Contents

Raport de Evaluare - Aplicație Password Vault

Paradigme de Proiectare a Aplicațiilor Web - 2025-2026

---

1. Oficiu (1.0 puncte)

Implementare Completă

Aplicația este complet funcțională și pregătită pentru evaluare:

- **Structură proiect**: Organizată conform best practices Spring Boot
- **Configurare**: Fișiere de configurare complete (`application.properties`, `logback-spring.xml`)
- **Docker**: Configurare Docker Compose pentru dezvoltare și producție
- **Dependențe**: Toate dependențele necesare definite în `pom.xml`
- **Baza de date**: Migrări Flyway pentru schema bazei de date
- **Documentație**: README.md cu instrucțiuni de instalare și utilizare

---

2. Secțiune Admin - CRUD pe 2 Entități (2.0 puncte)

Implementare Completă

Aplicația include **două secțiuni admin complete** implementate cu **MVC (Model-View-Controller)**:

2.1. Administrare Utilizatori (`/users`)

Controller: `UserAdminController.java` - **GET `/users`** - Listare toți utilizatorii cu statistici (total, activi, inactivi) - **GET `/users/{id}`** - Detalii utilizator - **GET `/users/create`** - Formular creare utilizator nou - **POST `/users/create`** - Salvare utilizator nou - **GET `/users/{id}/edit`** - Formular editare utilizator - **POST `/users/{id}/edit`** - Actualizare utilizator - **POST `/users/{id}/delete`** - Ștergere logică (soft delete)

Service: `UserAdminService.java` - Logica de business separată - Validări și transformări Entity ↔ DTO - Logging complet pentru toate operațiile

Templates Thymeleaf: - `users/index.html` - Listă cu statistici - `users/details.html` - Detalii utilizator - `users/create.html` - Formular creare - `users/edit.html` - Formular editare

Entitate: `User` cu cheie străină către `ServicePlan`

2.2. Administrare Planuri de Servicii (`/service-plans`)

Controller: `ServicePlanAdminController.java` - **GET `/service-plans`** - Listare toate planurile cu statistici - **GET `/service-plans/{id}`** - Detalii plan - **GET `/service-plans/create`** - Formular creare plan nou - **POST `/service-plans/create`** -

Salvare plan nou - **GET `/service-plans/{id}/edit`** - Formular editare plan - **POST `/service-plans/{id}/edit`** - Actualizare plan - **POST `/service-plans/{id}/delete`** - Ștergere fizică

Service: `ServicePlanAdminService.java` - Gestionare planuri și limite asociate (`PlanLimits`) - Validări de business - Invalidare cache la modificări - Logging detaliat

Templates Thymeleaf: - `service-plans/index.html` - Listă cu statistici - `service-plans/details.htm` - Detalii plan cu limite - `service-plans/create.html` - Formular creare cu toate câmpurile - `service-plans/edit.html` - Formular editare

Entitate: `ServicePlan` cu relație One-to-One cu `PlanLimits`

Caracteristici Implementate:

CRUD Complet pentru ambele entități: - **Create**: Formulare cu validare, salvare în baza de date - **Read**: Listare cu paginare logică, detalii individuale - **Update**: Formulare pre-populate, actualizare parțială - **Delete**: Ștergere logică pentru User, ștergere fizică pentru ServicePlan

Cheie Străină: - `User.servicePlanId` → `ServicePlan.id` (Many-to-One) - `PlanLimits.servicePlanId` → `ServicePlan.id` (One-to-One)

Validare: Bean Validation (`@Valid`, `@NotNull`, `@NotBlank`, etc.)

UI Modern: Bootstrap 5, design responsive, statistici în timp real

---

3. Secțiune Utilizator - Planuri și Funcționalitate Principală (1.0 puncte)

Implementare Completă

Aplicația include o **secțiune utilizator** implementată cu **API REST** care afișează planurile și controale specifice funcționalității principale:

3.1. Frontend React - VaultPage (`/vault`)

Fișier: `frontend/src/pages/VaultPage.jsx`

Funcționalități principale:

1. **Afișare Plan Curent**:
   - Nume plan (Free, Usual, Premium)
   - Contor items: `{items.length} / {planLimits.maxVaultItems} items`
   - Badge vizual pentru plan
2. **Controale Bazate pe Plan**:
   - **Buton “Add Item”**: Dezactivat când limita este atinsă
   - **Buton “Export”**: Vizibil doar dacă `planLimits.canExport === true`
   - **Buton “Import”**: Vizibil doar dacă `planLimits.canImport === true`
   - **Avertismente**: Mesaje când utilizatorul se apropie de limită

