Spring Boot

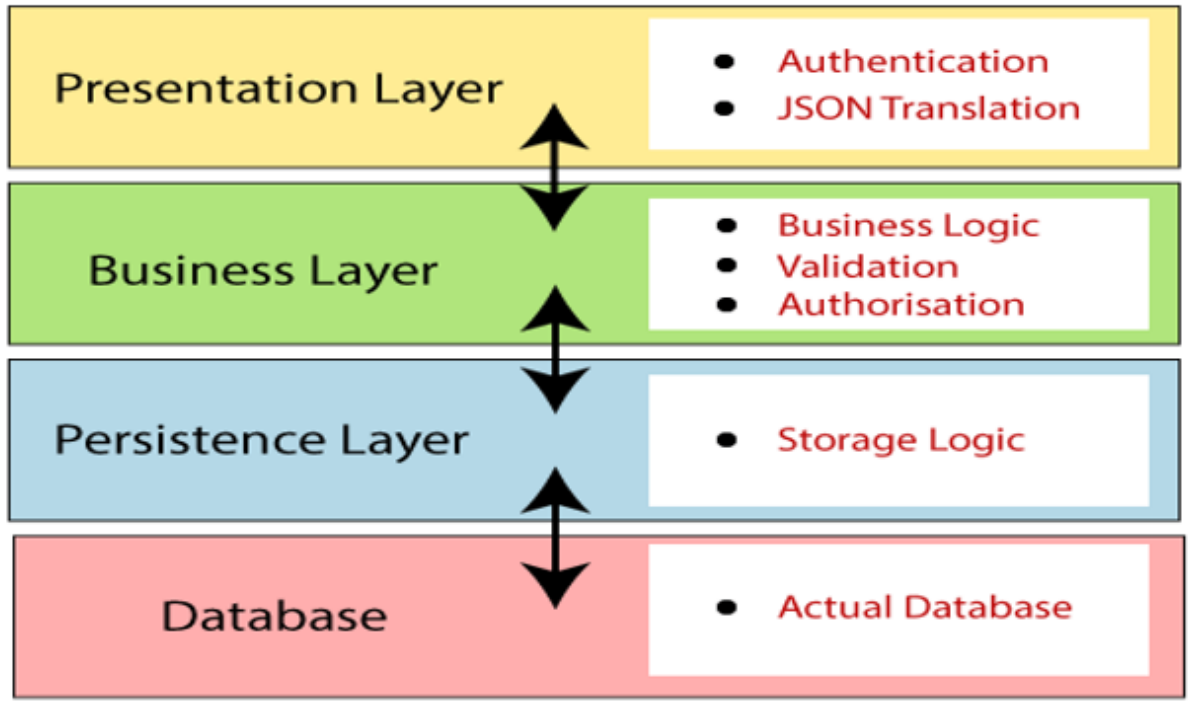
Spring Boot

- Spring Boot is a project that is built on the top of the Spring Framework.
- It provides an easier and faster way to set up, configure, and run both simple and web-based applications.
- Spring Boot is a Spring module that provides the RAD (Rapid Application Development) feature to the Spring framework.

Spring Boot Architecture

There are **four** layers in Spring Boot are as follows:

- Presentation Layer
- Business Layer
- Persistence Layer
- Database Layer

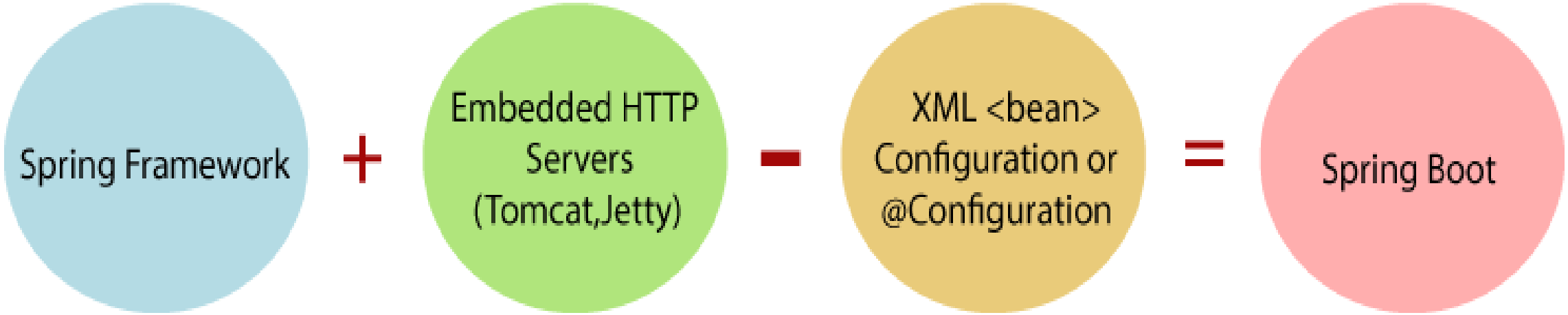


Presentation Layer: The presentation layer handles the HTTP requests, translates the JSON parameter to object, and authenticates the request and transfer it to the business layer. In short, it consists of **views** i.e., frontend part.

Business Layer: The business layer handles all the **business logic**. It consists of service classes and uses services provided by data access layers. It also performs **authorization** and **validation**.

Persistence Layer: The persistence layer contains all the **storage logic** and translates business objects from and to database rows.

Database Layer: In the database layer, **CRUD** (create, retrieve, update, delete) operations are performed.



In short, Spring Boot is the combination of **Spring Framework** and **Embedded Servers**.

Spring Framework	Spring Boot Framework
Spring Framework is a widely used Java EE framework for building applications.	**Spring Boot Framework** is widely used to develop **REST APIs**.
It simplifies Java EE development, boosting developer productivity.	It aims to shorten the code length and provide the easiest way to develop Web Applications.
The primary feature of the Spring Framework is **dependency injection**.	The primary feature of Spring Boot is **Autoconfiguration**.
The developer writes a lot of code (**boilerplate code**) to do the minimal task.	It **reduces** boilerplate code.
It does not provide support for an in-memory database.	It offers several plugins for working with an embedded and **in-memory** database such as **H2**.
Developers manually set dependencies in **`pom.xml`** for Spring projects.	Spring Boot's starters in **`pom.xml`** automatically handle dependency downloads.

