Contents

Documentație Proiect - Password Vault Application

Paradigme de Proiectare a Aplicațiilor Web - 2025-2026

---

1. Proiectare

1.1. Paradigme Utilizate

Aplicația **Password Vault** folosește următoarele paradigme de proiectare:

1.1.1. Model-View-Controller (MVC)

Implementare: Spring MVC cu Thymeleaf pentru interfețe web admin

Utilizare: - **Admin Panel**: Secțiuni pentru administrarea utilizatorilor și planurilor de servicii - **Controllers MVC**: `UserAdminController`, `ServicePlanAdminController`, `CompanyController`, `EmployeeController` - **Views**: Template-uri Thymeleaf în `src/main/resources/templates/` - **Models**: Entități JPA și ViewModels (DTO-uri pentru formulare)

Exemplu structură MVC:

```
Controller (UserAdminController)↓

Service (UserAdminService) - Logica de business↓

Repository (UserRepository) - Acces la date↓

Database (PostgreSQL)
```

1.1.2. RESTful API Implementare: Spring REST Controllers pentru API-uri

Utilizare: - **API Endpoints**: `/api/users`, `/api/vault`, `/api/service-plans`, etc. - **HTTP Methods**: GET, POST, PUT, DELETE conform standardelor REST - **Response Format**: JSON consistent cu wrapper `ApiResponse<T>` - **Status Codes**: 200, 201, 400, 404, 500

Exemplu endpoint:

```
@RestController
@RequestMapping("/api/users")
public class UserController {
    @GetMapping
    public ApiResponse<List<UserDTO>> getAllUsers() {
        // ...
    }
}
```

1.1.3. ORM Code First Implementare: JPA/Hibernate cu abordare Code First

Caracteristici: - **Entități JPA**: Definite în Java cu `@Entity`, `@Table`, `@Column` - **Migrări Flyway**: Schema bazei de date generată din entități și versionată - **Relații ORM**: `@ManyToOne`, `@OneToMany`, `@OneToOne` pentru relații între entități - **Lazy Loading**: `FetchType.LAZY` pentru optimizare performanță

Exemplu entitate:

```
@Entity
@Table(name = "users", schema = "vault_schema")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

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

1.1.4. Repository Pattern Implementare: Spring Data JPA Repositories

Utilizare: - Interfețe care extind `JpaRepository<T, ID>` - Query-uri custom cu `@Query` - Finder methods (ex: `findByUsername`, `findByEmail`) - Native queries pentru operații complexe

Exemplu repository:

```
public interface UserRepository extends JpaRepository<User, Long> {
    @Query("SELECT␣u␣FROM␣User␣u␣WHERE␣u.isDeleted␣=␣false")
    List<User> findAllNotDeleted();

    Optional<User> findByUsername(String username);
}
```

1.1.5. Service Layer Pattern **Implementare**: Servicii Spring cu `@Service`

Utilizare: - Separare logica de business de controllers - Transacții declarative cu `@Transactional` - Transformări Entity ↔ DTO - Validări de business

Exemplu service:

```
@Service
@Transactional
public class VaultItemService {
    public VaultItemDTO createVaultItem(Long userId, VaultItemCreateDTO dto) {
        // Logica de business
        // ăValidri
        // ăTransformri
    }
}
```

1.1.6. Dependency Injection **Implementare**: Spring IoC Container

Utilizare: - Constructor injection (preferat) - `@RequiredArgsConstructor` (Lombok) pentru injectare automată - Service-to-service dependencies - Repository injection în services

1.2. De Ce Au Fost Alese Aceste Paradigme?

1.2.1. MVC pentru Admin Panel **Motivație**: - **Rapid Development**: Thymeleaf permite crearea rapidă a interfețelor web - **Server-Side Rendering**: Ideal pentru aplicații admin cu date sensibile - **Integrare Naturală**: Spring MVC se integrează perfect cu Spring Boot - **SEO și Securitate**: Rendering pe server oferă mai mult control

Avantaje: - Formulare cu validare automată - Template reutilizabile (layout.html) - Integrare directă cu Spring Security (dacă e nevoie)

1.2.2. RESTful API pentru Frontend **Motivație**: - **Separație Frontend/Backend**: Frontend React poate fi dezvoltat independent - **Standardizare**: REST este standardul industriei pentru API-uri - **Scalabilitate**: API-ul poate servi multiple frontend-uri (web, mobile) - **Testabilitate**: API-urile REST sunt ușor de testat

Avantaje: - JSON lightweight și ușor de procesat - Stateless requests pentru scalabilitate - Cache-friendly pentru performanță

1.2.3. ORM Code First **Motivație**: - **Versionare Schema**: Flyway permite versionarea schimbărilor în schema - **Type Safety**: Entități Java oferă type safety la compile-time - **Productivity**: JPA reduce codul boilerplate pentru acces la date - **Portabilitate**: Codul este independent de baza de date specifică

Avantaje: - Migrări automate la deploy - Rollback ușor la versiuni anterioare - Documentație schema în cod

1.2.4. Service Layer Pattern Motivație: - **Separație Responsabilități**: Logica de business separată de HTTP handling - **Reutilizare**: Serviciile pot fi folosite de multiple controllers - **Testabilitate**: Serviciile pot fi testate independent de controllers - **Transacții**: Gestionare centralizată a transacțiilor

Avantaje: - Cod mai organizat și mai ușor de întreținut - Logica de business centralizată - Validări consistente

1.3. Arhitectura Aplicației

1.3.1. Diagramă Arhitectură Generală CLIENT LAYER

React Frontend (VaultPage.jsx) - Afișare planuri - CRUD vault items - Controale bazate pe plan - Export/Import vault

↓ HTTP/REST

PRESENTATION LAYER

REST Controllers (API)	MVC Controllers (Admin)
AuthController	UserAdminController
UserController	ServicePlanAdminController
UserPlanController	CompanyController
VaultItemController	EmployeeController
VaultController	
VaultExportController	
VaultImportController	
VaultShareController	
ServicePlanController	
AuditLogController	
HealthController	
StatsController	

↓

BUSINESS LAYER

Services: - UserService / UserAdminService - VaultItemService - ServicePlanService / ServicePlanAdminService - CompanyService / EmployeeService - AuditLogService - TokenService

Business Logic: - Validare limite plan - Soft delete - Cache management - Audit logging - Token generation/validation

↓

DATA ACCESS LAYER

Repositories (JPA): - UserRepository - VaultItemRepository - ServicePlanRepository - PlanLimitsRepository - CompanyRepository - EmployeeRepository - AuditLogRepository - PasswordHistoryRepository - SharedVaultItemRepository

Cache Service: - MemoryCacheService (ConcurrentHashMap)

↓

PERSISTENCE LAYER

PostgreSQL Database: - Schema: vault_schema - Migrări Flyway (13 versiuni) - Tabele: users, service_plans, plan_limits, vault_items, companies, employees, audit_logs, etc.

1.3.2. Flux de Date - Exemplu: Creare Vault Item

1. Frontend (React - VaultPage.jsx) - POST /api/vault - Headers: Authorization: Bearer - Body: { title, username, password, url, notes }

↓

2. REST Controller (VaultItemController) - Validare @Valid pe VaultItemCreateDTO - Extract userId din token (TokenService) - Verificare autentificare

↓

3. Service Layer (VaultItemService) - Verificare user există (UserRepository.findById) - Obține plan limits (ServicePlanService.getServicePlanWithLimits) - Validare maxVaultItems limit - Dacă limită atinsă: throw BusinessException - Validare maxPasswordLength (dacă e parolă generată) - Criptare parolă (dacă e necesar) - Salvare în DB (VaultItemRepository.save) - Audit log (AuditLogService.logAction)

↓

4. Repository (VaultItemRepository) - JPA Save operation - Hibernate persistence

↓

5. Database (PostgreSQL) - INSERT INTO vault_schema.vault_items - (user_id, title, username, password_hash, url, notes, ...)

↓

6. Response Chain - VaultItem entity → VaultItemDTO (transformare) - ApiResponse wrapper - JSON serialization

↓

7. Frontend (React) - Receive ApiResponse - Update items state - Refresh UI (item list) - Update item counter - Clear form

1.3.3. Module și Interacțiuni

Module principale:

1. **User Management Module**
   - Controllers: `UserController` (API) + `UserAdminController` (MVC)
   - Services: `UserService` + `UserAdminService`
   - Repository: `UserRepository`
   - Funcționalități: CRUD, soft delete, login tracking, plan management
2. **Vault Management Module**
   - Controllers: `VaultItemController`, `VaultController`, `VaultExportController`, `VaultImportController`, `VaultShareController` (API)

- Service: `VaultItemService`
- Repository: `VaultItemRepository`, `PasswordHistoryRepository`, `SharedVaultItemRepository`
- Funcționalități: CRUD items, validare limite, export/import, sharing, password history

3. **Service Plan Module**
   - Controllers: `ServicePlanController`, `UserPlanController` (API) + `ServicePlanAdminControll` (MVC)
   - Services: `ServicePlanService` + `ServicePlanAdminService`
   - Repositories: `ServicePlanRepository` + `PlanLimitsRepository`
   - Funcționalități: CRUD plans, cache, limite, plan upgrade/downgrade
4. **Authentication & Authorization Module**
   - Controllers: `AuthController` (API)
   - Service: `TokenService`
   - Funcționalități: Login, token generation/validation, session management
5. **Company/Employee Module**
   - Controllers: `CompanyController` + `EmployeeController` (MVC)
   - Services: `CompanyService` + `EmployeeService`
   - Repositories: `CompanyRepository` + `EmployeeRepository`
   - Funcționalități: CRUD companii și angajați
6. **Audit & Logging Module**
   - Controller: `AuditLogController` (API)
   - Service: `AuditLogService`
   - Repository: `AuditLogRepository`
   - Funcționalități: Logging automat, query-uri pe log-uri, filtrare
7. **System & Monitoring Module**
   - Controllers: `HealthController`, `StatsController` (API)
   - Funcționalități: Health checks, statistici aplicație
8. **Cache Module**
   - Service: `MemoryCacheService` (implementare `CacheService`)
   - Interface: `CacheService`
   - Funcționalități: In-memory caching cu TTL, pattern-based invalidation

---

2. Implementare

2.1. Business Layer – Explicat

Nivelul Business Layer (Services) este **nucleul aplicației** și conține toată logica de business specifică aplicației.

2.1.1. Structură Business Layer Servicii principale:

1. **UserService / UserAdminService**
   - **Responsabilități**:
     - Gestionare utilizatori (CRUD)
     - Validare username/email unic
     - Soft delete (marcare `isDeleted = true`)
     - Tracking login (increment `loginCount`, update `lastLoginAt`)

  - Transformare Entity ↔ DTO
- **Logica de business**:

```
// Soft delete - nu șterge fizic
user.setIsDeleted(true);
userRepository.save(user);

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

2. **VaultItemService**
   - **Responsabilități**:
     - CRUD vault items
     - Validare limite plan (maxVaultItems, maxPasswordLength)
     - Criptare/decriptare parole
     - Gestionare favorite
     - Audit logging automat
   - **Logica de business**:

```
// Validare ălimit la creare
var planDTO =
   servicePlanService.getServicePlanWithLimits(user.getServicePlan().getId());
long itemCount = vaultItemRepository.findByUserId(userId).size();
if (itemCount >= limits.getMaxVaultItems()) {
    throw new BusinessException("Maximum␣limit␣reached");
}

// Audit logging
auditLogService.logAction(userId, "CREATE_VAULT_ITEM", "Created:␣" +
   item.getTitle());
```

3. **ServicePlanService / ServicePlanAdminService**
   - **Responsabilități**:
     - CRUD service plans
     - Gestionare limite (PlanLimits)
     - Cache management (get/set/invalidate)
     - Filtrare planuri active
   - **Logica de business**:

```
// Cache check
List<ServicePlanDTO> cached =
   cacheService.getList(CACHE_KEY_ALL_PLANS, ServicePlanDTO.class);
if (cached != null) return cached;

// Query DB și cache
List<ServicePlanDTO> plans = // ... query
cacheService.set(CACHE_KEY_ALL_PLANS, plans, 60);

// Invalidare cache la ămodificri
cacheService.removeByPattern("service_plan");
```

4. **AuditLogService**

- **Responsabilități**:
  - Logging automat pentru acțiuni importante
  - Query-uri pe audit logs (by user, action, date range)
  - Filtrare și paginare
- **Logica de business**:

```
public void logAction(Long userId, String action, String details) {
    AuditLog log = new AuditLog();
    log.setUserId(userId);
    log.setAction(action);
    log.setDetails(details);
    log.setTimestamp(LocalDateTime.now());
    auditLogRepository.save(log);
}
```

2.1.2. Pattern-uri Utilizate în Business Layer **1. Service Layer Pattern** - Separare clară între controllers și business logic - Servicii reutilizabile de către multiple controllers

