**Student Management System – SpringBoot Backend API with MongoDB**

Goal – We have to build a Spring Boot backend API using MongoDB Spring Data and Mongo Template. Let's start with the project structure and then move on to the code.

**Project Structure**

StudentManagementSystem/

│

├── src/

│ ├── main/

│ │ ├── java/

│ │ │ └── com/

│ │ │ └── example/

│ │ │ └── studentmanagement/

│ │ │ ├── controller/

│ │ │ │ └── StudentController.java

│ │ │ ├── model/

│ │ │ │ └── Student.java

│ │ │ ├── repository/

│ │ │ │ └── StudentRepository.java

│ │ │ ├── service/

│ │ │ │ └── StudentService.java

│ │ │ └── StudentManagementApplication.java

│ │ │

│ │ └── resources/

│ │ └── application.properties

│ │

│ └── test/

│ └── java/

│ └── com/

│ └── example/

│ └── studentmanagement/

│ ├── controller/

│ │ └── StudentControllerTest.java

│ ├── repository/

│ │ └── StudentRepositoryTest.java

│ └── service/

│ └── StudentServiceTest.java

│

└── pom.xml

**pom.xml**

xml

Copy code

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.example</groupId>

<artifactId>student-management-system</artifactId>

<version>1.0-SNAPSHOT</version>

<packaging>jar</packaging>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.6.3</version>

</parent>

<dependencies>

<!-- Spring Boot Starter Web for RESTful APIs -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<!-- Spring Boot Starter Data MongoDB for MongoDB integration -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-mongodb</artifactId>

</dependency>

<!-- Lombok for reducing boilerplate code -->

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<optional>true</optional>

</dependency>

<!-- Test dependencies -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<!-- Maven Compiler Plugin -->

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.8.1</version>

<configuration>

<source>11</source>

<target>11</target>

</configuration>

</plugin>

</plugins>

</build>

</project>

**Student.java (Model)**

package com.example.studentmanagement.model;

import lombok.Data;

import org.springframework.data.annotation.Id;

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

@Data

@Document(collection = "students")

public class Student {

@Id

private String id;

private String firstName;

private String lastName;

private int age;

private String email;

}

**StudentRepository.java (Repository)**

package com.example.studentmanagement.repository;

import com.example.studentmanagement.model.Student;

import org.springframework.data.mongodb.repository.MongoRepository;

public interface StudentRepository extends MongoRepository<Student, String> {

}

**StudentService.java (Service)**

package com.example.studentmanagement.service;

import com.example.studentmanagement.model.Student;

import com.example.studentmanagement.repository.StudentRepository;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import java.util.List;

import java.util.Optional;

@Service

public class StudentService {

@Autowired

private StudentRepository studentRepository;

public List<Student> getAllStudents() {

return studentRepository.findAll();

}

public Optional<Student> getStudentById(String id) {

return studentRepository.findById(id);

}

public Student createStudent(Student student) {

return studentRepository.save(student);

}

public Student updateStudent(String id, Student student) {

if (studentRepository.existsById(id)) {

student.setId(id);

return studentRepository.save(student);

}

return null;

}

public void deleteStudent(String id) {

studentRepository.deleteById(id);

}

}

**StudentController.java (Controller)**

package com.example.studentmanagement.controller;

import com.example.studentmanagement.model.Student;

import com.example.studentmanagement.service.StudentService;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import java.util.List;

import java.util.Optional;

@RestController

@RequestMapping("/api/students")

public class StudentController {

@Autowired

private StudentService studentService;

@GetMapping

public ResponseEntity<List<Student>> getAllStudents() {

List<Student> students = studentService.getAllStudents();

return new ResponseEntity<>(students, HttpStatus.OK);

}

@GetMapping("/{id}")

public ResponseEntity<Student> getStudentById(@PathVariable String id) {

Optional<Student> student = studentService.getStudentById(id);

return student.map(value -> new ResponseEntity<>(value, HttpStatus.OK))

.orElseGet(() -> new ResponseEntity<>(HttpStatus.NOT\_FOUND));

}

@PostMapping

public ResponseEntity<Student> createStudent(@RequestBody Student student) {

Student createdStudent = studentService.createStudent(student);

return new ResponseEntity<>(createdStudent, HttpStatus.CREATED);

