MREN

Bharadwaj Rachakonda

Mail: rbharadwaj022@gmail.com

Index

- ➔ MongoDB, create, update, find
- ➔ Replace, Update and Delete Documents
- ➔ **FindAndModify**
- ➔ Cursor projection count doc
- ➔ **Aggregation / Functions / Aggregates**
- ➔ Indexes
- ➔ Data Modelling / Transactions
- ➔ MongoDB with Nodejs/ MongoClient
- ➔ Mongoose – Widely used to run MongoDB in Node.js

- ➔ Node.js/Express.js – Asynchronous non-Blocking I/O
- ➔ npm
- ➔ **request object**
- ➔ **response object**
- ➔ **User defined modules**
- ➔ Core Modules
- ➔ fs
- ➔ Fs/promises
- ➔ http
- ➔ Path
- ➔ events
- ➔ url
- ➔ Third-party modules
- ➔ Nodemon

- ➔ Express
- ➔ Middleware in Express cors, json, body-parser….
- ➔ ejs
- ➔ Express-validator
- ➔ Bcrypt
- ➔ jwt

MERN

MongoDB – Data That is Accessed Together Must be Stored Together

Introduction:

MongoDB is a document based No-SQL Data Base Management System. You can store data with out the need of structure to your collection hence each document in you collection can be unique. But this doesn't mean MongoDB is a schema less DBMS. MongoDB is a schema flexible DBMS. A document is similar to a JSON – JavaScript Object Notation the document is then converted to a BSON – Binary Script Object Notation by MongoDB Server. A Maximum limit of BSON is 16MB.

Insert and Find Documents:

- While using find we usually use the following Comparison operators

- `$gt` (greater than)
- `$lt` (less than)
- `$lte` (less than or equal to)
- `$gte` (greater than or equal to)

- And following logical operators

- `$and`
- `$or`

- And **$elemMatch** operator.

Insert One / insert Many:

- `db.collection.insertOne()`
- Inserts a single document into a collection. If the collection does not exist, then the insertOne() method creates the collection.
- `db.collection.insertMany()`
- Inserts multiple documents into a collection. Given an array of documents, insertMany() inserts each document in the array into the collection.
- **Example:**

```
try {
  db.products.insertMany( [
    { item: "card", qty: 15 },
    { item: "envelope", qty: 20 },
    { item: "stamps" , qty: 30 }
  ] );
} catch (e) {
  print (e);
}
```

Find/ find One and few comparison and $elemMatch operators:

- `db.collection.find(query, projection, options)`
- Selects documents in a collection or view and returns a cursor to the selected documents.
- Example:

  db.collection.find({ qty: { $gt: 4 } })

- Example:

  ```
  db.bios.find(
    { _id: { $in: [ 5, ObjectId("507c35dd8fada716c89d0013") ] } }
  )
  ```

- Example:

  db.bios.find({ birth: { $gt: new Date('1950-01-01') } })

- Example: for starts with N

  ```
  db.bios.find(
    { "name.last": { $regex: /^N/ } }
  )
  ```

- Example:

  db.bios.find({ birth: { $gt: new Date('1940-01-01'), $lt: new Date('1960-01-01') } })

- Example:

  ```
  db.bios.find( {
    birth: { $gt: new Date('1920-01-01') },
    death: { $exists: false }
  } )
  ```

- Example:

  db.bios.find({ contribs: { $all: ["ALGOL", "Lisp"] } })

- Example: where len is 4

db.bios.find({ contribs: { $size: 4 } })

- Example: The following operation finds all documents in the bios collection and returns only the name field, contribs field and _id field

  db.bios.find({ }, { name: 1, contribs: 1 })

- The following operation queries the bios collection and returns all fields *except* the first field in the name embedded document and the birth field:

  ```
  db.bios.find(
    { contribs: 'OOP' },
    { 'name.first': 0, birth: 0 }
  )
  ```

- The following operation queries the bios collection and returns the last field in the name embedded document and the first two elements in the contribs array:

  ```
  db.bios.find(
    { },
    { _id: 0, 'name.last': 1, contribs: { $slice: 2 } } )
  ```

- Example: returns all values even if products is array or single string if array value must match at least one element in array.

  ```
  db.accounts.find({ products: "InvestmentFund"})
  ```

- Example: Use the $elemMatch operator to find all documents that contain the specified subdocument. $elemMatch has more to it google it.

  ```
  db.sales.find({
   items: {
    $elemMatch: { name: "laptop", price: { $gt: 800 }, quantity: { $gte: 1 } },
   },
  })
  ```

- Find one is almost similar to find.

Logical operators:

- , acts as a $and operator
- Example:

  ```
  db.routes.find({
   $or: [{ dst_airport: "SEA" }, { src_airport: "SEA" }],
  ```

})

Replace, Update and Delete Documents :

ReplaceOne:

- `db.collection.replaceOne(filter, replacement, options)`
- Replaces a single document within the collection based on the filter. replaceOne() replaces the first matching document in the collection that matches the filter, using the replacement document.
- Example:

```
db.books.replaceOne(
  {
    _id: ObjectId("6282afeb441a74a98dbbec4e"),
  },
  {
    title: "Data Science Fundamentals for Python and MongoDB",
    isbn: "1484235967",
    publishedDate: new Date("2018-5-10"),
    thumbnailUrl:

"https://m.mediaamazon.com/images/I/71opmUBc2wL._AC_UY218_.jpg",
    authors: ["David Paper"],
    categories: ["Data Science"],
  }
)
```

UpdateOne / updateMany / update:

- `db.collection.updateOne(filter, update, options)`
- db.collection.updateOne() finds the first document that matches the filter and applies the specified update modifications.

- **$set: The $set operator replaces the value of a field with the specified value. And can also add a new field to document.**

Syntax: { $set: { <field1>: <value1>, ... } }

- **$push: The $push operator appends a specified value to an array. If the array doesn't exists a new array is created and assigned.**
  Syntax: { $push: { <field1>: <value1>, ... } }

- **Upsert: Insert a document with the provided information if the matching documents doesn't exist.**

- Examples:

```
db.podcasts.updateOne(
  {
    _id: ObjectId("5e8f8f8f8f8f8f8f8f8f8f8"),
  },
  {
    $set: {
      subscribers: 98562,
    },
  }
)
```

```
db.podcasts.updateOne(
  { title: "The Developer Hub" },
  { $set: { topics: ["databases", "MongoDB"] } },
  { upsert: true }
)
```

```
db.podcasts.updateOne(
  { _id: ObjectId("5e8f8f8f8f8f8f8f8f8f8f8") },
  { $push: { hosts: "Nic Raboy" } }
)
```

FindAndModify:

- `db.collection.findAndModify(document)`
- Updates and returns a single document. By default, the returned document does not include the modifications made on the update. To return the document with the modifications made on the update, use the new option.
- Example:

```
db.podcasts.findAndModify({
  query: { _id: ObjectId("6261a92dfee1ff300dc80bf1") },
  update: { $inc: { subscribers: 1 } },
  new: true,
})
```

UpdateMany:

- `db.collection.updateMany(filter, update, options)`
- Updates all documents that match the specified filter for a collection.
- Example:

```
db.books.updateMany(
  { publishedDate: { $lt: new Date("2019-01-01") } },
  { $set: { status: "LEGACY" } }
)
```

DeleteOne / DeleteMany:

- To delete documents, use the `deleteOne()` or `deleteMany()` methods. Both methods accept a filter document and an options object.
- Examples:

```
db.podcasts.deleteOne({ _id: Objectid("6282c9862acb966e76bbf20a") })
```

```
db.podcasts.deleteMany({category: “crime”})
```

Modifying Query Results:

Cursor.sort() and Cursor.limit():

- `cursor.sort(sort)`
- Specifies the order in which the query returns matching documents. You must apply sort() to the cursor before retrieving any documents from the database.
- Use `cursor.sort()` to return query results in a specified order. Within the parentheses of `sort()`, include an object that specifies the field(s) to sort by and the order of the sort. Use 1 for ascending order, and -1 for descending order.
- Examples:

```
db.companies.find({ category_code: "music" }).sort({ name: 1 });
```

```
db.companies.find({ category_code: "music" }).sort({ name: 1, _id: 1
});
```

- `cursor.limit()`
- Use `cursor.limit()` to specify the maximum number of documents the cursor will return. Within the parentheses of `limit()`, specify the maximum number of documents to return.
- A limit() value of 0 (i.e. .limit(0)) is equivalent to setting no limit.
- Examples:

```
db.companies
  .find({ category_code: "music" })
  .sort({ number_of_employees: -1, _id: 1 })
  .limit(3);
```

Projection:

- To specify fields to include or exclude in the result set, add a projection document as the second parameter in the call to `db.collection.find()`.

- Examples:

```
db.collection.find( <query>, { <field> : 1 }) → Inclusion
```

```
db.collection.find(query, { <field> : 0, <field>: 0 }) → Exclusion
```

```
db.inspections.find(
  { sector: "Restaurant - 818" },
  { business_name: 1, result: 1 }
)
```

```
db.inspections.find(
  { result: { $in: ["Pass", "Warning"] } },
  { date: 0, "address.zip": 0 }
)
```

```
➔ By Default the _id is included so to exclude it use following
    syntax
```

```
db.inspections.find(
  { sector: "Restaurant - 818" },
  { business_name: 1, result: 1, _id: 0 }
)
```

CountDocuments:

- **`db.collection.countDocuments(query, options)`**
- Use `db.collection.countDocuments()` to count the number of documents that match a query. `countDocuments()` takes two parameters: a query document and an options document.

