Spring Boot Explained for Kids

# 1. What is Spring Boot?

Imagine Spring Boot as a magical toy factory for building apps. Instead of putting together all the small pieces yourself, Spring Boot gives you a ready-to-use kit with lots of helpful parts so you can focus on your fun ideas.

# 2. How the App Starts

When you press the ‘run’ button (like starting a toy machine), Spring Boot wakes up and sets everything up for you. It loads all your blueprints (the code you wrote) and gets your app ready to play.

# 3. How the Database Works

Think of the database as a big treasure chest where you store your toys (data). In your code, you make Entity classes (like Actor, Movie, Genre) which are blueprints for how each toy looks. Spring Boot talks to a helper called Hibernate that reads these blueprints and creates matching boxes (tables) in the treasure chest automatically.

When you save a new toy (for example, creating a new Genre or Actor), Hibernate puts it into the right box. When you want to find a toy again, Hibernate goes to the chest and fetches it for you.

# 4. Configuration with application.properties

You have a special settings file called `application.properties`. It’s like the control panel for your toy factory. You tell it which chest (database) to use, how often to update the blueprints, and how much to log what’s happening.

# 5. How API Endpoints Work

API Endpoints are like doors in your toy factory where you can shout commands. When someone sends a request (like HTTP GET or POST), Spring Boot hears it at a door (controller), you handle the command in your code (service), and then you send back a toy (response in JSON). For example, calling `/api/movies` asks, “Please give me the list of movies,” and your code opens the door and hands over the list.

# 6. Final Tips

1. Entities are your blueprints.  
2. Repositories are helpers that save and fetch toys.  
3. Services hold the rules for playing with toys.  
4. Controllers are the doors that listen for playtime requests.  
5. application.properties is your control panel.  
Together, these parts let you build a fun, working app with Spring Boot!

**The typical flow for a new “Register User” feature would look like this:**

 **Entity** (e.g. User.java in entity/)

* Annotated with @Entity and @Table
* Fields like id, username, email, passwordHash, etc.
* This class defines **how your users table looks** in the database.

 **Repository** (e.g. UserRepository.java in repository/)

* An interface that extends JpaRepository<User,Long>
* Inherits methods like save(), findById(), findAll(), delete(), plus any custom query methods you declare (e.g. findByEmail(String email)).

 You don’t have to write any SQL or boilerplate CRUD code yourself.

 Under the hood, Spring will create a bean that implements UserRepository with methods like:

* User save(User user)
* Optional<User> findById(Long id)
* List<User> findAll()
* void deleteById(Long id)
* …and many more.

you’re telling Spring Data JPA two things:

1. **What entity you’re working with** (User)
2. **What type the entity’s primary key (ID) is** (Long)

 **Service** (e.g. RegisterUserService.java in service/)

* Annotated with @Service
* Depends on UserRepository (injected via constructor)
* Contains business logic:
  + register(...) to check email uniqueness, hash passwords, and then repository.save()
  + getById(), delete(), maybe activateAccount(), etc.

 **Controller** (e.g. RegisterUserController.java in controller/)

* Annotated with @RestController and @RequestMapping("/api/users")
* Depends on RegisterUserService (injected via constructor)
* Exposes HTTP endpoints:
  + @PostMapping → calls service.register(...)
  + @GetMapping("/{id}") → calls service.getById(id)
  + @DeleteMapping("/{id}") → calls service.delete(id)

# Entity explanation

**Package** com.example.movies.API.entity;  
this file lives in the folder com/example/movies/API/entity  
  
**Imports** are like grabbing tools from a toolbox so you can use them in this file:

* jakarta.persistence.\* gives you the building blocks to define database tables.
* LocalDate lets you work with dates nicely.
* HashSet & Set let you keep a list of things (without duplicates).

@Entity

@Table(name = "actors")

public class Actor {

 @Entity → “Hey Hibernate, this is a thing I want to store in the database.”

 @Table(name = "actors") → “Store all Actors in a table called actors.”

 public class Actor → “Here begins the recipe (class) for an Actor.”

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

* private Long id;  
  @Id → “This field is the unique name-tag (primary key) for each actor.”
* @GeneratedValue(...) → “Let the database pick a new number for each Actor (auto-increment).”
* private Long id; → “Store that number in a box called id.”

@Column(nullable = false)

private String name;  
 @Column(nullable = false) → “Make a column in the table and don’t allow it to be empty.”

 private String name; → “Each Actor has a name (text).”

