## 🧠 Core Idea

You said:

“Complete URL with path, http method, content type, request data, response data is endpoint.”

You’re right in **spirit**, but technically speaking, let’s break it down:

### 1️⃣ What is the **Endpoint**?

An **Endpoint** is the **unique address (URL)** inside your REST API  
that a client uses to perform **one specific operation**.

👉 So, the **Endpoint = Base URL + Path**

Example:

http://localhost:9999/RestProj/api/accounts/register

That’s a **single endpoint** for account registration.

### 2️⃣ What Makes an Endpoint Work

While the **endpoint** itself is mainly the **URL (base + path)**,  
it only becomes **usable** when you combine it with other pieces of info, like:

| **Element** | **Description** | **Example** |
| --- | --- | --- |
| **Base URL** | The main root address of your API | http://localhost:9999/RestProj/api/accounts |
| **Path** | The specific operation | /register or /find/{id} |
| **HTTP Method** | The type of action to perform | GET, POST, PUT, DELETE |
| **Content Type** | The format of data being exchanged | application/json |
| **Request Data** | The input sent by the client | { "name": "Ravi" } |
| **Response Data** | The output sent back by the server | { "status": "success" } |

### 💡 So in Simple Words:

The **Endpoint** is mainly the **unique URL** (Base URL + Path)  
that performs a **specific operation** in your API.

But to **fully define an endpoint** in real-time usage,  
we include **HTTP method**, **content type**, **request**, and **response** —  
because they describe **how** that endpoint behaves.

### 🧩 So You Can Say:

✅ **Technically:**

Endpoint = Base URL + Path

✅ **Practically (real-world meaning):**

Endpoint = URL + HTTP Method + Content Type + Request Data + Response Data

👉 So yes, **your statement is right in a real-world sense.**  
You just need to remember that “endpoint” literally means the **address**,  
but when we **document or use** endpoints, we always include:

* URL (Base + Path)
* HTTP Method
* Content Type
* Request Body (if any)
* Response

### 🧱 Example: Full Endpoint Definition

Let’s take one example 👇

#### 🎯 Operation: Register New Account

| **Property** | **Example** |
| --- | --- |
| **HTTP Method** | POST |
| **Base URL** | <http://localhost:9999/RestProj/api/accounts> |
| **Path** | /register |
| **Complete Endpoint (URL)** | <http://localhost:9999/RestProj/api/accounts/register> |
| **Content Type** | application/json |
| **Request Data** | { "name": "Ravi", "email": "ravi@mail.com", "balance": 10000 } |
| **Response Data** | { "message": "Account created successfully", "status": "success" } |

✅ This **whole definition** is what developers mean when they say:

“The endpoint for account registration is defined.”

### 🧠 Final Summary (in One Line)

🔹 **Technically:** Endpoint = Base URL + Path  
🔹 **Practically:** Endpoint = Complete URL + HTTP Method + Content Type + Request + Response

## 🧠 **API Documentation — Complete Explanation**

### 🔹 What is API Documentation?

**API Documentation** means writing **clear explanations** about how your **classes, methods, and APIs** work —  
so that other developers (or even you in the future) can understand and use them easily.

In simple words:

🗒️ API documentation is like a **user manual** for your code.

It explains:

* What the class or method does
* What parameters it accepts
* What it returns
* What exceptions it might throw
* and sometimes even **example usages**

### 🔹 Why We Need API Documentation

Imagine you are working in a team.  
Your teammate needs to use your method — but they don’t know how it works.

Instead of explaining every time, if your method is **well documented**,  
they can simply read the generated documentation (like a website page)  
and understand what to call and how.

✅ This saves time  
✅ Increases maintainability  
✅ Helps others reuse your APIs correctly  
✅ Makes your project professional and readable

### 🧩 **Types of Documentation in Java**

There are mainly **3 types** of comments in Java —  
two are ignored by the compiler, and one is used for **documentation generation**.

