# Course: 501

# Advanced Web Designing

# Unit-1: Concepts of NoSQL: MongoDB

**Unit-1: Concepts of NoSQL: MongoDB**

1.1 Concepts of NoSQL. Advantages and features.

1.1.1 MongoDB Datatypes (String, Integer, Boolean, Double, Arrays, Objects)

1.1.2 Database creation and dropping database

1.2 Create and Drop collections

1.3 CRUD operations (Insert, update, delete, find, Query and Projection operators)

1.4 Operators (Projection, update, limit (), sort ()) and

# Aggregation commands

## ➢ History of MongoDb

- MongoDB is a **NoSQL database** that was developed by **10gen**, a company founded by **Dwight Merriman and Eliot Horowitz** in 2007.

**Dwight Merriman**

**Eliot Horowitz**

- The **first version** of MongoDB was released in **2009**, and it quickly gained popularity among developers due to its ease of use, **scalability**, and **flexibility**.

- In 2013, **10gen changed its name to MongoDB Inc**. to better reflect its focus on the development of the MongoDB database.
- In 2017, MongoDB Inc. went public, and the company has continued to grow and expand its offerings, including the introduction of a **cloud-based database service called MongoDB Atlas**.
-

Today, MongoDB is used by a wide range of companies and organizations,

- The name "**MongoDB**" is **derived** from the word "**humongous**," reflecting the database's ability to store and manage large amounts of data.

## What is MongoDb?

- MongoDB is a popular **open-source**, **NoSQL** database that stores data in a **document-oriented format**.
- Unlike traditional relational databases, which store data in tables, MongoDB stores data as JSON-like documents, making it more flexible and scalable.
- It was developed by MongoDB Inc. and first released in 2009.

## Concepts of NoSQL

- NoSQL, also referred to as “not only SQL”, “nonSQL”, is an approach to database design that enables the storage and querying of data outside the traditional structures found in relational databases.
- 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.

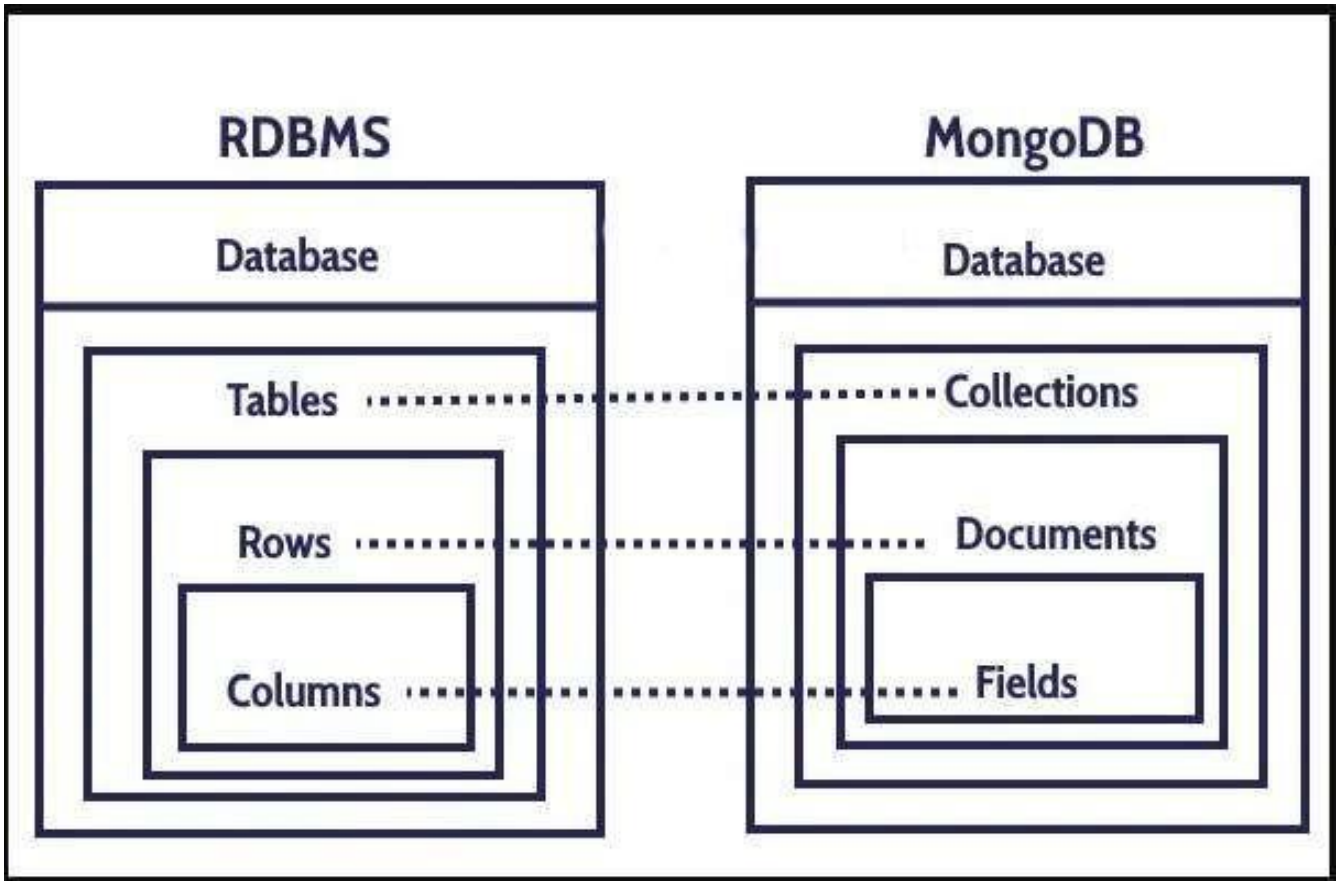
- NoSQL databases are generally classified into four main categories:

1. Document databases: These databases store data as semi-structured documents, such as JSON or XML, and can be queried using documentoriented query languages.
2. Key-value stores: These databases store data as key-value pairs, and are optimized for simple and fast read/write operations.

3. Column-family stores: These databases store data as column families, which are sets of columns that are treated as a single entity. They are optimized for fast and efficient querying of large amounts of data.
4. Graph databases: These databases store data as nodes and edges, and are designed to handle complex relationships between data.

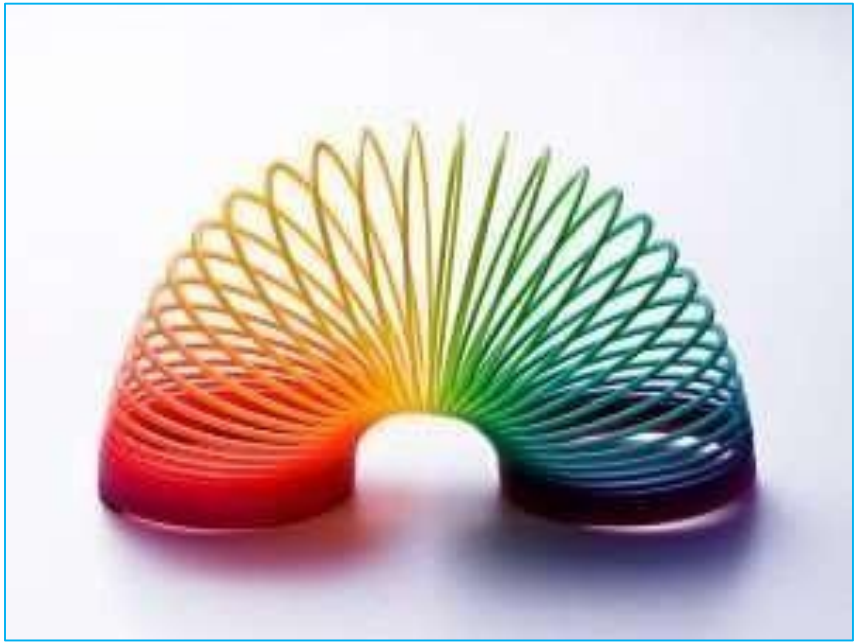
- Advantages ofNoSQL:

- There are many great features inbuilt with MongoDB. As compared to RDBMS, so let’s discuss MongoDB Benefits.



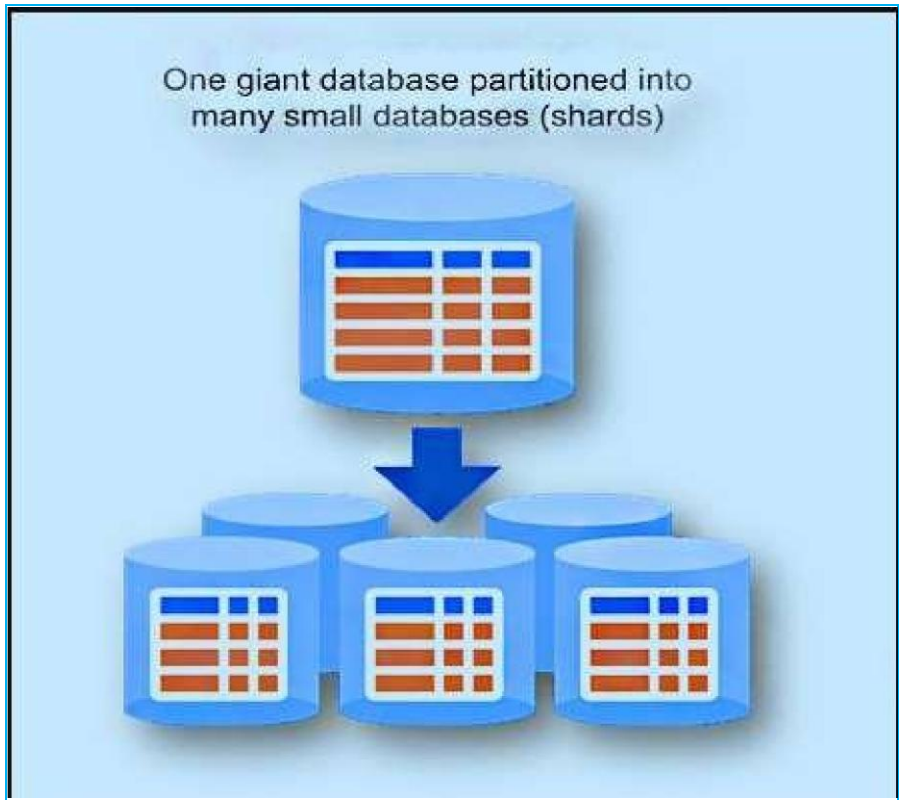
## Flexible Database

- We know that MongoDB is a schema-less database.
- That means we can have any type of data in a separate document.
- This thing gives us flexibility and a freedom to store data of different types.



## Sharding

- We can store a large data by distributing it to several servers connected to the application. If a server cannot handle such a  big data then there will be no failure condition. The term we can use here is “auto -sharding  ”.

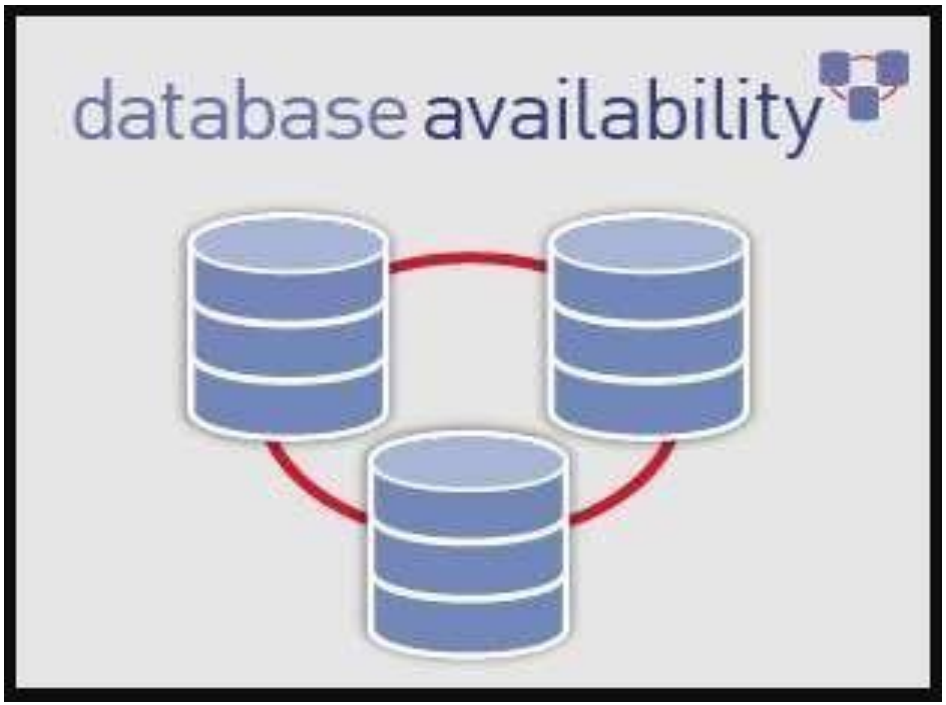

One giant database partitioned into
many small databases (shards)

## High Speed

- MongoDB is a document-oriented database. It is easy to access documents by indexing. Hence, it provides fast query response. The speed of MongoDB is 100 times faster than the relational database.

## High Availability

- MongoDB has features like replication and gridFS. These features help to increase data availability in MongoDB. Hence the performance is very high.



- **Scalability**

• A great advantage of MongoDB is that it is a horizontally scalable database. When you have to handle a large data, you can distribute it to several machines.

- **Easy Environment Setup**

• It is easier to setup MongoDB then RDBMS. It also provides JavaScript client for queries.

- **Full Technical Support**

• MongoDB Inc. provides professional support to its clients. If there is any problem, you can directly reacha MongoDB client support system.

✈ Disadvantages of NoSQL:

## ○ Joins not Supported

- MongoDB doesn't support joins like a relational database. Yet one can use joins functionality by adding by coding it manually. But it may slow execution and affect performance.

## ○ High Memory Usage

- MongoDB stores key names for each value pairs. Also, due to no functionality of joins, there is data redundancy. This results in increasing unnecessary usage of memory.

## Limited Data Size

- You can have document size, not more than 16MB.

- GUI is not available :

• GUI mode tools to access the database are not flexibly available in the market.

- Backup :

• Backup is a great weak point for some NoSQL databases like MongoDB.

MongoDB has no approach for the backup of data in a consistent manner.

- Large document size :

- Some database systems like MongoDB and CouchDB store data in JSON format.
- This means that documents are quite large (BigData, network bandwidth, speed), and having descriptive key names actually hurts since they increase the document size.

- **MongoDB is used in a wide variety of applications and industries, including:**

  - **Web and mobile applications:**

- MongoDB is often used as the primary database for web and mobile applications, where it provides high performance and scalability, as well as support for flexible data models.

  - **E-commerce:**

- MongoDB is used by many e-commerce sites to store product catalogs, customer data, and order information, as well as to provide real-time analytics and personalized recommendations.

- **Social networking:**

- MongoDB is used by social networking sites to store user profiles, activity feeds, and social graphs, as well as to provide real-time analytics and recommendations.

- **Gaming:**

- MongoDB is used in the gaming industry to store user data, game progress, and other game-related information, as well as to provide real-time analytics and recommendations.

- **Financial services:**

- MongoDB is used in the financial services industry to store and analyze large volumes of data, such as transaction data, customer data, and market data.



- **Healthcare:**

- MongoDB is used in the healthcare industry to store and manage patient data, medical records, and other healthcare-related information, as well as to provide real-time analytics and insights.

- **Government:**

- MongoDB is used by many government agencies to store and manage large volumes of data, such as census data, weather data, and traffic data, as well as to provide real-time analytics and insights.

- How to Install MongoDB on Windows

Step 1: Go to the Official MongoDB website

[https://www.mongodb.com/try/download/community-kubernetes-operator]

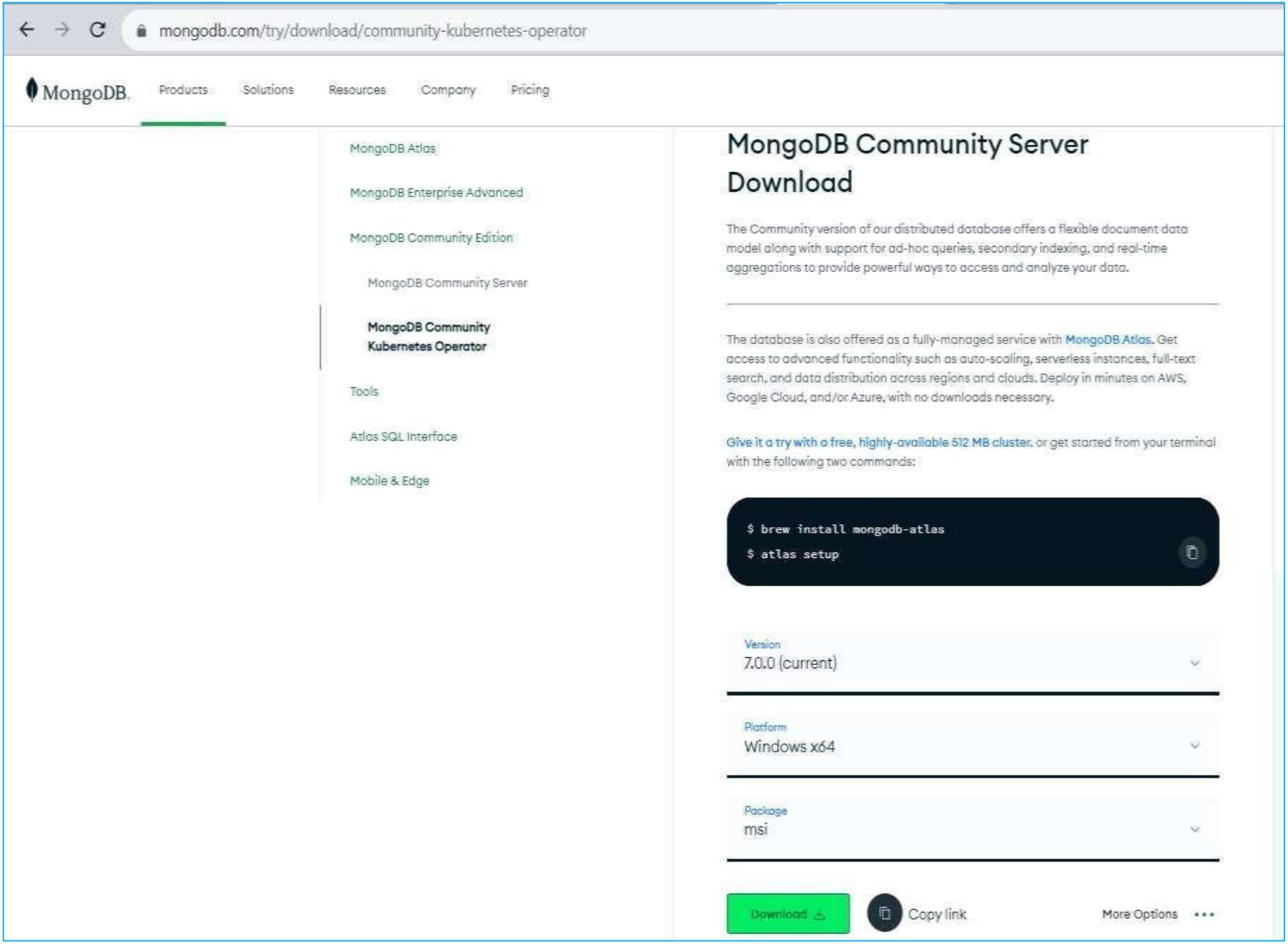
**Step 2:** Navigate to Products > Community Edition

**Step 3:** Select the appropriate installer file from the dropdown menus on the Community Edition page.

- In the version dropdown, select the latest version, 6.0.1(current)
- In the Platform dropdown, select Windows
- In the Package dropdown, select **msi**

Step 4: Click the green "Download" button. Wait for 2-5 minutes for the

file to download. (Depending on your internet speed)


mongodb.com/try/download/community-kubernetes-operator
MongoDB
Products
Solutions
Resources
Company
Pricing
MongoDB Atlas
MongoDB Enterprise Advanced
MongoDB Community Edition
MongoDB Community Server
MongoDB Community Kubernetes Operator
Tools
Atlas SQL Interface
Mobile & Edge
MongoDB Community Server Download
The Community version of our distributed database offers a flexible document data model along with support for ad-hoc queries, secondary indexing, and real-time aggregations to provide powerful ways to access and analyze your data.
The database is also offered as a fully-managed service with MongoDB Atlas. Get access to advanced functionality such as auto-scaling, serverless instances, full-text search, and data distribution across regions and clouds. Deploy in minutes on AWS, Google Cloud, and/or Azure, with no downloads necessary.
Give it a try with a free, highly-available 512 MB cluster. or get started from your terminal with the following two commands:
$ brew install mongodb-atlas
$ atlas setup
Version
7.0.0 (current)
Platform
Windows x64
Package
msi
Download
Copy link
More Options

- After the installer file has been downloaded, it's time to run the installer file.

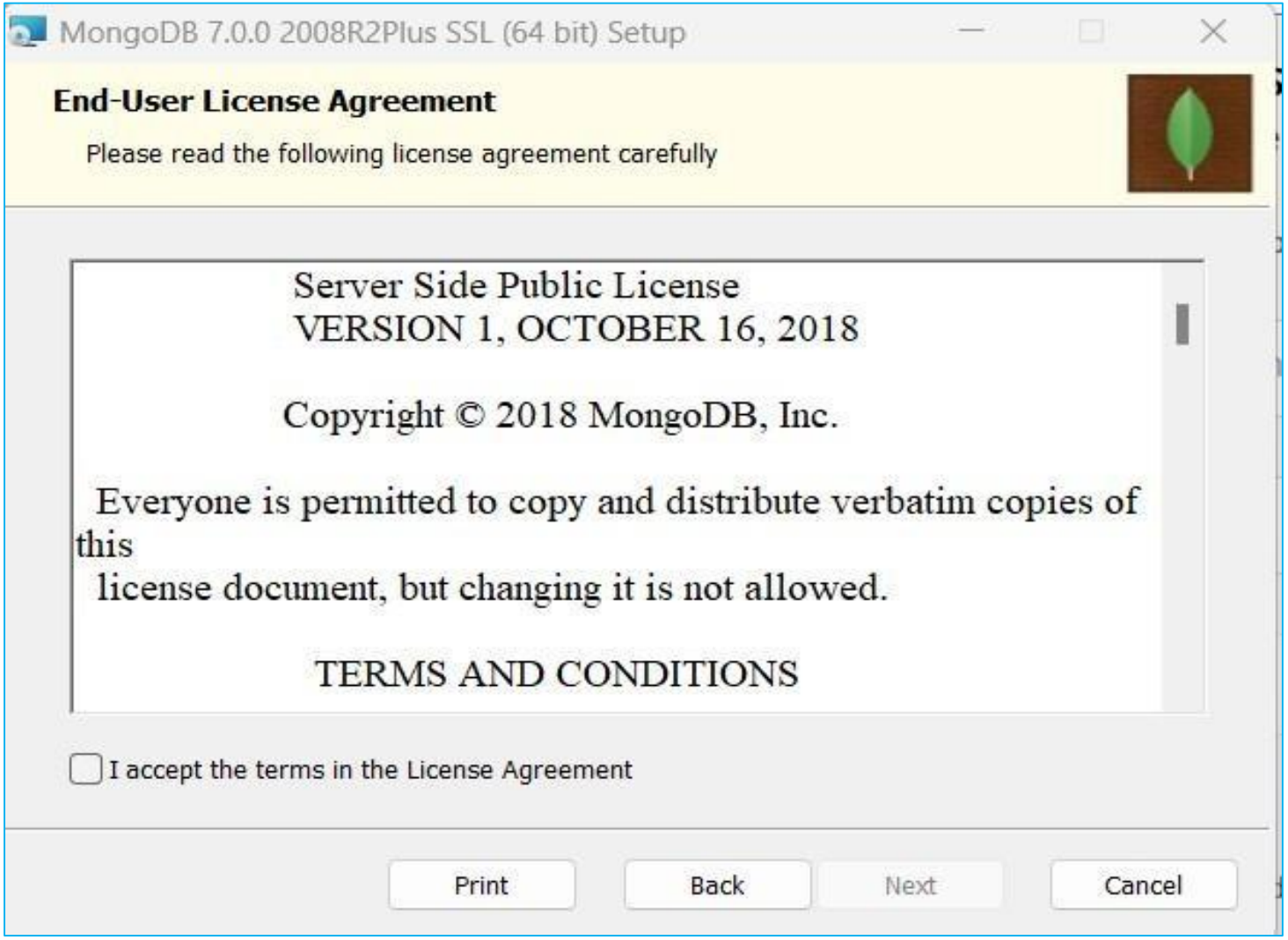
✈ **Procedure**

**Step: 1:** Go to the downloaded directory in your system (by default, it should be in the `Downloads` directory).

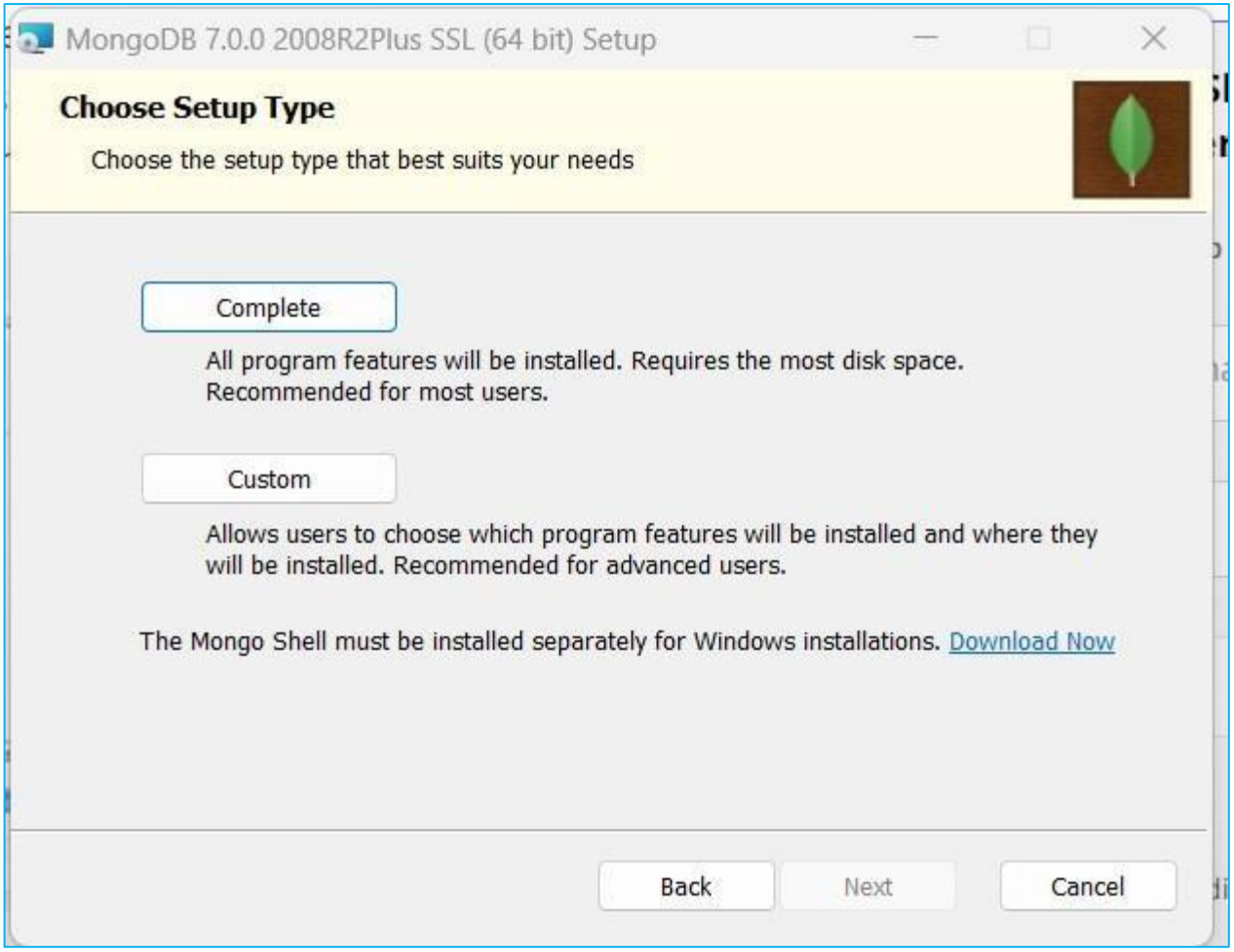
**Step 2:** Double-click on the .msi file. It will open the MongoDB setup windows.

- Step 3: It will open the MongoDB Community Edition installation wizard. This setup wizard guides you through the installation of MongoDB in your system. To continue the process, click "Next."

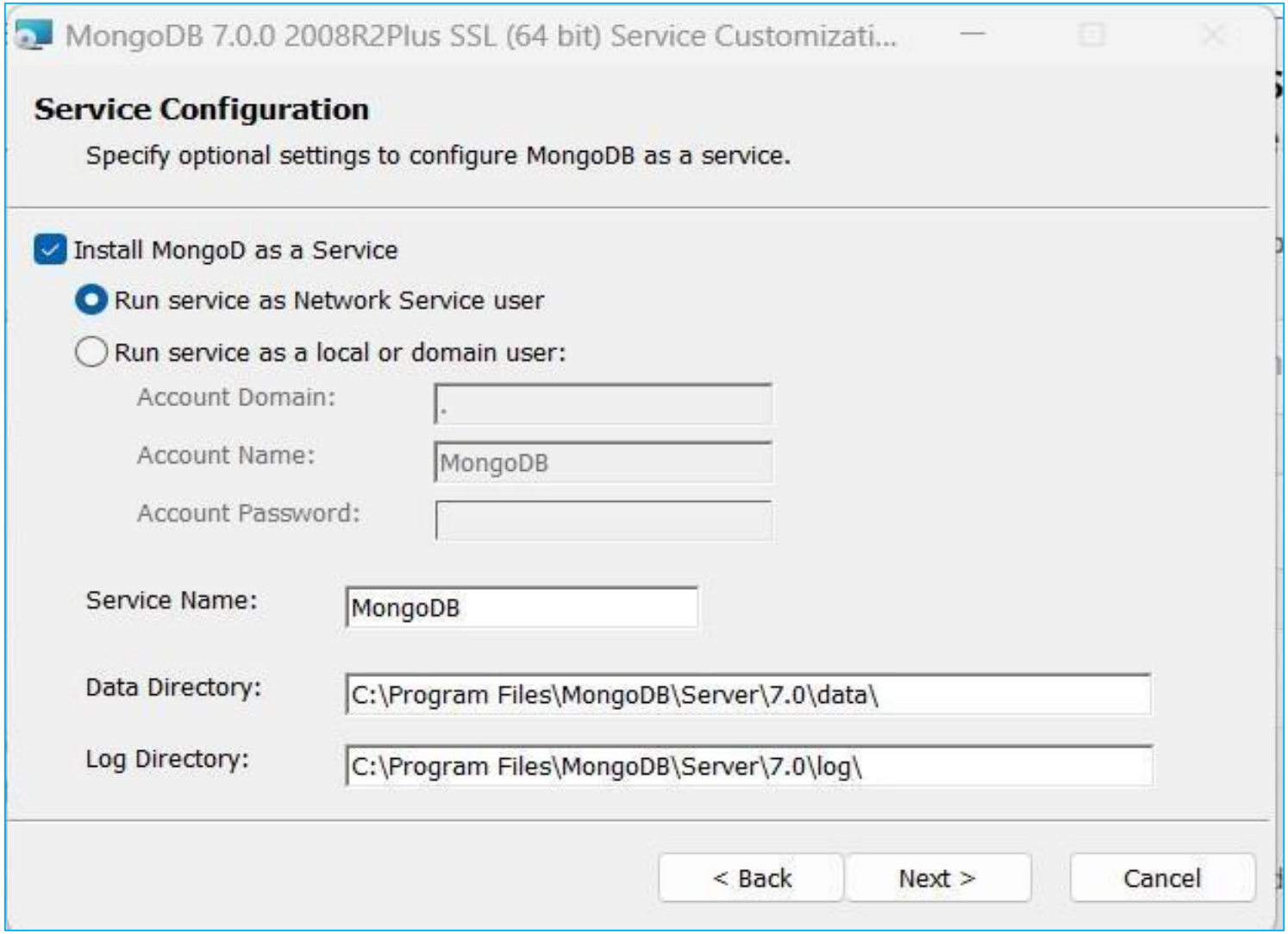
- Step 4: Read the End-User License Agreement, accept the terms and conditions, and then click the "Next" button to continue.