Advantages of Spring Boot

✅ 1. Auto Configuration

No need for manual XML config — Spring Boot auto-configures beans, JPA, DB, etc.

✅ 2. Embedded Server

Runs with built-in Tomcat/Jetty — no need to deploy WAR to external server.

✅ 3. Starter Dependencies

Provides pre-configured starters (e.g., web, JPA) — avoids version conflicts and saves setup time.

✅ 4. Production-Ready Features

Includes built-in monitoring, health checks, metrics via Spring Actuator.

✅ 5. Faster Development & Microservice Friendly

Quickly build REST APIs and microservices with minimal boilerplate and setup.

✅ Why Spring Boot is a combination of other Spring Projects:

1. **Spring Boot + Spring Web** → Easily build REST APIs with `@RestController`.
2. **Spring Boot + Spring Data JPA** → Simplifies database access using repositories.
3. **Spring Boot + Spring Security** → Provides built-in authentication and authorization.
4. **Spring Boot + Spring Batch** → Supports large-scale batch processing jobs.
5. **Spring Boot + Spring Cloud** → Helps build scalable microservices with service discovery, config, etc.
6. **Spring Boot + Spring Boot Actuator** → Adds ready-to-use monitoring and health check endpoints.

XML

- XML -Extensible Markup Language
- It was designed to store and transport data
- HTML – used for Web pages
- XML – used for store and transport data
- Both Humans and Machine can Understand.
- You can create your tags
- Html is presentation language
- Xml is descriptive language

```
<my-data>

   <sno>1</sno>

   <name> Surya </name>

   <place> Raj </place>

</my-data>
```

```
<?xml version="1.0"?>
<my course  year="first">

        <subject>
                <id>s1</id>
                <name>XML</name>
                <credits>4</credits>
        </subject>

        <subject>
                <id>s2</id>
                <name>JAVA</name>
                <credits>4</credits>
        </subject>

        <subject>
                <id>s3</id>
                <name>Python</name>
                <credits>4</credits>
        </subject>


</mycourse>
```

XML Rules

1. Must have a starting a **starting & closing tag**
2. XML tags are **case-sensitive.**
3. XML elements must be properly **nested**
4. Must have a **root element** and only one **root element**
5. XML attributes values must be **quoted**
6. XML tag names must not contains **spaces**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<my-data>
  <student>
    <sno>1</sno>
    <name> Surya </name>
    <place> Rjy </place>
  </student>
  <student>
    <sno>1</sno>
    <name> Surya </name>
    <place> Rjy </place>
  </student>

</my-data>
```

- **School Name**: Sunrise Public School
- **Class**: 10th Grade (Class 10A)
    - **Teacher**: Mrs. Shalini
    - **Students**:
        1. **Name**: Rahul, Age: 15, Roll: 101, Marks: 89
        2. **Name**: Anjali, Age: 14, Roll: 102, Marks: 92
        3. **Name**: Karan, Age: 15, Roll: 103, Marks: 85

XML Namespaces

JSON

- JSON stands for **J**ava**S**cript **O**bject **N**otation
- JSON is a **text format** for storing and transporting data
- JSON is "self-describing" and easy to understand
- The JSON syntax is derived from JavaScript object notation, but the JSON format is text only.
- Code for reading and generating JSON exists in many programming languages.
- If want two Applications Communicate each other over internet we need common data Exchange format, irrespective of any Frame works
- That Data exchange format is called Json or XML

```
{
    "rollno": 1,
    "firstName": "John",
    "lastName": "Doe",
    "email": "johndoe@example.com"
}
```

JSON supports the following data types:

- **String**: "Alice"
- **Number**: 25, 5.7
- **Boolean**: true, false
- **Null**: null
- **Object**: { "key": "value" }
- **Array**: ["value1", "value2"]

