

UNIVERSITATEA BABEȘ-BOLYAI
Facultatea de Științe Economice și Gestiunea Afacerilor
Informatică Economică

Lucrare de licență

Absolvent,
Mihaela-Elizabeta **BALICA**

Coordonator științific,
Prof. univ. dr. Gheorghe Cosmin **SILAGHI**

2022

UNIVERSITATEA BABEȘ-BOLYAI

Facultatea de Științe Economice și Gestiunea Afacerilor

Informatică Economică

Lucrare de licență

Artefacto

Aplicație web de ghidaj în muzeele de artă

Absolvent,

Mihaela-Elizabeta **BALICA**

Coordonator științific,

Prof. univ. dr. Gheorghe Cosmin **SILAGHI**

2022

Rezumat

Prezenta lucrare urmărește să surprindă procesul de dezvoltare a unei aplicații web care să întâmpine cerințele actuale ale muzeelor de artă și nevoia lor de digitalizare. În acest scop, aplicația funcționează atât ca un sistem de gestiune a operelor de artă deținute de muzee, precum și drept un asistent virtual pentru vizitatori. Interfața aplicației pune la dispoziție informații despre opere, printre care: artist, curent artistic, descriere și amplasare în muzeu, cu posibilitatea de a naviga între galerii. În timp ce administratorii pot edita aceste informații sau adăuga noi înregistrări, utilizatorii au opțiunea de a salva operele de artă favorite și de a adăuga comentarii pe paginile operelor de artă.

Cuprins

Abrevieri	5
Lista tabelelor și figurilor	6
Introducere	7
1. Identificarea și descrierea problemei de business	9
1.1 Motivație	9
1.2 Context	15
2. Cerințe de sistem	17
2.1 Surse de cerințe	17
2.2 Elicitația cerințelor	19
2.2.1 Metoda de elicitare 1 - Interviu	19
2.2.2 Metoda de elicitare 2 – Brainstorming	20
2.2.3 Modelul Use-Case	21
2.3 Documentarea cerințelor	24
2.3.1 Diagrama de activități	24
2.3.2 Diagrama de stări	26
2.4 Model de dezvoltare	27
3. Proiectarea sistemului informatic	28
3.1. Proiectarea logică	28
3.1.1 Arhitectura sistemului	30
3.1.2 Baza informațională	34
3.2. Proiectarea tehnică	35
3.2.1 Structura fizică a datelor	35
3.2.2 Procese și algoritmi	37
3.2.3 Tehnologii specifice	38
4. Implementare	40
4.1 Implementarea bazei de date	40
4.2 Implementarea aplicației backend	42
4.2.1 Clasa Entity	42
4.2.2 Interfața Repository	44
4.2.3 Clasa DTO	45
4.2.4 Interfața Mapper	46
4.2.5 Interfața Service	47
4.2.6 Clasa Controller	49
4.2.7 Componenta de securitate	51
4.3 Implementarea aplicației frontend	52
4.3.1 Componentele frontend	52
4.3.2 Navbar	55
4.3.3 Rutele și restricțiile de acces	56
4.3.4 User state management	57
5. Testare și evaluare	59
5.1 Testarea manuală	59
5.2 Testarea prin cereri HTTP	60
5.3 Evaluare	62
Concluzii	63
Glosar	65
Bibliografie	66
Anexe	68

Abrevieri

<i>API</i>	Application Programming Interface
<i>CRUD</i>	Create, Retrieve, Update, Delete (i.e. Creare, Regăsire, Actualizare, Ștergere)
<i>CSS</i>	Cascading Style Sheets
<i>DAO</i>	Data Access Object
<i>DTO</i>	Data Transfer Object
<i>HTML</i>	Hypertext Markup Language
<i>HTTP</i>	Hypertext Transfer Protocol
<i>IT</i>	Information Technology
<i>JDK</i>	Java Development Kit
<i>JPA</i>	Java Programming API
<i>NG</i>	Angular
<i>REST</i>	Representational State Transfer
<i>SQL</i>	Structured Query Language
<i>TS</i>	TypeScript
<i>UI</i>	User Interface
<i>URL</i>	Uniform Resource Locator
<i>UX</i>	User Experience

Lista tabelelor și figurilor

Tabele:

1. Actori-roluri-obiective	18
2. Procesul de testare manuală	60

Figuri:

Figură 1. Diagrama Fishbone a cauzelor	12
Figură 2. Diagrama de descompunere a obiectivelor.....	14
Figură 3. Diagrama Pareto a cauzelor	20
Figură 4. Modelul Use-Case general.....	22
Figură 5. Modelul Use-Case particular	23
Figură 6. Diagrama de activități.....	25
Figură 7. Diagrama de stări.....	26
Figură 8. Panoul de dezvoltare Kanban	27
Figură 9. Diagrama de flux de date	29
Figură 10. Arhitectura sistemului multi-layer.....	31
Figură 11. Diagrama de componente	33
Figură 12. Diagrama Entitate-Relație a bazei de date.....	35
Figură 13. Structura bazei de date în MySQL	41
Figură 14. Interfața Mapper	47
Figură 15. Clasa ArtworkServiceImpl	48
Figură 16. Metoda updateArtwork din clasa tip Service.....	48
Figură 17. Metoda getAllArtworksByRoomIdOrderByPosition	49
Figură 18. Exemplu de endpoint din clasa ArtworkController - adnotări.....	49
Figură 19. Exemplu de endpoint din clasa ArtworkController – metoda get	50
Figură 20. Endpoint în interfața Swagger	50
Figură 21. Componentele pachetului security.....	51
Figură 22. Autorizarea căilor API în Spring Security.....	51
Figură 23. Structura componentelor din dashboard.....	52
Figură 24. Elementele unei componente Angular.....	53
Figură 25. Fereastra de sign-up.....	53
Figură 26. Pagina principală a aplicației.....	54
Figură 27. Mesaj de solicitare a autentificării.....	54
Figură 28. Pagina unei galerii de artă	55
Figură 29. Navbar pentru desktop.....	55
Figură 30. Versiunea de mobil a aplicației	56
Figură 31. Rutele și modulele aplicației.....	56
Figură 32. RouteGuards pe rutele paginii unei galerii	57
Figură 33. Clasa AuthorisationGuard	57
Figură 34. User state management.....	58
Figură 35. Cerere GET trimisă din Postman.....	61
Figură 36. Cerere POST trimisă din Swagger.....	61
Figură 37. Cereri GET trimise de aplicația frontend.....	62

Introducere

Atât arta, cât și tehnologia își au originea în conceptul Greciei antice de *tékhnē*, care înseamnă artă, meșteșug (Marinescu, 2014), însă s-au dezvoltat într-o oarecare măsură independent și au ajuns să constituie obiecte foarte distincte de activitate și de studiu. Totuși, în momentul prezent, observăm cum tehnologia devine parte integrantă din tot mai multe domenii, fapt pentru care cele două ramuri au din nou posibilitatea să existe într-o interdependență de ansamblu. Fie că spunem că arta e un scop în sine, iar tehnologia e un instrument de îmbunătățire a realității, fie că susținem că tehnologia este o artă în sine, care este la fel de omniprezentă ca artele vizuale, avem dreptate.

În cadrul unui interviu despre imersiunea tehnologiei în mediul artistic, Jane Alexander, directorul sectorului informatic al Muzeului de Artă Cleveland, remarcă: „Cea mai bună întrebare a digitalizării [în muzee] nu te face să remarci tehnologia, ci să remarci arta.” (Song, 2017, tradus din engleză).

Putem spune, prin urmare, că tehnica aflată în permanentă evoluție ne pune la dispoziție un set de mijloace și procese care vor deveni esențiale în facilitarea experienței culturale și în intermedierea contactului cu mediul artistic, pentru experți și amatori deopotrivă.

În acest context prefigurat mai sus, am încercat, prin acest proiect, să ofer un răspuns practic și ilustrativ la problematica rezumată de întrebarea: *Cum poate fi utilizată tehnologia pentru a optimiza accesul publicului larg la artă și cultură?*

Acest demers se concretizează într-o aplicație web cu mai multe roluri: să ofere asistență și explicații persoanelor aflate într-o vizită fizică la muzeu, să fie o sursă de documentare și un intermediar cultural pentru cei care nu se pot deplasa până la muzeu, însă ar dori să-i studieze colecția și, la fel de important, să ofere personalului administrativ un sistem de gestiune și evidență a operelor de artă aflate în subordinea muzeului, dar și a vizitatorilor acestuia.

Obiectivul principal al acestei lucrări constă în descrierea și urmărirea ciclului de dezvoltare a aplicației: de la înțelegerea contextului și extragerea cerințelor, până la stabilirea arhitecturii și implementarea propriu-zisă. De asemenea, obiectivul de ansamblu al proiectului include și consolidarea cunoștințelor de programare și documentare, prin studiul și utilizarea principalelor tehnologii, anume Java Spring Boot și Angular, descrise în detaliu în capitolele ce urmează.

Structura acestui document are la bază întocmai etapele acestui proces de dezvoltare a aplicației, respectând următoarea ordine:

- primul capitol (*Identificarea și descrierea problemei de business*) descrie elementele preliminare caracteristice unui raport de analiză a mediului de business (cu subcapitole despre identificarea motivației și a contextului);
- analiza sistemului informatic se continuă în capitolul 2 (*Cerințe de sistem*), cu procesul de elicitare a cerințelor informaționale, bazat pe identificarea surselor, aplicarea metodelor de elicitare, documentarea cerințelor și stabilirea modelului de dezvoltare, toate surprinse în subcapitole distincte;
- capitolul 3 (*Proiectarea sistemului informatic*) are în vedere cele două etape – proiectarea logică (a arhitecturii și a bazei informaționale) și proiectarea tehnică (a structurii datelor și a principalilor algoritmi);
- în continuare, capitolul 4 (*Implementare*) se axează pe modul în care această arhitectură s-a concretizat, prin exemple de componente software, alături de funcționalitățile acestora;
- capitolul 5 (*Testare și Evaluare*) cuprinde metodele de testare și verificare întrebuințate, rezultatele acestora, precum și un subcapitol de evaluare a produsului;
- lucrarea se încheie cu un capitol aferent concluziilor și posibilităților viitoare de dezvoltare, urmate de un glosar al principalilor termeni tehnici, bibliografia surselor citate și anexele documentației.

1. Identificarea și descrierea problemei de business

Acest capitol are în vedere etapele procesului de analiză a contextului de business, cu scopul de a identifica punctual cerințele de sistem ale principalilor *stakeholderi*, cazurile de utilizare și cel mai potrivit model de dezvoltare pentru acest sistem informatic.

Identificarea problemei generale de business presupune înțelegerea contextului de ansamblu și a factorilor care au condus la această situație, urmată de stabilirea principalelor obiective care ar rezolva în cea mai mare măsură problema și care vor fi incluse ulterior în cerințele aplicației.

În acest scop, este necesar întâi să constatăm care este motivația care guvernează întregul proiect, generată atât de cauze subiective, cât și obiective. După aceea, putem corela aceste cauze cu posibile soluții formulate sub forma de obiective, iar apoi stabilim modul în care acestea se integrează în diferitele fațete ale contextului.

1.1 Motivație

Pentru înțelegerea motivației pe care se fundamentează necesitatea acestei aplicații, putem pleca de la premisa că tehnologia este un instrument deosebit de util în structurarea, automatizarea și eficientizarea proceselor din orice ramură de activitate.

Am constatat o reorientare semnificativă a mediului cultural spre această fuziune cu tehnologia informației, întrucât majoritatea instituțiilor cultural-artistice și-au făcut apariția în mediul digital, optând să își diversifice serviciile prin produse IT care să vină în întâmpinarea clienților lor. Pentru o perspectivă mai concretă, am putea menționa aplicații pentru industria cinematografică sau teatrală, de fotografie și design, aplicații pentru biblioteci, pentru obiective turistice sau, cele care ne interesează pe noi, pentru mediul artistic. Multe dintre marile muzee de artă ale lumii, spre exemplu *Rijksmuseum* (Amsterdam), *The British Museum* (Londra), *Galeria Națională de Artă* (Washington), *The Metropolitan Museum of Art* (New York City) sau *Musée d'Orsay* (Paris) și-au creat propriile aplicații mobile sau web, în care se prezintă expoziții permanente și temporare, explicații și informații suplimentare (Ayers, 2016).

În România, acest proces este încă emergent și are nevoie de mai mult sprijin și mai multe resurse pentru a lua amploare. Singurul muzeu din țara noastră care a reușit să ia o astfel

de inițiativă este *Muzeul Național de Artă al României* (București), cu aplicația *ARTmobile – Un muzeu la îndemână* (MNAR, n.d.).

Pentru a putea avea o privire de ansamblu asupra factorilor care determină nevoia de o astfel de aplicație-suport pentru muzeele de artă, am reunit principalele cauze și subcauze în *Figura 1*, sub forma unei diagrame tip Fishbone.

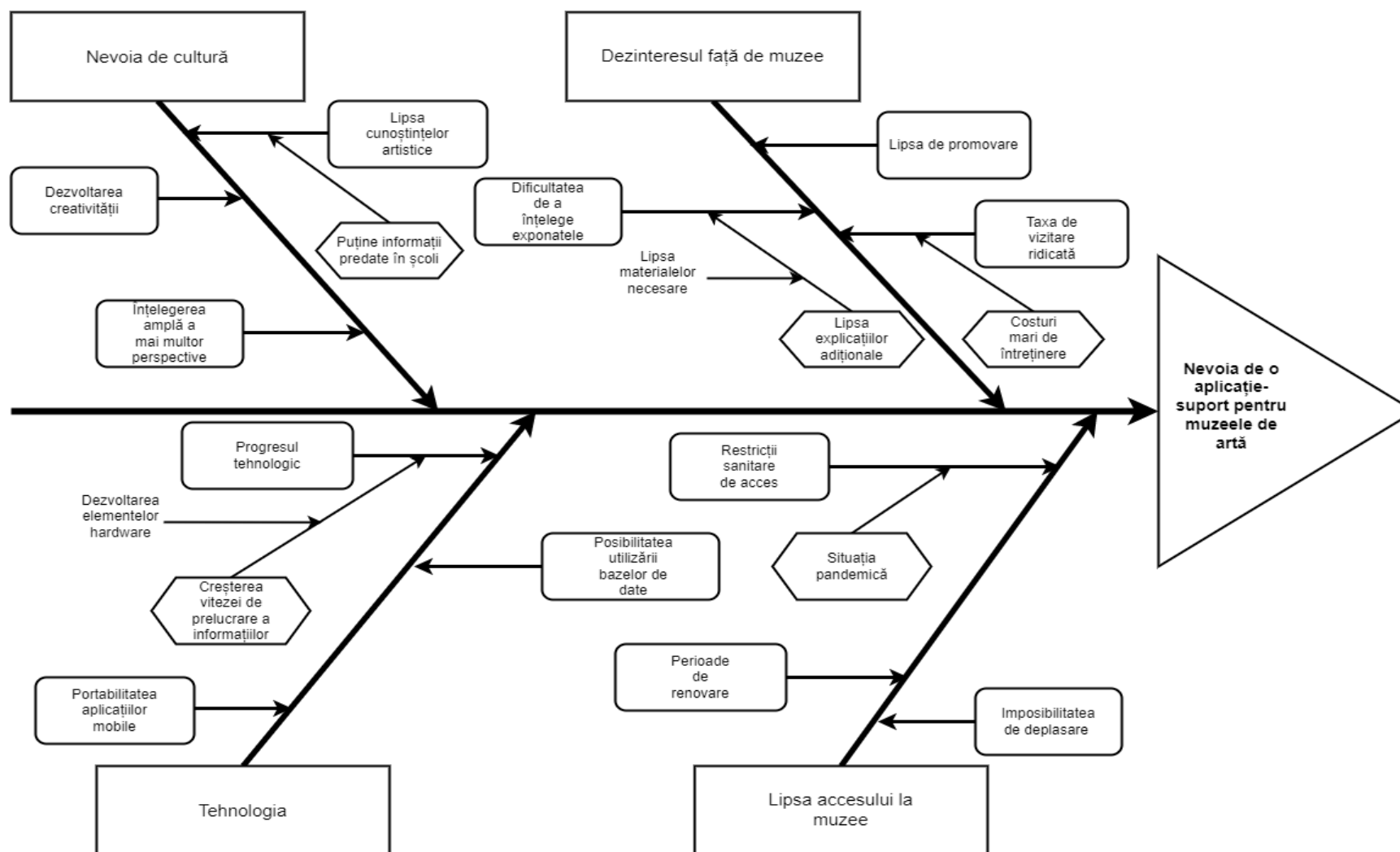
În acest caz, avem în vedere o aplicație web cu triplu rol:

- a) De a asista vizitatorii muzeelor de artă, oferindu-le explicații despre fiecare operă de artă pe care o observă, precum informații despre autor, curentul artistic, detalii de conținut și formă.
- b) De a oferi o bază de informații extinsă cu opere de artă și explicațiile aferente celor ce nu se pot deplasa fizic în muzeele respective.
- c) De a constitui un sistem de gestiune a acestei baze de date, cu posibilitatea de a efectua principalele operații de creare, regăsire, actualizare și stergere.

Cei patru factori principali care conduc la problema necesității dezvoltării acestei aplicații sunt:

1. Tehnologia – pe care o putem considera atât o cauză, cât și o soluție, iar drept subcauze, avem următoarele:
 - Progresul tehnologic, favorizat de creșterea vitezei de prelucrare a informațiilor, care a fost determinată de dezvoltarea permanentă a componentelor hardware;
 - Posibilitatea utilizării bazelor de date pentru gestiunea mai eficientă a informațiilor despre fiecare operă de artă;
 - Portabilitatea aplicațiilor mobile, întrucât majoritatea vizitatorilor ar avea un telefon mobil conectat la rețeaua wireless a muzeului.
2. Lipsa accesului la muzee – situație provocată de următoarele evenimente:
 - Perioadele de renovare sau reamenajare interioară, când muzeele sunt închise sau au un program restrâns;
 - Restricțiile de acces cauzate de situația pandemică – multe muzee au fost închise sau și-au redus capacitatea. În prezent, acest lucru s-a remediat, însă a condus la o conștientizare a nevoii unei variante alternative sau de rezervă tocmai pentru aceste situații;
 - Imposibilitatea unor oameni de a se deplasa până la anumite muzee, din diverse motive, precum lipsa timpului sau a posibilităților financiare.
3. Dezinteresul față de muzee – care derivă din mai multe subcauze:

- Lipsa de promovare a muzeelor – se întâmplă ca oamenii să nu știe că există anumite muzee sau galerii, iar adesea expozițiile temporare nu se bucură de prea mult succes pentru că este foarte costisitor să fie promovate corespunzător având în vedere durata restrânsă de timp în care vor fi vizitabile;
 - Dificultatea de a înțelege exponatele – poate cea mai importantă problemă pe care ar soluționa-o aplicația noastră – determinată adesea de lipsa explicațiilor care să însoțească operele de artă. Aceasta se întâmplă fie din lipsa resurselor necesare (costuri de achiziționare a materialelor fizice, costuri de tipărire și cu personalul necesar alcătuirii textului), fie deoarece curatorii le consideră necesare;
 - Taxa de vizitare ridicată, care este rezultatul costurilor mari de întreținere și de păstrare a anumitor condiții de temperatură, umiditate, iluminat etc. În plus, în majoritatea muzeelor se percep taxe pentru tururile ghidate, ceea ce îi descurajează pe mulți oameni să opteze pentru această variantă.
4. Nevoia de cultură – produsul unor subcauze precum:
- Lipsa cunoștințelor despre domeniul artistic, ca urmare a unei expuneri reduse la acest subiect în timpul educației școlare;
 - Necesitatea dezvoltării gândirii creative sau inovative, aptitudini ce devin din ce în ce mai căutate în mediul dinamic al societății contemporane;
 - Nevoia de a înțelege mai multe perspective de abordare a unor concepte, de a dezvolta gândirea critică și analitică.



