**Spring Boot MongoDB CURD Example with Maven**

we’re gonna build a Spring Boot Rest API example that use Spring Data MongoDB & Maven to make CRUD operations with MongoDB database.

* How to configure Spring Data to work with MongoDB Database
* How to define MongoDB Data Models and Repository interfaces
* Way to create Spring Rest Controller to process HTTP requests
* Way to use Spring Data MongoDB to interact with MongoDB Database

**We will build a Spring Boot MongoDB Rest CRUD API for a Tutorial application.**

Each Tutotial has id, title, description, published status.

APIs help to **create, retrieve, update, delete** Tutorials.

APIs also support custom finder methods such as find by published status or by title.

These are APIs that we need to provide:

|  |  |  |
| --- | --- | --- |
| Methods | Urls | Actions |
| POST | /api/tutorials | create new Tutorial |
| GET | /api/tutorials | retrieve all Tutorials |
| GET | /api/tutorials/:id | retrieve a Tutorial by :id |
| PUT | /api/tutorials/:id | update a Tutorial by :id |
| DELETE | /api/tutorials/:id | delete a Tutorial by :id |
| DELETE | /api/tutorials | delete all Tutorials |
| GET | /api/tutorials/published | find all published Tutorials |
| GET | /api/tutorials?title=[keyword] | find all Tutorials which title contains keyword |

– We make CRUD operations & finder methods using Spring Data MongoDB.

– Rest Controller will be created with the help of Spring Web MVC.

* Tutorial data model class corresponds to entity and table tutorials.
* TutorialRepository is an interface that extends [MongoRepository](https://docs.spring.io/spring-data/mongodb/docs/current/api/org/springframework/data/mongodb/repository/MongoRepository.html) for CRUD methods and custom finder methods. It will be autowired in TutorialController.
* TutorialController is a [RestController](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RestController.html) which has request mapping methods for RESTful requests such as: getAllTutorials, createTutorial, updateTutorial, deleteTutorial, findByPublished…
* Configuration for Spring Data MongoDB is in application.properties.
* pom.xml contains dependencies for Spring Boot Web MVC and Spring Data MongoDB.

**Create & Setup Spring Boot project**

Use [Spring web tool](https://start.spring.io/) or your development tool ([Spring Tool Suite](https://spring.io/tools), Eclipse, [Intellij](https://www.jetbrains.com/idea/download/)) to create a Spring Boot project.

Then open pom.xml and add these dependencies:

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-data-mongodb</artifactId>  </dependency> |

**Configure Spring Data MongoDB**

Under src/main/resources folder, open application.properties and add following lines.

|  |
| --- |
| spring.data.mongodb.database=bezkoder\_db  spring.data.mongodb.port=27017 |

**Define Data Model**

Our Data model is Tutorial with four fields: id, title, description, published.  
In model package, we define Tutorial class.

|  |
| --- |
| **model/Tutorial.java**  package com.bezkoder.spring.data.mongodb.model;  import org.springframework.data.annotation.Id;  import org.springframework.data.mongodb.core.mapping.Document;  @Document(collection = "tutorials")  public class Tutorial {  @Id  private String id;  private String title;  private String description;  private boolean published;  public Tutorial() {  }  public Tutorial(String title, String description, boolean published) {  this.title = title;  this.description = description;  this.published = published;  }  public String getId() {  return id;  }  public String getTitle() {  return title;  }  public void setTitle(String title) {  this.title = title;  }  public String getDescription() {  return description;  }  public void setDescription(String description) {  this.description = description;  }  public boolean isPublished() {  return published;  }  public void setPublished(boolean isPublished) {  this.published = isPublished;  }  @Override  public String toString() {  return "Tutorial [id=" + id + ", title=" + title + ", desc=" + description + ", published=" + published + "]";  }  } |

**@Document** annotation helps us override the collection name by “tutorials”.

**Create Repository Interface**

Let’s create a repository to interact with Tutorials from the database.  
In repository package, create TutorialRepository interface that extends MongoRepository.

