



UNIVERSITATEA DIN
BUCUREȘTI

FACULTATEA DE
MATEMATICĂ ȘI
INFORMATICĂ



SPECIALIZAREA INFORMATICĂ

Lucrare de licență

APLICAȚIE WEB PENTRU GESTIONAREA EFICIENTĂ A ANGAJAȚILOR ȘI A SARCINILOR DE LUCRU

Absolvent

Alexandra-Diana Ciocan

Coordonator științific

Lect.dr. Natalia Gabriela Ozunu

București, iunie 2024

Rezumat

Un instrument de ajutor pentru orice echipă ce dorește să lucreze organizat este o aplicație de management a sarcinilor de lucru. Aceasta permite centralizarea datelor necesare îndeplinirii muncii de către fiecare membru al echipei și analiza modului de muncă atât individual, cât și la nivel de grup. Totuși, gestiunea datelor dintr-o astfel de aplicație trebuie să fie cât mai simplă pentru a nu distra atenția de la proiect către operații administrative.

Din acest motiv, această lucrare se concentrează pe crearea unei astfel de aplicații cu integrarea unei funcționalități specializate în asignarea automată, într-un mod eficient și echitabil, a atribuțiilor.

Abstract

A task management application is a helpful tool for any team that strives to work in an organized manner. It allows users to centralise the data needed for each team member to carry out their work and analyse how the team works both individually and as a group. However, data management in such an application should be as simple as possible so as not to distract attention from the project to administrative things.

For this reason, this paper focuses on the creation of such an application with the integration of functionality specialised in the automatic assignment of tasks in an efficient and fair way.

Cuprins

1	Introducere	4
1.1	Motivație	4
1.2	Domenii abordate	4
1.3	Analiza competitorilor	4
1.4	Nevoi îndeplinite de aplicație	5
1.5	Structura lucrării	5
2	Preliminarii	6
2.1	Funcționalități principale	6
2.1.1	Actori	6
2.1.2	Cazuri uzuale de utilizare	6
2.2	Alegerea tehnologiilor	7
2.2.1	Typescript&React	7
2.2.2	Java&Spring	8
2.2.3	PostgreSQL	8
3	Arhitectura aplicației	9
3.1	Microservicii	9
3.2	taskage-core	10
3.3	taskage-db	11
3.4	taskage-client	11
3.5	taskage-smart-helper	12
4	Implementarea sistemului	15
4.1	Securitate	15
4.2	Persistență	18
5	Utilizarea aplicației și funcționalități	20
5.1	Diagrama de activitate	20
6	Concluzii	21

Capitolul 1

Introducere

1.1 Motivație

Prin contactul meu de până la momentual acutal cu mediul corporate și, prin extensie, cu metodologia Agile, am observat că o piedică în maximizarea performanței unei echipe o constituie asignarea haotică a sarcinilor de lucru. Aceasta rezultă într-o lipsă de echilibru în volumul de muncă, împreună cu posibilitatea neîncheierii sarcinilor până la data limită a livrării.

Astfel, aplicația centrală acestei lucrări își propune simplificarea gestionării proiectelor din cadrul unei companii și creșterea productivității prin automatizarea procesului de atribuire a sarcinilor de lucru. Utilizatorii cheie sunt managerii, care pot crea panouri cu sarcini de lucru și distribui eficient volumul de muncă, și angajații obișnuiți, care își pot vizualiza sarcinile zilnice și oferi noutăți legat de statusul lor.

1.2 Domenii abordate

Această lucrare are în vedere, în principiu, domeniul dezvoltării aplicațiilor web și al inteligenței artificiale. În cadrul proiectării aplicației, am folosit tehnologii mature precum Java & Spring pentru back-end, Typescript & React pentru front-end și baze de date relaționale, prin intermediul PostgreSQL [de adăugat detalii după implementarea agentului inteligent]

1.3 Analiza competitorilor

Investigând cele mai populare aplicații web de gestiune a sarcinilor de lucru în cadrul unei echipe, competitorii principali identificați sunt: Jira, Monday.com și Azure DevOps.

Fiecare dintre acestea excelează în puncte diferite. Jira este foarte bine adaptat metodologiei Agile, constituind un sistem bun de organizare a unui workflow, Monday.com

oferă căi de comunicare pentru echipă, iar Azure DevOps include chiar și funcționalități de version control, facilitând CI/CD.

