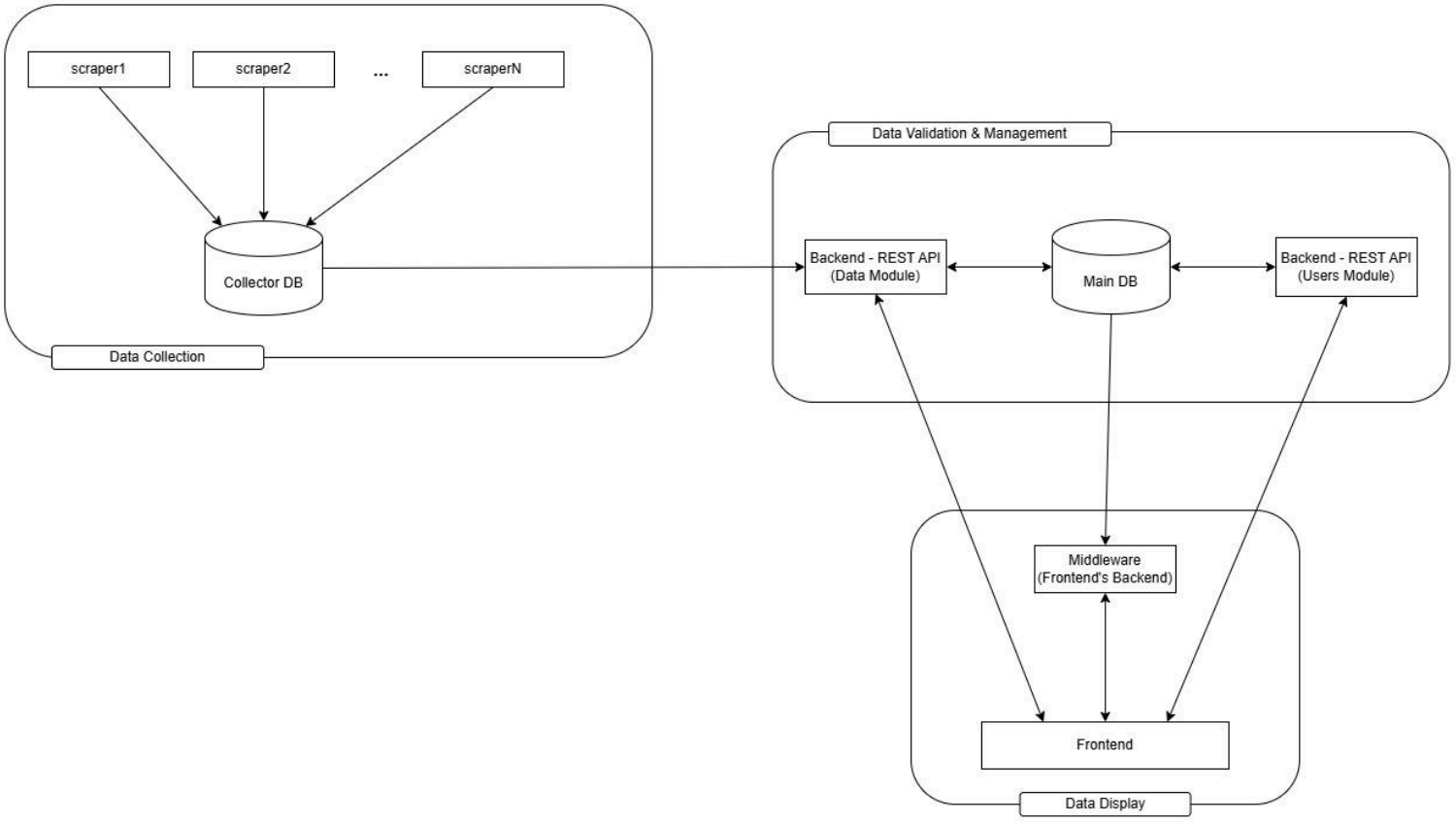
Diagrama Generală



Proiectul este împărțit în trei grupe principale, fiecare cu tehnologii diferite:

- **Data Collection:** Grupa responsabilă pentru colectarea datelor de zbor și stocarea lor într-o bază de date. Mai multe scrapere rulează în paralel pentru a colecta date de pe mai multe site-uri de zboruri. Pentru această grupă folosim Python, unul dintre cele mai facile limbaje pentru scraping.
- **Data Validation & Management:** Grupa responsabilă de manipularea datelor colectate anterior, aplicând validări, filtrări și actualizări. Datele din baza anterioară sunt procesate și mutate într-una principală. Această grupă reprezintă totodată și backend-ul aplicației, folosit și pe post de REST API. Pentru această grupă folosim Java, Spring Boot, o soluție robustă pentru acest tip de aplicație.
- **Data Display:** Prin accesări directe la baza de date (intermediate totuși printr-un “backend al frontend-ului” (middleware) sau prin apelări la REST API, afișăm datele colectate utilizatorilor. Pentru această grupă folosim C#, DevExpress, care dispune de o interfață foarte ușor de manipulat.

