* Tell me about yourself.
* Why do you think that you are eligible for this job?
* What are your weaknesses and strengths?
* How would you rate me as an interviewer?
* Where do you see yourself after five years?
* Why should I hire you?
* Are you a hard worker? What is the difference between hard work and smart work?
* Are you willing to relocate?
* If you won a 10 crore lottery, will you still work?
* What do you know about Hexaware?
* What do you do if any other organization pays you more than us?

# Resume questions

# <https://www.interviewbit.com/angular-interview-questions/>

# Action filter in MVC

## **There are 4 types of filters in asp.net mvc.** **1. Authorization filters** - Implements IAuthorizationFilter. Examples include AuthorizeAttribute and RequireHttpsAttribute. These filters run before any other filter. **2. Action filters** - Implement IActionFilter **3. Result filters** - Implement IResultFilter. Examples include OutputCacheAttribute.  **4. Exception filters** - Implement IExceptionFilter. Examples include HandleErrorAttribute.

## Few action filters in mvc: Authorize  ChildActionOnly HandleError OutputCache  RequireHttps ValidateInput ValidateAntiForgeryToken

If you want **"SecureMethod"** to be available only for authenticated users, then decorate it with **"Authorize"** attribute.  
[Authorize]  
public ActionResult SecureMethod()  
{  
    return View();  
}

To allow anonymous access to NonSecureMethod(), apply [AllowAnonymous] attribute. AllowAnonymous attribute is used to skip authorization enforced by Authorize attribute.   
[AllowAnonymous]  
public ActionResult NonSecureMethod()  
{  
    return View();  
}

// This method is accessible only by a child request. A runtime   
    // exception will be thrown if a URL request is made to this method  
    [ChildActionOnly]  
    public ActionResult Countries(List<String> countryData)  
    {  
        return View(countryData);  
    }

Child actions can also be invoked using **"RenderAction()"** HTMl helper as shown below.  
@{  
    Html.RenderAction("Countries", new { countryData = new List<string>() { "US", "UK", "India" } });  
}

Child action methods are different from NonAction methods, in that NonAction methods cannot be invoked using Action() or RenderAction() helpers.

**HandleError**

<customErrors mode="On">  
  <error statusCode="404" redirect="~/Error/NotFound"/>  
</customErrors>

**HandleErrorAttribute**is added to the GlobalFilters collection in global.asax

**OutputCache**

[OutputCache(Duration = 10)]  
public ActionResult Index()  
{  
    System.Threading.Thread.Sleep(3000);  
    return View(db.Employees.ToList());  
}

[RequireHttps]  
public string Login()  
{  
    return "This method should be accessed only using HTTPS protocol";  
}

Sensitive data such as login credentials, credit card information etc, must always be transmitted using HTTPS. Information transmitted over https is encrypted.

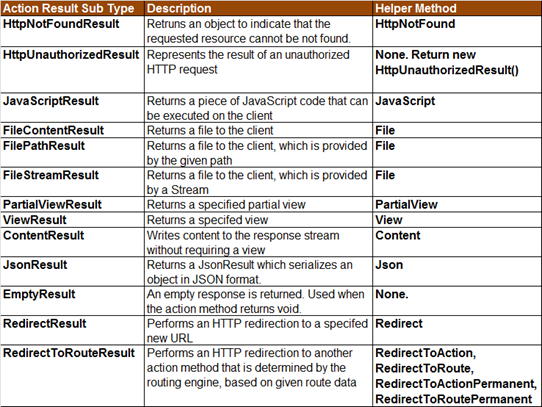
[HttpPost]  
[ValidateInput(false)]  
public string Index(string comments)  
{  
    return "Your Comments: " + comments;  
}  
   
At this point, you should be able to submit comments, with HTML tags in it.

