FULL STACK DEV

Spring IoC Container & Dependency Injection

Presented by:

Rajeev Khoodeeram

OCTOBER 2025

SPRING IOC CONTAINER (1)

- It's responsible for managing the lifecycle of your application's objects (beans) and their dependencies.

- In OO, we used objects - that is objects are created and destroyed when they are no longer used ...this is handled for us by the Spring framework

- In traditional programming, you're responsible for creating and managing the lifecycle of your objects.
    - Student s = new Student();

SPRING IOC CONTAINER (2)

- With IoC, the framework (Spring, in this case) takes over the responsibility of creating, configuring, and managing your objects.

- Instead of you calling the framework, the framework takes control over your objects .

- This "inversion" of control leads to more modular and testable code.

- You will notice so far, we have never called **new** on any of the main Spring classes.
    -

HOW DOES IT WORKS ?

- The Spring IoC container reads your configuration metadata (that is your annotations) and uses it to instantiate, configure, and assemble the objects.

- **@Service**: Indicates that an annotated class is a "Service". It's typically used for classes that encapsulate business logic.

- For example :
- @Service
- public class StudentService {
- // ... business logic for students
- }

ANNOTATIONS - @REPOSITORY

- @Repository: Indicates that an annotated class is a "Repository". It is typically used for classes that directly interact with the database (e.g., Data Access Objects - DAOs).

- It also enables automatic exception translation from persistence-specific exceptions to Spring's DataAccessException hierarchy.

- For example
- **@Repository**
- public class StudentRepository {
-   // ... database interaction
- }

CONTROLLER ANNOTATIONS

- **@Controller**: Indicates that an annotated class is a "Controller".
- This is used in Spring MVC applications to handle web requests and return views (e.g., Thymeleaf templates).
- For example
- **@Controller**
- public class StudentController {
-   // ... handles web requests, returns view names
- }
-

CONTROLLER ANNOTATIONS

- **@RestController**: A convenience annotation that combines @Controller and @ResponseBody.

- It's primarily used for building RESTful web services, where the methods return data directly (e.g., JSON or XML) rather than view names.

- For example
- **@RestController**
- public class StudentController {
- // ... handles API requests, returns data
- }
-

DEPENDENCY INJECTION

- DI = Injecting required dependencies into a class rather than creating them inside it.

- **Benefits**:
  - Promotes loose coupling
  - Easier unit testing
  - Cleaner, maintainable code

DEPENDENCY INJECTION

- **Without DI** :

```
class StudentService {
  private StudentRepo repo = new StudentRepo();
}
```

- **With DI** :

```
class StudentService {
  private StudentRepo repo;   → is injected here !
  public StudentService(StudentRepo repo) { this.repo = repo; }
}
```

DEPENDENCY INJECTION

- **@Repository**
- public class StudentRepo { ... }
-
- **@Service**
- public class StudentService {
-    **@Autowired**
-    private StudentRepo repo;
- }
-
- **@RestController**
- public class StudentController {
-    **@Autowired**
-    private StudentService service;
- }

FULL STACK DEV

Spring annotations for database operations

Presented by:

Rajeev Khoodeeram

OCTOBER 2025

SPRING BOOT - HTTP REQUESTS

- We have 5 main annotations that we will use more often for database operations for our Java services. These are :
- @RequestMapping → class level (base path)
- @GetMapping → Read
- @PostMapping → Create
- @PutMapping → Update
- @DeleteMapping → Delete

