PS05EMCA23 Web Application Frameworks (Angular Part)

Dr. J. V. Smart

Table of Contents

- Syllabus
- Introduction to the Angular Client-side Web Application Framework
- Important Concepts
- MVC (Model-View-Controller)
- MVVM (Model-View-ViewModel)
- Types of Data Binding in Angular
- Dependency Injection

Syllabus

COURSE NO: PS05EMCA23

Web Application Frameworks

w.e.f. June 2017

(3 Lectures & 1 Seminar/Tutorial per Week Total Marks: 100)

COURSE CONTENT:

1. **Basic Web Application Development Tools**
    - Introduction to HTML5, CSS3

- Interactive web pages using JavaScript
- The JQuery library
- JavaScript user interface library

2. **Website Development using WordPress**
   - Introduction to WordPress
   - Creating simple web sites using WordPress: Themes, Pages, Menus, Multimedia elements
   - Link management
   - Use of Plugins
   - Developing websites using plugins
3. **Client-side Web Application Framework**
   - Setting up Project, project organization and management
   - Templates
   - MVC Architecture
   - Data binding
   - Dependency injection
   - Routing
4. **Server-side Web Application framework**
   - Application structure
   - MVC Architecture
   - Routing
   - Helpers
   - Libraries
   - Form validation
   - Session management
   - Active record

MAIN REFERENCE BOOKS:

1. Dane Cameron, “HTML5, JavaScript and jQuery”, Wrox publication
2. David Sawyer McFarland, “CSS3”, O’reilly
3. Brad Green and Syham Seshadri, “AngularJS”, O’Reilly
4. Jake Spurlock, “Bootstrap”, O’Reilly
5. Mathew MacDonald, “WordPress”, O’Reilly
6. Thomas Myer, “Professional CodeIgniter”, Wrox Professional Guides

BOOKS FOR ADDITIONAL READING:

1. Valeri Karpov, Diego Netto, “Professional AngularJS”, Wrox publication
2. Zak Ruvalcaba, Anne Boehm, “HTML5 and CSS3”, Murach
3. Bear Bibeault, Yehuda Katz, “jQuery in action”, 2 nd edition, Dreamtech press
4. Karl Swedberg, Jonathan Chaffer, “jQuery 1.4 Reference Guide”, PACKT publishing
5. Other online references

Introduction to the Angular Client-side Web Application Framework

Angular (earlier called AngularJS) is an open-source client-side web application framework developed by Google. It uses the TypeScript, a superset of JavaScript.

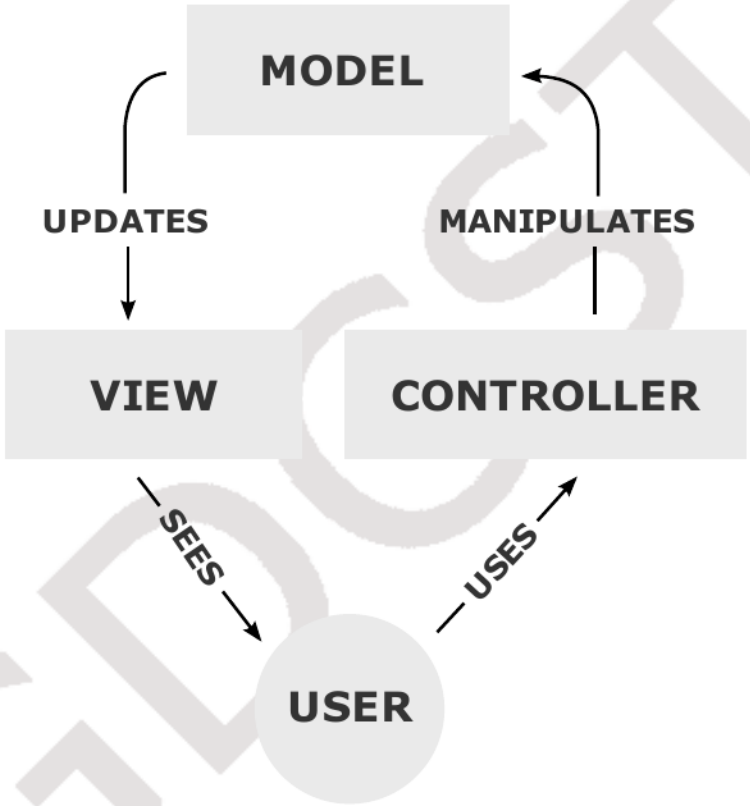
The Angular Logo

Important Concepts

MVC (Model-View-Controller)