**Custom Action Filter**

    public class TrackExecutionTime : ActionFilterAttribute, IExceptionFilter  
    {  
        public override void OnActionExecuting(ActionExecutingContext filterContext)  
        {  
            string message = "\n" + filterContext.ActionDescriptor.ControllerDescriptor.ControllerName +   
                " -> " + filterContext.ActionDescriptor.ActionName + " -> OnActionExecuting \t- " +   
                DateTime.Now.ToString() + "\n";  
            LogExecutionTime(message);  
        }  
  
        public override void OnActionExecuted(ActionExecutedContext filterContext)  
        {  
            string message = "\n" + filterContext.ActionDescriptor.ControllerDescriptor.ControllerName +   
                " -> " + filterContext.ActionDescriptor.ActionName + " -> OnActionExecuted \t- " +   
                DateTime.Now.ToString() + "\n";  
            LogExecutionTime(message);  
        }  
  
        public override void OnResultExecuting(ResultExecutingContext filterContext)  
        {  
            string message = filterContext.RouteData.Values["controller"].ToString() +   
                " -> " + filterContext.RouteData.Values["action"].ToString() +

                " -> OnResultExecuting \t- " + DateTime.Now.ToString() + "\n";  
            LogExecutionTime(message);  
        }  
  
        public override void OnResultExecuted(ResultExecutedContext filterContext)  
        {  
            string message = filterContext.RouteData.Values["controller"].ToString() +   
                " -> " + filterContext.RouteData.Values["action"].ToString() +   
                " -> OnResultExecuted \t- " + DateTime.Now.ToString() + "\n";  
            LogExecutionTime(message);  
            LogExecutionTime("---------------------------------------------------------\n");  
        }  
  
        public void OnException(ExceptionContext filterContext)  
        {  
            string message = filterContext.RouteData.Values["controller"].ToString() + " -> " +  
               filterContext.RouteData.Values["action"].ToString() + " -> " +  
               filterContext.Exception.Message + " \t- " + DateTime.Now.ToString() + "\n";  
            LogExecutionTime(message);  
            LogExecutionTime("---------------------------------------------------------\n");  
        }  
  
        private void LogExecutionTime(string message)  
        {  
            File.AppendAllText(HttpContext.Current.Server.MapPath("~/Data/Data.txt"), message);  
        }  
    }  
}  
public class HomeController : Controller  
{  
    [TrackExecutionTime]  
    public string Index()  
    {  
        return "Index Action Invoked";  
    }  
  
    [TrackExecutionTime]  
    public string Welcome()  
    {  
        throw new Exception("Exception ocuured");  
    }  
}

Why do we have so many sub-types?  
An action method in a controller, can return a wide range of objects. For example, an action method can return  
**1.** ViewResult  
**2.** PartialViewResult  
**3.** JsonResult  
**4.** RedirectResult etc..  
  
**What should be the return type of an action method - ActionResult or specific derived type?**  
It's a good practise to return specific sub-types, but, if different paths of the action method returns different subtypes, then I would return an ActionResult object. An example is shown below.  
public ActionResult Index()  
{  
    if (Your\_Condition)  
        return View();            // returns ViewResult object  
    else  
        return Json("Data");  // returns JsonResult object  
}  
  
**The following table lists**  
**1.** Action Result Sub Types  
**2.** The purpose of each sub-type  
**3.** The helper methods used to return the specific sub-type



if (Session["Key"] == null){

**RedirectToAction**("Login");

}

# Html helper

An HTML helper is a method that is used to render html content in a view. **HTML helpers** are implemented as **extension methods**.  
  
**For example**, to produce the HTML for a textbox with id="firstname" and name="firstname", we can type all the html in the view as shown below

<input type="text" name="firtsname" id="firstname" />  
  
**OR**  
  
**We can use the "TextBox" html helper.**  
@Html.TextBox("firstname")  
  
  
There are several overloaded versions. To set a value, along with the name, use the following overloaded version.  
@Html.TextBox("firstname", "John")  
  
**The above html helper, generates the following HTML**  
<input id="firstname" name="firstname" type="text" value="John" />

