Unit-5

Angular Modules, Services, Angular Forms:

- 5.1 Template-Driven Forms
- 5.2 Reactive Forms
- 5.3 Angular Pipes
- 5.4 Dependency Injection
- 5.5 Observables
- 5.6 Consuming RESTful web service with Angular

Angular Forms

- Angular provides two different approaches to handling user input through forms: reactive and template-driven.
  - Both allows to read user input entered in the form controls
  - capture user input events from the view
  - Validate the user input
  - Create a form model and data model to update and
  - Provide a way to track changes.

Template-Driven Form

- ✓ Simple Basic Form
- ✓ Easy to start with
- ✓ Based on template (HTML)

Reactive Form

- ✓ Complex form with more control
- ✓ Structure of form is defined in TypeScript class

Form Building Blocks/classes

- Angular forms have 3 important building blocks/classes to track the states and values of the forms and their controls:
- They are:
  - **FormControl**
  - **FormGroup**
  - **FormModel**
- **FormControl** will track the value and states of an individual input element such as a text box.
- **FormGroup** is used to track the value and states of a group of the form control. Form itself is considered as a form group.
- **FormModel**
  - is used as a data structure that represents the HTML form.
  - It will retain the form states, form values, and child controls.
  - Angular will automatically create the FormModel

5.2 Reactive Forms

- These are also known as model-driven forms.
- Created in the Component class.
- Define the structure of the form in the component class.
- Create the form model with FormGroup, FormControl and FormArray
- Then, we bind it to the HTML form in the template
- Angular reactive forms follow a model-driven approach to handle form input whose values can be changed over time.
- In reactive forms, we can
  - create and update a simple form control,
  - use multiple controls in a group,
  - validate form values, and
  - implement more advanced forms.

- Approach:
  - Reactive forms are forms where we define the structure of the form in the component class.
  1. Import FormsModule in the root module
  2. Import ReactiveFormsModule, FormGroup, FormControl, FormArray from @angular/forms
  3. Create the form model with FormGroup, FormControl, and FormArray classes.
  4. Create the HTML Form in the template resembling the Form Model
  5. Bind the HTML Form to the Form Model using formGroup and formControlName properties

5.1 Template-driven forms

- Template-driven forms can be useful for adding simple forms to an application.
- Template-driven forms rely on directives in the template to create and manipulate the underlying object model.
- The directive NgModel creates and manages a FormControl instance for a given form element.
- In Template-driven forms:
  - The form is set up using ngForm directive
  - controls are set up using the ngModel directive
  - ngModel provides the two-way data binding

Approach:

- Import FormsModule in the root module
- Build the basic form in HTML Template.
- Bind form controls to data properties present in component class using the ngModel directive and two-way data-binding syntax.
  - Name controls to make them accessible to ngModel.
- Create a form instance using ngForm
- Handle form submission using the ngSubmit output property of the form.

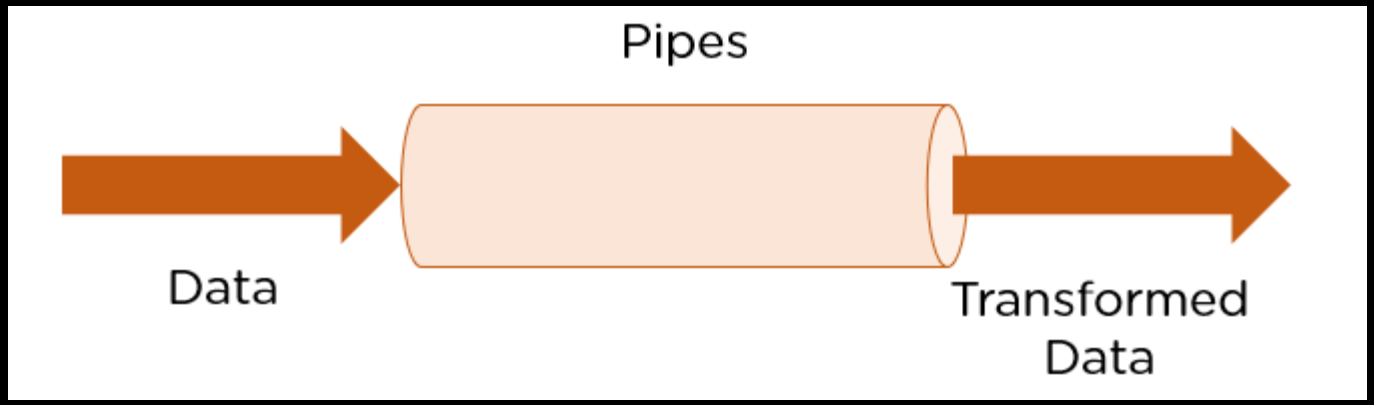
Type	Template	Component Class
Template driven approach	• Majority of the form responsibility will be managed by template. • We write the html statements for our form input elements. • Validation and error messages are handled in template • Two way data binding between template and data model • Form model will be generated automatically by angular	• Responsible for managing the properties for data binding. • Contains the Methods for form operations • Manage the data model for two way data binding
Reactive Forms	• Contains Form element • Contains input elements • data binding between template and **FormModel**	• We define the form model by creating the instance of the **FormControl** and **FormGroup**. • Validation and error messages are managed by component class. • Contains the properties and methods for managing the data model