}

@PutMapping("/{id}")

public ResponseEntity<Student> updateStudent(@PathVariable String id, @RequestBody Student student) {

Student updatedStudent = studentService.updateStudent(id, student);

if (updatedStudent != null) {

return new ResponseEntity<>(updatedStudent, HttpStatus.OK);

} else {

return new ResponseEntity<>(HttpStatus.NOT\_FOUND);

}

}

@DeleteMapping("/{id}")

public ResponseEntity<Void> deleteStudent(@PathVariable String id) {

studentService.deleteStudent(id);

return new ResponseEntity<>(HttpStatus.NO\_CONTENT);

}

}

**StudentManagementApplication.java (Main Application)**

package com.example.studentmanagement;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class StudentManagementApplication {

public static void main(String[] args) {

SpringApplication.run(StudentManagementApplication.class, args);

}

}

**application.properties (MongoDB Configuration)**

spring.data.mongodb.host=localhost

spring.data.mongodb.port=27017

spring.data.mongodb.database=studentdb

Now - Include the implementation using **MongoTemplate**.

**StudentService.java (Using MongoTemplate)**

package com.example.studentmanagement.service;

import com.example.studentmanagement.model.Student;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.data.mongodb.core.MongoTemplate;

import org.springframework.stereotype.Service;

import java.util.List;

@Service

public class StudentService {

@Autowired

private MongoTemplate mongoTemplate;

public List<Student> getAllStudents() {

return mongoTemplate.findAll(Student.class);

}

public Student getStudentById(String id) {

return mongoTemplate.findById(id, Student.class);

}

public Student createStudent(Student student) {

return mongoTemplate.insert(student);

}

public Student updateStudent(String id, Student student) {

student.setId(id);

return mongoTemplate.save(student);

}

public void deleteStudent(String id) {

mongoTemplate.remove(getStudentById(id));

}

}

Note – Here we use MongoTemplate for database operations instead of StudentRepository. This approach gives you more control over the queries and allows you to perform advanced operations if needed. You can inject MongoTemplate into your service class and use its methods like insert, save, find, and remove for CRUD operations.

**Here are some examples of simple queries on the Student collection using the** MongoTemplate find() method:

**1. Find All Students**

List<Student> students = mongoTemplate.findAll(Student.class);

**2. Find Student by ID**

String studentId = "60a7c2efb7a5cb001cb19b8d"; // Example student ID

Student student = mongoTemplate.findById(studentId, Student.class);

**3. Find Students by Age**

int age = 20; // Example age

Query query = new Query(Criteria.where("age").is(age));

List<Student> students = mongoTemplate.find(query, Student.class);

**4. Find Students by First Name and Last Name**

String firstName = "John"; // Example first name

String lastName = "Doe"; // Example last name

Query query = new

Query(Criteria.where("firstName").is(firstName).and("lastName").is(lastName));

List<Student> students = mongoTemplate.find(query, Student.class);

**5. Find Students with Age Greater Than or Equal to a Value**

int minAge = 18; // Example minimum age

Query query = new Query(Criteria.where("age").gte(minAge));

List<Student> students = mongoTemplate.find(query, Student.class);

**Queries using the aggregation pipeline - using MongoTemplate in Spring Boot:**

**1. Aggregation to Get Average Age of Students**

Aggregation aggregation = Aggregation.newAggregation(

Aggregation.group().avg("age").as("averageAge")

);

AggregationResults<AverageAgeResult> result = mongoTemplate.aggregate(aggregation, "students", AverageAgeResult.class);

AverageAgeResult averageAgeResult = result.getUniqueMappedResult();

Double averageAge = averageAgeResult != null ? averageAgeResult.getAverageAge() : null;

Define the AverageAgeResult class:

@Data

public class AverageAgeResult {

private Double averageAge;

}

**2. Aggregation to Count Students by Age Range**

Aggregation aggregation = Aggregation.newAggregation(

Aggregation.group(

CaseOperator

.when(Criteria.where("age").lte(20)).then("0-20")

.when(Criteria.where("age").lte(30)).then("21-30")

.otherwise("31+")

).count().as("count")

);

AggregationResults<AgeRangeCount> result = mongoTemplate.aggregate(aggregation, "students", AgeRangeCount.class);

List<AgeRangeCount> ageRangeCounts = result.getMappedResults();

Define the AgeRangeCount class:

@Data

public class AgeRangeCount {

private String ageRange;

private Long count;

}

**3. Aggregation to Find Students by Last Name Starting with a Specific Letter**

String startsWithLetter = "A"; // Example letter

Aggregation aggregation = Aggregation.newAggregation(

Aggregation.match(Criteria.where("lastName").regex("^" + startsWithLetter, "i"))

);

AggregationResults<Student> result = mongoTemplate.aggregate(aggregation, "students", Student.class);

List<Student> students = result.getMappedResults();

**4. Aggregation to Get Total Count of Students**

Aggregation aggregation = Aggregation.newAggregation(

Aggregation.group().count().as("totalCount")

);

AggregationResults<TotalCountResult> result = mongoTemplate.aggregate(aggregation, "students", TotalCountResult.class);

TotalCountResult totalCountResult = result.getUniqueMappedResult();

Long totalCount = totalCountResult != null ? totalCountResult.getTotalCount() : null;

Define the TotalCountResult class:

@Data

public class TotalCountResult {

private Long totalCount;

}

**5. Aggregation to Get Students Sorted by Age in Descending Order**

Aggregation aggregation = Aggregation.newAggregation(

Aggregation.sort(Sort.Direction.DESC, "age")

);

AggregationResults<Student> result = mongoTemplate.aggregate(aggregation, "students", Student.class);

List<Student> students = result.getMappedResults();

**6. Aggregation to Get Unique Distinct Age Values**

Aggregation aggregation = Aggregation.newAggregation(

Aggregation.group("age").first("age").as("age")

);

AggregationResults<DistinctAgeResult> result = mongoTemplate.aggregate(aggregation, "students", DistinctAgeResult.class);

List<DistinctAgeResult> distinctAges = result.getMappedResults();

Define the DistinctAgeResult class:

@Data

public class DistinctAgeResult {

private int age;

}

**7. Aggregation to Project Only Specific Fields of Students**

Aggregation aggregation = Aggregation.newAggregation(

Aggregation.project("firstName", "lastName")

);

AggregationResults<BasicStudentInfo> result = mongoTemplate.aggregate(aggregation, "students", BasicStudentInfo.class);

List<BasicStudentInfo> basicInfos = result.getMappedResults();

Define the BasicStudentInfo class:

@Data

public class BasicStudentInfo {

private String firstName;

private String lastName;

}

TEST DATA - To add test data for 30 students to MongoDB using the mongoimport command, you can create a JSON file with the student details and then import it into your MongoDB database.

1. Save the students.json file with the student data for all 30 students.
2. Open a terminal or command prompt and navigate to the directory where students.json is located.
3. Run the following mongoimport command to import the data into your MongoDB database:

mongoimport --db your\_database\_name --collection students --file students.json --jsonArray

Replace your\_database\_name with the name of your MongoDB database where you want to import the data.

[

{

"firstName": "John",

"lastName": "Doe",

"age": 20,

"email": "john.doe@example.com"

},

{

"firstName": "Jane",

"lastName": "Smith",

"age": 22,

"email": "jane.smith@example.com"

},

{

"firstName": "Michael",

"lastName": "Johnson",

"age": 25,

"email": "michael.johnson@example.com"

},

{

"firstName": "Emily",

"lastName": "Williams",

"age": 21,

"email": "emily.williams@example.com"

},

{

"firstName": "David",

"lastName": "Brown",

"age": 23,

"email": "david.brown@example.com"

},

{

"firstName": "Sarah",

"lastName": "Miller",

"age": 24,

"email": "sarah.miller@example.com"

},

{

"firstName": "James",

"lastName": "Davis",

"age": 22,

"email": "james.davis@example.com"

},

{

"firstName": "Emma",

"lastName": "Garcia",

"age": 21,

"email": "emma.garcia@example.com"

},

{

"firstName": "Daniel",

"lastName": "Martinez",

"age": 25,

"email": "daniel.martinez@example.com"

},

{

"firstName": "Olivia",

"lastName": "Hernandez",

"age": 20,

"email": "olivia.hernandez@example.com"

},

{

"firstName": "William",

"lastName": "Lopez",

"age": 24,

"email": "william.lopez@example.com"

},

{

"firstName": "Sophia",

"lastName": "Gonzalez",

"age": 23,

"email": "sophia.gonzalez@example.com"

