Spring Boot

REST API · Spring Data JPA · Spring Security, explained simply

Spring Boot Notes · easy to understand · free for students

1. Start with Spring Boot

- **Spring** is a big Java framework for building apps. **Spring Boot** makes it easy: you get **starters** (ready dependency bundles), **auto-configuration**, an **embedded server** (Tomcat) and a single jar you run with `java -jar`.

IDEA Spring is a full kitchen with every tool, but you set it all up. Spring Boot is a meal kit: the pieces come measured and ready, so you start cooking at once.



Each layer talks only to the next one. DTOs travel up, entities stay below the service

	Plain Spring	Spring Boot
Setup	Lots of manual config	Auto-configured
Server	Deploy a WAR to Tomcat	Embedded, just run the jar
Dependencies	Pick each version yourself	Starters with matching versions
Start	Deploy to a server	Run the main method

- You need **JDK 17 or higher** (Spring Boot 3 and newer), **Maven** or Gradle, an IDE like IntelliJ IDEA, **Postman** and a database (MySQL or PostgreSQL).

2. Project Setup and Structure

- Go to `start.spring.io`. Choose Maven, Java, Jar. Add **Spring Web**, **Spring Data JPA**, **Validation**, **Spring Security**, a **database driver** and **DevTools**.

Starter	Gives you
spring-boot-starter-web	REST APIs, Tomcat, JSON (Jackson)
spring-boot-starter-data-jpa	JPA, Hibernate, repositories
spring-boot-starter-validation	@Valid, @NotBlank, @Email
spring-boot-starter-security	Login and access control
spring-boot-starter-actuator	Health and metrics endpoints
spring-boot-starter-test	JUnit, Mockito, MockMvc

EXAMPLE Folder layout. The main class sits in the root package.

```
src/main/java/com/shop/
  ShopApplication.java
  controller/  service/  repository/
  entity/  dto/  config/  exception/
src/main/resources/
  application.properties
```

EXAMPLE Main class and database settings.

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

# application.properties
spring.datasource.url=jdbc:mysql://localhost:3306/shop
spring.datasource.username=root
spring.datasource.password=${DB_PASSWORD}
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

- Run: `./mvnw spring-boot:run`. Build: `./mvnw clean package`, then `java -jar target/app.jar`. The app opens at `localhost:8080`.

REMEMBER Spring scans only the main class package and below. A controller in a different root package will not be found.

3. Beans and Dependency Injection

- The **IoC container** creates your objects, called **beans**, and gives them to the classes that need them. This is **Dependency Injection**. You do not write `new` for services.

Annotation	Use
`@Component`	Any general bean
`@Service`	Business logic class
`@Repository`	Database access class
`@RestController`	Web layer class that returns JSON
`@Configuration` + `@Bean`	Create beans by hand, like third party objects

EXAMPLE Constructor injection (best way).

```
@Service
public class UserService {
  private final UserRepository repo;

  public UserService(UserRepository repo) {   // Spring passes it in
    this.repo = repo;
  }
}
```

EXAMPLE Make a bean yourself.

```
@Configuration
public class AppConfig {
  @Bean
  PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
  }
}
```

- Avoid `@Autowired` on fields. Constructor injection is easier to test. If two beans have the same type, pick one with `@Primary` or `@Qualifier`.
- Scope: ***singleton*** is the default (one shared instance). Others: `prototype` (new each time), `request`, `session`. Read settings with `@Value('${app.name}')`.

4. Building REST Controllers

- `@RestController` = `@Controller` + `@ResponseBody`. Returned objects become **JSON** automatically. Use **DTOs** (records) for input and output, not entities.

