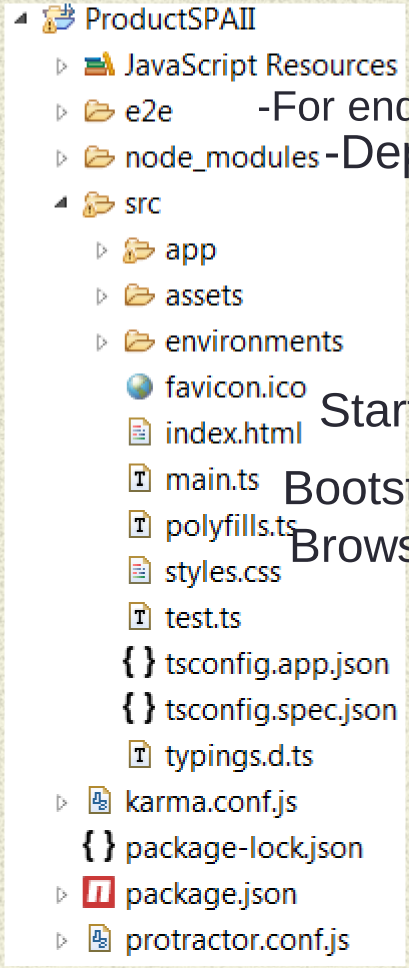
An Observable is like a **Stream** allows 0,1 or multiple events

Observables have lot of useful operators [Operators]

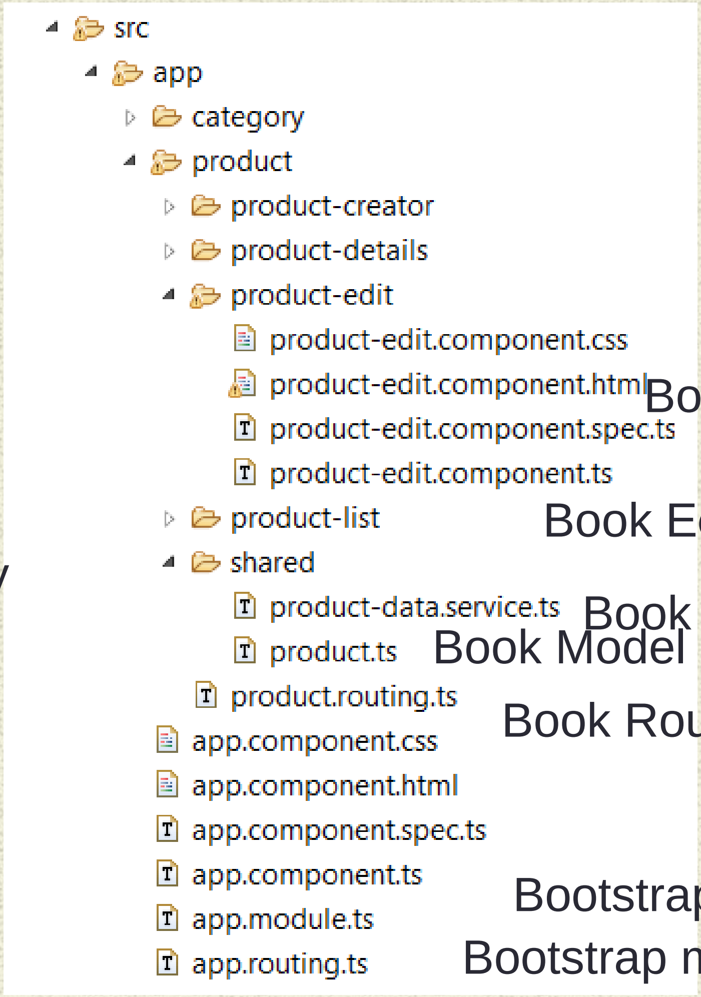
More Info:

Comparing Observables

# Angular 2+ Project

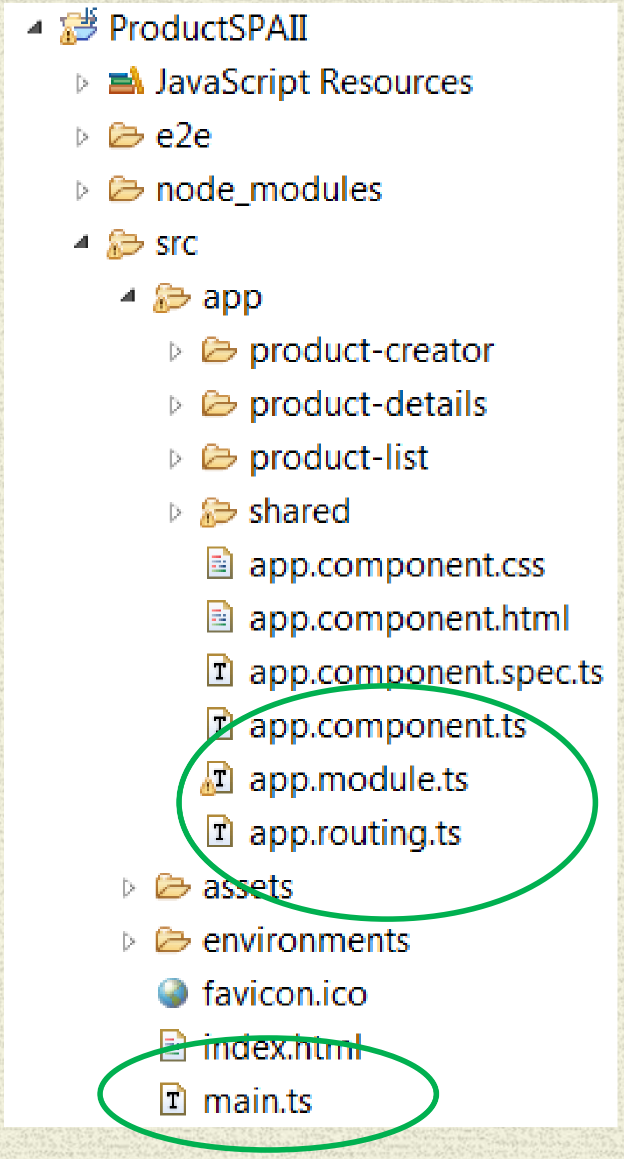


```
ProductSPAII
  JavaScript Resources
  e2e              -For end2end testing
  node_modules     -Dependencies
  src
    app
    assets
    environments
    favicon.ico
    index.html        Startup page
    main.ts           Bootstrap start
    polyfills.ts      Browser compatibility
    styles.css
    test.ts
    tsconfig.app.json
    tsconfig.spec.json
    typings.d.ts
  karma.conf.js
  package-lock.json
  package.json
  protractor.conf.js
```