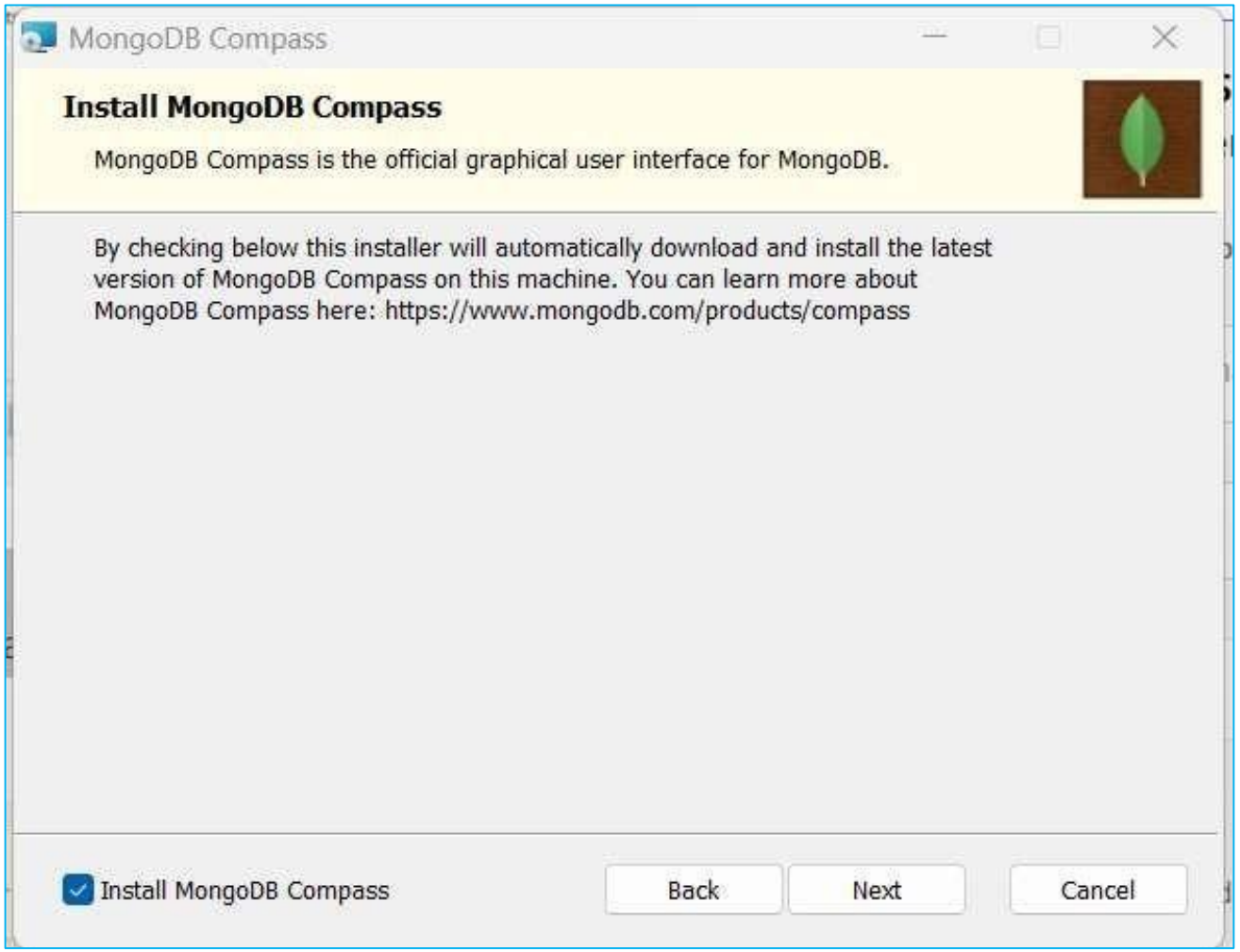
- Step 5: Next, you can choose either the Complete setup or Custom setup type to proceed. But for a beginner, we'drecommend using the Complete setup option. It installs MongoDB inthe default location. Select the Complete setup, and click "Next."



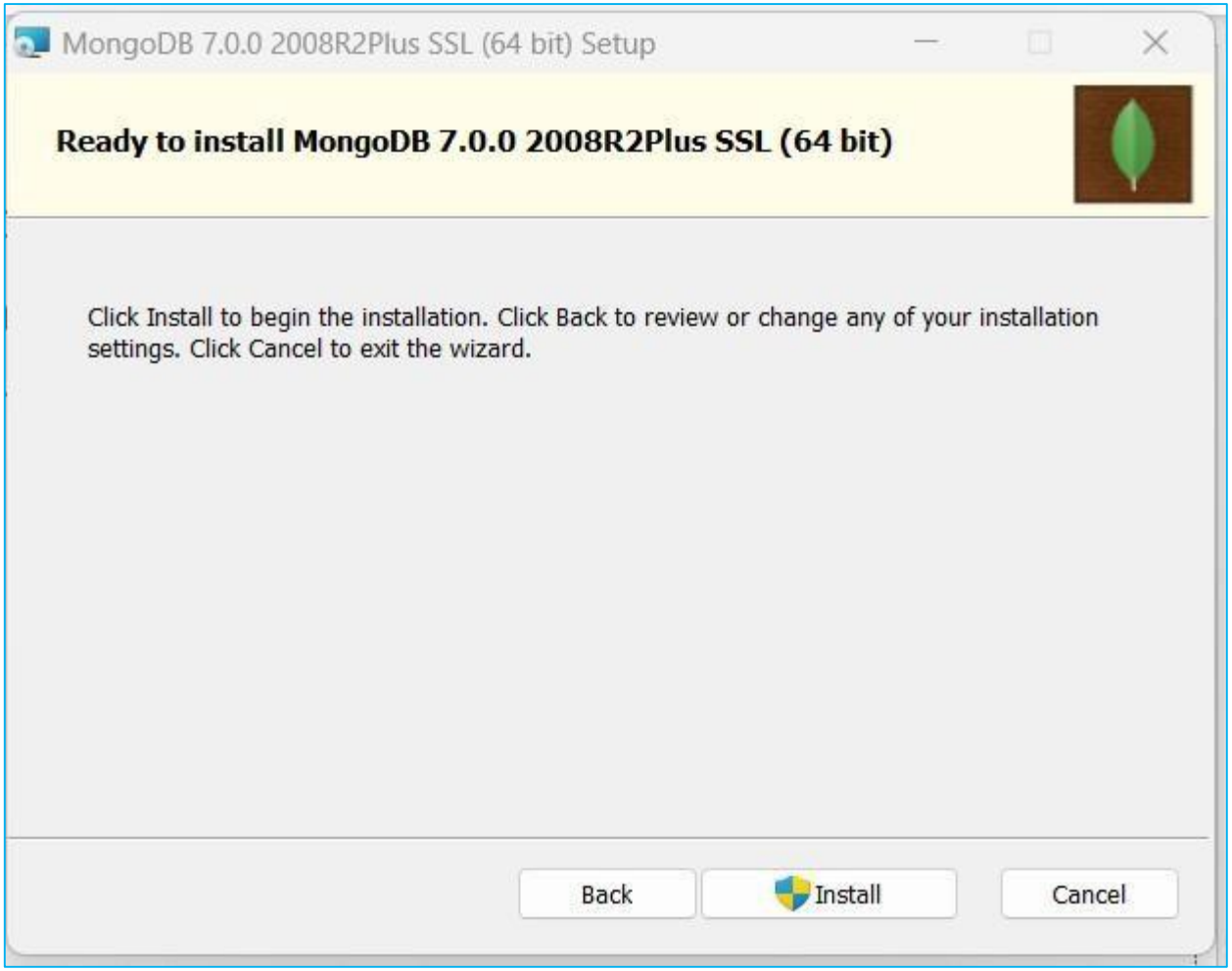
- Step 6: Select the "Install MongoD as a Service" option on the next page. Keep all other parameters as default. Click on the "Next" button.



- Step 7: In the next step, you will get an option to install MongoDB compass. Uncheck it if you don't want MongoDB compass to be installed on your device, and then click the "Next" button.



- Step 8: In the "Ready to install MongoDB" page, click the "Install" button, give administrator access, and wait for the installation to finish. Once installation is complete, you can click on the "Finish" button to finalize your installation.



# How to Install **MongoDB Shell** on Windows

Step 1: Download MongoDB Shell

- To begin with Windows MongoDB Shell Installation process, go to the download page at https://www.mongodb.com/try/download/shell.
- Choose your OS and your desired MongoDB version.
- Click Download.

MongoDB Atlas

MongoDB Enterprise Advanced

MongoDB Community Edition

Tools

**MongoDB Shell**

MongoDB Compass (GUI)

Atlas CLI

Atlas Kubernetes Operator

MongoDB CLI for Cloud Manager and Ops Manager

MongoDB Cluster-to-Cluster Sync

Relational Migrator

MongoDB Database Tools

MongoDB Connector for BI

Atlas SQL Interface

Mobile & Edge

TOOLS

# MongoDB Shell Download

MongoDB Shell is the quickest way to connect to (and work with) MongoDB. Easily query data, configure settings, and execute other actions with this modern, extensible command-line interface – replete with syntax highlighting, intelligent autocomplete, contextual help, and error messages.

**Note:** MongoDB Shell is an open source (Apache 2.0), standalone product developed separately from the MongoDB Server.

Learn more

Version
1.10.5

Platform
Windows 64-bit (8.1+)

Package
zip

Download

Copy link

More Options

## MongoDB Data Types

- MongoDB data types refer to the different types of data that can be stored in a MongoDB database.
- MongoDB supports a wide range of MongoDB data types, including strings, integers, doubles, booleans, dates, arrays, object IDs, regular expressions, and binary data.

✈ **String**

- In MongoDB, the string data type is used to store a sequence of UTF-8 characters.
- Strings are one of the most commonly used MongoDB data types, as they can be used to represent a wide range of text-based data, such as **names, addresses, descriptions, and more**.

✈ **Integer:**

- In MongoDB, the integer data type is used to store an integer value. We can store integer data type in two forms 32 -bit signed integer and 64 – bit signed integer.

✈ **Double**

- The Double data type in MongoDB is used to store floating-point numbers that require higher precision than the standard 32-bit float data type.
- Double data types are used to represent decimal values and are commonly used in financial applications or scientific calculations.

✈ **Boolean**

- The boolean data type in MongoDB represents a logical value that can be either true or false.

✈ **Arrays**

- This type is used to store arrays or list or multiple values into one key.



✈ **Object**

- In MongoDB, the Object data type is used to represent complex and nested data structures, such as documents within a collection. The Object data type is also known as the BSON document data type, where BSON stands for Binary JSON.

✈ **Null**

- This type is used to store a Null value.

✈ **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

- In MongoDB, the binary data type is used to store binary data as a sequence of bytes.
- This data type is commonly used to store **images, videos, audio files, and other non-textual data**.

## ✈ Regular Expression

- In MongoDB, the regular expression (regex) data type is used to store and search for text patterns in strings.

- Regular expressions are a powerful tool for string manipulation and pattern matching, and MongoDB provides several operators for working with them.



✈ **Date**

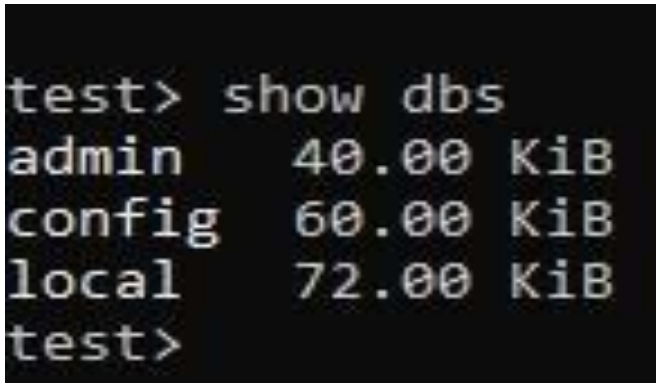
- This datatype is used to store the current date or time in UNIX time format.
- You can specify your own date time by creating object of Date and passing day, month, year into it.

✈ **Code**

- This datatype is used to store JavaScript code into the document.

## ✈ Database:

- The MongoDB database is a container for collections and it can store one or more collections.
- It is not necessary to create a database before you work on it.
- The **show dbs** command gives the list of all the databases.



## Creating a 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**

```
use DATABASE_NAME
```

- **Example**

```
>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.bca.insert({name:"Alex",age:21,state:"Gujarat"})
>show dbs
local      0.78125GB
mydb       0.23012GB
test       0.23012GB
```

AWD

# The dropDatabase() Method

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

```
db.dropDatabase()
```

## ✈ Collection:

- It is used to store a varied number of documents inside the database.
- As MongoDB is a Schema-free database, it can store the documents that are not the same in structure. Also, there is no need to define the columns and their datatype.

## There are 2 ways to create a collection.

### Method 1

- You can create a collection using the **createCollection**() database method.
- **Example**

```
db.createCollection(“student")
```

### Method 2

- You can also create a collection during the insert process.

- **Example**

```
db. student.insertOne({“name": "Alex",age:21,state:“Gujarat"})
```

## Display Collections(Tables)

- **Example**

```
show collections
```

## Exit The Mongosh Terminal

- **Example**

Ctrl + C  [ Pressed Two Times ]

**or**

quit()

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

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

- MongoDB stores data records as documents that are stored together in collections and the database stores one or more collections of documents.

❑ **Document:**

- A document is a basic unit of storing data into the database.
- A single record of a collection is also known as a document.
- Basically, It is a structure that compromises key & value pairs which is similar to the JSON objects. Documents have a great ability to store complex data.
- **For example:**

```
> var mydocument = {name: "gfg", country: "India", age: 21, status: 'A'}
>
```

- Here, the name, country, age and status are fields, and gfg, India, 21, A are their values.

# Insert Documents

- There are **2 methods** to insert documents into a MongoDB database.

## ✈insertOne()

- To insert a single document, use the insertOne() method.
- This method inserts a single object into the database.

- **Example**

```
db.posts.insertOne({
  title: "Post Title 1",
  body: "Body of post.",
  category: "News",
  likes: 1,
  tags: ["news", "events"],
  date: Date()
})
```

```
{
  acknowledged: true,
  insertedId: ObjectId("62c350dc07d768a33fdfe9b0")
}
Atlas atlas-8iy36m-shard-0 [primary] blog>
```

**Note:** If you try to insert documents into a collection that does not exist, MongoDB will create the collection automatically.

**Note:** When typing in the shell, after opening an object with curly braces "{" you can press enter to start a new line in the editor without executing the command.

The command will execute when you press enter after closing the braces.

## insertMany ()

- To insert multiple documents at once use the **insertMany ()** method.
- This method inserts an array of objects into the database.

- **Example**

```
db.posts.insertMany([
  {
    title: "Post Title 2",
    body: "Body of post.",
    category: "Event",
    likes: 2,
    tags: ["news", "events"],
    date: Date()
  },
  {
    title: "Post Title 3",
    body: "Body of post.",
    category: "Technology",
    likes: 3,
    tags: ["news", "events"],
    date: Date()
```

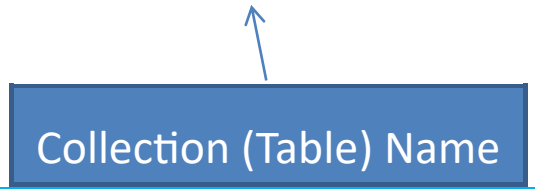
## Find Data [ Dipslay Data from Collection(Table) ]

- There are 2 methods to find and select data from a MongoDB collection, **find()** and **findOne()**.

### ✈ find()

- To select data from a collection in MongoDB, we can use the **find()** method.
- This method accepts a query object. If left empty, all documents will be returned.

- **Example** db.posts.find()


Collection (Table) Name

## ✈ findOne()

- To select only one document, we can use the findOne() method.
- This method accepts a query object. If left empty, it will return the first document it finds.
- **Note:** This method only returns the first match it finds.
- **Example**

```
db.posts.findOne()
```

Collection (Table) Name

## Delete Documents

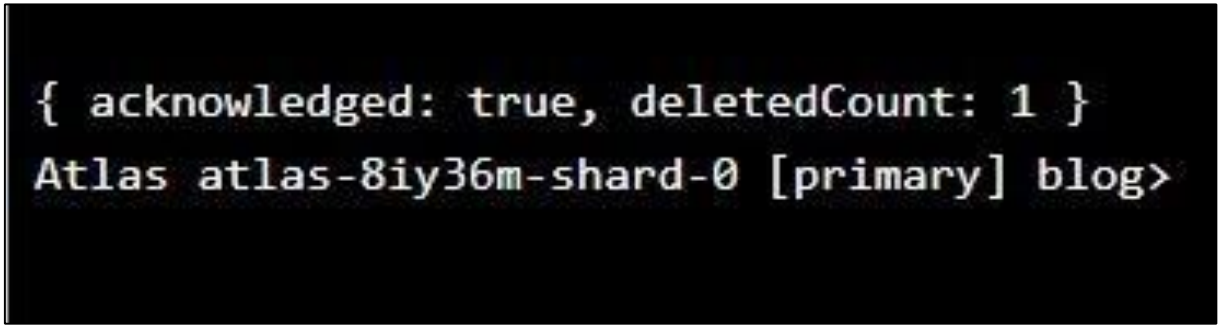
- We can delete documents by using the methods **deleteOne()** or **deleteMany()**.
- These methods accept a query object. The matching documents will be deleted.

## ✈ deleteOne()

- The deleteOne() method will delete the first document that matches the query provided.

## ○ Example

```
db.posts.deleteOne({ title: "Post Title 5" })
```



## ✈ deleteMany()

- The deleteMany() method will delete all documents that match the query provided.

## ○ Example

```
db.posts.deleteMany({ category: "Technology " })
```

```
{ acknowledged: true, deletedCount: 1 }
Atlas atlas-8iy36m-shard-0 [primary] blog>
```

## Update Document

- To update an existing document we can use the **updateOne()** or **updateMany()** methods.
- The first parameter is a query object to define which document or documents should be updated.
- The second parameter is an object defining the updated data.

### updateOne()

- The updateOne() method will update the first document that is found matching the provided query.

## Syntax:

```
db.COLLECTION_NAME.update({SELECTION_CRITERIA}, {$set:
{UPDATED_DATA}}, {
     upsert: <boolean>,
     multi: <boolean>,
     writeConcern: <document>,
     collation: <document>,
     arrayFilters: [ <filterdocument1>, ... ],
     hint:  <document|string>
   })
```

## Parameters:

- The first parameter is the Older value in the form of Documents. Documents are a structure created of file and value pairs, similar to JSON objects.
- The second parameter must contain a **$set** keyword to update the following specify document value.
- The third parameter is optional.

- **Example [First we can see all records(documents) from table]**

```
db.posts.find( { title: "Post Title 1" } )
```

```
[
  {
    _id: ObjectId("62c350dc07d768a33fdfe9b0"),
    title: 'Post Title 1',
    body: 'Body of post.',
    category: 'News',
    likes: 1,
    tags: [ 'news', 'events' ],
    date: 'Mon Jul 04 2022 15:43:08 GMT-0500 (Central Daylight Time)'
  }
]
Atlas atlas-8iy36m-shard-0 [primary] blog>
```

## MongoDB updateOne() method

- This methods updates a single document which matches the given filter.

```
db.posts.updateOne( { title: "Post Title 1" }, { $set: { likes: 2 } } )
```

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
Atlas atlas-8iy36m-shard-0 [primary] blog>
```

## Examples:

- In the following examples, we are working with:
- **Database:** gfg
- **Collections:** student
- **Document:** Three documents contains name and the age of the students

- **Update the name of the document whose name key has avi value to hello world.**

```
> use gfg
switched to db gfg
> db.student.find().pretty()
{ "_id" : ObjectId("5f8dd0790cf217478ba9355d"), "name" : "avi", "age" : 12 }
{
        "_id" : ObjectId("5f8dd08a0cf217478ba9355e"),
        "name" : "payal",
        "age" : 15
}
{
        "_id" : ObjectId("5f8dd0940cf217478ba9355f"),
        "name" : "prachi",
        "age" : 17
}
>
```

- Here, the **first parameter** is the document whose value to be changed **{name:"avi"}** and the **second parameter** is **set keyword** means to **set(update)** the following matched key value with the older key value.

```
db.student.update({name:"avi"},{$set:{name:"helloword"}})
```

- **Note:** The value of the key must be of the same datatype that was defined in the collection.

## Output

```
> db.student.update({name:"avi"}, {$set:{name:"helloword"}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.student.find().pretty()
{
        "_id" : ObjectId("5f8dd0790cf217478ba9355d"),
        "name" : "helloword",
        "age" : 12
}
{
        "_id" : ObjectId("5f8dd08a0cf217478ba9355e"),
        "name" : "payal",
        "age" : 15
}
{
        "_id" : ObjectId("5f8dd0940cf217478ba9355f"),
        "name" : "prachi",
        "age" : 17
}
>
```

## Example

- Update the age of the document whose name is prachi to 20.

```
db.student.update({name:"prachi"},{$set:{age:20}}
```

- Here, the first parameter is the document whose value to be changed {name:”prachi”} and the second parameter is set keyword means to set(update) the value of the age field to 20.

## Output

```
> db.student.update({name:"prachi"}, {$set:{age:20}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.student.find().pretty()
{
        "_id" : ObjectId("5f8dd0790cf217478ba9355d"),
        "name" : "helloword",
        "age" : 12
}
{
        "_id" : ObjectId("5f8dd08a0cf217478ba9355e"),
        "name" : "payal",
        "age" : 15
}
{
        "_id" : ObjectId("5f8dd0940cf217478ba9355f"),
        "name" : "prachi",
        "age" : 20
}
>
```

## ✈ updateMany()

- The updateMany() method will update all documents that match the provided query.
- When you update your document, the value of the _id field remains unchanged.
- This method can also add new fields in the document. Specify an empty document(**{}**) in the selection criteria to update all collection documents.

## Examples:

- In the following examples, we are working with:
- **Database:** gfg
- **Collection:** student
- **Document:** Three documents contains name and age of the students

```
> use gfg
switched to db gfg
> db.student.find().pretty()
{
        "_id" : ObjectId("600ebc010cf217478ba93570"),
        "name" : "aaksh",
        "age" : 15
}
{
        "_id" : ObjectId("600ebc010cf217478ba93571"),
        "name" : "nikhil",
        "age" : 18
}
{
        "_id" : ObjectId("600ebc010cf217478ba93572"),
        "name" : "vishal",
        "age" : 18
}
>
```

## Update single document

```
db.student.updateMany({name: "aaksh"}, {$set:{age: 20}})
```

```
> db.student.updateMany({name: "aaksh"}, {$set:{age: 20}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.student.find().pretty()
{
        "_id" : ObjectId("600ebc010cf217478ba93570"),
        "name" : "aaksh",
        "age" : 20
}
{
        "_id" : ObjectId("600ebc010cf217478ba93571"),
        "name" : "nikhil",
        "age" : 18
}
{
        "_id" : ObjectId("600ebc010cf217478ba93572"),
        "name" : "vishal",
        "age" : 18
}
>
```

- **Update multiple documents**

```
db.student.updateMany({age:18},{$set:{eligible:"true"}})
```

- Here, we update all the matched documents whose age is 18 to eligible: true

## Output

```
> db.student.updateMany({age:18},{$set:{eligible:"true"}})
{ "acknowledged" : true, "matchedCount" : 2, "modifiedCount" : 2 }
> db.student.find().pretty()
{
        "_id" : ObjectId("600ebc010cf217478ba93570"),
        "name" : "aaksh",
        "age" : 20
}
{
        "_id" : ObjectId("600ebc010cf217478ba93571"),
        "name" : "nikhil",
        "age" : 18,
        "eligible" : "true"
}
{
        "_id" : ObjectId("600ebc010cf217478ba93572"),
        "name" : "vishal",
        "age" : 18,
        "eligible" : "true"
}
>
```

## MongoDB Comparison Operators

- $eq • $gt
- $gte • $in • $lt

- $lte • $ne
- $nin

- **$eq**

- The $eq specifies the equality condition. It matches documents where the value of a field equals the specified value.

**Syntax:**

```
{ <field> : { $eq: <value> } }
```

**Example:**

```
db.books.find ( { price: { $eq: 300 } } )
```

- The above example queries the books collection to select all documents where

## $gt

- The $gt chooses a document where the value of the field is greater than the specified value.

**Syntax:**

```
{ field: { $gt: value } }
```

**Example:**

```
db.books.find ( { price: { $gt: 200 } } )
```

- **$gte**

- The $gte choose the documents where the field value is greater than or equal to a specified value.

**Syntax:**

```
{ field: { $gte: value } }
```

**Example:**

```
db.books.find ( { price: { $gte: 250 } } )
```

- **$in**

- The $in operator choose the documents where the value of a field equals any value in the specified array.

**Syntax:**

```
{ filed: { $in: [ <value1>, <value2>, .....] } }
```

**Example:**

```
db.books.find( { price: { $in: [100, 200] } } )
```

## $lt

- The $lt operator chooses the documents where the value of the field is less than the specified value.

**Syntax:**

```
{ field: { $lt: value } }
```

**Example:**

```
db.books.find ( { price: { $lt: 20 } } )
```

- **$lte**

- The $lte operator chooses the documents where the field value is less than or equal to a specified value.

**Syntax:**

```
{ field: { $lte: value } }
```

**Example:**

```
db.books.find ( { price: { $lte: 250 } } )
```

## $ne

- The $ne operator chooses the documents where the field value is not equal to the specified value.

**Syntax:**

```
{ <field>: { $ne: <value> } }
```

**Example:**

```
db.books.find ( { price: { $ne: 500 } } )
```

## $nin

- The $nin operator chooses the documents where the field value is not in the specified array or does not exist.

**Syntax:**

```
{ field : { $nin: [ <value1>, <value2>, .... ] } }
```

**Example:**

```
db.books.find ( { price: { $nin: [ 50, 150, 200 ] } } )
```

## What is MongoDB Projection?

- MongoDB Projection is a special feature allowing you to select only the necessary data rather than selecting the whole set of data from the document.
- For Example, If a Document contains 10 fields and only 5 fields are to be shown the same can be achieved using the Projections.

## This will enable us to:

- Project concise yet transparent data
- Filter data without impacting the overall database performance

## MongoDB Projection Operators

- MongoDB projection method positively impacts database performance as it reduces the workload of the find query when trying to retrieve specific data from a document, minimizing resource usage.
- To enhance the querying and reduce the workload, multiple operators can be used within a projection query like the ones below:

## Operators

- $
- $elemMatch

- $slice
- $meta

## $ Operator

- The $ operator limits the contents of an array from the query results to contain only the first element matching the query document.

**Syntax:**

```
db.books.find( { <array>: <value> ... },
    { "<array>.$": 1 } )
db.books.find( { <array.field>: <value> ...},
    { "<array>.$": 1 } )
```

## $elemMatch

- The content of the array field made limited using this operator from the query result to contain only the first element matching the element $elemMatch condition.

**Syntax:**

```
db.library.find( { bookcode: "63109" },
{ students: { $elemMatch: { roll: 102 } } } )
```

## $meta

- The meta operator returns the result for each matching document where the metadata associated with the query.

**Syntax:**

```
{ $meta: <metaDataKeyword> }
```

**Example:**

```
db.books.find(
<query>,
{ score: { $meta: "textScore" } }
```

## $slice

- It controls the number of values in an array that a query returns.

**Syntax:**

```
db.books.find( { field: value }, { array: {$slice: count } } );
```

**Example:**

```
db.books.find( {}, { comments: { $slice: [ 200, 100 ] } } )
```

## Update Operators

| Operator | Description |
|---|---|
| **$currentDate** | This operator is used to set the value of a field to current date, either as a Date or a Timestamp. |
| **$inc** | This operator is used to increment the value of the field by the specified amount. |
| **$min** | This operator is used only to update the field if the specified value is less than the existing field value |
| **$max** | This operator is used only to update the field if the specified value is greater than the existing field value. |
| **$mul** | This operator is used to multiply the value of the field by the specified amount. |

| | |
|---|---|
| **$rename** | This operator is used to rename a field. |
| **$setOnInsert** | This operator is used to set the value of a field if an update results in an insert of a document. It has no effect on update operations that modify existing documents. |

- **Increment the value of the field using $inc operator:**

```
db.Employee.update({"name.first": "Sumit"},
{$inc: {"personalDetails.salary": 3000}})
```

- **Comparing values (or numbers) using $max operator:**

```
db.Employee.update({"name.first": "Sumit"},
{$max: {"personalDetails.salary": 40000}})
```

- **Multiplying the value of a field using $mul operator:**

```
db.Employee.update({"name.first": "Sumit"},
{$mul: {"personalDetails.salary": 2}})
```

- **Updating the value of date field using $currentDate operator:**

```
db.Employee.updateOne({"name.first": "Om"},
{$currentDate: {joiningDate: true}})
```

- **Comparing values (or numbers) using $min operator:**

```
db.Employee.update({"name.first": "Sumit"},
{$min: {"personalDetails.salary": 5000}})
```

- **Renaming a field using $rename operator:**

```
db.Employee.update({"name.first": "Om"},
{$rename: {"department": "unit"}})
```

## MongoDB limit() Method

- In MongoDB, limit() method is used to limit the fields of document that you want to show. Sometimes, you have a lot of fields in collection of your database and have to retrieve only 1 or 2. In such case, limit() method is used.
- The MongoDB limit() method is used with find() method.

**Syntax:**

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

## Example

```
db.tybca.find().limit(1)
```

## skip() method

- In MongoDB, skip() method is used to skip the document. It is used with find() and limit() methods.

Syntax

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

- Execute the following query to retrieve only one document and skip 2 documents.

## Example

```
db.tybca.find().limit(1).skip(2)
```

## MongoDB sort() method

- In MongoDB, sort() method is used to sort the documents in the collection. This method accepts a document containing list of fields along with their sorting order.

## The sorting order is specified as 1 or -1.

- 1 is used for ascending order sorting.
- -1 is used for descending order sorting.

**Syntax:**

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

## Example

- Execute the following query to display the documents in descending order.

