According to B.C.A., M.C.A., B.Sc. (IT, CS), M.Sc. (IT, ICT, CS), B.E. (IT), PGDCA and Diploma Courses of Engineering Programs of Various Universities

Fundamentals of React & Angular JS with Mongo DB

Authors:

Dr. Ishaan Tamhankar [M.C.A., Ph.D.]
VIMAL TORMAL PODDAR BCA COLLEGE, PANDESARA, SURAT

Ms. Ritu Bhatiya [M.C.A.]
VIMAL TORMAL PODDAR BCA COLLEGE, PANDESARA, SURAT

Ms. Eeva Kapopara [M.C.A.]
ATMANAND SARASWATI SCIENCE COLLEGE, SURAT

[First Edition: 2022]

INDEX

Unit- 1
Concept of NoSQL: MongoDB

1.1 Concept of NOSQL

NoSQL databases are non-tabular databases and store data differently than relational tables. NoSQL databases come in a variety of types based on their data model. The main types are document, key-value, wide-column, and graph. They provide flexible schemas and scale easily with large amounts of data and high user loads.

When people use the term "NoSQL" stands for "non-SQL" while others say it stands for "not only SQL." Either way, most agree that NoSQL databases are databases that store data in a format other than relational tables.

NoSQL database features

Each NoSQL database has its own unique features. At a high level, many NoSQL databases have the following features:

- Flexible schemas
- Horizontal scaling
- Fast queries due to the data model
- Ease of use for developers

Check out What are the Benefits of NoSQL Databases? to learn more about each of the features listed above.

Types of NoSQL databases

Over time, four major types of NoSQL databases emerged: document databases, key-value databases, wide-column stores, and graph databases.

- Document databases store data in documents similar to JSON (JavaScript Object Notation) objects. Each document contains pairs of fields and values. The values can typically be a variety of types including things like strings, numbers, booleans, arrays, or objects.
- Key-value databases are a simpler type of database where each item contains keys and values.
- Wide-column stores store data in tables, rows, and dynamic columns.
- Graph databases store data in nodes and edges. Nodes typically store information about people, places, and things, while edges store information about the relationships between the nodes.

MongoDB Advantages and Disadvantages

Like other NoSQL databases, MongoDB doesn't require predefined schemas. It stores any type of data. This gives users the flexibility to create any number of fields in a document, making it easier to scale MongoDB databases compared to relational databases.

One of the advantages of using documents is that these objects map to native data types in a number of programming languages. Also, having embedded documents reduces the need for database joins, which can reduce costs.

A core function of MongoDB is its horizontal scalability, which makes it a useful database for companies running big data applications. In addition, sharding allows the database to distribute data across a cluster of machines. Newer versions of MongoDB also support the creation of zones of data based on a shard key.

MongoDB supports a number of storage engines and provides pluggable storage engine APIs that allow third parties to develop their own storage engines for MongoDB.

The DBMS also has built-in aggregation capabilities, which allow users to run MapReduce code directly on the database, rather than running MapReduce on Hadoop. MongoDB also includes its own file system called GridFS, akin to the Hadoop Distributed File System (HDFS). The use of the file system is primarily for storing files larger than BSON's size limit of 16 MB per document. These similarities allow MongoDB to be used instead of Hadoop, though the database software does integrate with Hadoop, Spark and other data processing frameworks.

Though there are some valuable benefits to MongoDB, there are some downsides to it as well. With its automatic failover strategy, a user sets up just one master node in a MongoDB cluster. If the master fails, another node will automatically convert to the new master. This switch promises continuity, but it isn't instantaneous -- it can take up to a minute. By comparison, the Cassandra NoSQL database supports multiple master nodes so that if one master goes down, another is standing by for a highly available database infrastructure.

MongoDB's single master node also limits how fast data can be written to the database. Data writes must be recorded on the master, and writing new information to the database is limited by the capacity of that master node.

Another potential issue is that MongoDB doesn't provide full referential integrity through the use of foreign-key constraints, which could affect data consistency. In addition, user authentication isn't enabled by default in MongoDB databases, a nod to the technology's popularity with developers. However, malicious hackers have targeted large numbers of unsecured MongoDB systems in ransom attacks, which led to the addition of a default setting that blocks networked connections to databases if they haven't been configured by a database administrator.

1.1.1 MongoDB Datatypes

MongoDB supports many datatypes. Some of them are –

- **String** – This is the most commonly used datatype to store the data. String in MongoDB must be UTF-8 valid.
- **Integer** – This type is used to store a numerical value. Integer can be 32 bit or 64 bit depending upon your server.
- **Boolean** – This type is used to store a boolean (true/ false) value.
- **Double** – This type is used to store floating point values.
- **Min/ Max keys** – This type is used to compare a value against the lowest and highest BSON elements.
- **Arrays** – This type is used to store arrays or list or multiple values into one key.
- **Timestamp** – ctimestamp. This can be handy for recording when a document has been modified or added.
- **Object** – This datatype is used for embedded documents.
- **Null** – This type is used to store a Null value.
- **Symbol** – This datatype is used identically to a string; however, it's generally reserved for languages that use a specific symbol type.
- **Date** – This datatype is used to store the current date or time in UNIX time format. You can specify your own date time by creating object of Date and passing day, month, year into it.
- **Object ID** – This datatype is used to store the document's ID.
- **Binary data** – This datatype is used to store binary data.
- **Code** – This datatype is used to store JavaScript code into the document.
- **Regular expression** – This datatype is used to store regular expression.

1.1.2 Database Creation and Dropping Database

The use Command

MongoDB **use DATABASE_NAME** is used to create database. The command will create a new database if it doesn't exist, otherwise it will return the existing database.

Syntax

Basic syntax of **use DATABASE** statement is as follows –

```
use DATABASE_NAME
```

Example

If you want to use a database with name **<mydb>**, then **use DATABASE** statement would be as follows –

```
>use mydb
switched to db mydb
```

To check your currently selected database, use the command **db**

```
>db
mydb
```

If you want to check your databases list, use the command **show dbs**.

```
>show dbs
local    0.78125GB
test     0.23012GB
```

Your created database (mydb) is not present in list. To display database, you need to insert at least one document into it.

```
>db.emp.insert({"name":"Hello India"})
>show dbs
local    0.78125GB
mydb     0.23012GB
test     0.23012GB
```

In MongoDB default database is test. If you didn't create any database, then collections will be stored in test database.

The dropDatabase() Method

MongoDB **db.dropDatabase()** command is used to drop a existing database.

Syntax

Basic syntax of **dropDatabase()** command is as follows –

db.dropDatabase()

This will delete the selected database. If you have not selected any database, then it will delete default 'test' database.

Example

First, check the list of available databases by using the command, **show dbs.**

```
>show dbs
local    0.78125GB
mydb     0.23012GB
test     0.23012GB
>
```

If you want to delete new database **<mydb>**, then **dropDatabase()** command would be as follows –

```
>use mydb
switched to db mydb
>db.dropDatabase()
>{ "dropped" : "mydb", "ok" : 1 }
>
```

Now check list of databases.

```
>show dbs
local    0.78125GB
test     0.23012GB
>
```

1.2 Create and Drop Collections

The createCollection() Method:

MongoDB **db.createCollection(name, options)** is used to create collection.

Syntax

Basic syntax of **createCollection()** command is as follows –

db.createCollection(name, options)

In the command, **name** is name of collection to be created. **Options** is a document and is used to specify configuration of collection.

Parameter	Type	Description
Name	String	Name of the collection to be created
Options	Document	(Optional) Specify options about memory size and indexing

Options parameter is optional, so you need to specify only the name of the collection. Following is the list of options you can use –

Field	Type	Description
Capped	Boolean	(Optional) If true, enables a capped collection. Capped collection is a fixed size collection that automatically overwrites its oldest entries when it reaches its maximum size. **If you specify true, you need to specify size parameter also.**
autoIndexId	Boolean	(Optional) If true, automatically create index on _id field.s Default value is false.
Size	number	(Optional) Specifies a maximum size in bytes for a capped collection. **If capped is true, then you need to specify this field also.**
Max	number	(Optional) Specifies the maximum number of documents allowed in the capped collection.

While inserting the document, MongoDB first checks size field of capped collection, then it checks max field.

Examples

Basic syntax of **createCollection()** method without options is as follows –

```
>use test
switched to db test
>db.createCollection("mycollection")
{ "ok" : 1 }
```

>

You can check the created collection by using the command **show collections**.

```
>show collections
mycollection
system.indexes
```

The following example shows the syntax of **createCollection()** method with few important options –

```
> db.createCollection("mycol", { capped : true, autoIndexID : true, size : 6142800, max : 10000 } ){
"ok" : 0,
"errmsg" : "BSON field 'create.autoIndexID' is an unknown field.",
"code" : 40415,
"codeName" : "Location40415"
}
```

In MongoDB, you don't need to create collection. MongoDB creates collection automatically, when you insert some document.

```
>db.emp.insert({"name" : "Ishaan"}),
WriteResult({ "nInserted" : 1 })
>show collections
mycol
mycollection
system.indexes
emp
```

The drop() Method

MongoDB's **db.collection.drop()** is used to drop a collection from the database.

Syntax

Basic syntax of **drop()** command is as follows –

```
db.COLLECTION_NAME.drop()
```

Example

First, check the available collections into your database **mydb**.

```
>use mydb
switched to db mydb
>show collections
mycol
```

```
mycollection
system.indexes
emp
>
```

Now drop the collection with the name **mycollection**.

```
>db.mycollection.drop()
true
>
```

Again check the list of collections into database.

```
>show collections
mycol
system.indexes
emp
```

drop() method will return true, if the selected collection is dropped successfully, otherwise it will return false.