- Examples:

```
db.trips.countDocuments({})
```

```
db.trips.countDocuments({ tripduration: { $gt: 120 }, usertype:
 "Subscriber" })
```

Aggregation / Functions / Aggregates:

- **Aggregation: An analysis and summary of data.**
- **Stage: An aggregation operation performed on the data.**
- **Aggregation pipeline: A series of stages completed one at a time, in order.**

- **Stages:** A single operation on data.
  $match: filters for data that matches criteria. Usually the first stage.
  $group: groups document based on criteria.
  $sort: puts the document in a specified order.
  $limit: limits the number of documents that are passed on to the next aggregation stage. Usually comes after sort.
  $project: determines output shape. Usually the last stage
  $count: counts the total number of documents in the pipeline.
  $out: writes the documents that are returned by an aggregation pipeline into a collection. It must be the last stage. And creates a new collection if it does not already exist.

- **Syntax:**

```
db.collection.aggregate([
    {
        $stage1: {
            { expression1 },
            { expression2 }...
        },
        $stage2: {
            { expression1 }...
        }
    }
])
```

- **Syntax / $match / $group:**

```
{
  $match: {
     "field_name": "value"
  }
}
```

```
{
  $group:
    {
      _id: <expression>, // Group key
      <field>: { <accumulator> : <expression> }
    }
 }
```

- Example:

```
 db.zips.aggregate([
{
  $match: {
     state: "CA"
   }
},
{
  $group: {
     _id: "$city",
     totalZips: { $count : { } }
   }
}
])
```

- **Syntax / $sort / $limit:**

```
 {
    $sort: {
        "field_name": 1
    }
}
```

```
{
  $limit: 5
}
```

- Example:

```
db.zips.aggregate([
{
  $sort: {
    pop: -1
  }
},
{
  $limit:  5
}
])
```

- Examples / Syntax / $project / $set / $count:

```
{
    $project: {
        state:1,
        zip:1,
        population:"$pop",
        _id:0
    }
}
```

```
{
   $set: {
         place: {
               $concat:["$city",",","$state"]
         },
         pop:10000
      }
   }
```

```
{
   $count: "total_zips"
}
```

- Example / Syntax / $out:

```
db.getSiblingDB("test").books.aggregate( [
    { $group : { _id : "$author", books: { $push: "$title" } }
},
    { $out : { db: "reporting", coll: "authors" } }
] )
```

Indexes

CreateIndex / getIndexes / single field indexes:

- `db.collection.createIndex(keys, options, commitQuorum)`
- It is similar to primary key in SQL but there can be more than one in mongo db. You can apply other constraints such as unique and … on the index.
- `db.collection.getIndexes()`
- Examples:

```
db.customers.createIndex({
  birthdate: 1
})
```

```
db.customers.createIndex({
  email: 1
},
{
  unique:true
})
```

```
db.customers.getIndexes()
```

```
db.customers.explain().find({
  birthdate: {
    $gt:ISODate("1995-08-01")
    }
  })
```

```
db.customers.explain().find({
  birthdate: {
    $gt:ISODate("1995-08-01")
    }
  }).sort({
    email:1
    })
```

Multikey Indexes:

- Indexes created on an array / subdocuments / subarrays field are called as a Multikey indexes.
- Examples: → Accounts in below example is an array of strings

```
db.customers.createIndex({
  accounts: 1
})
```

```
db.customers.explain().find({
  accounts: 627788
  })
```

```
db.customers.getIndexes()
```

Compound Indexes:

- It is an index on multiple fields. Like grouping values and keeping a key constraint on them. A compound index can only have one array if there is one in the group. It can be a multikey index if it includes an array.
- Examples:

```
db.customers.createIndex({
  active:1,
  birthdate:-1,
  name:1
})
```

```
db.customers.find({
  birthdate: {
    $gte:ISODate("1977-01-01")
    },
    active:true
    }).sort({
      birthdate:-1,
      name:1
      })
```

```
db.customers.getIndexes()
```

Deleting Indexes / hideIndex / dropIndex:

- `db.collection.hideIndex()`
- Hide index before deleting it.
- `db.collection.dropIndex(index)`
- This is used to delete an index. To delete multiple indexes use **dropIndexes**([‘Index1’, ‘Index2’, ……]).
- Examples:

```
db.customers.dropIndex(
  'active_1_birthdate_-1_name_1'
)
```

```
db.customers.dropIndex({
  active:1,
  birthdate:-1,
  name:1
})
```

```
db.customers.dropIndexes()
```

```
db.collection.dropIndexes([
  'index1name', 'index2name', 'index3name'
  ])
```

Data Modelling Intro:

➔ Before this topic there is another topic called Atlas Search, I skipped it because I am currently interested in only learning CLI commands.

Data Relationships:

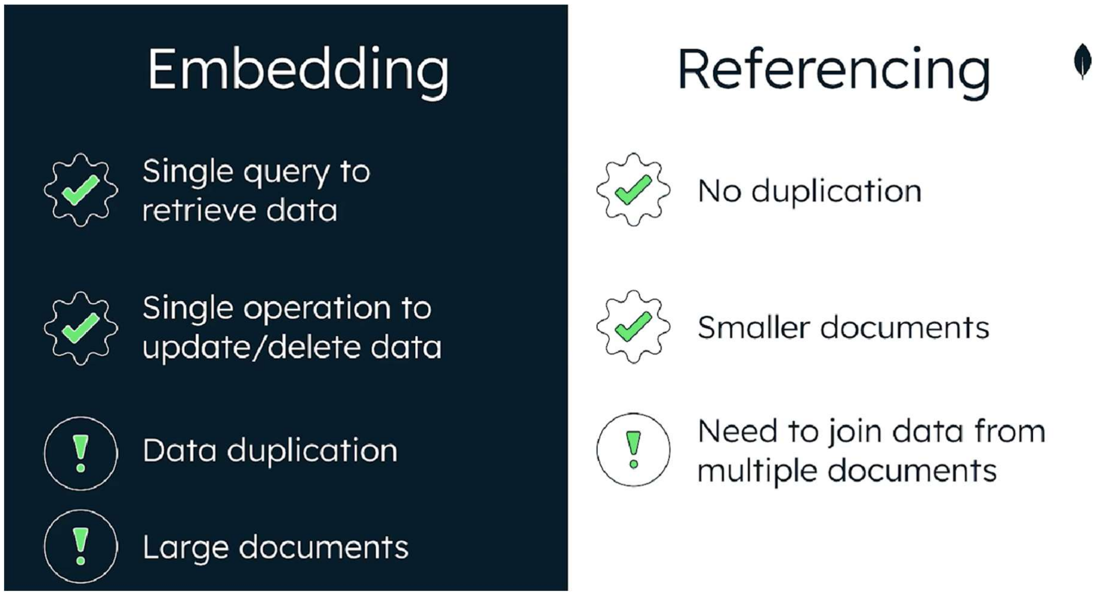
- There are three types of Relationship Types.

➔ **One-to-one:**
A relationship where a data entity in one set is connected to exactly one data entity

in another set. Example: A Single movie has only one Director

- ➔ **One-to-many:**
  A relationship where a data entity in one set is connected to any number of data entities in another set. Example: A move has many cast members i.e. actors
- ➔ **Many-to-many:**
  A relationship where any number of data entities in one set are connected to any number of data entities in another set.

- There are two main ways to model these relationships.
  - ➔ **Embedding:**
    We take related data and insert it into our document.
  - ➔ **Referencing:**
    We refer to documents in other collections in our document. Using references is sometimes called linking or data normalization.
- To understand when to use embedding and referencing use this



Embedding	Referencing
✓ Single query to retrieve data	✓ No duplication
✓ Single operation to update/delete data	✓ Smaller documents
! Data duplication	! Need to join data from multiple documents
! Large documents	

Transactions:

ACID Transactions:

- Transactions are used to solve few problems consider the following scenario.
  Two friends are out for a dinner and they decided to split the bill one of the friends pays the complete bill and other sends his contribution to the friend who paid the bill so during this transaction the amount must be deducted from one account and credited to another account. But what if the amount gets deducted but not credited. This issue is solved by the help of transactions.
- ACID transactions are a group of database operations that will be completed as a unit or not at all. It stands for
  - ➔ Atomicity – All operations either succeed or fail together.
  - ➔ Consistency – All changes made by operations are consistent with database constraints.

  - ➔ Isolation – Multiple transactions can happen at the same time without affecting the outcome of the other transaction.
  - ➔ Durability – All changes that are made by operations in a transaction will persist, no matter what.
- These are used in scenarios that involve the transfer of value from one record to another.

- Single document operations are already atomic in MongoDB. Multi-document operations are not atomic and requires few extra steps to become an ACID transactions.

- For this we use a Session – used to group database operations that are related to each other and should be run together. A transaction has a maximum runtime of less than one minute after the first write.

- **`Mongo.startSession(<options>)`**

- Example:

  ➔ To start a Transaction and commit

```
const session = db.getMongo().startSession()
session.startTransaction()
const account = session.getDatabase('< add database name
here>').getCollection('<add collection name here>')
//Add database operations like .updateOne() here
session.commitTransaction()
```

  ➔ To Abort a Transaction

```
const session = db.getMongo().startSession()
session.startTransaction()
const account = session.getDatabase('< add database name
here>').getCollection('<add collection name here>')
//Add database operations like .updateOne() here
session.abortTransaction()
```

That's It MongoDB is completed for now There is still more that you can learn from the documentation.

