



# DOCUMENTATIA PROIECTULUI

Sipos Bogdan

## 1. Introducere

În ultimii ani, arhitectura bazată pe microservicii a devenit un standard de facto în dezvoltarea aplicațiilor moderne, oferind scalabilitate, flexibilitate și o separare clară a responsabilităților. În loc să construim un monolit greu de întreținut, am decompozat sistemul în trei servicii autonome – **user-service**, **animal-service** și **exemplar-service** – fiecare responsabil pentru un set definit de operațiuni. Fiecare microserviciu are propria bază de date, propriul ciclu de dezvoltare, testare și rulare, și comunică cu celelalte prin API-uri REST bine documentate.

Proiectul implementat respectă principiile Domain-Driven Design (DDD), structurând fiecare serviciu pe layere specifice: **config** (securitate JWT, CORS, gateway intern), **controller** (expune endpoint-uri), **service** (logica de business), **domain** (entități JPA), **dto** (transfer de date) și **repository** (acces la date). În plus, între serviciile **exemplar** și **animal** există un mecanism de integrare printr-un client REST (AnimalApiService), care aplică principiul Proxy Pattern pentru a abstractiza apelurile HTTP.

Aplicația acoperă fluxurile tipice ale unui sistem de management al speciilor și al exemplarelor: crearea și autentificarea utilizatorilor, operații CRUD pentru specii și exemplare, filtrări și statistici agregate, precum și exportul datelor în formate multiple (CSV, JSON, XML, DOCX) printr-o implementare Strategy Pattern. Oferim astfel un cadru modular, testabil și extensibil, pregătit pentru containerizare (Docker, Kubernetes) și scalare independentă a fiecărui microserviciu. În continuare, această documentație prezintă analiza cerințelor, design-ul arhitectural, pattern-urile alese și diagramele UML aferente (clase, componente, secvență și ERD), împreună cu justificările tehnice pentru fiecare decizie.

Arhitectura propusă pentru acest proiect este distribuită, bazată pe paradigma microserviciilor, ceea ce presupune împărțirea funcționalității într-un set de servicii independente, fiecare responsabil pentru un subdomeniu clar al aplicației, conform principiilor Domain-Driven Design (DDD). Fiecare microserviciu are propria bază de date relațională (MySQL) și este dezvoltat, testat și implementat separat. Comunicația se realizează prin API-uri REST bine definite, iar validările între servicii (de exemplu, verificarea existenței unei specii înainte de a crea un exemplar) se fac printr-un client proxy intern (AnimalApiService).

## 2. Arhitectura sistemului

În această implementare există trei microservicii principale:

## 1. user-service

- **Rol:** Gestionarea utilizatorilor (înregistrare, autentificare JWT, CRUD, schimbare parolă, notificări prin email/SMS).
- **Stack:** Spring Security, JWT, UserDetailsServiceImpl, NotificationService
- **Date:** tabel users (id, username, email, password, user\_type)

## 2. animal-service

- **Rol:** Gestionarea speciilor de animale (CRUD, filtrare după categorie, dietType, habitat, statistici agregate, export în CSV/JSON/XML/DOCX).
- **Stack:** Spring Data JPA, AnimalExporterService, JFreeChart pentru statistici
- **Date:** tabel animals (id, name, category, diet\_type, habitat, average\_weight, average\_age)

## 3. exemplar-service

- **Rol:** Gestionarea exemplarelor (CRUD, mapare ExemplarDTO ↔ Exemplar, apel intern către animal-service pentru detalii specie, export exemplare).
- **Stack:** PlantUML, ExemplarFacade, AnimalApiService ca proxy REST
- **Date:** tabel exemplars (id, animal\_id, name, location, age, weight, notes)

Toate cererile HTTP dinspre client (React SPA) sunt centralizate printr-un **API Gateway** (NGINX sau Spring Cloud Gateway), care se ocupă de:

- Rutare către serviciul corespunzător (/api/auth/\*\*, /api/animal/\*\*, /api/exemplar/\*\*)
- Validarea și extragerea token-ului JWT
- Configurarea CORS la un singur punct
- Agregarea eventuală a răspunsurilor

### Avantaje ale acestei arhitecturi

- **Scalabilitate:** fiecare serviciu poate fi replicat independent în funcție de trafic.

