Java Spring Boot Notes

Commands, Examples, Project Structure & Base Application

Contents

What is Spring Boot?

Spring Boot is a framework for building Java applications quickly.
It provides:

- opinionated defaults (*"starter" dependencies*),
- embedded server (Tomcat/Jetty),
- auto-configuration,
- easy packaging as a `.jar`.

Mental model: You write Java classes (`@RestController`, `@Service`, `@Entity`), Spring Boot wires everything together and runs an HTTP server for you.

Basic Commands (Maven & Spring Boot)

Create a New Project (via Spring Initializr ZIP)

Often you use the website `start.spring.io`, download a ZIP, and unzip it.
Alternatively, you can use the CLI (if installed):

Generate a new project (Spring CLI)

```
spring init \
  --dependencies=web \
  --groupId=com.example \
  --artifactId=demo \
  --name=demo \
  demo-project
```

Run the Application

Inside the project root (where `pom.xml` is):

Run Spring Boot app with Maven

```
# compile and run with embedded Tomcat on default port 8080
mvn spring-boot:run
```

Build a .jar File

Build executable JAR

```
mvn clean package

# Then run the jar:
java -jar target/demo-0.0.1-SNAPSHOT.jar
```

Common Maven Commands

Useful Maven commands

```
mvn clean            # delete compiled artifacts (target/)
mvn compile          # compile source code
mvn test             # run tests
mvn package          # build JAR/WAR
mvn spring-boot:run  # run Spring Boot application
```

Project Structure (Analysis of src/)

Typical Maven Spring Boot Layout

```
demo/
  pom.xml
  src/
    main/
      java/
        com/example/demo/
          DemoApplication.java
          controller/
            HelloController.java
          service/
            HelloService.java
          model/
            User.java
      resources/
        application.properties
        static/
        templates/
    test/
      java/
        com/example/demo/
          DemoApplicationTests.java
```

Analysis of Key Files

- **pom.xml** Maven configuration: dependencies, plugins, Java version, build info.

- **DemoApplication.java** Main class with `@SpringBootApplication` and `main()` method. Entry point of the app.
- **controller/HelloController.java** Web layer. Defines HTTP endpoints with `@RestController`, `@GetMapping`, etc.
- **service/HelloService.java** Business logic. Annotated with `@Service`.
- **model/User.java** Data model / entity class (fields, getters, setters, etc.).
- **application.properties** Configuration: server port, DB URL, usernames, etc.

Pattern to remember: **Controller** handles HTTP, **Service** handles logic, **Repository** talks to database.

Making a Base Spring Boot Application

Main Application Class

DemoApplication.java

```java
package com.example.demo;

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

// @SpringBootApplication = @Configuration +
   @EnableAutoConfiguration + @ComponentScan
@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        // Bootstraps the application, starts Spring context and
           embedded server
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

Line-by-line Analysis

- `package com.example.demo;` – defines the Java package (folder structure).
- `@SpringBootApplication` – tells Spring Boot to:
  - search for components in this package and subpackages,
  - enable auto-configuration,
  - treat this class as configuration.
- `public static void main(...)` – standard Java entry point.
- `SpringApplication.run(...)` – starts Spring, creates Beans, runs embedded Tomcat.

Simple REST Controller

HelloController.java

```
package com.example.demo.controller;

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

@RestController // tells Spring this class handles HTTP requests
public class HelloController {

    @GetMapping("/hello")
    public String hello() {
        return "Hello from Spring Boot!";
    }
}
```

What happens when you run the app?

- You start the app with `mvn spring-boot:run`.
- Spring Boot starts Tomcat on port 8080.
- A GET request to `http://localhost:8080/hello` is routed to `hello()`.
- The method returns a `String` → sent as HTTP response body.

Using a Service Class

HelloService.java

```
package com.example.demo.service;

import org.springframework.stereotype.Service;

@Service
public class HelloService {

    public String buildMessage(String name) {
        return "Hello, " + name + "! Welcome to Spring Boot.";
    }
}
```

HelloController with Service Injection

