🌟Spring Boot — Zero to Hero

🐥Chapter 1 — Spring & Spring Boot Fundamentals (School Management Example)

Learning goals (after finishing this chapter):

- Understand what a *framework* is and why Spring exists.
- Understand core Spring concepts: IoC, DI, Bean, ApplicationContext, Entities, Repositories, Controllers.
- Learn how Spring Boot auto-configures common components and how to run a simple, real-world REST API for a **School Management System**.
- See full, beginner-friendly code examples (entities, repositories, services, controllers), request/response samples, configuration, testing hints, and troubleshooting.

🌀Target reader

This chapter assumes only **basic Java knowledge** (classes, objects, methods). Every new term is defined; whenever code appears, it is explained line-by-line.

🦘1.1 Key Definitions (beginner-friendly)

- **Framework:** A foundation with pre-built components and rules for building applications (e.g., Spring). Think of it as a building kit with prewired plumbing and electricity.
- **IoC (Inversion of Control):** Instead of classes instantiating their dependencies, the framework creates and supplies them. The *control* of creating objects is inverted (framework controls it).
- **DI (Dependency Injection):** A pattern implementing IoC — dependencies are "injected" into objects. Common forms: constructor injection (preferred), setter injection, field injection.
- **Bean:** An object managed by the Spring container. Beans are created, configured and wired by Spring.
- **ApplicationContext:** The Spring container that holds bean definitions and manages their lifecycle.
- **Entity:** A Java class mapped to a database table using JPA annotations (e.g., `@Entity`).
- **Repository:** A data access component (interface) that performs DB operations. In Spring Data, repositories are often interfaces that extend `JpaRepository`.

- **Controller:** A class that handles HTTP requests and produces responses (e.g., REST endpoints).
- **DTO (Data Transfer Object):** A plain object used to exchange data across process boundaries (e.g., request or response payloads).
- **REST (Representational State Transfer):** An architectural style for HTTP APIs where resources are identified by URLs and manipulated via HTTP verbs (GET, POST, PUT, DELETE).
- **CRUD:** Create, Read, Update, Delete — basic operations for persistent resources.

🐂1.2 Domain Overview — School Management System (conceptual)

We'll build a small REST API that models these domain entities:

- **Student** — id, name, age, classroom assignment, parents
- **Teacher** — id, name, subject
- **Classroom** — id, room number, a teacher, list of students
- **Parent** — id, name, contact info, linked students

Relationships (simplified):

- A **Student** belongs to one **Classroom** (Many students → One classroom).
- A **Classroom** has one **Teacher** (Many classrooms → One teacher), or alternatively a teacher can be assigned to many classrooms.
- **Student** and **Parent** have a Many-to-Many relationship: one student can have multiple parents; one parent can have multiple students.

We'll implement API endpoints to manage these resources.

🦠1.3 Project Setup (Maven) — `pom.xml` (minimal)

Explanation: We use Maven to manage dependencies. This snippet contains the essential Spring Boot starters for web, data-jpa, H2 (in-memory DB for learning), actuator for monitoring, and test support.

```
<!-- pom.xml (excerpt) -->
<project xmlns="http://maven.apache.org/POM/4.0.0" ...>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.school</groupId>
  <artifactId>school-management</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <parent>
    <groupId>org.springframework.boot</groupId>
```

```
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.1.0</version>
  </parent>

  <dependencies>
    <!-- Web (Spring MVC) -->
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <!-- JPA and H2 for quick demo -->
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
      <groupId>com.h2database</groupId>
      <artifactId>h2</artifactId>
      <scope>runtime</scope>
    </dependency>

    <!-- Actuator (monitoring) -->
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>

    <!-- Test -->
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <properties>
    <java.version>17</java.version>
  </properties>
</project>
```

> **Note:** Version numbers change over time — use the latest Spring Boot stable release when you create a real project.

