SYLLABUS

Class – B.Com/BBA/BAJMC III Year

Subject: - Web Designing Techniques

Unit	Contents
UNIT – I	**Introduction to Canva:** Overview of Canva interface and features, Creating an account and setting up a profile, Understanding the different types of design formats. **Design Basics:** Using different fonts and understanding color combinations, Designing with grids and layouts. **Creating Designs:** Uploading Images, Using Canva's templates and elements, Creating presentations, posters, and flyers.
UNIT – II	**Introduction to AngularJS:** Overview of AngularJS and its architecture, Setting up an AngularJS development environment, Creating a basic AngularJS application, AngularsJS Expressions and Modules, AngularJS Directives **Controllers and Services:** AngularJS Contollers, Creating and using controllers, Understanding the role of services in AngularJS, Creating and using services to share data and functionality
UNIT – III	**Data Binding and Scope:** AngularJS Data Binding, Using data binding to update the view in real-time, AngularJS Scopes, AngularJS Filters & Services, AngularJS Http and Tables **Forms and Validations:** Creating and validating forms with AngularJS
UNIT – IV	**Routing and Navigation:** Creating routes and configuring navigation in AngularJS, Using the AngularJS router to navigate between pages, Creating nested routes and using route parameters **Advanced Topics:** AngularJS API, CSS and Animations, AngularJS Application
UNIT – V	**Introduction to Bootstrap:** Overview of Bootstrap and its features, Setting up the development environment, Understanding the Bootstrap grid system **Layout and Navigation:** Creating and using the navbar component, Creating and using the grid system for layout **Typography and Tables:** Using typography classes to style text, Creating and using tables **Forms and Buttons:** Creating and using forms, Styling forms with Bootstrap classes, Creating and using buttons and button groups

UNIT-1

Canva is a user-friendly graphic design platform that offers a variety of tools and features, making it accessible for both beginners and experienced designers. Here's an overview of its interface and key features:

Interface Overview

1. **Dashboard**:
   - **Home Screen**: Displays recent designs, templates, and a search bar for finding projects or templates.
   - **Side Panel**: Contains options for accessing templates, folders, and the brand kit.
2. **Design Workspace**:
   - **Canvas Area**: Central area where your design is created and edited.
   - **Toolbar**: Located at the top, includes options for undo/redo, file settings, and download/share features.
   - **Side Panel (left)**:
     - **Elements**: Access to shapes, lines, illustrations, and icons.
     - **Text**: Options for adding text boxes, headings, and pre-styled text layouts.
     - **Photos**: A library of free and premium stock images.
     - **Templates**: Pre-designed layouts for various types of content (social media, presentations, posters, etc.).
     - **Backgrounds**: Options to set background colors or images.
3. **Layers Panel**:
   - Allows you to manage different elements in your design, helping with layering and arrangement.
4. **Grid and Alignment Tools**:
   - Helps ensure your design elements are properly aligned and spaced.

Key Features

1. **Templates**:
   - A vast library of customizable templates for different formats (social media posts, presentations, flyers, etc.).
2. **Drag-and-Drop Functionality**:
   - Easily add elements, images, and text to your design with simple drag-and-drop actions.
3. **Collaboration Tools**:
   - Share designs with team members for real-time collaboration and feedback.
4. **Brand Kit**:
   - Customize your design with brand colors, logos, and fonts for consistent branding.
5. **Photo Editing Tools**:
   - Basic editing features, such as cropping, filtering, and adjusting brightness/contrast.
6. **Animations and Effects**:
   - Add animated elements and transitions for more dynamic presentations or social media posts.

7. **Export Options**:
   - Download designs in various formats (JPEG, PNG, PDF) and share directly to social media platforms.
8. **Mobile App**:
   - Access and edit designs on-the-go with Canva's mobile application.
9. **Stock Media Library**:
   - Extensive collection of images, videos, and audio files for use in your projects.
10. **Content Planner**:
    - A feature that allows scheduling social media posts directly from Canva.

Creating an Account

1. **Visit the Canva Website**:
   - Go to canva.com or download the Canva app from your device's app store.
2. **Sign Up**:
   - Click on the **Sign up** button. You have several options:
     - **Email**: Enter your email address and create a password.
     - **Google**: Sign up using your Google account.
     - **Facebook**: Use your Facebook account to sign up.
     - **Apple**: If you're on an Apple device, you can sign up using your Apple ID.
3. **Confirm Your Email** (if applicable):
   - If you signed up with your email, check your inbox for a confirmation email from Canva. Click the link to verify your email address.

Setting Up Your Profile

1. **Log In**:
   - Once your account is created and verified, log in to Canva.
2. **Access Your Profile**:
   - Click on your profile icon or your name in the top right corner of the screen.
3. **Edit Profile**:
   - In the dropdown menu, select **Account settings** or **Profile** to access your profile settings.
   - Here, you can update:
     - **Name**: Add or change your display name.
     - **Profile Picture**: Upload an image to personalize your account.
     - **Email**: Update your email address if necessary.
     - **Password**: Change your password for security.
4. **Brand Kit (if applicable)**:
   - If you're using Canva for business, you can set up a Brand Kit, where you can upload your logo, select brand colors, and choose brand fonts for consistency in your designs.
5. **Explore Preferences**:

- Adjust your design preferences, including language and notification settings.

Canva offers a wide variety of design formats to cater to different needs and platforms. Here's an overview of the most common design formats available in Canva:

1. Social Media Graphics

- **Instagram Posts**: Square format ideal for feeds.
- **Instagram Stories**: Vertical format designed for story sharing.
- **Facebook Posts**: Versatile sizes for sharing images and updates.
- **Twitter Posts**: Specific dimensions for tweets.
- **LinkedIn Banners**: Professional graphics for your LinkedIn profile.

2. Presentations

- **Standard Presentation Slides**: Perfect for business or educational presentations.
- **Pitch Decks**: Designed for pitching ideas to stakeholders.
- **Webinar Slides**: Tailored for online presentations.

3. Print Materials

- **Flyers**: Promotional materials for events, sales, etc.
- **Brochures**: Multi-panel designs for detailed information.
- **Postcards**: For direct mail marketing or personal messages.
- **Business Cards**: Professional cards to share contact information.

4. Infographics

- **Statistical Infographics**: Visual representations of data and statistics.
- **Process Infographics**: Illustrations of workflows or step-by-step processes.
- **Timelines**: Showcasing historical events or project milestones.

5. Marketing Materials

- **Newsletters**: Email-friendly layouts for updates and announcements.
- **Ebooks**: Multi-page documents for sharing knowledge.
- **Social Media Ads**: Specific sizes for promotional content.

6. Event Materials

- **Invitations**: For parties, weddings, and other events.
- **Event Posters**: Eye-catching designs for promoting events.
- **Menus**: Stylish layouts for restaurants or gatherings.

7. Web Graphics

- **Blog Banners**: Visuals for blog posts or articles.
- **Website Headers**: Customizable headers for website branding.
- **YouTube Thumbnails**: Engaging visuals for video content.

8. Miscellaneous Formats

- **Certificates**: For awards or recognition.
- **Calendars**: Monthly or yearly layouts for planning.
- **T-shirt Designs**: Custom designs for apparel

Using Different Fonts in Canva

1. **Accessing Fonts**:
   - In the design workspace, click on any text box to access the text editing options.
   - In the toolbar above, you'll see a dropdown menu for fonts.
2. **Choosing Fonts**:
   - **Font Categories**: Canva categorizes fonts into various types (e.g., serif, sans-serif, display, handwritten). Explore these to find the style that fits your design.
   - **Filters**: Use filters to find fonts based on usage, such as "Free" or "Pro" (for Canva Pro users).
   - **Brand Fonts**: If you've set up a Brand Kit, your selected brand fonts will be easily accessible.
3. **Combining Fonts**:
   - **Pairing**: Combine different font types for contrast. A common approach is to pair a serif font for headings with a sans-serif font for body text.
   - **Hierarchy**: Use size and weight (bold, light) to establish a visual hierarchy, making headings stand out from the body text.
4. **Text Effects**:
   - Enhance your text with effects like shadows, outlines, or glows, which can add depth and emphasis.

Understanding Color Combinations

1. **Color Picker**:
   - Click on any element to bring up the color options. You can choose from preset colors or use the color picker to select a custom color.
2. **Color Theory Basics**:

- **Complementary Colors**: Colors opposite each other on the color wheel (e.g., blue and orange). They create high contrast and can be visually striking.
- **Analogous Colors**: Colors next to each other on the wheel (e.g., blue, blue-green, green). They create a harmonious look.
- **Triadic Colors**: Three colors evenly spaced on the color wheel (e.g., red, blue, yellow). This combination is vibrant and balanced.

3. **Color Palettes**:
   - Canva offers built-in color palettes to help you choose combinations that work well together. You can find these under the "Styles" tab in the left panel.
   - Alternatively, use tools like Adobe Color or Coolors to create and save custom palettes.
4. **Accessibility Considerations**:
   - Ensure sufficient contrast between text and background colors for readability. Use online contrast checkers to test your combinations.
5. **Brand Colors**:
   - If you have a Brand Kit, add your brand colors for consistency across all your designs. This helps in maintaining a cohesive visual identity.

Understanding Grids in Canva

1. **What Are Grids?**
   - Grids are a framework that helps structure your design by dividing it into equal sections. They provide guidelines for aligning and organizing elements consistently.
2. **Accessing Grids**:
   - To add a grid to your design, go to the **Elements** tab on the left panel and search for "grids."
   - You'll find different grid layouts (e.g., 2x2, 3x3) that you can drag and drop into your design.
3. **Using Grids**:
   - **Photo Grids**: Perfect for showcasing multiple images, such as a collage. Drag images into the grid sections, and they will automatically fit.
   - **Content Organization**: Use grids to neatly arrange text and graphics, helping to create a balanced layout.

Using Layouts in Canva

1. **Pre-Made Layouts**:
   - Canva offers a variety of pre-made layouts for different types of designs (social media posts, presentations, flyers, etc.).
   - Browse templates in the **Templates** section or under specific categories, and select one that fits your needs.
2. **Customizing Layouts**:

- Once you select a layout, you can customize it by changing colors, fonts, images, and other elements to match your branding or style.

3. **Creating Your Own Layouts**:
   - Start with a blank canvas and use grids to structure your design.
   - Use the **Rulers and Guides** feature by enabling it in the **File** menu. This helps you align elements more precisely.