- **Reziliență:** degradarea sau căderea unui serviciu nu compromite întregul sistem.
- **Cicluri de viață independente:** echipe diferite pot lansa versiuni noi fără a afecta celelalte servicii.
- **Modularitate:** codul este organizat pe layere (config, controller, service, domain, dto, repository), ușurând testarea și întreținerea.

### 3. Analiza cerințelor

În acest proiect, cerințele reflectă funcționalitățile unui sistem de gestiune a utilizatorilor, speciilor de animale și exemplarelor acestora, implementat pe microservicii.

#### 3.1 Cerințe funcționale

Cerințele sunt grupate pe cele trei microservicii:

##### user-service

- **Înregistrare:** utilizatorii își pot crea conturi noi (username, email, parolă, tip utilizator).
- **Autentificare:** login pe bază de JWT (email/parolă).
- **CRUD utilizatori:** administratorii și angajații pot crea, modifica, șterge și vizualiza utilizatori.
- **Filtrare și căutare:** listare paginată a utilizatorilor, filtrare după tipuri (CLIENT, EMPLOYEE, MANAGER, ADMIN).
- **Notificări:** trimitere de email la schimbarea parolei sau a datelor de profil.

##### animal-service

- **CRUD specii:** adăugare, modificare, ștergere și vizualizare speciilor de animale (name, category, dietType, habitat, weight, age).
- **Listare și filtrare:** sortare și filtrare după specie, dietType, habitat.
- **Statistici agregate:** calcul al mediei vârstei și greutatei pe categorii.
- **Export date:** descărcare listă specii în formate CSV, JSON, XML și DOCX.

## exemplar-service

- **CRUD exemplare:** gestionare exemplare concrete (animalId, name, location, age, weight, notes).
- **Îmbogățire date:** pentru fiecare exemplar, se apelează animal-service ca proxy REST ca să completeze detaliile speciei (AnimalApiService).
- **Listare și căutare:** paginare, filtrare după animal, locație, interval vârstă.
- **Export exemplare:** descărcare listă exemplare în același set de formate (CSV, JSON, XML, DOCX).

## 3.2 Cerințe non-funcționale

- **Scalabilitate:** fiecare microserviciu poate fi scalat independent orizontal (container Docker, Kubernetes).
- **Izolare și reziliență:** defectele într-un serviciu nu afectează celelalte; baze de date separate.
- **Securitate:** autentificare și autorizare bazate pe roluri (JWT + JwtAuthenticationFilter + SecurityConfig).
- **Modularitate și întreținere:** cod organizat pe layere (config, controller, service, domain, dto, repository) și ușor de testat (unit/integration).
- **Portabilitate:** rulare în containere Docker, configurare simplă de deployment cu Docker Compose sau Kubernetes.
- **Performanță:** folosirea cache-ului la nivel de serviciu acolo unde e necesar (ex. caching AnimalDTO).
- **Testabilitate:** fiecare serviciu oferă suite de teste unitare (JUnit, Mockito) și teste de integrare end-to-end (Postman/Newman).