🕊️1.4 Recommended Project Structure

```
src/main/java/com/school/management
 ├─ SchoolApplication.java         # main class
 ├─ domain/
 │    ├─ Student.java
 │    ├─ Teacher.java
 │    ├─ Classroom.java
 │    └─ Parent.java
 ├─ repository/
 │    ├─ StudentRepository.java
 │    └─ TeacherRepository.java
 ├─ service/
 │    ├─ StudentService.java
 │    └─ StudentServiceImpl.java
 ├─ controller/
 │    └─ StudentController.java
 └─ dto/
      └─ StudentDTO.java
```

🍂1.5 Main Application Class — `SchoolApplication.java`

```java
package com.school.management;

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

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

Explanation: `@SpringBootApplication` enables auto-configuration, component scanning, and Java-based configuration. The `main` method launches the embedded server and the Spring context.

🐣1.6 Entities (JPA) — full examples with explanations

`Student.java`

```java
package com.school.management.domain;

import jakarta.persistence.*;
import java.util.HashSet;
import java.util.Set;

@Entity
@Table(name = "students")
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String name;

    private Integer age;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "classroom_id")
    private Classroom classroom;

    @ManyToMany
    @JoinTable(name = "student_parents",
        joinColumns = @JoinColumn(name = "student_id"),
        inverseJoinColumns = @JoinColumn(name = "parent_id")
    )
    private Set<Parent> parents = new HashSet<>();

    // constructors, getters, setters
}
```

Notes:

- `@Entity` marks the class as a JPA entity.
- `@Id` and `@GeneratedValue` define the primary key.
- `@ManyToOne` indicates many students can belong to one classroom. `LAZY` fetch means related classroom data is loaded only when requested.
- `@ManyToMany` with `@JoinTable` models the student-parent relation.

`Parent.java`

```java
package com.school.management.domain;

import jakarta.persistence.*;
import java.util.HashSet;
import java.util.Set;

@Entity
@Table(name = "parents")
public class Parent {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String name;

    private String phone;

    @ManyToMany(mappedBy = "parents")
    private Set<Student> children = new HashSet<>();

    // constructors, getters, setters
}
```

`Classroom.java`

```java
package com.school.management.domain;

import jakarta.persistence.*;
import java.util.HashSet;
import java.util.Set;

@Entity
@Table(name = "classrooms")
public class Classroom {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String roomNumber;

    @OneToMany(mappedBy = "classroom")
    private Set<Student> students = new HashSet<>();
```

```
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "teacher_id")
    private Teacher teacher;

    // constructors, getters, setters
}
```

`Teacher.java`

```
package com.school.management.domain;

import jakarta.persistence.*;
import java.util.HashSet;
import java.util.Set;

@Entity
@Table(name = "teachers")
public class Teacher {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    private String subject;

    @OneToMany(mappedBy = "teacher")
    private Set<Classroom> classrooms = new HashSet<>();

    // constructors, getters, setters
}
```

> **Tip:** For learning, keep getters/setters and constructors concise. In production you can use Lombok to reduce boilerplate (`@Data`, `@NoArgsConstructor`, `@AllArgsConstructor`) but be aware of its trade-offs.

🦗 1.7 Repositories — Data access layer

Spring Data JPA reduces boilerplate for data access. You declare an interface and Spring provides the implementation at runtime.

```
package com.school.management.repository;
```

```java
import com.school.management.domain.Student;
import org.springframework.data.jpa.repository.JpaRepository;

public interface StudentRepository extends JpaRepository<Student, Long> {
    // Derived query example: find by name
    List<Student> findByName(String name);
}
```

Explanation: `JpaRepository<T, ID>` gives CRUD methods: `save()`, `findById()`, `findAll()`, `deleteById()`.

1.8 Services — Business logic with DI

Keep business logic in services and keep controllers thin.

```java
package com.school.management.service;

import com.school.management.domain.Student;
import java.util.List;

public interface StudentService {
    Student create(Student s);
    Student getById(Long id);
    List<Student> getAll();
    Student update(Long id, Student s);
    void delete(Long id);
}
```

Implementation:

```java
package com.school.management.service;

import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service
public class StudentServiceImpl implements StudentService {
    private final StudentRepository studentRepository;

    // constructor injection — preferred because it's immutable and testable
    public StudentServiceImpl(StudentRepository studentRepository) {
        this.studentRepository = studentRepository;
    }
```

```
    @Override
    @Transactional
    public Student create(Student s) {
        return studentRepository.save(s);
    }

    @Over
```