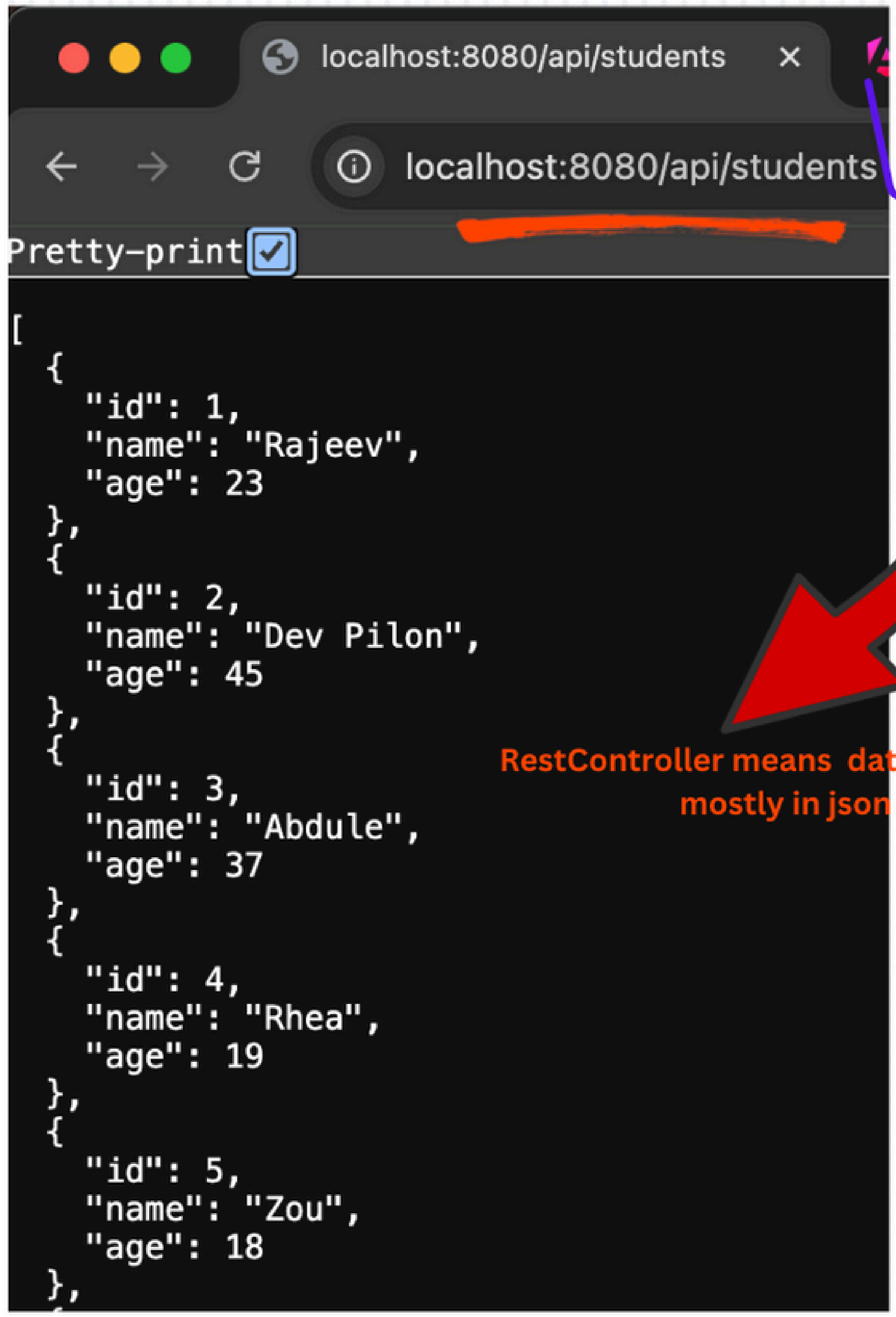
REQUESTMAPPING

- @RequestMapping: A versatile annotation for mapping web requests onto specific handler classes and/or handler methods.

- It can be used at the class level to define a base path for all methods in that controller, and at the method level for specific endpoints.

- @RestController
- **@RequestMapping("/api/students")** // Base path for all methods in this controller
- public class StudentController {
-   // ...
- }
  -

