IN28MINUTES

Creating Spring Boot and React Java Full Stack Application with Maven

in28minutes is providing amazing learning solutions (**100+** articles, **500+** Videos and **30+** Courses) to **85,000+** YouTube Subscribers, **350,000+** Udemy Learners and **1 Million+** SBT Readers. Our solutions help you **master** Full Stack, AWS, Azure, Docker, Kubernetes and Microservices **in 1-3 Months**.

This guide helps you create a Java full stack application with all the CRUD (Create, Read, Update and Delete) features using React as Frontend framework and Spring Boot as the backend REST API. We will be using JavaScript as the frontend language and Java as the backend language.

You will learn

- What is a full stack application?
- Why do we create full stack applications?
- How do you use React as a Frontend Framework?
- How do you use Spring to create Backend REST Service API?
- How do you call Spring Boot REST API from React using the axios framework?
- How and When to use different REST API Request Methods - GET, POST, PUT and DELETE?
- How do you perform CRUD (Create, Read, Update and Delete) operations using React as Frontend framework and Spring Boot as the backend REST API?
- How do you create a form in React using Formik?

Free Courses - Learn in 10 Steps

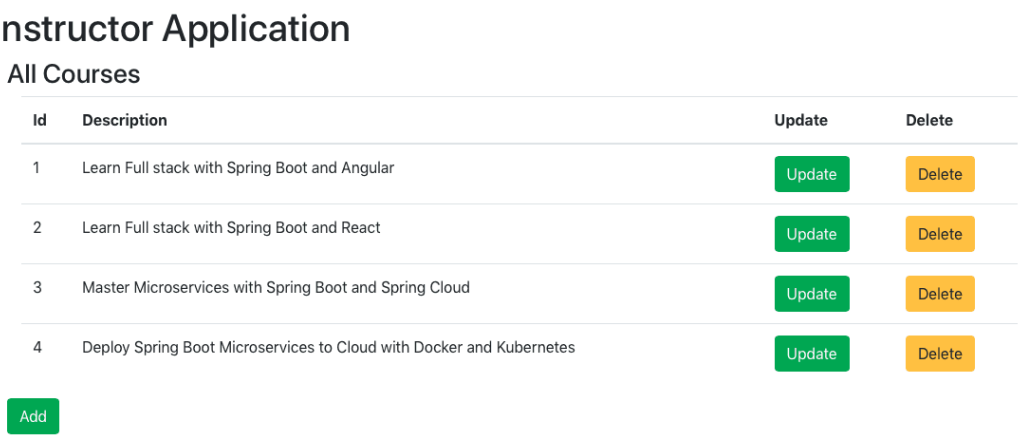
- FREE 5 DAY CHALLENGE - Learn Spring and Spring Boot
- Learn Spring Boot in 10 Steps
- Learn Docker in 10 Steps
- Learn Kubernetes in 10 Steps
- Learn AWS in 10 Steps

Step 0: Get an overview of the Full Stack Application

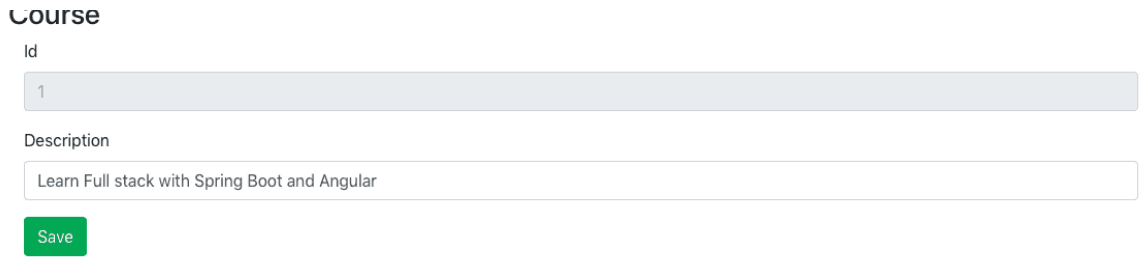
Understanding Basic Features of the Application

Following screenshot shows the application we would like to build:

It is a primary instructor portal allowing instructors to maintain their courses.

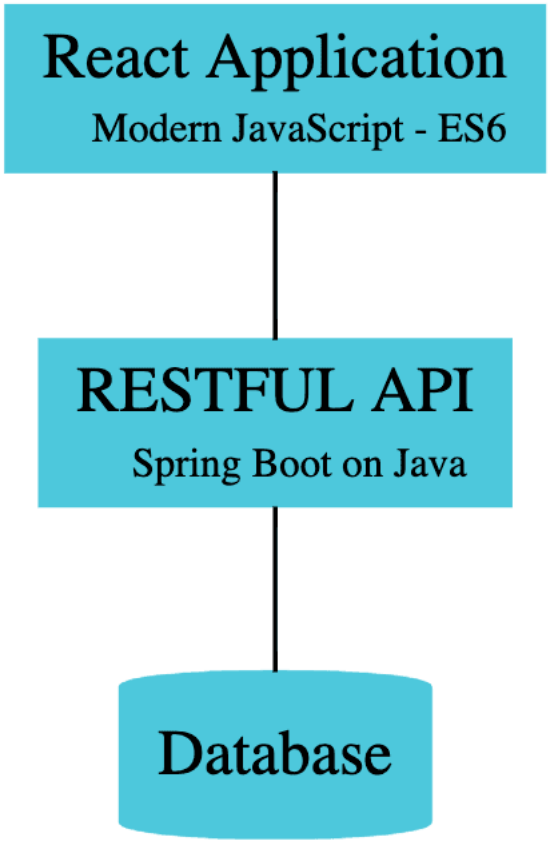


IN28MINUTES



Understanding Full Stack Architecture

Following Screenshot shows the architecture of the application we would create:



Important points to note:

- REST API is exposed using Spring Boot
- REST API is consumed from React Frontend to present the UI
- The Database, in this example, is a hardcoded in-memory static list.

You can find more details about Full Stack Architecture here - Full Stack Application Architecture - Spring Boot and React

Getting an overview of Spring Boot REST API Resources

In this guide, we will create these services using proper URIs and HTTP methods:

- `@GetMapping("/instructors/{username}/courses")` : Get Request Method exposing the list of courses taught by a specific instructor
- `@GetMapping("/instructors/{username}/courses/{id}")` : Get Request Method exposing the details of a specific course taught by a specific instructor
- `@DeleteMapping("/instructors/{username}/courses/{id}")` : Delete Request Method to delete a course belonging to a specific instructor
- `@PutMapping("/instructors/{username}/courses/{id}")` : Put Request Method to update the course details of a specific course taught by a specific instructor
- `@PostMapping("/instructors/{username}/courses")` : Post Request Method to create a new course for a specific instructor

IN28MINUTES

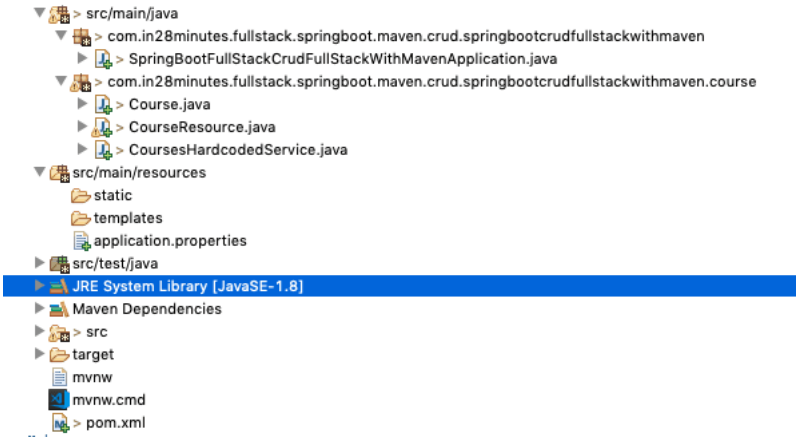
Downloading the Complete Maven Project With Code Examples

Following GitHub repository hosts the complete frontend and backend projects - https://github.com/in28minutes/spring-boot-react-fullstack-examples/tree/master/spring-boot-react-crud-full-stack-with-maven

> *Our Github repository has all the code examples - https://github.com/in28minutes/spring-boot-react-examples/*

Understanding Spring Boot REST API Project Structure

Following screenshot shows the structure of the Spring Boot project we create.

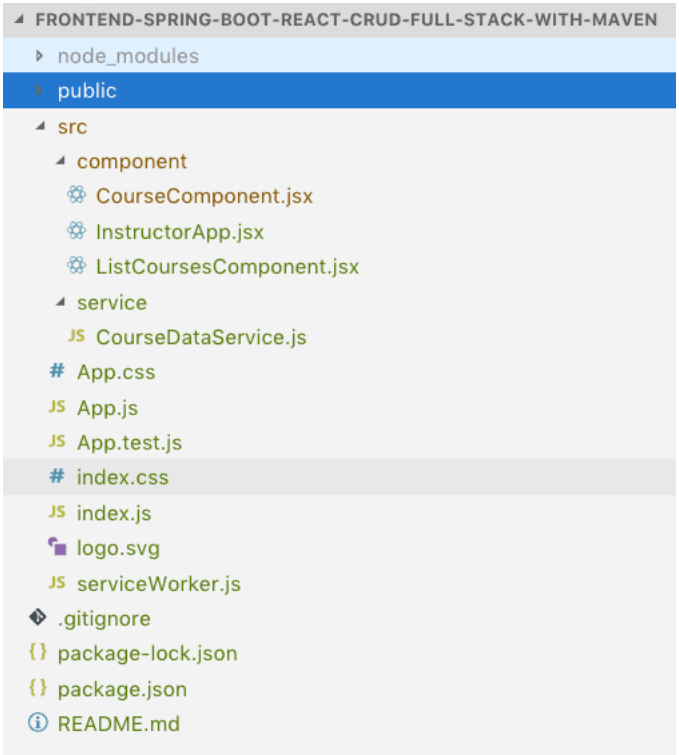


A few details:

- `CourseResource.java` - Rest Resource exposing all the service methods discussed above.
- `Course.java, CoursesHardcodedService.java` - Business Logic for the application. CoursesHardcodedService exposes a few methods we would invoke from our Rest Resource.
- `SpringBootFullStackCrudFullStackWithMavenApplication.java` - Launcher for the Spring Boot Application. To run the application, launch this file as Java Application.
- `pom.xml` - Contains all the dependencies needed to build this project. We use Spring Boot Starter Web and Spring Boot DevTools.

