[Java Script](#JS)

[JQuery](#JQuery)

[HTML](#HTMLSyntax)

[ASPNet](#ASP_Net)

[AJAXSolr](#AjaxSolr)

[Apcahe Solr & Lucene](#ApacheSolrLucene)

[Angular JS](#AngularJS)

Node JS

AWS Lambda

**ECMA Script & JAVA Script**

1. **Client side rendered views** enable building more responsive web applications w.r.t. server side rendering. Traditionally HTML rendering was done on server and sent to browser as front end was immature.
2. JAVA Script is a **client side scripting language**. When user goes to a website, webserver sends html back to user, it contains JS code and bowser executes code. This is in contrast to Server side scripting languages that don’t send code but sends output to client. **Web storm editor** is commonly used for writing/editing JS files, highlighting syntax errors and methods
3. **ECMA script is a standard** while JavaScript is **implementation of that standard.** Other implementations are **ActionScript and JScript**. ECMA script is used to manipulate the objects in web pages specified by the Document Object Model (DOM).
4. **W3C** is an international industry consortium that develops specifications to enhance the interoperability of web-related products. ECMA develops Standards and Technical Reports in the area of information and communication technology
5. **ECMA 2015 or ES6** includes classes and modules, iterators, for/of loops, array functions, promises, reflection & proxies. Promise refers to constructs used for synchronizing program execution in some concurrent programming languages
6. **Timers in JS execute in single thread** leading to queue up of events to be executed. setTimeout (function, delay) starts a timer that calling function after mentioned delay. setInterval(function, delay) function repeatedly executes given function
7. **SessionState** is specific to user data and **accessible across all pages**. '**ViewState’** is specific to a page in a session
8. Reading/Writing of a file can be handled in JS through **JS extensions and Active X objects**
9. **delete operator** is used to delete all variables and objects in a program but it cannot delete variables declared with **VAR keyword**
10. JS has Primitive & reference types. **Generic objects** are represented by VAR type. Exception handling is supported through Try catch. Output on screen done through **document object**
11. **onLoad function** is not executed until all the information on the page is loaded. **onDocumentReady** loads the code just after the DOM is loaded
12. **Closure** is a locally declared variable related to a function which stays in memory when the function has returned

*function greeter(name, age) {*

*var message = name + " says howdy!! He is” + age + " years old";*

*return function greet() {*

*console.log (message);*

*};*

*}*

*// Generate the closure*

***var JamesGreeter = greeter("James", 23);***

*// Use the closure*

***JamesGreeter();***

1. Anonymous function is declared without any named identifier & is inaccessible after declaration

*var anon = function() {alert('I am anonymous');};*

*anon();*

1. **Event Handlers** are extra attributes of object that includes event’s name and the action taken if the event takes place.

*<input type="button" value="Show Contents"* ***onClick="displayArray()"*** */>*

1. Deferred script delay execution of script till HTML parser is running
2. **Functions in JavaScript** are seen as objects and can be passed as arguments to other functions or returned as values from other functions, assigned to variables or stored in data structures.
3. **Nested functions** are called every time the main function is invoked.
4. The **escape () & unescape() functions** code & decode a string when transferring across network.
5. JavaScript codes can be hidden from old browsers using **“<!–” and “//–>”** after start and before end of <script> block
6. A JS function is written **within <script></script> tags** in HTML. It can be associated with an event on HTML control

*<input type="button" value="Show Contents"* ***onClick="displayArray()"*** */>*

*<div id="ResultsDiv"></div>*

A Click on button control calls JS function displayArray()

1. **A reference can be obtained to an HTML element** from within JS function. Also a new element can be created and appended to an existing element

*var ResultsDiv = document.getElementById("ResultsDiv");*

*var newDiv = document.createElement("div");*

*ResultsDiv.appendChild(newDiv);*

1. To change contents of a specified HTML element use **property innerHtml**

*var newDiv = document.createElement("div");*

*newDiv.innerHTML = "Item " + (i+1).toString();*

*ResultsDiv.appendChild(newDiv);*

1. To **replace child elements** of a given HTML element, first all existing child elements should be removed and then new elements added

*var ResultsDiv = document.getElementById("ResultsDiv");*

*while (ResultsDiv.hasChildNodes()){*

*ResultsDiv.removeChild(ResultsDiv.lastChild);*

*}*

1. JS supports **Associative arrays and collections**. Associative arraysare similar to maps, allowing **string based indexes**. Length property doesn’t work with Associative arrays. Collections allow using **arrays as keys**

*var emps = [emp1, emp2, emp3]*

emp1, emp2, emp3 are Associative arrays. Associative arraysalso provide **object based notations** to access array contents

1. To execute a JS function on page start up, use **windows.onload event**. Once everything related to HTML is loaded, assigned function gets executed

*windows.onload=DisplayResults;*

1. Functions can be used as **argument to other functions**, this type of declaration is used for **call backs**
2. A Function expression is assigning an anonymous function to a variable. A **Self executable anonymous function** us an anonymous function is **wrapped inside another set of parenthesis** to invoke itself at point of declaration. It is also referred to as immediately invoked function expression (IIFE)

*(function(){*

*...*

*})();*

IIFE returning a value

*var r = (function(){*

*...*

*return c;*

*})();*

1. JS has a built in **keyword "argument"** which will contain all elements passed to function as parameters in form of an array
2. **Scope of nested functions** in JS is within their parent, and they need to be executed as part of their parent. **A nested function can be returned from parent function** and then executed outside. **Inner function can be assigned to outer function**, in which case outer function loses it original definition. While first execution of outer function executes original definition, subsequent executions execute new definition. **An array** can have elements that **can themselves be functions**
3. Functions can alternatively be used to assign event handlers to controls

*function attachEvent(){*

*var btn = document.getementById("btnId");*

*btn.onclick = function (){*

...

}

}

This kind of declaration can be used to dynamically assign any function as an event handler for a UI element

1. A Method on a JS object can be **defined as an external function**, then assigned to object as a member

*function getAnnualSal(){*

*return this.sal \* 12*

*}*

*vor oEmp1 = {*

*EName: "Jugs";*

*Sal: 50000;*

***AnnSal: getAnnualSal;***

*}*

*alert(oEmp1.AnnSal().toString());*

1. Anonymous functions can also be defined as **methods from inside an object**

*vor oEmp1 = {*

*EName: "Jugs";*

*Sal: 50000;*

***AnnSal: function(){***

***return this.sal \* 12;***

***}***

*}*

1. Methods declared **inside a constructor function similar to above definition**. In this case all functions should be preceded with **"this" keyword**

var emp = function (eName, eSal) {

this.EName = eName;

this.Sal = eSal;

**this.GetAnnualSal = function {**

**return eSal \* 12;**

**}**

};

1. JS supports method chaining based on ability of a method to return an object or current object

var emp = function (eName, eSal) {

this.EName = eName;

this.Sal = eSal;

**this.IncrementSal = function (eInc) {**

**this.Sal += eInc;**

**return this;**

**}**

};

**Oemp.IncrementSal(500).IncrementSal(1500);**

1. Object Member preceded with **"this" are public in scope**, while ones with **var are private to object**

var emp = function (eName, eSal) {

this.EName = eName; //This is Public in scope

**var Sal = eSal; //This is private in scope**

this.IncrementSal = function (eInc) {

this.Sal += eInc;

return this;

}

};

1. Objects can be nested similar to functions. Nested objects can be created from within constructor also, as below

*var Dept = function (deptNo, deptName){*

*this.deptNo = deptNo;*

*this.deptName = deptName;*

*}*

*var Emp = function (eName, eSal, deptNo, deptName){*

*this.Name = eName;*

*this.Sal = eSal;*

***this.Department = new Dept(deptNo, deptName);***

*}*

*Var oEmp = new Emp(“Jag”. 5000, 10, “IT”)*

1. Functions written separately can be applied to properties of objects through **call, apply & bind** as object methods. This approach is used with objects having one or more common properties or method. Call, Apply and Bind attach function to an object, but call and apply **return result immediately**, while **bind returns a new object with attached function** that can be executed later. Apply is similar to call except that it is used with arrays. These functions are used as below

*functionname.****apply*** *(obj, [arg1, arg2, arg3]);*

*functionname.****call*** *(obj, arg1, arg2, arg3);*

*var* ***bound*** *= functionname.bind(obj);*

*bound (arg1, arg2, arg3);*

An example of usage is as below

*var emp = {*

*name: "Jags";*

*Sal: 50000;*

*}*

*var addToThis = function(incr){*

*return this.Sal + incr;*

*}*

*var sal =* ***addToThis.call(emp, 500);***

1. Every object has a shared space called **prototype**, generally used for **keeping default values or object methods**. Properties assigned to prototype space are generally overridden in an object when created. If not then property value from prototype is fetched. Use of prototype makes objects smaller

*var empProto = function(){*

*};*

***empProto. prototype.Salary = 0;***

***empProto.prototype.getAnnualSalary = function(){***

***return this.Salary \* 12;***

***}***

*var emp1 = new empProto();*

*emp1.Salary = 1000; ///Salary gets overridden*

1. **Closures** are functions with preserved data. In JS it is an inner function having access to outer functions variables along with its own and global variables. Function addTo is a closure below that stores parameter value “passed”.

*var addTo = function (passed){*

*var add = function(inner){*

*return passed + inner;*

*}*

*return add;*

*};*

*var addThree = new addTo(3);*

*var addFour = new addTo(4);*

1. JS is a prototype based language and inheritance in JS is based on prototype. Every function by definition has a **property called prototype** which by default is empty and to which properties & methods can be added. Any object created from this function will inherit methods defined in its prototype. In JS can a constructor can be created directly, and then using this constructor an object can be created. Every function expression in JS is a constructor
2. All objects in JS are created from a master object i.e. **Object.** Each function is created from Object and both have a **prototype property of their own**
3. An object can be looked through in detail using Console.dir method. Other methods used for getting object info are **instanceof** and **Constructor**

*Var x = function(){*

*}*

***Console.dir(x);***

***Console.log(x.Constructor); //displays member variables with assigned values***

1. Sub classing in JS is achieved by writing base class constructor function and a sub class constructor function. A call to base constructor is made in sub class constructor, which will inherit properties and method defined in Base class. Prototype of base class is inherited by creating a prototype for subclass supplying **base class prototype** as parameter and **then set constructor.** Prototype method can be overridden in subclass after inheriting from base class prototype

***//Base class***

*var Job = function(){*

*this.pays = true;*

*}*

*job.prototype.print = function(){*

*console.log(this.pays);*

*};*

***//Sub class***

*var TechJob = function(title, pays){*

***Job.call(this); //calls Job constructor and makes Job parent class***

*this.title = title;*

*this.pays=pays;*

*}*

***//Inheriting from base class prototype***

*TechJob.prototype = Object.create(Job.prototype)*

*TechJob.prototype.contructor = TechJob*

***//Overriding Prototype method in subclass***

*TechJob.prototype.print = function(){*

*console.log (“This is a Tech Job which pays ” +this.pays);*

*};*

1. Methods can be **added to master object**, in case a specific method is needed for any object that is created

Object.prototype.print = function () {

console.log (“This comes from Object prototype”);

};

1. A **Web worker** can be used to **spawn a separate script in back ground** which enables doing complex calculation without disturbing UI. Web worker feature is bowser specific and may not be supported by all browsers. There is a **window space & a worker space**. From window space there is post message which passes an object. On worker space message is received and a call back function uses this passed object to do some processing. Worker will then send message back to page with post message. On window space a call back function receives this message from worker and then do something i.e. update DOM.