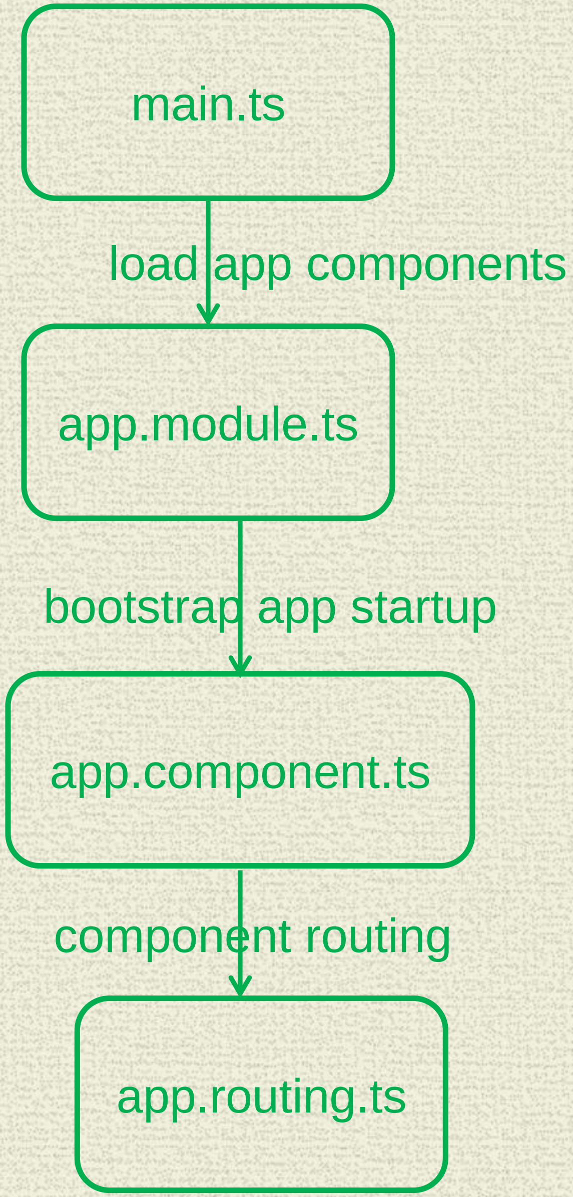


```
src
  app
    category
    product
      product-creator
      product-details
      product-edit
        product-edit.component.css
        product-edit.component.html     Book Editor View
        product-edit.component.spec.ts
        product-edit.component.ts       Book Edit "Controller"
      product-list
      shared
        product-data.service.ts         Book Rest Services
        product.ts                      Book Model
      product.routing.ts                Book Routing
    app.component.css
    app.component.html
    app.component.spec.ts
    app.component.ts                    Bootstrap component
    app.module.ts                       Bootstrap module
    app.routing.ts                      Application level Routing
```