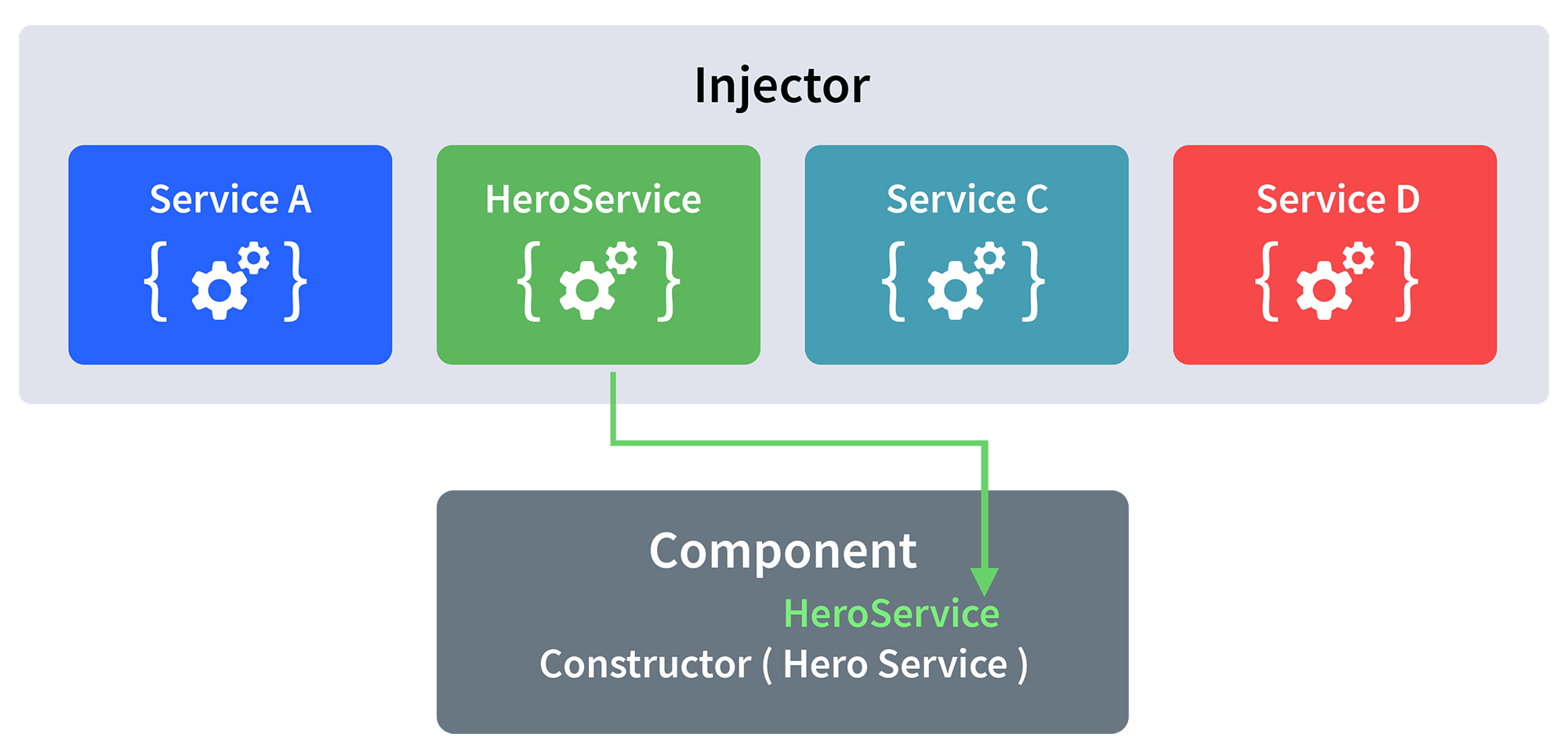
To set **HTML attributes**, use the following overloaded version. Notice that, we are passing HTML attributes (**style**& **title**) as an **anonymous type**.  
@Html.TextBox("firstname", "John", new { style = "background-color:Red; color:White; font-weight:bold", title="Please enter your first name" })  
Some of the **html attributes**, are reserved keywords. Examples include **class, readonly** etc. To use these attributes, use **"@"** symbol as shown below.

@Html.TextBox("firstname", "John", new { @class = "redtextbox", @readonly="true" })  
  
**To generate a label for "First Name"**  
@Html.Label("fisrtname", "First Name")  
  
**To generate a textbox to enter password, so that the input is masked**  
@Html.Password("Password")  
  
**To generate a multi-line textbox with 5 rows and 20 columns**  
@Html.TextArea("Comments", "", 5, 20, null)  
  
**To generate a hidden textbox**  
@Html.Hidden("id")  
  
**Hidden textbox** is used to store id values. Id values are not displayed on the page to the end user, but we need them to update data when the form is posted to the server.  
  
**Is it possible to create our own custom html helpers?**

Yes, we will discuss this in a later video session.  
  
**Is it mandatory to use HTML helpers?**  
No, you can type the required HTML, but using HTML helpers will greatly reduce the amount of HTML that we have to write in a view. Views should be as simple as possible. All the complicated logic to generate a control can be **encapsulated** into the **helper,** to keep views simple.

# 14. Explain the concept of Dependency Injection?

Dependency injection is an application design pattern which is implemented by Angular.  
It also forms one of the core concepts of Angular.  
  
