Spring Boot CRUD Operations

What is the CRUD operation?

The **CRUD** stands for **Create, Read/Retrieve, Update,** and **Delete**. These are the four basic functions of the persistence storage.

The CRUD operation can be defined as user interface conventions that allow view, search, and modify information through computer-based forms and reports. CRUD is data-oriented and the standardized use of **HTTP action verbs**. HTTP has a few important verbs.

* **POST:** Creates a new resource
* **GET:** Reads a resource
* **PUT:** Updates an existing resource
* **DELETE:** Deletes a resource

Within a database, each of these operations maps directly to a series of commands. However, their relationship with a RESTful API is slightly more complex.

Standard CRUD Operation

* **CREATE Operation:** It performs the INSERT statement to create a new record.
* **READ Operation:** It reads table records based on the input parameter.
* **UPDATE Operation:** It executes an update statement on the table. It is based on the input parameter.
* **DELETE Operation:** It deletes a specified row in the table. It is also based on the input parameter.

How CRUD Operations Works

CRUD operations are at the foundation of the most dynamic websites. Therefore, we should differentiate **CRUD** from the **HTTP** **action verbs**.

Suppose, if we want to **create** a new record, we should use HTTP action verb **POST**. To **update** a record, we should use the **PUT** verb. Similarly, if we want to **delete** a record, we should use the **DELETE**verb. Through CRUD operations, users and administrators have the right to retrieve, create, edit, and delete records online.

We have many options for executing CRUD operations. One of the most efficient choices is to create a set of stored procedures in SQL to execute operations.

The CRUD operations refer to all major functions that are implemented in relational database applications. Each letter of the CRUD can map to a SQL statement and HTTP methods.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **SQL** | **HTTP verbs** | **RESTful Web Service** |
| **Create** | INSERT | PUT/POST | POST |
| **Read** | SELECT | GET | GET |
| **Update** | UPDATE | PUT/POST/PATCH | PUT |
| **Delete** | DELETE | DELETE | DELETE |

Spring Boot CrudRepository

Spring Boot provides an interface called **CrudRepository**that contains methods for CRUD operations. It is defined in the package **org.springframework.data.repository**. It extends the Spring Data**Repository** interface. It provides generic Crud operation on a repository. If we want to use CrudRepository in an application, we have to create an interface and extend the **CrudRepository**.

**Syntax**

1. **public** **interface** CrudRepository<T,ID> **extends** Repository<T,ID>

where,

* **T** is the domain type that repository manages.
* **ID** is the type of the id of the entity that repository manages.

For example:

1. **public** **interface** StudentRepository **extends** CrudRepository<Student, Integer>
2. {
3. }

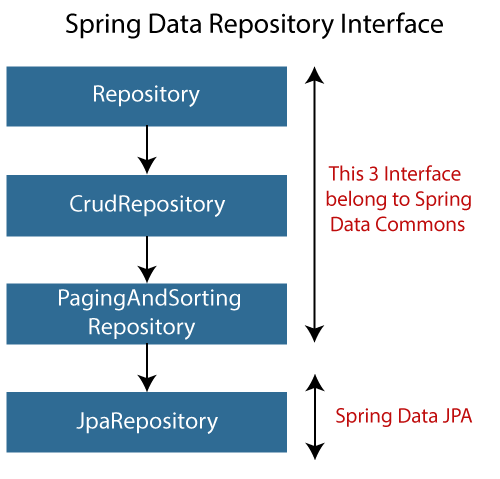
In the above example, we have created an interface named **StudentRepository** that extends CrudRepository. Where **Student** is the repository to manage, and **Integer** is the type of Id that is defined in the Student repository.

Spring Boot JpaRepository

JpaRepository provides JPA related methods such as flushing, persistence context, and deletes a record in a batch. It is defined in the package **org.springframework.data.jpa.repository.**JpaRepository extends both **CrudRepository** and **PagingAndSortingRepository.**

For example:

1. **public** **interface** BookDAO **extends** JpaRepository
2. {
3. }



Why should we use these interfaces?

* The interfaces allow Spring to find the repository interface and create proxy objects for that.
* It provides methods that allow us to perform some common operations. We can also define custom methods as well.