MVC (Model-View-Controller) is an architectural pattern for software

design. As the name suggets, an MVC application consists of three parts - Model, View and Controller.



The MVC Pattern

- **Model** The model is responsible for storing and manipulating the data of the application. It is also responsible for implementing the business logic
- **View** The view is a representation of (part of) the data that is presented to the user. Same data may be presented in different form by different views (e.g. table v/s chart)
- **Controller** The controller accepts user input in the form of events, optionally validate it and then either sends it to the model for action or selects the appropriate view to be displayed. Often, it is also responsible for fetching the data needed by the view from the model

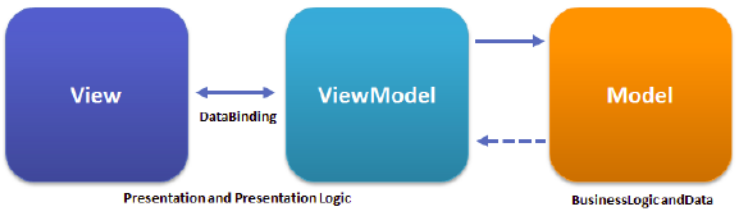
Major advantages of the MVC pattern include separation between presentation and logic, possibility of simultaneous independent development of the model, view and controller components and the ability to have multiple views for presenting the same data in different

ways. MVC pattern was conceived for GUI application development, but it is heavily used for web application development. Different frameworks implement the pattern in different ways. Several software architectural patterns based on MVC have been developed.

AngularJS (the first version of Angular) used MVC. Angular (version 2 onward) follows a component architecture and broadly follows the MVVM pattern.

MVVM (Model-View-ViewModel)

MVVM (Model-View-ViewModel) is an architectural pattern for software design that evolved from the MVC pattern. An MVVM application consists of three parts - Model, View and ViewModel. In most cases, MVVM implementations have data binding between the view and the view model.



The MVVM Pattern

- **Model** The model is responsible for storing and manipulating the data of the application. It is also responsible for implementing the business logic
- **View** The view is a representation of (part of) the data that is presented to the user. Same data may be presented in different form by different views (e.g. table v/s chart). The view gets the data to be displayed from the view model and presents them to the user. The view is also responsible for accepting user input, potentially validating it and then updating the view model. The view contains only the UX (User eXperience) elements and is developed by the designers. It contains almost no code
- **View Model** The view model fetches from the model the part of the data to be presented by the view. This component contains the code

for fetching the data, but has no user interface elements. The view model exposes the data from the model to the view, which presents them to the user

- **Data Binding** In most MVVM implementations, there is a two-way binding between the display elements in the view and the data in the view model; such that whenever the data in the view changes, the view model is updated automatically, and vice versa

A major advantage of the MVVM pattern over the MVC pattern is the complete separation between the UX (user interface design) and the coding needed to fetch the data to be displayed and to propagate the modifications made by the user in the view to the model. Another important advantage of the MVVM model is that since the views contain no code and the view model contains a representation of the view, automated testing frameworks can easily test the application by simulating user actions in the view by calling the methods in the view model that handle those actions.

The disadvantage of the MVVM pattern is the additional memory needed to hold the view model and the performance impact of automatic data binding.

In Angular, the model can be an object or array of objects of a TypeScript class that is like a POJO class (Plain Old Java Object). The HTML and CSS parts of the component represent the view. The TypeScript part of the component corresponds to the view model.