```
db.tybca.find().sort({"Course":-1})
```

## Aggregation Commands

| Name | Description |
| --- | --- |
| aggregate | Performs aggregation tasks such as $group using an aggregation pipeline. |
| count | Counts the number of documents in a collection or a view. |
| distinct | Displays the distinct values found for a specified key in a collection or a view. |

| | |
|---|---|
| mapReduce | Performs map-reduce aggregation for large data sets. |
| pipeline | The array that transforms the list of documents as a part of the aggregation pipeline. |

## Syntax

- Basic syntax of **aggregate()** method is as follows −

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

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

```
> db.mycol.aggregate([{$group : {_id : "$by_user", num_books : {$sum : 1}}}])
```

{ "_id" : “wfs_books", "num_books" : 2 } { "_id" : "Neo4j", "num_books" : 1 }
>

- The aggregate command does the aggregation operation using the aggregation pipeline.

- The aggregation pipeline allows the user to perform data processing from a record or other source using a stage-based application sequence.

## Aggregation Pipelines

- Aggregation operations allow you to group, sort, perform calculations, analyze data, and much more.
- Aggregation pipelines can have one or more "stages". The order of these stages are important. Each stage acts upon the results of the previous stage.

## Example

```
db.posts.aggregate([
  {
    $match: { likes: { $gt: 1 } }
  },
  {
    $group: { _id: "$category", totalLikes: { $sum: "$likes" } }
  }
])
```