**So what is dependency injection in simple terms?**  
Let’s break it down, **dependencies in angular are nothing but services which have a functionality**. Functionality of a service, can be needed by various components and directives in an application. Angular provides a smooth mechanism by which we can inject these dependencies in our components and directives.  
So basically, we are just making dependencies which are injectable across all components of an application.



Let’s understand how DI (Dependency Injection) works:  
  
Consider the following service, which can be generated using:

ng g service test

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

@Injectable({

providedIn: 'root'

})

export class TestService {

importantValue:number = 42;

constructor() { }

returnImportantValue(){

return this.importantValue;

}

}

As one can notice, we can create injectable dependencies by adding the **@Injectable** decorator to a class.  
  
We inject the above dependency inside the following component:

import { TestService } from './../test.service';

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

@Component({

selector: 'app-test',

templateUrl: './test.component.html',

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

})

export class TestComponent implements OnInit {

value:number;

constructor(private testService:TestService) { }

ngOnInit() {

this.value = this.testService.returnImportantValue();

}

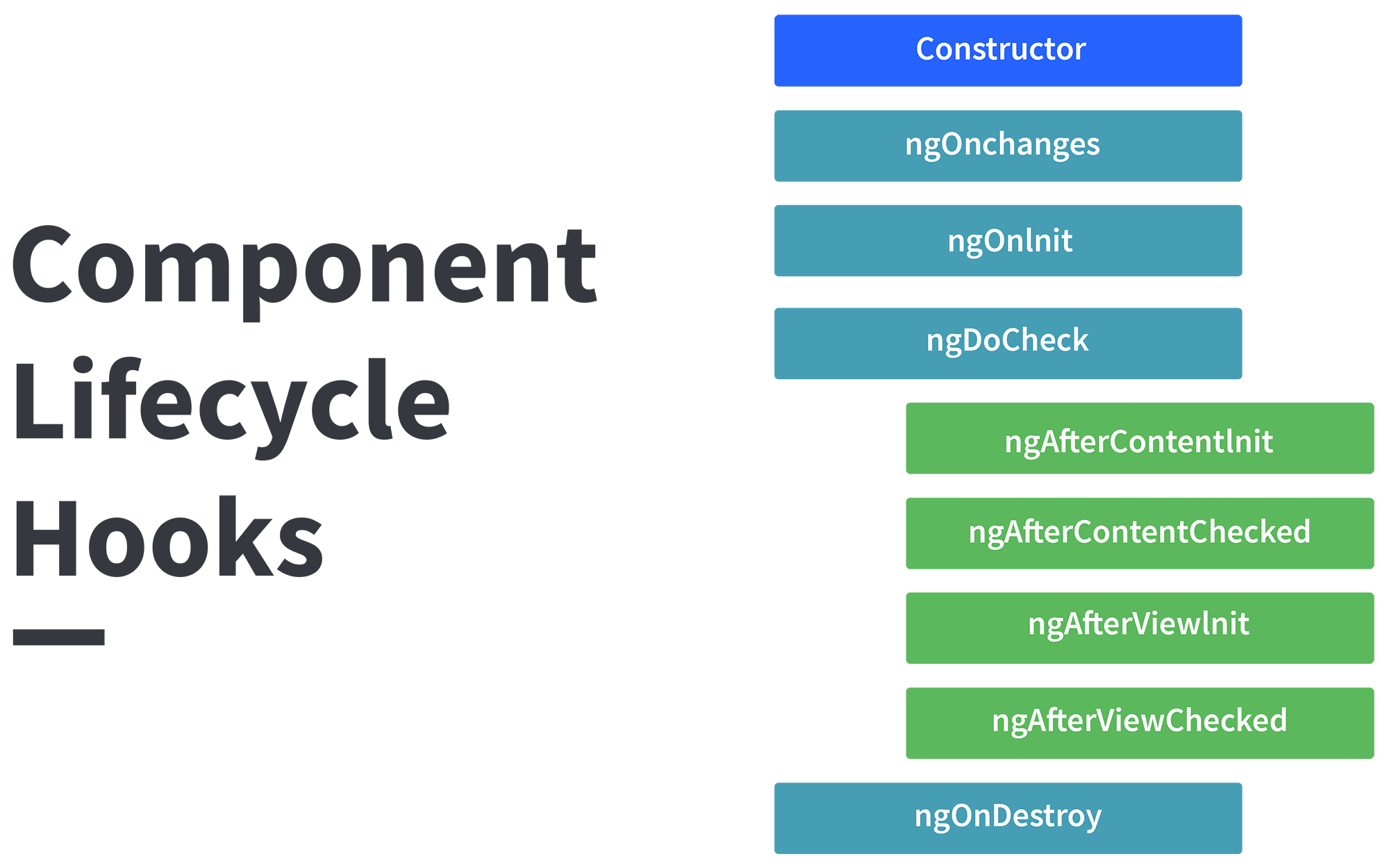
}

One can see we have imported our TestService at the top of the page. Then, we have created an instance inside the constructor of the component and implemented the **returnImportantValue** function of the service.  
  
From the above example, we can observe how angular provides a smooth way to inject dependencies in any component.

# Angular life cycle

### 7. What are lifecycle hooks in Angular? Explain a few lifecycle hooks.

Every component in Angular has a lifecycle, different phases it goes through from the time of creation to the time it's destroyed. Angular provides **hooks** to tap into these phases and trigger changes at specific phases in a lifecycle.



**ngOnChanges( )** This hook/method is called before **ngOnInit** and whenever one or more input properties of the component changes.  
This method/hook receives a SimpleChanges object which contains the previous and current values of the property.  
  
**ngOnInit( )** This hook gets called once, after the **ngOnChanges** hook.  
It initializes the component and sets the input properties of the component.  
  
**ngDoCheck( )** It gets called after **ngOnChanges** and **ngOnInit** and is used to detect and act on changes that cannot be detected by Angular.  
We can implement our change detection algorithm in this hook. **ngAfterContentInit( )** It gets called after the first **ngDoCheck** hook. This hook responds after the content gets projected inside the component.  
  