## 3.3 Cazuri de utilizare (rezumat)

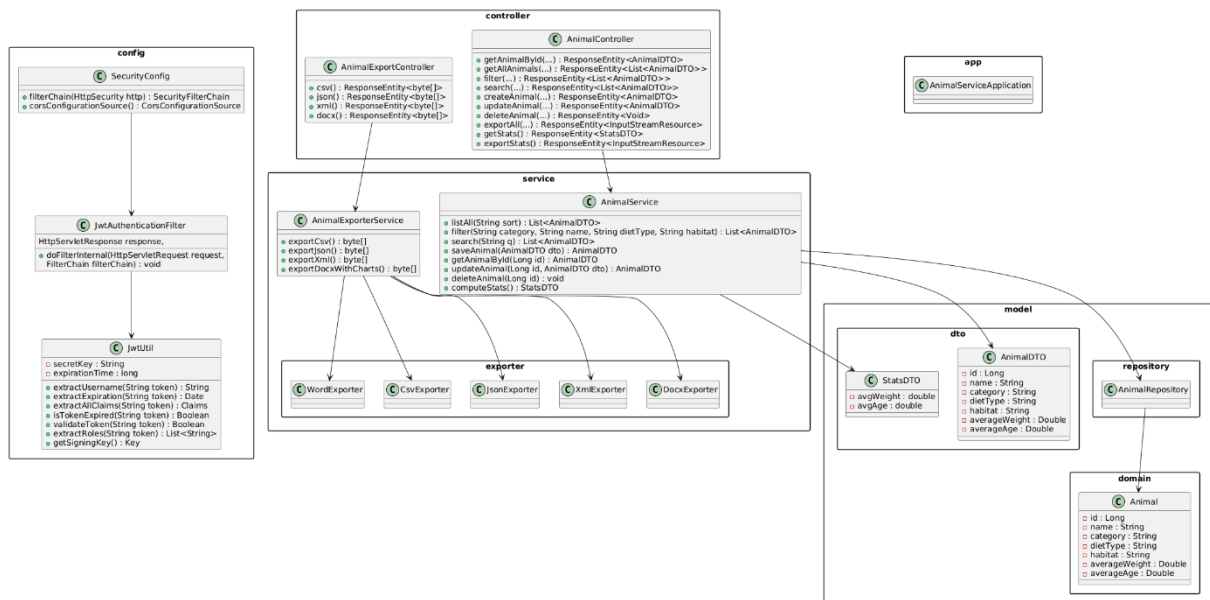
### 4.1 Diagrame de clase

Diagramele de clase oferă o vedere structurală a fiecărui microserviciu, evidențiind clasele principale, attributele și metodele lor, precum și relațiile dintre componente. În acest proiect, fiecare serviciu respectă principiile Domain-Driven Design (DDD), cu layere clare: **config**, **controller**, **service**, **domain**, **dto**, **repository** (sau **infrastructure**), plus eventuale **facade** sau **adapter** după caz.

- User (entitate JPA).
- UserType (enum).

- UserDTO, PasswordUpdateRequest, AuthRequest, AuthResponse – obiecte DTO.
- **Layer Repository / Infrastructure**
  - UserRepository – extinde JpaRepository<User, Long>.
- **Pattern-uri folosite:**
  - **Singleton** (bean-uri Spring),
  - **Filter** (JwtAuthenticationFilter),
  - **Builder**

#### 4.1.2 Diagrama de clase – animal-service



Microserviciul **animal-service** gestionează entitățile de tip specie:

- **Layer Config**
  - SecurityConfig, JwtUtil, JwtAuthenticationFilter – ca la user-service.
- **Layer Controller**
  - AnimalController – endpoint-uri /api/animal/\*\* pentru CRUD, filtrare, statistici și export.
  - AnimalExportController – sub-rute /api/animal/export pentru CSV, JSON, XML, DOCX.
- **Layer Service**





- ExemplarController – endpoint-uri /api/exemplar/\*\* pentru CRUD exemplare.
- **Layer Facade & Service**
  - ExemplarFacade – unifică apelurile la ExemplarService și AnimalApiService.
  - ExemplarService – logica CRUD, mapări Entitate↔DTO, apel intern la AnimalApiService.
  - AnimalApiService – client REST proxy către animal-service.
- **Layer Domain & DTO**
  - Exemplar (entitate JPA).
  - ExemplarDTO, ExemplarPhotoDTO – DTO-uri pentru transferul și încărcarea imaginilor.
- **Layer Repository**
  - ExemplarRepository – extinde JpaRepository<Exemplar, Long>.
- **Pattern-uri folosite:**
  - **Facade** (ExemplarFacade),
  - **DTO**,
  - **Filter** (JWT),
  - **Builder**
  - **Proxy**

## 4.2 Design Pattern-uri utilizate

Utilizarea sabloanelor de proiectare (design pattern-uri) în cadrul arhitecturii pe microservicii aduce beneficii majore în ceea ce privește claritatea, reutilizarea codului, testabilitatea și scalabilitatea sistemului. Conform cerințelor, în cadrul acestui proiect au fost implementate cinci design pattern-uri reprezentative:

### 1. Singleton

**Categorie:** Creational

**Locație:** Toate bean-urile Spring (@Service, @Component)

**Descriere:**

Spring gestionează fiecare bean cu scope singleton în mod implicit – de exemplu AnimalService, UserService sau NotificationService sunt create o singură dată

în context.