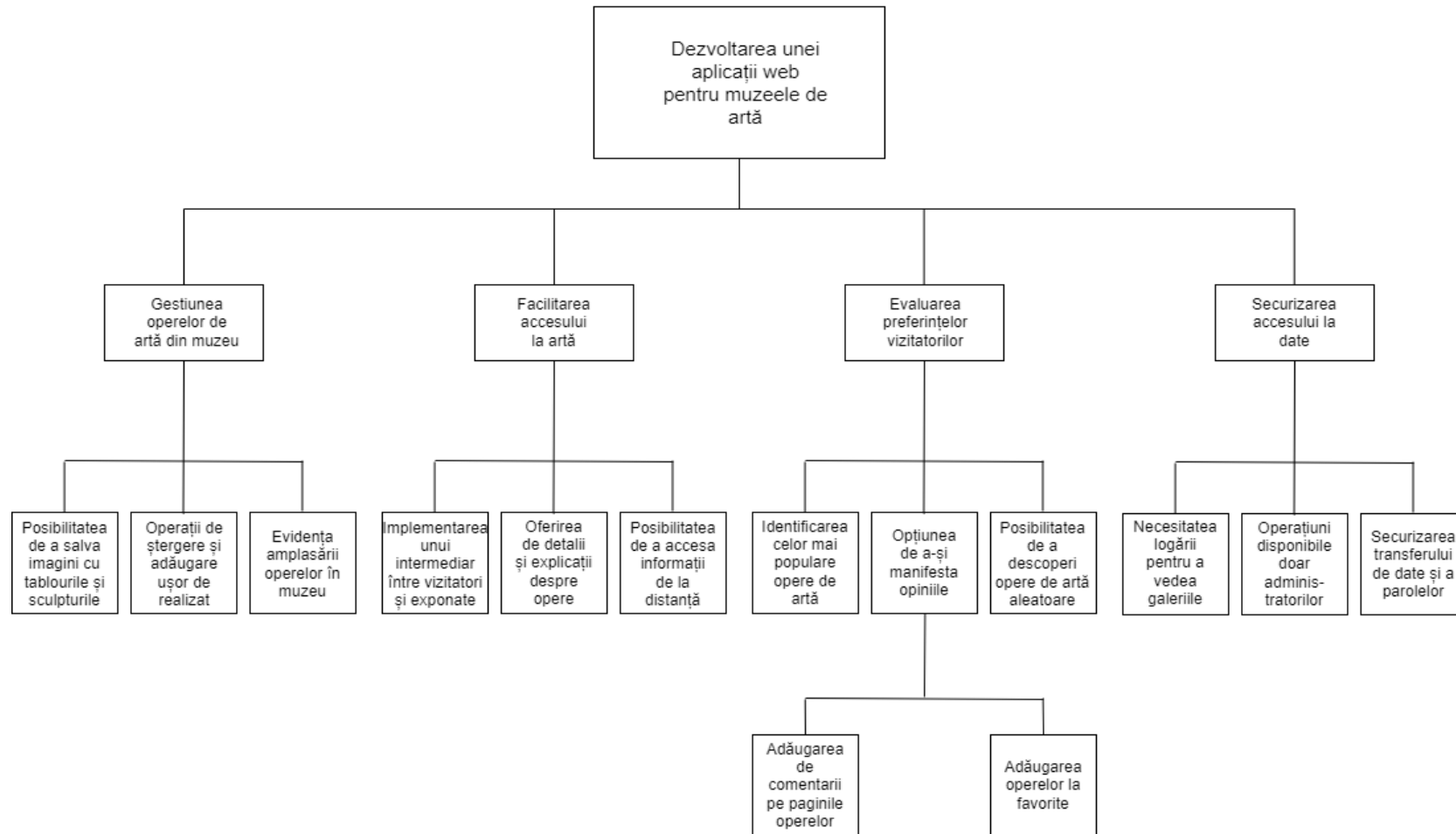
Figură 1. Diagrama Fishbone a cauzelor

Odată ce am stabilit că este nevoie de această aplicație, putem trece la Diagrama de descompunere a obiectivelor (*Figura 2*), ce are ca scop structurarea și segmentarea pe mai multe niveluri și procese a obiectivului principal, care constă, în acest caz, în dezvoltarea unei aplicații web care să funcționeze asemenea unui ghid virtual în cadrul vizitelor la muzeele de artă.

Acest parcurs este divizat în 4 mari subobiective, după cum urmează:

1. Gestiunea operelor de artă din muzeu are în vedere salvarea imaginilor acestora în baza de date, alături de informațiile esențiale (artist, curent, descriere etc.), posibilitatea de a efectua operațiile CRUD asupra înregistrărilor, precum și ținerea evidenței amplasării operelor în muzeu (în funcție de galerie și numărul camerei), pentru a le putea regăsi mai ușor la nevoie.
2. Obiectivul de facilitare a accesului la artă, care să contracareze subcauzele prezente pe ramura privind Lipsa accesului la muzee din diagrama Fishbone. Acesta se concretizează în implementarea acestui intermediar virtual între vizitatori și exponate (prin interfață), alături de oferirea explicațiilor necesare pentru a înțelege mai bine artefactele și, totodată, de posibilitatea de a accesa aceste informații de la distanță, fără necesitatea deplasării fizice la muzeu.
3. Evaluarea preferințelor vizitatorilor, în funcție de care muzeele ar putea determina ce expoziții viitoare ar fi mai potrivite și cum să amplaseze mai bine operele în muzeu. Pentru aceasta, ar fi necesară identificarea celor mai populare exponate prezente în galerii, oferirea posibilității vizitatorilor de a-și manifesta opiniile prin intermediul comentariilor și al opțiunii de adăugare la favorite, precum și adăugarea unei funcționalități care permite descoperirea unor opere de artă aleatoare, prin apăsarea unui buton cu *randomizare* – astfel, utilizatorii ar fi expuși la mai multe opere, fără a fi necesar să le caute propriu-zis prin navigarea pe paginile muzeelor și ale galeriilor.
4. Este foarte importantă și securizarea accesului la datele aplicației, pentru a putea preveni anumite atacuri, breșe sau pierderi de informații. Pentru aceasta, se urmărește necesitatea logării pentru a accesa paginile efective ale galeriilor (care conțin operele de artă și comentariile), restricționarea anumitor operațiuni (adăugare de opere, actualizare, ștergere) doar persoanelor cu rol de administrator și securizarea parolelor și a transferului datelor în interiorul aplicației.

Diagrama de descompunere a obiectivelor



Figură 2. Diagrama de descompunere a obiectivelor

1.2 Context

Contextul acestui sistem informatic cuprinde cele 4 fațete, după cum urmează:

1. **Fațeta subiect** care are în vedere, în primul rând, stakeholderii, dintre care fac parte reprezentanții muzeelor de artă (cei care au constatat necesitatea dezvoltării acestei aplicații), precum și personalul care va administra baza de date și va realiza mentenanța aplicației. De asemenea, sunt prezenți și utilizatorii – aici sunt vizați atât cei care vor să consulte pagina unui muzeu și colecția de artă, cât și, mai ales, cei care vor vizita muzeul și vor utiliza aplicația pentru a afla informații despre exponate.

Datele propriu-zise constau în detalii despre toate operele deținute de fiecare muzeu, însoțite de date despre localizarea în muzeu, despre autori și curentele artistice cărora le aparțin. De asemenea, se vor adăuga date despre activitatea utilizatorilor în aplicație: comentarii, adăugări la favorite.

2. **Fațeta utilizare** se concentrează pe interesul utilizatorilor de a obține informații despre operele de artă, prin următoarele procesul de navigare la pagina muzeului căutat, identificarea galeriei și a camerei, apoi afișarea explicațiilor despre respectivele opere de artă pe care fie le observă în timp real în muzeu, fie doar le studiază online. Aici există și posibilitatea de a naviga mai departe către biografia artistului sau către informațiile despre curentul artistic.

Prin crearea unui cont, utilizatorii au opțiunea de a adăuga comentarii la operele de artă și au acces la funcționalitatea de adăugare a operelor de artă în lista de favorite, pe care o pot consulta și edita ulterior.

3. **Fațeta IT** presupune următoarele aspecte:

Dacă muzeele dispun deja de baze de date privind operele de artă, atunci se va face doar o centralizare a acestor informații prin migrare sau acces API. În cazul în care nu există deja un sistem informatic pentru evidența datelor, va fi necesar un pas suplimentar pentru colectarea și structurarea datelor.

Prin același raționament, este necesară existența unei rețele interne de calculatoare (compatibilă cu aplicația) pentru administrarea permanentă a sistemului, verificarea și actualizarea datelor.

Nu în ultimul rând, condițiile tehnice includ și disponibilitatea unei rețele de internet WiFi pe întreg perimetrul muzeelor, la care vizitatorii să se conecteze pentru a putea accesa aplicația web de pe telefonul mobil.

Prima parte este axată pe implementarea unei baze de date relaționale de tip server persistent, în acest caz prin intermediul MySQL. În continuare, pe partea de backend, se utilizează framework-ul Java Spring Boot pentru construirea și configurarea serviciului REST API. Printre instrumentele necesare se numără JDK 17, Apache Maven și toate dependențele necesare, IntelliJ IDEA ca mediu de lucru.

Se realizează maparea între entitățile din baza de date și clasele Java, ceea ce permite efectuarea operațiilor tip CRUD.

Pentru frontend: framework-ul Angular bazat pe limbajul TypeScript, pentru implementarea design-ului, trimiterea cererilor HTTP către backend și afișarea rezultatelor. De asemenea, se adaugă framework-ul de CSS Bootstrap pentru asigurarea calității de *responsiveness* a aplicației, indiferent de tipul de device de pe care este accesată.

Toate aceste tehnologii vor fi descrise mai amănunțit în cadrul subcapitolului 3.2.3 *Tehnologii specifice*, din cadrul secțiunii de proiectare.

4. Fațeta dezvoltare

Dezvoltarea se realizează, în principiu, după metodologia Agile Kanban, pentru integrarea eficientă a fiecărei componente în aplicația web.

Elementele menționate anterior la fațeta IT vor fi segmentate în componente și subcomponente, apoi se determină sarcinile de lucru (*task-uri*) și se alcătuiesc panourile de lucru pentru a ține evidența progresului.

Întreg acest proces va fi detaliat în continuare, în cadrul subcapitolului 2.4 *Model de dezvoltare*, aparținând capitolului privind cerințele de sistem.

2. Cerințe de sistem

Odată identificate principalele obiective și fațete ale sistemului informatic, este necesar să le trecem să le trecem către stadiul și mai concret de cerințe funcționale și non-funcționale.

În definiția lui Dumitru Oprea (2005), o cerință informațională reprezintă „o funcție sau o caracteristică a noului sistem (...) [care] răspunde obiectivelor unei organizații și pentru a rezolva un set de probleme”. În conformitate cu acest concept, vom încerca în continuare să determinăm cele mai importante surse de cerințe, apoi aplicăm metodele de elicitare și documentăm cerințele obținute.



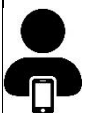
2.1 Surse de cerințe

În literatura de specialitate, se consideră că principalele surse de cerințe funcționale privind sistemul informatic sunt *stakeholderii*, dintre care se remarcă utilizatorii programului, beneficiarii (cei care solicită dezvoltarea aplicației) și personalul care va asigura mentenanța sistemului (Oprea, 2005). În consecință, pe aceștia îi regăsim în *Tabelul 1*, în calitate de actori, fiecare având roluri și obiective care le corespund.

Se începe cu reprezentanții muzeului și cei cu funcția de *product owner* – persoanele care asigură livrarea produsului cu valoarea scontată și stabilesc sarcinile de lucru (Schwaber & Sutherland, 2020). În vederea obținerii unei aplicații care să corespundă cât mai bine necesităților identificate, ei își vor comunica cerințele, restricțiile de timp și de buget. În continuare, vor colabora cu dezvoltatorii pentru a verifica periodic produsul, iar, după lansare, reprezentanții muzeului își vor încuraja vizitatorii să acceseze aplicația pentru a găsi explicațiile de care au nevoie.

Personalul ce va administra sistemul urmărește buna mentenanță a aplicației prin identificarea și soluționarea erorilor, alături de actualizarea datelor odată cu introducerea unor noi opere de artă sau a modificării amplasării lor în muzeu.

Utilizatorii vor accesa aplicația pentru a regăsi explicațiile necesare în vizita lor la muzeul de artă; în decursul navigării, ei pot salva operele la favorite și pot adăuga comentarii. Mai există și posibilitatea analizei arhivei muzeului pentru selectarea informațiilor căutate, în cazul utilizatorilor online care doresc să se documenteze fie înaintea vizitei la muzeu, fie fără să viziteze muzeul.

Actori	Rol	Obiective
 <p>Reprezentanții muzeului (manageri&product owners)</p>	Comunicarea necesităților, asigurarea finanțării, verificarea produsului final	<ul style="list-style-type: none"> • Obținerea unei aplicații care satisface cerințele • Încadrarea în buget și în limitele de timp • Îmbunătățirea experienței vizitatorilor prin promovarea utilizării aplicației
 <p>Administratorii de sistem</p>	Mentenanța aplicației și actualizarea datelor	<ul style="list-style-type: none"> • Menținerea aplicației în stare bună de funcționare • Identificarea, semnalarea și tratarea eventualelor erori • Asigurarea corespondenței între datele afișate de aplicație și realitatea din muzeu • Adăugarea noilor expozate
 <p>Utilizatorii aplicației</p>	Accesul și navigarea prin aplicație	<ul style="list-style-type: none"> • Folosirea aplicației pentru a găsi rapid explicații și a înțelege mai bine operele de artă în timpul vizitelor • Descoperirea diverselor colecții ale muzeelor de artă și selectarea celor preferate sau de interes • Posibilitatea de a afla informații despre operele de artă online, fără a fi nevoie să se deplaseze la muzeu

Tabel 1. Actori-roluri-obiective

2.2 Elicitația cerințelor

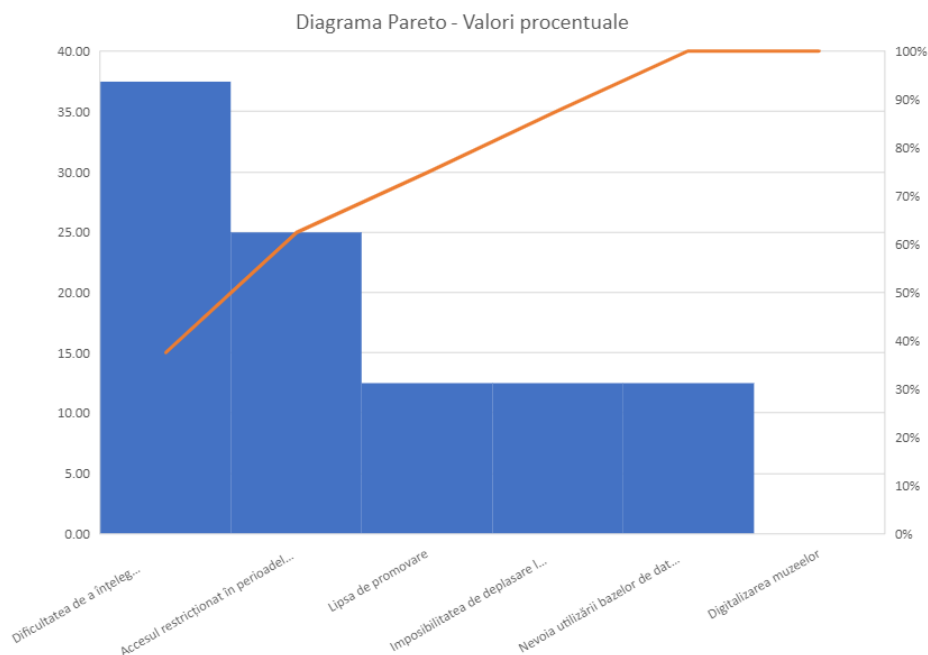
Pentru a asigura complexitatea aplicației și îndeplinirea cât mai bună a nevoilor beneficiarilor, optăm pentru două metode de elicitare a cerințelor: metoda interviului și metoda brainstorming, detaliate în subcapitolele următoare.

2.2.1 Metoda de elicitare 1 - Interviu

Interviul care se regăsește în *Anexa 1* se adresează principalilor *stakeholderi* ai sistemului, pentru a descoperi ce așteptări ar avea de la o astfel de aplicație și care consideră că ar fi cele mai importante caracteristici pe care să le includă. Întrebările sunt structurate în 3 categorii: pentru personalul administrativ, pentru personalul tehnic și pentru potențialii utilizatori (vizitatori ai muzeului). Având în vedere că nu am avut acces la personalul calificat al unui muzeu de artă pentru a le adresa primele 2 serii de întrebări, cerințele extrase se fundamentează pe răspunsurile celei de-a treia categorii, anume potențialii utilizatori.

Respondenții susțin că ar fi atât vizitatori ai muzeului, cât și utilizatori la distanță, iar toți ar accesa aplicația de pe telefon. Ei consideră că printre cele mai utile elemente într-o aplicație de ghidaj s-ar număra posibilitatea de a vedea informații generale despre obiectul de artă („stilul efectuat, autor, data creării, în ce expoziții a participat”), posibilitatea de a mări imaginea și de a vizualiza doar operele unui singur artist. La întrebarea *Ce ar face aplicația mai ușor de utilizat?*, au răspuns că o interfață ușor de navigat și un tutorial de început. Toți respondenții consideră utilă posibilitatea de a salva operele de artă în lista de favorite și sunt de părere că explicațiile despre operele de artă ar trebui să fie într-un limbaj accesibil și nu foarte detaliate.

