SE3040 – Application Frameworks
BSc (Hons) in Information Technology
Software Engineering Specialization
3rd Year
Faculty of Computing
SLIIT
2025 - Practical
Lab 07

MongoDB

MongoDB is a popular open-source NoSQL database. Unlike traditional relational databases that use tables and rows, MongoDB is document-oriented, using JSON-like documents with optional schemas to store data. MongoDB stores data in BSON (Binary JSON) format, which extends the JSON model to provide additional data types and to be efficient for encoding and decoding within different languages.

Work with MongoDB

- **Setup**
    - MongoDB Community Server Download

      https://www.mongodb.com/try/download/community

- Open **MongoDB Compass**
    - Launch MongoDB Compass. Wait for it to load and present the connection screen.
- Connect **to Your MongoDB Instance**
    - Connect to the default local MongoDB instance by clicking the 'Connect' button or enter your custom connection URI and connect.

- Create (Insert Data)

1. **Create a New Database and Collection**
    - Click on "Create Database". Name it `labDB` and create a collection named `users`.
    - Click "Create Database" to finalize.
2. **Insert Documents into the Collection**
    - Select the `users` collection in your **`labDB`** database.
    - Click on "Insert Document". In the dialog that appears, enter a document like `{"name": "John Doe", "age": 30, "email": "johndoe@example.com"}`.
    - Add a few more documents with different data for testing purposes.

- Read (Query Data)

1. **Find Documents in a Collection**
    - With the `users` collection selected, click on the "Find" tab.
    - Without any filter, clicking "Find" will show all documents.
    - To filter results, enter a query like `{"age": {"$gt": 25}}` to find all users older than 25.
    - `{"age": {"$gt": 25}, "name": "John"},` this query checks for documents where the `age` field is greater than 25 and the `name` field is exactly "John".

https://www.almabetter.com/bytes/cheat-sheet/mongodb

- Update (Modify Data)

1. **Update a Document**
    - In the `users` collection, find a document you want to update.
    - Click the "Edit Document" button in the document's entry.
    - Modify the document's content, such as changing the `name` or `age`, and then click "Update".

- Delete (Remove Data)

1. **Delete a Document**
    - Locate a document you wish to delete in the `users` collection.

- Click on the "Delete Document" button (trash bin icon) next to the document and confirm the deletion.

Part 1 : Node.js with MongoDB

- Setup Your Node.js Project

  Initialize a new Node.js project in your chosen directory by running `` `npm init -y` `` to create a `` `package.json` `` file.

- Install Dependencies

  Install necessary Node.js modules: Express for the web server, MongoDB driver to connect to your MongoDB database, and body-parser to parse incoming request bodies in a middleware. Use `` `npm install express mongodb mongoose body-parser` ``.

- Create the directories and files

  Create the folders (`routes`, `models`) and files (`server.js`, `users.js`, `user.js`).

- Create the Server and Connect to MongoDB

  In your project directory, create a file named `server.js`. Use the provided code to set up an Express server that connects to MongoDB and defines a POST route `/users` for inserting data into the `users` collection.

- ***`server.js`***

  This is the entry point of your application. It sets up the server and connects to MongoDB.

```
const express = require('express');
const mongoose = require('mongoose');
const userRoutes = require('./routes/users');


const app = express();
const port = 3000;

// Middleware to parse JSON bodies
app.use(express.json());

// MongoDB connection
mongoose.connect('mongodb://localhost:27017/myNodeProject', {
    useNewUrlParser: true,
```

```
    useUnifiedTopology: true
})
.then(() => console.log('MongoDB connected'))
.catch(err => console.error('MongoDB connection error:', err));

// Routes
app.use('/users', userRoutes);

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

- ***routes/users.js***

This file contains the routing for user-related operations.

```
const express = require('express');
const User = require('../models/user');

const router = express.Router();

// POST route to create a new user
router.post('/', async (req, res) => {
    try {
        const user = new User(req.body);
        const savedUser = await user.save();
        res.status(201).send(savedUser);
    } catch (error) {
        res.status(400).send(error);
    }
});

// GET route to fetch all users
router.get('/', async (req, res) => {
    try {
        const users = await User.find({});
        res.status(200).send(users);
    } catch (error) {
        res.status(500).send(error);
    }
```

```
});

// GET route to fetch a single user by id
router.get('/:id', async (req, res) => {
    try {
        const user = await User.findById(req.params.id);
        if (!user) {
            return res.status(404).send();
        }
        res.status(200).send(user);
    } catch (error) {
        res.status(500).send(error);
    }
});

