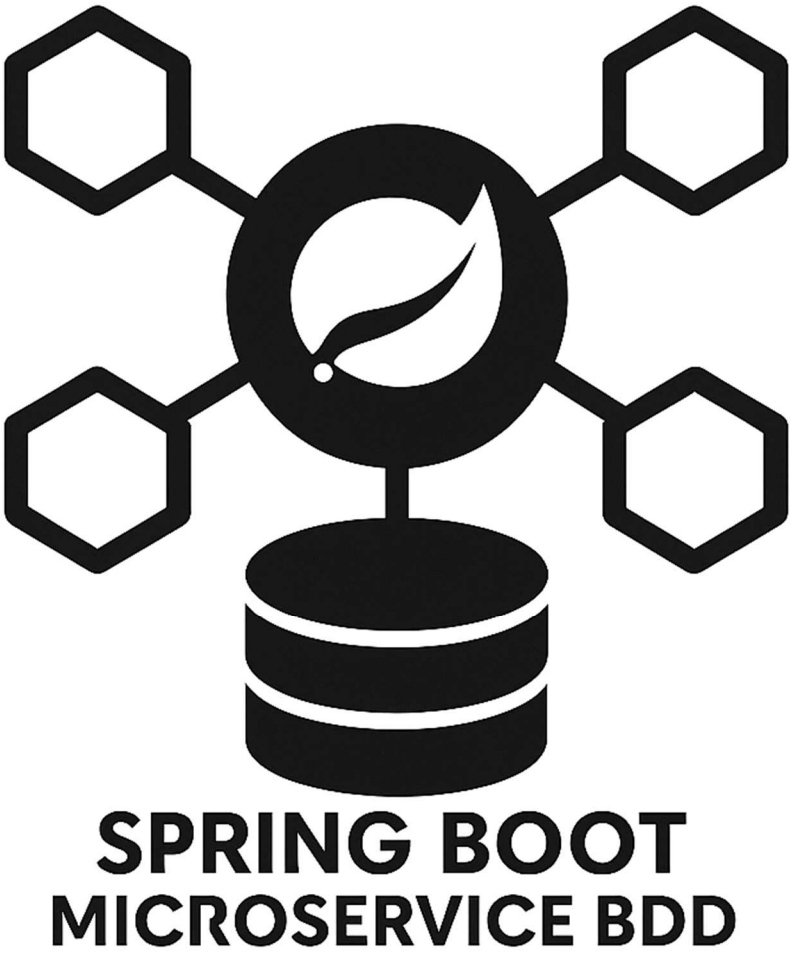
Approche BDD pour la transformation d’un monolithe en microservices avec Spring Boot



Dr. BADR EL KHALYLY

Introduction

La transformation d'une application monolithique vers une architecture microservices est un enjeu majeur pour améliorer la scalabilité, la maintenabilité et la rapidité de livraison logicielle. Ce document propose une démarche basée sur la méthode Behavior-Driven Development (BDD) afin d'accompagner cette migration.

Nous partons d'un exemple d'application monolithique développée avec Spring Boot, organisée autour de plusieurs contrôleurs métier. À travers la rédaction de scénarios BDD, nous analysons les comportements attendus du système et identifions les limites naturelles entre les différents domaines fonctionnels.

Cette démarche facilite la définition de microservices cohérents, autonomes et alignés sur les besoins métier, tout en assurant la continuité fonctionnelle grâce à des tests automatisés. Nous expliquons ensuite comment orchestrer la migration progressive, en garantissant une communication fluide entre les microservices nouvellement créés.