Steps to Upload Images in Canva

1. **Log In to Canva**:
   - Access your account on the Canva website or open the Canva app.
2. **Open a Design**:
   - Start a new project or open an existing design where you want to add images.
3. **Access the Uploads Tab**:
   - On the left panel, click on the **Uploads** tab. This is where you can manage your uploaded files.
4. **Upload Images**:
   - Click the **Upload media** button. This will open a file dialog.
   - Select the images you want to upload from your device (you can select multiple files at once).
   - You can also drag and drop images directly from your computer into the Canva interface.
5. **Supported Formats**:
   - Canva supports various file formats, including JPEG, PNG, SVG, and GIF. Ensure your images are in one of these formats.
6. **Using Uploaded Images**:
   - Once uploaded, your images will appear in the Uploads panel. Simply drag and drop them into your design.
   - You can resize and position the images as needed.
7. **Organizing Uploaded Images**:
   - To keep your uploads organized, you can create folders within the Uploads tab. Click on the **Folders** option and create new ones to categorize your images.
8. **Deleting Uploaded Images**:
   - If you need to remove an image, hover over it in the Uploads panel, click the three dots (more options), and select **Delete**.

Using Canva's Templates

1. **Accessing Templates**:
   - On the Canva homepage, you'll see a search bar and categories for various design types (e.g., social media, presentations, flyers).
   - You can either browse through categories or use the search bar to find a specific template.
2. **Selecting a Template**:

- Once you find a template you like, click on it to preview.
- If it fits your needs, click the **Use this template** button to open it in the design workspace.

3. **Customizing Templates**:
   - **Text**: Click on any text box to edit the content. You can change the font, size, color, and spacing.
   - **Images**: Replace any placeholder images with your own by dragging your uploaded images into the template or selecting from Canva's stock library.
   - **Colors**: Change the colors of elements to match your brand or personal style by clicking on the element and selecting a new color.
4. **Resizing Templates**:
   - If you need to change the size of your design, you can use the **Resize** feature (available in Canva Pro) to adjust it for different formats while maintaining the layout.
5. **Saving Custom Templates**:
   - If you customize a template extensively, you can save it as a new template by clicking **File** > **Make a copy**, allowing you to reuse your design.

Using Canva's Elements

1. **Accessing Elements**:
   - Click on the **Elements** tab in the left panel. Here, you'll find a wide variety of design elements, including:
     - **Shapes**: Basic geometric shapes for framing and layering.
     - **Lines and Dividers**: For creating sections or highlighting content.
     - **Icons**: A vast library of icons for visual representation of ideas.
     - **Illustrations**: Artistic graphics that can enhance your design.
     - **Photos**: A collection of stock images available for use.
2. **Searching for Elements**:
   - Use the search bar within the Elements tab to find specific graphics or icons by keywords.
3. **Adding Elements to Your Design**:
   - Drag and drop any element into your design workspace. You can resize, rotate, and position them as needed.
4. **Customizing Elements**:
   - Click on any element to change its color, size, or transparency. You can also group elements together for easier movement and alignment.
5. **Layering Elements**:
   - Use the layering feature to arrange elements. Right-click on an element to access options for bringing it forward or sending it backward in the design.

Creating presentations, posters, and flyers in Canva is easy and efficient thanks to its user-friendly interface and extensive library of templates and design elements. Here's a step-by-step guide for each type of design:

Creating Presentations

1. **Access the Presentation Format**:
   - From the Canva homepage, select **Presentations** from the template categories or search for "presentation" in the search bar.
2. **Choose a Template**:
   - Browse through the available presentation templates and select one that fits your theme or topic.
3. **Customize Your Slides**:
   - **Add Content**: Click on the text boxes to edit titles, subtitles, and body text. Add images by dragging them into placeholders or uploading your own.
   - **Transitions and Animations**: Click on the slide, then use the "Animate" option in the toolbar to add animations to text and images.
   - **Add Slides**: Use the **+ Add a new page** button to include more slides. You can duplicate existing slides for consistency.
4. **Presenting Your Slides**:
   - Click on the **Present** button in the top right corner to view your presentation in full-screen mode.
5. **Download or Share**:
   - Once complete, download your presentation as a PDF, PNG, or PPTX file, or share it directly via link or email.

Creating Posters

1. **Select the Poster Format**:
   - Go to the Canva homepage and choose **Posters** from the template categories or use the search bar.
2. **Choose a Poster Template**:
   - Browse the selection of poster templates and select one that aligns with your message or event.
3. **Customize Your Design**:
   - **Text and Images**: Edit the text by clicking on the boxes, and replace images as needed. Drag and drop images from your uploads or Canva's library.
   - **Colors and Fonts**: Adjust colors and fonts to match your branding or theme.
4. **Add Elements**:
   - Use the **Elements** tab to include shapes, icons, or illustrations that enhance your design.
5. **Download or Print**:
   - After finishing your design, download it in high-quality PDF format for printing or as an image file for digital sharing.

Creating Flyers

1. **Start with the Flyer Template**:
   - From the Canva homepage, navigate to **Flyers** in the template categories or search for "flyer."

2. **Select a Flyer Template**:
   - Choose a flyer template that suits your purpose (event, promotion, etc.).
3. **Customize Your Flyer**:
   - **Text and Graphics**: Edit the text to include relevant details and customize images to make your flyer visually appealing.
   - **Layout Adjustments**: Use grids or align elements to ensure a clean and organized layout.
4. **Add Call-to-Action**:
   - Make sure to include a clear call-to-action, such as RSVP details or a website link, to encourage engagement.
5. **Download or Print**:
   - Save your flyer in a format suitable for digital distribution or print.

UNIT-2

AngularJS is a very powerful JavaScript library. It is used in Single Page Application (SPA) projects. It extends HTML DOM with additional attributes and makes it more responsive to user actions. AngularJS is open source, completely free, and used by thousands of developers around the world. It is licensed under the Apache license version 2.0.

AngularJS is an open-source web application framework. It was originally developed in 2009 by Misko Hevery and Adam Abrons. It is now maintained by Google. Its latest version is 1.2.21.

AngularJS is a structural framework for dynamic web applications. It lets you

- use HTML as your template language and
- you extend HTML's syntax to express your application components clearly.
- Its data binding and dependency injection eliminate much of the code you currently have to write.
- it all happens within the browser, making it an ideal partner with any server technology.

General Features

The general features of AngularJS are as follows:

- AngularJS is a efficient framework that can create Rich Internet Applications (RIA).
- AngularJS provides developers an options to write client side applications using JavaScript in a clean Model View Controller (MVC) way.
- Applications written in AngularJS are cross-browser compliant. AngularJS automatically handles JavaScript code suitable for each browser.
- AngularJS is open source, completely free, and used by thousands of developers around the world. It is licensed under the Apache license version 2.0.

Overall, AngularJS is a framework to build large scale, high-performance, and easy- to-maintain web applications.

Core Features

The core features of AngularJS are as follows:

- **Data-binding:** It is the automatic synchronization of data between model and view components.
- **Scope:** These are objects that refer to the model. They act as a glue between controller and view.
- **Controller:** These are JavaScript functions bound to a particular scope.
- **Services:** AngularJS comes with several built-in services such as $http to make a XMLHttpRequests. These are singleton objects which are instantiated only once in app.

- **Filters:** These select a subset of items from an array and returns a new array.
- **Directives:** Directives are markers on DOM elements such as elements, attributes, css, and more. These can be used to create custom HTML tags that serve as new, custom widgets. AngularJS has built-in directives such as ngBind, ngModel, etc.

- **Templates:** These are the rendered view with information from the controller and model. These can be a single file (such as index.html) or multiple views in one page using *partials*.

- **Routing:** It is concept of switching views.

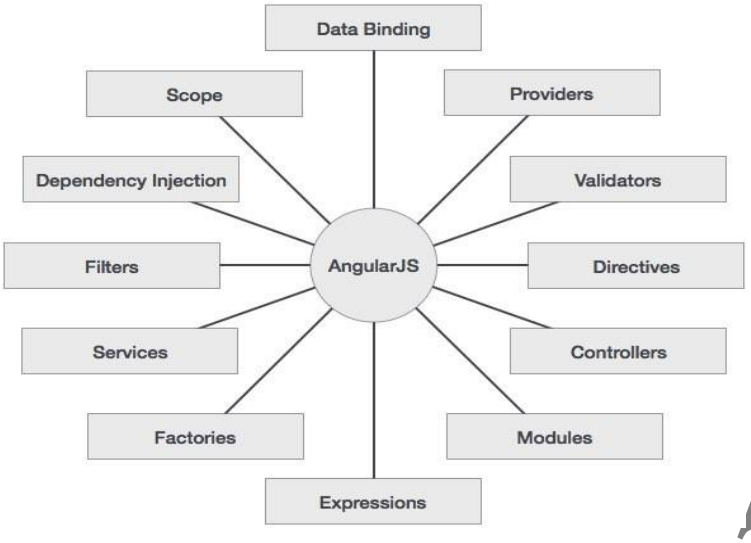
- **Model View Whatever:** MVW is a design pattern for dividing an application into different parts called Model, View, and Controller, each with distinct responsibilities. AngularJS does not implement MVC in the traditional sense, but rather something closer to MVVM (Model-View-ViewModel). The Angular JS team refers it humorously as Model View Whatever.

- **Deep Linking:** Deep linking allows to encode the state of application in the URL so that it can be bookmarked. The application can then be restored from the URL to the same state.

- **Dependency Injection:** AngularJS has a built-in dependency injection subsystem that helps the developer to create, understand, and test the applications easily.

Concepts

The following diagram depicts some important parts of AngularJS which we will discuss in detail in the subsequent chapters.



Advantages of AngularJS

The advantages of AngularJS are:

- It provides the capability to create Single Page Application in a very clean and maintainable way.
- It provides data binding capability to HTML. Thus, it gives user a rich and responsive experience.
- AngularJS code is unit testable.
- AngularJS uses dependency injection and make use of separation of concerns.
- AngularJS provides reusable components.
- With AngularJS, the developers can achieve more functionality with short code.
- In AngularJS, views are pure html pages, and controllers written in JavaScript do the business processing.

On the top of everything, AngularJS applications can run on all major browsers and smart phones, including Android and iOS based phones/tablets.

Disadvantages of AngularJS

Though AngularJS comes with a lot of merits, here are some points of concern:

- **Not secure** : Being JavaScript only framework, application written in AngularJS are not safe. Server side authentication and authorization is must to keep an application secure.

- **Not degradable**: If the user of your application disables JavaScript, then nothing would be visible, except the basic page.

AngularJS Directives

The AngularJS framework can be divided into three major parts:

- **ng-app** : This directive defines and links an AngularJS application to HTML.

- **ng-model** : This directive binds the values of AngularJS application data to HTML input controls.

- **ng-bind** : This directive binds the AngularJS application data to HTML tags.