Understanding React Frontend Project Structure

Following screenshot shows the structure of the React project we create.



> *Quick Tip: You can get a high-level overview of all files in the React Project Structure watching this video React Project Structure*

file.
- `ListCoursesComponent.jsx` - React Component for listing all the courses for an instructor.
- `CourseComponent.jsx` - React Component for editing Course Details and creating a new course
- `CourseDataService.js` - Service using axios framework to make the Backend REST API Calls.

Understanding the tools you need to build this project

- Maven 3.0+ for building Spring Boot API Project
- npm, webpack for building frontend
- Your favorite IDE. We use Eclipse for Java and Visual Studio Code for Frontend - JavaScript, TypeScript, Angular and React.
- JDK 1.8+
- Node v8+
- Embedded Tomcat, built into Spring Boot Starter Web

Installing Node Js (npm) & Visual Studio Code

- Playlist - [Click to see video Playlist](https://www.youtube.com/playlist?list=PLBBog2r6uMCQN4X3Aa_jM9qVjgMCHMWx6
- Steps){:target="_blank"}
    - Step 01 - Installing NodeJs and NPM - Node Package Manager
    - Step 02 - Quick Introduction to NPM
    - Step 03 - Installing Visual Studio Code - Front End JavaScript Editor

Installing Java, Eclipse & Embedded Maven

- Playlist - [Click to see video Playlist](https://www.youtube.com/playlist?list=PLBBog2r6uMCSmMVTW_QmDLyASBvovyAO3
- Steps){:target="_blank"}
    - 0 - Overview - Installation Java, Eclipse and Maven
    - 1 - Installing Java JDK
    - 2 - Installing Eclipse IDE
    - 3 - Using Embedded Maven in Eclipse
    - 4 - Troubleshooting Java, Eclipse and Maven

Creating Full Stack CRUD application with React and Spring Boot - Step By Step Approach

We will use a step by step approach to creating the full stack application

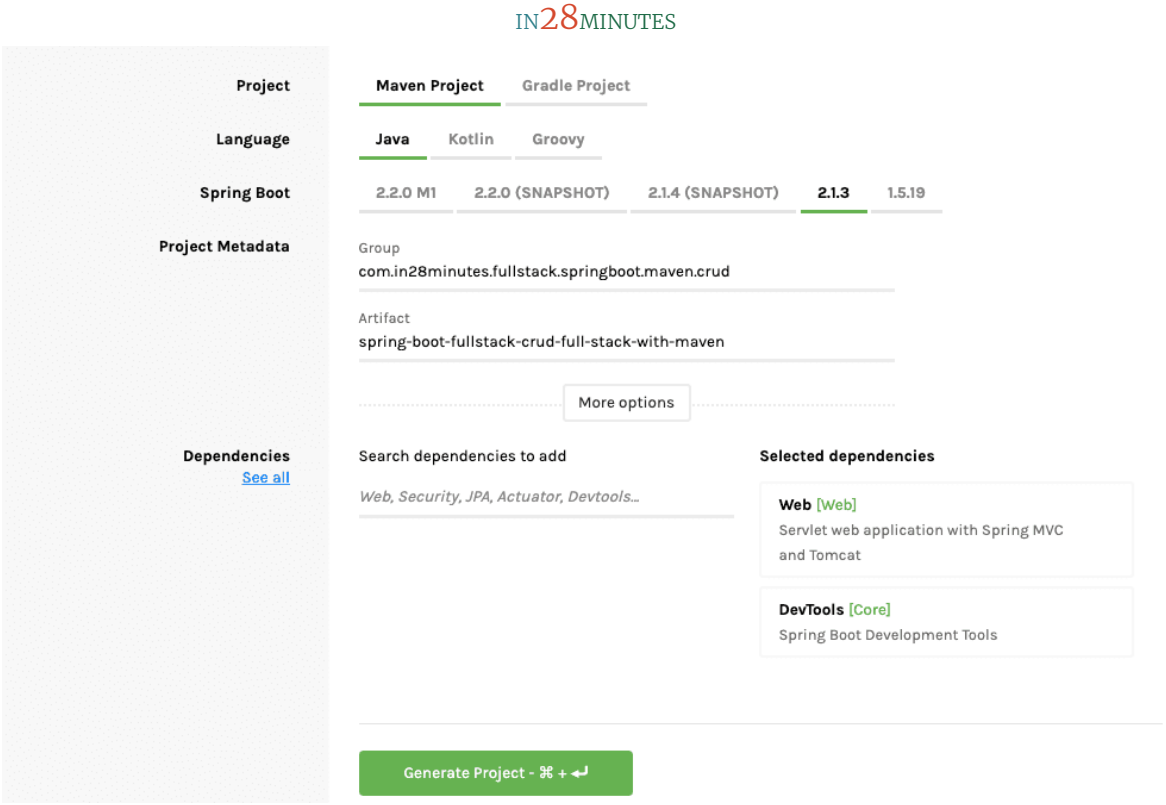
- Create a Spring Boot Application with Spring Boot Initializr
- Create a React application using Create React App
- Create the Retrieve Courses REST API and Enhance the React Front end to retrieve the courses using the axios framework
- Add feature to delete a course in React front end and Spring Boot REST API
- Add functionality to update course details in React front end and Spring Boot REST API
- Add feature to create a course in React front end and Spring Boot REST API

> *You can get an introduction to REST down here - Introduction to REST API*

Step 1: Bootstrapping Spring Boot REST API with Spring Initializr

Creating a REST service with Spring Initializr is a cake walk. We will use Spring Web MVC as our web framework.

Spring Initializr http://start.spring.io/ is great tool to bootstrap your Spring Boot projects.



As shown in the image above, the following steps have to be done

- Launch Spring Initializr and choose the following
  - Choose `com.in28minutes.fullstack.springboot.maven.crud` as Group
  - Choose `spring-boot-fullstack-crud-full-stack-with-maven` as Artifact
  - Choose following dependencies
    - Web
    - DevTools
- Click Generate Project.
- Import the project into Eclipse. File -> Import -> Existing Maven Project. For more details about creating Spring Boot Projects, you can read - Creating Spring Boot Projects

> *If you are new to Spring Boot, we recommend watching this video - Spring Boot in 10 Steps*

Step 2 - Bootstrapping React Frontend with Create React App

Create React App is an amazing tool to bootstrap your React applications.

Creating React Frontend Applications with Create React App is very simple.

Launch up your terminal/command prompt. Make sure that you have node installed.

```
npx create-react-app frontend-spring-boot-react-crud-full-stack-with-maven
```

> *You can find more information about using Create React App here - Create React App - Create and Launch a React Application*

> *You can get a high-level overview of all files in the React Project Structure here - React Project Structure*

Launching up React Frontend

You would need to cd to the project we created and execute `npm start`

```
cd frontend-spring-boot-react-crud-full-stack-with-maven
npm start
```

You would see the screen below:

```
You can now view spring-boot-react-crud-full-stack-with-maven in the browser.

  Local:            http://localhost:3000/
  On Your Network:  http://192.168.1.3:3000/

Note that the development build is not optimized.
To create a production build, use npm run build.
```

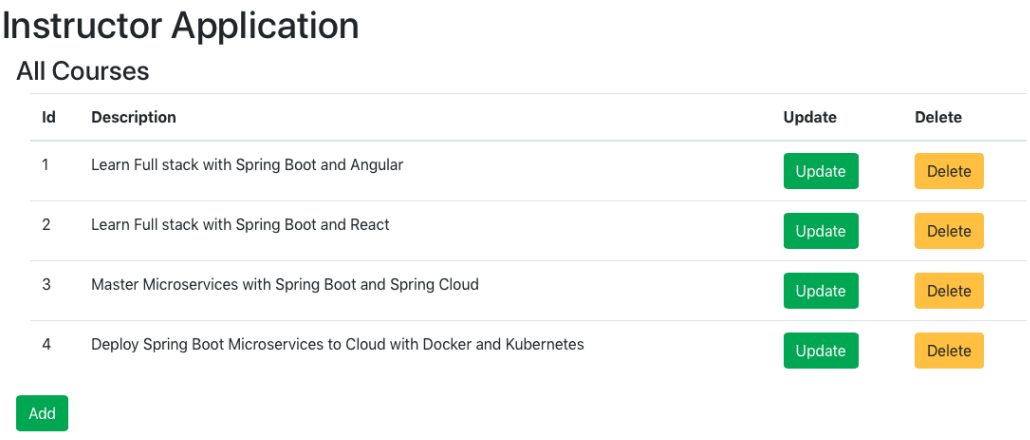
When you launch up the application in the browser at `http://localhost:3000/`, you would see the following welcome screen.

