Spring MVC CRUD with Example

Last Updated : 23 Jul, 2025

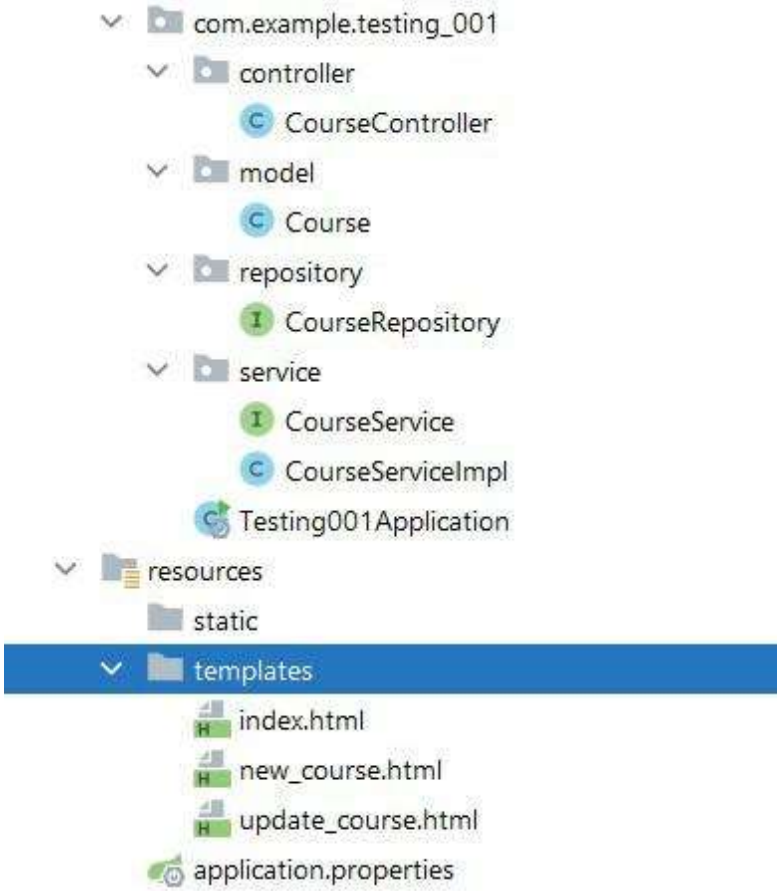
In this article, we will explore how to build a **Spring MVC CRUD application** from scratch. **CRUD** stands for **Create, Read/Retrieve**, **Update,** and **Delete**. These are the four basic operations to create any type of project. Spring MVC is a popular framework for building web applications. Spring MVC follows the **Model-View-Controller architecture**, which separates the application into three main components: The model (data), the view (UI), and the controller (business logic).

CRUD Example of Spring MVC

We will build a simple course-tracking **CRUD** application focused on the **Spring MVC** module.

Project Structure

The **project structure** for the **Spring MVC** application is as follows:



Step 1: Add Dependencies to build.gradle

To get started, create a Spring Boot project and add the following dependencies to your build.gradle file:

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
    implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
    implementation 'org.springframework.boot:spring-boot-starter-web'
    implementation 'org.projectlombok:lombok:1.18.20'
    developmentOnly 'org.springframework.boot:spring-boot-devtools'
    runtimeOnly 'com.h2database:h2'
    runtimeOnly 'mysql:mysql-connector-java'
```

```
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

These dependencies include:

- **Spring Data JPA** for database operations.
- **Thymeleaf** for rendering HTML templates.
- **H2 Database** for in-memory database support (you can switch to MySQL if needed).
- **Lombok** for reducing boilerplate code.

Step 2: Create the Model Layer

The **Model layer** represents the data structure of the application. In this case, we will create a Course entity that maps to a database table.

```
import lombok.*;
import javax.persistence.*;

@NoArgsConstructor
@AllArgsConstructor
@Data
@Entity
@Table(name = "courses")
public class Course {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "course_name")
    private String courseName;

    @Column(name = "instructor")
    private String instructor;

    @Column(name = "email")
    private String email;
}
```

The key annotations include:

- **@NoArgsConstructor:** This annotation generates constructors with no arguments.