```
[ { _id: 'News', totalLikes: 3 }, { _id: 'Event', totalLikes: 8 } ]
Atlas atlas-8iy36m-shard-0 [primary] blog>
```

# Unit-2: Fundamentals of React.js

**Unit-2: Fundamentals of React.js**

2.1 Overview of React

2.1.1 Concepts of React.

2.1.2 Using React with HTML

2.1.3 React Interactive components: Components within components and Files

2.1.4 Passing data through Props

2.2 Class components

2.2.1 React class and class components

2.2.2 Conditional statements, Operators, Lists

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

## ➢ History of React

- It was created by **Jordan Walke,** who was a software engineer at **Facebook.**
- It was initially developed and maintained by Facebook and was later used in its products like **WhatsApp** & **Instagram.**
- Facebook developed ReactJS in **2011** in its newsfeed section, but it was released to the public in the month of **May 2013.**

**Jordan Walke**

## ➢ What is React

- React is an open-source component-based front-end JavaScript library.
- It is used to create fast and interactive user interfaces for web and mobile applications.
- It is easy to create a dynamic application in React because it requires less coding and offer more functionality.
- It is used by big MNC and fresh new startups
- ReactJS is a declarative, efficient, and flexible JavaScript library for building reusable UI components.

- It is an open-source, component-based front end library responsible only for the view layer of the application.
- A JavaScript library for building user interfaces
- The user interface(UI) is the point of human-computer interaction and communication in a device.
- This can include dipslay screens,keyboards a mouse and the appearance of a desktop.

## ➢ Is React JS Library or Framework

- React is not framework.
- React is a JavaScript library for building user interfaces.
- It is also known as ReactJS and React.js so don’t confused if you read different notation in different places.
- React knows only one thing that is to create an awesome UI.

- **What Should you Know before Learning ReactJS**

- Basic knowledge of HTML,CSS and JavaScript
- Basic understanding  of how to use npm.

- **Where to Write ReactJS**

- **Text Editor/Source Code Editor**

- Visual Studio Code
- Notepad++
- Atom

- **Web Browser**

- Google Chrome,Firefox
- React Developer Tools

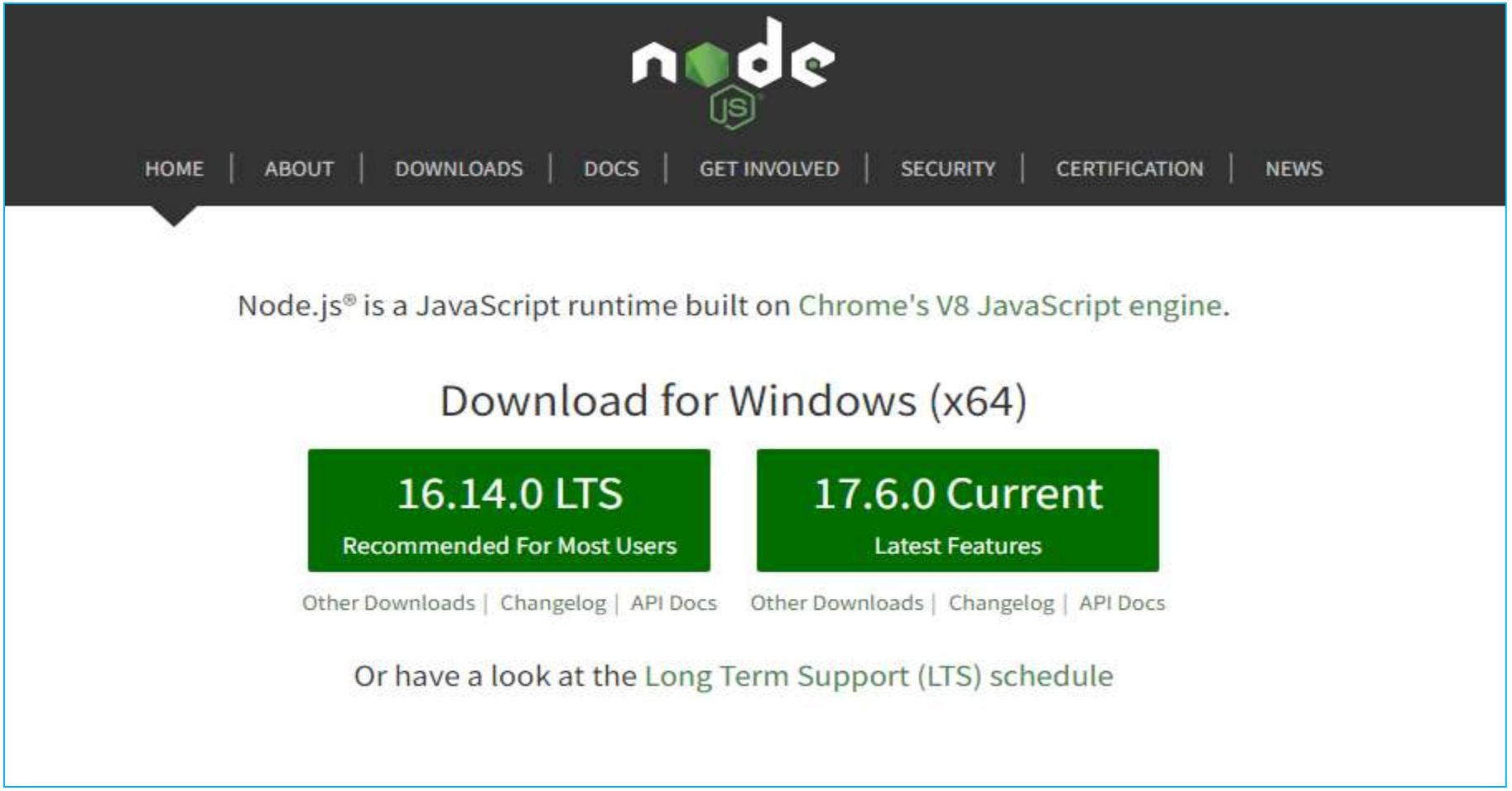
## ➢ Why Learn React

- The main objective of ReactJS is to develop User Interfaces (UI) that improves the speed of the apps.
- It uses virtual DOM (JavaScript object), which improves the performance of the app.
- The JavaScript virtual DOM is faster than the regular DOM. We can use ReactJS on the client and server-side as well as with other frameworks.
- It uses component and data patterns that improve readability and helps to maintain larger apps.

# ➢ Installation Reactjs on Windows:

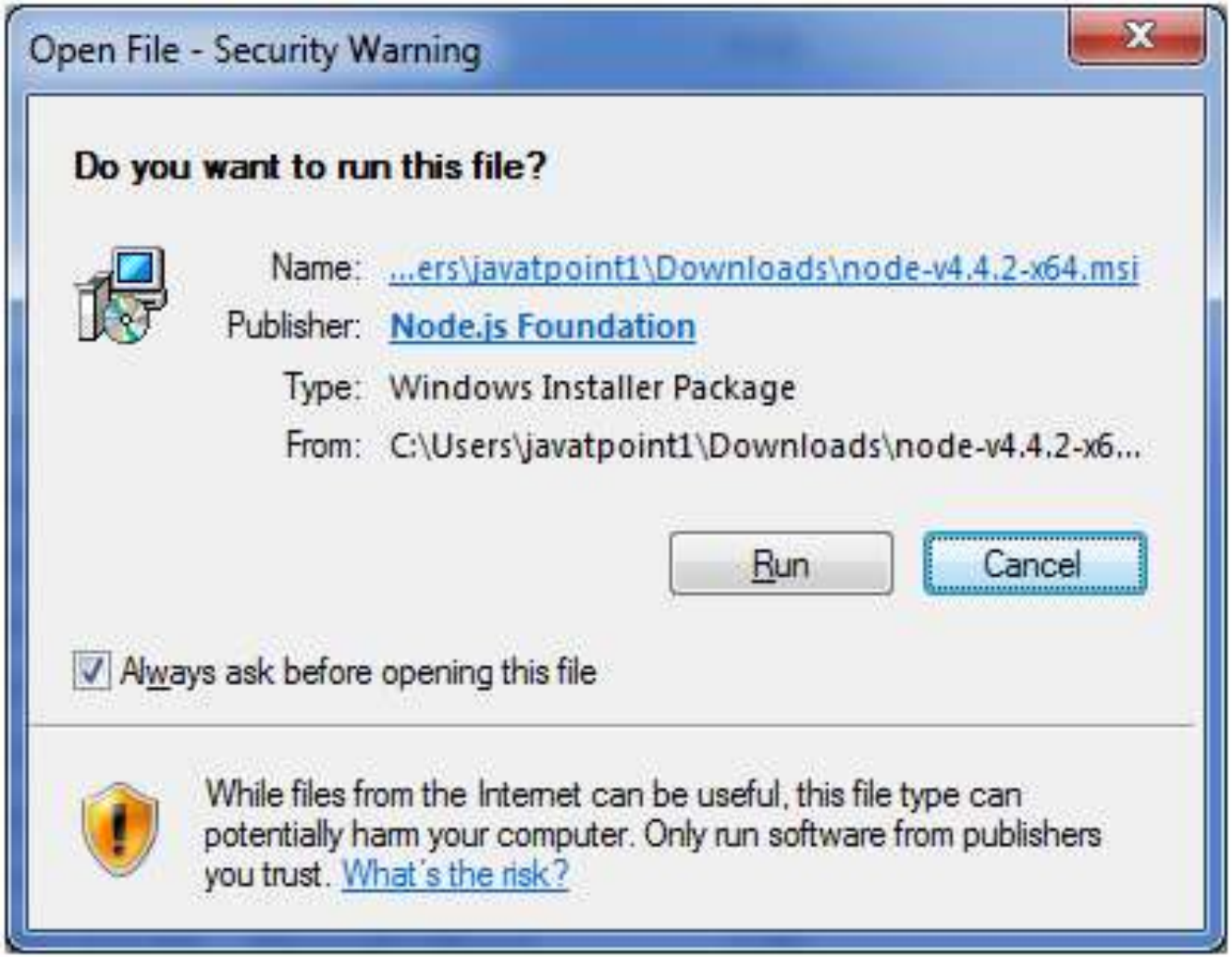
## ❑ Step 1:

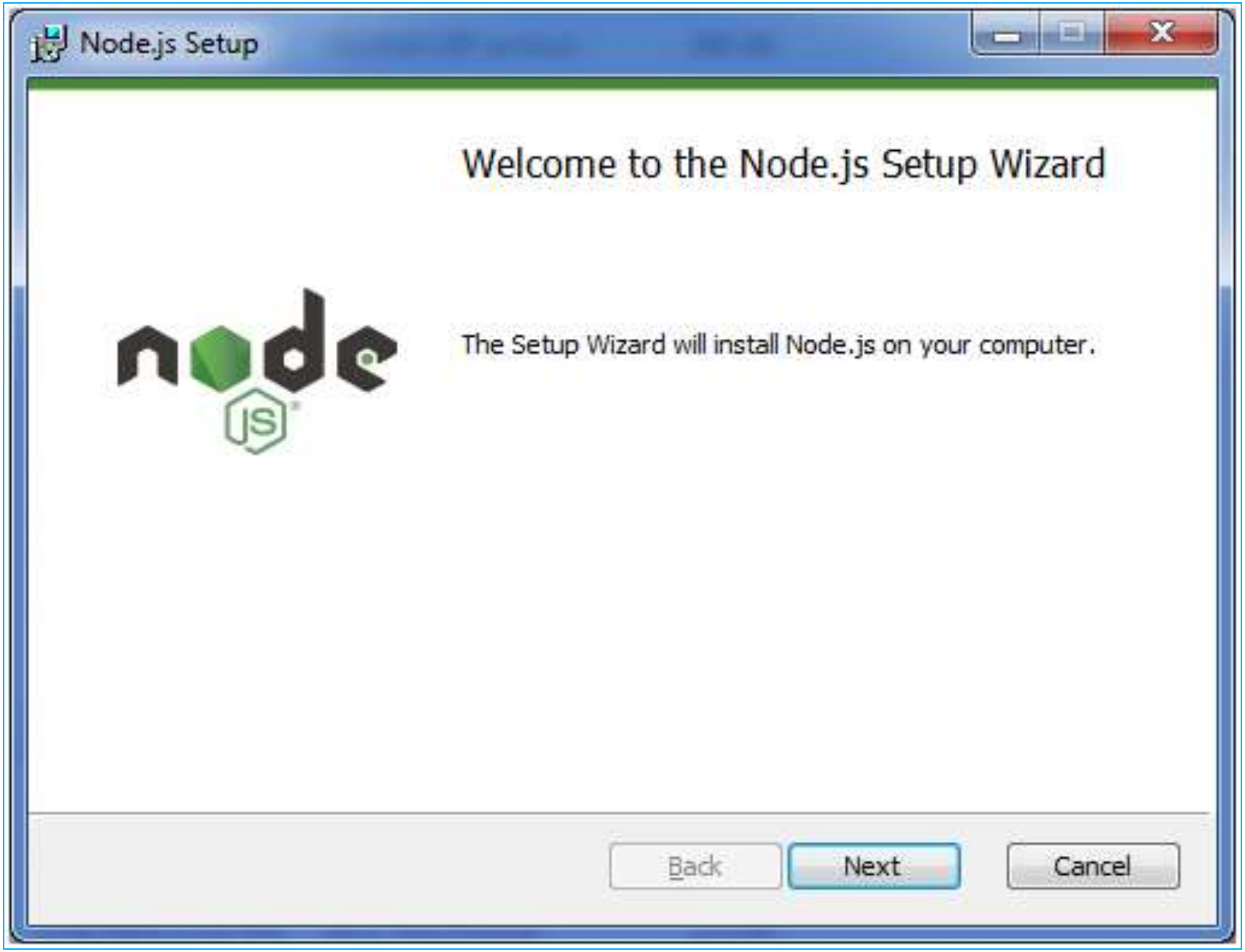
- Install Node.js installer for windows. Click on this link. Here install the LTS version (the one present on the left).
- Once downloaded open NodeJS without disturbing other settings, click on the **Next** button until it's completely installed.




Download the installer for windows by clicking on LTS or Current version button. Here, we will install the latest version LTS for windows that has long time support. However, you can also install the Current version which will have the latest features.

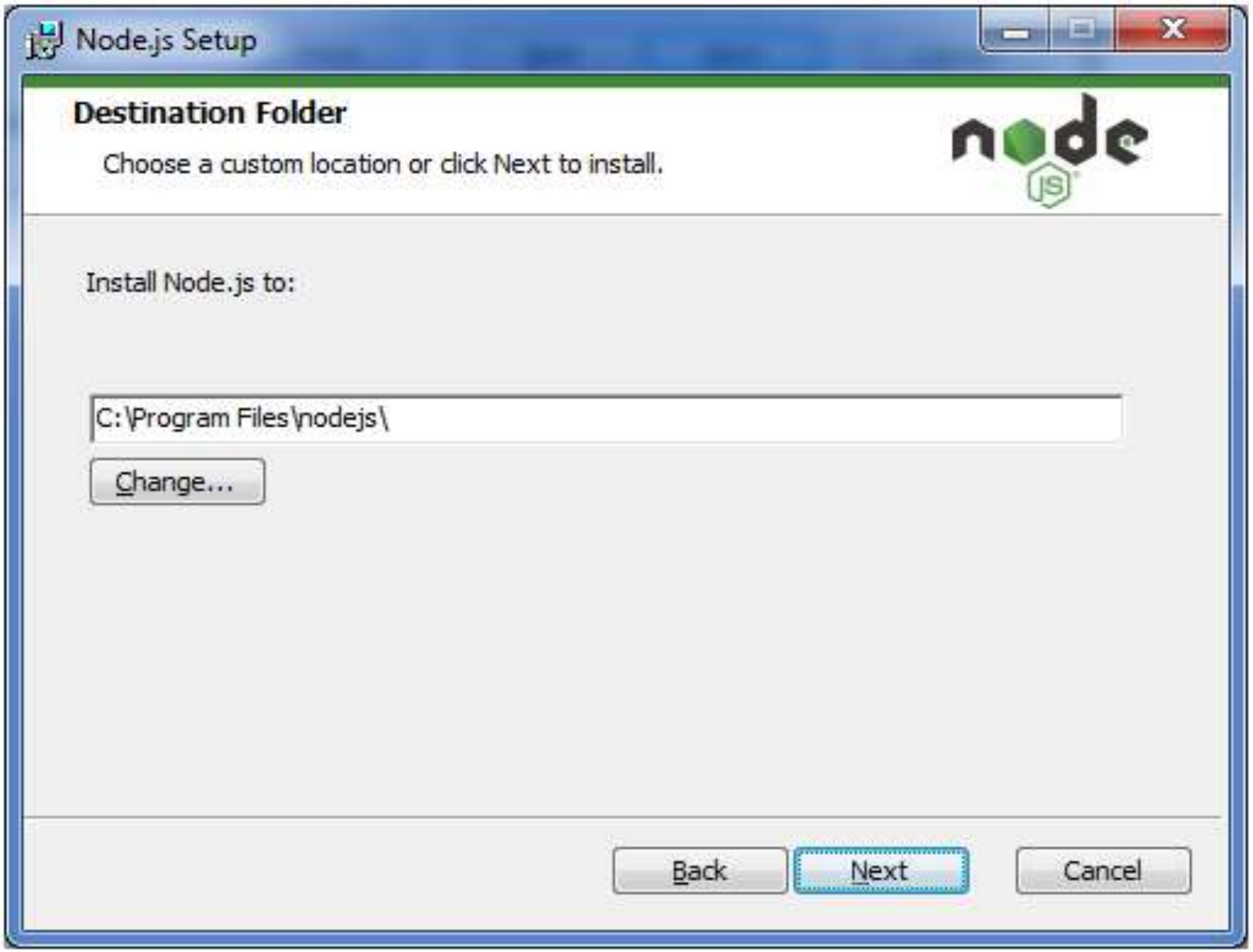
- **After you download the MSI, double-click on it to start the installation as shown below.**

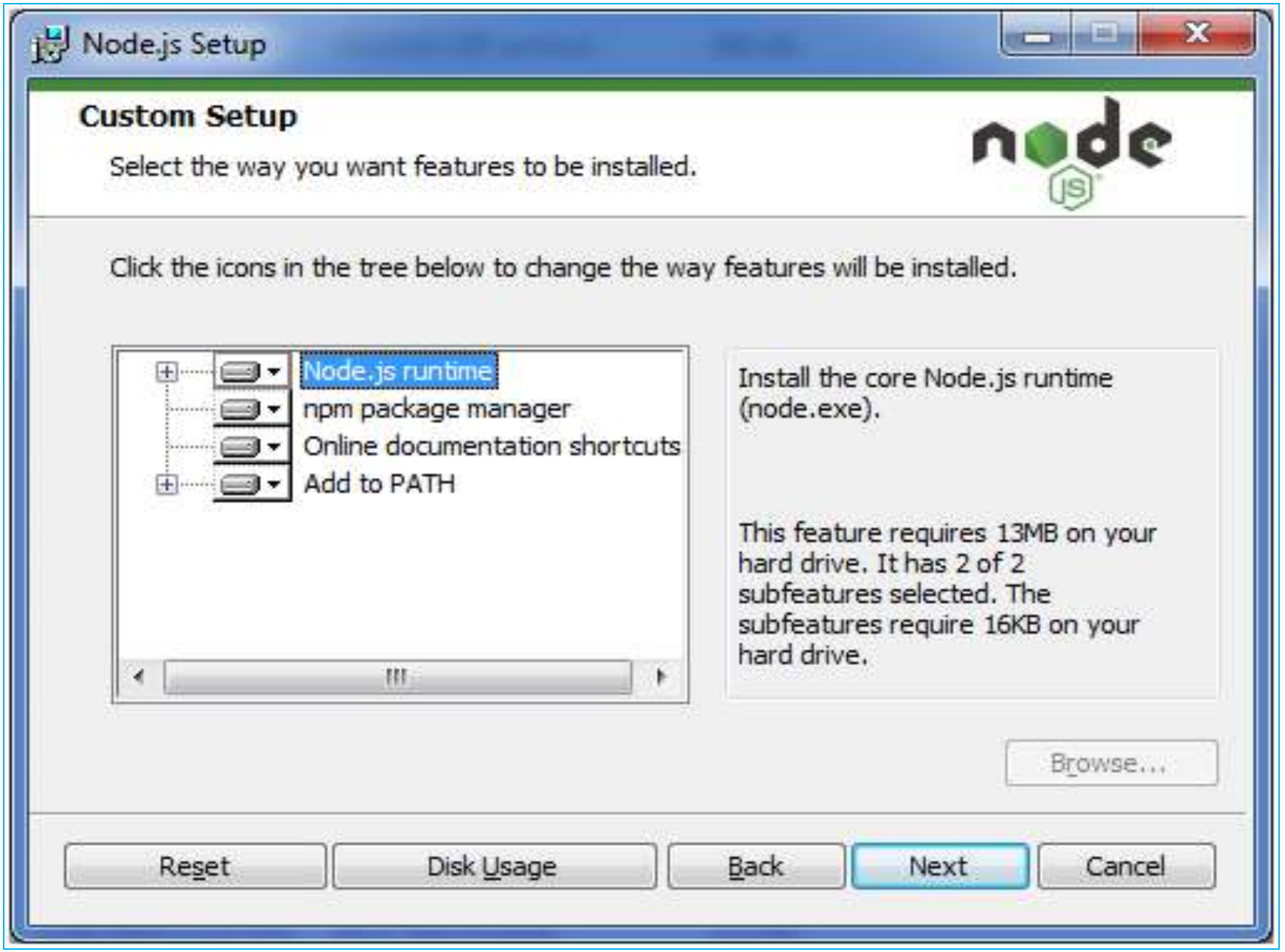



Node.js Setup
Welcome to the Node.js Setup Wizard
The Setup Wizard will install Node.js on your computer.
Back
Next
Cancel

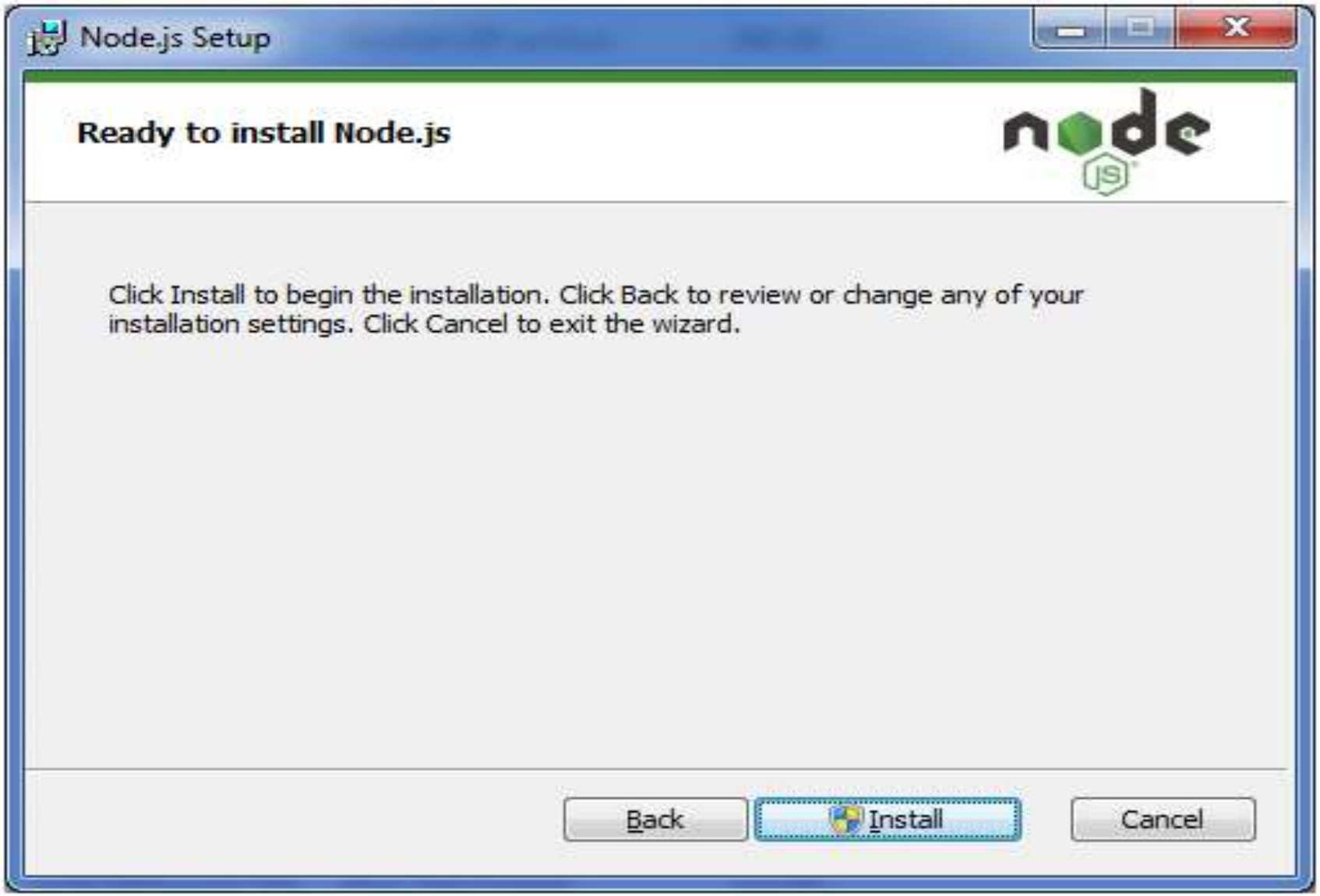
- **Accept the terms of license agreement.**

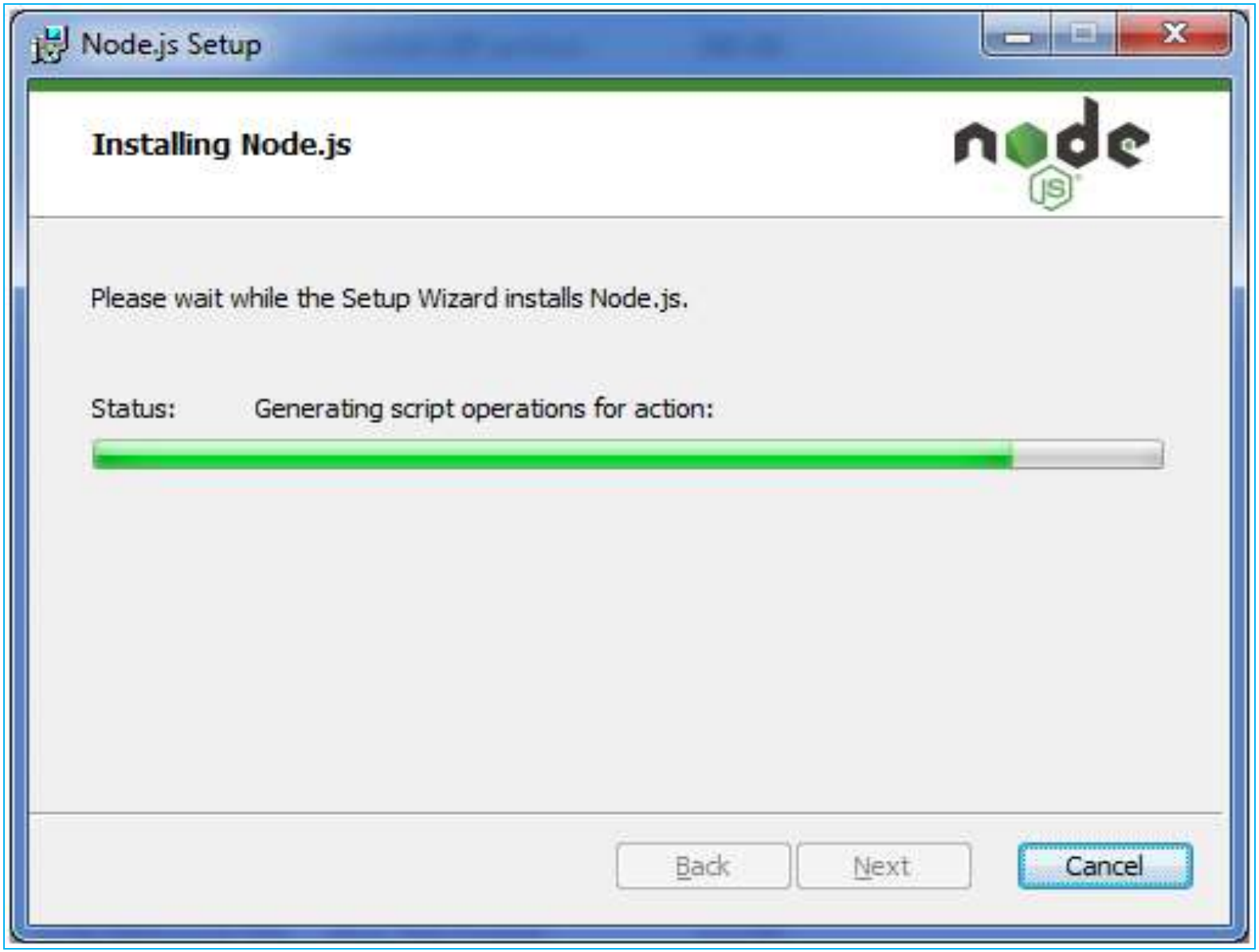
- **Choose the location where you want to install.**

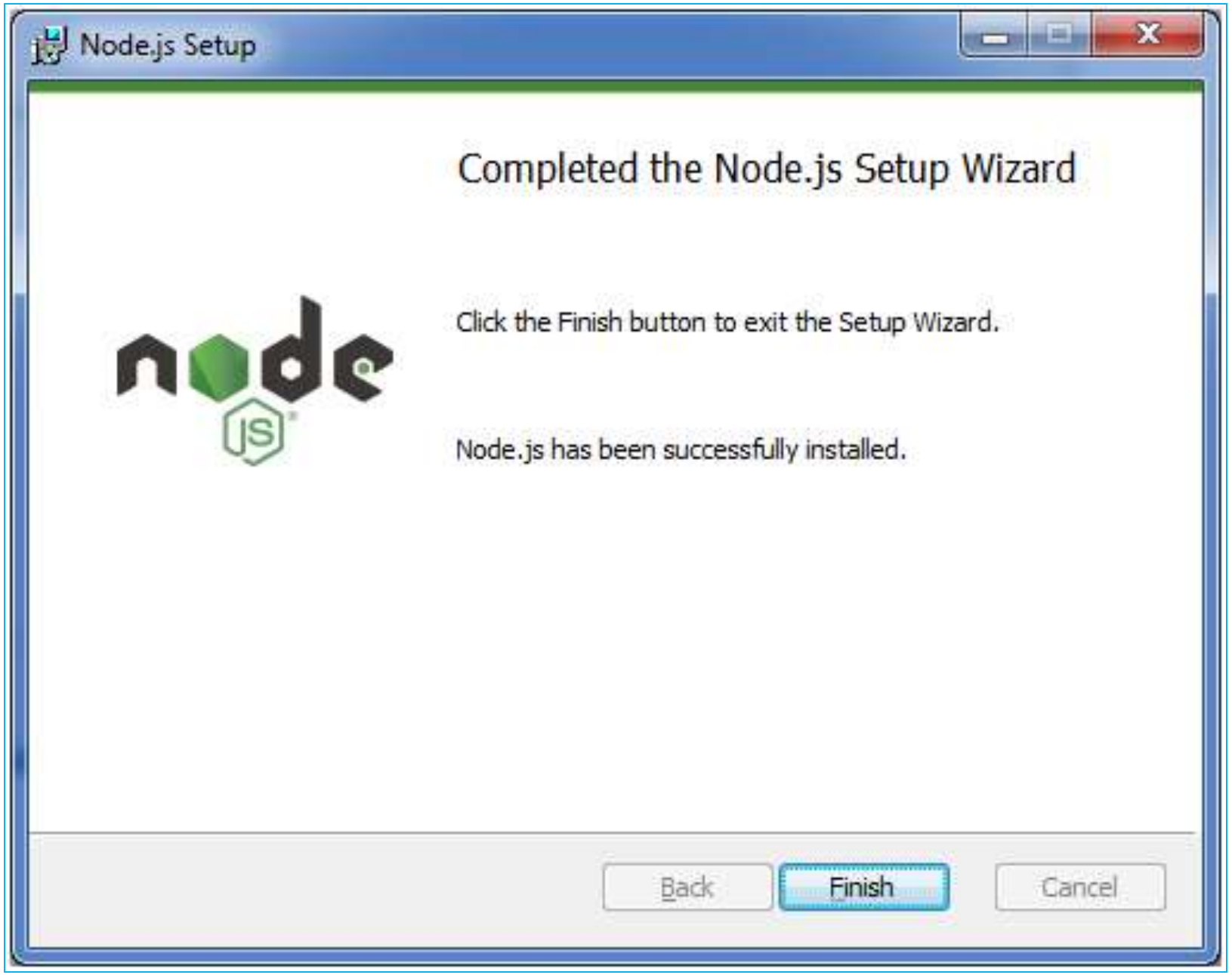



Node.js Setup
Custom Setup
Select the way you want features to be installed.
Click the icons in the tree below to change the way features will be installed.
Node.js runtime
npm package manager
Online documentation shortcuts
Add to PATH
Install the core Node.js runtime (node.exe).
This feature requires 13MB on your hard drive. It has 2 of 2 subfeatures selected. The subfeatures require 16KB on your hard drive.
Browse...
Reset
Disk Usage
Back
Next
Cancel

- **Ready to install:**

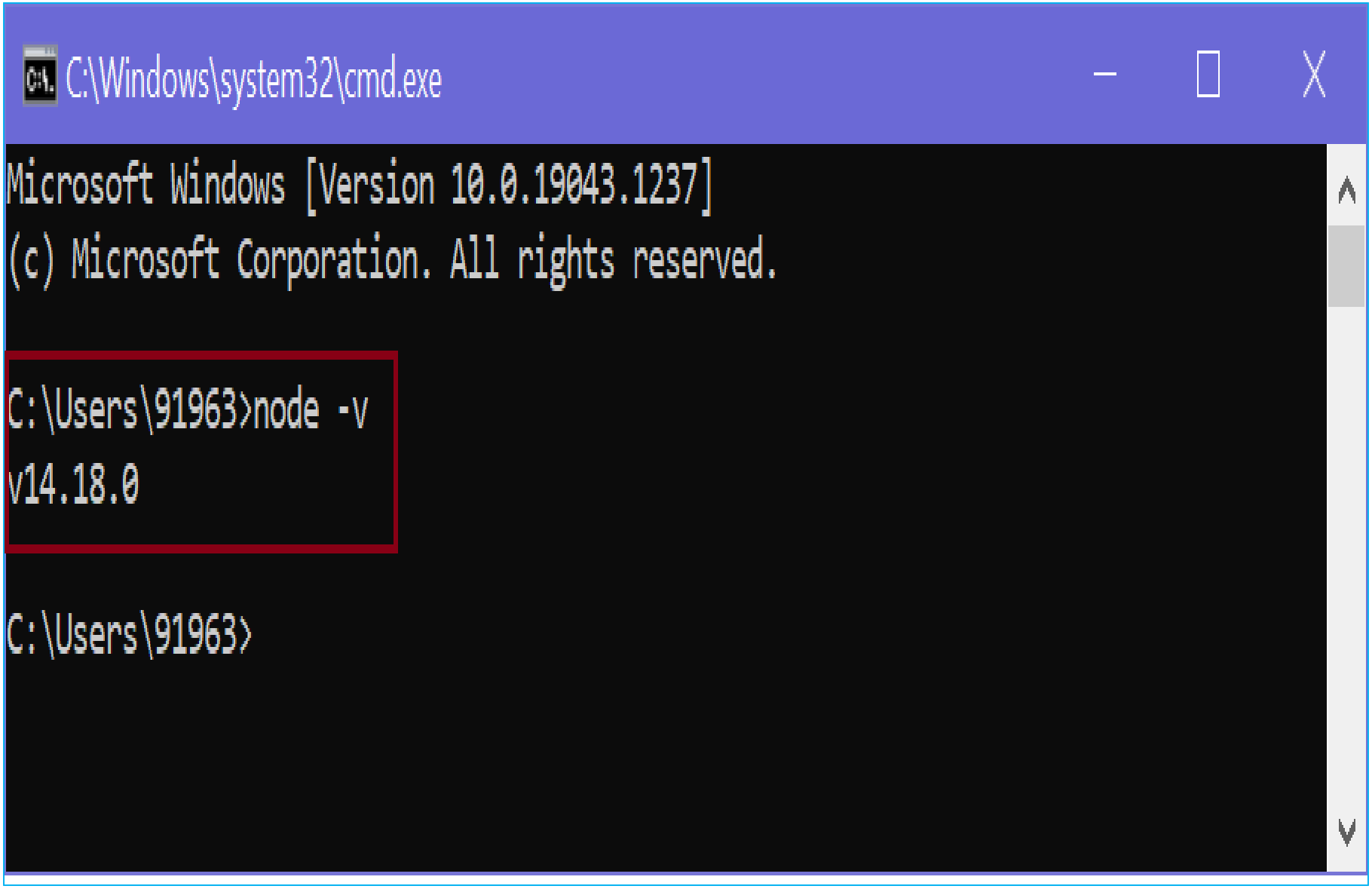



Node.js Setup
Installing Node.js
Please wait while the Setup Wizard installs Node.js.
Status: Generating script operations for action:
Back
Next
Cancel


Node.js Setup
Completed the Node.js Setup Wizard
Click the Finish button to exit the Setup Wizard.
Node.js has been successfully installed.
Back
Finish
Cancel

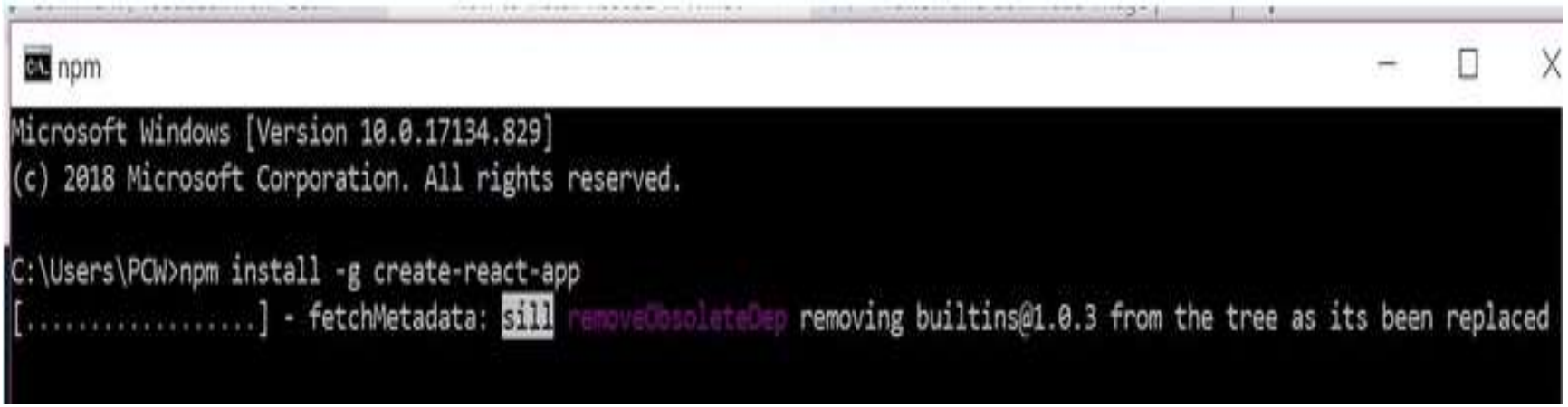
## ➢ Verify Installation

- Once you install Node.js on your computer, you can verify it by opening the command prompt and typing node -v.
- If Node.js is installed successfully then it will display the version of the Node.js installed on your machine, as shown below.


C:\Windows\system32\cmd.exe
Microsoft Windows [Version 10.0.19043.1237]
(c) Microsoft Corporation. All rights reserved.
C:\Users\91963>node -v
v14.18.0
C:\Users\91963>

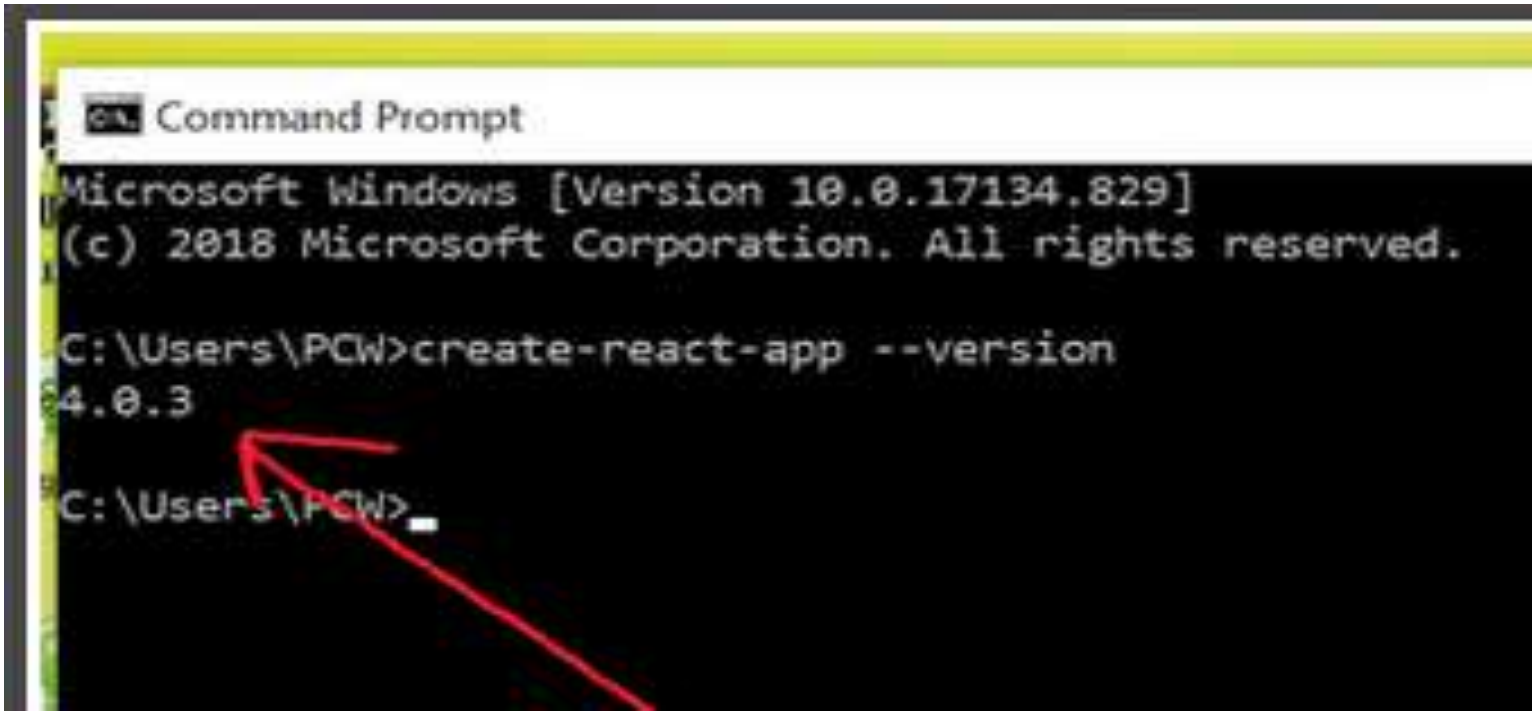
- **Step 3**: Now in the terminal run the below command:

npm install -g create-react-app




- It will globally install react app for you. To check everything  went well run the command

# create-react-app –version




- If everything went well it will give you the installed version of react app

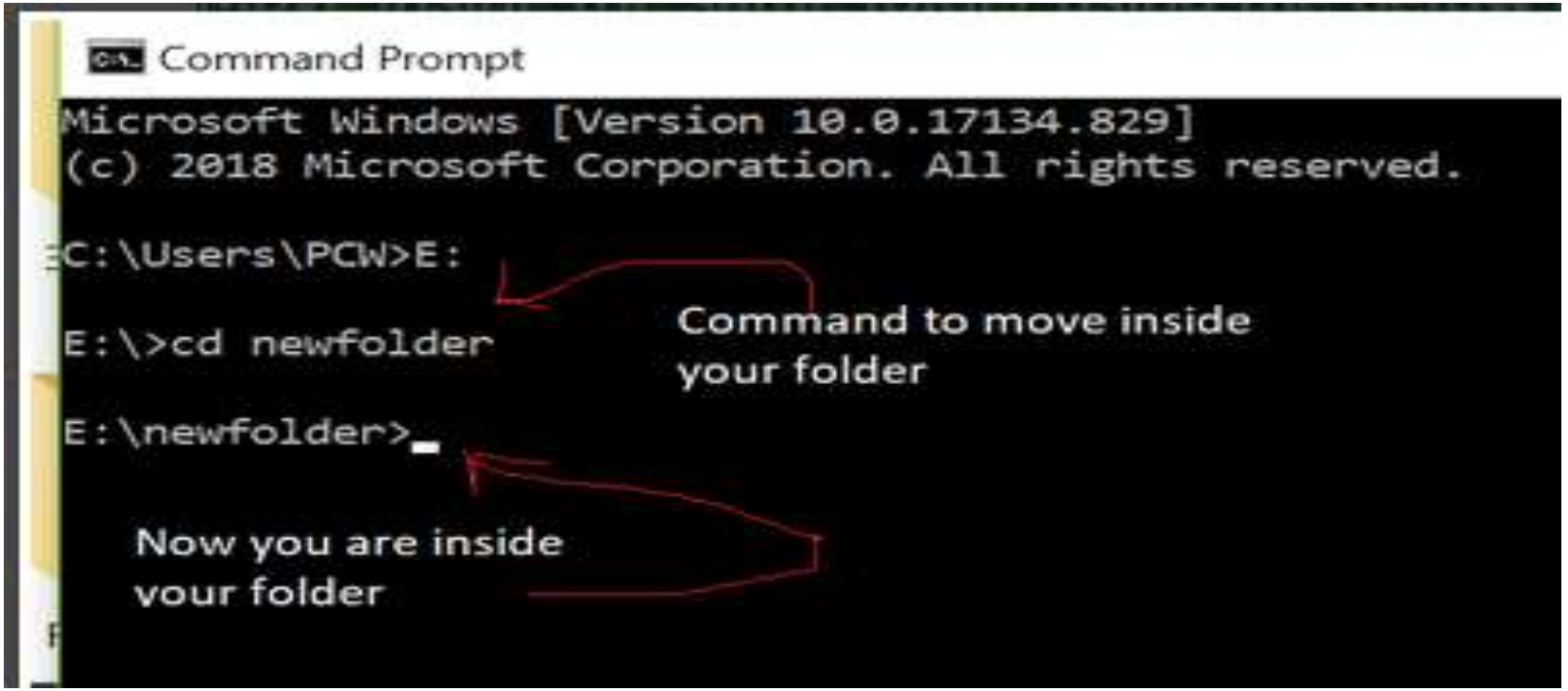
- **Step 4:** Now Create a new folder where you want to make your react app using the below command:

  **mkdir** newfolder

- **Note:** The *newfolder* in the above command is the name of the folder and can be anything.

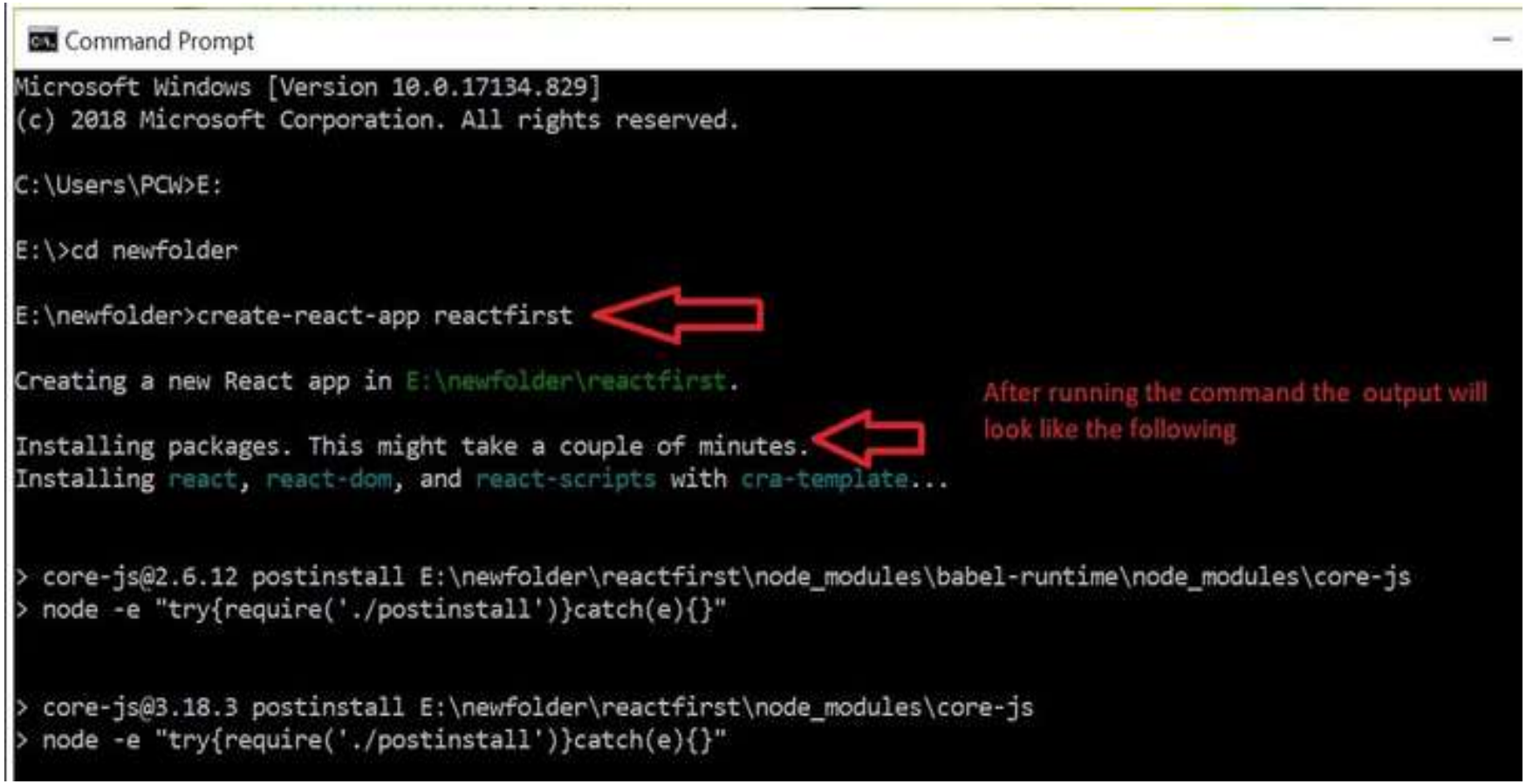
- Move inside the same folder using the below command:

**cd** newfolder (your folder name)



- **Step 5**: Now inside this folder run the command –>

  create-react-app reactfirst YOUR_APP_NAME



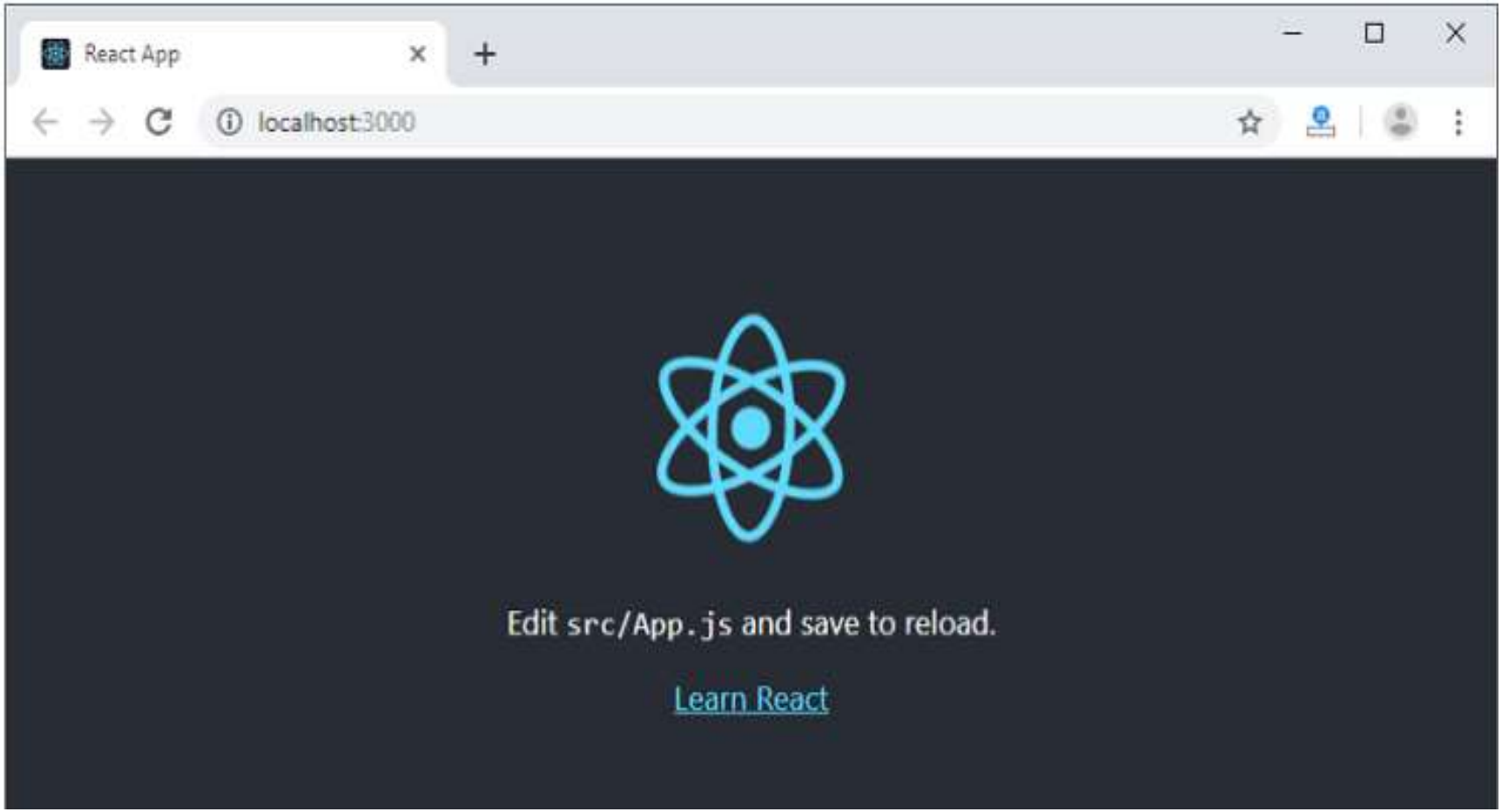

- It will take some time to install the required dependencies

## ➢ Create a new React project

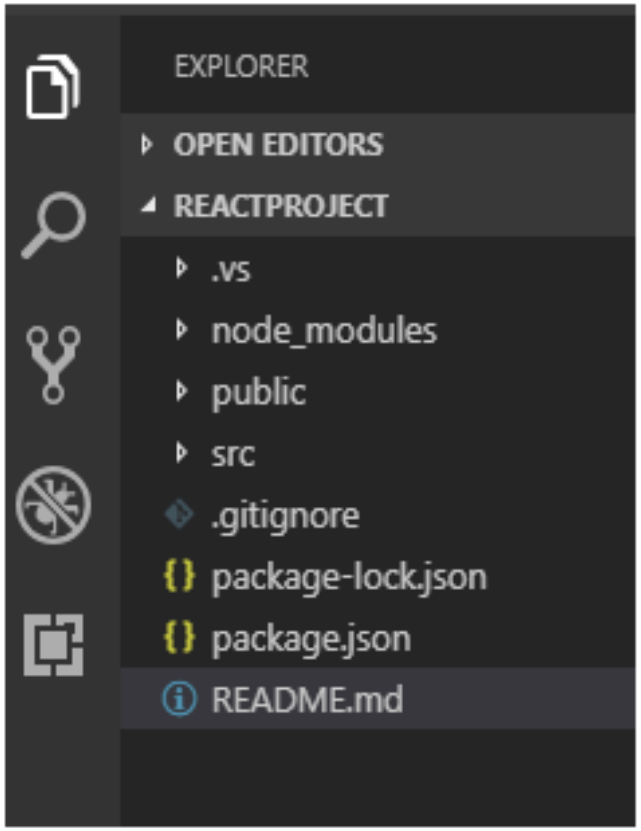
- Once the React installation is successful, we can create a new React project using **create-react-app** command.
- Here, I choose **"reactproject"** name for my project.

**C:\Users\myproject> create-react-app reactproject**

- NPM is a package manager which starts the server and access the application at default server http://localhost:3000. Now, we will get the following screen.



- Next, open the project on Code editor. Here, I am using Visual Studio Code. Our project's default structure looks like as below image.



- **In React application, there are several files and folders in the root directory. Some of them are as follows:**

❑ **node_modules:**

- It contains the React library and any other third party libraries needed.

❑ **public:**

- It holds the public assets of the application. It contains the index.
- html where React will mount the application by default on the  element.

❑ **src:**

- It contains the App.css, App.js, App.test.js, index.css, index.js, and serviceWorker.js files.
- Here, the App.js file always responsible for displaying the output screen in React.

❑ **package-lock.json:**

- It is generated automatically for any operations where npm package modifies either the node_modules tree or package.json.
- It cannot be published. It will be ignored if it finds any other place rather than the top-level package.

❑ **package.json:**

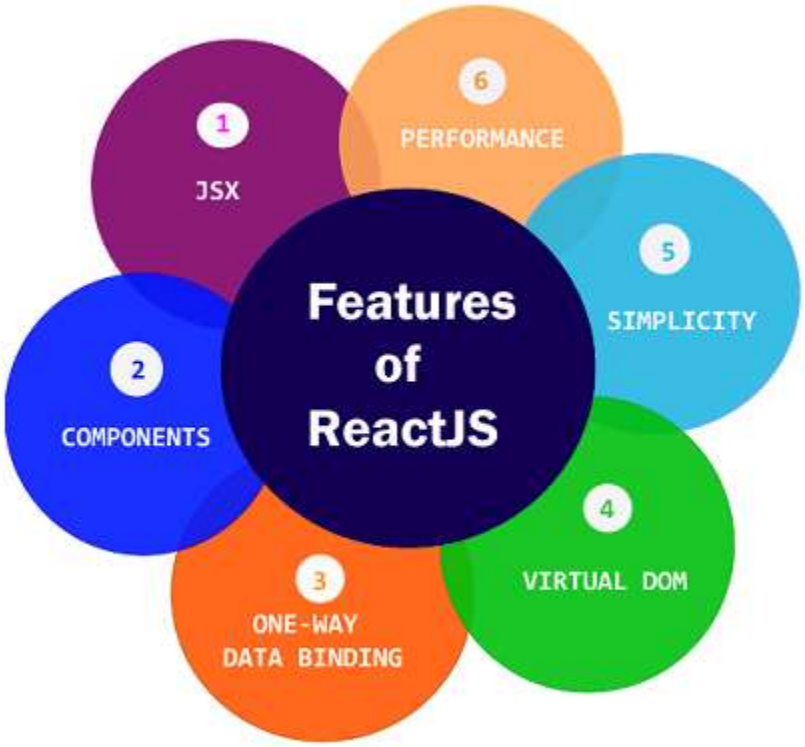
- It holds various metadata required for the project.
- It gives information to npm, which allows to identify the project as well as handle the project?s dependencies.

❑ **README.md:**

- It provides the documentation to read about React topics.

## ➢ React Features

- ReactJS gaining quick popularity as the best JavaScript framework among web developers.
- It is playing an essential role in the front-end ecosystem. The important features of ReactJS are as following.



- ❑ JSX
- ❑ Components
- ❑ One-way Data Binding
- ❑ Virtual DOM
- ❑ Simplicity
- ❑ Performance

## ➢ JSX

- JSX stands for JavaScript XML.
- It is a JavaScript syntax extension.
- Its an XML or HTML like syntax used by ReactJS.
- This syntax is processed into JavaScript calls of React Framework.
- It extends the ES6 so that HTML like text can co-exist with JavaScript react code.
- It is not necessary to use JSX, but it is recommended to use in ReactJS.

- **Components**

- ReactJS is all about components.
- ReactJS application is made up of multiple components, and each component has its own logic and controls.
- These components can be reusable which help you to maintain the code when working on larger scale projects.

- **One-way Data Binding**

- ReactJS is designed in such a manner that follows unidirectional data flow or one-way data binding.
- The benefits of one-way data binding give you better control throughout the application.
- If the data flow is in another direction, then it requires additional features. It is because components are supposed to be immutable and the data within them cannot be changed.

## ➢ Virtual DOM

- A virtual DOM object is a representation of the original DOM object.
- It works like a one-way data binding.
- Whenever any modifications happen in the web application, the entire UI is re-rendered in virtual DOM representation.
- Then it checks the difference between the previous DOM representation and new DOM.
- Once it has done, the real DOM will update only the things that have actually changed. This makes the application faster, and there is no wastage of memory.

- **Simplicity**

- ReactJS uses JSX file which makes the application simple and to code as well as understand.
- We know that ReactJS is a component-based approach which makes the code reusable as your need.
- This makes it simple to use and learn.

- **Performance**

- ReactJS is known to be a great performer. This feature makes it much better than other frameworks out there today.
- The reason behind this is that it manages a virtual DOM.
- The DOM is a cross-platform and programming API which deals with HTML, XML or XHTML.

- The DOM exists entirely in memory. Due to this, when we create a component, we did not write directly to the DOM.
- Instead, we are writing virtual components that will turn into the DOM leading to smoother and faster performance.

## ➢ Adding React to an HTML Page