```
{
  "College":"Samhitha",
  "place":"Hyd",
  "Year":"24/06/2024",
  "Students":[
              "Ramakrishna","ravi","rao"]
}
```

🧠 JSON Structure Task:

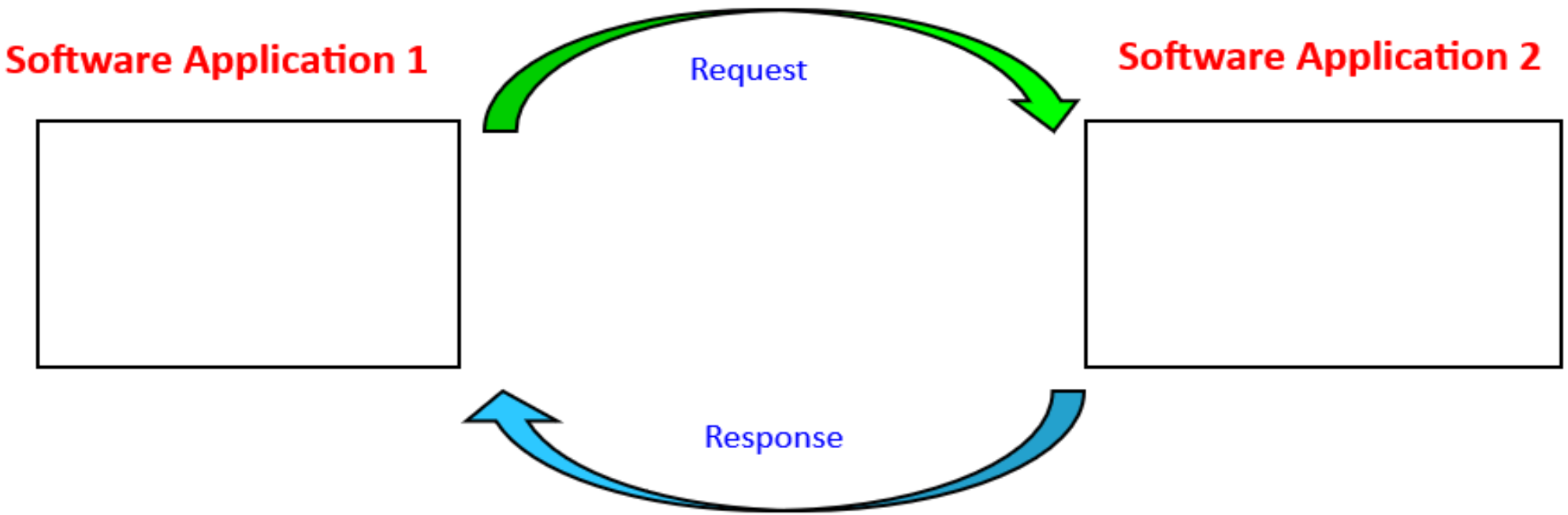
Represent the following data using JSON:

🎯 Sample Data

- **School Name**: Sunrise Public School
- **Class**: 10A
    - **Teacher**: Mrs. Shalini
    - **Students**:
        - Rahul (Age: 15, Roll: 101, Marks: 89)
        - Anjali (Age: 14, Roll: 102, Marks: 92)
        - Karan (Age: 15, Roll: 103, Marks: 85)

API - Application Programming Interface

- API are Mechanism that enable two software applications communicate each other using set of definitions and protocols.
- This two software should be any where either internet or local environment.
- Example SignInWithGoogle, Phone Pay etc.....



- **Private APIs** are intended for use only within a specific organization, providing access to internal systems and data.
- **Public APIs** are available to external developers, allowing them to interact with and integrate third-party applications and services.

Rest API

Rest API

- REST Stands for Representational State Transfer.
- It is Architectural pattern.
- Rest Defines a set of functions Get, Put, Delete, Patch etc..
- Converting data's State in database into representational from (Json/xml) and transfer as response from server to client.
- Client server Exchange Data using Http.

The important methods of HTTP are:

- **GET:** It reads a resource.
- **PUT:** It updates an existing resource.
- **POST:** It creates a new resource.
- **DELETE:** It deletes the resource.
- **PATCH:** It updates semi existing Data

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

REST Architectural Constraints

REST Architectural Constraints

- ✓ Client-server
- ✓ Stateless
- ✓ Cache(http Cache)
- ✓ Uniform Interface
- ✓ Layered System
- ✓ Code on Demand

HTTP also defines the:

- **404:** RESOURCE NOT FOUND
- **200:** SUCCESS
- **201:** CREATED
- **300:** INFORMATIONAL
- **401:** UNAUTHORIZED
- **500:** SERVER ERROR

Spring Boot Starters

Think of **starters** like **combo packs** or **starter kits**. You don't have to add each dependency one by one — just add 1 starter and you're ready to go.

```
  <dependency>spring-core</dependency>
  <dependency>spring-web</dependency>
  <dependency>spring-context</dependency>
  <dependency>jackson</dependency>
```

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

Spring Boot Autoconfiguration

Spring Boot **auto-magically configures** your project. You just focus on writing business logic, it configures beans, properties, etc., based on what's present in your project.

- `spring-boot-starter-web` ➡️ it will auto-create **DispatcherServlet**, **Tomcat server**, **Jackson**, etc.
- A DB dependency and `application.properties` ➡️ it auto-configures **DataSource**, **JPA repo**, etc.

Spring Boot DevTools

DevTools = Developer Power Tools

- Auto restart the server when you save a file.
- Live reloads in browser.
- Easier debugging and faster feedback loop.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-devtools</artifactId>
  <scope>runtime</scope>
</dependency>
```

❌ 1. Performance Killer

❌ 2. Security Risk

❌ 3. Memory Usage

Spring Boot Actuator

Actuator exposes internal metrics and health info of your app. Basically shows "What's going on inside?"

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

`/actuator/health` → App health status

`/actuator/info` → App info

`/actuator/metrics` → Memory, CPU, etc.

application.properties

```
management.endpoints.web.exposure.include=*
management.endpoint.health.show-details=always
```

✅ Safe Prod Config:

```properties
management.endpoints.web.exposure.include=health,info
management.endpoint.health.show-details=when-authorized
management.endpoint.env.enabled=false
management.endpoint.beans.enabled=false
management.endpoint.mappings.enabled=false
```

🔐 Add Security:

```properties
management.endpoints.web.base-path=/actuator
management.endpoint.health.roles=ADMIN
```

👉 Combine this with Spring Security to restrict access to `/actuator/*`.

Thymeleaf

Thymeleaf is a **server-side template engine** used in Spring Boot to render dynamic HTML pages.

💡 Why use Thymeleaf?

✅ Works with Spring MVC
✅ Uses **HTML5 syntax** (your file is real HTML!)
✅ Can show dynamic data using `${...}`
✅ Has conditions, loops, form binding, etc.
✅ Good for making **login pages, dashboards, admin panels**, etc.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

What is Thymeleaf

1. Thymeleaf is a modern server-side Java template engine for both web and standalone environments, capable of processing HTML, XML, JavaScript, CSS and even plain text.
2. The main goal of Thymeleaf is to provide an elegant and highly-maintainable way of creating templates.
3. It's commonly used to generate HTML views for web applications.
4. Thymeleaf is the best choice for developing Spring MVC web applications.

What kind of Templates can Thymeleaf process?

Out-of-the-box, Thymeleaf allows you to process six kinds of templates:

1. HTML
2. XML
3. TEXT
4. JAVASCRIPT
5. CSS
6. RAW (plain text)

How Thymeleaf Engine Works

Thymeleaf Standard Expressions

Five types of Thymeleaf standard expressions:

1. ${...} : Variable expressions
2. *{...} : Selection expressions
3. #{...} : Message (i18n) expressions
4. @{...} : Link (URL) expressions
5. ~{...} : Fragment expressions

Variable Expressions

Variable expressions are the most commonly used ones in thymeleaf templates. These expressions help bind the data from the template context(model) into the resulting html(view).

