NoSQL Database

NoSQL Database is referred as a non-SQL or non relational or Not Only SQL database. It provides a mechanism for storage and retrieval of data other than the tabular relations model used in relational databases. NoSQL databases don't use tables for storing data. It is generally used to store big data and real-time web applications.

History behind the creation of NoSQL Databases

- In the early 1970, Flat File Systems were used. Data was stored in flat files and the biggest problems with flat files are each company implements their own flat files and there are no standards. It is very difficult to store data in the files, retrieve data from files because there is no standard way to store data.

- Then the relational database was created by E.F. Codd and these databases answered the question of having no standard way to store data. But later relational database also get a problem that it could not handle big data, due to this problem there was a need of database which can handle every types of problems then NoSQL database was developed.

Features of NoSQL

Non-relational

- NoSQL databases never follow the relational model
- Never provide tables with flat fixed-column records
- Work with self-contained aggregates or BLOBs
- Doesn’t require object-relational mapping and data normalization
- No complex features like query languages, query planners,referential integrity joins, ACID

Schema-free

- NoSQL databases are either schema-free or have relaxed schemas
- Do not require any sort of definition of the schema of the data
- Offers heterogeneous structures of data in the same domain

Simple API

- Offers easy to use interfaces for storage and querying data provided
- APIs allow low-level data manipulation & selection methods
- Text-based protocols mostly used with HTTP REST with JSON
- Mostly used no standard based NoSQL query language

- Web-enabled databases running as internet-facing services

Distributed

- Multiple NoSQL databases can be executed in a distributed fashion
- Offers auto-scaling and fail-over capabilities
- Often ACID concept can be sacrificed for scalability and throughput
- Mostly no synchronous replication between distributed nodes Asynchronous Multi-Master Replication, peer-to-peer, HDFS Replication
- Only providing eventual consistency
- Shared Nothing Architecture. This enables less coordination and higher distribution.

Types of NoSQL Databases

NoSQL Databases are mainly categorised into four types: Key-value pair, Column-oriented, Graph-based and Document-oriented. Every category has its unique attributes and limitations. None of the above-specified database is better to solve all the problems. Users should select the database based on their product needs.

Types of NoSQL Databases:

1. Key-value Pair Based
2. Column-oriented Graph
3. Graphs based
4. Document-oriented

Advantages of NoSQL

- Can be used as Primary or Analytic Data Source
- Big Data Capability
- No Single Point of Failure
- Easy Replication
- No Need for Separate Caching Layer
- It provides fast performance and horizontal scalability.
- Can handle structured, semi-structured, and unstructured data with equal effect
- Object-oriented programming which is easy to use and flexible

- NoSQL databases don’t need a dedicated high-performance server
- Support Key Developer Languages and Platforms
- Simple to implement than using RDBMS
- It can serve as the primary data source for online applications.
- Handles big data which manages data velocity, variety, volume, and complexity
- Excels at distributed database and multi-data center operations
- Eliminates the need for a specific caching layer to store data
- Offers a flexible schema design which can easily be altered without downtime or service disruption

Disadvantages of NoSQL

- No standardization rules
- Limited query capabilities
- RDBMS databases and tools are comparatively mature
- It does not offer any traditional database capabilities, like consistency when multiple transactions are performed simultaneously.
- When the volume of data increases it is difficult to maintain unique values as keys become difficult
- Doesn’t work as well with relational data
- The learning curve is stiff for new developers
- Open source options so not so popular for enterprises.

Introduction to MongoDB

MongoDB is a non-relational document database that provides support for JSON-like storage. The MongoDB database has a flexible data model that enables you to store unstructured data, and it provides full indexing support,

- MongoDB stores data in flexible, JSON-like documents, meaning fields can vary from document to document and data structure can be changed over time

- The document model maps to the objects in your application code, making data easy to work with
- Ad hoc queries, indexing, and real time aggregation provide powerful ways to access and analyze your data
- MongoDB is a distributed database at its core, so high availability, horizontal scaling, and geographic distribution are built in and easy to use
- MongoDB is free to use.

MongoDB Datatypes

Following is a list of usable data types in MongoDB.

Data Types	Description
String	String is the most commonly used datatype. It is used to store data. A string must be UTF 8 valid in mongodb.
Integer	Integer is used to store the numeric value. It can be 32 bit or 64 bit depending on the server you are using.
Boolean	This datatype is used to store boolean values. It just shows YES/NO values.
Double	Double datatype stores floating point values.
Min/Max Keys	This datatype compare a value against the lowest and highest bson elements.
Arrays	This datatype is used to store a list or multiple values into a single key.
Object	Object datatype is used for embedded documents.
Null	It is used to store null values.
Symbol	It is generally used for languages that use a specific type.
Date	This datatype stores the current date or time in unix time format. It makes you possible to specify your own date time by creating object of date and pass the value of date, month, year into it.

MongoDB Create Database

How and when to create a database

- If there is no existing database, the following command is used to create a new database.

Syntax:

```
use DATABASE_NAME
```

- If the database already exists, it will return the existing database.

  Let' 's take an example to demonstrate how a database is created in MongoDB. In the following example, we are going to create a database "maintenanceDB".

Few commands for database operation

use DATABASE_NAME	To connect or create database
db	Return current connected/selected database instance
show dbs	Returns the list of databases
db.dropDatabase()	To delete / drop current/selected database

Example of Creating / Selecting a Database :

```
use maintenanceDB
Swithched to db maintenanceDB

local 0.078GB
```

- Here, your created database "maintenanceDB" will not be present in the list, insert at least one document into it to display database:

  ```
  db.buildings.insert({"name":"first Building"})
  ```

Example to list all databases on server:

```
Show dbs

-config
-admin
-maintenanceDB
```

Example to Drop a database:

```
db.dropDatabase();
```

Here, db represents connected database instance

MongoDB Create Collection

- In MongoDB, db.createCollection(name, options) is used to create collection. But usually you don’t need to create collection.
- MongoDB creates collection automatically when you insert some documents.

Syntax:

```
db.createCollection(name, options)
```

Here,

- **Name**: is a string type, specifies the name of the collection to be created.
- **Options**: is a document type, that specifies the memory size and indexing of the collection. It is an optional parameter.

Following is the list of options that can be used.

Field	Type	Description
Capped	Boolean	(Optional) If it is set to true, enables a capped collection. Capped collection is a fixed size collecction that automatically overwrites its oldest entries when it reaches its maximum size. If you specify true, you need to specify size parameter also.
AutoIndexID	Boolean	(Optional) If it is set to true, automatically create index on ID field. Its default value is false.
Size	Number	(Optional) It specifies a maximum size in bytes for a capped collection. Ifcapped is true, then you need to specify this field also.
Max	Number	(Optional) It specifies the maximum number of documents allowed in the capped collection.

- Let's take an example to create collection. In this example, we are going to create a collection name requests.

```
use maintenanceDB

db.createCollection("requests")


{ "ok" : 1 }
```

- To check the created collection, use the command "show collections".

```
show collections
```

How does MongoDB create collections automatically

- MongoDB creates collections automatically when you insert some documents.

For example:

- Insert a document named into a collection named request. The operation will create the collection if the collection does not currently exist.

```
db.users.insert({"type" : "admin"})

show collections
```

- If you want to see the inserted document, use the find() command.

Syntax:

```
db.collection_name.find()
```

Drop a Collection in MongoDB:

- If you want to drop a collection in MongoDB use drop() command in following way\

```
db.collection_name.drop()
```

What is CRUD in MongoDB?

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

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

When it comes to the individual CRUD operations:

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

Create Operation(Insert)

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

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

insertOne()

- As the namesake, insertOne() allows you to insert one document into the collection.

```
db.users.insertOne({name:"adminOne",password:"default"});
```

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

```
{
        "acknowledged" : true,
        "insertedId" : ObjectId("5fd989674e6b9ceb8665c57d")
}
```

insertMany()

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

```
db.users.insertMany(
    {name:"userOne",password:"default"},
    {name:"usertwo",password:"default"}
);
```

```
{
        "acknowledged" : true,
        "insertedIds" : [
                ObjectId("5fd98ea9ce6e8850d88270b4"),
                ObjectId("5fd98ea9ce6e8850d88270b5")
        ]
}
```

- on successful operation, two new documents are created. The function will return an object where “acknowledged” is “true” and “insertIds” are the newly created “ObjectIds”

Read Operation(Select)

- The read operations allow you to supply special query filters and criteria that let you specify which documents you want.
- Query modifiers may also be used to change how many results are returned.

MongoDB has two methods of reading documents from a collection:

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

find()

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

```
db.RecordsDB.find()
```

```
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "6 years",
"species" : "Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "3 years",
"species" : "Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
{ "_id" : ObjectId("5fd994efce6e8850d88270ba"), "name" : "Kevin", "age" : "8 years",
"species" : "Dog", "ownerAddress" : "900 W. Wood Way", "chipped" : true }
```

- Here we can see that every record has an assigned "ObjectId" mapped to the "_id" key.
- If you want to get more specific with a read operation and find a desired subsection of the records, you can use the filtering criteria to choose what results should be returned. One of the most common ways of filtering the results is to search by value.

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

```
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
```

findOne()

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

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

Let's take the following collection—say, *RecordsDB*, as an example.

```
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "8 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "6 years",
"species" : "Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "3 years",
"species" : "Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
{ "_id" : ObjectId("5fd994efce6e8850d88270ba"), "name" : "Kevin", "age" : "8 years",
"species" : "Dog", "ownerAddress" : "900 W. Wood Way", "chipped" : true }
```

And, we run the following line of code:

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

We would get the following result:

```
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "8 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
```

Notice that even though two documents meet the search criteria, only the first document that matches the search condition is returned.

Update Operations

- Like create operations, update operations operate on a single collection, and they are atomic at a single document level. An update operation takes filters and criteria to select the documents you want to update.
- You should be careful when updating documents, as updates are permanent and can't be rolled back. This applies to delete operations as well.

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

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

updateOne()

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

- To update a document, we provide the method with two arguments: an update filter and an update action.
- The update filter defines which items we want to update, and the update action defines how to update those items.
- We first pass in the update filter. Then, we use the “$set” key and provide the fields we want to update as a value. This method will update the first record that matches the provided filter.

```
db.RecordsDB.updateOne(
        {name: "Marsh"},
        {$set:
            {ownerAddress: "451 W. Coffee St. A204"}
        })
```

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

```
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "6 years",
"species" : "Dog", "ownerAddress" : "451 W. Coffee St. A204", "chipped" : true }
```

updateMany()

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

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

```
{ "acknowledged" : true, "matchedCount" : 3, "modifiedCount" : 3 }
```

```
> db.RecordsDB.find()
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "5", "species"
: "Dog", "ownerAddress" : "451 W. Coffee St. A204", "chipped" : true }
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "5", "species" :
"Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
{ "_id" : ObjectId("5fd994efce6e8850d88270ba"), "name" : "Kevin", "age" : "5", "species"
: "Dog", "ownerAddress" : "900 W. Wood Way", "chipped" : true }
```

replaceOne()

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

```
db.RecordsDB.replaceOne({name: "Kevin"}, {name: "Maki"})
```

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

```
> db.RecordsDB.find()
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "5", "species"
: "Dog", "ownerAddress" : "451 W. Coffee St. A204", "chipped" : true }
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "5", "species" :
"Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
{ "_id" : ObjectId("5fd994efce6e8850d88270ba"), "name" : "Maki" }
```

Delete Operations

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

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

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

deleteOne()

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

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

```
{ "acknowledged" : true, "deletedCount" : 1 }
```

```
> db.RecordsDB.find()
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "5", "species"
: "Dog", "ownerAddress" : "451 W. Coffee St. A204", "chipped" : true }
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "5", "species" :
"Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
```

deleteMany()

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

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

```
{ "acknowledged" : true, "deletedCount" : 2 }
```

```
db.RecordsDB.find()
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
```

Query Selector & Projection Operator in MongoDB

Comparison

Name	Description	Syntax
`$eq`	Matches values that are equal to a specified value.	{ : { $eq: } }
`$gt`	Matches values that are greater than a specified value.	{ field: { $gt: value } }
`$gte`	Matches values that are greater than or equal to a specified value.	{ field: { $gte: value } }
`$in`	Matches any of the values specified in an array.	{ field: { $in: [, , ...] } }
`$lt`	Matches values that are less than a specified value.	{ field: { $lt: value } }

Example

```
db.contributor.find({branch: {$eq: "CSE"}})
```

Logical

Name	Description	Syntax
`$and`	Joins query clauses with a logical `AND` returns all documents that match the conditions of both clauses.	{ $and: [{ }, { } , ... , { }] }
`$not`	Inverts the effect of a query expression and returns documents that do *not* match the query expression.	Syntax: { field: { $not: { } } }
`$nor`	Joins query clauses with a logical `NOR` returns all documents that fail to match both clauses.	{ $nor: [{ }, { }, ... { }] }
`$or`	Joins query clauses with a logical `OR` returns all documents that match the conditions of either clause.	{ $or: [{ }, { }, ... , { }] }

Example

```
db.contributor.find(
        {
            $and:
                [{branch: "CSE"}, {joiningYear: 2018}]
        }
    )
```

Element Query Operators

Name	Description	Syntax
`$exists`	Matches documents that have the specified field.	{ field: { $exists: } }
`$type`	Selects documents if a field is of the specified type.	{ field: { $type: } }

Example

```
db.spices.find( { saffron: { $exists: true } } )
```

- The results consist of those documents that contain the field `saffron`, including the document whose field `saffron` contains a null value.
-

What is Projection?