- **@AllArgsConstructor:** This annotation generates constructors with all field arguments.
- **@Data:** This annotation generates getters, setter, toString, required argument constructors, equals & hashCode.
- **@Entity:** This annotation defines that a class can be mapped to a table.
- **@Table:** This annotation specifies the name of the database table used for mapping.
- **@Id:** This annotation marks the primary key for the entity.
- **@GeneratedValue:** This annotation provides the specification of generation strategies for the values of the primary keys.
- **@Column:** This annotation marks the column name in the table for that particular attribute.

Step 3: Create the DAO (Data Access Object)/Repository Layer

The Repository layer interacts with the database. We will use Spring Data JPA to perform CRUD operations.

```
import com.example.testing_001.model.Course;
import org.springframework.data.jpa.repository.JpaRepository;

@Repository
public interface CourseRepository extends JpaRepository<Course, Long> {

}
```

The key annotation include:

- **@Repository:** Marks the class as a repository, indicating that it performs database operations.
- **JpaRepository:** JpaRepository is a JPA-specific extension of the Repository. It contains the full API of CrudRepository and PagingAndSortingRepository. So it contains API for basic CRUD operations and also API for pagination and sorting. Here we enable database operations for Employees. To learn more about spring data Jpa, refer this article.

Step 4: Create the Service Layer

The Service layer contains the business logic of the application. It acts as an intermediate between the Controller and Repository layers.

```
import com.example.testing_001.model.Course;
import java.util.List;
import org.springframework.data.domain.Page;

public interface CourseService {
    List<Course> getAllCourses();
    void saveCourse(Course course);
    Course getCourseById(long id);
    void deleteCourseById(long id);
    Page<Course> findPaginated(int pageNum, int pageSize,
                               String sortField,
                               String sortDirection);
}
```

Service Implemetation: CourseServiceImpl

The class CourseServiceImpl implements the CourseService interface and provides us with all the CRUD operations logic which is our business logic here.

```
import com.example.testing_001.model.Course;
import com.example.testing_001.repository.CourseRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Pageable;
import org.springframework.data.domain.Sort;
import org.springframework.stereotype.Service;

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

@Service
public class CourseServiceImpl implements CourseService{

    @Autowired
    private CourseRepository courseRepository;

    @Override
    public List<Course> getAllCourses() {
        return courseRepository.findAll();
    }
```

```
    @Override
    public void saveCourse(Course course) {
        this.courseRepository.save(course);
    }

    @Override
    public Course getCourseById(long id) {
        Optional<Course> optionalCourse = courseRepository.findById(id);
        Course course = null;
        if (optionalCourse.isPresent()) {
            course = optionalCourse.get();
        } else {
            throw new RuntimeException("Course not found for id : " + id);
        }
        return course;
    }

    @Override
    public void deleteCourseById(long id) {
        this.courseRepository.deleteById(id);
    }

    @Override
    public Page<Course> findPaginated(int pageNum, int pageSize, String
sortField, String sortDirection) {
        Sort sort =
sortDirection.equalsIgnoreCase(Sort.Direction.ASC.name()) ?
Sort.by(sortField).ascending() :
                Sort.by(sortField).descending();

        Pageable pageable = PageRequest.of(pageNum - 1, pageSize, sort);
        return this.courseRepository.findAll(pageable);
    }
}
```

The key annotation include:

- **@Service:** This annotation is a stereotype that is used to annotate classes at service layers.
- **@Autowired:** This annotation is used to autowired a bean into another bean.