1.3 CRUD Operation

The basic methods of interacting with a MongoDB server are called CRUD operations. CRUD stands for Create, Read, Update, and Delete. These CRUD methods are the primary ways you will manage the data in your databases.

CRUD operations describe the conventions of a user-interface that let users view, search, and modify parts of the database.

MongoDB documents are modified by connecting to a server, querying the proper documents, and then changing the setting properties before sending the data back to the database to be updated. CRUD is data-oriented, and it's standardized according to HTTP action verbs.

When it comes to the individual CRUD operations:

- The Create operation is used to insert new documents in the MongoDB database.
- The Read operation is used to query a document in the database.
- The Update operation is used to modify existing documents in the database.
- The Delete operation is used to remove documents in the database.

How to Perform CRUD Operation:

Create Operations

For MongoDB CRUD, if the specified collection doesn't exist, the create operation will create the collection when it's executed. Create operations in MongoDB target a single collection, not multiple collections. Insert operations in MongoDB are atomic on a single document level.

MongoDB provides two different create operations that you can use to insert documents into a collection:

- db.collection.insertOne()
- db.collection.insertMany()

insertOne()

As the namesake, insertOne() allows you to insert one document into the collection. For this example, we're going to work with a collection called RecordsDB. We can insert a single entry into our collection by calling the insertOne() method on RecordsDB. We then provide the information we want to insert in the form of key-value pairs, establishing the schema.

E.g

```
db.RecordsDB.insertOne({
   name: "Brado",
   age: "6 years",
   species: "Dog",
   ownerAddress: "Vesu",
   chipped: true
})
```

If the create operation is successful, a new document is created. The function will return an object where "acknowledged" is "true" and "insertID" is the newly created "ObjectId."

```
> db.RecordsDB.insertOne({
... name: "Brado",
... age: "6 years",
```

```
... species: "Dog",
... ownerAddress: "Vesu",
... chipped: true
... })
{
    "acknowledged" : true,
    "insertedId" : ObjectId("5fd989674e6b9ceb8665c57d")
}
```

insertMany()ss

It's possible to insert multiple items at one time by calling the insertMany() method on the desired collection. In this case, we pass multiple items into our chosen collection (RecordsDB) and separate them by commas. Within the parentheses, we use brackets to indicate that we are passing in a list of multiple entries. This is commonly referred to as a nested method.

```
db.RecordsDB.insertMany([{
  name: "Brad",
  age: "6 years",
  species: "Dog",
  ownerAddress: "Vesu",
  chipped: true},
   {name: "Lucy",
   age: "4 years",
   species: "Cat",
   ownerAddress: "Citylight",
   chipped: true}])
```

Read Operations

The read operations allow you to supply special query filters and criteria that let you specify which documents you want. The MongoDB documentation contains more information on the available query filters. Query modifiers may also be used to change how many results are returned.

MongoDB has two methods of reading documents from a collection:

- db.collection.find()
- db.collection.findOne()

find()

In order to get all the documents from a collection, we can simply use the find() method on our chosen collection. Executing just the find() method with no arguments will return all records currently in the collection.

e.g

```
db.RecordsDB.find
```

```
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Brad", "age" : "3 years",
"species" : "Dog", "ownerAddress" : "Vesu", "chipped" : true }
```

Here we can see that every record has an assigned "ObjectId" mapped to the "_id" key.

If you want to get more specific with a read operation and find a desired subsection of the records, you can use the previously mentioned filtering criteria to choose what results should be returned. One of the most common ways of filtering the results is to search by value.

```
db.RecordsDB.find({"species":"Cat"})
```

findOne()

In order to get one document that satisfies the search criteria, we can simply use the findOne() method on our chosen collection. If multiple documents satisfy the query, this method returns the first document according to the natural order which reflects the order of documents on the disk. If no documents satisfy the search criteria, the function returns null. The function takes the following form of syntax.

db.{collection}.findOne({query}, {projection})

And, we run the following line of code:

db.RecordsDB.find({"age":"8 years"})

Update Operations

Like create operations, update operations operate on a single collection, and they are atomic at a single document level. An update operation takes filters and criteria to select the documents you want to update.

You should be careful when updating documents, as updates are permanent and can't be rolled back. This applies to delete operations as well.

For MongoDB CRUD, there are three different methods of updating documents:

- db.collection.updateOne()
- db.collection.updateMany()
- db.collection.replaceOne()

updateOne()

We can update a currently existing record and change a single document with an update operation. To do this, we use the updateOne() method on a chosen collection, which here is "RecordsDB." To update a document, we provide the method with two arguments: an update filter and an update action.

The update filter defines which items we want to update, and the update action defines how to update those items. We first pass in the update filter. Then, we use the "$set" key and provide the fields we want to update as a value. This method will update the first record that matches the provided filter.

db.RecordsDB.updateOne({name: "Rahul"}, {$set:{ownerAddress: "451 Sai Shrushti,Vesu"}})

updateMany()

updateMany() allows us to update multiple items by passing in a list of items, just as we did when inserting multiple items. This update operation uses the same syntax for updating a single document.

db.RecordsDB.updateMany({species:"Dog"}, {$set: {age: "5"}})

replaceOne()

The replaceOne() method is used to replace a single document in the specified collection. replaceOne() replaces the entire document, meaning fields in the old document not contained in the new will be lost.

```
db.RecordsDB.replaceOne({name: "Rahul"}, {name: "Chetan"})
```

Delete Operations

Delete operations operate on a single collection, like update and create operations. Delete operations are also atomic for a single document. You can provide delete operations with filters and criteria in order to specify which documents you would like to delete from a collection. The filter options rely on the same syntax that read operations utilize.

MongoDB has two different methods of deleting records from a collection:

- db.collection.deleteOne()
- db.collection.deleteMany()

deleteOne()

deleteOne() is used to remove a document from a specified collection on the MongoDB server. A filter criteria is used to specify the item to delete. It deletes the first record that matches the provided filter.

```
db.RecordsDB.deleteOne({name:"Aman"})
```

deleteMany()

deleteMany() is a method used to delete multiple documents from a desired collection with a single delete operation. A list is passed into the method and the individual items are defined with filter criteria as in deleteOne().

```
db.RecordsDB.deleteMany({species:"Dog"})
```

1.4 Operators

The sort() Method

To sort documents in MongoDB, you need to use **sort()** method. The method accepts a document containing a list of fields along with their sorting order. To specify sorting order 1 and -1 are used. 1 is used for ascending order while -1 is used for descending order.

Syntax

The basic syntax of **sort()** method is as follows –

```
>db. COLLECTION_NAME.find().sort({KEY:1})
```

Please note, if you don't specify the sorting preference, then sort() method will display the documents in ascending order.

In MongoDB, projection means selecting only the necessary data rather than selecting whole of the data of a document. If a document has 5 fields and you need to show only 3, then select only 3 fields from them.

The find() Method

MongoDB's **find()** method, explained in MongoDB Query Document accepts second optional parameter that is list of fields that you want to retrieve. In MongoDB, when you execute **find()** method, then it displays all fields of a document. To limit this, you need to set a list of fields with value 1 or 0. 1 is used to show the field while 0 is used to hide the fields.

Syntax

The basic syntax of **find()** method with projection is as follows –

```
>db.COLLECTION_NAME.find({},{KEY:1})
```

The Limit() Method

To limit the records in MongoDB, you need to use **limit()** method. The method accepts one number type argument, which is the number of documents that you want to be displayed.

Syntax

The basic syntax of **limit()** method is as follows –

```
>db.COLLECTION_NAME.find().limit(NUMBER)
```

Example

Consider the collection myycol has the following data.

```
{_id : ObjectId("507f191e810c19729de860e1"), title: "MongoDB Overview"},
{_id : ObjectId("507f191e810c19729de860e2"), title: "NoSQL Overview"},
```

{_id : ObjectId("507f191e810c19729de860e3"), title: "MYDB Overview"}

Following example will display only two documents while querying the document.

```
>db.mycol.find({},{"title":1,_id:0}).limit(2)
{"title":"MongoDB Overview"}
{"title":"NoSQL Overview"}
>
```

If you don't specify the number argument in **limit()** method then it will display all documents from the collection.

MongoDB Skip() Method

Apart from limit() method, there is one more method **skip()** which also accepts number type argument and is used to skip the number of documents.

Syntax

The basic syntax of **skip()** method is as follows –

```
>db.COLLECTION_NAME.find().limit(NUMBER).skip(NUMBER)
```

Example

Following example will display only the second document.

```
>db.mycol.find({},{"title":1,_id:0}).limit(1).skip(1)
{"title":"NoSQL Overview"}
>
```

Please note, the default value in **skip()** method is 0.

The sort() Method

To sort documents in MongoDB, you need to use **sort()** method. The method accepts a document containing a list of fields along with their sorting order. To specify sorting order 1 and -1 are used. 1 is used for ascending order while -1 is used for descending order.

Syntax

The basic syntax of **sort()** method is as follows –

```
>db.COLLECTION_NAME.find().sort({KEY:1})
```

Example

Consider the collection myycol has the following data.