2. DTO Pattern - Transformare Entity ↔ DTO pentru API responses - Separare între modelul de date și API contract

3. Repository Pattern - Abstractizare acces la date - Query-uri custom pentru operații complexe

4. Transaction Management - `@Transactional` pentru consistență date - `@Transactional(readOnly = true)` pentru query-uri

5. Exception Handling - Excepții custom: `ResourceNotFoundException`, `BusinessException`, `ValidationException` - Propagare excepții către GlobalExceptionHandler

2.2. Librării Suplimentare Utilizate

2.2.1. Spring Boot Ecosystem **spring-boot-starter-web** - REST API support - Embedded Tomcat server - JSON serialization/deserialization

spring-boot-starter-data-jpa - JPA/Hibernate ORM - Spring Data JPA repositories - Transaction management

spring-boot-starter-thymeleaf - Template engine pentru MVC - Server-side rendering - Integration cu Spring MVC

spring-boot-starter-validation - Bean Validation (JSR-303) - `@Valid`, `@NotNull`, `@NotBlank`, etc. - Validare automată în controllers

2.2.2. Database & Migration **PostgreSQL Driver** - Driver JDBC pentru PostgreSQL - Connection pooling cu HikariCP

Flyway Core - Versionare schema bazei de date - Migrări automate la startup - Rollback support

2.2.3. Documentation & API

springdoc-openapi - Swagger UI pentru documentație API - OpenAPI 3.0 specification - Interactive API testing

2.2.4. Utilities

Lombok - `@Data`, `@Getter`, `@Setter` - reduce boilerplate - `@RequiredArgsConstructor` - constructor injection - `@Slf4j` - logging

SLF4J / Logback - Logging framework - Configurare via `logback-spring.xml` - File rolling, console output

2.3. Secțiuni de Cod sau Abordări Deosebite

2.3.1. Cache Implementation cu TTL

Fișier: `MemoryCacheService.java`

Abordare deosebită:

```
private static class CacheEntry {
    private final Object value;
    private final LocalDateTime expirationTime;

    public boolean isExpired() {
        return LocalDateTime.now().isAfter(expirationTime);
    }
}

// Pattern-based invalidation
public void removeByPattern(String pattern) {
    cache.keySet().removeIf(key -> key.startsWith(pattern));
}
```

Avantaje: - Thread-safe cu `ConcurrentHashMap` - TTL automat pentru expirare - Pattern-based invalidation pentru grupuri de cache

2.3.2. Soft Delete Implementation

Abordare deosebită:

```
// În loc de ștergere ăfizic
@Column(nullable = false, name = "is_deleted")
private Boolean isDeleted = false;

// Query-uri custom care ăfiltreaz automat
@Query("SELECT␣u␣FROM␣User␣u␣WHERE␣u.isDeleted␣=␣false")
List<User> findAllNotDeleted();

// Index pentru țăperforman
@Index(name = "idx_users_is_deleted", columnList = "is_deleted")
```

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

2.3.3. Validare Limite Plan în Multiple Straturi

Abordare deosebită - Validare atât în frontend cât și backend:

Frontend (UX):

```
// Buton dezactivat când limita e ăatins
disabled={planLimits && items.length >= planLimits.maxVaultItems}

// Mesaj de eroare clar
if (items.length >= planLimits.maxVaultItems) {
    setError(`Maximum limit (${planLimits.maxVaultItems}) reached`);
}
```

Backend (Securitate):

```
// Validare în service layer
long itemCount = vaultItemRepository.findByUserId(userId).size();
if (itemCount >= limits.getMaxVaultItems()) {
    throw new BusinessException("Maximum␣limit␣reached");
}
```

Avantaje: - UX bun (feedback imediat) - Securitate (validare în backend) - Consistență date

2.3.4. Dual Controller Pattern (API + MVC) Abordare deosebită - Același serviciu folosit de 2 tipuri de controllers:

```
// API Controller
@RestController
@RequestMapping("/api/users")
public class UserController {
    private final UserService userService; // API service
}

// MVC Controller
@Controller
@RequestMapping("/users")
public class UserAdminController {
    private final UserAdminService userAdminService; // MVC service
}
```

Avantaje: - Separare clară API vs Web UI - Servicii specializate pentru fiecare caz - Flexibilitate în viitor

2.3.5. Audit Logging Automat Abordare deosebită - Logging automat pentru toate operațiile importante:

```
@Service
public class VaultItemService {
    private final AuditLogService auditLogService;

    public VaultItemDTO createVaultItem(...) {
        // ... create logic
        auditLogService.logAction(userId, "CREATE_VAULT_ITEM", details);
    }
```

```
    public VaultItemDTO updateVaultItem(...) {
        // ... update logic
        auditLogService.logAction(userId, "UPDATE_VAULT_ITEM", details);
    }
}
```