3. **Operații CRUD pe Vault Items**:
    - **Create**: Adăugare item nou (cu validare limită plan)
    - **Read**: Listare toate items, filtrare favorite
    - **Update**: Editare item existent
    - **Delete**: Ștergere item
4. **Validări Bazate pe Plan**:
    - Verificare `maxVaultItems` la creare
    - Verificare `maxPasswordLength` la generare parolă
    - Mesaje de eroare clare când limitele sunt depășite

3.2. API Endpoints Utilizate

GET `/api/user/plan` - Obține planul curent al utilizatorului cu limite

```
{
  "success": true,
  "data": {
    "name": "Premium",
    "limits": {
      "maxVaultItems": 2000,
      "canExport": true,
      "canImport": true
    }
  }
}
```

GET `/api/vault` - Listă toate vault items pentru utilizatorul autentificat

POST `/api/vault` - Creează item nou (cu validare limită în backend)

PUT `/api/vault/{id}` - Actualizează item

DELETE `/api/vault/{id}` - Șterge item

GET `/api/vault/export/download` - Export JSON (doar dacă planul permite)

3.3. Logica de Business în Backend

Service: `VaultItemService.java` - Verificare limită `maxVaultItems` la creare - Verificare limită `maxPasswordLength` la generare parolă - Validare că item-ul aparține utilizatorului - Audit logging pentru toate operațiile

Exemplu validare limită:

```
long itemCount = vaultItemRepository.findByUserId(userId).size();
if (itemCount >= limits.getMaxVaultItems()) {
    throw new BusinessException("Maximum␣vault␣items␣limit␣reached");
}
```

---

4. Utilizare ORM (1.0 puncte)

Implementare Completă

Aplicația folosește **JPA/Hibernate** ca ORM cu abordare **Code First**:

4.1. Entități JPA

Entități principale: - `User` - Utilizatori cu soft delete - `ServicePlan` - Planuri de servicii - `PlanLimits` - Limite pentru planuri - `VaultItem` - Elemente din seif - `Company` - Companii - `Employee` - Angajați - `PasswordHistory` - Istoric parole - `AuditLog` - Log-uri de audit

4.2. Relații ORM

Many-to-One:

```
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "service_plan_id", nullable = false)
private ServicePlan servicePlan;
```

One-to-One:

```
@OneToOne(mappedBy = "servicePlan", cascade = CascadeType.ALL)
private PlanLimits limits;
```

One-to-Many:

```
@OneToMany(mappedBy = "user", cascade = CascadeType.ALL)
private List<VaultItem> vaultItems;
```

4.3. Repository Pattern

Interfețe JPA Repository: - `UserRepository` - Extinde `JpaRepository<User, Long>` - `ServicePlanRepository` - Cu query-uri custom - `VaultItemRepository` - Cu metode finder - `PlanLimitsRepository` - Cu query-uri native

Query-uri Custom:

```
@Query("SELECT␣u␣FROM␣User␣u␣WHERE␣u.isDeleted␣=␣false")
List<User> findAllNotDeleted();

@Query("SELECT␣sp␣FROM␣ServicePlan␣sp␣WHERE␣sp.isActive␣=␣true")
List<ServicePlan> findActivePlans();
```

4.4. Migrări Flyway (Code First)

13 migrări pentru schema bazei de date: - `V1__create_schema.sql` - Creare schema - `V2__create_service_plans.sql` - Tabel planuri - `V3__create_plan_limits.sql` - Tabel limite - `V4__create_users.sql` - Tabel utilizatori - `V5__create_vault_items.sql` - Tabel items - `V13__add_soft_delete_to_users.sql` - Soft delete

---

5. Utilizarea Nivelului Services (1.0 puncte)

Implementare Completă

Aplicația folosește **arhitectură în straturi** cu nivelul Services bine definit:

5.1. Servicii Implementate

Servicii principale: - `UserService` - Logica de business pentru utilizatori (API) - `UserAdminService` - Logica de business pentru admin panel (MVC) - `ServicePlanService` - Gestionare planuri cu cache - `ServicePlanAdminService` - Gestionare planuri în admin - `VaultItemService` - Logica de business pentru vault items - `CompanyService` - Gestionare companii - `EmployeeService` - Gestionare angajați - `AuditLogService` - Logging și query-uri pe audit logs - `TokenService` - Generare și validare token-uri