},

{

"firstName": "Alexander",

"lastName": "Wilson",

"age": 21,

"email": "alexander.wilson@example.com"

},

{

"firstName": "Isabella",

"lastName": "Anderson",

"age": 22,

"email": "isabella.anderson@example.com"

},

{

"firstName": "Ethan",

"lastName": "Thomas",

"age": 25,

"email": "ethan.thomas@example.com"

},

{

"firstName": "Mia",

"lastName": "Jackson",

"age": 20,

"email": "mia.jackson@example.com"

},

{

"firstName": "Jacob",

"lastName": "White",

"age": 24,

"email": "jacob.white@example.com"

},

{

"firstName": "Ava",

"lastName": "Harris",

"age": 23,

"email": "ava.harris@example.com"

},

{

"firstName": "Liam",

"lastName": "Martin",

"age": 21,

"email": "liam.martin@example.com"

},

{

"firstName": "Charlotte",

"lastName": "Thompson",

"age": 22,

"email": "charlotte.thompson@example.com"

},

{

"firstName": "Amelia",

"lastName": "Garcia",

"age": 25,

"email": "amelia.garcia@example.com"

},

{

"firstName": "Benjamin",

"lastName": "Clark",

"age": 20,

"email": "benjamin.clark@example.com"

},

{

"firstName": "Harper",

"lastName": "Rodriguez",

"age": 24,

"email": "harper.rodriguez@example.com"

},

{

"firstName": "Evelyn",

"lastName": "Lewis",

"age": 23,

"email": "evelyn.lewis@example.com"

},

{

"firstName": "Michael",

"lastName": "Walker",

"age": 21,

"email": "michael.walker@example.com"

},

{

"firstName": "Lucas",

"lastName": "Young",

"age": 22,

"email": "lucas.young@example.com"

},

{

"firstName": "William",

"lastName": "Allen",

"age": 25,

"email": "william.allen@example.com"

},

{

"firstName": "Ella",

"lastName": "King",

"age": 20,

"email": "ella.king@example.com"

},

{

"firstName": "Avery",

"lastName": "Wright",

"age": 24,

"email": "avery.wright@example.com"

}

]

**1. Add a Student (Create)**

* **Method:** POST
* **URL:** http://localhost:8080/api/students
* **Body:** JSON format with student details

{

"firstName": "John",

"lastName": "Doe",

"age": 20,

"email": "john.doe@example.com"

}

* Click on "Send" to add the student.

**2. Update a Student (Update)**

* **Method:** PUT
* **URL:** http://localhost:8080/api/students/{studentId}
* **Body:** JSON format with updated student details

{

"firstName": "John",

"lastName": "Smith",

"age": 21,

"email": "john.smith@example.com"

}

Replace {studentId} with the actual ID of the student you want to update.

* Click on "Send" to update the student.

**3. Delete a Student (Delete)**

* **Method:** DELETE
* **URL:** http://localhost:8080/api/students/{studentId} Replace {studentId} with the ID of the student you want to delete.
* Click on "Send" to delete the student.

**4. List All Students (Read)**

* **Method:** GET
* **URL:** http://localhost:8080/api/students
* Click on "Send" to get the list of all students.

**5. Query Students (Read with Query)**

For example, to get students with an age greater than or equal to 20:

* **Method:** GET
* **URL:** http://localhost:8080/api/students?age=20
* Click on "Send" to get the list of students matching the query parameter.

Here's an example of a new StudentService class in Java that includes aggregation-based queries using MongoTemplate in a Spring Boot application:

package com.example.studentmanagement.service;

import com.example.studentmanagement.model.Student;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.data.mongodb.core.MongoTemplate;

import org.springframework.data.mongodb.core.aggregation.Aggregation;

import org.springframework.data.mongodb.core.aggregation.AggregationResults;

import org.springframework.data.mongodb.core.query.Criteria;

import org.springframework.stereotype.Service;

import java.util.List;

@Service

public class StudentService {

@Autowired

private MongoTemplate mongoTemplate;

public Double getAverageAgeOfStudents() {

Aggregation aggregation = Aggregation.newAggregation(

Aggregation.group().avg("age").as("averageAge")

);

AggregationResults<AverageAgeResult> result = mongoTemplate.aggregate(aggregation, "students", AverageAgeResult.class);