**Avantaje:**

- Minimizează costul instanțierii
- Permite păstrarea stării (cache, configurări)
- Simplifică testarea și mocking-ul

## 2. Facade

**Categorie:** Structural

**Locație:** **exemplar-service**, clasa ExemplarFacade

**Descriere:**

ExemplarFacade oferă o interfață unificată pentru controller, ascunzând complexitatea apelurilor către ExemplarService și AnimalApiService.

**Avantaje:**

- Scade cuplarea dintre controller și serviciile de business
- Centralizează validările comune și logging-ul
- Facilitează mock-uirea în teste

## 3. Builder

**Categorie:** Creational

**Locație:** DTO-uri din toate serviciile, de ex. AnimalDTO, ExemplarDTO (adnotate cu @Builder Lombok)

**Descriere:**

Pattern-ul Builder permite crearea fluentă și sigură a obiectelor complexe. În AnimalDTO.builder()...build() se setează doar câmpurile necesare, evitând constructori cu parametri multipli și setteri expuși.

**Avantaje:**

- Cod mai curat și lizibil
- Obiecte configurabile imutabil, fără setter-i expuși
- Ușurează validările la creare

## 4. DTO (Data Transfer Object)

**Categorie:** Structural

**Locație:** Toate serviciile, clase \*DTO (UserDTO, AnimalDTO, ExemplarDTO, StatsDTO etc.)

**Descriere:**

DTO-urile transferă date între straturile aplicației și între microservicii, izolând modelul intern (entitățile JPA) de structura expusă prin API.

**Avantaje:**

- Ascund detaliile de persistență (lazy loading, relații)
- Permite formatarea și validarea datelor înainte de expunere
- Facilitează versionarea și evoluția API

## 5. Proxy

**Categorie:** Structural

**Locație:** **exemplar-service**, clasa AnimalApiService

**Descriere:**

AnimalApiService funcționează ca un Remote Proxy pentru animal-service, ascunzând logica HTTP (RestTemplate) și expunând o metodă simplă `fetchById(id)`. Poate fi extins ulterior cu caching sau circuit breaker.

**Avantaje:**

- Izolează consumatorii de detaliile rețelei
- Permite adăugarea facilă de funcționalități non-funcționale (cache, retry)
- Simplifică testarea prin mock-uire

## 5. Implementare

Faza de implementare a transpus arhitectura proiectată într-un sistem software complet funcțional, organizat în trei microservicii independente și un front-end React. Fiecare microserviciu este construit pe Java și Spring Boot, cu o separare clară a layere-lor **config**, **controller**, **service**, **domain**, **dto** și **repository**.

Componentă	Tehnologie / Instrument
------------	-------------------------

Backend	Java 21, Spring Boot 3.x
---------	--------------------------

<b>Componentă</b>	<b>Tehnologie / Instrument</b>
<b>Frontend</b>	React, React Router, react-i18next
<b>Baze de date</b>	MySQL (o instanță separată per microserviciu)
<b>Testare API</b>	Postman / Newman
<b>Testare unitară</b>	JUnit, Mockito
<b>Diagrame UML</b>	PlantUML

## 5.2 Justificarea limbajului Java

- **Ecosistem matur:** sute de biblioteci și framework-uri enterprise.
- **Spring Boot:** configurație rapidă pentru REST, JPA, validare și securitate.
- **Design patterns:** sintaxă clară pentru Singleton, Facade, Builder, DTO, Proxy.
- **Performanță și stabilitate:** garbage-collector optimizat, JVM bine cunoscut.

## 5.3 Justificarea Spring Boot

- **Auto-configurație:** pornire rapidă, minim de boilerplate.
- **Modularitate:** fiecare microserviciu poate include doar dependențele necesare.
- **Spring Security:** integrare nativă pentru JWT, filtre și autorizare pe roluri.
- **Spring Cloud (opțional):** extensie ușoară pentru API Gateway, service discovery, circuit breaker.

## 5.4 Comunicare între microservicii

Modelul de comunicare este **peer-to-peer (choreography)** între aceste servicii:

1. **Frontend** → **API Gateway** → unul dintre cele trei servicii, după rută.
2. **exemplar-service** → **animal-service** (prin `AnimalApiService.fetchById(id)`).
3. **user-service** nu apelează niciun alt serviciu intern.