You can import the created project into Visual Studio Code by using `File > Open > Select the project created earlier`. You can find more details here - Importing React App into Visual Studio Code

> *Cool! You are all set to rock and roll with React.*

Step 3 - Creating REST API for Retrieve All Courses and Connecting React Frontend

We would want to start building the screen shown below:



Let's start with building the course listing screen.

To be able to do that, we need to

- Create REST API for retrieving a list of courses.
- Connect the React Frontend to the backend REST API

Create REST API for retrieving a list of courses

Web Services, REST and Designing REST API, are pretty deep concepts. We would recommend to check this out for more - Designing Great REST API

We will create

- A model object `Course.java`

We will start with creating a model object `Course.java`. The snippet below shows the content of the model class. For the complete listing, refer `course/Course.java` in the complete code example at the end of this article.

```
public class Course {
  private Long id;
  private String username;
  private String description;
  //no arg constructor
  //constructor with 3 args
  //getters and setters
  //hashcode and equals
```

Next, let's create a Business Service. In this article, we will use hardcoded data.

```
@Service
public class CoursesHardcodedService {

  private static List<Course> courses = new ArrayList<>();
  private static long idCounter = 0;

  static {
    courses.add(new Course(++idCounter, "in28minutes", "Learn Full stack with Spring Boot and Angular"));
    courses.add(new Course(++idCounter, "in28minutes", "Learn Full stack with Spring Boot and React"));
    courses.add(new Course(++idCounter, "in28minutes", "Master Microservices with Spring Boot and Spring Cloud")
    courses.add(new Course(++idCounter, "in28minutes",
        "Deploy Spring Boot Microservices to Cloud with Docker and Kubernetes"));
  }

  public List<Course> findAll() {
    return courses;
  }
}
```

Few things to note

- Data is hardcoded
- findAll returns the complete list of courses
- You can see that the API of the Service is modelled around the Spring Data Repository interfaces. If you are familiar with JPA and Spring Data, you can easily replace this with a Service talking to a database.

Next, let create the REST Resource to retrieve the list of courses for an instructor.

```
@RestController
public class CourseResource {

  @Autowired
  private CoursesHardcodedService courseManagementService;

  @GetMapping("/instructors/{username}/courses")
  public List<Course> getAllCourses(@PathVariable String username) {
    return courseManagementService.findAll();
  }
}
```

Few things to note:

- `@RestController : Combination of @Controller and @ResponseBody` – Beans returned are converted to/from JSON/XML.
- `@Autowired private CoursesHardcodedService courseManagementService` – Autowire the CoursesHardcodedService so that we can retrieve details from business service.

If you launch up the Spring boot application and go to `http://localhost:8080/instructors/in28minutes/courses` in the browser, you would see the response from the API.

```
[
   {
      "id": 1,
      "username": "in28minutes",
      "description": "Learn Full stack with Spring Boot and Angular"
   },
   {
      "id": 2,
      "username": "in28minutes",
      "description": "Learn Full stack with Spring Boot and React"
   },
   {
      "id": 3,
      "username": "in28minutes",
      "description": "Master Microservices with Spring Boot and Spring Cloud"
   },
   {
      "id": 4,
      "username": "in28minutes",
      "description": "Deploy Spring Boot Microservices to Cloud with Docker and Kubernetes"
   }
]
```

IN28MINUTES

Enhancing React App to consume the REST API

To be able to enhance the React Application to consume the REST API, we would need to

- Create an Application Component - to represent the structure of the complete application and include it in `App.jsx` - `InstructorApp.jsx`
- Add the frameworks need to call the REST API - axios, display a form - formik and support routing - react-router-dom
- Create a view component for showing a list of course details and include it in the Application Component - `ListCoursesComponent.jsx`
- Invoking Retrieve Courses REST API from React Component - To enable this we will create a service to call the REST API using the axios framework - `CourseDataService.js`. `ListCoursesComponent.jsx` will make use of `CourseDataService.js`

Let's start with creating an Application Component - `InstructorApp.jsx`

/src/component/InstructorApp.jsx

```
import React, { Component } from 'react';

class InstructorApp extends Component {
    render() {
        return (
              <h1>Instructor Application</h1>
        )
    }
}

export default InstructorApp
```

Few things to note:

- One of the first things you would need to understand about React is the concept of the component. You can find more about a react component here - React Components
- `class InstructorApp extends Component` - Every React Class Component should extend a class called Component.
- `render()` - The render() method of a component returns what needs to be displayed as part of the component
- `export default InstructorApp` - Each JavaScript file is a module. If you wanted elements from a JavaScript module to be used in other JavaScript modules, we would need to export them. Here, we are making `InstructorApp` available for import in other components.

Let's update the `App.js` to display the InstructorApp component.

src/App.js

```
import React, { Component } from 'react';
import './App.css';
import InstructorApp from './component/InstructorApp';

class App extends Component {
  render() {
    return (
      <div className="container">
        <InstructorApp />
      </div>
    );
  }
}

export default App;
```

Few things to note:

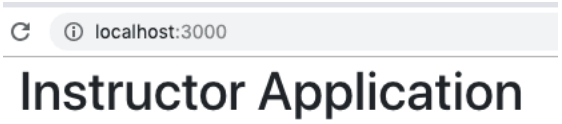
- `import InstructorApp from './component/InstructorApp'` - Importing the InstructorApp component class
- `<InstructorApp />` - Display the Instructor App component.

Let's update the App.css to use Bootstrap framework:

/src/App.css

```
@import url(https://unpkg.com/bootstrap@4.1.0/dist/css/bootstrap.min.css)
```

When you launch the React app in the browser, it will appear as shown below:



Adding Frameworks to React Application

In this project, we will make use of axios to execute REST APIs, react-router-dom to do the Routing between pages and formik to create forms. Let's stop the front end app running in the command prompt and execute these commands.

```
npm add axios
```

```
npm add formik
```

When commands execute successfully, you would see new entries in `package.json`

```
    "axios": "^0.18.0",
    "formik": "^1.5.1",
    "react-router-dom": "^5.0.0",
```

You can run 'npm start' to relaunch the front end app loading up all the new frameworks.

Creating a List Courses Component

Let's create a new component for showing the List of courses - `ListCoursesComponent.jsx`. For now, let's hardcode a course into the course list.

/src/component/ListCoursesComponent.jsx

```
class ListCoursesComponent extends Component {

    render() {
        return (
            <div className="container">
                <h3>All Courses</h3>
                <div className="container">
                    <table className="table">
                        <thead>
                            <tr>
                                <th>Id</th>
                                <th>Description</th>
                            </tr>
                        </thead>
                        <tbody>
                            <tr>
                                <td>1</td>
                                <td>Learn Full stack with Spring Boot and Angular</td>
                            </tr>
                        </tbody>
                    </table>
                </div>
            </div>
        )
    }
}

export default ListCoursesComponent
```

Things to Note:

- It's a simple component. Returning a hardcoded table displaying a list of courses.

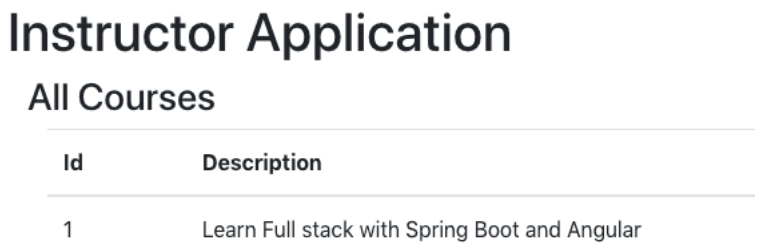
Let's update the InstructorApp component to display the ListCoursesComponent.

/src/component/InstructorApp.jsx

```
class InstructorApp extends Component {
    render() {
        return (<>
              <h1>Instructor Application</h1>
              <ListCoursesComponent/>
            </>
        )
    }
}

export default InstructorApp
```

We are importing the ListCoursesComponent and displaying it in the InstructorApp.

When you launch the React app in the browser, it will appear as shown below:



called axios.

> *Axios is a Promise based HTTP client for the browser and node.js*

Axios is a frontend framework that helps you make

- REST API calls with different request methods including GET, POST, PUT, DELETE etc
- Intercept Front end REST API calls and add headers and request content

Let's create a data service method to call the REST API.

/src/service/CourseDataService.js

```
import axios from 'axios'

const INSTRUCTOR = 'in28minutes'
const COURSE_API_URL = 'http://localhost:8080'
const INSTRUCTOR_API_URL = `${COURSE_API_URL}/instructors/${INSTRUCTOR}`

class CourseDataService {

    retrieveAllCourses(name) {
        return axios.get(`${INSTRUCTOR_API_URL}/courses`);
    }
}

export default new CourseDataService()
```

Important points to note:

- `const INSTRUCTOR_API_URL = `${COURSE_API_URL}/instructors/${INSTRUCTOR}`` - We are forming the URL to call in a reusable way.
- `axios.get(`${INSTRUCTOR_API_URL}/courses`)` - Call the REST API with the GET method.
- `export default new CourseDataService()` - We are creating an instance of CourseDataService and making it available for other components.

To make the REST API call, we would need to call the CourseDataService - retrieveAllCourses method from the ListCoursesComponent

Important snippets are shown below:

```
class ListCoursesComponent extends Component {

    constructor(props) {
        super(props)
        this.refreshCourses = this.refreshCourses.bind(this)
    }

    componentDidMount() {
        this.refreshCourses();
    }

    refreshCourses() {
        CourseDataService.retrieveAllCourses(INSTRUCTOR)//HARDCODED
            .then(
                response => {
                    console.log(response);
                }
            )
    }
    ....
}
```

Things to note:

- `componentDidMount()` - React defines a component lifecycle. componentDidMount will be called as soon as the component is mounted. We are calling refreshCourses as soon as a component is mounted.
- `this.refreshCourses = this.refreshCourses.bind(this)` - Any method in a react component should be bound to this. -`CourseDataService.retrieveAllCourses(INSTRUCTOR).then` - This would make the call to the REST API. You can define how to process the response in the then method.