MongoDB Practice and few new things – GFG

CRUD Operations with MongoDB:

- Connecting with Nodejs

```
// Connect to MongoDB Atlas
const { MongoClient } = require('mongodb');
const uri = 'mongodb+srv://<username>:<password>@<cluster-
url>/<dbname>?retryWrites=true&w=majority';
const client = new MongoClient(uri);
async function main() {
try {
await client.connect();
const database = client.db('sample');
const collection = database.collection('users');
// Insert a new document
const result = await collection.insertOne({ name: 'John
Doe', age: 30 });
// Find a document
const document = await collection.findOne({ name: 'John Doe' });
console.log(document);
// Update a document
await collection.updateOne({ name: 'John Doe' }, { $set: { age: 31 }
});
// Delete a document
await collection.deleteOne({ name: 'John Doe' });
} finally {
await client.close();
}
}
main().catch(console.error);
```

- **Creating a Collection. In MongoDB not Nodejs.**

```
// Creating a new database and collection
use my_database
db.createCollection("my_collection")
```

- **Finding Documents:**

```
const { MongoClient } = require('mongodb');
// MongoDB connection string
const uri = 'mongodb://localhost:27017/my_database';
// Connect to MongoDB
const client = new MongoClient(uri,
{ useNewUrlParser: true, useUnifiedTopology: true });
async function findDocuments() {
try {
await client.connect();
console.log('Connected to MongoDB');
// Access the database and collection
const database = client.db('my_database');
const collection = database.collection('my_collection');
// Find documents matching a query
const query = { age: { $gt: 30 } }; // Find documents where age is
greater than 30
const cursor = collection.find(query);
// Iterate over the cursor to access documents
await cursor.forEach(document => {
console.log('Found document:', document);
});
} catch (error) {
console.error('Error finding documents:', error);
} finally {
await client.close();
console.log('Disconnected from MongoDB');
}
}
findDocuments();
```

- **Using ForEach**:

```
// MongoDB query using comparison operators
const query = { age: { $gt: 30 } }; // Find documents where age is
greater than 30
const cursor = collection.find(query);
// Iterate over the cursor to access documents
```

```
await cursor.forEach(document => {
console.log('Found document:', document);
});
```

MongoDB with Nodejs:

Note: Explanation is not provided most of the time cause all the methods are already similar to what you have learnt in MongoDB the only new this is how to connect using MongoClient. Also, you would use mongoose most of the time rather than MongoClient.

Connecting:

- An Official MongoDB driver establishes secure connections to a MongoDB cluster and executes database operations on behalf of client applications.

```
npm install mongodb  // install the driver
```

- **Connecting:**

```
const { MongoClient } = require('mongodb');
// MongoDB connection URI
const uri = 'mongodb://localhost:27017/mydatabase';
// Create a new MongoClient instance
const client = new MongoClient(uri,
{ useNewUrlParser: true, useUnifiedTopology: true });
// Connect to MongoDB
async function connectToMongoDB() {
try {
await client.connect();
console.log('Connected to MongoDB successfully!');
} catch (error) {
```

```
console.error('Error connecting to MongoDB:', error);
}
}
// Call the connectToMongoDB function to establish a connection
connectToMongoDB();
```

- **Inserting:**

```
const db = client.db('mydatabase');
const collection = db.collection('mycollection');
// Insert a document into the collection
await collection.insertOne({ name: 'John', age: 30 });
```

- **Querying Documents:**

```
// Find documents that match a query
const result = await collection.find({ age: { $gt: 25 } }).toArray();
console.log(result);
```

- **Updating Documents:**

```
// Update documents that match a filter
await collection.updateOne({ name: 'John' }, { $set: { age: 35 } });
```

- **Deleting Documents**:

```
// Delete documents that match a filter
await collection.deleteOne({ name: 'John' });
```

- Examples:
  → Insert

```
const { MongoClient } = require('mongodb');
async function insertDocument() {
const uri = 'mongodb://localhost:27017/mydatabase';
const client = new MongoClient(uri,
{ useNewUrlParser: true, useUnifiedTopology: true });

try {
await client.connect();
const db = client.db();
const collection = db.collection('mycollection');
const document = { name: 'John', age: 30 };
const result = await collection.insertOne(document);
console.log('Document inserted successfully:', result.insertedId);
} catch (error) {
console.error('Error inserting document:', error);
} finally {
await client.close();
}
}
insertDocument();
```

→ Find

```
async function findDocuments() {
try {
```

```
await client.connect();
const db = client.db();
const collection = db.collection('mycollection');

const query = { age: { $gt: 25 } };
const result = await collection.find(query).toArray();
console.log('Documents found:', result);
} catch (error) {
console.error('Error finding documents:', error);
} finally {
await client.close();
}
}
findDocuments();
```

➔ Update

```
async function updateDocuments() {
try {
await client.connect();
const db = client.db();
const collection = db.collection('mycollection');
const filter = { name: 'John' };
const update = { $set: { age: 35 } };
const result = await collection.updateOne(filter, update);
console.log('Documents updated:', result.modifiedCount);
} catch (error) {
console.error('Error updating documents:', error);
} finally {
await client.close();
}
}
updateDocuments();
```

→ Delete

```
async function deleteDocuments() {
try {
await client.connect();
const db = client.db();
const collection = db.collection('mycollection');
const filter = { name: 'John' };
const result = await collection.deleteOne(filter);
console.log('Documents deleted:', result.deletedCount);
} catch (error) {
console.error('Error deleting documents:', error);
} finally {
await client.close();
}
}
deleteDocuments();
```

➔ Transaction

```
async function createTransaction() {
const session = client.startSession();
try {
await session.startTransaction();
const db = client.db();
const collection = db.collection('mycollection');
// Perform multiple operations within the transaction
await collection.insertOne({ name: 'Alice' }, { session });
await collection.updateOne({ name: 'Bob' }, { $set: { age: 40 } }, {
session });
// Commit the transaction
await session.commitTransaction();
} catch (error) {
console.error('Error creating transaction:', error);
await session.abortTransaction();
} finally {
session.endSession();
}
}
createTransaction();
```

- **Arrays / Update Arrays / Arrays Update:** arrays can be updated in MongoDB using **$push, $pull, $pop.**

- **Aggregation:**

```
const { MongoClient } = require('mongodb');
async function performAggregation() {
const uri = 'mongodb://localhost:27017/mydatabase';
const client = new MongoClient(uri,
{ useNewUrlParser: true, useUnifiedTopology: true });
try {
await client.connect();
const db = client.db();
const collection = db.collection('mycollection');

const pipeline = [
{ $match: { status: 'active' } },
{ $group: { _id: '$category', total: { $sum: '$quantity' } } },
{ $sort: { total: -1 } }
];
const result = await collection.aggregate(pipeline).toArray();
console.log('Aggregation result:', result);
} catch (error) {
console.error('Error performing aggregation:', error);
} finally {
await client.close();
}
}
performAggregation();
```

```
const pipeline = [
{ $match: { status: 'active' } },
{ $group: { _id: '$category', total: { $sum: '$quantity' } } },
{ $sort: { total: -1 } }
];
```

```
const pipeline = [
{ $match: { status: 'active' } },
{ $group: { _id: '$category', total: { $sum: '$quantity' } } }
];
```

```
const pipeline = [
{ $sort: { total: -1 } },
{ $project: { _id: 0, category: '$_id', total: 1 } }
];
```

Mongoose – Widely used to run MongoDB in Node.js:

Basics / Connection / Installation:

- **Installing Mongoose:**

```
npm install mongoose –save
```

- **Connecting to a database:**

```
// getting-started.js
```

```
const mongoose = require('mongoose');

main().catch(err => console.log(err));

async function main() {
  await mongoose.connect('mongodb://127.0.0.1:27017/test');
  // use `await
mongoose.connect('mongodb://user:password@127.0.0.1:27017/test');` if
your database has auth enabled
}
```

Schemas:

- **Defining your schema**:
  Everything in Mongoose starts with a Schema. Each schema maps to a MongoDB collection and defines the shape of the documents within that collection.

```
import mongoose from 'mongoose';
const { Schema } = mongoose;
const blogSchema = new Schema({
  title: String, // String is shorthand for {type: String}
  author: String,
  body: String,
  comments: [{ body: String, date: Date }],
  date: { type: Date, default: Date.now },
  hidden: Boolean,
  meta: {
    votes: Number,
    favs: Number
  }
});
```

The permitted SchemaTypes are: String Number Date Buffer Boolean Mixed ObjectId Array Decimal128 Map UUID

Schemas not only define the structure of your document and casting of properties, they also define document instance methods, static Model methods, compound indexes, and document lifecycle hooks called middleware.

- **Creating a model:**
  To use our schema definition, we need to convert our blogSchema into a Model we can work with. To do so, we pass it into mongoose.model(modelName, schema):

```
const Blog = mongoose.model('Blog', blogSchema);
// ready to go!
```

  Models take your schema and apply it to each document in its collection. Models are responsible for all document interactions like creating, reading, updating, and deleting (CRUD).

- **Inserting data:**

  ➔ **Method – 1 / save() :**

```
import mongoose from 'mongoose';
import Blog from './model/Blog';

mongoose.connect("mongodb+srv://mongo:mongo@cluster0.eyhty.mongodb.net/myFirstDatabase?retryWrites=true&w=majority")

// Create a new blog post object
const article = new Blog({
  title: 'Awesome Post!',
  slug: 'awesome-post',
  published: true,
  content: 'This is the best post ever',
  tags: ['featured', 'announcement'],
});

// Insert the article in our MongoDB database
await article.save();
```

  use the save() method to insert it into our MongoDB database.

  ➔ **Method – 2 / create :**

  This method is much better! Not only can we insert our document, but we also get returned the document along with its _id when we console log it

```
// Create a new blog post and insert into database
const article = await Blog.create({
  title: 'Awesome Post!',
  slug: 'awesome-post',
  published: true,
  content: 'This is the best post ever',
  tags: ['featured', 'announcement'],
});

console.log(article);
```

- **Update data:**

  We can directly edit the local object, and then use the save() method to write the update back to the database. I don't think it can get much easier than that!

```
article.title = "The Most Awesomest Post!!";
await article.save();
console.log(article);
```

- **Finding data:**

  We'll use a special Mongoose method, findById(), to get our document by its ObjectId.

```
const article = await Blog.findById("62472b6ce09e8b77266d6b1b").exec();
console.log(article);
```

- **Projecting document fields:**

```
const article = await Blog.findById("62472b6ce09e8b77266d6b1b", "title
slug content").exec();
console.log(article);
```

  The second parameter can be of type Object|String|Array<String> to specify which fields we would like to project. In this case, we used a String.