```java
package com.example.demo.controller;

import com.example.demo.service.HelloService;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloController {

    private final HelloService helloService;

    // Constructor injection (recommended)
    public HelloController(HelloService helloService) {
        this.helloService = helloService;
    }

    @GetMapping("/hello")
    public String hello(@RequestParam(defaultValue = "World") String
        name) {
        return helloService.buildMessage(name);
    }
}
```

Analysis of Dependency Injection

- `@Service` on `HelloService` – Spring manages it as a Bean.
- Constructor parameter `HelloService helloService` – Spring injects it automatically.
- `@RequestParam` – reads `?name=...` from the URL.

Configuration: application.properties

Basic Settings

```
# src/main/resources/application.properties

# change port (default 8080)
server.port=8081

# logging level
logging.level.root=INFO

# example datasource (H2 in-memory)
spring.datasource.url=jdbc:h2:mem:testdb
spring.jpa.hibernate.ddl-auto=update
```

Spring Boot reads `application.properties` (or `application.yml`) on startup and automatically configures many components (server, DB, logging, etc.).

More Java Examples (CRUD Skeleton)

Simple Model Class

User.java

```java
package com.example.demo.model;

public class User {

    private Long id;
    private String name;
    private String email;

    // Constructors
    public User() {}

    public User(Long id, String name, String email) {
        this.id    = id;
        this.name  = name;
        this.email = email;
    }

    // Getters and setters
    public Long getId() { return id; }
    public void setId(Long 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; }
}
```

Controller with Basic CRUD Endpoints (in-memory list)

UserController.java (in-memory)

```java
package com.example.demo.controller;

import com.example.demo.model.User;
import org.springframework.web.bind.annotation.*;

import java.util.ArrayList;
import java.util.List;

@RestController
@RequestMapping("/users")
public class UserController {

    private final List<User> users = new ArrayList<>();

    @GetMapping
    public List<User> getAllUsers() {
        return users;
    }

    @PostMapping
    public User createUser(@RequestBody User user) {
        // simple id assignment for demo
        user.setId((long) (users.size() + 1));
        users.add(user);
        return user;
    }

    @GetMapping("/{id}")
    public User getUser(@PathVariable Long id) {
        return users.stream()
                .filter(u -> id.equals(u.getId()))
                .findFirst()
                .orElse(null);
    }
}
```

What this demonstrates

- `@RequestMapping("/users")` – base path for all endpoints in this controller.
- `@GetMapping` – HTTP GET, returns all users or one user.
- `@PostMapping` – HTTP POST, creates a new user.
- `@RequestBody` – maps JSON body → `User` object.
- `@PathVariable` – maps `/users/{id}` → method parameter.

How to Build Your Own Base Project (Step-by-step)

Checklist

To create a base Spring Boot application:

1. Generate project (Spring Initializr or CLI).
2. Implement `DemoApplication.java` with `@SpringBootApplication`.
3. Add one **Controller** with `@RestController`.
4. Optionally add a **Service** class and inject it.
5. Configure basic properties in `application.properties`.
6. Run locally with `mvn spring-boot:run`.

Minimal Base Example

Minimal Base Application: main + controller

```java
package com.example.base;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
public class BaseApplication {

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

    @RestController
    static class BaseController {

        @GetMapping("/")
        public String index() {
            return "Base Spring Boot app is running!";
        }
    }
}
```

Why this is nice as a base

- Only one file to start with.
- You can later move `BaseController` to its own package.
- You already have a health-like endpoint at `/`.

Summary

Core ideas to keep in mind:

- Spring Boot starts an embedded server and auto-configures a lot for you.
- **Main class** with `@SpringBootApplication` is the entry point.
- **Controllers** define HTTP endpoints.
- **Services** hold business logic.
- **application.properties** controls behaviour (port, DB, logging, etc.).
- Maven commands: `mvn spring-boot:run`, `mvn package`, `java -jar ...`

This is enough base to start building real Java Spring Boot applications.