**ngAfterContentChecked( )** It gets called after **ngAfterContentInit** and every subsequent **ngDoCheck**. It responds after the projected content is checked.  
  
**ngAfterViewInit( )** It responds after a component's view, or a child component's view is initialized.  
  
**ngAfterViewChecked( )** It gets called after **ngAfterViewInit**, and it responds after the component's view, or the child component's view is checked.  
  
**ngOnDestroy( )** It gets called just before Angular destroys the component. This hook can be used to clean up the code and detach event handlers.  
  
Let’s understand how to use **ngOnInit** hook, since it’s the most oftenly used hook. If one has to process lot of data during component creation, it’s better to do it inside **ngOnInit** hook rather than the constructor:

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

@Component({

selector: 'app-test',

templateUrl: './test.component.html',

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

})

export class TestComponent implements OnInit {

constructor() { }

ngOnInit() {

this.processData();

}

processData(){

// Do something..

}

}

As you can see we have imported OnInit but we have used **ngOnInit** function. This principle should be used with the rest of the hooks as well.

# Directives in Angular

Directives in Angular 7 are Typescript class which is declared with decorator @Directive. These are the Document Object Model (DOM) instruction sets, which decide how logic implementation can be done.  
Angular directives can be classified into three types: 

1. **Component Directives:** It forms the main class and is declared by **@Component**. It contains the details on component processing, instantiated and usage at run time.  
   **Example:** It contains certain parameters some of them are shown in this example.

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

1. There are three parameters discussed below:
   * **Selector:** Tells the template tag which specifies the beginning and end of the component.
   * **templateURL:** Consists of the template used for the component.
   * **styleUrls:** It is of array type which consists of all the style format files used for the template.
2. **Structural Directives:** Structure directives manipulate the DOM elements. These directives have a \* sign before the directive. For example, \*ngIf and \*ngFor.  
   **Example:** Let’s look the implementation of \*ng-if-else and \*ng-for. Using them, we classify weekdays and weekends.   
   **Component file:**

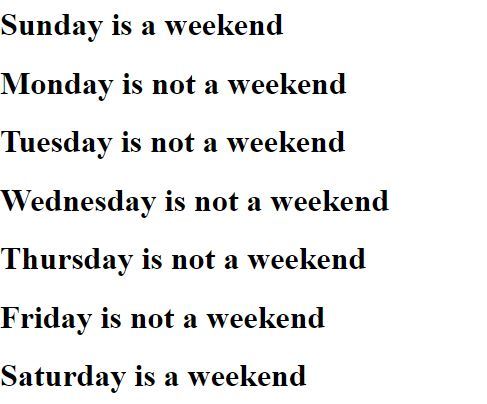
|  |
| --- |
| import { Component } from '@angular/core';   @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent {    Weekdays:Array =[  'Sunday', 'Monday', 'Tuesday',  'Wednesday', 'Thursday', 'Friday', 'Saturday']  } |