Atunci când vine vorba de automatizare, Jira și Monday.com pun la dispoziție un sistem bazat pe triggeri. Utilizatorul trebuie să configureze o regulă de forma "atunci când condiția ... este îndeplinită, execută ...". Astfel, e nevoie de efortul de analiză și decizie al utilizatorului pentru a configura toate aceste condiții. Azure DevOps, pe de altă parte, își axează posibilitățile de automatizare spre sarcinile manuale repetitive, în zona de CI/CD.

1.4 Nevoi îndeplinite de aplicație

Aplicația care este subiectul tezei își propune să construiască un algoritm inteligent care să scadă implicarea utilizatorului în procesul de decizie și analiză a informațiilor adunate, pentru a simplifica munca managerilor sau a Scrum Master-ilor, a crește echitabilitatea împărțirii volumului de muncă, și a crește per total mulțumirea și productivitatea în cadrul echipei.

Mai mult, această procesare a informației poate detecta și nivele neobișnuite de productivitate pentru un anumit angajat, pentru a fi ușor atât pentru acesta, cât și pentru manager, să depisteze semnale de alarmă pentru sănătatea mintală a angajului și să acționeze corespunzător.

1.5 Structura lucrării

[de adăugat descriere a fiecărui capitol după finalizarea lucrării]

Capitolul 2

Preliminarii

2.1 Funcționalități principale

2.1.1 Actori

Principalii actori ai cazurilor de utilizare sunt:

- Membru de echipă
- Manager
- Administrator

2.1.2 Cazuri uzuale de utilizare

- Manager:

1. În calitate de manager, aş vrea să adaug sarcini de lucru pentru echipa mea, cu detalii despre criteriile de acceptanță, prioritate, dificultate și termen limită, pentru ca acestea să fie cât mai clare pentru membri echipei.
2. În calitate de manager, aş vrea să pot vizualiza factorii de performanță precum disponibilitatea, volumul de muncă, calificarea, eficiența și atribuțiile fiecărui membru al echipei mele, pentru a evalua corect atribuirea de sarcini de lucru noi.
3. În calitate de manager, aş vrea să pot modifica factorii de performanță precum disponibilitatea, volumul de muncă, calificarea, eficiența și atribuțiile fiecărui membru al echipei mele, pentru ca aceștia să fie mereu actuali.
4. În calitate de manager, aş vrea să atribui automat sarcini de lucru, bazat pe detaliile sarcinii de lucru și pe factorii de performanță angajatului, pentru a le împărți cât mai echitabil și a maximiza productivitatea și șansele ca sarcina să fie îndeplinită până la termen.
5. În calitate de manager, aş vrea să pot atribui manual sarcini de lucru, bazat pe analiza mea asupra situației, pentru a ajusta eventualele erori ale recomandă-

rilor, a adapta panoul la noi informații relative la disponibilitate, sau a încuraja creșterea/scăderea productivității pentru un anumit membru al echipei

- Employee:
 1. În calitate de angajat obișnuit, aş vrea să îmi vizualizez sarcinile de lucru împreună cu detalii despre criteriile de acceptanță, prioritate, dificultate și termen limită, pentru a îmi putea organiza activitatea zilnică.
 2. În calitate de angajat obișnuit, aş vrea să pot actualiza statusul sarcinilor de lucru pentru ca managerul meu să le cunoască progresul.
 3. În calitate de angajat obișnuit, aş vrea să fiu notificat atunci când managerul îmi atribuie o sarcina de lucru nouă, pentru a percepe mai ușor schimbări ale panoului cu sarcinile mele.
- Administrator:
 1. În calitate de administrator, aş vrea să pot crea, actualiza sau elimina utilizatori și echipe, pentru a putea gestiona accesul la platformă și organizarea internă.
 2. În calitate de administrator, aş vrea să pot atribui manageri echipelor create, pentru ca fiecare echipă să fie gestionată de un manager.

2.2 Alegerea tehnologiilor

2.2.1 Typescript&React

Pentru partea de front-end a aplicației web am ales să utilizez Typescript ca limbaj și React ca bibliotecă, împreună cu componente reutilizabile din biblioteca de componente UI Ant Design.

Utilizarea limbajului Typescript, în defavoarea limbajului JavaScript, este explicată prin minimizarea erorilor cauzate de inconsistența adusă de faptul ca JavaScript este un limbaj „loosely typed” („dinamicamente tipat”) și de incertitudinea adusă de acest fapt. Astfel, prezența interfețelor și tipurilor de date bine stabilite facilitează dezvoltarea rapidă și stabilă.