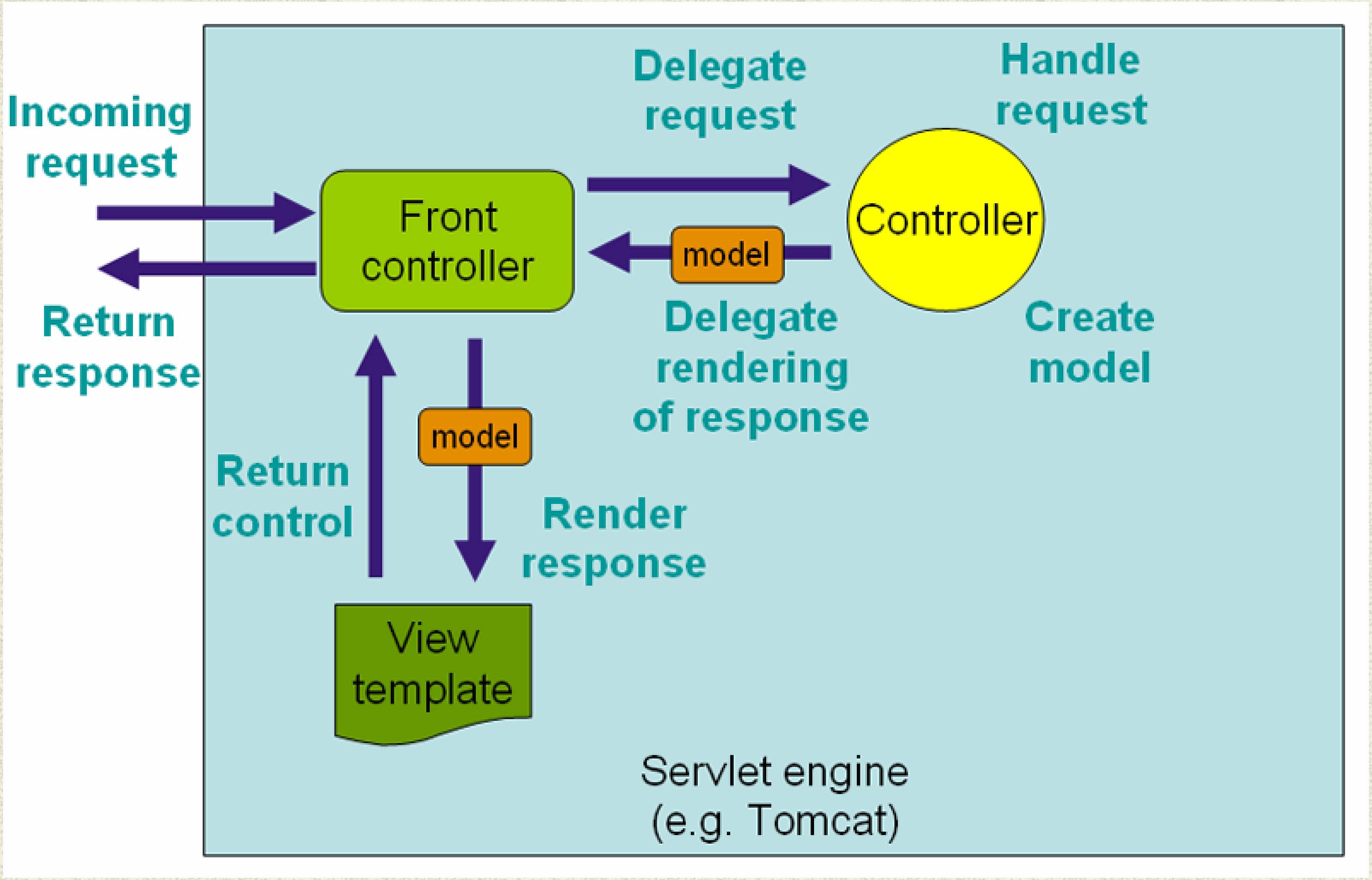
## Angular 2+ Startup



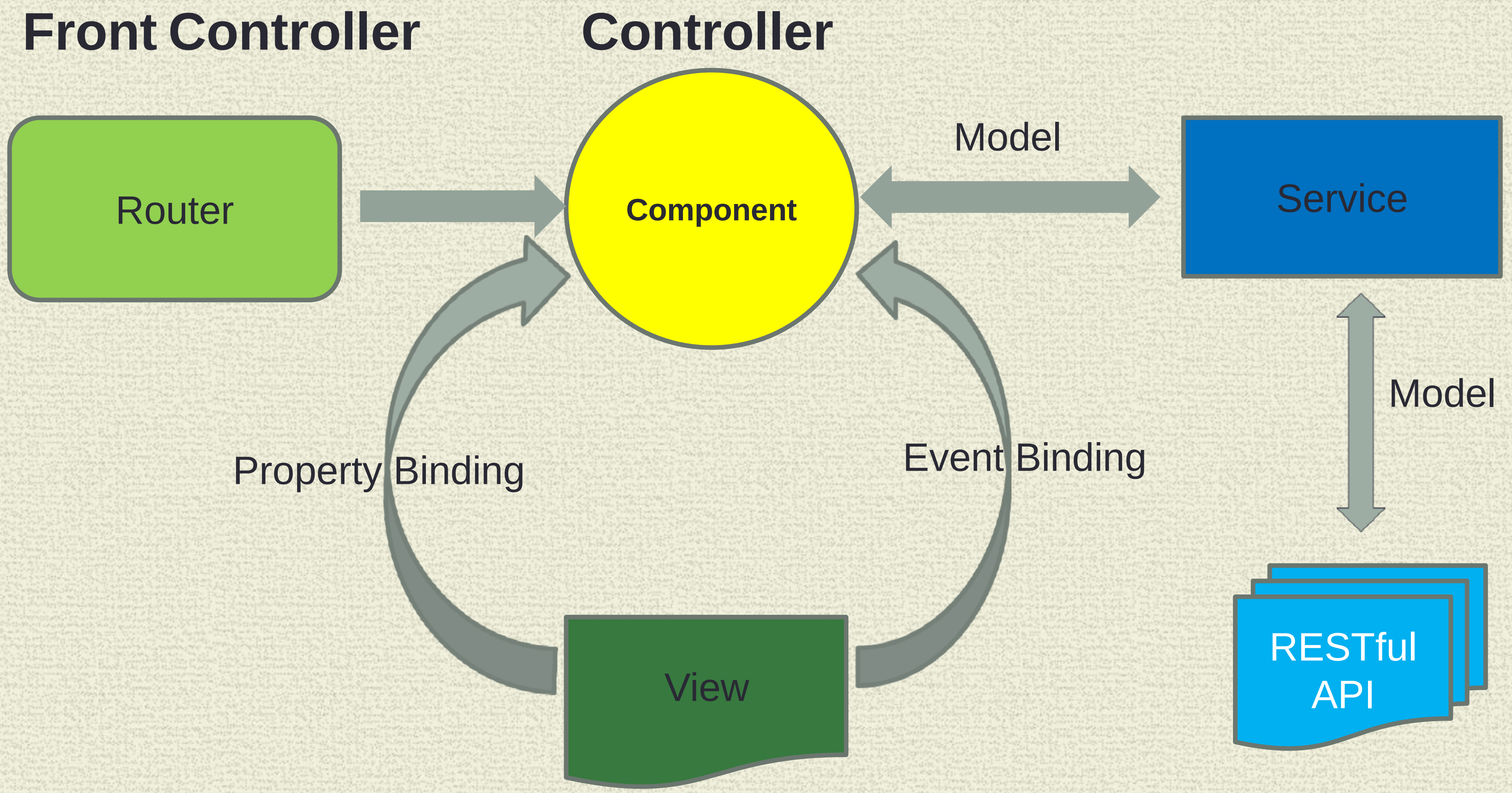
ProductSPAII
JavaScript Resources
e2e
node_modules
src
app
product-creator
product-details
product-list
shared
app.component.css
app.component.html
app.component.spec.ts
app.component.ts
app.module.ts
app.routing.ts
assets
environments
favicon.ico
index.html
main.ts



## Spring MVC Front Controller



## Angular 2+


Front Controller
Controller
Router
Component
Model
Service
Model
Property Binding
Event Binding
View
RESTful API

## Model - Typescript

**product.ts**

```
import { Category } from '../category/shared/category';

export class Product {
  id: number;
  name: string;
  description: string;
  price: number;
  category: Category;

  constructor() {
        this.category = new Category();
  }
```

## View

**product-list.component.html**

```
<div id="global">
    <h2>List of Products</h2>
  <ul class="product-list">

    <li *ngFor="let product of products"
        {{product.name}}
    </li>
  </ul>
 <div>
<a [routerLink]="['/products/new']"
   class="btn-primary btn-lg"><button>Add New Product</button>
</a>
```

[Cheat Sheet - Template Syntax]()

## Controller

### product-list.component.ts

```
import { Product } from '../shared/product';
import { ProductDataService } from '../shared/product-data.service';

@Component({
  templateUrl: './product-list.component.html',
 styleUrls: ['../app.component.css']
})

export class ProductListComponent implements OnInit {
  products: Product[] = [];

  constructor(
    private router: Router,
    private productDataService: ProductDataService) { }

  ngOnInit() {
    this.productDataService.getProducts()
         .subscribe(products => this.products = products );
```