When you run the react app in the browser right now, you would see the following errors in the console

```
[Error] Origin http://localhost:3000 is not allowed by Access-Control-Allow-Origin.
[Error] XMLHttpRequest cannot load http://localhost:8080/instructors/in28minutes/courses due to access control c
[Error] Failed to load resource: Origin http://localhost:3000 is not allowed by Access-Control-Allow-Origin. (co
[Error] Unhandled Promise Rejection: Error: Network Error
  (anonymous function) (0.chunk.js:1097)
  promiseReactionJob
```

The Backend Spring Boot REST API is running on http://localhost:8080, and it is not allowing requests from other servers - http://localhost:3000, in this example.

```
@RestController
public class CourseResource {
```

An important thing to note

- @CrossOrigin(origins = { "http://localhost:3000", "http://localhost:4200" }) - Allow requests from specific origins
- We will use 3000 to run React and Vue JS apps, and we use 4200 to run Angular apps. Hence we are allowing requests from both ports.

If you refresh the page again, you would see the response from server printed in the console.

We would need to use the data from the response and show it on the component.

> *In React, we use the state to do that.*

Following snippet highlights the significant changes

```
class ListCoursesComponent extends Component {

    constructor(props) {
        super(props)
        this.state = {
            courses: [],
            message: null
        }
        this.refreshCourses = this.refreshCourses.bind(this)
    }

    componentDidMount() {
        this.refreshCourses();
    }

    refreshCourses() {
        CourseDataService.retrieveAllCourses(INSTRUCTOR)//HARDCODED
            .then(
                response => {
                    console.log(response);
                    this.setState({ courses: response.data })
                }
            )
    }
    ....
}
```

Important things to note:

- `this.state = {courses: [],message: null}` - To display courses, we need to make them available to the component. We add courses to the state of the component and initialize it in the constructor.
- `response => {this.setState({ courses: response.data })}` - When the response comes back with data, we update the state.

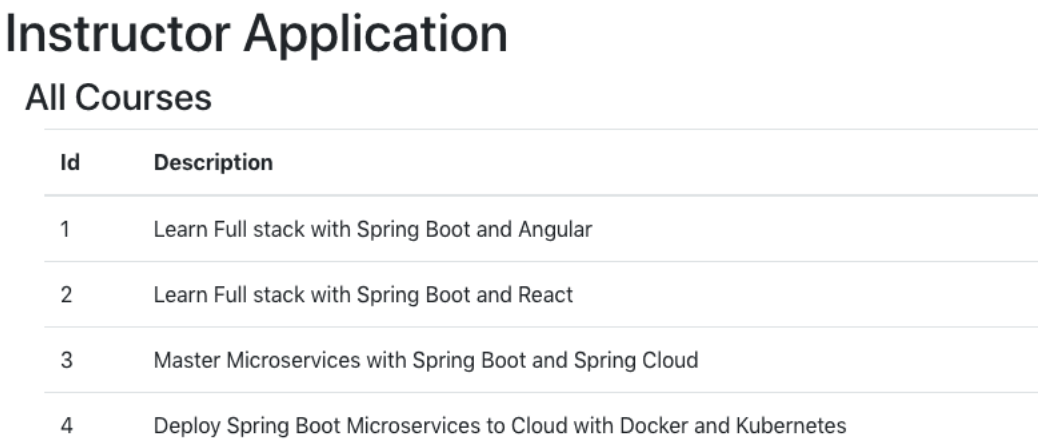
We have data in the state. How do we display it?

We need to update the render method.

```
    render() {
        return (
            <div className="container">
                <h3>All Courses</h3>
                <div className="container">
                    <table className="table">
                        <thead>
                            <tr>
                                <th>Id</th>
                                <th>Description</th>
                            </tr>
                        </thead>
                        <tbody>
                            {
                                this.state.courses.map(
                                    course =>
                                        <tr key={course.id}>
                                            <td>{course.id}</td>
                                            <td>{course.description}</td>
                                        </tr>
                                )
                            }
                        </tbody>
                    </table>
                </div>
            </div>
        )
    }
```

- `key={course.id}` - A key is used to uniquely identify a row.
- `{course.id}` - In JSX, we use {} to execute JavaScript code.

When you launch the React app in the browser, it will appear as shown below:



> *Congratulations! You have successfully integrated React with a REST API. Time to celebrate!*

Step 4: Adding Delete Feature to List Courses Page

To be able to do this

- We need a REST API in Spring Boot Backend for deleting a course
- We would need to update React frontend to use the API

Adding Delete Method in the Backend REST API

It should be easy.

Snippets below show how we create a simple deleteById method in CoursesHardcodedService and expose it from CourseResource.

```
@Service
public class CoursesHardcodedService {

  public Course deleteById(long id) {
    Course course = findById(id);

    if (course == null)
      return null;

    if (courses.remove(course)) {
      return course;
    }

    return null;
  }
```

```
public class CourseResource {

  @DeleteMapping("/instructors/{username}/courses/{id}")
  public ResponseEntity<Void> deleteCourse(@PathVariable String username, @PathVariable long id) {

    Course course = courseManagementService.deleteById(id);

    if (course != null) {
      return ResponseEntity.noContent().build();
    }

    return ResponseEntity.notFound().build();
  }
```

Important things to note:

- `@DeleteMapping("/instructors/{username}/courses/{id}")` - We are mapping the Delete Request Method with two path variables
- `@PathVariable String username, @PathVariable long id` - Defining the variables for Path Variables
- `ResponseEntity.noContent().build()` - If Request is successful, return no content back
- `ResponseEntity.notFound().build()` - If delete failed, return error - resource not found.

```
deleteCourse(name, id) {
    //console.log('executed service')
    return axios.delete(`${INSTRUCTOR_API_URL}/courses/${id}`);
}
```

We can add a delete button corresponding to each of the courses:

```
<td><button className="btn btn-warning" onClick={() => this.deleteCourseClicked(course.id)}>Delete</button></td>
```

On click of the button we are calling the `deleteCourseClicked` method passing the course `id`. The implementation for `deleteCourseClicked` is shown below:

When we get a successful response for delete API call, we set a `message` into state and refresh the courses list.

```
deleteCourseClicked(id) {
    CourseDataService.deleteCourse(INSTRUCTOR, id)
        .then(
            response => {
                this.setState({ message: `Delete of course ${id} Successful` })
                this.refreshCourses()
            }
        )

}
```

We display the message just below the header

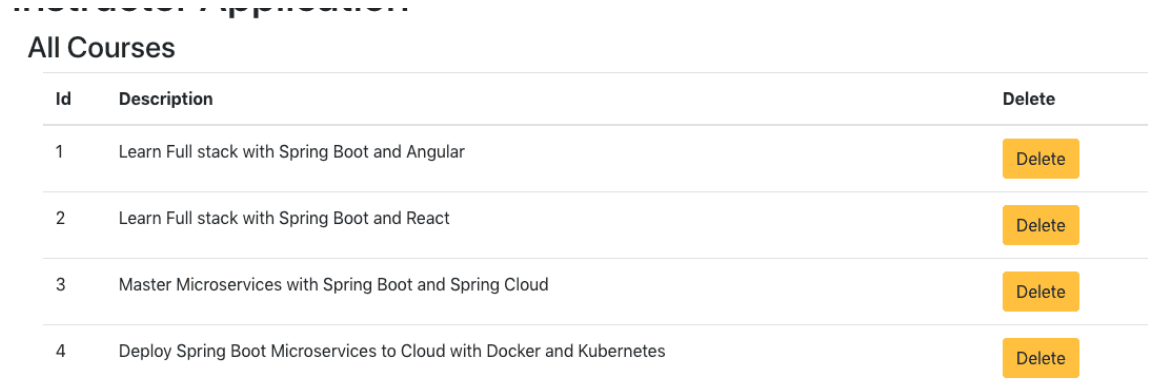
```
<h3>All Courses</h3>
{this.state.message && <div class="alert alert-success">{this.state.message}</div>}
```

Of course - we have to ensure that the method is bound to `this` in the constructor.

```
this.deleteCourseClicked = this.deleteCourseClicked.bind(this)
```

Complete `ListCoursesComponent`, at this stage, is shown below:

```
class ListCoursesComponent extends Component {
    constructor(props) {
        super(props)
        this.state = {
            courses: [],
            message: null
        }

    }

    render() {
        console.log('render')
        return (
            <div className="container">
                <h3>All Courses</h3>
                {this.state.message && <div class="alert alert-success">{this.state.message}</div>}
                <div className="container">
                    <table className="table">
                        <thead>
                            <tr>
                                <th>Id</th>
                                <th>Description</th>
                                <th>Delete</th>
                            </tr>
                        </thead>
                        <tbody>
                            {
                                this.state.courses.map(
                                    course =>
                                        <tr key={course.id}>
                                            <td>{course.id}</td>
                                            <td>{course.description}</td>
                                            <td><button className="btn btn-warning" onClick={() => this.deleteCo
                                        </tr>
                                )
                            }
                        </tbody>
                    </table>
                </div>
            </div>
        )
    }
```



When you click the delete button, the course will be deleted.

Updating Course Details

To be able to update the course details, we would need to create a new component to represent the todo form.