EXAMPLE DTO records.

```
public record UserRequest(@NotBlank String name, @Email String email) {}

public record UserResponse(Long id, String name, String email) {}
```

EXAMPLE A full CRUD controller.

```
@RestController
@RequestMapping("/api/users")
public class UserController {
  private final UserService service;
  public UserController(UserService service) { this.service = service; }

  @GetMapping
  public List<UserResponse> all() { return service.findAll(); }

  @GetMapping("/{id}")
  public UserResponse one(@PathVariable Long id) { return service.get(id); }

  @PostMapping
  public ResponseEntity<UserResponse> create(@Valid @RequestBody UserRequest req) {
    UserResponse saved = service.create(req);
    URI uri = URI.create("/api/users/" + saved.id());
    return ResponseEntity.created(uri).body(saved);
  }

  @PutMapping("/{id}")
  public UserResponse update(@PathVariable Long id,
                             @Valid @RequestBody UserRequest req) {
    return service.update(id, req);
  }

  @DeleteMapping("/{id}")
  @ResponseStatus(HttpStatus.NO_CONTENT)
  public void delete(@PathVariable Long id) { service.delete(id); }
}
```

Code	Use it when
200 OK	Read or update worked
201 Created	New resource made (add a Location header)
204 No Content	Delete worked
400 / 404 / 409	Bad input / not found / duplicate
401 / 403	Not logged in / not allowed
500	Unexpected server error

5. Request Data and Validation

Annotation	Reads
`@PathVariable`	Part of the URL: /users/**5**
`@RequestParam`	Query string: ?page=1&size=10
`@RequestBody`	JSON body turned into an object
`@RequestHeader`	A header like Authorization

EXAMPLE Search with paging and sorting.

```
@GetMapping("/search")
public Page<UserResponse> search(
    @RequestParam(defaultValue = "") String q,
    @RequestParam(defaultValue = "0") int page,
    @RequestParam(defaultValue = "10") int size) {
  return service.search(q, PageRequest.of(page, size, Sort.by("name")));
}
```

Validation	Meaning
@NotNull / @NotBlank / @NotEmpty	Not null / has text / not empty
@Size(min, max)	Length of text or list
@Min / @Max	Number range
@Email / @Pattern	Email format / regex
@Past / @Future	Date rules

EXAMPLE DTO with messages.

```
public record RegisterRequest(
    @NotBlank(message = "Name is required") String name,
    @Email(message = "Invalid email") String email,
    @Size(min = 8, message = "Use at least 8 characters") String password) {}
```

- Put `@Valid` before `@RequestBody` to switch validation on. For `@PathVariable` and `@RequestParam` rules, add `@Validated` on the class.

6. Exception Handling

- Throw your own exceptions in the service. One `@RestControllerAdvice` class turns them into clean JSON errors for the whole app.

EXAMPLE Custom exception and global handler.

```
public class NotFoundException extends RuntimeException {
  public NotFoundException(String msg) { super(msg); }
}

@RestControllerAdvice
public class GlobalHandler {
  @ExceptionHandler(NotFoundException.class)
  public ResponseEntity<Map<String, String>> notFound(NotFoundException e) {
    return ResponseEntity.status(404).body(Map.of("message", e.getMessage()));
  }

  @ExceptionHandler(MethodArgumentNotValidException.class)
  public ResponseEntity<?> invalid(MethodArgumentNotValidException e) {
    Map<String, String> errors = new HashMap<>();
    e.getBindingResult().getFieldErrors()
        .forEach(f -> errors.put(f.getField(), f.getDefaultMessage()));
    return ResponseEntity.badRequest().body(errors);
  }
}
```

Exception	Status
NotFoundException (yours)	404
MethodArgumentNotValidException	400 with field errors
DataIntegrityViolationException	409, like a duplicate email
AccessDeniedException	403
Exception (catch all)	500, log it, hide details

- Spring 6 also supports the standard **ProblemDetail** error format. Turn it on with `spring.mvc.problemdetails.enabled=true`. Never send stack traces to the client.

7. Entities and Repositories

- **JPA** is the standard for mapping Java classes to tables, **Hibernate** is the engine, and **Spring Data JPA** writes repository code for you. Imports are from `jakarta.persistence` in Boot 3 and newer.

EXAMPLE Entity class.