MongoDB provides a special feature that is known as **Projection**. It allows you to select only the necessary data rather than selecting whole data from the document. For example, a document contains 5 fields, i.e.,

```
{
name: "Roma",
age: 30,
branch: EEE,
department: "HR",
salary: 20000
}
```

But we only want to display the *name* and the *age* of the employee rather than displaying whole details. Now, here we use projection to display the name and age of the employee.

One can use projection with `db.collection.find()` method. In this method, the second parameter is the projection parameter, which is used to specify which fields are returned in the matching documents.

Projection Operators

Name	Description
`$`	Projects the first element in an array that matches the query condition.
`$elemMatch`	Projects the first element in an array that matches the specified `$elemMatch` condition.
`$slice`	Limits the number of elements projected from an array. Supports skip and limit slices.

Update Operators

Fields

Name	Description
`$currentDate`	Sets the value of a field to current date, either as a Date or a Timestamp.
`$inc`	Increments the value of the field by the specified amount.
`$min`	Only updates the field if the specified value is less than the existing field value.
`$max`	Only updates the field if the specified value is greater than the existing field value.
`$mul`	Multiplies the value of the field by the specified amount.
`$rename`	Renames a field.
`$set`	Sets the value of a field in a document.
`$setOnInsert`	Sets the value of a field if an update results in an insert of a document. Has no effect on update operations that modify existing documents.
`$unset`	Removes the specified field from a document.

Example

Create the `products` collection:

```
db.products.insertOne(   {      _id: 1,      sku: "abc123",      quantity: 10,      metrics:
{ orders: 2, ratings: 3.5 }    })
```

The following updateOne() operation uses the `$inc` operator to:

- increase the `"metrics.orders"` field by 1
- increase the `quantity` field by -2 (which decreases `quantity`)

```
db.products.updateOne(   { sku: "abc123" },   { $inc: { quantity: -2, "metrics.orders": 1
} })
```

The updated document would resemble:

```
{  _id: 1,  sku: 'abc123',  quantity: 8,  metrics: { orders: 3, ratings: 3.5 }}
```

What is Aggragation ?

- Aggregationis a way of processing a large number of documents in a collection by means of passing them through different stages.
- The stages make up what is known as a pipeline. The stages in a pipeline can filter, sort, group, reshape and modify documents that pass through the pipeline.
- One of the most common use cases of Aggregation is to calculate aggregate values for groups of documents.
- This is similar to the basic aggregation available in SQL with the GROUP BY clause and COUNT, SUM and AVG functions.

Aggragation: sort()

Method specifies the order in which the query returns the matching documents from the given collection. You must apply this method to the cursor before retrieving any documents from the database. It takes a document as a parameter that contains a field: value pair that defines the sort order of the result set. The value is 1 or -1 specifying an ascending or descending sort respectively.

- If a sort returns the same result every time we perform on same data, then such type of sort is known as a stable sort.
- If a sort returns a different result every time we perform on same data, then such type of sort is known as unstable sort.
- MongoDB generally performs a stable sort unless sorting on a field that holds duplicate values.
- We can use limit() method with sort() method, it will return first m documents, where m is the given limit.

- MongoDB can find the result of the sort operation using indexes.
- If MongoDB does not find sort order using index scanning, then it uses top-k sort algorithm.

Syntax:

```
db.Collection_Name.sort({field_name:1 or -1})
```

Parameter:

- The parameter contains a field: value pair that defines the sort order of the result set. The value is 1 or -1 that specifies an ascending or descending sort respectively. The type of parameter is a document.

Return:

- It returns the documents in sorted order.

Aggregation: $limit

This aggregation stage limits the number of documents passed to the next stage.

Example

```
db.movies.aggregate([ { $limit: 1 } ])
```

- This will return the 1 movie from the collection.

Aggregation Commands

Name	Description
`aggregate`	Performs aggragation 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.

Aggregation Methods

Name	Description
`db.collection.aggregate()`	Provides access to the aggragation pipeling
`db.collection.mapReduce()`	Performs map-reduce aggregation for large data sets.

What is the Aggregation Pipeline in MongoDB?

- The aggregation pipeline refers to a specific flow of operations that processes, transforms, and returns results. In a pipeline, successive operations are informed by the previous result.
- Let's take a typical pipeline:

  Input -> `$match` -> `$group` -> `$sort` -> `output`

- In the above example, input refers to one or more documents. `$match`, `$group`, and `$sort` are various stages in a pipeline.
- The output from the `$match` stage is fed into `$group` and then the output from the `$group` stage into `$sort`.
- These three stages collectively can be called an **aggregation pipeline**.
- Implementing a pipeline helps us to **break down queries into easier stages**. Each stage uses namesake operators to complete transformation so that we can achieve our goal.

Arithmetic Expression Operators

Arithmetic expressions perform mathematic operations during aggragation process on numbers. Some arithmetic expressions can also support date arithmetic.

Name	Description
`$abs`	Returns the absolute value of a number.
`$add`	Adds numbers to return the sum, or adds numbers and a date to return a new date. If adding numbers and a date, treats the numbers as milliseconds. Accepts any number of argument expressions, but at most, one expression can resolve to a date.
`$ceil`	Returns the smallest integer greater than or equal to the specified number.
`$divide`	Returns the result of dividing the first number by the second. Accepts two argument expressions.
`$exp`	Raises *e* to the specified exponent.
`$floor`	Returns the largest integer less than or equal to the specified number.
`$ln`	Calculates the natural log of a number.
`$log`	Calculates the log of a number in the specified base.
`$log10`	Calculates the log base 10 of a number.
`$mod`	Returns the remainder of the first number divided by the second. Accepts two argument expressions.
`$multiply`	Multiplies numbers to return the product. Accepts any number of argument expressions.
`$pow`	Raises a number to the specified exponent.
`$round`	Rounds a number to to a whole integer *or* to a specified decimal place.
`$sqrt`	Calculates the square root.
`$subtract`	Returns the result of subtracting the second value from the first. If the two values are numbers, return the difference. If the two values are dates, return the difference in milliseconds. If the two values are a date and a number in milliseconds, return the resulting date. Accepts two argument expressions. If the two values are a date and a number, specify the date argument first as it is not meaningful to subtract a date from a number.
`$trunc`	Truncates a number to a whole integer *or* to a specified decimal place.

Overview of React

- React, often pronounced as "React.js" or simply "React," is an open-source JavaScript library used for building user interfaces (UIs) and handling the view layer of web applications. Developed and maintained by Facebook (now Meta) and a community of developers, React has gained widespread popularity for its efficiency, modularity, and the ability to create dynamic and interactive UIs.
- React has revolutionized the way modern web applications are built, making it easier to create interactive, scalable, and maintainable UIs. Its popularity and extensive ecosystem make it a valuable skill for web developers.

key concepts and features of React

1. **Component-Based Architecture:** React follows a component-based architecture, where UIs are broken down into reusable building blocks called components. Each component encapsulates its own logic, state, and rendering, allowing for easier management and maintenance of complex UIs.
2. **Virtual DOM:** React uses a virtual DOM (Document Object Model) to improve performance. The virtual DOM is a lightweight representation of the actual DOM, and React updates the real DOM efficiently by comparing the changes in the virtual DOM. This minimizes direct manipulation of the actual DOM, which can be a resource-intensive process.
3. **Declarative Syntax:** React employs a declarative approach to building UIs. Developers describe what the UI should look like in a given state, and React handles the underlying updates to achieve that state. This is in contrast to imperative programming, where developers explicitly define each step to take to achieve a desired result.
4. **JSX (JavaScript XML):** JSX is a syntax extension for JavaScript that allows developers to write HTML-like code within their JavaScript code. JSX makes it easier to describe the structure of UI components and their interactions, enhancing the readability and maintainability of the code.
5. **State and Props:** React components can hold two types of data: state and props. State represents the internal data that can change over time, while props are the properties passed to a component from its parent component. By managing state and props, components can maintain their data and interactions.
6. **Component Lifecycle:** React components have a lifecycle with various phases, such as mounting, updating, and unmounting. Developers can hook into these lifecycle phases to perform actions like initializing state, fetching data, and cleaning up resources.

7. **Hooks:** Introduced in React 16.8, hooks are functions that allow developers to use state and other React features in functional components (components written as functions) instead of class components. Hooks provide a more concise and readable way to manage component logic.

8. **Context:** Context provides a way to share data between components without having to pass props explicitly at each level of the component tree. It's often used for managing global state or theme data.

9. **Routing:** While React itself focuses on the view layer, developers often use additional libraries like React Router to handle routing and navigation within a single-page application (SPA).

10. **Server-Side Rendering (SSR) and Static Site Generation (SSG):** React supports server-side rendering, where the initial rendering of a page occurs on the server before sending it to the client. This can improve initial page load times and SEO. Additionally, tools like Next.js build upon React to enable static site generation, pre-rendering entire pages as static HTML files.

11. **Community and Ecosystem:** React has a vibrant and active community, with a plethora of third-party libraries, tools, and resources available to streamline development. This ecosystem includes state management libraries like Redux and MobX, UI component libraries, testing frameworks, and more.

React has revolutionized the way modern web applications are built, making it easier to create interactive, scalable, and maintainable UIs. Its popularity and extensive ecosystem make it a valuable skill for web developers.

Using React With HTML

- The quickest way start learning React is to write React directly in your HTML files.
- To get an overview of what React is, you can write React code directly in HTML. But in order to use React in production, you need npm and Node.js installed.
- Start by including three scripts, the first two let us write React code in our JavaScripts, and the third, Babel, allows us to write JSX syntax and ES6 in older browsers.

```
<!DOCTYPE html>
<html>
  <head>
    <script src="https://unpkg.com/react@18/umd/react.development.js" crossorigin>
</script>
    <script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"
crossorigin></script>
    <script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
  </head>
  <body>

    <div id="mydiv"></div>

    <script type="text/babel">
      function Hello() {
        return <h1>Hello World!</h1>;
      }

      const container = document.getElementById('mydiv');
      const root = ReactDOM.createRoot(container);
      root.render(<Hello />)
    </script>

  </body>
</html>
```

- This way of using React can be OK for testing purposes, but for production you will need to set up a React environment.