Reactive Vs Template-driven

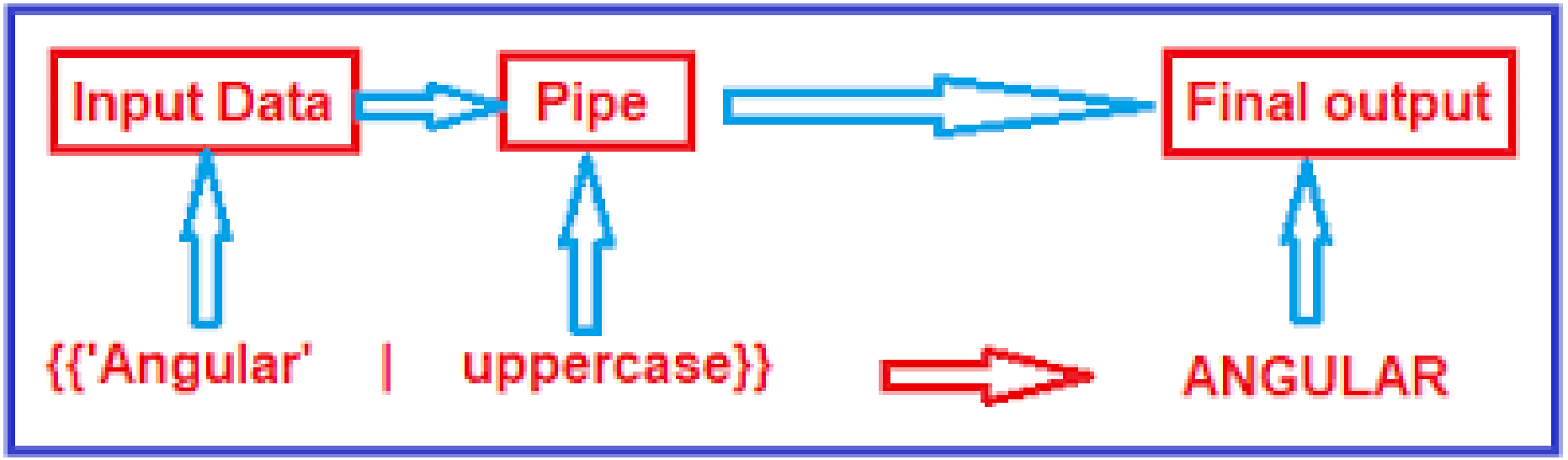
	REACTIVE	TEMPLATE-DRIVEN
Setup (form model)	More explicit, created in component class	Less explicit, created by directives
Data model	Structured	Unstructured
Predictability	Synchronous	Asynchronous
Form validation	Functions	Directives
Mutability	Immutable	Mutable
Scalability	Low-level API access	Abstraction on top of APIs

5.3 Angular Pipes

- Pipes are used to transform data.
- *Definition:* Pipes are simple functions to use in template expressions to accept an input value, process, and return a transformed value as the output.
- Angular Pipes are TypeScript classes with the @Pipe decorator.
- It is denoted by symbol **|**
- It takes integers, strings, arrays, and date as input separated with **|** to be converted in the format as required and display the same in the browser.
- They do not alter the data but change how they appear to the user.
- In Angular 1, **filters** are used which are later called as Pipes from Angular 2.


