Day 3

**@NgModule decorator**

@NgModule

Every Angular application consists of at least one module, the root module. We bootstrap that module to launch the application.

NgModules are TypeScript classes decorated with the [@NgModule](https://angular.io/api/forms/NgModel) decorator imported from the @angular/core package.

NgModule takes metadata and describes how to compile a component's template and how to create an injector at runtime. It identifies the module's own components, directives, and pipes and makes them public through the export property which can be used by external components.

The Angular CLI generates the basic *AppModule* (src/app/app.module.ts file) when creating a new application.

// imports

import { BrowserModule } from '@angular/platform-browser';

import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';

// @NgModule decorator with its metadata

@NgModule({

declarations: [AppComponent],

imports: [BrowserModule],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule {}

@NgModule takes the below metadata to launch the application:

* **declarations** — contains a list of components, directives, and pipes, which belong to this module.
* **imports** — contains a list of modules, which are used by the component templates in this module reference. For example, we import *BrowserModule* to have browser-specific services such as DOM rendering, sanitization, and location.
* **providers** — the list of service providers that the application needs.
* **bootstrap** — contains the root component of the application

Angular CLI creates an application with one component (AppComponent), so it is in both the declarations and the bootstrap arrays.

Bootstrapping in Angular:

The steps involved in starting an angular application:

* The *main.ts* is an entry point of an angular application.
* Then, we bootstrap an angular application and we pass *app.module.ts* as an argument. In *app.module.ts*, we tell the Angular to bootstrap the *AppComponent*.
* Then, Angular analyzes this *AppComponent* and knows there is an app-root selector defined.
* Now, Angular able to handle app-root in the *index.html* file.
* Finally, the *index.html* file is loaded on the browser.

References

* [Angular Docs - NgModules](https://angular.io/guide/ngmodules)
* [Angular Docs - Launching your app with a root module](https://angular.io/guide/bootstrapping)

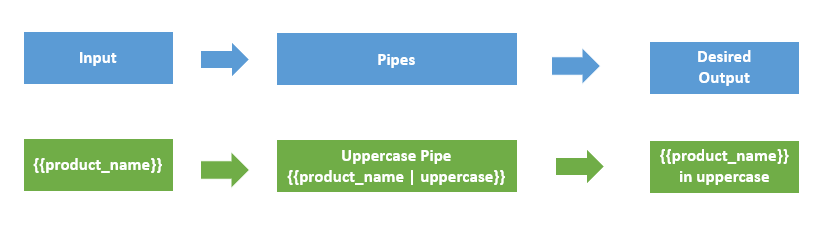
**Pipes**

# **Pipes**

Pipes provide a way to transform values in an Angular template. Pipes are used with a Pipe (|) character, and take integers, strings, arrays, and date as input and returns a desired formatted output which can be displayed in the browser.

For example, a Date object shows the date in this format: Sat Aug 03 2019 19:48:11 GMT+0530 (India Standard Time) which is not easy for the normal users to understand. It’s better to have the date in this format Saturday, 03 Aug 2019 07:50 PM. This can be achieved using pipes.

**Syntax:** {{title | uppercase}}



Pipes can be easily used in HTML templates. For example,

lastLoggedInTime = new Date(2018, 5, 25);

Here, the difference in using pipes.

<h3> Without date Pipe </h3>

<div>

Last Logged in @ {{lastLoggedInTime}}

<!-- OUTPUT Last Logged in @ Mon Jun 25 2018 11:48:11 GMT+0530 (India Standard Time) -->

</div>Copy

<h3> With date Pipe </h3>

<div>

Last Logged in @ {{lastLoggedInTime | date}}

<!-- OUTPUT Last Logged in @ Jun 25, 2018 -->

</div>

### **Parameterizing a pipe**

Pipes accept the any number of optional parameters to fine-tune their output. To add parameters to a pipe, follow the pipe name with a colon ( : ) and then the parameter value. For example,

<p>Date Of birth : {{ birthday | date:"MM/dd/yy" }} </p>

<!-- OUTPUT Date Of birth : 05/19/87 -->

### **Chaining pipes**

We can chain pipe together in effective combinations. For example,

<p>Date Of birth : {{ birthday | date | uppercase}} </p>

<!-- OUTPUT Date Of birth : MAY 19, 1997 -->

## Built-in pipes

Angular has the following built-in pipes:

* **Date pipe** - Used for formatting dates.
* **Decimal pipe** - Used for formatting numbers. The syntax for Decimal Pipe: {{ value\_expression | number : digitInfo }} where, digitInfo is optional. The represntaion of digitInfo: {minIntegerDigits}.{minFractionDigits}-{maxFractionDigits}. Where, minIntegerDigits - minimum integers before the decimal point (default is 1). minFractionDigits - minimum digits after the decimal point (default is 0). maxFractionDigits - maximum after the decimal point (default is 3). If minFractionDigits and maxFractionDigits value is 0, then the number is rounded off to nearest value. Example:

<p>{{3.676 | number}}</p>

<!--output '3.676'-->

<p>{{ 3.67 | number:'3.4-6'}}</p>

<!--output '003.1400'-->

<p>{{-2.5 | number:'1.0-0'}}</p>

<!--output '-3'-->

* **Currency pipe** - Used for formatting currencies. Example:

<p>{{75 | currency}}</p>

<!-- output '$75.00' -->

<p>A: {{ 75 | currency:'CAD'}}</p>

<!-- output 'CA$75.00' -->

<p>A: {{ 75 | currency:'CAD':'code'}}</p>

<!-- output 'CAD75.00' -->

* **Percent pipe** - Used for formatting percentage values. Example:

<p>{{ 26 | percent}}</p>

<!--output '26%'-->

<p>{{ 134.95 | percent:'4.3-5'}}</p>

<!--output '134.950%'-->

* **Slice pipe** - Used for Slicing strings. Example:

<p>{{'abcdefghij' | slice:0:4}} </p>

<!-- output 'abcd' -->

<p>{{'abcdefghij' | slice:4:0}} </p>

<!-- output '' -->

<p>{{'abcdefghij' |slice:-4}} </p>

<!-- output 'ghij' -->

* **Lowercase pipe** - Used for converting strings into lowercase. Example:

<p>{{'tHIs is sUndAY' | lowercase}}</p>

<!-- output "this is sunday" -->

* **Uppercase pipe** - Used for converting strings into uppercase. Example:

<p>{{'tHIs is sUndAY' | uppercase}}</p>

<!-- output "THIS IS SUNDAY" -->

* **Titlecase pipe** - Used for converting strings into title case. Example:

<p>{{'tHIs is a sUndAY' | titlecase}}</p>

<!-- output "This Is Sunday" -->

* **Json pipe** - Used for Converting values into a JSON-format representation. Example:

@Component({

selector: 'json-pipe',

template: `<div>

<p>With JSON pipe: {{object | json}} </p>

<!-- With JSON pipe: { "foo": "bar", "xyz": 3, "numbers": [ 1, 2, 3, 4, 5 ] } -->

<p>Without JSON pipe: {{object}} </p>

<!-- Without JSON pipe: [object Object] -->

</div>`

})

export class JsonPipeComponent {

object: Object = { "foo": "bar", "xyz": 3, "numbers": [ 1, 2, 3, 4, 5 ]};

}

* **Async Pipe** - Used for unwrapping values from an asynchronous primitive. The async pipe subscribes to an Observable or Promise and returns the latest value it has emitted. Example: Here, we bind the Promise with the template. Clicking the Resolve button resolves the promise.

@Component({

selector: 'async -pipe',

template: `<div>

<h3> Async Pipes for Promise</h3>

<button (click)="clicked()">{{ arrived ? 'Reset' : 'Resolve' }}</button>

<p>Wait for it... {{ logMessage | async }}</p>

</div>`

})

export class AsyncPipeComponent {

logMessage: Promise<string>|null = null;

arrived: boolean = false;

private resolve: Function|null = null;

constructor() { this.reset(); }

reset() {

this.arrived = false;

this.logMessage = new Promise<string>((resolve, reject) => { this.resolve = resolve; });

}

clicked() {

if (this.arrived) {

this.reset();

} else {

this.resolve !('Hello User!!!!');

this.arrived = true;

}

}

}

**Custom Pipes**

Custom Pipes

We can create custom pipes using the ng g pipe <pipe-name> command in the terminal with the Angular CLI.

**For example**, we create a custom pipe to count words by running the ng g pipe wordcount command in the terminal. The CLI creates 2 files - *wordcount.pipe.spec.ts* and *wordcount.pipe.ts* under *src/app* folder and updates the *app.module.ts* file.

**app.module.ts**

import { BrowserModule } from '@angular/platform-browser';

import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';

// Custom Pipe imported here by Angular CLI

import { WordcountPipe } from './wordcount.pipe';

@NgModule({

declarations: [

AppComponent,

WordcountPipe

],

imports: [

BrowserModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

In *wordcount.pipe.ts* file, we write the logic for word count.

import { Pipe, PipeTransform } from '@angular/core';

@Pipe({name: 'wordcount'})

export class WordcountPipe implements PipeTransform {

transform(value: any, args?: any): number {

return value.trim().split(' ').length;

}

}

* We import the @Pipe decorator from the core Angular library. If the Class is decorated with the @Pipe decorator, Angular knows that class is a pipe.
* In the @Pipe decorator, we define the pipe *name* that used within template expressions.
* The pipe class implements the **PipeTransform** interface to perform a transformation.
* There is a **transform** method that accepts an input value followed by optional parameters and returns the transformed value.

interface PipeTransform {

transform(value: any, ...args: any[]): any

}

In *app.component.html* file,

<p> {{ "Angular is an application design framework" | wordcount}} </p>

<!--output '6' -->

We can have additional arguments to the **transform** method for each parameter passed to the pipe.

*Example:*

@Pipe({name: 'powerUp'})

export class powerUpPipe implements PipeTransform {

transform(value: number, power?: number): number {

return Math.pow(value, isNaN(power) ? 1 : power);

}

}

In *app.component.html* file,

<p> {{ 2 | powerUp }} </p>

<!--output '2' -->

<p> {{ 2 | powerUp : 3}} </p>

<!--output '8' -->

<p> {{ 5 | powerUp : 2}} </p>

<!--output '25' -->

**1-way data binding**

## Data Binding

* In Angular, Data Binding provides the communication between a component and the DOM.
* It is a technique used to bind the data from an HTML template to a Component class or from a Component class to an HTML template.
* Angular allows both **one-way data binding** and **two-way data binding**.

## One-way data binding

One-way data binding allows us to manipulate the views through the models. If we make any changes in the class, it gets reflected in the template. In Angular, One-way data binding achieved through:

* Interpolation or String Interpolation
* Property binding
* Event binding

### **String Interpolation**

In String Interpolation, we bind the data from a typescript class to the template by using the expression in double curly braces. It is one-way from component class to the template. For example:{{ data }}.

**Example:** In app.component.ts file, we declared a name variable.

export class AppComponent {

name = "John";

}

In app.component.html file, we can access the name variable using string interpolation.

<p> Hello, {{username}} !!! </p>

### **Property Binding**

Property binding allows us to bind values to the attributes of HTML elements. Whenever the value of the component changes, the Angular updates the element attribute in the template. It's also a one-way binding from component class to template.

**Example:** In app.component.ts file, we declared a image variable.

export class AppComponent {

image = "https://assets.winni.in/c\_limit,dpr\_1,fl\_progressive,q\_80,w\_600/35184\_everlasting-paradise.jpeg";

}

In app.component.html file, we can access the name variable using string interpolation.

<h3>Property Binding</h3>

<img [src]="image">

<h3>String Interpolation</h3>

<img src = {{image}}>

String interpolation and property binding are somewhat interchangeable. If we need to set element properties to non-string data values, we must use Property binding, not String Interpolation.

### **Event Binding**

Event binding allows us to bind DOM events such as keystrokes, button clicks, mouse overs, touches, etc to a function in the component. It is one way from template to the component class.

**Example:** When we click the submit button, it calls the onSubmit() method defined in the component. Here, all DOM events are closed within parentheses.