Syntax:

${VariableName}

Variable Expressions Example

Consider we have added data to model in Spring MVC controller and in order to access the model data, we use thymeleaf Variable expressions:

```java
@Controller
public class MessageController {

    @GetMapping("message")
    public String message(Model model){
        model.addAttribute( attributeName: "message",  attributeValue: "Hello World!");
        return "message";
    }
}
```

Model variable Name

Model variable Value

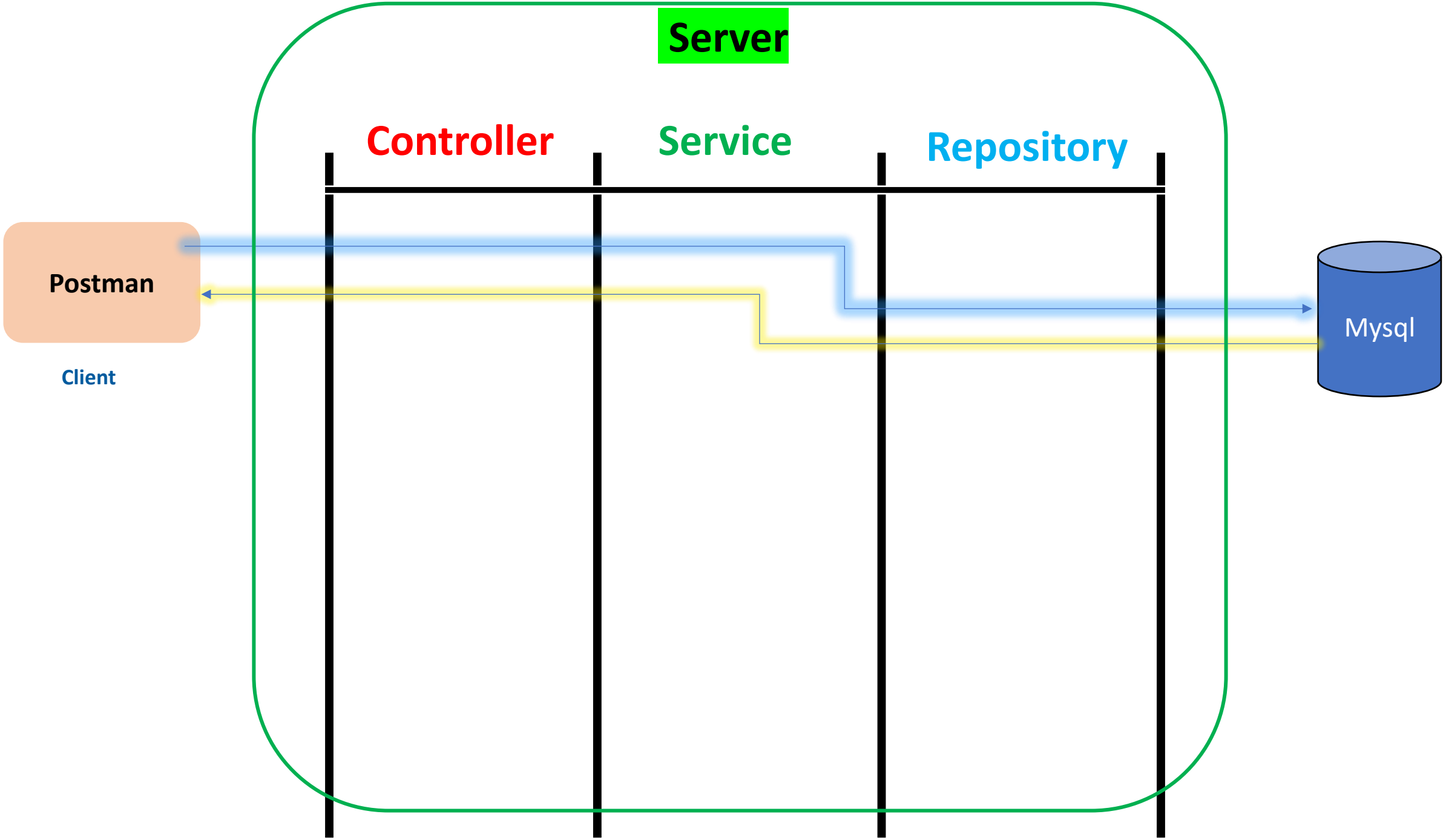
```html
<!DOCTYPE html>
<html lang="en"
      xmlns:th="http://www.thymeleaf.org"
>
<head>
    <meta charset="UTF-8">
    <title>Variable Expression</title>
</head>
<body>
    <h1 th:text="${message}"></h1>
</body>
</html>
```

Variable expression

Custom Queries

⚡ Common Patterns with `%`

Pattern	Meaning	Example Match
`'Sur%'`	Starts with "Sur"	`Surya`, `Suresh`, `Sur123`
`'%ya'`	Ends with "ya"	`Surya`, `Arya`, `Divya`
`'%ur%'`	Contains "ur"	`Surya`, `Gurman`, `Bureau`
`'%'`	Everything (any string)	Matches all rows


Server
Controller
Service
Repository
Postman
Client
Mysql

```sql
SELECT *
FROM your_table
WHERE email LIKE '%@gmail.com';
```

```java
List<User> findByNameIgnoreCase(String name);

@Query(value = "SELECT * FROM user_table WHERE email LIKE CONCAT('%', :domain, '%')", nativeQuery = true)
// JPQL: SELECT u FROM User u WHERE u.email LIKE CONCAT('%', :domain, '%')
List<User> findUsersByEmailDomain(@Param("domain") String domain);


@Query(value = "SELECT * FROM user_table WHERE name = :name AND age = :age", nativeQuery = true)
// JPQL: SELECT u FROM User u WHERE u.name = :name AND u.age = :age
List<User> findByNameAndAge(@Param("name") String name, @Param("age") int age);


@Query(value = "SELECT * FROM user_table WHERE age > :age", nativeQuery = true)
// JPQL: SELECT u FROM User u WHERE u.age > :age
List<User> findUsersOlderThan(@Param("age") int age);


@Query(value = "SELECT * FROM user_table WHERE email LIKE '%.com'", nativeQuery = true)
// JPQL: SELECT u FROM User u WHERE u.email LIKE '%.com'
List<User> findUsersWithComEmail();
```

```java
List<User> findByName(String name);

List<User> findByNameIgnoreCase(String name);

List<User> findByEmailContaining(String text);

List<User> findByAgeGreaterThan(int age);

List<User> findByActiveTrue();

List<User> findByNameAndAge(String name, int age);

List<User> findByNameOrEmail(String name, String email);
```