AverageAgeResult averageAgeResult = result.getUniqueMappedResult();

return averageAgeResult != null ? averageAgeResult.getAverageAge() : null;

}

public List<AgeRangeCount> getCountOfStudentsByAgeRange() {

Aggregation aggregation = Aggregation.newAggregation(

Aggregation.group(

Aggregation.switchCase()

.when(Criteria.where("age").lte(20)).then("0-20")

.when(Criteria.where("age").lte(30)).then("21-30")

.otherwise("31+")

).count().as("count")

);

AggregationResults<AgeRangeCount> result = mongoTemplate.aggregate(aggregation, "students", AgeRangeCount.class);

return result.getMappedResults();

}

public List<Student> getStudentsWithLastNameStartingWith(String letter) {

Aggregation aggregation = Aggregation.newAggregation(

Aggregation.match(Criteria.where("lastName").regex("^" + letter, "i"))

);

AggregationResults<Student> result = mongoTemplate.aggregate(aggregation, "students", Student.class);

return result.getMappedResults();

}

// Additional aggregation-based queries can be added here

// Define AverageAgeResult class for average age aggregation

private static class AverageAgeResult {

private Double averageAge;

public Double getAverageAge() {

return averageAge;

}

}

// Define AgeRangeCount class for age range count aggregation

private static class AgeRangeCount {

private String ageRange;

private Long count;

public String getAgeRange() {

return ageRange;

}

public Long getCount() {

return count;

}

}

}

In this StudentService class:

* The getAverageAgeOfStudents() method calculates the average age of all students using the aggregation pipeline.
* The getCountOfStudentsByAgeRange() method counts the number of students in different age ranges using the aggregation pipeline.
* The getStudentsWithLastNameStartingWith(String letter) method retrieves students whose last names start with a specified letter using the aggregation pipeline.

**1. Get Average Age of Students**

* **Method:** GET
* **URL:** http://localhost:8080/api/students/average-age
* Click on "Send" to execute the request and get the average age of students.

**2. Get Count of Students by Age Range**

* **Method:** GET
* **URL:** http://localhost:8080/api/students/count-by-age-range
* Click on "Send" to execute the request and get the count of students in different age ranges.

**3. Get Students with Last Name Starting with a Letter**

* **Method:** GET
* **URL:** http://localhost:8080/api/students/last-name-starting-with?letter=A Replace A with the letter you want to search for in last names.
* Click on "Send" to execute the request and get the students whose last names start with the specified letter.

These requests will correspond to the methods

getAverageAgeOfStudents(), getCountOfStudentsByAgeRange(), and getStudentsWithLastNameStartingWith(String letter)

**REST Controller Endpoints and Corresponding Service Methods**

1. **Create a Student**:
   * **Controller Endpoint**:
     + Method: POST
     + URL: http://localhost:8080/api/students
     + Body: JSON data for the new student
   * **Service Method**:
     + createStudent(Student student)
2. **Get All Students**:
   * **Controller Endpoint**:
     + Method: GET
     + URL: http://localhost:8080/api/students
   * **Service Method**:
     + getAllStudents()
3. **Get Student by ID**:
   * **Controller Endpoint**:
     + Method: GET
     + URL: http://localhost:8080/api/students/{id}
     + Replace {id} with the actual student ID
   * **Service Method**:
     + getStudentById(String id)
4. **Update Student**:
   * **Controller Endpoint**:
     + Method: PUT
     + URL: http://localhost:8080/api/students/{id}
     + Replace {id} with the actual student ID
     + Body: JSON data for the updated student
   * **Service Method**:
     + updateStudent(String id, Student student)
5. **Delete Student**:
   * **Controller Endpoint**:
     + Method: DELETE
     + URL: http://localhost:8080/api/students/{id}
     + Replace {id} with the actual student ID to delete
   * **Service Method**:
     + deleteStudent(String id)
6. **Get Average Age of Students**:
   * **Controller Endpoint**:
     + Method: GET
     + URL: http://localhost:8080/api/students/average-age
   * **Service Method**:
     + getAverageAgeOfStudents()
7. **Get Count of Students by Age Range**:
   * **Controller Endpoint**:
     + Method: GET
     + URL: http://localhost:8080/api/students/count-by-age-range
   * **Service Method**:
     + getCountOfStudentsByAgeRange()
8. **Get Students with Last Name Starting with a Letter**:
   * **Controller Endpoint**:
     + Method: GET
     + URL: http://localhost:8080/api/students/last-name-starting-with?letter=A
     + Replace A with the actual letter to search for
   * **Service Method**:
     + getStudentsWithLastNameStartingWith(String letter)