Types of Data Binding in Angular

1. **String Interpolation** This is a type of one-way data binding where a string contains some code (called template expression) enclosed in a set of double curly braces (`{{ code }}`). As the templates are reevaluated after every event; for performance reasons, there are many restrictions on the type of code that is allowed in a template expression. The common use is to get the value of a component variable / component property using `{{ variable }}` or `{{ variable.member }}` . This is a one-way binding; meaning that

only changes to the view model are propagated to the view, but changes to the view are not propagated to the view model

2. **Property Binding** Property Binding allows us to bind some property of a view element to a template expression. The property will change automatically when the value of the template expression changes. This is a one-way binding; meaning that only changes to the view model are propagated to the view, but changes to the view are not propagated to the view model. The syntax of property binding uses square brackets: `[property] = "template_expression"`. For example, `[disabled]="buttonDisabled"` where `buttonDisabled` is a variable in the component class.
3. **Event Binding** Event binding binds some event of a view element to code in the view model. This is a one-way binding; meaning that an event in the view can update the view model, but a change in the view model will not affect the view. The syntax of event binding uses round brackets: `(event)="template_expression"`. For example, `(click)='reloadBook();'` where `reloadBook()` is a method in the component class.
4. **Two-Way Data Binding** Two-way data binding is a combination of both property and event binding and it is a continuous synchronization of data between the view and the view model, which means that any changes to the view are propagated to the view model and any changes to the view model are propagated to the view immediately. Its syntax combines the syntax for property binding and event binding - it uses round brackets inside square brackets: `[(ngModel)]="book.title"`

Dependency Injection

Dependency injection is a technique whereby one object (the injector) provides the dependencies of another object (the client). A dependency is any object required by the client to perform. The dependency object is said to be injected into the client. Typically, a dependency provides some kind of service. Hence it is also called service. In Angular, a dependency is called `Injectable`. Dependency injection is one of the techniques for implementing the *Inversion of Control (IoC)* design pattern. Usually, the

dependency injection is carried out automatically by a framework.

- **Advantages**
  - **Separation of Concern** Each *concern* (functionality) in a software project is handled by a separate component or module, increasing the modularity and reducing coupling
  - **Increased Configurability** The services actually used by the client can be configured separately, often via a configuration file or code
  - **Ease of Unit Testing** Client code can be subjected to unit testing easily by using *mock* or *stub* implementations of the dependencies (services) instead of the real ones
  - **Simultaneous Independent Development** The client code and the services can be developed and unit-tested independently and simultaneously by different teams
- **Main Roles / Components**
  - **Client**
  - **Service (dependency)**
  - **Service Interface**
  - **Injector**

Examples

```
// Without dependency injection
export class BookGetComponent implements OnInit {

  books: Book[];
  bs : BookService;

  constructor(private bs: BookService) {

// The service class BookService is hard-coded and cannot be replaced
// class providing the same service

    bs = new BookService();

}

  ngOnInit() {
    this.bs
```

```
      .getBooks()
      .subscribe((data: Book[]) => {
        this.books = data;
    });
  }
}
```

```
// With dependency injection via the constructor
export class BookGetComponent implements OnInit {

  books: Book[];

// The BookService object bs is "injected" (provided) via the cosntruc
// When calling the cosntructor, it can be replaced by any other objec
// that implements the same interface

  constructor(private bs: BookService) { }

  ngOnInit() {
    this.bs
      .getBooks()
      .subscribe((data: Book[]) => {
        this.books = data;
    });
  }
}
```

```
// With dependency injection via a setter method
export class BookGetComponent implements OnInit {

  books: Book[];
  bs : BookService;

// The BookService object bs is "injected" (provided) by the setter me
// When calling the setter method, it can be replaced by any other obj
// that implements the same interface

    setBookService(private newBS: BookService) {
    this.bs = newBS;
  }

  constructor() { }

  ngOnInit() {
```

```
    this.bs
      .getBooks()
      .subscribe((data: Book[]) => {
        this.books = data;
    });
  }
}
```