P1C5 - Exposez vos fonctionnalités via une API REST

Vérifiez votre travail

Les statuts et les résultats. Les statuts rejoignent une énumération, dans le package `leave` :

```
enum LeaveStatus {
    PENDING,
    APPROVED,
    REFUSED
}

enum ApprovalResult { APPROVED, INSUFFICIENT_BALANCE, NOT_FOUND, NOT_PENDING }

enum RefusalResult { REFUSED, NOT_FOUND, NOT_PENDING }
```

Dans `LeaveRequest`, le champ `status`, son getter et son setter utilisent `LeaveStatus`. L'annotation `@Enumerated(EnumType.STRING)` permet à JPA de stocker les noms des constantes, comme `PENDING`, sous forme de chaînes dans la base, conformément à `data.sql`. Le DTO et le filtre utilisent eux aussi l'énumération : une valeur de filtre inconnue reçoit une réponse `400`.

Le repository des demandes. Il reçoit les deux requêtes de lecture dont le service a besoin :

```
interface LeaveRequestRepository extends Repository<LeaveRequest, Long> {

    Optional<LeaveRequest> findById(Long id);

    List<LeaveRequest> findAll(Sort sort);

    List<LeaveRequest> findByStatus(LeaveStatus status, Sort sort);

    LeaveRequest save(LeaveRequest request);
}
```

`findByStatus` est la méthode dérivée annoncée au chapitre 4 : le filtre est traduit en clause `where`. Le paramètre `Sort` ajoute le tri à la requête SQL, sans multiplier les noms de méthodes. Sans filtre, la route renvoie toutes les demandes : dans un vrai projet, il faudrait ajouter une pagination, que Spring Data prend en charge avec un paramètre `Pageable`. Elle reste hors du périmètre de ce cours.

Le service. Voici les méthodes nouvelles ou modifiées de `LeaveService`. Le constructeur, `getBalance` et `computeBalance` ne changent pas :

```
public List<LeaveRequest> getRequests(LeaveStatus status) {
    var byStartDate = Sort.by("startDate");
    return status == null
            ? leaveRequestRepository.findAll(byStartDate)
            : leaveRequestRepository.findByStatus(status, byStartDate);
}

public Optional<LeaveRequest> getRequest(long requestId) {
    return leaveRequestRepository.findById(requestId);
}
```

```
@Transactional
public Optional<LeaveRequest> create(long employeeId, LocalDate startDate, LocalDate endDate) {
    if (employeeRepository.findById(employeeId).isEmpty()) {
        return Optional.empty();
    }
    var request = new LeaveRequest();
    request.setEmployeeId(employeeId);
    request.setStartDate(startDate);
    request.setEndDate(endDate);
    request.setWorkingDays(countWorkingDays(startDate, endDate));
    request.setStatus(LeaveStatus.PENDING);
    return Optional.of(leaveRequestRepository.save(request));
}

@Transactional
public ApprovalResult approve(long requestId) {
    var found = leaveRequestRepository.findById(requestId);
    if (found.isEmpty()) {
        return ApprovalResult.NOT_FOUND;
    }
    var request = found.get();
    if (request.getStatus() != LeaveStatus.PENDING) {
        return ApprovalResult.NOT_PENDING;
    }
    var employee = employeeRepository.findById(request.getEmployeeId()).orElseThrow();

    if (computeBalance(employee) < request.getWorkingDays()) {
        request.setStatus(LeaveStatus.REFUSED);
        leaveRequestRepository.save(request);
        return ApprovalResult.INSUFFICIENT_BALANCE;
    }

    request.setStatus(LeaveStatus.APPROVED);
    employee.setDaysTaken(employee.getDaysTaken() + request.getWorkingDays());
    leaveRequestRepository.save(request);
    employeeRepository.save(employee);

    notifier.notify(employee, "Votre demande de congés est approuvée.");
    return ApprovalResult.APPROVED;
}

@Transactional
public RefusalResult refuse(long requestId) {
    var found = leaveRequestRepository.findById(requestId);
    if (found.isEmpty()) {
        return RefusalResult.NOT_FOUND;
    }
    var request = found.get();
    if (request.getStatus() != LeaveStatus.PENDING) {
        return RefusalResult.NOT_PENDING;
    }
    request.setStatus(LeaveStatus.REFUSED);
    leaveRequestRepository.save(request);
    return RefusalResult.REFUSED;
}

private static int countWorkingDays(LocalDate startDate, LocalDate endDate) {
    return (int) startDate.datesUntil(endDate.plusDays(1))
            .filter(day -> day.getDayOfWeek() != DayOfWeek.SATURDAY
                    && day.getDayOfWeek() != DayOfWeek.SUNDAY)
            .count();
}
```

La vérification du statut se trouve dans le service et non dans le contrôleur : c'est une règle métier, qui s'appliquera quel que soit le client. `save` renvoie la demande avec l'identifiant généré par la base : c'est

lui qui servira à construire l'en-tête `Location`. `datesUntil` produit les jours de la période, borne de fin exclue, d'où le `plusDays(1)`.