În plus, biblioteca React aduce o perspectivă structurată dezvoltării interfeței, prin definirea de componente reutilizabile în cadrul aplicației. O parte din componentele UI sunt utilizate din biblioteca Ant Design, folosite prin aspectul vizual unitar și formal, acompaniat de o documentație bine pusă la punct. Un dezavantaj al abordării prin componente este comunicarea între acestea, motiv pentru care am folosit state managerul MobX.

2.2.2 Java&Spring

Pentru partea de back-end a aplicației web am optat pentru limbajul Java și framework-ul Spring și ecosistemul acestuia, având în vedere opțiunile robuste pentru gestionarea logicii de afaceri, a securității și a interacțiunilor cu baza de date.

Versiunea de Java utilizată este Java 21, pentru a avea acces la cele mai noi update-uri ale limbajului, cu cele mai noi feature-uri, precum clase Record, fire virtuale de execuție, și Sequenced Collections.

Versiunea de Spring folosită este 6.1.3, ce vine împreună cu Spring Boot 3.2.2, alături de alte dependențe, printre cele mai notabile numărându-se Spring Security și Jakarta Persistence API (JPA).

Cele mai majore utilități puse la dispoziție de Spring și de ecosistemul de dependențe sunt:

- securitatea asigurată prin generarea de JSON Web Tokens pentru autentificarea și autorizarea utilizatorilor și a cererilor HTTP trimise;
- crearea tabelelor și conectarea la baza de date facilitată de ORM-ul JPA;
- injectarea dependențelor prin constructorii claselor, în scopul reducerii codului boilerplate.

[detaliere spring boot, spring security, spring jpa]

2.2.3 PostgreSQL

Pentru persistența datelor am ales PostgreSQL, un sistem open-source de gestiune pentru baze de date, pentru sistemul său robust de asigurare a integrității datelor și abilitatea de a manevra ușor cantități mari de date.

Capitolul 3

Arhitectura aplicației

3.1 Microservicii

Microserviciile sunt o arhitectură care presupune segmentarea unei aplicații într-o suită de servicii specializate, independente, ce comunică prin protocoale light-weight, în mod popular transmițând date prin HTTP într-un context RESTful. Unul din avantajele principale ale acestei arhitecturi este decuplarea serviciilor implicate. Eșecul unui serviciu nu afectează întreaga aplicație, astfel încât problemele se manifestă izolat și sunt mai ușor de controlat. Un alt motiv pentru care am considerat că această arhitectură este potrivită aplicației este versatilitatea permisă în alegerea tehnologiilor și a limbajelor de programare.

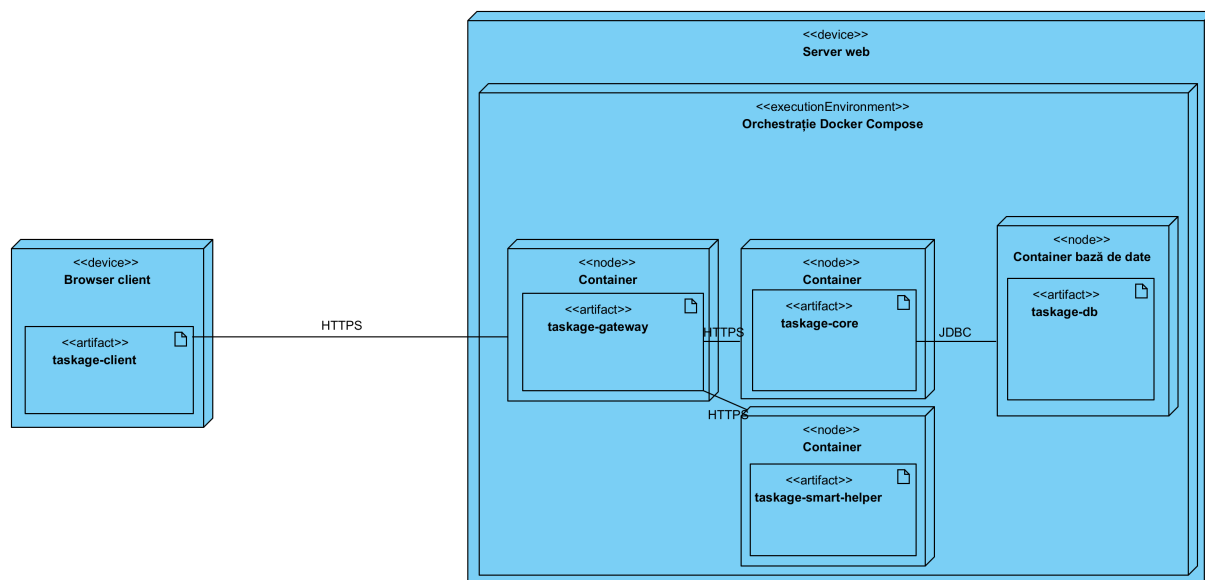


Figura 3.1: Arhitectura high-level

Figura 3.1 ilustrează caracteristica modulară a aplicației. Pe partea de front-end, artefactul aplicației React `taskage-client` rulează în browser-ul web al utilizatorului prin

