



AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

Wydział Fizyki i Informatyki Stosowanej

Projekt dyplomowy

**Zastosowanie metodyki Test Driven Development do tworzenia
aplikacji internetowych opartych na Java Spring**

**Test Driven Development of web applications
based on Java Spring**

Autor:
Kierunek studiów:
Opiekun pracy:

Hleb Shypula
Informatyka Stosowana
dr hab. inż. Maciej Wołoszyn

Kraków, 2021

Spis treści

1. Wprowadzenie	4
2. Wstęp teoretyczny	5
2.1 Definicja TDD.....	5
2.2 Wymagania TDD	6
2.3 Cykl TDD	6
2.4 Wzorce TDD	8
2.4.1 Sfałszuj to (ang. <i>Fake It</i>)	9
2.4.2 Triangulacja (ang. <i>Triangulate</i>)	9
2.4.3 Oczywista implementacja (ang. <i>Obvious Implementation</i>).....	10
2.4.4 Najpierw operator assert (ang. <i>Assert first</i>)	10
2.5 Dlaczego TDD	11
2.6 Fazy testowania.....	13
2.6.1 Testowanie jednostkowe.....	13
2.6.2 Testowanie integracyjne	13
2.6.3 Testowanie manualne	14
3. Wykorzystane technologie i narzędzia.....	15
3.1 Strona serwera.....	15
3.1.1 Java	15
3.1.2 Spring Framework	15
3.1.3 Spring Boot.....	16
3.1.4 PostgreSQL	17
3.2 Testowanie.....	17
3.3 Strona klienta.....	18
4. Opis aplikacji	20
4.1 Charakterystyka problemu	20
4.2 Wymagania.....	22
5. Implementacja	23
5.1 Podstawa modelu	23
5.2 Test integracyjny – warstwa danych.....	29
5.3 Rozszerzenie modelu	31
5.4 Wprowadzenie zmian	33
5.5 Bezpieczeństwo aplikacji	34

5.6 Problem edycji danych	38
5.7 Dodawanie pracowników	49
5.8 Test integracyjny – warstwa serwisu	49
5.9 Test integracyjny – warstwa kontrolera.....	52
5.10 Ostateczny design	54
5.11 Testy	59
6. Podsumowanie.....	60
6.1 Wnioski.....	60
6.2 Możliwości rozwoju	61
Załączniki	62
Bibliografia.....	63
Spis ilustracji	64
Spis listingów	65

1. Wprowadzenie

Ambicją porządnego dewelopera jest elegancki, elastyczny i zrozumiały kod, który można łatwo modyfikować, który działa poprawnie i nie sprawia przykrych niespodzianek podczas tworzenia oprogramowania. Aby zrealizować ten zamysł, można przygotować testy programu, zanim zostanie on napisany. To właśnie ta paradoksalna idea stanowi podstawę metodyki Test-Driven Development (programowanie sterowane testami). Czy to jest nielogiczne? Nie trzeba się śpieszyć z wyciąganiem pochopnych wniosków.

Głównym celem niniejszej pracy było napisanie aplikacji internetowej opartej na Java Spring przy zastosowaniu metodyki Test-Driven Development. A mianowicie:

- przedstawienie zasadniczych cech oraz wzorców metodyki Test-Driven Development;
- zdobywanie osobistego doświadczenia w programowaniu sterowanym testami;
- sprawdzenie czy poziom wejścia do tej metodyki jest akceptowalny dla programisty niezbyt doświadczonego w jakimkolwiek testowaniu.

Temat „Zastosowanie metodyki Test Driven Development do tworzenia aplikacji internetowych opartych na Java Spring” ma na celu wskazać Czytelnikowi, że w odróżnieniu od XP (*Extreme Programming*) metodyka TDD nie jest absolutna. XP mówi: „Mamy rzeczy, które powinniśmy opanować, zanim przejdziemy do kolejnego etapu”. Natomiast TDD – to nie tak konkretna metodyka. TDD zakłada, że podczas programowania w pełni wyobrażamy sobie odległość od momentu podjęcia decyzji o designie do etapu kontroli jakości otrzymanych wyników¹.

Główną tezę niniejszego opracowania jest stwierdzenie, że dzięki TDD kod będzie miał dużo lepszą jakość. Jeżeli tak, to napisanie kodu wymaganego dla pozytywnego wyniku testów, produkuje tezę podrzędną – programista będzie w stanie codziennie dostarczać w pełni działający produkt wraz z dodaną nową funkcjonalnością. Dzięki temu współpraca z klientem osiągnie nowy poziom.

¹ K. Beck, *TDD. Sztuka tworzenia dobrego kodu*, Helion 2014, s. 10.

2. Wstęp teoretyczny

2.1 Definicja TDD

Programowanie sterowane testami (ang. test-driven development, TDD) – technika tworzenia oprogramowania, polegająca na powtórzeniu bardzo krótkich cykli projektowania: najpierw pisze się test, pokrywający żądane zmiany, następnie pisze się kod potrzebny, aby test zakończył się sukcesem (“przeszedł”) i w końcu przeprowadza się refaktoryzację nowego kodu, żeby spełniał odpowiednie standardy. Kent Beck, uznawany za wynalazcę tej techniki, argumentował w 2003 r., że programowanie oparte na testach zachęca do prostego designu i wzbudza wiarę w siebie (ang. *inspires confidence*)².

W 1999 roku, kiedy TDD pojawiło się po raz pierwszy, technologia była ściśle związana z koncepcją *test-first* (najpierw test), stosowaną w programowaniu ekstremalnym, ale później pokazała się jako niezależna metodyka. TDD różni się od innych metod tym, że łączy programowanie z pisanem testów przez tego samego programistę. Ta koncepcja odnawia powszechny szacunek do testów tworzonych przez dewelopera³.

Test to procedura, która pozwala uzyskać potwierdzenie, że kod działa w spodziewany sposób (lub informację, że wymaga poprawienia). Testy funkcjonalne pozwalają zobaczyć funkcjonalność aplikacji z punktu widzenia użytkownika. Oznacza to, że takie testy mogą być rodzajem specyfikacji aplikacji. Można wydzielić: testy czarnej skrzynki, testy w przeglądarce, testy behawioralne, testy wydajnościowe. Natomiast tzw. testy jednostkowe testują aplikację od środka, z punktu widzenia programisty. Testowanie wtedy odbywa się w celu upewnienia się, że pojedyncze obiekty (moduły, podsystemy) działają poprawnie.

² <https://bit.ly/3xRUGCE>, *Czym jest TDD*, [dostęp: 14.08.2021].

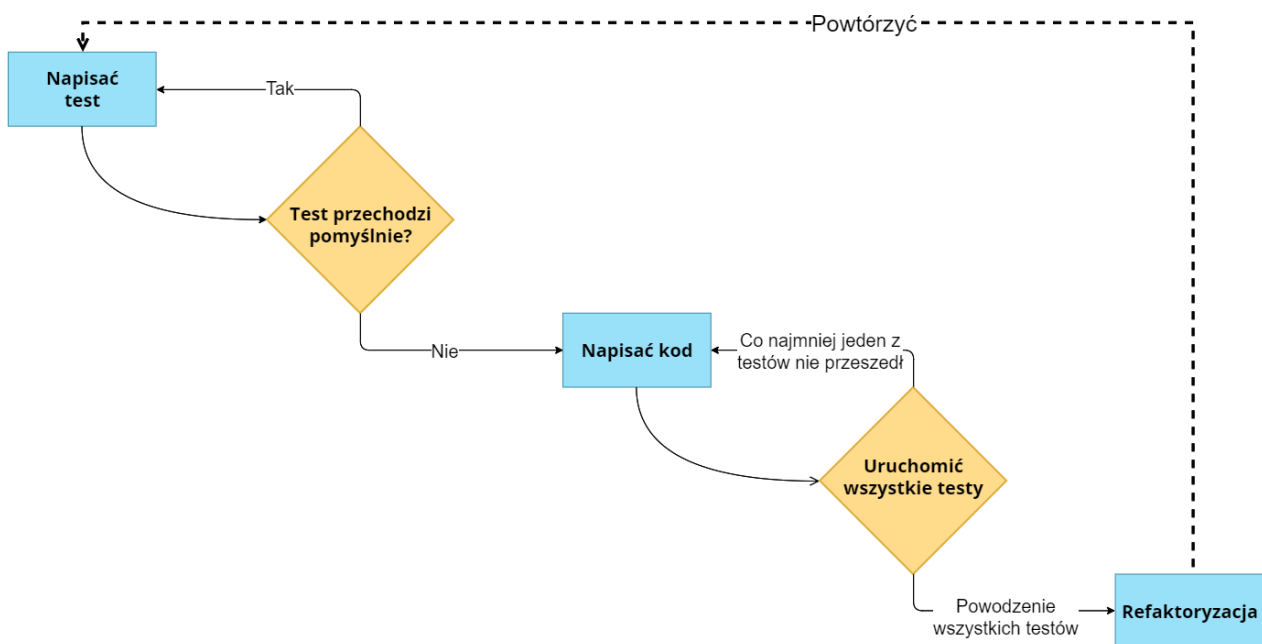
³ Tamże.

2.2 Wymagania TDD

Test Driven Development wymaga od dewelopera tworzenia automatycznych testów jednostkowych, które definiują wymagania dla kodu jeszcze przed napisaniem rzeczywistego kodu aplikacji. Test zawiera procedury sprawdzające czy spełnione są zadane warunki, a powodzenie testu weryfikuje zachowanie zgodne z zamierzeniami programisty. Deweloperzy często używają frameworków testowych do tworzenia i automatyzacji wykonywania zestawów testów. W praktyce testy jednostkowe obejmują krytyczne i nietrywialne sekcje kodu. Może to być kod podlegający częstym zmianom lub kod z dużą liczbą zależności.

TDD obejmuje nie tylko walidację, ale także wpływa na design programu. Opierając się na testach, programiści mogą szybciej wyobrazić sobie, jakich funkcjonalności potrzebuje użytkownik. Wtedy szczegóły interfejsu pojawiają się długo przed ostatecznym wdrożeniem rozwiązania.

2.3 Cykl TDD



Rysunek 1: Cykl TDD, składający się z pięciu kroków

1) Napisanie testu

W programowaniu opartym na testach wcielenie każdej nowej funkcjonalności do aplikacji zaczyna się od napisania testu. Aby to zrobić, programista powinien z pewnością rozumieć wymagania do nowej funkcjonalności. W tym celu brane są pod uwagę możliwe przypadki użycia i “historyjki użytkownika” (ang. *user stories*, opisujące wymagania z punktu widzenia docelowego użytkownika). Nowe wymagania mogą również wymuszać zmiany w istniejących testach. I właśnie ten pierwszy punkt cyklu odróżnia programowanie sterowane testami od technik, w których testy są tworzone po napisaniu kodu.