Step 5: Create the Controller Layer

In Spring MVC, the controller layer is the top layer of the architecture that is used to handle web requests. Then depending on the type of request, it passed it to corresponding layers.

```
import com.example.testing_001.model.Course;
```

```
import com.example.testing_001.service.CourseService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Page;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@Controller
public class CourseController {

    @Autowired
    private CourseService courseService;

    @GetMapping("/")
    public String viewHomePage(Model model) {
        return findPaginated(1, "courseName", "asc", model);
    }

    @GetMapping("/add")
    public String showNewCourseForm(Model model) {
        Course Course = new Course();
        model.addAttribute("course", Course);
        return "new_course";
    }

    @PostMapping("/save")
    public String saveCourse(@ModelAttribute("course") Course course) {
        // save Course to database
        courseService.saveCourse(course);
        return "redirect:/";
    }

    @GetMapping("/update/{id}")
    public String showFormForUpdate(@PathVariable( value = "id") long id,
Model model) {

        Course course = courseService.getCourseById(id);
        model.addAttribute("course", course);
        return "update_course";
    }

    @GetMapping("/delete/{id}")
    public String deleteCourse(@PathVariable (value = "id") long id) {

        this.courseService.deleteCourseById(id);
        return "redirect:/";
    }


    @GetMapping("/page/{pageNo}")
    public String findPaginated(@PathVariable (value = "pageNo") int pageNo,
                                @RequestParam("sortField") String sortField,
                                @RequestParam("sortDir") String sortDir,
                                Model model) {
```

```
        int pageSize = 5;

        Page<Course> page = courseService.findPaginated(pageNo, pageSize,
sortField, sortDir);
        List<Course> listCourses = page.getContent();

        model.addAttribute("currentPage", pageNo);
        model.addAttribute("totalPages", page.getTotalPages());
        model.addAttribute("totalItems", page.getTotalElements());

        model.addAttribute("sortField", sortField);
        model.addAttribute("sortDir", sortDir);
        model.addAttribute("reverseSortDir", sortDir.equals("asc") ? "desc"
: "asc");

        model.addAttribute("listCourses", listCourses);
        return "index";
    }
}
```

The key annotation include:

- **@Controller:** This annotation marks that the particular class serves the role of a controller.
- **@GetMapping:**This annotation marks the mapping of HTTP GET requests onto specific handler methods.
- **@PostMapping:** This annotation marks the mapping of HTTP POST requests onto specific method handlers.
- **@PathVariable:** Binds a method parameter to a URI template variable.
- **@ModelAttribute:** Binds form data to a model object.

Note: Change the GET mapping for course update and delete course to PUT mapping and DELETE mapping respectively. It is considered best practice to use mappings based on the functionality of the api.

Step 6: Thymeleaf Templates: Building the User Interface

We use Thymeleaf as our templating engine to create dynamic HTML pages instead of traditional jsps. Below is an example of the home page template:

Home page:

```
<!DOCTYPE html>
```

```
<!DOCTYPE html>
<html lang="en" xmlns:th="https://www.thymeleaf.org/">
<head>
  <meta charset="ISO-8859-1">
  <title>Course Tracker</title>

  <link rel="stylesheet"
        href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css"
        integrity="sha384-MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm81iuXoPkFOJwJ8ERdknLPMO"
        crossorigin="anonymous">

</head>
<body>

<div class="container my-2">
  <h1>Courses List</h1>

  <a th:href = "@{/add}" class="btn btn-primary btn-sm mb-3"> Add Course
</a>

  <table border="1" class = "table table-striped table-responsive-md">
    <thead>
    <tr>
      <th>
        <a th:href="@{'/page/' + ${currentPage} + '?sortField=courseName&sortDir=' + ${reverseSortDir}}">
          Course Name</a>
      </th>
      <th>
        <a th:href="@{'/page/' + ${currentPage} + '?sortField=instructor&sortDir=' + ${reverseSortDir}}">
          Course Instructor</a>
      </th>
      <th>
        <a th:href="@{'/page/' + ${currentPage} + '?sortField=email&sortDir=' + ${reverseSortDir}}">
          Course Email</a>
      </th>
      <th> Actions </th>
    </tr>
    </thead>
    <tbody>
    <tr th:each="course : ${listCourses}">
      <td th:text="${course.courseName}"></td>
      <td th:text="${course.instructor}"></td>
      <td th:text="${course.email}"></td>
      <td> <a th:href="@{/update/{id}(id=${course.id})}" class="btn btn-primary">Update</a>
        <a th:href="@{/delete/{id}(id=${course.id})}" class="btn btn-danger">Delete</a>
      </td>
    </tr>
    </tbody>
  </table>
```

```
    <div th:if = "${totalPages > 1}">
      <div class = "row col-sm-10">
        <div class = "col-sm-5">
          Total Rows: [[${totalItems}]]
        </div>
        <div class = "col-sm-3">
                        <span th:each="i: ${#numbers.sequence(1, totalPages)}">
                            <a th:if="${currentPage != i}" th:href="@{'/page/' +
${i}+ '?sortField=' + ${sortField} + '&sortDir=' + ${sortDir}}">[[${i}]]</a>
                            <span th:unless="${currentPage != i}">[[${i}]]
</span>     
                        </span>
        </div>
        <div class = "col-sm-1">
          <a th:if="${currentPage < totalPages}" th:href="@{'/page/' +
${currentPage + 1}+ '?sortField=' + ${sortField} + '&sortDir=' +
${sortDir}}">Next</a>
          <span th:unless="${currentPage < totalPages}">Next</span>
        </div>

        <div class="col-sm-1">
          <a th:if="${currentPage < totalPages}" th:href="@{'/page/' +
${totalPages}+ '?sortField=' + ${sortField} + '&sortDir=' +
${sortDir}}">Last</a>
          <span th:unless="${currentPage < totalPages}">Last</span>
        </div>
      </div>
    </div>
</div>
</body>
</html>
```