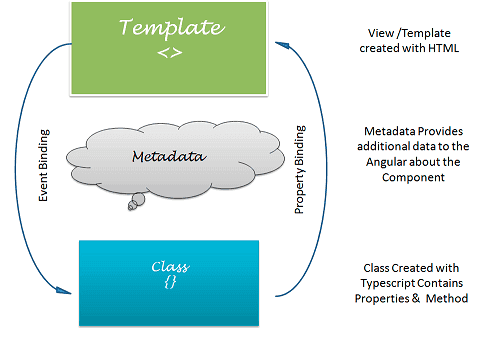
<input type="submit" (click) ="onSubmit()"/>

**2-way data binding**

Two-way data binding

In one-way data binding, any changes in the template are not reflected in the component class. To solve this, Angular provides two-way data binding.

Two-way data binding is achieved by combining **property binding** and **event binding** together.



Two-way data binding is useful in data entry forms. The Angular uses the ngModel directive to achieve two-way binding on HTML <form> elements.

To use the ngModel directive, we need to import the **FormsModule** package into our Angular module.

**Syntax:**

<input type="text" name="value" [(ngModel)]="value">

The ngModel directive is placed inside the square & parentheses and assigned to the property in the component class.

**Example** Import the **FormsModule** in the *app.module.ts* file.

import { BrowserModule } from '@angular/platform-browser';

import { NgModule } from '@angular/core';

import {FormsModule} from '@angular/forms';

import { AppComponent } from './app.component';

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

FormsModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

In *app.component.ts* file, we declare a name variable and in *app.component.html* file, we use the ngModel directive for two-way binding.

Enter your name: <input type="text" [(ngModel)]='name'><br>

<p>Hello, {{ name }} !!!</p>

**Services and Dependency Injection**

Services

* Services are used to organize and share business logic, models, data, or functions with different components of an Angular application.
* An Angular service is a singleton instance that can be injected into multiple components, where the component utilizes the functions defined in a service class. This allows us to reuse the code.

The following example illustrates why we need a service. For instance, consider an *EmployeeNames* component that displays the employee names. Here we declare the *employees* object array in the class and display the *name* in the template.

@Component({

selector: 'app-employee-names',

template:`<div>

<h2> Employee Names</h2>

<ul \*ngFor = "let emp of employees">

<li> {{ emp.name }} </li>

</ul>

</div>`,

styles: []

})

export class EmployeeNamesComponent {

public employees = [

{"id" : 12 , "name" : "Chris", "age" : 22 },

{"id" : 13 , "name" : "Joseph", "age" : 25 },

{"id" : 14 , "name" : "Alex", "age" : 35 }

];

}

Consider a scenario, if we need another component to display the employee list with id, name and age. To do so, we create an *EmployeeList* component and in that component also, we'll have the *employees* object array to display each employee's id, name and age.

@Component({

selector: 'app-employee-List',

template:`<div>

<h2> Employee List</h2>

<ul \*ngFor = "let emp of employees">

<li>{{ emp.id }} - {{ emp.name }} - {{ emp.age }}</li>

</ul>

</div>`,

styles: []

})

export class EmployeeNamesComponent {

public employees = [

{"id" : 12 , "name" : "Chris", "age" : 22 },

{"id" : 13 , "name" : "Joseph", "age" : 25 },

{"id" : 14 , "name" : "Alex", "age" : 35 }

];

}

Here, we are repeating the same *employees* object code in the two components. Also, the component is only responsible for controlling the view's logic, but in our case, it is also responsible for creating data *employees* object.

So, components can delegate these tasks to services. A service is a class that is used to share data across the multiple components, to implement the application logic, and for external communication.

In Angular, we have a file with \***.service.ts** extension where we define the service class. In that class, we can have a *employees* object code, which later can be injected into the components. We use a **Dependency Injection framework** to inject the service into components in our application.

Dependency Injection (DI)

**DI as a design pattern** - Consider, we have three classes *Engine*, *Tires*, and *Car*. Let's assume, we need an *Engine* instance and a *Tires* instance to create a *Car* instance. So, the *Car* class has two dependencies - *Engine* and *Tires*.

Class Engine{

constructor(){}

}

Class Tires{

constructor(){}

}

Class Car{

engine;

tires;

constructor()

{

this.engine = new Engine();

this.tires = new Tires();

}

}

The *Engine* and *Tires* are tightly coupled to the *Car*. For instance, Whenever we change the *Engine* and *Tires* class definition, then we need to manually change the *Car* Class definition. This problem can be solved by using the **Dependency Injection** design pattern.

Have a look at this code making use of the **Dependency Injection** design pattern. Instead of creating an instance of the *Engine* and *Tires* classes inside the *Car* constructor, we assign references to passed-in *Engine* and *Tires* objects.

Class Car{

engine;

tires;

constructor(engine, tires)

{

this.engine = engine;

this.tires = tires;

}

}

var engine1 = new Engine(newparameter);

var engine2 = new Engine();

var tires = new Tires();

var myCar = new Car(engine1, tires);

var otherCar = new Car(engine2, tires);

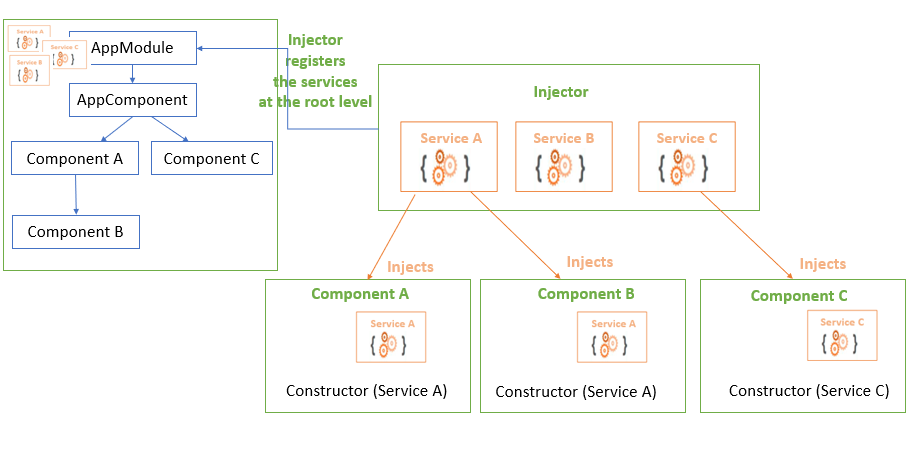
**Dependency Injection** allows the class to receive its dependencies from external sources rather than creating them itself.

With DI, we create a *Car* instance by passing the dependencies as parameters. Here the *Car* has only 2 dependencies (*Engine* and *Tires*), but what if the *Car* 10 or 20 dependencies? Then, we would have to create an instance for each of those dependencies before passing them as parameters. And if these dependencies in turn had dependencies on something else then we would have to create *those* dependencies. As the number of dependencies grows, it rapidly becomes difficult to manage the code.

To overcome this, we use **Dependency Injection as a Framework** in Angular. A DI framework uses an **Injector** where we register all those dependencies to be managed. If we need a *Car* instance, the injector will provide the *Car* instance. The injector is responsible for creating service instances and injecting them into components.

**NOTE -** In Angular, dependencies are typically services.

Using a Service



The Injector holds all the services, and registers them at the NgModule or component level based on their **provider**. A provider tells where in our application to register the service. The registered service can be accessed using a **DI token**. A DI token is a lookup key for the registered services.

The @Injectable() decorator marks a class as a service class that can be injected. The @Injectable() decorator has a providedIn property where we specify the **provider** of the decorated service class with the 'root' as a default, or any other module of our application.

**Example:**

Here, we create an Employee Service that shares the employee object array with the *EmployeeList* and *EmployeeDetail* component.

Run the ng g s employee command in the CLI, which creates *employee.service.ts* file and *employee.service.spec.ts* file under src/app folder.

In *employee.service.ts* file, you have a basic structure for an angular service.

import { Injectable } from '@angular/core';

@Injectable({

providedIn: 'root'

})

export class EmployeeService {

constructor() { }

}

An injector provides a **singleton instance of a service**, and can inject this same instance into multiple components.

A **hierarchy of injectors** at the NgModule and component level can provide different instances of a services to their own components and child components. We can configure injectors with different providers that can provide different implementations of the same services.

If we register the service in the EmployeeListComponent, then the service can be used by EmployeeListComponent and its child components. Other components in our application, can't use this service. If we register the service in the *AppModule*, then the service can be used by all the components in our application. To register the service, we use providedIn property in the @Injectable() decorator.

By default, AngularCLI sets the providedIn to 'root' registers the service at the module level.

AppModule <-- EmployeeService

|

AppComponent

|

| -----> EmployeeNamesComponent

|

| -----> EmployeeListComponent

**employee.service.ts** - Here we return the employee object array in the getEmployees(). After injecting this service into the EmployeeListComponent and EmployeeNamesComponent, both can call the getEmployees() method which returns employee object array.

@Injectable({ providedIn: 'root' })

export class EmployeeService {

getEmployees() : object {

return [

{"id" : 12 , "name" : "poko", "age" : 22 },

{"id" : 13 , "name" : "Joseph", "age" : 25 },

{"id" : 14 , "name" : "Alex", "age" : 35 }

];

}

}

When we create an instance of EmployeeService class, Angular injects the *EmployeeService* into that commponent. Here, \_employeeService is used as a lookup key for the EmployeeService.

\_employeeService : EmployeeService

In the EmployeeNamesComponent, we return the employee object array by calling \_employeeService.getEmployees() method definied inside the *EmployeeService*. Similarly we can access the employee object array in the EmployeeListsComponent. By doing so, EmployeeNamesComponent and EmployeeListsComponent controls only the view logic.

@Component({

selector: 'app-employee-names',

template:`<div>

<h2> Employee Names</h2>

<ul \*ngFor = "let emp of employees">

<li> {{ emp.name }} </li>

</ul>

</div>`,

styles: []

})

export class EmployeeNamesComponent {

public employees:object =[];

constructor(private \_employeeService : EmployeeService){

this.employees = \_employeeService.getEmployees();

}

}

Day 4

**Route guards**

Routing guards

Router guards are used to check whether the user should grant or remove access to certain parts of the navigation.

There are 4 different interfaces act as routing guards:

* CanActivate - decides if the route can be activated.
* CanActivateChild- decides if children of a route can be activated.
* CanLoad- decides if a route can be loaded.
* CanDeactivate- decides if the user can leave a route.

**Example:**

Run the ng g guard <guard-name> command in your terminal to generate a guard service. When we run the ng g guard admin command, the CLI creates a service class that implements any one of the guard interface.

*admin.guard.ts:*

import { Injectable } from '@angular/core';

import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/router';

import { Observable } from 'rxjs';

import { AuthService } from './auth/auth.service';

@Injectable({

providedIn: 'root'

})

export class AdminGuard implements CanActivate {

constructor(private authService: AuthService){}

canActivate(

next: ActivatedRouteSnapshot,

state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {

return this.authService.isLoggedIn;

}

}

* Adminguard is a class that implements the *CanActivate* interface and overrides the canActivate() method. The canActivate() method uses the following parameters:
  + next: ActivatedRouteSnapshot - Contains the information about a route associated with a component loaded in an outlet at a particular moment in time.
  + state: RouterStateSnapshot - Contains the information about router state at a particular moment in time.
* In this example, the canActivate() method to only allow access if the user is logged in. Here imported the *AuthService* to get the value of isLoggedIn property which holds true if the user logged in else false.
* We apply the guard to the routes, by imposing canActivate property of the path object. *admin-routing.module.ts*

const routes: Routes = [

{

path: 'admin',

component: ProjectComponent,

children: [

{

path: 'list',

component: EmployeeListComponent,

canActivate: [AdminGuard]

},

{

path: 'create',

component: EmployeeListComponent,

canActivate: [AdminGuard]

}

]

}

* Here, we can access the *EmployeeListComponent* and *EmployeeListComponent* only if we had logged in.

**HttpClient**

## HttpClient

Most front-end applications need to communicate with a server over the HTTP protocol, in order to download or upload data and accesss other back-end services. Angular provides a simplified client HTTP API for Angular applications, the HttpClient service class in [@angular/common/http](https://angular.io/api/common/http).

The Angular HttpClient offers testability features, typed request and response objects, request and response interception, Observable APIs, and streamlined error handling.

The HttpClient Module is already included when creating a new Angular app. We just need to register it in our Angular application. In src/app/app.module.ts file, import the HttpClient module to make use of the HttpClient service.

import { HttpClient } from '@angular/common/http';

Also, include the HttpClientModule in @NgModule's imports array.