Exemple d'application monolithique Spring Boot

Dans cet exemple, nous considérons une application monolithique conçue avec Spring Boot, qui gère un système de formation en ligne. Elle est organisée autour de plusieurs contrôleurs, chacun responsable d'un domaine fonctionnel précis :

1. Entités

```
// Role.java (Enum)
public enum Role {
   STUDENT, TRAINER
}

// User.java
@Entity
public class User {
   @Id @GeneratedValue
   private Long id;
   private String name;
   private String email;

   @Enumerated(EnumType.STRING)
   private Role role;
```

```
    // getters & setters
}

// Course.java
@Entity
public class Course {
    @Id @GeneratedValue
    private Long id;
    private String title;
    private String description;

    // getters & setters
}

// Enrollment.java
@Entity
public class Enrollment {
    @Id @GeneratedValue
    private Long id;

    @ManyToOne
    private User student;

    @ManyToOne
    private Course course;

    private LocalDateTime enrolledAt;

    // getters & setters
}
```

```
// Evaluation.java
@Entity
public class Evaluation {
    @Id @GeneratedValue
    private Long id;

    @ManyToOne
    private Enrollment enrollment;

    private Double grade;

    private String remarks;

    // getters & setters
}
```

---

2. Repositories

```
public interface UserRepository extends JpaRepository<User, Long> {
    List<User> findByRole(Role role);
}

public interface CourseRepository extends JpaRepository<Course, Long> {}

public interface EnrollmentRepository extends JpaRepository<Enrollment, Long> {
    List<Enrollment> findByStudentId(Long studentId);
    List<Enrollment> findByCourseId(Long courseId);
}

public interface EvaluationRepository extends JpaRepository<Evaluation, Long> {
    List<Evaluation> findByEnrollmentId(Long enrollmentId);
```

```
}
```

---

3. Services

```
@Service
public class UserService {
    @Autowired private UserRepository userRepository;

    public List<User> getUsersByRole(Role role) {
        return userRepository.findByRole(role);
    }
    public User save(User user) {
        return userRepository.save(user);
    }
    public Optional<User> getById(Long id) {
        return userRepository.findById(id);
    }
}

@Service
public class CourseService {
    @Autowired private CourseRepository courseRepository;

    public List<Course> getAll() {
        return courseRepository.findAll();
    }
    public Optional<Course> getById(Long id) {
        return courseRepository.findById(id);
    }
    public Course save(Course course) {
        return courseRepository.save(course);
    }
```

```
    public void delete(Long id) {
        courseRepository.deleteById(id);
    }
}

@Service
public class EnrollmentService {
    @Autowired private EnrollmentRepository enrollmentRepository;
    @Autowired private UserService userService;
    @Autowired private CourseService courseService;

    public Enrollment enrollStudent(Long studentId, Long courseId) {
        User student = userService.getById(studentId)
            .orElseThrow(() -> new RuntimeException("Student not found"));
        if (student.getRole() != Role.STUDENT)
            throw new RuntimeException("User is not a student");

        Course course = courseService.getById(courseId)
            .orElseThrow(() -> new RuntimeException("Course not found"));

        Enrollment enrollment = new Enrollment();
        enrollment.setStudent(student);
        enrollment.setCourse(course);
        enrollment.setEnrolledAt(LocalDateTime.now());
        return enrollmentRepository.save(enrollment);
    }

    public List<Enrollment> getEnrollmentsForStudent(Long studentId) {
        return enrollmentRepository.findByStudentId(studentId);
    }
}
```

```
@Service
public class EvaluationService {
    @Autowired private EvaluationRepository evaluationRepository;
    @Autowired private EnrollmentRepository enrollmentRepository;

    public Evaluation addEvaluation(Long enrollmentId, Double grade, String remarks) {
        Enrollment enrollment = enrollmentRepository.findById(enrollmentId)
            .orElseThrow(() -> new RuntimeException("Enrollment not found"));

        Evaluation evaluation = new Evaluation();
        evaluation.setEnrollment(enrollment);
        evaluation.setGrade(grade);
        evaluation.setRemarks(remarks);

        return evaluationRepository.save(evaluation);
    }

    public List<Evaluation> getEvaluationsByEnrollment(Long enrollmentId) {
        return evaluationRepository.findByEnrollmentId(enrollmentId);
    }
}
```