2) Uruchomienie wszystkich testów: nowe testy powinny się zakończyć niepowodzeniem

Właśnie napisany test nieuchronnie skończy się niepowodzeniem, ponieważ odpowiedni kod nie jest jeszcze napisany. Natomiast jeżeli test przeszedł pomyślnie, oznacza to, że rozważana funkcjonalność już istnieje lub napisany test ma wady. Wiedząc to, na tym etapie sprawdza się, czy napisane właśnie testy nie przechodzą. One także nie powinny przejść z oczywistych dla dewelopera powodów. Zwiększy to pewność (choć nie gwarantuje całkowicie), że test rzeczywiście sprawdza to, do czego został zaprojektowany.

3) Napisanie kodu

W tym momencie programista powinien zaprojektować nowy kod, aby wcześniej napisany test przeszedł pomyślnie. Ten kod nie musi być doskonały. Dozwolone jest, aby zdał „egzamin” w jakiś nieelegancki sposób. Jest to akceptowalne, ponieważ kolejne kroki go poprawią i wypolerują. Ważne jest, aby napisać kod specjalnie do przejścia konkretnego testu. Nie należy dodawać zbędnych, a za tym, niepokrytych testami funkcjonalności.

4) Uruchomienie wszystkich testów: wszystkie testy powinny się zakończyć powodzeniem

Jeśli wszystkie testy zakończą się powodzeniem, programista może mieć pewność, że kod spełnia wszystkie testowane wymagania. Dalej można przejść do ostatniego etapu cyklu.

5) Refaktoryzacja

Po osiągnięciu wymaganej funkcjonalności kod można „posprzątać”, czyli przeprowadzić refaktoryzację. Refaktoryzacja to proces zmiany wewnętrznej struktury programu, który nie wpływa

na jego zewnętrzne zachowanie i ma na celu ułatwienie zrozumienia jego działania, wyeliminowanie duplikacji kodu oraz ułatwienie wprowadzania zmian w najbliższej przyszłości⁴.

Opisany cykl powtarza się zgodnie ze schematem na rysunku 1, wdrażając coraz więcej nowych funkcjonalności. Jeśli nowy kod nie spełnia odpowiednich testów lub stare testy przestają przechodzić, programista powinien przejść do debugowania lub po prostu cofnąć się do poprzedniej wersji aplikacji i spróbować podejść do problemu z innej strony. Kroki powinny być małe, pozwoli to na łatwiejsze zidentyfikowanie błędów w przypadku ich wystąpienia, a w przypadku nierozwiązalnych trudności cofnięcie do poprzedniej wersji nie będzie aż tak bolesne.

2.4 Wzorce TDD

W tym podrozdziale zostaną omówione najważniejsze wzorce TDD, które wynikają z następującego stwierdzenia:

«Kiedy masz negatywny wynik testu, musisz to naprawić. Jeśli potraktujesz “czerwony pasek” jako sytuację do naprawienia tak szybko, jak to możliwe, odkryjesz, że możesz szybko dojść do “zielonego paska”. Użyj wzorców, aby test przeszedł (nawet jeśli wynik nie jest czymś, z czym chciałbyś żyć nawet przez godzinę)»⁵.

Kent Beck, twórca TDD

Tak więc najważniejszym celem opisanych poniżej wzorców jest jak najszybsze przejście do ostatniego, piątego punktu cyklu TDD – refaktoryzacji, ponieważ nadrzędnym przeznaczeniem programowania nadal jest napisanie rzeczywistego kodu aplikacji, a nie kodu testowego.

⁴ <https://bit.ly/3m94iqK>, *Trzy kroki TDD*, [dostęp: 15.08.2021].

⁵ K. Beck, *Test-Driven Development By Example*, Addison Wesley 2002, s. 143 (tłum. własne).

2.4.1 Sfalszuj to (ang. *Fake It*)

Jeżeli ma się test, który zakończył się niepowodzeniem, pierwszą rzeczą, którą należy zrobić, jest podstawienie stałej do testowanej metody. Po tym, jak test zaczął przechodzić, należy stopniowo przekształcać stałą na rzeczywistą implementację za pomocą zmiennych⁶.

Na pierwszy rzut oka ten wzorzec może się wydawać całkowicie bezużyteczny. Po co pisać kod, który wkrótce zostanie zastąpiony innym? Motywacją jest to, żeby mieć chociaż jakiś działający kod – to lepiej, niż nie mieć nic. Skorzystanie z tego wzorca wywołuje co najmniej dwa pozytywne efekty:

- Efekt psychologiczny – jeżeli przed oczami znajduje się „zielony” test, czujemy się zupełnie inaczej niż mając przed sobą czerwony kolor. Wtedy z pewnością można przystąpić do refaktoryzacji.
- Kontrola nad ilością pracy – programiści są przyzwyczajeni do przewidywania wszelkiego rodzaju problemów w przyszłości. Rozpoczęcie od konkretnego przykładu, a następnie rozwijanie kodu pomaga pozbyć się niepotrzebnych obaw. Można skoncentrować się na rozwiązaniu konkretnego problemu i dzięki temu lepiej wykonywać pracę. Kiedy przechodzi się do następnego testu, koncentrujemy się już na nim, ponieważ stwierdzono, że poprzedni test na pewno zadziała.

Jednak nie zawsze udaje się na pierwszy rzut oka zauważyć możliwość takich podstawień, więc dla większej przejrzystości należy użyć kolejnego wzorca TDD – „Triangulacja”.

2.4.2 Triangulacja (ang. *Triangulate*)

Zastosowanie wzorca „Triangulacja” wymaga od dewelopera zapewnienia modelu abstrakcji tylko w tym przypadku, gdy rozważane są dwa lub więcej przypadków użycia danej funkcjonalności. Czyli żeby lepiej zrozumieć logikę, a następnie napisać poprawny czysty kod, należy podejść do problemu z co najmniej dwóch stron (napisać dwa lub więcej testy). W odróżnieniu od wzorca „Sfalszuj to”, gdzie zastosowanie stałej polega na intuicji dewelopera, „Triangulacja” wprowadza bardziej konkretne zasady korzystania. Natomiast korzystać z tego wzorca należy tylko w tym przypadku, gdy

⁶ K. Beck, *TDD. Sztuka tworzenia dobrego kodu*, Helion 2014, s. 161.

się na pewno nie wie, jaka jest abstrakcja⁷. W przeciwnym przypadku można skorzystać z wzorca „Oczywista implementacja”.

2.4.3 Oczywista implementacja (ang. *Obvious Implementation*)

Wzorzec „Oczywista implementacja” najbardziej pasuje dla sytuacji, gdy realizacja jest bardzo prosta i przejrzysta. Wzorce „Sfałszuj to” oraz „Triangulacja” pozwalają stawiać małe kroki. Ale czasami występuje absolutna pewność, jak należy zrealizować pewną funkcjonalność. Wtedy od razu pisze się rzeczywisty kod bez żadnych „fałszywych” stałych. Ale korzystając z wzorca „Oczywista implementacja”, należy uważać, jak często napotyka się „czerwone” testy. Być może należy zwolnić i skorzystać z innych wzorców TDD⁸.

2.4.4 Najpierw operator assert (ang. *Assert first*)

Ten wzorzec nie jest najważniejszy w TDD, ale znacznie upraszcza tworzenie designu kodu źródłowego. Chodzi o to, żeby pisać operator assert nie po napisaniu ciała metody testowej, tylko przed⁹.

Pisząc test, rozwiązuje się kilka problemów jednocześnie, mimo że nie trzeba na razie myśleć o implementacji:

- Częścią czego jest nowa funkcjonalność? Czy jest to modyfikacja już istniejącej metody? Czy jest to nowa metoda istniejącej klasy? Czy musi to być nowa klasa?
- Jakie nazwy zmiennych przypisać użytym elementom?
- Jak sprawdzić poprawność wyniku działania metody?
- Co jest prawidłowym wynikiem działania metody?

Trudno jest znaleźć odpowiedzi na wszystkie te pytania, jeżeli będzie się to robiło jednocześnie. Ale dwa problemy z powyższej listy można łatwo oddzielić od wszystkich pozostałych: „Jak sprawdzić

⁷ K. Beck, *TDD. Sztuka tworzenia dobrego kodu*, Helion 2014, s. 163.

⁸ K. Beck, *TDD. Sztuka tworzenia dobrego kodu*, Helion 2014, s. 164.

⁹ K. Beck, *TDD. Sztuka tworzenia dobrego kodu*, Helion 2014, s. 140.

poprawność wyniku działania metody?” oraz „Co jest prawidłowym wynikiem działania metody?”. Rozwiązują się te dwa pytania bardzo prosto podczas napisania operatora `assert` przed napisaniem ciała metody testowej.

Praktyczne zastosowanie wszystkich wzorców z tego podrozdziału zostanie pokazane w rozdziale poświęconym implementacji.

2.5 Dlaczego TDD

Przed rozpoczęciem tworzenia aplikacji przemyślano, dlaczego należy użyć TDD oraz jakie problemy występujące w poprzednich projektach ta metodyka by pokonała. Podczas przygotowywania się do napisania pracy i czytania tematycznej literatury stworzona została lista bolączek, na które był narażony projekt, oraz które mogą zostać usunięte z pomocą TDD:

- większość testów były manualnych;
- jeżeli istniały testy automatyczne, to nie wykrywały one rzeczywistych problemów;
- testy automatyczne były pisane i przeprowadzane za późno, żeby wnieść użyteczny wkład w rozwój aplikacji;
- zawsze istniało coś ważniejszego do zrobienia niż napisanie testów;
- niemożliwość refaktoryzacji kodu ze względu na ryzyko zepsucia jakichś funkcjonalności;
- ciągły lęk przed wdrożeniem aplikacji, ponieważ na pewno jakaś funkcjonalność została pominięta podczas przeprowadzenia niewystarczającego testowania¹⁰.

Oczywiście, TDD nie pokona magicznie tych wszystkich problemów, ale istnieje nadzieja, że zostanie osiągnięty znaczny postęp na tych poziomach tworzenia oprogramowania.

Na początku wyszczególnijmy kilka zasadniczych zalet TDD:

- Zmniejsza się zależność od debugowania – ponieważ najpierw nacisk położony jest na pisanie testów, a następnie na generowanie kodu, aby przejść te testy, wielu programistów uważa, że znacznie zmniejsza się potrzeba debugowania. Ponieważ podczas pisania i kodowania testu

¹⁰ V. Farcic, A. Garcia, *Test-Driven Java Development*, Packt Publishing Ltd., Birmingham 2015, s. 2.

wymagane jest głębsze zrozumienie wymagań logicznych i funkcjonalnych, często można szybko zidentyfikować i rozwiązać przyczynę niepowodzenia testu.