1. **Template file:**

* html

|  |
| --- |
| <div \*ngFor="let day of Weekdays">  <ng-container \*ngIf =      "(day == 'Saturday' || day == 'Sunday'); else elseTemplate">    <h1>{{day}} is a weekend</h1>  </ng-container>  <ng-template #elseTemplate>    <h1>{{day}} is not a weekend</h1>  </ng-template>  </div> |

1. **Output:**



1. **Attribute Directives:** Attribute directives are used to change the look and behavior of the DOM element. It provides the facility to create our own directive.   
   **Example:** This example describes how to make our own directive.  
   Write command as follows:

ng g directive

1. **Directive:**

* javascript

|  |
| --- |
| import { Directive, ElementRef, OnInit } from '@angular/core';    @Directive({    selector: '[appChanges]'  })  export class ChangesDirective {    constructor(private eltRef: ElementRef) {        // Changing the background to green      this.eltRef.nativeElement.style.backgroundColor = 'green';      this.eltRef.nativeElement.style.color = 'white';      changing the text color to white    }      ngOnInit() {    }  } |

1. **The Component File:**

* javascript

|  |
| --- |
| import { Component, OnInit, Directive } from '@angular/core';  import { ChangesDirective } from '../changes.directive';    @Component({    selector: 'app-derived-directive',    templateUrl: './derived-directive.component.html',    styleUrls: ['./derived-directive.component.css']  })    export class DerivedDirectiveComponent implements OnInit {      isClicked:boolean=false;    constructor() { }    buttonClick(){        // Change controlled by button press      this.isClicked = true;    }    ngOnInit() {    }  } |

1. **The Template**

* html

|  |
| --- |
| <button>Click Here</button>        <div style="width: 220px;height: 50px">          <h1>GeeksForGeeks</h1>      </div>          <div style="color: green;width: 300px;height: 50px">          <h1> GeeksForGeeks</h1>      </div> |

1. **Output:**
   * **Before clicking the Button:**



* **After clicking the Button:**



# Sp optimization, execution plan

# <https://www.c-sharpcorner.com/article/sql-query-execution-plan/#:~:text=It%20is%20a%20graphical%20representation,engine%20to%20return%20some%20results>.

# SQL Function

# Unique PK

# Clustered,non clustered

# Web Api

# SOAP

# HTTP, which is lighter

# How to return json, behaviour

<https://www.c-sharpcorner.com/article/consuming-asp-net-web-api-rest-service-in-asp-net-mvc-using-http-client/>

# **Consuming ASP.NET Web API REST Service In ASP.NET MVC Using HttpClient**

using ConsumingWebAapiRESTinMVC.Models;

using Newtonsoft.Json;

using System;

using System.Collections.Generic;

using System.Net.Http;

using System.Net.Http.Headers;

using System.Threading.Tasks;

using System.Web.Mvc;

namespace ConsumingWebAapiRESTinMVC.Controllers

{

public class HomeController : Controller

{

//Hosted web API REST Service base url

string Baseurl = "<http://192.168.95.1:5555/>";

public async Task<ActionResult> Index()

{

List<Employee> EmpInfo = new List<Employee>();

using (var client = new HttpClient())

{

//Passing service base url

client.BaseAddress = new Uri(Baseurl);

client.DefaultRequestHeaders.Clear();

//Define request data format

client.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue("application/json"));

//Sending request to find web api REST service resource GetAllEmployees using HttpClient

HttpResponseMessage Res = await client.GetAsync("api/Employee/GetAllEmployees");

//Checking the response is successful or not which is sent using HttpClient

if (Res.IsSuccessStatusCode)

{

//Storing the response details recieved from web api

var EmpResponse = Res.Content.ReadAsStringAsync().Result;

//Deserializing the response recieved from web api and storing into the Employee list

EmpInfo = JsonConvert.DeserializeObject<List<Employee>>(EmpResponse);

}

//returning the employee list to view

return View(EmpInfo);

}

}

}

}

<https://www.c-sharpcorner.com/UploadFile/302f8f/Asp-Net-mvc-using-jquery-ajax/>

# **ASP.NET MVC Application - Using JQuery, AJAX**

**$.ajax () Method:**  
JQuery’s core method for creating Ajax requests. Here are some jQuery AJAX methods:

* **$.ajax()** Performs an async AJAX request.
* **$.get()** Loads data from a server using an AJAX HTTP GET request.
* **$.post()**Loads data from a server using an AJAX HTTP POST request.