MVC Architecture

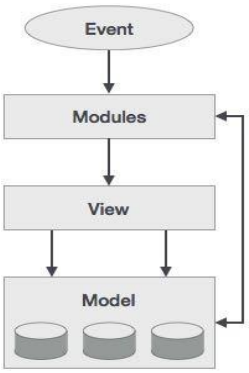
Model **V**iew **C**ontroller or MVC as it is popularly called, is a software design pattern for developing web applications. A Model View Controller pattern is made up of the following three parts:

- **Model** - It is the lowest level of the pattern responsible for maintaining data.

- **View** - It is responsible for displaying all or a portion of the data to the user.

- **Controller** - It is a software Code that controls the interactions between the Model and View.

MVC is popular because it isolates the application logic from the user interface layer and supports separation of concerns. The controller receives all requests for the application and then works with the model to prepare any data needed by the view. The view then uses the data prepared by the controller to generate a final presentable response. The MVC abstraction can be graphically represented as follows.



The Model

The model is responsible for managing application data. It responds to the request from view and to the instructions from controller to update itself.

The View

A presentation of data in a particular format, triggered by the controller's decision to present the data. They are script-based template systems such as JSP, ASP, PHP and very easy to integrate with AJAX technology.

The Controller

The controller responds to user input and performs interactions on the data model objects. The controller receives input, validates it, and then performs business operations that modify the state of the data model.

AngularJS is a MVC based framework. In the coming chapters, we will see how AngularJS uses MVC methodology.

First Application

Before creating actual *Hello World !* application using AngularJS, let us see the parts of a AngularJS application. An AngularJS application consists of following three important parts:

- **ng-app** : This directive defines and links an AngularJS application to HTML.
- **ng-model** : This directive binds the values of AngularJS application data to HTML input controls.
- **ng-bind** : This directive binds the AngularJS Application data to HTML tags.

Creating AngularJS Application

Step 1: Load framework

Being a pure JavaScript framework, it can be added using <Script> tag.

```
<script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.2.15/angular.min.js">
</script>
```

Step 2: Define AngularJS application using *ng-app* directive.

```
<div ng-app="">
...
</div>
```

Step 3: Define a model name using *ng-model* directive.

```
<p>Enter your Name: <input type="text" ng-model="name"></p>
```

```
<p>Hello <span ng-bind="name"></span>!</p>
```

Step 4: Bind the value of above model defined using *ng-bind* directive.

Executing AngularJS Application

Use the above-mentioned three steps in an HTML page.

testAngularJS.htm

```
<html>
<title>AngularJS First Application</title>
<body>
<h1>Sample Application</h1>
<div ng-app="">
   <p>Enter your Name: <input type="text" ng-model="name"></p>
   <p>Hello <span ng-bind="name"></span>!</p>
</div>
<script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.2.15/angular.min.js"></
/script>
</body>
```

Output

Open the file *testAngularJS.htm* in a web browser. Enter your name and see the result.

How AngularJS Integrates with HTML

- The ng-app directive indicates the start of AngularJS application.
- The ng-model directive creates a model variable named *name*, which can be used with the HTML page and within the div having ng-app directive.
- The ng-bind then uses the *name* model to be displayed in the HTML tag whenever user enters input in the text box.

- Closing tag indicates the end of AngularJS application

Overview of AngularJS Controllers

1. **Definition**:
   - A controller in AngularJS is defined using a JavaScript function. It initializes data and defines functions that can be called from the view.
2. **Usage**:
   - Controllers are defined using the `.controller` method of an AngularJS module. They are associated with specific views via the `ng-controller` directive.
3. **Scope**:
   - The `$scope` object is a special object in AngularJS that acts as the glue between the controller and the view. Properties added to the `$scope` object are accessible in the view.

Creating a Controller

Here's a basic example of how to create and use a controller in AngularJS:

```
javascript
// Define the AngularJS application module
var app = angular.module('myApp', []);

// Define a controller
app.controller('MyController', function($scope) {
    // Initialize some data
    $scope.greeting = "Hello, World!";

    // Define a function
    $scope.updateGreeting = function(newGreeting) {
        $scope.greeting = newGreeting;
    };
});
```

Using the Controller in the View

Once you've defined your controller, you can use it in your HTML view:

```
html
<!DOCTYPE html>
```

```
<html ng-app="myApp">
<head>
    <title>AngularJS Example</title>
    <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></
script>
    <script src="app.js"></script> <!-- Your AngularJS script -->
</head>
<body ng-controller="MyController">
    <h1>{{ greeting }}</h1>
    <input type="text" ng-model="newGreeting" placeholder="Enter new
greeting">
    <button ng-click="updateGreeting(newGreeting)">Update Greeting</button>
</body>
</html>
```

Key Concepts

1. **Two-Way Data Binding**:
   - AngularJS uses two-way data binding to synchronize the data between the model (the `$scope`) and the view (HTML). When the model changes, the view reflects the change and vice versa.
2. **Dependency Injection**:
   - AngularJS supports dependency injection, which allows you to inject services (like `$scope`, `$http`, etc.) into your controller functions for better modularity and testability.
3. **Controller As Syntax**:
   - For a more modern approach, you can use the "Controller As" syntax, which helps avoid issues with `$scope` by using `this` to refer to the controller instance.

```
javascript
app.controller('MyController', function() {
    var vm = this; // 'vm' stands for ViewModel
    vm.greeting = "Hello, World!";
    vm.updateGreeting = function(newGreeting) {
        vm.greeting = newGreeting;
    };
});
```

And in the view:

```
html
<body ng-controller="MyController as ctrl">
    <h1>{{ ctrl.greeting }}</h1>
```

```
    <input type="text" ng-model="ctrl.newGreeting" placeholder="Enter
new greeting">
    <button ng-click="ctrl.updateGreeting(ctrl.newGreeting)">Update
Greeting</button>
</body>
```

Setting Up Your AngularJS Application

First, ensure you have included AngularJS in your project. You can do this by adding the AngularJS library via a CDN link in your HTML file.

```
html
<!DOCTYPE html>
<html lang="en" ng-app="myApp">
<head>
    <meta charset="UTF-8">
    <title>AngularJS Example</title>
    <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></
script>
    <script src="app.js"></script> <!-- Your AngularJS script -->
</head>
<body>
    <!-- Your content will go here -->
</body>
</html>
```

2. Creating an AngularJS Module

In your `app.js` file, define an AngularJS module. A module is a container for different parts of your application, such as controllers, services, and directives.

```
javascript
var app = angular.module('myApp', []);
```

3. Creating a Controller

Now, create a controller within your module. Controllers are defined using the `.controller()` method of the module.

```
javascript
app.controller('MyController', function($scope) {
    // Initialize some data
    $scope.message = "Hello, AngularJS!";
```

```
    // Define a function to update the message
    $scope.updateMessage = function(newMessage) {
        $scope.message = newMessage;
    };
});
```

4. Using the Controller in the View

Next, you need to use the controller in your HTML view. This is done using the `ng-controller` directive.

```
html
<body ng-controller="MyController">
    <h1>{{ message }}</h1>
    <input type="text" ng-model="newMessage" placeholder="Enter new message">
    <button ng-click="updateMessage(newMessage)">Update Message</button>
</body>
```

5. Complete Example

Here's a complete example of an AngularJS application with a controller:

HTML (index.html):

```
html
<!DOCTYPE html>
<html lang="en" ng-app="myApp">
<head>
    <meta charset="UTF-8">
    <title>AngularJS Example</title>
    <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></
script>
    <script src="app.js"></script>
</head>
<body ng-controller="MyController">
    <h1>{{ message }}</h1>
    <input type="text" ng-model="newMessage" placeholder="Enter new message">
    <button ng-click="updateMessage(newMessage)">Update Message</button>
</body>
</html>
```

JavaScript (app.js):

```
javascript
var app = angular.module('myApp', []);

app.controller('MyController', function($scope) {
    // Initialize data
    $scope.message = "Hello, AngularJS!";

    // Function to update the message
    $scope.updateMessage = function(newMessage) {
        $scope.message = newMessage;
    };
});
```

6. Key Concepts

- **$scope**: The `$scope` object provides the context for your view. Properties and methods added to `$scope` are accessible in the view.
- **Two-Way Data Binding**: Changes made in the view automatically reflect in the `$scope` and vice versa.
- **Dependency Injection**: AngularJS uses dependency injection to make services like `$scope` available in your controllers.

What Are Services?

1. **Definition**:
   - Services are singleton objects that are instantiated once per application. They encapsulate business logic and provide functionalities that can be shared across controllers, directives, and other services.
2. **Purpose**:
   - They promote modularity and separation of concerns by allowing you to encapsulate reusable code, which improves maintainability and testability.

Key Characteristics of Services

1. **Singleton**:
   - A service is created only once and shared across the application. If you modify a service's state in one part of the application, that change is reflected everywhere else the service is used.
2. **Dependency Injection**:
   - AngularJS uses dependency injection to provide services to components that need them, enhancing modularity and making unit testing easier.

Types of Services

In AngularJS, you can create services using different methods:

1. **Service**:
   - A constructor function that can be instantiated. Use the `.service()` method to create a service.

```
javascript
app.service('MyService', function() {
    this.message = "Hello from MyService!";
    this.getMessage = function() {
        return this.message;
    };
});
```

2. **Factory**:
   - A factory function that returns an object. Use the `.factory()` method for more complex services or when you want to return an object with specific methods.

```
javascript
app.factory('MyFactory', function() {
    var service = {};
    service.message = "Hello from MyFactory!";
    service.getMessage = function() {
        return service.message;
    };
    return service;
});
```

3. **Provider**:
   - The most flexible way to create a service. It allows configuration before the service is instantiated. Use `.provider()` when you need to set up a service with specific configurations.

```
javascript
app.provider('MyProvider', function() {
    var message = "Default Message";
    this.setMessage = function(newMessage) {
        message = newMessage;
    };
    this.$get = function() {
        return {
            getMessage: function() {
```

```
                return message;
            }
        };
    };
});
```

Using Services in Controllers

Once a service is defined, it can be injected into controllers or other services. This is how you can access the service's properties and methods.

```
javascript
app.controller('MyController', function($scope, MyService) {
    $scope.message = MyService.getMessage();
    $scope.updateMessage = function(newMessage) {
        MyService.message = newMessage; // Assuming MyService.message is
public
    };
});
```

Example of a Complete AngularJS Application

Here's an example to illustrate the use of a service in an AngularJS application:

HTML (index.html):

```
html
<!DOCTYPE html>
<html lang="en" ng-app="myApp">
<head>
    <meta charset="UTF-8">
    <title>AngularJS Services Example</title>
    <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></
script>
    <script src="app.js"></script>
</head>
<body ng-controller="MyController">
    <h1>{{ message }}</h1>
    <input type="text" ng-model="newMessage" placeholder="Enter new message">
    <button ng-click="updateMessage(newMessage)">Update Message</button>
</body>
</html>
```

JavaScript (app.js):

```
javascript
var app = angular.module('myApp', []);

// Create a service
app.service('MyService', function() {
    this.message = "Hello from MyService!";
    this.getMessage = function() {
        return this.message;
    };
});

// Create a controller
app.controller('MyController', function($scope, MyService) {
    // Access service data
    $scope.message = MyService.getMessage();

    // Update message using service
    $scope.updateMessage = function(newMessage) {
        MyService.message = newMessage;
        $scope.message = MyService.getMessage();
    };
});
```

Advantages of Using Services

1. **Code Reusability**:
   - Services allow you to write common functionalities once and reuse them across the application, reducing code duplication.
2. **Separation of Concerns**:
   - By moving business logic out of controllers and into services, you create cleaner, more maintainable code.
3. **Testability**:
   - Services can be easily unit tested, which enhances the reliability of your application.
4. **Encapsulation**:
   - Services encapsulate data and behavior, making it easier to manage and change your code without affecting other parts of the application.