- Programowanie sterowane testami oferuje więcej niż tylko walidację kodu. Ma ono również wpływ na projektowanie aplikacji. Skupiając się początkowo na testach, łatwiej wyobrazić sobie, jakiej funkcjonalności potrzebuje użytkownik. W ten sposób programista zastanawia się nad szczegółami interfejsu przed napisaniem kodu. Testy zmuszają nas do pisania kodu łatwiejszego do testowania. Na przykład, porzucenie zmiennych globalnych oraz singletonów sprawia, że klasy są mniej powiązane ze sobą i łatwiejsze w użyciu. Silnie powiązany kod będzie znacznie trudniejszy do przetestowania. Ponieważ testy jednostkowe przyczyniają się do tworzenia przejrzystych i małych interfejsów, każda klasa będzie miała określoną (zwykle małą) rolę.
- Testy pozwalają na refaktoryzację kodu bez ryzyka zepsucia go. Wprowadzając zmiany w dobrze przetestowanym kodzie, ryzyko nowych błędów jest znacznie mniejsze. Jeśli nowa funkcjonalność prowadzi do błędów, testy natychmiast to wykryją. Dobrze pokryty testami kod jest łatwy do refaktoryzacji. Programiści będą mieli dużo większą swobodę, jeśli chodzi o wprowadzanie zmian w architekturze, które mają na celu ulepszenie projektu.
- Testy mogą służyć jako dokumentacja. Dobry kod powie jak działa lepiej niż jakakolwiek dokumentacja. Dokumentacja i komentarze do kodu mogą być nieaktualne – może to być mylące dla programistów czytających kod. A ponieważ dokumentacja, w przeciwieństwie do testów, nie może powiedzieć, że jest przestarzała, nierzadko zdarzają się sytuacje, w których dokumentacja nie odpowiada rzeczywistości.
- Praktyka pokazuje, że ogólny czas pracy nad projektem jest skrócony w porównaniu z tradycyjnymi metodami tworzenia oprogramowania. Podczas gdy całkowita liczba linii kodu wzrasta (dzięki testom), częstsze testowanie eliminuje błędy w procesie i znacznie wcześniej identyfikuje istniejące, zapobiegając problemom później¹¹.

¹¹ <https://bit.ly/3z4S7hW>, *O testach jednostkowych i TDD*, [dostęp: 18.08.2021].

2.6 Fazy testowania

2.6.1 Testowanie jednostkowe

Testowanie jednostkowe – faza testowania oprogramowania, pozwalająca sprawdzić poprawność działania pojedynczych elementów (jednostek) lub zestawu modułów z jednej sekcji kodu źródłowego aplikacji¹².

Ten typ testowania zakłada, że należy pisać testy dla każdej nietrywialnej metody. Pozwala to dość szybko sprawdzić, czy po dokonaniu zmian nie doszło do regresji kodu, czy się nie pojawiły błędy w już przetestowanych kawałkach kodu źródłowego.

W testach jednostkowych rozbudowanych komponentów aplikacji, które mają dużą ilość zależności, należy skorzystać z obiektów fikcyjnych, które te zależności nadpiszą. Ponieważ w taki sposób nie będzie istniała zależność pomiędzy modułami aplikacji. Ta zależność powinna być testowana w testach integracyjnych.

2.6.2 Testowanie integracyjne

Testowanie integracyjne – to jedna z faz testowania oprogramowania, w której poszczególne warstwy oprogramowania są łączone i testowane w grupie¹³.

Podczas testowania integracyjnego jednostki, nad którymi już wykonano testy jednostkowe, są łączone w moduły i testowane zgodnie z założeniami odpowiednich funkcjonalności.

- W testach integracyjnych tzw. interfejsu-repozytorium należy skupić się na interakcji tej klasy z źródłem danych.

Interfejs-repozytorium – to komponent, który hermetyzuje logikę wymaganą do uzyskania dostępu do źródeł danych. Implementuje on ogólną funkcjonalność dostępu do danych i zapewnia infrastrukturę potrzebną do uzyskania dostępu do bazy danych z warstwy modelu aplikacji¹⁴.

¹² <https://bit.ly/3qt6Os3>, *Klucze do udanego testowania jednostkowego – jak programiści testują swój własny kod*, [dostęp: 20.12.2021].

¹³ <https://bit.ly/3sust5M>, *Czym jest Testowanie integracyjne*, [dostęp: 20.12.2021].

¹⁴ <https://bit.ly/3pJeKqn>, *Zaprojektuj warstwę danych*, [dostęp: 27.10.2021].

Ten typ testowania integracyjnego przeprowadza się w celu upewnienia się, że mapowanie obiektowo-relacyjne działa poprawnie, semantyka zapytań (jeżeli są samodzielnie zdefiniowane) nie jest naruszona oraz konfiguracja połączeń działa bez zarzutu.

- W testach integracyjnych tzw. komponentów serwisowych należy natomiast skupić się na integracji tego serwisu z warstwą danych, czyli z interfejsem-repozytorium. W testach tego typu powinno się komunikować z rzeczywistym źródłem danych w celu sprawdzenia faktycznej integracji pomiędzy modułami.

Komponent serwisowy – we frameworku Spring to klasa oznaczona adnotacją `@Service`, implementuje logikę biznesową aplikacji.

- W testach integracyjnych klasy-kontrolera należy skupić się na interakcji tej klasy z serwisem, który natomiast komunikuje z interfejsem-repozytorium. Można powiedzieć, że w ten sposób sprawdza się integrację całego systemu. Czyli te testy integracyjne powinny być przeprowadzone jako ostatnie.

Dwie ww. fazy testowania wykorzystują wzorce xUnit¹⁵ „Asercja” (ang. *Assertion*)¹⁶. Aby upewnić się, że kod źródłowy działa poprawnie, należy napisać metodę, zwracającą wartość typu boolean, potwierdzającą pewne zachowanie kodu. Na szczęście teraz nie ma potrzeby pisania takich metod własnoręcznie, w tym celu framework JUnit dostarczą grupę metod z klasy `Assertions`.

2.6.3 Testowanie manualne

Testowanie manualne – faza testowania, którą można odnieść do etapu kontroli jakości. Przeprowadza się bez wykorzystania narzędzi programistycznych poprzez symulację działań użytkownika. W tym przypadku zwykli użytkownicy mogą również grać rolę testerów, informując programistów o znalezionych błędach.

¹⁵ xUnit – rodzina frameworków testowych.

¹⁶ K. Beck, TDD. *Sztuka tworzenia dobrego kodu*, Helion 2014, s. 167.

3. Wykorzystane technologie i narzędzia

3.1 Strona serwera

3.1.1 Java

Java jest obiektywnym językiem programowania rozwijanym przez Sun Microsystems od 1991 roku i oficjalnie wydanym 23 maja 1995 roku. Ten język programowania był pierwotnie nazywany Oak i wykorzystywany dla elektroniki użytkowej, ale później został przemianowany na Java i był używany do pisania appletów, aplikacji i oprogramowania serwerowego. Programy w języku Java mogą być przetłumaczone na kod bajtowy, który jest uruchamiany na wirtualnej maszynie Java (JVM) – jest to program, przetwarzający kod bajtowy na instrukcje sprzętowe¹⁷. Dla implementacji aplikacji, stanowiącej podstawę niniejszej pracy, użyto wersji JAVA SE 11.

3.1.2 Spring Framework

Spring to jeden z najczęściej używanych frameworków do tworzenia aplikacji korporacyjnych, zapewniający solidny model programowania i konfiguracji. Celem stworzenia tego frameworku było uproszczenie rozwoju aplikacji na popularnym stosie technologii Java EE firmy Oracle, który w dawnych czasach był bardzo złożony i nie zawsze łatwy do wykorzystania¹⁸.

Jedną z głównych cech Spring Framework jest wykorzystanie wzorca „Wstrzykiwanie zależności” (ang. *Dependency Injection, DI*)¹⁹. DI znacznie ułatwia implementację funkcjonalności, których potrzebują aplikacje, a także pozwala na tworzenie mało powiązanych klas, czyniąc je bardziej ogólnymi. Spring Framework pozwala także skorzystać z wzorca „Odwrócenie kontroli” (ang. *Inversion of Control, IoC*)²⁰.

¹⁷ <https://bit.ly/3sv52JH>, Czym jest Java? Definicja, znaczenie i funkcje platform Java, [dostęp: 23.12.2021].

¹⁸ <https://bit.ly/3poE6JI>, Co to jest Spring Framework? Niekonwencjonalny przewodnik, [dostęp: 23.12.2021].

¹⁹ Koncepcja *Dependency Injection* polega na przeniesieniu odpowiedzialności za tworzenie instancji obiektu z ciała metody poza klasę i przekazanie z powrotem już utworzonego obiektu.

²⁰ Inversion of Control to zasada inżynierii oprogramowania, która przenosi kontrolę obiektów lub części programu do kontenera lub frameworku.

Spring Framework może być używany we wszystkich warstwach architektury podczas tworzenia aplikacji internetowych, a także pozwala swobodnie łączyć moduły i łatwo je testować. Ten framework jest świetnym narzędziem do rozwiązania najbardziej skomplikowanych problemów biznesowych.

3.1.3 Spring Boot

Pomimo wielu zalet, jakie zawiera Spring Framework, długi proces przygotowawczy związany z jego konfiguracją doprowadził do stworzenia Spring Boot²¹.

Spring Boot to jeden z wielu projektów w ekosystemie Spring, ale w przeciwieństwie do większości jego „kolegów”, nie rozwiązuje żadnego konkretnego problemu, a raczej reprezentuje nowy etap w rozwoju Spring jako całości. Podczas gdy Spring Framework koncentruje się na zapewnieniu elastyczności, Spring Boot dąży do skrócenia długości kodu i uproszczenia tworzenia aplikacji internetowych. Wykorzystując konfigurację adnotacji i schematu, Spring Boot skraca czas tworzenia aplikacji. Ta funkcja pomaga tworzyć aplikacje typu *stand-alone* z mniejszymi lub prawie zerowymi kosztami konfiguracji.

Celem Spring Boot jest uproszczenie procesu tworzenia aplikacji opartych na Spring Framework poprzez budowanie ich w oparciu o gotowe zestawy komponentów oprogramowania (tzw. pakiety „startowe”). Zawierają one zestaw odpowiednio skonfigurowanych narzędzi, których powinno się użyć do rozwiązania konkretnego problemu. W projekcie dyplomowym zostały użyte następujące startery:

- Spring Boot starter Data JPA – zawiera komponenty do pracy z bazami danych zgodnie z podejściem „Mapowanie obiektowo-relacyjne” (ang. *Object-Relational Mapping, ORM*), działa w oparciu o Hibernate²².
- Spring Boot starter Web – zawiera komponenty do tworzenia aplikacji internetowych (np. na podstawie REST²³)

²¹ <https://bit.ly/3yXYLc>, *Podstawy Java: Czym jest Spring Boot*, [dostęp: 23.12.2021].

²² Hibernate – framework, implementujący interfejs *Java Persistence API*.

²³ Representational State Transfer – styl architektury oprogramowania, zapewniający interakcję komponentów rozproszonych aplikacji w sieci.