codul transpilat din TSX în JS. Acesta comunică cu serverul prin apeluri HTTP, utilizând componenta Spring Cloud taskage-gateway ca punct de intrare către funcționalitățile back-end. Serviciile de back-end comunică la rândul lor prin protocolul HTTP, iar conexiunea la baza de date este facilitată de JDBC.

[de completat legaturile taskage-smart-helper cu bd dupa ce ma hotarasc daca fac alt db sau continui pe acelasi]

3.2 taskage-core

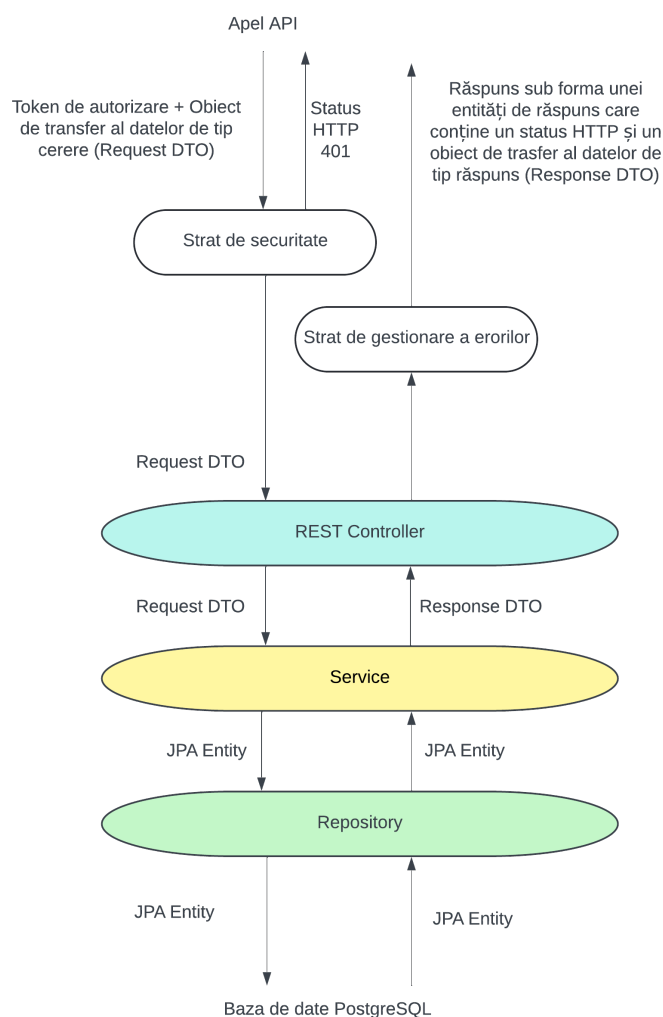


Figura 3.2: Fluxul datelor în componenta core

Componenta care se ocupă cu funcționalitățile cele mai de bază ale aplicației este taskage-core. Aici se realizează operațiile CRUD asupra entităților și se gestionează cererile uzuale ale platformei. Structura componentei urmează patternul Controller-Service-Repository, popular în aplicațiile Spring Boot. Acest model de structurare a proiectului

urmărește separarea sarcinilor unui sistem mare în părți mai ușor de manevrat. Fluxul datelor, reprezentat în figura 3.2, este controlat, iar fiecare strat are un scop bine definit.

Apelurile HTTP conțin un JWT token în header și trec prin filtrul de autorizare, implementat cu ajutorul Spring Securit. Dacă tokenul este aprobat ca fiind valabil și având criptat rolul corespunzător, apelul ajunge la end-point-ul asociat din controller. Altfel, aplicația întoarce un răspuns cu status HTTP 401(Unauthorized). Controllerele constituie API-ul aplicației și ca au ca unic scop expunerea unor funcționalități către agenți externi. Datele sunt primite sub forma unui obiect de transfer al datelor(Request DTO), care nu se traduce într-o entitate a bazei de date, fiind doar un mod de transmitere a informației între microservicii. Aceste DTO-uri trebuie de asemenea să respecte validările impuse câmpurilor prin Jakarta Validation.