// PATCH route to update a user by id
router.patch('/:id', async (req, res) => {
    try {
        const updatedUser = await User.findByIdAndUpdate(req.params.id, req.body,
{ new: true });
        if (!updatedUser) {
            return res.status(404).send();
        }
        res.status(200).send(updatedUser);
    } catch (error) {
        res.status(400).send(error);
    }
});

// DELETE route to delete a user by id
router.delete('/:id', async (req, res) => {
    try {
        const user = await User.findByIdAndDelete(req.params.id);
        if (!user) {
            return res.status(404).send('User not found');
        }
        // Send a message along with the name of the deleted user
        res.status(200).send({ message: 'User Deleted Successfully ', name:
user.name });
    } catch (error) {
        res.status(500).send(error);
    }
});


module.exports = router;
```

- **models/user.js**

This file defines the Mongoose schema and model for a user.

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

const userSchema = new mongoose.Schema({
    name: String,
    age: Number,
    email: {
        type: String,
        required: true,
        unique: true
    }
});

const User = mongoose.model('User', userSchema);

module.exports = User;
```

- `server.js` initializes the Express server and sets up the connection to MongoDB using Mongoose.
- `routes/users.js` defines a route to handle GET, POST, PATCH, DELETE requests for creating new users, leveraging the User model.
- `models/user.js` sets up a Mongoose schema and model for user data, which is then used by the routes to interact with the MongoDB database.

- Run the Server

  Start the server using `node server.js`. This will launch the Node.js server on `localhost:3000` and connect to the MongoDB database.

- Test the API

  Use a tool like Postman or curl to send a POST request to `http://localhost:3000/users` with a JSON payload to insert data into the MongoDB database.

- The `GET /` route fetches all users from the database.
- The `GET /:id` route fetches a single user by their unique ID.
- The `PATCH /:id` route updates a user's information based on their unique ID. This uses `findByIdAndUpdate` and returns the updated document.
- The `DELETE /:id` route removes a user from the database based on their unique ID.

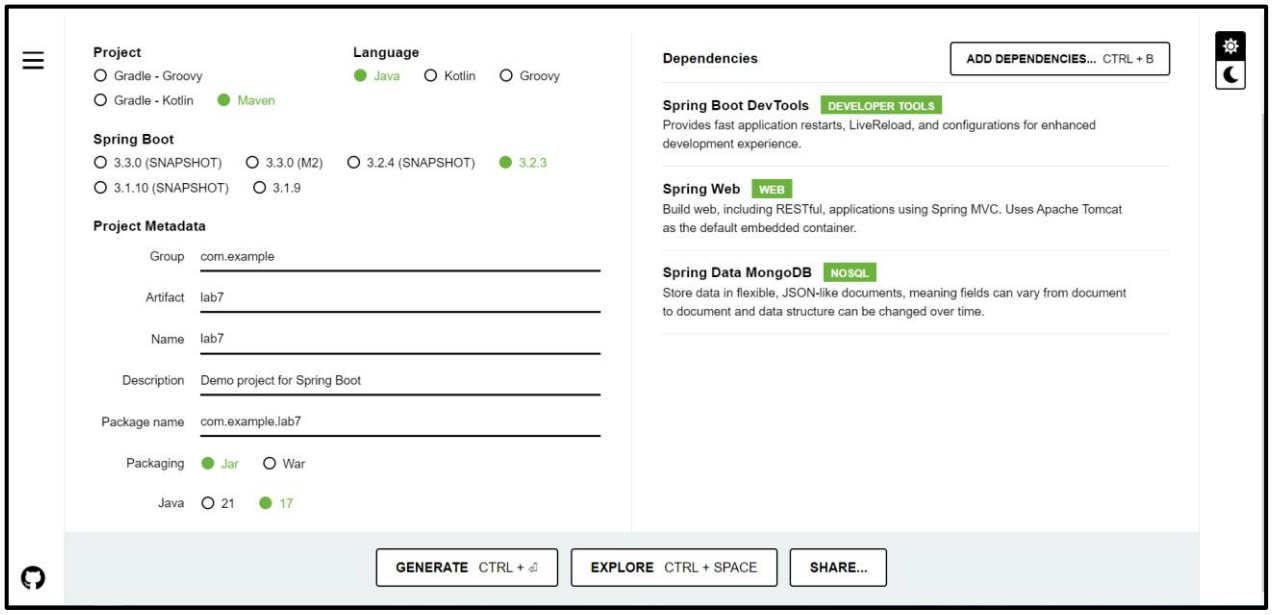
Part 2 : Spring Boot with MongoDB

Setup

1. **Spring Boot Project Initialization**

To create a Spring Boot project with Spring Initializr

1. **Access Spring Initializr**: Visit Spring Initializr.
2. **Project Metadata**: Enter the project's details, including Group, Artifact, Name, and Description.
3. **Project Type**: Choose Maven Project or Gradle Project based on your preference.
4. **Language**: Select Java as the programming language.
5. **Spring Boot Version**: Pick the latest stable version or your preferred one.
6. **Dependencies**: Add `Spring Web` for creating web applications and `Spring Data MongoDB` to integrate MongoDB.
7. **Generate Project**: Click the "Generate" button to download your project as a ZIP file.
8. **Project Setup**: Extract the ZIP file and open it in your Integrated Development Environment (IDE) like IntelliJ IDEA or VS Code.