- Spring Boot starter Logging – zawiera komponenty do logowania zdarzeń w czasie rzeczywistym.
- Spring Boot starter Test – zawiera komponenty do testowania automatycznego.
- Spring Boot starter Security – zawiera komponenty do zapewnienia bezpieczeństwa aplikacji. Wszystkie punkty końcowe części serwerowej projektu dyplomowego (oprócz /api/v1/auth/login) są skonfigurowane na przyjęcie zapytań tylko od autoryzowanych użytkowników.
- Spring Boot starter Validation – zawiera komponenty do walidacji komponentów modelu aplikacji.

3.1.4 PostgreSQL

PostgreSQL to bezpłatny system zarządzania obiektowo-relacyjnymi bazami danych. Posiada wiele innowacji, które zostały wdrożone w niektórych komercyjnych systemach zarządzania bazami danych znacznie później.

3.2 Testowanie

Dla testowania kodu źródłowego aplikacji użyto frameworka JUnit²⁴. Jest to framework, stworzony w celu testowania oprogramowania w języku Java. Leży u podstawy TDD i jest częścią rodziny frameworków testowych xUnit.

JUnit 5 to kombinacja modułów testowania – JUnit Platform, JUnit Jupiter oraz JUnit Vintage.

- JUnit Platform zapewnia uruchomienie testów na wirtualnej maszynie Java. Także zapewnia API (*application programming interface*) do stworzenia własnych frameworków testowych.
- JUnit Jupiter to połączenie nowego modelu programowania i modelu rozszerzeń do pisania testów.
- JUnit Vintage wyposaża silnik testowy w możliwość uruchomienia testów w starej, czwartej lub trzeciej wersji JUnit.

²⁴ <https://bit.ly/32hpzaa>, *JUnit 5 User Guide*, [dostęp: 24.12.2021].

W projekcie dyplomowym dla większości testów użyto nowej, piątej wersji frameworku. Natomiast dla testów integracyjnych kontrolerów użyto wersji czwartej, co ułatwia konfigurację. W piątej wersji JUnit klasa testowa powinna być oznaczona adnotacją `@ContextConfiguration`, także należy podać jako parametr adnotacji listę zależności, potrzebnych do uruchomienia testu. Natomiast w wersji czwartej nie trzeba samodzielnie definiować kontekst. W moim przypadku ułatwiło to sprawę, ponieważ przy próbie użycia piątej wersji JUnit dostawały się nieoczekiwane ostrzeżenia, których chciałbym unikać. W taki sposób ostateczny zestaw wszystkich testów aplikacji uruchamia się bez żadnych ostrzeżeń.

Także na potrzeby testów użyto frameworku Mockito²⁵. Testując kod (przede wszystkim podczas testów jednostkowych), często istnieje potrzeba, aby wyposażyć testowany element w instancje klas, z których powinien korzystać podczas pracy. Często te instancje nie muszą być w pełni funkcjonalne – wręcz przeciwnie, muszą zachowywać się w określony sposób, aby ich zachowanie było proste i przewidywalne. Takie obiekty nazywane są „stubami” lub obiektami fikcyjnymi. Aby je zdobyć, można tworzyć alternatywne testowe implementacje interfejsów, dziedziczyć niezbędne klasy z nadpisaniem funkcjonalności itd., ale wszystko to jest raczej niewygodne. Wygodniejszym rozwiązaniem pod każdym względem są wyspecjalizowane frameworki do tworzenia obiektów fikcyjnych. Jednym z najbardziej powszechnych i jest Mockito.

3.3 Strona klienta

Część webowa projektu dyplomowego jest napisana z użyciem biblioteki React języku JavaScript. Jest to najpopularniejsza biblioteka do tworzenia interfejsu użytkownika (ang. *user interface*, *UI*). React oferuje świetną odpowiedź na dane wprowadzane przez użytkowników, wykorzystując nową metodę renderowania stron internetowych. Komponenty React (kawałki interfejsu) są łatwe do implementowania, ponieważ do ich tworzenia używa się JSX (*JavaScript XML*) – opcjonalnego rozszerzenia składni JavaScript, które pozwala łączyć HTML z JavaScript²⁶.

²⁵ <https://bit.ly/3yReuIE>, *Mockito i jak go używać*, [dostęp: 24.12.2021].

²⁶ <https://bit.ly/3ms25pm>, *Tech 101: What Is React JS*, [dostęp: 24.12.2021].

Wszystkie komponenty są napisane z użyciem biblioteki React MUI²⁷. Ta biblioteka z nowoczesnym designem zapewnia świeży wygląd aplikacji.

²⁷ <https://bit.ly/3Epzl6M>, *Biblioteka React UI, której zawsze chciałeś*, [dostęp: 26.12.2021].

4. Opis aplikacji

4.1 Charakterystyka problemu

Zadaniem aplikacji, na przykładzie której zilustrowane zostanie zastosowanie metodyki TDD, jest kontrola oraz obsługa procedury i warunków prowadzenia certyfikacji zawodowej pracowników medycznych, farmaceutycznych i innych pracowników opieki zdrowotnej zgodnie z rozporządzeniem ministerstwa zdrowia Białorusi z dnia 28 maja 2021 r.²⁸.

Niżej przedstawiono tłumaczenie najbardziej znaczących zagadnień z ww. rozporządzenia. Wiedząc, że testy mogą służyć jako dokumentacja, w rozdziale poświęconym implementacji pokazano jak poniższe urywki najpierw są podstawą testów, a później implementacji w kodzie. Od razu chciałbym zaznaczyć, że czytelnik tej pracy nie powinien się martwić, jeżeli jakieś szczegóły wymagań nie będą zrozumiałe na bieżąco, ponieważ „logika biznesowa” aplikacji nie ma zasadniczego znaczenia dla tematu niniejszej pracy inżynierskiej.

Certyfikacja jest przeprowadzana w celu nadania lub potwierdzenia kategorii kwalifikacyjnej pracownika. Nadanie lub potwierdzenie kategorii kwalifikacyjnej odbywa się sekwencyjnie – druga, pierwsza, wyższa.

Rozporządzenie określa, że pracownicy sfery medycznej, mający w dniu wejścia rozporządzenia w życie (23 lipca 2021 r.) obecną kategorię kwalifikacyjną, powinni przejść certyfikację zawodową w celu potwierdzenia własnej kategorii w ciągu 4 lat i 9 miesięcy od ww. daty.

Certyfikacji podlegają:

- kadra kierownicza i specjaliści medyczni oraz farmaceutyczni, pracownicy opieki zdrowotnej;
- przedstawiciele służby zdrowia;
- kadra dydaktyczna państwowych placówek edukacyjnych, realizujących szkolenia i przekwalifikowanie specjalistów o profilu kształcenia „Medycyna” (oraz specjalność „Farmacja”);

²⁸ <https://bit.ly/2ZeP7Ty>, Rozporządzenie ministerstwa zdrowia Białorusi z dnia 28 maja 2021 r. №70, o profesjonalnej certyfikacji pracowników medycznych, farmaceutycznych oraz innych pracowników służby zdrowia, [dostęp: 20.12.2021].

- inni pracownicy medyczni o wykształceniu wyższym lub średnim, pracujący w jednostkach służby zdrowia, placówkach edukacyjnych, szkoleniowych, specjaliści o profilu kształcenia „Opieka zdrowotna”.

Pracownicy mogą ubiegać się o wyższą (sekwencyjnie kolejną od aktualnej) kategorię kwalifikacyjną nie wcześniej niż trzy lata od daty nadania poprzedniej kategorii kwalifikacyjnej. Certyfikacja na potwierdzenie kategorii kwalifikacyjnej przeprowadzana jest raz na pięć lat. Dokumentację na potwierdzenie kategorii kwalifikacyjnej należy złożyć do komisji atestacyjnej nie później niż trzy miesiące przed wygaśnięciem kategorii kwalifikacyjnej. Certyfikacja w celu nadania drugiej kategorii kwalifikacyjnej i potwierdzenia kategorii kwalifikacji jest obowiązkowa.

Z obowiązku certyfikacji zwolnieni są:

- pracownicy, którzy pracowali na odpowiednim stanowisku krócej niż rok;
- kobiety w ciąży;
- pracownicy, przebywające na długotrwałym (powyżej czterech miesięcy) leczeniu – na 6 miesięcy po powrocie do pracy;
- osoby odbywające staż w szkole wyższej, studia doktoranckie – w okresie studiów, szkolenia;
- osoby przebywające na urlopie macierzyńskim, na urlopie wychowawczym dla dziecka do ukończenia trzeciego roku życia – na rok po urlopie.

Kategorie kwalifikacyjne są przydzielane na podstawie kwalifikacji, odpowiednie z stanowiskiem pracownika (prowizor, prowizor-organizator, farmaceuta, lekarz organizujący opiekę zdrowotną, lekarz-ekspert itd.).

Aby nadać pracownikowi kategorię kwalifikacyjną, konieczne jest:

- druga kategoria kwalifikacyjna – mieć wiedzę teoretyczną oraz praktyczne umiejętności z zakresu działalności zawodowej, doświadczenie w pracy w odpowiedniej specjalności – co najmniej 3 lata, odbyć min. 100 godzin szkoleń zawodowych (dla pracowników o wykształceniu średnim medycznym lub farmaceutycznym – co najmniej 50 godzin);
- pierwsza kategoria kwalifikacyjna – mieć wiedzę teoretyczną oraz praktyczne umiejętności z zakresu działalności zawodowej, doświadczenie w pracy w odpowiedniej specjalności – co najmniej 6 lat, staż pracy w drugiej kategorii kwalifikacyjnej – co najmniej 3 lata, odbyć min. 160 godzin szkoleń zawodowych (dla pracowników o wykształceniu średnim medycznym lub farmaceutycznym – co najmniej 80 godzin);

- wyższa kategoria kwalifikacyjna – mieć wiedzę teoretyczną oraz praktyczne umiejętności z zakresu działalności zawodowej, doświadczenie w pracy w odpowiedniej specjalności – co najmniej 9 lat, staż pracy w pierwszej kategorii kwalifikacyjnej – co najmniej 3 lata, odbyć min. 160 godzin szkoleń zawodowych (dla pracowników o wykształceniu średnim medycznym lub farmaceutycznym – co najmniej 80 godzin).

Aby potwierdzić kategorię kwalifikacyjną pracownik powinien odbyć min. 160 godzin szkoleń zawodowych (dla pracowników o wykształceniu średnim medycznym lub farmaceutycznym – co najmniej 80 godzin).