# Front Controller [AKA Router]

```
const routes: Routes = [
  {
    path: '',            // default
    redirectTo: '/products/new',
    pathMatch: 'full'
  },
  {
     path: 'products/new',
    component: ProductCreatorComponent
  },
  {
    path: 'products/:id',
    component: ProductDetailsComponent
  },
  {
    path: 'products',
    component: ProductListComponent
```

NOTE: not “network” URLs but SPA “on the client URLs”

## Main Point

- The fundamental value of the MVC pattern is the implemenation of the principle of Separation of Concerns [SoC]. An SPA implements a variation on the MVC pattern making clear use of the SoC principle.
- *Fundamental values [AKA laws of nature] will always be relevant and applicable in a variety of situations.*

## Dependency Injection

## Dependency Injection

**@Inject(Http) - Argument Level injection indicator**

```
export class ProductDataService { ...
        constructor(@Inject(Http) private http) { }
```

Place on EVERY argument

**@Injectable() – Alternative to @Inject**

Class level– means inject ALL constructor params

```
@Injectable()
export class ProductDataService { ...
          constructor(private http: Http) { }
```

Analogous to @Autowired on a constructor

Dependency Injection

## Component Dependency Injection

Components are a "special case"

@Component "automatically" generates identifier for constructor parameter [does NOT need @Injectable()]

```
export class ProductListComponent implements OnInit
```

"Automatically Injected"

```
constructor( private productDataService:
                                ProductDataService)
```

# Dependency Instance Creation "Configuration"

A provider create instances of the services for DI

At the application level [root NgModule] creates a singleton of the listed resources

**EXAMPLE**

App.module.ts

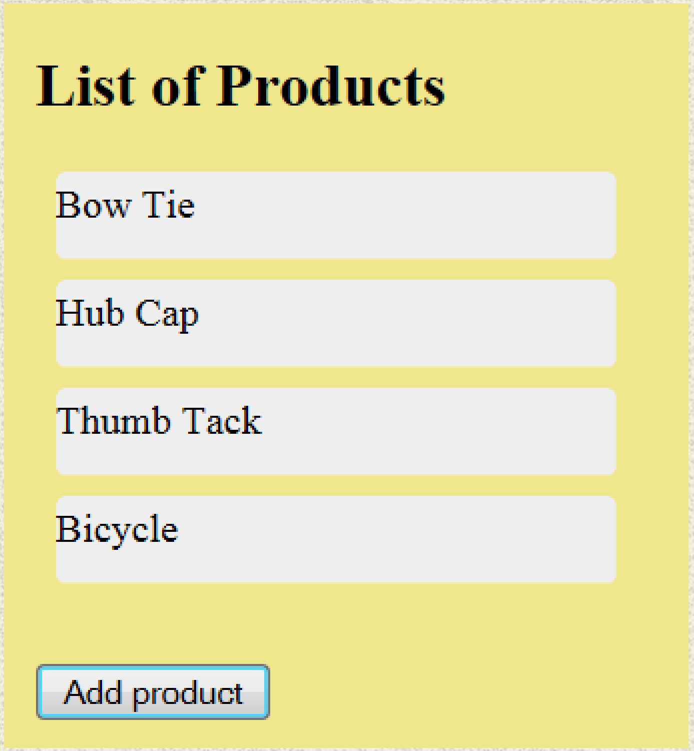
```
@NgModule({
  imports: [
       ...
     HttpModule
  ],
 providers: [
         ProductDataService,
 ],
```

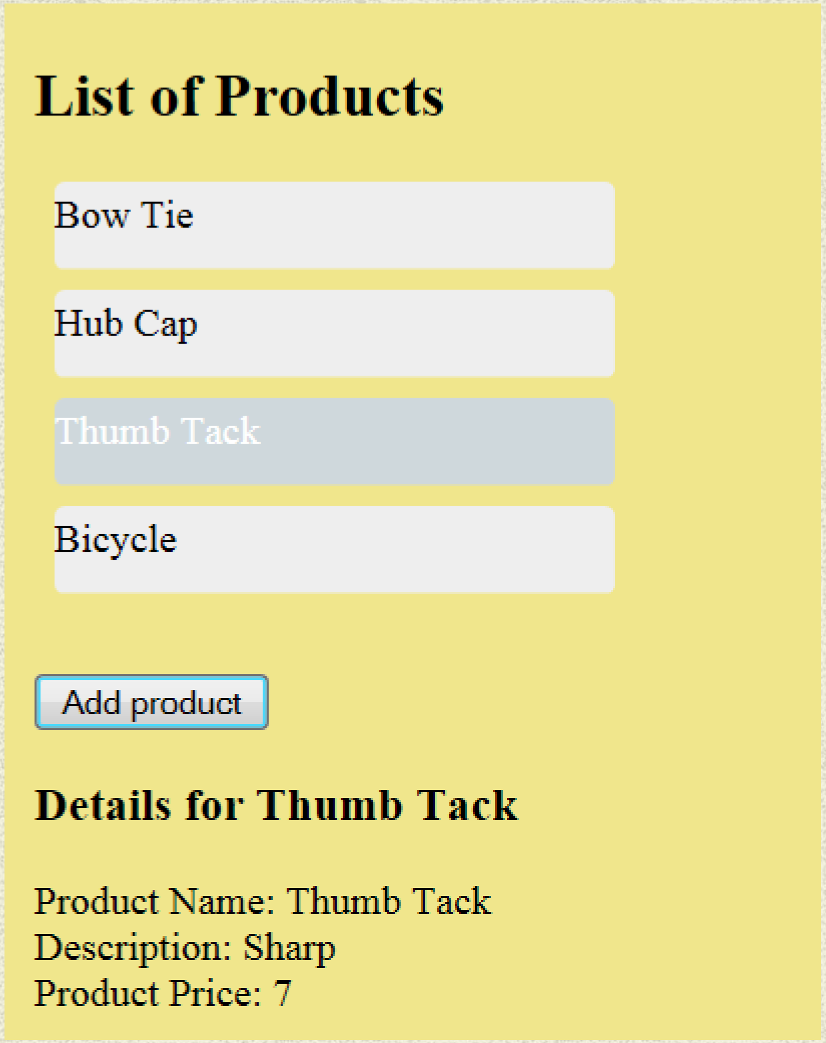
Http is analogous to a Spring provided Bean declaration
Declared in HttpModule
Used in Slide 26

Analogous to registering an application specified service provider for creation & injection
Used in Slide 27