@NgModule({

imports: [

BrowserModule,

HttpClient

]

})

Now, the Angular HttpClient is ready to use or inject with the Angular service or component.

The HttpClient service is used for communication between front-end web apps and backend services. This communication is done over the HTTP protocol. The HttpClient service is available as an injectable class, with methods to perform HTTP requests. The Angular HttpClient Methods are request(), delete(), get(), patch(), post(), put(), head(), jsonp(), and options(). All HttpClient methods return an **Observable** of something. In general, an observable can return multiple values over time. An observable from HttpClient always emits a single value and then completes, never to emit again.

The HttpHeaders service is used for the header configuration options of an HTTP request. HTTP Headers let the client and the server share additional information about the HTTP request or response. For example, we use the content-type header to indicate the media type of the resource like JSON, text, blob, etc.

### **Handling Errors with HttpClient**

By using Angular's HttpClient along with catchError from RxJS, we can easily write a function to handle errors within each service. HttpClient will also conveniently parse JSON responses and returns an observable object.

There are two categories of errors which need to be handled differently:

* Client-side: Network problems and front-end code errors. With HttpClient, these errors return ErrorEvent instances.
* Server-side: AJAX errors, user errors, back-end code errors, database errors, file system errors. With HttpClient, these errors return HTTP Error Responses.

By verifying if an error is an instance of ErrorEvent, we can figure out which type of error we have and handle it accordingly.

To catch errors, we "pipe" the observable result from http.get() (or any HttpClient methods) through an RxJS catchError() operator. Also, we add the retry(1) function to the pipe to retry all requests once before failing.

## Example:

We are going to create a fake backend server using the [json-server](https://www.npmjs.com/package/json-server" \t "_blank) NPM module in our Angular app. This module will allow us to communicate with a server to which we can send and receive data locally.

Run the npm install -g json-servercommand to set the fake json-server globally.

In the root folder of the Angular project, create a folder by the name of backend and also create a file by the name of database.json. This file will have our fake JSON data. Add some fake data to the database.json file:

{

"employees": [

{"id" : 12 , "name" : "Chris", "age" : 22 },

{"id" : 13 , "name" : "Joseph", "age" : 25 },

{"id" : 14 , "name" : "Alex", "age" : 35 }

]

}

We are done setting up a fake JSON server in our Angular application. To start the fake JSON server, run the json-server --watch backend/database.json command in the terminal. Now, your fake json server is up and running on the port **3000**. You are able to view this employees array by visiting <http://localhost:3000/employees> on the browser. Now that our server is ready, we communicate with the server through HTTP Requests.

We create service file that allow us to handle all HTTP requests to our application. All HttpClient methods return an **observable** object, so we need to cast the observable object into an Employee type.

Before, we create the service file we need to create an interface to define the employee type. So, the observable object returned by the HttpClient Methods can be cast to the Employee type.

export interface Employee{

id : number;

name : string;

age : number;

}

Let's create a employee.service.ts file to handle all HTTP requests. We import the HttpClient and HttpHeaders services to make the HTTP request work. Here, we create CRUD operations using HttpClient methods (GET, POST, PUT, DELETE) and also there is some error handling logic in it.

import { Injectable } from '@angular/core';

import { HttpClient, HttpHeaders } from '@angular/common/http';

import { Observable, throwError } from 'rxjs';

import { catchError, retry } from 'rxjs/operators';

import { Employee } from './Employee';

@Injectable({providedIn: 'root'})

export class EmployeeService {

// Base url

baseurl = 'http://localhost:3000/employees/';

constructor(private http: HttpClient) { }

// Http Headers

httpOptions = {

headers: new HttpHeaders({

'Content-Type': 'application/json'

})

}

// POST

CreateEmployee(data): Observable<Employee> {

return this.http.post<Employee>(this.baseurl , JSON.stringify(data), this.httpOptions)

.pipe(

retry(1),

catchError(this.errorHandl)

)

}

// GET

GetEmployee(id): Observable<Employee> {

return this.http.get<Employee>(this.baseurl + id)

.pipe(

retry(1),

catchError(this.errorHandl)

)

}

// GET

GetEmployees(): Observable<Employee> {

return this.http.get<Employee>(this.baseurl)

.pipe(

retry(1),

catchError(this.errorHandl)

)

}

// PUT

UpdateEmployee(id, data): Observable<Employee> {

return this.http.put<Employee>(this.baseurl + id, JSON.stringify(data), this.httpOptions)

.pipe(

retry(1),

catchError(this.errorHandl)

)

}

// DELETE

DeleteEmployee(id){

return this.http.delete<Employee>(this.baseurl + id, this.httpOptions)

.pipe(

retry(1),

catchError(this.errorHandl)

)

}

// Error handling

errorHandl(error) {

let errorMessage = '';

if(error.error instanceof ErrorEvent) {

// Get client-side error

errorMessage = error.error.message;

} else {

// Get server-side error

errorMessage = `Error Code: ${error.status}\nMessage: ${error.message}`;

}

console.log(errorMessage);

return throwError(errorMessage);

}

}

Let's make an **HTTP POST Request** to add one employee to the employees array in the local server using HttpClient service.

In the app.component.ts file,

export class AppComponent implements OnInit{

constructor(private employeeService : EmployeeService){}

new\_employee = {

"id" : "18",

"name" : "Grace",

"age" :"22"

};

ngOnInit(){

this.employeeService.CreateEmployee(this.new\_employee)

.subscribe(data =>{

console.log("Post Request for creating new employee");

console.log("id: " + data.id);

console.log("name: " + data.name);

console.log("age: " + data.age);

}

);

}

}

//Logs:

//Post Request for creating new employee

// id: 18

// name: Grace

// age: 22

Now let's make an **HTTP GET Request** to get a specific employee details from the employees array in the local server using HttpClient service.

In the app.component.ts file,

export class AppComponent implements OnInit{

constructor(private employeeService : EmployeeService){}

employee : Employee;

ngOnInit(){

this.employeeService.GetEmployee(12)

.subscribe(data =>{

console.log("GET Request to get a employee with id - 12");

console.log("name: " + data.name);

console.log("age: " + data.age);

}

);

}

}

//Logs:

//GET Request to get a employee with id - 12

// name: Chris

// age: 22

Let's make an **HTTP GET Request** to get all the employee details in the employees array.

In the app.component.ts file,

@Component({

selector: 'app-root',

template: `

<div>

<h2> Employee List</h2>

<ul \*ngFor = "let emp of employees">

<li>{{ emp.id }} - {{ emp.name }} - {{emp.age}}</li>

</ul>

</div>

`,

styleUrls: ['./app.component.css']

})

export class AppComponent implements OnInit{

constructor(private employeeService : EmployeeService){}

employees : Employee;

ngOnInit(){

this.employeeService.GetEmployees()

.subscribe(data =>{

this.employees = data;

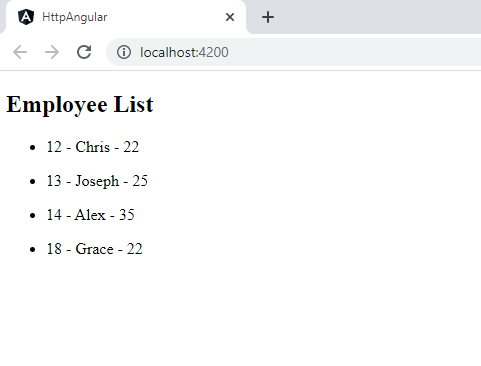
}

);

}

}

Output:



Let's make an **HTTP PUT Request** to update a specific employee details in the employees array.

export class AppComponent implements OnInit{

constructor(private employeeService : EmployeeService){}

employee = {

"name": "Adam",

"age": "28"

}

ngOnInit(){

this.employeeService.UpdateEmployee(18, this.employee )

.subscribe(data =>{

console.log("PUT Request to update the employee with id - 18");

console.log("updated name :" + data.name);

console.log("updated age :" + data.age);

}

);

}

}

//Logs:

//PUT Request to update the employee with id - 18

//updated name :Adam

//updated age :28

Let's make an **HTTP DELETE Request** to delete a specific employee in the employees array.

export class AppComponent implements OnInit{

constructor(private employeeService : EmployeeService){}

ngOnInit(){

this.employeeService.DeleteEmployee(18)

.subscribe(data =>{

console.log("DELETE Request to delete the employee with id - 18");

console.log(data);

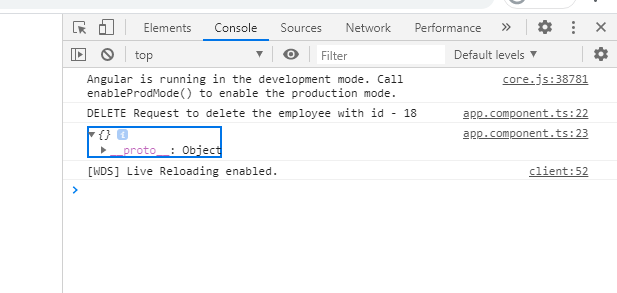
}

);

}

}

Console Logs:



We can request a specific employee's details by passing the id in the request URL. If the id in the request URL is not present in the employee array, it results in an server- side error. These errors can be handled by the error handler method defined in the EmployeeService.

//Here, we make get request for the already deleted employee record which returns HTTP error response.

ngOnInit(){

this.employeeService.GetEmployee(18)

.subscribe(data =>{

console.log("GET Request to get a employee with id - 18");

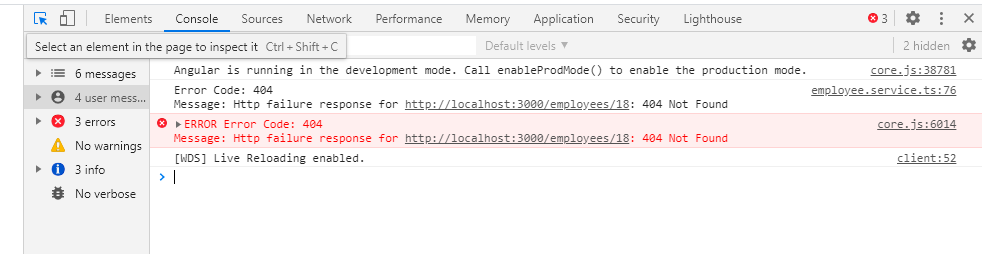
console.log(data);

}

);

}

Console Logs:



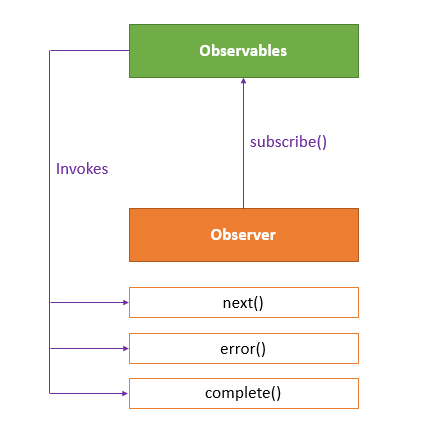
**RxJS - Observables**

## Observables

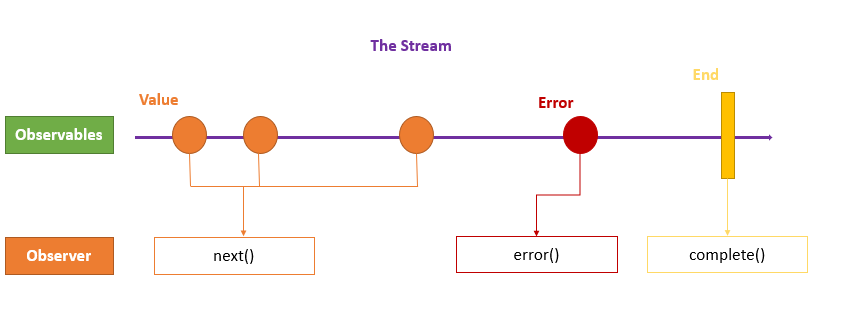
Observables provide support for passing messages between parts of your application. We can use observables for event handling, asynchronous programming, and handling multiple values.

The obserser pattern is similar to the [publish/subscribe](https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern) design pattern. Observables define a function for publishing values, but it is not executed until a consumer subscribes to it. The subscribed consumer then receives notifications until the function completes, or until they unsubscribe.