Let's start with creating a simple component.

/src/component/CourseComponent.jsx

```
class CourseComponent extends Component {
      render() {
        return (
              <h1>Course Details</h1>
        )
    }

}

export default CourseComponent
```

Implementing Routing

When the user clicks the update course button on the course listing page, we would want to route to the course page. How do we do it? That's where Routing comes into the picture.

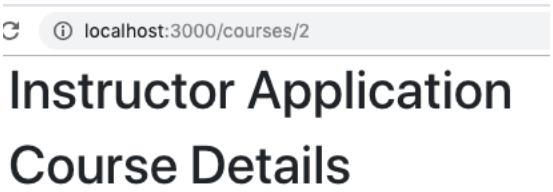
/src/component/InstructorApp.jsx

```
class InstructorApp extends Component {
    render() {
        return (
            <Router>
                <>
                    <h1>Instructor Application</h1>
                    <Switch>
                        <Route path="/" exact component={ListCoursesComponent} />
                        <Route path="/courses" exact component={ListCoursesComponent} />
                        <Route path="/courses/:id" component={CourseComponent} />
                    </Switch>
                </>
            </Router>
        )
    }
}

export default InstructorApp
```

We are defining a Router around all the components and configuring paths to each of them.

- `http://localhost:3000/` takes you to home page
- `http://localhost:3000/courses` takes you to course listing page
- `http://localhost:3000/courses/2` takes you to course page

When you launch the React app in the browser using this URL `http://localhost:3000/courses/2`, it will appear as shown below:



/src/component/ListCoursesComponent.jsx

```
.....

<th>Update</th>

....
<td><button className="btn btn-success" onClick={() => this.updateCourseClicked(course.id)}>Update</button></td>
```

We can create the add `updateCourseClicked` method to redirect to Course Component and add the binding in the constructor method.

```
...
this.updateCourseClicked = this.updateCourseClicked.bind(this)

...
updateCourseClicked(id) {
    console.log('update ' + id)
    this.props.history.push(`/courses/${id}`)
}
```

Adding Add button to Course Listing Page

Let's add an Add button at the bottom of Course Listing Page.

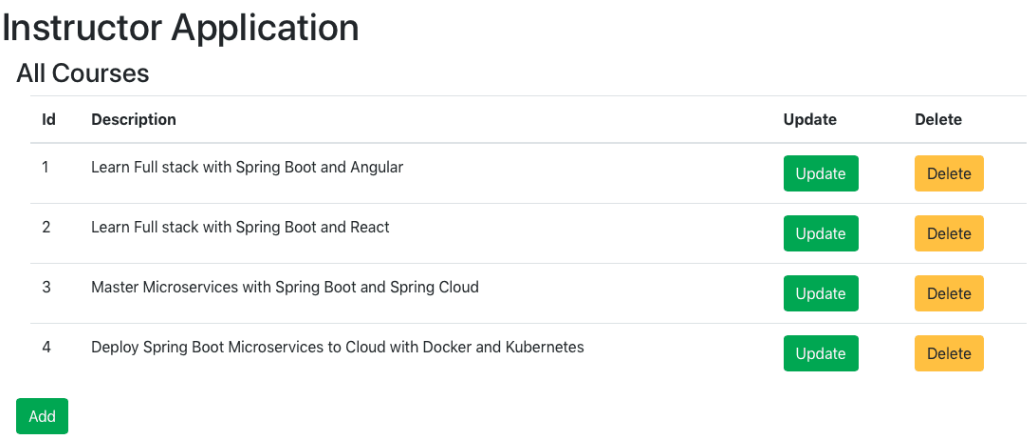
/src/component/ListCoursesComponent.jsx

```
<div className="row">
    <button className="btn btn-success" onClick={this.addCourseClicked}>Add</button>
</div>
```

Let's add the appropriate binding and the method to handle click of Add button.

```
//In constructor
this.addCourseClicked = this.addCourseClicked.bind(this)


addCourseClicked() {
    this.props.history.push(`/courses/-1`)
}
```

When you launch the React app in the browser using this URL `http://localhost:3000`, it will appear as shown below:



Clicking any of the Update or Add buttons would take you to the Course Component.

Create API to Retrieve Specific Course Details

Now that we have the course component beinging rendered on the click of update button, let's start focusing on getting the course details from the REST API.

Let's add `findById` method to `CoursesHardcodedService`. It retrieves the details of a specific course based on it.

```
public Course findById(long id) {
  for (Course course: courses) {
    if (course.getId() == id) {
      return course;
    }
  }
  return null;
}
```

```
@GetMapping("/instructors/{username}/courses/{id}")
public Course getCourse(@PathVariable String username, @PathVariable long id) {
  return courseManagementService.findById(id);
}
```

Invoking the API from Course Component

> *How do we invoke the retrieve course details from the React frontend?*

Let's add `retrieveCourse` method to `CourseDataService`

```
retrieveCourse(name, id) {
    return axios.get(`${INSTRUCTOR_API_URL}/courses/${id}`);
}
```

We would want to call the retrieveCourse method in CourseDataService on the load of CourseComponent.

> *How do we do it?*

> *Yes. componentDidMount is the right solution.*

Before we get to it we would need to be able to get the course id from the URL. In the course details page, we are redirecting to the url `/courses/${id}`. From the path parameter, we would want to capture the id. We can use `this.props.match.params.id` to get the id from path parameters.

The code listing below shows the updated CourseComponent.

```
import React, { Component } from 'react'
import { Formik, Form, Field, ErrorMessage } from 'formik';
import CourseDataService from '../service/CourseDataService';

const INSTRUCTOR = 'in28minutes'

class CourseComponent extends Component {

    constructor(props) {
        super(props)

        this.state = {
            id: this.props.match.params.id,
            description: ''
        }

    }

    componentDidMount() {

        console.log(this.state.id)

        // eslint-disable-next-line
        if (this.state.id == -1) {
            return
        }

        CourseDataService.retrieveCourse(INSTRUCTOR, this.state.id)
            .then(response => this.setState({
                description: response.data.description
            }))
    }

    render() {

        let { description, id } = this.state

        return (
            <div>
                <h3>Course</h3>
                <div>{id}</div>
                <div>{description}</div>
            </div>
        )
    }
}

export default CourseComponent
```

We are setting the details of the course into state.

We are initializing state in the constructor.

```
this.state = {
    id: this.props.match.params.id,
```

In `componentDidMount`, we are calling the `CourseDataService.retrieveCourse` to get the details for a course. Once we have the details, we are updating the state.

```
CourseDataService.retrieveCourse(INSTRUCTOR, this.state.id)
    .then(response => this.setState({
        description: response.data.description
    }))
```

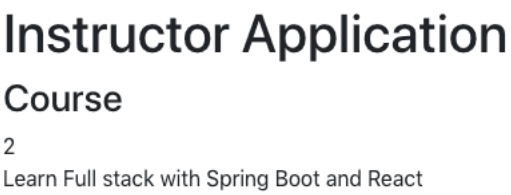
We are updating the `render` method to show the course details from component `state`.

```
render() {

    let { description, id } = this.state

    return (
        <div>
            <h3>Course</h3>
            <div>{id}</div>
            <div>{description}</div>
        </div>
    )
}
```

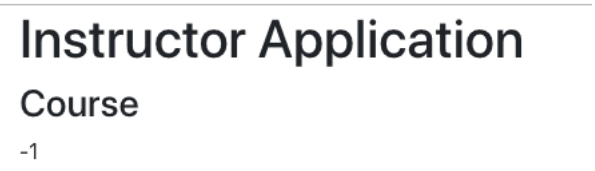
`let { description, id } = this.state` is called destructing. This is similar to writing code shown below.

```
let description = this.state.description
let id = this.state.id
```

When you try to update a course, you would see the screen below.



When you try to create a course, you would see the screen below.



This is because of the logic in `componentDidMount` to not invoke the course API for a new todo.

```
if (this.state.id == -1) {
    return
}
```

Create a Form using Formik

Now that we have loaded up the details of a specific course, let's shift out our attention to editing them and saving them back to the database.

To edit course details, we need a form. The most popular form framework with React is formik. We already added formik to our `package.json` using the command `npm add formik`.

Let's now create a simple form using formik.

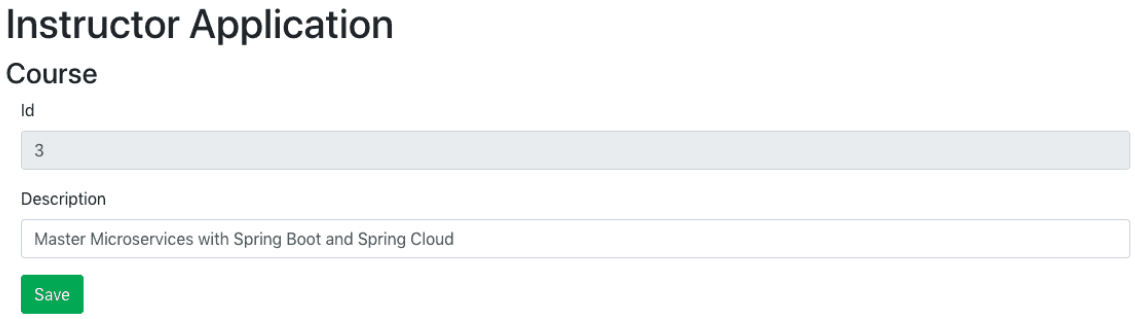
```
render() {

    let { description, id } = this.state

    return (
        <div>
            <h3>Course</h3>
            <div className="container">
                <Formik
                    initialValues={{ id, description }}
                >
```

```
                                    <label>Id</label>
                                    <Field className="form-control" type="text" name="id" disabled />
                                </fieldset>
                                <fieldset className="form-group">
                                    <label>Description</label>
                                    <Field className="form-control" type="text" name="description" />
                                </fieldset>
                                <button className="btn btn-success" type="submit">Save</button>
                            </Form>
                        )
                    }
                </Formik>

            </div>
        </div>
    )
}
```

Following are some of the important details:

- `let { description, id } = this.state` - Creating local variable using destructuring
- `<Formik initialValues={{ id, description }}>` - Initialing Formik with the values loaded from state
- `<Field className="form-control" type="text" name="id" disabled />` - Creating a disabled text element for id. The name of element should match the name in state.
- `<Field className="form-control" type="text" name="description" />` - Creating a text element for description. The name of element should match the name in state.
- `<button className="btn btn-success" type="submit">Save</button>` - Adding a submit button.