```
<!DOCTYPE html>
<html lang="en">
<title>Test React</title>
<!-- Load React API -->
<script src= "https://unpkg.com/react@16/umd/react.production.min.js"></script>
<!-- Load React DOM-->
<script src= "https://unpkg.com/react-dom@16/umd/react-dom.production.min.js"></script>
<!-- Load Babel Compiler -->
<script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js"></script>
<body>
        <script type="text/babel">
                //  JSX Babel code goes here
        </script>
</body>
</html>
```

## ➢ What is Babel?

- Babel is a JavaScript compiler that can translate markup or programming languages into JavaScript.
- With Babel, you can use the newest features of JavaScript (ES6 - ECMAScript 2015).
- Babel is available for different conversions. React uses Babel to convert JSX into JavaScript.

## ➢ What is JSX?

- JSX stands for **J**ava**S**cript **X**ML.
- JSX is an XML/HTML like extension to JavaScript.
- JSX is a combination of HTML and JavaScript.
- You can embed JavaScript objects inside the HTML elements.
- JSX is not supported by the browsers, as a result Babel compiler transcompile the code into JavaScript code.
- JSX makes codes easy and understandable.
- It is easy to learn if you know HTML and JavaScript.

➢ **Example**

- Here, we will write JSX syntax in JSX file and see the corresponding JavaScript code which transforms by preprocessor(babel).

❑ **JSX File**

```
<div>Hello ReactJS</div>
```

❑ **Corresponding Output**

```
React.createElement("div", null, "Hello  ReactJS ");
```

- The above line creates a **react element** and passing **three arguments** inside where the first is the name of the element which is div, second is the **attributes** passed in the div tag, and last is the **content** you pass which is the "Hello ReactJS."

## ➢ React DOM Render

- The method ReactDom.render() is used to render (display) HTML elements:

## ➢ Example

```
<!DOCTYPE html>
<html lang="en">
<title>Test React</title>
<script src=
"https://unpkg.com/react@16/umd/react.production.min.js"></script>
<script src= "https://unpkg.com/react-dom@16/umd/react-
dom.production.min.js"></script>
<script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js"></script>
<body>
        <div id="id01">Hello World!</div>
```

```
<script type="text/babel">
ReactDOM.render(
  <h1>Hello React!</h1>,
  document.getElementById('id01'));
</script>


</body>
</html>
```

- **Output**

## ➢ JSX Expressions

- Expressions can be used in JSX by wrapping them in curly **{}** braces.

```
<!DOCTYPE html>
<html lang="en">

<title>Test React</title>
<script src=
"https://unpkg.com/react@16/umd/react.production.min.js"></script>
<script src= "https://unpkg.com/react-dom@16/umd/react-
dom.production.min.js"></script>
<script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js"></script>

<body>
<div id="id01">Hello World!</div>
```

```
<script type="text/babel">
const name = 'John Doe';

ReactDOM.render(
  <h1>Hello {name}</h1>,
  document.getElementById('id01'));
</script>

</body>
</html>
```

- **Output**

## ➢ React Elements

- React applications are usually built around a single **HTML element**.
- React developers often call this the **root node** (root element):

```
<div id="root"></div>
```

**React elements** look like this:

```
const element = <h1>Hello React!</h1>
```

Elements are **rendered** (displayed) with the ReactDOM.render() method:

```
ReactDOM.render(element, document.getElementById('root'));
```

➢ **Example**

```
<!DOCTYPE html>
<html lang="en">

<title>Test React</title>
<script src= "https://unpkg.com/react@16/umd/react.production.min.js"></script>
<script src= "https://unpkg.com/react-dom@16/umd/react-
dom.production.min.js"></script>
<script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js"></script>

<body>
<div id="root"></div>

<script type="text/babel">
ReactDOM.render(<h1>Hello React!</h1>, document.getElementById('root'));
</script>

</body>
</html>
```

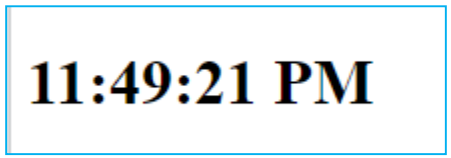
**Output**

- **Example**

```
<!DOCTYPE html>
<html lang="en">
<title>Test React</title>
<script src=
"https://unpkg.com/react@16/umd/react.production.min.js"></script>
<script src= "https://unpkg.com/react-dom@16/umd/react-
dom.production.min.js"></script>
<script src="https://unpkg.com/babel-
standalone@6.15.0/babel.min.js"></script>
<body>
        <div id="root"></div>
```

```
<script type="text/babel">
function tick() {
  const element = (<h1>{new Date().toLocaleTimeString()}</h1>);
  ReactDOM.render(element, document.getElementById('root'));
}

setInterval(tick, 1000);
</script>

</body>
</html>
```

**Output**



## ➢ Advantage of ReactJS

### ❑ Easy to Learn and Use

- ReactJS is much easier to learn and use. It comes with a good supply of documentation, and training resources.
- Any developer who comes from a JavaScript background can easily understand and start creating web apps using React.
- It is not fully featured but has the advantage of open-source JavaScript User Interface(UI) library, which helps to execute the task in a better manner.

## ❑ Creating Dynamic Web Applications Becomes Easier

- To create a dynamic web application specifically with HTML strings was tricky because it requires a complex coding, but React JS solved that issue and makes it easier.
- It provides less coding and gives more functionality. It makes use of the JSX(JavaScript Extension), which is a particular syntax letting HTML quotes and HTML tag syntax to render particular subcomponents.
- It also supports the building of machine-readable codes.

## ❑ Reusable Components

- A ReactJS web application is made up of multiple components, and each component has its own logic and controls.
- These components are responsible for outputting a small, reusable piece of HTML code which can be reused wherever you need them.
- The reusable code helps to make your apps easier to develop and maintain.
- 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.

## ❑ Performance Enhancement

- ReactJS improves performance due to virtual DOM. The DOM is a cross-platform and programming API which deals with HTML, XML or XHTML.
- Most of the developers faced the problem when the DOM was updated, which slowed down the performance of the application. ReactJS solved this problem by introducing virtual DOM.
- The React Virtual DOM exists entirely in memory and is a representation of the web browser's DOM.
- Due to this, when we write a React component, we did not write directly to the DOM.
- Instead, we are writing virtual components that react will turn into the DOM, leading to smoother and faster performance.

## ❑ The Support of Handy Tools

- React JS has also gained popularity due to the presence of a handy set of tools.
- These tools make the task of the developers understandable and easier.
- The React Developer Tools have been designed as Chrome and Firefox dev extension and allow you to inspect the React component hierarchies in the virtual DOM.
- It also allows you to select particular components and examine and edit their current props and state.

## ❑ Known to be SEO Friendly

- Traditional JavaScript frameworks have an issue in dealing with SEO. The search engines generally having trouble in reading JavaScript-heavy applications.
- Many web developers have often complained about this problem.
- ReactJS overcomes this problem that helps developers to be easily navigated on various search engines.
- It is because React.js applications can run on the server, and the virtual DOM will be rendering and returning to the browser as a regular web page.

## ❑ The Benefit of Having JavaScript Library

- Today, ReactJS is choosing by most of the web developers.
- It is because it is offering a very rich JavaScript library.
- The JavaScript library provides more flexibility to the web developers to choose the way they want.

## ❑ Scope for Testing the Codes

- ReactJS applications are extremely easy to test.
- It offers a scope where the developer can test and debug their codes with the help of native tools.

## ➢ Disadvantage of ReactJS

### ❑ The high pace of development

- The high pace of development has an advantage and disadvantage both.
- In case of disadvantage, since the environment continually changes so fast, some of the developers not feeling comfortable to relearn the new ways of doing things regularly.
- It may be hard for them to adopt all these changes with all the continuous updates.
- They need to be always updated with their skills and learn new ways of doing things.

## ❑ Poor Documentation

- It is another cons which are common for constantly updating technologies.
- React technologies updating and accelerating so fast that there is no time to make proper documentation.
- To overcome this, developers write instructions on their own with the evolving of new releases and tools in their current projects.

## ❑ View Part

- ReactJS Covers only the UI Layers of the app and nothing else.
- So you still need to choose some other technologies to get a complete tooling set for development in the project.

## ❑ JSX as a barrier

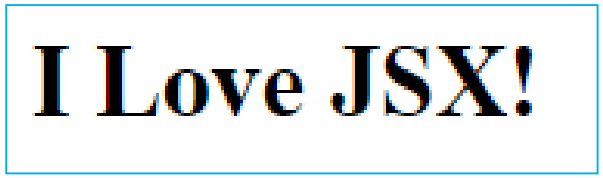
- ReactJS uses JSX. It's a syntax extension that allows HTML with JavaScript mixed together.
- This approach has its own benefits, but some members of the development community consider JSX as a barrier, especially for new developers.
- Developers complain about its complexity in the learning curve.

## ➢ Example 1

**JSX:**

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

const myElement = <h1>I Love JSX!</h1>;

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

## ➢ Output



➢ **Example 2**

**Without JSX:**

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

const myElement = React.createElement('h1', {}, 'I do not use JSX!');

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

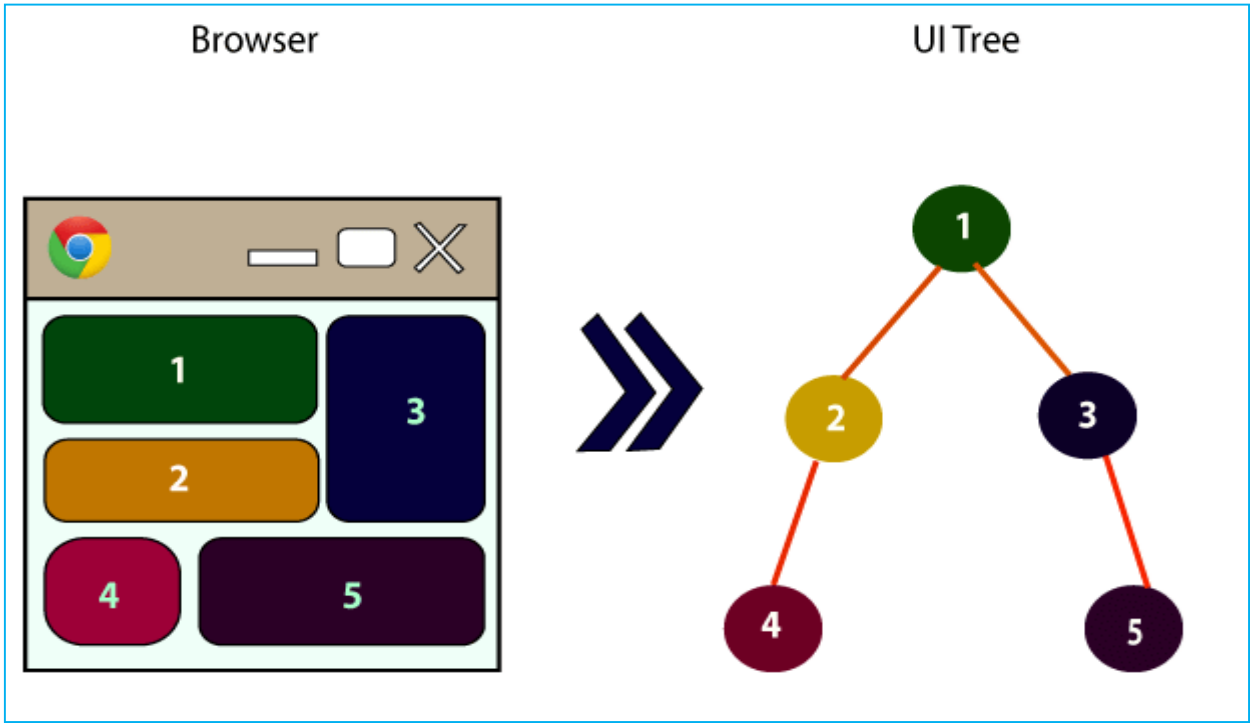
➢ **Output**

I do not use JSX!

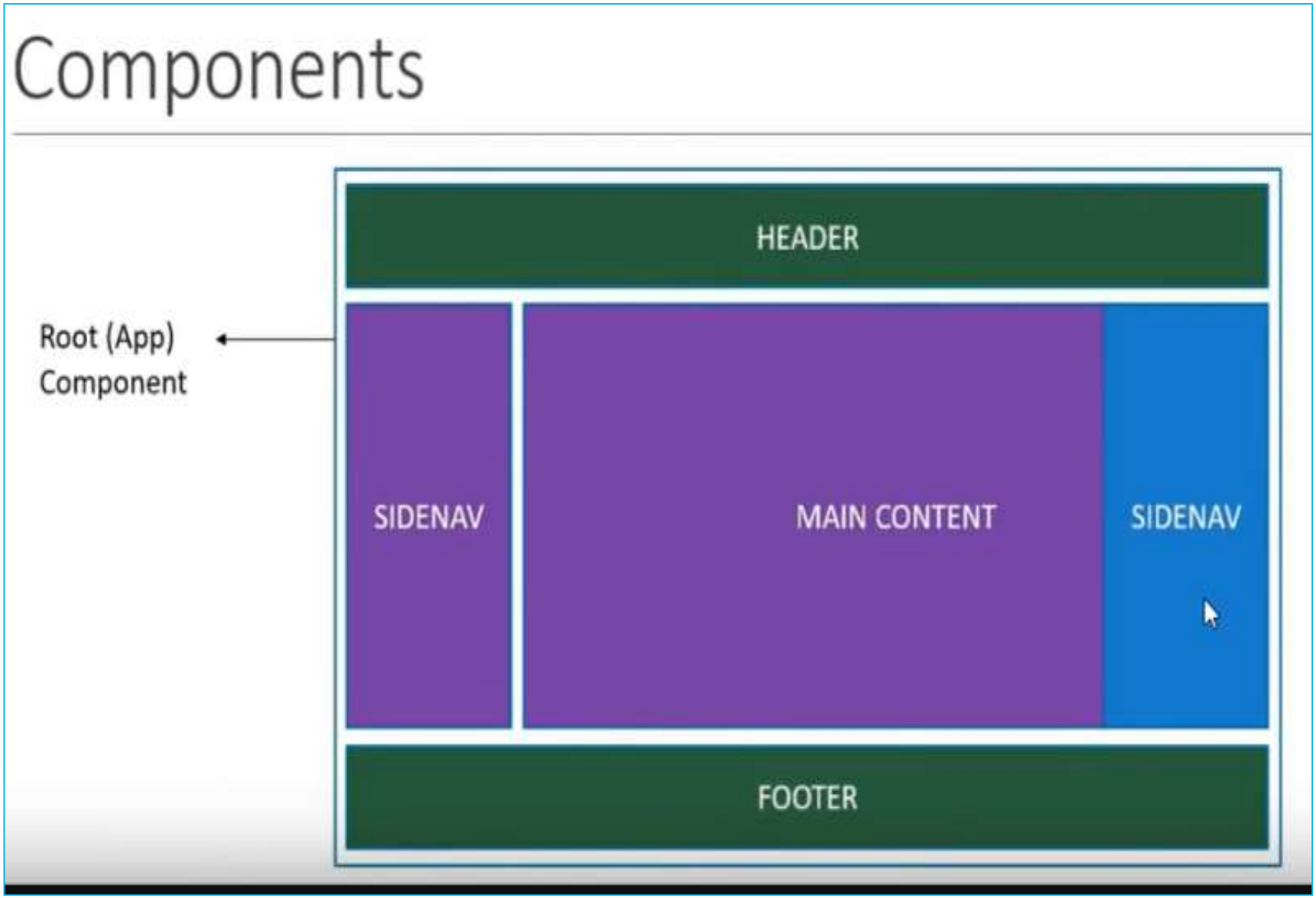
## ➢ React Components

- Components are independent and reusable bits of code.
- They serve the same purpose as JavaScript functions, but work in isolation and return HTML.
- 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.

- Every React component have their own structure, methods as well as APIs.
- They can be reusable as per your need. For better understanding, consider the entire UI as a tree.
- Here, the root is the starting component, and each of the other pieces becomes branches, which are further divided into sub-branches



# Components


HEADER
Root (App)
Component
SIDENAV
MAIN CONTENT
SIDENAV
FOOTER

➢ **In ReactJS, we have mainly two types of components. They are**

- Functional Components
- Class Components

❑ **Functional Components**

- In React, function components are a way to write components that only contain a render method and don't have their own state.
- They are simply JavaScript functions that may or may not receive data as parameters.
- We can create a function that takes props(properties) as input and returns what should be rendered.

- 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, are easier to understand

## ➢ Create Your First Component

- When creating a React component, the component's name *MUST* start with an upper case letter.

➢ **A valid functional component can be shown in the below example.**

```
function WelcomeMessage(props) {
  return <h1>Welcome to the , {props.name}</h1>;
}
```

- 

## Example

Create a Function component called `Car`

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

## ❑ Class Component

- 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.
- Class components are more complex than functional components.
- It requires you to extend from React.

- Component and create a render function which returns a React element.
- 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 a Class component called `Car`

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

- **Valid class component is shown in the below example.**

```
class MyComponent extends React.Component {
  render() {
    return (
      <div>This is main component.</div>
    );
  }
}
```

## ❑ Rendering a Component

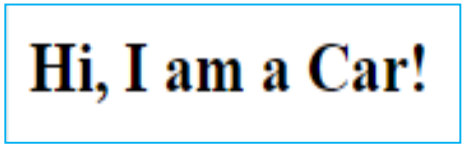
- Now your React application has a component called Car, which returns an <h2> element.
- To use this component in your application, use similar syntax as normal HTML: <Car />

➢ **Example**

❑ **Display the Car component in the "root" element:**

```
import React from 'react';

import ReactDOM from 'react-dom/client';

function Car() {

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

}

const root =

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

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

**Output**



## ➢ 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.
- They are **read-only** components.
- It is an object which stores the value of attributes of a tag and work similar to the HTML attributes.
- It gives a way to pass data from one component to other components.
- It is similar to function arguments.
- Props are passed to the component in the same way as arguments passed in a function.

## ➢ Example

- Use an attribute to pass a color to the Car component, and use it in the render() function:

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

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

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

- **Output**

I am a red Car!

## ➢ Example

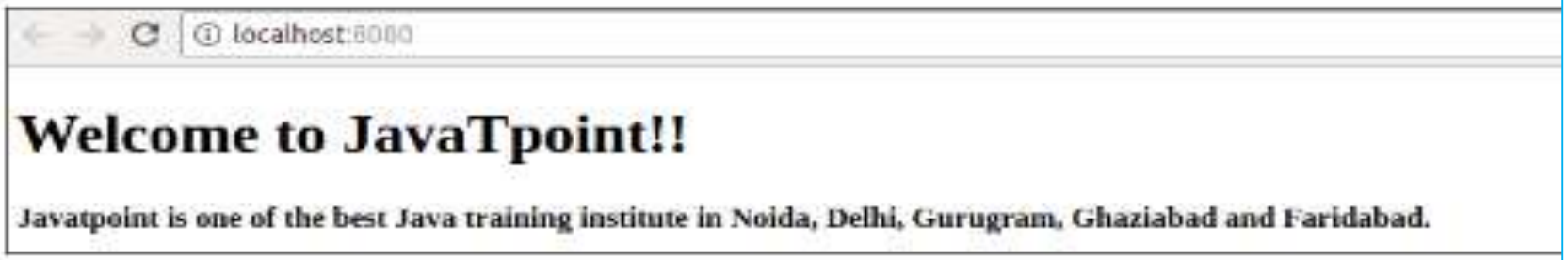
- **App.js**

```
import React, { Component } from 'react';
class App extends React.Component {
  render() {
    return (
        <div>
         <h1> Welcome to { this.props.name } </h1>
         <p> <h4> Javatpoint is one of the best Java training institute in Noida, Delhi, Gurugram, Ghazi
        </div>
    );
  }
}
export default App;
```

- **Main.js**

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

ReactDOM.render(<App name = "JavaTpoint!!" />, document.getElementById('app'));
```

**Output**



## ➢ Components in Components

- We can refer to components inside other components:

## ➢ Example

- Use the Car component inside the Garage component:

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

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

function Garage() {
  return (
    <>
      <h1>Who lives in my Garage?</h1>
      <Car />
    </>
  );
}

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

- **Output**

**Who lives in my Garage?**

**I am a Car!**

## ➢ 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:

### Example

This is the new file, we named it "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.**

- **Example**
- Now we 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 />);
```

- **Output**

**Hi, I am a Car!**

## ➢ React Conditional Rendering

- In React, conditional rendering works the same way as the conditions work in JavaScript.
- We use JavaScript operators to create elements representing the current state, and then React Component update the UI to match them.
- From the given scenario, we can understand how conditional rendering works. Consider an example of handling a **login/logout** button.
- The login and logout buttons will be separate components.
- If a user logged in, render the **logout component** to display the logout button.
- If a user not logged in, render the **login component** to display the login button. In React, this situation is called as **conditional rendering**.

- **There is more than one way to do conditional rendering in React. They are given below.**

- if
- ternary operator
- logical && operator
- switch case operator
- Conditional Rendering with enums

## ➢ if

- It is the easiest way to have a conditional rendering in React in the render method.
- It is restricted to the total block of the component.
- IF the condition is **true**, it will return the element to be rendered.
- if JavaScript operator to decide which component to render.

➢ **It can be understood in the below example.**

```
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;
  if (isGoal) {
    return <MadeGoal/>;
  }
  return <MissedGoal/>;
}

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

- **Logical && operator**

- This operator is used for checking the condition. If the condition is **true**, it will return the element **right** after **&&**, and if it is **false**, React will **ignore** and skip it.

- **Syntax**

```
{
    condition &&
    // whatever written after && will be a part of output.
}
```

- We can understand the behavior of this concept from the below example.
- If you run the below code, you will not see the **alert** message because the condition is not matching.

```
(' reactjs' ==  ' reactjs ') && alert('This alert will never be shown!')
```

- If you run the below code, you will see the **alert** message because the condition is matching.

```
(10 > 5) && alert('This alert will be shown!')
```

➢ **Example**

```
import React from 'react';
import ReactDOM from 'react-dom';
// Example Component
function Example()
{
   return(<div>
         {
            (10 > 5) && alert('This alert will be shown!')
         }
         </div>
      );
}
```

- You can see in the above output that as the condition **(10 > 5)** evaluates to true, the alert message is successfully rendered on the screen.

## ➢ Ternary operator

- The ternary operator is used in cases where two blocks alternate given a certain condition.
- This operator makes your if-else statement more concise.
- It takes **three** operands and used as a shortcut for the if statement.

### Syntax

```
condition ?  true : false
```

If the condition is **true**, **statement1** will be rendered. Otherwise, **false** will be rendered.

- **Example**

```
render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      Welcome {isLoggedIn ? 'Back' : 'Please login first'}.
    </div>
  );
}
```

- **Switch case operator**

- Sometimes it is possible to have multiple conditional renderings.
- In the switch case, conditional rendering is applied based on a different state.

- **Example**

```
function NotificationMsg({ text}) {
  switch(text) {
    case 'Hi All':
      return <Message: text={text} />;
    case 'Hello JavaTpoint':
      return <Message text={text} />;
    default:
      return null;
  }
}
```

# ➢ Conditional Rendering with enums

- In JavaScript, an object can be used as an enum when it is used as a map of key-value pairs.
- An **enum** is a great way to have a multiple conditional rendering.
- It is more **readable** as compared to switch case operator.
- It is perfect for **mapping** between different **state**.
- It is also perfect for mapping in more than one condition.
- It can be understood in the below example.

- **Example**

```
const ENUMOBJECT = {
  a: '1',
  b: '2',
  c: '3',
};
```

➢ **Example**

- We want to create three different components Foo, Bar and Default. We want to show these components based on a certain state.