În privința întrebării *Care considerați că sunt cele mai importante cauze ce determină necesitatea unei astfel de aplicații?*, am realizat o diagramă Pareto pe baza răspunsurilor (*Figura 3*). Se poate observa că respondenții consideră dificultatea de a înțelege exponatele drept cea mai semnificativă cauză, urmată de accesul restricționat în muzee în perioadele de renovare. O importanță medie și egală o au imposibilitatea de deplasare la muzeu, lipsa de promovare și nevoia utilizării bazelor de date pentru gestiunea operelor de artă.



Figură 3. Diagrama Pareto a cauzelor

2.2.2 Metoda de elicitare 2 – Brainstorming

Odată ce obținem răspunsurile intervievaților și le centralizăm, putem conduce o sesiune de *brainstorming* pentru a stabili ce presupune fiecare cerință și pentru a identifica altele nemenționate până acum.

Într-o definiție concisă, conceptul de *brainstorming* desemnează procesul creativ de a genera idei în cadrul unui grup, cu scopul de a găsi soluții pentru anumite probleme (Besant, 2016). Putem structura acest demers creativ în 3 etape:

1. Stabilirea obiectivelor ședinței de *brainstorming*, anume:
 - Identificarea contextului în care se va dezvolta și implementa aplicația;
 - Analiza rezultatelor interviului și formularea clară a cerințelor de sistem;
 - Găsirea de posibile soluții pentru fiecare problemă;
 - Identificarea rolurilor necesare pentru echipa ce va realiza proiectul; eventual, propunerea și selecția membrilor echipei.
2. Generarea ideilor într-un mod constructiv și proactiv:
 - Stabilirea variabilelor contextuale, a platformei pe care se va implementa aplicația și regimul în care va funcționa;
 - Propunerea principalelor funcționalități;
 - Propunerea celor mai potrivite tehnologii și framework-uri.
3. Potrivirea și organizarea ideilor

- Descrierea sumară a unui șablon al cerințelor și eventual al arhitecturii, care se va defini mai concret în faza de proiectare;
- Stabilirea modelului de dezvoltare și a principalelor termene limită;
- Alegerea tehnologiilor și a resurselor necesare, în concordanță cu bugetul propus;
- Stabilirea cerințelor de performanță și eficiență.

2.2.3 Modelul Use-Case

În urma analizei categoriilor de utilizatori ai aplicației și a cerințelor acestora, putem delimita anumite cazuri concrete de utilizare, pentru a putea structura mai bine caracteristicile aplicației în continuare.

Astfel, diagrama din *Figura 4* prezintă modelul de ansamblu al cazurilor de utilizare, declanșate de următorii actori:

1. Utilizatorul – persoana care s-a logat cu un cont în aplicație. Acesta reprezintă o generalizare a cazului de utilizator-vizitator, persoană care are acces la un număr mai restrâns de funcționalități până la crearea unui cont.

Utilizatorului îi corespund următoarele cazuri de utilizare:

- a) Navigarea prin colecția muzeului: se pot vedea listele cu galeriile muzeului, sălile și operele de artă aflate în fiecare categorie.
- b) Din listele generale se poate trece la vizualizarea paginilor operelor de artă, ce conțin explicații, informații despre artist, curentul artistic etc. Această trecere implică și pasul de autentificare, realizată pe baza adresei de email și a parolei, pentru a putea intra în cont. Aceasta include și pasul de verificare a corectitudinii datelor introduse, urmată de trecerea la versiunea aplicației disponibilă utilizatorilor sau afișarea unui mesaj de eroare în cazul în care emailul și parola nu corespund.
- c) Pe pagina unei opere de artă, utilizatorii logați au de asemenea opțiunile de a adăuga acea operă la favorite și de a posta un comentariu cu impresia lor.
- d) Separat, se poate vizualiza pagina de opere favorite, de unde, prin extensie, se pot elimina elemente din lista de favorite.

2. Administratorul aplicației – persoana din cadrul muzeului care se ocupă de buna funcționare a aplicației și de actualizarea datelor. Acesta are nevoie de următoarele funcționalități suplimentare:

- a) Gestiunea bazei de date cu opere de artă, direct din interfața aplicației. Acest ansamblu de operațiuni este detaliat mai târziu în *Figura 4* și implică de asemenea autentificarea în

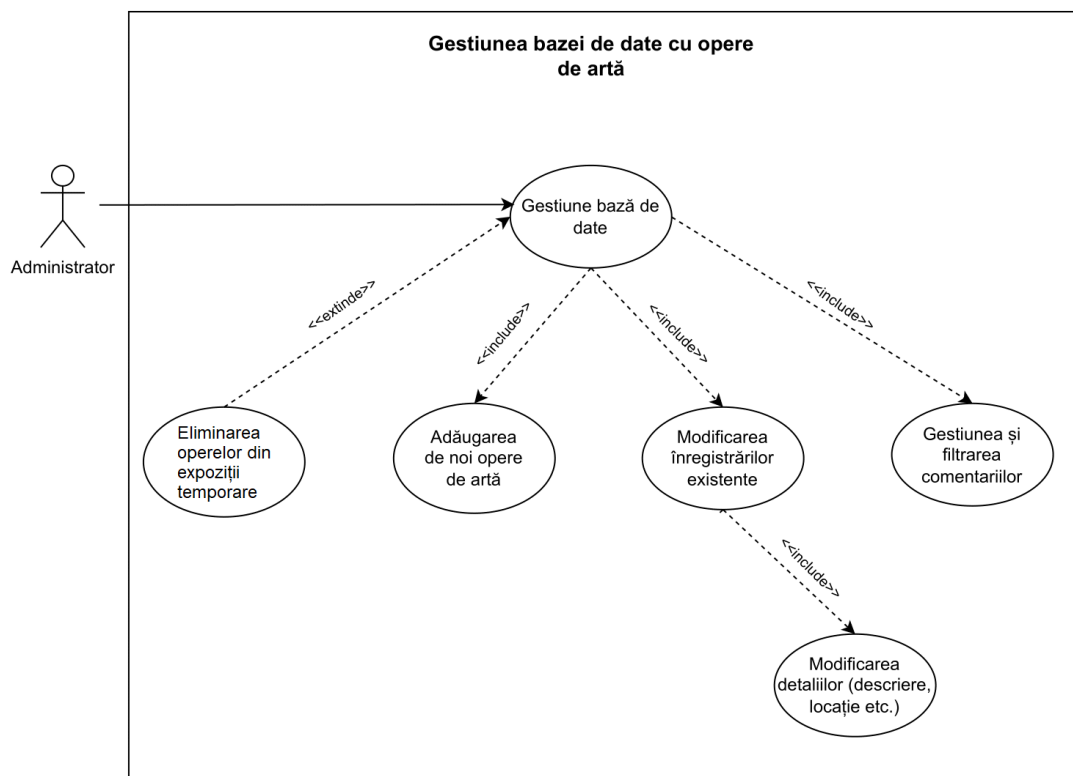
prealabil. Implicit, contul cu rol de administrator extinde funcționalitățile contului de utilizator simplu, având dreptul de a accesa paginile operelor de artă.

- b) Identificarea și soluționarea promptă a eventualelor erori. Pentru aceasta, aplicația trebuie să includă logarea erorilor, metode de gestiune a excepțiilor etc.
3. Specialistul IT care, la nevoie, colaborează cu administratorul pentru a corecta eventualele erori și a îmbunătăți performanța aplicației.



Figură 4. Modelul Use-Case general

După cum am menționat, vom analiza în detaliu cazul de utilizare al administratorului de sistem care dorește să realizeze operații tip CRUD asupra bazei cu opere de artă.



Figură 5. Modelul Use-Case particular

Cea de-a doua diagramă Use-Case (*Figura 5*) prezintă cazuri concrete în care administratorul ar dori să intervină asupra datelor stocate despre colecția muzeului. Pentru a facilita acest proces, contul cu drept de administrator are acces la următoarele operațiuni:

1. Adăugarea de noi opere de artă, caz în care se completează câmpuri tip input cu toate datele necesare, precum titlu, artist, dată, explicații, imagine etc., apoi se declanșează adăugarea unei noi înregistrări în baza de date prin metoda de submit.
2. Modificarea înregistrărilor existente – administratorul poate alege să schimbe datele uneia dintre operele de artă deja prezente în baza de date, actualizându-i, spre exemplu, locația, în cazul în care a fost mutată într-o altă sală.
3. Gestiunea și filtrarea comentariilor – administratorul are dreptul de a șterge anumite comentarii care încalcă regulile de bună purtare.
4. Eliminarea operelor din expozițiile temporare, întrucât, în momentul în care expoziția temporară se încheie, expodatele vor dispărea și din aplicație.

2.3 Documentarea cerințelor

După cum am observat până acum, cerințele informaționale sunt destul de diverse, astfel încât pot fi clasificate în cerințe funcționale – reflectă procesele pe care sistemul trebuie să le realizeze – și cerințe non-funcționale, axate pe proprietățile sistemului și pe caracteristicile tehnice (Oprea, 2005).

Urmărind această taxonomie, putem încadra și cerințele principale ale sistemului nostru astfel:

- Cerințe funcționale: vizualizarea informațiilor despre muzee și galerii, accesarea informațiilor despre operele de artă, adăugarea la favorite, adăugarea de comentarii, gestiunea bazei de date prin adăugarea, modificarea sau ștergerea operelor de artă și a datelor despre muzee și galerii.
- Cerințe non-funcționale: securitatea și confidențialitatea datelor, interfața ușor de utilizat, timp de răspuns prompt.

Pe baza acestor elemente, putem realiza mai departe diagramele de documentare a cerințelor, mai exact diagrama de activități, urmată de diagrama de stări.

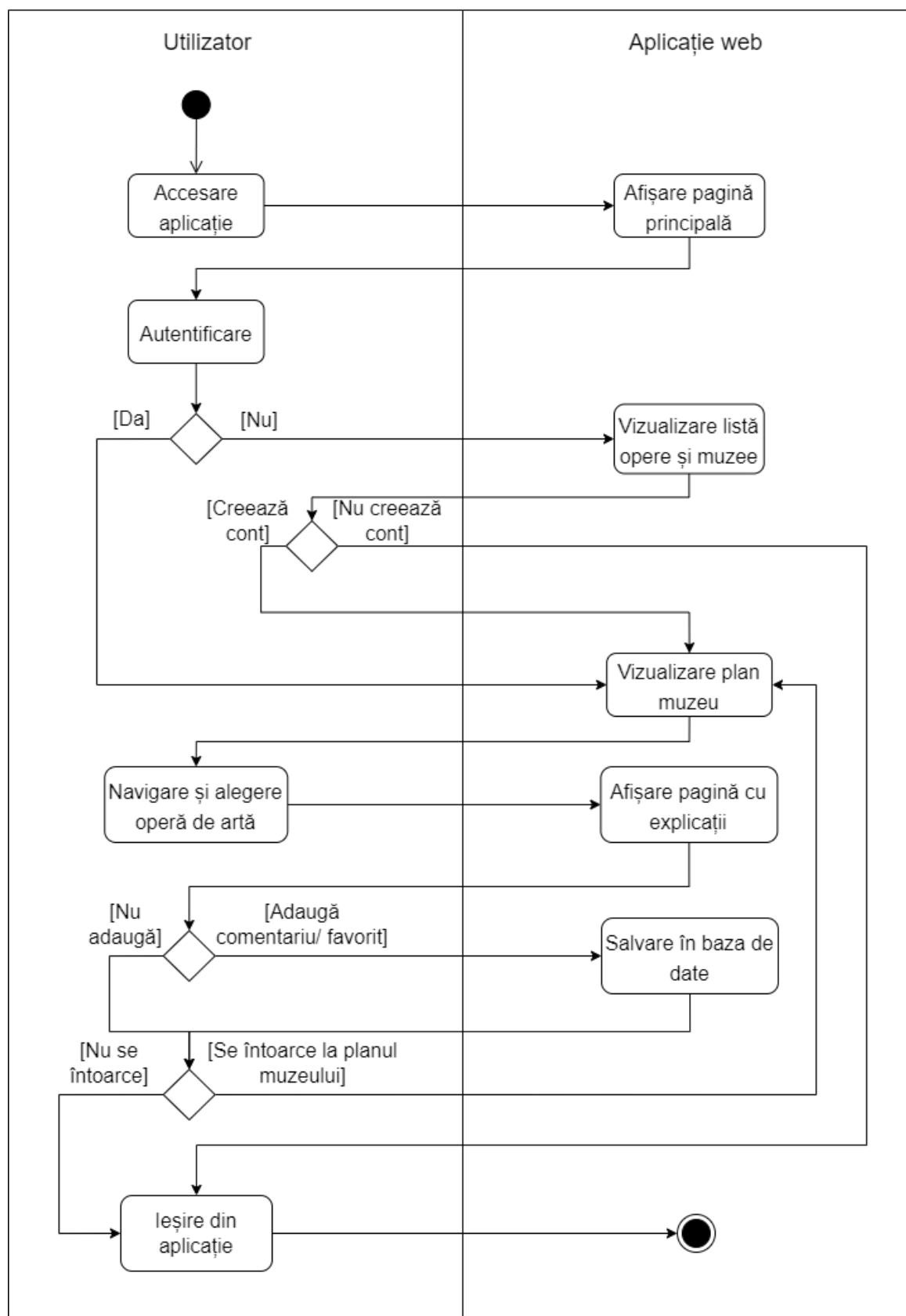
2.3.1 Diagrama de activități

Diagrama prezintă fluxul de activitate al actorului-utilizator în cadrul aplicației web. Acesta declanșează procesul prin accesarea aplicației, eveniment ce conduce la afișarea paginii principale (*homepage*).

Dacă se are în vedere un simplu utilizator-vizitator, acesta poate vizualiza doar lista cu muzeele și operele de artă. În urma creării unui cont, userul va avea acces la planul muzeului, de unde poate naviga prin diversele liste cu galeriile și camerele muzeului, putând alege o anumită operă de artă. În acel moment, se deschide o nouă pagină cu imaginea operei și cu explicații detaliate despre aceasta.

Pe această pagină (a operei de artă), utilizatorul autentificat are posibilitatea de a adăuga un comentariu și de a adăuga opera de artă la favorite, ceea ce declanșează operația de salvare în baza de date. În continuare, utilizatorul se poate întoarce la planul muzeului, pentru a vizualiza un alt element din colecția muzeului, iar procesul descris mai sus se reia. În caz contrar, utilizatorul iese din aplicație, iar procesul se încheie.

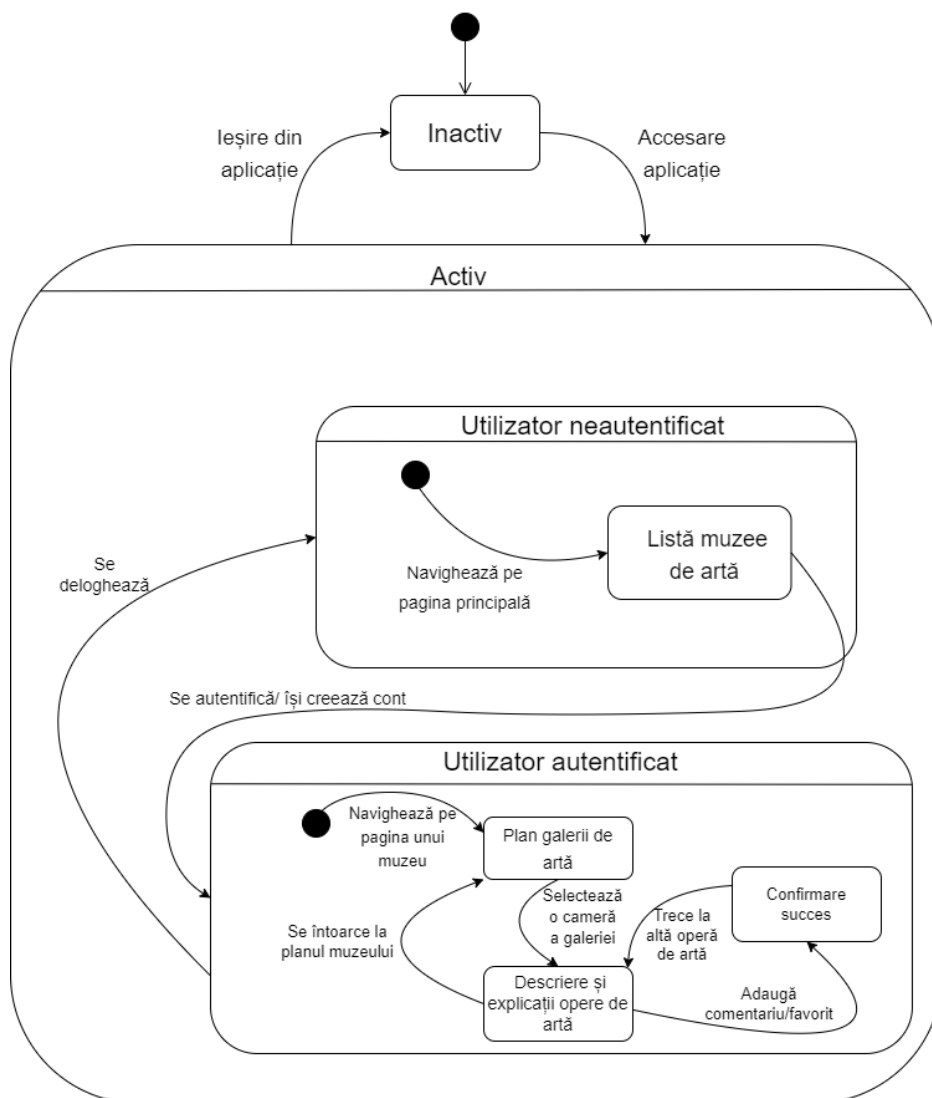
Diagrama de activități
Aplicație de ghidaj în muzeele de artă



Figură 6. Diagrama de activități

2.3.2 Diagrama de stări

În această diagramă (*Figura 7*) se prezintă principalele stări și evenimente de tranziție din cadrul aplicației web, după cum urmează:



Figură 7. Diagrama de stări

1. Starea de inactivitate, întreruptă prin accesarea aplicației și declanșată de ieșirea din aplicație.
2. Starea activă de funcționare a aplicației, ce cuprinde două stări esențiale:
 - a) Utilizator neautentificat – în această situație, aplicația oferă doar posibilitatea de naviga de pe pagina principală către lista cu muzee de artă și operele acestora. În momentul în care vizitatorul se loghează sau își creează un cont, se va face trecerea în următoarea stare de ansamblu a aplicației:
 - b) Utilizator autentificat – aici, actorul poate naviga pe pagina unui anumit muzeu pentru a-i vedea planul galeriilor și al sălilor. De acolo, se poate selecta o anumită cameră, moment în

care se va afișa pagina cu detalii și explicații despre fiecare operă din acea cameră. În cadrul acestei pagini, utilizatorul are opțiunea de a posta un comentariu sau de a adăuga opera respectivă la favorite, caz în care se face salvarea în baza de date, urmată de o confirmare.