`@OneToOne`

1 user ↔ 1 profile

In `@OneToOne` → Foreign key goes in any one table

```java
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String username;
    private String email;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "profile_id")  // foreign key column
    private Profile profile;
}
```

```java
public class Profile {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String bio;
    private String address;
}
```

```json
{
  "username": "surya99",
  "email": "surya@mail.com",
  "profile": {
    "bio": "Java Full Stack Dev",
    "address": "Chennai"
  }
}
```

`cascade = CascadeType.ALL`

Cascade means: "When I do something to `User`, also do it to `Profile`"

`CascadeType.ALL` = includes:

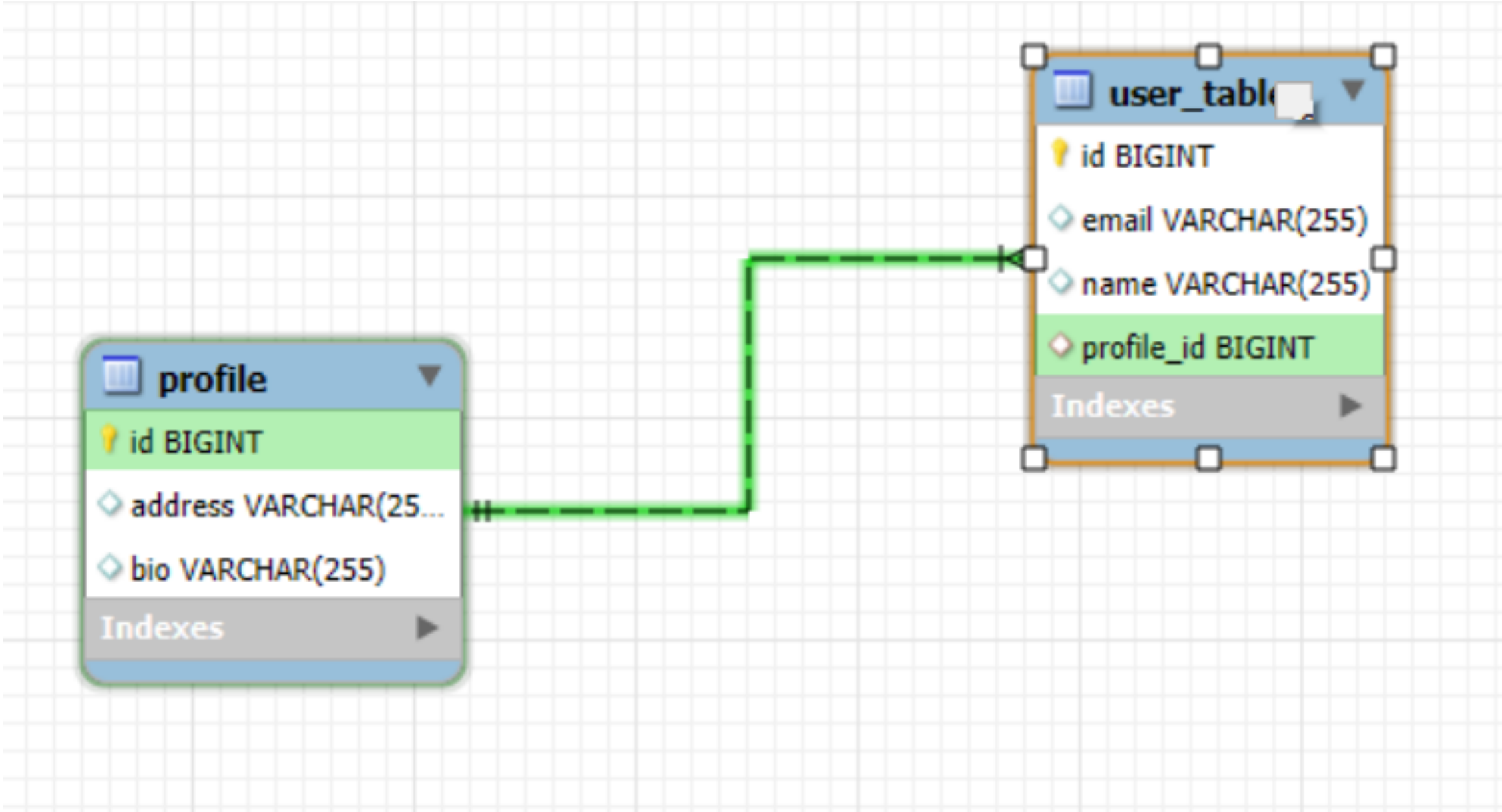
- `PERSIST` → save user ➜ save profile
- `MERGE` → update user ➜ update profile
- `REMOVE` → delete user ➜ delete profile (⚠️ careful!)
- `DETACH`, `REFRESH`, etc.

