Case Study Title-1: Employee Info API using Spring Boot AutoConfiguration

Objective:
To build a simple Spring Boot application that exposes an API endpoint to retrieve basic employee information using **Spring Boot AutoConfiguration**. The endpoint will be tested via a browser and Postman using only @GetMapping.

Background:
Spring Boot simplifies application setup with its **AutoConfiguration** feature.
Instead of manually defining bean configurations, Spring Boot intelligently guesses what you need and configures it behind the scenes. This case study helps you understand:
• What AutoConfiguration does.
• How to leverage it using minimal configuration.
• How to expose a basic REST endpoint with @GetMapping.

Components Involved:
1. Spring Boot Starter Web – Automatically brings in all dependencies for building REST APIs.
2. AutoConfiguration – Behind the scenes, it configures the DispatcherServlet, Tomcat server, and other beans automatically.
3. REST Controller – A simple Java class using @RestController and @GetMapping.
4. **Browser/Postman** – For testing the GET API.

Scenario:
You are a developer working in the HR software team. Your task is to expose employee information (like name, ID, and department) through a simple HTTP GET API without manually configuring
any server, servlet, or web.xml file.

Steps in the Case Study:
1. **Create the Spring Boot Project•**

Use Spring Initializr (https://start.spring.io)

• Project metadata:
◦ Group: com.company
◦ Artifact: employee-api
• Dependencies:
◦ Spring Web

2. Directory Structure AutoCreated by Spring Boot
Spring Boot automatically generates the following:

```
src/
main/
java/
com.company.employeeapi/
EmployeeApiApplication.java
controller/
EmployeeController.java
resources/
application.properties
```

3.Understanding AutoConfiguration

//EmployeeApiApplication.java

```
package com.company.employeeapi;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class EmployeeApiApplication {
    public static void main(String[] args) {
        SpringApplication.run(EmployeeApiApplication.class, args);
    }
}
```

• No need to configure DispatcherServlet, JSON converter, or server port.
• When you add spring-boot-starter-web, it:
◦ Configures embedded Tomcat server.
◦ Registers Jackson for JSON conversion.

◦ Sets up DispatcherServlet for handling REST requests.
◦ Starts server on port 8080.

4.Creating a Simple GET Endpoint

• The @RestController and @GetMapping("/employee") annotations automatically expose a REST endpoint due to AutoConfiguration.

//EmployeeController.java

```
package com.company.employeeapi.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import java.util.Map;

@RestController

public class EmployeeController {

    @GetMapping("/employee")
    public Map<String, Object> getEmployee() {
        return Map.of(
            "id", 101,
            "name", "John Doe",
            "department", "Engineering"
        );
    }
}
```

5. Running the Application

• Just run the main class EmployeeApiApplication.java.
• Spring Boot auto-starts the embedded server and makes the endpoint live.

6.Testing the API

Open browser or Postman.
Hit: http://localhost:8080/employee
Expected JSON output:

```
{
"id": 101,
"name": "John Doe",
"department": "Engineering"
}
```

2. Spring Boot – Actuators
Case Study: Monitoring an Inventory System

Problem Statement:
You deploy an Inventory Management app and want to **monitor** its health, memory usage, bean loading, and environment settings without building these endpoints manually.

Key Concept:
Spring Boot **Actuator** exposes production-ready features like health checks, metrics, beans, and custom endpoints.

Scenario:
You add the spring-boot-starter-actuator dependency, and enable the / actuator endpoint in application.properties.
With zero code changes, you get:
• /actuator/health → Health of the service.
• /actuator/beans → Beans created in the container.
• /actuator/metrics → JVM and HTTP metrics.
• /actuator/env → Current environment values.

Project Setup

Go to Spring Initializr and create the project with:

Group: com.company

Artifact: inventory-system

Dependencies:

Spring Web (for REST API)

Spring Boot Actuator (for monitoring)

Packaging: Jar

Java Version: 17 (or your installed version)

//pom.xml(If doing manually without spring initializer)

```
<dependency>

   <groupId>org.springframework.boot</groupId>

   <artifactId>spring-boot-starter-actuator</artifactId>

</dependency>
```

//InventorySystemApplication.java

```
package com.company.inventorysystem;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class InventorySystemApplication {
   public static void main(String[] args) {
      SpringApplication.run(InventorySystemApplication.class, args);
   }
}
```

//application.properties

```
management.endpoints.web.exposure.include=*
```

//InventoryController.java

```
package com.company.inventorysystem.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import java.util.Map;

@RestController
public class InventoryController {

   @GetMapping("/inventory")
   public Map<String, Object> getInventory() {
      return Map.of(
         "itemId", 201,
         "itemName", "Laptop",
         "quantity", 50
```

```
        );
    }
}
```

//Running the Application

Run InventorySystemApplication.java

//Open browser or Postman and check:

Health Check

```
GET http://localhost:8080/actuator/health
```

Sample Response:

```
{
  "status": "UP"
}
```

Beans Loaded in Spring Context

```
GET http://localhost:8080/actuator/beans
```

Metrics (Memory, CPU, HTTP Requests)

```
GET http://localhost:8080/actuator/metrics
```

Environment Variables

```
GET http://localhost:8080/actuator/env
```