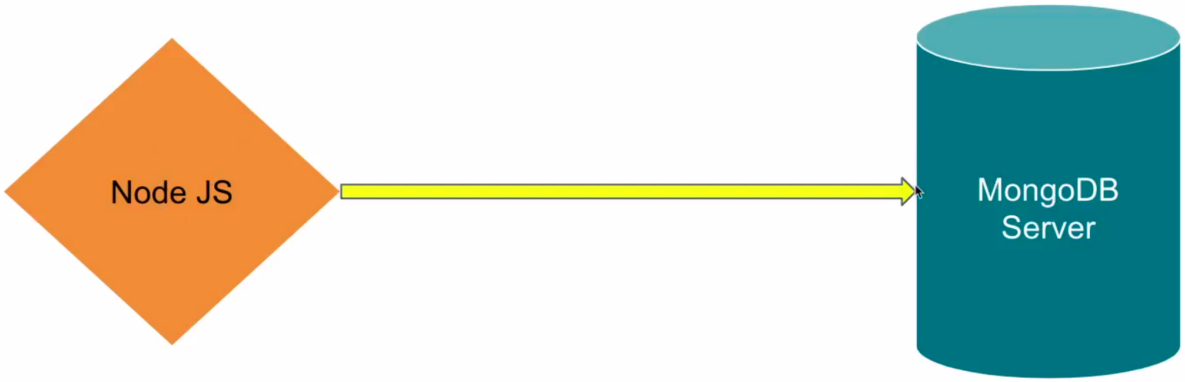
MongoDB with NodeJS - I

MongoDB Driver



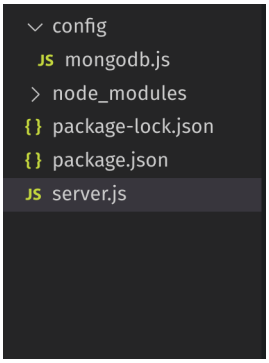
The MongoDB driver is a crucial link between your Node.js application and the MongoDB database.

Installation of MongoDB Driver

- It provides programming tools and interfaces that empower your application to interact with MongoDB seamlessly.
- Install MongoDB driver by running the command: `npm i mongodb`

Implementing MongoDB with NodeJS

- Create a 'config' directory to establish a separation of configurations in our project. Within this directory, create a 'mongodb.js' file to facilitate the connection to MongoDB:



- Import MongoClient:

```
import { MongoClient } from 'mongodb';
```

- Define Connection URL:

```
const url = 'mongodb://localhost:27017/mydb';
```

localhost:27017: The hostname and port number of the MongoDB server.
mydb : The name of the specific database you want to connect to.

- Connection to MongoDB:
  Use the connect method of the MongoClient instance to establish a connection to the MongoDB server

```
import {MongoClient} from 'mongodb';

const url = "mongodb://localhost:27017/mydb";

const connectToMongoDB = () => {
    MongoClient.connect(url)
        .then(client => {
            console.log("Mongodb is connected");
        })
```

```
        .catch(err => {
            console.log(err);
        })
}

export default connectToMongoDB;
```

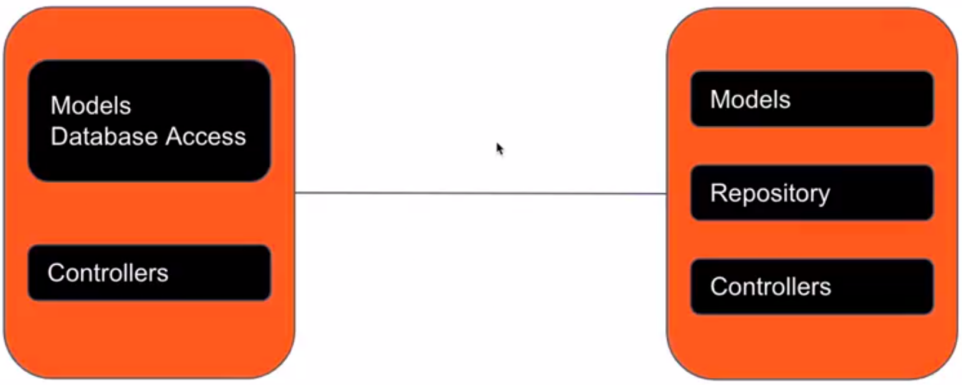
We have used **.connect( )** method of MongoClient that returns a promise and then we have exported the **connecToMongoDB** function.