Tot de aici, utilizatorul se va putea întoarce la starea precedentă, anume la vizualizarea planului muzeului, pentru a selecta o următoare opera de artă, pe măsură ce avansează prin muzeu.

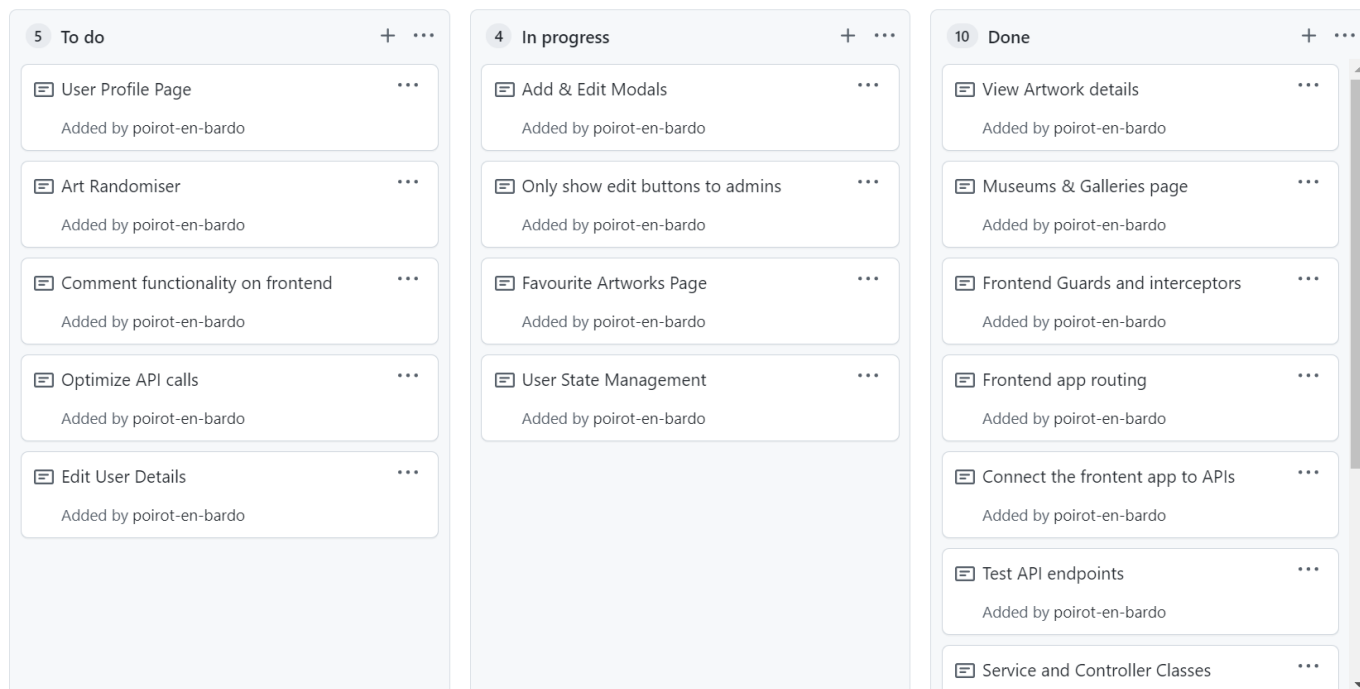
Prin delogarea utilizatorului, aplicația revine la starea de la punctul a) și se revocă drepturile de vizualizare a planului muzeului, a paginilor de explicații și a listei de opere favorite.

2.4 Model de dezvoltare

Modelul de dezvoltare software preferat pentru acest proiect este reprezentat de sistemul Kanban, din cadrul metodologiei Agile.

În varianta sa clasică, sistemul Kanban presupune un panou (*board*) separat în 3 secțiuni principale: *To Do* (de făcut), *In Progress* (sarcini în progress sau în desfășurare) și *Done* (activități terminate). Scopul său este să optimizeze fluxul de lucru, să țină evidența progresului și să minimizeze întârzierile (Ahmad et al., 2013).

În acest caz, am utilizat funcționalitatea *Projects* disponibilă pe GitHub, unde se pot realiza și completa șabloane tip Kanban, sub forma din *Figura 8*:



Figură 8. Panoul de dezvoltare Kanban

Fiecare subobiectiv din coloana din stânga trece în coloana centrală în momentul începerii lucrului, iar apoi în coloana din dreapta la finalizarea sa.

3. Proiectarea sistemului informatic

Până în acest punct, ne-am concentrat pe analiza sistemului informatic și pe extragerea caracteristicilor și a funcționalităților dorite. Toate acestea vor deveni subiectul demersului de proiectare, care constă în definirea structurii aplicației și a bazei de date, elaborarea planului arhitecturii de sistem și stabilirea algoritmilor și a tehnologiilor cele mai potrivite pentru implementarea care va avea loc ulterior.

3.1. Proiectarea logică

Pentru a putea înțelege mai bine modul în care circulă datele în aplicație prin comunicarea și interacțiunea frontend – backend – baza de date și retur, le putem vizualiza sub forma unui flux permanent, cât timp aplicația funcționează. Acest flux este declanșat de către unul dintre actori (utilizator sau administrator) și poate fi observat în *Figura 9*, care constituie diagrama de flux de date.

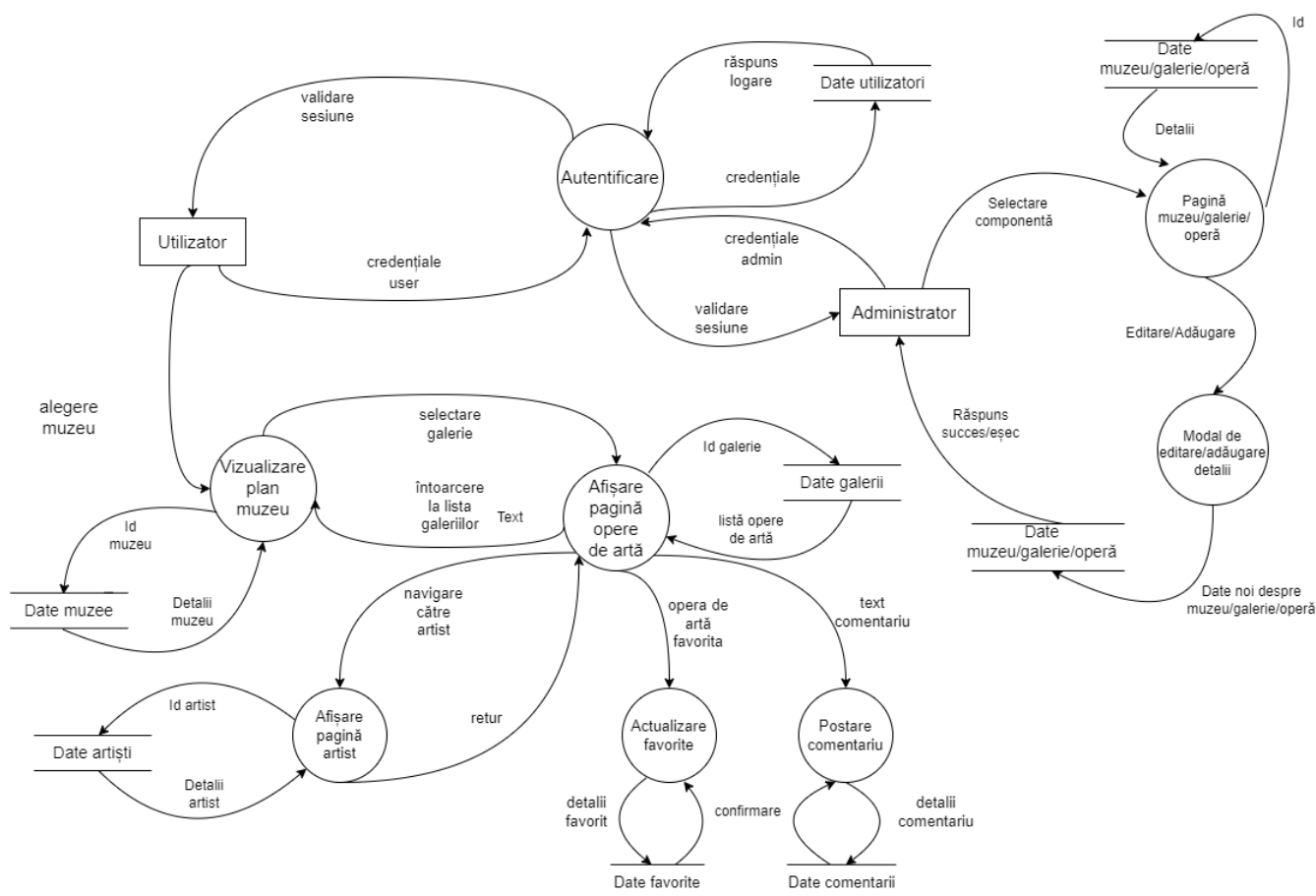
În cadrul diagramei, *Autentificarea* reprezintă un proces comun, indiferent de tipul utilizatorului, iar aceasta se desfășoară sub forma următoarei secvențe: se primesc credențialele introduse de utilizator (emailul și parola sau datele de creare a unui nou cont), acestea sunt transmise mai departe către componenta de logare, ce comunică cu baza de date; se realizează verificarea în urma căreia se primește un răspuns afirmativ sau negativ, iar în funcție de acesta se returnează o sesiune de logare validă sau un mesaj de eroare.

Mai departe, utilizatorul logat obișnuit (vizitator al muzeului) poate declanșa navigarea prin colecția muzeului, care cuprinde următoarele subproces:

- *Vizualizarea planului unui muzeu* ales din listă – context ce permite selectarea galeriei, camerei și a operei de artă; în acest context, se face un apel către baza de date cu id-ul muzeului și se primesc înapoi detaliile aferente;
- *Afișarea paginii operei de artă* – cu imagine, detalii și descriere, de asemenea primite în urma unui apel către baza cu datele galeriilor; aici există și posibilitatea întoarcerii la planul muzeului;
- *Afișarea paginii despre artist*, prin navigare de pe pagina unei opere de artă și retur, cu un apel către baza de date despre artiști, prin intermediul id-ului artistului căutat;

- *Adăugarea unei opere de artă la favorite*, ce declanșează inserarea unei noi înregistrări în baza de date, urmată de un răspuns de confirmare;
- *Postarea unui comentariu* pe pagina operei de artă, care va fi salvat implicit și în baza de date.

În completarea funcționalităților descrise mai sus, utilizatorul care are și rol de administrator poate efectua și operații de adăugare, actualizare și ștergere asupra componentelor aferente muzeului, anume galeriei, camere și opere de artă.



Figură 9. Diagrama de flux de date

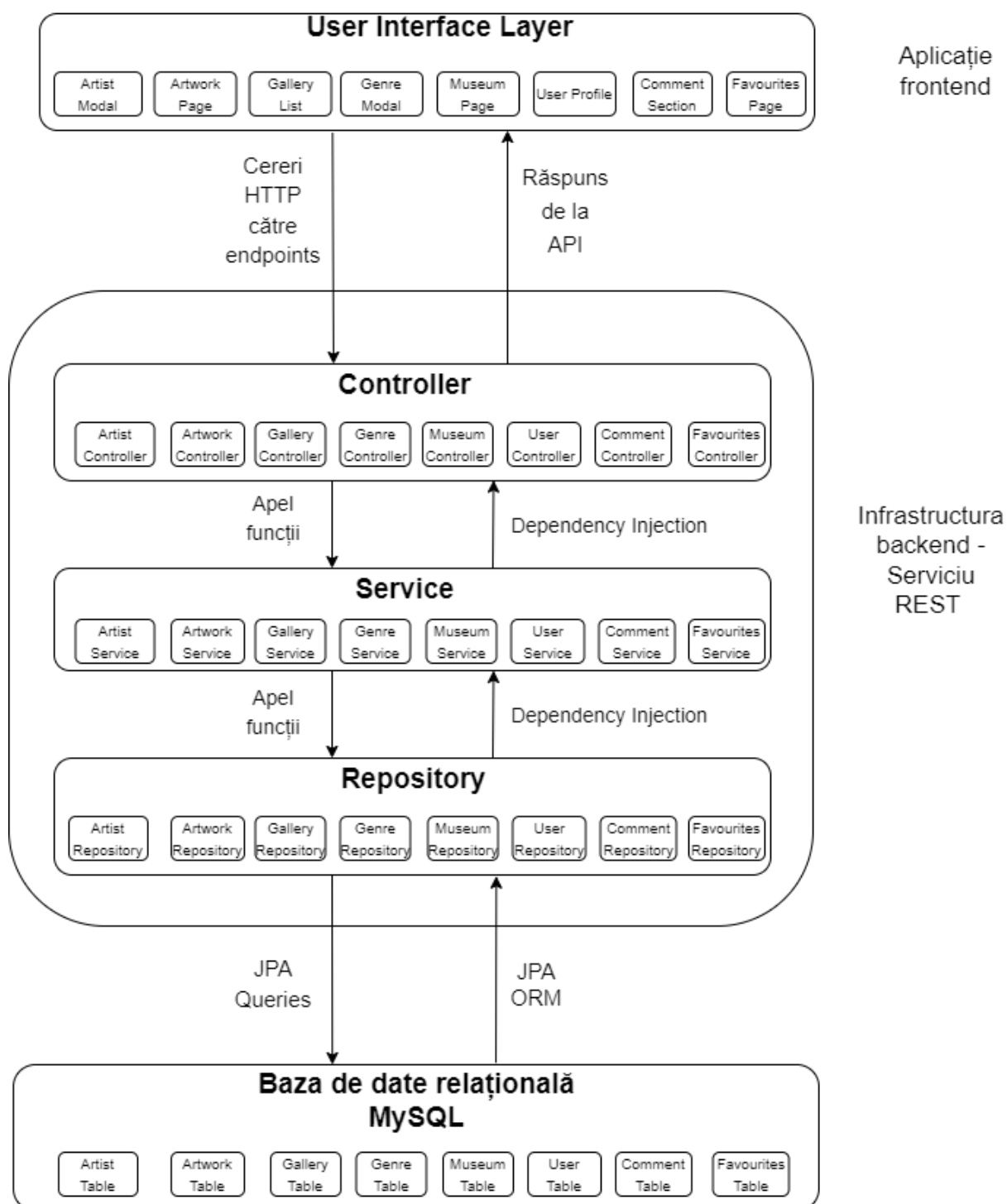
Pentru operațiile CRUD, administratorul începe prin navigarea pe pagina componentei ce se vrea a fi modificată (pe care se vor încărca detaliile obținute din baza de date), apoi selectează fie editare, fie adăugare înregistrare nouă, moment în care se deschide o fereastră tip modal în care să introducă noile date. Acestea vor fi trimise, la salvare (*submit*), mai departe către baza de date și se va returna un răspuns de succes sau de eșec.

3.1.1 Arhitectura sistemului

Urmând modelul arhitecturii cu mai multe straturi (n-tier/multilayer), aplicația este structurată pe 3 niveluri principale, după cum urmează (*Figura 10*):

1. User Interface Layer – reprezentat de aplicația frontend (în Angular și TypeScript), ce gestionează interacțiunea utilizatorilor cu aplicația și comunică mai departe cu serviciul de backend. Aceasta conține componente de vizualizare pentru fiecare modul important al aplicației.
2. Spring REST API – serviciul REST de backend, format din următoarele elemente:
 - Controller (Presentation Layer), care primește cererile HTTP din componenta frontend, trimite parametrii pentru prelucrare mai departe spre Service (interfața service este injectată în controller), iar la final returnează răspunsul API către frontend. Precum fiecare nivel din arhitectura de backend, pachetul controller cuprinde clase particulare pentru fiecare componentă distinctă din aplicație, dintre care cele mai importante sunt: Artist, Artwork, Gallery, Genre, Museum, User, Comment, Favourites.
 - Service (Business Logic Layer), care acționează drept intermediar și separator între Controller și Repository. Aici se fac mapările de date (entități către DTO <<Data Transfer Object>> și invers), se realizează eventuale prelucrări și se apelează funcțiile din Repository prin injectarea interfeței respective.
 - Repository (Persistence Layer) conține maparea propriu zisă (Object-Relational Modelling) a entităților din baza de date în clase Java cu toate atributele conținute și în tabele. Acesta folosește tehnologia JPA (Java Persistence API) pentru a interoga și modifica baza de date.
3. Baza de date relațională realizată folosind sistemul MySQL. Aici se salvează toate datele necesare în tabelele aferente fiecărei entități menționate mai sus: Artist, Artwork, Gallery, Genre, Museum, User, Comment, Favourites, la care se adaugă tabele de intersecție și tabele aferente componentei de securitate. Acestea vor fi detaliate în secțiunea 2.2.1 *Structura fizică a datelor*.