An observable can deliver multiple values of any type (literals, messages, or events). The API for receiving values is the same whether the values are delivered synchronously or asynchronously and the setup and the logic are both handled by the observable. We'll only concentrate on subscribing to consume values, and when done, unsubscribing. The observer subscribes to the observable to receive the value. To execute the observable we have created and begin receiving notifications, we call its subscribe() method, passing an observer. The unsubscribe() method is called to stop receiving notifications.



The observer defines three callback methods - next(), error(), and complete(). The observable invokes the next() callback whenever the value arrives in the stream. It passes the value as the argument to the next callback. If the error occurs, then the error() callback is invoked. It invokes the complete() callback when the stream completes.



**Example:**

import {Observable, of } from "rxjs";

export class AppComponent {

// Create simple observable that emits three values

myObservable = of(1, 2, 3);

// Create observer object

myObserver = {

next: x => console.log('Observer got a next value: ' + x),

error: err => console.error('Observer got an error: ' + err),

complete: () => console.log('Observer got a complete notification'),

};

// Execute with the observer object

constructor(){

this.myObservable.subscribe(this.myObserver);

}

// Logs:

// Observer got a next value: 1

// Observer got a next value: 2

// Observer got a next value: 3

// Observer got a complete notification

}

**NOTE:** The of(...items)returns an Observable instance that synchronously delivers the values provided as arguments.

### **Promises vs Observables**

In Angular, we can use either Promises or Observables for handling asynchronous data.

**The difference is** - a Promise emits a single value while Observable can emit multiple values. So, while handling a HTTP request, a Promise can manage a single response for the same request, but if there are multiple responses to the same request, then we have to use an Observable.

**Example:**

const promise = new Promise((data) =>{

data(1);

data(2);

data(3); }).then(element => console.log('Promise '+ element));

// Logs:

// Promise 1

const observable = new Observable((data) => {

data.next(1);

data.next(2);

data.next(3); }).subscribe(element => console.log('Observable ' + element));

// Logs:

//Observable 1

//Observable 2

//Observable 3

**Setting up routing**

Routing

In Single Page Applications, only one page is requested from the server, and it provides multiple views dynamically rather than loading the new pages from the server.

The Router mechanism in Angular provides a way to navigate from one view to another view in the application.

Angular provides a RouterModule that has the necessary service providers and directives for navigating through application views. The **router** defines navigation of views on a single page and interprets URL links to determine which views to create or destroy, and which components to load or unload. A **routing component** imports the Router module, and its template contains a RouterOutlet element where it can display views produced by the router.

**Example:**

* Run the ng new routing-app --routing command to generate a basic Angular app with an app routing module, where we can configure our routes.
* To use the Angular router, an app needs to have at least two components so that it can navigate from one to the other. Run these command ng g c first and ng g c second to generate 2 components - *FirstComponent* and *SecondComponent*.
* Make sure that in index.html file we have <base href="/"> in the <head> section. The href attribute is set to "/" so that the application constructs the URL while navigating.
* We need to import these two components in the *app-routing.module.ts* file.

import { FirstComponent } from './first/first.component';

import { SecondComponent } from './second/second.component';

* Also, import the *AppRoutingModule* into *AppModule* and add it to the import array. The Angular CLI performs import this by default. When we implement the routing feature in the existing application, we need to import *AppRoutingModule* into *AppModule*.
* In the app routing module, the CLI creates a Routes array used to define our routes.

import { NgModule } from '@angular/core';

import { Routes, RouterModule } from '@angular/router';

const routes: Routes = [];

@NgModule({

imports: [RouterModule.forRoot(routes)],

exports: [RouterModule]

})

export class AppRoutingModule { }

* Each route in the Routes array is an object with two properties:
  + *path* - URL path for the route.
  + *component* - the corresponding angular component for the specified path.

const routes: Routes = [

{ path: 'first-component', component: FirstComponent },

{ path: 'second-component', component: SecondComponent },

];

* <router-outlet> - works as a placeholder to load the different components dynamically based on the activated component.
* *routerLink* - is an attribute to an anchor tag which sets the route for the component.
* In the *AppComponent*, we add our routes to the application. Angular updates the view depending upon the selected route.

<h1>Angular Router App</h1>

<nav>

<ul>

<li><a routerLink="/first-component" routerLinkActive="active">First Component</a></li>

<li><a routerLink="/second-component" routerLinkActive="active">Second Component</a></li>

</ul>

</nav>

<router-outlet></router-outlet>

**RxJS - Subjects**

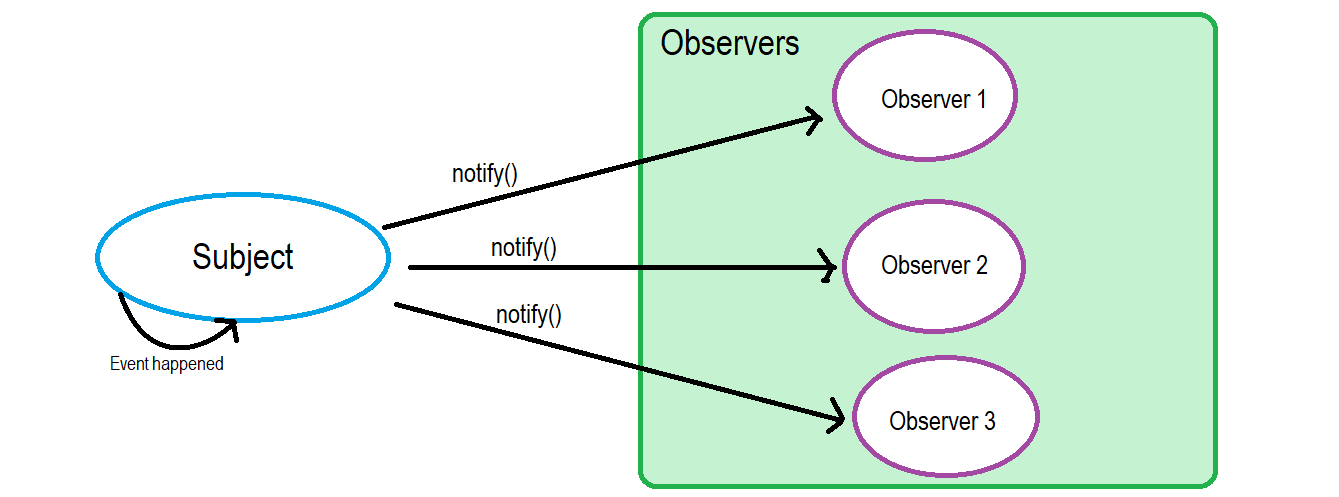
## RxJS - Subjects

**R**eactive E**x**tensions for **J**ava**S**cript([RxJS](https://rxjs.dev/guide/overview" \t "_blank)) is a framework for reactive programming using observables that makes it easier to write asynchronous code.

An RxJS Subject is a special type of Observable that allows values to be multicasted to many Observers.

By default an RxJS Observable is unicast, i.e. each subscribed observer has an independent execution of the Observable, whereas multicast means that the Observable execution is shared by multiple Observers.

Every Subject is an Observer. It is an object with the methods next(v), error(e), and complete(). To feed a new value to the Subject, just call next(theValue), and it will be multicast to the Observers registered to listen to the Subject. We subscribe to the Subject to recieve values normally.



**Example:** Here, we have two Observers attached to a Subject and we feed some values to the Subject.

import { Subject } from "rxjs";

export class AppComponent implements OnInit{

ngOnInit(){

const subject = new Subject();

//First Observer

subject.subscribe({

next: (data) => console.log('First observer prints '+ data)

});

subject.next(1);

//Second Observer

subject.subscribe({

next: (data) => console.log('Second observer prints '+ data)

});

subject.next(34);

subject.next(14);

}

//Logs:

//First observer prints 1

//First observer prints 34

//Second observer prints 34

//First observer prints 14

//Second observer prints 14

}

In this example, we have two observers for the subject that returns data values. If we notice, the second observer did not receive the very first next value because the subject simultaneously holds and efficiently distributes the values according to scope and definition.

## Subject Variants

There are 3 subject variants:

### **Behavior subject**

It used to temporarily store the current data value of any observer declared before it. **Example:**

ngOnInit(){

const subject = new BehaviorSubject(0);

//First Observer

subject.subscribe({

next: (data) => console.log('First observer prints '+ data)

});

subject.next(1111);

subject.next(2222);

//Second Observer

subject.subscribe({

next: (data) => console.log('Second observer prints '+ data)

});

subject.next(3333);

}

//Logs:

//First observer prints 0

//First observer prints 1111

//First observer prints 2222

//Second observer prints 2222

//First observer prints 3333

//Second observer prints 3333

Here, subject stores last data value '2222' and then reports by the new observer even though it was defined after the reference to it. Also, the new constructor takes an initial value.

### **Replay subject**

Replay Subject provides a option to choose how many values we want to emit from the last observer. This subject stores and then passes the last specificed option values to the new observer. **Example:**

ngOnInit(){

const subject = new ReplaySubject(2);

//First Observer

subject.subscribe({

next: (data) => {

return console.log('First observer prints ' + data);

}

});

subject.next(1111);

subject.next(2222);

//Second Observer

subject.subscribe({

next: (data) => {

return console.log('Second observer prints ' + data);

}

});

subject.next(3333);

}

//Logs:

//First observer prints 1111

//First observer prints 2222

//Second observer prints 1111

//Second observer prints 2222

//First observer prints 3333

//Second observer prints 3333

Here, subject stores last 2 data values ('1111' & '2222') and then reports them to the new observer.

### **Async subject**

A Async Subject emits the last value to observers when the sequence is completed.

ngOnInit(){

const subject = new AsyncSubject();

//First Observer

subject.subscribe({

next: (data) => console.log('First observer prints '+ data)

});

subject.next(1);

subject.next(2);

//Second Observer

subject.subscribe({

next: (data) => console.log('Second observer prints '+ data)

});

subject.next(3);

subject.complete();

}

//Logs:

//First observer prints 3

//Second observer prints 3

Here, we execute the subject only after a complete method is called.

**Pub/Sub design pattern**

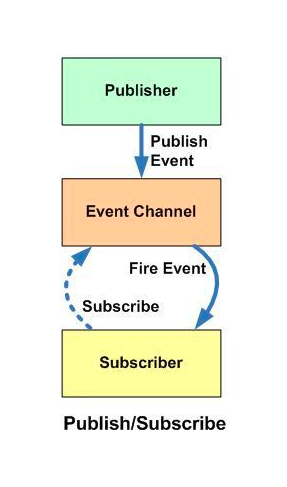
Publisher/Subscriber Design Pattern

The Publisher/Subscriber design pattern describes the flow of messages between applications, devices, or services.

A message is published by **Publishers** to a **Channel**, that will be consumed by all **Subscribers** monitoring that channel.

When the Publisher pushes messages to a channel (live-feed data streams), the subscribers who subscribed to this channel are immediately notified. Any publisher may also be a subscriber. Messages can be text, sensor data, audio, video, or other digital content.

The Pub-Sub pattern is usually implemented in an asynchronous way. In the Observer pattern the observers are aware of the observable, but in Pub-Sub pattern, publishers and subscribers don’t need to know each other. They simply communicate with the help of message queues.



**Example - Pub-Sub pattern**

class PubSub {

constructor() {

this.handlers = [];

}

//publisher publishes the topic to the channel

publish(event, args) {

this.handlers.forEach(topic => {

if (topic.event === event) {

topic.handler(args)

}

})

}

// subscriber gets notifications when there is a new feed in the subscribed channel

subscribe(event, handler, context) {

if (typeof context === 'undefined') { context = handler; }

this.handlers.push({ event: event, handler: handler.bind(context) });

}

}

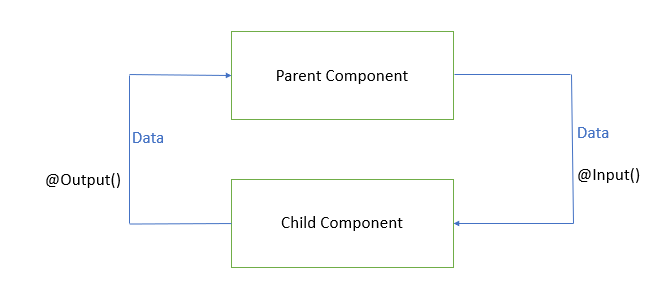
export default new PubSub();

**Event Emitters**

# **Event Emitters in Angular**

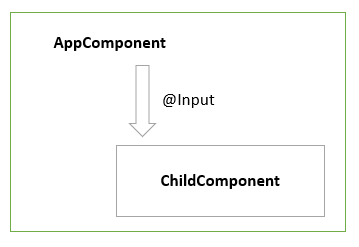
An [EventEmitter](https://angular.io/api/core/EventEmitter" \t "_blank) is used to emit custom events synchronously or asynchronously, and register handlers for those events by subscribing to an instance.