*<script>*

*if(window.Worker) {*

*var myWorker = new Worker("worker.js");*

*var message = {addThis: {num1:1, num2:3}};*

***//This passes an object to worker***

*myWorker.postMessage(message);*

***//this callback function waits for any response from worker***

*myWorker.onmessage = function(e){*

***///message from worker received and "result" extracted out***

*console.dir(e.data.result);*

*};*

*}*

*</script>*

*///****worker.js***

*this.onmessage = function (e) {*

***///e contains object that is sent from page***

*if(e.data.addThis !== undefined){*

***//performs operation, wraps in another object, result and posts back to page***

*this.postMessage(result: e.data.addThis.Num1 + e.data.addThis.Num2);*

*}*

*}*

1. Web worker doesn’t have access to document object, window object or parent object i.e. **cannot do DOM manipulation**. It has access to navigator object i.e. having access to useragent, geolocation, cookeyenabled. It has read only access to location object i.e. host, hostname, href, pathname etc. It has access to XmlHttpRequest and ApplicationCache. A worker **allows spawning other workers**
2. Arrow functions in ES6, defining functions using fat arrow expression

*Let mult = (a,b) => {return a\*b};*

This is equivalent to

*Mult = function (a, b) {*

*return a\*b;*

*}*

1. Arrow functions are good to use in cases where there are nested functions and there is need to **access this of parent function** from inner function. Arrow functions don’t support argument array which is available by default to normal function. However workaround is to use ES6 introduced syntax (…n)

*Var x* ***= (…N)*** *=> {return N [0]; };*

This is equivalent to

*X = function (){*

*return argument [0];*

*}*

1. Template String and other string functions are introduced in latest JS ES6, making it easier to do string operations. Use of back ticks (`) with string is more powerful & is useful with templates i.e. react JS. Back ticks are **used in interpolation** using $ symbol. Numeric are auto converted to String

*let x = `whatever`;*

*let y = 21;*

*let z = `my name is ${x} and age is ${y}*

*console.log(z); //****outputs ‘my name is whatever and age is 21’***

1. **Tagged templates** transform a template string by placing function name before string template. A string template is passed as an argument to a function, having 2 parts – a regular string and arguments. Both regular string and arguments are tokenized separately. This is **useful in internationalization**

*let one = 1*

*let two = 2*

*let tagged = function (strArray, ...N){*

*console.log (strArray)'*

*console.log (N);*

*}*

*//****separates string portion and arguments portion into separate arrays***

*tagged `adding ${one} and ${two} give ${one + two}`;*

1. ES6 has included new string function i.e. startWith, endWith, include, repeat. **Raw function** recently introduced keeps string as it is without interpreting chars like \n or other escapes

*let raw = String.raw `Not newline: \n`;*

*//****Outputs Not newline: \n` including newline escape***

*Console.log(raw);*

1. In JS an event bubbles up from child to parent while Capture happens from top to bottom. In normal case if child button has an event handler then it gets executed first followed by parent event handler. In capturing phase event handling is from top to bottom i.e. click on child button executes parent event handler first. Capture phase is enabled by setting last parameter as true in addEventListener

*var p = document.querySelector('#parent');*

*p.addEventListener('click' function(){*

*console.log("Parent clicked");*

*},* ***true****);* ***//setting true enables capture phase***

*var c = document.querySelector('#child');*

*c.addEventListener('click' function(){*

*console.log("Child clicked");*

*},* ***true****);*

1. JS supports recursion; however its usage is not advised as it depends on stack size of browser and can fail if recursive calls exceed browser stack size. Tail call optimization in next version is introduced to address this issue
2. **Var vs let:** A variable defined anywhere in a function using var gets hoisted to top of function as soon as it is referred inside. Let has block scoping unlike var which has functional scope and also variable defined using let doesn’t get hoisted.
3. Constants in JS are defined using **const keyword** or through **Object.defineProperty**

*Object.defineProperty (window, ‘PI’, {*

*Value: 3.14,*

***Writeable: false***

*});*

*Console.log(PI); //outputs 3.14*

1. **SetTimeout** is natively available in JS in global scope. It is not part of main JS engine but a part of web API having DOM related or browser related functions. It is used to run a piece of cade after a particular time interval. This can be cancelled using **clearTimeout**

*let timeOutId = windows.setTimeOut(function(), milliseconds)*

*clearTimeout(timeOutId)*

1. SetTimeout exhibits asynchronous behaviour. Any complex calculation could be wrapped into SetTimeout so while computation takes place, page continues to load. Calculation comes from event queue after complete page is loaded. JS is non-blocking as anything asynchronous would come through **event queue** after entire **execution stack** is emptied.
2. In JS there is an execution stack, events table and events queue. Normal statement like **console.log goes to execution stack**. SetTimeOut goes to event table and after timeout goes to **Event Queue**. Items on **event queue have lower priority than on event stack**. Once stack gets empty, then event loop takes items from event queue
3. **Setinterval** is similar to SetTimeout except that it will keep executing code at specified interval unless explicitly stopped using **clearInterval(timeOutId)**
4. **Bootstrap** is html & JS framework for developing **responsive mobile apps** on web. Bootstrap code includes a css and a JS file that can be added through a CDN to project

**JQuery**

1. JQuery is **lightweight JS Library** simplifying programming with JS. It has a large development community and many plugins. **JQuery 2.x** is supported for modern browsers only while 1.X works for older versions of IE also. JQuery 2.X is smaller as there is no backward compatibility code To use JQuery its library should be referenced in html file as below

*<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js"></script>*

*<script src="http://ajax.googleapis.com/ajax/libs/jqueryui/1.8.24/jquery-ui.min.js"></script>*

1. **$(document).ready (handler)** is JQuery event invoked when entire DOM is fully loaded before JS code gets executed. This in invoked before all images, css etc. are loaded. Other way is to put JS script at bottom of html body so that all elements get available before script is executed. **$(windows).load** is fired when DOM and all contents like images css are fully loaded. Most of JS script accesses DOM elements and document.ready **is good place for putting in JS code**. However some cases where **image dimensions** are needed to be accessed $(windows).load should be used since entire image needs to be loaded to get dimensions
2. **JQuery selectors** allow selecting HTML elements in DOM by ElementId, TagName, class Name, attribute etc. It uses JS function **document.getElemetById** behind scene**.**

$('#example') //example is ID attribute of html elememt

$('.small') //selects all elements by class small

$('.small', '.big') // selects all elements by class small or big

$('div.small', '.big') // selects div elements by class small or any element with class big

1. By default attribute value comparison is **case sensitive**. In order to make it case insensitive **comparison filter method** on returned element collection is used

*$('div[title]').****filter****(function(){*

*return $(this).attr('title').ToLowerCase() == "divTitle";*

*}).css('border', '5px solid red');*

This returns all elements with attribute having value **“divTitle”** in case insensitive manner and applies a 5px red border around those elements.

1. Comparison is also through regular expression

*$('div[title]').filter(function(){*

*return* ***RegExp*** *('Div', 'i').test($(this).attr('title').ToLowerCase();*

*}).css('border', '5px solid red');*

This applies a solid red border to all HTML elements having attribute title with value containing “Div” making **case insensitive comparison**

1. All input elements including textarea, select & button elements are selected using **$(:input)** while **$(input)** selects only those elements having input tag
2. To iterate over a particular type of field i.e. text field, use **type attribute** as filter to $input

*$(‘input [type="text"]').each (function () {*

*alert ($(this).val());*

*});*

Performance wise it is **better to use $(input[type = "text"]) over $(:input[type = "text"])** as $(:input[type = "text"]) scans all input elements including textareas and selects (combo boxes)

1. **JQuery variable with $ prefix** indicates that it is a JQuery object on which standard functions can be called i.e. $test.text();

Var $table = $(‘#htmlTable”) //declares table and assigns to element with id htmlTable

If($table.length > 0) //checks length of html element

$table.DataTable(

{

data: product //array of data items

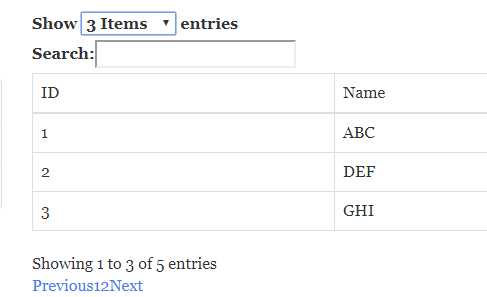
pageLength: 5

lengthMenu: [[3,5,10,-1], [‘3 items’, ‘5 items’, ’10 items’, ‘ALL’]]

}

);

Assigns DataTable plugin to html element, taking array of data items, and selection menu for no. of items to be displayed per page



1. Data Plugin reads JSON data from web server and renders it in tabular form providing functionality like search, pagination, items per page etc. This is through ajax property, to which JSON url is provided. A request is sent to web server and values are retrived in columns property as a collection.