{_id : ObjectId("507f191e810c19729de860e1"), title: "MongoDB Overview"}

{_id : ObjectId("507f191e810c19729de860e2"), title: "NoSQL Overview"}
{_id : ObjectId("507f191e810c19729de860e3"), title: "MYDB Overview"}

Following example will display the documents sorted by title in the descending order.

```
>db.mycol.find({},{"title":1,_id:0}).sort({"title":-1})
{"title":"MYDB Overview"}
{"title":"NoSQL Overview"}
{"title":"MongoDB Overview"}
>
```

Please note, if you don't specify the sorting preference, then **sort()** method will display the documents in ascending order.

Aggregations operations process data records and return computed results. Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result. In SQL count(*) and with group by is an equivalent of MongoDB aggregation.

The aggregate() Method

For the aggregation in MongoDB, you should use **aggregate()** method.

Syntax

Basic syntax of **aggregate()** method is as follows –

```
>db.COLLECTION_NAME.aggregate(AGGREGATE_OPERATION)
```

Example

In the collection you have the following data –

```
{
  _id: ObjectId(7df78ad8902c)
  title: 'MongoDB Overview',
  description: 'MongoDB is no sql database',
  by_user: 'ishaan',
  url: 'http://www.jump2learn.com',
  tags: ['mongodb', 'database', 'NoSQL'],
  likes: 100
},
{
  _id: ObjectId(7df78ad8902d)
  title: 'NoSQL Overview',
```

```
 description: 'No sql database is very fast',
 by_user: 'chetan',
 url: 'http://www.jump2learn.com',
 tags: ['mongodb', 'database', 'NoSQL'],
 likes: 10
},
{
  _id: ObjectId(7df78ad8902e)
 title: 'Neo4j Overview',
 description: 'Neo4j is no sql database',
 by_user: 'Neo4j',
 url: 'http://www.neo4j.com',
 tags: ['neo4j', 'database', 'NoSQL'],
 likes: 750
},
```

Now from the above collection, if you want to display a list stating how many tutorials are written by each user, then you will use the following **aggregate()** method –

```
> db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$sum : 1}}}])
{ "_id" : "tutorials point", "num_tutorial" : 2 }
{ "_id" : "Neo4j", "num_tutorial" : 1 }
>
```

Sql equivalent query for the above use case will be **select by_user, count(*) from** mycol **group by by_user**.

In the above example, we have grouped documents by field **by_user** and on each occurrence of by user previous value of sum is incremented. Following is a list of available aggregation expressions.

Expression	Description	Example
$sum	Sums up the defined value from all documents in the collection.	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$sum : "$likes"}}}])

$avg	Calculates the average of all given values from all documents in the collection.	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$avg : "$likes"}}}])
$min	Gets the minimum of the corresponding values from all documents in the collection.	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$min : "$likes"}}}])
$max	Gets the maximum of the corresponding values from all documents in the collection.	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$max : "$likes"}}}])
$push	Inserts the value to an array in the resulting document.	db.mycol.aggregate([{$group : {_id : "$by_user", url : {$push: "$url"}}}])
$addToSet	Inserts the value to an array in the resulting document but does not create duplicates.	db.mycol.aggregate([{$group : {_id : "$by_user", url : {$addToSet : "$url"}}}])
$first	Gets the first document from the source documents according to the grouping. Typically this makes only sense together with some previously applied "$sort"-stage.	db.mycol.aggregate([{$group : {_id : "$by_user", first_url : {$first : "$url"}}}])
$last	Gets the last document from the source documents according to the grouping. Typically this makes only sense together with some previously applied "$sort"-stage.	db.mycol.aggregate([{$group : {_id : "$by_user", last_url : {$last : "$url"}}}])

Pipeline Concept

In UNIX command, shell pipeline means the possibility to execute an operation on some input and use the output as the input for the next command and so on. MongoDB also supports same concept in aggregation framework. There is a set of possible stages and each

of those is taken as a set of documents as an input and produces a resulting set of documents (or the final resulting JSON document at the end of the pipeline). This can then in turn be used for the next stage and so on.

Following are the possible stages in aggregation framework -

- **$project** - Used to select some specific fields from a collection.
- **$match** - This is a filtering operation and thus this can reduce the amount of documents that are given as input to the next stage.
- **$group** - This does the actual aggregation as discussed above.
- **$sort** - Sorts the documents.
- **$skip** - With this, it is possible to skip forward in the list of documents for a given amount of documents.
- **$limit** - This limits the amount of documents to look at, by the given number starting from the current positions.
- **$unwind** - This is used to unwind document that are using arrays. When using an array, the data is kind of pre-joined and this operation will be undone with this to have individual documents again. Thus with this stage we will increase the amount of documents for the next stage.

Important Questions:

1. Explain MongoDB Datatypes.
2. Explain CURD Operation.
3. Explain Advantages of MongoDB.
4. How to Add, Update, Delete operation Perform in MongoDB.
5. Explain Operators in MongoDb.

Unit- 2
Fundamentals of React.Js

2.1 Overview of React

2.1.1. Concepts of React

ReactJS is one of the most popular JavaScript front-end libraries which has a strong foundation and a large community.

It is an open-source, component-based front end library which is responsible only for the view layer of the application.

Although React is a library rather than a language, it is widely used in web development. The library first appeared in May 2013 and is now one of the most commonly used frontend libraries for web development.

ReactJS is a **declarative, efficient,** and flexible **JavaScript library** for building reusable UI components. The main objective of ReactJS is to develop User Interfaces (UI) that improves the speed of the apps. We can use ReactJS on the client and server-side as well as with other frameworks.

A ReactJS application is made up of multiple components, each component responsible for outputting a small, reusable piece of HTML code. The components are the heart of all React applications. These Components can be nested with other components to allow complex applications to be built of simple building blocks.

ReactJS uses virtual DOM based mechanism to fill data in HTML DOM. The virtual DOM works fast as it only changes individual DOM elements instead of reloading complete DOM every time.

The DOM (Document Object Model) is an object which is created by the browser each time a web page is loaded. It dynamically adds or removes the data at the back end and when any modifications were done, then each time a new DOM is created for the same page. This repeated creation of DOM makes unnecessary memory wastage and reduces the performance of the application.

2.1.2. Using react with HTML

- JSX stands for JavaScript XML. — Extension
- JSX allows us to write HTML in React.
- JSX makes it easier to write and add HTML in React.
- JSX allows us to write HTML elements in JavaScript and place them in the DOM without any createElement() and/or appendChild() methods.
- JSX converts HTML tags into react elements.
- JSX provides you to write HTML/XML-like structures (e.g., DOM-like tree structures) in the same file where you write JavaScript code, then pre-processor will transform these expressions into actual JavaScript code. Just like XML/HTML, JSX tags have a tag name, attributes, and children. Components are like functions that return HTML elements. Components are independent and reusable bits of code. They serve the same purpose as JavaScript functions, but work in isolation and return HTML.

  React renders HTML to the web page by using a function called ReactDOM.render().

2.1.3. React Interactive Components: Components within Components & Files

React components are like functions that return HTML elements. React components let you break up the user interface into separate pieces that can then be reused and handled independently.

A Component is considered as the core building blocks of a React application. It makes the task of building UIs much easier. Each component exists in the same space, but they work independently from one another and merge all in a parent component, which will be the final UI of your application.

There are two types of components:

1. Class Components
2. Function Components

1. Class Components:

- A class component must include the extends React.Component statement.
- This statement creates an inheritance to React.Component, and gives your component access to React.Component's functions.
- The component also requires a render() method, this method returns HTML.
- You can pass data from one class to other class components.
- You can create a class by defining a class that extends Component and has a render function.

Example:

- **Create Fruit Component**

In App.js file

```
import logo from './logo.svg';
import './App.css';

import React, { Component } from 'react'


export default class Car extends React.Component {
 render() {
  return <h2>Hi, I am a Car!</h2>;
 }
}
```

Or

In App.js file

```
import logo from './logo.svg';
import './App.css';
import Car from './Components/Car'
```

```
export default Car
```

In Car.js file

```
import React, {Component} from "react";

class Car extends React.Component{
 render(){
  return (
  <div>
   <h1>My Car</h1>
   <h5>This is React js examole</h5>
  </div>

  )
 }
}
export default Car
```

2. **Function Component:**
   - A Function component also returns HTML, and behaves much the same way as a Class component, but Function components can be written using much less code.
   - The functional component is also known as a stateless component because they do not hold or manage state.

Example:

- **Create Car Component**
- **In App.js file**

```
import logo from './logo.svg';
import './App.css';
import Car from './Components/Car'
import { Component } from 'react';
class App extends Component{
 render(){
  return(
   <div className='App'>
    <Car/>
   </div>
  );
 }
}
export default App
```

- **In Car.js file**

```
import React, {Component} from "react";
function Car() {

    return <h2>Hi, I am a Car!</h2>;

 }
export default Car
```

- **Components within Component:**

We can refer to components inside other components:

Use the mango component inside the fruit component:

Example:

```
function mango() {
 return <h2>I am a mango!</h2>;
}
function fruit() {
 return (
  <>
   <h1>Which fruit you are?</h1>
   <mango />
  </>
 );
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<fruit />);
```

- **Components in Files**

React is all about re-using code, and it is recommended to split your components into separate files.

To do that, create a new file with a .js file extension and put the code inside it:

Note that the filename must start with an uppercase character.

First create new file named Car.js

```
function Car() {
  return <h2>Hi, I am a Car!</h2>;
        }
```

```
export default Car;
```

To be able to use the Car component, you have to import the file in your application.Import the "Car.js" file in the application, and we can use the Car component as if it was created here.