Nu există un orchestrator central: fiecare serviciu își asumă responsabilitatea propriului flux de lucru și, atunci când are nevoie de date din altă parte, apelează direct serviciul corespunzător.

## 5.5 Considerații privind Frontend-ul

- **React SPA** consumă doar API-ul expus de API Gateway.
- **Internaționalizare:** react-i18next suportă ușor traduceri RO/EN/FR.
- **Izolare:** prezentarea (componenta UI) este complet separată de logica de business (backend).
- **Extensibilitate:** adăugarea unui nou microserviciu nu impune modificări în codul frontend, ci doar un nou endpoint în API Gateway.

## 5.6 Etape de Dezvoltare

Următoarele etape au fost parcurse pentru a transforma specificația și design-ul în aplicație funcțională:

### 1. Inițializare proiecte Spring Boot

- Generarea scheletelor de proiect pentru user-service, animal-service și exemplar-service via [Spring Initializr](#).
- Adăugarea dependențelor de Web, JPA, MySQL, Lombok și Security acolo unde a fost necesar.

### 2. Definirea modelului de date

- Crearea entităților JPA (User, Animal, Exemplar) și maparea relațiilor în clase.
- Generarea scripturilor schema.sql și data.sql pentru populări de test.

### 3. Implementare strat Config & Securitate

- Configurarea JWT în SecurityConfig, JwtUtil și JwtAuthenticationFilter.
- Setarea politicilor CORS și a interceptorului pentru RestTemplate (în exemplar-service).

### 4. Construirea layer-elor Controller

- Definirea contractelor REST în AuthController și UserController.
- Crearea endpoint-urilor de CRUD în AnimalController și ExemplarController.

- Testare manuală cu Postman pentru fiecare rută.

## **5. Implementare layer-ilor Service și Facade**

- Dezvoltarea logicii de business în UserService, AnimalService, ExemplarService.
- Introducerea ExemplarFacade pentru orchestrarea apelurilor interne și mapări DTO.

## **6. DTO & Builder**

- Adăugarea claselor DTO (UserDTO, AnimalDTO, ExemplarDTO, StatsDTO) cu @Builder și @Data.
- Metode statice de conversie fromEntity() și toEntity().

## **7. Implementare exporturi și strategii**

- Refactorizarea exportului din animal-service folosind ExportStrategy (CSV, JSON, XML, DOCX).
- Înregistrarea fiecărei strategii ca bean Spring și maparea dinamică în AnimalExporterService.