Arhitectura sistemului multilayer



Figură 10. Arhitectura sistemului multi-layer

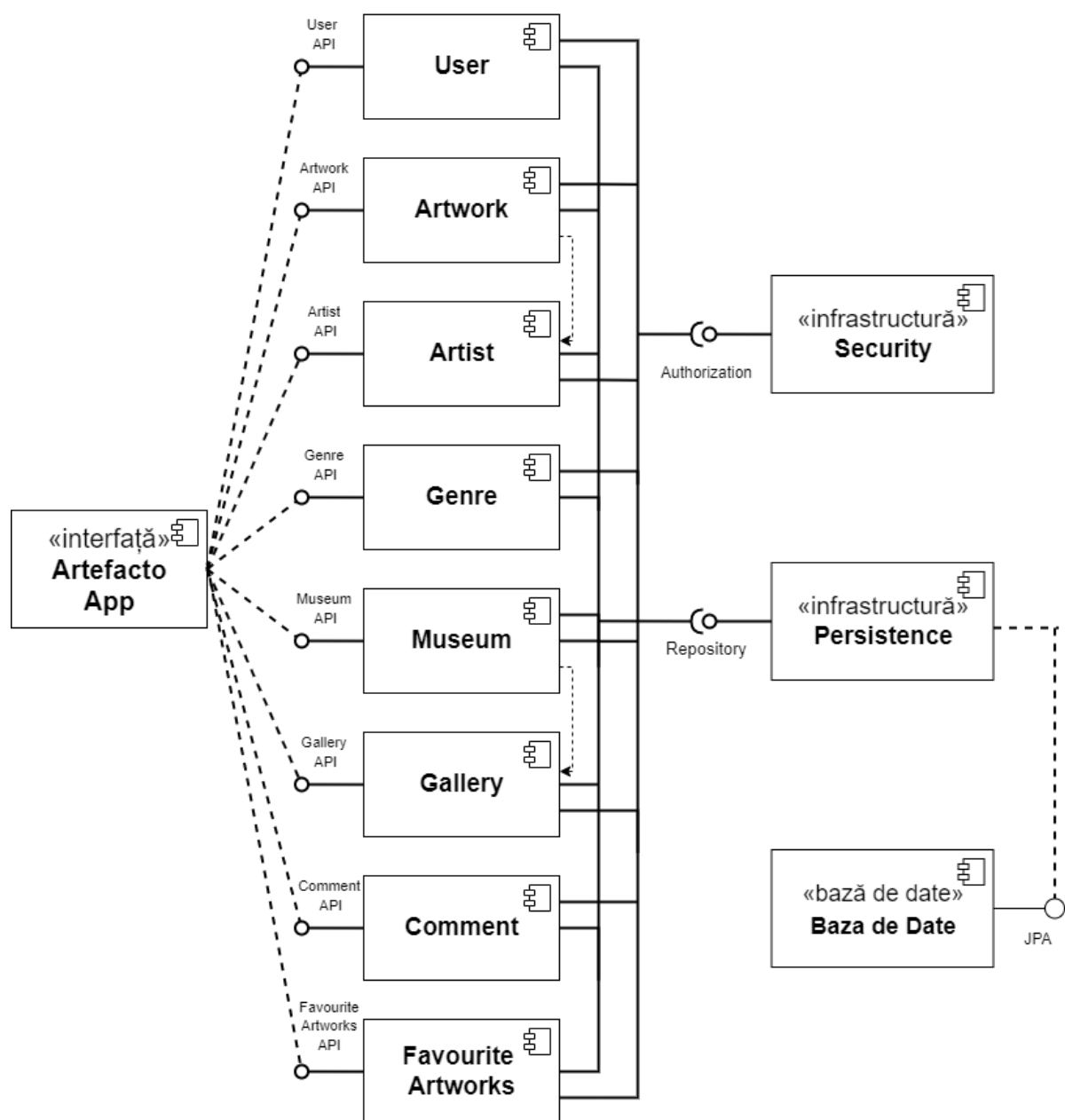
Pentru o altă perspectivă asupra modului în care elementele de mai sus comunică între ele și interacționează cu sistemul de securitate, putem face referință la *Figura 11*. Această diagramă se axează pe reprezentarea componentelor sistemului aplicației, după cum urmează:

1. Baza de date relațională de tip MySQL, care stochează datele despre toate entitățile necesare aplicației, atât cele prezente mai departe în;
2. Persistența, element de infrastructură ce comunică prin intermediul interfeței JPA (Java Programming API) cu baza de date și realizează operațiile CRUD. De asemenea, stratul de persistență este accesibil mai departe tuturor componentelor tip Service prin interfețele tip Repository;
3. Securitatea, o altă componentă de infrastructură ce realizează operațiunile de autentificare și autorizare. Aceasta comunică, de asemenea, cu următoarele componente prin filtrul de autorizare;

Urmează componentele tip controller de la punctele 4-11, care au și câte un REST endpoint aferent pentru gestionarea cererilor tip HTTP:

4. User – cuprinde funcționalitatea de creare cont, logare și gestiunea utilizatorilor;
5. Artwork – vizează gestiunea operelor de artă;
6. Artist – se ocupă cu evidența artiștilor ale căror opere se află în muzee;
7. Genre – se ocupă cu gestiunea genurilor și curentelor artistice;
8. Museum – componenta de gestiune a muzeelor de artă;
9. Gallery – realizează gestiunea galeriilor fiecărui muzeu;
10. Comment – are în vedere gestiunea comentariilor adăugate de utilizatori pe paginile operelor de artă;
11. Favourite Artworks – vizează gestiunea operelor salvate de fiecare utilizator la favorite;
12. Interfața de interacțiune cu utilizatorul (Artefacto App UI) care comunică prin API-uri cu celelalte componente definitorii pentru arhitectura de backend.

Diagrama de componente



Figură 11. Diagrama de componente

3.1.2 Baza informațională

Aplicația este menită să gestioneze fluxuri complexe de date care, în anumite cazuri, sunt stocate în forma inițială în care sunt obținute, în timp ce în alte cazuri sunt prelucrate și apoi sunt inserate în baza de date.

Putem spune că baza informațională cuprinde toate aceste categorii de date primare, intermediare și finale, care vor fi transformate ulterior în entități de sine-stătătoare, atribute și relații, redate în detaliu în subcapitolul următor – 3.2 *Proiectarea tehnică*.

Având în vedere că domeniul de activitate al aplicației este cel al artelor vizuale, se deduce faptul că sistemul lucrează cu date, informații și trăsături ce caracterizează operele de artă. Cele mai notabile dintre acestea ar fi: titlul unei opere de artă, anul în care a fost realizată, curentul artistic și descrierea sa; pe lângă toate datele de tip text, este foarte importantă și imaginea operei de artă, care va fi afișată în aplicație alături de explicații. În relație directă cu opera de artă se află și artistul, caracterizat și acesta de date precum anii în care a trăit, naționalitatea și biografia. Operele de artă sunt găzduite de muzee, iar aici intervine o altă suită de informații de bază despre acestea și despre modul în care sunt structurate în galerii, etaje, camere și modul în care operele de artă sunt amplasate în acestea.

De obicei, galeriile muzeelor de artă sunt clasificate în funcție de curentul artistic (spre exemplu artă barocă, realism, expresionism, artă abstractă), de perioadă (artă primitivă, de ev mediu, renașcentistă, contemporană), sau de o anumită cultură sau civilizație (artă egipteană, asiatică, flamandă etc.). Pornind de la această segmentare, este important și modul în care sunt plasate operele de artă în cadrul fiecărei galerii, pentru a putea realiza o potrivire între ordinea acestora din aplicație și configurația unei săli dintr-o galerie.

Trecând la segmentul de gestiune a utilizatorilor, este evident că la crearea unui cont aceștia vor furniza un email și o parolă, date la care se adaugă informații precum nume și prenume. Din considerente de securitate, această parolă trebuie criptată, astfel că în baza de date va ajunge doar un șir de caractere imposibil de descifrat în sens invers. Prin activitatea lor în cadrul aplicației, utilizatorii generează înregistrări de comentarii și de opere de artă favorite. Ei vor putea vedea și comentariile altor utilizatori, însă lista de favorite este privată pentru fiecare în parte.

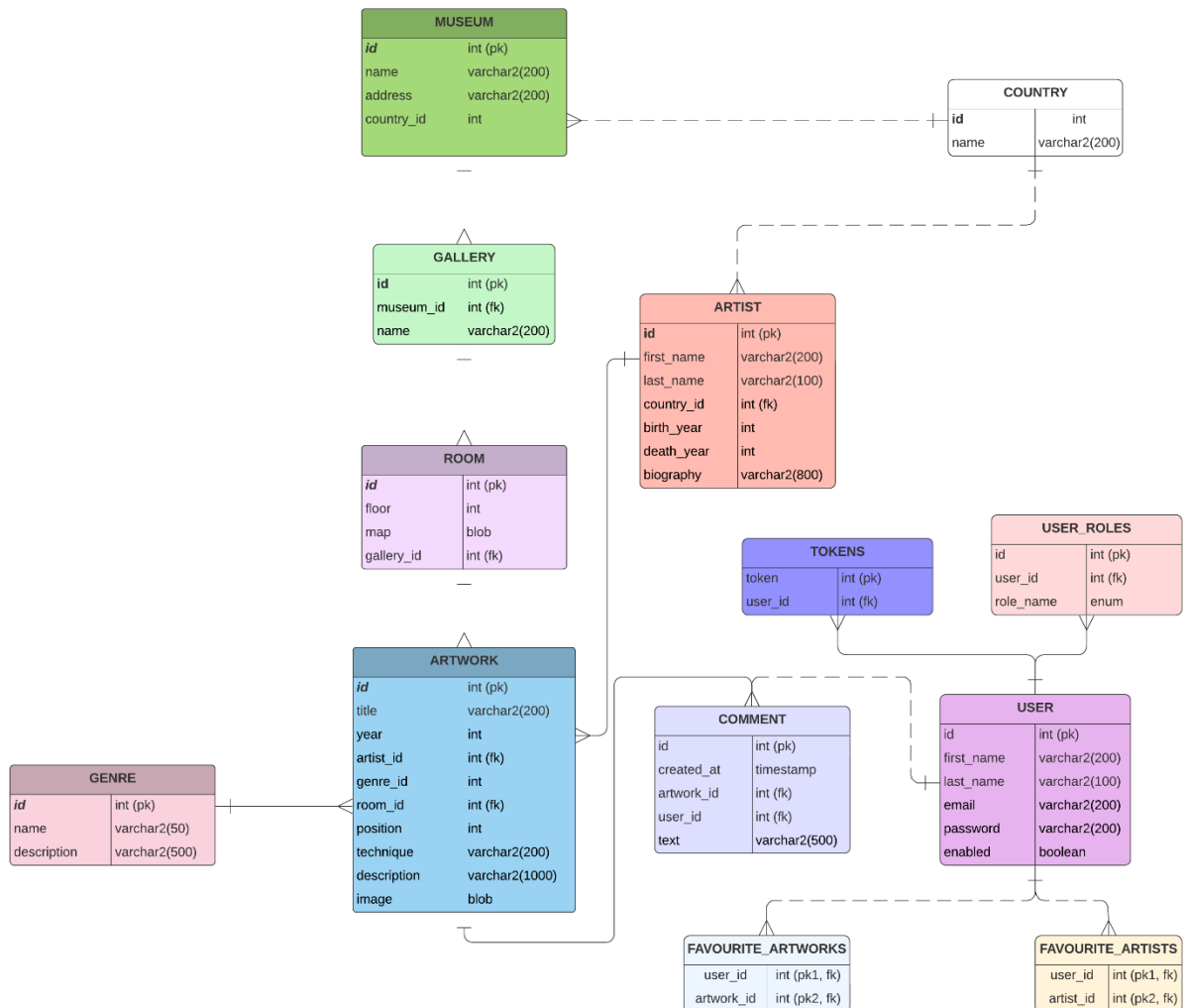
Toate aceste date sunt disponibile și pentru administratori, o altă categorie de *stakeholderi*, care vor avea și dreptul de a le edita (exemplu modificarea amplasării operei într-o altă sală) sau de a adăuga noi înregistrări (precum adăugarea unei noi galerii).

3.2. Proiectarea tehnică

Acest subcapitol urmărește să surprindă ultimul pas care precede implementarea, anume stabilirea concretă a structurii fizice a entităților din baza de date, a claselor de obiecte ce le corespund, a algoritmilor fundamentali în realizarea aplicației și a tehnologiilor utilizate pentru implementarea acestora.

3.2.1 Structura fizică a datelor

Baza de date relațională este formată din entități care înglobează în mod structurat datele prezentate în cadrul subcapitolului despre baza informațională (3.1.2), în urma procesului de filtrare, prelucrare și curățare.



Figură 12. Diagrama Entitate-Relație a bazei de date

Diagrama entitate-relație (*Figura 12*) surprinde structura bazei de date, formată din următoarele tabele și relații:

1. Country – tabela de evidență a țărilor, cu attributele: id (int), name (varchar).
2. Museum – tabela ce cuprinde muzeele disponibile în aplicație, cu attributele: id (int), name (varchar), address (varchar), country_id (int) cheie străină către tabela Country.
3. Gallery – tabela galeriilor de artă din fiecare muzeu, cu attributele: id (int), name (varchar), museum_id (int) cheie străină către Museum.
4. Room – tabela sălilor din fiecare galerie, cu attributele: id (int), floor (int), map (blob) – harta sălii, dacă există și gallery_id – cheie străină spre Galler.
5. Artist – tabela artiștilor, cu attributele: id, first_name, last_name, birth_year, death_year, biography, country_id cheie străină către Country.
6. Genre – tabela cu curentelor artistice, cu attributele: id, name, description.
7. Artwork – tabela operelor de artă, cu attributele: id, title, year, position (indexul din camera în care se află), technique (stilul în care e realizat exponatul, spre exemplu ulei pe pânză, acrilic, argilă, sculptură în marmură etc.), description (explicațiile despre operă), image_path (fisierul cu imaginea), artist_id cheie străină spre Artist, genre_id cheie străină către Genre, room_id (cheie străină către Room).
8. User – tabela utilizatorilor, cu attributele: id, first_name, last_name, email, password, și *enabled* (boolean) – atribut ce indică dacă acest cont de utilizator e activat sau nu; în mod implicit, atributul este setat pe *True*.
9. User_roles – tabela care stochează rolurile utilizatorilor, cu attributele: id, user_id (cheie străină către User) și role_name (un termen din enumerația {'ROLE_USER', 'ROLE_ADMIN'}). Această tabelă este necesară în eventualitatea în care dorim ca un cont să dețină atât rol de user, cât și de administrator.
10. Tokens – tabela ce ține evidența identificatorilor sesiunilor active ale utilizatorilor; aceasta conține câmpurile token (un string unic ce are și rol de cheie primară) și user_id (cheie străină către User).
11. Comment – tabela comentariilor, cu attributele: id, created_at (timestamp cu momentul în care a fost creat comentariul), text (conținutul comentariului), artwork_id – cheie străină către Artwork, user_id – cheie străină către User.
12. Favourite Artworks – tabela operelor de artă salvate la favorite de un utilizator, cu attributele: user_id – cheie străină spre User, artwork_id – cheie străină spre Artwork – acestea două constituie o cheie primară compusă.

13. Favourite Artists – tabela artiștilor salvați ca favoriți de un utilizator, cu atributele: `user_id` – cheie străină spre User, `artist_id` – cheie străină spre Artist – formează o cheie primară compusă, la fel ca la Favourite_Artworks.

3.2.2 Procese și algoritmi

Logica pe care se bazează procesul de comunicare între componenta backend și baza de date presupune tehnica ORM (*Object–relational mapping*, adică reprezentarea obiectual-relațională).

Acest sistem presupune crearea unor clase care să aibă o structură corespunzătoare entităților din baza de date. În consecință, fiecare câmp din baza de date va fi reprezentat de un atribut al clasei aferente, cu tip de date similar, spre exemplu: `id` – INT și `int`, `nume` – VARCHAR și String, `image` – BLOB și `byte[]` (array de biți).

Hibernate ORM, un framework Java propriu tehnologiei Spring descrise în subcapitolul următor, va asigura corelarea datelor și transferul corespunzător din entități în obiecte și invers.

Un alt proces important este reprezentat de maparea DAO – DTO. DAO (Data Access Object) reprezintă acele obiecte prezentate mai sus, care corespund entităților din baza de date. În schimb, DTO (Data Transfer Object) este un obiect care preia din DAO doar acele atribute necesare pentru a fi transferate mai departe prin microserviciu. Spre exemplu, DAO User va conține implicit și un atribut *password*, însă acesta nu va fi trimis niciodată prin *endpointul API*, întrucât înaintea trimerii datelor, acestea vor fi mapate pe DTO User, obiect care nu conține câmpul *password* și care poate fi transferat în siguranță.

În acest context, un algoritm important este cel de criptare a parolelor, concretizat în Java Spring Security prin clasa *BCryptPasswordEncoder*, care conține o funcție de hash într-un singur sens (*encode*) ce folosește și un *salt* (un șir aleatoriu de biți) pentru a crea un hash unic pentru fiecare parolă. Astfel, chiar dacă doi utilizatori au aceeași parolă, acestea nu vor avea același hash, fapt ce previne atacurile în masă (O’Neil, 2017).

Mai departe, algoritmul de securitate Spring JWT (JSON Web Token) presupune preluarea credențialelor de autentificare și generarea unui *token* (un șir de caractere cu rol de cheie) în cazul în care acestea sunt valide. Pe baza acestui token, aplicația va realiza mai departe autorizarea, permițând sau respingând anumite cereri în funcție de rolurile pe care le are acel utilizator (user sau admin).

Nu în ultimul rând, respingerea cererilor se realizează prin aruncarea unor excepții personalizate (*custom exceptions*), care sunt apoi gestionate de un controller special ce extinde clasa *ResponseEntityExceptionHandler*. Acesta se asigură că rularea aplicației nu este

întreruptă la întâlnirea excepțiilor, returnându-se în schimb mesaje de eroare sub formă de *ResponseEntity*, însoțite de un cod standard al erorii (*400 Bad Request*, *401 Unauthorized*, *403 Forbidden* etc.).

3.2.3 Tehnologii specifice

Fiind un proiect *full-stack* (care implică atât componenta backend, cât și componenta frontend), ne sunt necesare mai multe limbaje de programare și mai multe tehnologii suport. În acest context, aplicația se bazează pe următoarele instrumente, programe și tehnologii:

- Spring Boot pentru componenta backend – un *framework* Java care facilitează dezvoltarea aplicațiilor web prin crearea de microservicii (module cu funcție unică, responsabile de transferul datelor prin protocolul HTTP). Acesta se bazează pe 2 *design pattern-uri* importante: *Dependency Injection*, prin care Spring injectează obiectele fără a fi nevoie să le creăm noi, și *Inversion of Control*, un *container* care gestionează întreg ciclul de viață al obiectelor – noi folosim librăriile și dăm obiectelor un anumit comportament (IBM, 2020). În cadrul acestui *framework*, se folosesc și librăriile Lombok, care generează șabloane de cod precum *getter*, *setter*, constructori și MapStruct, care realizează maparea entitate – DTO (<https://spring.io/projects/spring-boot>, <https://projectlombok.org/>, <https://mapstruct.org/>).
- Maven – instrument pentru build, anume convertirea codului sursă într-o formă ce poate fi rulată (Verma, 2020) și gestiunea dependențelor: <https://maven.apache.org/>.
- Angular 13 pentru componenta frontend – *framework* TypeScript, pe baza NodeJS 16.14.2: <https://angular.io/>, alături de HTML5 și CSS. Angular se bazează pe module, componente și programare asincronă și este utilizat pentru crearea de *Single Page Applications* (Aplicații web cu pagină unică), adică aplicații care rulează în *browser* și fac schimb de date fără a fi nevoie să se reîncarce paginile (Khezami, 2020).
- Bootstrap – *framework* CSS utilizat din considerente estetice și pentru a asigura adaptabilitatea aplicației indiferent de tipul ecranului (responsiveness) <https://getbootstrap.com/>
- Limbajul SQL și software-ul MySQL Workbench pentru gestiunea bazei de date: <https://www.mysql.com/products/workbench/>
- IntelliJ IDEA drept mediu de implementare și dezvoltare: <https://www.jetbrains.com/idea/>
- Suita Swagger pentru documentarea API-ului – Application Programming Interface, adică un intermediar care permite comunicarea între mai multe aplicații individuale (Freeman,

2019) – și evidența endpoint-urilor, care sunt căile de acces prin care API-ul trimite și primește date (Juviler, n.d.) : <https://swagger.io/>

- Aplicația Postman pentru testarea API-ului prin cereri HTTP tip GET, PUT, POST, DELETE: <https://www.postman.com/>
- Programul Zotero pentru evidența referințelor și sistemul de citare APA 7th: <https://www.zotero.org/>
- Microsoft Word, Draw.io și LucidChart pentru redactarea lucrării și realizarea diagramelor: <https://app.diagrams.net/>, <https://www.lucidchart.com/pages/>
- Instrumentele Git și GitHub pentru controlul versiunilor aplicației (*version control system*): <https://git-scm.com/>, <https://github.com/>.