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import Car from './Car.js';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Car />);
```

- **Props:**
  - Components can be passed as props, which stands for properties.
  - Props are like function arguments, and you send them into the component as attributes.
  - Use an attribute to pass a color to the Car component, and use it in the render() function:

```
function Car(props) {
  return <h2>I am a {props.color} Car!</h2>;
  }
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Car color="red"/>);
```

  - Props are also how you pass data from one component to another, as parameters.
  - Send the "brand" property from the Garage component to the Car component:

```
function Car(props) {
  return <h2>I am a { props.brand }!</h2>;
}

function Garage() {
  return (
    <>
      <h1>Who lives in my garage?</h1>
      <Car brand="Ford" />
    </>
  );
}
```

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Garage />);
```

- Create a variable named carName and send it to the Car component:

```
function Car(props) {
  return <h2>I am a { props.brand }!</h2>;
}

function Garage() {
  const carName = "Ford";
  return (
    <>
      <h1>Who lives in my garage?</h1>
      <Car brand={ carName } />
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Garage />);
```

React allows you to pass props but only down the family tree. A parent can only pass information to the children. Children can not pass props up to the parent. This is the one way data flow of React.

2.2. Class Components

2.2.1 React Class & Class Components:

- React **class based components** are made up of multiple functions that add functionality to the application). All **class based components** are child classes for the Component class of ReactJS.

- Creation of class-based components. Create a React app and edit the **App.js** as:

```
import React from "react";

class App extends React.Component {
render() {
        return <h1>Hello Everyone!</h1>;
}
}
```

```
export default App;
```

2.2.2 Conditional statements, Operators & Lists:

- In react js we can conditionally render components.

1. **If statement:**
   If is a javascript operator to decide which component to render.
   For example here we are using 2 components:

```
function pendingwork()
 {
   Return <h1>Pending</h1>;
 }

function workdone()
 {
   Return <h1>Done</h1>;
 }
```

Now, we will make another component that will choose which component to render based on condition.

```
import React from 'react';
import ReactDOM from 'react-dom/client';

function pendingwork()
        {
          Return <h1>Pending</h1>;
        }

      function workdone()
        {
          Return <h1>Done</h1>;
        }
function Done(props) {
 const isDone = props.isDone;
 if (isDone) {
  return <workdone/>;
 }
 return <pendingwork/>;
}

Const root =
ReactDOM.createRoot(document.getElementById('root
'));
root.render(<Work isDone={false} />);
```

Here, the value of isDone is false so the output will be Pending

If we will put true instead of false then output will be Done

```
const root =
ReactDOM.createRoot(document.getElementById('root
'));
root.render(<Goal isGoal={true} />);
```

2. **Logical && operator:**

Another way to render a react component is by using && Operator.

```
import React from 'react';
import ReactDOM from 'react-dom/client';

function Garage(props) {
 const cars = props.cars;
 return (
  <>
   <h1>Garage</h1>
   {cars.length > 0 &&
    <h2>
     You have {cars.length} cars in your garage.
    </h2>
   }
  </>
 );
}

const cars = ['Ford', 'BMW', 'Audi'];
const root =
ReactDOM.createRoot(document.getElementById('root
'));
root.render(<Garage cars={cars} />);
```

If cars.length is equal to true, the expression after && will render.
Try to empty the cars array:

```
import React from 'react';

import ReactDOM from 'react-dom/client';

function Garage(props) {

 const cars = props.cars;
```

```
return (
 <>
  <h1>Garage</h1>
  {cars.length > 0 &&
   <h2>
    You have {cars.length} cars in your garage.
   </h2>
  }
 </>
);
}
const cars = [];
const root =
ReactDOM.createRoot(document.getElementById('root'));
root.render(<Garage cars={cars} />);
```

3. Ternary Operator:

This is also used to render components conditionally.

condition ? true : false

Return the MadeGoal component if isGoal is true, otherwise return the MissedGoal component:

```
import React from 'react';
import ReactDOM from 'react-dom/client';
function MissedGoal() {
        return <h1>MISSED!</h1>;
}
function MadeGoal() {
```

```
        return <h1>GOAL!</h1>;
    }
    function Goal(props) {
     const isGoal = props.isGoal;
        return (
            <>
                { isGoal ? <MadeGoal/> : <MissedGoal/> }
            </>
        );
    }

    const root =
    ReactDOM.createRoot(document.getElementById('root'));

    root.render(<Goal isGoal={false} />);
```

React Lists:

- You will render lists with some type of loop in react.
- The JavaScript map() array method is used for this.
- A map is a data collection type where data is stored in the form of pairs. It contains a unique key.
- The value stored in the map must be mapped to the key.
- We cannot store a duplicate pair in the map().
- It is because of the uniqueness of each stored key.
- It is mainly used for fast searching and looking up data.

```
import React from 'react';
import ReactDOM from 'react-dom/client';

function Car(props) {
 return <li>I am a { props.brand }</li>;
}

function Garage() {
 const cars = ['Ford', 'BMW', 'Audi'];
 return (
  <>
```

```
        <h1>Who lives in my garage?</h1>
        <ul>
      {cars.map((car) => <Car brand={car} />)}
     </ul>
    </>
  );
}
const root =
ReactDOM.createRoot(document.getElementById('root'));
root.render(<Garage />);

/*
If you run this example in your create-react-app,
you will receive a warning that there is no "key" provided for the list items.
*/
```

- Keys allow React to keep track of elements. This way, if an item is updated or removed, only that item will be re-rendered instead of the entire list.
- Keys need to be unique to each sibling. But they can be duplicated globally.
- Generally, the key should be a unique ID assigned to each item.

```
import React from 'react';
import ReactDOM from 'react-dom/client';
function Car(props) {
 return <li>I am a { props.brand }</li>;
}
function Garage() {
 const cars = [
  {id: 1, brand: 'Ford'},
  {id: 2, brand: 'BMW'},
  {id: 3, brand: 'Audi'}
 ];
 return (
  <>
        <h1>Who lives in my garage?</h1>
        <ul>
    {cars.map((car) => <Car key={car.id} brand={car.brand} />)}
   </ul>
  </>
 );
}

const root =
ReactDOM.createRoot(document.getElementById('root'));
root.render(<Garage />);
```

2.2.3 React Events: Adding events, Passing arguments, Event objects:

- Like HTML DOM events, React can perform actions based on user events.
- React has the same events as HTML: click, change, mouseover etc.

Adding Events

- React events are written in camelCase syntax:
  (camelCase is a naming convention in which the first letter of each word in a compound word is capitalized, except for the first word.)
  onClick instead of onclick.
- React event handlers are written inside curly braces:
  onClick={shoot} instead of onClick="shoot()".

Example:

```
<button onClick={shoot}>Take the Shot!</button>

import React from 'react';

import ReactDOM from 'react-dom/client';

function Football() {

 const shoot = () => {

  alert("Great Shot!");

 }

 return (

  <button onClick={shoot}>Take the shot!</button>

 );

}

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Football />);
```

Passing Arguments:

To pass an arguments to an event handler we have to use arrow function.

Example:

```
import React from 'react';
import ReactDOM from 'react-dom/client';
function Football() {
  const shoot = (a) => {
    alert(a);
  }
  return (
    <button onClick={() => shoot("Goal!")}>Take the
shot!</button>
  );
}
const root =
ReactDOM.createRoot(document.getElementById('root'));
root.render(<Football />);
```

React Event Object

Event handlers have access to the React event that triggered the function.

Following example is of click event:

Arrow Function: Sending the event object manually:

```
import React from 'react';
import ReactDOM from 'react-dom/client';

function Football() {
  const shoot = (a, b) => {
    alert(b.type);
                /*
'b' represents the React event that triggered the function.
    In this case, the 'click' event
                */
  }

  return (
    <button onClick={(event) => shoot("Goal!", event)}>Take the
shot!</button>
  );
}

const root =
ReactDOM.createRoot(document.getElementById('root'));
root.render(<Football />);
```

Important Questions:

1. Explain React JS Components.
2. How to Pass Data Through Props?
3. Explain Concept of React Js.
4. Explain Conditional Statements and Operators.

Unit- 3
Forms and Hooks in React JS

3.1 Forms

Form is a crucial part of web application. Web form is an interactive online page that allows to take inputs from the user whenever it is required. Web forms can be availed in modern web browsers using various web-oriented languages. Web forms are collection of various components like text boxes, labels, buttons, Textarea, etc. For example: We are looking at a photograph on social media. Suppose when we looked at photo the likes of the same was 65 and within fraction of time the likes get changed to 85 and the page did not refresh. This is what performed by the portion of react JS.

- **Adding a form in React JS**

Same as HTML we can add a form in React JS and allow the user to interact with the web page. But the difference is, in HTML when we submit the form page will be refreshed and in React JS will not happen in such a way. Form in React JS is added using function component.

Example:

```
import React from 'react';
import ReactDOM from 'react-dom/client';
function FormDemo1() {
 return (
  <form>
   <label>Enter name:
    <input type="text" />
   </label>
  </form>
 )
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<FormDemo1 />);
```

In the above example form named FormDemo.js is added with the only one textbox element. It will work as normal form in HTML as no other elements are there.

- **Handling Forms in React JS**

Forms are handled by DOM (Document Object Model) in HTML. In React JS forms are handled by the components. When forms are handled by components, data is stored in component state.

Components can be of two types:

- Controlled components: It is same as the HTML form inputs and which is handled by the DOM.
- Uncontrolled components: Rather than controlling data only at clicking submit button, uncontrolled components have functions which controls data being passed on every change made.