Le calcul des jours ouvrés ignore les jours fériés. C'est une simplification du cours. Dans un vrai projet, ce serait une nouvelle question à poser à Samuel, et sans doute une nouvelle stratégie.

Les contrôleurs. Le solde a son propre contrôleur, sous l'adresse des salariés :

```
@RestController
@RequestMapping("/api/employees")
class BalanceController {

    private final LeaveService leaveService;

    BalanceController(LeaveService leaveService) {
        this.leaveService = leaveService;
    }

    @GetMapping("/{id}/balance")
    BalanceDto getBalance(@PathVariable long id) {
        return leaveService.getBalance(id)
                .map(balance -> new BalanceDto(id, balance))
                .orElseThrow(() -> new ResponseStatusException(
                        HttpStatus.NOT_FOUND, "Ce salarié n'existe pas."));
    }
}
```

Les demandes ont le leur :

```
@RestController
@RequestMapping("/api/leave-requests")
class LeaveRequestsController {

    private final LeaveService leaveService;

    LeaveRequestsController(LeaveService leaveService) {
        this.leaveService = leaveService;
    }

    @GetMapping
    List<LeaveRequestDto> getAll(@RequestParam(required = false) LeaveStatus status) {
        return leaveService.getRequests(status).stream()
                .map(LeaveRequestDto::from)
                .toList();
    }

    @GetMapping("/{id}")
    LeaveRequestDto getById(@PathVariable long id) {
        return leaveService.getRequest(id)
                .map(LeaveRequestDto::from)
                .orElseThrow(() -> notFound());
    }

    @PostMapping
    ResponseEntity<LeaveRequestDto> create(@Valid @RequestBody CreateLeaveRequestDto dto) {
        var request = leaveService.create(dto.employeeId(), dto.startDate(), dto.endDate())
                .orElseThrow(() -> new ResponseStatusException(
                        HttpStatus.BAD_REQUEST, "Ce salarié n'existe pas."));

        var location = ServletUriComponentsBuilder.fromCurrentRequest()
                .path("/{id}")
                .buildAndExpand(request.getId())
                .toUri();
        return ResponseEntity.created(location).body(LeaveRequestDto.from(request));
    }
```

```
    @PostMapping("/{id}/approval")
    LeaveRequestDto approve(@PathVariable long id) {
        return switch (leaveService.approve(id)) {
            case APPROVED, INSUFFICIENT_BALANCE -> getById(id);
            case NOT_FOUND -> throw notFound();
            case NOT_PENDING -> throw alreadyProcessed();
        };
    }

    @PostMapping("/{id}/refusal")
    LeaveRequestDto refuse(@PathVariable long id) {
        return switch (leaveService.refuse(id)) {
            case REFUSED -> getById(id);
            case NOT_FOUND -> throw notFound();
            case NOT_PENDING -> throw alreadyProcessed();
        };
    }

    private static ResponseStatusException notFound() {
        return new ResponseStatusException(HttpStatus.NOT_FOUND, "Cette demande n'existe pas.");
    }

    private static ResponseStatusException alreadyProcessed() {
        return new ResponseStatusException(HttpStatus.CONFLICT, "Cette demande a déjà été traitée.");
    }
}
```

Plusieurs détails méritent votre attention :

- les méthodes qui répondent toujours `200` renvoient directement le DTO ; seule `create` a besoin de `ResponseEntity`, pour choisir le statut `201` et ajouter l'en-tête `Location` ;
- `ServletUriComponentsBuilder.fromCurrentRequest()` part de l'adresse de la requête reçue, `/api/leave-requests`, et lui ajoute l'identifiant de la demande créée ;
- `@PathVariable long id` attend un nombre : une adresse comme `leave-requests/abc` produit un `400` au format ProblemDetail, sans même appeler votre méthode ;
- dans `create`, `dto.employeeId()` ne peut pas être `null` : si le champ manquait, `@Valid` aurait déjà provoqué un `400` ;
- dans `approve` et `refuse`, un `case` peut lever une exception au lieu de produire une valeur. Le `switch` reste exhaustif : si un résultat s'ajoute un jour, le contrôleur ne compilera plus tant qu'il ne l'aura pas traité ;
- après une décision, `getById` relit la demande pour renvoyer son état à jour. Cela coûte une requête de plus, en échange d'un service qui n'a pas à connaître le format de la réponse.