4. Implementare

Implementarea sistemului este realizată în vederea separării celor 2 aplicații individuale – backend Java Spring și frontend Angular. Astfel, aplicația Spring mapează baza de date MySQL și oferă un set de endpointuri care sunt apoi apelate de aplicația frontend pentru extracția și afișarea datelor existente, respectiv trimiterea datelor de adăugat sau modificat.

În acest context, capitolul curent este segmentat în 3 mari subcapitole care descriu implementarea bazei de date în MySQL, a microserviciilor în Spring și a interfeței în Angular.

4.1 Implementarea bazei de date

Baza de date este implementată utilizând MySQL Workbench, prin intermediul scripturilor SQL de creare de tabele.

După crearea schemei intitulate *ART*, am rulat comenzile pentru crearea fiecărui tabel. Pentru referință, întrucât aceste scripturi au o sintaxă similară, observăm aici codul pentru crearea entității *ARTWORK*, ce conține câmpurile prezentate anterior în capitolul 3.2.1
Structura fizică a datelor:

```
USE ART;
```

```
CREATE TABLE ARTWORK (  
    ID INT PRIMARY KEY AUTO_INCREMENT,  
    TITLE VARCHAR(200) NOT NULL,  
    YEAR INT NOT NULL,  
    POSITION INT,  
    TECHNIQUE VARCHAR(200),  
    DESCRIPTION VARCHAR(1000),  
    IMAGE_PATH BLOB,  
    ARTIST_ID INT,  
    GENRE_ID INT,  
    ROOM_ID INT,  
    CONSTRAINT artwork_artist_id_fk FOREIGN KEY (ARTIST_ID) REFERENCES  
    ARTIST(ID),  
    CONSTRAINT artwork_genre_id_fk FOREIGN KEY (GENRE_ID) REFERENCES  
    GENRE(ID),  
    CONSTRAINT artwork_room_id_fk FOREIGN KEY (ROOM_ID) REFERENCES  
    ROOM(ID));
```


În continuare, are loc adăugarea înregistrărilor în fiecare tabel folosind comanda INSERT astfel:

```
USE ART;
```

```
INSERT INTO ARTWORK
```

```
VALUES (
```

```
'The School of Athens', 1511, 1, 'fresco',
```

```
'This painting represents all the greatest mathematicians, philosophers and scientists from  
classical antiquity gathered together sharing their ideas and learning from each other. These  
figures all lived at different times, but here they are gathered together under one roof.',  
LOAD_FILE('athens.jpg'), 8, 1, 4);
```

În acest demers, am constatat că imaginea nu se putea încărca, întrucât fișierul depășea limitele tipului BLOB, așa că este necesar să schimbăm tipul câmpului image_path în MEDIUMBLOB:

```
USE ART;
```

```
ALTER TABLE ARTWORK
```

```
MODIFY COLUMN IMAGE_PATH MEDIUMBLOB;
```

Se repetă asemenea operații de creare, inserare și editare, până ce se obține structura finală a bazei de date:



Figură 13. Structura bazei de date în MySQL

4.2 Implementarea aplicației backend

După cum am menționat anterior, aplicația backend este implementată utilizând Java Spring Boot și mediul de lucru IntelliJIDEA.

Arhitectura multi-layer presupune existența claselor tip entitate (DAO), DTO, interfețe de tip persistență (Repository), service și controller, alături de alte clase necesare pentru configurare, toate acestea fiind detaliate în continuare.

Pentru aproape fiecare entitate din baza de date se implementează fiecare dintre clasele menționate mai sus, mai exact pentru *Artist*, *Artwork*, *Comment*, *Country*, *FavouriteArtist*, *FavouriteArtwork*, *Gallery*, *Genre*, *Museum*, *Room*, *Token*, *User*, *UserRole*. La acestea se adaugă și microserviciile *Role* și *Authentication*, necesare pentru componenta de securitate. Întrucât toate aceste componente au similitudini în structura și logica de proiectare, ne limităm la a prezenta câte un exemplu din fiecare tip de clasă aparținând microserviciului *Artwork*, pentru consecvență.

4.2.1 Clasa Entity

Pachetul *models* cuprinde toate clasele tip *Entity* sau DAO, care corespund entităților din baza de date. Dintre acestea, clasa *ArtworkEntity* are următoarea structură:

```
@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
@Table(name = "ARTWORK")
public class ArtworkEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "ID", updatable = false, unique = true)
    private int id;
    @Column(name = "TITLE")
    private String title;
    @Column(name = "YEAR")
    private int year;
    @Column(name = "TECHNIQUE")
    private String technique;
```

```

@Column(name = "DESCRIPTION")
private String description;
@Column(name = "IMAGE_PATH")
private byte[] imagePath;
@Column(name = "POSITION")
private int position;
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "ARTIST_ID", referencedColumnName = "ID")
private ArtistEntity artist;
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "GENRE_ID", referencedColumnName = "ID")
private GenreEntity genre;
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "ROOM_ID", referencedColumnName = "ID")
private RoomEntity room;
@OneToMany(mappedBy = "artwork", cascade = CascadeType.ALL)
private List<CommentEntity> comments;
}

```

Adnotarea *@Entity* proprie Spring indică tipul acestei clasei (o entitate care corespunde unui tabel), iar adnotarea *@Table(name = "ARTWORK")* precizează numele tabelului din MySQL, pentru a asigura maparea.

Adnotările *@Data*, *@NoArgsConstructor* și *@AllArgsConstructor* aparțin sistemului Lombok, care generează funcții de bază pentru prelucrarea datelor precum *getter*, *setter*, *equals*, *toString*, *hashCode* și constructori.

Fiecare atribut propriu obiectului are o adnotare *@Column*, care indică numele coloanei corespunzătoare din tabel, iar cheile străine sunt mapate folosind *@JoinColumn* și adnotarea *@ManyToOne* sau *@OneToMany*, în funcție de tipul relației. Tehnologia Spring este capabilă să creeze un atribut din tipul clasei care corespunde tabelului către care face referință cheia străină (*ArtistEntity*, *GenreEntity*, *RoomEntity*, *CommentEntity*).

De asemenea id-ul entității este desemnat prin *@Id* și *@GeneratedValue(strategy = GenerationType.IDENTITY)*, o strategie de autoincrementare a id-ului care asigură unicitatea în interiorul unei tabele.

Alte adnotări întâlnite în cadrul altor entități ar fi: @IdClass pentru cheile primare compuse, @JsonIgnore pentru a evita incluziunile tip buclă infinită cauzate de cheile străine, @Enumerated(EnumType.STRING) pentru câmpurile de tip enumerație din baza de date.

4.2.2 Interfața Repository

Utilizând entitățile definite în subcapitolul precedent, interfețele tip persistență sau *repository* realizează comunicarea cu baza de date, moștenind clasa *JpaRepository* – o extensie tip Java Programming API ce permite efectuarea operațiunilor CRUD (Create, Retrieve, Update, Delete) asupra bazei de date (Rout, 2021).

JpaRepository are metode predefinite foarte utile pentru gestiunea bazei de date, precum *findById()*, *save()*, *deleteById()*, *findAll()* – acestea permit regăsirea înregistrărilor sau efectuarea modificărilor în funcție de valoarea unui câmp precum id-ul.

Pentru exemplificare, putem observa mai jos codul care stă la baza *ArtworkRepository*, clasa repository care gestionează operele de artă:

```
public interface ArtworkRepository extends JpaRepository<ArtworkEntity, Integer> {
    List<ArtworkEntity> findAllByArtistIdOrderByTitle(int artistId);
    List<ArtworkEntity> findAllByGenreIdOrderByTitle(int genreId);
    List<ArtworkEntity> findAllByRoomIdOrderByPosition(int roomId);
}
```

Clasa se bazează pe tuplul <ArtworkEntity, Integer>, anume entitatea gestionată și tipul de date al id-ului acesteia. În principiu, această clasă este un *Spring Bean* – un obiect instanțiat și gestionat de către containerul Spring – iar cele trei metode definite în interfață sunt implementate automat și pot fi folosite mai departe prin *Dependency Injection* – „injectarea” obiectului *repository* în clasele unde avem nevoie să îi folosim metodele, fără a-l crea prin constructori de fiecare dată.

Toate celelalte interfețe *repository* sunt similare, gestionând entitățile deja menționate, și se regăsesc în pachetul *persistence*.

4.2.3 Clasa DTO

Pentru a asigura eficiența cererilor GET de preluare a datelor și a evita fenomenul de *overfetching* (în care se returnează mai multe date decât sunt necesare), fiecărei clase de tip entitate îi corespund și două clase de tip DTO (*Data Transfer Object*), anume *RequestDTO* și *ResponseDTO*. Acest *design pattern* permite agregarea datelor pentru transferul prin API și, totodată, sporește securitatea aplicației, asigurând faptul că date sensibile precum parolele nu sunt transferate odată cu obiectele cerute.

Începem cu *ArtworkResponseDTO*, clasă ce conține câmpurile ce vor fi primite în *ReponseEntity* (obiectul de răspuns al unei cereri API):

```
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
public class ArtworkResponseDTO {
    private int id;
    private String title;
    private String technique;
    private int year;
    private String description;
    private byte[] imagePath;
    private int roomId;
    private int position;
    private String artistFirstName;
    private String artistLastName;
    private String genreName;
    private int genreId;
}
```

Observăm că, în completarea atributelor caracteristice prezente în *ArtworkEntity*, aici se mai adaugă niște câmpuri din alte tabele (*artistFirstName*, *artistLastName*, *genreName*, *genreId*), obținute prin maparea realizată folosind biblioteca *MapStruct*, explicată în

subcapitolul următor. În acest caz, se previn cererile multiple către API pentru a prelua și date despre artist și curentul artistic, înglobând aceste date deja în obiectul de răspuns.

Mai departe, putem observa și obiectul care mapează o cerere, folosit în principal pentru cererile tip PUT și POST:

```
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
public class ArtworkRequestDTO {
    private String title;
    private String technique;
    private String description;
    private byte[] imagePath;
    private int roomId;
    private int year;
    private int position;
    private int artistId;
    private int genreId;
}
```

Aici, se primesc și id-urile corespunzătoare cheilor străine (*roomId*, *artistId*, *genreId*), care asigură legarea directă a noilor obiecte de cele deja existente.

Adnotările `@Getter`, `@Setter`, `@AllArgsConstructor` și `@NoArgsConstructor` aparțin de biblioteca Lombok, deja menționată.

4.2.4 Interfața Mapper

MapStruct este o altă dependență Spring Boot care realizează maparea claselor de tip *Bean*. Precum la *repository*, și în acest caz folosim *Inversion of Control (IoC)*, containerul Spring care creează și configurează obiectele, gestionându-le întreg ciclul de viață (Rout, 2022). Astfel, definim interfața *ArtMapper* (Figura 14), adnotată cu `@Mapper`, pentru a anunța că acesta este un *Bean* care va fi injectat ulterior:

```

@Mapper(componentModel = "spring")
public interface ArtMapper {

    @Mappings({
        @Mapping(target = "countryName", source = "country.name")
    })
    ArtistResponseDTO artistEntityToArtistResponseDTO(ArtistEntity
artistEntity);

    @Mappings({
        @Mapping(target = "country.id", source = "countryId")
    })
    ArtistEntity artistRequestDTOToArtistEntity(ArtistRequestDTO
artistRequestDTO);

    @Mappings({
        @Mapping(target = "id", source = "artistId"),
        @Mapping(target = "country.id", source =
"artistRequestDTO.countryId")
    })
    ArtistEntity artistRequestDTOToArtistEntityWithId(Integer artistId,
ArtistRequestDTO artistRequestDTO);

    @Mappings({
        @Mapping(target = "artistFirstName", source =
"artist.firstName"),
        @Mapping(target = "artistLastName", source =
"artist.lastName"),
        @Mapping(target = "genreName", source = "genre.name"),
        @Mapping(target = "genreId", source = "genre.id"),
        @Mapping(target = "roomId", source = "room.id")
    })
    ArtworkResponseDTO artworkEntityToArtworkResponseDTO(ArtworkEntity
artworkEntity);

```

Figură 14. Interfața Mapper

Aici, realizăm o serie de mapări între entități și DTO-uri (Request și Response), simple și sub formă de liste, precum și o mapare însoțită de Id (*artistRequestDTOToArtistEntityWithId*), necesară pentru cererile tip PUT, în care se primește și id-ul obiectului care trebuie modificat, iar acesta trebuie căutat în baza de date folosind metoda *findById* definită în *repository*.

Mapările sunt indicate prin intermediul adnotațiilor *@Mapping*, grupate cu ajutorul lui *@Mappings*, fiecare conținând numele atributului din clasa sursă și pe cel care îi corespunde în clasa țintă.

4.2.5 Interfața Service

Următorul strat al arhitecturii este cel de *Business Logic*, cuprins în interfețele *Service* și în implementările lor. Acestea reprezintă un fel de intermediar între stratul persistent de *Repository* și *Controllerele* în care se configurează punctele de acces ale API-ului.

În clasele Service, au loc principalele operațiuni de procesare a datelor extrase din baza de date și pregătirea acestora pentru a fi returnate de *endpoint*, respective procesarea datelor primite prin PUT sau POST și maparea lor către entități.

Vom analiza clasa *ArtworkServiceImpl* (implementarea interfeței *ArtworkService*) și cele mai importante metode ale sale:

```
@Service
@AllArgsConstructor
public class ArtworkServiceImpl implements ArtworkService {

    private final ArtworkRepository artworkRepository;
    private final ArtMapper artMapper;
```

Figură 15. Clasa *ArtworkServiceImpl*

În primă fază, se injectează obiectele tip *ArtworkRepository* și *ArtMapper*, sub forma design pattern-ului *Singleton* (o singură instanță în toată aplicația). Majoritatea metodelor clasei *service* realizează o mapare DAO-DTO sau invers și apelează una dintre metodele clasei tip *repository*. Spre exemplu, în *Figura 16*, observăm metoda de actualizare (*update*) ce va fi apelată în cazul cererilor PUT:

```
@Override
@Transactional
public ArtworkResponseDTO updateArtwork(int id, ArtworkRequestDTO
artworkRequestDTO) {
    ArtworkEntity oldArtwork =
artworkRepository.findById(id).orElseThrow(() ->
        new ArtNotFoundException(ErrorCode.ERR_03_ARTWORK_NOT_FOUND));
    if (oldArtwork == null) {
        return null;
    } else {
        ArtworkEntity artworkEntity =
artMapper.artworkRequestDTOToArtworkEntityWithId(id, artworkRequestDTO);
        return
artMapper.artworkEntityToArtworkResponseDTO(artworkRepository.save(
artworkEntity));
    }
}
```

Figură 16. Metoda *updateArtwork* din clasa tip Service

Întâi, se caută entitatea de actualizat prin intermediul metodei *findById* a obiectului *repository*. În cazul în care acesta nu este găsit, se aruncă o excepție personalizată care va fi gestionată de controllerul special de excepții. Dacă instanța există, are loc maparea id-ului său pe o entitate formată din noile date, apoi această entitate este salvată în baza de date.

În *Figura 17*, se poate observa metoda care returnează toate operele de artă aferente unei camere, ordonate crescător în funcție de indicele poziției din acea sală. În primă instanță, se returnează un obiect de tip *Optional*, pentru a lua în calcul situația în care există camere fără

opere de artă care să le corespundă. În cazul în care obiectul nu este vid, se realizează maparea listei de entități într-o listă de *ResponseDTO-uri*.

```
@Override
public List<ArtworkResponseDTO> getAllArtworksByRoomIdOrderByPosition(int
roomId) {
    Optional<List<ArtworkEntity>> artworkListOptional =
Optional.ofNullable(
        artworkRepository.findAllByRoomIdOrderByPosition(roomId));
    if (artworkListOptional.isEmpty()) {
        throw new ArtNotFoundException(ErrorCode.ERR_03_ARTWORK_NOT_FOUND);
    }
    return |
artMapper.artworkEntityListToArtworkResponseDTOList(artworkListOptional
.get());
}
```

Figură 17. Metoda `getAllArtworksByRoomIdOrderByPosition`

4.2.6 Clasa Controller

Clasa de tip *Controller* este cea care mapează punctele de acces (*endpoints*) ale API-ului și corespunde stratului de prezentare (*Presentation Layer*) al arhitecturii.

Pe același principiu de Inversion of Control, se injectează aici obiectul tip service:

```
private final ArtworkService artworkService;
```

Fiecare endpoint are anotări specifice `@GetMapping`, `@PostMapping`, `@PutMapping` sau `@DeleteMapping`, însoțite de anotări caracteristice documentării API-ului prin intermediul instrumentelor *Swagger*: `@Operation` (numele operației disponibile), `@ApiResponses` (răspunsurile posibile ale aceluia endpoint), `@Content` (tipul conținutului aferent), `@Schema` (clasa care corespunde aceluia răspuns).

```
@Operation(summary = "Get artworks by and room id")
@ApiResponses(value = {
    @ApiResponse(responseCode = "500", description = "Server error",
        content = @Content(mediaType = "application/json", schema =
@Schema(implementation =
            ArtInternalServerErrorException.class))),
    @ApiResponse(responseCode = "404", description = "Artworks not
found",
        content = @Content(mediaType = "application/json", schema =
@Schema(implementation =
            ArtNotFoundException.class))),
    @ApiResponse(responseCode = "400", description = "Invalid request",
        content = @Content(mediaType = "application/json", schema =
@Schema(implementation =
            ArtBadRequestException.class))),
    @ApiResponse(responseCode = "200", description = "Successful
retrieval",
        content = @Content(mediaType = "application/json", array =
@ArraySchema(schema = @Schema(
            implementation = ArtworkResponseDTO.class))))
})
```

Figură 18. Exemplu de endpoint din clasa `ArtworkController` - adnotări

```

@GetMapping("/user/artworks")
public ResponseEntity<List<ArtworkResponseDTO>>
getArtworksByRoomIdOrderByPosition(
    @RequestParam(required = true) int roomId
) {
    return new ResponseEntity<>(artworkService.
        getAllArtworksByRoomIdOrderByPosition(roomId), HttpStatus.OK);
}

```

Figură 19. Exemplu de endpoint din clasa ArtworkController – metoda get

Toate endpointurile aplicației backend încep cu */api* (<http://localhost:8080/api>), urmate de */user* sau */admin*, în funcție de rolul necesar pentru a accesa acel endpoint (situație gestionată mai departe în configurațiile de securitate), iar mai apoi o cale sugestivă precum */artworks*, respectiv */artwork/{id}* pentru operațiile privind o singură operă de artă. În cazul din *Figura 19*, în loc de un parametru al căii tip *@PathVariable*, am utilizat un parametru tip *query string*, extras prin adnotația *@RequestParam*.