Example:

```
import { useState } from "react";
import ReactDOM from 'react-dom/client';
function FormDemo2() {
 const [name, setName] = useState("");
 return (
  <form>
   <label>Enter name:
    <input type="text" value={name} onChange={(e) => setName(e.target.value)}/>
   </label>
  </form>)
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<FormDemo2 />);
```

In above example to control the changes in data event handler is added in onChange attribute.

- **Submitting a form in React JS**

Form submission is handled using even handler in onSubmit attribute for the form element.

Example:

```
import { useState } from "react";
import ReactDOM from 'react-dom/client';
function MyForm() {
 const [name, setName] = useState("");
 const handleSubmit = (event) => {event.preventDefault();
  alert(`The name you entered was: ${name}`);
 }
 return (
  <form onSubmit={handleSubmit}>
    <label>Enter name:
      <input type="text" value={name} onChange={(e) => setName(e.target.value)}/>
    </label>
    <input type="submit" />
  </form>
 )
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<FormDemo3 />);
```

3.1.1 event.target.name, event.target.event and React Memo

To handle the values of more than one input field event handlers event.target.name and event.target.value is used. Values of multiple input fields are managed by giving name attribute to each element. Square bracket is used around the property name to update the state. State is generally initialized with empty object.

React memo is considered as a higher order component. It stops rendering the components which results the same value and reuses the last rendered value. It works as a performance booster.

3.1.2 Components

Textarea

In react JS textarea is not working like HTML because in HTML we put value in starting and ending tag while in React JS value of textarea is placed in value attribute. useState hook is used to manage the textarea.

Example:

```
import { useState } from 'react';
import ReactDOM from 'react-dom/client';
function DemoTextArea() {
 const [textarea, setTextarea] = useState("The content of a textarea goes in the value attribute");
 const handleChange = (event) => {setTextarea(event.target.value)}
 return (
  <form>
   <textarea value={textarea} onChange={handleChange} />
  </form>
 )
}
```

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<DemoTextArea />);
```

Dropdown list

Dropdown list is also known as select box. Selected value is placed in value attribute of the select tag.

Example:

```
import { useState } from "react";
import ReactDOM from "react-dom/client";
function DemoDropdownlist() {
 const [myCar, setMyCar] = useState("i20");
 const handleChange = (event) => {setMyCar(event.target.value)
 }
 return (
  <form>
   <select value={myCar} onChange={handleChange}>
    <option value="i20"> i20</option>
    <option value="KIA"> KIA </option>
    <option value="Baleno"> Baleno </option>
   </select>
  </form>
 )
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<DemoDropdownlist/>);
```

3.2 Hooks

Hooks are introduced in React 16.8 version. It allows to use react features without writing in a class. It can not work in the class. There are two rules for hook. Hooks can not be called inside the looping structure, conditions and nested functions. Hook can not be called from regular javascript functions.

Advantages

Hook improves component readability.

It is easier to work and to test with hooks.

It implements composable and reusable logic.

3.2.1 useState, useEffect, useContext

useState: It is used to tract the state in a function component.

Example:

```
import { useState } from "react";
import ReactDOM from "react-dom/client";
function DemoState() {
 const [color, setColor] = useState("red");
 return (
  <>
   <h1>My favorite color is {color}!</h1>
   <button type="button" onClick={() => setColor("blue")}>Blue</button>
  </>
 )
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<DemoState />);
```

useEffect: To perform side effects in component useEffect hook is used. It runs on every render.

Example:

```
import { useState, useEffect } from "react";
import ReactDOM from "react-dom/client";
function Timer() {
 const [count, setCount] = useState(0);
 useEffect(() => {setTimeout(() => {setCount((count) => count + 1);}, 1000);});
 return <h1>I've rendered {count} times!</h1>;
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Timer />);
```

useContext: It is use with useState hook to share state between deeply nested components.

3.2.2 useRef, useReducer, useCallback, useMemo

useRef: It is used to store mutable values which will not be updated when re-rendered. It returns only one item which is current object.

useReducer: It is like useState hook but useReducer is used to keep track of multiple pieces of state which is depends on complex logic.

Syntax : useReducer(<reducer>, <initialState>)

useCallback: It identifies the resource intensive functions and stops them to run automatically.

useMemo: It is same as useCallback but useMemo keep track of cache value which should not be recalculated. The only difference is useMemo returns memorized value and useCallback returns memorized function.

3.2.3 Hook: Building custom hook, advantages and use

Hooks are considered reusable functions. custom hooks are nothing but a javascript function only. Custom hook name starts with "use". "use" keyword declares that the function is bound to follow the rules of hook.

Advantages:

It provides reusability of the code.

Other benefits are same as the inbuilt hooks.

Example:

```
import React, { useState, useEffect } from 'react';
const useCustomh = title => {useEffect(() => {document.title = title;}, [title])}
function CustomCounter() {
const [count, setCount] = useState(0);
const incrementCount = () => setCount(count + 1);
useCustomh(`You clicked ${count} times`);
return (
  <div>
   <p>You clicked {count} times</p>
   <button onClick={incrementCount}>Click me</button>
  </div>
 )
}
export default CustomCounter;
```

In the above example useCustomh is a custom hook which takes a string argument. In the custom hook useEffect hook is called.

Important Questions

1) How can forms be added and submitted in React JS?
2) Explain form handling method in React JS with an example.
3) What is event handler? Explain event.target.name and event.target.event.
4) Describe below listed components of form with an example.
    a. Textarea
    b. Dropdown list
5) What is hook? List and explain different built-in hooks.
6) How does the custom hook differ from built in hook?

Unit- 4
Angular JS

4.1 Concepts and characteristics of Angular JS

Angular JS is an opensource framework to develop dynamic web applications. It was developed in 2009 by Misko Hevery and Adam Abrons. Now a days it is maintained by the google. Latest version of it is 1.2.21. It follows MVC (Model View Controller) architecture.

Features:

- **MVC** – This is widely used structure which separates business logic layer, data layer and presentation layer into different section.
- **Data model binding** – No need to write special code to bind data to HTML controls, it is done just by adding few lines of code.
- **Writing less code** – As compared to DOM, less lines of javascript code is required to design any application.
- **Ready unit testing** – There is a framework called "Karma" which provide hassle free unit testing of Angular JS applications.

Advantages:

As it is an opensource framework minimal error and issues can be expected.

It provides Unit and Integration testing.

By providing concept of single page application, though you access different functionalities of web applications still you stay on same page.

4.1.1 Expressions in Angular JS

To bind application data in HTML expressions are used in Angular JS. Expressions in Angular JS are written in curly braces.

Syntax: {{expression}}

Expressions of Angular JS gives the output at the same place where they are written.

Example written below define the different types of expressions.