*$table.DataTable({*

*lengthMenu: [[3,5,10,-1], ['3 Items', '6 Items', '10 Items', 'ALL']],*

*pageLength: 5,*

*ajax : {*

*url : jsonUrl,*

*dataSrc : ""*

*},*

*columns: [*

*{*

*data: 'name'*

*},*

*{*

*data: 'brand'*

*},*

*{*

*data: 'unitPrice'*

*},*

*{*

*data: 'quantity'*

*}*

*]*

*});*

This makes an ajax call to web app and from JSON response, retrieved data values for name,brand, unitPrice, quantity. A column value is modified through mrender function, provided as property to column

1. **JQuery checked selector** selects all radio button elements or checkbox elements as specified in filter. $('input[type="radio"]:checked') selects all radio buttons that are checked. $('input[type="checkbox"]:checked')

*$(document).ready(function () {*

*$('#btnSubmit').click(function(){*

***var result = $('input[type="radio"]:checked');***

*if (result.length > 0) {*

***//display selected radio button item at html element “divResult”***

*$('#divResult').html(result.val() + " is checked");*

*}*

***var result1 = $('input[type="checkbox"]:checked');***

*if (result1.length > 0) {*

*var resultString = result.length + " checkboxe(s) checked<br/>";*

***//iterates over checked checkbox collection***

*result1.each(function () {*

***//creates list of items line by line***

*resultString += $(this).val() +* ***"<br/>";***

*});*

*$('#divResult').html(resultString);*

*}*

*})*

*});*

1. To select values from **different groups of checkboxes**, name attribute identifying check box group should be used instead of type attribute. Since name is different for each group a variable is passed to selector i.e. var result **= $('input[name="' + groupName + '"]:checked');**

*$(document).ready(function () {*

*$('#btnGetSelectedSkills').click(function () {*

*getSelectedCheckBoxes('skills');*

*});*

*var getSelectedCheckBoxes = function (groupName) {*

***var result = $('input[name="' + groupName + '"]:checked');***

*if (result.length > 0) {*

*var resultString = result.length + " checkboxe(s) checked<br/>";*

*result.each(function () {*

*resultString += $(this).val() + "<br/>";*

*});*

*$('#divResult').html(resultString);*

*}*

*};*

*});*

1. **Accessing Text of selected checkbox or radio button:** In Web User Interface checkbox value is some cases is different from display text**. text() function doesn’t work for checkbox elements**. To make text accessible, label type should be used within html. Both label and box get displayed in same line

*<input type="checkbox" name="skills" value="js" />*

*<label* ***for="option-js****">JavaScript</label>*

*result.each(function () {*

*var selectedValue = $(this).val();*

*resultString += selectedValue + " - "*

*+* ***$('label[for="option-' + selectedValue + '"]').text()*** *+ "<br/>";*

*});*

1. To select all selected options from a select element **use: selected selector.** Selected options are iterated using JQuery implicit iterator

*$('#selectCountries').change(function () {*

***var selectedOptions = $('#selectCountries option:selected');***

*if (selectedOptions.length > 0) {*

*var resultString = '';*

***selectedOptions.each(function () {***

***resultString += 'Value = ' + $(this).val() +***

***', Text = ' + $(this).text() + '<br/>';***

***});***

*$('#divResult').html(resultString);*

*}*

*});*

1. JS provides **‘attr’ method** to get or set an attribute value of DOM element. Multiple attributes can be set on elements using a JSON object. This can also be done through method chaining

*$('div').attr({*

*'title': 'New Div Title',*

*'style': 'border:3px solid red;',*

*'name': 'My Div'*

*});*

Attributes can be removed through **removeAttr method**

*$('div[title]').removeAttr('title');*

1. **Wrap function** of JQuery wraps an html structure around each element in set of matched elements. **Unwrap** removes html structure. **Wrapinner** wraps html structure around **content of each html element.** Unwrap in this case will remove original elements since they are parent of wrapped element
2. **$.each** in JQuery is used to iterate over Java Script object

*var jsObject = {*

*'USA': 'Washington D.C.',*

*'India': 'New Delhi',*

*'UK': 'London',*

*'Australia': 'Canberra'*

*};*

*var result = '';*

***$.each*** *(jsObject, function (key, value) {*

*result += key + ' - ' + value + '<br/>';*

*});*

1. **each vs map: “**each” method is an immutable iterator returning original array while “map” method returns a new array. With each method, returning false **terminates iteration** while with map iteration cannot be terminated. Returning false does not apply processing to a specific array element but resumes for subsequent elements.
2. A **change event in JS** is fired on value change for input, textarea and select elements or when check box or radio button is clicked. For input, textarea and select elements change event is fired when control loses focus. For buttons and check boxes it is fired on a click. To handle change events on all input elements, input selector should be used instead of ID selector. A **class selector** can be better option as it is available on all element types. This requires adding a class name to HTML declaration

*<td><input id="txtLastName" type="text"* ***class="inputRequired"*** */></td>*

$('.inputRequired').**change** (function () {

        if (result == '') {

            result += $(this).val();

        }

        else {

            result += ', ' + $(this).val();

        }

        $('#divResult').html(result);

})

1. Within html, an icon can be added using **img tag pointing to path of image**. A text on a html form can be hidden using **style display: none**. Mouseover and mouseout events occur when mouse is moved over an element**. fadeIn** makes an hidden element visible while **fadeout** makes visible element hidden again. Mouseover and mouseout expect a callback as a parameter. **hover is shorthand** for mouseover and mouseout expecting 2 callback functions as parameters
2. When an event occurs, relevant data about that event is placed into the **event object**. This includes X and Y coordinates of the mouse pointer when the event occurred, the HTML element that fired the event, which mouse button is clicked etc. With JQuery event object is available to callback function as parameter. A **raw JS event object** can be converted to JQuery event object through **$.event.fix()** to make it work in previous browser versions. JQuery provides **one() method** that executes the handler at most once per element per event type.

*$(document).ready(function () {*

*$('#btn').click(function (event) {*

*getEventDetails(event);*

*})*

*function getEventDetails(event) {*

***var e = $.event.fix(event);***

*var eventDetails = "Event = " + e.type +*

*"<br/> X = " + e.pageX +*

*"<br/>Y = " + e.pageY +*

*"<br/>Target Type = " + e.target.type +*

*"<br/>Target Tag Name = " + e.target.tagName;*

*$('#divResult').html(eventDetails);*

*}*

*});*

1. A right click on a webpage triggers event “**contextmenu”.** preventDefault() function of this event prevents default browser action from happening i.e. preventing context menu from appearing on right click or getting redirected on clicking a hyperlink
2. A **Scroll event** applies to windows objects & scrollable frames or elements having **overflow CSS style** to scroll

<div style="width:300px; height:200px; border:1px solid black; **overflow:scroll**">

1. To detect which mouse button is clicked, **“event.which” and “event.button” properties** are provided. With JS event object working with IE8 or earlier version event.button property used. With later versions of IE event.which property is used
2. **Bind and Unbind methods** are used alternatively to bind handlers to events in JQuery. With JQuery 1.7 and higher, on()/off() methods get used instead of bind()/unbind(). bind supports binding to multiple events also

*$('#btnClickMe').bind('click', function () {*

*$('#divResult').html('Button Clicked');*

*});*

1. Event handler function automatically receives an event object. Event object has **a data property** using which data passed to an event is accessed. Data is passed using JSON object. If no data is passed data property of event object is null

$(document).ready(function () {

$('#btnClickMe').on('click', **{**

**firstName: 'John',**

**lastName: 'Doe'**

**}**, sayHello);

function sayHello(event) {

if (event.data == null) {

alert('No name provided');

}

else {

alert('Hello ' + event.data.firstName +

(event.data.lastName != null ? ' ' + event.data.lastName : ''));

}

}

});

1. Adding event to dynamically created object i.e. creating new list item on click of a button. By default when page loads event handler gets added only to existing items on page. This needs to be done explicitly while creating a new item

*$('#btnAdd').on('click', function () {*

*var newListItem = $('<li>New List Item</li>').on('click', function () {*

*$(this).fadeOut(500);*

*});*

*$('ul').append(newListItem);*

*});*

1. **Event delegation** allows attaching a single event listener, to a parent element, that will fire for all descendants matching a selector, can be done through on() and delegate() functions. Binding event handler to unordered list ('ul') will result only in one copy of event handler in memory. There is no need to add handler now to each created item. Whatever selector is specified within on() method, event of that selector is bubbled up to parent

*$('ul').on('click',* ***'li',*** *function () { //****click event on all <li> items will bubble up to parent***

*$(this).fadeOut(500);*

*});*

*$('#btnAdd').on('click', function () {*

***//new list item gets dynamically added as <li> child***

*$('ul').append('<li>New List Item</li>');*

*});*

1. An event can be attached to an HTML element in multiple ways using JQuery, raw JS or in plain html

//**adding event in plain HTML**

*<input type="button" value="Show Contents"* ***onClick="******clickHandler()"*** */>*

//**adding event in raw JS**

*document.getEventById('btn').addEventListener('click',* ***clickHandler****);*

1. Adding multiple event handlers can be prevented using off() and on() methods in chain

*$('#btn').off('click').on('click', function () {*

*alert('Button Clicked');*

*});*

1. An attribute on an element can be set or removed using **“attr” and “removeAttr” methods.**

*$(this).attr('disabled', 'disabled');*

*btnStop.removeAttr('disabled');*

$.inArray returns index of an element in an array

1. JQuery animate function enables animating CSS properties, provided as JSON object in argument. Animate has 2 variations.

.*animate( properties [, duration ] [, easing ] [, complete ] )*

*.animate( properties, options )*

First version has optional parameters **- easing & complete**. Easing is used to change speed/direction of animation while in progress. Complete is a callback function that executes after animation is completed. In animation call below, font size of div element increases up to 50 points within 1000 milliseconds, providing animation effects as per **default easing** value

*$('#myDiv').animate({'font-size': '50'}, 1000);*

1. To change position of object on html, **position property** is set in css to **relative, fixed or absolute.** Animate function can then be chained creation a moving effect on an object

*$(document).ready(function () {*

*$('#myImage').click(function () {*

***//chaining of animate method on an image object where ‘left’ & ‘top’ are position properties***

*$(this)*

*.animate({ 'left': '300' })*

*.animate({ 'top': '200' })*

*.animate({ 'left': '10' })*

*.animate({ 'top': '10' });*

*});*

*});*

When animate calls are chained, they are put into a **queue (fx)** to be executed serially. A queue is associated with a div element

1. In order to perform **animation in parallel** css properties should be included as a single json object. This uses second version of animate with queue property set to false

$('#myDiv1')

.animate({ 'width': 500 }, { duration: 1500, queue: false })

.animate({ 'padding': 20 }, { duration: 1500, queue: false })

1. A class in JQuery is defined as a JSON object with a set of properties like background- color, font-size, width etc. Once applied to an element, these properties are rendered on html page. Value of a property can be obtained or modified using .css method.
2. Present versions of browsers **allow change of HTML UI element property ‘type’** from within JQuery. This is useful in cases of show password option where input type is changed from ‘password’ to ‘text’ when show password checkbox is selected. **In previous browser versions**, approach is to remove existing UI element and add new element of text type.
3. A **floating div element** is always visible as page is scrolled. This is achieved by setting css property ‘position' on element **as 'fixed' when scrolling out of page**. Otherwise position is set to relative. A fixed position is relative to browser window while relative position is relative to frame

$(document).ready(function () {

            var floatingDiv = $("#divfloating");

            var floatingDivPosition = floatingDiv.position();

            $(window).scroll(function () {

                var scrollPosition = $(window).scrollTop();

                if (scrollPosition >= floatingDivPosition.top) {

**floatingDiv.css({**

**'position': 'fixed',**

**'top': 3**

                    });

                } else {

                    floatingDiv.css({

                        'position': 'relative',

                        'top': 0

                    });

                }

            });

        });

1. AJAX (Asynchronous Java script and XML) allows parts of page to be updated without reloading entire page. Server sends only limited data required by page such as help text for selected text box. **JQuery AJAX load method** loads data from a server and puts the returned data into the selected element.

*$(selector).load(URL,data,callback);*

**URL is resource on server** to which request is sent, **data in optional JSON object** or string that is sent to the server along with the URL request, **complete is callback** function called when the request completes. Code below loads HTML data from the server. When a text box receives focus, the help text associated with that field is loaded from HTML on server and displayed. When the focus is lost help text disappears.