To know more [click](https://learn.jquery.com/ajax/jquery-ajax-methods/).  
  
**$.ajax () Method Configuration option:**  
Options that we use:

* async:
* type:
* url:
* data:
* datatype:
* success:
* error:

Let’s have details overview:  
  
**async**  
Set to false if the request should be sent synchronously. Defaults to true.  
  
Note that if you set this option to false, your request will block execution of other code until the response is received.  
  
**Example:**

1. async: **false**,

**type**  
This is type of HTTP Request and accepts a valid HTTP verb.  
  
The type of the request, "POST" or "GET" defaults to "GET". Other request types, such as "PUT" and "DELETE" can be used, but they may not be supported by all the web browsers.  
  
**Example:**

1. type: "POST",

**url**  
The URL for the request.  
  
**Example:**

1. url: "/Home/JqAJAX",

**data**  
The data to be sent to the server. This can either be an object or a query string.  
  
**Example:**

1. data: JSON.stringify(model\_data),

**dataType**  
The type of data you expect back from the server. By default, jQuery will look at the MIME type of the response if no dataType is specified.  
  
Accepted values are text, xml, json, script, html jsonp.  
  
**Example:**

1. dataType: "json",

**contentType**  
This is the content type of the request you are making. The default is 'application/x-www-form-urlencoded'.  
  
**Example:**

1. contentType: 'application/json; charset=utf-8',

**success**  
A callback function to run if the request succeeds. The function receives the response data (converted to a JavaScript object if the DataType was JSON), as well as the text status of the request and the raw request object.

1. Success: function (result) {
2. $('#result').html(result);
3. }

**error**  
A callback function to run if the request results in an error. The function receives the raw request object and the text status of the request.

1. error: function (result) {
2. alert('Error occured!!');
3. },

**Let’s Post Values using JQuey,Ajax:**  
We often use the jQuery Ajax method in ASP.NET Razor Web Pages. Here is a sample code:

1. <input type="submit" id="btnSubmit" value="Add Student" />
2. <script type="text/javascript">
3. $(document).ready(function() {
4. $(function() {
5. $('#btnSubmit').click(function(**event**) {
6. **event**.preventDefault();
7. var Student = {
8. ID: '10001',
9. Name: 'Shashangka',
10. Age: 31
11. };
12. $.ajax({
13. type: "POST",
14. url: "/Home/JqAJAX",
15. data: JSON.stringify(Student),
16. dataType: "json"
17. contentType: 'application/json; charset=utf-8',
18. success: function(data) {
19. alert(data.msg);
20. },
21. error: function() {
22. alert("Error occured!!")
23. }
24. });
25. });
26. });
27. });
28. </script>

**Controller Action:**

1. // GET: Home/JqAJAX
2. [HttpPost]
3. **public** ActionResult JqAJAX(Student st) {
4. **try** {
5. **return** Json(**new** {
6. msg = "Successfully added " + st.Name
7. });
8. } **catch** (Exception ex) {
9. **throw** ex;
10. }
11. }

**Posting JSON**  
JSON is a data interchange format where values are converted to a string. The recommended way to create JSON is include the JSON.stringify method. In this case we have defined:

1. JSON.stringify(Student)

And the data type set to:

1. datatype: "json"

And the content type set to application/json

1. contentType: 'application/json; charset=utf-8'

**Syntax:**  
JSON.stringify(value[, replacer[, space]])  
  
**Post Script:**

1. var Student = {
2. ID: '10001',
3. Name: 'Shashangka',
4. Age: 31
5. };
6. $.ajax({
7. type: "POST",
8. url: "/Home/JqAJAX",
9. data: JSON.stringify(Student),
10. contentType: 'application/json; charset=utf-8',
11. success: function(data) {
12. alert(data.msg);
13. },
14. error: function() {
15. alert("Error occured!!")
16. }
17. });

**Controller Action:**

1. // GET: Home/JqAJAX
2. [HttpPost]
3. **public** ActionResult JqAJAX(Student st) {
4. **try** {
5. **return** Json(**new** {
6. msg = "Successfully added " + st.Name
7. });
8. } **catch** (Exception ex) {
9. **throw** ex;
10. }
11. }

**JSON Response Result:**  
Sent data format:{"ID":"10001","Name":"Shashangka","Age":31}Received Data format:{"msg":"Successfully added Shashangka"}  
  
**Let’s Post JavaScript Objects:**  
To send JavaScript Objects we need to omit the JSON.stringify(Student) method and we need to pass the plain object to the data option. In this case we have defined:

1. data: Student

And the datatype set to:

1. datatype: "html"

And the content type set to default.