---

4. Contrôleurs REST

```
@RestController
@RequestMapping("/users")
public class UserController {
    @Autowired private UserService userService;

    @GetMapping
```

```
    public List<User> getUsersByRole(@RequestParam Role role) {
        return userService.getUsersByRole(role);
    }

    @PostMapping
    public User createUser(@RequestBody User user) {
        return userService.save(user);
    }
}

@RestController
@RequestMapping("/courses")
public class CourseController {
    @Autowired private CourseService courseService;

    @GetMapping
    public List<Course> getAllCourses() {
        return courseService.getAll();
    }

    @GetMapping("/{id}")
    public ResponseEntity<Course> getCourse(@PathVariable Long id) {
        return courseService.getById(id)
                .map(ResponseEntity::ok)
                .orElse(ResponseEntity.notFound().build());
    }

    @PostMapping
    public Course createCourse(@RequestBody Course course) {
        return courseService.save(course);
    }
```

```
    @PutMapping("/{id}")
    public ResponseEntity<Course> updateCourse(@PathVariable Long id, @RequestBody
Course course) {
        return courseService.getById(id).map(existing -> {
            existing.setTitle(course.getTitle());
            existing.setDescription(course.getDescription());
            return ResponseEntity.ok(courseService.save(existing));
        }).orElse(ResponseEntity.notFound().build());
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteCourse(@PathVariable Long id) {
        courseService.delete(id);
        return ResponseEntity.noContent().build();
    }
}

@RestController
@RequestMapping("/enrollments")
public class EnrollmentController {
    @Autowired private EnrollmentService enrollmentService;

    @PostMapping
    public Enrollment enrollStudent(@RequestParam Long studentId, @RequestParam Long
courseId) {
        return enrollmentService.enrollStudent(studentId, courseId);
    }

    @GetMapping("/student/{studentId}")
    public List<Enrollment> getEnrollmentsForStudent(@PathVariable Long studentId) {
        return enrollmentService.getEnrollmentsForStudent(studentId);
```

```
    }
}

@RestController
@RequestMapping("/evaluations")
public class EvaluationController {
    @Autowired private EvaluationService evaluationService;

    @PostMapping
    public Evaluation addEvaluation(
            @RequestParam Long enrollmentId,
            @RequestParam Double grade,
            @RequestParam(required = false) String remarks) {
        return evaluationService.addEvaluation(enrollmentId, grade, remarks);
    }

    @GetMapping("/enrollment/{enrollmentId}")
    public List<Evaluation> getEvaluations(@PathVariable Long enrollmentId) {
        return evaluationService.getEvaluationsByEnrollment(enrollmentId);
    }
}
```

Contexte

- Tu as une application monolithique Spring Boot avec plusieurs contrôleurs (ex: UserController, CourseController, etc.).
- Tu souhaites découper ce monolithe en microservices indépendants.
- Tu veux faire cela de manière structurée, avec des scénarios de comportement (BDD) pour bien comprendre les besoins métier, les limites naturelles et garantir que chaque microservice a un périmètre clair.

Pourquoi BDD dans cette démarche ?

- BDD permet d'écrire les comportements attendus du système en langage naturel, compréhensible par tous (technique, métier).

- Cela facilite la découverte des **domaines fonctionnels** (bounded contexts) qui formeront les microservices.
- Permet d'identifier clairement les **frontières fonctionnelles**, évitant les microservices trop gros ou trop petits.
- Les scénarios BDD aident aussi à définir les APIs et les contrats de communication entre microservices.

Étapes détaillées de l'approche

---

1. Analyse initiale : recensement des contrôleurs et de leurs responsabilités

- Extraire la liste des contrôleurs existants dans le monolithe.
- Pour chaque contrôleur, comprendre précisément quelles fonctionnalités métier il supporte.