Setting up React environment

Setting up a React development environment involves installing the necessary tools and dependencies to start building React applications. Here's a step-by-step guide to help you set up your React development environment:

1. **Node.js and npm:** React applications require Node.js and npm (Node Package Manager) to manage dependencies and run scripts. Visit the official Node.js website and download the LTS (Long-Tived Support) version, which includes npm.

   - Node.js website: https://nodejs.org/

   To check if Node.js and npm are installed, run these commands in your terminal:

   `node -v`

   `npm -v`

2. **Create React App (CRA):** Create React App is a tool that sets up a new React project with a pre-configured development environment. It includes everything you need to get started without having to set up the build process manually.

   Install Create React App globally using npm:

   ```
   npm install -g create-react-app
   ```

3. **Create a New React Project:** Use Create React App to generate a new React project. Replace `my-react-app` with your preferred project name:

   ```
   npx create-react-app my-react-app
   ```

   ```
   cd my-react-app
   ```

4. **Start the Development Server:** Once your project is created, navigate to its directory and start the development server:

   ```
   npm start
   ```

   This will start the development server and open your React app in a web browser. The server will automatically reload the app whenever you make changes to the source code.

5. **Project Structure:** Create React App sets up a basic project structure for you. Common directories and files include:

   - `src`: This is where your application's source code resides.
   - `public`: Contains the public assets and the `index.html` file where your app is initially rendered.
   - `node_modules`: Contains the installed dependencies.
   - `package.json`: Defines project metadata, scripts, and dependencies.
   - `App.js` and `index.js`: These are the entry points for your React app.

6. **Building and Deployment:** To build your React app for production, use the following command:

   bashCopy code

   ```
   npm run build
   ```

   This will create an optimized build of your app in the `build` directory. You can then deploy the contents of this directory to a web server.

> This is a basic setup. As your React projects become more complex, you might need to integrate additional tools and libraries for state management, routing, styling, and more. The development environment you create now will serve as the foundation for building powerful and efficient React applications.

React Components

- In React, components are the building blocks used to create user interfaces. Components encapsulate UI elements, logic, and behavior, making it easier to manage and organize complex applications.
- React applications are typically composed of multiple nested components that work together to create a cohesive user experience.
- There are two main types of components in React: functional components and class components.

1. **Functional Components:** Functional components are defined as JavaScript functions. They receive props (input data) as arguments and return JSX (JavaScript XML) to define the UI. Functional components are simple and lightweight, and they're the recommended way to define components in React.

```
import React from 'react';

function FunctionalComponent(props) {
  return <h1>Hello, {props.name}!</h1>;
}

export default FunctionalComponent;
```

2. **Class Components:** Class components are defined as ES6 classes that extend React's `Component` class. They have a more complex syntax compared to functional components but provide additional features, such as state and lifecycle methods. However, with the introduction of React Hooks, functional components can handle state and lifecycle functionality as well.

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

class ClassComponent extends Component {
  render() {
    return <h1>Hello, {this.props.name}!</h1>;
  }
}

export default ClassComponent;
```

3. **Props:** Props (short for "properties") are a way to pass data from a parent component to a child component. Props are read-only and help components communicate and share information.

   In the Given Example `name` attribute is prop and `Alice` is the value which is being passed from `ParentComponent` to `FunctionalComponent`

```
import React from 'react';
import FunctionalComponent from './FunctionalComponent';

function ParentComponent() {
  return <FunctionalComponent name="Alice" />;
}

export default ParentComponent;
```

4. **State:** State is a way to manage data that can change over time within a component. State is specific to class components, but with the introduction of React Hooks, functional components can also manage state using the `useState` hook.

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

class StateExample extends Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Increment
        </button>
      </div>
    );
  }
}

export default StateExample;
```

5. **Lifecycle Methods (Class Components):** Class components have lifecycle methods that allow you to control and respond to different phases of a component's existence, such as mounting, updating, and unmounting.

   With the introduction of React Hooks, functional components can achieve similar behavior using hooks like `useEffect`.

> React components, whether functional or class-based, are at the core of building applications using the React library. They provide a structured way to manage UI elements, state, props, and interactions, making it easier to create dynamic and interactive web applications.

Interactive Components in React

Interactive components in React are UI elements that respond to user actions, such as clicks, input changes, and more. React provides a straightforward way to create interactive elements by combining state management, event handling, and component rendering. Here are some common examples of interactive components in React:

1. **Buttons:**

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

function InteractiveButton() {
  const [clickCount, setClickCount] = useState(0);

  const handleButtonClick = () => {
    setClickCount(clickCount + 1);
  };

  return (
    <div>
      <button onClick={handleButtonClick}>Click me</button>
      <p>Clicked {clickCount} times</p>
    </div>
  );
}

export default InteractiveButton;
```

2. **Input Fields:** Input fields allow users to enter text or data. You can capture the input value using the `useState` hook and update it based on user input.

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

function InteractiveInput() {
  const [inputValue, setInputValue] = useState('');

  const handleInputChange = (event) => {
    setInputValue(event.target.value);
  };

  return (
    <div>
      <input type="text" value={inputValue} onChange={handleInputChange} />
      <p>Input value: {inputValue}</p>
    </div>
  );
}

export default InteractiveInput;
```

3. **Checkbox and Radio Buttons:** Checkbox and radio button components allow users to select options. Use state to manage the selected option(s).

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

function InteractiveCheckbox() {
  const [isChecked, setIsChecked] = useState(false);

  const handleCheckboxChange = () => {
    setIsChecked(!isChecked);
  };

  return (
    <div>
      <label>
        <input type="checkbox" checked={isChecked} onChange={handleCheckboxChange} />
        Check me
      </label>
    </div>
  );
}

export default InteractiveCheckbox;
```

4. **Dropdowns (Select):** Dropdowns or select components let users choose from a list of options. Use the `value` attribute and `onChange` event to manage the selected option.

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

function InteractiveSelect() {
  const [selectedOption, setSelectedOption] = useState('');

  const handleSelectChange = (event) => {
    setSelectedOption(event.target.value);
  };

  return (
    <div>
      <select value={selectedOption} onChange={handleSelectChange}>
        <option value="option1">Option 1</option>
        <option value="option2">Option 2</option>
        <option value="option3">Option 3</option>
      </select>
      <p>Selected option: {selectedOption}</p>
    </div>
  );
}

export default InteractiveSelect;
```

> These are just a few examples of how you can create interactive components in React.

Components within Components and Files in React

In React, components can be organized within components and across files to create a modular and maintainable application. This modular approach enables you to manage the complexity of your application by breaking it down into smaller, reusable pieces. Let's explore how components can be nested within each other and how you can structure your files in a React application:

Components Within Components (Nested Components):

You can nest components within each other to create a hierarchy of UI elements. This is useful for creating complex user interfaces where different parts of the UI are encapsulated into their own components. Here's an example of nesting components:

```
// Button.js
import React from 'react';

function Button(props) {
  return <button onClick={props.onClick}>{props.label}</button>;
}

export default Button;
```

```
// App.js
import React from 'react';
import Button from './Button'; // Import the nested Button component

function App() {
  const handleClick = () => {
    console.log('Button clicked!');
  };

  return (
    <div>
      <h1>Hello, React!</h1>
      <Button label="Click me" onClick={handleClick} />
    </div>
  );
}

export default App;
```

File Structure Organization:

A well-organized file structure is crucial for maintaining a React application. Here's a common file structure pattern that you might use:

lessCopy code

```
src/
  components/
    Button.js
    Header.js
    // Other component files
  pages/
    Home.js
    About.js
    // Other page files
  App.js
  index.js
```

- `src/components`: This directory holds all your reusable components. Each component has its own file, making it easy to locate and manage them.
- `src/pages`: This directory contains your page-level components. Each page is a composition of smaller components, allowing you to define different views of your application.
- `App.js`: The main component that acts as the entry point for your application. It might assemble different components or pages.
- `index.js`: The entry point of your application that renders the `App` component into the DOM.

This structure keeps your code organized, makes it easier to collaborate with other developers, and enhances maintainability as your application grows.

Remember that you can organize your files in a way that best suits your project's needs. The goal is to create a structure that promotes reusability, readability, and maintainability while following best practices.

Class Components

A "React class" generally refers to a class-based component in React. In earlier versions of React, class components were the primary way to create components with state and lifecycle methods. However, with the introduction of React Hooks, functional components have become the preferred approach due to their simplicity and reusability. Nevertheless, understanding class components is still important, as you may encounter them in older codebases or when working on projects that haven't been migrated to functional components with hooks.

Here's an overview of a class component in React:

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

class ClassComponent extends Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  componentDidMount() {
    console.log('Component has mounted');
  }

  componentDidUpdate() {
    console.log('Component has updated');
  }

  componentWillUnmount() {
    console.log('Component will unmount');
  }

  handleIncrement = () => {
    this.setState({ count: this.state.count + 1 });
  };

  render() {
    return (
      <div>
        <h1>Class Component Example</h1>
        <p>Count: {this.state.count}</p>
        <button onClick={this.handleIncrement}>Increment</button>
      </div>
    );
  }
}

export default ClassComponent;
```

In this example:

- We're importing `React` and `Component` from the `react` module.
- The `ClassComponent` class extends `Component`, which provides access to React's lifecycle methods.
- We initialize the component's state in the constructor.
- `componentDidMount` is called after the component has been added to the DOM.
- `componentDidUpdate` is called when the component updates (e.g., state changes).

- `componentWillUnmount` is called just before the component is removed from the DOM.
- The `handleIncrement` method updates the state when the button is clicked.
- The `render` method returns the JSX that defines the component's UI.

Conditional Statements, Operators, Lists

In React, you can use conditional statements, operators, and lists to dynamically render UI components based on different conditions or data. These concepts allow you to create dynamic and interactive user interfaces. Here's how you can use them:

Conditional Statements:

You can use conditional statements like `if`, `else if`, and `else` within your JSX to render different components or content based on specific conditions.

```
import React from 'react';

function ConditionalComponent(props) {
  if (props.isLoggedIn) {
    return <p>Welcome, {props.username}!</p>;
  } else {
    return <p>Please log in.</p>;
  }
}

export default ConditionalComponent;
```

Operators:

You can use JavaScript operators within your JSX to evaluate conditions and display content accordingly.

```
import React from 'react';

function OperatorComponent(props) {
  const isEven = props.number % 2 === 0;

  return (
    <div>
      <p>The number is {isEven ? 'even' : 'odd'}.</p>
    </div>
  );
}

export default OperatorComponent;
```

Lists (Mapping over Arrays):

To render lists of items dynamically, you can use the `map` function to iterate over an array and generate UI components for each item.

```
import React from 'react';

function ListComponent(props) {
  const items = props.items;

  return (
    <ul>
      {items.map((item, index) => (
        <li key={index}>{item}</li>
      ))}
    </ul>
  );
}

export default ListComponent;
```

React Events

React provides a way to handle events in components, similar to how you would handle events in regular HTML elements. However, there are some differences and considerations to keep in mind when working with React events. Here's an overview of React events and how to use them:

Handling Events in React:

1. **Event Handling Syntax:** In React, event handlers are defined using camelCase instead of lowercase as in HTML. Additionally, you pass a function reference to the event handler instead

of a string.

```
import React from 'react';

function EventComponent() {
  const handleClick = () => {
    console.log('Button clicked');
  };

  return <button onClick={handleClick}>Click me</button>;
}

export default EventComponent;
```

2. **Event Object:** React event handlers receive an event object as the first parameter. This event object is a wrapper around the native browser event and has some React-specific properties and methods.