#### 1️⃣ **Block Comment**

/\* text \*/

* Used for writing **multi-line** explanations.
* Compiler **ignores** everything between /\* and \*/.
* Mostly used inside code to describe logic.

✅ Example:

/\* This part handles user authentication

and verifies credentials from the database \*/

#### 2️⃣ **Single-line Comment**

// text

* Used for **short, single-line** explanations.
* Compiler **ignores** everything after // on that line.

✅ Example:

// Validate user input

#### 3️⃣ **Documentation Comment**

/\*\* documentation \*/

* This is also called a **Doc Comment**.
* It starts with /\*\* and ends with \*/.
* The **Javadoc tool** reads this comment and creates **automatic API documentation** in HTML format.

✅ Example:

/\*\*

\* This class handles all account operations like

\* register, find, update, and delete.

\*/

public class AccountService { ... }

### ⚙️ **What is the Javadoc Tool?**

🧰 javadoc is a **tool provided by JDK** that automatically reads your documentation comments  
and converts them into **HTML documentation pages**.

In simple words:

It transforms your code comments into a professional **API reference website**.

**📘 API Documentation in REST**

After developing REST APIs, we can provide API documentation in the following ways:

1. **Using separate text documents**  
   → Create .txt or .md files manually to describe APIs (URL, method, request/response format).  
   ➤ Non-interactive and hard to maintain.
2. **Using API documentation comments and Javadoc tool**  
   → Write Javadoc comments (/\*\* ... \*/) and generate static HTML docs using the javadoc command.  
   ➤ Still non-interactive (you can only read, not test).
3. **Using Swagger / Swagger UI (Interactive Documentation)**  
   → Automatically generates a web page where you can **view and test** APIs directly from your browser.  
   ➤ Most popular and industry-standard approach.
4. **Using OpenAPI Specification**  
   → OpenAPI defines the **format** for describing REST APIs (like JSON schema).  
   ➤ Swagger is one of the **tools** that implements this specification.  
   (So, companies use Swagger because it’s a user-friendly tool built on top of OpenAPI.)

**💡 Simple Analogy**

| **Tool** | **Type** | **Analogy** |
| --- | --- | --- |
| Text docs | Manual | Writing API details in a notebook |
| Javadoc | Semi-automatic | Creating a book of API info, but read-only |
| Swagger | Interactive | Website where you can **read + test APIs live** |
| OpenAPI | Standard | Grammar rules that Swagger follows |

## 🌍 **Core Concept of Swagger API**

When we create REST APIs in a Spring Boot project, we usually test them using **Postman** — right?  
But Postman requires you to:

* Remember all API URLs
* Set the correct HTTP method (GET, POST, PUT, DELETE, etc.)
* Set headers, request body (JSON), and so on

That’s a bit **manual** and **not user-friendly**, especially when someone new joins your project or when you want to share API documentation with testers or other developers.

👉 **Swagger solves this problem.**

### What Swagger Does:

* Automatically generates **interactive documentation** for your APIs.
* Shows all available **endpoints**, **HTTP methods**, **request/response models**, and **example inputs**.
* Allows you to **test APIs directly** from the browser — just like Postman — without opening any external tool.
* Helps your team understand the API easily — what to send and what to expect as output.

So in short:

**Swagger = Live + Interactive API Documentation + Built-in Testing Tool**

## 🧱 **How Swagger Works in Spring Boot**

To use Swagger in Spring Boot, we usually use a library called **Springfox**, which integrates Swagger automatically with your REST controllers.