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

  @Column(nullable = false, length = 50)
  private String name;

  @Column(nullable = false, unique = true)
  private String email;

  @Enumerated(EnumType.STRING)
  private Role role = Role.USER;
  // constructors, getters, setters
}
```

ddl-auto	Meaning
none / validate	Do nothing / check tables match entities (use in production)
update	Add missing columns. Fine for learning only
create / create-drop	Rebuild tables on start. Data is lost

EXAMPLE Repository with no SQL.

```
public interface UserRepository extends JpaRepository<User, Long> {
  Optional<User> findByEmail(String email);
  boolean existsByEmail(String email);
  List<User> findByNameContainingIgnoreCase(String q);
  Page<User> findByRole(Role role, Pageable pageable);
}
```

- Free methods: `save`, `findById`, `findAll`, `deleteById`, `existsById`, `count`. Method names are parsed into queries: `findBy`, `And`, `Or`, `Between`, `Containing`, `OrderBy`, `Top`.

8. Queries and Relationships

EXAMPLE Your own queries with @Query.

```
@Query("SELECT u FROM User u WHERE u.email LIKE %:domain")
List<User> byDomain(@Param("domain") String domain);

@Query(value = "SELECT * FROM users WHERE city = ?1", nativeQuery = true)
List<User> byCity(String city);
```

Relation	Annotation	Example
Many to one	@ManyToOne + @JoinColumn	Many orders, one user
One to many	@OneToMany(mappedBy)	One user, many orders
Many to many	@ManyToMany + @JoinTable	Students and courses
One to one	@OneToOne	User and profile

EXAMPLE Orders belong to a user.

```
@Entity
public class Order {
  @Id @GeneratedValue
  private Long id;

  @ManyToOne(fetch = FetchType.LAZY)
  @JoinColumn(name = "user_id")
  private User user;          // owning side, has the foreign key
}

// inside User
@OneToMany(mappedBy = "user", cascade = CascadeType.ALL)
private List<Order> orders = new ArrayList<>();
```

- **LAZY** loads related data only when used, **EAGER** loads it at once. Prefer LAZY. Returning entities with two-way links to JSON can loop forever, so return DTOs.

REMEMBER The N+1 problem: one query for the list, then one more query for every item. Fix it with `JOIN FETCH` in `@Query` or with `@EntityGraph`.

9. Service Layer and Transactions

- The **service** holds business rules, calls repositories and converts entity to DTO. Controllers stay thin, repositories stay simple.

EXAMPLE A service with transactions.

```
@Service
public class UserService {
  private final UserRepository repo;
  public UserService(UserRepository repo) { this.repo = repo; }

  @Transactional(readOnly = true)
  public UserResponse get(Long id) {
    User u = repo.findById(id)
        .orElseThrow(() -> new NotFoundException("User " + id + " not found"));
    return toResponse(u);
  }

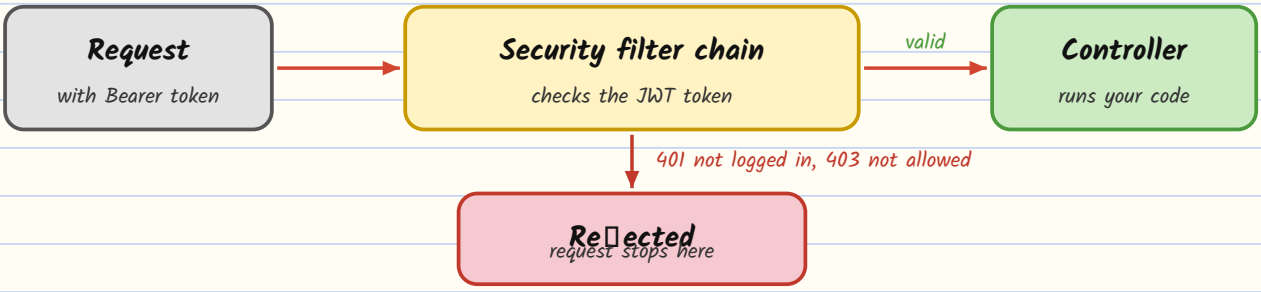
  @Transactional
  public UserResponse create(UserRequest req) {
    if (repo.existsByEmail(req.email()))
      throw new ConflictException("Email already used");
    User u = new User();
    u.setName(req.name());
    u.setEmail(req.email());
    return toResponse(repo.save(u));
  }