@Column(nullable = false)

* private LocalDate birthDate;
  + Must exist (not null).
  + Stored as a LocalDate (year-month-day) in a box called birthDate.

@ManyToMany(mappedBy = "actors")

private Set<Movie> movies = new HashSet<>();  
 @ManyToMany(mappedBy = "actors") → “Actors and Movies are friends and can have many of each other.”

 mappedBy = "actors" → “The Movie side owns the relationship, so just listen for it here.”

 private Set<Movie> movies = new HashSet<>(); → “This Actor has a bag (Set) of Movies they appeared in.”

* protected Actor() { /\* JPA \*/ }  
  A **no-argument constructor** that Hibernate needs behind the scenes.
* Marked protected so regular code can’t accidentally use it.

public Actor(String name, LocalDate birthDate) {

this.name = name;

this.birthDate = birthDate;

}  
A **convenient constructor** for you: “Give me a name and birthDate, and I’ll make a new Actor.”  
 public Long getId() { return id; }

public String getName() { return name; }

public void setName(String name) { this.name = name; }

public LocalDate getBirthDate() { return birthDate; }

public void setBirthDate(LocalDate birthDate) { this.birthDate = birthDate; }

public Set<Movie> getMovies() { return movies; }

* }  
  **Getter methods** (getX) let other parts of your code peek inside those private boxes.
* **Setter methods** (setX) let you change some of those boxes (like name or birthDate).
* No setter for id → once the database gives an id, you don’t change it.
* You can only **get** the list of movies—to add/remove a movie, you manipulate that Set<Movie> directly.

### In summary

* This class tells Hibernate “please make me an actors table with columns id, name, birthDate.”
* It also sets up a “many-to-many” link to the movies table.
* The constructors and getters/setters let you create and manipulate Actor objects in your Java code.

ActorController step by step, in the simplest terms possible—think of this class as the “front door” to your application for anything having to do with Actors. I’ll show each bit of code and explain what it’s doing.  
  
**Imports** bring in other code you need:

* Actor is your data blueprint (so you can accept it as JSON and return it).
* ActorService is the helper that does all the “work” (talking to the database).
* @RestController, @RequestMapping, @PostMapping, etc., are Spring annotations that turn simple methods into web endpoints.
* HttpStatus lets you choose HTTP response codes (e.g. 201 Created).
* LocalDate & List are basic Java types you’ll use in method signatures.

@RestController

@RequestMapping("/api/actors")

* public class ActorController {  
  @RestController
  + **Tells Spring:** “This class has methods that should listen for HTTP requests and return data (usually as JSON).”
* @RequestMapping("/api/actors")
  + **Tells Spring:** “All the URLs in here start with /api/actors.”
  + So if a method is annotated @GetMapping, it really listens at /api/actors + that method’s path.

private final ActorService actorService;

public ActorController(ActorService actorService) {

this.actorService = actorService;

}  
 **Field (actorService)**

* Holds a reference to your Service layer (the part that actually knows how to create, fetch, update, delete actors in the database).

 **Constructor**

* Spring Boot will automatically “give” you an ActorService when it makes this controller.
* Think of this like giving your front-door guard (ActorController) a phone line (actorService) to call the database helpers.
*  **ActorService** (in the **service** folder) is like the “expert team” that knows how to save, fetch, update, or delete actors in the database.
*  **ActorController** (in the **controller** folder) is like the “reception desk” that greets web requests and then calls the expert team to do the work.

private final ActorService actorService;  
Think: “I, the reception desk, have a phone line to the expert team.”

public ActorController(ActorService actorService) {

this.actorService = actorService;

}  
 Think: “When you build the reception desk, you hand it the phone line to the experts.”

 **You don’t create the phone line yourself**—someone else (Spring) does, and gives it to the desk.

**Spring’s role**

* Because you marked ActorService with @Service, Spring says:  
  “I will build exactly one expert team (one ActorService object) and keep it in my “registry.””
* Because you asked in your constructor for an ActorService, Spring finds that one object and hands it to your controller when it builds the reception desk.

Inside **ActorController**, anytime you need to work with actors, you use that actorService phone line:

* **Greet a request to list actors**

java

CopyEdit

actorService.getAll();

“Hello expert team, please give me all actors.”