Dacă DTO-ul a fost validat, acesta este trimis mai departe pentru procesare în zona cu logica de afaceri a aplicației, și anume serviciile. Serviciile procesează informația și eventual o mapează ca entități JPA spre a o transmite mai departe depozitelor(repositories). Depozitele reprezintă stratul de acces al datelor și se ocupă cu toate interacțiunile cu baza de date. Dacă totul se realizează cu succes, se va returna o entitate de răspuns cu statusul HTTP 200, eventual conținând un Response DTO cu datele cerute. Dacă se aruncă vreo excepție, aceasta va fi prinsă de clasa care se ocupă cu gestiunea erorilor la nivel global, GlobalExceptionHandler, și se va trimite înapoi un răspuns sugestiv.

3.3 taskage-db

Nu numai interacțiunea cu baza de date este gestionată de Spring JPA, ci și schema acesteia, prin utilizarea unor configurări specifice. Astfel, fiecare clasă adnotată „@Entity” devine un tabel din diagrama 3.3, prin intermediul instrumentului de persistare a datelor Hibernate și al metadatelor oferite prin adnotări.

Tabela „app_users” are câmpul special „auth_role” care poate lua una din valorile „ROLE_BASIC”(asociat membrilor normali ai unei echipe), „ROLE_MANAGER”(asociat managerilor echipelor) și „ROLE_ADMIN”(asociat administratorului paginii), urmând formatul înțeles de Spring Security. Speciale mai sunt tabelele de tip dicționar „priorities” și „statuses”, care nu implementează operații CRUD, ci servesc la stocarea unor date care altfel ar fi fost stocate în structuri de tip enum. Această alegere ajută la organizarea eficientă a codului și permite manevrarea mai ușoară a valorilor, putând fi chiar și actualizate la runtime.

3.4 taskage-client

Componenta care constituie front-end-ul aplicației este taskage-client. Aceasta este un proiect React care folosește MobX pentru gestionarea stării. Una din provocările prin-

cipale la construcția unui proiect în React este lipsa de structură interentă tehnologiei. Prin comparație cu un framework popular, Angular, proiectele React nu au o structură de fișiere și un flux de date bine-definite, acestea rămânând responsabilitatea dezvoltatorului. Cele două mari abordări pentru structurarea aplicației sunt: abordarea orientată pe categorie și abordarea orientată spre funcționalitate. Prima din cele două este mai puțin scalabilă, organizând fișierele după tipul lor (componente, pagini, modele, etc...) în timp ce a doua abordare rafinează mai departe structura, sortând componentele în funcție de funcționalitatea efectivă pe care o implementează. Am ales cea de-a doua alternativă pentru că densitatea componentelor devenise prea mare pentru a putea fi gestionate la un loc.

Deși front-end-ul reprezintă un SPA (Single Page Application), încărcarea paginilor se face pe baza rutelor pentru eficiență. Rutele sunt atât publice, cât și private. Cele private se pot accesa pe baza unor criterii precum tipul utilizatorului, iar în cazul lipsei privilegiilor de acces, acestea vor redirecționa utilizatorul către o pagină cu un mesaj sugestiv. Rutele servesc pagini, implementate în folderul „pages”, iar paginile randează diverse componente care, dacă au logică complexă, sunt plasate într-un folder cu nume sugestiv funcționalității de care țin, în folderul „features”. Folderul „components” este rezervat pentru componente reutilizabile, generice.

Stocarea datelor necesare în componente aflate la distanță mare pe ierarhia din DOM se face în „stores”, concept specific MobX. Tot aici se fac și apelurile către server și gestionarea datelor utilizatorului spre setarea corespunzătoare a tokenului în header-ul cererilor.

3.5 taskage-smart-helper

[de adăugat capitol despre microserviciul python după ce are o structură bine definită]