Avantaje: - Audit trail complet - Compliance (GDPR, etc.) - Debugging și troubleshooting

---

3. Utilizare

3.1. Pașii de Instalare

3.1.1. Instalare și Configurare pentru Programator Cerințe preliminare: - Java 21 sau mai nou - Maven 3.6+ - Docker și Docker Compose - Git

Pași de instalare:

1. **Clonează repository-ul**:

```
git clone <repository-url>
cd PPAW
```

2. **Configurează variabile de mediu** (opțional):

```
# ăCreeaz șfiier .env (țopional, valorile default țăfuncioneaz)
export
    SPRING_DATASOURCE_URL=jdbc:postgresql://localhost:5432/password_vault
export SPRING_DATASOURCE_USERNAME=postgres
export SPRING_DATASOURCE_PASSWORD=postgres
export SERVER_PORT=8080
```

3. **Pornește aplicația cu Docker Compose**:

```
docker compose up
```

   Această comandă va:

   - Porni PostgreSQL pe portul 5432
   - Porni aplicația Spring Boot pe portul 8080
   - Rula migrările Flyway automat
   - Popula baza de date cu date de test (dacă există)

4. **Verifică că aplicația rulează**:

   - Aplicație: http://localhost:8080
   - Swagger UI: http://localhost:8080/swagger-ui.html
   - Health Check: http://localhost:8080/api/health

5. **Pentru dezvoltare locală** (fără Docker):

```
# șPornete PostgreSQL local
# ăAjusteaz application.properties cu conexiunea la DB

# ăRuleaz țaplicaia
mvn spring-boot:run
```

Configurare IDE (IntelliJ IDEA / Eclipse): - Importă proiectul ca Maven project - Configurează Java 21 SDK - Ajustează run configuration pentru Spring Boot

Debugging: - Port debug: 5005 (configurat în docker-compose.yml) - Conectează debugger la `localhost:5005`

3.1.2. Instalare și Configurare la Beneficiar Cerințe preliminare: - Server cu Docker și Docker Compose instalat - Acces la portul 8080 (sau alt port configurat) - PostgreSQL (poate fi pe același server sau extern)

Pași de instalare:

1. **Transferă fișierele aplicației pe server**:

```
# ăCopiaz întregul proiect pe server
scp -r PPAW/ user@server:/opt/password-vault/
```

2. **Configurează variabile de mediu pe server**:

```
cd /opt/password-vault

# ăEditeaz docker-compose.yml sau ăcreeaz .env
# ăAjusteaz:
# - Portul țaplicaiei (ădac e nevoie)
# - țCredeniale baza de date
# - Volume paths pentru logs
```

3. **Configurează baza de date**:

```
# ăDac PostgreSQL e pe alt server:
# ăAjusteaz SPRING_DATASOURCE_URL în docker-compose.yml

# ăDac PostgreSQL e local:
# Docker Compose va crea containerul automat
```

4. **Pornește aplicația**:

```
docker compose up -d  # -d pentru detached mode
```

5. **Verifică logs**:

```
docker compose logs -f password-vault
```

6. **Verifică că aplicația rulează**:

```
curl http://localhost:8080/api/health
# Ar trebui ăs returneze: {"status":"UP",...}
```

7. **Configurare reverse proxy** (opțional, pentru producție):

```
# Nginx configuration
server {
    listen 80;
    server_name password-vault.example.com;

    location / {
        proxy_pass http://localhost:8080;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }
}
```

Backup baza de date:

```
# Backup zilnic (cron job)
pg_dump -h localhost -U postgres password_vault > backup_$(date +%Y%m%d).sql
```

Update aplicație:

```
# șOprete țaplicaia
docker compose down

# ăActualizeaz codul
git pull  # sau ăcopiaz noile șfiiere

# șReconstruiete și șrepornete
docker compose up -d --build
```

3.2. Mod de Utilizare

3.2.1. Utilizare Admin Panel (MVC) **Acces**: http://localhost:8080/users

Funcționalități:

1. **Administrare Utilizatori** (`/users`):
   - **Listare**: Vezi toți utilizatorii cu statistici (total, activi, inactivi)
   - **Creare**: Click "Adaugă Utilizator Nou" → Completează formularul → Salvează
   - **Editare**: Click pe utilizator → "Editează Utilizator" → Modifică → Salvează
   - **Ștergere**: Click "Șterge Utilizator" → Confirmă (soft delete)
   - **Detalii**: Click pe utilizator pentru a vedea detalii complete
2. **Administrare Planuri de Servicii** (`/service-plans`):
   - **Listare**: Vezi toate planurile cu statistici
   - **Creare**: Click "Adaugă Plan Nou" → Completează:
     - Nume plan, preț, monedă
     - Limite: maxVaultItems, maxPasswordLength, etc.