```
<html>
  <head>
    <title>AngularJS Expressions</title>
```

```
</head>
<body>
  <div ng-app = "" ng-init = "quantity = 2;cost = 30; student =
{firstname:'Ram',lastname:'Patel',rollno:1}; marks = [80,90,75,73,60]">
    <p> Example of string expression</p>
    <p>Hello {{student.firstname + " " + student.lastname}}!</p>
    <p> Example of number expression</p>
    <p>Expense on Books : {{cost * quantity}} Rs</p>
    <p> Example of object</p>
    <p>Roll No: {{student.rollno}}</p>
    <p> Example of array</p>
    <p>Marks(Math): {{marks[3]}}</p>
  </div>
  <script src = "https://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js">
  </script>
</body>
</html>
```

4.1.2 Setting up environment

One can download Angular JS library from https://angularjs.org/ source. By clicking on the above link below kind of screen will be displayed which will show two options to download Angular JS library.

Image 1 Two options to download Angular JS Library

Example:

By clicking on "View GitHub ", one will be diverted to GitHub and will get all the latest scripts. Or on can go by clicking "Download Angular JS 1" and a dialog box will be shown which will be look as displayed in below image.

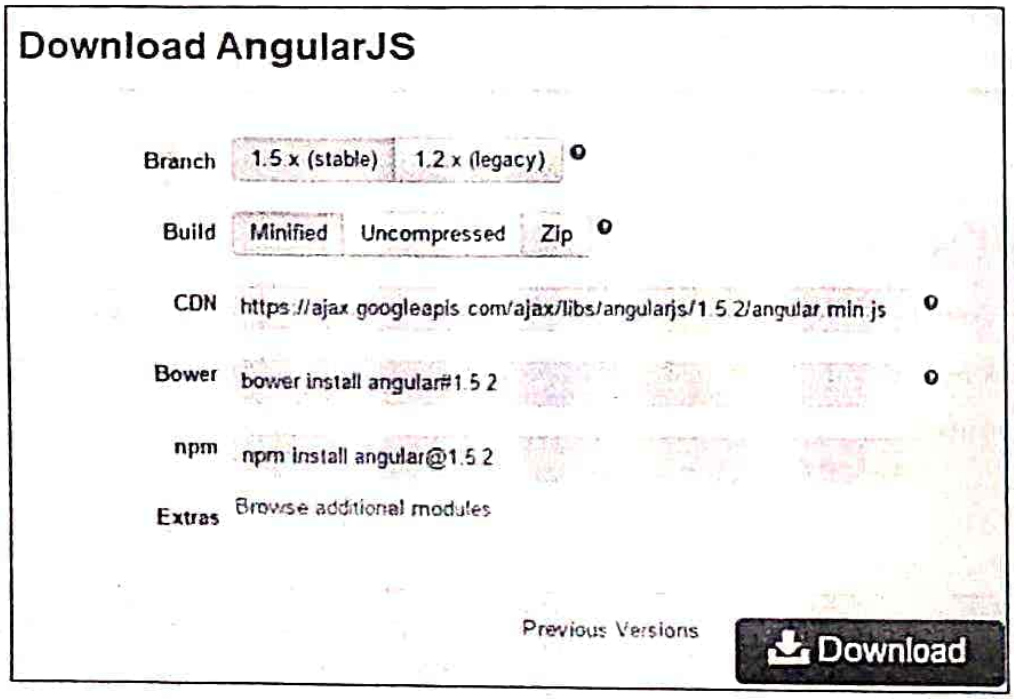

Image 2 Download by clicking on "Download Angular JS 1"

Above screen shows various options to use Angular JS as follow:

Downloading and hosting files locally – There are two options legacy and lates. Legacy will give versions below 1.2.x and latest will give versions above 1.3.x. One can also have minimized, zipped or uncompressed version.

CDN (Content Delivery Network) access – In this case google is a host and it gives access to regional data centers.

4.1.3 Understanding MVC architecture

MVC stands for Model View Controller. It is a designing pattern for web applications. It is very popular as it separates business logic layer, data layer and presentation layer. MVC architecture contains three parts.

Model – This layer is responsible for managing application data. It gives response to the requests from view and controller.

View – It is responsible for representing data to users. They are script-based templates like ASP, PHP, JSP, etc.

Controller – It maintains the relation between models and views. Controller receives inputs, validates the same and performs the business operations.

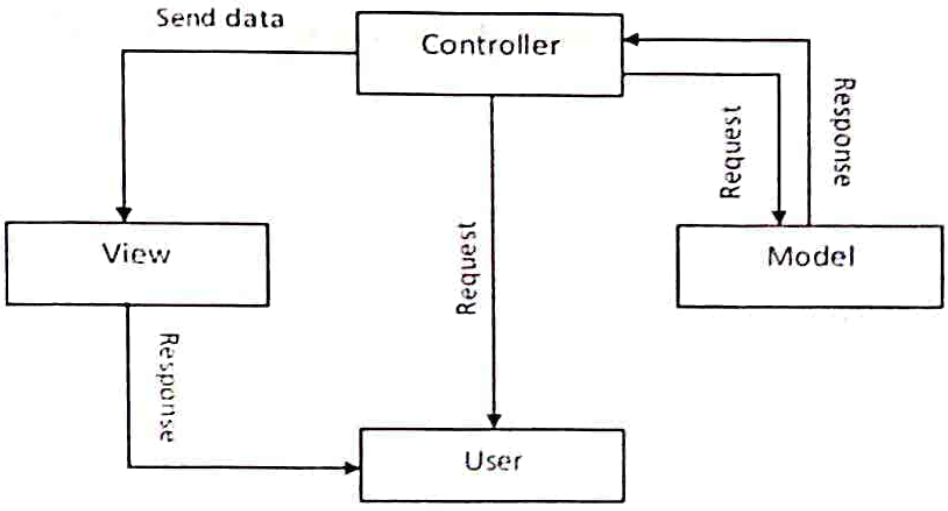

Image 3 Model View Controller

4.2. AngularJS Directives

Extended HTML attributes with ng- prefix are known as Angular JS Directives. Some of them are as follows.

ng-app – It initializes Angular JS application.

ng-init – This directive initializes application data.

ng-controller – It defines controller object for an application.

ng-model – It binds the value of HTML controls to application data.

Example:

```
<div ng-app="" ng-init="firstName='XYZ'">
<p>Name: <input type="text" ng-model="firstName"></p>
<p>You wrote: {{ firstName }}</p>
</div>
```

ng-repeat – used on an array of objects

Example:

```
<div ng-app="" ng-
init="names=[{name:'Ram',country:'India'},{name:'Sita',country:'USA'},{name:'Jay',country:'
UK'}]">
<ul>
 <li ng-repeat="x in names">
  {{ x.name + ', ' + x.country }}
 </li>
</ul>
</div>
```

4.2.1 Some other directives

ng-class – It specifies CSS classes dynamically on HTML elements. This directive is supported by all the HTML elements.

Syntax: <element ng-class="expression"></element>

Expression returns one or more CSS class names.

ng-animate – This directive provides support for CSS-based animations.

ng-show – It shows the HTML element if the given expression is true otherwise it hides.

Syntax: <element ng-show="expression"></element>

It specifies an expression that will show the element only if the expression returns true.

ng-hide – This directive is used to hide the HTML element if the expression is set to true.

Syntax: <element ng-hide="expression"></element>

It specifies an expression that will hide the element if the expression returns true.

4.2.2 Angular JS controller

Angular JS controller is defined using ng-controller directive. It is used to control the flow of the data. It contains attributes/properties and functions. Controller accepts $scope as a parameter which specifies the module or application to be controlled.

Example

```
<script>
 function studController($scope) {
   $scope.student = {
     firstName: "Ram",
     lastName: "Patel",
     fullName: function() {
       var studObject;
       studObject = $scope.student;
```

```
        return studObject.firstName + " " + studObject.lastName;
      }
    };
  }
</script>
```

The studController is defined as a JavaScript object with $scope as an argument. The $scope refers to application which uses the studController object. The $scope.student is a property of studController object. The firstName and the lastName are two properties of $scope.student object. We pass the default values to them. The property fullName is the function of $scope.student object, which returns the combined name. In the fullName function, it gets the student object and then return the combined name.

Now we can use studentController's student property using ng-model or using expressions as follows:

```
Enter first name: <input type = "text" ng-model = "student.firstName"><br>
Enter last name: <input type = "text" ng-model = "student.lastName"><br>
<br>
You are entering: {{student.fullName()}}
```

We bound student.firstName and student.lastname to two input boxes.

We bound student.fullName() to HTML.

4.2.3 Filters

Angular JS uses filter to format the data. They can be used in expression or directives using pipe sign.

uppercase – It converts text to upper case.

Syntax:

```
{{expression | uppercase}}
</div>
```

lowercase – It converts text to lower case.

Syntax:

{{expression | lowercase }}

currency – It formats text in currency format.

Syntax:

{{ expression | currency }}

orderby – It arranges data in an order as specified in the expression

Syntax:

orderBy:expression

Important Questions

1) List and describe features of Angular JS.
2) What is expression in Angular JS? Explain various expression with an example.
3) Describe MVC architecture in detail.
4) List and describe various directives of Angular JS.
5) What do you mean by Filter? Explain uppercase, lowercase, currency and order by filters.

Unit- 5

Angular Js: Single Page Application

5.1. Single Page Application in Angular JS

AngularJS is an open-source front-end web application framework based on JavaScript, it is widely used for building Single Page Application (SPA). It extends the traditional HTML with new attributes and data-binding components that enable us to serve dynamic page content. In this article, we'll learn how to build a Single Page Application (SPA) using AngularJS and AngularJS Route library.

5.1.1 What Is Single Page Application?

AngularJS Single page application (SPA) is a web application that is contained in a single page. In a single page application all our code (JS, HTML, CSS) is loaded when application loads for the first time. Loading of the dynamic contents and the navigation between pages is done without refreshing the page.

Step 1:- This is the simple part. Here we will create a simple index.html file and add a simple layout with a navigation bar. We will also load angular and angular route library via CDN.

index.html

```
<html>
<head>
  <title>
    Angular Routing
  </title>
</head>
<style>
  #nav{
    width: 19%;
    display: inline-block;
    background: #429cd6;
    border: 1px solid #3c87b2;

  }
  #page-content{
    width: 79%;
    display: inline-block;
    vertical-align: top;
    padding: 8px;
```

```
        padding-top: 0px;
      }
      ul{
        list-style: none;
        padding-right: 10px;
      }
      li{
        background-color: #ffffff;
        color: white;
        margin: 5px;
        text-decoration: none;
        text-align: center;
        padding: 3px;
        font-size: 20px;
        font-family: 'Helvetica', sans-serif;
      }
      a{
        color: black;
        text-decoration: none;
      }
    </style>
    <!-- Load AngularJS and Angular Route JS -->
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular-
    route.js"></script>
    <body ng-app="myApp">

    <!-- AngularJS Navigation -->
    <div id="nav">
      <ul>
        <li><a href="#!">Home</a></li>
        <li><a href="#!about">About</a></li>
        <li><a href="#!services">Services</a></li>
        <li><a href="#!projects">Projects</a></li>
      </ul>
    </div>
    <div id="page-content">
      <h1>AngularJS Single Page Application - W3Adda</h1>
```

```
  <!-- this is where page content will be loaded -->
  <div ng-view></div>
</div>
<!-- Load AngularJS Module and Angular Route Configuration -->
<script type="text/javascript" src="script.js"></script>
</body>
</html>
```

Step 2:-

Angular Application

Every angular application starts from creating a angular module. Module is a container that holds the different parts of your application such as controllers, service, etc. In this step we will create a simple javascript file **script.js** and put the following code in it to create our angular module.

script.js

```
// create the module and name it myApp
var app = angular.module("myApp", ['ngRoute']);
```

The **app** variable is initialized as angular module and named as **"myApp"**, the **ng-app** directive is used to bind the angular application to a container. It is important to use the same module name to bind using **ng-app** directive.

Step 3:-

Configure Routes

Since we are making a single page application and we don't want any page refreshes. Here we'll use Angular's routing capabilities to setup single page application.We will use ngRoute module for that. Notice that we have added ngRoute as a part of the parameter passed to our angular module. The ngRoute module provides routing, deeplinking services and directives to our angular applications. We will be using **$routeProvider** in **app.config** to configure our application routing. Let's open our **script.js** file and put the following in it –

script.js

```
var app = angular.module("myApp", ['ngRoute']);
// configure routes
app.config(function($routeProvider){
```

```
$routeProvider
.when("/", {template:"<p>AngularJS Single Page Application(SPA).</p>"})
.when("/about", {templateUrl : "about.html"})
.when("/services", {templateUrl : "services.html"})
.when("/projects", {templateUrl : "projects.html"})
});
```

Now we have defined our routes with **$routeProvider**. As you can see by the configuration, you can specify the **route**, the **template** file to use, and even a **controller**.

AngularJS Template

Writing HTML code inside template is very constraining and to counter it one can use templateURL and instead of writing HTML code directly we specify the HTML file that will be injected corresponding to route. Now, we just need to define the page templates that will be injected.

about.html

```
<h2>About</h2>
<p>About Page Contents</p>
```

services.html

```
<h2>Services</h2>
<p>Service Page Contents</p>
```

projects.html

```
<h2>Projects</h2>
<p>Project Page Contents</p>
```

AngularJS View

To finish off with this tutorial, we need to specify the container where content of each page will be loaded in our layout. In AngularJS, there is a **ng-view** directive that will injects the template file of the current route (/about , /services or /projects) in the main layout file. Notice that we have added **ng-view** directive in main (**index.html**) layout file.