În documentația API-ului din interfața Swagger, endpoint-ul *Artwork* este afișat ca în *Figura 20*:

The image shows the Swagger API documentation interface. At the top, it says "Artwork requests". Below this, there are several endpoints listed with their HTTP methods and descriptions:

- PUT** `/api/admin/artwork/{id}` Update an artwork
- DELETE** `/api/admin/artwork/{id}` Delete an Artwork by id
- POST** `/api/admin/artwork` Add a new artwork
- GET** `/api/user/artworks` Get artworks by and room id
- GET** `/api/user/artwork` Get artworks by artist id
- GET** `/api/user/artwork/{id}` Get Artwork by id

Below the endpoints, there is a "Parameters" section. It contains a table with two columns: "Name" and "Description".

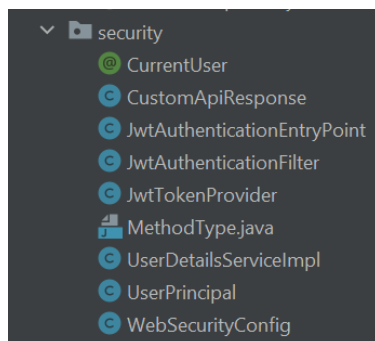
Name	Description
id * required integer(\$int32) (path)	id

At the bottom right of the parameters section, there is a blue button labeled "Execute".

Figură 20. Endpoint în interfața Swagger

4.2.7 Componenta de securitate

Securitatea în cadrul aplicației backend este asigurată de implementarea serviciului Spring Security prin JWT (JSON Web Token) pentru autentificare și autorizare. Elementele acestei tehnologii se regăsesc în pachetul *security*, cu următoarea structură (Figura 21):



Figură 21. Componentele pachetului security

În principiu, *framework-ul* de securitate filtrează fiecare cerere primită, verificând dacă un anumit URL este accesibil cu sau fără autentificare și ce roluri (user/admin) au acces la el – partea de autorizare. Acest sistem implementează interfețele *UserDetails* și *UserDetailsService*, care comunică cu *UserRepository* și conțin metode de gestiune a utilizatorilor, se folosește de funcția *.authenticate* a interfeței predefinite *AuthenticationManager* și generează un token unic de identificare a sesiunii utilizatorului prin intermediul *JwtTokenProvider*.

Mai departe, *WebSecurityConfig*, care moștenește clasa abstractă *WebSecurityConfigurerAdapter*, mapează căile API-ului în funcție de autoritățile necesare pentru a le accesa, în cadrul metodei *.configure* surprinsă în Figura 22:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .cors()
        .and()
        .csrf()
        .disable()
        .sessionManagement()
        .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
        .and()
        .authorizeRequests()
        .antMatchers(HttpMethod.OPTIONS, "**").permitAll()
        .antMatchers("/").permitAll()
        .antMatchers("/swagger-ui/**", "/v3/api-docs/**").permitAll()
        .antMatchers("/api/authentication/**").permitAll()
        .antMatchers("/api/open/**").permitAll()

        .antMatchers("/api/user/**").hasAnyAuthority("ROLE_USER,ROLE_ADMIN")
        .antMatchers("/api/users/**").hasAnyAuthority("ROLE_USER,
        ROLE_ADMIN")
        .antMatchers("/api/admin/**").hasAuthority("ROLE_ADMIN")
}
```

Figură 22. Autorizarea căilor API în Spring Security

4.3 Implementarea aplicației frontend

Având în vedere răspunsurile primite la interviu privind tipul dispozitivului de pe care s-ar accesa aplicația, este evident faptul că este necesară o aplicație portabilă, potrivită pentru ecrane mici, care să fie accesată de vizitatorii care se deplasează fizic prin muzeu.

În scopul de a asigura această calitate de *responsiveness* – adaptabilitatea interfeței la orice tip de ecran – aplicația frontend este dezvoltată utilizând *framework-ul* de TypeScript Angular, folosind principii de design și instrumente din modulul Bootstrap.

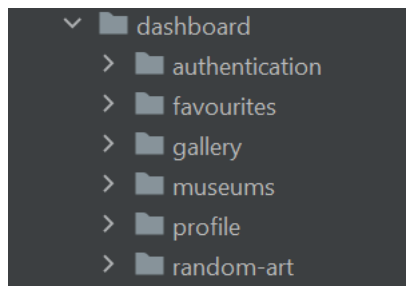
În continuare, ne îndreptăm atenția către modul în care este structurată și implementată aplicația frontend.

4.3.1 Componentele frontend

În general, fiecare funcționalitate a aplicației este gestionată de o componentă distinctă, cu propriul modul de declarații și modul de rutare.

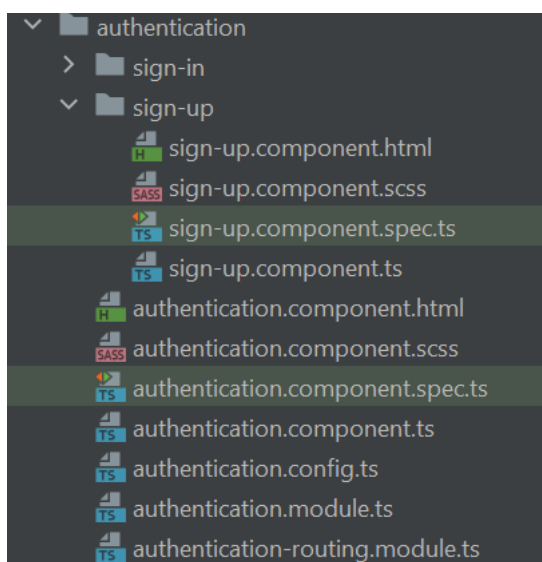
În consecință, pachetul *dashboard* conține componentele principale, în timp ce componentele recurente în ansamblul aplicației se regăsesc în pachetul *shared*, iar elementele esențiale precum clasa componentei de bază (extinsă de toate componentele) și fișierele navbar se regăsesc în pachetul *core*.

Astfel, fiecare dintre componentele majore ale interfeței se regăsesc într-unul din subpachetele din *dashboard* (Figura 23):



Figură 23. Structura componentelor din dashboard

O componentă conține un fișier HTML cu structura vizuală a interfeței, precum și un fișier TypeScript cu funcțiile apelate pe acea pagină (Figura 24):




Figură 24. Elementele unei componente Angular

Pentru exemplificare, alegem componentele sign-in și sign-up din *Figura 24*, al căror design implementat se poate observa în *Figura 25* – se poate comuta între cele 2 pagini prin butonul din dreapta sus Sign Up/Sign In:

Artefacto

MuseumsArt Randomiser

 Log in

Sign up

Already a user? [Sign In!](#)

First name

Field is required

Last name


Field is required

Email

Field is required

Password

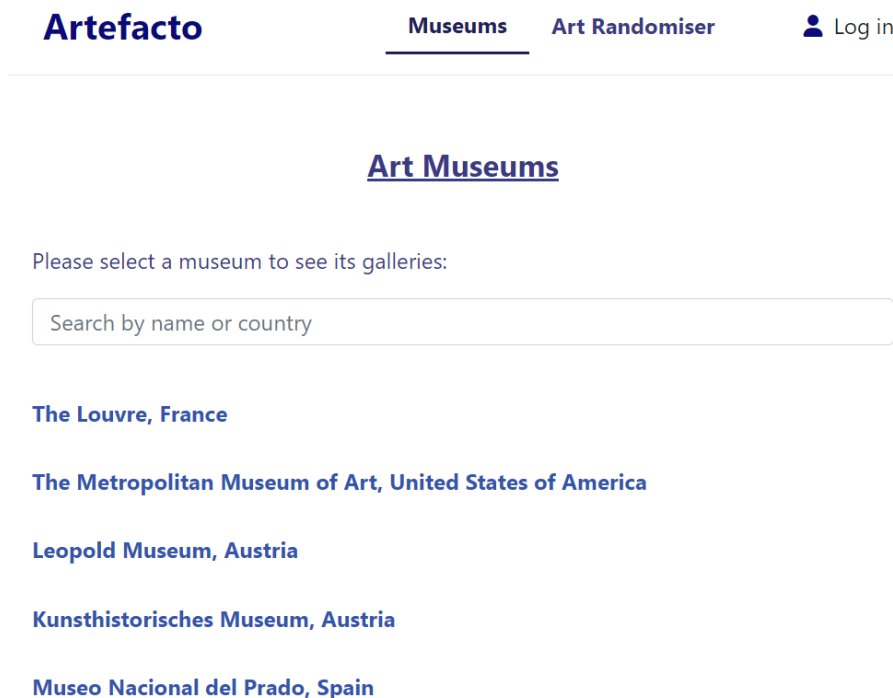
Field is required



Sign Up

Figură 25. Fereastra de sign-up

Un vizitator al aplicației web poate accesa pagina principală ce conține lista muzeelor și fără a se loga. De aici (*Figura 26*), poate selecta un muzeu și poate naviga la pagina muzeului, ce conține o listă a galeriilor.



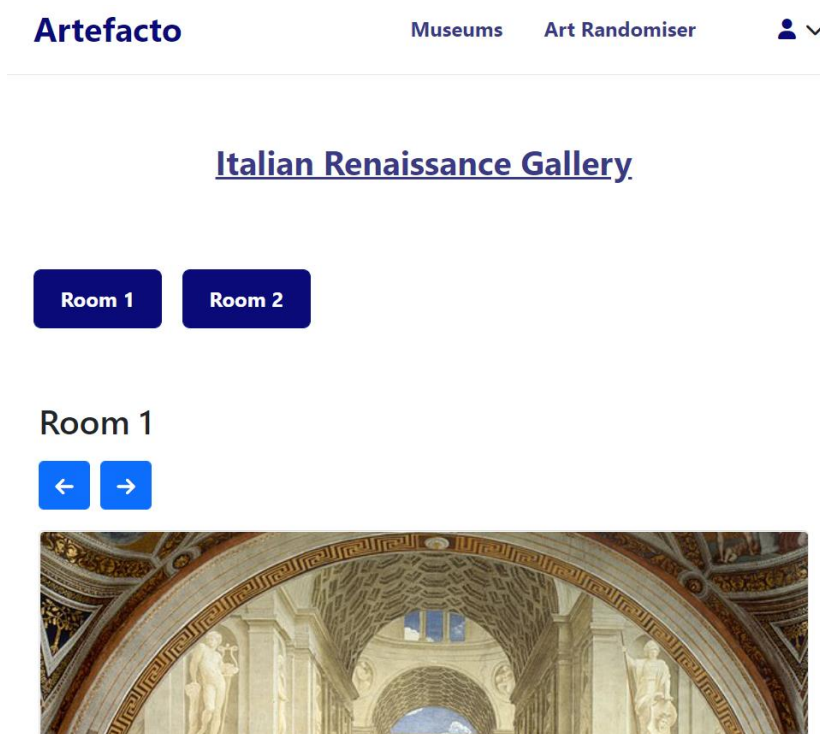
Figură 26. Pagina principală a aplicației

În acel punct, devine obligatoriu ca vizitatorul să se logheze pentru a putea vizualiza pagina unei galerii, cu operele de artă aferente (*Figura 27*).



Figură 27. Mesaj de solicitare a autentificării

Odată logat, utilizatorul va putea vizualiza pagina cu sălile galeriei și operele de artă din fiecare sală (*Figura 28*):



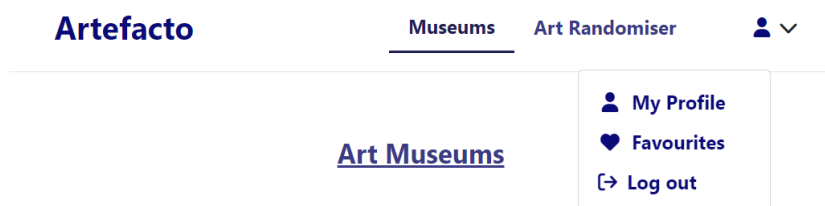
Figură 28. Pagina unei galerii de artă

În funcție de rolul utilizatorului (user/admin), pe pagini sunt sau nu sunt prezente butoane de editare și adăugare a galeriilor, camerelor sau operelor de artă.

4.3.2 Navbar

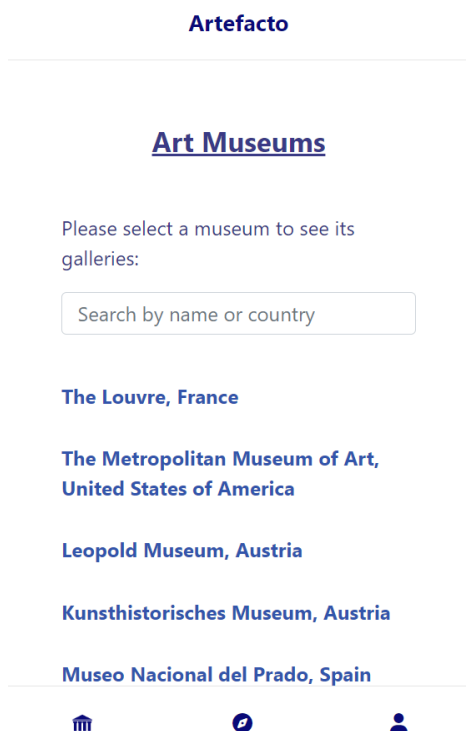
Din considerente de *responsiveness*, aplicația are două feluri de navbar (bară de navigare), una pentru versiunea desktop și una pentru versiunea mobilă, iar trecerea între acestea se face automat, în funcție de dimensiunea ecranului. Acestea două se regăsesc în directorul *navbars* din *core*.

Navbarul pentru versiunea desktop (*Figura 29*) conține tab-urile *Museums* (lista muzeelor), *Art Randomiser* (pagina ce afișează o operă de artă aleatoare) și opțiunile profilului: *My Profile*, *Favourites* și *Sign-Out*.



Figură 29. Navbar pentru desktop

În schimb, navbarul pentru versiunea de mobil (*Figura 30*) folosește pictograme (*icons*) sugestive pentru taburile descrise explicit în versiunea desktop:



Figură 30. Versiunea de mobil a aplicației

Aceste elemente sunt dezvoltate utilizând clasele specifice *Bootstrap* – *navbar*, respectiv *container*.

4.3.3 Rutele și restricțiile de acces

Rutele principale ale aplicației se regăsesc în modulul *app-routing.module.ts* (*Figura 31*), unde fiecărei rute îi corespunde un modul din *dashboard* care gestionează traficul pe acea rută:

```
const routes: Routes = [
  { path: '', redirectTo: 'museums', pathMatch: 'full' },
  { path: 'museums', loadChildren: () =>
import('./dashboard/museums/museums.module').then(m => m.MuseumsModule)},
  { path: 'random_artwork', loadChildren: () => import('./dashboard/random-
art/random-art.module').then(m => m.RandomArtModule)},
  { path: 'profile', loadChildren: () =>
import('./dashboard/profile/profile.module').then(m => m.ProfileModule)},
  { path: 'favourites', loadChildren: () =>
import('./dashboard/favourites/favourites.module').then(m =>
m.FavouritesModule)},
  { path: 'authentication', loadChildren: () =>
import('./dashboard/authentication/authentication.module').then(m =>
m.AuthenticationModule)},
  { path: 'gallery', loadChildren: () =>
import('./dashboard/gallery/gallery.module').then(m => m.GalleryModule)},
```

Figură 31. Rutele și modulele aplicației

Restricțiile de acces se realizează în modulele de rutare ale fiecăreia dintre componentele principale, utilizând *RouteGuards*, care permit accesul pe o anumită rută numai utilizatorilor cu unul dintre rolurile prezente în *neededRoles* (Figura 32):

```
const routes: Routes = [
  {
    path: ':galleryId',
    component: GalleryComponent,
    canActivate: [AuthorisationGuard],
    data: {neededRoles: ['ROLE_USER', 'ROLE_ADMIN']},

    children: [
      { path: 'room/:roomId', component: RoomComponent},|
```

Figură 32. RouteGuards pe rutele paginii unei galerii

De gestiunea și controlul acestor restricții se ocupă clasa *AuthorisationGuard* (Figura 33), care verifică dacă utilizatorul curent are între roluri unul dintre cele necesare pentru a accesa calea respectivă.

```
@Injectable()
export class AuthorisationGuard implements CanActivate {
  constructor(private authorisationService: AuthorisationService) {
  }

  canActivate(route: ActivatedRouteSnapshot): Observable<boolean> {
    const allowedRoles = route.data['neededRoles'];
    return this.authorisationService.getSignedInUser().pipe(map(
      (response) => {
        return allowedRoles.some((item: string) =>
response.roles.includes(item));
      }));
  }
}
```

Figură 33. Clasa AuthorisationGuard

4.3.4 User state management

Controlul stării utilizatorului curent pe tot parcursul aplicației se realizează folosind *pattern-ul* de state management (NGXS) caracteristic Angular.

Principiul pe care se bazează este existența unei „stări” a utilizatorului, care este accesibilă oriunde în aplicație și se modifică prin una dintre acțiunile descrise de metodele prezente în *user.action.ts* (*GetLoggedUser* și *RemoveLoggedUser*) din directorul *shared/redux/user*. Acest mod de lucru previne necesitatea de a face cereri multiple către backend pentru a verifica rolurile utilizatorului logat pe fiecare pagină pe care acesta navighează, fapt ce eficientizează aplicația.