Exemple :

Contrôleur	**Responsabilité principale**
UserController	Gestion des utilisateurs (étudiants, formateurs)
CourseController	Gestion des cours
EnrollmentController	Gestion des inscriptions
EvaluationController	Gestion des évaluations

---

2. Définition des scénarios BDD (Given-When-Then) par contrôleur

Pour chaque contrôleur, rédiger des scénarios BDD qui décrivent les comportements métier attendus. Cela peut être fait en collaboration avec les métiers, PO, analystes.

Exemple pour UserController :

```
Feature: Gestion des utilisateurs

  Scenario: Créer un nouvel étudiant
    Given je suis un administrateur
    When je crée un utilisateur avec le rôle STUDENT
    Then l'utilisateur est enregistré dans la base
```

```
Scenario: Lister les formateurs
  Given des utilisateurs avec différents rôles
  When je demande la liste des utilisateurs avec rôle TRAINER
  Then je reçois uniquement les formateurs
```

Chaque scénario correspond à un comportement fonctionnel.

---

3. Identification des limites fonctionnelles (Bounded Contexts) à partir des scénarios

- Chaque groupe cohérent de scénarios métier (par exemple liés aux utilisateurs) constitue un **Bounded Context**.
- Ces Bounded Contexts correspondent à des microservices potentiels.

Ici :

- User Management → User Service
- Course Management → Course Service
- Enrollment Management → Enrollment Service
- Evaluation Management → Evaluation Service

---

4. Validation des frontières par analyse des dépendances

- Étudier les dépendances entre contrôleurs : quels contrôleurs/appels dépendent les uns des autres ?
- Identifier les dépendances critiques qui pourraient nécessiter des API entre microservices.

---

5. Définition des API entre microservices

- À partir des scénarios BDD, définir les contrats d'API REST qui permettront la communication entre microservices.
- Par exemple, Enrollment Service doit appeler User Service pour vérifier que l'étudiant existe.

---

6. Rédaction des scénarios BDD d'intégration

- Écrire des scénarios BDD qui couvrent les interactions entre microservices.

Exemple :

```
Feature: Inscription d'un étudiant à un cours
```

```
Scenario: Inscrire un étudiant existant à un cours existant
  Given un étudiant existant avec l'id 123
  And un cours existant avec l'id 456
  When je crée une inscription pour cet étudiant et ce cours
  Then l'inscription est validée et enregistrée

Scenario: Inscrire un étudiant non existant
  Given un étudiant avec l'id 999 qui n'existe pas
  When je tente de créer une inscription pour cet étudiant
  Then un message d'erreur "Étudiant non trouvé" est retourné
```

Ces scénarios guident la conception des appels inter-microservices.

---

7. Découpage progressif et migration

- On migre progressivement chaque contrôleur vers un microservice indépendant.
- Le monolithe diminue à chaque migration.
- Utiliser des tests automatisés issus des scénarios BDD pour garantir que les comportements restent corrects.

---

8. Implémentation

Exemple d’organisation projet (4 projets Maven/Gradle)

```
user-service/
  src/main/java/...
  application.properties (BDD User)

course-service/
  src/main/java/...
  application.properties (BDD Course)

enrollment-service/
  src/main/java/...
  application.properties (BDD Enrollment)
```

evaluation-service/

src/main/java/...

application.properties (BDD Evaluation)