Akumulacja wymaganego czasu szkoleń zawodowych, niezbędnych do nadania (potwierdzenia) kategorii kwalifikacyjnej, odbywa się w ciągu 5 lat. Warunkiem koniecznym do nadania (potwierdzenia) kategorii kwalifikacyjnej jest samodzielne opanowanie materiałów szkoleniowych, dla pracowników o wykształceniu wyższym medycznym w wymiarze co najmniej 80 godzin (dla pracowników z wykształceniem średnim medycznym lub farmaceutycznym – co najmniej 40 godzin). Pracownicy, którym nadano kategorię kwalifikacyjną, którzy przeszli na nowe stanowisko służbowe, są dopuszczeni do obrony na równorzędną kategorię kwalifikacyjną na nowym stanowisku pracy za dwa lata od daty rozpoczęcia pracy na nowym stanowisku. W tym czasie ważna jest kategoria kwalifikacyjna przypisana na poprzednim stanowisku pracy.

4.2 Wymagania

Aplikacja powinna:

- pobierać ogólne dane pracowników firmy z pliku CSV, wyeksportowanego z programu księgowego «1С Бухгалтерия»²⁹;
- pozwalać na ręczne wprowadzenie oraz modyfikację wszystkich potrzebnych danych dotyczących certyfikacji pracownika;
- obliczać oraz przedstawiać program certyfikacji osobno dla każdego pracownika zgodnie z zasadami opisanymi w odpowiednim dekrete;
- generować raporty w postaci plików PDF.

²⁹ <https://bit.ly/30Z26tf>, 1С:Бухгалтерия 8, [dostęp: 27.12.2021].

5. Implementacja

Aplikacja powinna być stworzona przy pomocy programowania sterowanego testami. Oznacza to, że należy zacząć implementację od czystego projektu. Nie ma na razie nic oprócz dokumentu, który stanowi logikę biznesową programu. Czyli należy iść z implementacją krok po kroku, czasami te kroki będą małe, czasami duże. Można pisać testy w taki sposób, że każdy z nich będzie wymagał dodania jednej linii kodu i niewielkiej refaktoryzacji. Z drugiej strony można pisać testy tak skomplikowane, że będą wymagać dopisania setek linii kodu i godzinnej refaktoryzacji. Nie ma lepszego podejścia z tych dwóch. W TDD powinno się umieć pracować na oba te sposoby.

Zacząć najlepiej od mniejszych kroków, korzystając z wzorców „Sfałszuj to” oraz „Triangulacja”, żeby nauczyć się pracować w odpowiednim tempie i odczuć rytm TDD. Kiedy uda się zdobyć pewność siebie i wszystko będzie szło gładko, będzie można spróbować przejść na większą prędkość, czyli skorzystać z wzorca „Oczywista implementacja” – testy będą wymagały napisania większego kawałka kodu, jeżeli od razu wiadomo jak ten kod ma wyglądać. Ale zawsze należy wracać do mniejszych kroków, jeśli pojawi się trywialny błąd tam, gdzie implementacja wydawała się być oczywista.

Cztery pierwsze podrozdziały tej części niniejszej pracy zostały napisane równolegle z pisanem kodu źródłowego aplikacji, częściowo użytego w listingach tej części pracy. W ten sposób chciałem pokazać rzeczywiste użycie TDD gdy dostaje się pewien problem (funkcjonalność do zaimplementowania), który należy rozwiązać od zera, dzieląc na podproblemy.

5.1 Podstawa modelu

Pierwszy krok implementacji programu nazwany został „podstawą modelu”, ponieważ tu zostało pokazane powstanie kilku podstawowych klas modelu aplikacji. Także zademonstrowano dodanie do tych klas wymaganych do przejścia testów pól oraz metod. Powinno to dać impuls do dalszego rozwoju logiki programu. TDD zakłada, że powinno się pisać tylko ten kod, którego wymaga test. Czyli daje to prawo aby brać poszczególne wycinki z dokumentu stanowiącego logikę biznesową, pisać testy, a następnie kod, by przejść te testy. Pierwszy urywek brzmi tak:

Pracownicy sfery medycznej, mający w dniu 23 lipca 2021 r. obecną kategorię kwalifikacyjną, powinni odbyć certyfikację zawodową w celu potwierdzenia własnej kategorii w ciągu czterech lat i dziewięciu miesięcy od ww. daty. Dokumenty potwierdzające odbycie certyfikacji zawodowej powinny być złożone do komisji nie później niż trzy miesiące do końca pięciu lat od ww. daty³⁰.

Teraz należy ustrukturyzować to jako listę podpunktów:

1. Istnieje data punktu odniesienia.
2. Odnosi się to tylko do pracowników mających na tę datę obecną kategorię.
3. Pracownicy powinni odbyć certyfikację w ciągu czterech lat i dziewięciu miesięcy od tej daty.
4. Dokumenty na potwierdzenie kategorii powinny trafić do komisji nie później niż trzy miesiące do końca pięciu lat od tej daty.

Najlepiej najpierw zająć się podpunktem 2, który wymusza stworzenie pracownika zawierającego kategorię. To zdanie brzmi jak test, przedstawiony na listingu 1.

```
1. @Test
2. public void testEmployeeCategory() {
3. }
```

Listing 1: Metoda testowa oznaczona adnotacją frameworku testowego JUnit

Teraz na tak prostym przykładzie zostanie pokazane jak działa wzorzec „Najpierw operator assert”. Czyli najpierw dodaje się do testu metodę statyczną `assertEquals`, sprawdzającą kategorię pracownika. Zostało pokazane to na listingu 2. I właśnie w tym momencie wymyśla się nazwy poszczególnych klas i pól.

```
1. @Test
2. public void testEmployeeCategory() {
3.     assertEquals(employee.category, "Kategoria X");
4. }
```

Listing 2: Do metody testowej została dołączona metoda statyczna, porównująca dwie wartości

³⁰ <https://bit.ly/2ZeP7Ty>, Rozporządzenie ministerstwa zdrowia Białorusi z dnia 28 maja 2021 r. №70, o profesjonalnej certyfikacji pracowników medycznych, farmaceutycznych oraz innych pracowników służby zdrowia, s. 1, [dostęp: 20.12.2021].

Następnie przy pomocy dynamicznych podpowiedzi IDE dodaje się nieistniejący obiekt, co widać na listingu 3.

```
1. @Test
2. public void testEmployeeCategory() {
3.     Employee employee = new Employee();
4.     assertEquals(employee.category, "Kategoria X");
5. }
```

Listing 3: W metodzie testowej został stworzony obiekt klasy Employee

Napisany właśnie test nawet się nie kompiluje, ale łatwo to poprawić. Najprościej to zrobić na sztywno dodając brakujące rzeczy, czyli klasę Employee, która zawiera pole category (patrz: lis. 4).

```
1. public class Employee {
2.     public String category;
3. }
```

Listing 4: Klasa Employee, zawierająca publiczne pole typu *String* – category

Widać, że nie jest to zadowalające rozwiązanie – istnieje pole publiczne w klasie. Ale teraz test się kompiluje i można zobaczyć jeden z kroków TDD w akcji. Napisano test, który nie przechodzi. Przyczynę można zobaczyć na listingu 5.

```
org.opentest4j.AssertionFailedError:
Expected :null
Actual   :Kategoria X
```

Listing 5: Przyczyna nieudania się testu oraz oczekiwany i rzeczywisty parametry metody statycznej assertEquals

Najważniejsze jest, że istnieje postęp. Nieudany test to też wynik. Teraz należy oderwać się od globalnego celu „odbycie certyfikacji pracownikiem w terminie” i przełączyć się na mały kroczek z kategorią. Chciałbym aby test przeszedł, a nie mam teraz na celu dostania pięknego rozwiązania. Najmniejsza zmiana, która sprawi, że test przebiegnie pomyślnie, została pokazana na listingu 6. Takie podejście jest jednym z opisanych we wstępie teoretycznym wzorców TDD i nazywa się

„Sfałszuj to”. Czyli chcemy zwracać stałą i stopniowo zamieniać ją na zmienną, dopóki nie otrzymamy docelowego kodu.

```
1. public class Employee {  
2.     public String category = "Kategoria X";  
3. }
```

Listing 6: Publiczne pole category w klasie Employee zostało zainicjalizowane

Ale jak można teraz zauważyć, istnieje duplikowanie między testem a kodem realizacji. Nie pozwoli to napisać kolejnego testu i go przejść, jeżeli parametry wejściowe będą inne (np. inna kategoria). Można naprawić to w oczywisty sposób. Także należy pamiętać, że powinno się teraz poruszać małymi krokami, żeby nauczyć się takiego tempa i odczuć rytm TDD. Duplikowanie usuwa się poprzez dodanie konstruktora z odpowiednim parametrem, można też od razu pozwolić sobie uczynić pole klasy prywatnym i wygenerować getter. Dodany kod został zademonstrowany na listingu 7.

```
1. public class Employee {  
2.     private String category;  
3.  
4.     public Employee(String category) {  
5.         this.category = category;  
6.     }  
7.  
8.     public String getCategory() {  
9.         return category;  
10.    }  
11. }
```

Listing 7: Dodanie konstruktora z parametrem category, gettera do tego pola,
które teraz już jest prywatne

Następnie należy wnieść zmiany do testu, co jest pokazane na listingu 8.

```
1. @Test
2. public void testEmployeeCategory() {
3.     Employee employee = new Employee("Kategoria X");
4.     assertEquals(employee.getCategory(), "Kategoria X");
5. }
```

Listing 8: Metoda testowa, w której obiekt jest tworzony za pomocą konstruktora z parametrem, a wartość pola category pobiera się za pomocą gettera

Test dalej przechodzi pomyślnie, kod wygląda już lepiej. Można teraz przejść do kolejnych podpunktów problemu. Na przykład do podpunktu 1, dla przypomnienia brzmi on tak:

1. Istnieje data punktu odniesienia jako 23.07.2021 r.

Należy gdzieś przechowywać tę datę, ponieważ jest to data wejścia rozporządzenia w życie i w przyszłości będzie potrzeba nieraz z niej skorzystać. Wybrano sposób przechowywania jako stała w już istniejącej klasie, pokazano to na listingu 9. Można to zrobić bez żadnych testów, ponieważ zostanie to przetestowane w innych testach do większych części kodu.

```
1. public class Employee {
2.     public static final LocalDate ACT_ENTRY_INTRO_FORCE_DATE =
3.         LocalDate.of(2021,7,23);
4.     //...
5. }
```

Listing 9: Publiczne niezmiennicze pole statyczne typu *LocalDate*, znajdujące się w klasie *Employee* i przechowujące datę wejścia rozporządzenia w życie

Dalej można przejść do reszty podpunktów – 3 oraz 4:

3. **Pracownicy powinni odbyć certyfikację w ciągu czterech lat i dziewięciu miesięcy od tej daty.**
4. **Dokumenty na potwierdzenie kategorii powinny trafić do komisji nie później niż trzy miesiące do końca pięciu lat od tej daty.**