|  |
| --- |
| **repository/TutorialRepository.java**  package com.bezkoder.spring.data.mongodb.repository;  import java.util.List;  import org.springframework.data.mongodb.repository.MongoRepository;  import com.bezkoder.spring.data.mongodb.model.Tutorial;  public interface TutorialRepository extends MongoRepository<Tutorial, String> {  List<Tutorial> findByTitleContaining(String title);  List<Tutorial> findByPublished(boolean published);  } |

Now we can use MongoRepository’s methods: save(), findOne(), findById(), findAll(), count(), delete(), deleteById()… without implementing these methods.

We also define custom finder methods:  
– findByTitleContaining(): returns all Tutorials which title contains input title.  
– findByPublished(): returns all Tutorials with published having value as input published.

The implementation is plugged in by [Spring Data MongoDB](https://docs.spring.io/spring-data/mongodb/docs/current/reference/html/#reference) automatically.

**Create Spring Rest APIs Controller**

Finally, we create a controller that provides APIs for creating, retrieving, updating, deleting and finding Tutorials.

|  |
| --- |
| **controller/TutorialController.java**  package com.bezkoder.spring.data.mongodb.controller;  ...  import com.bezkoder.spring.data.mongodb.model.Tutorial;  import com.bezkoder.spring.data.mongodb.repository.TutorialRepository;  @CrossOrigin(origins = "http://localhost:8081")  @RestController  @RequestMapping("/api")  public class TutorialController {  @Autowired  TutorialRepository tutorialRepository;  @GetMapping("/tutorials")  public ResponseEntity<List<Tutorial>> getAllTutorials(@RequestParam(required = false) String title) {    }  @GetMapping("/tutorials/{id}")  public ResponseEntity<Tutorial> getTutorialById(@PathVariable("id") String id) {    }  @PostMapping("/tutorials")  public ResponseEntity<Tutorial> createTutorial(@RequestBody Tutorial tutorial) {    }  @PutMapping("/tutorials/{id}")  public ResponseEntity<Tutorial> updateTutorial(@PathVariable("id") String id, @RequestBody Tutorial tutorial) {    }  @DeleteMapping("/tutorials/{id}")  public ResponseEntity<HttpStatus> deleteTutorial(@PathVariable("id") String id) {    }  @DeleteMapping("/tutorials")  public ResponseEntity<HttpStatus> deleteAllTutorials() {    }  @GetMapping("/tutorials/published")  public ResponseEntity<List<Tutorial>> findByPublished() {    }  } |

– @CrossOrigin is for configuring allowed origins.  
– @RestController annotation is used to define a controller and to indicate that the return value of the methods should be be bound to the web response body.  
– @RequestMapping("/api") declares that all Apis’ url in the controller will start with /api.  
– We use @Autowired to inject TutorialRepository bean to local variable.

Now I will show you how to implement each controller’s CRUD methods.

**Create Operation**

We use @PostMapping annotation for handling POST HTTP requests.  
A new Tutorial will be created by MongoRepository.save() method.

|  |
| --- |
| @PostMapping("/tutorials")  public ResponseEntity<Tutorial> createTutorial(@RequestBody Tutorial tutorial) {  try {  Tutorial \_tutorial = tutorialRepository.save(new Tutorial(tutorial.getTitle(), tutorial.getDescription(), false));  return new ResponseEntity<>(\_tutorial, HttpStatus.CREATED);  } catch (Exception e) {  return new ResponseEntity<>(null, HttpStatus.INTERNAL\_SERVER\_ERROR);  }  } |

**Retrieve Operations**

We use @GetMapping annotation for handling GET HTTP requests, then Repository’s findAll(), findByTitleContaining(title), findByPublished() method to get the result.