## Data Binding

## Javascript “data binding”





Details for Thumb Tack

Product Name: Thumb Tack
Description: Sharp
Product Price: 7

Result of Javascript function →

**See ProductMVC Demo**

# Plain Javascript - “manual” data binding

“Event Binding”

```
// Click on Product [row]  in product-details
      $("#productList").find("tr").click(function() {
         var productId=  $(this).find("td:first").html();
         showProduct(productId);
       });
// Get product object from Server...
      showProduct =  function(productId) {
        $.ajax({ url: contextRoot + '/' + productId,
            type: 'GET',  async:false, dataType: "json",
        success: function (response) {
           displayProduct(response);
        } ...
      displayProduct = function(product) {
       $('#details').html("");
       var details = '<h3>Details for ' + product.name + '</h3>';
       details += 'Product Name: ' + product.name + '<br/>';
       details += 'Description: ' + product.description + '<br/>';
       details += 'Product Price: ' + product.price;
       $("#details").html(details);

       $('#details').show();
```

“Property” Binding”

**See ProductMVC Demo**

# Angular One Way Binding

```
<div id="global">
    <h2>List of Products</h2>
 <ul class="product-list">

  <li *ngFor="let product of products"
     [class.selected]="product === selectedProduct"
      (click)="onSelect(product)" >
      {{product.name}}
    </li>
    </ul>
 </div>
<product-details [product]="selectedProduct"></product-details>
```

Event Binding - product-list Component

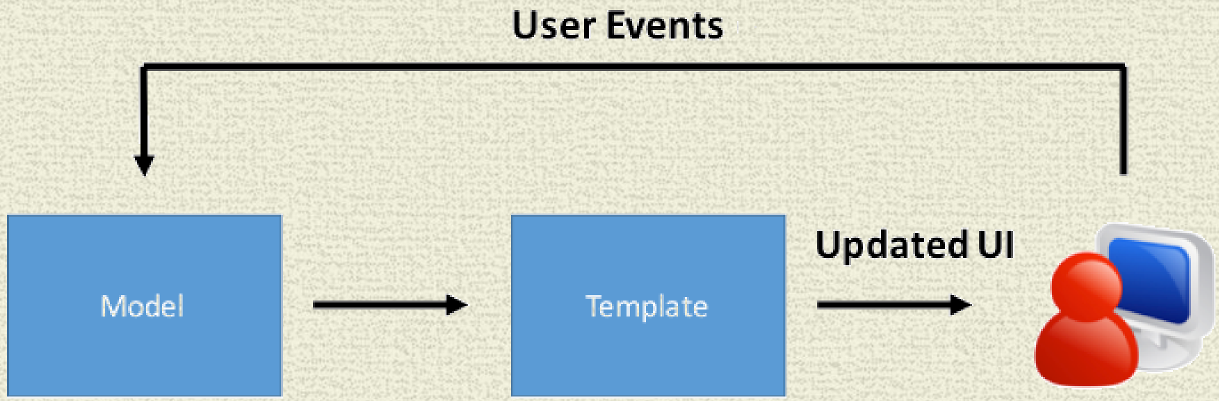
We have "Modularized" Property Binding
In separate component –
product-details component
Using @Input

**See ProductSPA Demo** **Template Syntax**

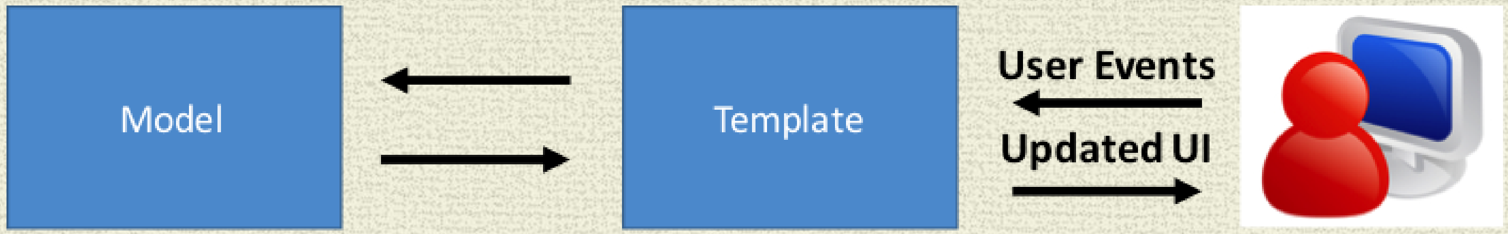
**Also See Form input in ProductSPAllOneway Demo**

# Angular 2+ One Way AND Two Way Data Binding

**One-way data binding flows from the model to the UI ONLY.** When the UI element is always updated only from the model, it has one source of data. The element itself is not directly updated by the triggered event.