  private UserResponse toResponse(User u) {
    return new UserResponse(u.getId(), u.getName(), u.getEmail());
  }
}
```

- **@Transactional** makes all database steps succeed together or roll back together. It rolls back on `RuntimeException` by default. Checked exceptions need `rollbackFor`.
- It works through a proxy, so calling a transactional method from another method of the same class will not start a transaction. Put it on service methods, not controllers.

10. Spring Security Basics

- Adding the security starter locks every endpoint. Spring prints a random password in the console. You then define your own rules.



Term	Meaning
Authentication	Who are you? Login check
Authorization	What may you do? Role check
UserDetailsService	Loads a user from your database
PasswordEncoder	Hashes passwords, use BCrypt
SecurityFilterChain	The rules: which URL needs what
Role	Group like USER or ADMIN. hasRole('ADMIN') checks ROLE_ADMIN

EXAMPLE Security rules (Spring Security 6 style).

```
@Configuration
@EnableMethodSecurity
public class SecurityConfig {
  @Bean
  SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.csrf(csrf -> csrf.disable())          // stateless token API
        .sessionManagement(s -> s
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS))
        .authorizeHttpRequests(a -> a
            .requestMatchers("/api/auth/**").permitAll()
            .requestMatchers("/api/admin/**").hasRole("ADMIN")
            .anyRequest().authenticated());
    return http.build();
  }
}
```

EXAMPLE Load users from the database.

```
@Service
public class DbUserDetailsService implements UserDetailsService {
  private final UserRepository repo;
  public DbUserDetailsService(UserRepository repo) { this.repo = repo; }

  public UserDetails loadUserByUsername(String email) {
    User u = repo.findByEmail(email)
        .orElseThrow(() -> new UsernameNotFoundException("No such user"));
    return org.springframework.security.core.userdetails.User
        .withUsername(u.getEmail())
        .password(u.getPassword())
        .roles(u.getRole().name())
        .build();
  }
}
```

- Turn CSRF off only for ***stateless*** APIs that send tokens in headers. Keep it on for apps that use cookies or sessions. Method rules: `@PreAuthorize('hasRole("ADMIN")')`.

II. Tokens and Roles

- **JWT** flow: the user logs in, the server returns a signed token, the client sends `Authorization: Bearer token` on each call, and a ***filter*** checks the token before your

controller runs.

EXAMPLE *JwtService (jjwt library).*

```
@Service
public class JwtService {
  private final SecretKey key;
  public JwtService(@Value("${app.jwt.secret}") String secret) {
    this.key = Keys.hmacShaKeyFor(secret.getBytes(StandardCharsets.UTF_8));
  }

  public String create(String email) {
    return Jwts.builder()
        .subject(email)
        .issuedAt(new Date())
        .expiration(new Date(System.currentTimeMillis() + 3_600_000))
        .signWith(key)
        .compact();
  }

  public String emailFrom(String token) {
    return Jwts.parser().verifyWith(key).build()
        .parseSignedClaims(token).getPayload().getSubject();
  }
}
```

EXAMPLE *Filter that reads the token.*

```
@Component
public class JwtFilter extends OncePerRequestFilter {
  private final JwtService jwt;
  private final UserDetailsService users;
  // constructor omitted

  protected void doFilterInternal(HttpServletRequest req,
      HttpServletResponse res, FilterChain chain)
      throws ServletException, IOException {
    String header = req.getHeader("Authorization");
    if (header != null && header.startsWith("Bearer ")) {
      String email = jwt.emailFrom(header.substring(7));
      UserDetails u = users.loadUserByUsername(email);
      var auth = new UsernamePasswordAuthenticationToken(
          u, null, u.getAuthorities());
      SecurityContextHolder.getContext().setAuthentication(auth);
    }
    chain.doFilter(req, res);
  }
}
// in SecurityConfig:  http.addFilterBefore(jwtFilter, UsernamePasswordAuthenticationFilter.class);
```

EXAMPLE Login endpoint.