- **Deleting data:**

  Just like in the standard MongoDB Node.js driver, we have the deleteOne() and deleteMany() methods.

```
const blog = await Blog.deleteOne({ author: "Jesse Hall" })
console.log(blog)

const blog = await Blog.deleteMany({ author: "Jesse Hall" })
console.log(blog)
```

- **Validation:**

  Validators only run on the create or save methods.

```
const blogSchema = new Schema({
  title:  {
    type: String,
    required: true,
  },
  slug:  {
    type: String,
    required: true,
    lowercase: true,
  },
  published: {
    type: Boolean,
    default: false,
  },
  author: {
    type: String,
    required: true,
  },
  content: String,
  tags: [String],
  createdAt: {
    type: Date,
    default: () => Date.now(),
    immutable: true,
  },
  updatedAt: Date,
  comments: [{
    user: String,
    content: String,
    votes: Number
  }]
});
```

- **Other useful methods / exists / where / select:**

  The **exists**() method returns either null or the ObjectId of a document that matches the provided query.

```
 const blog = await Blog.exists({ author: "Jesse Hall" })

console.log(blog)
```

Mongoose also has its own style of querying data. The **where**() method allows us to chain and build queries.

```
// Instead of using a standard find method
const blogFind = await Blog.findOne({ author: "Jesse Hall" });

// Use the equivalent where() method
const blogWhere = await Blog.where("author").equals("Jesse Hall");
console.log(blogWhere)
```

To include projection when using the where() method, chain the **select**() method after your query.

```
const blog = await Blog.where("author").equals("Jesse
Hall").select("title author")
console.log(blog)
```

- Multiple Schemas:

It's important to understand your options when modelling data. If you're coming from a relational database background, you'll be used to having separate tables for all of your related data. Generally, in MongoDB, data that is accessed together should be stored together. You should plan this out ahead of time if possible. Nest data within the same schema when it makes sense. If you have the need for separate schemas, Mongoose makes it a breeze. Let's create another schema so that we can see how multiple schemas can be used together. We'll create a new file, User.js, in the model folder.

```
import mongoose from 'mongoose';
const {Schema, model} = mongoose;

const userSchema = new Schema({
  name: {
    type: String,
    required: true,
  },
  email: {
    type: String,
    minLength: 10,
    required: true,
    lowercase: true
  },
});
```

```
const User = model('User', userSchema);
export default User;
```

Now we'll reference this new user model in our blog schema for the author and comments.user.

```
import mongoose from 'mongoose';
const { Schema, SchemaTypes, model } = mongoose;

const blogSchema = new Schema({
  ...,
  author: {
    type: SchemaTypes.ObjectId,
    ref: 'User',
    required: true,
  },
  ...,
  comments: [{
    user: {
      type: SchemaTypes.ObjectId,
      ref: 'User',
      required: true,
    },
    content: String,
    votes: Number
  }];
});
...
```

You'll now see only the user _id in the author field. So, how do we get all of the info for the author along with the article?
We can use the populate() Mongoose method.

```
const article = await Blog.findOne({ title: "Awesome Post!"
}).populate("author");
console.log(article);
```

- Middleware:

  In Mongoose, middleware are functions that run before and/or during the execution of asynchronous functions at the schema level.
  Here's an example. Let's update the updated date every time an article is saved or updated.
  We'll add this to our Blog.js model.

```
blogSchema.pre('save', function(next) {
  this.updated = Date.now(); // update the date every time a blog post is
saved
  next();
});
```

Besides pre(), there is also a post() mongoose middleware function.

That's it I will update from here on If there are any changes or new things that I discovered in MongoDB or Mongoose. You can see the documentation here and here is the quick start.

Node.js/Express.js – Asynchronous non-Blocking I/O

Introduction:

Before I continue if you want to learn Postman or brush up your skills review **here**

Node.js is an open-source and platform-independent (cross-platform) JavaScript runtime environment. Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. A Node.js app runs in a single process, without creating a new thread for every request. Node.js provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking and generally, libraries in Node.js are written using non-blocking paradigms, making blocking behaviour the exception rather than the norm.

Installation: You can Install Node.js from https://nodejs.org/download/

Requirements:
Learn these before starting with Node.js i.e. basics of JavaScript

- Lexical Structure
- Expressions
- Data Types
- Classes
- Variables
- Functions
- **this** operator
- Arrow Functions
- Loops
- Scopes
- Arrays
- Template Literals

- Strict Mode
- ECMAScript 2015 (ES6) and beyond
- Asynchronous programming and callbacks
- Timers
- Promises
- Async and Await
- Closures
- The Event Loop

npm:

npm is a package manager for Node.js it is a acronym for Node Package Manager. Here are few npm commands to start with Node.js.

```
npm install
```

If a project has a **package.json** file, by running above command it will install everything the project needs, in the **node_modules** folder, creating it if it's not existing already.

```
npm install <package-name>
```

You can also install a specific package by running above command. Furthermore, since npm 5, this command adds **<package-name>** to the **package.json** file *dependencies*. Before version 5, you needed to add the flag **--save**.

Often, you'll see more flags added to this command:

- **--save-dev** installs and adds the entry to the **package.json** file *devDependencies*
- **--no-save** installs but does not add the entry to the **package.json** file *dependencies*
- **--save-optional** installs and adds the entry to the **package.json** file *optionalDependencies*
- **--no-optional** will prevent optional dependencies from being installed

Shorthand's of the flags can also be used:

- -S: **--save**
- -D: **--save-dev**
- -O: **--save-optional**

The difference between *devDependencies* and *dependencies* is that the former contains development tools, like a testing library, while the latter is bundled with the app in production.

```
npm update
```

Updating is also made easy, by running above command. **npm** will check all packages for a newer version that satisfies your versioning constraints.

```
npm update <package-name>
```

You can specify a single package to update as well using the above command.

```
npm install <package-name>@<version>
```

You can install a specific version of a package, by running above command

```
npm run <task-name>
```

The package.json file supports a format for specifying command line tasks that can be run by using above command.

You can simplify long commands that are difficult to write or remember in **scripts** of package.json file and run them with the help of key specified in the package.json file.

Note: Node.js uses common-js syntax which enables you to use module, exports, __filename, __modulename, require inside the file. But if you want to use the ECMA script syntax mention "type": "module" in the package.json file by this you won't be able to utilize the common-js methods that are mentioned above but you can import the modules using import statement and use await directly with in the file without wrapping the statements inside a function and to export use export default or export.

Note: The DOM manipulation and other browser objects such as document and window are not available in Node.js.

request object:
The below example shows all available methods in the request object.

Example:

```
const express = require('express');
const cookieParser = require('cookie-parser'); // Middleware for cookies
const app = express();
const PORT = 3000;

// Middleware for parsing JSON and URL-encoded bodies
app.use(express.json());
app.use(express.urlencoded({ extended: true }));

// Middleware for parsing cookies
app.use(cookieParser());

// Example: Logging HTTP request information
app.use((req, res, next) => {
    console.log(`HTTP Method: ${req.method}`);          // Logs the HTTP
method (e.g., GET, POST)
    console.log(`URL: ${req.url}`);                    // Logs the full URL
path with query string
    console.log(`HTTP Version: ${req.httpVersion}`);   // Logs the HTTP
version (e.g., 1.1 or 2.0)
    console.log('Headers:', req.headers);             // Logs request headers
as an object
    console.log('Content-Type:', req.headers['content-type']); // Logs the
Content-Type header (if available)
```

```
    next();
});

// Example: Handling `req.query`
app.get('/user', (req, res) => {
    console.log("Query Parameters:", req.query); // Access query parameters
    console.log("Name:", req.query.name);       // Example:
http://localhost:3000/user?name=John
    console.log("Age:", req.query.age);         // Example:
http://localhost:3000/user?age=25
    res.send('Query parameters logged.');
});

// Example: Handling `req.body`
app.post('/profile', (req, res) => {
    console.log("Request Body:", req.body); // Access the request body (parsed
as an object)
    res.send('Request body logged.');
});

// Example: Handling `req.params`
app.get('/student/profile/:start/:end', (req, res) => {
    console.log("Route Parameters:", req.params); // Access route parameters
    console.log("Starting Page:", req.params.start); // Example:
/student/profile/1/10
    console.log("Ending Page:", req.params.end);     // Example:
/student/profile/1/10
    res.send('Route parameters logged.');
});

// Example: `req.path`, `req.hostname`, and `req.ip`
app.get('/info', (req, res) => {
    console.log("Request Path:", req.path);      // Path of the request (e.g.,
/info)
    console.log("Host Name:", req.hostname);    // Hostname of the request
(e.g., localhost)
    console.log("IP Address:", req.ip);         // IP address of the client
    res.send('Path, hostname, and IP logged.');
});

// Example: `req.cookies`
app.get('/cookies', (req, res) => {
    console.log("Cookies:", req.cookies); // Logs cookies sent by the client
    res.cookie('example', 'Hello');      // Sets a cookie named 'example'
    res.send('Cookies logged.');
});

// Start the server
```

```
app.listen(PORT, (err) => {
    if (err) console.error(err);
    console.log(`Server listening on PORT ${PORT}`);
});
```

response object:
The commonly used response objects are as shown below**(to end a request use req.send/req.end this is a mandatory step).**

- **res.status(code):**
  Sets the HTTP status code for the response.

  Example:
  ```
  res.status(404).send('Not Found');
  ```

- **res.set(field, [value]):**
  Sets a response header. The `field` is the header name, and `value` is the header value.

  Example:

```
res.set('Content-Type', 'application/json');
```

- **res.cookie(name, value, [options]):**
  Sets a cookie with the specified name, value, and optional settings.

  Example:

```
res.cookie('user', 'JohnDoe', { maxAge: 900000, httpOnly: true });
```

- **res.redirect([status], url):**
  Redirects the client to a different URL. The status code is optional; if not provided, it defaults to 302 Found.

  Example:

```
res.redirect(301, 'https://new-url.com'); // Permanent redirect (301)
```

- **res.send([body|status], [body]):**
  Sends a response to the client. If a body is provided, it will be sent as the response body. If the first argument is a status code, it sets the status code and the second argument is the body.