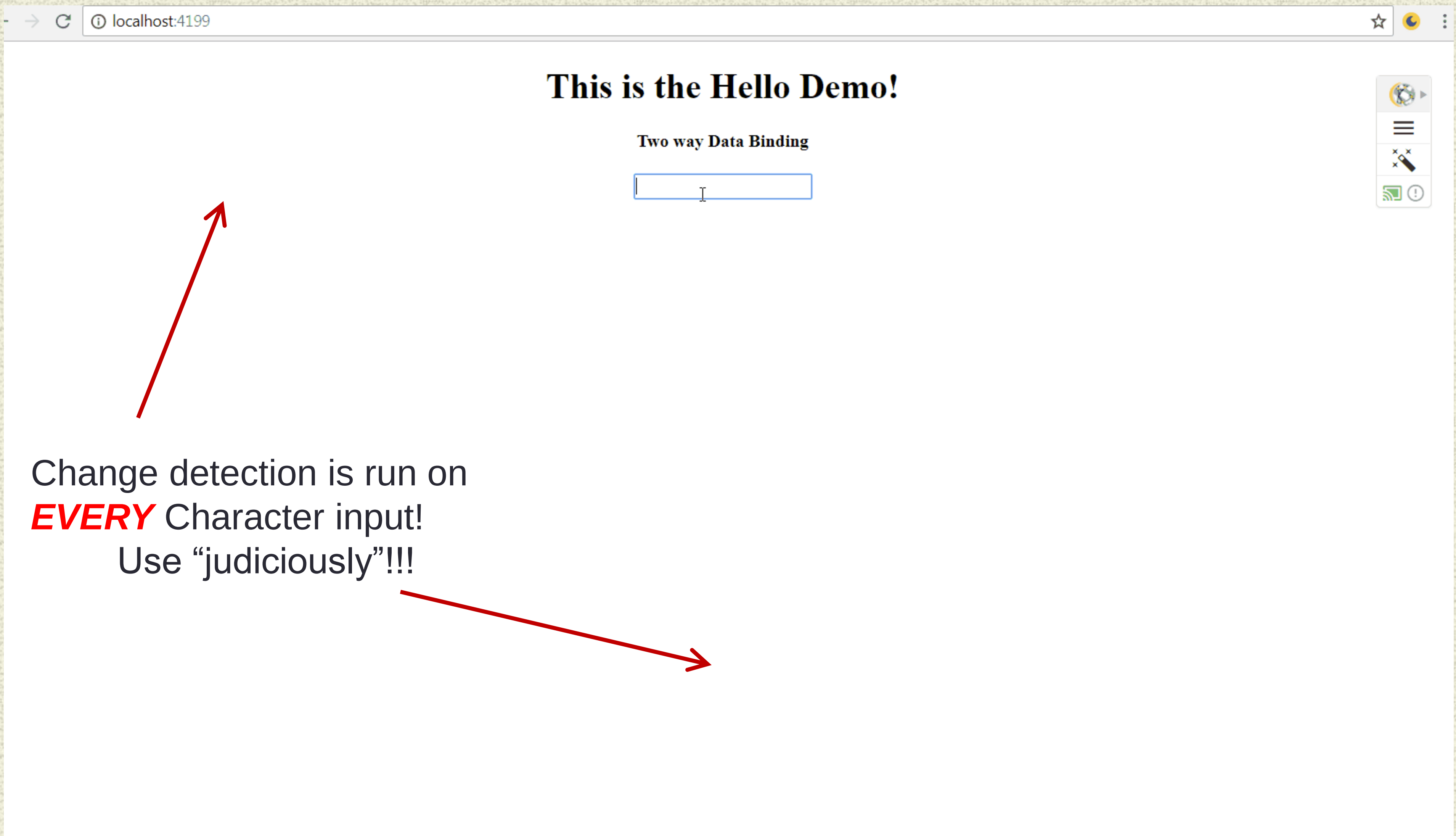


**Two-way data binding also allows the flow of data from the UI to the model.** The UI element is updated from BOTH the model and user input -- **TWO** sources of data.



The implementation of data binding [Both types] involves **Change Detection** – a mechanism to determine if changes to the model have occurred.

## Angular 2+
## Two Way Data Binding Example



This is the Hello Demo!

Two way Data Binding

Change detection is run on *EVERY* Character input!
Use "judiciously"!!!

## Two Way Data Binding Example

Two-way data binding consists of event binding AND property binding

Using **ngModel Directive**

```
<input [(ngModel)]="username">
<p>Hello {{username}}!</p>
```

**Event binding**

**Property binding**

**“Manually” Without ngModel**

```
<input [value]="username" (input)="updateName($event)">
<p>Hello {{username}}!</p>
```

**[value]=”username”** - Binds username to the input value property

**(input)=”expression”** - Binds to the input element input event

**updateName($event)** - Executed on input event

**$event** - Angular event expression has value of event’s payload

**Also See Form input in ProductSPAII Demo**

## Main Point

Data Binding in Angular 2+provides a comprehensive
[1 & 2 way binding] and a way to dynamically synchronize
[Change Detection] the entire SPA.

*Transcendental Consciousness is a state of Complete Synchronization.*

## Angular 2+ Forms

Angular offers two form-building technologies:
*reactive* forms and *template-driven* forms.

| **Template Driven Forms** | **Reactive Forms** |
|---|---|
| Form resources created by template directives | Form resources defined in component |
| Easy to use | Longer learning curve |
| Simple Scenarios | Complex Scenarios |
| Minimal component code [controller] | More component code - Less HTML |
| Two way data binding | Two way data binding “not supported” Assumes immutable data model |
| Async Page Creation | Sync Page Creation |
| Unit testing problematic | Easier unit testing |

## FormControl FormGroup FormArray

**FormControl Class**

FormControl represents a form control element; input box, select box, radio buttons, dropdown etc. Tracks the value and validation status of an individual form control.

**FormGroup Class**

Aggregates the values of each child FormControl into one object, with each control name as the key. Tracks the value and validity state of a group of FormControl instances.

**FormArray Class**

Variation of FormGroup. Variable length array of FormControls. Useful for dynamically adding controls to the form.

## Form Directives

### Template-driven Form

**ngForm Directive**

**Creates** a top-level FormGroup instance and binds it to a form to track form input values and validation status

***By default ngForm is hidden***

**ngModel Directive**

**Creates** a FormControl instance from a domain model and binds it to a form control element.

### Reactive Form

**FormGroupDirective**

Binds an existing FormGroup to a DOM element.

**FormControlName**

Syncs a FormControl in an existing FormGroup to a form control by name.

# Template-driven Form

```
<form  (ngSubmit)="saveProduct()" >
 <legend>Add a product</legend>
   <p>
        <label for="name">Product Name: </label>
        <input type="text" [(ngModel)]="product.name" id="name" name= "name">
   </p>
   <p>
      <label for="description">Description: </label>
      <input type="text" [(ngModel)]="product.description" id="descri
                                                    name= "description" >
    </p>
    <p>
        <label for="price">Price: </label>
        <input type="text" [(ngModel)]="product.price" id="price" name= "price">
    </p>
    <p id="buttons">
        <button type="submit" id="submit">Add Product</button>
    </p>
</form>
```

ngForm is hidden – could get it by adding #productForm="ngForm"

FormControls Added to ngForm FormGroup

## Reactive Component

```
export class ProductCreatorComponent {
  product: Product = new Product();
  productForm: FormGroup;
  private subscription: any;

  constructor(private router: Router,
              private productDataService: ProductDataService,
              private formBuilder: FormBuilder) {}

  ngOnInit() {
    this.productForm = this.formBuilder.group({
      name: '',
      description: '',
      price: ''
    });

  }
```