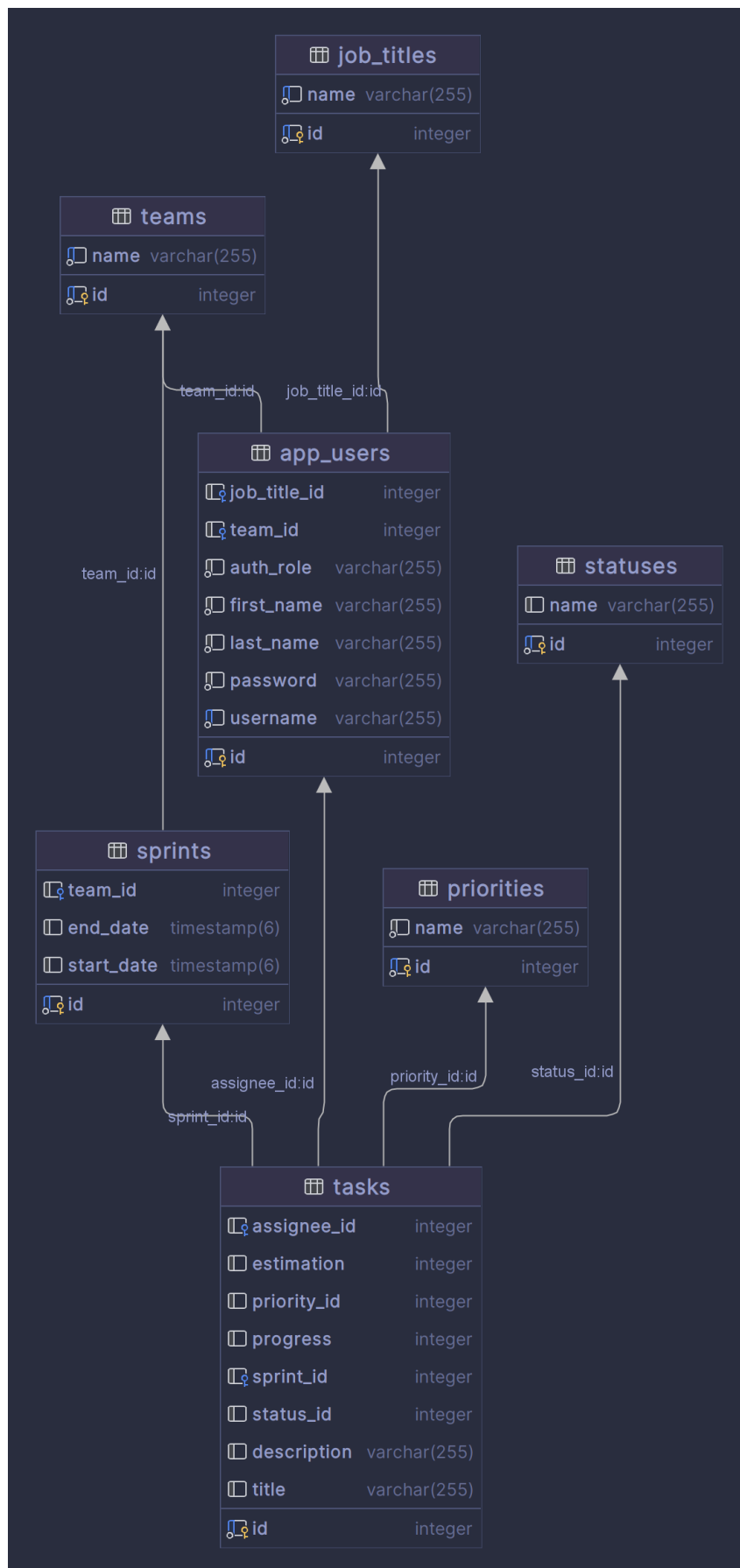


Figura 3.3: Diagrama bazei de date

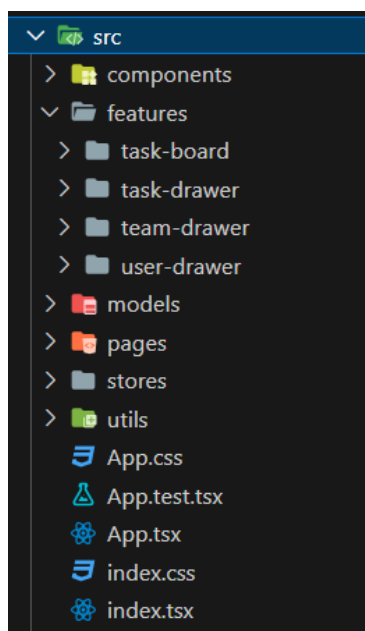


Figura 3.4: Structura de fișiere a aplicației React

Capitolul 4

Implementarea sistemului

4.1 Securitate

Aplicația este securizată din două perspective: taskage-core implementează o componentă personalizată de rută privată, în timp ce endpoint-urile REST expuse de către componenta taskage-core sunt securizate utilizând modulul de securitate pus la dispoziție de către Spring, anume Spring Security.