Pipes
Data
Transformed
Data

Example:


Input Data
Pipe
Final output
{{'Angular' | uppercase}}
ANGULAR

Built-in pipes

- Angular provides built-in pipes for typical data transformations.
- The following are commonly used built-in pipes for data formatting:
- DatePipe: Formats a date value according to locale rules.
- UpperCasePipe: Transforms text to all upper case.
- LowerCasePipe: Transforms text to all lower case.
- CurrencyPipe: Transforms a number to a currency string, formatted according to locale rules.
- DecimalPipe: Transforms a number into a string with a decimal point, formatted according to locale rules.
- PercentPipe: Transforms a number to a percentage string, formatted according to locale rules.
- SlicePipe: Creates a new Array or String containing a subset (slice) of the elements.

Syntax:

- To apply a pipe, use the pipe operator (|) within a template expression:
  - {{ expression | pipename }}
- Pipes with parameters:
  - Follow the pipe name with a colon (:) and the parameter value
  - **{{ expression | pipename : param-value }}**
  - If the pipe accepts multiple parameters, separate the values with colons.
  - {{ expression | pipename : param-value1 : param-value2 }}

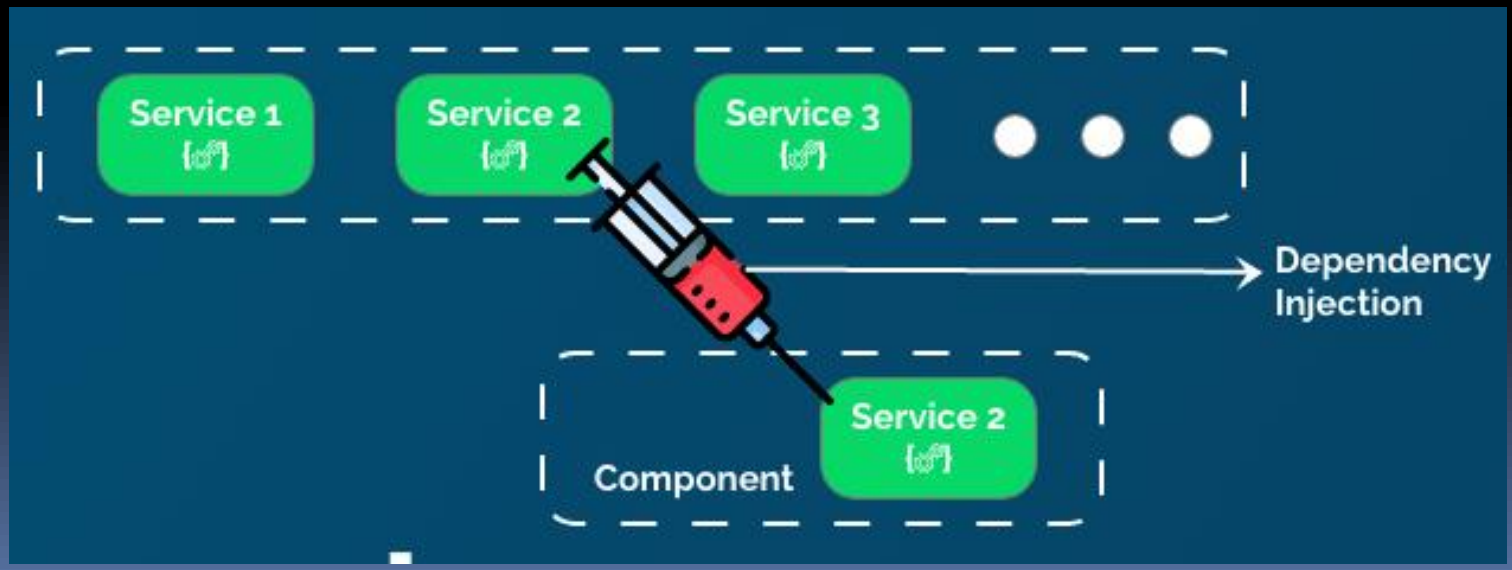
- Chained Pipes:
  - Chain pipes so that the output of one pipe becomes the input to the next.
  - {{ expression | pipe1 | pipe2 }}