"Builds" FormGroup with FormControls

Binds to the productForm FormGroup declared in component

# Reactive Form

```
<form [formGroup]="productForm" (ngSubmit)="saveProduct(productForm.value)" >
    <fieldset>
        <legend>Add a product</legend>
        <p>
            <label for="name">Product Name: </label>
             <input formControlName="name" >
         </p>
        <p>
            <label for="description">Description: </label>
              <input formControlName="description">
            </p>
        <p>
            <label for="price">Price: </label>
              <input formControlName="price">
         </p>
        <p id="buttons">
            <button type="submit"  id="submit">Add Product</button>
        </p>
```

Bind to the FormControls declared in component

## Validation

Angular has a set of  validators that is similar to the HTML5 attributes:

| Constraint | Description |
|---|---|
| min | Must be an integer <= the value |
| max | Must be an integer >= the value |
| required | Must have a value |
| email | Must have valid email syntax |
| minLength | Must have a value of a minimum length. |
| maxLength | Must have a value of a maximum length. |
| pattern | Must match the regular expression defined in the regexp element. |

Template driven forms use HTML attributes

Reactive forms use functions

Angular built-in Validators

## Template-driven Validation

```
<p>
   <label for="description">Description: </label>
   <input type="text" [(ngModel)]="product.description"
            id="description" name="description"
            required #description="ngModel">
</p>
 <!--  show error under conditions: valid AND touched element -->
<div id="errors" [hidden]="description.valid || (!description.touched)">
        <div *ngIf="description.errors?.required">
            Description is required.
        </div>
</div>
```

# Reactive Validation

## Component

```
this.productForm = this.formBuilder.group({
      name: ['',Validators.required],
      description: '',
      price: ''
     });
```

## HTML Form

```
<p>
             <label for="description">Description: </label>
              <input formControlName="name" >
</p>
<!--  show error under conditions: valid AND touched element -->
<div id="errors" [hidden]="description.valid || (!description.touched)">
       <div *ngIf="description.errors?.required">
            Description is required.
       </div>
</div>
```

## Client Side Validation Caveat

Client side validation doesn't remove the need for validation on the ***server side***.

Client side validation “reduces” invalid form data BUT invalid data can still exist:

- Non-compliant browsers (e,g. without HTML5
- Complex constraints not appropriate to client side
- “Creative” individuals trying to trick your web application

***Server side validation is ALWAYS necessary***

## Main Point

Angular 2+ has two form-building technologies:

*reactive* forms and *template-driven* forms.

Template forms are simpler to use & supports 2 way binding

Reactive forms handle complex scenarios better.

*Scientific studies show that participants in the Transcendental Meditation Program attain a natural integration of diverse styles of brain functioning & are able to respond more flexibly and dynamically to tasks, as needed.*

## Angular & I18n

Angular only works with one language at a time, you have to completely reload the application to change the language.

- Displays dates, number, percentages, and currencies by locale.

- Translates text in component templates
- Messages are stored in XML files
- Modifying the application invalidates existing message files
- Translations are baked into the application during compilation

## Angular 2+ i18n File Format

Angular 2+ uses the

**XML Localisation Interchange File Format [XLIFF 1.2]**

for i18n support

It is an OASIS standard designed as a format to exchange localization information

The Angular i18n Cookbook Internationalization (i18n) describes in detail, how to markup your templates and how to extract the i18n information …

**From the Angular i18n Cookbook:**

*In a large translation project, you would send the messages.fr.xlf file to a French translator who would enter the translations using an XLIFF file editor.*

***We can Improve on that!!***

# Generic i18n Process

1. Mark the template content with "i18n" to indicate translation targets

```
<legend i18n>Add a product</legend>
```

2. Run the Angular tool (ng-xi18n) to extract the content into a "language" file

```
ng xi18n --output-path src/locale --locale en
                    [messages.xlf]
```

3. Copy and translate the extracted file for every supported language

```
                    [messages.fr.xlf, messages.zh.xlf]
```

4. Compile/run a special version of your app every supported language

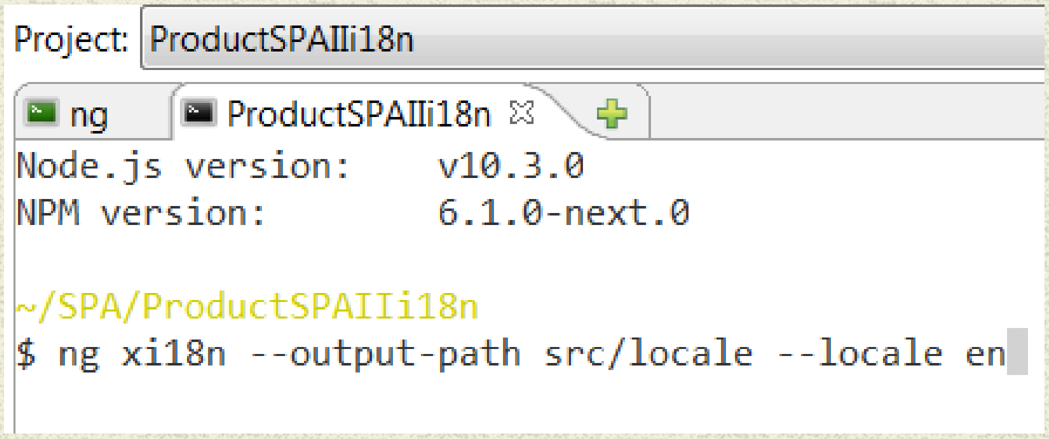
```
ng serve --aot --i18n-file=src/locale/messages.fr.xlf --locale=fr
ng serve --aot --i18n-file=src/locale/messages.zh.xlf --locale=zh
```

***ANY changes in ANY [html] file requires steps 2-4 repeated!!!!***

***We can Improve on that!!***

## EXAMPLE: Create French language version

- Create locale folder under src
- Run xi18n from Terminal+
- *ng xi18n --output-path src/locale --locale en*



Project: ProductSPAIIi18n

ng | ProductSPAIIi18n

```
Node.js version:    v10.3.0
NPM version:        6.1.0-next.0

~/SPA/ProductSPAIIi18n
$ ng xi18n --output-path src/locale --locale en
```

- Generates messages.xlf file – for english in src/locale
- Add <target> for every <source> with French translation and save as .fr.xlf
- Example