```
import React from 'react';

function EventObjectComponent() {
  const handleClick = (event) => {
    console.log('Button clicked');
    console.log('Event type:', event.type);
    console.log('Target element:', event.target);
  };

  return <button onClick={handleClick}>Click me</button>;
}

export default EventObjectComponent;
```

3. **Binding Event Handlers:** When passing a method as a callback to an event handler, you need to make sure that the method is bound to the correct instance of the component. You can achieve this by using the arrow function syntax or binding the method in the constructor.

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

class BindingComponent extends Component {
  constructor(props) {
    super(props);
    this.state = { message: 'Hello' };

    // Binding the method in the constructor
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    console.log(this.state.message);
  }

  render() {
    return <button onClick={this.handleClick}>Click me</button>;
  }
}

export default BindingComponent;
```

Adding Event

In React, you can add event handling to components to make them respond to user interactions. Events like clicks, input changes, and more can be handled using event listeners. Here's how you can add events to React components:

1. **Functional Components:**

   In functional components, you can directly define event handlers as functions within the component's body.

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

function EventComponent() {
  const [count, setCount] = useState(0);

  const handleClick = () => {
    setCount(count + 1);
  };

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={handleClick}>Increment</button>
    </div>
  );
}

export default EventComponent;
```

2. **Class Components:**

   In class components, you can define event handlers as methods within the component's class. Make sure to bind the event handler to the correct instance of the component if you're using class components.

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

class EventComponent extends Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

  handleClick = () => {
    this.setState({ count: this.state.count + 1 });
  };

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={this.handleClick}>Increment</button>
      </div>
    );
  }
}

export default EventComponent;
```

3. **Passing Parameters:**

   If you need to pass parameters to an event handler, you can use an arrow function to wrap the handler. This is especially useful when mapping over lists.

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

function ListComponent() {
  const [items, setItems] = useState(['Apple', 'Banana', 'Orange']);

  const handleItemClick = (item) => {
    console.log(`Clicked: ${item}`);
  };

  return (
    <ul>
      {items.map((item, index) => (
        <li key={index} onClick={() => handleItemClick(item)}>
          {item}
        </li>
      ))}
    </ul>
  );
}

export default ListComponent;
```

React supports a wide range of events, such as `onClick`, `onChange`, `onSubmit`, `onMouseOver`, and many more. You can attach event handlers to various JSX elements to create interactive and dynamic user interfaces. Just remember to follow the proper syntax for functional components, class components, and event handler functions.

Forms in React

Forms in React work similarly to HTML forms but with some additional considerations due to React's component-based nature. React provides a way to manage form inputs and their state, handle user input, and submit data. Here's how you can work with forms in React:

Controlled Components:

In React, form inputs are typically controlled components, which means their value is controlled by the component's state. You use the `value` prop and the `onChange` event handler to manage the input value.

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

function FormComponent() {
  const [name, setName] = useState('');
  const [email, setEmail] = useState('');

  const handleSubmit = (event) => {
    event.preventDefault();
    console.log('Name:', name);
    console.log('Email:', email);
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Name:
        <input type="text" value={name} onChange={(e) => setName(e.target.value)} />
      </label>
      <label>
        Email:
        <input type="email" value={email} onChange={(e) => setEmail(e.target.value)} />
      </label>
      <button type="submit">Submit</button>
    </form>
  );
}

export default FormComponent;
```

Select and Textarea:

For `<select>` and `<textarea>` elements, you can use the same controlled component approach.

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

function SelectTextareaComponent() {
  const [selectedOption, setSelectedOption] = useState('option2');
  const [textareaValue, setTextareaValue] = useState('');

  const handleSelectChange = (event) => {
    setSelectedOption(event.target.value);
  };

  const handleTextareaChange = (event) => {
    setTextareaValue(event.target.value);
  };

  return (
    <div>
      <select value={selectedOption} onChange={handleSelectChange}>
        <option value="option1">Option 1</option>
        <option value="option2">Option 2</option>
        <option value="option3">Option 3</option>
      </select>
      <textarea value={textareaValue} onChange={handleTextareaChange} />
    </div>
  );
}

export default SelectTextareaComponent;
```

Uncontrolled Components:

In some cases, you might need to work with uncontrolled components where you don't manage the input value in the component's state. However, controlled components are recommended as they provide more control and predictable behavior.

Handling Form Submission:

Use the `onSubmit` event handler on the `<form>` element to handle form submission. Prevent the default behavior using `event.preventDefault()` to avoid a page reload.

```
import React from 'react';

function UncontrolledComponent() {
  const handleSubmit = (event) => {
    event.preventDefault();
    const name = event.target.name.value;
    const email = event.target.email.value;
    console.log('Name:', name);
    console.log('Email:', email);
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Name:
        <input type="text" name="name" />
      </label>
      <label>
        Email:
        <input type="email" name="email" />
      </label>
      <button type="submit">Submit</button>
    </form>
  );
}

export default UncontrolledComponent;
```

Working with forms in React involves using controlled components to manage input values and providing event handlers for form submission and user input. This approach ensures that React's state management handles form data effectively.

event.target

In React, when handling events, the `event` object represents the event itself and contains information about the event that occurred. The `event.target` property refers to the DOM element that triggered the event. It's important to note that `event.target` is a plain JavaScript DOM element, not a React component.

Here's an example of how you might use `event.target` to handle an `onClick` event:

```
import React from 'react';

function ClickComponent() {
  const handleClick = (event) => {
    // event.target refers to the DOM element that was clicked
    const clickedElement = event.target;
    console.log('Element clicked:', clickedElement);
  };

  return <div onClick={handleClick}>Click me!</div>;
}

export default ClickComponent;
```

In this example, when the `<div>` element is clicked, the `handleClick` function is called, and the `event.target` property points to the specific `<div>` element that was clicked.

Remember that React's strength lies in its component-based architecture, and it encourages you to work with components rather than directly manipulating the DOM. In most cases, you'll handle events within React components, and the information you extract from `event.target` might be used to determine how to update the component's state or trigger other React-related actions.

event.target.name

In React, `event.target.name` is commonly used to access the `name` attribute of the form element that triggered an event. This is particularly useful when you have multiple form inputs and you want to identify which input triggered a specific event, such as an `onChange` event. By using `event.target.name`, you can dynamically update the corresponding state property for the input.

Here's an example of how `event.target.name` is used in a controlled component to handle form inputs:

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

function FormComponent() {
  const [formData, setFormData] = useState({
    name: '',
    email: ''
  });

  const handleChange = (event) => {
    const { name, value } = event.target;
    setFormData((prevData) => ({
      ...prevData,
      [name]: value
    }));
  };

  const handleSubmit = (event) => {
    event.preventDefault();
    console.log('Form data:', formData);
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Name:
        <input type="text" name="name" value={formData.name} onChange={handleChange} />
      </label>
      <label>
        Email:
        <input type="email" name="email" value={formData.email} onChange={handleChange}
/>
      </label>
      <button type="submit">Submit</button>
    </form>
  );
}

export default FormComponent;
```

In this example, each input element has a `name` attribute corresponding to a property in the `formData` state object. The `handleChange` function uses `event.target.name` to dynamically update the corresponding property when the input value changes. This way, you can manage multiple form inputs using a single state object.

Using `event.target.name` allows you to create more generic and reusable event handlers for your form inputs, as you can adapt the handler's behavior based on the name of the input that triggered the event.

React Memo

React's `memo` function is a higher-order component that you can use to optimize the rendering performance of functional components. It helps prevent unnecessary re-renders by memoizing the component's output based on its props. This is particularly useful when dealing with components that might re-render frequently, but their props haven't changed.

Here's how you can use `React.memo` to optimize functional components:

```
import React from 'react';

// Regular functional component
function RegularComponent(props) {
  console.log('RegularComponent rendered');
  return <div>{props.value}</div>;
}

// Memoized functional component using React.memo
const MemoizedComponent = React.memo(RegularComponent);

function App() {
  const [count, setCount] = React.useState(0);

  return (
    <div>
      <button onClick={() => setCount(count + 1)}>Increment Count</button>
      <MemoizedComponent value={count} />
    </div>
  );
}

export default App;
```

In this example, the `RegularComponent` renders whenever its parent component (`App`) renders, regardless of whether the `value` prop changes. This can lead to unnecessary re-renders.

By wrapping `RegularComponent` with `React.memo`, we create the `MemoizedComponent`. Now, the `MemoizedComponent` will only re-render when its `value` prop changes, optimizing performance by avoiding unnecessary renders.

- Keep in mind Useing `React.memo` for components that have a significant number of re-renders and receive the same props most of the time.

Component : Textarea

In React, you can create a textarea component by using the `<textarea>` HTML element. To make it a controlled component, you'll manage its value using React's state. Here's how you can create a textarea component in a functional component:

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

function TextareaComponent() {
  const [text, setText] = useState('');

  const handleTextareaChange = (event) => {
    setText(event.target.value);
  };

  return (
    <div>
      <h2>Textarea Component</h2>
      <textarea
        value={text}
        onChange={handleTextareaChange}
        rows={4} // Specify the number of visible rows
        cols={50} // Specify the number of visible columns
        placeholder="Enter your text here"
      />
      <p>Character Count: {text.length}</p>
    </div>
  );
}

export default TextareaComponent;
```

In this example:

- The `text` state variable is used to store the value of the textarea.
- The `handleTextareaChange` function is called whenever the content of the textarea changes. It updates the `text` state with the new value.
- The `value` prop of the `<textarea>` element is set to the `text` state, making it a controlled component.
- The `rows` and `cols` props define the initial number of visible rows and columns in the textarea.
- The `placeholder` prop provides a placeholder text that appears before the user enters any content.

Component : Drop Down List(Select)

To create a dropdown list (select element) in React, you can use the `<select>` HTML element along with the `<option>` elements to define the available options. You'll also use React's state to manage the selected option. Here's how you can create a dropdown list in a functional component:

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

function DropdownComponent() {
  const [selectedOption, setSelectedOption] = useState('option2');

  const handleSelectChange = (event) => {
    setSelectedOption(event.target.value);
  };

  return (
    <div>
      <h2>Dropdown List Component</h2>
      <select value={selectedOption} onChange={handleSelectChange}>
        <option value="option1">Option 1</option>
        <option value="option2">Option 2</option>
        <option value="option3">Option 3</option>
      </select>
      <p>Selected Option: {selectedOption}</p>
    </div>
  );
}

export default DropdownComponent;
```

In this example:

- The `selectedOption` state variable is used to store the value of the selected option in the dropdown.
- The `handleSelectChange` function is called whenever the selected option changes. It updates the `selectedOption` state with the new value.
- The `value` prop of the `<select>` element is set to the `selectedOption` state, making it a controlled component.
- Each `<option>` element within the `<select>` element defines an available option. The `value` attribute corresponds to the value of the option.
- The text between the opening and closing `<option>` tags is the visible label for the option.

You can enhance the dropdown component by populating options dynamically from an array, setting a default value, or adding additional attributes like `disabled` and `label`. This basic example demonstrates how to create a controlled dropdown list in a React functional component.

Hooks

React Hooks are a set of functions that allow you to "hook into" React state and lifecycle features from functional components. They were introduced in React 16.8 and provide a more concise and readable way to work with state, side effects, and lifecycle behavior in functional components.

Advantages of React Hooks

React Hooks introduced a new way of writing functional components in React, providing several advantages that enhance development and code organization. Here are some key advantages of using React Hooks:

1. **Simplicity and Readability:** Hooks allow you to write components in a more straightforward and concise manner. The code becomes more readable and easier to understand, reducing the cognitive load when working with complex components.
2. **Elimination of Class Components:** Hooks enable you to use state and lifecycle features without having to use class components. This leads to more consistent codebases and makes it easier to migrate from class components to functional components.
3. **Reusability with Custom Hooks:** Custom hooks allow you to encapsulate and share logic across components. This promotes code reuse and modularity, making it easier to maintain and test your application.
4. **Functional Composition:** Hooks encourage the use of functional programming concepts by enabling you to break down logic into smaller, composable functions. This promotes better separation of concerns and makes it easier to reason about the behavior of your components.
5. **Local State Management:** Hooks like `useState` provide a straightforward way to manage local component state. This is particularly useful for smaller components that don't require the complexity of a centralized state management solution.
6. **Lifecycle Management:** The `useEffect` hook simplifies handling side effects and mimics the behavior of lifecycle methods. It provides a centralized place to manage all component-related side effects.
7. **Performance Optimization:** Hooks like `useMemo` and `useCallback` help optimize performance by preventing unnecessary re-renders and calculations. This is especially valuable for components with heavy computation or rendering.
8. **Ease of Learning:** For developers new to React, Hooks provide a more cohesive and straightforward way to learn React concepts without the need to grasp class component intricacies.