  Example:

```
res.send('Hello, World!');
```

```
res.status(200).send({ message: 'Success' });
```

- **res.json([status|body], [body]):**
  Sends a JSON response. Similar to res.send(), but ensures that the response body is JSON.

  Example:

```
res.json({ success: true });
```

- **res.render(view, [locals], callback):**
  Renders a view template and sends the rendered HTML as the response, you will see this later in the ejs template.

  Example:

```
res.render('index', { title: 'Home' });
```

- **res.writeHeader(statusCode, [headers]):**
  This method is typically used to send the HTTP response headers and status code. However, in Express, this is usually done with `res.status(code)` and `res.set(field, value)`. You may use `res.writeHeader()` directly in lower-level HTTP handling.

  Example:

```
res.writeHeader(200, { 'Content-Type': 'text/html' });
```

- **res.writeHeaders([headers]):**
  Sets the headers for the response.

  Example:

```
res.writeHeaders({
    'Content-Type': 'text/html',
    'Cache-Control': 'no-cache'
  });
```

- **res.write(chunk, [encoding]):**
  Writes data to the response stream. It is generally used for streaming data to the client.

  Example:

```
res.write('Hello');
res.write(' World!');
```

- **res.end([data], [encoding]):**
  Ends the response. This is usually used after streaming data or when no further content will be sent.

  Example:

```
res.write('Hello, World!');
res.end();  // Ends the response stream
```

- **res.statusCode:**
  The `statusCode` property can be set directly to specify the HTTP status code for the response.

  Example:

```
res.statusCode = 404;
res.end('Not Found');
```

- **res.sendFile(path, [options], [callback]):**
  Sends a file as a response. It automatically sets the appropriate content type based on the file's extension.

  Example:

```
res.sendFile('/path/to/file.html');
```

Modules:

Modules in Node.js are generally categorized into three types.

a) Core modules
b) User defined modules
c) Third – party modules

- **User defined modules:**
  User defined modules are created by the developer these modules are simply .js files that export methods or variables using exports or module.exports these then can be utilized using the require method if you are using a common-js syntax else use **export function_name** and if function is to be exported on default use **export default function_name**. Then use import to use the module.

  Example:

  main/modules/addition.js

```
const add = (a, b) => {
  return a + b;
};

const subtract = (a, b) => {
  return a - b;
};

exports.add = add;

exports.subtract = subtract;
```

main/main.js

```
const { add, subtract } = require("./modules/addition");

console.log(add(1, 1));
```

main/modules/addition.js – can also be written as following

```
const add = (a, b) => {
  return a + b;
};

const subtract = (a, b) => {
  return a - b;
};

module.exports = { add, subtract };
```

- **Core modules:**
  Core modules are the prebuilt modules that are installed along with Node.js.
  Syntax: require(`module_name`)
  Core modules can directly be imported using the module name i.e. there is no need to mention the complete path of the module.
  - ➔ http
  - ➔ assert
  - ➔ fs
  - ➔ path
  - ➔ process
  - ➔ os
  - ➔ querystring
  - ➔ url
  - ➔ connect

- ➔ events

- **fs module:**
  fs module is used to manipulate files in Node.js, It contains many methods and the commonly used methods are mentioned below.

  writeFile - writing & creating a file:
  This method is asynchronous and we need to use await to make it synchronous, use it when the following line's doesn't depend on it.
  syntax: **fs.writeFile(file, data[, options], callback)**

Example:
```
const fs = require("fs");


fs.writeFile("hello.txt", "Hello, world!", (err) => {
  if (err) {
    console.error(err);
    return;
  }
  console.log("File written successfully");
});
```

writeFileSync - writing & creating a file:
This is a synchronous version of writeFile
syntax: **fs.writeFileSync(file, data[, options])**

Example:

```
const fs = require('fs');

fs.writeFileSync('hello.txt', 'Hello from Node.js!');
```

readFile – reading a file:
This is a asynchronous operation to read a file similar to writeFile
syntax: **fs.readFile(path[, options], callback)**

Example:

```
const fs = require("fs");

fs.write("hello.txt", "Hello, world!", (err) => {
  if (err) {
    console.error(err);
    return;
  }
  console.log("File written successfully");
  fs.readFile("hello.txt", "utf8", (err, data) => {
    if (err) {
      console.error(err);
      return;
    }
    console.log(data);
  });
});
```

The above outputted data is not in a readable format to convert it use **data.toString()**
This is file but it results to a callback hell

readFileSync – reading a file:
This is too similar to above two methods hence we will not have anything to explain here.

appendFile – appending a file:
when you write into a file if the file already exists the previous content is overwritten to avoid this, we use a method appendFile which is similar to writeFile but it appends the content at the end of the file, there is another method appendFileSync with similar functionality but it is synchronous.
syntax: **fs.appendFile(path, data[, options], callback)**

Example:

```
const fs = require("fs");

fs.appendFile("message.txt", "data to append", (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

- **fs/promises:**

This is a module in fs that helps to write synchronous code with the help of promises.

Example:

You can use **appendFile** with the similar syntax

```
const fs = require("fs/promises");

// you can await a promise but this is not a module this is commonjs

let a = fs.readFile("a.txt", "utf-8");
a.then((data) => {
  fs.writeFile("b.txt", data);
});
```

- **path module:**
  Note: console.log is not written for ease of writing.

  extname:
  The path.extname() method returns the extension of the path, from the last occurrence of the . (period) character to end of string in the last portion of the path.
  syntax: **path.extname(path)**

  Example:

```
path = require("path");
path.extname("index.html");
// Returns: '.html'
```

  dirname:
  The path.dirname() method returns the directory name of a path.
  syntax: **path.dirname(path)**

  Example:

```
path = require("path");

path.dirname("/foo/bar/baz/asdf/quux");
// Returns: '/foo/bar/baz/asdf'
```

  basename:
  The path.basename() method returns the last portion of a path
  syntax: **path.basename(path[, suffix])**

  Example:

```
path = require("path");

path.basename("/foo/bar/baz/asdf/quux.html");
// Returns: 'quux.html'
```

  join:
  The path.join() method joins all given path segments together using the platform-specific separator as a delimiter, then normalizes the resulting path.
  syntax: **path.join([...paths])**

  Example:

```
path = require("path");
```

```
console.log(path.join("/foo", "bar", "baz/asdf", "quux", ".."));
// Returns: '/foo/bar/baz/asdf'
```

- **http module / creating a http server:**
  HTTP module is used to create the http server in Node.js the below code is an example showing how to create the endpoints.

  Example:

```
const http = require("http");

// Create an HTTP server
const server = http.createServer((req, res) => {
  const { method, url } = req;

  // Set up a route handler
  if (url === "/api/resource") {
    switch (method) {
      case "GET":
        res.writeHead(200, { "Content-Type": "application/json" });
        res.end(JSON.stringify({ message: "This is a GET request" }));
        break;

      case "POST":
        let postData = "";
        req.on("data", (chunk) => {
          postData += chunk;
        });
        req.on("end", () => {
          res.writeHead(201, { "Content-Type": "application/json" });
          res.end(
            JSON.stringify({
              message: "Data received via POST",
              data: JSON.parse(postData),
            })
          );
        });
        break;

      case "PUT":
        let putData = "";
        req.on("data", (chunk) => {
          putData += chunk;
        });
        req.on("end", () => {
          res.writeHead(200, { "Content-Type": "application/json" });
          res.end(
            JSON.stringify({
              message: "Resource updated via PUT",
              data: JSON.parse(putData),
            })
          );
        });
        break;

      case "DELETE":
        res.writeHead(200, { "Content-Type": "application/json" });
        res.end(JSON.stringify({ message: "Resource deleted via DELETE" }));
        break;

      default:
        res.writeHead(405, { "Content-Type": "application/json" });
        res.end(JSON.stringify({ message: `Method ${method} not allowed` }));
        break;
    }
  } else {
    // Handle 404 for undefined routes
    res.writeHead(404, { "Content-Type": "application/json" });
    res.end(JSON.stringify({ message: "Route not found" }));
  }
});

// Start the server on port 3000
server.listen(3000, () => {
  console.log("Server running at http://localhost:3000/");
});
```

- **connect module:**
  Don’t learn this just learn express that is enough.

- **events module:**
  This module is used to create and trigger events an event is just like a method that executes when triggered.

  Example:

```
const EventEmitter = require('events');

// Create a new instance of EventEmitter
const myEmitter = new EventEmitter();

// Define a listener for the 'greet' event
myEmitter.on('greet', (name) => {
    console.log(`Hello, ${name}!`);
});

// Define another listener for a custom event
myEmitter.on('farewell', (name) => {
    console.log(`Goodbye, ${name}!`);
});

// Emit the 'greet' event
myEmitter.emit('greet', 'Alice');

// Emit the 'farewell' event
myEmitter.emit('farewell', 'Bob');

// Add an event listener that fires only once
myEmitter.once('intro', (name) => {
    console.log(`Welcome, ${name}! This message will only be displayed
once.`);
});

// Emit the 'intro' event multiple times
myEmitter.emit('intro', 'Charlie'); // Triggered
myEmitter.emit('intro', 'Charlie'); // Won't trigger

console.log('Program completed.');
```

- **url module:**
  The url module in Node.js provides utilities for URL parsing, resolution, and manipulation. It helps in working with URLs and extracting components such as the hostname, pathname, query parameters, etc.

  Example:

```
const url = require('url');

// Example URL string
const urlString =
'https://example.com:8080/path/page?name=John&age=30#section1';