*$(document).ready (function () {*

*var textBoxes = $('input[type="text"]');*

*textBoxes.focus(function () {*

***var helpDiv = $(this).attr('id') + 'HelpDiv';***

***$('#' + helpDiv).load ('Help.html #' + helpDiv);***

***});***

*textBoxes.blur(function () {*

*var helpDiv = $(this).attr('id') + 'HelpDiv'; $('#' + helpDiv).html('');*

*});*

*});*

1. AJAX load function can also be used to access data from a database on server. On a webserver, an ASPX page is configured to load data from SQL server DB using ADO. Net and C#. Within html page then this **APSX page is provided as parameter** to JQuery load function

*var textBoxes = $('input[type="text"]');*

*textBoxes.focus(function () {*

*var helpDiv = $(this).attr('id');*

***$('#' + helpDiv + 'HelpDiv').load('GetHelpText.aspx', { HelpTextKey: helpDiv });***

*});*

1. Callback function has 3 parameters – **response, status, xhr**. If Ajax request completes successfully, response parameter contains response of request else it contains information about exception. Status tells if request has completed successfully or an error has occurred, **xhr contains more information about status** i.e. status text, status etc.
2. **JQuery get** can be used to load any form of data from server - xml, json, script, html while **load function only loads html data** from server. Load function requires a selector first. The HTML content retrieved from the server is automatically inserted into the DOM elements matched by the selector. With **get() function** selector is not required first. Instead callback function is used to insert the data received from the server into the DOM elements

*$.get('url', 'data', function (response) {*

*$('selector').html(response);*

*});*

1. JQuery load function can be used to make **either a get or post request**, If data parameter is specified to load, POST request is issued. JQuery also has post method to do same. When data is sent using GET request it is appended to URL. With POST request it is included in body. With GET there is **size limit on data** depending on client & server
2. JQuery load(), get(), post() are wrapper methods & **use ajax() under hood**. ajax() method takes a **json object in form of settings**, in which properties are set based on get()/post() requests

*$.ajax({*

*url: 'GetHelpText.aspx',*

*data: { HelpTextKey: helpDiv },*

*success: function (response, status, xhr) {*

*var jQueryXml = $(response);*

*var textElement = jQueryXml.find("Text");*

*$('#' + helpDiv + 'HelpDiv').html(textElement.text());*

*},*

*dataType: 'xml',*

*method: 'post'*

*});*

1. To enable an ASP. Net web service to be called by JS web service should be **decorated with [System.Web.Script.Services.ScriptService].** Web service is invoked using JQuery ajax() method. When a button is clicked, JQuery ajax() invokes web service method **'EmployeeService.asmx/GetEmployeeById'** passing it a parameter empId. XML data is returned from web service that gets parsed using **JQuery find method** and is set in HTML UI fields

*$('#btnGetEmployee').click(function () {*

*var empId = $('#txtId').val();*

*$.ajax({*

*url: 'EmployeeService.asmx/GetEmployeeById',*

*data: { employeeId: empId },*

*method: 'post',*

*dataType: 'xml',*

*success: function (data) {*

*var jQueryXml = $(data);*

*$('#txtName').val(jQueryXml.find('Name').text());*

*$('#txtGender').val(jQueryXml.find('Gender').text());*

*$('#txtSalary').val(jQueryXml.find('Salary').text());*

*},*

*error: function (err) {*

*alert(err);*

*}*

*});*

*});*

1. To handle JSON data from ASP .Net web service there are two approaches - In first approach only JQuery code is modified by using **contentType option & using JSON.stringify() method** to convert JSON data sent as JSON string

*$.ajax({*

*url: 'EmployeeService.asmx/GetEmployeeById',*

***contentType: "application/json; charset=utf-8",***

***data: JSON.stringify({ employeeId: empId }),***

***dataType: "json",***

*method: 'post',*

*success: function (data) {*

***$('#txtName').val(data.d.Name);***

*$('#txtGender').val(data.d.Gender);*

*$('#txtSalary').val(data.d.Salary);*

*},*

*error: function (err) {*

*alert(err);*

*}*

*});*

1. To invoke an **aspx page method** from code behind file using JQuery, the method should be made **static and decorated with [System.Web.Services.WebMethod]**

*public partial class WebForm1 : System.Web.UI.Page*

*{*

***[System.Web.Services.WebMethod] //This is invoked using JQuery***

*public static Employee GetEmployeeById(int employeeId)*

*{*

*Employee employee = new Employee();*

*string cs = ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;*

*using (SqlConnection con = new SqlConnection(cs))*

*{*

*SqlCommand cmd = new SqlCommand("spGetEmployeeById", con);*

*cmd.CommandType = CommandType.StoredProcedure;*

*cmd.Parameters.Add(new SqlParameter()*

*{*

*ParameterName = "@Id",*

*Value = employeeId*

*});*

*con.Open();*

*SqlDataReader rdr = cmd.ExecuteReader();*

*while (rdr.Read())*

*{*

*employee.ID = Convert.ToInt32(rdr["Id"]);*

*employee.Name = rdr["Name"].ToString();*

*employee.Gender = rdr["Gender"].ToString();*

*employee.Salary = Convert.ToInt32(rdr["Salary"]);*

*}*

*}*

*return employee;*

*}*

1. **Window height is browser viewport** i.e. size of window within browser while **document height** contains everything even above or below visible area. These along with scrollbar position are used to detect if user has scrolled to the bottom of the page

*$(document).ready(function () {*

*var floatingDiv = $("#divfloating");*

*var floatingDivPosition = floatingDiv.position();*

*$(window).scroll(function () {*

*var scrollPosition = $(window).scrollTop();*

***if (scrollPosition == $(document).height() - $(window).height()) {***

***floatingDiv.html('Reached the bottom of the page');***

***}***

*});*

*});*

1. In order to use **JQuery autocomplete widget, JQueryUI is required**, from where JQuery autocomplete widget can be used. This can be obtained from JQueryUI site. By default all UI widgets are selected for download but this behaviour can be customized. Minified versions should be used for prod environment, Files required to be present in project are jquery-ui.js, jquery-ui.css and images and they are to be referenced in JavaScript section of web form or HTML. Autocomplete method gets enabled with input box after these files are included, accepting optionally settings as argument as a JSON object.
2. Creating **Auto complete input box using JQuery** with ASP. Net Application
   1. At backend there is a stored procedure that selects name from a DB table where name matches whatever typed in text box
   2. In ASP .Net form there is a Name text box that should suggest names as user starts typing, and a complete name can be chosen from suggestions
   3. A DB connection string to a database is included in **web.config file**
   4. A generic handler to project (.ashx) is added to project. This handler implements IHttpHandler interface. It has a processRequst method that should be overridden to contain code to retrieve data from database. Data is serialized to JSON format using Java Script serializer.
   5. The handler code **reads connection string from web.config file** using **ConfiurationManager** and it creates an **SQLConnection object** using that connection string. Then **SqlCommand object** is created and used to execute stored procedure. Command type is set to storedProc and assigned parameter. **Parameter name** is the one **passed to stored procedure.** Connection object is opened and command executed using **EndExecuteReader ().** Result is obtained in **SQLDataReader**. Each row is looped through to create a list of names. Instance of **JavaScriptSerializer** **class** is created and list of **names** serialized **to json format**, and write to response stream
   6. Add a web form to project now, add to it ASP.Net text box, that has autocomplete enabled. Its source option is to be set to **handler (ashx)** that actually returns suggestions as a dropdown
3. Autocomplete source option can be **explicitly set to array of strings** or an array of objects with **label/value properties**. In this case label properties get displayed under suggestions and value gets inserted into Text box when item is selected
4. Using Autocomplete with ASMX web service
   1. Add a web service (ASMX) to project above & enable calling from script. There is no context object is web service hence a string argument is passed to method. List of names is returned, no serialization is required
   2. Source option can be set as a callback function, expects a **request object &** **response callback function** as argument. Request.term property (argument to stored procedure) will contain what user will type in text box. **term property** can also come from text box through text box ID, but it is preferred getting from request object
   3. Ajax request is issued with url property as service, request issued as post, and content type set as application/json since json string is passed to server. Data passed is converted to json string using JSON.Stringify method to which JS object having a term property is passed. Type of data expected from server is again JSON and should be specified in dataType property.
   4. Response argument represents a callback function. **Data retrieved from web service** **is passed to this callback function** and will take care of showing suggestions for textbox. There is also **Minlength option** that specifies no. of characters entered before search is performed.
5. Accordion is used to display collapsible content panels used for presenting info in limited amount of space. HTML for accordion panel needs pairs of header & content panels. **accordion() function** can then be called on container div element
6. Using Accordion to display dynamic content ie Country and Description from database
   1. In ASP. Net project create a class containing fields corresponding to columns in DB table ie Country
   2. Add ASMX service to project and add method GetCountries() that returns a list of country objects
   3. From JQuery issue an Ajax request, loop through returned list using JQuery each(), dynamically build header and content panels required by accordion.
   4. Set html string created as inner html for accordion container and invoke accordion() method. Accordion has **collapsible property** used to close all panels. By default one panel is always active. Options are specified through Java Script object. **Active property** is set to false if at initial load all panels are required to be collapsed,. It can be set to an **integer value** if a specific panel is to be made active
7. Using **APS.NET repeater ctrl** with accordion
   1. Add a reference to country service created. Add a web form to project. Inside accordion container add repeater control. In code behind file get countries from WS and bind to repeater ctrl.
   2. Within repeater specify structure required. Use <ItemTemplate> and Eval() methods for binding. Same can be done with list view ctrl also

*<div id="accordion" style="width: 600px">*

*<asp:Repeater ID="repeaterCountries" runat="server">*

*<ItemTemplate>*

*<h3>*

*<%#Eval("Name") %>*

*</h3>*

*<div>*

*<%#Eval("CountryDescription") %>*

*</div>*

*</ItemTemplate>*

*</asp:Repeater>*

*</div>*

1. Producing Tabs in JQuery: html for tabbed widget is an ordered or unordered list and tab() function is then applied to container. Dynamic tab elements can be generated using repeater control
2. Date Picker plugin: It requires an HTML textbox on which **datepicker() method** is applied. It is highly customizable. Options can be specified using JS object and passed to datepicker function. In order to **localize Date Picker**, download corresponding localized version of file an include in script section
3. A slider can be added to a div element by applying **slider() method**. Options can be specified as JS object, Slider options support start, stop & slide events.
4. **JQuery tooltip widget** allows a lot of customization compared to native tooltip. Use tool tip function on element selector. To use tooltip widget on all elements use document object

*$('#txtName').tooltip();*

*$(document).tooltip();*

1. **Content option** of tooltip function can be used to provide content for tooltip, overriding **existing title attribute**. Html can also be used within content for tooltip effects. It also supports a function that can used to load tooltip using AJAX from database. Use track option for tooltip to follow mouse pointer. **Show/hide options** specify animation for tooltip. **effect property** provides different types of animation effect