Grupa **Data Validation & Management** poate doar să citească informații din baza de date colectoare, nu poate interacționa cu scraperele sau modifica direct în baza colectoare datele.

Modulul de useri este responsabil de autentificarea și autorizarea utilizatorilor, utilizând propriul sistem de generare de JWT tokens.

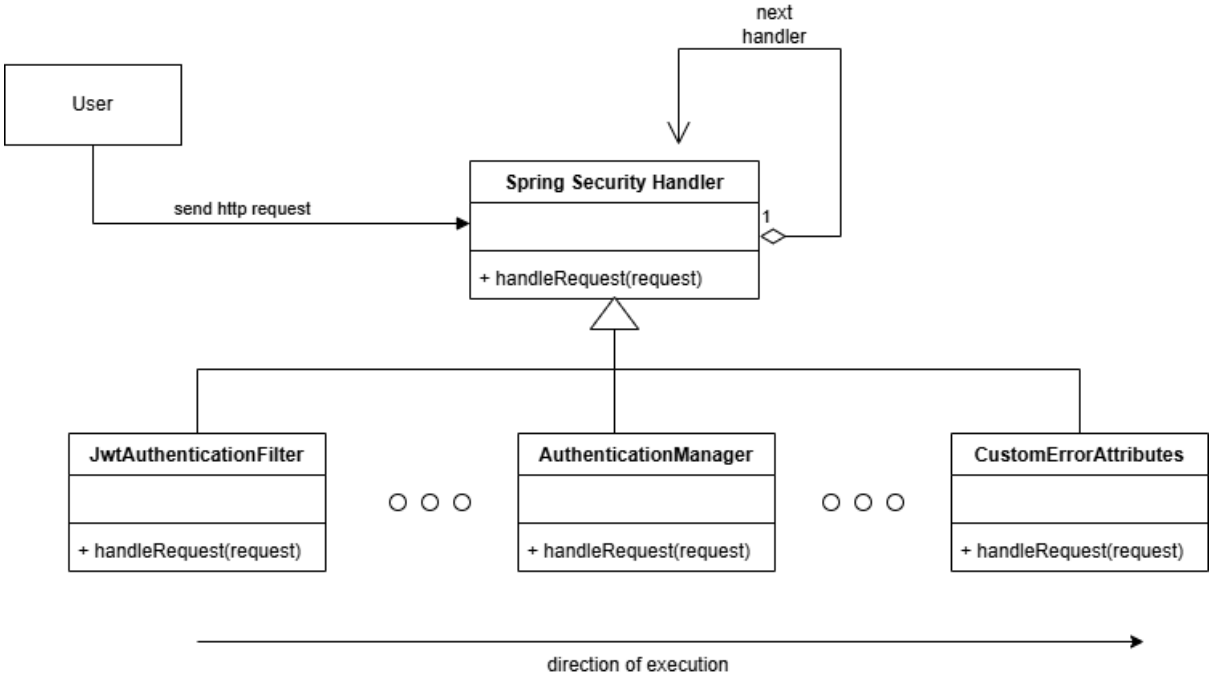
Modulul de date este responsabil pentru parsarea datelor colectate și oferirea lor utilizatorilor în formate acceptabile.

Design Patterns

Chain of Responsibility

Spring Security by default are implementat un chain of responsibility pentru a trata request-urile web. Aplică filtre, procesează requestu, face logging, și triggeruiește evenimente terțe.

În diagrama de mai jos este o abstractizare (un overview) a ce se întâmplă de fapt în aplicație, adică funcțiile pot să varieze ca denumire și Spring Security are un sistem mai complex de a insera next handler, dar în principiu fiecare concrete handler are o funcție echivalentă cu nextHandler() pentru a avansa în chain.



Request-ul este primit de handler-ul de Spring Security și trimis către filtre. În aplicația noastră, am customizat trei pași importanți în chain of responsibility pentru a realiza autentificarea.

JwtAuthenticationFilter este un filtru care se execută în faza incipientă a chain-ului, și are rolul de a extrage token-ul din headere și a seta în SecurityContext informațiile extrase din acesta. Dacă token-ul este invalid, chain-ul se oprește aici.