Creating Custom Pipes

- Angular makes provision to create custom pipes that convert the data in the format that you desire.
- Steps:
  - Create a custom pipe using the following Angular CLI command:
    - ng g pipe <name_of_the_pipe>
  - Decorate pipe.ts file with the @Pipe decorator and pass the name property to it.
  - Implement the '**PipeTransform'** interface in the class.
  - Implement the **transform** method imposed due to the interface.
  - Return the transformed data with the pipe.
  - Import the custom pipe in the required module

5.4 Dependency Injection

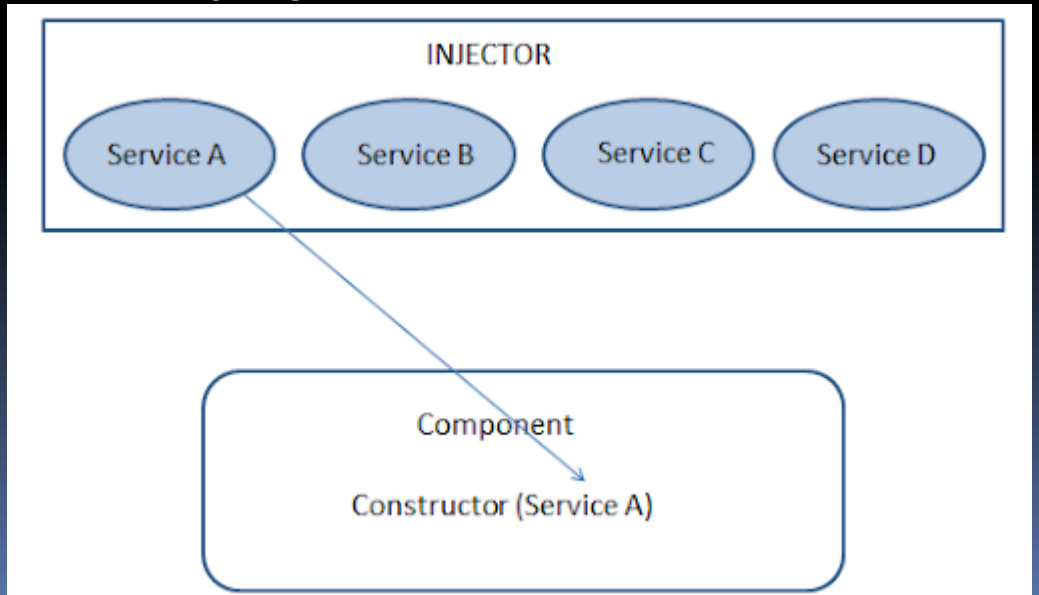
- Dependencies are services or objects that a class needs to perform its function.
- *Dependency injection* is a way to supply a new instance of a class with the fully-formed dependencies it requires.
- *Definition:* It is a coding pattern in which classes receive their dependencies from external sources rather than creating them.
- Angular uses dependency injection to provide new components with the services they need.



- DI (Dependency Injection) is a combination of two terms:
  - **Dependency: Dependency is an object or service that can be used in any other object.**
  - **Injection: An Injection is a process of passing the dependency to a dependent object that would use it.**

Procedure:

- When Angular creates a component, it first asks an **injector** for the services that the component requires.
- An injector maintains a container of service instances that it has previously created.
- If a requested service instance is not in the container, the injector makes one and adds it to the container before returning the service to Angular.
- When all requested services have been resolved and returned, Angular can call the component's constructor with those services as arguments. This is called as *dependency injection*.