* **Greet a request to create a new actor**

java

CopyEdit

actorService.create(newActor);

“Hello expert team, please save this new actor for me.”

### You never do new ActorService() inside the controller—Spring already did that for you once, and you just use the provided instance. Why this is good

* **Single shared expert team**: There’s exactly one ActorService handling all requests, so it can manage transactions, caching, or shared resources.
* **Clear separation**: The controller only deals with web details (URLs, HTTP methods), and the service only deals with business logic and the database.
* **Testable**: You could hand a fake “expert team” (a mock ActorService) to the controller in a unit test, without using a database.

@PostMapping

@ResponseStatus(HttpStatus.CREATED)

public Actor create(@RequestBody Actor actor) {

return actorService.create(actor);

1. }  
   @PostMapping
   * **Listens for** HTTP **POST** requests at /api/actors (because of the class-level path).
2. @ResponseStatus(HttpStatus.CREATED)
   * **Says** “When this method finishes successfully, reply with HTTP status 201 (Created).”
3. public Actor create(@RequestBody Actor actor)
   * **Parameters**
     + @RequestBody Actor actor means “take the JSON body of the request, turn it into an Actor object.”
   * **Body**
     + Calls actorService.create(actor) to save it in the database.
   * **Return value**
     + Returns the saved Actor (with its new database ID) back to the caller as JSON.

@GetMapping

public List<Actor> getAll(@RequestParam(required = false) String name) {

if (name != null) {

return actorService.findByName(name);

}

return actorService.getAll();

}  
 @GetMapping

* **Listens for** HTTP **GET** requests at /api/actors.

 @RequestParam(required = false) String name

* **Reads** an optional query parameter ?name=.... If present, Spring puts its value into the name variable.

 **Logic**

* If name was provided, call findByName(name) to do a search.
* Otherwise, call getAll() to list every actor.

 **Returns** a list of Actor objects as JSON.

@GetMapping("/{id}")

public Actor getById(@PathVariable Long id) {

return actorService.getById(id);

1. }  
   @GetMapping("/{id}")
   * **Listens for** GET at /api/actors/123 (where 123 is any number).
2. @PathVariable Long id
   * **Takes** the 123 from the URL and puts it into the id parameter.
3. **Body**
   * Calls actorService.getById(id), which either finds the actor or throws a “not found” error.
4. **Return**
   * Sends back the found Actor as JSON.

@PatchMapping("/{id}")

public Actor update(

@PathVariable Long id,

@RequestBody Actor partial

) {

return actorService.update(

id,

partial.getName(),

partial.getBirthDate()

);

}  
 @PatchMapping("/{id}")

* **Listens for** HTTP **PATCH** at /api/actors/123. PATCH means “I want to change just these fields.”

 **Parameters**

* @PathVariable Long id gets the 123.
* @RequestBody Actor partial turns the JSON body (which might only have name or birthDate) into an Actor object where only those fields are set.

 **Body**

* Calls actorService.update(id, newName, newBirthDate).

 **Return**

* The updated Actor as JSON.

@DeleteMapping("/{id}")

@ResponseStatus(HttpStatus.NO\_CONTENT)

public void delete(

@PathVariable Long id,

@RequestParam(defaultValue = "false") boolean force

) {

actorService.delete(id, force);

}

}  
 @DeleteMapping("/{id}")

* **Listens for** HTTP **DELETE** at /api/actors/123.

 @ResponseStatus(HttpStatus.NO\_CONTENT)

* **Replies** with HTTP 204 (No Content) on success—meaning “the delete worked, and there’s nothing to show.”

 **Parameters**

* @PathVariable Long id for the actor’s ID.
* @RequestParam(defaultValue = "false") boolean force reads ?force=true or defaults to false.

 **Body**

* Calls actorService.delete(id, force), which either removes the actor or throws an error if it’s in use and force is false.

 **Return**

* void means no JSON body; just the 204 status code.

### Putting it all together

* **Controller** methods map HTTP verbs (GET, POST, PATCH, DELETE) + URLs to Java methods.
* Each method converts web data (path variables, query params, JSON bodies) into Java parameters.
* Then it calls the **Service** layer (actorService) to do the real work.
* Finally it returns results (or errors) back to the client as JSON and the appropriate HTTP status code.

That’s how your **ActorController** sits at the “front door”: it listens for web requests about actors, pulls the data you send in, calls the service to talk to the database, and then hands back whatever the service returns.