*$('#txtName').tooltip({*

*content: '<b><u>tool tip content</u></b>'*

*track: true*

*show : {delay:10, duration:500, effect: 'slideDown'},*

*});*

1. Ajax call in JS is used through various options. **URL option** has specified web service method that JS Ajax call should invoke. **method option** specifies request type ie post or get. **data option** contains parameter passed to web service method as json object. **dataType option** is format expected back from server. **success option** specifies a callback function **that receives data** from which info required is retrieved (i.e. tooltip from database by stored procedure).
2. **async option** should be set to false in Ajax call as function returns asynchronously before return value is populated, asyc value false ensures execution of success callback only after web service has returned

*$('.displayTooltip').tooltip({*

*content: getTooltip*

*});*

*function getTooltip() {*

*var returnValue = '';*

*$.ajax({*

*url: 'TooltipService.asmx/GetTooltip',*

*method: 'post',*

***//$(this) is context from which id is used, here element collection retuned by class selector***

*data: { fieldName: $(this).attr('id') },*

*dataType: 'json',*

***async: false,***

*success: function (data) {*

*returnValue = data.TooltipText;*

*}*

*});*

*return returnValue;*

*}*

1. Progress bar can be applied using **$('#progress').progressbar() on div element**. There is determinate and indeterminate progress bar. For determinate progress bar actual status can be determined through **value option** that should be set to integer value or false. For **indeterminate progress bar** value is set to false. It is used with uploading files i.e. on email attachment
2. To add text on Progress bar a span element is included within progress bar <div> element and **position property relative** is applied to it using style attribute. Style attribute should also set on **parent <div> element** **with position relative**; else text positioning happens relative to browser window. In code below when a button is clicked, progress bar gets updated and a **change event** is raised. Text on progress bar is updated in response to change event within an anonymous function

*$('#btn').click(function () {*

*progressbarDiv.progressbar({*

*value: parseInt($('#ddlPercentage').val()),*

*change: function () {*

*$('#progressBar-label').text(progressbarDiv.progressbar('value') + '%');*

*}*

*});*

*});*

1. To process file upload from within JQuery use **FormData** to set data on Form which in this case are uploaded files. Setting **contentType to false** tells JQuery not to set any content header (used with raw data). By default JQuery will try to convert form data to string and setting **processData to false** prevents this behaviour in case of files. Within success callback progress bar text is set to complete **using text() method**. **fadeout()/fadein()** method enables fading in/out of progress bar within time specified

*$("#btnUpload").click(function (event) {*

***var files = $("#FileUpload1")[0].files;***

*if (files.length > 0) {*

*var formData = new FormData();*

*for (var i = 0; i < files.length; i++) {*

*formData.append(files[i].name, files[i]);*

*}*

*var progressbarLabel = $('#progressBar-label');*

*var progressbarDiv = $('#progressbar');*

*$.ajax({*

*url: 'UploadHandler.ashx',*

*method: 'post',*

*data: formData,*

***contentType: false,***

***processData: false,***

*success: function () {*

***progressbarLabel.text('Complete');***

***progressbarDiv.fadeOut(2000);***

*},*

*error: function (err) {*

*alert(err.statusText);*

*}*

*});*

*progressbarLabel.text('Uploading...');*

*progressbarDiv.progressbar({*

*value: false*

*}).fadeIn(500);*

*}*

*})*

1. To get **menu widget** create a structure in html with parent child relationship, most commonly used html element is unordered list. A JQuery menu is applied on html list using menu() method. Dimensions on menu can be changed by controlling css class applied to menu. This can be found using browser developer tools (F12)

*$('#menu').menu()*

1. An item can be disabled by applying **class "ui-state-disabled"**. Icons can be added to menu by applying class names within a span element. Icons are available at **api.jquery.ui/theming/icons**

*<li class="ui-state-disabled">MP</li>*

*<li><span class="ui-icon ui-icon-flag"></span>USA</li>*

1. Different icons can be **applied to submenu using icons option and submenu property** within JQuery; menu is disabled using disable option. Selected menu item text can be obtained from **select event**

*$('#menu').menu({*

*icons: { submenu: 'ui-icon-circle-arrow-e' }*

***select:******function (event, ui) {***

*alert(ui.item.text());*

*}*

*});*

1. JQuery allows **adding class dynamically** to a list item using **AddClass method**. **appendTo** allows list item to be added dynamically to unordered list <ul>. Similarly <ul> can be appended to <li> using appendTo

*function buildMenu(parent, items) {*

*$.each(items, function () {*

*var li = $('<li>' + this.MenuText + '</li>');*

*if (!this.Active) {*

***li.addClass('ui-state-disabled');***

*}*

***li.appendTo(parent);***

*if (this.List && this.List.length > 0) {*

*var ul = $('<ul></ul>');*

***ul.appendTo(li);***

*buildMenu(ul, this.List);*

*}*

*});*

1. **JQuery select menu** extends functionality of native html select. Apply **selectmenu() method** to select structure in native html. **Optgroup element** is used to group related options in dropdown list, group is not selectable

*<select id="selectMenu">*

*<optgroup label="USA">*

*<option value="1">New York</option>*

*<option value="2">Los Angeles</option>*

*</optgroup>*

*<optgroup label="India">*

*<option value="3">New Delhi</option>*

*<option value="4">Bangalore</option>*

*</optgroup>*

*</select>*

1. **Button icon** is image seen on right hand side of dropdown list, default icon is **ui-icon-triangle-1-s** that can be changed. JQuery width option is used to control width of select menu. Select event is used to retrieve value of selected item

*$('#selectMenu').selectmenu({*

*icons: { button: 'ui-icon-circle-arrow-s' }*

*width : 200*

*select: function (event, ui) {*

*alert('Label = ' + ui.item.label + ' ' + 'Value = ' + ui.item.value);*

*}*

*});*

*});*

1. From within JQuery tbody and thead section are accessible through selectors. To add a row dynamically to tbody section, obtain a ref to tbody section, create a <tr> element and append it to tbody section

***var tbody = $('#employees tbody');***

*success: function (data) {*

*$(data).each(function () {*

***var tr = $('<tr></tr>')***

***tr.append('<td>' + this.FirstName + '</td>')***

***tr.append('<td>' + this.LastName + '</td>')***

***tr.append('<td>' + this.Email + '</td>')***

***tbody.append(tr);***

*})*

*}*

1. To create a dialog in JQuery use **dialog() method on div element**. Dialog widget can be customized through options. A dialog can be provided arguments ie **‘open’, 'close'.** Buttons can be added on to a dialog through buttons property

*dialogDiv.dialog({*

***autoOpen: false, //dialog doesn’t pop up until an event occurs***

*modal: true,*

*buttons: {*

*'Create': CreateEmployee****,//Create is button title, CreateEmployee is executed***

*'Cancel': function () {*

*dialogDiv.dialog('close');* ***//closes open dialog***

*clearInputFields();*

*}*

*}*

1. **Datatable is JQuery plugin** that renders data in tabular form providing search, sorting pagination support. Datatable plugin has ccs & js files that should be included in JQuery header. These files are downloadable **from datatables.net**

*<link rel="stylesheet" type="text/css" href="//cdn.datatables.net/1.10.7/css/jquery.dataTables.min.css" />*

*<script src****="//cdn.datatables.net/1.10.7/js/jquery.dataTables.min.js">***

*</script>*

1. To render using datatable plugin apply **datatable() method** table defined in html. For customization, **html table can be wrapped in a <div> element** and requisite properties can be applied to div element that the get applied to datatable plugin

<*div style="width: 500px; border: 1px solid black; padding: 5px">*

*$('#datatable').datatable(); // datatable is ID of html table*

**padding attribute** in html moves controls off the border by specified pixels

1. Datatable accepts options in form of JSON object. A **render event handler** option with data allows displaying data in specified format i.e. Date in “dd/mm/yy” format instead of string received from backend. To **exclude data from a particular column** in search or from sorting, make option **'searchable' as false** for that column data and **'sortable' as false.** Bydefaultthese are true. To turn off paging on entire datatable, set **option paging** to false. To fix height of table set **scrollY to a value** , scroll bar appears as defaultif length of data exceeds table height

$('#datatable').dataTable({

paging: true, **//enable data in multiple pages**

sort: true, **//sortable on columns**

searching: true, **//columns searched when string entered in search box**

scrollY: 200, **//fixes height to 200 pixels**

data: data,

columns: [

{ 'data': 'Id' },

{ 'data': 'Name' },

{ 'data': 'Gender' },

{

'data': 'WebSite',

'sortable': false, **//Sorting control will not be present with this col**

'searchable': false, **//Website col will not be searched**

**'render': function (webSite) //displays col Website as hyperlink**

{

return '<a href=' + webSite + '>'

+ webSite.substr(0, 10) + '...' + '</a>';

}

},

{

'data': 'HireDate',

**//converts Date received to “dd/mm/yyyy” format**

**'render': function (jsonDate)** {

var date = new Date(parseInt(jsonDate.substr(6)));

var month = date.getMonth() + 1;

return month + "/" + date.getDate() + "/" + date.getFullYear();

}

}

]

});

1. To add a search box dynamically within footer for each column in datatable, use code below. While putting search box in column header instead of footer, in order to prevent sort behaviour on clicking inside search box, **stopPropogation() method** should be used

***$('#datatable tfoot th').****each(function ()* ***//retrieve all <th> elements in footer***

*{*

***//for each <th> element, obtain its corresponding index in header, get title***

*var title = $('#datatable thead th').eq($(this).index()).text();*

*//****create a search box dynamically, add informative text through ‘placeholder’***

*$(this).html('<input type="text" placeholder="Search ' + title + '" />');*

*});*

1. To enable search, get columns from datatable plugin in form of collection using **datatableinstance.columns().**To search a column for a value use **datatablecolumn.search()** method. Iteration over all columns is done with **every() method and draw() method** retrieves renders matching rows in datatable plugin. Use **on() method** to associate keyup and change event to search text box

datatableInstance.columns().every(function () {

var dataTableColumn = this;

var searchTextBoxes = $(this.header()).find('input');

**searchTextBoxes.on('keyup change', function ()** { //keyup and change events are added

dataTableColumn.search(this.value).draw();

});

1. Controlling visibility of columns in datatable plugin through links on page: To set hyperlink for columns use anchor element along with **data-columnindex** property. Links are made clickable use href attribute. All **anchor elements** are retrieved using **class selector** and column can be retrieved using id instance. To toggle visibility use **visible() function of column**