```
const Foo = () => {
  return <button>FOO</button>;
};
const Bar = () => {
  return <button>BAR</button>;
};
const Default = () => {
  return <button>DEFAULT</button>;
};
```

- **We'll now be creating an object that can be used as an enum.**

```
const ENUM_STATES = {
  foo: <Foo />,
  bar: <Bar />,
  default: <Default />
};
```

- **Let's now create a function that will take state as a parameter and return components based on "state". The "EnumState" function below is quite self-explanatory.**

```
function EnumState({ state }) {
  return <div>{ENUM_STATES[state]}</div>;
}
```

- **The state property key above helps us to retrieve the value from the object. You can see that it is much more readable compared to the switch case operator.**
- **Let's create an Enum component, which will pass the values of "state" to the function "EnumState".**

```
class Enum extends React.Component {
  render() {
    return (
      <div>
        <h1>Conditional Rendering with enums</h1>
        <EnumState state="default"></EnumState>
        <EnumState state="bar"></EnumState>
        <EnumState state="foo"></EnumState>
      </div>
    );
  }
}
ReactDOM.render(<Enum />, document.getElementById("app"));
```

## ➢ React Lists

- **Lists** are very useful when it comes to developing the UI of any website.
- Lists are mainly used for displaying menus on a website, for example, the navbar menu.
- In regular JavaScript, we can use arrays for creating lists.
- you will render lists with some type of loop.
- The JavaScript map() array method is generally the preferred method.

➢ **Example**

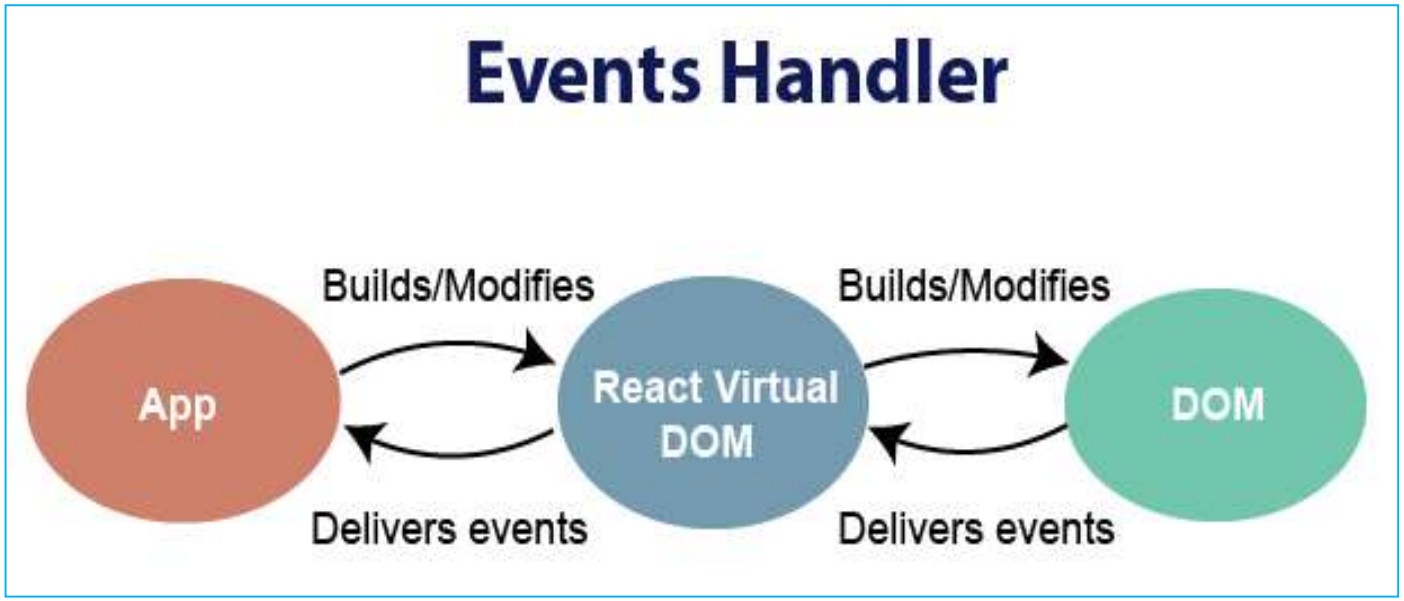
```
import React from 'react';
import ReactDOM from 'react-dom';
const numbers = [1,2,3,4,5];
const updatedNums = numbers.map((number)=>{
        return <li>{number}</li>;
});
ReactDOM.render(
        <ul>
                {updatedNums}
        </ul>,
        document.getElementById('root')
);
```

**Output**

- 1
- 2
- 3
- 4
- 5

## ➢ React Events

- An event is an action that could be triggered as a result of the user action or system generated event.
- For example, a mouse click, loading of a web page, pressing a key, window resizes, and other interactions are called events.
- React has its own event handling system which is very similar to handling events on DOM elements.
- The react event handling system is known as Synthetic Events. The synthetic event is a cross-browser wrapper of the browser's native event.




- Handling events with react have some syntactic differences from handling events on DOM. These are:
- React events are named as **camelCase** instead of **lowercase**.
- With JSX, a function is passed as the **event handler** instead of a **string**. For example:

- **Event declaration in plain HTML:**

```
<button onclick="showMessage()">
    Hello JavaTpoint
</button>
```

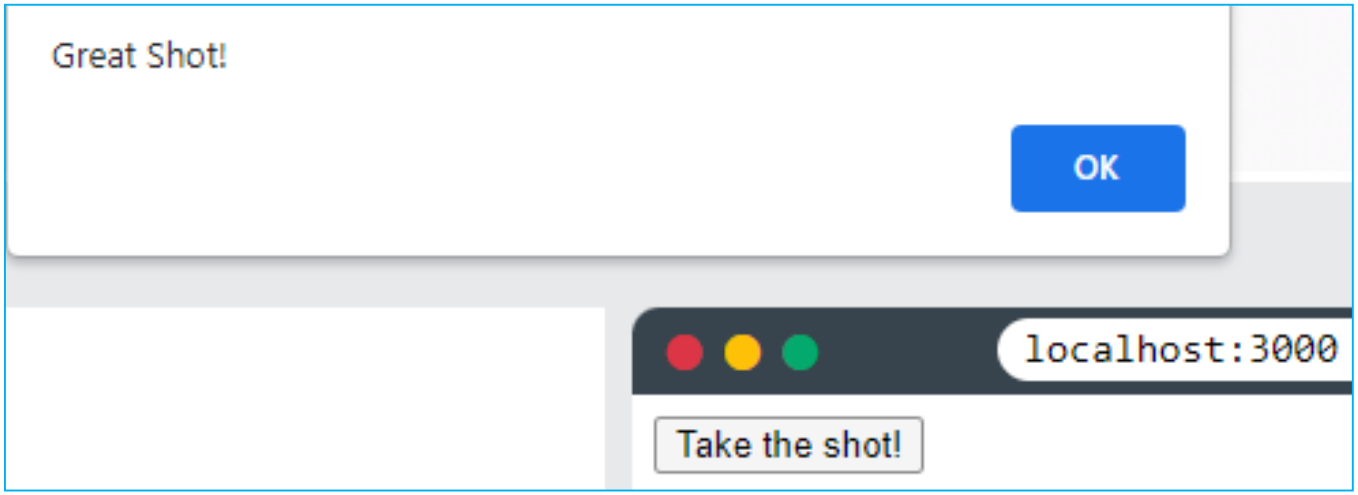
- **Event declaration in React:**

```
<button onClick={showMessage}>
    Hello JavaTpoint
</button>
```

## ➢ Example

```
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 />);
```

## ➢ Output



## ➢ Passing Arguments

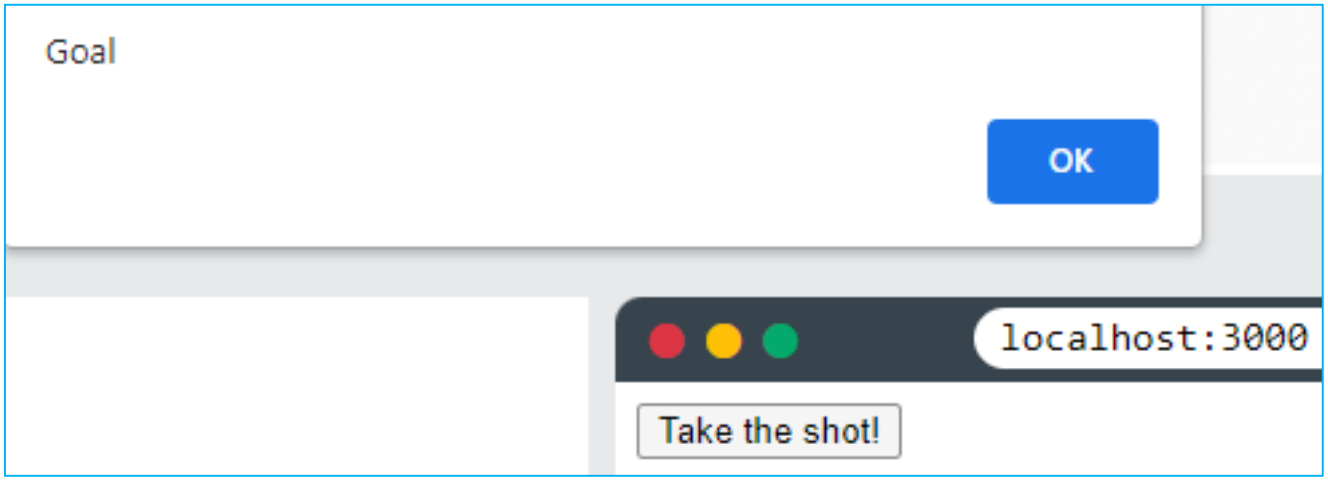
- To pass an argument to an event handler, use an arrow function.

### ❑ Example:

- Send "Goal!" as a parameter to the shoot function, using arrow function:

```
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 />);
```

## ➢ Output



## ➢ React Event Object

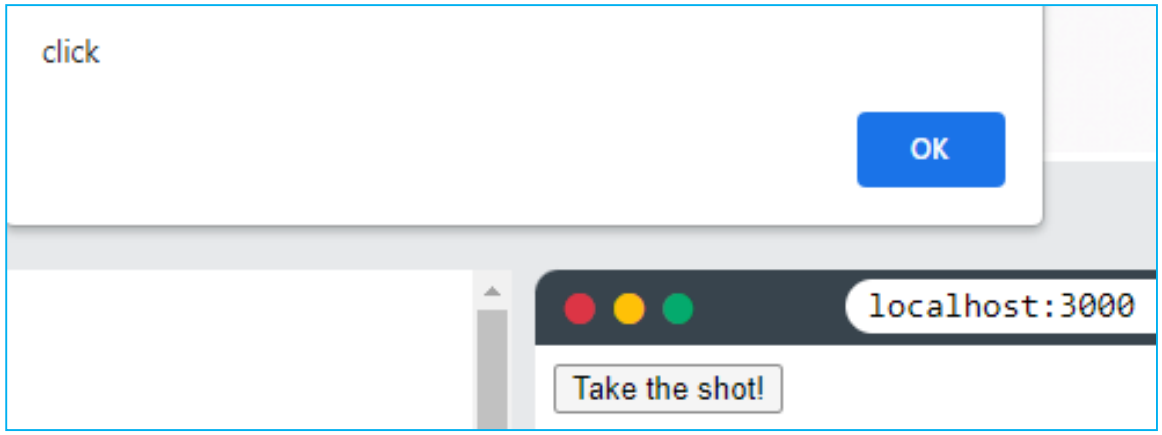
- Event handlers have access to the React event that triggered the function.
- In our example the event is the "click" event.

## ❑ Example:

- 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 />);
```

## ➢ Output



# Unit-2: Forms and Hooks in React.JS

**Unit-2: Forms and Hooks in React.JS**

2.3 Forms: (Adding forms, Handling forms, Submitting forms)

2.3.1 event.target.name and event. Target.event, React Memo

2.3.2 Components (TextArea, Drop down list (SELECT))

2.4 Hooks: Concepts and Advantages

2.4.1 useState, useEffect, useContext

2.4.2 useRef, useReducer, useCallback, useMemo

2.3.3 Hook: Building custom hook, advantages and use

## ➢ React Forms

- Forms are an integral part of any modern web application. It allows the users to interact with the application as well as gather information from the users.
- Forms can perform many tasks that depend on the nature of your business requirements and logic such as authentication of the user, adding user, searching, filtering, booking, ordering, etc.
- A form can contain text fields, buttons, checkbox, radio button, etc.

## ➢ Creating Form

- React offers a stateful, reactive approach to build a form.
- The component rather than the DOM usually handles the React form.
- In React, the form is usually implemented by using controlled components.

➢ There are mainly two types of form input in React.

- Uncontrolled component
- Controlled component

## ➢ Uncontrolled component

- The uncontrolled input is similar to the traditional HTML form inputs.
- The DOM itself handles the form data.
- Here, the HTML elements maintain their own state that will be updated when the input value changes.
- To write an uncontrolled component, you need to use a ref to get form values from the DOM.
- In other words, there is no need to write an event handler for every state update.
- You can use a ref to access the input field value of the form from the DOM.

## ➢ Controlled Component

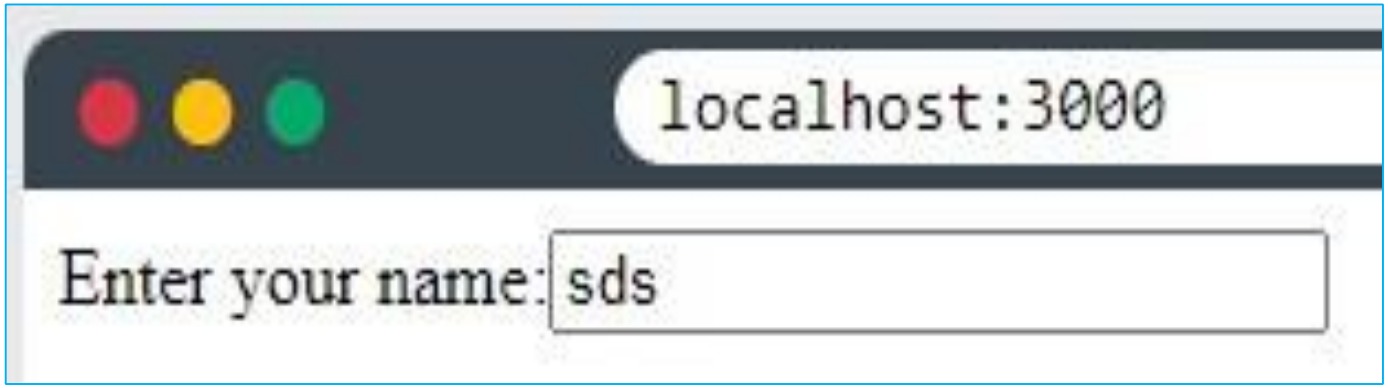
- In HTML, form elements typically maintain their own state and update it according to the user input.
- In the controlled component, the input form element is handled by the component rather than the DOM.
- Here, the mutable state is kept in the state property and will be updated only with **setState()** method.
- Controlled components have functions that govern the data passing into them on every **onChange event**, rather than grabbing the data only once, e.g., when you click a **submit button**.
- This data is then saved to state and updated with setState() method.
- This makes component have better control over the form elements and data.

## ➢ Example

- Add a form that allows users to enter their name:

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

function MyForm() {
  return (
    <form>
      <label>Enter your name:
        <input type="text" />
      </label>
    </form>
  )
}

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

➢ **Output**




- This will work as normal, the form will submit and the page will refresh.
- But this is generally not what we want to happen in React.
- We want to prevent this default behavior and let React control the form.

## ➢ Handling Forms

- Handling forms is about how you handle the data when it changes value or gets submitted.
- In HTML, form data is usually handled by the DOM.
- In React, form data is usually handled by the components.
- When the data is handled by the components, all the data is stored in the component state.
- You can control changes by adding event handlers in the onChange attribute.
- We can use the useState Hook to keep track of each inputs value and provide a "single source of truth" for the entire application.

➢ **Example:**

- Use the **useState** Hook to manage the input:

```
import { useState } from "react";
import ReactDOM from 'react-dom/client';

function MyForm() {
  const [name, setName] = useState("");

  return (
    <form>
      <label>Enter your name:
        <input
          type="text"
          value={name}
          onChange={(e) => setName(e.target.value)}
        />
      </label>
    </form>
  )
}

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

## ➢ Submitting Forms

- You can control the submit action by adding an event handler in the **onSubmit** attribute for the **<form>** :

## ➢ Example:

- Add a submit button and an event handler in the onSubmit attribute:

```
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 your 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(<MyForm />);
```

## ➢ Output

## ➢ event.target.name

- The target property returns the element **where the event occured**.
- The target property is read-only.
- The target property returns the element on which the event occurred, opposed to the currentTarget property, which returns the element whose event listener triggered the event.

## ➢ Syntax

*event*.target

- In React, event. target refers to the HTML element that triggered the event. For instance, if you have a button component in React, you can add an onClick listener to it to listen for click events.
- When the button is clicked, the event object that is passed to the handler function will have the event.
- **Example**

Get the name of the element where the event occurred:

```
let text = event.target.tagName;
```

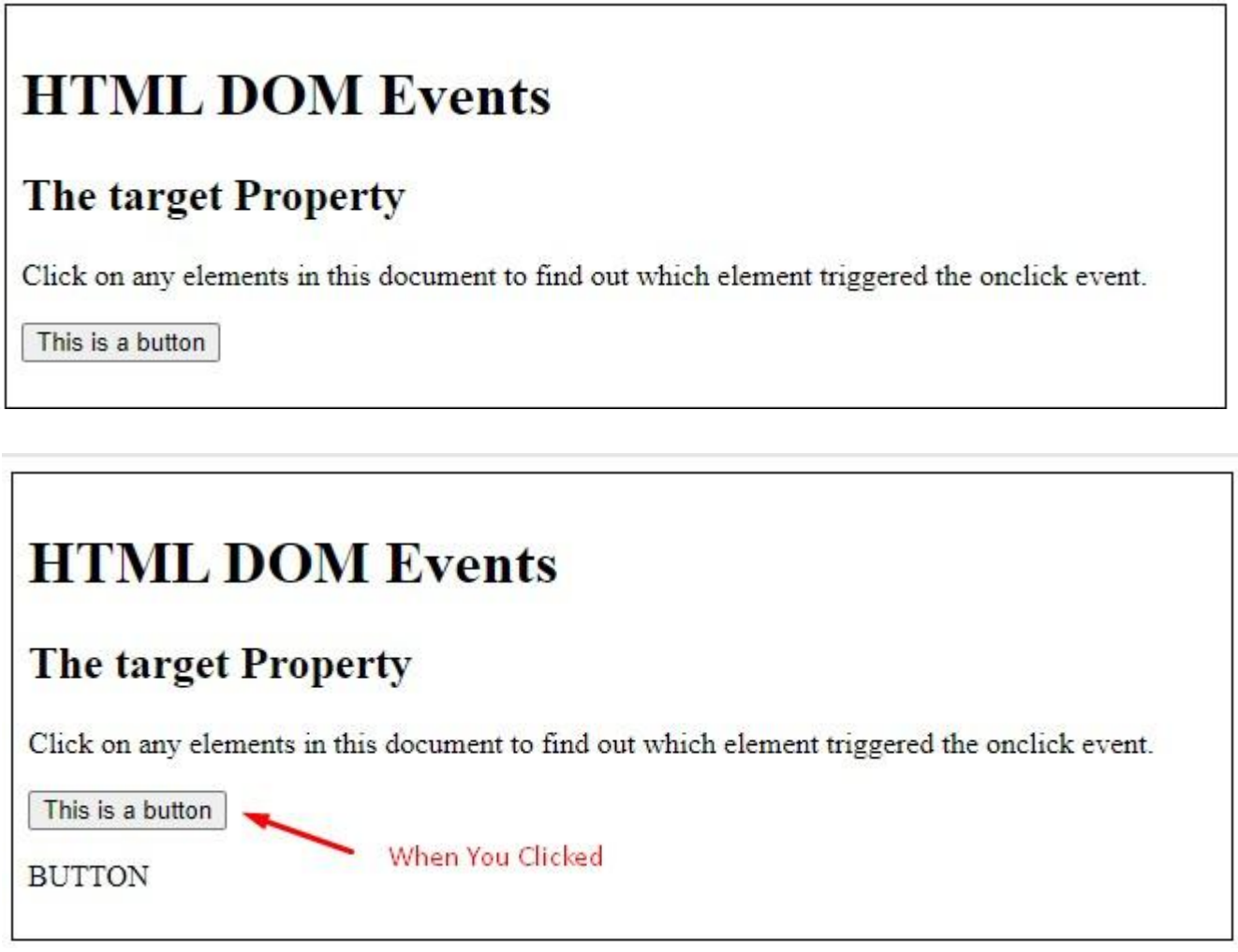
Get the element where the event occurred:

```
const element = event.target;
```

➢ **Example**

```
<!DOCTYPE html>
<html>
<body onclick="myFunction(event)" style="border:1px solid black;padding:8px">
<h1>HTML DOM Events</h1>
<h2>The target Property</h2>

<p>Click on any elements in this document to find out which element triggered
the onclick event.</p>

<button>This is a button</button>

<p id="demo"></p>

<script>
function myFunction(event) {
  let text = event.target.nodeName;
  document.getElementById("demo").innerHTML = text;
}
</script>

</body>
</html>
```

## ➢ Output


HTML DOM Events
The target Property
Click on any elements in this document to find out which element triggered the onclick event.
This is a button
HTML DOM Events
The target Property
Click on any elements in this document to find out which element triggered the onclick event.
This is a button
When You Clicked
BUTTON

## ➢ React Memo

- Using memo will cause React to skip rendering a component if its props have not changed.
- This can improve performance.
- React Memo is a higher-order component that wraps around a component to memoize the rendered output and avoid unnecessary renderings.
- This improves performance because it memoizes the result and skips rendering to reuse the last rendered result.
- **React**.**memo** is a function that you can use to optimize the render performance of pure function components and hooks.

- **Example:**

```
const MyComponent = React.memo(function MyComponent(props) {

/* render using props */

});
```

- **Components (TextArea, Drop down list (SELECT))**
- **TextArea**

- One frequently used form control is textarea, which is used to get multi-line input from a user. It's different from a normal text input, which allows only single-line input.
- A good example of a use case for textarea is an address field.
- TextArea is a controlled component.
- This means that the visible text will always match the contents of the value prop. In this example, notice how value is stored within this. state .
- The onChange function will set the new value when the user enters or removes a character in the textarea.

- **Example**

```
function TextAreaExample() { const [value,
  setValue] = React.useState(''); return (
    <div>
      <Label text="Business description">
        <TextArea
          value={value}
          placeholder="Tell us about your business"
          onChange={v => setValue(v)}
        />
      </Label>
    </div>
  );
}
```

## ➢ Dropdowns

- A Dropdown in React JS list is a graphical user interface element that gives users a list of possibilities and allows them to select one value from the list.
- There are two statuses in the dropdown menu: active and inactive.
- Only one discount is displayed while the dropdown list is fixed.
- By activating the list, all accessible options in the list are revealed, and This situation may alter this value.
- You may use a dropdown list as one of many different lists in your apps.
- It's an excellent technique to provide various options and let the user select one from the list.

## ➢ Creating a Dropdown in React

**Js** import * as React from 'react';

```
const App = () => { return (
 <div>
 <select>
     <option value="fruit">Fruit</option>
     <option value="vegetable">Vegetable</option>
     <option value="meat">Meat</option>
  </select>
   </div>
 );
};
export default App;
```

## ➢ Hooks

- Hooks are used to give functional components an access to use the states and are used to manage side-effects in React.
- They let developers use state and other React features without writing a class For example- State of a component It is important to note that hooks are not used inside the classes.
- Hooks allow us to "hook" into React features such as state and lifecycle methods.
- Hooks are backward-compatible, which means it does not contain any breaking changes. Also, it does not replace your knowledge of React concepts.

## ➢ When to use a Hooks

- If you write a function component, and then you want to add some state to it, previously you do this by converting it to a class. But, now you can do it by using a Hook inside the existing function component.

## ➢ Rules for using hooks

- Only functional components can use hooks
- Calling of hooks should always be done at top level of components
- Hooks should not be inside conditional statements

## ➢ Advantages of Using React Hooks

### ❑ Simplified Code

- One of the primary benefits of using React Hooks is that it simplifies the codebase.
- Hooks eliminate the need for class components, which often require more boilerplate code and can be harder to read and understand. With Hooks, developers can write cleaner, more intuitive code.

### ➢ Improved Reusability

- React Hooks make it easier to reuse stateful logic across components. With custom hooks, developers can extract component logic into reusable functions.
- This promotes cleaner, more modular code, and reduces duplication.

## ❑ Easier Testing and Debugging

- Functional components that use Hooks are generally easier to test and debug than class components.
- Since Hooks promote separation of concerns and a more functional programming style, developers can write more predictable and testable code.

## ❑ Reduced Bundle Size

- By using functional components with Hooks instead of class components, developers can reduce the overall size of their application bundle.
- This can lead to faster load times and improved performance for users.

## ➢ React Hooks Installation

❑ To use React Hooks, we need to run the following commands:

```
$ npm install react@16.8.0-alpha.1 --save

$ npm install react-dom@16.8.0-alpha.1 --save
```

- The above command will install the latest React and React-DOM alpha versions which support React Hooks.
- Make sure the **package.json** file lists the React and React-DOM dependencies as given below.

```
"react": "^16.8.0-alpha.1",

"react-dom": "^16.8.0-alpha.1",
```

## ➢ React.js — Basic Hooks (useState, useEffect, & useContext)

### ❑ useState

- The React useState Hook allows us to track state in a function component.
- State generally refers to data or properties that need to be tracking in an application.

### ➢ Import useState

- To use the useState Hook, we first need to import it into our component.

### ➢ Example:

At the top of your component, `import` the `useState` Hook.

```
import { useState } from "react";
```

## ➢ Initialize useState

- We initialize our state by calling useState in our function component.
- useState accepts an initial state and returns two values:
- The current state.
- A function that updates the state.

### Example:

Initialize state at the top of the function component.

```
import { useState } from "react";

function FavoriteColor() {
  const [color, setColor] = useState("");
}
```

## ➢ React useEffect Hooks

- The useEffect Hook allows you to perform side effects in your components.
- Some examples of side effects are: fetching data, directly updating the DOM, and timers.
- useEffect accepts two arguments. The second argument is optional.
- useEffect(<function>, <dependency>)

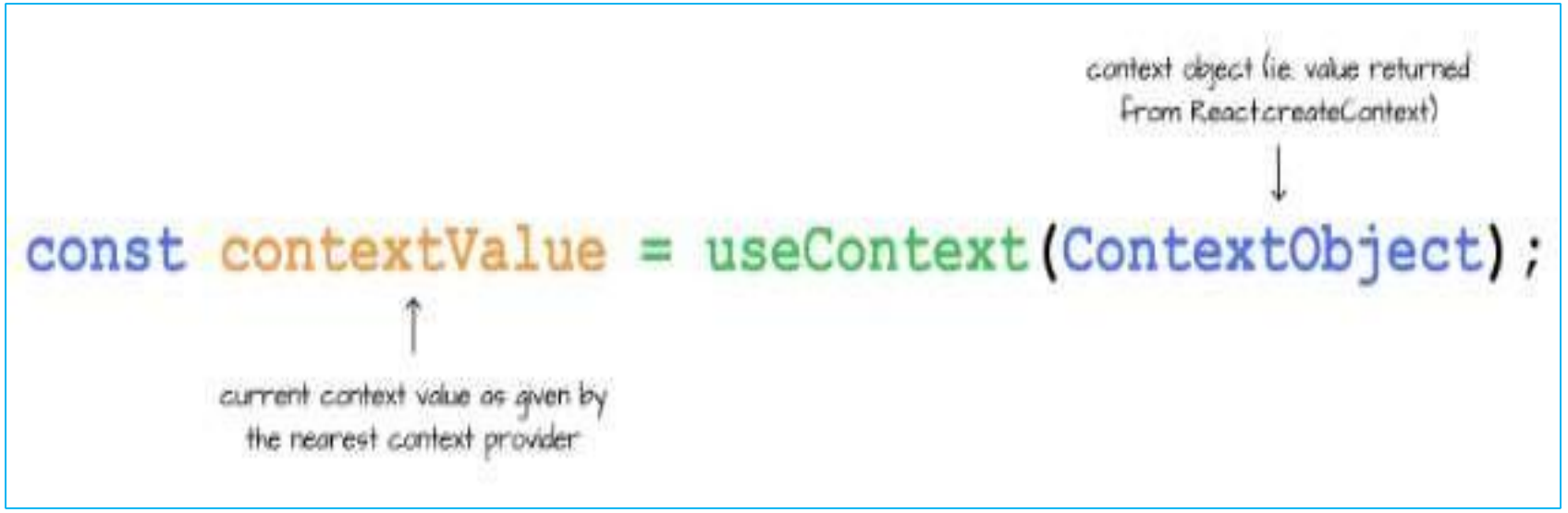
- **Example**

```
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 have rendered {count} times!</h1>;
}

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

- **Output**

## ➢ useContext Hook

- The useContext Hook provides function components access to the **context** value for a context object. It:
- Takes the context object (i.e., value returned from React.createContext) as the one argument it accepts.
- And returns the current context value as given by the nearest context provider.



- With this in mind, we'll have the <Child /> component in our example use the useContext hook to access the data property available in our application context and render its value in its markup.

- **Example**

```
import React, { createContext, useContext } from
"react"; const Context = createContext(); const Child = ()
=> {
  const context = useContext(Context);
  return <div>{context.data}</div>;
};
const App = () => {
  return (
    <Context.Provider value={{ data: "Data from context!" }}>
      <Child />
    </Context.Provider>
  );
};
```

## ➢ React useRef Hook

- The useRef is a hook that allows to directly create a reference to the DOM element in the functional component.
- The useRef hook is a new addition in React 16.8. To learn useRef the user should be aware about refs in React. Unlike useState if we change a value in useRef it will not re-render the webpage

## ➢ Reasons to use useRef hook

- The main use of useRef hook is to access the DOM elements in a more efficient way as compared to simple refs. Since useRef hooks preserve value across various re-renders and do not cause re-renders whenever a value is changed they make the application faster and helps in caching and storing previous values

## ➢ Importing useRef hook

- To import the useRef hook, write the following code at the top level of your component

```
import { useRef } from 'react';
```

- **Example**

```
import React, {Fragment, useRef} from 'react'; function
App() {
// Creating a ref object using useRef hook
const focusPoint = useRef(null); const
onClickHandler = () => {
focusPoint.current.value =
    "The quick brown fox jumps over the lazy dog";
    focusPoint.current.focus();
};
return (
    <Fragment>
    <div> <button onClick={onClickHandler}> ACTION </button> </div>
```

```
<label>
    Click on the action button to focus
    and populate the text.
    </label><br/>
    <textarea ref={focusPoint} />
    </Fragment>
);
};
export default App;
```

## ➢ useReducer

- The useReducer(reducer, initialState) hook accepts 2 arguments: the *reducer* function and the *initial state*. The hook then returns an array of 2 items: the current state and the *dispatch* function.
- The useReducer Hook is similar to the useState Hook.
- It allows for custom state logic.
- If you find yourself keeping track of multiple pieces of state that rely on complex logic, useReducer may be useful.

## ➢ Syntax

- The useReducer Hook accepts two arguments.

useReducer(<reducer>, <initialState>)

## ➢ Example

```
import { useReducer } from 'react';

function MyComponent() {
  const [state, dispatch] = useReducer(reducer, initialState);

  const action = {
    type: 'ActionType'
  };

  return (
    <button onClick={() => dispatch(action)}>
      Click me
    </button>
  );
}
```

## ➢ useCallback

- The React useCallback Hook returns a memoized callback function.
- Think of memoization as caching a value so that it does not need to be recalculated.
- This allows us to isolate resource intensive functions so that they will not automatically run on every render.
- The useCallback Hook only runs when one of its dependencies update.
- This can improve performance.