Springfox provides 2 main dependencies:

1. springfox-swagger2 → for core Swagger configuration
2. springfox-swagger-ui → for the web interface (the UI you open in the browser)

After adding these dependencies and doing a small configuration, Swagger automatically scans your controllers and displays documentation at a URL like:

http://localhost:8080/swagger-ui.html

## ⚙️ **Step-by-Step Setup**

### 🧩 Step 1: Add Swagger Dependencies

In your pom.xml, add the following:

<!-- Swagger Core Library -->

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-swagger2</artifactId>

<version>2.9.2</version>

</dependency>

<!-- Swagger UI Library -->

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-swagger-ui</artifactId>

<version>2.9.2</version>

</dependency>

📝 **Explanation:**

* The first dependency (swagger2) tells Spring Boot to enable Swagger support.
* The second dependency (swagger-ui) gives us a nice user interface to see and test APIs from the browser.

### 🧩 Step 2: Create a Swagger Configuration Class

You need to create one configuration class to tell Swagger how to scan your project.

// File: SwaggerConfig.java

package in.orcas.config;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import springfox.documentation.builders.PathSelectors;

import springfox.documentation.builders.RequestHandlerSelectors;

import springfox.documentation.spi.DocumentationType;

import springfox.documentation.spring.web.plugins.Docket;

import springfox.documentation.swagger2.annotations.EnableSwagger2;

@Configuration // Marks this as a configuration class

@EnableSwagger2 // Enables Swagger2 support

public class SwaggerConfig {

@Bean

public Docket api() {

// Docket is the main class for Swagger configuration

return new Docket(DocumentationType.SWAGGER\_2)

.select()

// Tell Swagger to scan only our REST Controller package

.apis(RequestHandlerSelectors.basePackage("in.orcas.controller"))

.paths(PathSelectors.any()) // Include all URL paths

.build();

}

}

📝 **Explanation of Code:**

* @Configuration: Tells Spring that this is a configuration class.
* @EnableSwagger2: Activates Swagger2.
* Docket: It’s a builder class from Springfox that defines what to include in documentation.
  + DocumentationType.SWAGGER\_2: tells it to use Swagger version 2.
  + .apis(RequestHandlerSelectors.basePackage("in.orcas.controller")):  
    Swagger will only scan the given package for @RestController classes.
  + .paths(PathSelectors.any()): includes all endpoints found.

### 🧩 Step 3: Create a Simple RestController

Let’s create a sample REST controller so Swagger has something to document.

// File: StudentController.java

package in.orcas.controller;

import org.springframework.web.bind.annotation.\*;

import in.orcas.model.Student;

@RestController

@RequestMapping("/student")

public class StudentController {

// Simple GET API to fetch student info

@GetMapping("/{id}")

public Student getStudent(@PathVariable Integer id) {

return new Student(id, "John", "john@gmail.com");

}

// Simple POST API to add student info

@PostMapping("/add")

public String addStudent(@RequestBody Student s) {

return "Student added: " + s.getName();

}

}

📝 **Explanation of Code:**

* @RestController: Marks this as a REST controller.
* @RequestMapping("/student"): All APIs inside will start with /student.
* @GetMapping("/{id}"): Used to fetch data by ID.
* @PostMapping("/add"): Used to add new student data (expects JSON).

### 🧩 Step 4: Model Class

// File: Student.java

package in.orcas.model;

public class Student {

private Integer id;

private String name;

private String email;

public Student() {} // Default constructor

public Student(Integer id, String name, String email) {

this.id = id;

this.name = name;

this.email = email;

}

// Getters & Setters

public Integer getId() { return id; }

public void setId(Integer id) { this.id = id; }

public String getName() { return name; }

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

public String getEmail() { return email; }

public void setEmail(String email) { this.email = email; }

}

### 🧩 Step 5: Run and Test Swagger

Start your Spring Boot application and open this URL in your browser:

👉 [**http://localhost:8080/swagger-ui.html**](http://localhost:8080/swagger-ui.html)

You’ll see:

* List of all APIs (GET, POST, etc.)
* Request/Response model
* “Try it out” button to test APIs directly
* Automatically generated input boxes for parameters

## 💡 **Advantages of Using Swagger**

| **Problem Without Swagger** | **Solution With Swagger** |
| --- | --- |
| Must use Postman manually | Browser UI built-in |
| Must remember URLs & methods | Swagger lists all |
| Difficult to share documentation | Auto-generated docs |
| No direct connection between docs and API code | Always up-to-date with your code |