CrudRepository vs. JpaRepository

|  |  |
| --- | --- |
| **CrudRepository** | **JpaRepository** |
| CrudRepository does not provide any method for pagination and sorting. | JpaRepository extends PagingAndSortingRepository. It provides all the methods for implementing the pagination. |
| It works as a **marker** interface. | JpaRepository extends both **CrudRepository** and **PagingAndSortingRepository**. |
| It provides CRUD function only. For example **findById(), findAll(),** etc. | It provides some extra methods along with the method of PagingAndSortingRepository and CrudRepository. For example, **flush(), deleteInBatch().** |
| It is used when we do not need the functions provided by JpaRepository and PagingAndSortingRepository. | It is used when we want to implement pagination and sorting functionality in an application. |

Spring Boot CRUD Operation Example

Let's set up a Spring Boot application and perform CRUD operation.

**Step 1:** Open Spring Initializr [http://start.spring.io](http://start.spring.io/).

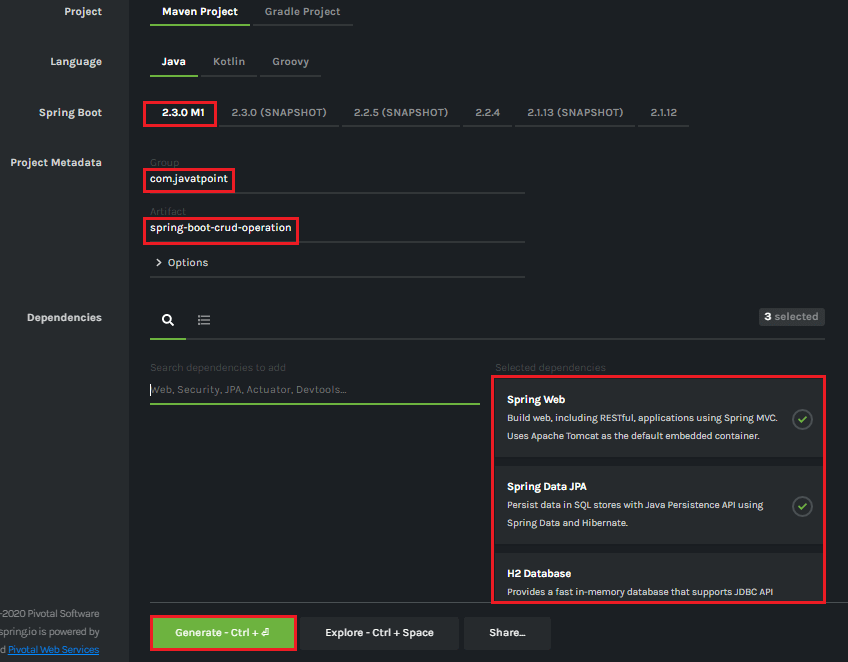
**Step 2:** Select the Spring Boot version **2.3.0.M1.**

**Step 2:** Provide the **Group** name. We have provided **com.javatpoint.**

**Step 3:** Provide the **Artifact** Id. We have provided **spring-boot-crud-operation.**

**Step 5:** Add the dependencies **Spring Web, Spring Data JPA,**and**H2 Database.**

**Step 6:** Click on the **Generate** button. When we click on the Generate button, it wraps the specifications in a **Jar** file and downloads it to the local system.



**Step 7: Extract** the Jar file and paste it into the STS workspace.

**Step 8: Import** the project folder into STS.

File -> Import -> Existing Maven Projects -> Browse -> Select the folder spring-boot-crud-operation -> Finish

It takes some time to import.

**Step 9:** Create a package with the name **com.javatpoint.model**in the folder **src/main/java.**

**Step 10:** Create a model class in the package **com.javatpoint.model.**We have created a model class with the name **Books.**In the Books class, we have done the following:

* Define four variable **bookid, bookname, author,**and
* Generate Getters and Setters.  
  Right-click on the file -> Source -> Generate Getters and Setters.
* Mark the class as an **Entity** by using the annotation **@Entity.**
* Mark the class as **Table** name by using the annotation **@Table.**
* Define each variable as **Column** by using the annotation **@Column.**