Jeżeli wiadomo, że to się odnosi tylko do pracowników, którzy na moment wejścia rozporządzenia w życie mają obecną kategorię, to logiczne jest, że klasa odwzorowująca pracownika (*Employee*) ma

zawierać pole na przechowywanie daty potwierdzenia (nadania) kategorii. Także zgodnie z wymaganiami aplikacji istnieje potrzeba informowania użytkownika o zbliżającym się ostatecznym terminie (ang. *deadline*) dla potwierdzenia kategorii pracownika. Zakłada się, że użytkownik będzie chciał samodzielnie wybierać preferowany termin wcześniejszego powiadomienia w dniach. Można pozwolić sobie trochę przyspieszyć i napisać większy test, pokazany na listingu 10. W tym teście został stworzony nowy obiekt typu `NotificationTerm`, oraz zostało sprawdzone, czy ilość dni między datami jest zgodna z wartością pola `days` obiektu `notificationTerm`. Jest to test jednostkowy, nie sprawdza się tutaj integracji pomiędzy warstwami aplikacji.

```
1. @Test
2. public void testEmployeeDeadlineNotification() {
3.     Employee employee = new Employee();
4.     employee.setCategory("Kategoria X");
5.     employee.setCategoryAssignmentDeadlineDate(Employee
6.         .ACT_ENTRY_INTRO_FORCE_DATE.plusYears(5));
7.
8.     employee.setDocsSubmitDeadlineDate(Employee
9.         .ACT_ENTRY_INTRO_FORCE_DATE.plusYears(4).plusMonths(9));
10.
11.     NotificationTerm notificationTerm = new NotificationTerm();
12.
13.     notificationTerm.setDays(365);
14.     assertEquals(Math.abs(DAYS.between(employee
15.         .getCategoryAssignmentDeadlineDate(), LocalDate.of(2025, 7, 23))),
16.         notificationTerm.getDays());
17.
18.     notificationTerm.setDays(200);
19.     assertTrue(Math.abs(DAYS.between(employee
20.         .getDocsSubmitDeadlineDate(), LocalDate.of(2025, 7, 23)))
21.         > notificationTerm.getDays());
22. }
```

Listing 10: Test sprawdzający, czy ilość dni między datami jest zgodna z obiektem `notificationTerm`

Napisany właśnie test zakończył się powodzeniem po dopisaniu odpowiednich klas, pól i metod nawet bez dodawania do nich żadnych wpisanych na sztywno wartości, ponieważ wszystko się inicjalizuje w miejscu. Właśnie takie podejście jest jednym z wzorców TDD opisanych we wstępie teoretycznym i nazywa się „Oczywista implementacja”. Teraz można byłoby przetestować klasę `NotificationTerm`, jako odwzorowanie tabeli w bazie danych. Będzie to test integracyjny z warstwą bazodanową, opisany w kolejnym podrozdziale.

5.2 Test integracyjny – warstwa danych

W testach integracyjnych z warstwą danych należy skupić się na interakcji pomiędzy interfejsem-repozytorium a źródłem danych. Stworzono nową klasę testową, która jest skonfigurowana na potrzeby testów integracyjnych z warstwą danych. Różnicą pomiędzy testami jednostkowymi a integracyjnymi jest to, że teraz obiekt klasy NotificationTerm wysyła się do bazy danych. A następnie tworzy się nowy obiekt, który inicjalizuje się znalezionym na podstawie id rekordem z bazy. W taki sposób sprawdza się integracja z warstwą danych. Kod klasy jest pokazany na listingu 11.

```
1. @RunWith(SpringRunner.class)
2. @AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
3. @DataJpaTest
4. @ActiveProfiles("test")
5. public class DAOIntegrationTests {
6.
7.     @Autowired
8.     private TestEntityManager entityManager;
9.
10.    @Test
11.    public void testNotificationTermDAO() {
12.        NotificationTerm notificationTerm = new NotificationTerm();
13.        notificationTerm.setDays(365);
14.
15.        entityManager.persist(notificationTerm);
16.        entityManager.flush();
17.
18.        NotificationTerm foundNotificationTerm = entityManager
19.            .find(NotificationTerm.class, notificationTerm.getId());
20.
21.        assertEquals(365, foundNotificationTerm.getDays());
22.    }
```

Listing 11: Klasa testowa skonfigurowana na potrzeby testów integracyjnych z bazą danych

W liniach 2 oraz 3 listingu 11 dodano adnotacje, które mówią klasie testowej, że będą to testy integracyjne z warstwą danych, oraz że należy komunikować nie z wbudowanym, tylko z rzeczywistym źródłem danych. Także dobrą funkcjonalnością tak skonfigurowanej klasy testowej jest to, że dane zapisujące się podczas testów do bazy zostaną usunięte po zakończeniu testów. Czyli schemat bazy danych wróci do poprzedniego stanu. Ale zdecydowano skonfigurować inne źródło danych na potrzeby testów, ponieważ testowanie na realnej bazie danych projektu jest bardzo złą praktyką. A żeby klasa testowa wiedziała, że należy skorzystać z bazy testowej, dodano odpowiednią adnotację w linii 4. Także wywołuje się metodę flush w linii 16, która powoduje rzeczywiste

wykonanie zapytania w bazie. Pozwala to uzyskać id wysłanego przed chwilą obiektu na potrzeby dalszego odnalezienia go w bazie.

W celu poprawnej kompilacji oraz przejścia testu do klasy `NotificationTerm` zostały dodane kilka niezbędnych adnotacji frameworku Hibernate, co jest pokazane na listingu 12.

```
1. @Entity
2. public class NotificationTerm {
3.
4.     @Id
5.     @GeneratedValue(strategy = GenerationType.AUTO)
6.     @Column(columnDefinition = "serial")
7.     private Long id;
8.
9.     private int days;
10.
11.     // getter, setter
12. }
```

Listing 12: Kod klasy `NotificationTerm`, która jest encją odwzorowania obiektowo-relacyjnego (skrót ang. *ORM*)

Podsumowując podrozdziały 5.1 oraz 5.2 można powiedzieć, że udało się rozłożyć jeden wielki problem terminowej certyfikacji pracowników na 4 podproblemy, co pozwoliło skupić się osobno nad każdym i zrealizować to w miarę płynnie i bez stresu. Nie napisano żadnej linii kodu, której by nie wymagał test. Dlatego warto powiedzieć, że kod działa bezpiecznie, co wzbudza pewność siebie i chęć kontynuowania programowania kolejnych zagadnień logiki biznesowej.

5.3 Rozszerzenie modelu

Z wymagań aplikacji pośrednio wynika fakt, że pracownik może być aktywny lub nieaktywny. Zatem można dodać kolejny test (patrz: lis. 13).

```
1. @Test
2. public void testEmployeeActive() {
3.     Employee employee = new Employee();
4.     employee.setActive(true);
5.
6.     assertTrue(employee.isActive());
7. }
```

Listing 13: Metoda testowa, sprawdzająca, czy pracownik jest aktywny

Teraz na tak prostym przykładzie zostanie pokazane zastosowanie kolejnego wzorca TDD „Triangulacja”. Jak wynika z teorii, najpierw należy zastosować metodę zwracającą stałą, czyli skorzystać z wzorca fałszywej implementacji „Sfałszuj to” (patrz: lis. 14).

```
1. public class Employee {
2.     // ...
3.     public boolean isActive() {
4.         return true;
5.     }
6. }
```

Listing 14: Metoda isActive w klasie Employee zwracająca stałą

Test przechodzi pomyślnie. Ale co się stanie, jak zostanie napisany kolejny test, pokazany na listingu 15? Zostały tam dodane metody przygotowujące dane do testowania oraz czyszczące, ponieważ te dane będą się często powtarzać dla różnych testów.

```
1. private Employee employee;
2.
3. @BeforeEach
4. public void before() {
5.     employee = new Employee();
6. }
7.
8. @AfterEach
9. public void after() {
10.     employee = null;
11. }
```

```

11.}
12.
13.@Test
14.public void testEmployeeActive() {
15.    employee.setActive(true);
16.    assertTrue(employee.isActive());
17.}
18.
19.@Test
20.public void testEmployeeNotActive() {
21.    employee.setActive(false);
22.    assertFalse(employee.isActive());
23.}

```

Listing 15: Metody before oraz after, a także dwie metody testowe, sprawdzające, czy pracownik jest aktywny

Jeden z testów z listingu 15 przechodzi pomyślnie, a drugi nie. Przyczyną jest to, że metoda `isActive` zwraca teraz stałą `true`. Napisano drugi test, który inicjalizuje pole `active` wartością przeciwną, czyli `false`. W taki sposób udało się podejść do przypadku testowego z drugiej strony, czyli skorzystać z triangulacji. Pozwoliło to zobaczyć, że klasa `Employee` ma teraz wadę – metodę zwracającą stałą. Można to w bardzo prosty sposób zlikwidować, co zostało zademonstrowane na listingu 16.

```

1. public class Employee {
2.
3.    // ...
4.
5.    private boolean active;
6.
7.    public void setActive(boolean active) {
8.        this.active = active;
9.    }
10.
11.    public boolean isActive() {
12.        return active;
13.    }
14.}

```

Listing 16: Część kodu klasy `Employee`, zawierająca prywatne pole `active` oraz odpowiedni getter oraz setter

W tak prostym przykładzie były tylko dwie wartości graniczne – true oraz false. Natomiast zademonstrowany wzorec „Triangulacja” jest bardzo przydatny w sytuacjach, gdy logika jest bardzo skomplikowana. W takich przypadkach można triangulować problem z kilku stron, żeby złapać sens i napisać poprawny kod.

5.4 Wprowadzenie zmian

Czytając opis logiki biznesowej aplikacji zauważono, że pracownik może posiadać kategorię z konkretnie wyróżnionej listy: druga, pierwsza, wyższa. Także należy pamiętać, że w podpunkcie 4.1 operowało się na pracownikach mających obecną kategorię. Czyli logiczne jest stwierdzenie, że pracownicy bez kategorii też będą występować w aplikacji. Sugeruje to, aby zastosować typ wyliczeniowy z czterema wartościami. Także trzeba będzie zmienić kod testów, ponieważ na razie kategoria jest typu String. Po dokonaniu niezbędnych zmian oraz po poprawieniu błędów kompilacji wszystkie testy przechodzą pomyślnie, a nowy typ wyliczeniowy oraz jego zastosowanie w klasie Employee widać na listingu 17.

```
1. public enum Category {
2.
3.     HIGHEST("Wyższa"),
4.     FIRST("Pierwsza"),
5.     SECOND("Druga"),
6.     NONE("Brak");
7.
8.     private final String label;
9.
10.    Category(String label) {
11.        this.label = label;
12.    }
13.
14.    @Override
15.    public String toString() {
16.        return label;
17.    }
18.}
```

```
1. public class Employee {
2.     private Category category;
3.     // ...
4. }
```

Listing 17: Typ wyliczeniowy Category oraz jego zastosowanie w klasie Employee

5.5 Bezpieczeństwo aplikacji

Na chwilę odchodząc od tematu TDD, chciałbym zademonstrować, jak jest skonstruowana warstwa bezpieczeństwa aplikacji. Nie jest to ani bezpośrednim tematem pracy, ani bardzo ważnym wymogiem programu, ale zdecydowałem zaimplementować to w celu wykorzystania w dalszych etapach rozwoju aplikacji, kiedy powstanie potrzeba rozdzielenia użytkowników na role oraz rozbicia schematu na więcej niż jedną, korzystającą z aplikacji, firmę medyczną.