In summary, React Hooks bring multiple advantages to the table by simplifying code, enhancing reusability, promoting functional programming practices, and providing a more efficient way to work with state, effects, and lifecycle management in functional components.

useState Hook

The `useState` hook is one of the most commonly used hooks in React. It allows you to add state to functional components, making them dynamic and capable of re-rendering when the state changes. Here's how you can use the `useState` hook:

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

function Counter() {
  // useState returns an array with the current state value and a function to update the
state
  const [count, setCount] = useState(0);

  const handleIncrement = () => {
    setCount(count + 1); // Update the state with a new value
  };

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={handleIncrement}>Increment</button>
    </div>
  );
}

export default Counter;
```

In this example:

- The `useState` hook is imported from the `react` module.
- Inside the `Counter` component, `useState(0)` initializes a state variable named `count` with an initial value of `0`.
- `count` holds the current value of the state, and `setCount` is a function that allows you to update `count` and trigger a re-render.
- When the "Increment" button is clicked, the `handleIncrement` function updates the state by calling `setCount`, which causes the component to re-render with the updated value.

Remember that state updates using `useState` are asynchronous, and React batches multiple state updates for better performance. To update state based on the previous state, you should use the functional update pattern:

```
setCount((prevCount) => prevCount + 1);
```

This ensures that you're working with the latest state value and avoids potential race conditions.

`useState` is a fundamental hook that enables you to bring dynamic behavior to your functional components, allowing you to manage and display changing data within your UI.

useEffect Hook

The `useEffect` hook in React allows you to perform side effects in functional components, such as data fetching, DOM manipulation, or subscribing to events. It replaces lifecycle methods like `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` in class components. Here's how you can use the `useEffect` hook:

```
import React, { useState, useEffect } from 'react';

function Timer() {
  const [seconds, setSeconds] = useState(0);

  useEffect(() => {
    // This function will be executed after every render
    const interval = setInterval(() => {
      setSeconds((prevSeconds) => prevSeconds + 1);
    }, 1000);

    // Clean-up function: runs before the next effect and when the component unmounts
    return () => clearInterval(interval);
  }, []); // Empty dependency array means the effect runs only after the initial render

  return <p>Seconds: {seconds}</p>;
}

export default Timer;
```

In this example:

- The `useEffect` hook is imported from the `react` module.
- Inside the `Timer` component, `useEffect` takes two arguments: the effect function and an array of dependencies.
- The effect function is executed after the component renders and after every update if any of the dependencies change. It can contain side-effect code.
- In this case, the effect function sets up an interval to update the `seconds` state every second.
- The clean-up function returned from the effect is executed before the next effect runs and when the component unmounts. It's used here to clear the interval.
- Since an empty dependency array is passed (`[]`), the effect runs only after the initial render and doesn't depend on any specific prop or state change.

Here are some important points about using the `useEffect` hook:

- If the dependency array is not provided (`useEffect(() => {...});`), the effect runs after every render.
- If the dependency array is an empty array (`useEffect(() => {...}, []);`), the effect runs only after the initial render.
- If the dependency array contains variables (`useEffect(() => {...}, [variable1, variable2]);`), the effect runs whenever any of those variables change.
- Returning a function from the effect is a best practice for cleanup, especially for scenarios like event listeners, subscriptions, and intervals.

`useEffect` is a powerful tool for handling side effects in functional components and helps you manage the lifecycle of your component in a clear and concise way.

useContext Hook

The `useContext` hook in React allows you to access the context values provided by the nearest ancestor `Context.Provider` component in the component tree. It provides a way to share data between components without having to pass props through all the intermediary components. Here's how you can use the `useContext` hook:

1. **Create a Context:**

   First, you need to create a context using the `React.createContext` function. This creates a context object that you can use to provide and consume values.

   ```
   import React from 'react';

   const MyContext = React.createContext();

   export default MyContext;
   ```

2. **Provide Context Values:**

   Wrap your component tree with the `Context.Provider` component to provide context values to the components that need them.

```
import React from 'react';
import MyContext from './MyContext';

function App() {
  const value = 'Hello from Context';

  return (
    <MyContext.Provider value={value}>
      <ChildComponent />
    </MyContext.Provider>
  );
}

export default App;
```

3. **Consume Context Values:**

   In the component where you want to access the context value, use the `useContext` hook.

```
import React, { useContext } from 'react';
import MyContext from './MyContext';

function ChildComponent() {
  const contextValue = useContext(MyContext);

  return <p>{contextValue}</p>;
}

export default ChildComponent;
```

In this example, the `ChildComponent` is able to access the context value provided by the `MyContext.Provider` in the `App` component. This allows you to share data, such as theme preferences, authentication status, or any other global state, between different parts of your application without the need for prop drilling.

Remember that `useContext` hooks into the nearest `Context.Provider` ancestor. If you have nested contexts, each `useContext` call will correspond to the closest ancestor that provides the context value.

Using the `useContext` hook makes it more convenient to access shared values and promotes cleaner and more modular code in your React applications.

useRef, useReducer, useCallback, useMemo

1. **useRef:** The `useRef` hook allows you to create a mutable ref object that persists across renders. Refs are commonly used to access DOM elements directly, store references to values that don't trigger re-renders, or manage imperative behavior.

   ```
   import React, { useRef, useEffect } from 'react';

   function FocusableInput() {
     const inputRef = useRef(null);

     useEffect(() => {
       inputRef.current.focus();
     }, []);

     return <input ref={inputRef} />;
   }

   export default FocusableInput;
   ```

2. **useReducer:** The `useReducer` hook is an alternative to `useState` for managing more complex state logic. It takes a reducer function and an initial state and returns the current state and a dispatch function. It's suitable for cases where state transitions involve multiple actions and require more advanced state management.

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

function reducer(state, action) {
  switch (action.type) {
    case 'INCREMENT':
      return { count: state.count + 1 };
    case 'DECREMENT':
      return { count: state.count - 1 };
    default:
      return state;
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, { count: 0 });

  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={() => dispatch({ type: 'INCREMENT' })}>Increment</button>
      <button onClick={() => dispatch({ type: 'DECREMENT' })}>Decrement</button>
    </div>
  );
}

export default Counter;
```

3. **useCallback:** The `useCallback` hook is used to memoize functions, preventing unnecessary re-creations of functions across renders. It's particularly useful when passing functions to child components to ensure consistent reference equality.

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

function ParentComponent() {
  const [count, setCount] = useState(0);

  const increment = useCallback(() => {
    setCount((prevCount) => prevCount + 1);
  }, []);

  return <ChildComponent increment={increment} />;
}

function ChildComponent({ increment }) {
  return <button onClick={increment}>Increment</button>;
}

export default ParentComponent;
```

4. **useMemo:** The `useMemo` hook is used to memoize the result of a computation, preventing redundant calculations. It takes a function and a dependency array and returns the memoized result. Useful for optimizing performance when computing derived data.

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

function ExpensiveComponent({ data }) {
  const processedData = useMemo(() => {
    // Expensive data processing logic
    return data.map((item) => item * 2);
  }, [data]);

  return <div>{processedData.join(', ')}</div>;
}

export default ExpensiveComponent;
```

Each of these hooks (`useRef` , `useReducer` , `useCallback` , and `useMemo`) serves a specific purpose in enhancing your React components' functionality, performance, and overall development experience.

Building Custom hook

Building a custom hook in React allows you to encapsulate reusable logic and share it across multiple components. Custom hooks are named like regular functions and can use other built-in hooks or even other custom hooks. Here's how you can create a simple custom hook:

Let's create a custom hook that manages a counter:

```javascript
import React, { useState } from 'react';

// Custom hook
function useCounter(initialCount = 0, step = 1) {
  const [count, setCount] = useState(initialCount);

  const increment = () => {
    setCount((prevCount) => prevCount + step);
  };

  const decrement = () => {
    setCount((prevCount) => prevCount - step);
  };

  return { count, increment, decrement };
}

function CounterComponent() {
  const { count, increment, decrement } = useCounter(0, 2);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
}

export default CounterComponent;
```

In this example:

1. The `useCounter` custom hook encapsulates the state and logic related to the counter.
2. The `useCounter` hook returns an object with the current `count` value and functions `increment` and `decrement`.
3. The `CounterComponent` uses the `useCounter` custom hook to manage and display the counter.

Custom hooks can be shared and reused across different components, promoting code reusability and maintaining a clean separation of concerns. This example demonstrates a basic custom hook, but you can create more complex ones that encapsulate more advanced logic and functionality.

Advantages of Custom Hook

Custom hooks offer several advantages when building applications in React:

1. **Code Reusability:** Custom hooks allow you to encapsulate logic into reusable functions that can be shared across multiple components. This reduces code duplication and promotes a more modular and maintainable codebase.
2. **Abstraction of Complex Logic:** You can encapsulate complex logic or state management patterns within a custom hook, making it easier for other components to use that logic without needing to understand its implementation details.
3. **Better Organization:** Custom hooks help organize your code by centralizing related logic into a single location. This makes it easier to manage and update functionality throughout your application.
4. **Separation of Concerns:** Custom hooks promote separation of concerns by allowing you to isolate specific functionality, such as data fetching, form handling, or animations, into dedicated hooks. This keeps your components cleaner and focused on rendering.
5. **Cleaner Components:** By abstracting away complex logic into custom hooks, your components become cleaner and more focused on rendering UI. This separation improves readability and maintainability.
6. **Testing:** Custom hooks can be tested independently, allowing you to write unit tests for the encapsulated logic without the need to consider the component's rendering.
7. **Component Composition:** Custom hooks can be combined to create more powerful hooks or to assemble complex behavior by reusing existing functionality. This encourages a higher level of composability in your application.
8. **Easier Debugging:** Isolating logic in custom hooks can simplify debugging, as you can focus on specific functionality in isolation without the noise of the entire component.

Overall, custom hooks are a powerful tool that promotes code reusability, organization, and maintainability, making your React development more efficient and effective.

Use of Custom Hooks

Custom hooks in React can be used for a wide range of purposes to encapsulate reusable logic and stateful behavior. Here are some common scenarios where custom hooks can be beneficial:

1. **Data Fetching:** Create a custom hook to handle data fetching from APIs. This can help centralize and standardize data fetching logic across your application.
2. **Form Handling:** Abstract away form state management, validation, and submission logic into a custom hook. This promotes consistency and simplifies form-related code in your components.

3. **Authentication and Authorization:** Implement authentication and authorization logic in a custom hook to manage user sessions, tokens, and access control.

4. **Global State Management:** Build custom hooks for managing global state using context or other state management libraries, offering a more streamlined API for your components to interact with global state.

5. **Media Handling:** Create a custom hook to handle media-related functionality, such as image uploading, resizing, or video playback.

Custom hooks are a powerful way to abstract away complex or reusable logic, promoting code reusability, modularity, and better organization in your React application.

What is AngularJS ?

AngularJS is an open-source JavaScript framework developed by Google for building dynamic web applications. It was initially released in 2010 and gained significant popularity among web developers due to its ability to create single-page applications (SPAs) with a more structured and organized approach.

AngularJS follows the Model-View-Controller (MVC) architectural pattern, which helps separate concerns and makes it easier to manage complex applications. Here's a brief overview of the components in AngularJS:

1. **Model**: This represents the data and the business logic of the application. In AngularJS, the model is usually defined using JavaScript objects or JSON data.

2. **View**: The view is responsible for rendering the data and presenting it to the user. In the context of AngularJS, the view is typically defined using HTML templates with additional directives and expressions provided by the framework.