When you try to update a course, you would see the screen below.



Adding Handling of Submit Event

Let's try to handle the Submit event now.

Let's create an onSubmit method to log the values

```
onSubmit(values) {
    console.log(values);
}
```

Let's bind the `onSubmit` method to `this` in the constructor.

```
this.onSubmit = this.onSubmit.bind(this)
```

It's time to tie up the form with the onSubmit method. The key snippet is `onSubmit={this.onSubmit}`.

```
<Formik
      initialValues={{ id, description }}
      onSubmit={this.onSubmit}
                >
```

When you click Submit, the form details are now printed to the console.

```
{id: "2", description: "Learn Microservices"}
```

Adding Validation using Formik

What's a form without validation?

Let's add a validate method.

```
validate(values) {
    let errors = {}
    if (!values.description) {
        errors.description = 'Enter a Description'
    } else if (values.description.length < 5) {
```

```
    return errors
}
```

We are adding two validations:

- check for empty description
- check for a minimum length of 5

You can add other validations as you need.

As usual, let's bind it to `this` in the constructor.

```
this.validate = this.validate.bind(this)
```

Let's ties this up with the form. The key snippet is `validate={this.validate}`. We do not want to validate on change of value or on blur of the field. Let's keep things simple. `enableReinitialize={true}` is needed to ensure that we can reload the form for existing todo.

```
<Formik
      initialValues={{ id, description }}
      onSubmit={this.onSubmit}
      validateOnChange={false}
      validateOnBlur={false}
      validate={this.validate}
      enableReinitialize={true}>
```

If you run the page right now and submit invalid description, you would see that validations prevent the form from getting submitted.

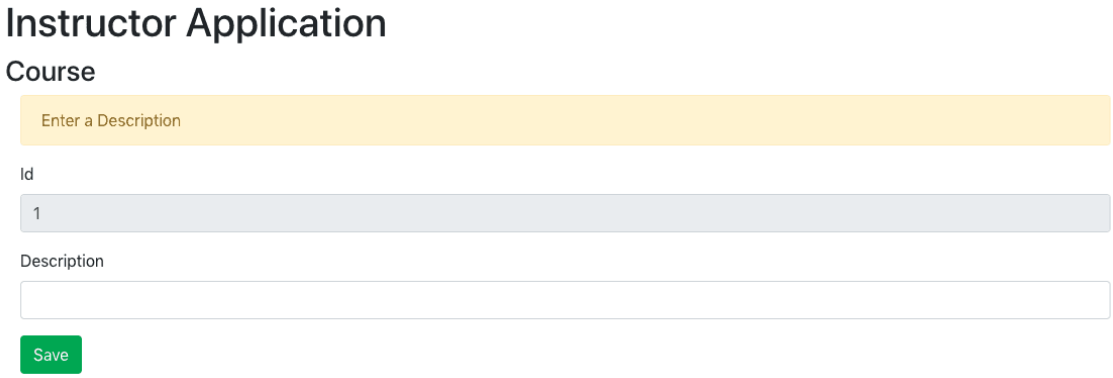
> *How do we see validation messages on the screen?*

Formik provides ErrorMessage.

Let's add error message to the field:

```
<ErrorMessage name="description" component="div"
    className="alert alert-warning" />
```

When you try to update a course, you would see the screen below.



Updating Course Details on click of submit

Now that the form is ready, we would want to call the backend API to save the course details.

Let's quickly create the API to Update and Create Courses.

Create API to Update Course

Let's add a `save` method to `CoursesHardcodedService` to handle creation and updation of course.

```
public Course save(Course course) {
  if (course.getId() == -1 || course.getId() == 0) {
    course.setId(++idCounter);
    courses.add(course);
  } else {
    deleteById(course.getId());
    courses.add(course);
  }
  return course;
}
```

```
@PutMapping("/instructors/{username}/courses/{id}")
public ResponseEntity<Course> updateCourse(@PathVariable String username, @PathVariable long id,
    @RequestBody Course course) {

  Course courseUpdated = courseManagementService.save(course);

  return new ResponseEntity<Course>(courseUpdated, HttpStatus.OK);
}
```

Adding API to Create Course

Let's add a method to the Resource class to create the course. We are using POST method to create the course. On course updation, we are returning a status of CREATED.

```
@PostMapping("/instructors/{username}/courses")
public ResponseEntity<Void> createCourse(@PathVariable String username, @RequestBody Course course) {

  Course createdCourse = courseManagementService.save(course);

  // Location
  // Get current resource url
  /// {id}
  URI uri = ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}").buildAndExpand(createdCourse.getId(
      .toUri();

  return ResponseEntity.created(uri).build();
}
```

Invoking Update and Create APIs from Course Screen

Now that the REST API is ready, let's create the frontend methods to call them.

Let's create respective methods in the `CourseDataService`. `updateCourse` uses a `put` and `createCourse` uses `post`.

```
class CourseDataService {
  updateCourse(name, id, course) {
      return axios.put(`${INSTRUCTOR_API_URL}/courses/${id}`, course);
  }

  createCourse(name, course) {
      return axios.post(`${INSTRUCTOR_API_URL}/courses/`, course);
  }
```

Let's update the `CourseComponent` to invoke the right service on the click of the submit button.

```
  onSubmit(values) {
      let username = INSTRUCTOR

      let course = {
          id: this.state.id,
          description: values.description,
          targetDate: values.targetDate
      }

      if (this.state.id === -1) {
          CourseDataService.createCourse(username, course)
              .then(() => this.props.history.push('/courses'))
      } else {
          CourseDataService.updateCourse(username, this.state.id, course)
              .then(() => this.props.history.push('/courses'))
      }

      console.log(values);
  }
```

We are creating a course object with the updated details and calling the appropriate method on the `CourseDataService`. Once the request is successful, we are redirecting the user to the course listing page using `this.props.history.push('/courses')`.

Complete Code Example

/spring-boot-react-crud-full-stack-with-maven/frontend-spring-boot-react-crud-full-stack-with-maven/public/index.html

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8" />
  <link rel="shortcut icon" href="%PUBLIC_URL%/favicon.ico" />
```

```
    manifest.json provides metadata used when your web app is installed on a
    user's mobile device or desktop. See https://developers.google.com/web/fundamentals/web-app-manifest/
  -->
  <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
  <!--
    Notice the use of %PUBLIC_URL% in the tags above.
    It will be replaced with the URL of the `public` folder during the build.
    Only files inside the `public` folder can be referenced from the HTML.

    Unlike "/favicon.ico" or "favicon.ico", "%PUBLIC_URL%/favicon.ico" will
    work correctly both with client-side routing and a non-root public URL.
    Learn how to configure a non-root public URL by running `npm run build`.
  -->
  <title>My Full Stack Application with Spring Boot and React</title>
</head>

<body>
  <noscript>You need to enable JavaScript to run this app.</noscript>
  <div id="root"></div>
  <!--
    This HTML file is a template.
    If you open it directly in the browser, you will see an empty page.

    You can add webfonts, meta tags, or analytics to this file.
    The build step will place the bundled scripts into the <body> tag.

    To begin the development, run `npm start` or `yarn start`.
    To create a production bundle, use `npm run build` or `yarn build`.
  -->
</body>

</html>
```

/spring-boot-react-crud-full-stack-with-maven/frontend-spring-boot-react-crud-full-stack-with-maven/public/manifest.json

```
{
  "short_name": "React App",
  "name": "Create React App Sample",
  "icons": [
    {
      "src": "favicon.ico",
      "sizes": "64x64 32x32 24x24 16x16",
      "type": "image/x-icon"
    }
  ],
  "start_url": ".",
  "display": "standalone",
  "theme_color": "#000000",
  "background_color": "#ffffff"
}
```

/spring-boot-react-crud-full-stack-with-maven/frontend-spring-boot-react-crud-full-stack-with-maven/src/App.css

```
@import url(https://unpkg.com/bootstrap@4.1.0/dist/css/bootstrap.min.css)
```

/spring-boot-react-crud-full-stack-with-maven/frontend-spring-boot-react-crud-full-stack-with-maven/src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import * as serviceWorker from './serviceWorker';

ReactDOM.render(<App />, document.getElementById('root'));

// If you want your app to work offline and load faster, you can change
// unregister() to register() below. Note this comes with some pitfalls.
// Learn more about service workers: https://bit.ly/CRA-PWA
serviceWorker.unregister();
```