Angular uses @Input and @Output decorators to flow data between components. We can also use Angular services to flow data between the components. If we have to pass data into a component we use the @Input decorator, and if we have to emit the event or data from a component we use @Output decorator with the EventEmitter API.

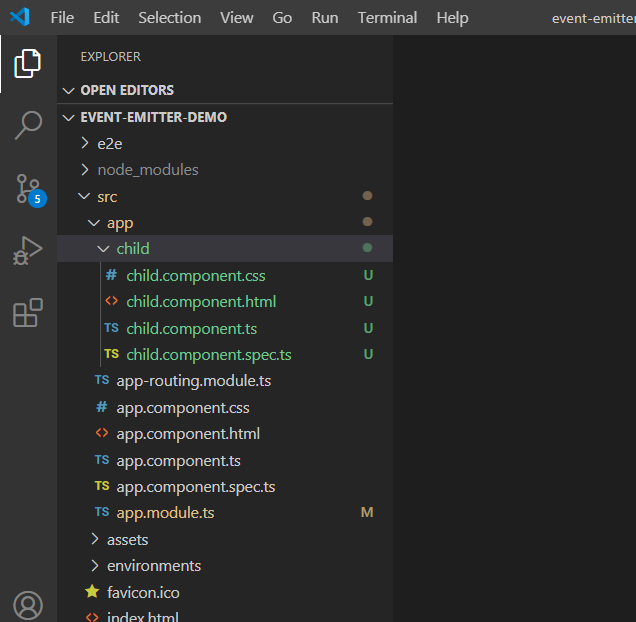


## @Input decorator

In Angular, the [@Input](https://angular.io/api/core/Input) decorator is defined in the [@angular/core](https://angular.io/api/core) package that marks a class field as an **input property** and supplies configuration metadata.



**Example:** Let us create an angular application with at least one child component. Run the ng new event-emitter-demo CLI command to create an Angular application. Here, we already have an AppComponent considered as a parent component. Run the ng g component child CLI command to create an child component for the AppComponent.



In child.component.ts, we create a count property and decorate with the @Input(), which implies that value of count property will be set outside from the ChildComponent.

import { Component, Input } from '@angular/core';

@Component({

selector: 'app-child',

template: `

<p> Data from the AppComponent --- count = {{count}}</p>

`

})

export class ChildComponent {

@Input() count: number;

}

In app.component.ts, we use **property binding** to pass the count property value from the AppComponent to the ChildComponent.

import { Component } from '@angular/core';

@Component({

selector: 'app-root',

template: `

<h2>@Input Example</h2>

<app-child [count]='count'></app-child>

`

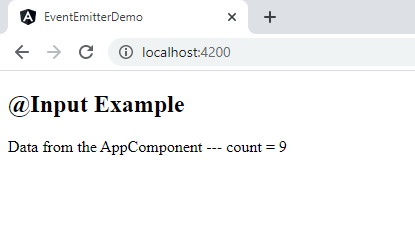
})

export class AppComponent {

count = 9;

}

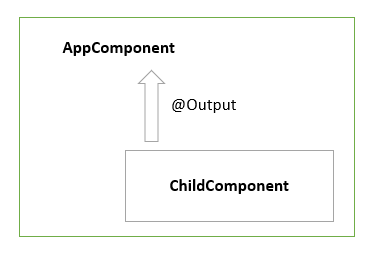
When we run this application, we are able to see the below output:



## Component events with EventEmitter and @Output

In Angular, a component can emit an event using [@Output](https://angular.io/api/core/Output) and [EventEmitter](https://angular.io/api/core/EventEmitter" \t "_blank) API in the [@angular/core](https://angular.io/api/core) package.

@Output decorator that marks a class field as an **output property** and supplies configuration metadata.



**Example:** To emit data and event out from a component, we create an instance of EventEmitter and annotate that property and with @Output decorator. This instance calls emit() method to emit a payload which can be received by an event object $event.

In child.component.ts, we create a change property and decorate with the @Output()and bound a new instance of EventEmitter to it.

Also, we have two methods - increment() and decrement() which updates the value of the count property based on the event (clicking on the increment count/ decrement count buttons) and emits the event changes to its parent component.

Here, the change property calls the emit() method that emits the count value which can be received by event object $event.

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

@Component({

selector: 'app-child',

template: `

<p> Click this button to increment the count: </p>

<button (click)='increment()'>increment count</button> <br>

<p> Click this button to decrement the count: </p>

<button (click)='decrement()'>decrement count</button>

`

})

export class ChildComponent {

@Input()

count: number = 0;

@Output()

change: EventEmitter<number> = new EventEmitter<number>();

increment() {

this.count++;

this.change.emit(this.count);

console.log("incrementing count in the child component....." + this.count + " --- passing to AppComponent");

}

decrement() {

this.count--;

this.change.emit(this.count);

console.log("decrementing count in the child component....." + this.count + " --- passing to AppComponent");

}

}

In app.component.ts, we use **event binding** to get the count property value from the ChildComponent to the AppComponent.

The .emit() method emits the changes to our (change) event listener we set up in the AppComponent, to which our countChange($event) callback will be invoked, and the data associated with the event will be given to us via the $event property.

We assign this.count with the event that’s passed back inside the countChange() method.

import { Component } from '@angular/core';

@Component({

selector: 'app-root',

template: `

<h2>Event Emitters</h2>

<p> At AppComponent, count = {{ count }} </p>

<app-child [count]='count' (change)= 'countChange($event)'></app-child>

`

})

export class AppComponent {

count = 9;

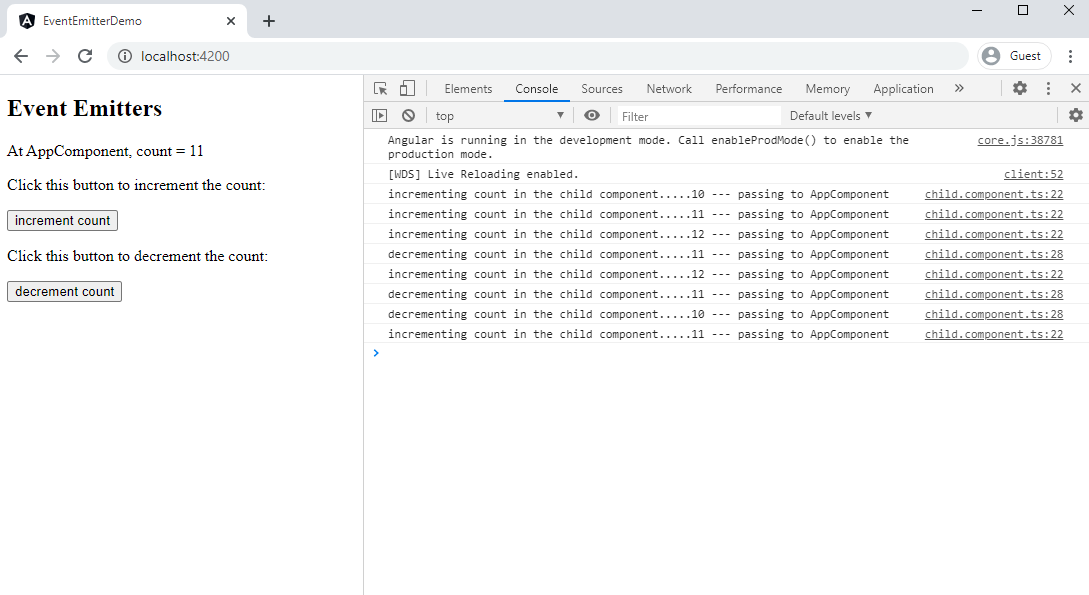
countChange(event) {

this.count = event;

}

}

When we run this application, we are able to see the below output:



In this example, we use property binding to send count value from the parent component to child component and we use custom event binding to get the updated count value from the child component to parent component.

Day 5

**Testing in Angular with Jasmine & Karma**

## Testing in Angular with Jasmine & Karma

Testing often for each feature is important because it helps the design of implementations. It allows for refactoring (changes to pre-existing code). It also helps while adding brand new features since you can ensure you are not breaking any existing code by testing. Tests are also useful documentation and helps make developers more confident about their code. Angular uses the [Jasmine test framework](https://jasmine.github.io/) and [Karma](https://karma-runner.github.io/1.0/index.html), a task runner, for running the tests.

##### **Jasmine**

Jasmine is a behavior-driven development framework for testing JavaScript code. It does not depend on any other JavaScript frameworks. It does not require a DOM. And it has a clean, obvious syntax so that you can easily write tests.

##### **Karma**

Karma uses a configuration file in order to set the startup file, the reporters, the testing framework, the browser among other things.

##### **Step By Step guide to Component Testing**

* Create the function you want to create in the appropriate component.

@Component({

selector: 'lightswitch-comp',

template: `

<button (click)="clicked()">Click me!</button>

<span>{{message}}</span>`

})

export class LightswitchComponent {

isOn = false;

clicked() { this.isOn = !this.isOn; }

get message() { return `The light is ${this.isOn ? 'On' : 'Off'}`; }

}

In this case the function is called clicked and it toggles a light switch on and off when the user clicks the button.

* Go to the automatically generated spec.ts file in the component.

describe('LightswitchComp', () => {

it('#clicked() should toggle #isOn', () => {

const comp = new LightswitchComponent();

expect(comp.isOn).toBe(false, 'off at first');

comp.clicked();

expect(comp.isOn).toBe(true, 'on after click');

comp.clicked();

expect(comp.isOn).toBe(false, 'off after second click');

});

it('#clicked() should set #message to "is on"', () => {

const comp = new LightswitchComponent();

expect(comp.message).toMatch(/is off/i, 'off at first');

comp.clicked();

expect(comp.message).toMatch(/is on/i, 'on after clicked');

});

});

Now there is a test for the click function. When testing using Karma, this test should be automatically detected and run.

* Run all tests using 'ng test' This should run all the tests present and output something similar to below on the console.

10% building modules 1/1 modules 0 active

...INFO [karma]: Karma v1.7.1 server started at http://0.0.0.0:9876/

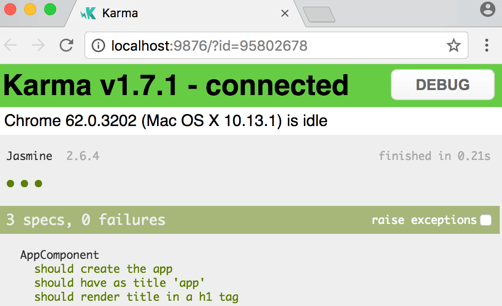
...INFO [launcher]: Launching browser Chrome ...

...INFO [launcher]: Starting browser Chrome

...INFO [Chrome ...]: Connected on socket ...

Chrome ...: Executed 3 of 3 SUCCESS (0.135 secs / 0.205 secs)

A web browser tab should open with the Jasmine HTML Reporter like the picture below.



Testing of services is similar. For other cases please reference [Angular Docs Extra Testing](https://angular.io/guide/testing#more-information-on-testing)

**Angular Review**

## Angular Review

Angular is a **TypeScript-based open-source framework** used to develop **dynamic web applications**.

Angular is a Single Page Application (SPA) Framework. An SPA is a web application that fits on a single page. All our code (JavaScript , HTML , CSS ) is retrieved with a single page load. Navigation between pages is performed without refreshing the whole page.

##### **AngularJS vs Angular**

[AngularJS](https://en.wikipedia.org/wiki/AngularJS) is a **JavaScript-based open-source front-end web framework**. It doesn't support the features of a server-side programming language, nor dynamic loading of the page.

[Angular](https://en.wikipedia.org/wiki/Angular_(web_framework)) is a **complete rewrite** of AngularJS. AngularJS was completely based on **controllers** and **scopes** whereas, Angular uses **component hierarchy** as its main architecture.

##### **Angular CLI**

The [Angular CLI](https://cli.angular.io/) is a command-line interface for Angular that helps us to get started with creating an Angular application. Angular CLI creates an Angular application and uses the [Typescript](https://app.revature.com/) programming language, [Webpack](https://app.revature.com/) for Module bundling, Karma for unit testing, and Protractor for end-to-end testing. The Angular CLI takes care of the configuration and initialization of various libraries. It also allows us to add components, directives, services, etc, to already existing Angular applications.

