



Case Study Title: 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

- 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.

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"  
}  
☐  
☐
```

Package - `Com.company.employee_api.controller`

Class - `EmployeeController.java`

Code :

```
package com.company.employee_api.controller;
```

```
import java.util.HashMap;
```

```
import java.util.Map;
```

```
import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RestController;


@RestController


public class EmployeeController {


    @GetMapping("/employee")


    public Map<String, Object> getEmployeeInfo(){


        Map<String, Object> emp = new HashMap<>();


        emp.put("id", 101);


        emp.put("name", "John Doe");


        emp.put("department", "Engineering");


        return emp;


    }

}
```

EmployeeApiApplication.java

```
package com.company.employee_api;

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);

    }

}
```

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.

Code :

pom.xml – Add Actuator Dependency

```
<dependencies>
```

```
    <!-- Spring Boot Starter Web -->
```

```

<dependency>

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

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

</dependency>

<!-- Spring Boot Actuator -->

<dependency>

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

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

</dependency>

</dependencies>

```

InventoryMonitorApplication.java:

```

package com.inventory;

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

@SpringBootApplication

public class InventoryMonitorApplication {

    public static void main(String[] args) {

        SpringApplication.run(InventoryMonitorApplication.class, args);

    }

}

```

application.properties

//Enable all actuator endpoints

management.endpoints.web.exposure.include=*

// Show full health details in the response

management.endpoint.health.show-details=always