**Books.java**

1. **package** com.javatpoint.model;
2. **import** javax.persistence.Column;
3. **import** javax.persistence.Entity;
4. **import** javax.persistence.Id;
5. **import** javax.persistence.Table;
6. //mark class as an Entity
7. @Entity
8. //defining class name as Table name
9. @Table
10. **public** **class** Books
11. {
12. //Defining book id as primary key
13. @Id
14. @Column
15. **private** **int** bookid;
16. @Column
17. **private** String bookname;
18. @Column
19. **private** String author;
20. @Column
21. **private** **int** price;
22. **public** **int** getBookid()
23. {
24. **return** bookid;
25. }
26. **public** **void** setBookid(**int** bookid)
27. {
28. **this**.bookid = bookid;
29. }
30. **public** String getBookname()
31. {
32. **return** bookname;
33. }
34. **public** **void** setBookname(String bookname)
35. {
36. **this**.bookname = bookname;
37. }
38. **public** String getAuthor()
39. {
40. **return** author;
41. }
42. **public** **void** setAuthor(String author)
43. {
44. **this**.author = author;
45. }
46. **public** **int** getPrice()
47. {
48. **return** price;
49. }
50. **public** **void** setPrice(**int** price)
51. {
52. **this**.price = price;
53. }
54. }

**Step 11:** Create a package with the name **com.javatpoint.controller**in the folder **src/main/java.**

**Step 12:** Create a Controller class in the package **com.javatpoint.controller.**We have created a controller class with the name **BooksController.**In the BooksController class, we have done the following:

* Mark the class as **RestController**by using the annotation**@RestController.**
* Autowire the **BooksService** class by using the annotation **@Autowired**.
* Define the following methods:
  + **getAllBooks():** It returns a List of all Books.
  + **getBooks():** It returns a book detail that we have specified in the path variable. We have passed bookid as an argument by using the annotation @PathVariable. The annotation indicates that a method parameter should be bound to a URI template variable.
  + **deleteBook():** It deletes a specific book that we have specified in the path variable.
  + **saveBook():** It saves the book detail. The annotation @RequestBody indicates that a method parameter should be bound to the body of the web request.
  + **update():** The method updates a record. We must specify the record in the body, which we want to update. To achieve the same, we have used the annotation @RequestBody.

**BooksController.java**

1. **package** com.javatpoint.controller;
2. **import** java.util.List;
3. **import** org.springframework.beans.factory.annotation.Autowired;
4. **import** org.springframework.web.bind.annotation.DeleteMapping;
5. **import** org.springframework.web.bind.annotation.GetMapping;
6. **import** org.springframework.web.bind.annotation.PathVariable;
7. **import** org.springframework.web.bind.annotation.PostMapping;
8. **import** org.springframework.web.bind.annotation.PutMapping;
9. **import** org.springframework.web.bind.annotation.RequestBody;
10. **import** org.springframework.web.bind.annotation.RestController;
11. **import** com.javatpoint.model.Books;
12. **import** com.javatpoint.service.BooksService;
13. //mark class as Controller
14. @RestController
15. **public** **class** BooksController
16. {
17. //autowire the BooksService class
18. @Autowired
19. BooksService booksService;
20. //creating a get mapping that retrieves all the books detail from the database
21. @GetMapping("/book")
22. **private** List<Books> getAllBooks()
23. {
24. **return** booksService.getAllBooks();
25. }
26. //creating a get mapping that retrieves the detail of a specific book
27. @GetMapping("/book/{bookid}")
28. **private** Books getBooks(@PathVariable("bookid") **int** bookid)
29. {
30. **return** booksService.getBooksById(bookid);
31. }
32. //creating a delete mapping that deletes a specified book
33. @DeleteMapping("/book/{bookid}")
34. **private** **void** deleteBook(@PathVariable("bookid") **int** bookid)
35. {
36. booksService.delete(bookid);
37. }
38. //creating post mapping that post the book detail in the database
39. @PostMapping("/books")
40. **private** **int** saveBook(@RequestBody Books books)
41. {
42. booksService.saveOrUpdate(books);
43. **return** books.getBookid();
44. }
45. //creating put mapping that updates the book detail
46. @PutMapping("/books")
47. **private** Books update(@RequestBody Books books)
48. {
49. booksService.saveOrUpdate(books);
50. **return** books;
51. }
52. }

**Step 13:** Create a package with the name **com.javatpoint.service**in the folder **src/main/java.**

**Step 14:** Create a **Service** class. We have created a service class with the name **BooksService**in the package **com.javatpoint.service.**