Część serwerowa polega na uwierzytelnianiu Spring Boot JWT (JSON Web Token) przez Spring Security, co zapewnia uwierzytelnianie oparte na tokenach i autoryzację opartą na rolach. Także została zaimplementowana pełna walidacja wartości wejściowych przy rejestracji.

Ta warstwa aplikacji została przetestowana za pomocą narzędzia do wysyłania zapytań REST Postman. Na rysunku 2 została pokazana odpowiedź serwera przy próbie rejestracji użytkownika z błędnymi danymi wejściowymi, wysyłając zapytanie POST do punktu końcowego (ang. *endpoint*) `/api/v1/auth/signup`. W tym przypadku polem z błędną wartością jest nazwa użytkownika. Dostaje się systemową wiadomość, że rozmiar wartości nazwy użytkownika musi być w przedziale od 5 do 32 znaków. Wiadomość jest w języku rosyjskim, ponieważ jest generowana automatycznie w języku systemu.



http://hlebshyupla.us-east-2.elasticbeanstalk.com/api/v1/auth/signup POST URL params Headers (2)

form-data x-www-form-urlencoded raw JSON

```
1 {
2   "username": "te",
3   "password": "test_password",
4   "email": "testEmail@gmail.com",
5   "role": [
6     "user"
7   ]
8 }
```

Send Preview Add to collection Reset

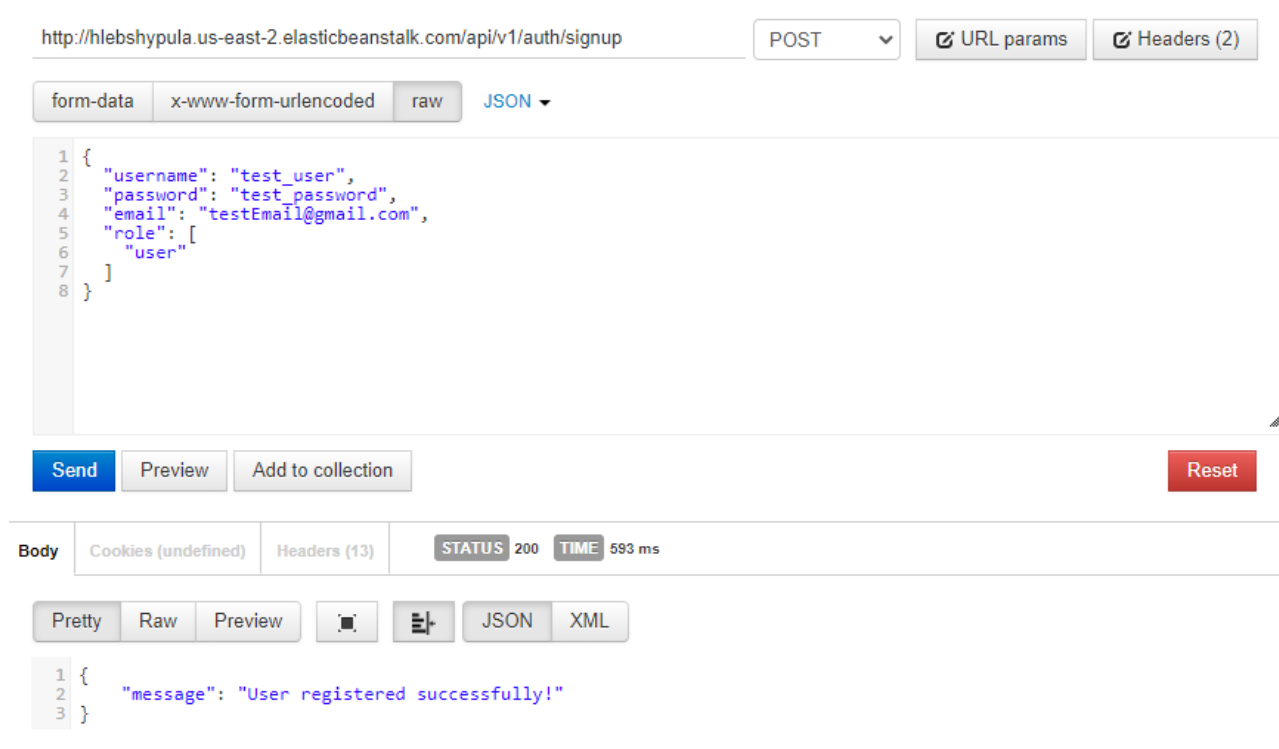
Body Cookies (undefined) Headers (13) STATUS 400 TIME 175 ms

Pretty Raw Preview JSON XML

```
1 {
2   "status": 400,
3   "message": "Validation failed for argument [0] in public org.springframework.http.ResponseEntity<?>
hlebshyupla.controller.authentication.AuthController.registerUser(hlebshyupla.security.payload.request.SignupRe
quest): [Field error in object 'signupRequest' on field 'username': rejected value [te]; codes
[Size.signupRequest.username,Size.username,Size.java.lang.String,Size]; arguments
[org.springframework.context.support.DefaultMessageSourceResolvable: codes [signupRequest.username,username];
arguments []; default message [username],32,5]; default message [размер должен находиться в диапазоне от 5 до
32]]",
4   "errors": {
5     "username": "размер должен находиться в диапазоне от 5 до 32"
6   }
7 }
```

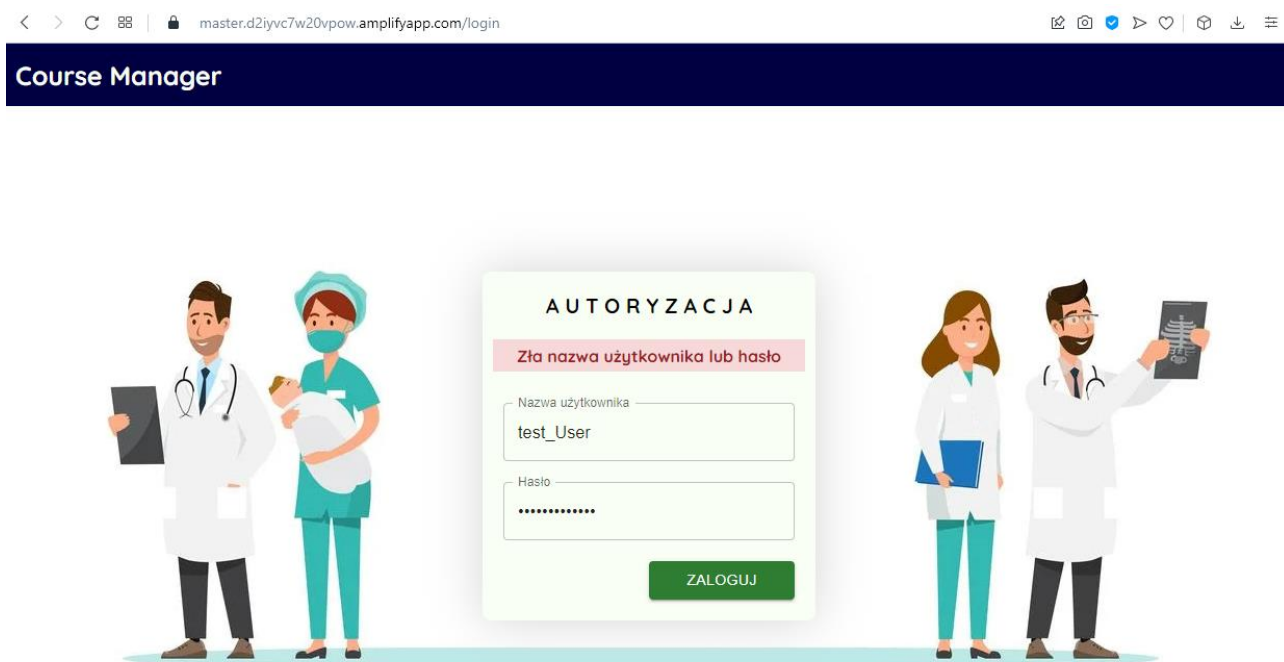
Rysunek 2: Odpowiedź serwera przy próbie rejestracji użytkownika z błędną nazwą użytkownika

Natomiast przy rejestracji z poprawnymi danymi dostaje się wiadomość o udanej rejestracji użytkownika, co jest pokazane na rysunku 3.



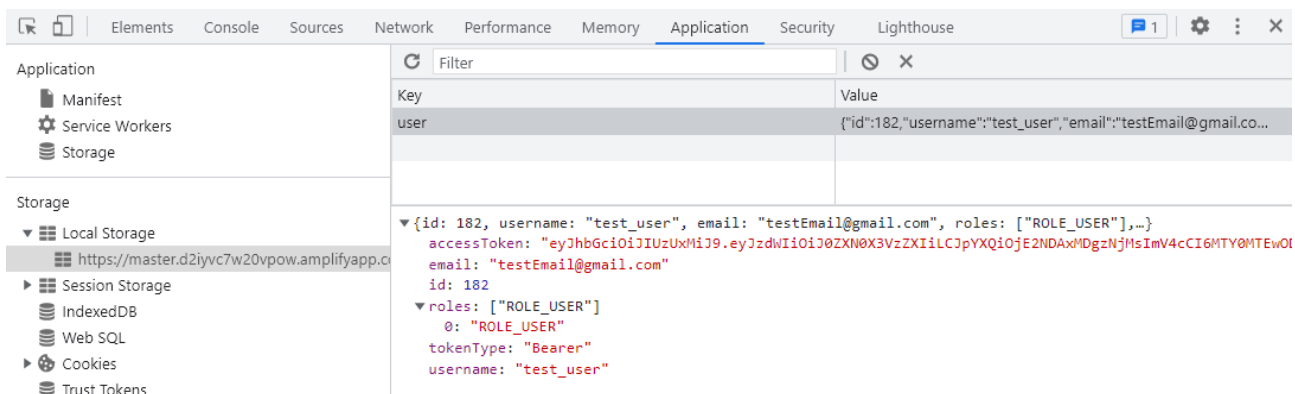
Rysunek 3: Poprawna rejestracja użytkownika

Po stronie klienta zostało zaimplementowane okno logowania (patrz: rys. 4). Przy próbie wejścia na stronę z błędnymi danymi użytkownik dostaje komunikat o błędzie.