3. **Controller**: The controller acts as an intermediary between the model and the view. It handles user input, updates the model, and ensures that the view reflects the current state of the model. Controllers are defined using JavaScript functions in AngularJS.

AngularJS introduced several concepts that were innovative at the time, such as data binding (automatic synchronization between the model and the view), dependency injection (managing and injecting dependencies into components), and directives (extending HTML with custom behavior and functionality).

Concepts & Charactristics Of AngularJS

AngularJS introduced several key concepts that were instrumental in its popularity and success. These concepts helped developers build dynamic and maintainable web applications. Here are some of the core concepts of AngularJS:

1. **Two-Way Data Binding**: AngularJS introduced a powerful feature called two-way data binding, which allows automatic synchronization between the model (data) and the view (UI). When the model changes, the view is updated automatically, and vice versa. This reduces the need for manual DOM manipulation.

2. **Controllers**: Controllers in AngularJS are JavaScript functions that contain the business logic of a specific part of the application. They act as intermediaries between the model and the view. Controllers manipulate the model and respond to user interactions, updating the view accordingly.

3. **Directives**: Directives are markers on a DOM element that tell AngularJS to attach a certain behavior or functionality to that element. They enable the creation of reusable UI components and custom behaviors. Common directives include `ng-repeat` (for iterating over arrays), `ng-model` (for binding input elements to data), and `ng-show`/`ng-hide` (for toggling element visibility).

4. **Templates**: Templates are HTML files that define the structure of the user interface. AngularJS templates can include directives, which allow dynamic rendering and binding of data to the UI.

5. **Scope**: The scope is a JavaScript object that represents the context in which expressions are evaluated. Scopes serve as a bridge between controllers and views, enabling data binding and communication between the two.

6. **Filters**: Filters are used to format and transform data displayed in the view. They can be applied to expressions in templates to modify the way data is presented to the user.

7. **Services**: Services in AngularJS are singleton objects that provide specific functionality and can be injected into controllers, directives, and other components. Common examples of services include the `$http` service for making HTTP requests and the `$rootScope` service for managing the top-level scope.

8. **Dependency Injection**: AngularJS employs dependency injection to manage and provide instances of various components (controllers, services, etc.). This promotes modularization and testability of code by decoupling components and their dependencies.

9. **Routing**: AngularJS introduced the concept of routing to create single-page applications (SPAs) with different views. The `ngRoute` module provides tools for defining routes and managing navigation within an application.

10. **Modules**: AngularJS applications are typically organized into modules. Modules encapsulate related components, making the application more modular, maintainable, and testable.

These concepts collectively contributed to the development of dynamic and interactive web applications using AngularJS.

Expression in AngularJS

In AngularJS, expressions are a way to bind dynamic values to the HTML templates. They allow you to interpolate values, perform calculations, and manipulate data directly within the template. Expressions are typically enclosed in double curly braces `{{ }}`. Here's how expressions work for different data types:

1. **Numbers**:

```
<p>5 + 3 = {{ 5 + 3 }}</p>
<p>{{ 10 / 2 }}</p>
```

2. **Strings**:

```
<p>{{ 'Hello, ' + 'AngularJS' }}</p>
<p>{{ 'Length: ' + 'OpenAI'.length }}</p>
```

3. **Objects**: Assuming you have an object defined in your scope:

```
$scope.person = {     firstName: 'John',     lastName: 'Doe' };
```

You can use its properties in expressions:

```
<p>{{ person.firstName }} {{ person.lastName }}</p>
```

4. **Arrays**: Assuming you have an array defined in your scope:

```
$scope.numbers = [1, 2, 3, 4, 5];
```

You can use array elements in expressions:

```
<p>{{ numbers[0] }} {{ numbers[2] }}</p>
```

5. **Functions**: You can also call functions defined in your scope from expressions:

```
$scope.calculateSum = function(a, b) {     return a + b; };
```

```
<p>3 + 7 = {{ calculateSum(3, 7) }}</p>
```

6. **Conditional Expressions**: AngularJS expressions also support ternary conditional expressions:

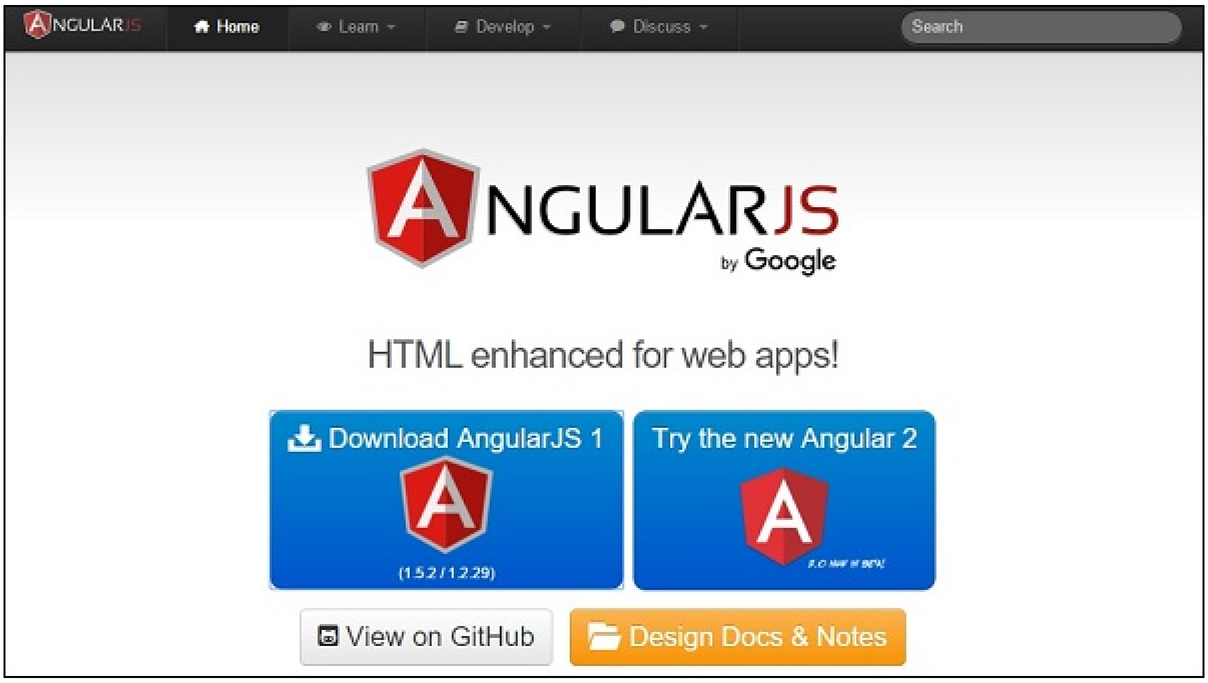
```
$scope.isEven = function(num) {     return num % 2 === 0; };
```

```
<p>{{ 8 }} is even: {{ isEven(8) ? 'Yes' : 'No' }}</p>
```

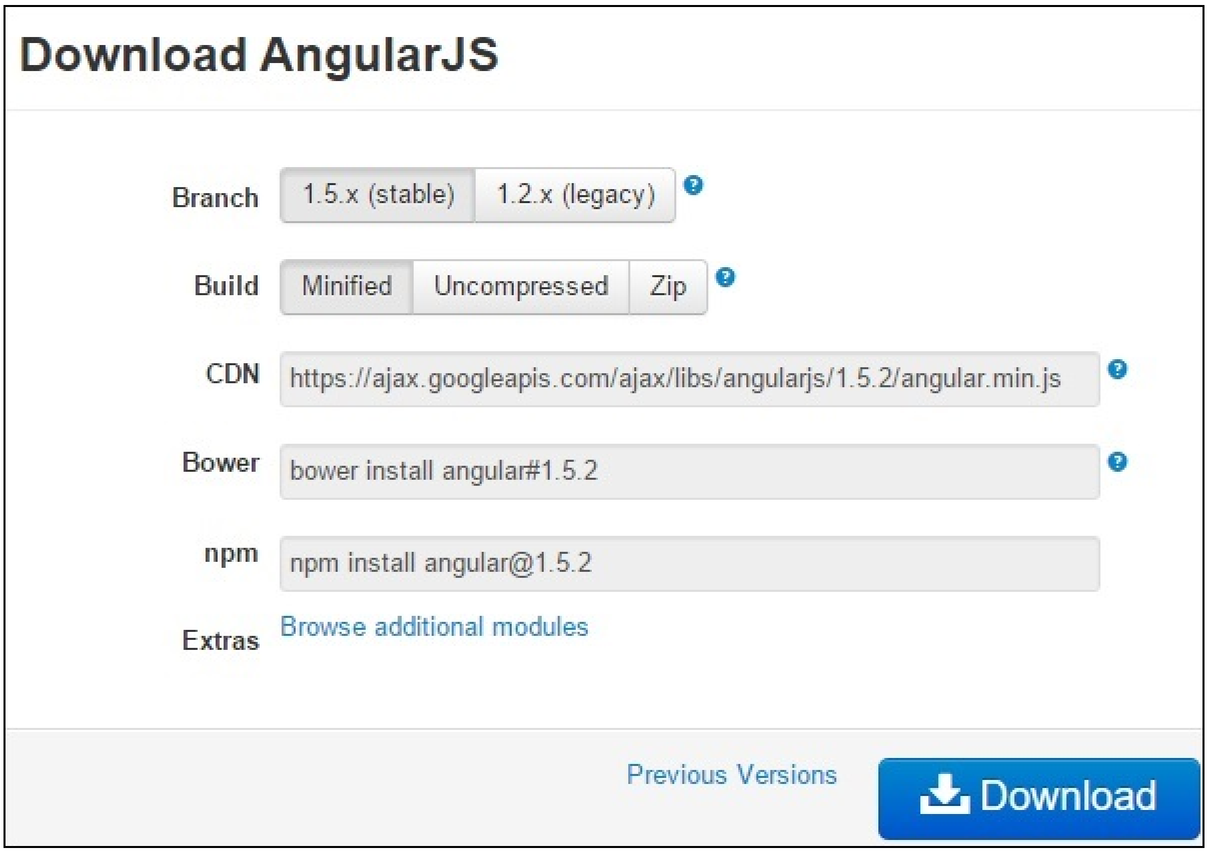
- It's important to note that AngularJS expressions are evaluated within the context of the current scope. The expressions are not full JavaScript; they are a subset designed specifically for use within AngularJS templates.
- Expressions do not support control flow statements (like loops or conditional statements) and cannot directly access global variables or execute arbitrary JavaScript code.
- Expressions are meant for relatively simple calculations and data binding within the context of templates.

Setting up Environment

- This chapter describes how to set up AngularJS library to be used in web application development. It also briefly describes the directory structure and its contents.
- 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 –


Download AngularJS

Branch: 1.5.x (stable) | 1.2.x (legacy)

Build: Minified | Uncompressed | Zip

CDN: https://ajax.googleapis.com/ajax/libs/angularjs/1.5.2/angular.min.js

Bower: bower install angular#1.5.2

npm: npm install angular@1.5.2

Extras: Browse additional modules

This screen gives various options of using Angular JS as follows −

- **Downloading and hosting files locally**
    - There are two different options : Legacy and Latest. The names themselves are self-descriptive. The Legacy has version less than 1.2.x and the Latest come with version 1.3.x.
    - We can also go with the minimized, uncompressed, or zipped version.
- **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 re-download it.

> We are using the CDN versions of the library throughout this tutorial.

Example

Now let us write a simple example using AngularJS library. Let us create an HTML file *myfirstexample.html* shown as below −

```
<!doctype html>
<html>
   <head>
      <script src =
"https://ajax.googleapis.com/ajax/libs/angularjs/1.5.2/angular.min.js"></script>
   </head>

   <body ng-app = "myapp">
      <div ng-controller = "HelloController" >
         <h2>Welcome {{helloTo.title}} to the world of Tutorialspoint!</h2>
      </div>

      <script>
         angular.module("myapp", [])

         .controller("HelloController", function($scope) {
            $scope.helloTo = {};
            $scope.helloTo.title = "AngularJS";
         });
      </script>
    </body>
</html>
```

Let us go through the above code in detail –

Include AngularJS We include the AngularJS JavaScript file in the HTML page so that we can use it –

```
<head>
   <script src = "https://ajax.googleapis.com/ajax/libs/angularjs/1.4.8/angular.min.js">
   </script>
</head>
```

You can check the latest version of AngularJS on its official website.

