Backend REST APIs — Spring Boot, Node & Django

Web Technologies — Lecture 10 (10a, 10b, 10c)

Masoud Hamad

State University of Zanzibar (SUZA)

Semester II, 2025/2026

Outline

Three Stacks, Same Pattern

The same REST API can be built in many languages. We'll see three popular stacks.

Stack	Language	Common for
Spring Boot	Java	Enterprise, banks, government
Node.js + Express	JavaScript	Startups, full-stack JS shops
Django REST Framework	Python	Data-heavy, ML/AI products

Plan: 10a Spring Boot in depth, 10b Node + 10c Django for comparison.

The Same Resource in Three Stacks

We'll build the same endpoint everywhere:

```
GET    /api/students          -> list
POST   /api/students          -> create
GET    /api/students/{id}     -> read one
PUT    /api/students/{id}     -> update
DELETE /api/students/{id}     -> delete
```

Why Spring Boot?

- Most widely used Java backend framework
- "Convention over configuration" — sensible defaults
- Auto-configures embedded server, JSON, validation, etc.
- Huge ecosystem (Security, Data, Cloud, Batch...)
- You already know Java from OOP and Advanced Java

Project Generation

Use `start.spring.io` or the CLI:

```
spring init --dependencies=web,data-jpa,h2 students-api
cd students-api
./mvnw spring-boot:run
```

Default port: 8080. Default profile: `dev` (in-memory H2 DB).

Entity (JPA)

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

    @Column(nullable = false)
    private String name;

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

    private String course;

    // getters / setters / constructors
}
```

Repository — Zero SQL

```
public interface StudentRepository
        extends JpaRepository<Student, Long> {

    List<Student> findByCourse(String course);
    Optional<Student> findByEmail(String email);
}
```

Spring Data generates the implementation at runtime — you write *zero* SQL for these.

Service Layer

```
@Service
public class StudentService {
    private final StudentRepository repo;

    public StudentService(StudentRepository repo) { this.repo = repo; }

    public List<Student> all() { return repo.findAll(); }

    public Student byId(Long id) {
        return repo.findById(id)
            .orElseThrow(() -> new ResponseStatusException(NOT_FOUND));
    }

    public Student create(Student s) { return repo.save(s); }
    public void delete(Long id)      { repo.deleteById(id); }
}
```

Controller — The HTTP Layer

```
@RestController
@RequestMapping("/api/students")
public class StudentController {
    private final StudentService svc;
    public StudentController(StudentService svc) { this.svc = svc; }

    @GetMapping              public List<Student> list()              { return svc.
        all(); }
    @GetMapping("/{id}")     public Student get(@PathVariable Long id){ return svc.
        byId(id); }
    @PostMapping             @ResponseStatus(CREATED)
    public Student create(@Valid @RequestBody Student s)              { return svc.
        create(s); }
    @DeleteMapping("/{id}")  @ResponseStatus(NO_CONTENT)
    public void delete(@PathVariable Long id)                         { svc.delete(
        id); }
}
```

Validation & Errors

```
public class Student {
    @NotBlank private String name;
    @Email    @NotBlank private String email;
}
```

```
@RestControllerAdvice
public class ApiExceptionHandler {
    @ExceptionHandler(MethodArgumentNotValidException.class)
    @ResponseStatus(BAD_REQUEST)
    public Map<String, String> handleValidation(MethodArgumentNotValidException e) {
        return Map.of("error", e.getMessage());
    }
}
```

application.properties

```
spring.datasource.url=jdbc:postgresql://localhost:5432/students
spring.datasource.username=postgres
spring.datasource.password=${DB_PASS}
spring.jpa.hibernate.ddl-auto=update
server.port=8080
```

For production, replace H2 with PostgreSQL by adding the driver dependency and these properties — no code changes.

Setup

```
npm init -y
npm install express better-sqlite3 zod
```

Same API in Express

```
import express from "express";
import Database from "better-sqlite3";

const app = express();
app.use(express.json());
const db = new Database("students.db");
db.exec(`CREATE TABLE IF NOT EXISTS students (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  name TEXT NOT NULL, email TEXT UNIQUE, course TEXT)`);

app.get("/api/students", (req, res) => {
  res.json(db.prepare("SELECT * FROM students").all());
});

app.post("/api/students", (req, res) => {
  const { name, email, course } = req.body;
  const r = db.prepare("INSERT INTO students(name,email,course) VALUES(?,?,?)")
              .run(name, email, course);
  res.status(201).json({ id: r.lastInsertRowid, name, email, course });
});

app.delete("/api/students/:id", (req, res) => {
  db.prepare("DELETE FROM students WHERE id = ?").run(req.params.id);
```

Express Trade-offs

Pros

- Tiny, unopinionated, very fast to learn
- Same language as the frontend
- Massive npm ecosystem

Cons

- You assemble the stack (DB, validation, auth, logging)
- No built-in dependency injection or service layering
- Easy to write spaghetti without discipline

Setup

```
pip install django djangorestframework
django-admin startproject backend .
python manage.py startapp students
```

Same API in DRF

```
# students/models.py
class Student(models.Model):
    name   = models.CharField(max_length=100)
    email  = models.EmailField(unique=True)
    course = models.CharField(max_length=20, blank=True)

# students/serializers.py
class StudentSerializer(serializers.ModelSerializer):
    class Meta: model = Student; fields = "__all__"

# students/views.py
class StudentViewSet(viewsets.ModelViewSet):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer

# urls.py
router = DefaultRouter()
router.register(r"students", StudentViewSet)
urlpatterns = [path("api/", include(router.urls))]
```

$\sim$**15 lines** for full CRUD — batteries included.

DRF Trade-offs

Pros

- ViewSets + Routers = full CRUD with almost no code
- Built-in admin, auth, permissions, pagination, filtering
- Excellent for data-heavy, ML-adjacent backends

Cons

- Heavier than Express
- "Magic" makes debugging harder for newcomers
- Async support is newer/less mature than Node

When to Choose What

Choose...	**When...**
Spring Boot	Strong typing, large team, enterprise integration
Node + Express	Same language as frontend, lots of I/O, real-time
Django REST	Data heavy, ML/AI, you want admin + auth for free

All three are excellent. Pick what your team knows.

Common Best Practices

- Validate input at the boundary (DTOs / serializers)
- Use **layers**: controller $\rightarrow$ service $\rightarrow$ repository
- Return proper status codes (201 created, 204 deleted, etc.)
- Document with OpenAPI / Swagger
- Log structured JSON, not free-form strings
- Cover happy + edge cases with integration tests

Summary

- REST APIs follow the same pattern in any language
- Spring Boot: layered, type-safe, enterprise-ready
- Express: minimal, flexible, JS everywhere
- Django REST: opinionated, batteries-included
- Pick once, then focus on the **API design**, not the language

Next: Authentication, sessions & web security.

Questions?