`@JoinColumn(name = "profile_id")`

Tells JPA to **create a foreign key** column in the `user` table

It will create a column in the `user` table called `profile_id`

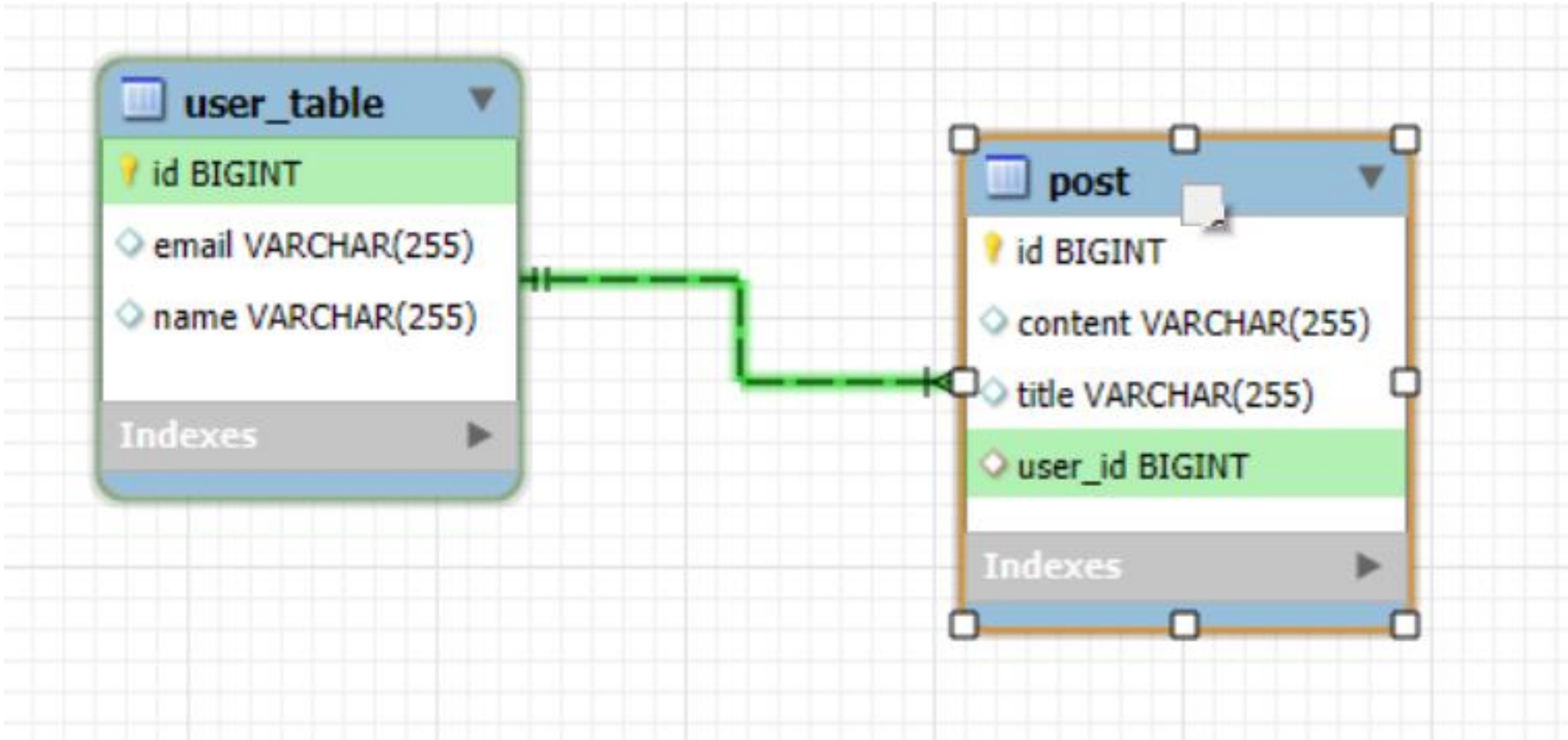
which points to the `profile.id`


user_table
id BIGINT
email VARCHAR(255)
name VARCHAR(255)
profile_id BIGINT
Indexes
profile
id BIGINT
address VARCHAR(25...
bio VARCHAR(255)
Indexes

`@OneToMany` → FK must go on the "many" side (child table)

```java
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    // One User ➡️ Many Posts
    @OneToMany(
        mappedBy = "user",
        cascade = CascadeType.ALL,
        fetch = FetchType.LAZY
    )
    private List<Post> posts;
}
```

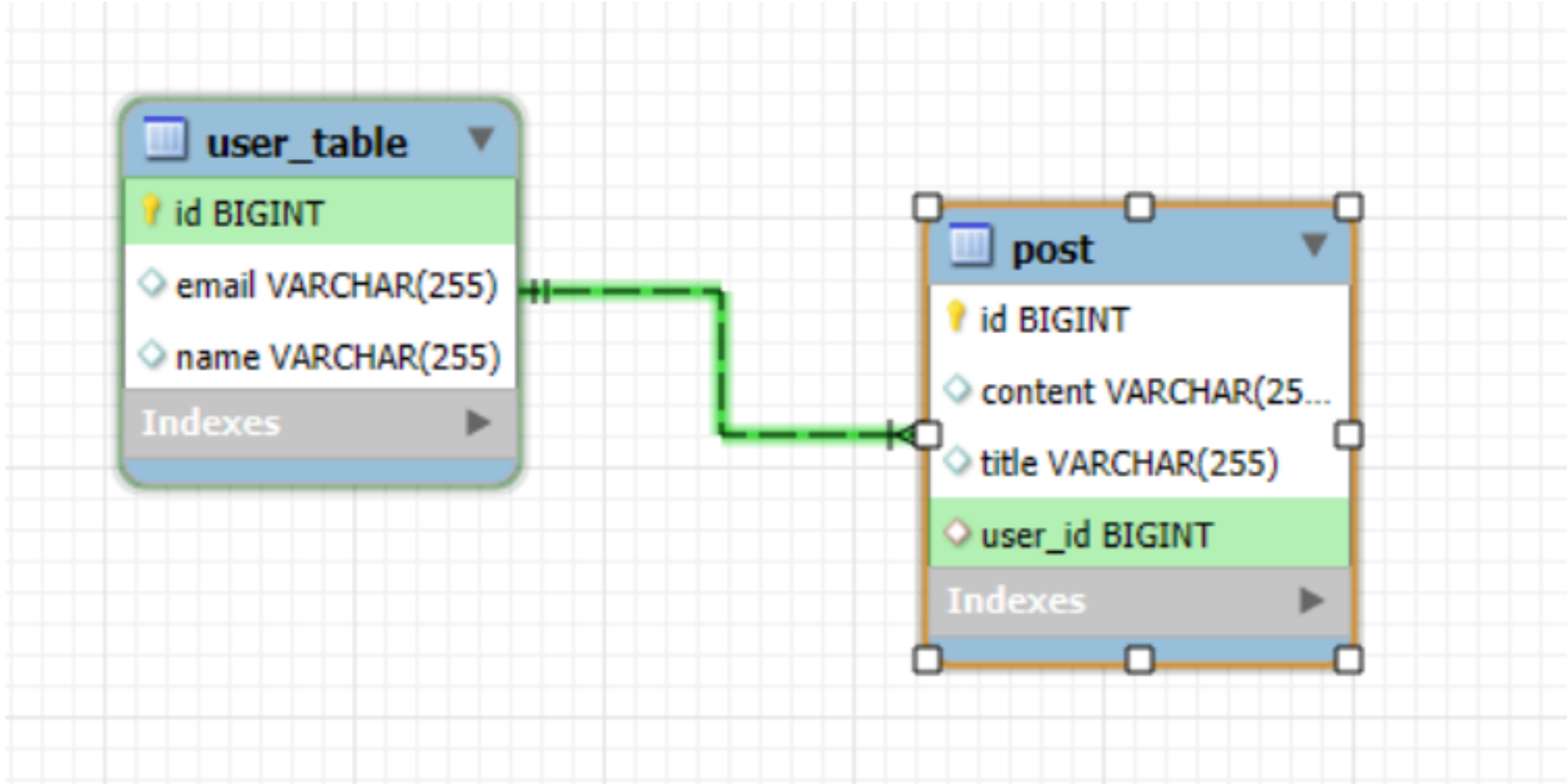
```java
public class Post {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String title;
    private String content;

    // Many Posts belong to One User
    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "user_id")  // foreign key column
    private User user;
}
```