// Parse the URL using the modern URL class
const parsedUrl = new URL(urlString);

console.log('Full URL:', parsedUrl.href); // Full URL
console.log('Protocol:', parsedUrl.protocol); // Protocol: https:
console.log('Host:', parsedUrl.host); // Host: example.com:8080
console.log('Hostname:', parsedUrl.hostname); // Hostname: example.com
console.log('Port:', parsedUrl.port); // Port: 8080
console.log('Pathname:', parsedUrl.pathname); // Pathname: /path/page
console.log('Search:', parsedUrl.search); // Search: ?name=John&age=30
console.log('Hash:', parsedUrl.hash); // Hash: #section1
console.log('Query Params:', parsedUrl.searchParams); // Query Params:
URLSearchParams { 'name' => 'John', 'age' => '30' }

// Access individual query parameters
console.log('Name:', parsedUrl.searchParams.get('name')); // Name: John
console.log('Age:', parsedUrl.searchParams.get('age')); // Age: 30

// Adding and deleting query parameters
parsedUrl.searchParams.append('newParam', 'newValue');
console.log('Updated Query Params:', parsedUrl.searchParams.toString()); //
Updated Query Params: name=John&age=30&newParam=newValue
parsedUrl.searchParams.delete('age');
console.log('After Deleting "age":', parsedUrl.searchParams.toString()); //
After Deleting "age": name=John&newParam=newValue

// Convert parsed URL back to string
console.log('Formatted URL:', parsedUrl.toString());
```

- Third-Party Modules:
  In Node.js third party modules are the modules that are installed externally using the `npm install <package_name>` command

  few most commonly used third party modules are

  ➔ Express
  ➔ Jsonwebtoken
  ➔ express-validator
  ➔ bcryptjs
  ➔ mongoose (this one is already covered)
  ➔ nodemon
  ➔ concurrently

  If possible, I will cover the easiest once first / if it depends on others, I will cover the important once.

- **nodemon:**
  nodemon is a tool that helps develop Node.js based applications by automatically restarting the node application when file changes in the directory are detected.
  Installation: ***npm install -g nodemon***
  Syntax: `nodemon myapp.js`
  nodemon stops running on the result of a error or a keyboard interruption.

- **express:**

express is a module developed specifically to create web application the original http module has few limitations such as the security must be implemented from initial state of application, get and post requests must be written by the user though custom code, to get out of this burden we use express.
Installation: **npm install –save express**

Example: Hello World

```
const express = require('express')
const app = express()
const port = 3000

app.get('/', (req, res) => {
  res.send('Hello World!')
})

app.listen(port, () => {
  console.log(`Example app listening on port ${port}`)
})
```

Serving static files in Express:
To serve static files such as images, CSS files, and JavaScript files, use the express.static built-in middleware function in Express, To use multiple static assets directories, call the express.static middleware function multiple times.
Syntax: **express.static(root, [options])**

For example, use the following code to serve images, CSS files, and JavaScript files in a directory named public:

```
app.use(express.static('public'))
```

To create a virtual path prefix (where the path does not actually exist in the file system) for files that are served by the express.static function, specify a mount path for the static directory, as shown below:

```
app.use('/static', express.static('public'))
```

.all, .post, .get, .put, .delete:
There is a special routing method, app.all()used to load middleware functions at a path for ***all*** HTTP request methods.

For more on routing visit **here**

Route parameters:
Route parameters are named URL segments that are used to capture the values specified at their position in the URL. The captured values are populated in the req.paramsobject, with the name of the route parameter specified in the path as their respective keys.

Example:

```
app.get('/users/:userId/books/:bookId', (req, res) => {
  res.send(req.params)
})
/*
```

Output:
Route path: /users/:userId/books/:bookId
Request URL: http://localhost:3000/users/34/books/8989
req.params: { "userId": "34", "bookId": "8989" }

Since the hyphen (-) and the dot (.) are literally interpreted, they can be used along with route parameters for useful purposes.
For more on routing visit **here**

express.Router:
Use the express.Routerclass to create modular, mountable route handlers. A Routerinstance is a complete middleware and routing system;

Example:

Create a router file named birds.jsin the app directory, with the following content:

```
const express = require('express')
const router = express.Router()

// middleware that is specific to this router
router.use((req, res, next) => {
  console.log('Time: ', Date.now())
  next()
})
// define the home page route
router.get('/', (req, res) => {
  res.send('Birds home page')
})
// define the about route
router.get('/about', (req, res) => {
  res.send('About birds')
})

module.exports = router
```

Then, load the router module in the app:

```
const birds = require('./birds')


// ...


app.use('/birds', birds)
```

The app will now be able to handle requests to /birdsand /birds/about

req.query:

it gets the query parameters req.params only gets the slug parameters.

Example:

```
const express = require('express');
const app = express();
const port = 3000;

app.get('/query', (req, res) => {
  // Print req.query
  console.log(req.query);
  res.send(req.query);
});

app.listen(port, () => {
  console.log(`Server is running on http://localhost:${port}`);
});
```

Output:

URL: http://localhost:3000/query?name=John&age=30

Console Output: { name: 'John', age: '30' }

req.body:

It gets the data sent into the body of request, no example here I know how it works.

res.sendFile:

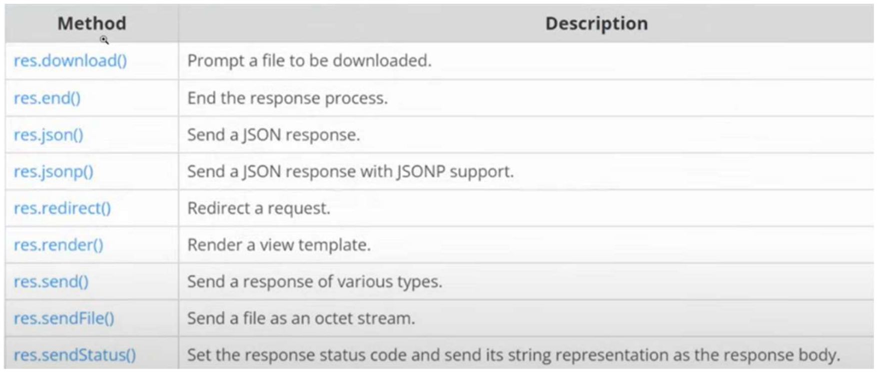
It is used to serve html files or send any file to user, syntax is already provided in the response object above. Here the {root: __dirname} is important else we need to specify the absolute path.

Example:

```
const express = require("express");
const app = express();

app.get("/download", (req, res) => {
  res.sendFile("templates/example.pdf", { root: __dirname });
});

const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server is running on http://localhost:${PORT}`);
});
```



Method	Description
res.download()	Prompt a file to be downloaded.
res.end()	End the response process.
res.json()	Send a JSON response.
res.jsonp()	Send a JSON response with JSONP support.
res.redirect()	Redirect a request.
res.render()	Render a view template.
res.send()	Send a response of various types.
res.sendFile()	Send a file as an octet stream.
res.sendStatus()	Set the response status code and send its string representation as the response body.

The above are few more commonly used methods.

Middleware in express:

When you use the app.use this is added as a middle ware if no path is provided this middle ware runs for every endpoint, if you want it for a specific endpoint, you can pass it with in the function (get, post, …), A middle ware can be implemented in three ways the following example show's all those 3 ways, 3rd party middle wares also exist which we will explore later. The middleware takes 3 parameters req, res, next use next() to pass req, res to next middle ware else if you want to end request with current middleware use res.end() which stops req from moving to next middleware.

```
const express = require("express");
const app = express();

/*
1. Global Middleware:
   Logs all incoming requests, regardless of the route.
*/
app.use((req, res, next) => {
  console.log(`[Global Middleware] ${req.method} ${req.url}`);
  next(); // Pass control to the next middleware or route handler
});

/*
2. Specific Route Middleware:
   Checks authentication for a protected route.
*/
const checkAuth = (req, res, next) => {
  console.log("[Specific Middleware] Authentication check.");
  const isAuthenticated = true; // Example condition
  if (isAuthenticated) {
    next(); // Pass control to the next middleware or route handler
  } else {
    res.status(403).send("Forbidden");
  }
};

// Apply specific middleware to a single route
app.get("/protected", checkAuth, (req, res) => {
  res.send("Welcome to the protected route!");
});

/*
3. Middleware for routes starting with `/api/`:
   Handles requests to all API-related endpoints.
*/
const apiMiddleware = (req, res, next) => {
  console.log(`[API Middleware] ${req.method} ${req.url}`);
  next(); // Pass control to the next middleware or route handler
};

// Apply API middleware to all `/api/` routes
app.use("/api", apiMiddleware);

// Define some API routes
app.get("/api/users", (req, res) => {
  res.send("List of users");
});

app.get("/api/products", (req, res) => {
  res.send("List of products");
});

// A public route with no middleware
app.get("/", (req, res) => {
  res.send("Public Home Page");
});

// Start the server
const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server is running on http://localhost:${PORT}`);
});
```

There are also other types of middleware's one is error handling middleware it takes another object in addition to these that is **err** which you can use for error handling.

Below are most commonly used built in middle ware (express.static is already seen)

express.json / express.urlencoded / bod-parser / cors :

Example with syntax:

```javascript
const express = require("express");
const bodyParser = require("body-parser");
const cors = require("cors");

const app = express();

/*
1. express.json():
   - Parses incoming JSON requests.
   - Useful for APIs that send data as JSON (e.g., `Content-Type:
application/json`).
*/
app.use(express.json());

/*
2. express.urlencoded():
   - Parses URL-encoded data (form submissions).
   - Useful when data is sent using `application/x-www-form-urlencoded`
format.
*/
app.use(express.urlencoded({ extended: true })); // `extended: true` allows
parsing of nested objects

/*
3. body-parser:
   - Used to handle raw data, JSON, or URL-encoded data.
   - While `express.json` and `express.urlencoded` handle JSON and form data
respectively, `body-parser` gives more flexibility for raw data types like
text.
   - It's modular and works well for legacy setups.
*/
app.use(bodyParser.text()); // Parses raw text sent in the body
app.use(bodyParser.json()); // Parses JSON (alternative to express.json)

/*
4. cors:
   - Allows Cross-Origin Resource Sharing.
   - Enables your API to handle requests from different domains.
*/
app.use(cors()); // Enables all CORS requests by default