```
<div id="page-content">
```

```
<h1>AngularJS Single Page Application </h1>
<!-- this is where page content will be loaded -->
<div ng-view></div>
</div>
```

Output:-

5.1.2 Embedding Angular.Js Script in HTML

The ng-bind-html Directive is used to bind the innerHTML of an HTML element to application data and remove dangerous code from the HTML string. $sanitize service is a must for **ng-bind-html** directive. It is supported by all HTML elements.

Syntax:

```
<element ng-bind-html="expression"> Contents... </element>
```

Approach:

- Initialize the libraries that we are using. Here, we are mainly use angular-sanitize.js library.
- Create app and its controller scope.
- Define the controller variable.
- Call the app and controller to body of html.
- Inside the body use span tag and use attribute ng-bind-html and assign the value as the scope variable.

Example:

```
<html>
<head>
  <meta charset="utf-8">
</head>
```

```
<body ng-app="myApp" ng-controller="myCtrl">
  <span ng-bind-html="message"></span>
  <script src=
"https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.6.9/angular.min.js"
        charset="utf-8">
    </script>
  <script src=
"https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular-sanitize.js"
        charset="utf-8">
</script>
  <script type="text/javascript">
  var app=angular.module('myApp',['ngSanitize']);
  app.controller('myCtrl',function($scope){
    $scope.message = '<h1>Jump2Learn</h1>';
  });
  </script>
</body>
</html>
```

5.1.3 Routing Capability of AngularJS

If you want to navigate to different pages in your application, but you also want the application to be a SPA (Single Page Application), with no page reloading, you can use the ngRoute module.

The ngRoute module *routes* your application to different pages without reloading the entire application.

Example:

Navigate to "red.htm", "green.htm", and "blue.htm":

```
<body ng-app="myApp">
```

```
<p><a href="#/!">Main</a></p>

<a href="#!red">Red</a>
<a href="#!green">Green</a>
<a href="#!blue">Blue</a>

<div ng-view></div>

<script>
var app = angular.module("myApp", ["ngRoute"]);
app.config(function($routeProvider) {
 $routeProvider
 .when("/", {
  templateUrl : "main.htm"
 })
 .when("/red", {
  templateUrl : "red.htm"
 })
 .when("/green", {
  templateUrl : "green.htm"
 })
 .when("/blue", {
  templateUrl : "blue.htm"
 });
});
</script>
</body>
```

$routeProvider

With the $routeProvider you can define what page to display when a user clicks a link.

Example:

Define a $routeProvider:

```
var app = angular.module("myApp", ["ngRoute"]);
app.config(function($routeProvider) {
 $routeProvider
 .when("/", {
  templateUrl : "main.htm"
 })
 .when("/india", {
```

```
    templateUrl : "jump2learn.htm"
  })
  .when("/paris", {
    templateUrl : "bepositive.htm"
  });
});
```

Controllers:

With the $routeProvider you can also define a controller for each "view".

Example:

```
var app = angular.module("myApp", ["ngRoute"]);
app.config(function($routeProvider) {
 $routeProvider
 .when("/", {
   templateUrl : "main.htm"
 })
 .when("/india", {
   templateUrl : "india.htm",
   controller : "indiaCtrl"
 })
 .when("/paris", {
   templateUrl : "paris.htm",
   controller : "parisCtrl"
 });
});
app.controller("indiaCtrl", function ($scope) {
 $scope.msg = "I love india";
});
app.controller("parisCtrl", function ($scope) {
 $scope.msg = "I love Paris";
});
```

The "india.htm" and "paris.htm" are normal HTML files, which you can add AngularJS expressions as you would with any other HTML sections of your AngularJS application.

5.2 HTML DOM Directives

5.2.1 AngularJS HTML DOM

In AngularJS, some directives can be used to bind application data to attributes of HTML DOM elements.

These directives are:

Directive	Description
ng-disabled	It disables a given control.
ng-show	It shows a given control.
ng-hide	It hides a given control.
ng-click	It represents an Angular JS click event.

ng-disabled directive: The ng-disabled directive binds AngularJS application data to the disabled attribute of HTML elements. In the below code, it binds a model to a checkbox.

```
<input type = "checkbox" ng-model = "enableDisableButton">Disable Button
button ng-disabled = "enableDisableButton">Click</button>
```

ng-show directive: The ng-show directive shows or hides an HTML element. In the below code, it binds a model to a checkbox

```
<input type = "checkbox" ng-model = "showHide1">Show Button
button ng-show = "showHide1">Click</button>
```

ng-hide directive: The ng-hide directive hides or shows an HTML element. In the below code, it binds a model to a checkbox.

```
<input type = "checkbox" ng-model = "showHide2">Hide Button
<button ng-hide = "showHide2">Click </button>
```

ng-click directive: The ng-click directive counts the total clicks an HTML element. In the below code, it binds a model to a checkbox.

```
<p>Total click: {{ clickCounter }}</p>
It;button ng-click = "clickCounter = clickCounter + 1">Click </button>
```

5.2.2 AngularJS Module

In AngularJS, a module defines an application. It is a container for the different parts of your application like controller, services, filters, directives etc.

A module is used as a Main() method. Controller always belongs to a module.

How to create a module

The angular object's module() method is used to create a module. It is also called AngularJS function angular.module

```
<div ng-app="myApp">...</div>
<script>
var app = angular.module("myApp", []);
</script>
```

Here, "myApp" specifies an HTML element in which the application will run.

Now we can add controllers, directives, filters, and more, to AngularJS application.

How to add controller to a module

If you want to add a controller to your application refer to the controller with the ng-controller directive.

Example:

```
<!DOCTYPE html>
<html>
<script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.4.8/angular.min.js"></
script>
<body>
<div ng-app="myApp" ng-controller="myCtrl">
{{ firstName + " " + lastName }}
</div>
<script>
var app = angular.module("myApp", []);
app.controller("myCtrl", function($scope) {
  $scope.firstName = "Ishaan";
  $scope.lastName = "Tamhankar";
});
</script>
</body>
</html>
```

How to add directive to a module

AnglarJS directives are used to add functionality to your application. You can also add your own directives for your applications.

Following is a list of AngularJS directives:

Directive	Description
ng-app	It defines the root element of an application.
ng-bind	It binds the content of an html element to application data.
ng-bind-html	Itbinds the innerhtml of an html element to application data, and also removes dangerous code from the html string.
ng-bind-template	It specifies that the text content should be replaced with a template.
ng-blur	It specifies a behavior on blur events.
ng-change	It specifies an expression to evaluate when content is being changed by the user.
ng-checked	It specifies if an element is checked or not.
ng-class	It specifies css classes on html elements.
ng-class-even	It is same as ng-class, but will only take effect on even rows.
ng-class-odd	It is same as ng-class, but will only take effect on odd rows.
ng-click	It specifies an expression to evaluate when an element is being clicked.
ng-cloak	It prevents flickering when your application is being loaded.
ng-controller	It defines the controller object for an application.
ng-copy	It specifies a behavior on copy events.
ng-csp	It changes the content security policy.
ng-cut	It specifies a behavior on cut events.
ng-dblclick	It specifies a behavior on double-click events.
ng-disabled	It specifies if an element is disabled or not.
ng-focus	It specifies a behavior on focus events.
ng-form	It specifies an html form to inherit controls from.
ng-hide	It hides or shows html elements.

ng-href	It specifies a URL for the <a> element.
ng-if	It removes the html element if a condition is false.
ng-include	It includes html in an application.
ng-init	It defines initial values for an application.
ng-jq	It specifies that the application must use a library, like jQuery.
ng-keydown	It specifies a behavior on keydown events.
ng-keypress	It specifies a behavior on keypress events.
ng-keyup	It specifies a behavior on keyup events.
ng-list	It converts text into a list (array).
ng-model	It binds the value of html controls to application data.
ng-model-options	It specifies how updates in the model are done.
ng-mousedown	It specifies a behavior on mousedown events.
ng-mouseenter	It specifies a behavior on mouseenter events.
ng-mouseleave	It specifies a behavior on mouseleave events.
ng-mousemove	It specifies a behavior on mousemove events.
ng-mouseover	It specifies a behavior on mouseover events.
ng-mouseup	It specifies a behavior on mouseup events.
ng-non-bindable	It specifies that no data binding can happen in this element, or it's children.
ng-open	It specifies the open attribute of an element.
ng-options	It specifies <options> in a <select> list.
ng-paste	It specifies a behavior on paste events.
ng-pluralize	It specifies a message to display according to en-us localization rules.
ng-readonly	It specifies the readonly attribute of an element.
ng-repeat	It defines a template for each data in a collection.
ng-required	It specifies the required attribute of an element.

ng-selected	It specifies the selected attribute of an element.
ng-show	It shows or hides html elements.
ng-src	It specifies the src attribute for the <img> element.
ng-srcset	It specifies the srcset attribute for the <img> element.
ng-style	It specifies the style attribute for an element.
ng-submit	It specifies expressions to run on onsubmit events.
ng-switch	It specifies a condition that will be used to show/hide child elements.
ng-transclude	It specifies a point to insert transcluded elements.
ng-value	It specifies the value of an input element.

How to add directives

Example:

```
<!DOCTYPE html>
<html>
<script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.4.8/angular.min.js">
</script>
<body>

<div ng-app="myApp" w3-test-directive></div>
<script>
var app = angular.module("myApp", []);
app.directive("w3TestDirective", function() {
  return {
    template : "This is a directive constructor. "
  };
});
</script>
</body>
</html>
```

Modules and controllers in file

In AngularJS applications, you can put the module and the controllers in JavaScript files.

In this example, "myApp.js" contains an application module definition, while "myCtrl.js" contains the controller:

Example:

```
<!DOCTYPE html>
<html>
<script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.4.8/angular.min.js">
</script>
<body>
<script src="myApp.js"></script>
<script src="myCtrl.js"></script>
</body>
</html>
```

Here "myApp.js" contains:

```
app.controller("myCtrl", function($scope) {
  $scope.firstName  = "Ishaan";
  $scope.lastName= "Tamhankar";
});
```

Here "myCtrl.js" contains:

```
<div ng-app="myApp" ng-controller="myCtrl">
{{ firstName + " " + lastName }}
</div>
```

This example can also be written as:

```
<!DOCTYPE html>
<html>
<body>
<script  src="http://ajax.googleapis.com/ajax/libs/angularjs/1.4.8/angular.min.js"></
script>
<div ng-app="myApp" ng-controller="myCtrl">
```

```
{{ firstName + " " + lastName }}
</div>
<script>
var app = angular.module("myApp", []);
app.controller("myCtrl", function($scope) {
  $scope.firstName = "Ishaan";
  $scope.lastName = "Tamhankar";
});
</script>
</body>
</html>
```

5.2.3 AngularJS Forms

AngularJS facilitates you to create a form enriches with data binding and validation of input controls.

Input controls are ways for a user to enter data. A form is a collection of controls for the purpose of grouping related controls together.

Following are the input controls used in AngularJS forms:

- input elements
- select elements
- button elements
- textarea elements

5.2.3 Forms(Events, Data Validation, Controller)

AngularJS provides multiple events that can be associated with the HTML controls. These events are associated with the different HTML input elements.

Following is a list of events supported in AngularJS:

- ng-click
- ng-dbl-click
- ng-mousedown
- ng-mouseup
- ng-mouseenter

- ng-mouseleave
- ng-mousemove
- ng-mouseover
- ng-keydown
- ng-keyup
- ng-keypress
- ng-change

Data Binding

ng-model directive is used to provide data binding.

Let's take an example where ng-model directive binds the input controller to the rest of your application

See this example:

```
<!DOCTYPE html>
<html lang="en">
<script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.4.8/angular.min.js">
</script>
<body>
<div ng-app="myApp" ng-controller="formCtrl">
 <form>
  First Name: <input type="text" ng-model="firstname">
 </form>
</div>
<script>
var app = angular.module('myApp', []);
app.controller('formCtrl', function($scope) {
  $scope.firstname = "Ajeet";
});
</script>
</body>
</html>
```

AngularJS form example

```
<!DOCTYPE html>
<html>
<head>
<title>Angular JS Forms</title>
<script src = "http://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js">
</script>

<style>
table, th , td {
  border: 1px solid grey;
  border-collapse: collapse;
  padding: 5px;
}

table tr:nth-child(odd) {
  background-color: lightpink;
}

table tr:nth-child(even) {
  background-color: lightyellow;
}
</style>
</head>
<body>

<h2>AngularJS Application</h2>
<div ng-app = "mainApp" ng-controller = "studentController">

<form name = "studentForm" novalidate>
<table border = "0">
<tr>
<td>Enter first name:</td>
<td><input name = "firstname" type = "text" ng-model = "firstName" required>
<span style = "color:red" ng-show = "studentForm.firstname.$dirty &&
studentForm.firstname.$invalid">
```

```
<span ng-show = "studentForm.firstname.$error.required">First Name is
required.
</span>
  </span>
 </td>
</tr>

<tr>
 <td>Enter last name: </td>
 <td><input name = "lastname"  type = "text" ng-model = "lastName" required>

   <span style = "color:red" ng-show = "studentForm.lastname.$dirty &&
   studentForm.lastname.$invalid">

   <span ng-show = "studentForm.lastname.$error.required">Last Name is
   required.</span>

   </span>
 </td>
</tr>
<tr>
   <td>Email: </td>
   <td><input name = "email" type = "email" ng-model = "email" length =
   "100" required>
   <span style = "color:red" ng-show = "studentForm.email.$dirty &&
   studentForm.email.$invalid">
   <span ng-show = "studentForm.email.$error.required">Email is
   required.</span>
   <span ng-show = "studentForm.email.$error.email">Invalid email
   address.</span>
   </span>
 </td>
</tr>
<tr>
 <td>
   <button ng-click = "reset()">Reset</button>
```

```
            </td>
            <td>
              <button ng-disabled = "studentForm.firstname.$dirty &&
                studentForm.firstname.$invalid || studentForm.lastname.$dirty &&
                studentForm.lastname.$invalid || studentForm.email.$dirty &&
                studentForm.email.$invalid" ng-click="submit()">Submit</button>
            </td>
          </tr>
        </table>
      </form>
    </div>
    <script>
      var mainApp = angular.module("mainApp", []);
      mainApp.controller('studentController', function($scope) {
        $scope.reset = function(){
          $scope.firstName = "Ritu";
          $scope.lastName = "Bhatiya";
          $scope.email = "ritubhatiya@gmail.com";
        }
        $scope.reset();
      });
    </script>
  </body>
</html>
```

AngularJS Form Validation

AngularJS provides client-side form validation. It checks the state of the form and input fields (input, textarea, select), and lets you notify the user about the current state.

It also holds the information about whether the input fields have been touched, or modified, or not.

Following directives are generally used to track errors in an AngularJS form:

- **$dirty** - states that value has been changed.
- **$invalid** - states that value entered is invalid.
- **$error** - states the exact error.

AngularJS Form Validation Example

```
<!DOCTYPE html>
<html>
<head>
<title>Angular JS Forms</title>
<script src = "http://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js">
</script>

<style>
table, th , td {
  border: 1px solid grey;
  border-collapse: collapse;
  padding: 5px;
}

table tr:nth-child(odd) {
  background-color: lightpink;
}

table tr:nth-child(even) {
  background-color: lightyellow;
}
</style>

</head>
<body>

<h2>AngularJS Sample Application</h2>
<div ng-app = "mainApp" ng-controller = "studentController">

<form name = "studentForm" novalidate>
<table border = "0">
<tr>
<td>Enter first name:</td>
<td><input name = "firstname" type = "text" ng-model = "firstName" required>
```

```
            <span style = "color:red" ng-
show = "studentForm.firstname.$dirty && studentForm.firstname.$invalid">
              <span ng-
show = "studentForm.firstname.$error.required">First Name is required.</span>
            </span>
           </td>
          </tr>
          <tr>
           <td>Enter last name: </td>
           <td><input name = "lastname"  type = "text" ng-model = "lastName" required>
            <span style = "color:red" ng-
show = "studentForm.lastname.$dirty && studentForm.lastname.$invalid">
              <span ng-
show = "studentForm.lastname.$error.required">Last Name is required.</span>
            </span>
           </td>
          </tr>

          <tr>
           <td>Email: </td><td><input name = "email" type = "email" ng-
model = "email" length = "100" required>
            <span style = "color:red" ng-
show = "studentForm.email.$dirty && studentForm.email.$invalid">
              <span ng-
show = "studentForm.email.$error.required">Email is required.</span>
              <span ng-
show = "studentForm.email.$error.email">Invalid email address.</span>
            </span>
           </td>
          </tr>
          <tr>
           <td>
            <button ng-click = "reset()">Reset</button>
           </td>
           <td>
            <button ng-disabled = "studentForm.firstname.$dirty &&
             studentForm.firstname.$invalid || studentForm.lastname.$dirty &&
```

```
            studentForm.lastname.$invalid || studentForm.email.$dirty &&
            studentForm.email.$invalid" ng-click="submit()">Submit</button>
          </td>
        </tr>
      </table>
      </form>
    </div>
    <script>
    var mainApp = angular.module("mainApp", []);
    mainApp.controller('studentController', function($scope) {
      $scope.reset = function(){
        $scope.firstName = "ishaan";
        $scope.lastName = "tamhankar";
        $scope.email = "ishaantamhankar@gmail.com";
      }
       $scope.reset();
    });
    </script>
    </body>
</html>
```

Important Questions:

1. What is Single Page Application?
2. What is $RoutesProvider Service?
3. What is HTML DOM Directives?
4. Explain form with Example.
5. Explain AngularJs form Validation.