user_table
id BIGINT
email VARCHAR(255)
name VARCHAR(255)
Indexes
post
id BIGINT
content VARCHAR(255)
title VARCHAR(255)
user_id BIGINT
Indexes

@ManyToOne

```java
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String email;
}
```

```java
public class Post {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String title;
    private String content;

    // 👉 Many posts can belong to one user
    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "user_id") // FK created in this table
    private User user;
}
```


user_table
id BIGINT
email VARCHAR(255)
name VARCHAR(255)
Indexes
post
id BIGINT
content VARCHAR(25...
title VARCHAR(255)
user_id BIGINT
Indexes

@ManyToMany

```java
public class Student {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @ManyToMany
    @JoinTable(
        name = "student_subject", // join table name
        joinColumns = @JoinColumn(name = "student_id"),
        inverseJoinColumns = @JoinColumn(name = "subject_id")
    )
    private List<Subject> subjects;
}
```

```java
public class Subject {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String subjectName;

    @ManyToMany(mappedBy = "subjects")
    private List<Student> students;
}
```

`@JoinTable(...)`

We're creating a **third table** called `student_subject`

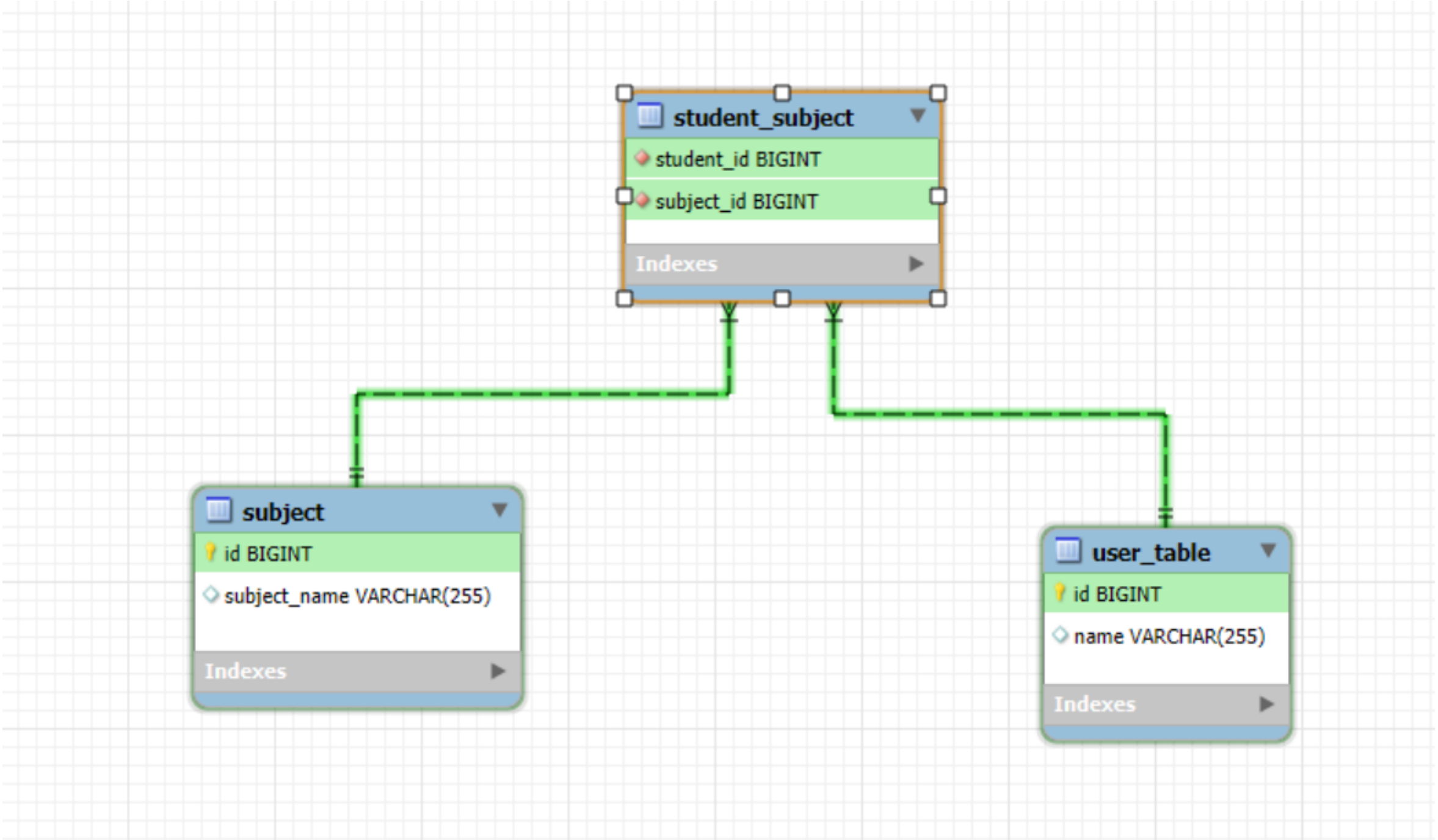
I'm not creating the join table. `Student` already did it — I'm just using it!