Point to AngularJS app

Next, it is required to tell which part of HTML contains the AngularJS app. You can do this by adding the ng-app attribute to the root HTML element of the AngularJS app. You can either add it to the html element or the body element as shown below –

```
<body ng-app = "myapp">
</body>
```

View

```
<div ng-controller = "HelloController" >
   <h2>Welcome {{helloTo.title}} to the world of Tutorialspoint!</h2>
</div>
```

ng-controller tells AngularJS which controller to use with this view. *helloTo.title* tells AngularJS to write the model value named helloTo.title in HTML at this location.

Controller

The controller part is –

```
<script>
   angular.module("myapp", [])
      .controller("HelloController", function($scope) {
      $scope.helloTo = {};
      $scope.helloTo.title = "AngularJS";
   });
</script>
```

This code registers a controller function named HelloController in the angular module named *myapp*. The controller function is registered in angular via the angular.module(...).controller(...) function call.

The $scope parameter model is passed to the controller function. The controller function adds a *helloTo* JavaScript object, and in that object it adds a *title* field.

Execution

Save the above code as *myfirstexample.html* and open it in any browser. You get to see the following output –

```
Welcome AngularJS to the world of Tutorialspoint!
```

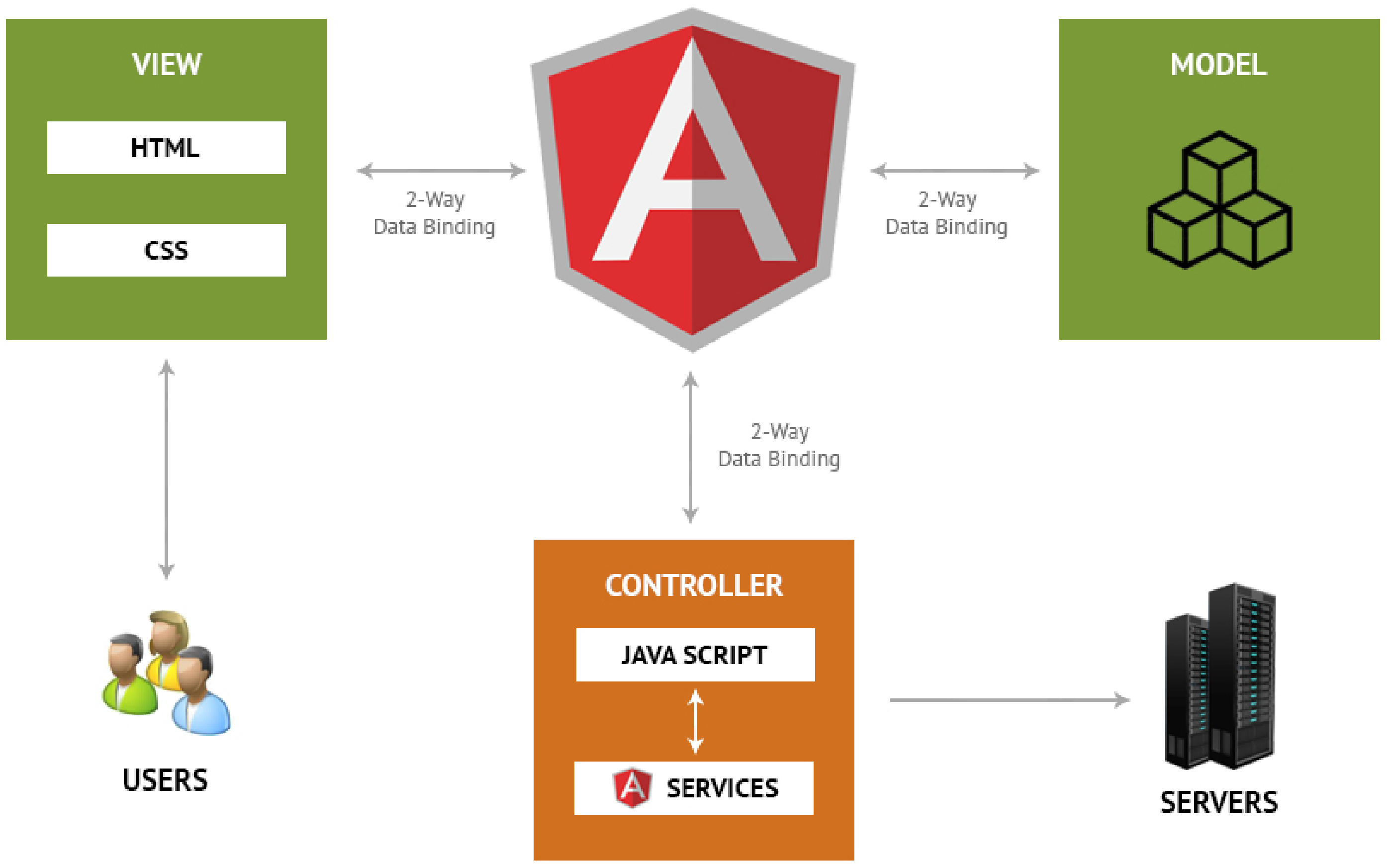
What happens when the page is loaded in the browser ? Let us see –

- HTML document is loaded into the browser, and evaluated by the browser.
- AngularJS JavaScript file is loaded, the angular *global* object is created.
- The JavaScript which registers controller functions is executed.
- Next, AngularJS scans through the HTML to search for AngularJS apps as well as views.
- Once the view is located, it connects that view to the corresponding controller function.

- Next, AngularJS executes the controller functions.
- It then renders the views with data from the model populated by the controller. The page is now ready.

Understanding MVC architecture

In AngularJS, the Model-View-Controller (MVC) architectural pattern is a fundamental concept that helps structure and organize your application code. AngularJS's interpretation of MVC is often referred to as MVVM (Model-View-ViewModel), but the principles are similar. Here's how MVC/MVVM is applied in AngularJS:



Model:

1.
   - In AngularJS, the model represents the data and the business logic of your application. It's the actual information that your application works with, such as user input, server responses, and other data.
   - The model in AngularJS is usually defined within the controllers or services. Controllers manage and manipulate the data that forms the model.
   - The `$scope` object in AngularJS is a key player here. It acts as the glue between the controller and the view, allowing data binding and communication between the two.
2. **View**:

- The view is the user interface that presents the data from the model to the user. In AngularJS, the view is typically defined using HTML templates enriched with AngularJS directives and expressions.
- Directives provide custom behaviors and enable data binding, which establishes a connection between the model and the view.
- AngularJS allows you to create dynamic views that update automatically when the underlying data changes.

3. **Controller**:

    - Controllers in AngularJS act as intermediaries between the model and the view. They handle user interactions, update the model, and ensure that the view accurately reflects the current state of the model.
    - Controllers contain the business logic that processes data and prepares it for presentation in the view.
    - The `$scope` object plays a significant role in controllers. It allows controllers to expose data and functions to the view, enabling data binding and interaction.

In the context of AngularJS, you can visualize the flow of MVC/MVVM like this:

1. **Model**: The application data is stored in JavaScript objects within controllers or services. These objects represent the state of the application.

2. **View**: HTML templates are used to define how the data from the model should be presented to the user. AngularJS directives and expressions are used to bind data and define behavior within the view.

3. **Controller**: Controllers contain the logic that manipulates the model data. They respond to user interactions and update the model accordingly. The updated model data is then automatically reflected in the view due to data binding.

AngularJS Directives

In AngularJS, directives are a powerful feature that extends HTML with new attributes or elements. They allow you to create reusable components and enhance the behavior of your application's UI. Directives play a significant role in separating concerns and making code more modular and maintainable. Here's an overview of AngularJS directives:

Built-in Directives: AngularJS comes with a set of built-in directives that you can use to enhance your application's functionality and create dynamic UI elements. Some common built-in directives include:

- `ng-repeat` : Iterates over a collection and generates HTML elements for each item.
- `ng-if` and `ng-show` / `ng-hide` : Conditionally show or hide elements based on expressions.
- `ng-model` : Binds an input element's value to a property in the model, enabling two-way data binding.
- `ng-click` : Attaches a click event to an element and executes an expression when clicked.
- `ng-class` and `ng-style` : Dynamically add classes or apply inline styles based on conditions.

Let Us Explore Some of Them in Detail:

1. **`ng-app`** : The `ng-app` directive is used to define the root of the AngularJS application. It marks the HTML element where the AngularJS framework will be initialized. You typically place this directive on the `<html>` or `<body>` element of your HTML document.

   Example:

   ```
   <!DOCTYPE html>

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

2. **`ng-init`** : The `ng-init` directive allows you to initialize values in the AngularJS scope. While it's convenient for simple cases, it's generally recommended to use controllers for more complex initialization logic.

   Example:

   ```
   <div ng-init="name = 'John'">
        <p>Hello, {{ name }}</p>
   </div>
   ```

3. **`ng-controller`** : The `ng-controller` directive associates a controller with a specific section of the HTML. It defines the scope for that section of the HTML where the controller's properties and functions can be accessed.

   Example:

   ```
   <div ng-controller="myController">
       <p>{{ greeting }}</p>
   </div>
   ```

4. **`ng-model`** : The `ng-model` directive binds an input, textarea, or select element to a property in the AngularJS scope. It enables two-way data binding, meaning changes to the input value update the scope, and changes to the scope value update the input.

   Example:

   ```
   <input type="text" ng-model="username">
   <p>Hello, {{ username }}</p>
   ```

5. **`ng-repeat`** : The `ng-repeat` directive is used to iterate over a collection (like an array or an object) and generate HTML elements for each item. It's particularly useful for rendering lists of data.

   Example:

   ```
   <ul>
       <li ng-repeat="item in items">{{ item }}</li>
   </ul>
   ```

- These directives are fundamental to building dynamic and interactive applications using AngularJS.
- They allow you to create a seamless connection between the UI and your application's logic, making it easier to manage and maintain your codebase.

Some Other Directives

1. **`ng-class`** : The `ng-class` directive allows you to conditionally apply CSS classes to an HTML element based on expressions in your AngularJS application. It's useful for dynamically changing the styling of elements.

   Example:

   ```
   <div ng-class="{ 'highlight': isHighlighted, 'error': isError }">
       Content
   </div>
   ```

   In this example, the class `highlight` will be applied if `isHighlighted` is truthy, and the class `error` will be applied if `isError` is truthy.

2. `ng-animate` (deprecated): The `ng-animate` directive was used in earlier versions of AngularJS to animate elements when they are added, removed, or updated in the DOM. However, this directive has been removed in favor of using the `ngAnimate` module.

3. **`ng-show` and `ng-hide`**: The `ng-show` and `ng-hide` directives are used to conditionally show or hide elements based on expressions. When the expression evaluates to `true`, the element is shown; when it evaluates to `false`, the element is hidden.

   Example:

   ```
   <div ng-show="showElement">
       This element is shown.
   </div>
   <div ng-hide="hideElement">
       This element is hidden.
   </div>
   ```

   In this example, the first `<div>` will be visible if `showElement` is truthy, and the second `<div>` will be hidden if `hideElement` is truthy.

Expressions and Controllers

1. **Expressions**:

   Expressions in AngularJS are used to bind dynamic values to the HTML templates. They are typically enclosed within double curly braces `{{ }}`. Expressions can contain variables, literals, operators, and function calls, allowing you to perform simple calculations and display data from the model.

   Example:

   ```
   <p> Hello, {{ name }} </p>
   <p> Age: {{ age }} </p>
   <p> Total: {{ price * quantity }} </p>
   ```

   In this example, `name`, `age`, `price`, and `quantity` are variables that are part of the model, and their values will be interpolated into the HTML.

2. **Controllers**:

   Controllers in AngularJS are JavaScript functions that act as the bridge between the model and the view. They contain the application's business logic and interact with the model to update data that's presented in the view. Controllers manage the scope of the application.

Example:

```
angular.module('myApp', [])
       .controller('myController', function($scope) {
             $scope.name = 'John';
             $scope.age = 30;
             $scope.price = 10;
             $scope.quantity = 5;
});
```

In this example, a controller named `myController` is defined. It attaches properties like `name`, `age`, `price`, and `quantity` to the `$scope` object, making them accessible to the view. The controller initializes the initial values of these properties.

You can then use this controller in your HTML:

```
<div ng-app="myApp" ng-controller="myController">
     <p>Hello, {{ name }}</p>
     <p>Age: {{ age }}</p>
     <p>Total: {{ price * quantity }}</p>
</div>
```

The `ng-controller` directive associates the controller with a specific section of the HTML.

- Controllers are important for keeping your application organized and maintainable.
- They allow you to encapsulate logic and data manipulation, ensuring that the view remains clean and focused on presentation.

Filters In AngularJS

In AngularJS, filters are used to format and transform data displayed in the UI. They allow you to modify the presentation of data before it's rendered to the user. Filters are applied to expressions within double curly braces `{{ }}` in templates. Here are some commonly used AngularJS filters:

1. **`currency`**: Formats a number as a currency string using the specified currency code.