Exemple concret : Enrollment Service appelle User Service et Course Service

EnrollmentService (dans enrollment-service)

```
@Service
public class EnrollmentService {
    private final RestTemplate restTemplate;

    @Autowired
    public EnrollmentService(RestTemplate restTemplate) {
        this.restTemplate = restTemplate;
    }

    public Enrollment enrollStudent(Long studentId, Long courseId) {
        // Vérifier que l'étudiant existe dans User Service
        ResponseEntity<User> userResponse = restTemplate.getForEntity(
                "http://user-service/users/" + studentId, User.class);
        if (!userResponse.getStatusCode().is2xxSuccessful()) {
            throw new RuntimeException("Student not found");
        }
        User student = userResponse.getBody();
        if (student.getRole() != Role.STUDENT) {
            throw new RuntimeException("User is not a student");
        }

        // Vérifier que le cours existe dans Course Service
        ResponseEntity<Course> courseResponse = restTemplate.getForEntity(
                "http://course-service/courses/" + courseId, Course.class);
        if (!courseResponse.getStatusCode().is2xxSuccessful()) {
```

```
            throw new RuntimeException("Course not found");
        }
        Course course = courseResponse.getBody();

        // Créer l'inscription
        Enrollment enrollment = new Enrollment();
        enrollment.setStudentId(studentId);
        enrollment.setCourseId(courseId);
        enrollment.setEnrolledAt(LocalDateTime.now());

        // Sauvegarder dans la base Enrollment
        return enrollmentRepository.save(enrollment);
    }
}
```

Ici user-service et course-service doivent être accessibles via un service discovery (ex : Eureka) ou via une URL fixe.

5. Simplification : utiliser OpenFeign pour l'appel REST

Ajoute dans enrollment-service :

```
@FeignClient(name = "user-service")
public interface UserClient {
    @GetMapping("/users/{id}")
    User getUserById(@PathVariable Long id);
}

@FeignClient(name = "course-service")
public interface CourseClient {
    @GetMapping("/courses/{id}")
    Course getCourseById(@PathVariable Long id);
}
```

Puis dans EnrollmentService :

```
@Service
public class EnrollmentService {
    private final UserClient userClient;
    private final CourseClient courseClient;

    public EnrollmentService(UserClient userClient, CourseClient courseClient) {
        this.userClient = userClient;
        this.courseClient = courseClient;
    }

    public Enrollment enrollStudent(Long studentId, Long courseId) {
        User student = userClient.getUserById(studentId);
        if (student.getRole() != Role.STUDENT) {
            throw new RuntimeException("User is not a student");
        }
        Course course = courseClient.getCourseById(courseId);

        Enrollment enrollment = new Enrollment();
        enrollment.setStudentId(studentId);
        enrollment.setCourseId(courseId);
        enrollment.setEnrolledAt(LocalDateTime.now());
        return enrollmentRepository.save(enrollment);
    }
}
```

Autres conseils pour le découpage microservices

- Base de données : Chaque microservice avec sa propre base (ex: Postgres, MySQL, Mongo, etc.).
- Sécurité : Gestion des rôles et authentification peut être déléguée à un microservice Auth ou gérée via API Gateway (ex: OAuth2, JWT).

- Service Discovery : Utiliser Spring Cloud Netflix Eureka pour enregistrer les services et les retrouver dynamiquement.
- API Gateway : Utiliser Spring Cloud Gateway pour router les appels client vers les microservices.
- Logs & Monitoring : Centraliser les logs avec ELK (Elasticsearch, Logstash, Kibana) ou autre outil.

Résumé de la démarche BDD dans la transformation

Étape	**Objectif**	**Résultat**
1. Recensement contrôleurs	Comprendre la structure fonctionnelle	Liste contrôleurs + responsabilités
2. Rédaction scénarios BDD	Décrire précisément le comportement métier	Scénarios métier clairs et validés
3. Identification BC	Définir les périmètres fonctionnels	Définition des microservices à créer
4. Analyse dépendances	Identifier liens et appels entre modules	Cartographie des dépendances
5. Définition API	Formaliser les échanges entre microservices	Contrats d'API REST ou events
6. Scénarios intégration	Garantir la cohérence multi-services	Tests d'intégration basés sur le comportement
7. Migration progressive	Découper, migrer et valider	Microservices autonomes et testés

---

Avantages de cette approche

- **Compréhension métier forte** → microservices alignés sur le métier.
- **Collaboration facilitée** avec les équipes métiers.
- **Validation continue** grâce aux scénarios BDD automatisés.
- **Limitation du risque** de rupture fonctionnelle à chaque étape de migration.
- **Chaque microservice a sa base de donnés**