Services in AngularJS are singleton objects that encapsulate reusable functionality, making them accessible from different parts of your application. They help maintain a clear separation of concerns, promoting code reusability and easier maintenance.

Creating a Service

You can create a service using either the `.service()`, `.factory()`, or `.provider()` methods. Here's how to create a simple service using the `.service()` method.

Example: Simple Data Service

1. Setting Up Your AngularJS Module

First, create a module for your application.

```
javascript
var app = angular.module('myApp', []);
```

2. Creating a Service

Define a service to manage shared data.

```
javascript
app.service('DataService', function() {
    var self = this; // Reference to the service instance

    // Shared data
    self.data = {
        message: "Hello from DataService!"
    };

    // Function to get the data
    self.getMessage = function() {
        return self.data.message;
    };

    // Function to update the data
    self.updateMessage = function(newMessage) {
        self.data.message = newMessage;
    };
});
```

Using the Service in Controllers

Now, you can inject this service into your controllers to access or modify the shared data.

Example: Controller Implementation

3. Creating a Controller

Define a controller that uses the `DataService`.

```
javascript
app.controller('MainController', function($scope, DataService) {
    // Accessing shared data from the service
    $scope.message = DataService.getMessage();

    // Function to update the message using the service
    $scope.updateMessage = function(newMessage) {
        DataService.updateMessage(newMessage);
        $scope.message = DataService.getMessage(); // Update the view
    };
});
```

Complete Example

Here's how everything fits together in a complete AngularJS application.

HTML (index.html):

```
html
<!DOCTYPE html>
<html lang="en" ng-app="myApp">
<head>
    <meta charset="UTF-8">
    <title>AngularJS Data Service Example</title>
    <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></
script>
    <script src="app.js"></script> <!-- Your AngularJS script -->
</head>
<body ng-controller="MainController">
    <h1>{{ message }}</h1>
    <input type="text" ng-model="newMessage" placeholder="Enter new message">
    <button ng-click="updateMessage(newMessage)">Update Message</button>
</body>
</html>
```

JavaScript (app.js):

```
javascript
var app = angular.module('myApp', []);

// Create a service
app.service('DataService', function() {
    var self = this; // Reference to the service instance
```

```
    // Shared data
    self.data = {
        message: "Hello from DataService!"
    };

    // Function to get the data
    self.getMessage = function() {
        return self.data.message;
    };

    // Function to update the data
    self.updateMessage = function(newMessage) {
        self.data.message = newMessage;
    };
});

// Create a controller
app.controller('MainController', function($scope, DataService) {
    // Accessing shared data from the service
    $scope.message = DataService.getMessage();

    // Function to update the message using the service
    $scope.updateMessage = function(newMessage) {
        DataService.updateMessage(newMessage);
        $scope.message = DataService.getMessage(); // Update the view
    };
});
```

Advantages of Using Services

1. **Data Sharing**: Services allow different components (like controllers) to share data seamlessly without needing to pass data directly between them.
2. **Separation of Concerns**: Business logic can be kept separate from the user interface logic, making the code cleaner and easier to maintain.
3. **Reusability**: You can reuse the same service across multiple controllers and components, reducing code duplication.
4. **Testability**: Services can be easily tested in isolation, making unit testing more straightforward.

UNIT-3

AngularJS, data binding

In AngularJS, **data binding** is a powerful feature that allows automatic synchronization of data between the **model** (JavaScript objects) and the **view** (HTML elements). This means that when the data in the model changes, the view automatically reflects those changes, and vice versa, without needing additional code to update the DOM.

Types of Data Binding in AngularJS

AngularJS supports several types of data binding to facilitate communication between the model and the view:

1. **One-way Data Binding**
2. **Two-way Data Binding**

1. Two-Way Data Binding

Two-way data binding is one of the core features of AngularJS and allows the **model** and **view** to be automatically synchronized. If the data in the model changes, the view is updated automatically, and if the view (UI) changes (for example, if a user types in an input field), the model is updated as well.

This is achieved by using **ng-model** in AngularJS, which binds an HTML input element to a model property.

Example of Two-Way Data Binding

In the following example, we will demonstrate two-way data binding using an input field:

JavaScript (Controller)

```
javascript
var app = angular.module('myApp', []);

app.controller('MainController', function($scope) {
    // Model property bound to the view
    $scope.name = "John Doe";
});
```

HTML (View)

```
html
<!DOCTYPE html>
<html ng-app="myApp">
<head>
    <title>AngularJS Two-Way Data Binding Example</title>
</head>
<body ng-controller="MainController">
    <h2>Hello, {{ name }}!</h2>
    <input type="text" ng-model="name">
    <p>Your name is: {{ name }}</p>
</body>
</html>
```

Breakdown:

- The `{{ name }}` syntax is **one-way data binding** that outputs the value of the `name` property from the model to the view.
- The `ng-model="name"` creates **two-way data binding** between the input field and the `name` property. As the user types in the input box, the `name` property in the model gets updated automatically.
- The changes in the input field are reflected immediately in both the `h2` and the `p` tag without the need for any manual DOM updates.

How It Works:

- The **`ng-model`** directive binds the `name` property from the AngularJS model to the input element. This means that if the model changes (e.g., through user input), the view updates, and vice versa.
- Whenever you update the `name` in the JavaScript code, the input field value will automatically reflect this change, and if the user types into the input field, the `name` property in the model will be updated.

2. One-Way Data Binding

One-way data binding refers to binding data from the **model** to the **view**. In this case, any changes made to the model will automatically update the view, but changes made in the view will not affect the model.

This is usually achieved by using **{{expression}}** or the `ng-bind` directive.

Example of One-Way Data Binding

JavaScript (Controller)

```
javascript
var app = angular.module('myApp', []);

app.controller('MainController', function($scope) {
    // Model property bound to the view
    $scope.message = "Welcome to AngularJS!";
});
```

HTML (View)

```
html
<!DOCTYPE html>
<html ng-app="myApp">
<head>
    <title>AngularJS One-Way Data Binding Example</title>
</head>
<body ng-controller="MainController">
    <h2>{{ message }}</h2> <!-- One-way data binding -->
</body>
</html>
```

Breakdown:

- `{{ message }}` binds the value of the `message` model property to the `<h2>` tag. Whenever the `message` changes in the controller, the view (HTML) is updated.
- However, any changes made in the view (such as editing the content inside the `<h2>` tag) will not change the value of the `message` property in the model.

3. One-Way Data Binding with ng-bind

Instead of using `{{ expression }}` syntax, you can use `ng-bind` to bind data to HTML elements.

Example of Using `ng-bind`

HTML (View)

```
html
<!DOCTYPE html>
```

```
<html ng-app="myApp">
<head>
    <title>AngularJS ng-bind Example</title>
</head>
<body ng-controller="MainController">
    <h2 ng-bind="message"></h2> <!-- One-way data binding using ng-bind -->
</body>
</html>
```

In this case, `ng-bind="message"` binds the `message` property to the content of the `<h2>` element. The result is the same as `{{ message }}`, but `ng-bind` has the advantage of preventing the "flash of uncompiled content" (i.e., the page is briefly rendered without the data, and then the data is injected).

4. Data Binding with ng-model for Forms

You can use two-way data binding to capture user input in forms and bind it directly to model properties. This makes it easy to collect form data and manage the state of the form.

Example of Data Binding in Forms

JavaScript (Controller)

```
javascript
var app = angular.module('myApp', []);

app.controller('FormController', function($scope) {
    $scope.user = {
        name: '',
        email: ''
    };
});
```

HTML (View)

```
html
<!DOCTYPE html>
<html ng-app="myApp">
<head>
    <title>AngularJS Form Binding</title>
</head>
<body ng-controller="FormController">
    <form name="userForm">
        <label>Name:</label>
```

```
        <input type="text" ng-model="user.name" placeholder="Enter your
name">

        <label>Email:</label>
        <input type="email" ng-model="user.email" placeholder="Enter your
email">

        <h3>Your Name: {{ user.name }}</h3>
        <h3>Your Email: {{ user.email }}</h3>
    </form>
</body>
</html>
```

How It Works:

- The `ng-model="user.name"` and `ng-model="user.email"` directives bind the input fields to the `user` object properties. As the user types into the inputs, the `user.name` and `user.email` properties get updated automatically.
- This is two-way data binding, meaning that both the view (HTML form) and the model (JavaScript object) are kept in sync.

AngularJS Scopes,

In AngularJS, **scopes** are an essential concept for linking the controller and the view, and they play a central role in the data-binding process. The **$scope** object is used to store the data and functions that are accessible in a particular view (HTML) controlled by a controller.

Understanding **scope** is key to grasping how AngularJS manages data flow and interacts with the view, especially when you're working with controllers, directives, or even services.

What is `$scope` in AngularJS?

- The `$scope` object is an **execution context** for expressions.
- It provides a bridge between the controller (JavaScript) and the view (HTML).
- Variables defined in `$scope` are available for **data binding** in the view, and changes to these variables are reflected in the view automatically (two-way data binding).

When you create a controller in AngularJS, `$scope` is injected into the controller function by AngularJS. It is used to expose variables and methods to the view.

How `$scope` Works in AngularJS

In an AngularJS controller, the `$scope` object is used to share data and functions between the controller and the view. Anything you assign to `$scope` becomes available in the view, and the view can also update the `$scope` model (in the case of two-way binding).