// Routes
app.post("/json", (req, res) => {
  console.log("Received JSON:", req.body); // Parsed JSON data
  res.send("JSON received successfully!");
});
```

```
app.post("/form", (req, res) => {
  console.log("Received Form Data:", req.body); // Parsed form data
  res.send("Form data received successfully!");
});

app.post("/raw-text", (req, res) => {
  console.log("Received Raw Text:", req.body); // Parsed raw text
  res.send("Raw text received successfully!");
});

// Example CORS-enabled route
app.get("/cors-example", (req, res) => {
  res.json({ message: "CORS is enabled!" });
});

// Start the server
const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server is running on http://localhost:${PORT}`);
});
```

- **template ejs:**
  ejs is a template engine used just like jinja template in flask but its syntax is different, learn more about ejs from here.
  Installation: `npm install ejs`

Important: `app.set("view engine", "ejs");`

Example:
views/index.ejs → The file must be in views folder

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>EJS Syntax Example</title>
  </head>
  <body>
    <h1>EJS Syntax Demonstration</h1>

    <!-- 1. Output a simple variable -->
    <h2>Welcome, <%= username %>!</h2>

    <!-- 2. Output a variable with escaping -->
    <p>Your bio: <%= bio %></p>

    <!-- 3. Output raw HTML -->
    <p>Raw HTML content: <%- rawHtml %></p>

    <!-- 4. Conditional Statements -->
    <h3>Conditional Example</h3>
    <% if (isLoggedIn) { %>
    <p>You are logged in!</p>
    <% } else { %>
    <p>You are not logged in. Please log in.</p>
    <% } %>

    <!-- 5. Loops -->
    <h3>List of Items:</h3>
    <ul>
      <% items.forEach(item => { %>
      <li><%= item %></li>
      <% }); %>
    </ul>

    <!-- 6. Include Partial (Commented for demonstration) -->
    <% /* <%- include('partials/header') %> */ %>

    <!-- 7. Inline JavaScript -->
    <h3>Numbers:</h3>
    <ul>
      <% for (let i = 1; i <= 5; i++) { %>
      <li>Number: <%= i %></li>
      <% } %>
    </ul>

    <!-- 8. Commenting -->
    <% /* This is a server-side EJS comment that won't appear in the HTML */
%>

    <!-- 9. Default Values -->
    <h3>Default Value Example</h3>
    <p>Favorite Color: <%= favoriteColor || 'Not specified' %></p>

    <!-- 10. Escaping vs Non-Escaping -->
    <h3>Escaping Content</h3>
    <p>
      Escaped: <%= "
      <script>
        alert("Hello!");
      </script>
      " %>
    </p>
    <p>
      Unescaped: <%- "
      <script>
        alert("Hello!");
      </script>
      " %>
    </p>
  </body>
</html>
```

server.js

```
const express = require("express");
const app = express();

// Set EJS as the view engine
app.set("view engine", "ejs");

app.get("/", (req, res) => {
  const data = {
    username: "John Doe",
    bio: "I love coding & <b>open source</b>!",
    rawHtml: "<b>This is bold text rendered as raw HTML.</b>",
    isLoggedIn: true,
    items: ["Apple", "Banana", "Cherry"],
    favoriteColor: null, // Demonstrates default value
  };

  res.render("index", data);
});

const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server is running on http://localhost:${PORT}`);
});
```

Explanation of Syntax:

1. **`<%= variable %>`**:
   - Outputs the value of the variable, escaping HTML characters for security.
2. **`<%- variable %>`**:
   - Outputs the raw content of the variable without escaping HTML.
3. **`<% JavaScript code %>`**:
   - Executes JavaScript code, such as loops and conditionals, without displaying it in the output.
4. **Conditionals (`if`, `else`)**:
   - Control the rendering of sections based on conditions.
5. **Loops (`forEach`, `for`)**:
   - Used to iterate over arrays or numbers and dynamically render content.

6. **Include Partials**:
   - `<%- include('partials/filename') %>`: Includes another EJS file (e.g., a header or footer).
7. **Inline JavaScript**:
   - Use `<% %>` to execute JavaScript inside the template.
8. **Comments**:
   - `<% /* comment */ %>`: Server-side comments that do not appear in the final HTML.
9. **Default Values**:
   - The `||` operator allows you to set default values for variables.
10. **Escaping vs Non-Escaping**:
    - Demonstrates the difference between `<%= %>` (escaped) and `<%- %>` (raw, unescaped) syntax.

- **express-validator:**
  It allows you to combine them in many ways so that you can validate and sanitize your express requests, and offers tools to determine if the request is valid or not, which data was matched according to your validators, and so on, learn more on here.
  Installation: `npm install express-validator`

  .notEmpty():
  cannot be empty, in this example query is used(parameters passed in url (not as a slug)), the same example goes with the body and params.

  Example:

```
const express = require('express');
const { query } = require('express-validator');
const app = express();

app.use(express.json());
app.get('/hello', query('person').notEmpty(), (req, res) => {
  res.send(`Hello, ${req.query.person}!`);
});

app.listen(3000);
```

  Handling validation errors:
  For the validation errors the module provides a function for verifying the validation result with the validationResult function

  Example:

```
const express = require('express');
const { query, validationResult } = require('express-validator');
const app = express();

app.use(express.json());
app.get('/hello', query('person').notEmpty(), (req, res) => {
  const result = validationResult(req);
  if (result.isEmpty()) {
    return res.send(`Hello, ${req.query.person}!`);
  }

  res.send({ errors: result.array() });
});

app.listen(3000);
```

Now if you access http://localhost:3000/hello again, what you'll see is the following JSON content:

```
{
  "errors": [
    {
      "location": "query",
      "msg": "Invalid value",
      "path": "person",
      "type": "field"
    }
  ]
}
```

Sanitizing inputs:

While the user can no longer send empty person names, it can still inject HTML into your page! This is known as the Cross-Site Scripting vulnerability (XSS). Let's see how it works. Go to http://localhost:3000/hello?person=<b>John</b>, and you should see "Hello, **John**!". While this example is fine, an attacker could change the person query string to a <script> tag which loads its own JavaScript that could be harmful. In this scenario, one way to mitigate the issue with express-validator is to use a **sanitizer**, most specifically escape, which transforms special HTML characters with others that can be represented as text.

Example:

```
const express = require('express');
const { query, validationResult } = require('express-validator');
const app = express();

app.use(express.json());
app.get('/hello', query('person').notEmpty().escape(), (req, res) => {
  const result = validationResult(req);
  if (result.isEmpty()) {
    return res.send(`Hello, ${req.query.person}!`);
  }

  res.send({ errors: result.array() });
});
```

```
app.listen(3000);
```

Accessing validated data:
This application is pretty simple, but as you start growing it, it might become quite repetitive to type req.body.fieldName1, req.body.fieldName2, and so on. To help with this, you can use matchedData(), which automatically collects all data that express-validator has validated and/or sanitized:

Example:

```
const express = require('express');
const { query, matchedData, validationResult } = require('express-validator');
const app = express();

app.use(express.json());
app.get('/hello', query('person').notEmpty().escape(), (req, res) => {
  const result = validationResult(req);
  if (result.isEmpty()) {
    const data = matchedData(req);
    return res.send(`Hello, ${data.person}!`);
  }

  res.send({ errors: result.array() });
});

app.listen(3000);
```

What are validation chains?
Validation chains are created by functions such as body(), param(), query(), and so on. They have this name because they wrap the value of a field with validations (or sanitizations), and each of its methods returns itself. This pattern is usually called **method chaining**, hence why the name *validation chain*.
Example:

```
app.post(
    '/newsletter',
    // For the `email` field in `req.body`...
    body('email')
      // ...mark the field as optional
      .optional()
      // ...and when it's present, trim its value, then validate it as an
email address
      .trim()
      .isEmail(),
    maybeSubscribeToNewsletter,
  );
```

A validation chain has three kinds of methods: validators, sanitizers and modifiers.
Validators determine if the value of a request field is valid. This means checking if the field is in the format that you expect it to be.

Sanitizers transform the field value. They are useful to remove noise from the value, to cast the value to the right JavaScript type, and perhaps even to provide some basic line of defence against threats.

Modifiers define how validation chains behave when they are run. This might include adding conditions on when they should run, or even which error message a validator should have.

View all the possible methods here this is what you might be looking for.

Each of an array's items is **individually** validated/sanitized according to these rules.
For example, a validation chain body('ids').isNumber() would find two errors when req.body.ids is [5, '33', 'abc', 'def'].

Reusing validation chains:
Validation chains are **mutable**. This means that calling methods on one will cause the original chain object to be updated, just like any references to it. If you wish to reuse the same chain, it's a good idea to return them from functions:

Example:

```
const createEmailChain = () => body("email").isEmail();
app.post("/login", createEmailChain(), handleLoginRoute);
app.post(
  "/signup",
  createEmailChain().custom(checkEmailNotInUse),
  handleSignupRoute
);
```

Note: better to use square brackets than dot notation in objects

Whole-body selection:
Sometimes a request's body is not an object or an array, but you still want to select it for validation/sanitization. This can be done by omitting the field path, or by using an empty string, I never used this putting this for future use cases. Both yield the same result:
Example:

```
app.post(
  "/recover-password",
  // These are equivalent.
  body().isEmail(),
  body("").isEmail(),
  (req, res) => {
    // Handle request
  }
);
```

Wildcards:
Sometimes you will want to apply the same rules to all items of an array, or all keys of an object. That's what the *, also known as the wildcard, is for.