*<div style="padding: 5px; padding-left: 0px">*

*<b>Show/Hide Column : </b>*

***<a class="showHideColumn" data-columnindex="0">Id</a> -***

*<a class="showHideColumn" data-columnindex="1">First Name</a> -*

*</div>*

*$('.showHideColumn').on('click', function (e) {*

*e.preventDefault();*

*var tableColumn =*

*datatableInstance.column($(this).attr('data-columnindex'));*

***tableColumn.visible(!tableColumn.visible());***

*});*

1. Server side processing using ASP. Net handler: With ASP. Net handler parameters sent from JQuery Ajax call are received in **context parameter of ProcessRequest method** of handler, and can then be retrieved inside ProcessRequest code. With web service , method receives as parameters
2. To execute stored procedure from within asp handler method, use **SQLCommand object**, setting its command type as stored proc. Create **SQLParameter instance** & Add parameters to command object. Use **ExeuteReader** of command object to execute stored procedure and **receive results in SqlDataReader.**
3. SQLCommand can also be used to execute SQL queries in which case command type is not set. **ExceuteScalar** is used in case query returns an aggregate.
4. For compatibility with datatable plugin, a JSON object returned for ASP .Net handler should have **documented properties** in plugin, & this object should be serialized using JavaSCriptSerializer class and written to **context object.**
5. Within jQueryResource that provides server side data is given by **option sAjaxSource**. To enable processing on server **bSeverside option** is set as true
6. Server Side processing using Asp.Net web service: With web service data is posted from JQuery Datatable plugin to service method (**sServerMethod: 'post'**)
7. To **enable exporting Datatable plugin data** to PDF, copying to clipboard, printing, exporting to excel or CSV, table tools plugin for Datatable is used. It is available at <https://www.datatables.net/download/index>

*<script src="//cdn.datatables.net/tabletools/2.2.4/js/dataTables.tableTools.min.js">*

*<link rel="stylesheet" type="text/css"*

*href="//cdn.datatables.net/tabletools/2.2.4/css/dataTables.tableTools.css" />*

1. Within JQuery get reference to Datatable & table tool plugin. **fnContainer() function** of table tools will return a container in which all buttons will be present. Use insertBefore function to make container appear before Datatable wrapper. **datatable\_wrapper** wraps entire content of datatable as can be seen from developer tools. This div element is injected by JQuery

*var table = $('#datatable').dataTable();*

*var tableTools = new $.fn.dataTable.TableTools(table)*

*$(tableTools.fnContainer()).insertBefore('#datatable\_wrapper');*

1. Table tools rely on **flashswf files** for actions these can be found on link **cdn.datatables.net/tabletools/2.2.4/swf**. There are 2 files and one has capability to export to pdf. Options to table tool plugin can be specified using **json object.** This includes **sSwfPath.** After adding this option function of buttons gets enabled. No. of buttons can be Customized using **option aButtons,** which isan array specifying what button are required. To further customize print option use a JS object, **specify sExtends property** for button to extend (print in this case), set **bShowAll** to false. This can be done in cases **where only current page rather than all are to be printed**. To customize xls option, use JS object, **set sFileName option to \*.xls**. To change button text use sButtonText option. **In PDF** if header/footer are not needed, set **bFooter/ bHeader to false**

$(document).ready(function () {

var table = $('#datatable').dataTable();

var tableTools = new $.fn.dataTable.TableTools(table, {

'aButtons': [

{

'sExtends': 'xls',

'sButtonText': 'Save to Excel',

'sFileName': 'Data.xls'

},

{

'sExtends': 'print',

'bShowAll': true,

},

{

'sExtends': 'pdf',

'bFooter': false

},

'copy',

'csv'

],

'sSwfPath': '//cdn.datatables.net/tabletools/2.2.4/swf/copy\_csv\_xls\_pdf.swf'

});

$(tableTools.fnContainer()).insertBefore('#datatable\_wrapper');

});

**HTML Syntax**

1. To **enable native tool tip** for an element it should have title attribute

*<input id="txtName" type="text" title="Your full name as it appears in passport" />*

1. To associate a **label with text box** use for attribute with label providing it with textbox ID

*<label id="lblName" for="txtName" >Name</label>*

*<input id="txtName" type="text" title="Your full name as it appears in passport" />*

1. **Element <table>** defines a table in html form. <tr> within table represent a row <td> a cell within in a row. There is a **border property** that if set displays border around table cells. By default border is thick & setting **collapsible property** makes border thin. A table can have a **<thead>, <tbody> and <tfoot> section**

*<table id="employees" style="border-collapse: collapse" border="1">*

1. A **class** can be used to select a group of elements on form. To make this work elements should have an class **attribute** with specified value

*<input id="firstName" class="displayTooltip" title="" type="text" />*

*$('.displayToolTip').tooltip()*

1. A combo box or select element is defined in html using <select> along with <option> <input> elements sepcified after <select> are placed on same line

Select :

<select id="ddlPercent">

<option value="10">10</option>

<option value="20">20</option>

<input type="button" id="btn" value="Set Value" />

1. **Styles** can be applied to html elements **through css method**

*$('[title = "myTitle"]').css ('border', '5px solid red');*

This applies a solid red border to all HTML elements having an attribute title with value myTitle

1. Using **style attribute** on div or span element properties can be set i.e. width, height, position etc. Container div properties get applied to child div elements ie progress bar width. To position span element relative to its parent div element use **position property** on **parent div element as relative** and on **span element as absolute.** To hide div element set **display property** as none

<div style="width: 300px">

<div id="progressbar" style="**position: relative; display: none**">

<span style="**position: absolute**; left: 35%; top: 20%" id="progressBar-label">

Uploading...

</span>

</div>

</div>

1. An unordered list is created starting with **<ul> tag** i.e. <ul id="menu">. Items in unordered list are represented with **<li> tag.** An <li> element can further include a <ul> as a child
2. An **anchor elemen**t is used to make enclosed element clickable

'<a href=' + webSite + '>'+webSite+ '</a>

1. **Classes** can be added or overridden within **style section in html**. Classes are applied to JQuery widgets by default ie ui-menu class applied to menu widget

*<style>*

*.ui-menu { width: 150px; }*

*</style>*

*<style>*

*.showHideColumn {*

*cursor: pointer;*

*color: blue;*

*}*

*</style>*

**ASP. Net Application**

1. Creating a stored procedure

create proc spGetTooltip

**@FieldName nvarchar (50**) //parameter is specified using @

as

Begin

**///Stored procedure starts here**

Select \* from tblTooltip where FieldName = @FieldName

End

1. A stored procedure is accessed within **web service method using ADO.NET** connection is created to database, stored procedure is invoked & result is returned. JS serializer class serializes object returned by web service and writes serialized string to response. A stored procedure can be used to get data from multiple tables in which case there can be multiple select statements.
2. Code to load data from DB executing stored procedure can also be written **in code behind file**. Stored procedure is alternatively executed using **SqlDataAdapter** and results are returned into **a Dataset object**. DataSet object will contain a **collection of tables** which can be iterated using DatRows
3. Connection string to database is included in web.config file

<add name="DBCS"

      connectionString="server=.;database=SampleDB;integrated security=SSPI"/>

1. A class file encapsulating fields in database is added to the project (i.e. ToolTip.cs) Object of this class would be created in web service method.

public class Tooltip {

public string FieldName { get; set; }

public string TooltipText { get; set; }

}

1. To upload files from a ASP. Net web service, add an upload folder to ASP project for holding uploaded files. Add a **generic handler** for doing actual file upload, making use of **context object** coming into method as parameter

*public void ProcessRequest(HttpContext context)*

*{*

*if (context.Request.Files.Count > 0)*

*{*

*HttpFileCollection selectedFiles = context.Request.Files;*

*for (int i = 0; i < selectedFiles.Count; i++)*

*{*

*System.Threading.Thread.Sleep(1000);*

*HttpPostedFile PostedFile = selectedFiles[i];*

*///Path.GetFileName is used to make it qork across mutlipe browsers*

*String FileName = context.Server.MapPath("~/Uploads/" + Path.GetFileName(PostedFile.FileName));*

*PostedFile.SaveAs(FileName);*

*}*

*}*

*}*

1. **Asp: FileUpload control** in a web form allows selection of files for uploading. To enable multiple file selection set is **allowMultiple property** to true.
2. **A repeater control** is used to create an html structure dynamically; it is encapsulated inside a <div> or a <select> element. To use option groups inside a repeater, optgrp is used. To populate repeater, bind data returned in code behind to repeater

*protected void Page\_Load(object sender, EventArgs e)*

*{*

*repeaterCountries.DataSource = GetSelectMenuData();*

*repeaterCountries.DataBind();*

*}*

1. This binds object returned to repeater. Within repeater html structure in web form file repeater properties can be bound to element attributes using **Eval function.** Multiple repeaters can be nested within a repeater

***<asp:Repeater ID="repeaterCountries" runat="server">***

*<ItemTemplate>*

***<optgroup label="<%#Eval("Name") %>">***

*<asp:Repeater ID="repeaterCities" runat="server"*

*DataSource='<%# Eval("Cities")%>'>*

*<ItemTemplate>*

***<option value="<%#Eval("Id") %>">***

*<%#Eval("Name") %>*

*</option>*

*</ItemTemplate>*

*</asp:Repeater>*

*</optgroup>*

*</ItemTemplate>*

*</asp:Repeater>*

1. To enable an ASP.Net web service to be called by a script (JS or JQuery) annotate web service class with [System.Web.Script.Services.ScriptService]

**AJAX Solr Architecture**

1. AJAX Solr is a **JavaScript library/framework** for creating **user interfaces to Apache Solr**, requiring an AJAX implementation for communication with Solr. To use AjaxSolr, A **Manager object** should be defined that extends AbstractManager object from library, overriding function executeRequest().Manager takes URL of Solr application as parameter. AJAX Solr by **default uses the select servlet** to talk to Solr, alternatively servlet parameter can be passed to manager along with URL as a JSON object
2. Manager has methods **'addWidget' and 'setStore'** using which it attaches with itself ParameterStore and Widgets. Manager's init() method initializes ParameterStore and widgets.
3. ParameterStore defines functions for getting, setting, and removing Solr parameters. Some parameters for Solr may be specified multiple times i.e. facet.field, fq, others specified only once ie (q and start). **The isMultiple() method** returns true if parameter may be specified multiple times. Functions get(), values(), add(), remove(), find(), addByValue(), removeByValue() behave differently depending on parameter frequency. string () returns the Solr parameters as a query string, **parseString()** does opposite.
4. Manager’s **doRequest() method** sends a request to Solr. It calls **each widget’s** **beforeRequest() method** in which widget can perform some action before request is sent to Solr. Then it invokes executeRequest() method to send the request. A search query is constructed within executeRequest() method using select servlet and Parameter Store in format below

*http://localhost:8983/solr/select?q=\*%3A\*&fq=topics%3Awheat&wt=json&json.wrf=?*

1. **Manager.jquery.js file** shipped with AJAX Solr implements the executeRequest() method using jQuery. On receiving response from Solr, **Manager caches the JSON response** in its response property, calls each widget’s afterRequest() method where widgets inspect response property and updates UI interface accordingly
2. **Solr parameters** are represented **as Parameter object** having properties val, local, remove.
3. AJAX Solr stores Solr parameters in a ParameterStore. Some parameters like **facet & facet.field** don’t change and are set when manager is initialized. Others like **q & fq** are changeable and can be modified by one or more widgets. A basic search query is built by **adding "q" parameter** to store, matching all documents

Manager.store.addByValue('q', '\*:\*');

Request is then sent to Solr through **Manager.doRequest() API** and search results are returned in response object as can be seen by checking **Manager.response in browser** developer tools; Solr uses **SearchHandler servlet** as its default request handler

1. **Widgets update UI** as per response received. All widgets inherit from AbstractWidgets. A widget takes an ID & **an optional target** i.e. a CSS selector for the HTML element that widget updates after each Solr request. AJAX Solr enables writing new widgets inheriting from available abstract widgets
2. Documents in the Solr response are displayed in results widget that is added to Manager.