Example of `$scope` in a Controller

Controller:

```
javascript
var app = angular.module('myApp', []);

app.controller('MainController', function($scope) {
    // Data is stored on the $scope object
    $scope.name = 'John Doe';

    // A function attached to $scope
    $scope.greet = function() {
        alert('Hello, ' + $scope.name);
    };
});
```

View:

```
html
<!DOCTYPE html>
<html ng-app="myApp">
<head>
    <title>AngularJS Scopes Example</title>
</head>
<body ng-controller="MainController">
    <h1>Hello, {{ name }}!</h1> <!-- Bind data from $scope to the view -->
    <button ng-click="greet()">Greet</button> <!-- Call a function from
$scope -->
</body>
</html>
```

Breakdown:

- The `name` property is bound to the view using `{{ name }}`.
- The `greet()` function, which is also defined on `$scope`, is called when the user clicks the button. It shows an alert that uses the `name` value from `$scope`.

Types of Scopes in AngularJS

In AngularJS, scopes can be divided into two main types:

1. **Controller Scope**
2. **Child Scope**

1. Controller Scope (Default Scope)

When you define a controller using `$scope`, you're typically working with the **controller scope**. The data and methods attached to `$scope` in a controller are available only to that controller's view (i.e., the view in which the controller is applied). These properties and functions can be accessed directly in the HTML template.

Example:

```
javascript
app.controller('MainController', function($scope) {
    $scope.message = "Hello from MainController!";
});
```

HTML (View):

```
html
<h1>{{ message }}</h1>
```

- In this case, `$scope.message` is available in the view associated with `MainController`.

2. Child Scope (Inheritance of Scope)

In AngularJS, scopes are hierarchical, meaning that **child controllers inherit the scope of their parent controllers**. This is important because a child scope can access properties from its parent scope, but the parent scope cannot directly access data from the child scope.

- Child scopes are created when you use directives like `ng-repeat`, `ng-if`, or `ng-controller` inside other controllers.
- Child scopes can **override** or **add to** the data from the parent scope.

Example of Child Scope

```
javascript
app.controller('ParentController', function($scope) {
    $scope.parentData = "This is parent scope data";
});

app.controller('ChildController', function($scope) {
```

```
    $scope.childData = "This is child scope data";
});
```

HTML (View):

```
html
<div ng-controller="ParentController">
    <p>{{ parentData }}</p>
    <div ng-controller="ChildController">
        <p>{{ parentData }}</p> <!-- Inherited from ParentController -->
        <p>{{ childData }}</p> <!-- Defined in ChildController -->
    </div>
</div>
```

- The `ChildController` has access to `parentData` because it inherits the parent scope. However, the `ParentController` cannot access `childData`, as that belongs to the child scope.

Isolated Scope in Directives

In AngularJS, you can define **isolated scopes** in custom directives, where a directive gets its own scope that is **not inherited** from the parent controller. This is useful when creating reusable components or directives that need their own scope.

```
javascript
app.directive('myDirective', function() {
    return {
        restrict: 'E',
        scope: {
            localData: '@'
        },
        template: '<h1>{{ localData }}</h1>'
    };
});
```

In this example, `myDirective` has its own isolated scope, which doesn't inherit anything from the parent scope. The `localData` attribute is passed into the directive.

Key Concepts for Using `$scope`

1. Two-Way Data Binding

With `$scope`, AngularJS automatically keeps the model (controller data) and view in sync. If you update the model, the view will reflect the changes, and if the user changes the view (e.g., input fields), the model is updated as well.

```
html
<input type="text" ng-model="name">
<p>Your name is: {{ name }}</p>
```

- In this example, `ng-model="name"` binds the input field to `$scope.name`. The data entered into the input will automatically be reflected in the paragraph and vice versa.

2. Event Handling with `$scope`

You can attach functions to `$scope` and use them to handle events in the view, such as button clicks or other user interactions.

```
javascript
$scope.showAlert = function() {
    alert("Button clicked!");
};
html
<button ng-click="showAlert()">Click Me</button>
```

- In this example, when the button is clicked, the `showAlert` function will be triggered, which displays an alert.

3. Scope Life Cycle

The `$scope` object is tied to the life cycle of the controller. When the controller is instantiated, the `$scope` is created and populated. When the controller is destroyed (for example, when a view is removed), the `$scope` is also destroyed.

- **$scope.$destroy()** can be used to manually destroy the scope when necessary, for instance, in cases where you want to clean up event listeners or other resources associated with the scope.

`$scope` VS `$rootScope`

While `$scope` is used for communication between a controller and its associated view, **`$rootScope`** is a global object that can be accessed from anywhere within the AngularJS

application. It's typically used to store application-wide data that needs to be accessible across all controllers and views.

Example of `$rootScope`:

```
javascript
app.controller('MainController', function($rootScope) {
    $rootScope.globalMessage = "This is a global message";
});
```

HTML:

```
html
<h2>{{ globalMessage }}</h2>
```

- `$rootScope.globalMessage` is available globally throughout the application, unlike `$scope`, which is limited to the scope of the current controller or view.

AngularJS Filters & Services

In AngularJS, **filters** and **services** are essential tools for manipulating data and structuring the application logic in a modular and reusable way. These concepts help manage tasks such as transforming data for display in the view or sharing functionality across different parts of the app. Below is a breakdown of what filters and services are, how they work, and examples of how to use them in AngularJS.

1. Filters in AngularJS

Filters in AngularJS are used to format or transform data before displaying it in the view. Filters can be applied to expressions within the view (e.g., `{{ expression | filter }}`) or to model data directly in controllers and services.

Key Characteristics of Filters:

- **Transform Data**: Filters are used to format, sort, or manipulate data before it's displayed in the view.
- **Reusable**: Filters are reusable across different parts of an AngularJS application.
- **Chainable**: Multiple filters can be chained together in one expression.
- **Built-in and Custom Filters**: AngularJS comes with many built-in filters, but you can also create your own custom filters.

Built-in Filters

AngularJS includes several built-in filters that can be used to format and transform data in different ways.

1. **`currency`**: Formats a number as currency.

   ```
   html
   <div>{{ amount | currency }}</div>
   ```

   Example: `12345 | currency` → `$12,345.00`

2. **`date`**: Formats a date value.

   ```
   html
   <div>{{ date | date:'yyyy-MM-dd' }}</div>
   ```

   Example: `new Date() | date:'shortDate'` → `9/13/21`

3. **`filter`**: Filters an array based on a provided condition.

   ```
   html
   <ul>
       <li ng-repeat="item in items | filter: 'apple'">{{ item }}</li>
   </ul>
   ```

4. **`uppercase`** / **`lowercase`**: Converts text to uppercase or lowercase.

   ```
   html
   <div>{{ text | uppercase }}</div>
   ```

5. **`json`**: Converts an object to a JSON string.

   ```
   html
   <div>{{ data | json }}</div>
   ```

6. **`limitTo`**: Limits the number of items to display from an array or string.

   ```
   html
   <div>{{ items | limitTo: 3 }}</div>
   ```

7. **`orderBy`**: Sorts an array based on a given expression.

```
html
<ul>
    <li ng-repeat="item in items | orderBy:'name'">{{ item.name }}</li>
</ul>
```

Example: Using Built-in Filters

Let's say you have an array of numbers and you want to display them as currency:

HTML:

```
html
<!DOCTYPE html>
<html ng-app="myApp">
<head>
    <title>AngularJS Filters Example</title>
</head>
<body ng-controller="MainController">
    <div>
        <h3>Product Prices:</h3>
        <ul>
            <li ng-repeat="price in prices">{{ price | currency }}</li>
        </ul>
    </div>
</body>
</html>
```

JavaScript (Controller):

```
javascript
var app = angular.module('myApp', []);

app.controller('MainController', function($scope) {
    $scope.prices = [99.99, 129.50, 159.95, 89.99];
});
```

- The `currency` filter formats each price as currency, adding a dollar sign and formatting the number.

Creating Custom Filters

You can create your own filters in AngularJS using the `filter()` method on an AngularJS module. Custom filters can be used to transform data in ways that the built-in filters don't provide.

Example: Custom Filter

Let's create a filter that converts text to title case (capitalizing the first letter of each word).

JavaScript (Custom Filter):

```
javascript
var app = angular.module('myApp', []);

app.filter('titleCase', function() {
    return function(input) {
        if (!input) return '';
        return input.replace(/\b\w/g, function(char) {
            return char.toUpperCase();
        });
    };
});

app.controller('MainController', function($scope) {
    $scope.text = 'hello world';
});
```

HTML:

```
html
<!DOCTYPE html>
<html ng-app="myApp">
<head>
    <title>Custom Filter Example</title>
</head>
<body ng-controller="MainController">
    <h1>{{ text | titleCase }}</h1>
</body>
</html>
```

- The `titleCase` filter capitalizes the first letter of each word in the string. If `text` is "hello world", it would display "Hello World".

2. Services in AngularJS

Services in AngularJS are used to organize and share business logic and functionality across different parts of an application. A service in AngularJS is typically a function or object that is used to encapsulate reusable logic, such as managing data, making HTTP requests, or performing calculations.

Key Characteristics of Services:

- **Singleton**: Services are singleton objects, meaning they are instantiated only once during the lifetime of the application and are shared across controllers and other services.
- **Reusable**: Services can be injected into controllers, directives, and other services, promoting code reuse and separation of concerns.
- **Manage Business Logic**: Services encapsulate business logic or data access, making your controllers cleaner and easier to maintain.
- **Dependency Injection (DI)**: AngularJS uses DI to inject services into controllers, directives, and other components.

Creating and Using Services in AngularJS

Example: Basic Service

Let's create a simple service to manage a list of users.

JavaScript (Service Definition):

```
javascript
var app = angular.module('myApp', []);

app.service('UserService', function() {
    var users = ['Alice', 'Bob', 'Charlie'];

    this.getUsers = function() {
        return users;
    };

    this.addUser = function(user) {
        users.push(user);
    };
});

app.controller('MainController', function($scope, UserService) {
    $scope.users = UserService.getUsers();

    $scope.addUser = function() {
        UserService.addUser($scope.newUser);
        $scope.newUser = '';
    };
});
```

HTML (View):

```
html
<!DOCTYPE html>
<html ng-app="myApp">
<head>
    <title>AngularJS Service Example</title>
</head>
<body ng-controller="MainController">
    <h3>User List:</h3>
    <ul>
        <li ng-repeat="user in users">{{ user }}</li>
    </ul>

    <input type="text" ng-model="newUser">
    <button ng-click="addUser()">Add User</button>
</body>
</html>
```

Breakdown:

- The **`UserService`** stores a list of users and provides methods to get and add users.
- The **`MainController`** uses this service to get the list of users and add new ones.
- The `UserService` is injected into the controller, making it available for use within the controller's scope.

Using `factory()` vs `service()`

In AngularJS, there are multiple ways to create services. The two most common methods are `service()` and `factory()`.

- **`service()`**: This creates a constructor function. The object created by `service()` is constructed using the `this` keyword and is returned to the controller or other components.
- **`factory()`**: This returns an object or function directly. It can be more flexible if you need to return different types of values.