##### **Angular File Structure**

Generally, We use Visual Studio Code or Webstrom as a Code Editor for creating and editing Angular Applications. You can download and install Visual Studio Code from this website: <https://code.visualstudio.com/download>

The file structure of the Angular application described below:

The **e2e** folder at the top level contains source files for a set of end-to-end tests and test-specific configuration files. The **node\_modules** folder provides npm packages to the entire workspace. The **src** folder contains the source files which give information about application logic, data, and assets.

* **app** - this folder contains the component files.
  + **app.component.ts** - used to define the logic for the app's root component (AppComponent).
  + **app.component.html** - used to define the HTML template associated with the root AppComponent.
  + **app.component.css** - used to define the base CSS stylesheet for the root AppComponent.
  + **app.component.spec.ts** - used to define the unit test for the root AppComponent.
  + **app.module.ts** - used to define the root module (AppModule) and helps the Angular to assemble the application. All components, including the AppComponent, must be declared inside the AppModule.
* **assets** - this folder contains image and other asset files.
* **environments** - this folder contains build configuration options for particular target environments.
* **favicon.ico** - An icon to used for an application in the bookmark bar.
* **index.html** - The main HTML page that is served when someone visits your site. The CLI automatically adds all JavaScript and CSS files when building your app, so you typically don't need to add any <script> or <link> tags here manually.
* **main.ts** - The main entry point for an application. Compiles the application with the JIT compiler and bootstraps the application's root module (AppModule) to run in the browser.
* **polyfills.ts** - Provides polyfill scripts for browser support.
* **styles.css** - Lists CSS files that applies the styles for a project.
* **test.ts** - The main entry point for unit tests used in the application.
* **.editorconfig** - this file contains configuration for code editors.
* **.gitignore** - it specifies untracked files that Git should ignore.
* **angular.json** - holds CLI configuration defaults for all projects in the workspace. It includes configuration options for the build, serve, and test tools.
* **browserslist** - used to configure the sharing of target browsers and Node.js versions among various front-end tools.
* **karma.conf.js** - it contains application-specific Karma configuration.
* **package-lock.json** - this provides version information for all packages installed into node\_modules by the npm client.
* **package.json** - used to configure npm package dependencies that are available to all projects in the workspace.
* **README.md** - An introductory documentation for the root app.
* **tsconfig.app.json** - it holds application-specific TypeScript configuration, including TypeScript and Angular template compiler options.
* **tsconfig.json** - holds default TypeScript configuration for projects in the workspace.
* **tslint.json** - holds default TSLint configuration for projects in the workspace. TSLint is an extensible static analysis tool that checks TypeScript code for readability, maintainability, and functionality errors.

##### **TypeScript**

* [Typescript](https://www.typescriptlang.org/) is a **typed superset of JavaScript**.
* It is used by Angular.
* It is an **open-source** and **object-oriented programming language** that supports classes, interfaces, inheritance, modules, etc., developed by **Microsoft**.
* It can be used to **manipulate the DOM** for adding or removing elements, similar to JavaScript.
* It is **portable** across browsers, devices, and operating systems.
* It supports **strong typing** or **static typing**, unlike JavaScript.
* TypeScript files are saved with a .ts extension and then compiled into JavaScript using the TypeScript compiler.
* TypeScript gets compiled to JavaScript, which can run on any JavaScript runtime environment.(Eg: Node.js)

##### **Angular Components**

Components are the basic building blocks in the Angular application. Components contain the data & UI logic that defines the view and behavior of the web application.

Components in Angular are defined using a [@Component](https://angular.io/api/core/Component) decorator. It includes a selector, template, style, and other properties, and it specifies the metadata required to process the component.

Angular applications can have multiple components. Each component handles a small part of UI. These components work together to produce the complete user interface of the application. An Angular application has one **root component** (AppComponent) which is specified in the bootstrap array under the main **ngModule** module defined in the **app.module.ts** file.

##### **@NgModule**

Every Angular application consists of at least one module, the root module. We bootstrap that module to launch the application.

NgModules are TypeScript classes decorated with the [@NgModule](https://angular.io/api/forms/NgModel) decorator imported from the @angular/core package.

NgModule takes metadata and describes how to compile a component's template and how to create an injector at runtime. It identifies the module's own components, directives, and pipes and makes them public through the export property which can be used by external components.

##### **Angular Routing**

In Single Page Applications, only one page is requested from the server, and it provides multiple views dynamically rather than loading the new pages from the server.

The Router mechanism in Angular provides a way to navigate from one view to another view in the application.

Angular provides a RouterModule that has the necessary service providers and directives for navigating through application views. The **router** defines navigation of views on a single page and interprets URL links to determine which views to create or destroy, and which components to load or unload. A **routing component** imports the Router module, and its template contains a RouterOutlet element where it can display views produced by the router.

##### **Conclusion**

Angular is a great tool for building web applications. You will be using it to display dynamic web pages which link back to your backend. It can be hard to navigate but provides quality tools for testing, storing objects, communicating with backend and stylizing.

**Introduction to SDLC**

# **SDLC - Software Development Life Cycle**

### **Helpful References/Links**

* [SDLC - Waterfall Model (tutorialspoint)](https://www.tutorialspoint.com/sdlc/sdlc_waterfall_model.htm)
* [SDLC - Agile Model (tutorialspoint)](https://www.tutorialspoint.com/sdlc/sdlc_agile_model.htm)
* [Scrum - Framework (tutorialspoint)](https://www.tutorialspoint.com/scrum/scrum_framework.htm)
* [Kanban (Atlassian)](https://www.atlassian.com/agile/kanban)
* [The Benefits and Pitfalls of Pair Programming in the Workplace (freeCodeCamp)](https://www.freecodecamp.org/news/the-benefits-and-pitfalls-of-pair-programming-in-the-workplace-e68c3ed3c81f/)
* [Story points and estimation (Atlassian)](https://www.atlassian.com/agile/project-management/estimation)
* [Extreme Programming (Agile Alliance)](https://www.agilealliance.org/glossary/xp/#q=~(infinite~false~filters~(postType~(~'post~'aa_book~'aa_event_session~'aa_experience_report~'aa_glossary~'aa_research_paper~'aa_video)~tags~(~'xp))~searchTerm~'~sort~false~sortDirection~'asc~page~1))

## Software Development Lifecycle (SDLC)

SDLC is the series of steps that we go through when creating new products. It encompasses methodologies (broader categories of development concepts and practices) and frameworks (a more detailed implementation of a methodology's ideas). We will be discussing the Waterfall and Agile methodologies in addition to the Scrum, Kanban, Scrumban, and eXtreme Programming frameworks (all of which are frameworks of the Agile methodology). Although the order/repetition of the steps in the two methodologies is different, they both contain the same overarching steps that are characteristic of SDLC as a whole; these are gathering requirements, analyzing those requirements, designing a solution, developing the solution, testing the solution, user acceptance testing (UAT), releasing the solution, and finally maintaining the product. These eight steps can be categorized into three phases: the design phase (gathering/analyzing requirements and designing the solution), the development phase (developing, testing, and UAT), and the deliver phase (release and maintenance). We'll begin our journey into the methodologies and frameworks by first discussing the methodologies and how they implement/order these steps and phases.

## The Waterfall Methodology

Waterfall is a very rigid methodology that follows a strict series of steps. Waterfall practitioners follow each of the eight SDLC steps in order, never revisiting a previous stage; just like a natural waterfall, the methodology only flows in one direction. E.g. once development has begun, Waterfall practitioners accept no new requirements from the client. In fact, this exclusion of the client is one of the major drawbacks to Waterfall. Let's further explore Waterfall's drawbacks and benefits.

### **Waterfall Drawbacks**

As stated in the previous section, the Waterfall methodology excludes the client in every step between requirement gathering and deployment - your client better have given you a complete set of requirements at the beginning because once development starts, you're not accepting any new feature requests. Additionally, you should pray that they are satisfied with the end result because once you deploy the product, you are only maintaining existing features - never adding new ones. The one-direction flow makes changes to previous stages impossible. Finally, Waterfall practitioners do not test until all development has been completed, so a bug or problem that may have been developed months prior will not be noticed until the entire project has entered the testing phase.

### **Benefits of Waterfall**

The same rigidness that hinders the methodology can also work to its advantage. There is no room for confusion in Waterfall because there is a clear barrier between each step, making it easy for team members to keep track of where they are in the project. This clear barrier also allows for a clean transfer of knowledge between steps. For example, when moving from development to testing, team members will already know every aspect of the project that will ever need to be tested; because there is no revisiting past steps, they do not have to worry about any future functionality in need of scrutiny. This clarity also applies to the entire Waterfall lifecycle - programmers have an understanding of what the finished product will look like and do as soon as the design phase has been completed because all requirements are gathered before design (or development, for that matter) begins. Lastly, unlike the Agile methodology (which we will detail in a bit), Waterfall does not require any specific procedural knowledge - followers simply intuitively move from one step and phase to the next.

## Term Definitions

Before we dive into Agile, let's define some basic terms that will help us understand and intelligently discuss Agile planning and analysis.

**User Story**: An individual feature of/requirement for a project in Agile development.

**Epic**: A group of related features that is broken down into multiple user stories.

**Story Point**: A level of difficulty assigned to a user story through the use of a sequence of numbers that increases with increasing difficulty.

**Sprint**: A brief period of development (almost always less than four weeks, sometimes as short as one week) generally culminating in a release of related features.

**Velocity**: The sum of story points of all user stories completed during a sprint. Velocity allows Agile teams to more accurately predict how many user stories can be completed in future sprints.

## The Agile Methodology

Where Waterfall is rigid, Agile is fluid. This fluidity has brought popularity, largely owing to the ability to adjust to changing requirements and requests from the client. For now, let's talk about the benefits and drawbacks of the methodology as a whole. Later, We will explore four Agile frameworks.

### **Agile Drawbacks**

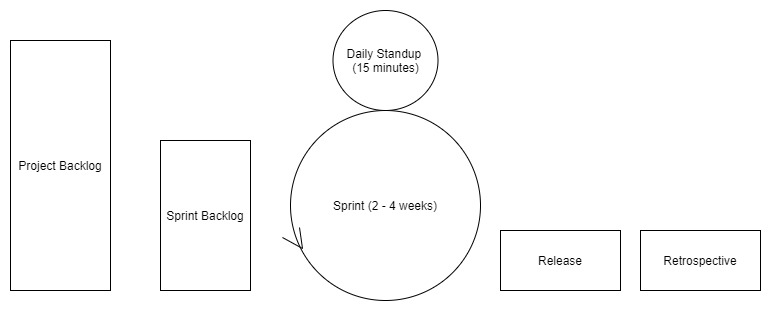
Although flexibility can be advantageous, it can also lead to bad practices. For example, developers may be tempted to avoid fully designing their solutions prior to implementation because they know that the methodology allows for revisiting past SDLC steps. This is not how Agile should be utilized. Rather, the Agile fluidity should be used to adapt to changing requirements from the client and unforeseen circumstances that are not simply the result of poor, or nonexistent, planning. Allowing for change can also encourage unpredictability - another drawback of the methodology. Finally, as we will detail when talking about the Agile frameworks, properly following Agile practices necessitates knowledge of those practices - the methodology is more complicated than the comparatively simple Waterfall approach.

### **Benefits of Agile**

Although we highlighted the dangers of allowing for changing feature requests in the last section, the ability to handle and respond to these shifts is a benefit. Additionally, the Agile flexibility extends to release times - because the Agile release cycle is iterative, any features that cannot be included in the upcoming release can simply be made as part of the following deployment. Multiple releases also allow for faster movement through the testing phase because developers are testing a small set of features, not the entire project at once as in Waterfall. Lastly, the methodology allows for initial ambiguity. For example, suppose Feature B is built on top of Feature A. Because we have the ability to put both features in separate releases, we can wait until the release containing Feature A is deployed and we are in the design phase of the following iteration before designing Feature B. This allows for the solution to Feature B to be modified based on an implementation of Feature A that may be different than we had originally planned.

### **The Scrum Framework**

We will first explore Scrum, arguably the simplest of the Agile frameworks. It enforces certain ceremonies, led by a Scrum Master who ensures that the team is following Scrum practices. Development is broken into two to four week sprints.