```
<p>{{ price | currency:'USD' }}</p>
```

2. **`date`**: Formats a date object or string into a human-readable date string.

```
<p>{{ myDate | date:'yyyy-MM-dd HH:mm:ss' }}</p>
```

3. `uppercase` and `lowercase` : Converts a string to uppercase or lowercase.

```
<p>{{ 'Hello World' | uppercase }}</p>
<p>{{ 'Hello World' | lowercase }}</p>
```

4. `number` : Formats a number to a specified number of decimal places.

```
<p>{{ value | number:2 }}</p>
```

5. `orderBy` : Orders an array of objects based on a specified property.

```
<ul>
    <li ng-repeat="item in items | orderBy:'name'">{{ item.name }}</li>
</ul>
```

6. `filter` : Filters an array based on a search term or criteria.

```
<ul>
     <li ng-repeat="item in items | filter:searchText">{{ item.name }}</li>
</ul>
```

7. `limitTo` : Limits the number of items displayed in an array or string.

```
<p>{{ text | limitTo:100 }}</p>
<ul>
    <li ng-repeat="item in items | limitTo:5">{{ item.name }}</li>
</ul>
```

8. `Custom Filters` : You can also create custom filters to perform more specific transformations.

```
angular.module('myApp').filter('customFilter', function() {
    return function(input) {
        // Custom logic to transform input
        return transformedOutput;
    };
});
```

```
<p>{{ data | customFilter }}</p>
```

- These are just a few examples of AngularJS filters. Filters are a powerful way to manipulate data directly within templates, reducing the need for complex logic in controllers.
- However, be mindful not to overuse filters, especially when dealing with large datasets, as they can impact performance.

Single Page Application using AngularJS

Creating a single-page application (SPA) using AngularJS involves structuring your project, defining routes, using controllers, and implementing views. SPAs provide a more fluid and responsive user experience by loading content dynamically within a single HTML page, eliminating the need for full page reloads.

- Here's a step-by-step guide to creating a simple Single Page Application (SPA) using AngularJS, including creating a module, defining a controller, embedding AngularJS script in HTML, using `ngRoute` for routing, and navigating to different pages:

1. **Setting Up Your Project**:

   Create a new directory for your project and set up the basic structure:

   ```
   my-spa/
   ├── index.html
   ├── js/ │
           ├── angular.js │
           └── app.js
   ├── views/ │
              ├── home.html │
              └── about.html
   ```

2. **Embedding AngularJS Script in HTML**:

   In your `index.html` file, include the AngularJS script using a `<script>` tag:

```html
<!DOCTYPE html>
<html ng-app="myApp">
<head>
    <meta charset="UTF-8">
    <title>My SPA with AngularJS</title>
    <script src="js/angular.js"></script>
    <script src="js/app.js"></script>
</head>
<body>
    <header>
        <a href="#/">Home</a>
        <a href="#/about">About</a>
    </header>

    <main ng-view></main>

    <footer>
        <!-- Footer content here -->
    </footer>
</body>
</html>
```

3. **Creating a Module and Defining a Controller**:

   In your `app.js` file, define the AngularJS module and controllers:

```javascript
angular.module('myApp', ['ngRoute'])

.controller('HomeController', function($scope) {
    $scope.message = 'Welcome to the Home Page!';
})

.controller('AboutController', function($scope) {
    $scope.message = 'This is the About Page.';
});
```

4. **Using `ngRoute` for Routing**:

   AngularJS provides the `ngRoute` module for routing. Install it using npm or include it from a CDN:

```html
<script src="https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.8.2/angular-
route.min.js"></script>
```

5. **Configuring Routes**:

Configure routes in your `app.js` file using the `$routeProvider` service:

```
.config(function($routeProvider) {
    $routeProvider
        .when('/', {
            templateUrl: 'views/home.html',
            controller: 'HomeController'
        })
        .when('/about', {
            templateUrl: 'views/about.html',
            controller: 'AboutController'
        })
        .otherwise({
            redirectTo: '/'
        });
});
```

6. **Creating Templates for Views**:

   Create `home.html` and `about.html` templates in the `views/` directory:

```
<!-- home.html -->
<div>
     <h1>{{ message }}</h1>
</div>
```

```
<!-- about.html -->
<div>
     <h1>{{ message }}</h1>
</div>
```

7. **Navigating Between Pages**:

   Use the `<a>` tag with `ng-href` to navigate between pages:

```
<a ng-href="#/">Home</a>
<a ng-href="#/about">About</a>
```

AngularJS Routing Capability

AngularJS provides routing capabilities through the `ngRoute` module. This module allows you to create single-page applications (SPAs) where different routes correspond to different views/templates and controllers. Here's an overview of how routing works in AngularJS using the `ngRoute` module:

1. **Setting Up `ngRoute`**:

   Include the `angular-route.js` script in your HTML. You can download it or use a CDN link.

   ```
   <script src="https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.8.2/angular-
   route.min.js"></script>
   ```

2. **Configuring Routes**:

   In your AngularJS module's configuration block, use the `$routeProvider` service to configure routes. Specify a template URL and a controller for each route.

   ```
   angular.module('myApp', ['ngRoute'])
       .config(function($routeProvider) {
           $routeProvider
               .when('/', {
                   templateUrl: 'views/home.html',
                   controller: 'HomeController'
               })
               .when('/about', {
                   templateUrl: 'views/about.html',
                   controller: 'AboutController'
               })
               .when('/contact', {
                   templateUrl: 'views/contact.html',
                   controller: 'ContactController'
               })
               .otherwise({
                   redirectTo: '/'
               });
       });
   ```

3. **Defining Views and Controllers**:

   Create HTML templates for each route and define controllers for them.

```
.controller('HomeController', function($scope) {
    $scope.message = 'Welcome to the Home Page!';
})

.controller('AboutController', function($scope) {
    $scope.message = 'This is the About Page.';
})

.controller('ContactController', function($scope) {
    $scope.message = 'Contact us for more information.';
});
```

4. **Using `ng-view` Directive**:

   In your main HTML file, use the `ng-view` directive to indicate where the content of each route should be inserted.

   ```
   <main ng-view></main>
   ```

5. **Navigating Between Routes**:

   Use the `ng-href` directive to navigate between different routes.

   ```
   <a ng-href="#/">Home</a>
   <a ng-href="#/about">About</a>
   <a ng-href="#/contact">Contact</a>
   ```

6. **Run Your Application**:

   Start a local server and open your application in a web browser. As you click the navigation links, the content will change without the need for full page reloads.

- AngularJS's routing capabilities, provided by the `ngRoute` module, enable you to build SPAs where different sections of your application are dynamically loaded based on the route.

Modules (Application, Controller)

In AngularJS, modules play a crucial role in organizing and structuring your application. They provide a way to encapsulate different parts of your code and define dependencies between various components. Two common types of modules in AngularJS are "Application Modules" and "Controller Modules." Let's take a closer look at each:

1. **Application Modules**:

An application module is the top-level module that defines your entire AngularJS application. It acts as a container for various components, such as controllers, services, directives, and more. The application module is initialized using the `angular.module` function.

Here's how you can define an application module:

```
// Creating an application module named 'myApp'
angular.module('myApp', []);
```

In this example, the application module named `'myApp'` is created, and the empty array `[]` is used to define the module's dependencies. You can include other modules as dependencies, such as `'ngRoute'` for routing or custom modules you've created.

2. **Controller Modules**:

   Controller modules are used to define controllers, which manage the behavior and logic of specific sections of your application's UI. Each controller module is attached to the application module as a part of its dependencies.

   Here's an example of defining a controller module and attaching it to the application module:

```
// Creating a controller module named 'myControllers'
angular.module('myControllers', [])
        .controller('HomeController', function($scope) {
        // Controller logic for the home page
        $scope.message = 'Welcome to the Home Page!';
    });
```

   In this example, a controller module named `'myControllers'` is created and a controller named `'HomeController'` is defined within it. The `$scope` object is used to manage data and interactions between the view and the controller.

   You can then attach the `'myControllers'` module as a dependency to the `'myApp'` application module:

```
// Attaching the 'myControllers' module to the 'myApp' application module
angular.module('myApp', ['myControllers']);
```

   Now, the `'myApp'` application module includes the `'myControllers'` module and can access the controllers defined within it.

Forms (Events, Data validation, ng-click)

In AngularJS, forms are an essential part of building dynamic and interactive web applications. AngularJS provides a set of directives and features to handle form-related tasks such as capturing user input, data validation, and handling events. Let's explore some key concepts related to forms in AngularJS:

1. **Form Elements and ng-model**:

   AngularJS provides the `ng-model` directive to bind form elements (like input fields, checkboxes, and radio buttons) to properties in your model. This enables two-way data binding, where changes in the UI are reflected in your model and vice versa.

   Example:

   ```
   <input type="text" ng-model="username">
   <input type="password" ng-model="password">
   ```

2. **Form Submission and ng-submit**:

   The `ng-submit` directive allows you to specify a function to be executed when the form is submitted. This function can handle form validation, data processing, and interactions.

   Example:

   ```
   <form ng-submit="submitForm()">
        <!-- Form Input fields here -->
        <button type="submit">Submit</button>
   </form>
   ```

   ```
   angular.module('myApp')
        .controller('FormController', function($scope) {
            $scope.submitForm = function() {
                // Handle form submission logic here
            };
        });
   ```

3. **Data Validation**:

   AngularJS provides built-in form validation features using CSS classes and directives like `ng-required`, `ng-minlength`, `ng-maxlength`, and more. You can also use CSS classes like `ng-valid`, `ng-invalid`, `ng-pristine`, and `ng-dirty` to style form elements based on their validation state.

   Example:

```html
<input type="email" ng-model="email" ng-required="true">
 <span ng-show="myForm.email.$error.email && myForm.email.$dirty">
     Invalid email address.
</span>
```

4. **ng-click and Form Interactions**:

   The `ng-click` directive allows you to bind a function to the click event of an element, such as a button. This is useful for triggering actions when a user interacts with a form.

   Example:

```html
<button ng-click="clearForm()">Clear Form</button>
```

```javascript
angular.module('myApp')
     .controller('FormController', function($scope) {
         $scope.clearForm = function() {
             // Reset form data and validation
             $scope.email = '';
             $scope.password = '';
             $scope.myForm.$setPristine();
         };
     });
```

AngularJS's form-related directives and features make it easier to create interactive forms with dynamic validation and smooth user interactions.