5.2. Caracteristici Services

Separație de Responsabilități: - Controllers doar pentru HTTP handling - Services pentru logica de business - Repositories pentru acces la date

Transacții:

```
@Transactional
public VaultItemDTO createVaultItem(Long userId, VaultItemCreateDTO
   createDTO) {
    // Logica de business
}

@Transactional(readOnly = true)
public List<ServicePlanDTO> getAllServicePlans() {
    // Read-only transaction
}
```

Dependency Injection:

```
@Service
@RequiredArgsConstructor
public class VaultItemService {
    private final VaultItemRepository vaultItemRepository;
    private final UserRepository userRepository;
    private final ServicePlanService servicePlanService;
}
```

Transformări Entity ↔ DTO: - Toate serviciile transformă entități în DTO-uri pentru API - Separare clară între modelul de date și API

---

6. Implementare Logică de Business Specifică (2.0 puncte)

Implementare Completă

Aplicația include **logică de business complexă** implementată în nivelul Services și utilizată pe pagina principală:

6.1. Validare Limite Plan la Creare Vault Item

Service: `VaultItemService.createVaultItem()`

Logica implementată:

```
// țObine limitele planului utilizatorului
var planDTO =
   servicePlanService.getServicePlanWithLimits(user.getServicePlan().getId());
var limits = planDTO.getLimits();

// ăVerific ădac utilizatorul a atins limita ămaxim
long itemCount = vaultItemRepository.findByUserId(userId).size();
if (itemCount >= limits.getMaxVaultItems()) {
    throw new BusinessException(
        String.format("Maximum␣vault␣items␣limit␣(%d)␣reached.␣Please␣
           upgrade.",
        limits.getMaxVaultItems())
    );
}
```

Utilizare în frontend (`VaultPage.jsx`): - Buton “Add Item” dezactivat când `items.length >= planLimits.maxVaultItems` - Mesaj de eroare afișat utilizatorului - Validare și în backend pentru securitate

6.2. Validare Lungime Parolă Bazată pe Plan

Logica implementată:

```
// ăVerific lungimea parolei generate
if (generatedPassword.length() > limits.getMaxPasswordLength()) {
    throw new BusinessException(
        String.format("Password␣length␣exceeds␣plan␣limit␣(%d␣characters)",
        limits.getMaxPasswordLength())
    );
}
```

6.3. Controale Funcționalități Bazate pe Plan

Frontend (`VaultPage.jsx`):

```
// Export doar ădac planul permite
{planLimits?.canExport && (
    <button onClick={handleExport}>Export</button>
```

```
)}

// Import doar ădac planul permite
{planLimits?.canImport && (
    <label>Import
        <input type="file" onChange={handleImport} />
    </label>
)}
```

Backend (VaultExportController.java):

```
// ăVerific ădac planul permite export
var planDTO =
   servicePlanService.getServicePlanWithLimits(user.getServicePlan().getId());
if (!planDTO.getLimits().getCanExport()) {
    return ResponseEntity.status(HttpStatus.FORBIDDEN)
        .body("Export␣not␣allowed␣for␣your␣plan");
}
```

6.4. Soft Delete pentru Utilizatori

Logica implementată (UserService.deleteUser()):

```
// În loc de ștergere ăfizic
// userRepository.delete(user);

// ăMarcheaz ca șters logic
user.setIsDeleted(true);
userRepository.save(user);
```

Filtrare în query-uri:

```
@Query("SELECT␣u␣FROM␣User␣u␣WHERE␣u.isDeleted␣=␣false")
List<User> findAllNotDeleted();
```

6.5. Cache pentru Service Plans

Logica implementată (ServicePlanService):

```
// ăVerific cache înainte de query DB
String cacheKey = CACHE_KEY_ALL_PLANS;
List<ServicePlanDTO> cached = cacheService.getList(cacheKey,
   ServicePlanDTO.class);
if (cached != null) {
    logger.debug("Returning␣cached␣service␣plans");
    return cached;
}

// ăDac nu e în cache, query DB și ăsalveaz în cache
List<ServicePlanDTO> plans = // ... query DB
cacheService.set(cacheKey, plans, 60); // 60 minute TTL
```