Les contrôleurs restent fidèles à la règle du chapitre 3 : aucune règle métier, aucun accès direct aux données. Ils traduisent les résultats du service en statuts HTTP et les entités en DTO.

La configuration. `application.properties` contient une ligne de plus que celle du chapitre 4 : `spring.mvc.problemdetails.enabled=true`. Depuis que `ApiExceptionHandler` existe, cette ligne n'a plus d'effet, puisque Spring Boot n'enregistre son gestionnaire que si vous n'en fournissez pas. Vous pouvez la garder, ou la retirer. Le `pom.xml` compte deux dépendances de plus : `spring-boot-starter-validation` et `springdoc-openapi-starter-webmvc-ui`.

Les tests. La classe de test reçoit `@Transactional`, et le test du bug de Leïla utilise la nouvelle route. Il lit le solde dans le JSON avec JsonPath, une bibliothèque fournie par `spring-boot-starter-test` :

```
@SpringBootTest
@AutoConfigureMockMvc
```

```
@Transactional
class LeaveApprovalTests {

    @Autowired
    MockMvcTester mvc;

    @MockitoBean
    LeaveNotifier notifier;

    @Test
    void balanceOfFixedTermEmployeeCountsOnlyFullMonths() throws Exception {
        var result = mvc.get().uri("/api/employees/4/balance").exchange();

        double balance = JsonPath.read(result.getResponse().getContentAsString(), "$.balance");
        assertThat(balance).isCloseTo(4.56, within(0.01));
    }

    // Les autres tests passent de mvc.get().uri("/api/leave/approve?id=1")
    // à mvc.post().uri("/api/leave-requests/1/approval").
}
```

Dans une classe de test, l'injection par champ avec `@Autowired` reste la norme : c'est JUnit, et non Spring, qui crée la classe.

Les deux nouveaux tests vérifient le contrat lui-même :

```
@Test
void approvingRequestTwiceReturnsConflict() {
    assertThat(mvc.post().uri("/api/leave-requests/1/approval")).hasStatusOk();

    assertThat(mvc.post().uri("/api/leave-requests/1/approval"))
            .hasStatus(HttpStatus.CONFLICT);
}

@Test
void creatingRequestEndingBeforeItStartsReturnsBadRequest() {
    assertThat(mvc.post().uri("/api/leave-requests")
            .contentType(MediaType.APPLICATION_JSON)
            .content("""
                    {"employeeId": 4, "startDate": "2026-07-10", "endDate": "2026-07-06"}
                    """))
            .hasStatus(HttpStatus.BAD_REQUEST);
}
```

Le premier test rejoue exactement l'incident de vendredi. S'il avait existé plus tôt, Amina n'aurait pas perdu cinq jours. Le corps JSON du second est écrit directement dans un bloc de texte : vous voyez exactement ce que le client envoie, et vous évitez au passage le piège des deux versions de Jackson.

La distinction attendue est la suivante : une API qui **fonctionne** répond aux appels que son auteur a prévus ; une API qui respecte un **contrat** se comporte de façon prévisible pour tous les clients, y compris ceux qu'il n'a pas prévus, comme l'aperçu de lien d'une messagerie.

Erreurs fréquentes

- **Garder un** `@GetMapping` **pour une action qui modifie des données**, même « provisoirement » : c'est exactement ce qui a provoqué l'incident.
- **Recevoir une entité avec** `@RequestBody` : le client peut alors choisir le statut et le nombre de jours de sa demande.

- **Oublier** `spring-boot-starter-validation` **ou** `@Valid` : les annotations du DTO compilent, mais ne sont jamais vérifiées.
- **Déclarer** `@NotNull` **sur un** `long` : un type primitif n'est jamais `null`, la règle ne détecte pas un champ absent.
- **Vérifier le statut de la demande dans le contrôleur** : la règle fonctionne pour l'API, mais pas pour un autre client du service.
- **Répondre** `404` **pour un salarié inexistant dans le corps de la requête** : l'adresse existe, ce sont les données qui sont invalides.
- **Supprimer le test de la notification parce qu'il échoue** : le problème vient de la base partagée entre les tests, que `@Transactional` règle.