The above image shows the Scrum lifecycle. Before the sprint begins, the team moves some user stories from the project backlog (the list of user stories for the entire project) to the sprint backlog (the list of features that the team is planning to complete in the upcoming sprint). Throughout the sprint, the Scrum Master leads 15 minute daily standups, daily meetings where each team member details the progress they made the previous day, their plans for the coming day, and any roadblocks they have run into. The standup creates an environment where each team member knows how the project in totality is progressing, even though they may only be working on one component. It also allows for a set time where programmers can ask their peers, who may have more expertise and experience in a specific area, or simply a fresh pair of eyes, for help with any issues. As any programmer can attest, there are some bugs that you will never find by yourself.

After the product is released at the end of each sprint, the team holds a final Scrum ceremony: the retrospective. Here, teams put any unaccomplished tasks back into the project backlog and discuss what went well in the sprint and how to continue that success, as well as what was unsuccessful and how to improve on those setbacks.

#### **Scrum Advantages**

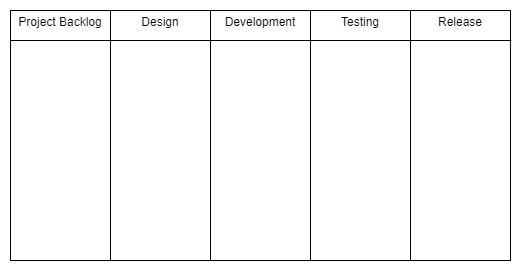
Daily standups bring many benefits, particularly the ability to quickly identify any setbacks that developers on the team are experiencing and then quickly solve those problems with the help of others. The relatively short sprints allow for constant releases, which in turn incorporate the client throughout the process of developing the entire project. Involving the client allows for the team to adjust to new client feedback and feature requests, which results in an end product that more accurately reflects their client's desires, leading to increased client satisfaction.

#### **Scrum Disadvantages**

As we touched on when discussing Agile, the very fluidity that gives Scrum benefits also brings in uncertainty. Because the final deadline and complete requirement list are not known at the beginning of the project, it is difficult to estimate costs or a final release date. Additionally, this fluidity results in feature creep, a continuous expansion of proposed functionalities. Feature creep occurs because Scrum followers are held to no firm deadlines, so there is no reason to exclude certain user stories in order to stick to a release date and they can unceasingly add new features, never truly completing the project. Finally, Scrum's ceremonies require engagement from all team members in order to properly function. For example, the daily standup loses its effectiveness if a developer is running into setbacks but never voices them to the group.

### **Kanban**

The Kanban board, a vehicle for a visual representation of the progress of a project, is the hallmark trait of the Kanban framework. The image below shows an empty example Kanban board.



There is no firm requirement for the number of columns on the board, but there should generally be at least one column per SDLC phase. Kanban cards, visual representations of a single user story or work item, are placed in the columns. Team members who have completed their current task can simply refer to the board and choose a new card from the backlog to work on. There should never be more tasks in progress - i.e. not in the backlog/equivalent column or release/equivalent column - than there are developers on the team.

#### **Kanban Advantages**

Kanban is event-driven, removing even the small pressure of sprint deadlines found in the Scrum framework. It also allows for specialists; for example, one team member can be a testing specialist. In this scenario, once user stories have been moved into the testing/equivalent column, the testing specialist can begin working on them and the developer who had previously been responsible for that task can start work on another card in the backlog. Specialization can result in developers building stronger expertise in a particular area. Additionally, the Kanban board can always take new client requirements and is persistent throughout releases. Finally, it allows the entire team to view the current state and progress of the project.

#### **Kanban Disadvantages**

The capacity of the Kanban board also brings drawbacks, particularly the danger that the board's size will grow to the point where it becomes unnecessarily complex and difficult to understand/navigate. Like Scrum, Kanban also requires commitment from its devotees - an out-of-date Kanban board doesn't help (and can even hurt) the team because developers will end up working on the same tasks and will not have an accurate picture of the evolution of the project. But unlike Scrum (with timed sprints) and Waterfall (with a set project deadline from the beginning), this framework has no timing element, which can result in an even more extreme version of feature creep and constant release delays/uncertainty.

### **Scrumban**

We will only briefly touch on Scrumban, a combination of Scrum and Kanban. Scrumban pairs the visual representation of Kanban with the ceremonies and sprints of Scrum. Doing so mitigates Kanban's timing issues with the help of Scrum's sprints. It also gives Scrum practitioners the specialization capabilities found in Kanban (an argument between specialization and cross-functionality can be reserved for another time).

### **eXtreme Programming (XP)**

XP is the most verbose of the Agile frameworks we have examined. There are strict procedures designed to encourage XP's values of communication, simplicity, feedback, courage, and respect. As these values would suggest, this framework seeks to go beyond improving product quality by also improving team cohesion and the lives and happiness of individual members. XP procedures include weekly cycles (one-week sprints), quarterly cycles (i.e. quarterly releases), continuous integration (ensuring that code from new sprints can easily be assimilated into the existing code base), incremental design (initial high-level design of features and functionalities, leaving the specifics for later), the use of user stories and epics, test-driven development (TDD), pair programming, and other practices. In addition, the XP team includes a member of the customer in order to fulfill the goal of faster feedback.

Let's further detail TDD and pair programming because we may be implementing these practices at various points throughout training. TDD is the approach of designing the tests that your code must pass before writing the actual code (in contrast to the usual practice of writing the code then writing the tests), allowing the developer to have a clearer understanding of what their program must accomplish. But take note that using TDD is not an invitation to write test-specific code, your code should still be able to handle a variety of input, not just the input that you know your tests will give.

Pair programming is an approach where teams are broken into groups of two while developing. Each group includes a pilot and a navigator. The pilot is actively writing code, while the navigator is planning how the code that the pilot is writing at that moment will tie into the larger user story, epic, and project codebase. Although this may seem like it will double development time, pair programming is actually more efficient than you may expect because the pairs can collaborate and solve issues faster and narrow their focus due to lessened responsibility.

#### **XP Downsides**

The amount of XP procedures and practices can make following the framework difficult even for dedicated teams. In addition, the involvement of the client on the development team can be a hindrance - non-technically inclined clients may insist on inefficient implementations or question logically sound programmatic approaches. XP's weekly iterations can cause developers to sacrifice general best practices in order to make deadlines. For example, documentation may be written hastily and be incomplete. The frameworks’ short sprints and intimate inclusion of the customer can invite frequent changes, meaning that documentation can quickly become outdated.

Frequent iterations combined with incremental design can also be a dangerous combination as programmers may not have time within a sprint to refine their solutions to maximize code efficiency and good design principles; they may only have time to write code that simply clears the bar set by the customer requirements. Observing these drawbacks as a whole, it becomes clear that ironically, despite its lofty goal of improving employee happiness, this framework can actually be quite stressful to use.

#### **XP Benefits**

That being said, XP is not irredeemable. There can be advantages to such a close relationship with the customer - the constant communication minimizes or even erases the danger of sunk costs due to time spent developing features that, upon delivery, the client decides they no longer want or need. The framework's focus on communication and pair programming can increase the group's cohesion, openness, awareness of the entire project, and commitment to each other. Finally, although its pace can be stressful, the brief development periods and focus on continuous integration result in the quick delivery of working solutions.

### **Agile Planning**

Now that we've explored various Agile frameworks. let's revisit the terms we defined at the beginning of this module and detail how they are used in project planning. At the beginning of a sprint (if using Scrum, Scrumban, or XP) or upon receiving a new feature request (if using Kanban), teams form client requirements into epics and then break those epics into individual user stories. After that process is complete, each user story is assigned story points. In our practices, we will be assigning story points using the Fibonacci sequence - i.e. a 1 will be the absolute easiest task, 2 the next most difficult, followed by 3, 5, 8, etc. Lastly, each story is given a time estimate.

The art of assigning story points and time estimates is just that - an art. It takes practice to gain a solid understanding of what tasks are easier/more difficult than others for your team of developers, so don't be discouraged if your initial attempts are not particularly accurate - that's part of the learning process. Even though these estimates will not be perfect, they allow the group to have a better idea of which tasks will be easier/more difficult and take more/less time, so they can plan accordingly and accomplish goals with more ease than they would by simply flying blindly into development.

## A Home for Waterfall

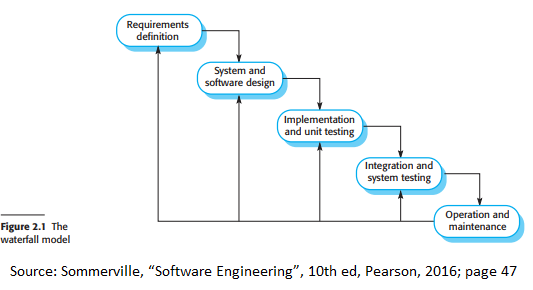
We'll end this module on the same methodology we started with: the Waterfall methodology. For the vast majority of this module, we have explored Agile and its frameworks. This deep focus may lead you to believe that Agile development is always preferred and there is never a scenario appropriate for Waterfall use, but this is not the case. The Waterfall methodology thrives in the right situations, particularly those with a strict deadline for the final release of the product and strict budgetary constraints. Often, these circumstances can be found in government projects - government agencies and departments are under pressure from legislators and executives to stick to an allocated budget and fulfill requests within a set time frame.

Let's broaden the sentiment expressed in the last paragraph - there is no perfect, one size fits all SDLC approach. As you probably noticed throughout our discussion, many times the features of a particular methodology or framework also brought inherent difficulties. Some circumstances may call for one framework, others may necessitate the use of another. Although Agile and particularly Scrum are incredibly popular, the important thing is to find what works for you and your team and allows you to most successfully implement the phases of SDLC.

**Agile vs Waterfall**

## Agile vs Waterfall

##### [**Waterfall model**](https://en.wikipedia.org/wiki/Waterfall_model)



As seen in the image above, the waterfall model has 5 main stages:

* **Requirements definition**
  + The system's components, goals, functionalities, services, and constraints will be determined and written in detail in the documentation. This documentation will be known as the system specification.
* **System and software design**
  + Determine the overall system architecture
* **Implementation and unit testing**
  + This stage involves breaking down the software design into components and verifying whether the components address requirement specifications
* **Integration and system testing**
  + Every component is combined to form a single system and is tested to see if they components mesh well with each other.
* **Operation and maintenance**
  + The system is shipped out and maintenance (improving the system and fixing any bugs that were not discovered earlier.

The waterfall model is a straight forward model where the software process follows a linear pattern; for it to advance to the next stage, the prior stage must be completed. Only after the "operation and maintenance" step is completed can the project go to earlier steps as it sees fit. The waterfall model is commonly known as a plan-driven model.

In trying to develop innovative software, the model has these shortcomings:

* The users don't really know how they want the software to behave without trying it out.
* The analysis and design phases ignore the difficulties that might be faced in actually implementing this innovative idea.
* The implementation phase proceeds without testing all along the way, so problems aren't detected until after it is completed.
* The "environment" the software was designed for will be changing while this long process, with major decisions locked-in at the start, is proceeding. Examples:
* The laws regarding the real-world activity the software is meant to enhance can change.
* The competitive environment of the real-world activity can change.
* The technological landscape assumed by the analysis phase can change.

However, these strict step-by-step processes and lack of foresight into potential future changes do not mean that this model is useless. Because each stage must be fully completed and understood, this model is very beneficial for large projects that multiple teams from multiple sperate locations work on a workflow would be more efficient if each team was on the same page/understanding as each other.

##### [**Agile**](https://www.agilealliance.org/)

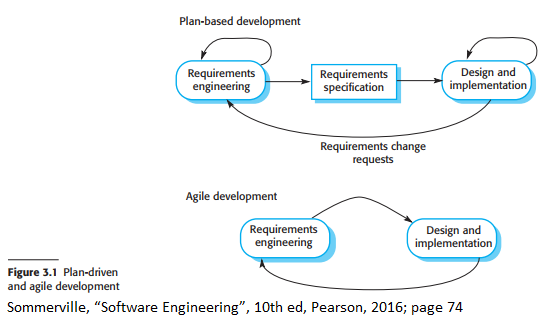
Rapid development and delivery is an important factor for software engineering as software needs to be able to address new changes to requirements. To address this, agile methods were created to put more emphasis on the end product rather than the documentation that may be made along the way. In other words, of working long hours on documentation, the agile approach aimed to cut the overhead caused by the documentation and focus on the development of the actual code.