Angular Dependency Injection Framework

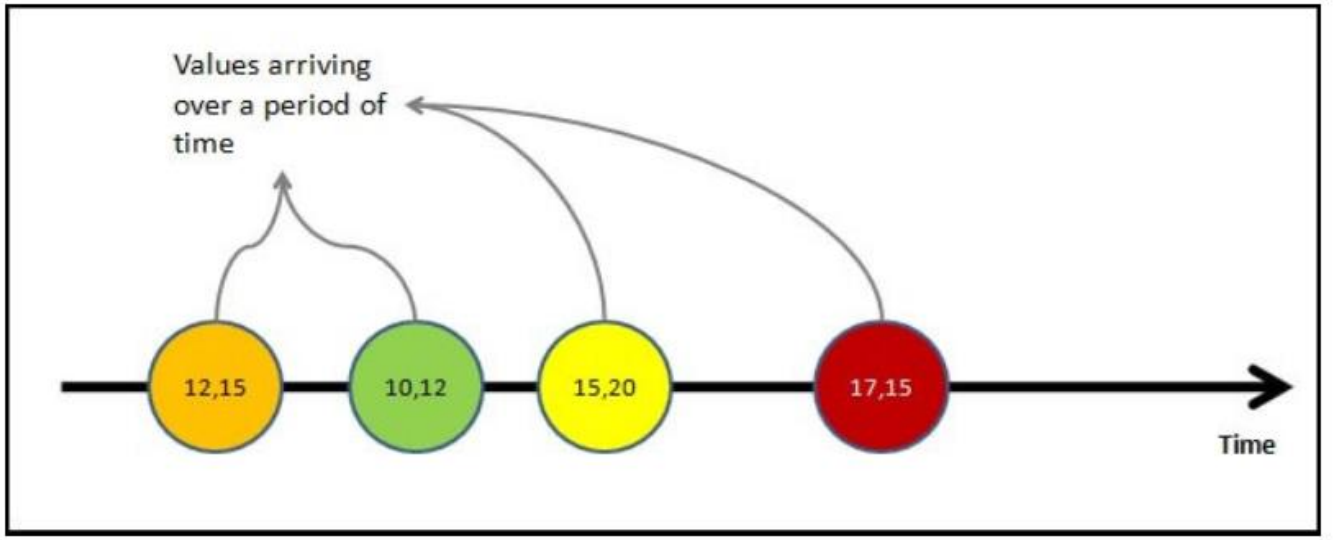
- **Angular's DI framework provides dependencies to a class upon instantiation.**
- Angular Dependency Injection Framework mainly consists of this major parts:
  - @Injectable() decorator
  - Consumer class
  - Dependency
  - Provider
  - Injector

- 1. **@Injectable():** The @Injectable() lets Angular know that a class can be used with the dependency injector.
- 2. **Consumer:** This is the Component or class that needs the dependencies or functionality.
- 3. **Dependency:**
  - The Service that is being injected.
  - To inject dependency into a class, we need to declare @Injectable() decorator in to that class.
- 4. **Provider:**
  - It maintains the list of all the dependencies.
  - It provides the instance of dependencies to the injector.
  - We can define providers list(Array) in component it self in meta data or can define in module file.
- 5. **Injector:**
  - Injector is responsible for injecting the instance of the dependency(Service) to the Consumer(Component).
  - To create an instance, injector looks for a provider.

- Creating an injectable service
  - To generate a new service use the following CLI command:
    - ng g service <service-name>
  - Include @Injectable decorator to the service class
  - Add this service to the providers array present in the @NgModule decorator array of Root module.
- Injecting services
  - To inject a dependency in a component's constructor(), supply a constructor argument with the dependency type.
  - Constructor(instance : service-name) { }

5.5 Observables

- Observables provide support for passing messages between parts of your single-page application.
- They are used frequently in Angular and are a technique for event handling, asynchronous programming (similar to promises), and handling multiple values
- **"Observable emits the value from the stream asynchronously"**
  - **Data Stream** – a sequence of data which is emitted asynchronously over a period of time .
  - **Example:**



mouse click events as data streams

- An observable can deliver multiple values of any type – literals, messages, events, depending on the context.
- The Observable on its own is useless unless someone consumes the value emitted by the observable. We call them **observers** or **subscribers**.
- The **observer must subscribe** with the observable to receive the value from the observable.
- **Observables are declarative**—that is, you define a function for publishing values, but it is not executed until a consumer subscribes to it.
- The observable starts emitting the value as soon as the observer or consumer subscribes to it.
- The **subscribed observers then receives notifications/values** until the function completes, or until they unsubscribe.
- **The observers communicate with the Observable using callbacks**

- While subscribing using subscribe() method observer optionally passes the three callbacks.
- next(), error() & complete()