*Manager.addWidget(new AjaxSolr.ResultWidget({*

*id: 'result',*

*target: '#docs' //widget will update elements matching #docs selector*

*}));*

Next define **afterReguest() method** of results widget to iterate through response object and append documents retrieved to its target property

*afterRequest: function () {*

*$(this.target).empty();*

*for (var i = 0, l = this.manager.response.response.docs.length; i < l; i++) {*

*var doc = this.manager.response.response.docs[i];*

*$(this.target).append(****this.template(doc));***

*}*

*}*

template() function creates an html formatted structure for each doc. Retrieved docs are now displayed in browser

1. For each document topics, organizations and exchanges are **retrieved as facetlinks(),** a method that iterates through topics, organizations and exchanges associated with document and adds to a collection. Each facetlink has a **click event handler** attached that adds or modifies parameters on ParameterStore and performs search request on Solr

*facetHandler: function (facet\_field, facet\_value) {*

*var self = this;*

*return function () {*

*self.manager.store.remove('fq');*

***self.manager.store.addByValue('fq', facet\_field + ':' + AjaxSolr.Parameter.escapeValue(facet\_value));***

***self.doRequest(0);***

*return false;*

*};*

},

1. **Faceting** is arrangement of **search results into categories** based on indexed terms. Response to faceted search provides numerical count & matching docs. Facets are enabled if **facet parameter** is set to true. Other parameters are ignored if facet is not set. **facet.query** allows specifying an arbitrary query. Solr determines unique terms for a field and returns a count for each of them. facet.query parameter can be provided multiple times. facet.field identifies a field to be treated as a facet. It iterates over each Term in the field and generates a facet count using that Term as the constraint. Can be specified multiple times
2. Adding a pager widget paginates search results

*Manager.addWidget(new AjaxSolr.PagerWidget({*

*id: 'pager',*

*target: '#pager',*

*prevLabel: '&lt;',*

*nextLabel: '&gt;',*

*innerWindow: 1,*

*renderHeader: function (perPage, offset, total) {*

*$('#pager-header').html($('<span></span>').text('displaying ' + Math.min(total, offset + 1) + ' to ' + Math.min(total, offset + perPage) + ' of ' + total));*

*}*

*}));*

1. A tag cloud widget is created extending AbstractFacetWidget of AJAX Solr library. For each facet field - 'topics', 'organisations', 'exchanges', add widget instances to manager. These fields can be customized based on Solr application requirements. This widget will take a facet field as a parameter

var fields = [ 'topics', 'organisations', 'exchanges' ];

for (var i = 0, l = fields.length; i < l; i++) {

Manager.addWidget(new AjaxSolr.TagcloudWidget({

id: fields[i],

target: '#' + fields[i],

field: fields[i]

}));

}

1. Within afterRequest method of this widget, count for each of facet field is obtained, and all facets within that filed added to an array A hyperlink is created on each facet, and a click handler added. Click handler implemented by AbstractFacetWidget, creates a search request on a facet and sends request to solr

*afterRequest: function () {*

*if (this.manager.response.facet\_counts.facet\_fields[this.field] === undefined) {*

*$(this.target).html('no items found in current selection');*

*return;*

*}*

*var maxCount = 0;*

*var objectedItems = [];*

*for (var facet in this.manager.response.facet\_counts.facet\_fields[this.field]) {*

*var count = parseInt(this.manager.response.facet\_counts.facet\_fields[this.field][facet]);*

*if (count > maxCount) {*

*maxCount = count;*

*}*

*objectedItems.push({ facet: facet, count: count });*

*}*

*objectedItems.sort(function (a, b) {*

*return a.facet < b.facet ? -1 : 1;*

*});*

*$(this.target).empty();*

*for (var i = 0, l = objectedItems.length; i < l; i++) {*

*var facet = objectedItems[i].facet;*

*$(this.target).append(*

*$('<a href="#" class="tagcloud\_item"></a>')*

*.text(facet)*

*.addClass('tagcloud\_size\_' + parseInt(objectedItems[i].count / maxCount \* 10))*

*.click(this.clickHandler(facet))*

*);*

*}*

*}*

1. Create a new widget extending AjaxSolr.AbstractWidget and add it to Manager. Within its afterRequest method collects all the fq parameter values from ParameterStore. For each parameter value an hyperlink is created, that gets removed when link is clicked and request sent to Solr. New links that get created as received form Solr response are displayed

**Apache Solr and Lucene**

1. Most concepts Solr uses are derived from Lucene. Lucene allows building custom components and integrating them with Solr. Projects like Solr & elastics search use Lucene. **Lucene jars** are added to class path of Java application from maven
2. Lucene can scale to large datasets. Features of Lucene are sufficient for large scale ecommerce applications. Solr is based on **NoSQL** and can handle large datasets
3. Lucene performs **indexing of docs** and enables search. It uses Inverted index, which is a lookup Data structure locating document based on word occurring in them. Google search engine does more than what Lucene or Solr does. Approach is same based on everything being indexed. In order to use Lucene as a web crawler, a third party framework should be used that scans all pages on web and gives to Lucene.
4. Contents acquired by an application have to be formed into a doc before sending it to Lucene. Before storing a doc **Lucene applies analysers** to doc for filtering or breaking into tokens. It uses Boolean and vector space model to categorize searched docs
5. Analyser is used as a strategy for index writing. It purges docs of useless contents i.e. space, stop words etc. An analyser consists of a **tokenizer and multiple filters**. A Tokenizer decides how analyser has to split stream of chars i.e. WhiteSpaceAnalyzer splits words using white space; StandardAnalyzer ignores numeric chars while generating tokens. A single analyser can be used at a time. However **Custom analyser** can be used to club features of multiple analysers in a single analyser.
6. Tokenizers include **pre tokenizer and post tokenizer** e.g. pre tokenizer cleans up html tags to extract content. Post tokenizer includes stemming, text normalization, synonym expansion etc. **Query types** are term query, wild card query, fuzzy query, range query, Boolean query etc.

**Angular JS Overview**

1. Angular JS is open source FE web app framework. It simplifies development & testing of apps by providing client side MVC/MVVM architectures. Angular framework first reads html page, which contain custom tags that it interprets as **directives**. Directives bind i/o parts of page to model
2. Angular 2 is typescript based and uses a hierarchy of components instead of scope. It include **ES6 (ECMA 2015) features** i.e. lambdas, iterators & reflection. Supports class based OOP, static typing & generics. ES6 syntax for Java6 includes classes & modules
3. **Typescript** is superset of Java script and adds optional static typing. It supports definition files like C++ headers that can contain type information of existing JavaScript libraries. Type Script headers are available for Node JS, JQuery, Mongo DB
4. A Typescript file is compiled into java script through **TS compiler**. If run in watch mode changes to TS scripts get compiled on fly and rendered in browser. Provides object oriented features through classes/interfaces/generics & others like variable declaration of specified type, array swapping, template string etc.
5. A class is marked as **Angular component** using decorators and provides **metadata** for usage of component at runtime. **Services** are used to access data & are shared across components. **Angular Quick Start project** on Git hub contains basic dependencies needed for an angular app. As it is based on typescript, it avoids need for directly working with JS, rather requires editing only TS files which get compiled to JS. **Ng directives** are still available for use in **html metadata** of component. Services are injected into component using **@Injectable decorator** on service class and passing it into component constructor

**Angular JS**

1. Directive **ng-app** sets boundary of angular app. It can be moved to particular section of html also such as a div element **ng-model** maps input element value to a property in current scope instance, **ng-show** & **ng-hide** display or hide html element based on expression.
2. A **Module** in angular is container for different parts of application such as. Controllers, services, filter etc. Also specifies how angular app should be bootstrapped. Module is defined in a JS file and then included in HTML.

Var testAngularApp = angular

.module(“TestApp”, [])

.controller(“Controller1”, function($scope, $http, $log) {

$http({method: “GET’,

url : ‘webservice method’})

.then (function (response){

$scope.propertyName = response.data;

})

});

Optional array parameter to module specifies dependencies. Empty array means no dependency

1. To **bootstrap angular app** with this module, ng-app directive should be set to module name in HTML file

*<body ngApp = "TestApp">*

1. A **Controller** function that builds a model for view to display. **$scope** in controller function is passed by angular framework automatically. A Model if attached to $scope object within controller and then becomes available to view. Controller can be **registered with module** either separately or through method chaining in a single statement. A Controller can be attached to html element using **ng-controller** directive.

myApp.Controller("controller name", controllerFunction($scope){

});

1. **ng-src attribute** used with images ensures request is sent only after angular has evaluated bind expression. Html element attributes can also be dynamically assigned using binding expressions.

<div>

<img **src={{country.flag}}** alt = {{country.name + ' Flag'}} style="height:100px; width:200px" />

</div>

When DOM is parsed request is sent to server to load image. At that time angular binding expressions are not parsed, Hence 404 error comes initially. .A second request is made after binding expression is evaluated and then image gets displayed

1. **ng-model directive** when assigned as property of HTML element ensures when view is changed, model gets updated.

*<input type="text" ng-model="message" /> //message is property of scope*

A model gets created while something is entered in input box, updating model data from view. ng-model can be used with any of element => textarea, input, select

1. **ng-repeat directive** is like for each. $index property with ng-repeat returns index of current employee. ng-repeat can be used in **nested expressions** through a combination of html unordered and ordered list elements <ul> & <li>

<tr ng-repeat = "emp in employees" //employees is a model

<td>**{{emp.firstname}}** - Index = {{$index}} - Parent Index = {{$parent.$index}} </td>

<td>{{emp.lastname}}</td>

</tr>

It displays each employee’s first name and last name in separate rows on a page.

1. A Filter is applied using pipe within binding expression braces **{{expression | filtername: parameter}**}. Common formatting filters include lowercase, uppercase, number etc. A **Custom filter** is created in AJS **using .filter method on module**. It takes in an anonymous function which returns another function. It applied with pipe char as with regular angular filters
2. To make table header clickable, assign **ng-click attribute** to <th> element. Set its value as sort column name. Use ng-class attribute to get an arrow on sorted column.