Add course page:

```
<!DOCTYPE html>
<html lang="en" xmlns:th="https://www.thymeleaf.org/">
<head>
    <meta charset="ISO-8859-1">
    <title>Course </title>
    <link rel="stylesheet"

href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.c
ss"
          integrity="sha384-
MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm81iuXoPkFOJwJ8ERdknLPMO"
          crossorigin="anonymous">
</head>
<body>
<div class="container">
    <h1>Course Tracker</h1>
    <hr>
```

```
    <h2>Save Course</h2>

    <form action="#" th:action="@{/save}" th:object="${course}"
          method="POST">
        <input type="text" th:field="*{courseName}"
               placeholder="Course Name" class="form-control mb-4 col-4">

        <input type="text" th:field="*{instructor}"
               placeholder="Instructor Name" class="form-control mb-4 col-
4">

        <input type="text" th:field="*{email}"
               placeholder="Course Email" class="form-control mb-4 col-4">

        <button type="submit" class="btn btn-info col-2"> Save
Course</button>
    </form>

    <hr>

    <a th:href = "@{/}"> Back to Course List</a>
</div>
</body>
</html>
```

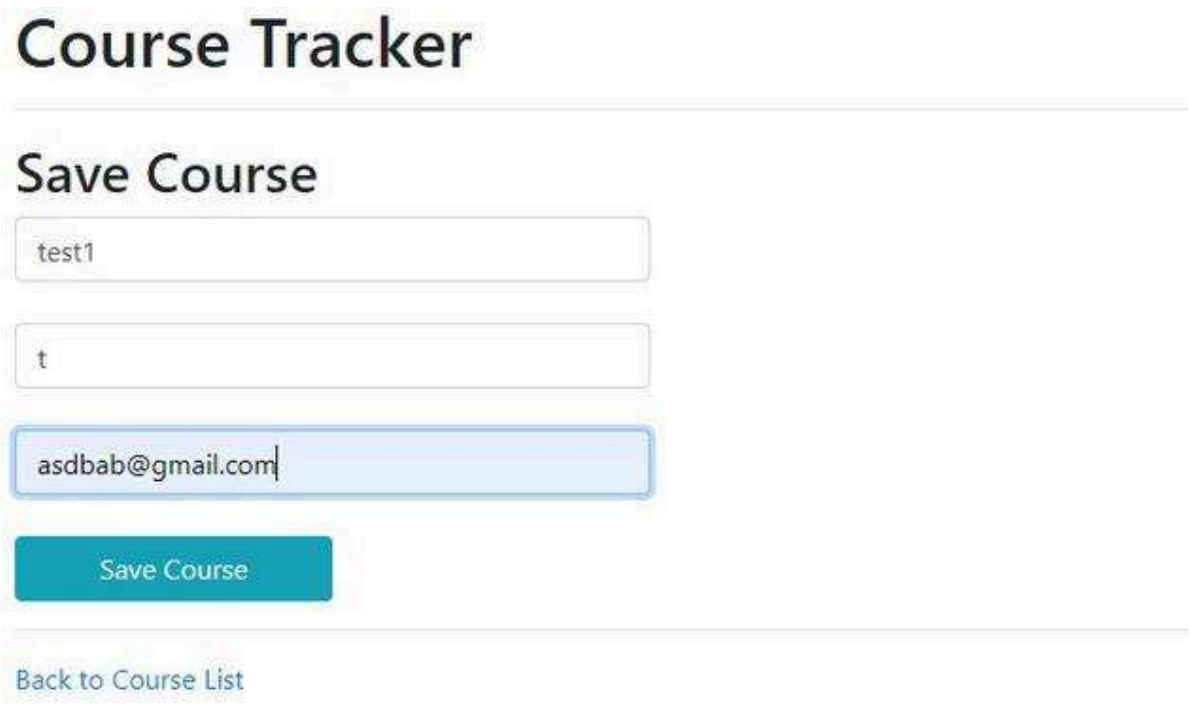
Update course page:

```
<!DOCTYPE html>
<html lang="en" xmlns:th="https://www.thymeleaf.org/">
<head>
    <meta charset="ISO-8859-1">
    <title>Course Tracker</title>

    <link rel="stylesheet"

href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.c
ss">
</head>
<body>
<div class="container">
    <h1>Course Tracker</h1>
    <hr>
    <h2>Update Course</h2>

    <form action="#" th:action="@{/save}" th:object="${course}"
          method="POST">

        <!-- Add hidden form field to handle update -->
        <input type="hidden" th:field="*{id}" />
        <input type="text" th:field="*{courseName}" class="form-control mb-4
```