**Postman Requests**

Now, let's create Postman requests for each of the above endpoints:

1. **Create Student**:
   * Method: POST
   * URL: http://localhost:8080/api/students
   * Body: JSON data for the new student
2. **Get All Students**:
   * Method: GET
   * URL: http://localhost:8080/api/students
3. **Get Student by ID**:
   * Method: GET
   * URL: http://localhost:8080/api/students/{id}
4. **Update Student**:
   * Method: PUT
   * URL: http://localhost:8080/api/students/{id}
   * Body: JSON data for the updated student
5. **Delete Student**:
   * Method: DELETE
   * URL: http://localhost:8080/api/students/{id}
6. **Get Average Age of Students**:
   * Method: GET
   * URL: http://localhost:8080/api/students/average-age
7. **Get Count of Students by Age Range**:
   * Method: GET
   * URL: http://localhost:8080/api/students/count-by-age-range
8. **Get Students with Last Name Starting with a Letter**:
   * Method: GET
   * URL: http://localhost:8080/api/students/last-name-starting-with?letter=A

To test the REST controller and service class APIs using JUnit and Mockito in a Spring Boot application, we'll first create a testing service that covers the controller and service methods. We'll also make any necessary changes to the pom.xml file and project structure to include testing dependencies and configurations.

**1. Project Structure Changes**

Assuming your project structure is standard, you'll typically have a src/main/java directory for main code and a src/test/java directory for test code. Here's an example structure:

- src

- main

- java

- com

- example

- studentmanagement

- controller

- StudentController.java

- model

- Student.java

- service

- StudentService.java

- test

- java

- com

- example

- studentmanagement

- service

- StudentServiceTest.java (New)

**2. Pom.xml Changes**

Add the necessary dependencies for testing in your pom.xml file:

<dependencies>

<!-- Spring Boot Test Starter -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<!-- Mockito for mocking objects -->

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>3.12.4</version>

<scope>test</scope>

</dependency>

</dependencies>

**3. Testing Service Class (StudentServiceTest.java)**

Create a new test class for StudentService:

package com.example.studentmanagement.service;

import com.example.studentmanagement.model.Student;

import org.junit.jupiter.api.Test;

import org.junit.jupiter.api.extension.ExtendWith;

import org.mockito.InjectMocks;

import org.mockito.Mock;

import org.mockito.junit.jupiter.MockitoExtension;

import java.util.Arrays;

import java.util.List;

import static org.mockito.Mockito.\*;

@ExtendWith(MockitoExtension.class)

public class StudentServiceTest {

@Mock

private StudentRepository studentRepository; // Assuming you have a repository for students

@InjectMocks

private StudentService studentService;

@Test

public void testCreateStudent() {

Student student = new Student();

student.setFirstName("John");

student.setLastName("Doe");

student.setAge(25);

student.setEmail("john.doe@example.com");

when(studentRepository.save(any(Student.class))).thenReturn(student);

Student savedStudent = studentService.createStudent(student);

verify(studentRepository, times(1)).save(student);

assertNotNull(savedStudent);

assertEquals("John", savedStudent.getFirstName());

// Add more assertions as needed

}

@Test

public void testGetAllStudents() {

Student student1 = new Student();

student1.setFirstName("John");

student1.setLastName("Doe");

student1.setAge(25);

student1.setEmail("john.doe@example.com");

Student student2 = new Student();

student2.setFirstName("Jane");

student2.setLastName("Smith");

student2.setAge(30);

student2.setEmail("jane.smith@example.com");

List<Student> students = Arrays.asList(student1, student2);

when(studentRepository.findAll()).thenReturn(students);

List<Student> allStudents = studentService.getAllStudents();

verify(studentRepository, times(1)).findAll();

assertNotNull(allStudents);

assertEquals(2, allStudents.size());

// Add more assertions as needed

}

// Add more test methods for other service methods like getStudentById, updateStudent, deleteStudent, etc.