## **8. Proxy REST intern**

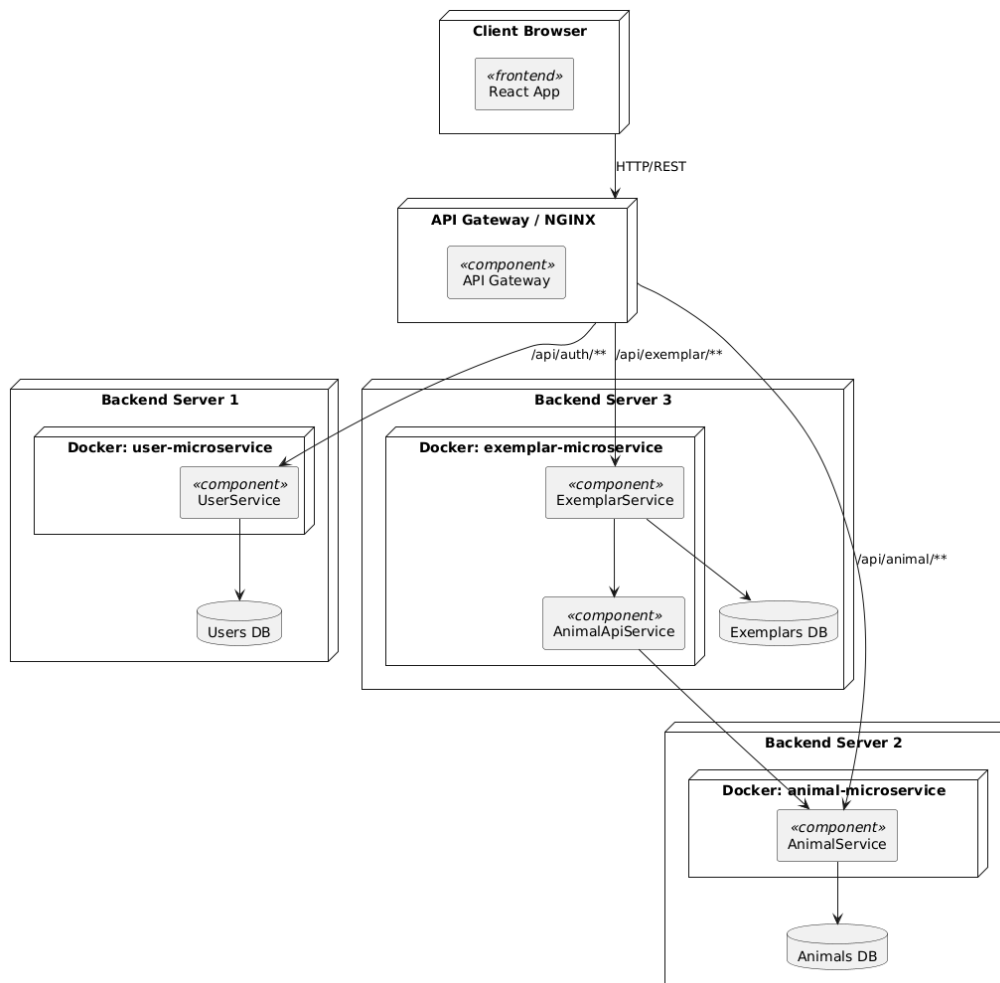
- Definirea AnimalApiService în exemplar-service pentru apeluri HTTP către animal-service.
- Testarea integrării end-to-end: crearea unui exemplar și verificarea că DTO-ul este îmbogățit corect.

## **9. Testare și validare**

- Scenarii de testare cu Postman/Newman pentru toate endpoint-urile.

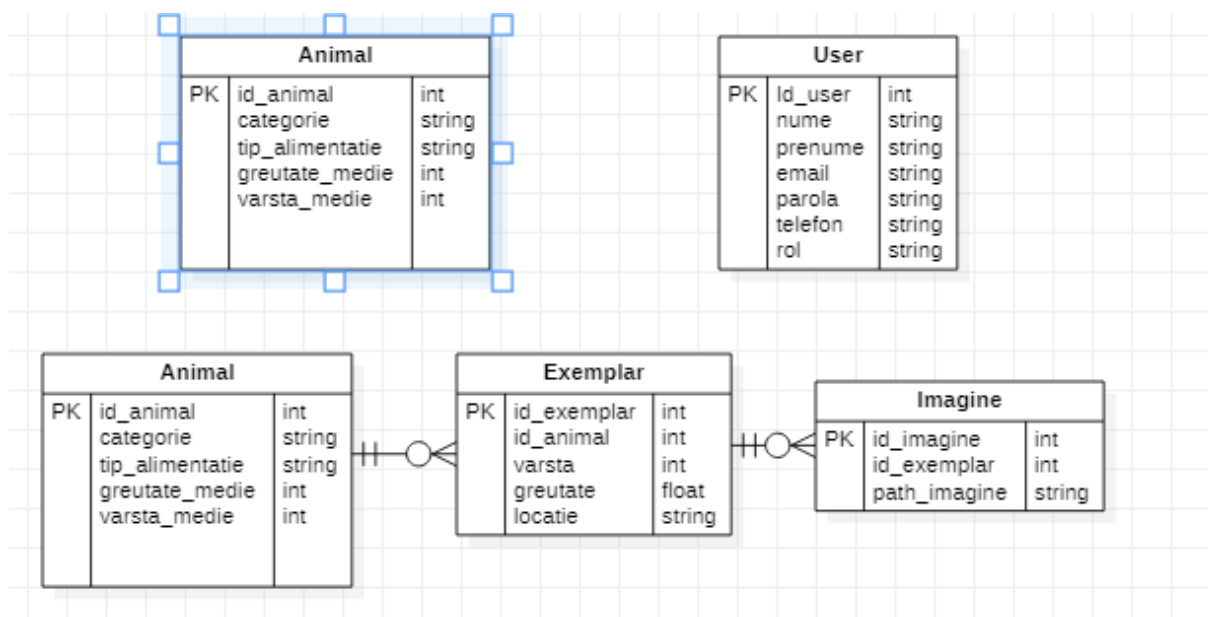
## **10. Documentație și diagrame UML**

- Generare diagrame de clase, componente și secvență în PlantUML.