`joinColumns = @JoinColumn(name = "student_id")`

"In the `student_subject` table, one column will be `student_id` — that links to the student"

`inverseJoinColumns = @JoinColumn(name = "subject_id")`

"The other column will be `subject_id` — that links to the subject"


student_subject
student_id BIGINT
subject_id BIGINT
Indexes
subject
id BIGINT
subject_name VARCHAR(255)
Indexes
user_table
id BIGINT
name VARCHAR(255)
Indexes

🧩 What is AOP?

AOP = Aspect Oriented Programming

💡 It's like adding **extra features** to your app without messing with your actual code.

Basic Terms

Term	Simple meaning
Aspect	The extra logic (e.g. logging)
Advice	When to run that logic (before, after, etc.)
JoinPoint	The exact place in your app where it runs (like a method)
Pointcut	A filter: “Only run on these methods”
Weaving	Spring puts your logic into the app (like magic glue ✨)

AOP ANNOTATIONS

Annotation	Meaning
`@Aspect`	Marks the class as an aspect (this is where the magic logic lives)
`@Before`	Run code **before** method starts
`@After`	Run code **after** method ends
`@AfterReturning`	Run code if method completes **successfully**
`@AfterThrowing`	Run code if method **throws exception**
`@Around`	Surround the method — run **before + after**, can also **block** it

```java
@Aspect
@Component
public class LoggingAspect {

    @Before("execution(* com.surya.example.services.*.*(..))")
    public void logMethodDetails(JoinPoint joinPoint) {
        System.out.println("📣 Called Method: " + joinPoint.getSignature().getName());
        System.out.println("📦 Arguments: " + Arrays.toString(joinPoint.getArgs()));
        System.out.println("🎯 Target Object: " + joinPoint.getTarget());
    }
}
```

Weaving	Spring injecting your advice logic into real method calls at **runtime**, using proxies, without modifying original method code

```java
// 🔷 AfterReturning Advice
@AfterReturning(
    pointcut = "execution(* com.surya.example.controllers.UserController.addUser(..))",
    returning = "returnedValue"
)
public void logAfterReturning(JoinPoint joinPoint, Object returnedValue) {
    log.warn("📦 [AOP - AFTER RETURNING] Method: {} returned with value: {}",
            joinPoint.getSignature().getName(), returnedValue);
}
```

```java
@AfterThrowing(
    pointcut = "execution(* com.surya.example.services.UserServiceImpl.updateUserById(..))",
    throwing = "ex"
)
public void logAfterThrowing(JoinPoint joinPoint, Throwable ex) {
    log.error("❌ [AOP - AFTER THROWING] Method: {} threw exception: {}",
            joinPoint.getSignature().getName(), ex.getMessage());
}
```

```java
@Aspect
@Component
@Slf4j
public class LoggingAspect {

    @Before("execution(* com.surya.example.controllers.UserController.addUser(..))")
    public void logBefore(JoinPoint joinPoint) {
        log.warn("🚀 [AOP - BEFORE] Method: {}", joinPoint.getSignature().getName());
    }

    @Around("execution(* com.surya.example.controllers.UserController.addUser(..))")
    public Object logAround(ProceedingJoinPoint joinPoint) throws Throwable {
        log.warn("⏳ [AOP - AROUND - BEFORE] Method: {}", joinPoint.getSignature().getName());
        Object result = joinPoint.proceed();
        log.warn("☑ [AOP - AROUND - AFTER] Method: {}", joinPoint.getSignature().getName());
        return result;
    }

    @After("execution(* com.surya.example.controllers.UserController.addUser(..))")
    public void logAfter(JoinPoint joinPoint) {
        log.warn("🎉 [AOP - AFTER] Method: {}", joinPoint.getSignature().getName());
    }
```

```java
@AfterReturning(
    pointcut = "execution(* com.surya.example.controllers.UserController.addUser(..))",
    returning = "returnedValue"
)
public void logAfterReturning(JoinPoint joinPoint, Object returnedValue) {
    log.warn("📦 [AOP - AFTER RETURNING] Returned: {}", returnedValue);
}

@AfterThrowing(
    pointcut = "execution(* com.surya.example.services.UserServiceImpl.updateUserById(..))",
    throwing = "ex"
)
public void logAfterThrowing(JoinPoint joinPoint, Throwable ex) {
    log.error("❌ [AOP - AFTER THROWING] Method: {} threw exception: {}",
              joinPoint.getSignature().getName(), ex.getMessage());
}
```

What is Logging?

Logging is the process of recording information about the application's runtime behavior.

It helps developers understand what's happening inside the app (info, warnings, errors, etc.).

Logging in Spring Boot

1. **Spring Boot uses SLF4J with Logback by default**
   - SLF4J = Logging API (standard interface)
   - Logback = Default logging implementation

Common Logging Libraries

- SLF4J (API)
- Logback (Default in Spring Boot)
- Log4j2 (Alternative)
- JUL (Java Util Logging)

Logging Levels (from lowest to highest)

1. TRACE – Most detailed logs (used for debugging)
2. DEBUG – Useful for development
3. INFO – General info about application progress
4. WARN – Something unexpected, but not breaking
5. ERROR – Serious issue, something failed
6. OFF – Turns off logging

```properties
logging.level.root=INFO
logging.level.com.surya.example=DEBUG
```