}

In this test class:

* We use Mockito to mock the StudentRepository dependency of StudentService.
* Each test method verifies the behavior of a specific service method by setting up mock data and expectations.

Make sure to adjust the test methods and mock data according to your actual service methods and business logic.

**Running Tests**

You can run these tests using your IDE's built-in test runner or using Maven. For Maven, use the command mvn test to execute all tests in the project.

This setup provides a foundation for testing your Spring Boot application's controller and service layer using JUnit and Mockito. Adjustments may be needed based on your specific project setup and requirements.

An end-to-end test for a Spring Boot application involving MongoDB typically involves testing the entire application stack, including controller endpoints, service layer, and the integration with the MongoDB database. Here's a recommended approach using Spring Boot's testing capabilities along with an embedded MongoDB instance for integration testing.

**1. Add Test Dependencies to pom.xml**

Ensure you have the necessary dependencies for testing:

xml

Copy code

<dependencies>

<!-- Spring Boot Test Starter -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<!-- Spring Data MongoDB Test -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-mongodb</artifactId>

<scope>test</scope>

</dependency>

<!-- Embedded MongoDB for integration testing -->

<dependency>

<groupId>de.flapdoodle.embed</groupId>

<artifactId>de.flapdoodle.embed.mongo</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

**2. Configure Embedded MongoDB for Testing**

Create a test configuration class to set up the embedded MongoDB instance:

package com.example.studentmanagement.testconfig;

import de.flapdoodle.embed.mongo.MongodExecutable;

import de.flapdoodle.embed.mongo.MongodStarter;

import de.flapdoodle.embed.mongo.config.MongodConfigBuilder;

import de.flapdoodle.embed.mongo.distribution.Version;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration

public class TestMongoConfig {

@Bean(initMethod = "start", destroyMethod = "stop")

public MongodExecutable mongodExecutable() throws Exception {

return MongodStarter.getDefaultInstance()

.prepare(new MongodConfigBuilder()

.version(Version.Main.PRODUCTION)

.net(new de.flapdoodle.embed.mongo.config.Net("localhost", 27017, false))

.build());

}

}

**3. Write End-to-End Integration Tests**

Create an integration test class to test the entire application stack:

package com.example.studentmanagement.integration;

import com.example.studentmanagement.model.Student;

import com.example.studentmanagement.repository.StudentRepository;

import com.example.studentmanagement.service.StudentService;

import org.junit.jupiter.api.AfterEach;

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;

import org.springframework.boot.test.context.SpringBootTest;

import org.springframework.http.MediaType;

import org.springframework.test.web.servlet.MockMvc;

import org.springframework.test.web.servlet.request.MockMvcRequestBuilders;

import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.\*;

@SpringBootTest

@AutoConfigureMockMvc

public class StudentIntegrationTest {

@Autowired

private MockMvc mockMvc;

@Autowired

private StudentRepository studentRepository;

@Autowired

private StudentService studentService;

@BeforeEach

public void setup() {

// Initialize test data or perform any setup needed

}

@AfterEach

public void cleanup() {

// Cleanup after each test

studentRepository.deleteAll();

}

@Test

public void testCreateStudent() throws Exception {

String studentJson = "{ \"firstName\": \"John\", \"lastName\": \"Doe\", \"age\": 25, \"email\": \"john.doe@example.com\" }";

mockMvc.perform(MockMvcRequestBuilders.post("/api/students")

.contentType(MediaType.APPLICATION\_JSON)

.content(studentJson))

.andExpect(status().isCreated());

Student savedStudent = studentService.getStudentByEmail("john.doe@example.com");

assertNotNull(savedStudent);

assertEquals("John", savedStudent.getFirstName());

// Add more assertions as needed

}

// Add more integration test methods for other controller endpoints and service methods

}

In this integration test class:

* We use @SpringBootTest to load the entire Spring application context.
* @AutoConfigureMockMvc is used to auto-configure the MockMvc instance for testing controller endpoints.
* We perform end-to-end testing by making requests to controller endpoints and verifying the behavior against the actual service layer and database interactions.

Adjust the test methods and assertions according to your specific application logic and requirements.

**Running Tests**

You can run these tests using your IDE's test runner or using Maven with the command mvn test.

This approach ensures that your entire application stack, including the interaction with MongoDB, is tested in an integrated manner, providing confidence in the functionality of your Spring Boot application.