## 6. Modelarea Bazei de Date (ERD)

Fiecare microserviciu deține propria schemă relațională, izolată de celelalte, pentru a păstra principiul **database-per-service**. Mai jos este descrierea entităților și a relațiilor din fiecare bază de date:



## 6.1 user-service (Users DB)

### Tabel: users

Coloana	Tip	Constraint
id	BIGINT	PK, auto-increment
username	VARCHAR(100)	NOT NULL, UNIQUE
email	VARCHAR(150)	NOT NULL, UNIQUE
password	VARCHAR(255)	NOT NULL
phone	VARCHAR(20)	
user_type	VARCHAR(20)	NOT NULL (CLIENT, EMPLOYEE, MANAGER...)
created_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP
updated_at	TIMESTAMP	ON UPDATE CURRENT_TIMESTAMP

*Nicio relație externă – serviciul este autonom.*

## 6.2 animal-service (Animals DB)

### Tabel: animals

Coloana	Tip	Constraint
id	BIGINT	PK, auto-increment



Coloana	Tip	Constraint
name	VARCHAR(100)	NOT NULL
category	VARCHAR(50)	NOT NULL
diet_type	VARCHAR(50)	NOT NULL
habitat	VARCHAR(100)	
average_weight	DECIMAL(8,2)	
average_age	DECIMAL(5,2)	
created_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP
updated_at	TIMESTAMP	ON UPDATE CURRENT_TIMESTAMP

*Niciun FK extern – toate datele speciilor se păstrează local.*

### 6.3 exemplar-service (Exemplars DB)

**Tabel: exemplars**

Coloana	Tip	Constraint
id	BIGINT	PK, auto-increment
animal_id	BIGINT	NOT NULL — face referința “logică” către animals.id
name	VARCHAR(100)	NOT NULL
location	VARCHAR(100)	
age	DECIMAL(5,2)	
weight	DECIMAL(8,2)	
notes	TEXT	
created_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP
updated_at	TIMESTAMP	ON UPDATE CURRENT_TIMESTAMP

**Note:**

- `animal_id` nu are constrângere FK directă în baza de date pentru a păstra decuplarea fizică (eventuală folosire de caching sau replicare). Legătura logică este realizată în cod prin `AnimalApiService`.
- Dacă se dorește integritate referențială strictă, s-ar putea adăuga un FK către `animals` din `animal-service`, dar adesea se evită cross-DB constraints în arhitecturi microservicii.

## 7. Descrierea aplicației

Pagina de primire unde avem opțiunea de conectare la aplicație ca utilizator care nu este vizitator (Angajat, manager, admin) dar și un buton care ne duce la pagina principală.

### Zoo App

Română

English

Français

# Welcome to the Zoo App

Go to Home

Log In


Pagina principala Home care este destinata vizualizarii animalelor si a exemplarelor de catre vizitator.

# Welcome to the Zoo App

## Animals

ID	Name	Category	Diet Type	Habitat	Average Weight	Average Age
14	Dragon	Reptile	Carnivore	Castle	21	23
4	Giraffe	Mammal	Herbivore	Savannah	600	25
3	Hippos	Mammal	Herbivore	Savannah	800	25
16	Zebra	Mammal	Herbivore	Savannah	234	12

## Exemplars

ID	Animal ID	Name	Species	Images	Location	Age	Weight	Notes
53	4	ferty	Giraffe	 images-0	Bird Aviar	1.2	0.5	Newly arrived from rescue
55	3	FORTY	Hippos		12	213	21	123
59	4	bravo	Giraffe		21	21	123	123123qwasd

Pagina de conectare care va redirectiona utilizatorul pe pagina corespunzătoare după rol.

## Zoo App

Română

English

Français

## Log In

admin

.....