        - Features: canExport, canImport, canShare, etc.
    - **Editare**: Modifică plan existent și limitele sale
    - **Ștergere**: Șterge plan (fizic, cu confirmare)
3. **Administrare Companii** (`/companies`):
    - CRUD complet pentru companii
4. **Administrare Angajați** (`/employees`):
    - CRUD complet pentru angajați

3.2.2. Utilizare API (REST)

Base URL: http://localhost:8080/api

Autentificare:

```
POST /api/auth/login
Content-Type: application/json

{
  "username": "user1",
  "password": "password123"
}

Response:
{
  "success": true,
  "data": {
    "token": "abc123...",
    "user": { ... }
  }
}
```

Operații Vault Items:

```
# ă List toate items
GET /api/vault
Authorization: Bearer <token>

# ă Creeaz item nou
POST /api/vault
Authorization: Bearer <token>
Content-Type: application/json

{
  "title": "Gmail",
  "username": "user@gmail.com",
  "password": "secure123",
  "url": "https://gmail.com"
}

# ă Actualizeaz item
PUT /api/vault/1
Authorization: Bearer <token>
Content-Type: application/json

{
```

```
  "title": "Gmail␣Updated",
  "password": "newpassword123"
}

# Șterge item
DELETE /api/vault/1
Authorization: Bearer <token>
```

Operații Service Plans:

```
# ăList toate planurile
GET /api/service-plans

# ăList doar planurile active
GET /api/service-plans?active=true

# țObine plan cu limite
GET /api/service-plans/1/with-limits
```

Export Vault:

```
GET /api/vault/export/download
Authorization: Bearer <token>

# ăReturneaz JSON file cu toate items
```

3.2.3. Utilizare Frontend React

Acces: http://localhost:3000 (dacă frontend-ul e pornit separat)

Funcționalități principale:

1. **Login**:
   - Introdu username și parolă
   - La autentificare reușită, se încarcă planul utilizatorului
2. **Vault Page** (`/vault`):
   - **Afișare plan**: Badge cu numele planului și contor items
   - **Add Item**: Buton pentru adăugare item nou
     - Dezactivat când limita e atinsă
     - Formular cu: title, username, password, url, notes
   - **Edit Item**: Click pe item → Modifică → Salvează
   - **Delete Item**: Click “Delete” → Confirmă
   - **Export**: Buton vizibil doar dacă planul permite
     - Descarcă JSON cu toate items
   - **Import**: Buton vizibil doar dacă planul permite
     - Încarcă fișier JSON
   - **Favorites**: Marchează items ca favorite
3. **Validări Vizuale**:
   - Avertisment când utilizatorul se apropie de limită (80% din maxVaultItems)
   - Eroare când limita e atinsă
   - Mesaje clare pentru fiecare acțiune

3.2.4. Documentație API (Swagger) Acces: http://localhost:8080/swagger-ui.html

Funcționalități: - Vezi toate endpoint-urile disponibile - Testează API-urile direct din browser - Vezi schema-uri JSON pentru request/response - Exemple de request-uri

3.2.5. Logs și Monitoring Logs aplicație:

```
# Logs în Docker
docker compose logs -f password-vault

# Logs în șfiier (ădac configurat)
tail -f logs/application.log
```

Health Check:

```
curl http://localhost:8080/api/health
```

Statistici:

```
curl http://localhost:8080/api/stats
```

Concluzie

Aplicația **Password Vault** este o aplicație web completă care demonstrează utilizarea corectă a paradigmelor de proiectare moderne:

Arhitectură solidă cu separare clară a responsabilităților
Multiple paradigme (MVC, REST API, ORM Code First)
Business logic complexă pentru validări și limitări
Caracteristici avansate (Cache, Logging, Soft Delete)
Documentație completă pentru instalare și utilizare

Aplicația este pregătită pentru utilizare în producție și poate fi extinsă ușor cu funcționalități noi.