Here's a comparison of the two:

```
javascript
// Using service
app.service('UserService', function() {
    var users = ['Alice', 'Bob', 'Charlie'];

    this.getUsers = function() {
        return users;
    };
```

```
});

// Using factory
app.factory('UserService', function() {
    var users = ['Alice', 'Bob', 'Charlie'];

    return {
        getUsers: function() {
            return users;
        }
    };
});
```

The main difference is that `factory()` returns an object directly, while `service()` returns an object created using a constructor function

AngularJS HTTP and Tables

In AngularJS, the **$http** service is used to make asynchronous HTTP requests (such as GET, POST, PUT, DELETE) to retrieve or send data to a server, which is commonly used in web applications to interact with backend APIs. Once the data is retrieved, you can dynamically display it in the view, often in **tables**. Below is an overview of how to use **$http** for AJAX requests and **ng-repeat** for displaying data in tables.

1. Using $http for HTTP Requests

What is `$http`?

The **`$http`** service is part of AngularJS's built-in **AngularJS HTTP API** that allows the application to communicate with the server. It enables you to send requests to an external API or server and process the response. The `$http` service is based on the **Promise** pattern, allowing for asynchronous requests and responses.

Common HTTP Methods:

1. **GET**: Used to retrieve data from a server.
2. **POST**: Used to send data to a server (usually for creating new data).
3. **PUT**: Used to update existing data on the server.
4. **DELETE**: Used to delete data from the server.

Syntax of `$http`

The `$http` service methods return a **promise** object, which you can chain `.then()` or `.catch()` to handle the success and error cases.

Example of GET Request:

```
javascript
$http({
    method: 'GET',
    url: 'https://api.example.com/data'
}).then(function(response) {
    // success callback
    console.log('Data received:', response.data);
}, function(error) {
    // error callback
    console.error('Error fetching data:', error);
});
```

Alternatively, `$http.get()`, `$http.post()`, etc., are shorthand methods for common HTTP actions.

Example: Fetching Data with `$http`

Below is an example where you fetch a list of users from a mock API and display them in a table.

2. Example: Fetching Data and Displaying in Tables

Step 1: Creating the AngularJS Application

Create an AngularJS module, a controller, and a simple HTML structure to display the data in a table.

JavaScript (Controller & $http Request)

```
javascript

var app = angular.module('myApp', []);

app.controller('UserController', function($scope, $http) {
    // Define the URL for the GET request
    var apiUrl = 'https://jsonplaceholder.typicode.com/users';

    // Make the HTTP GET request
    $http.get(apiUrl)
```

```
        .then(function(response) {
            // Success: Bind the data to the scope
            $scope.users = response.data;
        }, function(error) {
            // Error: Log error to the console
            console.error('Error fetching data:', error);
        });
});
```

HTML (View with Table)

```
html

<!DOCTYPE html>
<html ng-app="myApp">
<head>
    <title>AngularJS $http Example</title>
    <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></
script>
    <style>
        table { width: 100%; border-collapse: collapse; }
        table, th, td { border: 1px solid black; padding: 8px; }
        th { background-color: #f2f2f2; }
    </style>
</head>
<body ng-controller="UserController">

    <h2>List of Users</h2>

    <table>
        <thead>
            <tr>
                <th>Name</th>
                <th>Email</th>
                <th>Phone</th>
                <th>Website</th>
            </tr>
        </thead>
        <tbody>
            <tr ng-repeat="user in users">
                <td>{{ user.name }}</td>
                <td>{{ user.email }}</td>
                <td>{{ user.phone }}</td>
                <td>{{ user.website }}</td>
            </tr>
        </tbody>
    </table>

</body>
```

```
</html>
```

How It Works:

1. The **$http.get()** request fetches data from the `https://jsonplaceholder.typicode.com/users` API.
2. Once the data is received successfully, it's bound to the `users` array in the **`$scope`** object.
3. The data is displayed in the HTML table using **ng-repeat**. Each row represents a user from the `users` array.

Explanation of ng-repeat:

- **ng-repeat="user in users"** is used to loop over the `users` array in the controller and render each user's details (name, email, etc.) in a new `<tr>` (table row) for every user.

3. Handling POST, PUT, and DELETE Requests

In addition to GET requests, you can also handle **POST**, **PUT**, and **DELETE** requests using `$http`.

Example: Using POST Request to Send Data

Suppose you want to send data to the server to add a new user:

JavaScript (Controller with POST Request)

```
javascript

app.controller('UserController', function($scope, $http) {
    $scope.newUser = {
        name: '',
        email: '',
        phone: '',
        website: ''
    };

    $scope.addUser = function() {
        $http.post('https://jsonplaceholder.typicode.com/users',
$scope.newUser)
            .then(function(response) {
                // Success: Add the new user to the list
                $scope.users.push(response.data);
                $scope.newUser = {};  // Clear the form
            }, function(error) {
```

```
            // Error: Log error
            console.error('Error adding user:', error);
        });
    };
});
```

HTML (Form for Adding a New User)

```
html

<h3>Add New User</h3>
<form ng-submit="addUser()">
    <input type="text" ng-model="newUser.name" placeholder="Name" required>
    <input type="email" ng-model="newUser.email" placeholder="Email"
required>
    <input type="text" ng-model="newUser.phone" placeholder="Phone" required>
    <input type="text" ng-model="newUser.website" placeholder="Website"
required>
    <button type="submit">Add User</button>
</form>
```

- The `addUser()` function sends the new user data as a **POST** request to the server, and once the data is added, the new user is added to the `users` array and displayed in the table.

4. Deleting Data with $http DELETE Request

You can also delete items from your table and send a **DELETE** request to the server.

JavaScript (Controller with DELETE Request)

```
javascript

$scope.deleteUser = function(userId, index) {
    $http.delete('https://jsonplaceholder.typicode.com/users/' + userId)
        .then(function(response) {
            // Success: Remove user from the local list
            $scope.users.splice(index, 1);
        }, function(error) {
            // Error: Log error
            console.error('Error deleting user:', error);
        });
};
```

HTML (Delete Button)

```
html

<button ng-click="deleteUser(user.id, $index)">Delete</button>
```

- The **deleteUser()** function sends a **DELETE** request to remove the user from the API. Upon success, the user is removed from the local `users` array using the `splice()` method.
- The **ng-click="deleteUser(user.id, $index)"** is used to call the `deleteUser()` method when the "Delete" button is clicked.

5. Pagination in Tables (Optional)

For large datasets, you may want to implement **pagination**. While AngularJS doesn't have a built-in pagination component, you can easily create your own or use a third-party module like **ng-table** or **angular-ui-bootstrap**.

Simple Pagination Example:

1. Define `itemsPerPage` and `currentPage` in the controller.
2. Use **`ng-repeat`** with **`limitTo`** and **`slice`** to paginate through the array.

```
javascript

$scope.itemsPerPage = 5;
$scope.currentPage = 1;

$scope.paginatedUsers = function() {
    var startIndex = ($scope.currentPage - 1) * $scope.itemsPerPage;
    return $scope.users.slice(startIndex, startIndex + $scope.itemsPerPage);
};
```

In the view, you would loop over `paginatedUsers()` instead of `users` directly:

```
html

<tr ng-repeat="user in paginatedUsers()">
    <td>{{ user.name }}</td>
    <td>{{ user.email }}</td>
    <td>{{ user.phone }}</td>
    <td>{{ user.website }}</td>
</tr>
```

You can create navigation buttons to increase or decrease the `currentPage` value.

Creating and Validating Forms in AngularJS

In AngularJS, forms are integral to capturing and validating user input. AngularJS provides robust features for form handling, including **data binding**, **form validation**, and **custom**

validation. You can easily track the form's state (whether it's valid or invalid), display error messages, and customize validation rules.

Key Concepts for AngularJS Forms:

1. **Form Directives**: AngularJS provides built-in directives to bind form inputs to the model and validate the data. These directives include:
   - `ng-model`: Binds input fields to the model.
   - `ng-submit`: Triggers a function when the form is submitted.
   - `ng-required`, `ng-minlength`, `ng-maxlength`, `ng-pattern`: Built-in validation rules.
   - `ng-invalid`, `ng-valid`: These classes are applied to form fields based on their validation state.
2. **Form Validation**: AngularJS forms are validated automatically by Angular when the form is submitted. You can use built-in validation checks (like required fields, minimum length, regex patterns) and display custom error messages.
3. **Form Control State**: AngularJS tracks the state of each form field using the `$dirty`, `$pristine`, `$valid`, and `$invalid` properties. You can use these to determine whether the field has been touched or if it's in a valid state.

1. Basic Form Setup with AngularJS

Let's start by creating a simple form with input fields for **name** and **email**, and validate the form.

Example: Simple Form with Basic Validation

HTML: Simple Form

```
html

<!DOCTYPE html>
<html ng-app="myApp">
<head>
    <title>AngularJS Form Validation</title>
    <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></
script>
    <style>
        .error { color: red; }
    </style>
</head>
<body ng-controller="FormController">
```

```
    <h2>Registration Form</h2>

    <form name="registrationForm" ng-submit="submitForm()" novalidate>
        <!-- Name Field -->
        <div>
            <label for="name">Name:</label>
            <input type="text" id="name" name="name" ng-model="user.name"
required>
            <span class="error" ng-show="registrationForm.name.$touched &&
registrationForm.name.$invalid">Name is required</span>
        </div>

        <!-- Email Field -->
        <div>
            <label for="email">Email:</label>
            <input type="email" id="email" name="email" ng-model="user.email"
required>
            <span class="error" ng-show="registrationForm.email.$touched &&
registrationForm.email.$invalid">Valid email is required</span>
        </div>

        <!-- Submit Button -->
        <button type="submit" ng-
disabled="registrationForm.$invalid">Submit</button>
    </form>

    <pre>Form Data: {{ user | json }}</pre>

    <script>
        var app = angular.module('myApp', []);

        app.controller('FormController', function($scope) {
            $scope.user = {};

            // Submit function
            $scope.submitForm = function() {
                if ($scope.registrationForm.$valid) {
                    alert('Form submitted successfully!');
                } else {
                    alert('Form has errors!');
                }
            };
        });
    </script>

</body>
</html>
```

Explanation:

- **Form Structure**: The form uses `ng-submit="submitForm()"` to trigger the form submission when the user clicks the submit button.
- **Validation**:
  - `required` validation is used on both the **name** and **email** fields.
  - The `ng-show` directive is used to display error messages when a form field has been touched (`$touched`) and is invalid (`$invalid`).
  - The **submit** button is disabled if the form is invalid, using `ng-disabled="registrationForm.$invalid"`.
- **Form State**: `registrationForm.$invalid` checks if the form is invalid, and `$touched` checks if the field has been interacted with.