```



```

Figură 34. User state management

Astfel, în *Figura 34*, datele utilizatorului respectă modelul definit în interfața *AuthoriseResponseModel* (ce corespunde cu *UserResponseDTO* din backend), iar pentru a obține aceste date se injectează clasa *Store* în constructorul paginii de interes și se folosește decoratorul *@Select* pentru a prelua starea utilizatorului din *Store*.

Această stare (*loggedInUser*) este utilizată ulterior pentru a verifica dacă există un user logat, în funcție de care se afișează un buton de *Login* sau un buton de *My Profile*, și apoi pentru a-i afișa detaliile în pagina de profil.

5. Testare și evaluare

Asigurarea calității și a fiabilității sistemului presupune testarea diferitelor funcționalități implementate, documentarea și soluționarea erorilor.

În acest scop, funcționarea și integritatea aplicației sunt verificate, cu precădere, prin două metode: testarea manuală și verificarea endpointurilor backend prin cereri HTTP din Swagger și Postman, detaliate în subcapitolele ce urmează.

5.1 Testarea manuală

Procesul de testare manuală a aplicației *Artefacto* presupune verificarea funcționalităților prin simularea diferitelor cazuri de utilizare, urmată de evaluarea modului în care aplicația răspunde, evaluarea interactivității interfeței, estimarea timpului de așteptare și, mai ales, determinarea posibilelor *bug-uri* și erori care ar afecta experiența utilizatorilor aplicației.

Acest demers este documentat în *Tabelul 2*, care înregistrează întreg parcursul de testare manuală: etapele, acțiunile întreprinse și rezultatele observate.

Etapa de testare	Acțiuni	Rezultate
Navigarea fără logare	1. Accesarea paginii principale 2. Verificarea casetei de căutare 3. Selectarea unui muzeu 4. Testarea funcționalității care deschide fereastra Google Maps 5. Căutarea unei galerii în casetă 6. Selectarea unei galerii	- Pagina principală este accesibilă și pentru vizitatorii nelogați. - Ambele casete de căutare funcționează - Fereastra Google Maps se deschide cu locația muzeului - Apare un mesaj de alertă care solicită logarea pentru a accesa o galerie, deci restricția pe acea rută funcționează - Se trimite cereri get cu id null al utilizatorului către backend. Ar trebui remediat prin verificarea în user state management

Logarea utilizatorului	<ol style="list-style-type: none"> 1. Navigarea pe pagina de login 2. Comutarea spre pagina de sign-up și înapoi 3. Introducerea unor credențiale incorecte + Submit 4. Introducerea credențialelor corecte + Submit 	<ul style="list-style-type: none"> - Pagina de login funcționează; se poate trece pe pagina de sign-up și se poate reveni - La introducerea credențialelor incorecte, apare mesajul <i>toast</i> de eroare la logare - La introducerea credențialelor valide, logarea se realizează cu succes
Navigarea utilizatorului logat	<ol style="list-style-type: none"> 1. Repetarea pașilor de la prima etapă 2. Accesarea paginii unei galerii 3. Vizualizarea operelor dintr-o cameră 4. Navigarea prin view-ul tip <i>carousel</i> cu opere de artă 5. Accesarea Art Randomiser 6. Accesarea profilului 	<ul style="list-style-type: none"> - Paginile galeriilor sunt accesibile utilizatorilor logați - Se pot vizualiza cardview-urile cu operele de artă - <i>TypeError: Cannot read properties of undefined (reading '0')</i> la încărcarea paginii unei galerii - Celelalte componente funcționează așa cum ar trebui
Delogarea	<ol style="list-style-type: none"> 1. Logarea 2. Apăsarea butonului <i>Sign Out</i> 	<ul style="list-style-type: none"> - Utilizatorul este delogat - Se revine la http://localhost:4200/ - Apare un mesaj de eroare în consolă <i>Http failure response for http://localhost:8080/api/user/null</i>

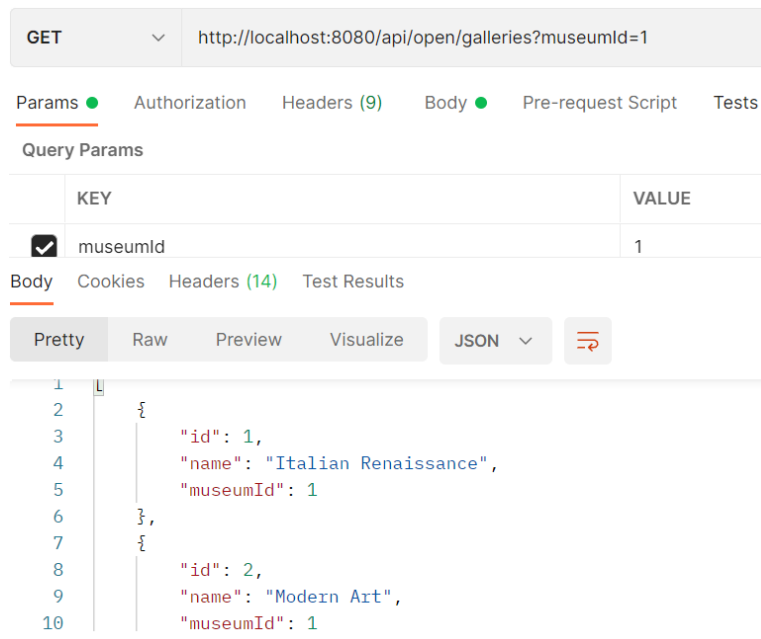
Tabel 2. Procesul de testare manuală

5.2 Testarea prin cereri HTTP

Pentru verificarea cererilor HTTP, s-au utilizat Postman, Swagger și consola *Network* din *Google Chrome developer tools*.

Asemenea cereri se trimit pentru verificarea bunei funcționări a tuturor endpointurilor, însă, pentru exemplificare, vom realiza câte o cerere prin fiecare tehnologie menționată.

Pentru început, trimitem o cerere GET folosind Postman (*Figura 35*), către adresa <http://localhost:8080/api/open/galleries>, la care concatenăm parametrul *museumId=1*. Răspunsul este cel așteptat, anume lista galeriilor muzeului cu id-ul 1.



Figură 35. Cerere GET trimisă din Postman

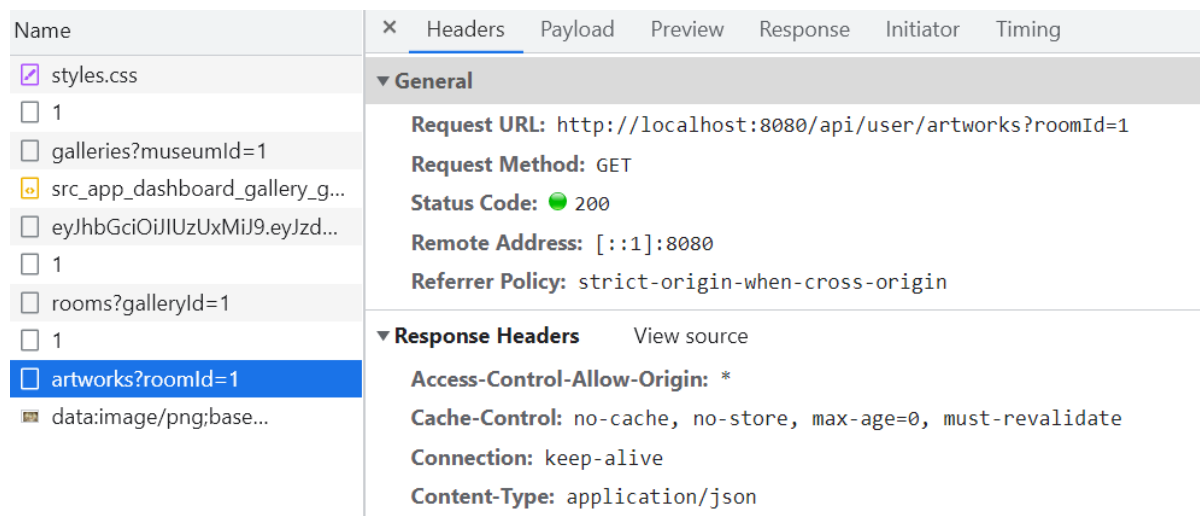
Mai departe, încercăm să trimitem o cerere tip POST din Swagger (*Figura 36*), către <http://localhost:8080/api/admin/artist>. Utilizăm Swagger, întrucât acesta adaugă implicit în header token-ul utilizatorului curent și face accesibil endpointul pentru administratori.

Completăm corpul cererii (*Request body*) de tip JSON cu datele unui artist, executăm cererea și primim răspunsul de confirmare 201 (*Created*).



Figură 36. Cerere POST trimisă din Swagger

Nu în ultimul rând, verificăm statusul cererilor din consola Network a aplicației frontend (*Figura 37*). Navigând prin aplicație, observăm că se trimite cereri tip GET către localhost care au *Status Code 200 (Success)*.



Figură 37. Cereri GET trimise de aplicația frontend

5.3 Evaluare

Din câte am observat, aplicația performează în conformitate cu media așteptărilor, în majoritatea cazurilor funcționând fără erori majore sau întreruperi. Cererile trimise sunt mapate corect în baza de date, iar datele sunt extrase destul de prompt.

În etapa de testare, am remarcat anumite erori TypeScript, precum și niște cereri către API-ul `/users` care ar putea fi prevenite.

Chiar dacă este minimalistă, interfața face navigarea mai intuitivă, pentru a ajunge cu ușurință la centrul de interes, anume paginile cu explicații despre operele de artă.

Faptul că există niște pagini de început disponibile fără logare este util deoarece îi familiarizează pe vizitatori cu aplicația și îi poate convinge să își creeze un cont pentru a accesa restul conținutului.

Drept îmbunătățiri, s-ar putea face redirectare după delogare înapoi pe pagina cu lista muzeelor și s-ar putea adăuga mai multe funcționalități, discutate mai departe în secțiunea de posibile direcții de dezvoltare din capitolul *Concluzii*.

Concluzii

Această lucrare a încercat să ofere o perspectivă cât mai completă asupra întregului proiect de dezvoltare a aplicației web *Artefacto*, un sistem informatic având rolul de a funcționa asemenea unui ghid în muzeele de artă.

S-a început cu partea de analiză a mediului de business, urmată de capitolul de proiectare a sistemului informatic, noțiuni pe baza cărora s-a realizat implementarea, iar rezultatul a fost testat și evaluat.

Aplicația este compusă dintr-o bază de date MySQL, o componentă backend de tip serviciu REST dezvoltat în *framework-ul* Java Spring Boot, precum și o componentă frontend realizată folosind Angular, un *framework* TypeScript.

API-ul implementat în backend oferă o suită de microservicii care gestionează cererile de tip HTTP, iar interfața oferă utilizatorilor obișnuiți posibilitatea de a naviga pe paginile muzeelor, ale galeriilor și ale operelor de artă, pentru a vedea detalii și explicații, a posta comentarii sau a le adăuga la favorite. Utilizatorii cu rol de administrator pot edita informațiile sau adăuga noi înregistrări.

Un muzeu de artă ar avea beneficii în urma utilizării aplicației, întrucât aceasta îmbunătățește fluxul prin muzeu, pe baza un circuit bine-stabilit prin ordinea camerelor și a operelor de artă, dar oferă vizitatorilor și posibilitatea de a regăsi mai ușor acele opere de interes, nefiind nevoie să parcurgă întreg muzeul pentru a le căuta. De asemenea, vizitatorii ar avea o experiență culturală mai plăcută, deoarece ar descoperi mai multe informații noi și ar înțelege mai bine exponatele din muzeu.

În privința posibilelor direcții viitoare de dezvoltare, consider că există potențial pe mai multe segmente. În primul rând, aplicația ar putea fi extinsă prin noi funcționalități adiacente precum adăugarea de recenzii muzeelor, introducerea unei secțiuni speciale de expoziții temporare, eventual și cu un calendar, implementarea posibilității de a achiziționa un bilet la muzeu direct din aplicație, pentru a reduce timpul de așteptare la coadă.

În al doilea rând, după cum am menționat în introducere, tehnologia avansează într-un mod impresionant, iar noile tendințe axate pe inteligența artificială ar putea fi integrate și în acest domeniu. Spre exemplu, s-ar putea adăuga o componentă de *Natural Language Processing* (ramură a inteligenței artificiale care se ocupă cu procesarea și înțelegerea limbajului natural), pentru a analiza comentariile utilizatorilor și a determina care opere de artă sunt cele mai populare sau cele mai apreciate. În mod similar, o funcționalitate interesantă și

deosebit de utilă ar fi un scanner bazat pe *Computer Vision* (capacitatea sistemelor bazate pe inteligența artificială de a recunoaște și a interpreta imagini), care le-ar permite utilizatorilor să facă poză unei opere de artă și să obțină imediat detalii despre aceasta, fără a mai fi necesar să parcurgă întreaga listă cu galerii și săli pentru a o identifica.

Consider că întregul demers al dezvoltării acestei aplicații m-a ajutat să îmi însușesc noi deprinderi despre cum să analizez contextul pentru care este menită aplicația și cum să determin cele mai utile funcționalități. Totodată, ocazia de a lucra cu două tehnologii noi într-un proiect *full-stack* a fost o provocare în urma căreia pot spune că mi-am dezvoltat adaptabilitatea. Nu în ultimul rând, acest proiect mi-a permis să îmi cultiv pasiunea pentru artele vizuale și pentru interdisciplinaritate.

Glosar

<i>brainstorming</i>	procesul de a genera idei în cadrul unui grup, pentru a găsi soluții posibile unei probleme
<i>design pattern</i>	tipar prin care se rezolvă anumite probleme în dezvoltarea software
<i>endpoint</i>	punct de comunicare prin cereri HTTP cu un API
<i>framework</i>	sistem ce facilitează dezvoltarea anumitor componente software
<i>homepage</i>	pagina principală (de întâmpinare) a unui site
<i>product owner</i>	persoanele care asigură livrarea produsului cu valoarea scontată și stabilesc sarcinile de lucru
<i>randomizare</i>	selectare aleatoare
<i>responsiveness</i>	capacitatea interfeței de a se adapta la dimensiunile ecranului
<i>stakeholderi</i>	persoanele care au anumite interese sau contribuții într-un proiect

Bibliografie

- Ahmad, M. O., Markkula, J., & Oivo, M. (2013). Kanban in Software Development: A Systematic Literature Review. *Conference: Software Engineering and Advanced Applications*. <https://doi.org/10.1109/SEAA.2013.28>
- Ayers, J. (2016, decembrie 7). Top Museum Apps to bring you closer to art. *Appypie*. <https://www.appypie.com/top-museum-apps>
- Besant, H. (2016). The Journey of Brainstorming. *Journal of Transformative Innovation*, 2(1). <https://www.regent.edu/journal/journal-of-transformative-innovation/the-history-of-brainstorming-alex-osborn/>
- Freeman, J. (2019, august 8). What is an API? Application programming interfaces explained. *InfoWorld*. <https://www.infoworld.com/article/3269878/what-is-an-api-application-programming-interfaces-explained.html>
- IBM. (2020, martie 25). *Java Spring Boot*. <https://www.ibm.com/cloud/learn/java-spring-boot/>
- Juviler, J. (n.d.). What Is an API Endpoint? (And Why Are They So Important?). *HubSpot*. <https://blog.hubspot.com/website/api-endpoint>
- Khezami, K. (2020, noiembrie). *What Is Angular?* [Medium]. <https://medium.com/swlh/what-is-angular-3aad3b1046ec>
- Marinescu, P. C. (2014, noiembrie 9). Techne sau ars? *Vanitologia*. <https://vanitologia.wordpress.com/2014/09/11/techne-sau-ars/>
- MNAR. (f.a.). Aplicații. *Muzeul Național de Artă al României*. Preluat în 10 iunie 2022, din <https://www.mnar.arts.ro/exploreaza>
- O'Neil, B. (2017, august 13). Hashing Passwords in Java With BCrypt. *DZone*. <https://dzone.com/articles/hashing-passwords-in-java-with-bcrypt>

- Oprea, D. (2005). Determinarea cerințelor pentru noul sistem. În *Analiza sistemelor informaționale* (pp. 309–326). Editura Universității „Alexandru Ioan Cuza”.
- Rout, A. R. (2021, decembrie 28). Spring Boot JpaRepository with Example. *GeeksforGeeks*.
<https://www.geeksforgeeks.org/spring-boot-jparepository-with-example/>.
- Rout, A. R. (2022, februarie 18). Spring – Understanding Inversion of Control with Example. *GeeksforGeeks*. <https://www.geeksforgeeks.org/spring-understanding-inversion-of-control-with-example>.
- Schwaber, K., & Sutherland, J. (2020). *Ghidul Scrum*.
<https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-Romanian.pdf>
- Song, K. (2017, septembrie 24). Virtual reality and Van Gogh collide—Technology is turning museums into a booming industry. *CNBC*. <https://www.cnbc.com/2017/09/22/how-technology-is-turning-museums-into-a-booming-industry.html>
- Verma, R. (2020, iulie). Software Build Process – All You Need to Know! *DevOpsBuzz*.
<https://devopsbuzz.com/software-build-knowledge/>

Link-uri sursă ale tehnologiilor utilizate, menționate în subcapitolul 3.2.3:

<https://spring.io/projects/spring-boot>
<https://projectlombok.org/>
<https://mapstruct.org/>
<https://maven.apache.org/>
<https://angular.io/>
<https://getbootstrap.com/>
<https://www.mysql.com/products/workbench/>
<https://www.jetbrains.com/idea/>
<https://swagger.io/>
<https://www.postman.com/>
<https://www.zotero.org/>
<https://app.diagrams.net/>
<https://www.lucidchart.com/pages/>
<https://git-scm.com/>
<https://github.com/>

Anexe

Anexa 1. Interviu pentru elicitarea cerințelor de sistem

1. Întrebări pentru personalul administrativ al muzeului:

- Care sunt problemele pe care să le soluționeze aplicația? Ce cauze au provocat aceste probleme? Ce soluții se întrevăd?
- Care sunt caracteristicile pe care reprezentanții muzeului doresc să le includă aplicația?
- Care este intervalul de timp în care se dorește realizarea aplicației? Cât de flexibil și de realist este?
- Care este bugetul aferent proiectului? Este concordant cu valoarea estimată de specialiștii IT?
- Care sunt prevederile de confidențialitate?
- Se dorește mentenanța ulterioară a produsului?

2. Întrebări pentru personalul tehnic al muzeului:

- Care este suportul tehnic existent în prezent? Se au în vedere actualizări și îmbunătățiri, dacă va fi cazul?
- Ce sistem de baze de date este utilizat? Cum sunt structurate datele? Se pot migra în altă bază de date sau se preferă un serviciu REST?
- Există un server local și ce performanță are?
- Ce calificări are personalul angajat? Există specialiști care ar putea colabora la dezvoltarea aplicației?
- Care sunt cerințele funcționale? Cum și-ar dori să arate interfața? Ce opțiuni de navigare prin aplicație se doresc?
- Ce dificultăți tehnice ați întâmpinat până în prezent? Cum le-ați rezolvat?

3. Întrebări pentru potențialii utilizatori:

- În ce categorie de utilizatori v-ați încadra – vizitator al muzeului sau utilizator la distanță?
- Care ar fi cele mai utile elemente într-o aplicație de ghidaj în muzeele de artă?
- De pe ce dispozitive ați accesa cel mai des aplicația (telefon, laptop, tabletă)?
- Ce ar face aplicația mai ușor de utilizat?
- Vi se pare utilă posibilitatea de a salva anumite opere de artă în lista de favorite?

- Cât de complexe credeți că ar trebui să fie explicațiile despre operele de artă?
- Care dintre următoarele considerați că sunt cele mai importante cauze ce determină necesitatea unei astfel de aplicații: (1) dificultatea de a înțelege exponatele, (2) nevoia de digitalizare a muzeelor, (3) lipsa de promovare, (4) imposibilitatea de deplasare la muzeu, (5) nevoia utilizării bazelor de date pentru a ține evidența exponatelor, (6) accesul restricționat în perioadele de renovare?