P1C5 - Exposez vos fonctionnalités via une API REST

Vérifiez votre travail

Les statuts et le service. Les statuts rejoignent des constantes, comme les types de contrat au chapitre 3 :

```
public static class LeaveStatuses
{
    public const string Pending = "Pending";
    public const string Approved = "Approved";
    public const string Refused = "Refused";
}

public enum ApprovalResult { Approved, InsufficientBalance, NotFound, NotPending }

public enum RefusalResult { Refused, NotFound, NotPending }
```

Voici les méthodes nouvelles ou modifiées de `LeaveService`. Le constructeur, `GetBalance` et `ComputeBalance` ne changent pas :

```
public IReadOnlyList<LeaveRequest> GetRequests(string? status) =>
    _db.LeaveRequests
        .Where(r => status == null || r.Status == status)
        .OrderBy(r => r.StartDate)
        .ToList();

public LeaveRequest? GetRequest(int requestId) => _db.LeaveRequests.Find(requestId);

public LeaveRequest? Create(int employeeId, DateTime startDate, DateTime endDate)
{
    if (_db.Employees.Find(employeeId) is null) return null;

    var request = new LeaveRequest
    {
        EmployeeId = employeeId,
        StartDate = startDate,
        EndDate = endDate,
        WorkingDays = CountWorkingDays(startDate, endDate),
        Status = LeaveStatuses.Pending,
    };

    _db.LeaveRequests.Add(request);
    _db.SaveChanges();
    return request;
}

public ApprovalResult Approve(int requestId)
{
    var request = _db.LeaveRequests.Find(requestId);
    if (request is null) return ApprovalResult.NotFound;
    if (request.Status != LeaveStatuses.Pending) return ApprovalResult.NotPending;

    var employee = _db.Employees.Find(request.EmployeeId)!;
```

```
    if (ComputeBalance(employee) < request.WorkingDays)
    {
        request.Status = LeaveStatuses.Refused;
        _db.SaveChanges();
        return ApprovalResult.InsufficientBalance;
    }

    request.Status = LeaveStatuses.Approved;
    employee.DaysTaken += request.WorkingDays;
    _db.SaveChanges();

    _notifier.Notify(employee, "Votre demande de congés est approuvée.");
    return ApprovalResult.Approved;
}

public RefusalResult Refuse(int requestId)
{
    var request = _db.LeaveRequests.Find(requestId);
    if (request is null) return RefusalResult.NotFound;
    if (request.Status != LeaveStatuses.Pending) return RefusalResult.NotPending;

    request.Status = LeaveStatuses.Refused;
    _db.SaveChanges();
    return RefusalResult.Refused;
}

private static int CountWorkingDays(DateTime startDate, DateTime endDate) =>
    Enumerable.Range(0, (endDate.Date - startDate.Date).Days + 1)
        .Select(offset => startDate.Date.AddDays(offset))
        .Count(day => day.DayOfWeek is not DayOfWeek.Saturday and not DayOfWeek.Sunday);
```

Dans `GetRequests`, le filtre et le tri précèdent `ToList()` : ils sont traduits en SQL, comme vous l'avez vu au chapitre 4. La vérification du statut, elle, se trouve dans le service et non dans le contrôleur : c'est une règle métier, qui s'appliquera quel que soit le client.

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 rejoint le contrôleur des salariés :

```
[ApiController]
[Route("api/employees")]
public sealed class EmployeesController : ControllerBase
{
    private readonly LeaveService _leaveService;

    public EmployeesController(LeaveService leaveService) =>
        _leaveService = leaveService;

    [HttpGet("{id:int}/balance")]
    public ActionResult<BalanceDto> GetBalance(int id)
    {
        var balance = _leaveService.GetBalance(id);
        if (balance is null) return NotFound();

        return new BalanceDto(id, balance.Value);
    }
}
```