Add the Database Connection to the application.

The MongoDB connection details in a Spring Boot application are typically specified in the `application.properties` or `application.yml` file within the `src/main/resources` directory. Here, you configure the database name, port, and host for MongoDB.

Application properties (`application.properties`)

```
spring.application.name=lab7

spring.data.mongodb.database=lab7
spring.data.mongodb.port=27017
spring.data.mongodb.host=localhost
```

This setup tells Spring Boot to connect to the MongoDB server running on localhost with port `27017` and use the `lab7` database. Spring Boot automatically uses these properties to configure the connection to MongoDB.

Define the Model: In the `User` class, ensure all necessary fields are defined and annotated properly for MongoDB document mapping.

User Model (`User.java`)

```java
package com.example.lab7.model;

import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

@Document(collection = "users")
public class User {
    @Id
    private String id;
    private String name;
    private int age;
    private String email; // Additional field example

    // Constructors, Getters, and Setters
    public User() {}

    public User(String name, int age, String email) {
        this.name = name;
        this.age = age;
        this.email = email;
    }

    // Getter and setter methods for id, name, age, and email
    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
```

```
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }


}
```

Create Repository Interface: The `UserRepository` should extend `MongoRepository` for MongoDB operations.

Repository (`UserRepository.java`)

```
package com.example.lab7.repository;

import com.example.lab7.model.User;
import org.springframework.data.mongodb.repository.MongoRepository;

public interface UserRepository extends MongoRepository<User, String> {
    // Custom database queries can be defined here
}
```

Implement Service Layer: In `UserService`, implement methods for business logic, utilizing `UserRepository` for database interactions.

User Service (`UserService.java`) -Add methods for creating, updating, and deleting users to support full CRUD operations.

```java
package com.example.lab7.service;

import com.example.lab7.model.User;
import com.example.lab7.repository.UserRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.Optional;

@Service
public class UserService {
    private final UserRepository userRepository;

    @Autowired
    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public List<User> getAllUsers() {
        return userRepository.findAll();
    }

    public User createUser(User user) {
        return userRepository.save(user);
    }

    public Optional<User> getUserById(String id) {
        return userRepository.findById(id);
    }

    public User updateUser(String id, User user) {
        user.setId(id);
        return userRepository.save(user);
    }

    public void deleteUser(String id) {
        userRepository.deleteById(id);
    }
}
```

Develop REST Controller: In `UserController`, define methods for handling HTTP requests (GET, POST, PUT, DELETE) that interact with `UserService`.

User Controller (`UserController.java`)

```
package com.example.lab7.controller;

import com.example.lab7.model.User;
import com.example.lab7.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/users")
public class UserController {
    private final UserService userService;

    @Autowired
    public UserController(UserService userService) {
        this.userService = userService;
    }

    @GetMapping
    public List<User> getAllUsers() {
        return userService.getAllUsers();
    }

    @PostMapping
    public User createUser(@RequestBody User user) {
        return userService.createUser(user);
    }

    @GetMapping("/{id}")
    public ResponseEntity<User> getUserById(@PathVariable String id) {
        return userService.getUserById(id)
                          .map(ResponseEntity::ok)
                          .orElseGet(() -> ResponseEntity.notFound().build());
    }

    @PutMapping("/{id}")
    public ResponseEntity<User> updateUser(@PathVariable String id, @RequestBody
User user) {
```

```
        return userService.getUserById(id)
                          .map(existingUser ->
ResponseEntity.ok(userService.updateUser(id, user)))
                          .orElseGet(() -> ResponseEntity.notFound().build());
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<?> deleteUser(@PathVariable String id) {
        return userService.getUserById(id)
                          .map(user -> {
                              userService.deleteUser(id);
                              return ResponseEntity.ok().build();
                          })
                          .orElseGet(() -> ResponseEntity.notFound().build());
    }
}
```

Run and Test: Start the Spring Boot application using this command.

mvnw.cmd spring-boot:run

And test the REST API endpoints using a tool like Postman to ensure they are working and data is being stored in MongoDB.