- **Syntax:**

```
const memoizedCallback = useCallback(
  () => {
    doSomething(a, b);
  },
  [a, b],
);
```

- **Example**

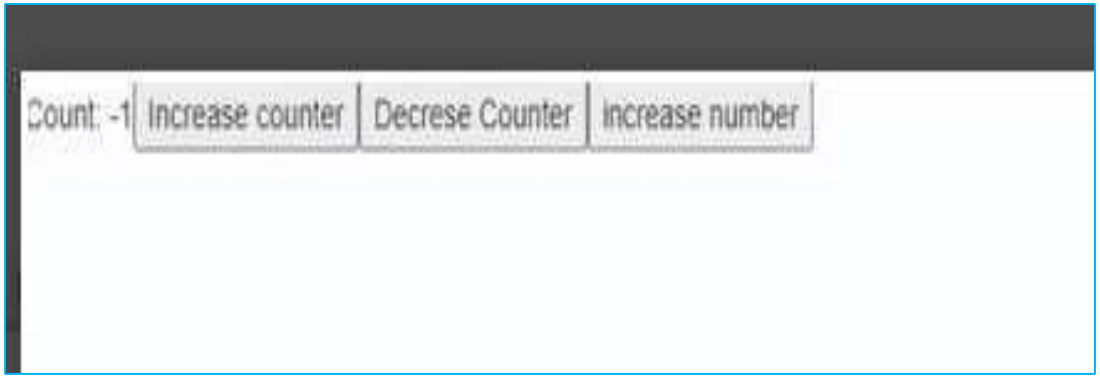
```
import React, { useState, useCallback } from 'react'

var funccount = new Set(); const App = () => {

const [count, setCount] = useState(0) const

[number, setNumber] = useState(0) const

incrementCounter = useCallback(() => {

setCount(count + 1)

}, [count])

const decrementCounter = useCallback(() => {

setCount(count - 1)

}, [count])
```

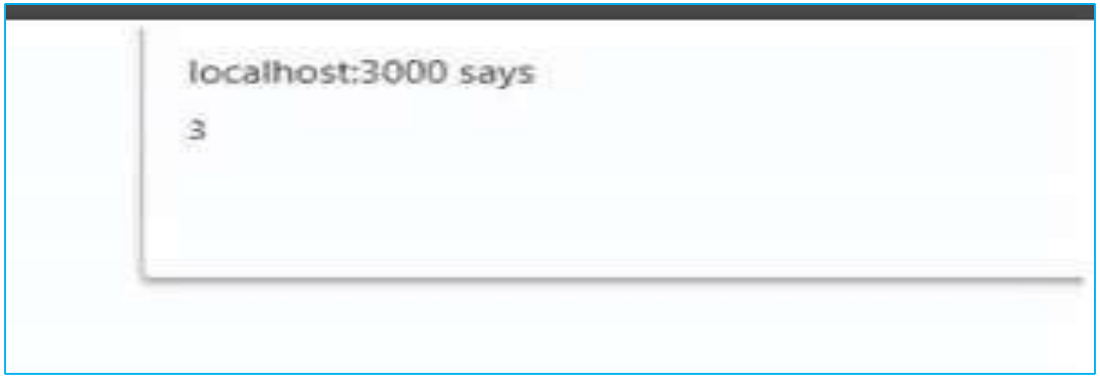
```
const incrementNumber = useCallback(() => {

setNumber(number + 1)

}, [number])
funccount.add(incrementCounter);
funccount.add(decrementCounter);
funccount.add(incrementNumber);
alert(funccount.size);
```

```
return (
    <div>
    Count: {count}
    <button onClick={incrementCounter}>
        Increase counter
    </button>
    <button
    onClick={decrementCounter}>
        Decrease Counter
    </button>
    <button  onClick={incrementNumber}>
        increase number
    </button>
```

```
    </div>)} export
default App;
```

## ➢ Output


Count: -1
Increase counter
Decrese Counter
increase number


localhost:3000 says
3

## ➢ useMemo

- The React useMemo Hook returns a memoized value. Think of memoization as caching a value so that it does not need to be recalculated.
- The useMemo Hook only runs when one of its dependencies update. This can improve performance.
- The React useMemo Hook returns a memoized value.
- Think of memoization as caching a value so that it does not need to be recalculated.
- The useMemo Hook only runs when one of its dependencies update.
- This can improve performance.

➢ **Example**

```
import { useState, useMemo } from 'react';

export function CalculateFactorial() {
  const [number, setNumber] = useState(1);
  const [inc, setInc] = useState(0);

  const factorial = useMemo(() => factorialOf(number), [number]);

  const onChange = event => {
    setNumber(Number(event.target.value));
  };
  const onClick = () => setInc(i => i + 1);

  return (
    <div>
      Factorial of
      <input type="number" value={number} onChange={onChange} />
      is {factorial}
      <button onClick={onClick}>Re-render</button>
    </div>
  );
}

function factorialOf(n) {
  console.log('factorialOf(n) called!');
  return n <= 0 ? 1 : n * factorialOf(n - 1);
}
```

## ➢ Building a custom hook

- Creating a custom hook is the same as creating a JavaScript function whose name starts with “use”. It can use other hooks inside it, return anything you want it to return, take anything as parameters.
- **Note:** It is important to name your custom hooks starting with “use”, because without it React can’t realize that it is a custom hook and therefore can’t apply the rules of hooks to it. So, you should name it starting with “use”.

## ➢ Example

```
// First Component

import React from "react";

// importing the custom hook
import useCustomHook from "./useCustomHook";

function FirstComponent(props){

    // ClickedButton = resetCounter;
    const clickedButton = useCustomHook(0 , "FirstComponent");

    return (
        <div>
            <h1> This is the First Component</h1>
            <button onClick={clickedButton}>
                  Click here!
            </button>
        </div>
    );
}

export default FirstComponent;
```

## ➢ Advantages of Building a custom hook

### ❑ Reusability

- Custom React JS hooks offer reusability as when a custom hook is created, it can be reused easily, which ensures clean code and reduces the time to write the code.
- It also enhances the rendering speed of the code as a custom hook does not need to be rendered again and again while rendering the whole code.

## ➢ Readability

- Instead of using High-Order Components (HOCs), one can use custom hooks to improve the readability of the code.
- Complex codes can become hard to read if layers of providers surround the components, consumers, HOCs, render props, and other abstractions, generally known as wrapper hell.
- On the other hand, using custom React JS hooks can provide cleaner logic and a better way to understand the relationship between data and the component.

## ❑ Testability

- Generally, the test containers and the presentational components are tested separately in React.
- This is not a trouble when it comes to unit tests. But, if a container contains several HOCs, it becomes difficult as you will have to test the containers and the components together to do the integration tests.

# Unit-3: Fundamentals of Angular

**Unit-3: Fundamentals of Angular**

3.1 Concepts and characteristics of Angular JS

3.1.1 Expressions in Angular JS (Numbers, Strings, Objects, Arrays)

3.1.2 Setting up Environment, Angular JS Filters

3.1.3 Understanding MVC (Model, View, Controller) architecture

3.2 AngularJS Directive (ng-app, ng-init, ng-controller, ng-model, ng-repeat)

3.2.1 Some other dire ctives: ng-class, ng-animate, ng-show, ng-hide

3.2.2 Expressions and Controllers

3.2.3 Filters (Uppercase, Lowercase, Currency, order by)

## ➢ History of AngularJS:

- AngularJS was originally developed in 2008-2009 by Miško Hevery and Adam Abrons at Brat Tech LLC, as software for the online JSON storage service, in order to ease to development of the applications for the enterprise, that has been valued by the megabyte.
- It is now maintained by Google.
- AngularJS was released with version 1.6, which contains the component-based application architecture concept.

## ➢ AngularJS

- **AngularJS** is a Javascript open-source front-end structural framework that is mainly used to develop single-page web applications(SPAs).
- It is a continuously growing and expanding framework which provides better ways for developing web applications.
- It changes the static HTML to dynamic HTML.
- Its features like dynamic binding and dependency injection eliminate the need for code that we have to write otherwise.

- AngularJS is rapidly growing and because of this reason, we have different versions of AngularJS with the latest stable being 1.7.9.
- It is also important to note that Angular is different from AngularJS.
- It is an open-source project which can be freely used and changed by anyone.

- It extends HTML attributes with Directives, and data is bound with HTML.

- AngularJS is a JavaScript framework written in JavaScript.

## ➢ Why use AngularJS?

### ❑ Easy to work with:

- All you need to know to work with AngularJS is the basics of HTML, CSS, and Javascript, not necessary to be an expert in these technologies.

### ❑ Time-saving:

- AngularJs allows us to work with components and hence we can use them again which saves time and unnecessary code.

### ❑ Ready to use a template:

- AngularJs is mainly plain HTML, and it mainly makes use of the plain HTML template and passes it to the DOM and then the AngularJS compiler. It traverses the templates and then they are ready to use.

## ❑ Directives:

- AngularJS's directives allow you to extend HTML with custom elements and attributes.
- This enables you to create reusable components and define custom behaviors for your application.
- Directives make it easier to manipulate the DOM, handle events, and encapsulate complex UI logic within a single component.

## ➢ Features of AngularJS

• The core features of AngularJS are as follows −

### ❑ Data-binding

• It is the automatic synchronization of data between model and view components.

### ❑ Scope

• These are objects that refer to the model. They act as a glue between controller and view.

### ❑ Controller

• These are JavaScript functions bound to a particular scope.

### ❑ Services

• AngularJS comes with several built-in services such as $http to make a XMLHttpRequests. These are singleton objects which are instantiated only once in app.

## ❑ Filters

• These select a subset of items from an array and returns a new array.

## ❑ Directives

- Directives are markers on DOM elements such as elements, attributes, css, and more.
- These can be used to create custom HTML tags that serve as new, custom widgets. AngularJS has built-in directives such as ngBind, ngModel, etc.

## ❑ Templates

- These are the rendered view with information from the controller and model.
- These can be a single file (such as index.html) or multiple views in one page using *partials*.

- **Routing** −

- It is concept of switching views.

- **Model View Whatever** −

- MVW is a design pattern for dividing an application into different parts called Model, View, and Controller, each with distinct responsibilities.
- AngularJS does not implement MVC in the traditional sense, but rather something closer to MVVM (Model-View-ViewModel). The Angular JS team refers it humorously as Model View Whatever.

- **Deep Linking** −

- Deep linking allows to encode the state of application in the URL so that it can be bookmarked.
- The application can then be restored from the URL to the same state.

## ❑ Dependency Injection –

- AngularJS has a built-in dependency injection subsystem that helps the developer to create, understand, and test the applications easily.

## ➢ AngularJS - Environment Setup

- When you open the link https://angularjs.org/, you will see there are two options to download AngularJS library –

- **View on GitHub** − By clicking on this button, you are diverted to GitHub and get all the latest scripts.
- **Download AngularJS 1** − By clicking on this button, a screen you get to see a dialog box shown as −

- **CDN access** – You also have access to a CDN. The CDN gives you access to regional data centers. In this case, the Google host. The CDN transfers the responsibility of hosting files from your own servers to a series of external ones. It also offers an advantage that if the visitor of your web page has already downloaded a copy of AngularJS from the same CDN, there is no need to redownload it.

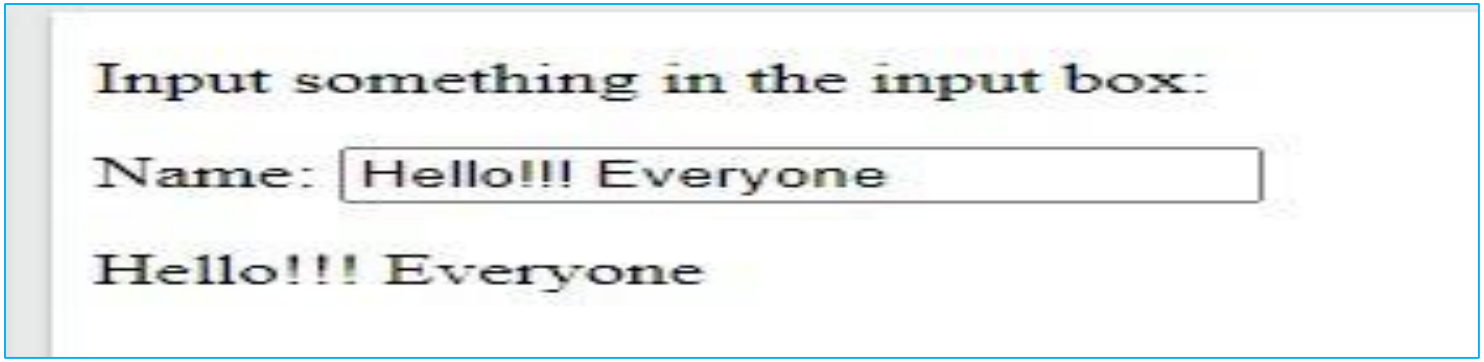
## ➢ AngularJS Extends HTML

- AngularJS extends HTML with **ng-directives**.
- The **ng-app** directive defines an AngularJS application.
- The **ng-model** directive binds the value of HTML controls (input, select, textarea) to application data.
- The **ng-bind** directive binds application data to the HTML view.

➢ **Example**

```
<!DOCTYPE html>
<html>
<script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js">
</script>
<body>

<div ng-app="">

<p>Input something in the input box:</p>
<p>Name: <input type="text" ng-model="name"></p>
<p ng-bind="name"></p>

</div>

</body>
</html>
```

## ➢ Output


Input something in the input box:
Name: Hello!!! Everyone
Hello!!! Everyone

## ➢ Expressions in Angular JS (Numbers, Strings, Objects, Arrays)

- Expressions are variables which were defined in the double braces {{ }}.

## ➢ Syntax:

- A simple example of an expression is {{5 + 6}}.
- Angular.JS expressions are used to bind data to HTML the same way as the ng-bind directive.
- AngularJS displays the data exactly at the place where the expression is placed.

## ➢ Example

```
</head>
<body>
<script src="lib/angular.js"></script>
<script src="lib/jquery-1.11.3.min.js"></script>
<script src="lib/bootstrap.js"></script>

<h1> Guru99 Global Event</h1>

<div ng-app="">

    Addition :

    {{ 6+9 }}

<div >
</body>
</html>
```

1

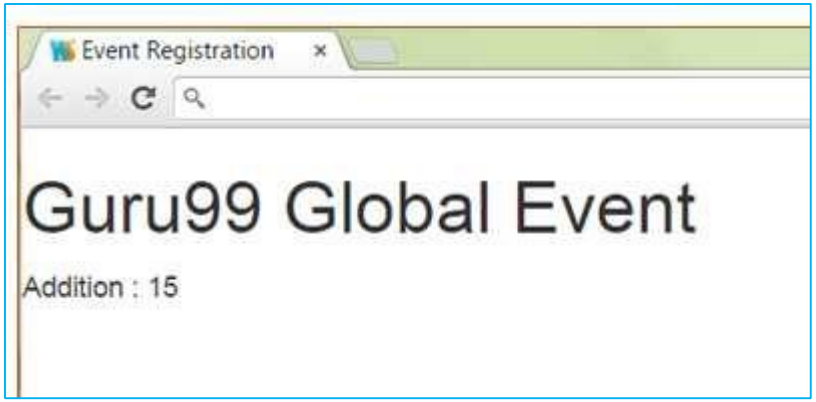
The ng-app is not defined with any application name.

2

A simple addition operation inside an expression

➢ **Example**

**Output**




```
<!DOCTYPE html>
<html>
<head><meta chrset="UTF 8">
   <title>Event Registration</title> </head>
<body>
    <script src="https://code.angularjs.org/1.6.9/angular-route.js"></script>
    <script src="https://code.angularjs.org/1.6.9/angular.min.js"></script>
    <h1> Guru99 Global Event</h1>
    <div ng-app=""> Addition : {{6+9}} </div>
</body>
</html>
```

## ➢ Angular.JS Numbers

- Expressions can be used to work with numbers as well. Let's look at an example of Angular.JS expressions with numbers.
- In this example, we just want to show a simple multiplication of 2 number variables called margin and profit and displayed their multiplied value.

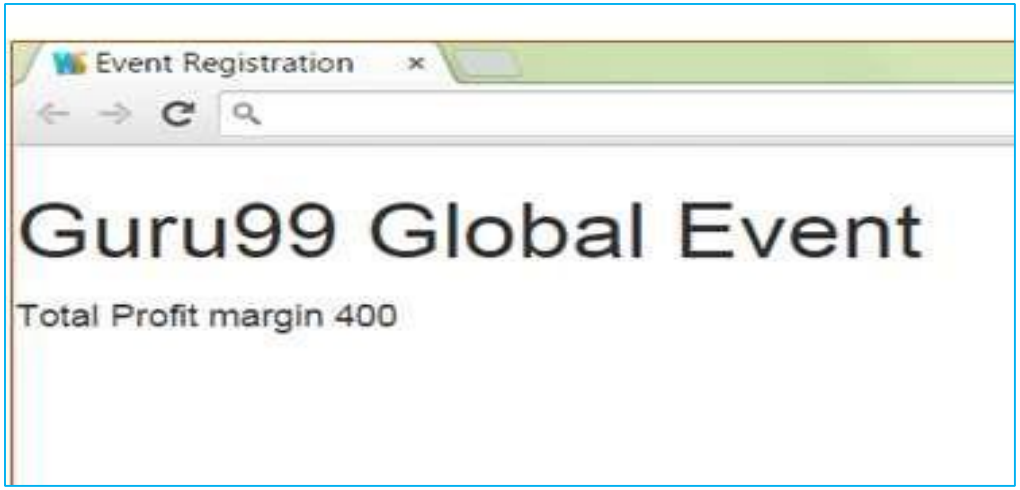
## Example

```
</head>
<body>
<script src="lib/angular.js"></script>
<script src="lib/jquery-1.11.3.min.js"></script>
<script src="lib/bootstrap.js"></script>

<h1> Guru99 Global Event</h1>

<div ng-app="" ng-init="margin=2;profit=200">

    Total Profit margin

    {{ margin*profit }}

<div>
</body>
</html>
```

1 ng-init tag to initialize variables

2 variables being used in an expression

## ➢ Output


Event Registration
Guru99 Global Event
Total Profit margin 400

## ➢ AngularJS Strings

- Expressions can be used to work with strings as well. Let's look at an example of Angular JS expressions with strings.
- In this example, we are going to define 2 strings of "firstName" and "lastName" and display them using expressions accordingly.

## ➢ Example

```
</head>
<body>
<script src="lib/angular.js"></script>
<script src="lib/jquery-1.11.3.min.js"></script>
<script src="lib/bootstrap.js"></script>

<h1> Guru99 Global Event</h1>

<div ng-app="" ng-init="firstName='Guru';lastName='99'">

      
   First Name : {{ firstName }}<br>   
   Last Name  : {{ lastName }}

</script>
</body>
</html>
```

1

ng-init tag to initialize string variables

2

variables names being displayed.

## ➢ Angular.JS Objects

- Expressions can be used to work with JavaScript objects as well.
- Let’s look at an example of Angular.JS expressions with javascript objects. A javascript object consists of a name-value pair.
- Below is an example of the syntax of a javascript object.

## ➢ Syntax:

```
var car = {type:"Ford", model:"Explorer", color:"White"};
```

## ➢ Example

```
</head>
<body>
<script src="lib/angular.js"></script>
<script src="lib/jquery-1.11.3.min.js"></script>
<script src="lib/bootstrap.js"></script>

<h1> Guru99 Global Event</h1>

<div ng-app="" ng-init="person={firstName:'Guru',lastName:'99'}">


       
    First Name : {{ person.firstName  }}<br>   

    Last Name  : {{ person.lastName  }}

</script>
</body>
</html>
```

1

Creating an object variable with 2 key value pairs

2

Accessing each value of the object person via it's key value pairs

## ➢ Output


Event Registration
Guru99 Global Event
First Name : Guru
Last Name : 99

## ➢ AngularJS Arrays

- Expressions can be used to work with arrays as well. Let's look at an example of Angular JS expressions with arrays.
- In this example, we are going to define an array which is going to hold the marks of a student in 3 subjects. In the view, we will display the value of these marks accordingly.

## ➢ Example

```
<head>
    <meta charset="UTF-8">
    <title>Event Registration</title>
    <link rel="stylesheet" href="css/bootstrap.css"
</head>
<body>
<script src="lib/angular.js"></script>
<script src="lib/jquery-1.11.3.min.js"></script>
<script src="lib/bootstrap.js"></script>

<h1> Guru99 Global Event</h1>

<div ng-app="" ng-init="marks=[1,15,19]">

    Student Marks<br>   

    Subject1 : {{ marks[0]  }}<br>   
    Subject2 : {{ marks[1]  }}<br>   
    Subject3 : {{ marks[2]  }}<br>   

</script>
</body>
</html>
```

1 Initializing an array using the ng-init directive

2 Accessing each array value

## ➢ Output


Event Registration
Guru99 Global Event
Student Marks
Subject1 : 1
Subject2 : 15
Subject3 : 19

## ➢ AngularJS Filters

- AngularJS provides filters to transform data:
- **currency** Format a number to a currency format.
- **date** Format a date to a specified format.
- **filter** Select a subset of items from an array.
- **json** Format an object to a JSON string.
- **limitTo** Limits an array/string, into a specified number of elements/characters.
- **lowercase** Format a string to lower case.
- **number** Format a number to a string.
- **orderBy** Orders an array by an expression.
- **uppercase** Format a string to upper case.

## ➢ Adding Filters to Expressions

- Filters can be added to expressions by using the pipe character |, followed by a filter.
- The uppercase filter format strings to upper case:

- **Example**

```
<!DOCTYPE html>
<html>
<script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js">
</script>
<body>

<div ng-app="myApp" ng-controller="personCtrl">

<p>The name is {{ lastName | uppercase }}</p>

</div>

<script>
angular.module('myApp', []).controller('personCtrl', function($scope) {
    $scope.firstName = "John",
    $scope.lastName = "Doe"
});
</script>

</body>
</html>
```

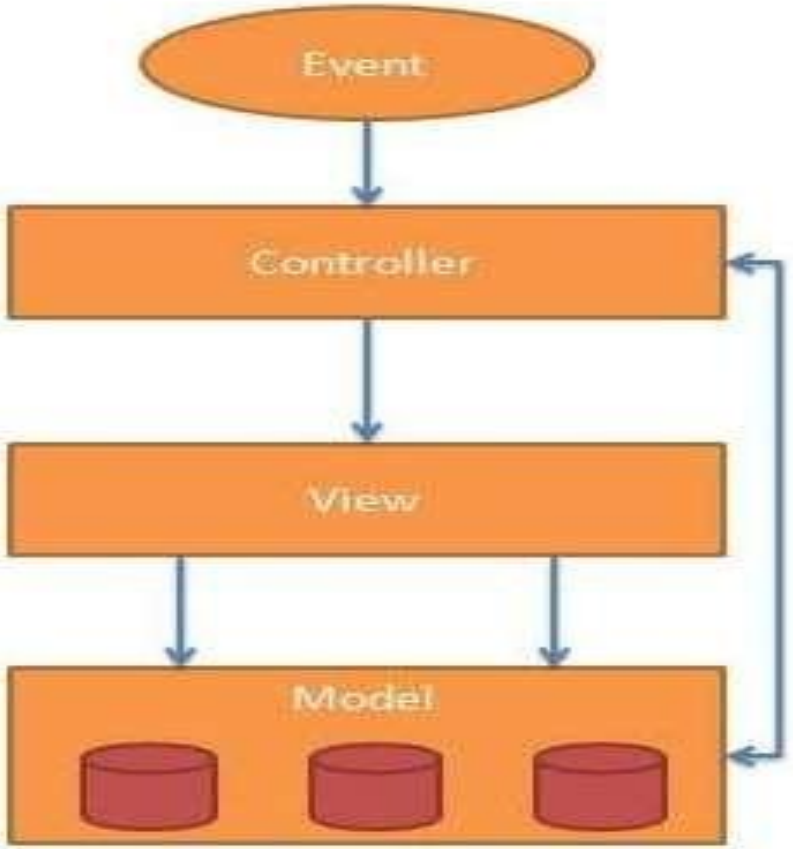
- **Output**

```
The name is DOE
```

## ➢ Understanding MVC (Model, View, Controller) architecture

## ➢ AngularJS - MVC Architecture

- MVC stands for Model View Controller.
- It is a software design pattern for developing web applications.
- It is very popular because it isolates the application logic from the user interface layer and supports separation of concerns.
- **M**odel **V**iew **C**ontroller or MVC as it is popularly called, is a software design pattern for developing web applications.
- A Model View Controller pattern is made up of the following three parts −




- **The Model**

- The model is responsible for managing application data. It responds to the request from view and to the instructions from controller to update itself.

## ➢ The View

- A presentation of data in a particular format, triggered by the controller's decision to present the data.
- They are script-based template systems such as JSP, ASP, PHP and very easy to integrate with AJAX technology.

## ➢ The Controller

- The controller responds to user input and performs interactions on the data model objects.
- The controller receives input, validates it, and then performs business operations that modify the state of the data model.
- AngularJS is a MVC based framework. In the coming chapters, we will see how AngularJS uses MVC methodology.

## ➢ AngularJS – Directives

### ❑ ng-app directive

- The ng-app directive starts an AngularJS Application.
- It defines the root element. It automatically initializes or bootstraps the application when the web page containing AngularJS Application is loaded.
- It is also used to load various AngularJS modules in AngularJS Application.
- In the following example, we define a default AngularJS application using ng-app attribute of a  element.

```
<div ng-app = "">
   ...
</div>
```

## ❑ ng-init directive

- ng-init directive initializes an AngularJS Application data. It defines the initial values for an AngularJS application.
- In following example, we'll initialize an array of countries. We're using JSON syntax to define array of countries.

```
<div ng-app = "" ng-init = "countries = [{locale:'en-IND',name:'India'}, {locale:'er
PAK',name:'Pakistan'}, {locale:'en-AUS',name:'Australia'}]">
  ...
</div>
```

## ❑ ng-model directive:

- ng-model directive defines the model/variable to be used in AngularJS Application.
- In following example, we've defined a model named "name".

```
<div ng-app = "">
  ...
  <p>Enter your Name: <input type = "text" ng-model = "name"></p>
</div>
```

## ❑ ng-repeat directive

- ng-repeat directive repeats html elements for each item in a collection. In following example, we've iterated over array of countries.

```
<div ng-app = "">
  ...
  <p>List of Countries with locale:</p>

  <ol>
    <li ng-repeat = "country in countries">
      {{ 'Country: ' + country.name + ', Locale: ' + country.locale }}
    </li>
  </ol>
```

## ❑ ng-controller Directive

- The AngularJS ng-controller directive adds a controller class to the view (your application). It is the key aspect which specifies the principles behind the Model-View-Controller design pattern.
- It facilitates you to write code and make functions and variables, which will be parts of an object, available inside the current HTML element. This object is called scope.
- This is supported by all HTML elements.

### ➢ Syntax:

```
<element ng-controller="expression"></element>
```

## ➢ Example

```
<!DOCTYPE html>
<html>
<script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js">
</script>
<body>

<div ng-app="myApp" ng-controller="myCtrl">

Full Name: {{firstName + " " + lastName}}

</div>

<script>
var app = angular.module('myApp', []);
app.controller('myCtrl', function($scope) {
    $scope.firstName = "John";
    $scope.lastName = "Doe";
});
</script>

<p>This example shows how to define a controller, and how to use variables
made for the scope.</p>

</body>
</html>
```

## ➢ Output

Full Name: John Doe

This example shows how to define a controller, and how to use variables made for the scope.

## ❑ ng-class Directive

- The ng-class directive dynamically binds one or more CSS classes to an HTML element.
- The value of the ng-class directive can be a string, an object, or an array.
- If it is a string, it should contain one or more, space-separated class names.
- As an object, it should contain key-value pairs, where the key is the class name of the class you want to add, and the value is a boolean value. The class will only be added if the value is set to true.

Syntax

```
<element ng-class="expression"></element>
```

➢ **Example**

```
<!DOCTYPE html>
<html>
<script
    src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></s
    cript>
<style>
.sky {
   color:white;
   background-color:lightblue;
   padding:20px;
   font-family:"Courier New";
}
```

```
.tomato { background-
    color:coral; padding:40px;
    font-family:Verdana;
}
</style>
<body ng-app="">
<p>Choose a class:</p>
```

```
<select ng-model="home">
<option value="sky">Sky</option>
<option value="tomato">Tomato</option>
</select>
<div ng-class="home">
  <h1>Welcome Home!</h1>
  <p>I like it!</p>
</div>
</body>
</html>
```

**Output**

## ➢ Angular Animations

## ❑ What Does ngAnimate Do?

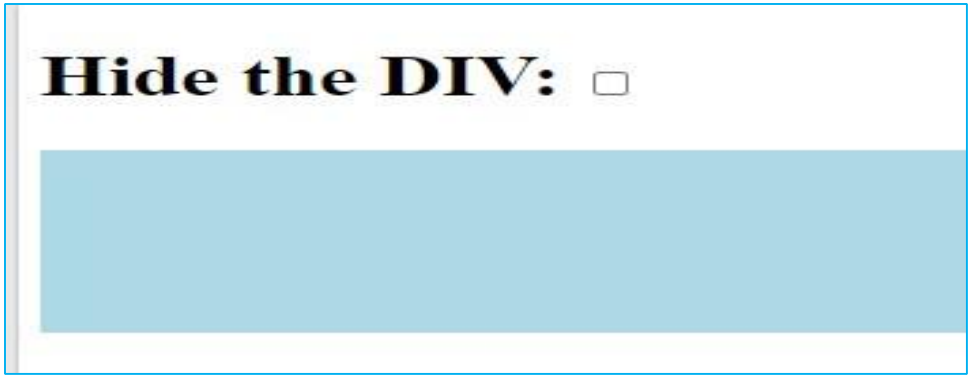
- The ngAnimate module adds and removes classes.
- The ngAnimate module does not animate your HTML elements.
- However, when ngAnimate notices certain events, such as hiding or showing an HTML element, the element receives some pre-defined classes that can be used to create animations.

- **Example**

```
<!DOCTYPE html>
<html>
<style>
div {
  transition: all linear 0.5s;
  background-color: lightblue;
  height: 100px; width: 100%;
  position: relative; top: 0;
  left: 0;
}
```

```
.ng-hide { height: 0; width: 0;
  background-color: transparent;
  top:-200px; left: 200px;
}
</style>
<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/angularanimate.js">
</script>
```

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

<h1>Hide the DIV: <input type="checkbox" ng-model="myCheck"></h1>

<div ng-hide="myCheck"></div>

</body>

</html>
```




- **Output**

## ❑ ng-show Directive

- The AngularJS ng-show directive is used to show or hide the given HTML element according to the expression given to the ng-show attribute.
- It shows the specified HTML element if the given expression is true, otherwise it hides the HTML element.
- It is supported by all HTML elements.

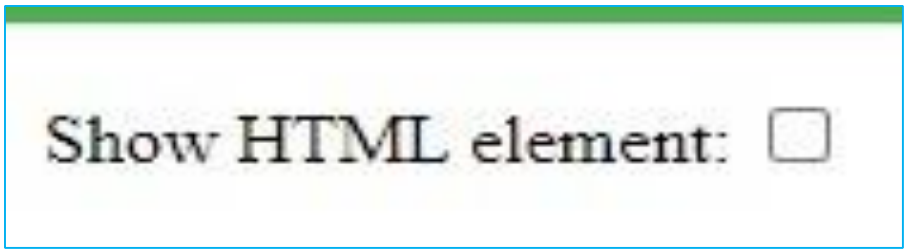
**Syntax:**

```
<element ng-show="expression"></element>
```

➢ **Example**

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.8/angular.min.js">
</script>
<body ng-app="">
Show HTML element: <input type="checkbox" ng-model="myVar">
<div ng-show="myVar">
<h1>Welcome to Angulajs</h1>
<p>A solution of all technology.</p>
</div>
</body>
</html>
```

➢ **Output**


Show HTML element:


Show HTML element:
Welcome to Angulajs
A solution of all technology.