2. Advanced Validation with AngularJS

You can add more complex validations, such as checking the length of the input or matching it against a regular expression.

Example: Complex Validation with Min/Max Length and Email Pattern

HTML: Advanced Validation Form

```
html

<!DOCTYPE html>
<html ng-app="myApp">
<head>
    <title>Advanced Form Validation</title>
    <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></
script>
    <style>
        .error { color: red; }
    </style>
</head>
<body ng-controller="FormController">

    <h2>Registration Form with Advanced Validation</h2>

    <form name="advancedForm" ng-submit="submitForm()" novalidate>
        <!-- Name Field -->
        <div>
            <label for="name">Name:</label>
            <input type="text" id="name" name="name" ng-model="user.name" ng-
minlength="3" ng-maxlength="50" required>
```

```
            <span class="error" ng-show="advancedForm.name.$touched &&
advancedForm.name.$error.required">Name is required</span>
            <span class="error" ng-show="advancedForm.name.$touched &&
advancedForm.name.$error.minlength">Name must be at least 3 characters
long</span>
            <span class="error" ng-show="advancedForm.name.$touched &&
advancedForm.name.$error.maxlength">Name must be less than 50
characters</span>
        </div>

        <!-- Email Field -->
        <div>
            <label for="email">Email:</label>
            <input type="email" id="email" name="email" ng-model="user.email"
ng-pattern="/^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}$/" required>
            <span class="error" ng-show="advancedForm.email.$touched &&
advancedForm.email.$error.required">Email is required</span>
            <span class="error" ng-show="advancedForm.email.$touched &&
advancedForm.email.$error.pattern">Enter a valid email address</span>
        </div>

        <!-- Submit Button -->
        <button type="submit" ng-
disabled="advancedForm.$invalid">Submit</button>
    </form>

    <pre>Form Data: {{ user | json }}</pre>

    <script>
        var app = angular.module('myApp', []);

        app.controller('FormController', function($scope) {
            $scope.user = {};

            // Submit function
            $scope.submitForm = function() {
                if ($scope.advancedForm.$valid) {
                    alert('Form submitted successfully!');
                } else {
                    alert('Form has errors!');
                }
            };
        });
    </script>

</body>
</html>
```

Key Points:

- **ng-minlength and ng-maxlength**: These are used to specify the minimum and maximum length of the name input.
- **ng-pattern**: This is used to validate the email input against a custom regular expression for email format validation.
- **Validation Feedback**: Each error message is conditionally displayed using `ng-show` when the field is **touched** and **invalid**.

3. Custom Validation in AngularJS

You can also create custom validators for more complex logic (e.g., validating two fields that need to match, custom business logic, etc.).

Example: Custom Validator (Confirm Password)

JavaScript: Custom Validator for Confirm Password

```
javascript

var app = angular.module('myApp', []);

app.controller('FormController', function($scope) {
    $scope.user = {};

    // Custom password confirmation validator
    $scope.passwordMatch = function() {
        return $scope.user.password === $scope.user.confirmPassword;
    };

    $scope.submitForm = function() {
        if ($scope.registrationForm.$valid) {
            alert('Form submitted successfully!');
        } else {
            alert('Form has errors!');
        }
    };
});
```

HTML: Adding Custom Validator to Form

```
html

<form name="registrationForm" ng-submit="submitForm()" novalidate>
    <!-- Password Field -->
    <div>
```

```
        <label for="password">Password:</label>
        <input type="password" id="password" name="password" ng-
model="user.password" required>
    </div>

    <!-- Confirm Password Field -->
    <div>
        <label for="confirmPassword">Confirm Password:</label>
        <input type="password" id="confirmPassword" name="confirmPassword"
ng-model="user.confirmPassword" required
               ng-pattern="user.password" ng-
messages="registrationForm.confirmPassword.$error">
        <span class="error" ng-show="!passwordMatch()">Passwords do not
match</span>
    </div>

    <button type="submit" ng-disabled="registrationForm.$invalid ||
!passwordMatch()">Submit</button>
</form>
```

Explanation:

- The `passwordMatch` function is used to compare the **password** and **confirmPassword** fields.
- If they don't match, the error message "Passwords do not match" is shown.
- The submit button is disabled if the form is invalid or the passwords don't match.

UNIT-4

Routing and navigation in AngularJS are crucial for creating single-page applications (SPA) that allow users to navigate between different views without refreshing the entire page. AngularJS provides a powerful `ngRoute` module that helps you manage application state, routes, and navigation.

Here's an overview of how to create and configure routes in AngularJS, including nested routes and route parameters:

1. Setting Up Routing with `ngRoute`

To get started, you need to include the `ngRoute` module in your AngularJS application. This module provides the `$routeProvider` service, which allows you to define routes.

Step 1: Include the AngularJS and `ngRoute` script

First, ensure you include the necessary scripts in your HTML:

```
html

<script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></
script>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular-
route.min.js"></script>
```

Step 2: Create the AngularJS application module

In your JavaScript file, create an AngularJS module and inject `ngRoute` as a dependency:

```
javascript

var app = angular.module('myApp', ['ngRoute']);
```

2. Defining Routes

Once you have your app and `ngRoute` set up, you can define routes using `$routeProvider` in the config block of your application.

Example: Simple Route Definition

```
javascript

app.config(function($routeProvider) {
    $routeProvider
        .when('/home', {
            templateUrl: 'home.html',
            controller: 'HomeController'
        })
        .when('/about', {
            templateUrl: 'about.html',
            controller: 'AboutController'
        })
        .otherwise({
            redirectTo: '/home'
        });
});
```

In this example:

- The `/home` route loads `home.html` and uses `HomeController`.
- The `/about` route loads `about.html` and uses `AboutController`.
- The `.otherwise()` method is used to define a default route (if the user enters an unknown path).

Example: Home Controller and View

```
javascript

app.controller('HomeController', function($scope) {
    $scope.message = 'Welcome to the Home page!';
});
```

The view (`home.html`) could be a simple HTML file:

```
html

<div>
    <h1>{{ message }}</h1>
</div>
```

3. Navigating Between Views

To navigate between different routes, use the `ng-view` directive in your main HTML layout, and `ng-href` or `$location` in controllers or views to manage routing.

Example: Main Layout (`index.html`)

```
html

<!DOCTYPE html>
<html ng-app="myApp">
<head>
    <title>AngularJS Routing Example</title>
</head>
<body>
    <div>
        <a href="#/home">Home</a> |
        <a href="#/about">About</a>
    </div>
    <div ng-view></div>  <!-- Where views will be injected -->
</body>
</html>
```

The `ng-view` directive is where AngularJS dynamically loads the templates (`home.html`, `about.html`, etc.) based on the current route.

4. Nested Routes

AngularJS also supports nested views, which allow you to create complex page structures with multiple sections. To configure nested routes, use the `views` property in `$routeProvider`.

Example: Nested Routes

```
javascript

app.config(function($routeProvider) {
    $routeProvider
        .when('/user/:id', {
            templateUrl: 'user.html',
            controller: 'UserController',
            resolve: {
                user: function($route, UserService) {
                    return UserService.getUser($route.current.params.id);
                }
            },
            views: {
                'userDetails': {
                    templateUrl: 'userDetails.html',
                    controller: 'UserDetailsController'
                },
                'userPosts': {
                    templateUrl: 'userPosts.html',
                    controller: 'UserPostsController'
```

```
                }
            }
        })
        .otherwise({
            redirectTo: '/'
        });
});
```

In this example, the `/user/:id` route has two nested views:

- `'userDetails'` displays the user's details.
- `'userPosts'` displays the user's posts.

Example: Main Layout for Nested Views (`index.html`)

```
html

<div>
    <a href="#/user/1">User 1</a> |
    <a href="#/user/2">User 2</a>
</div>

<div ng-view></div>  <!-- Main content view -->
<div ng-view="userDetails"></div>  <!-- Nested user details view -->
<div ng-view="userPosts"></div>  <!-- Nested user posts view -->
```

5. Route Parameters

Route parameters allow you to pass dynamic data to views via the URL. You define parameters using the `:parameterName` syntax in the route path.

Example: Route with Parameters

```
javascript

app.config(function($routeProvider) {
    $routeProvider
        .when('/user/:id', {
            templateUrl: 'user.html',
            controller: 'UserController'
        })
        .otherwise({
            redirectTo: '/'
        });
});
```

In this case, `:id` is a route parameter.

Example: Accessing Route Parameters in the Controller

In the controller, you can access route parameters using `$routeParams`:

```
javascript

app.controller('UserController', function($scope, $routeParams) {
    var userId = $routeParams.id;  // Accessing the 'id' parameter from the
URL
    $scope.message = 'User ID: ' + userId;
});
```

6. Using `$location` for Navigation

AngularJS provides the `$location` service for programmatically navigating between routes.

Example: Using `$location` for Navigation

```
javascript

app.controller('HomeController', function($scope, $location) {
    $scope.goToAbout = function() {
        $location.path('/about');  // Navigate to the /about route
    };
});
```

In the HTML, bind the function to a button or link:

```
html
<button ng-click="goToAbout()">Go to About Page</button>
```

AngularJS API Overview

AngularJS (also known as Angular 1.x) is a JavaScript-based framework primarily used for building dynamic web applications. It provides a set of built-in services, directives, and controllers that allow developers to create complex web applications more efficiently. Although AngularJS is no longer actively maintained (since the release of Angular 2+), it was a popular framework for single-page applications (SPA) in the past.

Key Components of AngularJS API:

1. **Modules**:

- A module is a container for the different parts of an AngularJS application like controllers, services, and directives.
- A module can be created with `angular.module('moduleName', [])` and can be configured or run with `.config()` and `.run()` respectively.

2. **Controllers**:
   - Controllers are JavaScript functions that are used to manage the scope of the application. They define the behavior of the view by manipulating the model.
   - Example:

```
javascript

angular.module('app', [])
  .controller('MainCtrl', function($scope) {
    $scope.message = "Hello, AngularJS!";
  });
```

3. **Directives**:
   - Directives are used to extend HTML with new behavior and functionality. AngularJS provides several built-in directives like `ng-model`, `ng-repeat`, etc.
   - Example:

```
html

<div ng-repeat="item in items">
  {{ item.name }}
</div>
```

4. **Services**:
   - Services are reusable pieces of code that provide a specific functionality. You can create a custom service to handle logic, data fetching, etc.
   - Example:

```
javascript

angular.module('app', [])
  .service('DataService', function($http) {
    this.getData = function() {
      return $http.get('/api/data');
    };
  });
```

5. **Filters**:
   - Filters are used to format data for display in the view. AngularJS has several built-in filters like `currency`, `date`, `filter`, etc.