1. contentType: 'application/x-www-form-urlencoded'

**Post Script:**

1. var Student = {
2. ID: '10001',
3. Name: 'Shashangka',
4. Age: 31
5. };
6. $.ajax({
7. type: "POST",
8. url: "/Home/JqAJAX",
9. data: Student,
10. contentType: 'application/x-www-form-urlencoded',
11. datatype: "html",
12. success: function(data) {
13. alert(data.msg);
14. },
15. error: function() {
16. alert("Error occured!!")
17. }
18. });

**Controller Action:**

1. // GET: Home/JqAJAX
2. [HttpPost]
3. **public** ActionResult JqAJAX(Student st) {
4. **try** {
5. **return** Json(**new** {
6. msg = "Successfully added " + st.Name
7. });
8. } **catch** (Exception ex) {
9. **throw** ex;
10. }
11. }

**JavaScript Objects Response Result:**  
Sent data format:ID=10001&Name=Shashangka&Age=31Received Data format:{"msg":"Successfully added Shashangka"}  
  
**Let’s Post JavaScript Arrays:**  
To send Array we need to omit the JSON.stringify(Student) method and we need to pass the plain object to the data option. In this case we have defined:

1. data: Student

And the datatype set to:

1. datatype: "html"

And the content type set to default

1. contentType: 'application/x-www-form-urlencoded'

**Post Script:**

1. var ID = ["Shashangka", "Shekhar", "Mandal"];
3. $.ajax({
4. type: "POST",
5. url: "/Home/JqAJAX",
6. data: {
7. values: ID
8. },
9. datatype: "html",
10. contentType: 'application/x-www-form-urlencoded',
11. success: function(data) {
12. alert(data.msg);
13. },
14. error: function() {
15. alert("Error occured!!")
16. }
17. });

**Controller Action:**

1. // GET: Home/JqAJAX
2. [HttpPost]
3. **public** ActionResult JqAJAX(**string**[] values) {
4. **try** {
5. **return** Json(**new** {
6. msg = String.Format("Fist Name: {0}", values[0])
7. });
8. } **catch** (Exception ex) {
9. **throw** ex;
10. }
11. }

**Array Response Result:**  
Sent data format:values[]=Shashangka&values[]=Shekhar&values[]=MandalReceived Data format:{"msg":"Fist Name: Shashangka"}  
  
Hope this will help to understand different datatypes and Ajax posting. Thanks!

# Angular services, components, module, directive

# Why angular

# SOLID

# MVC authentication

# One view multiple controller

# MVC folder structure

# Partial view

# Repository pattern

# My .net core project flow

# Angular data between components

# Components to view

# Trigger in sql

# Dom in jQuery

# jQuery with ajax

# Design patterns

# MVC architecture

# @@error

<https://www.c-sharpcorner.com/UploadFile/f0b2ed/exception-handling-in-sql-server/>

@@ERROR return the error number for last executed T-SQL statements. It returns 0 if the previous Transact-SQL statement encountered no errors else return an error number.

1. **Update** Employee **set** Salary=19000 **Where** Emp\_IID=5
2. IF @@ERROR = 547
3. PRINT 'A check constraint violation occurred.';

# Handling in SP

# Debugging the SP

# Wcf

# Api on only http?

# Rest principal

# @output in sp

# multiple End points in API

# Factory design pattern

# jQuery able vs Innumerable

# Bundling and minification

# Testing in MVC, can views be tested without controller

# Why you choose angular

# New features in angular 8

# DOM

# MVC api

# .net core api

# Angular api

# Api call from client on different server

# IActionResut return trype

# What is viewresult

# Different results from action method

# MVC folder structure

# Viewstart in which folder?

# Web api just for http??

# Why to use Sp?

# Delete truncate

# MVC filestream output

# Unit testing for all