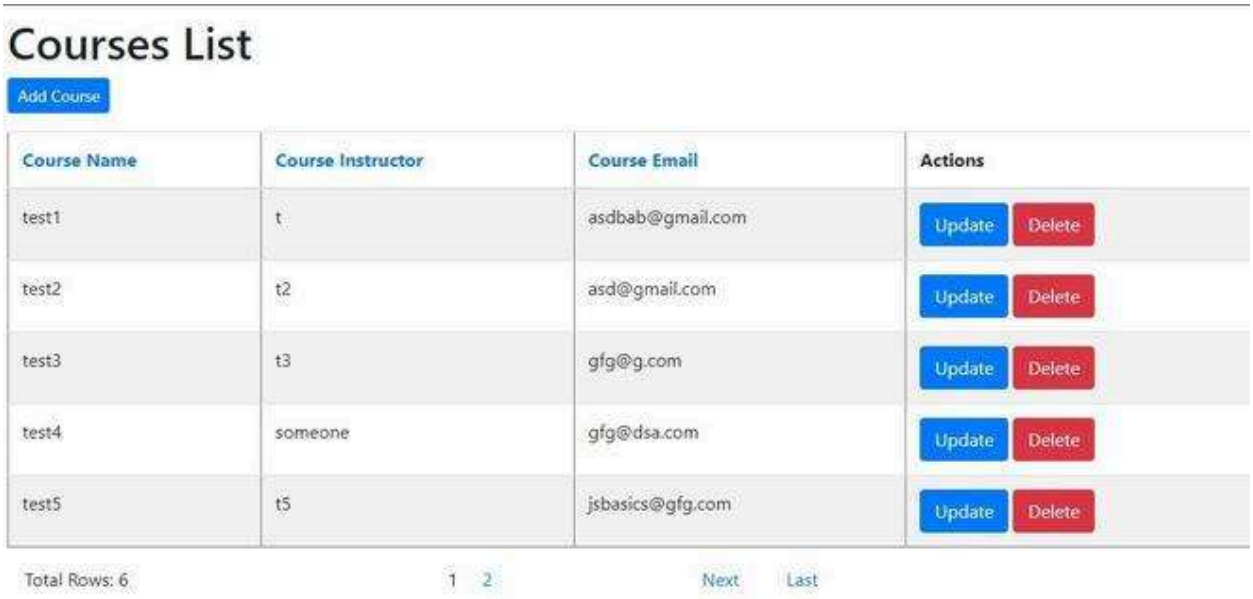
```
col-4" placeholder="Course name">
        <input type="text" th:field="*{instructor}" class="form-control mb-4
col-4" placeholder="Course instructor">
        <input type="text" th:field="*{email}" class="form-control mb-4 col-
4" placeholder="Email">
        <button type="submit" class="btn btn-info col-2"> Update
Course</button>
    </form>

    <hr>

    <a th:href = "@{/}"> Back to Course List</a>
</div>
</body>
</html>
```