- Import the above function in server.js

```
import express from 'express';
import connectToMongoDB from './config/mongodb.js';
const app = express();

app.listen(8080, () => {
      console.log('server is running on port 8080');
      connectToMongoDB();
})
```

Repository Pattern



The repository pattern is a software design concept that promotes the separation of concerns and enhances the maintainability and scalability of applications by providing an organised approach to interact with data sources, such as databases.

Benefits of Repository Pattern

- **Abstraction**: It abstracts data source details, providing a consistent interface for data operations across diverse storage mechanisms.
- **Modularity**: Repositories encapsulate data logic, enabling modular and reusable code components.
- **Maintenance**: Changes to data source or structure are localised within repositories, simplifying maintenance efforts.
- **Testing**: Repositories facilitate isolated unit testing by allowing mock implementations.
- **Caching**: Data caching can be implemented within repositories for improved performance.
- **Query Logic**: Complex queries and filtering logic are centralised in repositories.
- **Database Agnosticism**: The pattern enables flexibility in switching between different data sources.
- **Security**: Repositories can enhance security through parameterised queries and validation.

The Use Case of the Repository Pattern

Scenario: Online Bookstore Management System

Explanation:

In this example, we'll use the mongodb package to interact with MongoDB. The repository pattern will help us separate the data access logic from the business logic.

MongoDB Connection:

```
//databaseConnection.js

import { MongoClient } from 'mongodb';

const url = 'mongodb://localhost:27017';
const dbName = 'bookstore';

let client;

async function connect() {
  client = new MongoClient(url);
  await client.connect();
  console.log('Connected to MongoDB');
}

function getDatabase() {
  return client.db(dbName);
}

export { connect, getDatabase };
```

Book Repository:

```
import { ObjectID } from 'mongodb';
import { getDatabase } from './databaseConnection';
```

```
class BookRepository {
  async getAll() {
    const db = getDatabase();
    return await db.collection('books').find().toArray();
  }

  async getById(bookId) {
    const db = getDatabase();
    return await db.collection('books').findOne({ _id:
ObjectID(bookId) });
  }

  async create(bookData) {
    const db = getDatabase();
    const result = await
db.collection('books').insertOne(bookData);
    return result.ops[0];
  }

  async update(bookId, bookData) {
    const db = getDatabase();
    return await db.collection('books').findOneAndUpdate(
      { _id: ObjectID(bookId) },
      { $set: bookData },
      { returnOriginal: false } // If returnOriginal is set to
true (which is the default), the operation will return the
original document (before the update) as the result of the
operation. The document in the database will be updated with the
new values, but the returned result will be the state of the
document before the update.
    );
  }
```

```
  async delete(bookId) {
    const db = getDatabase();
    await db.collection('books').deleteOne({ _id: ObjectID(bookId)
});
  }
}

export default BookRepository;
```

Usage:

Here's what we're going to do:

Connect to the Database: We'll start by establishing a connection to the MongoDB database using the connect function.

Create Books: We'll utilise the BookRepository to create two book entries: "The Great Gatsby" by F. Scott Fitzgerald and "To Kill a Mockingbird" by Harper Lee.

Retrieve All Books: We'll fetch and display all the books stored in the collection using the getAll method.

Update a Book Title: We'll demonstrate updating a book's title using the update method after fetching it by its ID.

Delete a Book: We'll delete one of the books from the collection using the delete method.

```
import { connect } from './databaseConnection';
```

```
import BookRepository from './bookRepository';

(async () => {
  try {
    await connect();

    const bookRepo = new BookRepository();

    const book1 = await bookRepo.create({
      title: 'The Great Gatsby',
      author: 'F. Scott Fitzgerald',
    });

    const book2 = await bookRepo.create({
      title: 'To Kill a Mockingbird',
      author: 'Harper Lee',
    });

    console.log('Books created:', book1, book2);

    const allBooks = await bookRepo.getAll();
    console.log('All books:', allBooks);

    const bookToUpdate = await bookRepo.getById(book1._id);
    if (bookToUpdate) {
      bookToUpdate.title = 'Updated Title';
      const updatedBook = await bookRepo.update(bookToUpdate._id,
bookToUpdate);
      console.log('Updated book:', updatedBook);
    }

    await bookRepo.delete(book2._id);
```

```
    console.log('Book deleted:', book2._id);
  } catch (error) {
    console.error('Error:', error);
  } finally {
    if (client) {
      client.close();
      console.log('Disconnected from MongoDB');
    }
  }
})();
```

In this example, we're using the mongodb package to interact directly with MongoDB. The databaseConnection.js file handles the database connection, and the BookRepository class encapsulates the data access logic. This implementation adheres to the repository pattern and separates data access from business logic.

