



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. 2. 3. 4. Spring Boot Starter Web – Automatically brings in all dependencies for building REST APIs. AutoConfiguration – Behind the scenes, it configures the DispatcherServlet, Tomcat server, and other beans automatically.

REST Controller – A simple Java class using @RestController and @GetMapping.

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" }`

pom.xml:

```
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0

        https://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <!-- ✅ Your project's metadata -->

    <groupId>com.company</groupId>

    <artifactId>employee-api</artifactId>
```

<version>0.0.1-SNAPSHOT</version>

<name>Employee API</name>

<!--  Spring Boot parent (very important) -->

<parent>

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

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

<version>3.2.0</version> <!-- You can also use 3.1.5 or any latest version -->

<relativePath/> <!-- lookup parent from repository -->

</parent>

<!--  Java version -->

<properties>

<java.version>17</java.version> <!-- Change to 21 if you're using Java 21 -->

</properties>

<!--  Spring dependencies -->

<dependencies>

<!-- Spring Web for REST APIs -->

<dependency>

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

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

</dependency>

</dependencies>

<!--  Spring Boot plugin -->

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>
```

application.properties:

```
server.port=8082

spring.application.name=employee-api
```

src/main/java-com.company.employeeapi.controller:

EmployeeController:

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

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> getEmployee() {

        return Map.of(

            "id", 101,

            "name", "John Doe",


            "department", "Engineering"
```


```
);  
  
}  
  
}
```


EmployeeApiApplication:

```
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);  
  
    }  
  
}
```

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

pom.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-  
instance"
```

```
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
```

```
  <modelVersion>4.0.0</modelVersion>
```

```
  <parent>
```

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

```
    <artifactId>spring-boot-starter-parent</artifactId>
```

```
    <version>3.5.4</version>
```

```
    <relativePath/> <!-- lookup parent from repository -->
```

```
  </parent>
```

```
  <groupId>com.company</groupId>
```

```
  <artifactId>inventory-monitor</artifactId>
```

```
  <version>0.0.1-SNAPSHOT</version>
```

```
  <name>inventory-monitor</name>
```

```
  <description>com.company.inventorymonitor</description>
```

```
  <url/>
```

```
  <licenses>
```

```
    <license/>
```

```
  </licenses>
```

```
  <developers>
```

```
    <developer/>
```

```
  </developers>
```

```
  <scm>
```

```
    <connection/>
```

```
    <developerConnection/>
```

```
    <tag/>
```

```
    <url/>
```

```
  </scm>
```

```
<properties>

    <java.version>17</java.version>

</properties>

<dependencies>

    <dependency>

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

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

    </dependency>

    <dependency>

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

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

    </dependency>

    <dependency>

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

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

        <scope>test</scope>

    </dependency>

</dependencies>

<build>

    <plugins>

        <plugin>

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

            <artifactId>spring-boot-maven-plugin</artifactId>

        </plugin>

    </plugins>

</build>
```

</project>

application.properties:

server.port=8085


spring.application.name=inventory-monitor


management.endpoints.web.exposure.include=*

management.endpoint.health.show-details=always

InventoryMonitorApplication:

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. 2. 3. 4. Spring Boot Starter Web – Automatically brings in all dependencies for building REST APIs. AutoConfiguration – Behind the scenes, it configures the DispatcherServlet, Tomcat server, and other beans automatically.

REST Controller – A simple Java class using @RestController and @GetMapping.

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"

pom.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
        https://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <!-- ✅ Your project's metadata -->

    <groupId>com.company</groupId>

    <artifactId>employee-api</artifactId>

    <version>0.0.1-SNAPSHOT</version>

    <name>Employee API</name>

    <!-- ✅ Spring Boot parent (very important) -->

    <parent>

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

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

        <version>3.2.0</version> <!-- You can also use 3.1.5 or any latest version -->

        <relativePath/> <!-- lookup parent from repository -->

    </parent>

    <!-- ✅ Java version -->

    <properties>

        <java.version>17</java.version> <!-- Change to 21 if you're using Java 21 -->

    </properties>

    <!-- ✅ Spring dependencies -->

    <dependencies>

        <!-- Spring Web for REST APIs -->
```


```
<dependency>

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

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

</dependency>

</dependencies>

<!--  Spring Boot plugin -->

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>
```

application.properties:

server.port=8082

spring.application.name=employee-api

src/main/java-com.company.employeeapi.controller:

EmployeeController:

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

```
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> getEmployee() {
```

```
        return Map.of(
```

```
            "id", 101,
```

```
            "name", "John Doe",
```

```
            "department", "Engineering"
```

```
        );
```

```
    }
```

```
}
```

EmployeeApiApplication:

```
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);
```

```
    }
```

```
}
```

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

pom.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
```

```
    <modelVersion>4.0.0</modelVersion>
```

```
    <parent>
```

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

```
        <artifactId>spring-boot-starter-parent</artifactId>
```

```
        <version>3.5.4</version>
```

```
        <relativePath/> <!-- lookup parent from repository -->
```

```
    </parent>
```

```
    <groupId>com.company</groupId>
```

```
    <artifactId>inventory-monitor</artifactId>
```

```
    <version>0.0.1-SNAPSHOT</version>
```

```
    <name>inventory-monitor</name>
```

```
    <description>com.company.inventorymonitor</description>
```

```
    <url/>
```

```
    <licenses>
```

```
        <license/>
```

</licenses>

<developers>

<developer/>

</developers>

<scm>

<connection/>

<developerConnection/>

<tag/>

<url/>

</scm>

<properties>

<java.version>17</java.version>

</properties>

<dependencies>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

<scope>test</scope>

</dependency>

```
</dependencies>
```

```
<build>
```

```
<plugins>
```

```
<plugin>
```

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

```
<artifactId>spring-boot-maven-plugin</artifactId>
```

```
</plugin>
```

```
</plugins>
```

```
</build>
```

```
</project>
```

application.properties:

```
server.port=8085
```

```
spring.application.name=inventory-monitor
```

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

```
management.endpoint.health.show-details=always
```

InventoryMonitorApplication:

```
package com.company.inventory_monitor;
```

```
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);
    }

}
```

InventoryController:

```
package com.company.inventorymonitor.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> getInventoryStatus() {

        return Map.of(

            "status", "available",

            "items", 120,

            "location", "Warehouse A"

        );

    }

}
```