* getAllTutorials(): returns List of Tutorials, if there is title parameter, it returns a List in that each Tutorial contains the title
* getTutorialById(): returns Tutorial by given id
* findByPublished(): return published Tutorials

|  |
| --- |
| @GetMapping("/tutorials")  public ResponseEntity<List<Tutorial>> getAllTutorials(@RequestParam(required = false) String title) {  try {  List<Tutorial> tutorials = new ArrayList<Tutorial>();  if (title == null)  tutorialRepository.findAll().forEach(tutorials::add);  else  tutorialRepository.findByTitleContaining(title).forEach(tutorials::add);  if (tutorials.isEmpty()) {  return new ResponseEntity<>(HttpStatus.NO\_CONTENT);  }  return new ResponseEntity<>(tutorials, HttpStatus.OK);  } catch (Exception e) {  return new ResponseEntity<>(null, HttpStatus.INTERNAL\_SERVER\_ERROR);  }  }  @GetMapping("/tutorials/{id}")  public ResponseEntity<Tutorial> getTutorialById(@PathVariable("id") String id) {  Optional<Tutorial> tutorialData = tutorialRepository.findById(id);  if (tutorialData.isPresent()) {  return new ResponseEntity<>(tutorialData.get(), HttpStatus.OK);  } else {  return new ResponseEntity<>(HttpStatus.NOT\_FOUND);  }  }  @GetMapping("/tutorials/published")  public ResponseEntity<List<Tutorial>> findByPublished() {  try {  List<Tutorial> tutorials = tutorialRepository.findByPublished(true);  if (tutorials.isEmpty()) {  return new ResponseEntity<>(HttpStatus.NO\_CONTENT);  }  return new ResponseEntity<>(tutorials, HttpStatus.OK);  } catch (Exception e) {  return new ResponseEntity<>(HttpStatus.INTERNAL\_SERVER\_ERROR);  }  } |

**Update Operation**

@PutMapping will help us handle PUT HTTP requests.  
– updateTutorial() receives id and a Tutorial payload.  
– from the id, we get the Tutorial from database using findById() method.  
– then we use the payload and save() method for updating the Tutorial.

|  |
| --- |
| @PutMapping("/tutorials/{id}")  public ResponseEntity<Tutorial> updateTutorial(@PathVariable("id") String id, @RequestBody Tutorial tutorial) {  Optional<Tutorial> tutorialData = tutorialRepository.findById(id);  if (tutorialData.isPresent()) {  Tutorial \_tutorial = tutorialData.get();  \_tutorial.setTitle(tutorial.getTitle());  \_tutorial.setDescription(tutorial.getDescription());  \_tutorial.setPublished(tutorial.isPublished());  return new ResponseEntity<>(tutorialRepository.save(\_tutorial), HttpStatus.OK);  } else {  return new ResponseEntity<>(HttpStatus.NOT\_FOUND);  }  } |

**Delete Operation**

We use @DeleteMapping for DELETE HTTP requests.  
There are 2 methods:

* deleteTutorial(): delete a Tutorial document with given id
* deleteAllTutorials(): remove all documents in tutorials collection

The operations is done with the help of MongoRepository’s deleteById() and deleteAll() method.

|  |
| --- |
| @DeleteMapping("/tutorials/{id}")  public ResponseEntity<HttpStatus> deleteTutorial(@PathVariable("id") String id) {  try {  tutorialRepository.deleteById(id);  return new ResponseEntity<>(HttpStatus.NO\_CONTENT);  } catch (Exception e) {  return new ResponseEntity<>(HttpStatus.INTERNAL\_SERVER\_ERROR);  }  }  @DeleteMapping("/tutorials")  public ResponseEntity<HttpStatus> deleteAllTutorials() {  try {  tutorialRepository.deleteAll();  return new ResponseEntity<>(HttpStatus.NO\_CONTENT);  } catch (Exception e) {  return new ResponseEntity<>(HttpStatus.INTERNAL\_SERVER\_ERROR);  }  } |