IN28MINUTES

```
import React, { Component } from 'react'
import CourseDataService from '../service/CourseDataService';

const INSTRUCTOR = 'in28minutes'

class ListCoursesComponent extends Component {
    constructor(props) {
        super(props)
        this.state = {
            courses: [],
            message: null
        }
        this.deleteCourseClicked = this.deleteCourseClicked.bind(this)
        this.updateCourseClicked = this.updateCourseClicked.bind(this)
        this.addCourseClicked = this.addCourseClicked.bind(this)
        this.refreshCourses = this.refreshCourses.bind(this)
    }

    componentDidMount() {
        this.refreshCourses();
    }

    refreshCourses() {
        CourseDataService.retrieveAllCourses(INSTRUCTOR)//HARDCODED
            .then(
                response => {
                    //console.log(response);
                    this.setState({ courses: response.data })
                }
            )
    }

    deleteCourseClicked(id) {
        CourseDataService.deleteCourse(INSTRUCTOR, id)
            .then(
                response => {
                    this.setState({ message: `Delete of course ${id} Successful` })
                    this.refreshCourses()
                }
            )

    }

    addCourseClicked() {
        this.props.history.push(`/courses/-1`)
    }

    updateCourseClicked(id) {
        console.log('update ' + id)
        this.props.history.push(`/courses/${id}`)
    }

    render() {
        console.log('render')
        return (
            <div className="container">
                <h3>All Courses</h3>
                {this.state.message && <div class="alert alert-success">{this.state.message}</div>}
                <div className="container">
                    <table className="table">
                        <thead>
                            <tr>
                                <th>Id</th>
                                <th>Description</th>
                                <th>Update</th>
                                <th>Delete</th>
                            </tr>
                        </thead>
                        <tbody>
                            {
                                this.state.courses.map(
                                    course =>
                                        <tr key={course.id}>
                                            <td>{course.id}</td>
                                            <td>{course.description}</td>
                                            <td><button className="btn btn-success" onClick={() => this.updateCo
                                            <td><button className="btn btn-warning" onClick={() => this.deleteCo
                                        </tr>
                                )
                            }
                        </tbody>
                    </table>
                    <div className="row">
                        <button className="btn btn-success" onClick={this.addCourseClicked}>Add</button>
                    </div>
                </div>
            </div>
        )
    }
}

export default ListCoursesComponent
```

IN28MINUTES

/spring-boot-react-crud-full-stack-with-maven/frontend-spring-boot-react-crud-full-stack-with-maven/src/component/InstructorApp.jsx

```
import React, { Component } from 'react';
import ListCoursesComponent from './ListCoursesComponent';
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom'
import CourseComponent from './CourseComponent';

class InstructorApp extends Component {
    render() {
        return (
            <Router>
                <>
                    <h1>Instructor Application</h1>
                    <Switch>
                        <Route path="/" exact component={ListCoursesComponent} />
                        <Route path="/courses" exact component={ListCoursesComponent} />
                        <Route path="/courses/:id" component={CourseComponent} />
                    </Switch>
                </>
            </Router>
        )
    }
}

export default InstructorApp
```

/spring-boot-react-crud-full-stack-with-maven/frontend-spring-boot-react-crud-full-stack-with-maven/src/component/CourseComponent.jsx

```
import React, { Component } from 'react'
import { Formik, Form, Field, ErrorMessage } from 'formik';
import CourseDataService from '../service/CourseDataService';

const INSTRUCTOR = 'in28minutes'

class CourseComponent extends Component {
    constructor(props) {
        super(props)

        this.state = {
            id: this.props.match.params.id,
            description: ''
        }

        this.onSubmit = this.onSubmit.bind(this)
        this.validate = this.validate.bind(this)

    }

    componentDidMount() {

        console.log(this.state.id)

        // eslint-disable-next-line
        if (this.state.id == -1) {
            return
        }

        CourseDataService.retrieveCourse(INSTRUCTOR, this.state.id)
            .then(response => this.setState({
                description: response.data.description
            }))
    }

    validate(values) {
        let errors = {}
        if (!values.description) {
            errors.description = 'Enter a Description'
        } else if (values.description.length < 5) {
            errors.description = 'Enter atleast 5 Characters in Description'
        }

        return errors

    }

    onSubmit(values) {
        let username = INSTRUCTOR

        let course = {
            id: this.state.id,
            description: values.description
        }
```

```
        } else {
            CourseDataService.updateCourse(username, this.state.id, course)
                .then(() => this.props.history.push('/courses'))
        }

        console.log(values);
    }

    render() {

        let { description, id } = this.state

        return (
            <div>
                <h3>Course</h3>
                <div className="container">
                    <Formik
                        initialValues={{ id, description }}
                        onSubmit={this.onSubmit}
                        validateOnChange={false}
                        validateOnBlur={false}
                        validate={this.validate}
                        enableReinitialize={true}
                    >
                        {
                            (props) => (
                                <Form>
                                    <ErrorMessage name="description" component="div"
                                        className="alert alert-warning" />
                                    <fieldset className="form-group">
                                        <label>Id</label>
                                        <Field className="form-control" type="text" name="id" disabled />
                                    </fieldset>
                                    <fieldset className="form-group">
                                        <label>Description</label>
                                        <Field className="form-control" type="text" name="description" />
                                    </fieldset>
                                    <button className="btn btn-success" type="submit">Save</button>
                                </Form>
                            )
                        }
                    </Formik>

                </div>
            </div>
        )
    }
}

export default CourseComponent
```

/spring-boot-react-crud-full-stack-with-maven/frontend-spring-boot-react-crud-full-stack-with-maven/src/index.css

```
body {
  margin: 0;
  padding: 0;
  font-family: -apple-system, BlinkMacSystemFont, "Segoe UI", "Roboto", "Oxygen",
    "Ubuntu", "Cantarell", "Fira Sans", "Droid Sans", "Helvetica Neue",
    sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
}

code {
  font-family: source-code-pro, Menlo, Monaco, Consolas, "Courier New",
    monospace;
}
```

/spring-boot-react-crud-full-stack-with-maven/frontend-spring-boot-react-crud-full-stack-with-maven/src/App.test.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

it('renders without crashing', () => {
  const div = document.createElement('div');
  ReactDOM.render(<App />, div);
  ReactDOM.unmountComponentAtNode(div);
});
```

IN28MINUTES

crud-full-stack-with-maven/src/serviceWorker.js

```
// This optional code is used to register a service worker.
// register() is not called by default.

// This lets the app load faster on subsequent visits in production, and gives
// it offline capabilities. However, it also means that developers (and users)
// will only see deployed updates on subsequent visits to a page, after all the
// existing tabs open on the page have been closed, since previously cached
// resources are updated in the background.

// To learn more about the benefits of this model and instructions on how to
// opt-in, read https://bit.ly/CRA-PWA

const isLocalhost = Boolean(
  window.location.hostname === 'localhost' ||
    // [::1] is the IPv6 localhost address.
    window.location.hostname === '[::1]' ||
    // 127.0.0.1/8 is considered localhost for IPv4.
    window.location.hostname.match(
      /^127(?:\.(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)){3}$/
    )
);

export function register(config) {
  if (process.env.NODE_ENV === 'production' && 'serviceWorker' in navigator) {
    // The URL constructor is available in all browsers that support SW.
    const publicUrl = new URL(process.env.PUBLIC_URL, window.location.href);
    if (publicUrl.origin !== window.location.origin) {
      // Our service worker won't work if PUBLIC_URL is on a different origin
      // from what our page is served on. This might happen if a CDN is used to
      // serve assets; see https://github.com/facebook/create-react-app/issues/2374
      return;
    }

    window.addEventListener('load', () => {
      const swUrl = `${process.env.PUBLIC_URL}/service-worker.js`;

      if (isLocalhost) {
        // This is running on localhost. Let's check if a service worker still exists or not.
        checkValidServiceWorker(swUrl, config);

        // Add some additional logging to localhost, pointing developers to the
        // service worker/PWA documentation.
        navigator.serviceWorker.ready.then(() => {
          console.log(
            'This web app is being served cache-first by a service ' +
              'worker. To learn more, visit https://bit.ly/CRA-PWA'
          );
        });
      } else {
        // Is not localhost. Just register service worker
        registerValidSW(swUrl, config);
      }
    });
  }
}

function registerValidSW(swUrl, config) {
  navigator.serviceWorker
    .register(swUrl)
    .then(registration => {
      registration.onupdatefound = () => {
        const installingWorker = registration.installing;
        if (installingWorker == null) {
          return;
        }
        installingWorker.onstatechange = () => {
          if (installingWorker.state === 'installed') {
            if (navigator.serviceWorker.controller) {
              // At this point, the updated precached content has been fetched,
              // but the previous service worker will still serve the older
              // content until all client tabs are closed.
              console.log(
                'New content is available and will be used when all ' +
                  'tabs for this page are closed. See https://bit.ly/CRA-PWA.'
              );

              // Execute callback
              if (config && config.onUpdate) {
                config.onUpdate(registration);
              }
            } else {
              // At this point, everything has been precached.
              // It's the perfect time to display a
              // "Content is cached for offline use." message.
              console.log('Content is cached for offline use.');

              // Execute callback
              if (config && config.onSuccess) {
                config.onSuccess(registration);
              }
            }
```

```
    .catch(error => {
      console.error('Error during service worker registration:', error);
    });
}

function checkValidServiceWorker(swUrl, config) {
  // Check if the service worker can be found. If it can't reload the page.
  fetch(swUrl)
    .then(response => {
      // Ensure service worker exists, and that we really are getting a JS file.
      const contentType = response.headers.get('content-type');
      if (
        response.status === 404 ||
        (contentType != null && contentType.indexOf('JavaScript') === -1)
      ) {
        // No service worker found. Probably a different app. Reload the page.
        navigator.serviceWorker.ready.then(registration => {
          registration.unregister().then(() => {
            window.location.reload();
          });
        });
      } else {
        // Service worker found. Proceed as normal.
        registerValidSW(swUrl, config);
      }
    })
    .catch(() => {
      console.log(
        'No internet connection found. App is running in offline mode.'
      );
    });
}

export function unregister() {
  if ('serviceWorker' in navigator) {
    navigator.serviceWorker.ready.then(registration => {
      registration.unregister();
    });
  }
}
```

/spring-boot-react-crud-full-stack-with-maven/frontend-spring-boot-react-crud-full-stack-with-maven/src/service/CourseDataService.js