Invalidare cache la modificări:

```
// La creare/update/delete plan
cacheService.removeByPattern("service_plan");
cacheService.removeByPattern("service_plans");
```

6.6. Audit Logging Automat

Logica implementată (`AuditLogService`):

```
// Logging automat pentru ț operaii importante
auditLogService.logAction(userId, "CREATE_VAULT_ITEM",
    "Created␣vault␣item:␣" + item.getTitle());
```

---

7. Complexitatea Implementării (1.0 puncte)

Implementare Completă

Aplicația include **multiple caracteristici avansate**:

7.1. Cache în Memorie

Implementare: `MemoryCacheService.java`

Caracteristici: - Cache cu TTL (Time To Live) configurable - Pattern-based invalidation - Thread-safe cu `ConcurrentHashMap` - Serializare JSON pentru obiecte complexe - Logging pentru cache hits/misses

Utilizare: - Cache pentru `ServicePlan` entities (60 minute TTL) - Invalidare automată la modificări - Reducere semnificativă a query-urilor la DB

7.2. Logging Complet (SLF4J/Logback)

Configurare: `logback-spring.xml`

Caracteristici: - Logging la nivel INFO pentru operații importante - Logging la nivel DEBUG pentru detalii - Logging la nivel ERROR pentru excepții - Hibernate SQL queries la nivel DEBUG - Logging în fișier cu rolling daily - Logging în console pentru dezvoltare

Exemplu logging:

```
logger.info("Creating␣new␣vault␣item␣for␣user␣id:␣{}␣with␣title:␣{}", userId,
   title);
logger.debug("Checking␣plan␣limits␣for␣user␣id:␣{}", userId);
logger.error("Error␣creating␣vault␣item", e);
```

7.3. Dependency Injection

Utilizare Spring DI: - Constructor injection cu `@RequiredArgsConstructor` (Lombok) - Field injection pentru configurații - Service-to-service injection - Repository injection în services

Exemplu:

```
@Service
@RequiredArgsConstructor
public class VaultItemService {
    private final VaultItemRepository vaultItemRepository;
    private final UserRepository userRepository;
    private final ServicePlanService servicePlanService;
    private final AuditLogService auditLogService;
}
```

7.4. Soft Delete

Implementare: - Câmp `isDeleted` în entitatea `User` - Query-uri custom care filtrează utilizatorii șterși - Index pe `is_deleted` pentru performanță - Migrare Flyway pentru adăugare câmp

Avantaje: - Păstrare istoric - Posibilitate de restaurare - Audit trail complet

7.5. Alte Caracteristici Avansate

Exception Handling Global: `GlobalExceptionHandler` pentru răspunsuri consistente

Bean Validation: Validare automată cu `@Valid`, `@NotNull`, etc.

DTO Pattern: Separare între entități și API responses

Transaction Management: `@Transactional` pentru consistență date

Flyway Migrations: Versionare schema bazei de date

Swagger/OpenAPI: Documentație API automată

---

8. Documentație (1.0 puncte)

Documentație Completă

Aplicația include documentație detaliată:

8.1. README.md

- Instrucțiuni de instalare
- Configurare Docker
- Listă completă API endpoints

- Structură proiect
- Tehnologii utilizate

8.2. Documentație API (Swagger)

- Accesibilă la `/swagger-ui.html`
- Documentație interactivă
- Exemple de request/response
- Schema-uri JSON

8.3. Comentarii în Cod

- JavaDoc pentru clase principale
- Comentarii pentru logica complexă
- Explicații pentru query-uri custom

8.4. Documentație Separată

- Fișier `DOCUMENTATIE.md` cu structură completă
- Explicații despre paradigme utilizate
- Arhitectură aplicație
- Pași de instalare pentru programator și beneficiar

Concluzie

Aplicația **Password Vault** îndeplinește toate cerințele din grila de evaluare:

Arhitectură solidă cu separare clară a responsabilităților
CRUD complet pe 2 entități în admin panel (MVC)
Secțiune utilizator funcțională cu controale bazate pe plan
ORM Code First cu JPA/Hibernate și Flyway
Nivel Services bine organizat cu logica de business
Logică de business complexă pentru validări și limitări plan
Caracteristici avansate: Cache, Logging, DI, Soft Delete
Documentație completă pentru instalare și utilizare

Aplicația este pregătită pentru evaluare și demonstrează o înțelegere profundă a paradigmelor de proiectare a aplicațiilor web.