Sign In

Pentru angajat după înregistrare va fi direcționat către meniul de la angajat unde sunt opțiunile de adăugare, ștergere sau actualizare atât pentru exemplare cat si pentru animal.

## Zoo App

Română

English

Français

## Employee Dashboard

Animals

Exemplars

Add New Animal

ID	Species	Category	Actions
14	Dragon	Reptile	<a href="#">View</a> <a href="#">Edit</a> <a href="#">Delete</a>
4	Giraffe	Mammal	<a href="#">View</a> <a href="#">Edit</a> <a href="#">Delete</a>
3	Hippos	Mammal	<a href="#">View</a> <a href="#">Edit</a> <a href="#">Delete</a>
16	Zebra	Mammal	<a href="#">View</a> <a href="#">Edit</a> <a href="#">Delete</a>

Pentru pagina de manageri avem aceleași funcționalități ca la angajat plus opțiunea de vizualizare a unor statistici si cea de exportare a acestora.

Zoo App

RomânăEnglishFrançais

Manager Dashboard

Animals by Category

Category	Animals per Category
Mammal	3.0
Reptile	1.0

Diet Type Breakdown

Diet Type	Percentage
Carnivore	25%
Herbivore	75%

Averages

Average Age: 21.25

Average Weight: 413.75

Export to Word

Animal Management

Add New Animal

Animal Management

Add New Animal

ID	Name	Category	Diet	Actions
14	Dragon	Reptile	Carnivore	<a href="#">Edit</a> <a href="#">Delete</a>
4	Giraffe	Mammal	Herbivore	<a href="#">Edit</a> <a href="#">Delete</a>
3	Hippos	Mammal	Herbivore	<a href="#">Edit</a> <a href="#">Delete</a>
16	Zebra	Mammal	Herbivore	<a href="#">Edit</a> <a href="#">Delete</a>

Exemplar Management

Add New Exemplar

ID	Name	Animal ID	Location	Actions
53	ferty	4	Bird Aviar	<a href="#">Edit</a> <a href="#">Delete</a>
55	FORTY	3	12	<a href="#">Edit</a> <a href="#">Delete</a>
59	bravo	4	21	<a href="#">Edit</a> <a href="#">Delete</a>

Pagina speciala pentru admin are aceleași funcționalități ca cea de la manager plus opțiunea de efectuare de operații precum ștergere, actualizare si creare a

utilizatorilor opțiuni care sunt urmate de o notificare automata pe mail a acestora.

## 8. Concluzii

Proiectul implementat pune în practică o arhitectură modernă, bazată pe microservicii, adaptată nevoilor de gestionare a utilizatorilor, speciilor de animale și exemplarelor acestora. Aplicând principiile **Domain-Driven Design**, soluția oferă:

- **Scalabilitate**  
Fiecare microserviciu (user-service, animal-service, exemplar-service) poate fi scalat independent în funcție de volum.
- **Modularitate și întreținere facilă**  
Codul este organizat pe layere clare (config, controller, service, domain, dto, repository) și pe bounded contexts, ceea ce ușurează testarea unităților și a integrărilor.
- **Separarea datelor**  
Fiecare serviciu deține propria bază de date MySQL, eliminând dependențele fizice între scheme și permițând evoluția independentă.
- **Design pattern-uri relevante**
  - **Singleton**: bean-urile Spring (ex. NotificationService/UserService) — o singură instanță pe context.
  - **Facade**: ExemplarFacade pentru a simplifica interfața către controller.
  - **Builder**: DTO-urile (AnimalDTO, ExemplarDTO, UserDTO) — crearea fluentă și sigură a obiectelor.
  - **DTO**: obiecte de transfer pentru a izola modelul intern al entităților.
  - **Proxy**: AnimalApiService — client REST local pentru apeluri la animal-service, extensibil cu cache sau retry.
- **Securitate și izolarea accesului**  
Autentificarea și autorizarea bazate pe JWT (filter + config), cu roluri clare (CLIENT, EMPLOYEE, MANAGER, ADMIN).

În ansamblu, proiectul demonstrează nu doar stăpânirea tehnologiilor (Java, Spring Boot, React), ci și aplicarea coerentă a principiilor de arhitectură software și pattern-urilor de proiectare, asigurând un sistem robust, flexibil și ușor de întreținut.