```
import axios from 'axios'

const INSTRUCTOR = 'in28minutes'
const COURSE_API_URL = 'http://localhost:8080'
const INSTRUCTOR_API_URL = `${COURSE_API_URL}/instructors/${INSTRUCTOR}`

class CourseDataService {

    retrieveAllCourses(name) {
        //console.log('executed service')
        return axios.get(`${INSTRUCTOR_API_URL}/courses`);
    }

    retrieveCourse(name, id) {
        //console.log('executed service')
        return axios.get(`${INSTRUCTOR_API_URL}/courses/${id}`);
    }

    deleteCourse(name, id) {
        //console.log('executed service')
        return axios.delete(`${INSTRUCTOR_API_URL}/courses/${id}`);
    }

    updateCourse(name, id, course) {
        //console.log('executed service')
        return axios.put(`${INSTRUCTOR_API_URL}/courses/${id}`, course);
    }

    createCourse(name, course) {
        //console.log('executed service')
        return axios.post(`${INSTRUCTOR_API_URL}/courses/`, course);
    }
}

export default new CourseDataService()
```

/spring-boot-react-crud-full-stack-with-maven/frontend-spring-boot-react-crud-full-stack-with-maven/src/logo.svg

```
        <path d= M666.3 296.5c0-32.5-40.7-63.3-103.1-82.4 14.4-63.6 8-114.2-20.2-130.4-6.5-3.8-14.1-5.6-22.4-5.6
        <circle cx="420.9" cy="296.5" r="45.7"/>
        <path d="M520.5 78.1z"/>
    </g>
</svg>
```

/spring-boot-react-crud-full-stack-with-maven/frontend-spring-boot-react-crud-full-stack-with-maven/src/App.js

```
import React, { Component } from 'react';
import './App.css';
import InstructorApp from './component/InstructorApp';

class App extends Component {
  render() {
    return (
      <div className="container">
        <InstructorApp />
      </div>
    );
  }
}

export default App;
```

/spring-boot-react-crud-full-stack-with-maven/frontend-spring-boot-react-crud-full-stack-with-maven/package.json

```
{
  "name": "spring-boot-react-crud-full-stack-with-maven",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "axios": "^0.18.0",
    "formik": "^1.5.1",
    "react": "^16.8.5",
    "react-dom": "^16.8.5",
    "react-router-dom": "^5.0.0",
    "react-scripts": "2.1.8"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": "react-app"
  },
  "browserslist": [
    ">0.2%",
    "not dead",
    "not ie <= 11",
    "not op_mini all"
  ]
}
```

/spring-boot-react-crud-full-stack-with-maven/backend-spring-boot-react-crud-full-stack-with-maven/pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<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>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.3.RELEASE</version>
    <relativePath /> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.in28minutes.fullstack.springboot.maven.crud</groupId>
  <artifactId>spring-boot-fullstack-crud-full-stack-with-maven</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>spring-boot-fullstack-crud-full-stack-with-maven</name>
  <description>Demo project for Spring Boot</description>
```

```
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-devtools</artifactId>
      <scope>runtime</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>

</project>
```

/spring-boot-react-crud-full-stack-with-maven/backend-spring-boot-react-crud-full-stack-with-maven/src/test/java/com/in28minutes/fullstack/springboot/react/maven/crud/springbootr

```
package com.in28minutes.fullstack.springboot.react.maven.crud.springbootreactcrudfullstackwithmaven;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@SpringBootTest
public class SpringBootReactCrudFullStackWithMavenApplicationTests {

  @Test
  public void contextLoads() {
  }

}
```

/spring-boot-react-crud-full-stack-with-maven/backend-spring-boot-react-crud-full-stack-with-maven/src/main/resources/application.properties

---

/spring-boot-react-crud-full-stack-with-maven/backend-spring-boot-react-crud-full-stack-with-maven/src/main/java/com/in28minutes/fullstack/springboot/maven/crud/springbootcrudfu

```
package com.in28minutes.fullstack.springboot.maven.crud.springbootcrudfullstackwithmaven;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringBootFullStackCrudFullStackWithMavenApplication {

  public static void main(String[] args) {
    SpringApplication.run(SpringBootFullStackCrudFullStackWithMavenApplication.class, args);
  }

}
```

IN28MINUTES

crud-full-stack-with-maven/src/main/java/com/in28minutes/fullstack/springboot/maven/crud/springbootcrudfu

```
package com.in28minutes.fullstack.springboot.maven.crud.springbootcrudfullstackwithmaven.course;

import java.util.ArrayList;
import java.util.List;

import org.springframework.stereotype.Service;

@Service
public class CoursesHardcodedService {

  private static List<Course> courses = new ArrayList<>();
  private static long idCounter = 0;

  static {
    courses.add(new Course(++idCounter, "in28minutes", "Learn Full stack with Spring Boot and Angular"));
    courses.add(new Course(++idCounter, "in28minutes", "Learn Full stack with Spring Boot and React"));
    courses.add(new Course(++idCounter, "in28minutes", "Master Microservices with Spring Boot and Spring Cloud")
    courses.add(new Course(++idCounter, "in28minutes",
        "Deploy Spring Boot Microservices to Cloud with Docker and Kubernetes"));
  }

  public List<Course> findAll() {
    return courses;
  }

  public Course save(Course course) {
    if (course.getId() == -1 || course.getId() == 0) {
      course.setId(++idCounter);
      courses.add(course);
    } else {
      deleteById(course.getId());
      courses.add(course);
    }
    return course;
  }

  public Course deleteById(long id) {
    Course course = findById(id);

    if (course == null)
      return null;

    if (courses.remove(course)) {
      return course;
    }

    return null;
  }

  public Course findById(long id) {
    for (Course course : courses) {
      if (course.getId() == id) {
        return course;
      }
    }

    return null;
  }
}
```

/spring-boot-react-crud-full-stack-with-maven/backend-spring-boot-react-crud-full-stack-with-maven/src/main/java/com/in28minutes/fullstack/springboot/maven/crud/springbootcrudfu

```
package com.in28minutes.fullstack.springboot.maven.crud.springbootcrudfullstackwithmaven.course;

public class Course {
  private Long id;
  private String username;
  private String description;

  public Course() {

  }

  public Course(long id, String username, String description) {
    super();
    this.id = id;
    this.username = username;
    this.description = description;
  }
```

```
  }

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

  public String getUsername() {
    return username;
  }

  public void setUsername(String username) {
    this.username = username;
  }

  public String getDescription() {
    return description;
  }

  public void setDescription(String description) {
    this.description = description;
  }

  @Override
  public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((description == null) ? 0 : description.hashCode());
    result = prime * result + ((id == null) ? 0 : id.hashCode());
    result = prime * result + ((username == null) ? 0 : username.hashCode());
    return result;
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    Course other = (Course) obj;
    if (description == null) {
      if (other.description != null)
        return false;
    } else if (!description.equals(other.description))
      return false;
    if (id == null) {
      if (other.id != null)
        return false;
    } else if (!id.equals(other.id))
      return false;
    if (username == null) {
      if (other.username != null)
        return false;
    } else if (!username.equals(other.username))
      return false;
    return true;
  }

}
```

/spring-boot-react-crud-full-stack-with-maven/backend-spring-boot-react-crud-full-stack-with-maven/src/main/java/com/in28minutes/fullstack/springboot/maven/crud/springbootcrudfu

```
package com.in28minutes.fullstack.springboot.maven.crud.springbootcrudfullstackwithmaven.course;

import java.net.URI;
import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.servlet.support.ServletUriComponentsBuilder;

@CrossOrigin(origins = { "http://localhost:3000", "http://localhost:4200" })
@RestController
public class CourseResource {

  @Autowired
```

```
    public List<Course> getAllCourses(@PathVariable String username) {
      return courseManagementService.findAll();
    }

    @GetMapping("/instructors/{username}/courses/{id}")
    public Course getCourse(@PathVariable String username, @PathVariable long id) {
      return courseManagementService.findById(id);
    }

    @DeleteMapping("/instructors/{username}/courses/{id}")
    public ResponseEntity<Void> deleteCourse(@PathVariable String username, @PathVariable long id) {

      Course course = courseManagementService.deleteById(id);

      if (course != null) {
        return ResponseEntity.noContent().build();
      }

      return ResponseEntity.notFound().build();
    }

    @PutMapping("/instructors/{username}/courses/{id}")
    public ResponseEntity<Course> updateCourse(@PathVariable String username, @PathVariable long id,
        @RequestBody Course course) {

      Course courseUpdated = courseManagementService.save(course);

      return new ResponseEntity<Course>(course, HttpStatus.OK);
    }

    @PostMapping("/instructors/{username}/courses")
    public ResponseEntity<Void> createCourse(@PathVariable String username, @RequestBody Course course) {

      Course createdCourse = courseManagementService.save(course);

      // Location
      // Get current resource url
      /// {id}
      URI uri = ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}").buildAndExpand(createdCourse.getId(
          .toUri();

      return ResponseEntity.created(uri).build();
    }

  }
```

IN28MINUTES