*<th* ***ng-click='sortData('name')*** *>Name <div* ***ng-class="getSortClass('name')"*** *></div></th>*

This applies **sortData() function** when click event is sent on “Name” column. An arrow class is applied depending on value returned by getSortClass method

1. A **service in angular** is an object that provides some sort of service that can be used within angular, It has props and methods. Angular has **built in services** like $http, $resource etc. factory method is used to create custom service and register with angular
2. **$http service** in angular is used to communicate with remote http server. $http service has a single input parameter i.e. configuration. It **returns a promise object** i.e. data returned may not be available immediately. Returned value is obtained in **then method**, taking **success call back** function as argument. An **error call back function** can also be provided along with success callback function that takes **reason object** as parameter. reason object has **data property** which is assigned to $scope inside callback function
3. **$log service** provides logging support, used to log response to console. $log service is also injected in function like $http service
4. **$anchorscroll service** is used to jump to specified element on page. **$location service** hash method appends hash fragments to the URL. $anchorscroll() method reads the hash fragment in the URL and jumps to that element on the page. **yOffset property** specifies the vertical scroll-offset

*$scope.scrollTo = function (scrollLocation) {*

*$location.hash(scrollLocation);*

*$anchorScroll.yOffset = 20;*

*$anchorScroll();*

*}*

1. Angular JS framework **ng-boilerplate** provides a project structure for starting angular JS project. It includes a **grunt file** that provides configuration of grunt build system (comparable to Ant) & **Bower** which is a project package manager (comparable to maven) used mostly for managing f**ront end JS frameworks.** Grunt is task manager and these together are configured in ng-boilerplate project, makes **creating angular projects much easier.**
2. Grunt/bower both run on **node js**, mainly a platform for **developing server side applications**. Installing node js installs **node package manager (npm)** which is used to install both grunt and bower

*npm install -g grunt-cli //cmd prompt should be running in admin mode*

*npm install -g bower*

ng-boilerplate should be put in **webapp directory** of project

*git clone https://github.com/ngbp/ngbp.git app*

1. Grunt file in project **references many tasks**. These are user tasks that other users have built each having a website defining configuration options for these tasks. To use these tasks, **a taskConfig JSON object** is defined containing different configurations to these tasks
2. **package.JSON** file helps download different grunt dependencies using npm. **Npm install** command downloads all dependencies to **node\_modules package**
3. Bower is used to manage JS and AJS dependencies. **.bowerrc file** contains a directory to download bower dependencies, & a **configuration file bower.json**. It contains reference to many client side dependencies. **bower install** command install all these dependencies, which also get **automatically added to project html file** as compiled java script.
4. **grunt watch** command Watches changes to JS source files and rebuilds application every time there is a change

**MEAN Stack with Angular Frontend**

1. Install angular CLI and Angular Source directory.

*npm uninstall -g @angular/cli*

*npm cache verify*

*npm install -g @angular/cli@latest*

*npm install --save-dev @angular/cli@latest*

*ng new angular-src*

*ng serve from angular-src folder*

A quick start angular project gets created with a module and a component and launched at <http://localhost:4200>

1. Edit **angular-cli.json** file to point output directory to **public** folder from default dist
2. **app.module.ts file** is meeting place for all components. All components & services will be imported in here along with declarations. Services will be imported in **provider array**. Modules will get imported in import section
3. Component decorator @Component allows adding metadata
4. New components and services are generated through angular CLI. It auto imports components into app module

*ng g component navbar*

*ng g service authservice*

1. **HTML code** for component is put inside html file created for component
2. Angular application is a reactive system, based on change detection**.**
3. **Observables** in Angular help manage asynchronous data, such as data coming from a backend service. To use observables, angular uses a third-party library called **Reactive Extensions (RxJS).** ReactiveX provides asynchronous programming support with observable streams
4. Map operator of RxJS transforms items emitted by Observable by applying a function to each item

*return this.http.post('http://localhost:3000/user/register', user,{headers: headers}).map(res => res.json());*

1. Observable should be **subscribed to** in calling function

*this.authService.registerUser(user).****subscribe****(data => {*

*if(data.success){*

*//Display a flash message*

*//navigate to login route*

*} else {*

*//Show error message*

*}*

*});*

1. **Ngmodel** directive will be used to link input elements on HTML form to fields in angular component

*<div class="form-group">*

*<label>Username</label>*

*<input type="text" class="form-control"* ***[(ngModel)]="username" name="username"****>*

*</div>*

Links username on form control to username field in Login Component

1. **Angular services** should be injected as dependency in constructor of component, passing them as arguments

constructor(

private authService:AuthService,

private router:Router,

private flashMessage:FlashMessagesService

) { }

1. **Router module** will be used to navigate based on form action. FlashMessagesService will be used to display status on form
2. Use “ng build” command to install application angular app in public folder
3. To **redirect** all unspecified **routes** to index page

*app.use('\*', (req, res) => {*

*res.sendfile(path.join(\_\_dirname, 'public/index.html')));*

*})*

1. To deploy mean app using **Heroku**, modify port on node server

*Const port = process.env.PORT || 8080*

1. Backend service URL calls from angular services should be modified, then application built using ng build

*return this.http.post('/user/register’ …)*

*ng build*

1. Heroku account creation and installation of Heroku CLI for Windows/Linus etc. Install heroku after CLI installation
2. Git installation (if not there) and git folder initialization using git init from within project folder
3. Prepare heroku to receive project source code using **heroku create** fromproject folder. A url link for application is created
4. Add all files to git using “**git .”** from project folder and commit and push to heroku

*Git config –global user.email “email address”*

*Git config –global user.name “user name”*

*Git commit –am “first commit”*

*Git push heroku master*

1. URL link should open project front end now however functionality won’t work
2. To set up mongo DB heroku add on for mongo is used i.e. mLab mongoDB
3. Set up an mLab account and set up remote database. Create a user. Set up received database connection string in db configuration file on backend server
4. Push changes to heroku

*Git add .*

*Git commit –am “linked DB”*

*Git push heroku master*

1. Add a domain name from heroku UR settings

**Node JS**

1. Node JS enables creating server side applications. Node JS code is run on server unlike JS that runs in browser.
2. Node JS modules are JS files. Functions are exported from modules to enable use in other files.

*module.exports.avatar = funcName;*

*var movies = require(./movies) //imports movies module, refers to movies.js file*

*module.exports = mongoose.model('user', userSchema);*

*var User =* ***require****('../model/user'); //user schema is imported in another module as ‘User’*

1. Core modules are built in with node JS e.g. FS module for working with files, http module for server code
2. **Express** is a web framework for node and sets up entire structure for project. It allows using **built in middleware or writing custom ones** to respond to http requests. It s **routes** to go to diff pages in web application. Start-up request is sent to **Index.js** when home page is accessed

*router.get ('/', function(req, res, next){*

*res.render(index. {title: 'Express'}) //passed to tile variable*

*})*

1. Node JS has a very large ecosystem of open-source libraries. Dependencies appear in package.json file and a new folder node\_module gets created

*npm install cookie-parser --save*

***npm install express –save***

*npm install cookie-parser --save*

*npm install body-parser ejs ejs-mate express-session –save*

*npm -install express-session*

1. **express-session** provides a simple server side session middleware for express. Default server side session storage is not good for production environment & also does not scale beyond a single process. There are **alternate session stores** which use a backend as session store. **connect-mongo** is one of them using mongo DB based store, It allows storage of session data in mongo dB
2. **Mongoose** is required to be installed to save session data into Mongo db. mongoose allows connection to several databases at a time or to a single database

*npm install connect-mongo --save*

*npm install mongoose --save*

*var mongoose = require(‘mongoose’); //include mongoose in main JS file*

*mongoose.connect('mongodb://localhost/dbname')*

1. **Ejs-mate** provides express layout & template functions for ejs template engine. It helps write **reusable code for web pages** like header footer etc. It makes changing data in header or footer easier without repetitions

**AWS Lambda**

1. Lambda is **AWS compute service** where code can be uploaded which is then executed using AWS infrastructure. Each piece of code is a function on AWS Lambda platform . Uploaded functions should be stateless, but stateful info can be stored by connecting to dynamo DB/ S3 services. Lambdas are triggered from events coming from other AWS services i.e. when new file gets uploaded to S3 bucket, or through http calls. Lambda exposes Functions as a service
2. Amazon **API Gateway** is a collection of resources and methods. When a method is invoked through HTTPS endpoint API Gateway invokes the Lambda function. Lambda function is connected to api gateway through **AWS lambda console (using trigger)**. Function can then be called as web service through a URL in browser
3. Lambda function can also be connected using **API Gateway console**. Steps are Creating a New API, Edit Integration Request & Create a body template which is a **JSON doc** passed on to function. Lambda function makes use of parameters, using values from **event parameter** which is passed in by default. Parameters provide in URL are populated inside event parameter
4. Apex program manages server less arch via AWS lambda. It is an attempt to fix usability of Lambda & Api gateway and convenient as Managing clusters through ECS agents is still lot of overhead
5. To work with apex, a New user for Apex is created in aws accounts through IAM. Access policies for Lambda & IAM are attached to this user. AWS CLI should be installed and configured ie **aws configure**. Configuration involves **setting access key & secret key of apex user** created through IAM. This ensures limited rights as per attached policies to apex user i.e. Create Security Group rights may not be available for apex user as for admin user.
6. **Apex init** configures new project on command line. It creates **project.json file** that contain info about functions in that project ie. Memory usage, role under which these functions execute etc. These values can be overridden for each function. **Apex deploy** creates a function and it appears under lambda functions on AWS Lambda console. Apex automatically **updates version and alias** of function if changed.
7. **Cloud watch** section on AWS for can be used for scheduling functions to run at a specific time on AWS. It is good for auto scheduled emails
8. Using Lambda to **write data do dynamodb:** Lambda functions accessing DynamoDB use document client which simplifies working with items in Amazon DynamoDB. It simplifies JS development. JavaScript objects passed in as parameters are marshalled into AttributeValue required by DynamoDB. Put method creates or replaces an item in dynamoDB
9. By default lambda doesn’t have permission to access dynamo DB, hence lambda function cannot be directly invoked through AWS Lambda dashboard. To access other ASW services specific rights should be given to **lambda function role.** This can be done through **IAM, by attaching a new policy to that role**. Policies allow access limited to specified tables in dynamoDB through table ARNs. After associating new policy Lambda function is invokable from Lambda Console
10. **Reading from dynamo DB**: DocClient.scan and DocClient.query methods are used to fetch data. Scan method can be a performance bottleneck for large data as it looks through entire table. Query run using docClient.get and can only be on primary key
11. For **a FE application to fetch and update data in Dynamo DB**, first an API around Lambda functions is required through which FE can access BE system. **GET/POST methods** are then created under resource entries. FE will post data through API system and API will send it through Lambda. Body mapping template of type Application/Json will be used for parameters. CORS should be enabled so that any one from any domain can access API. Deploying API will give an **Invoke URL** which can then be accessed through AJAX requests in FE HTML file.