Every agile method share these characteristics:

* Specification, design, and implementation are mixed as system specification, design documentation, and user requirements are usually minimized to save time during development or are automatically generated.
* Systems are usually developed in increments so that consumers and other stakeholders can give their input during the development phase
* Many tools are used to aid during development. Examples of such tools are tools that automatically test your code, help you make the process of designing and creating the user interface easier, etc.

Agile has the following limitations:

* It is not the ideal solution for small developer teams.
* It has a higher cost compared to other workflow models.
* The project can go off target if the project manager is not sure of the direction to take.

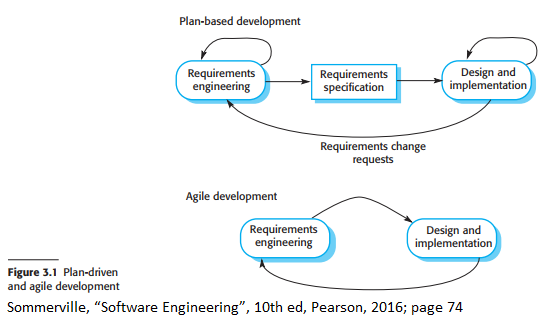


The above image depicts a plan based development lifecycle vs an agile development cycle.

**Agile/Scrum Concepts**

## Agile Concepts

### **Agile**



The image above outlines how plan-driven processes and agile development are different. Plan-driven processes have their iterations within separate stages. Documentation is then created to move to the next stage. However, in the agile approach, the activities requirements and implementation are done at the same time.

Rapid development and delivery is an important factor for software engineering as software needs to be able to address new changes to requirements. Agile methods were developed to account for these developments.

Agile characteristics:

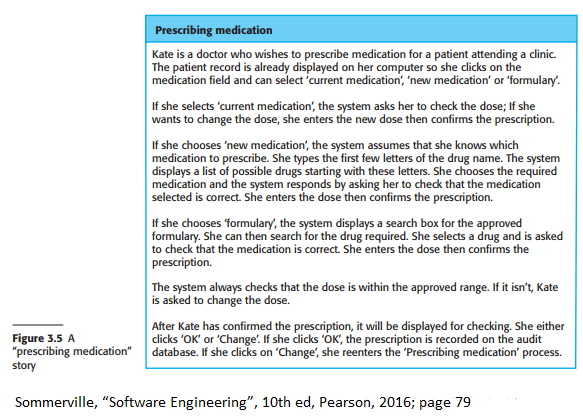
* The scope of the project is kept dynamic so it can easily be adjusted by changing situations or conditions.
* Systems are usually developed in increments so that consumers and other stakeholders can give their input during the development phase. This helps keep the vision of the stakeholders and development team aligned.
* Many tools are used to aid during development like tools that automatically test your code, help you make the process of designing and creating the user interface easier, etc. This generally helps save time and labor to focus more on the goals and targets of the project.

#### [**Agile manifesto**](https://agilemanifesto.org/)

The agile manifesto is a collection of principles that explain the reasoning behind the agile approach:

* **Customer involvement**
  + Stakeholders should give their input in all stages of the development process to show what components should be prioritized and give input that can be analyzed to evaluate the progress of every iteration.
* **Embrace change**
  + Change is inevitable therefore the system should be designed in such a way to relieve the potential stress caused by change.
* **Incremental delivery**
  + Software should be incrementally developed in small batches to test components and allow for feedback.
* **Maintain simplicity**
  + Systems should be simple where applicable.
* **People, not process**
  + People's skills should be recognized and used.

#### **Agile User Stories**



The above image depicts an example of a user story. In this case it describes what a doctor would see when prescribing medication to a patient on the required application.

To combine requirements elicitation, which is the process of getting requirements from customers/clients, with development, user stories are used. User stories are what the name implies: a story on how the user interacts with the software. "Story cards" are developed from these stories and focus on what stakeholders need from the software. These stories and needs will become a list of specifications that need to be met by the development team.

Advantages of user stories:

* Stories are more relatable than documentation
* Stories can suggest what are key requirements the software must have or do. In other words, they are helpful for requirements elicitation.

Disadvantages of user stories:

* Users may not go over all the requirements they need from one story.
* Stories may not give a good view of the requirements.
* Terminology differs from developer and consumer as consumers are more familiar with the software.
* This may lead to consumers leaving out details when describing how they use an application as they are used to using it and may deem actions useless or forgettable.

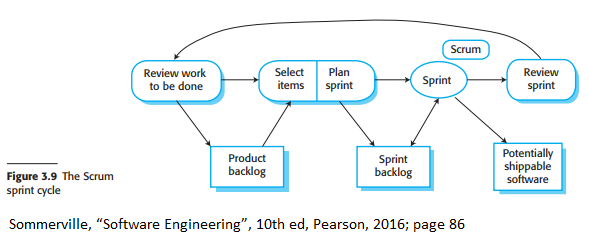
#### **Agile Refactoring**

Code should be refactored, or in other words improved, constantly. Refactoring includes renaming names of methods and attributes, removing duplicate code to reorganize class hierarchies and relationships, replacing code that is similar to each other, etc. The objective of doing this is to make developers understand the software more which would lead to less of a need for documentation due to the code being easier to understand and structured well.

#### **Agile Test First Development**

Test first or test driven development means that tests are written before the actual code. This allows tests to run alongside the development of the code to find any errors early. It also means that developers understand exactly what they are supposed to be making before starting to write code. Users can also be more involved this way as their data or dummy data can be used to test the system.

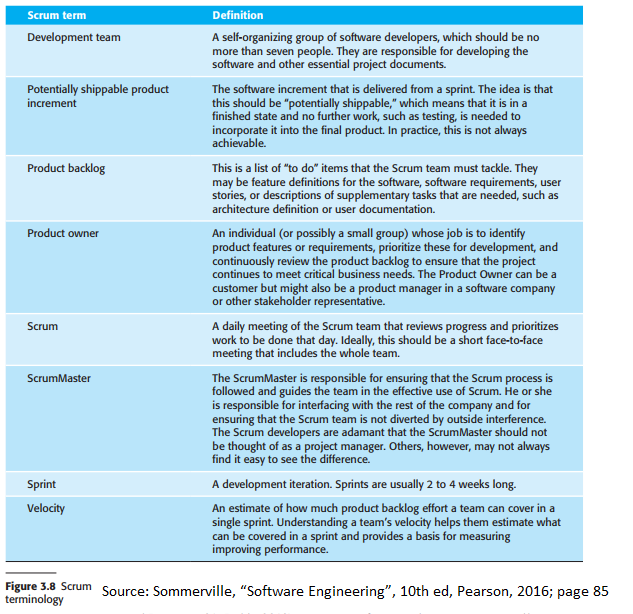
#### **Scrum**



The above image depicts a Scrum sprint cycle.

Scrum is an agile approach that focuses on the management and organization of iterative development. There are three main phases in Scrum:

* **Initial Phase**: begin by outlining objectives for the project and a rough design of the software architecture.
* **Sprint Cycles**: are the incremental portion of development; each cycle will develop an increment of the project.
* **Closure phase**: involves wrapping up the project and completing any documentation that is required



The above image explains what different commonly used Scrum terms mean.

Scrum sprints have a fixed length (usually 2 - 4 weeks) and begin by the creation of a list of objectives and work that should be done, this is called the product backlog. The selection phase is next, where features and functionalities are selected from the product backlog. These will be the components that will be worked on during the sprint as they are usually the components deemed to be the most important and can be reasonably done well during the time frame. After this stage, teams are organized to develop the software.

Teams will hold short daily meetings known as Scrums to discuss current progress and any changes they need to make towards prioritization of features. This allows everyone on the team to know what is currently happening and can re-plan activities quickly if needed. The Scrum Master will help organizing the sprint. They are tasked to arrange the daily Scrums, track what is being done and what has to be done, communicate with stakeholders and management, and other such organizational activities.

Scrum benefits:

* Development is broken into multiple iterations that are manageable and understandable.
* The whole process is visible as the whole team knows the current progress of development due to the daily scrums.
* Change will not cause delays as the team can quickly adapt to the change.
* Customers will get fast delivery of increments which will allow them to give feedback earlier.

**Story pointing**

## [Story Points](https://www.atlassian.com/agile/project-management/estimation)

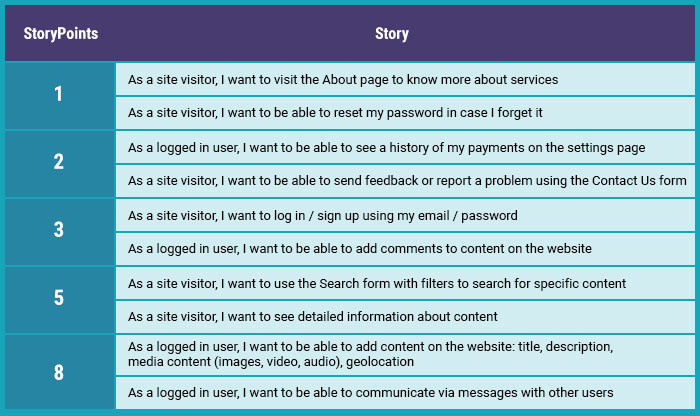
Agile development has the product backlog which is a list of work that contains short descriptions of all the desired features for a product. The engineering team has to come up with estimated time of completion for the required features, this is called the estimation process. During this time it is vital to have good discussions on each requirement as this will help the whole team with understanding what exactly needs to be done and how it will be accomplished. Story points are important at this time.

##### **Story Points**

Story points are measurement units for expressing how much effort will have to go into fully implementing/the completion of a requirement or product backlog item. The team as a whole assigns story points based on complexity, amount of work and risk of change/uncertainty. Values are assigned to break down work into smaller sections. This process also helps understanding of the task to better prepare teams for the required work.

Story points benefits:

* The discussion makes teams more prepared and ready for the upcoming workload.
* The discussion also helps understanding of each feature and its Implementation.
* Story points reward team members based on difficulty instead of time spent.
* This process repeated over and over for different projects will lead to a better understanding of the capabilities of the team and time required.



The image above depicts an example of story points for a project. (Image from [RubyGarage](https://rubygarage.org/blog/how-to-estimate-with-story-points" \t "_blank))

**Scrum Ceremonies**

# **Scrum Ceremonies**

Scrum ceremonies are meetings used during the life cycle of the sprint in a project. All of the involved groups use them to give progress updates, promote general understanding between teams, create and track user stories, as well as gather data for sprint metrics. Scrum ceremonies are as time-sensitive as the sprints and are meant to only take as much time as is necessary.

### **Types of Scrum Ceremonies**

* Sprint Planning
* Daily Scrum/ Daily Stand-Up
* Sprint Review
* Sprint Retrospective

## Sprint Planning

Sprint planning is vital for the success of any sprint. This ceremony includes the development team, scrum master, and product owner and happens before the start of every sprint. The scope of the sprint, sprint goals, metrics and other specifics of the sprint(such as user stories and team composition) are decided and agreed upon.

* This ceremony is the longest: it is typically an hour per week of sprint

## Daily Scrum

Daily Scrum meetings are known as daily standups and are used to communicate progress, blockers, as well as communicate today’s agenda. It involves individuals "standing up" in front of the team to explain what they had done since the last standup and what they were planning on doing today.

* This scrum ceremony is usually brief (under 15 minutes)

## Sprint Review

A sprint review involves all the parties interested in its progress. Invite as many people as you can to attend and give feedback. This is time for the scrum team to verbalize their hard work, progress, successes, and failures over the course of the sprint. The feedback on a sprint is vital for future sprints to be improved.

* typically runs between 45 minutes to an hour, excluding demo time

## Sprint Retro

Sprint reviews are about the product. In contrast, Sprint retrospective is about the process. This is time for the scrum master to bring forward the sprints metrics and for the scrum team as a whole to assess inefficiencies they came across in the last sprint. Then as a team, they make plans for improvement which are taken to the next sprint planning meeting.

-typically 45 minutes long

### **Additional Resources:**

<https://www.scrumalliance.org/about-scrum/events> <https://www.atlassian.com/agile/scrum/ceremonies> <https://geekbot.com/blog/scrum-ceremonies-and-artifacts-what-wasnt-in-the-scrum-guide/>