Hashing Passwords

- Hashing passwords is a crucial practice in security to protect sensitive user credentials.
- It involves converting plain-text passwords into irreversible and unique strings of characters using cryptographic algorithms.
- This transformation ensures that attackers cannot easily decipher the original passwords even if a database breach occurs.

Working with bcrypt

- Bcrypt is a widely used password-hashing algorithm that enhances security by transforming passwords into irreversible hash values.
- Bcrypt is favoured for its key features: salting and multiple rounds of hashing. Salting involves adding a random value (salt) to each password before hashing, preventing attackers from using precomputed hash tables (rainbow tables) to crack passwords.
- To install it, run the command: `npm i bcrypt`

Basic usage

- After installing it, we have to import it.

```
import bcrypt from 'bcrypt';
```

- The hashPassword function asynchronously generates a secure hash from a plain-text password and returns the hash. It utilises the bcrypt.genSalt and bcrypt.hash functions.

```
async function hashPassword(plainPassword) {
    const saltRounds = 12;
    const salt = await bcrypt.genSalt(saltRounds);
    const hash = await bcrypt.hash(plainPassword, salt);
    return hash;
}
```

The greater the amount of salt, the more intricate the complexity becomes, so it should be kept at a minimum.

- Now we can compare two passwords using **.compare** method.

```
async function authenticateUser(plainPassword, storedHash) {
    const passwordsMatch = await bcrypt.compare(plainPassword,
storedHash);
    return passwordsMatch; // Returns true if passwords match
}
```

Understanding dotenv

- dotenv is a popular npm package that simplifies the management of environment variables in Node.js applications.
- It enables developers to store sensitive configuration information, API keys, and other settings in a .env file, separate from the source code.
- dotenv loads the variables from the .env file into the environment, making them accessible via the process.env object.

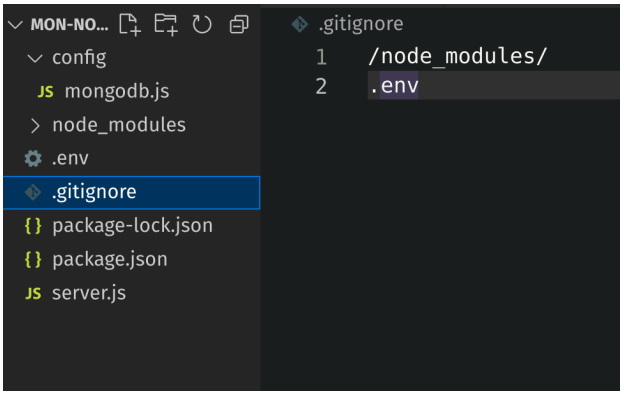
Reasons for using dotenv

- Security: Sensitive data is kept outside of the source code, reducing the risk of accidental exposure.
- Readability: The separation of configuration data from code improves the clarity and maintainability of the application.
- Ease of Use: dotenv simplifies the process of loading environment variables, requiring only a single line of code.
- Portability: Developers can easily share the same codebase across different environments without changing the source code.
- Configuration: dotenv enhances the configuration process by providing a standardised method to handle environment-specific variables.

Using dotenv to Safely Configure MongoDB url

- Install it using the command: `npm i dotenv`

- Create .env file in root directory of the app.



Ensure your .gitignore file includes it to prevent sensitive information and configuration data from being accidentally pushed to version control systems like Git.

- Save the important URLs or keys of passwords in the .env file.

```
DB_URL = mongodb://localhost:27017/mydb
JWT_SECRET = your_jwt_secret_key
```

- Import dotenv wherever needed, and call the **.config()** method to use the variables.

```
import {MongoClient} from 'mongodb';
import dotenv from 'dotenv';

dotenv.config();

const url = process.env.DB_URL;
...
```

.

Secure Car Dealership Management with Repository Pattern in Node.js

In this concise guide, we'll build a secure car dealership management system using Node.js, the Repository Pattern, the bcrypt library for password security, and the dotenv library for environment variables. Our scenario involves a car dealership managing its inventory and sales.