Example:
Let's imagine that the endpoint for updating a user's profile accepts their addresses and siblings:

```
{
  "addresses": {
    "home": { "number": 35 },
    "work": { "number": 501 }
  },
  "siblings": [{ "name": "Maria von Validator" }, { "name": "Checky McCheckFace" }]
}
```

In order to validate that the address numbers are all integers, and that the name of the siblings are set, you could have the following validation chains:

```
app.post(
  "/update-user",
  body("addresses.*.number").isInt(),
  body("siblings.*.name").notEmpty(),
  (req, res) => {
    // Handle request
  }
);
```

Globstars:
Globstars extend wildcards to an infinitely deep level. They can be used when you have an unknown level of nested fields, and want to validate/sanitize all of them the same way.

Example:
For example, imagine that your endpoint handles the update of a company's organizational chart. The structure is recursive, so it looks roughly like this:

```
{
  "name": "Team name",
  "teams": [{ "name": "Subteam name", "teams": [] }]
}
```

You can use a globstar (**) to target any field, no matter how deep it is in the request. The following example checks that all fields called name, including the one at the root of the req.body, are set:

```
app.put("/update-chart", body("**.name").notEmpty(), (req, res) => {
  // Handle request
});
```

Customizing express-validator:
Custom validators must return a truthy value to indicate that the field is valid, or falsy to indicate it's invalid. Custom validators can be asynchronous, in which case it can return a promise. The returned promise is awaited on, and it must resolve in order for the field to be valid. If it rejects, the field is deemed invalid.

Examples:

```
app.post(
  "/create-user",
  body("email").custom(async (value) => {
    const user = await UserCollection.findUserByEmail(value);
    if (user) {
      throw new Error("E-mail already in use");
    }
  }),
  (req, res) => {
    // Handle the request
  }
);
```

Custom sanitizers don't have many rules. Whatever the value that they return, is the value that the field will acquire.

Error Messages:
A validator-level message applies only when the field fails a specific validator. This can be done by using the .withMessage() method:
Example:

```
body("email").isEmail().withMessage("Not a valid e-mail address");
```

If a custom validator throws, the thrown value will be used as its error message.

A field-level message is set when you create the validation chain. It's used as a fallback message when a validator doesn't override its error message.
Example:

```
body("json_string", "Invalid json_string")
  // No message specified for isJSON, so use the default "Invalid json_string"
  .isJSON()
  .isLength({ max: 100 })
  // Overrides the default message when `isLength` fails
  .withMessage("Max length is 100 bytes");
```

Schema validation:
Schemas are an object-based way of defining validations or sanitizations on requests. They offer exactly the same functionality as regular validation chains - in fact, under the hood, express-validator deals all in validation chains!
Schemas are plain JavaScript objects that you pass to the checkSchema() function

Example:

```
checkSchema({
  username: {
    errorMessage: "Invalid username",
    isEmail: true,
  },
  password: {
    isLength: {
      options: { min: 8 },
      errorMessage: "Password should be at least 8 chars",
    },
  },
});
```

Everything works just the same.

Methods of validation:

.custom():
We already seen it in previous examples.

.exists():
Adds a validator to check if the field exists.
Syntax:

```
exists(options?: {
    values?: 'undefined' | 'null' | 'falsy',
  }): ValidationChain
```

.isArray():
Adds a validator to check that a value is an array, You can also check that the array's length is greater than or equal to options.min and/or that it's less than or equal to options.max..
Syntax:

```
isArray(options?: { min?: number; max?: number }): ValidationChain
```

.isObject():
Adds a validator to check that a value is an object.

.isString():
Adds a validator to check that a value is a string.

.notEmpty():
Adds a validator to check that a value is a string that's not empty. This is analogous to .not().isEmpty().

isBoolean():
checks if value is Boolean.

isDate():
checks if it is a date

isDecimal():
checks if it is a decimal value

isEmail():
checks if it is a email.

isFloat():
checks if it is a float

Syntax:
```
isFloat(options?: {

    min?: number;
    max?: number;
    lt?: number;
    gt?: number;
    locale?: AlphanumericLocale;
  }): ValidationChain
```

isIn():
checks if it is in certain values.

Syntax:
```
isIn(values: readonly any[]): ValidationChain
```

isInt():
checks if the value is integer.

Syntax:
```
isInt(options?: {

    min?: number;
    max?: number;
    lt?: number;
    gt?: number;
    allow_leading_zeroes?: boolean;
  }): ValidationChain
```

isJSON():
checks if value is json.

isJWT():
checks if it is a JWT Token.

isLength():
checks the length you can also give a fixed value

```
Syntax:
isLength(options: {

    min?: number;
    max?: number;
  }): ValidationChain
```

isLowercase() / isMobilePhone() / isNumeric() / isUppercase():
You can understand what these are

isStrongPassword():
Checks if it is a strong password or not based on given values
Syntax:

```
isStrongPassword(options?: {
    minLength?: number;
    minLowercase?: number;
    minUppercase?: number;
    minNumbers?: number;
    minSymbols?: number;
    returnScore?: boolean;
    pointsPerUnique?: number;
    pointsPerRepeat?: number;
    pointsForContainingLower?: number;
    pointsForContainingUpper?: number;
    pointsForContainingNumber?: number;
    pointsForContainingSymbol?: number;
  }): ValidationChain
```

.default():
give it a default value

```
Syntax:
default(defaultValue: any): ValidationChain
```

.toArray() / .toLowerCase() / .toUpperCase():
You know what they do and other methods are shown below.

escape() / unescape() / ltrim(chars?: string) / rtrim() / toBoolean() / toDate() / toFloat() / toInt() / trim() / .not() / .optional() / .withMessage()

.hide():

Hide the field's value in errors returned by validationResult(). If the value is confidential information (such as api key), you might want to call this method to prevent exposing it. If hiddenValue is set, it's set as the value in the errors for this field.

```
Syntax:
hide(hiddenValue?: string): ValidationChain
```

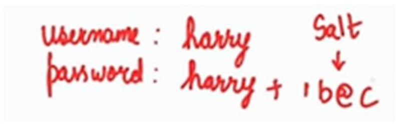
Example:

```
// Omits the value in the errors
query('api_key').custom(isValidKey).hide();

// Replaces the value in the errors with '*****'
query('api_key').custom(isValidKey).hide('*****');
```

this is more than enough you can check more in the documentation.

- **bcryptjs:**
  watch **here** for more clear understanding. Why use bcrypt.js when user sends the password and the hacker has a rainbow table (a table of common passwords mapped to their hashes) your application will be vulnerable, to avoid this we apply hashing, hashing (a one-way function) encrypts the data and the password can't be revered and can only be compared. Hence even if your database is near hands of a hacker since he can't get the password's he can't do anything. Up on this we add another layer called salt why add salt, if the hacker has a rainbow table application is still vulnerable to another layer of protection is added, here we add an additional value to the password which is not predictable, the salt is not stored in dB it is only at the backend. Bcryptjs provides everything in its package as shown in below
  Installation: ***npm install bcryptjs***
  example.



Example:
(on signup)

```
router.post("/createuser", validation(), async (req, res) => {
  const errors = validationResult(req);

  if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() });
  }

  try {
    const salt = await bcrypt.genSalt(10);

    const secPass = await bcrypt.hash(req.body.password, salt);

    const user = await User.create({
      name: req.body.name,
      password: secPass,
    });

    const data = {
      user: {
        id: user.id,
        name: user.name,
      },
    };

    const authToken = jwt.sign(data, JWT_SECRET);

    res.json({ authToken });
  } catch (error) {
    if (error.code === 11000) {
      // Duplicate key error
      return res.status(400).json({ errors: "User already exists" });
    }

    console.error(error);

    res.status(500).json({ errors: "Server error" });
  }
});
```

(on login)

```
router.post("/login", validation(), async (req, res) => {
  const errors = validationResult(req);

  if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() });
  }

  const { name, password } = req.body;

  try {
    let user = await User.findOne({ name });
    if (!user) {
      return res.status(400).json({ errors: "Login with right credentials" });
    }

    const passwordCompare = await bcrypt.compare(password, user.password);

    if (!passwordCompare) {
      return res.status(400).json({ errors: "Login with right credentials" });
    }
    const data = {
      user: {
        id: user.id,
        name: user.name,
      },
    };
    const authToken = jwt.sign(data, JWT_SECRET);
    res.json({ authToken });
  } catch (error) {
    console.log(error.message);
    res.status(500).json({ errors: "Some Error Occured" });
  }
});
```

- **JWT:**
  After authentication user data is passed to the browser which is later taken to retrieve further information but what if the data is changed by user, false data can't be detected and due to this the data of other users might get exposed, to protect from this we use JWT. With JWT change in data can easily be detected, JWT contains mainly 3 parts the HEADER, PAYLOAD, VERIFY SIGNATURE.
  HEADER includes the algorithm used for encryption, verify signature is the jwt secreat.
  You can learn more on the **ebook**
  Installation: ***npm install jsonwebtoken***
  Example:
  Creating token (on signup) JWT_SECRET can be any string.

```
router.post("/createuser", validation(), async (req, res) => {
  const errors = validationResult(req);

  if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() });
  }

  try {
    const salt = await bcrypt.genSalt(10);

    const secPass = await bcrypt.hash(req.body.password, salt);

    const user = await User.create({
      name: req.body.name,
      password: secPass,
    });

    const data = {
      user: {
        id: user.id,
        name: user.name,
      },
    };
    const authToken = jwt.sign(data, JWT_SECRET);

    res.json({ authToken });
  } catch (error) {
    if (error.code === 11000) {
      // Duplicate key error
      return res.status(400).json({ errors: "User already exists" });
    }

    console.error(error);

    res.status(500).json({ errors: "Server error" });
  }
});
```

Validating Token, accessing data (on every request)

```
const jwt = require("jsonwebtoken");

const fetchUser = (req, res, next) => {
  const token = req.header("auth-token");
  if (!token) {
    return res
      .status(401)
      .json({ error: "Please authenticate using a valid token" });
  }
  try {
    const data = jwt.verify(token, "Bharat");
    req.user = data.user;
    next();
  } catch (error) {
    return res
      .status(401)
      .json({ error: "Please authenticate using a valid token" });
  }
};

module.exports = fetchUser;
```

…..soon long term session validation.