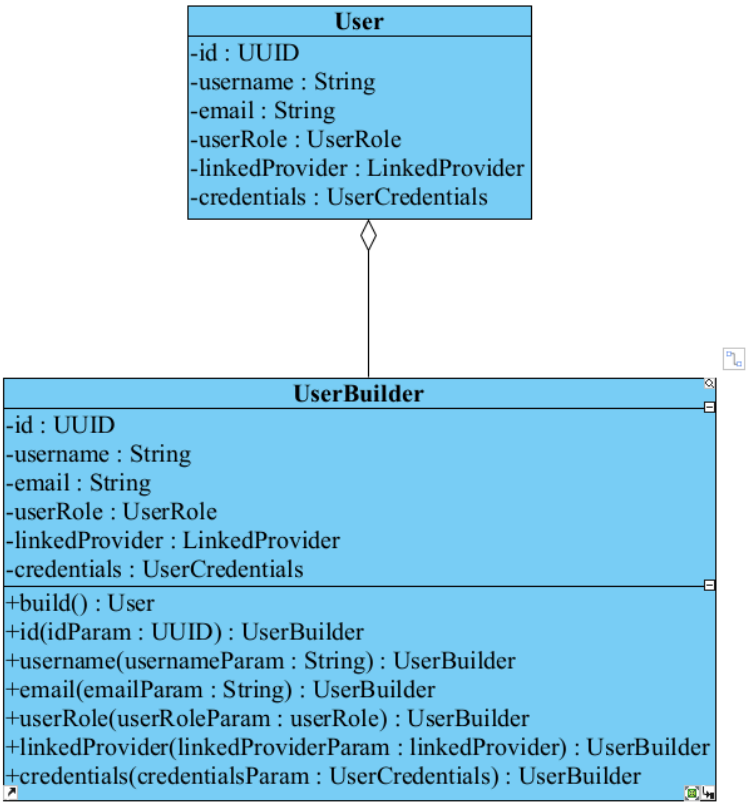
AuthenticationManager este folosit pentru a valida tokenuri externe (care nu sunt generate din aplicația noastră).

CustomErrorAttributes este folosit în caz de erori aruncate de-a lungul chain-ului și a trimite mai departe un response valid. Noi îl folosim pentru a prinde erori cu status code 500, și a masca eroarea neprevăzută cu un hash, astfel neoferind user-ului informații despre codul intern.

Punctele dintre JwtAuthenticationFilter, AuthenticationManager și CustomErrorAttributes semnifica existența unor alte filtre default implementate de Spring Security.

Builder

Odată cu crearea un obiect complex, un developer poate avea nevoie de parametrii opționali la acesta. Una dintre potențialele soluții este aplicarea unui Design Pattern deja cunoscut cum ar fi Builder pattern.



În proiectul nostru, unul dintre obiectele care a avut nevoie de Builder pattern a fost obiectul de User. După cum se poate observa, obiectul de tip User are mai multe atribute care nu sunt necesare la crearea obiectului, prin urmare am decis să implementăm un builder care ne ajuta, la nevoie, să mai adăugăm alte atribute opționale precum și control asupra celor deja existente. Un mare avantaj al acestui pattern este faptul că permite adăugarea a unui număr extins de atribute, fără a mai implementa pentru fiecare atribut un Constructor separat. Un potențial dezavantaj este acela că, Builder pattern adaugă complexitate codului care îl poate face mai greu de citit pentru alți developeri.

În diagrama de mai sus, vedem cum avem clasa User care cuprinde mai multe atribute. Pentru a putea adăuga atribute opționale, ne folosim de clasa UserBuilder care are cate o metoda pentru inițializarea fiecărui atribut prezent în clasa inițială.

Singleton

Fără doar și poate, cel mai utilizat Design Pattern în proiectul nostru este Singleton-ul. Acesta este implementat la toate nivelele în Spring Boot, în special în arhitecturile REST precum cea a proiectului nostru. Convenția arhitecturilor REST presupune împărțirea aplicației în cel puțin 4 niveluri: Model, Controller, Service, Repository. Dintre acestea implicit toate, cu excepția Modelului, sunt Singleton-uri. Controller-ul controlează și mapeaza rutele din API, Service-ul menține logic bussiness-ul aplicației într-un singur loc, iar Repository este folosit ca Middleman între API și Baza de Date. Obiectele adnotate cu “@Component”, “@Service”, “@RestController” ș.a.m.d sunt păstrate de Spring Boot intern ca o singură instanță care ulterior poate fi injectată în alte clase, fără a se creea obiecte noi.
Avantajele folosirii unui Singleton este aceea ca menține obiectele create (pentru care existența mai multor instanțe nu are sens) unice, ceea ce ajută la gestionarea resurselor și a timpilor de set-up. Standard-ul de getInstance() este înlocuit cu dependency injection-ul realizat de Spring Boot.