## ❑ ng-hide Directive

- The AngularJS ng-hide directive is used to hide the HTML element if the expression is set to true.
- The element is shown if you remove the ng-hide CSS class and hidden, if you add the ng-hide CSS class onto the element. The ng-hide CSS class is predefined in AngularJS and sets the element's display to none.

```
<element ng-hide="expression"></element>
```

As a CSS class:

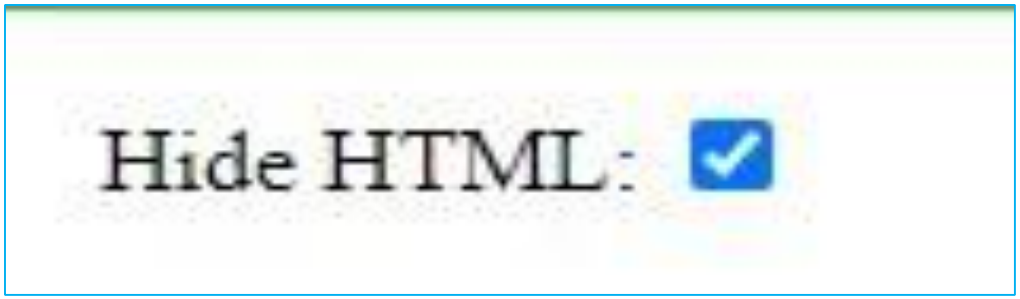
```
<element class="ng-hide"></element>
```

➢ **Example**

```
<!DOCTYPE html>
<html>
<script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.4.8/angular.min.js"></script>
<body ng-app="">
Hide HTML: <input type="checkbox" ng-model="myVar">
<div ng-hide="myVar">
<h1>Welcome to JavaTpoint!</h1>
<p>A  solution of all technologoes.</p>
</div>
</body>
</html>
```

## Output


Hide HTML:
Welcome to JavaTpoint!
A solution of all technologoes.


Hide HTML:

## ➢ Expressions and Controllers

### ❑ AngularJS Expressions

- AngularJS expressions can be written inside double braces: {{ expression }}.
- AngularJS expressions can also be written inside a directive: ngbind="expression".
- AngularJS will resolve the expression, and return the result exactly where the expression is written.
- AngularJS expressions are much like JavaScript expressions: They can contain literals, operators, and variables.
- Example {{ 5 + 5 }} or {{ firstName + " " + lastName }}

- **Example**

```
<!DOCTYPE html>
<html>
<script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js">
</script>
<body>

<div ng-app>
<p>My first expression: {{ 5 + 5 }}</p>
</div>

</body>
</html>
```

- **Output**

```
My first expression: 10
```

- **If you remove the ng-app directive, HTML will display the expression as it is, without solving it:**

```
<!DOCTYPE html>
<html>
<script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js">
</script>
<body>

<p>Without the ng-app directive, HTML will display the expression as it is,
without solving it.</p>

<div>
<p>My first expression: {{ 5 + 5 }}</p>
</div>

</body>
</html>
```

## ➢ Output

Without the ng-app directive, HTML will display the expression as it is, without solving it.

My first expression: {{ 5 + 5 }}

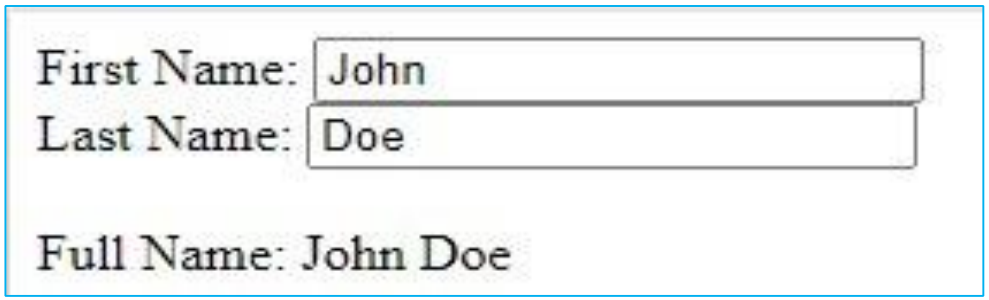
## ➢ AngularJS Controllers

- AngularJS applications are controlled by controllers.
- The **ng-controller** directive defines the application controller.
- A controller is a **JavaScript Object**, created by a standard JavaScript **object constructor**.

## ➢ Example

```
<!DOCTYPE html>
<html>
<script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js">
</script>
<body>

<div ng-app="myApp" ng-controller="myCtrl">

First Name: <input type="text" ng-model="firstName"><br>
Last Name: <input type="text" ng-model="lastName"><br>
<br>
Full Name: {{firstName + " " + lastName}}

</div>

<script>
var app = angular.module('myApp', []);
app.controller('myCtrl', function($scope) {
    $scope.firstName = "John";
    $scope.lastName = "Doe";
});
</script>

</body>
</html>
```

- **Output**



## ➢ AngularJS Filters

- AngularJS Filters allow us to format the data to display on UI without changing original format.
- Filters can be used with an expression or directives using pipe | sign.

{{expression | filterName:parameter }}

➢ Angular includes various filters to format data of different data types. The following table lists important filters.

| Filter Name | Description |
|---|---|
| Uppercase | Converts string to upper case. |
| Lowercase | Converts string to lower case. |
| Currency | Formats numeric data into specified currency format and fraction. |

| | |
|---|---|
| orderBy | Sorts an array based on specified predicate expression. |

## ➢ Uppercase/lowercase filter

- The uppercase filter converts the string to upper case and lowercase filter converts the string to lower case.

Example: uppercase & lowercase filters

```
<!DOCTYPE html>
<html >
<head>
    <script src="~/Scripts/angular.js"></script>
</head>
<body ng-app>
    <div ng-init="person.firstName='James';person.lastName='Bond'">
        Lower case: {{person.firstName + ' ' + person.lastName | lowercase}} <br />
        Upper case: {{person.firstName + ' ' + person.lastName | uppercase}}
    </div>
</body>
</html>
```

Result:

```
Lower case: james bond
Upper case: JAMES BOND
```

## Currency Filter

- The currency filter formats a number value as a currency. When no currency symbol is provided, default symbol for current locale is used.

- AngularJS currency filter is used to convert a number into a currency format.

- If no currency format is specified currency filter uses the local currency format.

  Parameters: It contains 2 parameters as mentioned above and described below:

  symbol: It is an optional parameter.

  ```
  {{ expression | currency : 'currency_symbol' : 'fraction'}}
  ```

- **Parameters:** It contains 2 parameters as mentioned above and described below:

- **symbol:** It is an optional parameter. It is used to specify the currency symbol. The currency symbol can be any character or text.

- **fractionsize:** It is an optional parameter. It is used to specify the number of decimals.

Example: Currency filter

```
<!DOCTYPE html>
<html >
<head>
    <script src="~/Scripts/angular.js"></script>
</head>
<body ng-app="myApp">
    <div ng-controller="myController">
        Default currency: {{person.salary | currency}} <br />
        Custom currency identifier: {{person.salary | currency:'Rs.'}} <br />
        No Fraction: {{person.salary | currency:'Rs.':0}} <br />
        Fraction 2: <span ng-bind="person.salary| currency:'GBP':2"></span>
    </div>
    <script>
        var myApp = angular.module('myApp', []);

        myApp.controller("myController", function ($scope) {
            $scope.person = { firstName: 'James', lastName: 'Bond', salary: 100000}
        });
    </script>
</body>
</html>
```

## ➢ Output

Result:

```
Default currency: $100,000.00
Custom currency identifier: Rs.100,000.00
No Fraction: Rs.100,000
Fraction 2: GBP100,000.00
```

## ➢ orderBy filter

- The orderBy filter sorts an array based on specified expression predicate.

{{ expression | orderBy : predicate_expression : reverse}}

- Example: orderBy filter

```
<!DOCTYPE html>
<html>
<head>
    <script src="~/Scripts/angular.js"></script>
</head>
<body ng-app="myApp">
    <div ng-controller="myController">
        <select ng-model="SortOrder">
            <option value="+name">Name (asc)</option>
            <option value="-name">Name (dec)</option>
            <option value="+phone">Phone (asc)</option>
            <option value="-phone">Phone (dec)</option>
        </select>
        <ul ng-repeat="person in persons | orderBy:SortOrder">
            <li>{{person.name}} - {{person.phone}}</li>
        </ul>
    </div>
```

```
    <script>
        var myApp = angular.module('myApp', []);

        myApp.controller("myController", function ($scope) {

            $scope.persons = [{ name: 'John', phone: '512-455-1276' },
                              { name: 'Mary', phone: '899-333-3345' },
                              { name: 'Mike', phone: '511-444-4321' },
                              { name: 'Bill', phone: '145-788-5678' },
                              { name: 'Ram', phone: '433-444-8765' },
                              { name: 'Steve', phone: '218-345-5678' }]

            $scope.SortOrder = '+name';
        });
    </script>
</body>
</html>
```

- The above example displays a list of person names and phone numbers in a particular order specified using orderBy:SortOrder filter.
- SortOrder is a model property and will be set to the selected value in the dropdown.

- Therefore, based on the value of SortOrder, ng-repeat directive will display the data.

# Unit-3: Angular JS: Single page application:

**Unit-3: Angular JS: Single page application:**

Single page application using AngularJS

Create a module, Define Simple controller

Embedding AngularJS script in HTML

AngularJS's routine capability

$routeProvider service from ngRoute

Navigating different pages

HTML DOM directives

ng-disabled, ng-show, ng-hide, ng-click

Modules (Application, Controller)

Forms (Events, Data validation, ng-click)

## ➢ Single page application using AngularJS

- Single page application (SPA) is a web application that fits on a single page.
- All your code (JS, HTML, CSS) is retrieved with a single page load. And navigation between pages performed without refreshing the whole page.

## ➢ Advantages of Single Page Application:

### ❑ Team collaboration:

- Single-page applications are excellent when more than one developer is working on the same project.
- It allows backend developers to focus on the API, while frontend developers can focus on creating the user interface based on the backend API.

### ❑ Caching:

- The application sends a single request to the server and stores all the received information in the cache. This proves beneficial when the client has poor network connectivity.

### ❑ Fast and responsive:

- As only parts of the pages are loaded dynamically, it improves the website's speed.

- **Debugging is easier:**

- Debugging single-page applications with chrome is easier since such applications are developed using AngularJS Batarang and React developer tools.

- **Linear user experience:** Browsing or navigating through the website is easy.

## ➢ Disadvantages of Single Page Application:

### ❑ SEO optimization:

- SPAs provide poor SEO optimization.
- This is because single-page applications operate on JavaScript and load data at once server.
- The URL does not change and different pages do not have a unique URL.
- Hence it is hard for the search engines to index the SPA website as opposed to traditional server-rendered pages.

### ❑ Browser history:

- A SPA does not save the users' transition of states within the website. A browser saves the previous pages only, not the state transition.
- Thus when users click the back button, they are not redirected to the previous state of the website.

- To solve this problem, developers can equip their SPA frameworks with the HTML5 History API.

❑ **Security issues:**

- Single-page apps are less immune to cross-site scripting (XSS) and since no new pages are loaded, hackers can easily gain access to the website and inject new scripts on the client-side.

❑ **Memory Consumption:**

- Since the SPA can run for a long time sometimes hours at a time, one needs to make sure the application does not consume more memory than it needs. Else, users with low-memory devices may face serious performance issues.

❑ **Disabled Javascript:**

- Developers need to chalk out ideas for users to access the information on the website for browsers that have Javascript disabled.

## ➢ Pros

### ❑ No Page Refresh

- When you are using SPA, you don't need to refresh the whole page, just load the part of the page which needs to be changed. Angular allows you to preload and cache all your pages, so you don't need extra requests to download them.

### ❑ Better User Experience

- SPA feels like a native application: fast and responsive.

### ❑ Ability to Work Offline

- Even if user loses internet connection, SPA can still work because all the pages are already loaded.

## ➢ Cons

### ❑ More Complex to Build

- You need to write pretty much JavaScript, handle shared state between pages, manage permissions, etc.

### ❑ SEO

- To index your SPA app, search engine crawlers should be able to execute JavaScript. Only recently, Google and Bing started indexing Ajax-based pages by executing JavaScript during crawling. You need to create static HTML snapshots especially for search engines.

### ❑ Initial Load is Slow

- SPA needs to download more resources when you open it.

### ❑ Client Should have JavaScript Enabled

- Of course, SPA requires JavaScript. But fortunately, almost everyone has JavaScript enabled.

## ➢ Angular Application

- Every angular application starts from creating a module. Module is a container for the different parts of your application: controllers, service, etc.

```javascript
var app = angular.module('myApp', []);
```

- **Let's define a simple controller:**

```javascript
app.controller('HomeController', function($scope) {
  $scope.message = 'Hello from HomeController';
});
```

- After we created module and controller, we need to use them in our HTML.
- First of all, we need to include Angular script and ***app.js*** that we built.
- Then, we need to specify our module in **ng-app** attribute and controller in **ng-controller** attribute.

HTML

```
<!doctype html>
<html ng-app="myApp">
  <head>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.4.7/angular.min.js"></script>
  </head>
  <body ng-controller="HomeController">
    <h1>{{message}}</h1>
    <script src="app.js"></script>
  </body>
</html>
```

- **Output**


If you have done this correctly, you should see:
localhost:8000
Hello from HomeController

## ➢ Creating a Module

- A module is created by using the AngularJS function angular.module

```
<div ng-app="myApp">...</div>

<script>

var app = angular.module("myApp", []);

</script>
```

- The "myApp" parameter refers to an HTML element in which the application will run.
- Now you can add controllers, directives, filters, and more, to your AngularJS application.

## ➢ Adding a Controller

- Add a controller to your application, and refer to the controller with the ng-controller directive:

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/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 = "HP";
    $scope.lastName = "Company";
});
</script>

</body>
</html>
```

- **Output**

## ➢ Modules and Controllers in Files

- It is common in AngularJS applications to 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:

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body>

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

<script src="myApp.js"></script>
<script src="myCtrl.js"></script>

</body>
</html>
```

- **Output**

## ➢ Embedding AngularJS script in HTML

### ❑ Angularjs include html:

- HTML does not support embedding html pages within html page.
- But we can achieve this functionality using AngularJS.
- AngularJS provides the ng-include directive to embed HTML pages within a HTML page.

- **Example:**
- **testable.htm**

```
<p>Students with country name:</p>
        <table>
            <tr>
                <th>Name</th>
                <th>Country</th>
            </tr>

            <tr ng-repeat = "student in students">
               <td>{{ student.name }}</td>
               <td>{{ student.country }}</td>
            </tr>
        </table>
```

- **Include above html file**

```
<html>
   <head>
    <title>AngularJS Include Example</title>
    <script src=
     "http://ajax.googleapis.com/ajax/libs/angularjs/1.4.8/angular.min.js">
    </script>
    <style>
     table, th , td {
      border: 1px solid grey;
      border-collapse: collapse;
      padding: 5px;
     }

     table tr:nth-child(odd) {
      background-color: #f2f2f2;
     }

     table tr:nth-child(even) {
      background-color: #ffffff;
     }
    </style>
   </head>
```

```
    <body>
        <h1>AngularJS Include Example.</h1>

        <div ng-app = ""
             ng-init = "students=[{name:'Prabhjot',country:'US'},
                                  {name:'Nidhi',country:'Sweden'},
                                  {name:'Kapil',country:'India'}]"
                                  >
          <div ng-include="'testTable.htm'"></div>
        </div>
    </body>
</html>
```

Output:

# AngularJS Include Example.

Students with country name:

| Name | Country |
|---|---|
| Prabhjot | US |
| Nidhi | Sweden |
| Kapil | India |

## ➢ AngularJS's routine capability

- **Routing** in AngularJS is used when the user wants to navigate to different pages in an application but still wants it to be a single-page application.
- AngularJS routes enable the user to create different URLs for different content in an application.
- The **ngRoute** module helps in accessing different pages of an application without reloading the entire application.
- We can build Single Page Application (SPA) with AngularJS.
- It is a web app that loads a single HTML page and dynamically updates that page as the user interacts with the web app.
- AngularJS supports SPA using routing module ngRoute. This routing module acts based on the url.
- When a user requests a specific url, the routing engine captures that url and renders the view based on the defined routing rules.

- AngularJS also provides the ability to pass parameters in routes, which means, it allows us to dynamically generate routes and handle different data based on the parameters.
- We can define route patterns with placeholders for parameters, and AngularJS will extract the values from the URL and make them available in your controller.

- This parameterization of routes can be useful for creating dynamic pages or handling specific data queries within a single-page application.

## ➢ $routeProvider

- $routeProvider is used to configure the routes. It helps to define what page to display when a user clicks a link.
- It accepts either when() or otherwise() method.
- The ngRoute must be added as a dependency in the application module.
- With the $routeProvider you can define what page to display when a user clicks a link.
- Define the $routeProvider using the config method of your application. Work registered in the config method will be performed when the application is loading.
- Routing allows us to create Single Page Applications. To do this, we use *ngview* and *ng-template directives*, and *$routeProvider services*.
- We use *$routeProvider* to configure the routes.

- The *config()* takes a function that takes the *$routeProvider* as a parameter and the routing configuration goes inside the function.
- The *$routeProvider* is a simple API that accepts either when() or otherwise() method. We need to install the *ngRoute module*.
- f 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.

- Now your application has access to the route module, which provides the $routeProvider.
- Use the $routeProvider to configure different routes in your application:

```
app.config(function($routeProvider) {
  $routeProvider
  .when("/", {
    templateUrl : "main.htm"
  })
  .when("/red", {
    templateUrl : "red.htm"
  })
  .when("/green", {
    templateUrl : "green.htm"
  })
  .when("/blue", {
    templateUrl : "blue.htm"
  });
});
```

## ➢ Navigating different pages

- We will be building an application, which will display a login page when a user requests for base url - *http://localhost/*. Once the user logs in successfully, we will redirect it to student page *http://localhost/student/{username}* where username would be logged in user's name.
- In our example, we will have one layout page - index.html, and two HTML templates - login.html and student.html.
- Index.html - layout view
- login.html - template
- student.html - template

- **The following is a main layout view - index.html.**

Example: Layout view - Index.html

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title></title>
    <script src="Scripts/angular.js"></script>
    <script src="Scripts/angular-route.js"></script>
    <link href="Content/bootstrap.css" rel="stylesheet" />
</head>
<body ng-app="ngRoutingDemo">
    <h1>Angular Routing Demo</h1>

    <div ng-view>

    </div>
```

```
<script>
    var app = angular.module('ngRoutingDemo', ['ngRoute']);

    app.config(function ($routeProvider) {

        $routeProvider.when('/', {
            templateUrl: '/login.html',
            controller: 'loginController'
        }).when('/student/:username', {
            templateUrl: '/student.html',
            controller: 'studentController'
        }).otherwise({
            redirectTo: "/"
        });

    app.controller("loginController", function ($scope, $location) {

        $scope.authenticate = function (username) {
            // write authentication code here..

            $location.path('/student/' + username)
        };

    });
```

```
        app.controller("loginController", function ($scope, $location) {

            $scope.authenticate = function (username) {
                // write authentication code here..

                $location.path('/student/' + username)
            };

        });

        app.controller("studentController", function ($scope, $routeParams) {
            $scope.username = $routeParams.username;
        });

    });
    </script>
</body>
</html>
```

## ➢ HTML DOM directives

- AngularJS has directives for binding application data to the attributes of HTML DOM elements.

❑ **The following directives are used to bind application data to the attributes of HTML DOM elements –**

| Sr.No. | Name & Description |
|---|---|
| 1 | **ng-disabled**<br>disables a given control. |
| 2 | **ng-show**<br>shows a given control. |
| 3 | **ng-hide**<br>hides a given control. |
| 4 | **ng-click**<br>represents a AngularJS click event. |

## ❑ ng-disabled Directive

- The **ng-disabled** directive binds AngularJS application data to the disabled attribute of HTML elements.

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body>

<div ng-app="" ng-init="mySwitch=true">
<p>
<button ng-disabled="mySwitch">Click Me!</button>
</p>
<p>
<input type="checkbox" ng-model="mySwitch"/>Button
</p>
<p>
{{ mySwitch }}
</p>
</div>

</body>
</html>
```

- **Output**

➢ **Application explained:**

- The **ng-disabled** directive binds the application data **mySwitch** to the HTML button's **disabled** attribute.
- The **ng-model** directive binds the value of the HTML checkbox element to the value of **mySwitch**.
- If the value of **mySwitch** evaluates to **true**, the button will be disabled:

```
<p>
<button disabled>Click Me!</button>
</p>
```

If the value of **mySwitch** evaluates to **false**, the button will not be disabled:

```
<p>
<button>Click Me!</button>
</p>
```

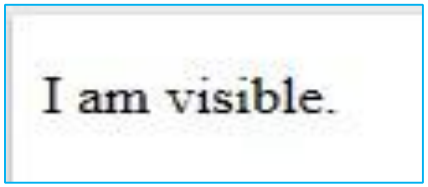
## ❑ ng-show Directive

- The **ng-show** directive shows or hides an HTML element.

### ➢ Example

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body>

<div ng-app="">

<p ng-show="true">I am visible.</p>

<p ng-show="false">I am not visible.</p>

</div>

</body>
</html>
```

### ➢ Output



## ❑ ng-hide Directive

- The **ng-hide** directive hides or shows an HTML element.

### ➢ Example

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body>

<div ng-app="">

<p ng-hide="true">I am not visible.</p>

<p ng-hide="false">I am visible.</p>

</div>

</body>
</html>
```

### ➢ Output

## ➢ ng-click Directive

- The ng-click directive defines AngularJS code that will be executed when the element is being clicked.

## ➢ Example

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body>

<div ng-app="myApp" ng-controller="myCtrl">

<button ng-click="count = count + 1">Click Me!</button>

<p>{{ count }}</p>

</div>
<script>
var app = angular.module('myApp', []);
app.controller('myCtrl', function($scope) {
    $scope.count = 0;
});
</script>

</body>
</html>
```

- **Output**


## ➢ Modules (Application, Controller)

## ❑ AngularJS Modules

- A module in AngularJS is a container of the different parts of an application such as controller, service, filters, directives, factories etc. It supports separation of concern using modules.
- AngularJS stops polluting global scope by containing AngularJS specific functions in a module.

## ❑ Application Module

- An AngularJS application must create a top level application module. This application module can contain other modules, controllers, filters, etc.

Example: Create Application Module

```
<!DOCTYPE html>
<html >
<head>
    <script src="~/Scripts/angular.js"></script>
</head>
<body ng-app="myApp">
    @* HTML content *@
    <script>
        var myApp = angular.module('myApp', []);

    </script>
</body>
</html>
```

- In the above example, the angular.module() method creates an application module, where the first parameter is a module name which is same as specified by ng-app directive.
- The second parameter is an array of other dependent modules [].
- In the above example we are passing an empty array because there is no dependency.

➢ **Note:**

- The angular.module() method returns specified module object if no dependency is specified.
- Therefore, specify an empty array even if the current module is not dependent on other module.
- Now, you can add other modules in the myApp module.

## ❑ Create Controller Module

➢ The following example demonstrates creating controller module in myApp module.

Example:Create Controller Module

```
<!DOCTYPE html>
<html >
<head>
    <script src="~/Scripts/angular.js"></script>
</head>
<body ng-app="myApp">
    <div ng-controller="myController">
        {{message}}
    </div>
    <script>
        var myApp = angular.module("myApp", []);

        myApp.controller("myController", function ($scope) {
            $scope.message = "Hello Angular World!";
        });
    </script>
</body>
</html>
```

- In the above example, we have created a controller named "myController" using myApp.controller() method.
- Here, myApp is an object of a module, and controller() method creates a controller inside "myApp" module.

## ➢ Forms (Events, Data validation, ng-click)

### ❑ 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

## ➢ Events

- AngularJS provides multiple events associated with the HTML controls. For example, ng-click directive is generally associated with a button. AngularJS supports the following events −
- 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

## ➢ ng-click

- The ng-click directive tells AngularJS what to do when an HTML element is clicked.

### Syntax

```
<element ng-click="expression"></element>
```

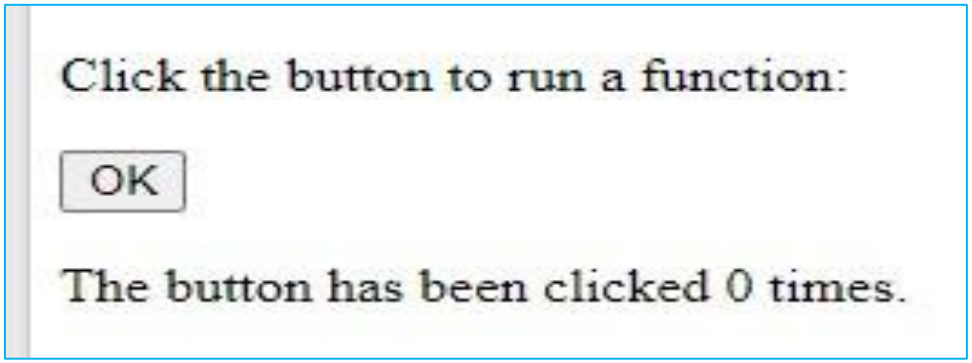
### Parameter Values

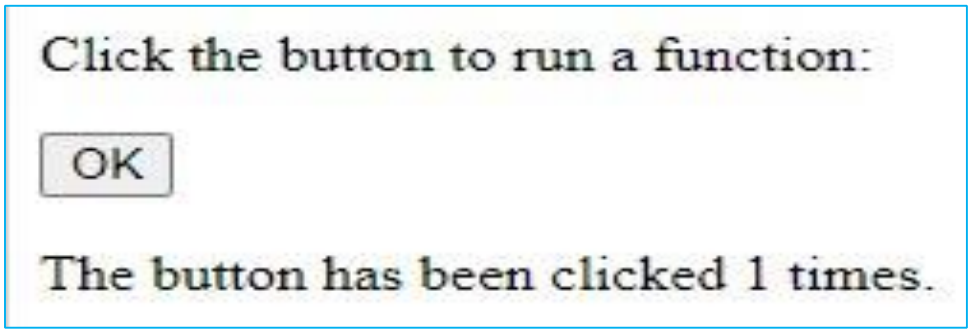
| Value | Description |
| --- | --- |
| *expression* | An expression to execute when an element is clicked. |

## ➢ Example

```
<!DOCTYPE html>
<html>
<script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js">
</script>
<body ng-app="myApp">

<div ng-controller="myCtrl">
  <p>Click the button to run a function:</p>
  <button ng-click="myFunc()">OK</button>
  <p>The button has been clicked {{count}} times.</p>
</div>

<script>
angular.module('myApp', [])
  .controller('myCtrl', ['$scope', function($scope) {
    $scope.count = 0;
    $scope.myFunc = function() {
      $scope.count++;
    };
  }]);
</script>
</body>
</html>
```

- **Output**


Click the button to run a function:
OK
The button has been clicked 0 times.


Click the button to run a function:
OK
The button has been clicked 1 times.

## ➢ Form Validation

- AngularJS offers client-side form validation.
- AngularJS monitors the state of the form and input fields (input, textarea, select), and lets you notify the user about the current state.
- AngularJS also holds information about whether they have been touched, or modified, or not.
- You can use standard HTML5 attributes to validate input, or you can make your own validation functions.
- Client-side validation cannot alone secure user input. Server side validation is also necessary.

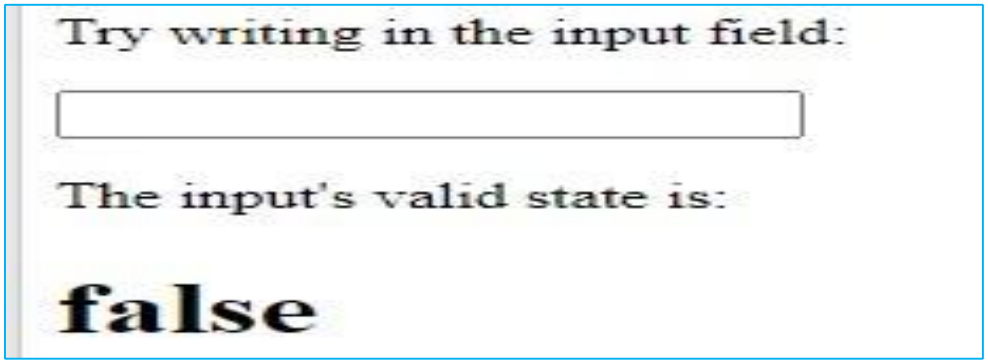
## ❑ Required

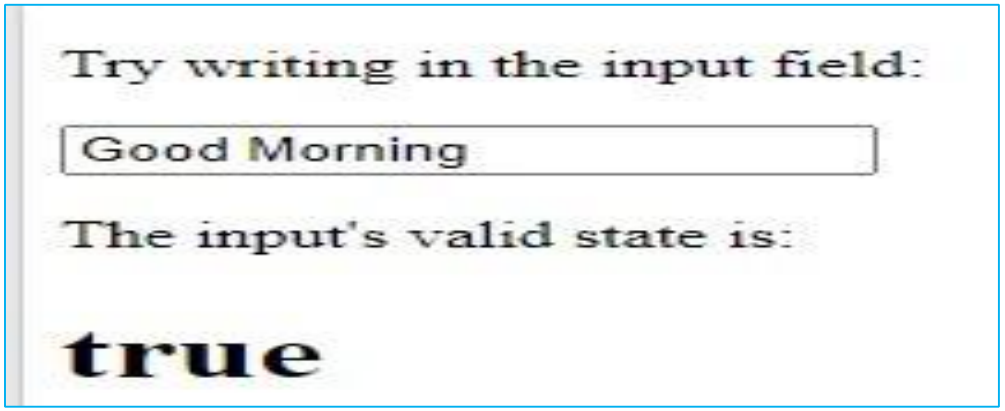
- Use the HTML5 attribute required to specify that the input field must be filled out:

## ➢ Example

```
<!DOCTYPE html>
<html>
<script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js">
</script>
<body ng-app="">

<p>Try writing in the input field:</p>

<form name="myForm">
<input name="myInput" ng-model="myInput" required>
</form>

<p>The input's valid state is:</p>
<h1>{{myForm.myInput.$valid}}</h1>

</body>
</html>
```

- **Output**


Try writing in the input field:
The input's valid state is:
false


Try writing in the input field:
Good Morning
The input's valid state is:
true

## ➢ Example

```
<html>
<script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js">
</script>
<body ng-app="">

<p>Try leaving the first input field blank:</p>

<form name="myForm">
<p>Name:
<input name="myName" ng-model="myName" required>
<span ng-show="myForm.myName.$touched && myForm.myName.$invalid">The name is
required.</span>
</p>

<p>Address:
<input name="myAddress" ng-model="myAddress" required>
</p>

</form>

<p>We use the ng-show directive to only show the error message if the field
has been touched AND is empty.</p>

</body>
</html>
```

## ➢ Output

Try leaving the first input field blank:

Name:

Address:

We use the ng-show directive to only show the error message if the field has been touched AND is empty.

Try leaving the first input field blank:

Name: The name is required.

Address:

We use the ng-show directive to only show the error message if the field has been touched AND is empty.