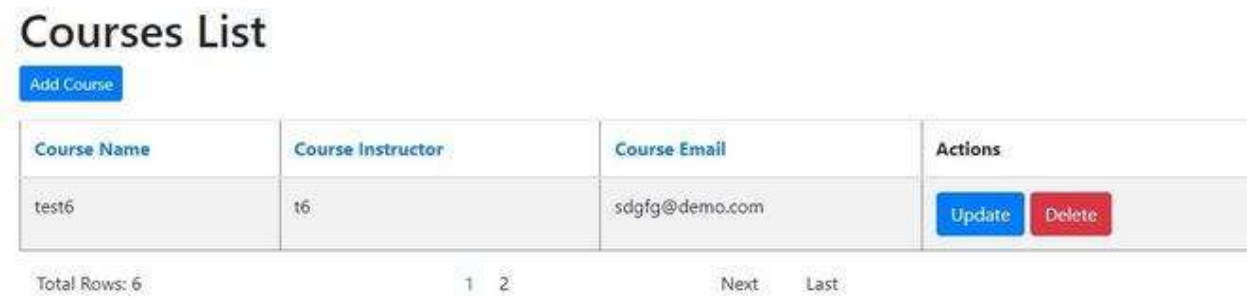
Step 7: Test the CRUD Operations

1. Creating a course:

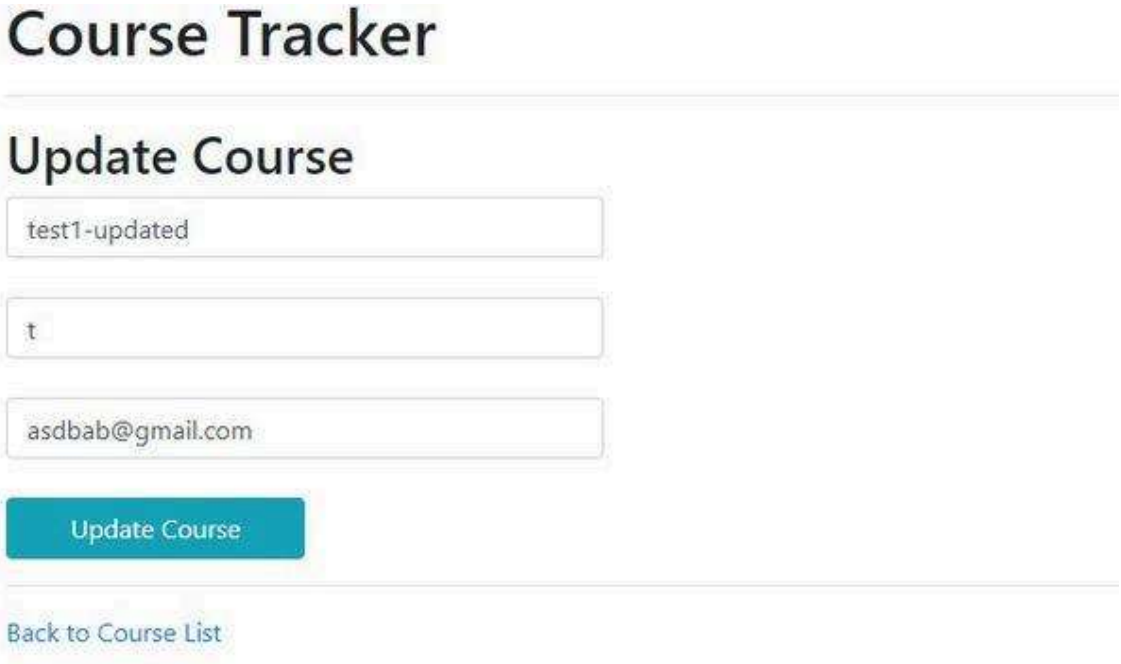


2. Reading courses:





3. Updating course:



4. Course updated successfully:

5. Deleting course:

Comment More info Advertise with us