- Example:

```
html

<div>{{ amount | currency }}</div>
```

6. **Dependency Injection (DI)**:
   - Dependency Injection is a core concept in AngularJS. It allows for components (like controllers, services, etc.) to request dependencies (such as other services or values) rather than creating them manually.
   - Example:

```
javascript

angular.module('app', [])
  .controller('MainCtrl', function($scope, DataService) {
    DataService.getData().then(function(data) {
      $scope.items = data;
    });
  });
```

7. **Routing**:
   - AngularJS uses the `ngRoute` module for client-side routing to create single-page applications with multiple views. Routes are defined with the `ngRoute` provider and are mapped to specific controllers.
   - Example:

```
javascript

angular.module('app', ['ngRoute'])
  .config(function($routeProvider) {
    $routeProvider
      .when('/home', {
        templateUrl: 'home.html',
        controller: 'HomeCtrl'
      })
      .when('/about', {
        templateUrl: 'about.html',
        controller: 'AboutCtrl'
      })
      .otherwise({
        redirectTo: '/home'
      });
  });
```

CSS and Animations in AngularJS

AngularJS provides support for CSS manipulations and animations in two primary ways: **ngClass/ngStyle** for dynamic CSS binding, and **ngAnimate** for animations.

1. Dynamic CSS Binding (`ngClass` and `ngStyle`):

- These directives allow you to bind CSS classes and inline styles to model data dynamically.
- **ngClass**: Used to toggle CSS classes dynamically based on an expression.

```
html
<div ng-class="{active: isActive, 'text-success': isSuccess}">
  This div will toggle classes based on the scope values.
</div>
```

- **ngStyle**: Allows inline styles to be applied dynamically.

```
html
<div ng-style="{'color': textColor, 'font-size': fontSize + 'px'}">
  This div has dynamic styles.
</div>
```

2. AngularJS Animations (ngAnimate):

- AngularJS has a built-in animation module, `ngAnimate`, that allows you to create animations when elements are added or removed from the DOM.
- To enable animations in AngularJS, you need to include the `ngAnimate` module in your application and use CSS transitions, keyframes, or AngularJS-specific animation hooks (`ng-enter`, `ng-leave`, etc.).
- Example with CSS animation:

```
css
.fadeIn {
  transition: opacity 1s ease-in-out;
  opacity: 0;
}
.fadeIn.ng-enter {
  opacity: 1;
}
```

- Example with `ngAnimate`:

```
html
<div ng-animate="'fadeIn'">
  <p>This element will fade in on insert!</p>
</div>
```

- The `$animate` service in AngularJS allows you to programmatically control animations:

```
javascript
angular.module('app', ['ngAnimate'])
  .controller('MainCtrl', function($scope, $animate) {
    $scope.addElement = function() {
      var newElement = angular.element('<div class="fadeIn">New
Element</div>');
      angular.element(document.body).append(newElement);
      $animate.enter(newElement);
    };
  });
```

UNIT-5

Overview of Bootstrap and Its Features

Bootstrap is a free, open-source front-end framework that helps developers design and build responsive and mobile-first websites quickly. Initially created by Twitter, it has since become one of the most widely used frameworks for web development.

Key Features of Bootstrap:

1. **Responsive Grid System**: The grid system in Bootstrap allows web pages to adjust automatically to various screen sizes and devices.
2. **Pre-designed Components**: Bootstrap includes ready-to-use components like navigation bars, modals, alerts, cards, and forms that save time in development.
3. **Utility Classes**: Utility classes for spacing, text alignment, color, visibility, and much more help developers make quick adjustments to their layouts.
4. **Customizable**: You can customize Bootstrap to your needs, either by using its extensive configuration options or by writing custom CSS.
5. **JavaScript Plugins**: Bootstrap includes optional JavaScript plugins for components like carousels, modals, dropdowns, and more.

Advantages of Bootstrap:

- **Consistency**: Provides a consistent look and feel across web projects.
- **Speed**: Enables rapid prototyping and quicker web development.
- **Cross-Browser Compatibility**: Bootstrap ensures your site will look good across all major browsers and devices.
- **Built-in Mobile-First Design**: Ensures websites are responsive and optimized for mobile users.

Setting Up the Development Environment

To start using Bootstrap, follow these steps:

1. **Via CDN (Content Delivery Network)**: This is the easiest method. Simply include Bootstrap's CSS and JavaScript files in your HTML:

```
html
```

```
<head>
  <!-- Bootstrap CSS -->
  <link
href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.cs
s" rel="stylesheet">
</head>
<body>
  <!-- Your HTML content goes here -->

  <!-- Bootstrap JS & Popper.js -->
  <script src="https://code.jquery.com/jquery-3.5.1.slim.min.js"></script>
  <script
src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.0.6/dist/umd/popper.min.js
"></script>
  <script
src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js">
</script>
</body>
```

2. **Via Download**: Download the Bootstrap source files from getbootstrap.com. Then link the Bootstrap CSS and JavaScript in your project files.
3. **Via NPM** (for Node.js environments):

   ```
   npm install bootstrap
   ```

   After installation, link the Bootstrap files into your project.

Understanding the Bootstrap Grid System

The Bootstrap grid system is based on a 12-column layout. It allows you to create flexible and responsive layouts by dividing the page into rows and columns. The grid is responsive, meaning the layout will automatically adjust based on the screen size.

Structure:

1. **Container**: The `.container` class defines the width of the page and helps center the content.
2. **Rows**: The `.row` class is used to define a row of columns.
3. **Columns**: Inside a row, you use column classes like `.col-md-4`, `.col-sm-6`, etc., to define the size of each column. These classes are responsive, so you can specify how many columns a section should span on different screen sizes.

Example of the Grid System:

```
html

<div class="container">
  <div class="row">
    <div class="col-12 col-md-6">
      <p>Column 1: Takes full width on small screens and 50% on medium and
larger screens.</p>
    </div>
    <div class="col-12 col-md-6">
      <p>Column 2: Same behavior as column 1.</p>
    </div>
  </div>
</div>
```

Layout and Navigation

Creating and Using the Navbar Component

The **navbar** component is one of the most popular and commonly used Bootstrap components. It helps you create navigation bars for websites that are responsive and mobile-friendly.

Basic Example of a Navbar:

```
html

<nav class="navbar navbar-expand-lg navbar-light bg-light">
  <a class="navbar-brand" href="#">MyWebsite</a>
  <button class="navbar-toggler" type="button" data-toggle="collapse" data-
target="#navbarNav" aria-controls="navbarNav" aria-expanded="false" aria-
label="Toggle navigation">
    <span class="navbar-toggler-icon"></span>
  </button>
  <div class="collapse navbar-collapse" id="navbarNav">
    <ul class="navbar-nav">
      <li class="nav-item active">
        <a class="nav-link" href="#">Home</a>
      </li>
      <li class="nav-item">
        <a class="nav-link" href="#">Features</a>
      </li>
      <li class="nav-item">
        <a class="nav-link" href="#">Pricing</a>
      </li>
    </ul>
  </div>
```

```
</nav>
```

- The `.navbar` class creates the basic navbar.
- `.navbar-expand-lg` makes the navbar responsive, showing a toggle button on smaller screens.
- `.navbar-light` and `.bg-light` define the color scheme of the navbar.

Creating and Using the Grid System for Layout

The grid system allows you to create complex layouts by combining rows and columns. You can use various column sizes for different screen widths (e.g., `.col-md-4` for medium screens).

Example:

```
html

<div class="container">
  <div class="row">
    <div class="col-md-4">Left Sidebar</div>
    <div class="col-md-8">Main Content</div>
  </div>
</div>
```

This will create a layout with a left sidebar that takes up 4 columns and the main content taking up 8 columns on medium-sized screens and above.

Typography and Tables

Using Typography Classes to Style Text

Bootstrap includes several utility classes to style text, including changing the font size, alignment, weight, and color.

Common Typography Classes:

- `.h1`, `.h2`, `.h3`, `.h4`, `.h5`, `.h6`: These classes create headers of different sizes.
- `.lead`: Makes text appear larger for introductory paragraphs.
- `.text-center`, `.text-left`, `.text-right`: Aligns text accordingly.
- `.font-weight-bold`, `.font-italic`: Makes text bold or italic.
- `.text-primary`, `.text-success`, `.text-danger`: Applies Bootstrap theme colors to text.

Example:

```
html

<h1 class="text-center">Welcome to My Website</h1>
<p class="lead text-muted">This is an introductory paragraph styled with
Bootstrap typography classes.</p>
```

Creating and Using Tables

Bootstrap provides a variety of table styles for quick and easy customization.

Basic Table:

```
html

<table class="table">
  <thead>
    <tr>
      <th>Name</th>
      <th>Age</th>
      <th>Occupation</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>John Doe</td>
      <td>28</td>
      <td>Developer</td>
    </tr>
    <tr>
      <td>Jane Smith</td>
      <td>34</td>
      <td>Designer</td>
    </tr>
  </tbody>
</table>
```


- `.table` creates a basic table.
- `.table-striped` adds zebra-striping to the table rows.
- `.table-bordered` adds borders around table cells.
- `.table-hover` highlights rows when you hover over them.

Forms and Buttons

Creating and Using Forms

Bootstrap provides several classes to make forms more user-friendly and responsive.

Example of a Basic Form:

```
html

<form>
  <div class="form-group">
    <label for="exampleInputEmail1">Email address</label>
    <input type="email" class="form-control" id="exampleInputEmail1" aria-
describedby="emailHelp" placeholder="Enter email">
  </div>
  <div class="form-group">
    <label for="exampleInputPassword1">Password</label>
    <input type="password" class="form-control" id="exampleInputPassword1"
placeholder="Password">
  </div>
  <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

- `.form-group` groups label and input together.
- `.form-control` applies styling to input elements.
- `.btn` and `.btn-primary` are used to style buttons.

Styling Forms with Bootstrap Classes

Bootstrap includes several classes for customizing form elements:

- `.form-control-lg`, `.form-control-sm`: Adjust the size of form inputs.
- `.is-valid`, `.is-invalid`: Provides styling for validation states.

Example:

```
html
<input type="text" class="form-control form-control-lg" placeholder="Large
input">
<input type="text" class="form-control form-control-sm" placeholder="Small
input">
```

Creating and Using Buttons and Button Groups

Buttons in Bootstrap are highly customizable with various classes for color, size, and style.

Example of Buttons:

```
html
<button type="button" class="btn btn-primary">Primary Button</button>
<button type="button" class="btn btn-secondary">Secondary Button</button>
```

Button Groups:

You can group multiple buttons together using `.btn-group`:

```
html
<div class="btn-group" role="group" aria-label="Basic example">
  <button type="button" class="btn btn-primary">Left</button>
  <button type="button" class="btn btn-primary">Middle</button>
  <button type="button" class="btn btn-primary">Right</button>
</div>
```