**BooksService.java**

1. **package** com.javatpoint.service;
2. **import** java.util.ArrayList;
3. **import** java.util.List;
4. **import** org.springframework.beans.factory.annotation.Autowired;
5. **import** org.springframework.stereotype.Service;
6. **import** com.javatpoint.model.Books;
7. **import** com.javatpoint.repository.BooksRepository;
8. //defining the business logic
9. @Service
10. **public** **class** BooksService
11. {
12. @Autowired
13. BooksRepository booksRepository;
14. //getting all books record by using the method findaAll() of CrudRepository
15. **public** List<Books> getAllBooks()
16. {
17. List<Books> books = **new** ArrayList<Books>();
18. booksRepository.findAll().forEach(books1 -> books.add(books1));
19. **return** books;
20. }
21. //getting a specific record by using the method findById() of CrudRepository
22. **public** Books getBooksById(**int** id)
23. {
24. **return** booksRepository.findById(id).get();
25. }
26. //saving a specific record by using the method save() of CrudRepository
27. **public** **void** saveOrUpdate(Books books)
28. {
29. booksRepository.save(books);
30. }
31. //deleting a specific record by using the method deleteById() of CrudRepository
32. **public** **void** delete(**int** id)
33. {
34. booksRepository.deleteById(id);
35. }
36. //updating a record
37. **public** **void** update(Books books, **int** bookid)
38. {
39. booksRepository.save(books);
40. }
41. }

**Step 15:** Create a package with the name **com.javatpoint.repository**in the folder **src/main/java.**

**Step 16:** Create a **Repository** interface. We have created a repository interface with the name **BooksRepository**in the package **com.javatpoint.repository.**It extends the **Crud Repository** interface.

**BooksRepository.java**

1. **package** com.javatpoint.repository;
2. **import** org.springframework.data.repository.CrudRepository;
3. **import** com.javatpoint.model.Books;
4. //repository that extends CrudRepository
5. **public** **interface** BooksRepository **extends** CrudRepository<Books, Integer>
6. {
7. }

Now we will configure the datasource **URL, driver class name, username,**and**password,**in the **application.properties**file.

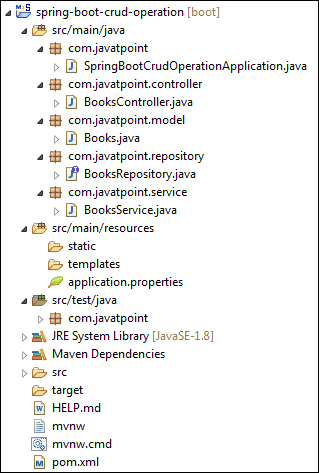
**Step 17:** Open the **application.properties** file and configure the following properties.

**application.properties**

1. spring.datasource.url=jdbc:h2:mem:books\_data
2. spring.datasource.driverClassName=org.h2.Driver
3. spring.datasource.username=sa
4. spring.datasource.password=
5. spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
6. #enabling the H2 console
7. spring.h2.console.enabled=**true**

Note: Do not forget to enable the H2 console.

After creating all the classes and packages, the project directory looks like the following.



Now we will run the application.

**Step 18:** Open **SpringBootCrudOperationApplication.java**file and run it as Java Application.

**SpringBootCrudOperationApplication.java**

1. **package** com.javatpoint;
2. **import** org.springframework.boot.SpringApplication;
3. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
4. @SpringBootApplication
5. **public** **class** SpringBootCrudOperationApplication
6. {
7. **public** **static** **void** main(String[] args)
8. {
9. SpringApplication.run(SpringBootCrudOperationApplication.**class**, args);
10. }
11. }

Note: In the next steps we will use rest client Postman. So, ensure that the Postman application is already installed in your system.

**Step 19:** Open the **Postman**and do the following:

* Select the **POST**
* Invoke the URL http://localhost:8080/books.
* Select the **Body**
* Select he Content-Type **JSON (application/json).**
* Insert the data. We have inserted the following data in the Body:

{

    "bookid": "5433",

    "bookname": "Core and Advance Java",

    "author": "R. Nageswara Rao",

    "price": "800"

}

* Click on the **Send**

When the request is successfully executed, it shows the **Status:200 OK**. It means the record has been successfully inserted in the database.

Similarly, we have inserted the following data.