</trans-unit>

<source>List of Products</source>

**<target>Liste de Produits</target>**

## “Custom” Support to modify i18n Files

**Reduces the boilerplate repetitive process[Steps 2-4] in the “Generic i18n Process”**

Tool name is Xliffmerge

**Download here:**

https://www.npmjs.com/package/ngx-i18nsupport?activeTab=readme

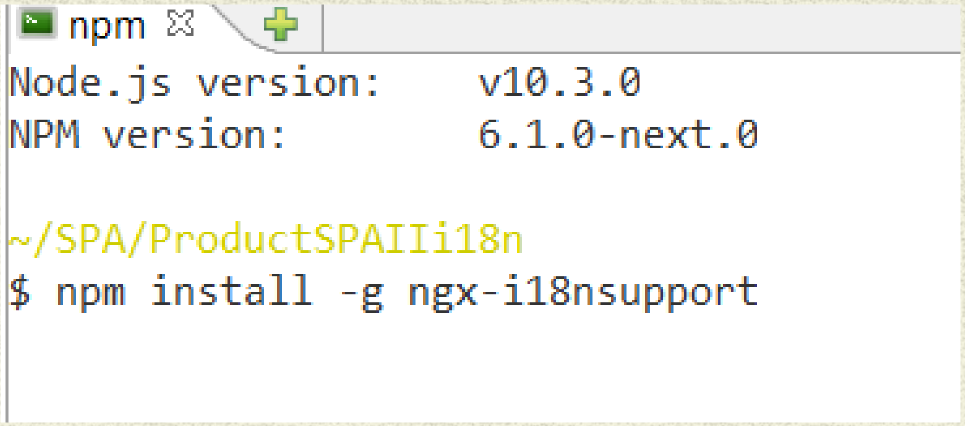
**Install:**

npm install –g ngx-i18nsupport

**Create config file in src folder:**

xliffmerge.json config file

```
  "xliffmergeOptions": {
    "srcDir": "src/locale",
    "genDir": "src/locale"
```



```
Node.js version:     v10.3.0
NPM version:         6.1.0-next.0

~/SPA/ProductSPAIIi18n
$ npm install -g ngx-i18nsupport
```

**Run Xliffmerge  to create “editable” versions:**

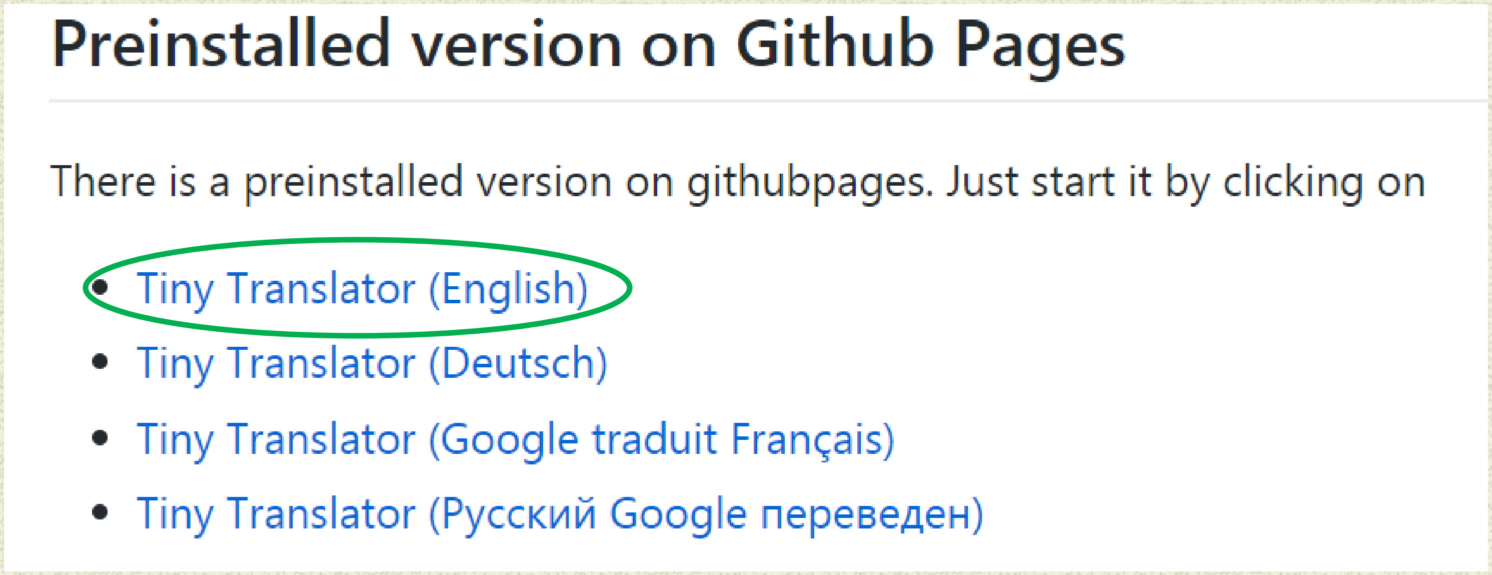
```
ng xi18n --output-path src/locale  && xliffmerge --profile
             src/xliffmerge.json en fr"
```

Assumes *ng xi18n* has been already run to create default language version

# XLIFF “sample” File editor

**…send [.xlf] to a translator "*who fills in the translations using one of the many XLIFF file editors*". (citation from the Angular cookbook).**

**Pre-installed version of “tiny-translator” can be run “on the web”:**
[https://github.com/martinroob/tiny-translator](https://github.com/martinroob/tiny-translator)


Preinstalled version on Github Pages

There is a preinstalled version on githubpages. Just start it by clicking on

- Tiny Translator (English)
- Tiny Translator (Deutsch)
- Tiny Translator (Google traduit Français)
- Tiny Translator (Русский Google переведен)

**See Translate i18n Document**

# “Improved” i18n Process

1. Mark the template content with “i18n” to indicate translation targets

```
<legend i18n>Add a product</legend>
```

2. Run the Angular tool (ng-xi18n) to extract the content into a “language” file

```
ng xi18n --output-path src/locale --locale en«
                    [messages.xlf]
```

3. Copy Run the Xliffmerge to create “editable” versions

```
ng xi18n --output-path src/locale  && xliffmerge –profile
        src/xliffmerge.json en fr
```

4. Run “Tiny Translator” to edit French version
5. Compile/run a special version of your app every supported language

```
ng serve --aot --i18n-file=src/locale/messages.fr.xlf --locale=fr
```

**ProductSPAIIi18n “edit” Demo**