Ruta privată a modului front-end, observabilă în figura 4.1, primește ca prop rolul care poate accesa ruta respectivă pentru a putea utiliza componenta în contextul oricărui rol. Aceasta folosește un hook de tip `useEffect` pentru a detecta schimbarea paginii și a utilizatorului logat. Pentru oricare din aceste schimbări, încarcă un ecran de loading până verifică accesul, apoi, după ce metoda `checkAccess` determina asincron răspunsul, identifică dacă utilizatorul nu are acces la pagină pentru că nu este logat sau pentru că rolul său nu are privilegii la pagina respectivă. Dacă accesul este oferit, se poate randa un outlet în care să se afișeze pagina cerută, altfel utilizatorul este redirecționat către o pagină cu mesaj sugestiv, precum cel din figura 4.2.

În cadrul aplicației, în urma conectării cu succes al unui utilizator, pe bază de nume de utilizator și parolă, back-end-ul eliberează un JSON Web Token (JWT), pe baza căruia cererile HTTP făcute ulterior de către client pot fi autorizate. Mai exact, în urma autentificării cu succes a utilizatorului se va crea un JWT, semnat de către aplicație, căruia îi sunt atașate rolurile utilizatorului autentificat. Un astfel de rol este de forma „ROLE_[nume_rol]”, unde [nume_rol] poate fi orice denumire aleasă, însă, în cadrul proiectului avem trei tipuri de utilizatori: „ROLE_BASIC”(asociat membrilor normali ai unei echipe), „ROLE_MANAGER”(asociat managerilor echipelor) și „ROLE_ADMIN”(asociat administratorului paginii), fiecare având stocat în baza de date propriile privilegii și permisiuni, sub formă de roluri.

[de adăugat o diagrama cu filtrul prin JWT]

În plus, JWT-ul eliberat de către serviciul Spring are și o perioadă predefinită de

```

const PrivateRoute = observer(
  ({ allowedAuthRole }: { allowedAuthRole: String }) => {
    const location = useLocation();
    const [isAllowed, setIsAllowed] = useState(false);
    const [loading, setLoading] = useState(true);
    const [navigateToRoute, setNavigateToRoute] = useState<string>("");

    useEffect(() => {
      const checkAccess = async () => {
        if (userStore.currentUser) {
          const currentUser = userStore.currentUser;
          if (currentUser.user.authRole === allowedAuthRole) {
            setIsAllowed(true);
          } else {
            setNavigateToRoute(UNAUTHORIZED_ACCESS_LINK);
          }
        } else {
          setNavigateToRoute(NOT_AUTHENTICATED_LINK);
        }
        setLoading(false);
      };

      checkAccess();
    }, [userStore.currentUser, location]);

    return loading ? (
      <Spin />
    ) : isAllowed ? (
      <Outlet />
    ) : (
      <Navigate to={navigateToRoute} replace />
    );
  }
);

export default PrivateRoute;

```

Figura 4.1: Componenta PrivaterRoute

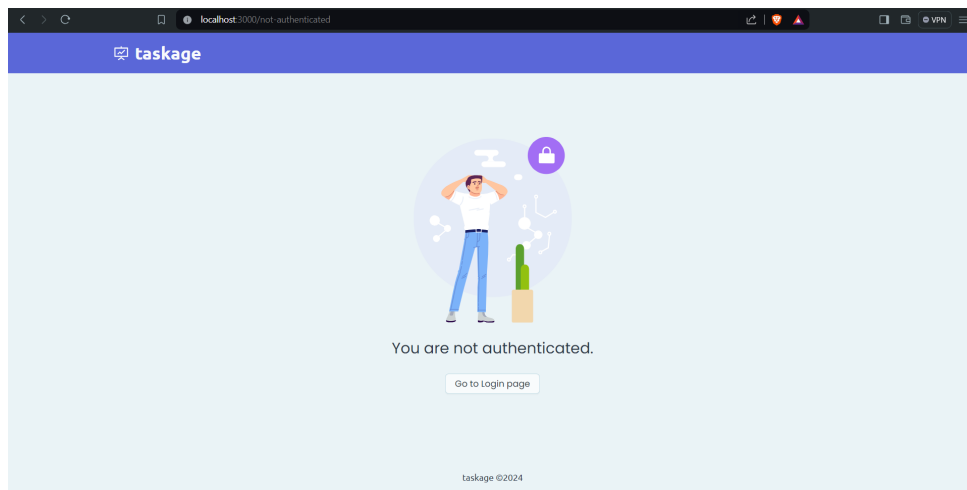


Figura 4.2: Pagina de redirectionare pentru utilizatorii neautentificați

existență („time to live”), configurat la compilare.