## ✅ **Summary**

| **Concept** | **Meaning** |
| --- | --- |
| **Swagger** | Tool for API documentation + live testing |
| **Springfox** | Library that integrates Swagger with Spring Boot |
| **swagger2 dependency** | Core Swagger logic |
| **swagger-ui dependency** | User interface in browser |
| **Docket class** | Main configuration builder |
| **SwaggerConfig class** | Defines how Swagger should behave |

Eg: SwaggerExample

**💡 How Swagger (Springdoc) Knows About Your Controllers**

Even though you didn’t explicitly mention your controller package in SwaggerConfig,  
**Springdoc OpenAPI** automatically finds all your REST APIs.

Let’s see *how and why* it happens.

**🧠 1. Spring Boot’s Component Scanning**

When your app starts, the main class (like SwaggerExampleApplication.java) has:

@SpringBootApplication

public class SwaggerExampleApplication {

public static void main(String[] args) {

SpringApplication.run(SwaggerExampleApplication.class, args);

}

}

This annotation:

@SpringBootApplication

is actually a combination of 3 annotations:

@Configuration

@EnableAutoConfiguration

@ComponentScan

The important one here is:

@ComponentScan

✅ **It automatically scans all classes** (controllers, services, configs, etc.) **under the same base package and its subpackages**.

So, since your main class is in:

package in.orcas;

Spring Boot automatically scans:

in.orcas.controller

in.orcas.service

in.orcas.repository

in.orcas.config

...

That’s how your controllers are **automatically registered** with Spring Boot — even if you don’t mention them in the Swagger config.

**🧩 2. Springdoc’s Integration with Spring Boot**

Now, the **Springdoc OpenAPI library** is very smart.  
It hooks into the Spring Boot runtime and automatically detects:

* All classes annotated with @RestController
* Their mappings (@GetMapping, @PostMapping, etc.)
* Request/response models (@RequestBody, @PathVariable, etc.)

It reads all of that information and **auto-generates** your Swagger documentation dynamically.

So when you open:

http://localhost:9090/swagger-ui/index.html

Springdoc scans your running application context and says:

“Ah, here are the controllers! Let me display them as endpoints.”

That’s why you don’t need to tell Swagger manually *which package to scan*.

**🧠 3. Why We Needed Manual Configuration in Old Springfox**

In the old springfox-swagger2 version, you had to write code like:

.apis(RequestHandlerSelectors.basePackage("in.orcas.controller"))

Because Springfox didn’t automatically integrate with Spring Boot’s scanning mechanism.

But **Springdoc** does.  
It reads directly from the **Spring Web layer** at runtime — that’s why it’s so much simpler and more powerful.

**🧩 4. Optional: Restrict Swagger to a Specific Package (If You Ever Want)**

If in the future you want Swagger to document only specific packages (for example, only in.orcas.api),  
you can still configure it manually like this:

springdoc.packages-to-scan=in.orcas.controller

springdoc.paths-to-match=/student/\*\*

👉 Add these lines in application.properties.

📝 **Explanation:**

* springdoc.packages-to-scan → Only scans controllers inside that package.
* springdoc.paths-to-match → Only documents endpoints starting with /student.

But usually, this isn’t needed for most projects.

**✅ Summary**

| **Concept** | **Description** |
| --- | --- |
| @SpringBootApplication | Automatically triggers package scanning |
| Springdoc OpenAPI | Hooks into Spring Boot’s Web layer |
| Auto-detection | Finds all @RestController and endpoint annotations |
| Manual config | Only needed if you want to limit scanning |
| Advantage | No extra setup — zero configuration needed for docs |

💬 **In short:**

Swagger knows your controller paths because Spring Boot automatically registers them in the application context, and Springdoc OpenAPI automatically scans those registered controllers at runtime.