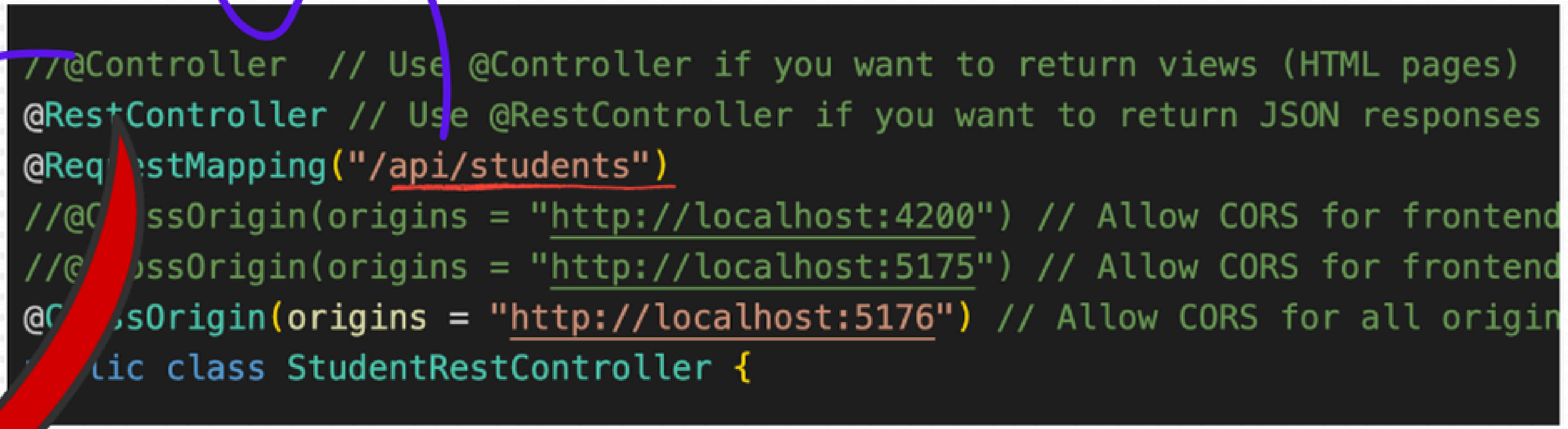
REQUESTMAPPING



localhost:8080/api/students

Pretty-print ✓

```
[
  {
    "id": 1,
    "name": "Rajeev",
    "age": 23
  },
  {
    "id": 2,
    "name": "Dev Pilon",
    "age": 45
  },
  {
    "id": 3,
    "name": "Abdule",
    "age": 37
  },
  {
    "id": 4,
    "name": "Rhea",
    "age": 19
  },
  {
    "id": 5,
    "name": "Zou",
    "age": 18
  },
```



```
//@Controller  // Use @Controller if you want to return views (HTML pages)
@RestController // Use @RestController if you want to return JSON responses
@Req  stMapping("/api/students")
//@C  ssOrigin(origins = "http://localhost:4200") // Allow CORS for frontend
//@  ossOrigin(origins = "http://localhost:5175") // Allow CORS for frontend
@C  sOrigin(origins = "http://localhost:5176") // Allow CORS for all origin
   lic class StudentRestController {
```

RestController means data will be returned - mostly in json format

HTTP SPECIFIC : GETMAPPING

- @GetMapping: Maps HTTP GET requests. Used for retrieving resources.
- @GetMapping // Maps to /api/students (if @RequestMapping is at class level)
-

```
@GetMapping
public ResponseEntity<List<Student>> getAllStudents() {
List<Student> students = studentService.getAllStudents();

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

HTTP SPECIFIC : POSTMAPPING

- @PostMapping: Maps HTTP POST requests. Used for creating new resources.
-
- @PostMapping // Maps to /api/students
- public Product createStudent(@RequestBody Student newStudent) {
-   // ... save new student
-   return newSudent;
- }

HTTP SPECIFIC : PUTMAPPING

- @PutMapping: Maps HTTP PUT requests.

- Used for updating existing resources (often for full replacement of a resource).

- @PutMapping("/{id}") // Maps to /api/students/{id}
- public Product updateStudent(@PathVariable Long id, @RequestBody Student updatedStudent) {
-   // ... update student by ID
-   return updatedStudent;
- }

HTTP SPECIFIC : DELETEMAPPING

- @DeleteMapping: Maps HTTP DELETE requests. Used for deleting resources.

- @DeleteMapping("/{id}") // Maps to /api/students/{id}
- public ResponseEntity<Void> deleteStudent(@PathVariable Long id) {
-   // ... delete student by ID
-   return ResponseEntity.noContent().build(); // Return 204 No Content
- }

FULL STACK DEV

spring boot

HTTP Status codes

Presented by:

Rajeev Khoodeeram

OCTOBER 2025

HTTP STATUS CODES

- Returning appropriate HTTP status codes is crucial for building well-behaved RESTful APIs.

- It provides clear communication to the client about the outcome of their request.
  - 200 OK: The request was successful. (e.g., GET, PUT, POST success)
  - 201 Created: The request has been fulfilled and resulted in a new resource being created. (e.g., POST success for resource creation)
  - 204 No Content: The server successfully processed the request and is not returning any content. (e.g., DELETE success)
  -

HTTP STATUS CODES

- 400 Bad Request: The server cannot process the request due to client error (e.g., malformed request syntax, invalid request message framing, or deceptive request routing).

- 401 Unauthorized: Authentication is required and has failed or has not yet been provided.

- 403 Forbidden: The server understood the request but refuses to authorize it. (e.g., insufficient permissions)

HTTP STATUS CODES

- 404 Not Found: The requested resource could not be found.

- 405 Method Not Allowed: The request method is known by the server but has been disabled and cannot be used.

- 500 Internal Server Error: A generic error message, given when an unexpected condition was encountered and no more specific message is suitable.

EXAMPLE 1

- // Example: GET a resource
- @**GetMapping**("/{id}")
- public ResponseEntity<String> getItem(@PathVariable Long id) {
-   if (id == 1L) {
-     return new ResponseEntity<>("Item Found!", HttpStatus.OK); // **200** OK
-   } else {
-     return new ResponseEntity<>("Item Not Found", HttpStatus.NOT_FOUND); // **404** Not Found
-   }
- }

EXAMPLE 2

- // Example: POST to create a resource

- **@PostMapping**
- public ResponseEntity<String> createItem(@RequestBody String itemDetails) {
- // Logic to save item
- System.out.println("Creating item: " + itemDetails);
- return new ResponseEntity<>("Item Created Successfully", HttpStatus.CREATED); // **201** Created
- }

EXAMPLE 3

- // Example: DELETE a resource

- @**DeleteMapping**("/{id}")
- public ResponseEntity<Void> deleteItem(@PathVariable Long id) {
-   // Logic to delete item
-   System.out.println("Deleting item with ID: " + id);
-   return new ResponseEntity<>(HttpStatus.NO_CONTENT); // **204** No Content
- }
- }