De asemenea, modul în care este configurat modulul Security se află într-o clasa adnotată cu „@Configuration”, în cadrul căreia suprascriem metode precum security-FilterChain, passwordEncoder și corsConfigurationSource, pentru a configura gradul de securitate aplicat end-point-urilor REST. Adnotările facilitează munca dezvoltatorului, constiuind un mod de a asocia metadata diferitelor clase, pe care JVM-ul le va interpreta pentru compilare. SecurityFilterChain determină filtrarea securizată a cererilor pe baza token-ului, passwordEncoder criptează parolele înainte de salvarea în baza de date, iar corsConfigurationService limitează accesul cross-origin pentru a bloca cererile care vin de pe IP-uri care nu corespund clientului nostru.

```
@Configuration
@EnableWebSecurity
@RequiredArgsConstructor
@EnableGlobalMethodSecurity(securedEnabled = true)
public class SecurityConfig {
```

Figura 4.3: Adnotările clasei SecurityConfig

În plus, tokenul de autorizare și datele utilizatorului curent sunt stocate în localStorage-ul browserului spre autentificarea automată. După părăsirea paginii, la o nouă accesare, se verifica dacă credențialele se află în localStorage și dacă token-ul este încă valid, print-un apel suplimentar către backend care decriptează token-ul și verifică time to live-ul setat anterior.

[de intrebat daca e nevoie sa detaliez cu code snippets din spring security]

4.2 Persistență

Atât crearea schemei bazei de date, cât și interacțiunea cu baza de date, s-au realizat prin ORM-ul (Object Relational Mapping) oferit de Spring, anume Spring JPA (Jakarta Persistence API). Am optat pentru această opțiune prin configurarea specifică Hibernate, unde `ddl-auto` poate fi setat pentru transformarea automată a modelelor adnotate „@Entity” în scripturi SQL:

```
spring.jpa.hibernate.ddl-auto=create
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.properties.hibernate.format_sql=true
```

Figura 4.4: Configurările JPA

Acesta oferă un nivel de abstractizare peste strat-ul de SQL, permițând crearea de tabele, relații și constrângeri prin adnotări, precum „@Entity” pentru a semnala că această clasă reprezintă de fapt un model pentru tabela din baza de date, „@Id” pentru a semnala că un câmp dintr-o clasă asociată unei tabele este cheie primară, și altele asemenea. Spre exemplu, clasa „Teams” din figura 4.5 beneficiază de metadata ce descriu structura viitoare tabele din baza de date. După cum se observă, JPA ne oferă posibilitatea de a seta algoritm de generare automată a cheilor primare, de a crea coloane read-only, de a adăuga constrângeri atât banale, precum nullable și unique, cât și constrângeri relaționale precum @OneToMany. Un alt avantaj este specificarea metodei de aducere a datelor, care poate fi LAZY sau EAGER. Având în vedere că lucrăm cu o aplicație web, de ajutor mai este adnotarea @JsonIgnore care ne permite să manipulăm forma obiectului trimis prin HTTP.

Mai departe, interacțiunea cu baza de date este simplificată, fiind mai prietenoasă cu utilizatorul decât standardul Java, ce implică clasa JDBCTemplate, prin intermediul căreia ai scrie SQL nativ. În schimb, utilizând JPA, procesul este vast simplificat, având ca premise doar crearea unei interfețe ce derivează una deja existentă, generică, anume JpaRepository<TIP, TIP_ID>.

În cadrul acestei metode, Spring JPA permite definirea de metode doar descriind acțiunea necesară. Spre exemplu, pentru a găsi toți utilizatorii dintr-o echipă anume, am putea specifica o metodă cu o semnătură precum: „findAllByTeam(String)”, iar implementarea ar fi rezolvată de către ORM-ul utilizat, astfel, oferind una dintre cele mai prietenoase API-uri pentru a interacționa cu baza de date.

```

@Entity
@Table(name = "teams")
public class Team {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(updatable = false)
    private Integer id;

    @Column(nullable = false, unique = true)
    private String name;

    @OneToMany(mappedBy = "team", fetch = FetchType.LAZY)
    private Set<User> users = new HashSet<>();

    @OneToMany(mappedBy = "team", fetch = FetchType.LAZY, cascade =
        CascadeType.ALL)
    @JsonIgnore
    private Set<Sprint> sprints = new HashSet<>();
}

```

Figura 4.5: Configurările JPA

Capitolul 5

Utilizarea aplicației și funcționalități

5.1 Diagrama de activitate

[de realizat diagrama de activitate pentru fluxurile uzuale]

Capitolul 6

Concluzii