```
@PostMapping("/api/auth/login")
public Map<String, String> login(@RequestBody LoginRequest req) {
  authManager.authenticate(
      new UsernamePasswordAuthenticationToken(req.email(), req.password()));
  return Map.of("token", jwt.create(req.email()));
}

@Bean
AuthenticationManager authManager(AuthenticationConfiguration c) throws Exception {
  return c.getAuthenticationManager();
}
```

Code	Meaning
401 Unauthorized	Not logged in, or the token is missing, wrong or expired
403 Forbidden	Logged in, but the role is not allowed

REMEMBER Keep `app.jwt.secret` (at least 32 characters) in an environment variable, never in Git. Use short token life. Catch `JwtException` in the filter so a bad token gives 401, not 500. Allow your React origin with a CORS configuration.

12. Config, Profiles and Testing

- Use ***profiles*** for different settings: `application-dev.properties` and `application-prod.properties`. Switch with `spring.profiles.active=prod` or the env variable `SPRING_PROFILES_ACTIVE`. Environment variables override files.

EXAMPLE Custom settings and logging.

```
app.jwt.secret=${JWT_SECRET}
app.jwt.expiry-minutes=60
logging.level.org.hibernate.SQL=DEBUG
management.endpoints.web.exposure.include=health,info
```

- Add ***Actuator*** and open `/actuator/health` to see if the app is alive. Hosting platforms use it as a health check.

Test type	Use
`@SpringBootTest`	Whole app, slowest, few of these
`@WebMvcTest`	Only controllers, with MockMvc
`@DataJpaTest`	Only repositories, with an in-memory or Testcontainers database
Mockito	Unit test services with fake repositories

EXAMPLE Controller test.

```
@WebMvcTest(UserController.class)
class UserControllerTest {
  @Autowired MockMvc mvc;
  @MockitoBean UserService service;   // @MockBean in older Boot

  @Test
  @WithMockUser
  void returnsUser() throws Exception {
    when(service.get(1L)).thenReturn(new UserResponse(1L, "Asha", "a@x.com"));
    mvc.perform(get("/api/users/1"))
       .andExpect(status().isOk())
       .andExpect(jsonPath("$.name").value("Asha"));
  }
}
```

EXAMPLE Dockerfile.

```
FROM eclipse-temurin:21-jre
COPY target/app.jar app.jar
ENTRYPOINT ["java", "-jar", "app.jar"]
```

13. Projects and Practice

Project	What you practice
1. Student management API	CRUD, DTOs, validation, exceptions
2. Blog API with login	JWT, roles, relations, pagination
3. Library system	Many to many, transactions, search
4. E-commerce backend	Orders, stock, @Transactional, admin role
5. Wallet or banking API	Transactions, locking, rollback, audit
6. Job portal	Roles, file upload, filters, email

➤ Common mistakes

Mistake	Fix
Field injection with @Autowired	Use constructor injection
Returning entities from controllers	Return DTOs
No @Transactional on multi step changes	Add it on the service method
N+1 queries	JOIN FETCH or @EntityGraph
ddl-auto=update in production	Use validate and Flyway or Liquibase
Plain passwords and secrets in code	BCrypt and environment variables
Main class in a wrong package	Keep it in the root package

➤ Practice questions

1. What is Spring Boot? What are starters and auto-configuration?
2. Explain IoC, Dependency Injection and beans.
3. @Component vs @Service vs @Repository. @Controller vs @RestController.
4. PUT vs PATCH. @PathVariable vs @RequestParam.
5. How do you validate input and handle exceptions globally?
6. JPA vs Hibernate vs Spring Data JPA. What is the N+1 problem?
7. What does @Transactional do? Why can it silently fail?
8. Explain how JWT authentication works with Spring Security.
9. 401 vs 403. What is CSRF and when can it be disabled?
10. How do profiles and environment variables work?

- Build one complete API with login, validation and tests. Then add Docker and deploy it.