Les demandes ont leur propre contrôleur :

```
[ApiController]
[Route("api/leave-requests")]
public sealed class LeaveRequestsController : ControllerBase
```

```
{
    private readonly LeaveService _leaveService;

    public LeaveRequestsController(LeaveService leaveService) =>
        _leaveService = leaveService;

    [HttpGet]
    public IEnumerable<LeaveRequestDto> GetAll(string? status) =>
        _leaveService.GetRequests(status).Select(LeaveRequestDto.From);

    [HttpGet("{id:int}")]
    public ActionResult<LeaveRequestDto> GetById(int id)
    {
        var request = _leaveService.GetRequest(id);
        if (request is null) return NotFound();

        return LeaveRequestDto.From(request);
    }

    [HttpPost]
    public ActionResult<LeaveRequestDto> Create(CreateLeaveRequestDto dto)
    {
        var request = _leaveService.Create(dto.EmployeeId, dto.StartDate!.Value, dto.EndDate!.Value);
        if (request is null)
        {
            ModelState.AddModelError(nameof(dto.EmployeeId), "Ce salarié n'existe pas.");
            return ValidationProblem();
        }

        return CreatedAtAction(nameof(GetById), new { id = request.Id }, LeaveRequestDto.From(request));
    }

    [HttpPost("{id:int}/approval")]
    public ActionResult<LeaveRequestDto> Approve(int id) => _leaveService.Approve(id) switch
    {
        ApprovalResult.NotFound => NotFound(),
        ApprovalResult.NotPending => AlreadyProcessed(),
        _ => LeaveRequestDto.From(_leaveService.GetRequest(id)!),
    };

    [HttpPost("{id:int}/refusal")]
    public ActionResult<LeaveRequestDto> Refuse(int id) => _leaveService.Refuse(id) switch
    {
        RefusalResult.NotFound => NotFound(),
        RefusalResult.NotPending => AlreadyProcessed(),
        _ => LeaveRequestDto.From(_leaveService.GetRequest(id)!),
    };

    private ObjectResult AlreadyProcessed() =>
        Problem(statusCode: StatusCodes.Status409Conflict, title: "Cette demande a déjà été traitée.");
}
```

Plusieurs détails méritent votre attention :

- `{id:int}` est une **contrainte de route** : une adresse comme `leave-requests/abc` ne correspond à aucune action et renvoie `404`, sans même appeler le contrôleur ;
- dans `Create`, les `!.Value` sont sûrs : si les dates manquaient, la validation automatique aurait déjà répondu `400` ;
- `CreatedAtAction` renvoie `201` avec un en-tête `Location` qui pointe vers `GetById`, par exemple `/api/leave-requests/12` : le client sait où retrouver la demande créée ;
- après une décision, `GetRequest` n'interroge pas la base une seconde fois. Comme vous l'avez vu au chapitre 4, `Find` renvoie l'entité déjà chargée par le même `DbContext`, le temps de la requête.

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.

Program.cs. Quatre lignes s'ajoutent à celles du chapitre 4 :

```
builder.Services.AddProblemDetails();
builder.Services.AddOpenApi();

var app = builder.Build();

app.UseExceptionHandler();

if (app.Environment.IsDevelopment())
{
    app.MapOpenApi();
}
```

Les tests. Le test du bug de Leïla utilise la nouvelle route et lit le DTO :

```
[Fact]
public async Task Balance_of_fixed_term_employee_counts_only_full_months()
{
    var client = _factory.CreateClient();

    var balance = await client.GetFromJsonAsync<BalanceDto>("/api/employees/4/balance");

    Assert.Equal(4.56, balance!.Balance, precision: 2);
}
```

Les deux nouveaux tests vérifient le contrat lui-même :

```
[Fact]
public async Task Approving_a_request_twice_returns_conflict()
{
    var client = _factory.CreateClient();

    await client.PostAsync("/api/leave-requests/7/approval", null);
    var response = await client.PostAsync("/api/leave-requests/7/approval", null);

    Assert.Equal(HttpStatusCode.Conflict, response.StatusCode);
}

[Fact]
public async Task Creating_a_request_ending_before_it_starts_returns_bad_request()
{
    var client = _factory.CreateClient();
    var dto = new { employeeId = 4, startDate = "2026-07-10", endDate = "2026-07-06" };

    var response = await client.PostAsJsonAsync("/api/leave-requests", dto);

    Assert.Equal(HttpStatusCode.BadRequest, response.StatusCode);
}
```

Le premier test rejoue exactement l'incident de vendredi. S'il avait existé plus tôt, Amina n'aurait pas perdu cinq jours.

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 `GET` pour une action qui modifie des données**, même « provisoirement » : c'est exactement ce qui a provoqué l'incident.

- **Recevoir une entité dans l'action de création** : le client peut alors choisir le statut et le nombre de jours de sa demande.
- **Déclarer** `[Required]` **sur un** `DateTime` **non nullable** : la règle ne détecte jamais 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.