Scenario:
You are developing a car dealership management system for "Speedy Motors." The system should securely store user credentials, manage car inventory, and track sales.

1. User Authentication:

Step 1: Repository Setup

In user.repository.js:

```
import { getDB } from '../config/mongodb.js';
import bcrypt from 'bcrypt';
import { ApplicationError } from
'../../error-handler/applicationError.js';

class UserRepository {
    // Constructor and other methods...

    async signUp(newUser) {
        try {
```

```
            // ... Database setup
            const hashedPassword = await
bcrypt.hash(newUser.password, 10);
            newUser.password = hashedPassword;
            // Insert user into the collection
        } catch (err) {
            throw new ApplicationError("Database Error", 500);
        }
    }

    async signIn(email, password) {
        try {
            // ... Database setup
            const user = await collection.findOne({ email });

            if (user && await bcrypt.compare(password,
user.password)) {
                return user;
            } else {
                return null;
            }
        } catch (err) {
            throw new ApplicationError("Database Error", 500);
        }
    }
}

export default UserRepository;
```

Step 2: Controller and Routes

In user.controller.js:

```
import UserRepository from './user.repository.js';

export default class UserController {

  constructor() {
    this.userRepository = new UserRepository();
  }

  async signUp(req, res) {
    const { name, email, password, role } = req.body;
    const user = { name, email, password, role };
    const createdUser = await this.userRepository.signUp(user);
    res.status(201).send({ message: "User registered", user:
createdUser });
  }

  async signIn(req, res) {
    const { email, password } = req.body;
    const user = await this.userRepository.signIn(email,
password);
    if (user) {
        res.status(200).send({ message: "Sign in successful", user
});
    } else {
        res.status(401).send({ message: "Invalid credentials" });
    }
  }
```

```
}
```

2. Car Inventory Management:

Step 1: Repository Setup

In car.repository.js:

```
import { getDB } from '../../config/mongodb.js';
import { ApplicationError } from
"../../error-handler/applicationError.js";

class CarRepository {
    // Constructor and other methods...

    async add(newCar) {
        try {
            // ... Database setup
            await collection.insertOne(newCar);
            return newCar;
        } catch (err) {
            throw new ApplicationError("Database Error", 500);
        }
    }

    async getAll() {
        try {
            // ... Database setup
            const cars = await collection.find().toArray();
```

```
            return cars;
        } catch (err) {
            throw new ApplicationError("Database Error", 500);
        }
    }

    // Other methods for updating, filtering, and more...
}

export default CarRepository;
```

Step 2: Controller and Routes

In car.controller.js:

```
import CarRepository from './car.repository.js';

export default class CarController {

  constructor() {
    this.carRepository = new CarRepository();
  }

  async addCar(req, res) {
    const { make, model, year, price } = req.body;
    const newCar = { make, model, year, price };
    const createdCar = await this.carRepository.add(newCar);
    res.status(201).send({ message: "Car added", car: createdCar
});
```

```
  }

  async getAllCars(req, res) {
    const cars = await this.carRepository.getAll();
    res.status(200).send(cars);
  }

  // Other methods for managing car inventory...
}
```

3. Environment Variables with dotenv:

Create a .env file:

```
DB_CONNECTION_STRING=mongodb://localhost:27017/mydb
SECRET_KEY=mysecretkey
```

In config/mongodb.js:

```
import dotenv from 'dotenv';
dotenv.config();

// Use process.env.DB_CONNECTION_STRING in your code
```

Conclusion:

By implementing the Repository Pattern, secure user authentication using bcrypt, and utilizing environment variables with dotenv, we've built a robust car dealership management system. This scenario-based approach covers user registration,

authentication, car inventory, and sales tracking. The approach ensures data security, integrity, and scalability in your application.

Summarising it

Let's summarise what we have learned in this module:

- We learned how to establish a connection to the MongoDB database from a Node.js application.
- Using MongoClient, we accessed the database, performed CRUD operations, and interacted with collections to store and retrieve data.
- We explored the repository pattern, a design principle that separates data access logic from the rest of the application.
- We delved into the crucial aspect of password security by hashing user passwords using the bcrypt library.
- We also learned about environment variables, which helps in safeguarding our critical data such as database key.

Some Additional Resources:

- **MongoDB Node.js Driver**
- **bcrypt**
- **dotenv**