{

    "bookid": "0982",

    "bookname": "Programming with Java",

    "author": "E. Balagurusamy",

    "price": "350"

}

{

    "bookid": "6321",

    "bookname": "Data Structures and Algorithms in Java",

    "author": "Robert Lafore",

    "price": "590"

}

{

    "bookid": "5433",

    "bookname": "Effective Java",

    "author": "Joshua Bloch",

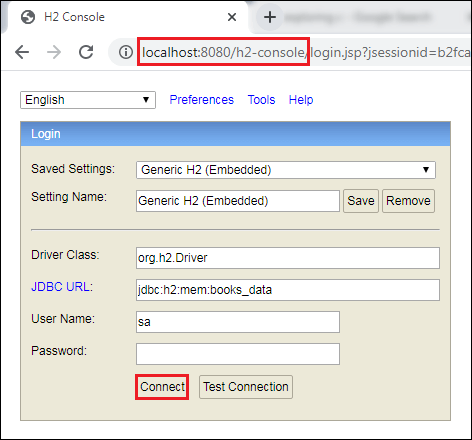
    "price": "670"

}

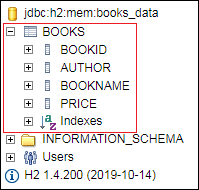
Let's access the H2 console to see the data.

**Step 20:** Open the browser and invoke the URL http://localhost:8080/h2-console. Click on the **Connect** button, as shown below.

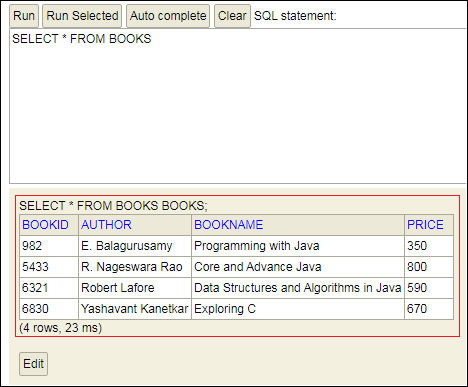
jdbc:h2:mem:books\_data



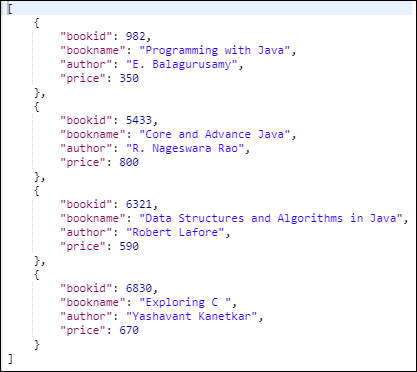
After clicking on the **Connect** button, we see the **Books** table in the database, as shown below.



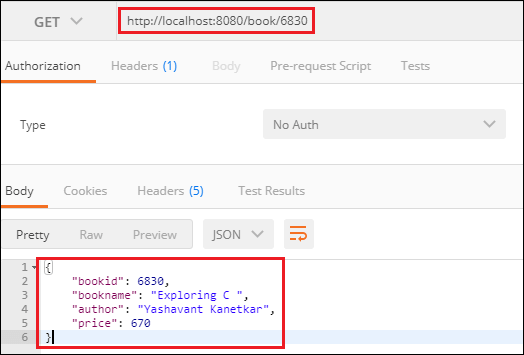
**Step 21:** Click on the **Books** table and then click on the **Run** button. The table shows the data that we have inserted in the body.



**Step 22:** Open the **Postman** and send a **GET** request with the URL http://localhost:8080/books. It returns the data that we have inserted in the database.

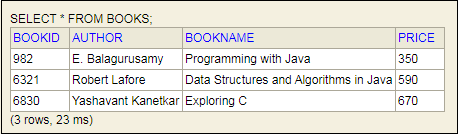


Let's send a **GET** request with the URL http://localhost:8080/book/{bookid}. We have specified the **bookid 6830**. It returns the detail of the book whose id is 6830.



Similarly, we can also send a **DELETE** request to delete a record. Suppose we want to delete a book record whose id is **5433**.

Select the **DELETE** method and invoke the URL http://localhost:8080/book/5433. Again, execute the **Select** query in the H2 console. We see that the book whose id is **5433** has been deleted from the database.



Similarly, we can also update a record by sending a **PUT** request. Let's update the price of the book whose id is **6321**.

* Select the **PUT**
* In the request body, paste the record which you want to update and make the changes. In our case, we want to update the record of the book whose id is 6321. In the following record, we have changed the price of the book.

1. {
2. "bookid": "6321",
3. "bookname": "Data Structures and Algorithms in Java",
4. "author": "Robert Lafore",
5. "price": "500"
6. }

* Click on the **Send**

Now, move to the H2 console and see the changes have reflected or not. We see that the price of the book has been changed, as shown below.