Rysunek 4: Okno logowania wraz z komunikatem o błędzie

Po poprawnym zalogowaniu serwer wysyła w odpowiedzi dane użytkownika, które są zapisywane do Local Storage przeglądarki w odpowiedniej domenie (patrz: rys. 5). Jednym z pól zwróconych w odpowiedzi jest token JWT, który przechowuje w sobie zahasowane dane i służy do uwierzytelniania użytkownika podczas zapytań do punktów końcowych, które są skonfigurowane na przyjęcie zapytań tylko od autoryzowanych użytkowników lub od użytkowników określonych roli. Token łączy się do zapytania jako nagłówek. Dalej po stronie serwera przy otrzymaniu zapytania odbywa się sprawdzenie poprawności tokenu w specjalnie napisanej klasie tzw. filtrze. Przy próbie zapytania z błędnym tokenem, filtr to rozpozna i rzuci odpowiedni wyjątek (patrz: rys. 6).



Rysunek 5: Dane użytkownika zapisane do *Local Storage* przeglądarki



Rysunek 6: Odpowiedź serwera przy próbie zapytania z błędnym tokenem

5.6 Problem edycji danych

Podczas pisania pierwszych testów związanych z edycją danych pracownika, napotkano problem. Przy ścisłej walidacji pól klasy Employee (patrz: lis. 18), znika możliwość częściowej edycji stanu obiektu.

```
1. @Entity
2. @CategoryDatesNotNull
3. public class Employee {
4.
5.     @Id
6.     @GeneratedValue(strategy = GenerationType.AUTO)
7.     @Column(columnDefinition = "serial")
8.     Long id;
9.
10.    /// Main info
11.    /// Id from accounting app
12.    @NotNull(message = "foreignId cannot be null")
13.    Long foreignId;
14.    @NotBlank(message = "fullName cannot be blank")
15.    private String fullName;
16.    @NotNull(message = "hiringDate cannot be null")
17.    private LocalDate hiringDate;
18.    @NotBlank(message = "jobFacility cannot be blank")
19.    private String jobFacility;
20.    @NotBlank(message = "position cannot be blank")
21.    private String position;
22.
23.    /// Category
24.    private String qualification;
```

```

25.     private Category category;
26.     private String categoryNumber;
27.     private LocalDate categoryAssignmentDate;
28.     private LocalDate categoryAssignmentDeadlineDate;
29.     private LocalDate docsSubmitDeadlineDate;
30.     private LocalDate categoryPossiblePromotionDate;
31.
32.     /// Courses
33.     @OneToMany(mappedBy = "employee", fetch = FetchType.EAGER)
34.     @JsonIgnore
35.     private Set<Course> courses;
36.     private int courseHoursSum;
37.
38.     /// Exemption
39.     private Exemption exemption;
40.     private LocalDate exemptionStartDate;
41.     private LocalDate exemptionEndDate;
42.     private boolean exemptioned;
43.
44.     /// Active / inactive
45.     private boolean active;
46.
47.     /// Education
48.     private Education education;
49.     private String eduName;
50.     private LocalDate eduGraduationDate;
51.
52.     /// statyczne pola, konstruktory, gettery, settery ...
53.
54. }

```

Listing 18: Pola klasy Employee oznaczone adnotacjami walidacji oraz adnotacjami frameworku Hibernate

Jak widać na listingu 18, adnotacjami walidacji (`@NotNull`, `@NotBlank`) oznaczone są tylko pola w sekcji Main info, ponieważ te pola są inicjalizowane na początku życia obiektu i są z nim związane przez cały cykl życia. Wartości tych pól przychodzą z zewnątrz (z aplikacji księgowej) i na ich podstawie tworzy się nowy obiekt pracownika. Natomiast pola z innych sekcji (Category, Exemption itd.) na początku powinny mieć wartość null. Użytkownik aplikacji powinien mieć możliwość osobno edytować każdą z tych sekcji stanu obiektu.

Typowym sposobem edycji stanu obiektu w aplikacjach internetowych opartych na REST API jest wysyłanie zapytania PUT, zawierającego w ciele nowy stan obiektu, który nadpisuje stan poprzedni.

Teraz problem jest taki, że, korzystając z zapytań PUT, nie ma możliwości aby oznaczyć jakieś pole z sekcji Category adnotacją @NotNull, ponieważ przy próbie edycji sekcji Main info nadal następuje nadpisanie wszystkich pól obiektu. Na przykład, jeżeli pole jest null i nie ma potrzeby zmiany wartości, to wartość nadal się zmieni z null na null. A to właśnie wywołuje naruszenie ograniczenia, które nakłada się na pole poprzez adnotację @NotNull.

Przeglądając wzorce rozwiązujące ten problem, zdecydowano skorzystać ze zmodyfikowanego wzorca wykorzystującego obiekt transferu danych (ang. *Data Transfer Object*, *DTO*). Polega to na tym, że do częściowej modyfikacji stanu obiektu używa się zapytania PATCH, które w ciele zawiera DTO dla odpowiedniej sekcji obiektu klasy Employee. W dalszej części podpunktu przebieg działań zostanie pokazany na jednym z pięciu przykładów użycia tego podejścia w aplikacji.

Chcąc zmodyfikować wykształcenie pracownika, użytkownik powinien wybrać osobę z listy (patrz: rys. 7) i wejść na stronę z danymi.

Course Manager

PRACOWNICY

ZAPLANUJ KURSY

POKAŻ DANE

Imię i nazwisko	Stanowisko	Kategoria	Termin potwierd...	Termin dostarcz...	Pozostała ilość ...
Adam Babacki	Prowizor	Brak	10.12.2024	10.09.2024	100
Karol Swiderski	Farmaceuta	Brak	08.12.2024	08.09.2024	50
Karolina Kowalska	Farmaceuta	Brak	16.12.2026	16.09.2026	0
Vladimir Ivanov	Prowizor-organizator	Brak	11.08.2022	11.05.2022	50
Hleb Shypula	Farmaceuta				

1 row selected

Rows per page: 100 1-5 of 5

DODAJ

Prześlij plik .csv, pobrany z aplikacji księgowej

Rysunek 7: Strona z listą wszystkich pracowników

Na stronie z danymi użytkownik powinien kliknąć na przycisk „Edytuj” na polu z wykształceniem. Ten przycisk jest oznaczony czerwonym kwadracikiem na rysunku 8.

Course Manager

PRACOWNICY

Dane osobowe

Aktywny

Imię i nazwisko: Adam Babacki
Data zatrudnienia: 13.10.2019
Miejsce pracy: Apteka №3
Stanowisko: Prowizor

Kategoria

Brak

Kwalifikacja: Farmaceuta
Kategoria: Brak
Numer:
Data nadania:
Termin potwierdzenia: 10.12.2024
Termin dostarczenia dokumentów: 10.09.2024
Możliwe nadanie kolejnej kategorii: 10.12.2024

EDYTUJEDYTUJ RĘCZNIE

20

Pozostała ilość godzin do 10.12.2024

+ DODAJ KURS

Wykształcenie

Wyższe

Rodzaj: Wyższe
Szkoła: AGH
Data zakończenia: 10.12.2021

EDYTUJ

Zwolnienie

Zwolniony

Nazwa	Opis	Ilość godzin	Data początku	Data końca
Kurs 1		80	14.12.2021	30.12.2021

Rysunek 8: Strona z danymi pracownika oraz oznaczony na czerwono przycisk do edycji wykształcenia

Następnie użytkownik powinien wprowadzić dane w formularzu oraz kliknąć na przycisk „Zatwierdź” (patrz: rys. 9).

Edytuj wykształcenie

Imię i nazwisko

Adam Babacki

Rodzaj

Wyższe

Nazwa szkoły

UJ kraków

Data zakończenia

31.03.2022

ANULUJ

ZATWIERDŹ

Rysunek 9: Formularz do edycji wykształcenia pracownika

W tym momencie ze strony klienta na serwer wysyła się zapytanie PATCH do punktu końcowego `/api/v1/employees/{id}/education` poprzez wbudowane w język JavaScript Fetch API. Kod źródłowy funkcji wysyłającej zapytanie jest umieszczony na listingu 19.

```

1. export const patchEmployeeEducation = (id, patch) => {
2.   return patchEmployee(id, patch, "/education");
3. };
4.
5. const patchEmployee = (id, patch, path) => {
6.   return fetch(API_BASE_URL + "/" + id + path, {
7.     method: "PATCH",
8.     body: JSON.stringify(patch),
9.     headers: Object.assign(
10.      {},
11.      { "Content-type": "application/merge-patch+json" },
12.      authHeader()
13.    ),
14.  })
15.    .then((response) => {
16.      if (response.ok) {
17.        return response.json();
18.      } else {
19.        throw new Error(response.clone().json());
20.      }
21.    })
22.    .catch((error) => {
23.      console.log(error);
24.    });
25. };

```

Listing 19: Funkcje w języku JavaScript odpowiadające za wysyłanie zapytania PATCH w celu edycji wykształcenia pracownika

Po stronie serwera zapytanie przyjmuje się w metodzie, pokazanej na listingu 20.

```

1. @PatchMapping(path =("/{id}/education",
2.   consumes = "application/merge-patch+json")
3. public ResponseEntity<Employee> patchEmployee(
4.   @PathVariable Long id,
5.   @RequestBody @Valid EmployeeEducationPatchDto employeeEducationPatchDto) {
6.   return ResponseEntity.ok(employeeDataService
7.     .patch(id, employeeEducationPatchDto));
8. }

```

Listing 20: Metoda w klasie REST kontrolera, przyjmująca zapytanie PATCH

Jak widać na listingu 20, wywołuje się metoda `patch` serwisu³¹ `EmployeeDataService`, która przyjmuje jako parametry id pracownika oraz DTO wykształcenia (patrz: lis. 22). DTO wykształcenia (patrz: lis. 21) zawiera tylko te pola z klasy `Employee`, które odpowiadają za sekcję `Education` z listingu 18. Także na tym etapie można dodać adnotacje, odpowiadające za walidację tych pól. Teraz już to nie przeszkadza innym sekcjom klasy `Employee`, ponieważ jest wyosobniono w oddzielną klasę. A żeby zwalidować obiekt, należy oznaczyć go adnotacją `@Valid`, jak to jest zrobione na listingu 20 w 5 linii.

```
1. public class EmployeeEducationPatchDto extends EmployeePatchDto {
2.
3.     @NotNull(message = "Education cannot be null")
4.     private Education education;
5.     @NotBlank(message = "Education name cannot be blank")
6.     private String eduName;
7.     @NotNull(message = "Education graduation date cannot be empty")
8.     private LocalDate eduGraduationDate;
9.
10.    /// getter, setter
11.
12. }
```

Listing 21: Kod DTO `EmployeeEducationPatchDto` wraz z adnotacjami walidacji

```
1. private void patchEmployeeEducation(Employee employee, EmployeePatchDto
   employeePatchDto) {
2.     BeanUtils.copyProperties(employeePatchDto, employee);
3. }
```

Listing 22: Kod metody `patch` serwisu `EmployeeDataService`, odpowiadający za kopiowanie wartości DTO do obiektu klasy `Employee`

W taki, opisany powyżej, sposób osiągnięta została częściowa edycja stanu obiektu klasy `Employee`, nie tracąc mocy walidacji. Zademonstrowano to na prostszym przykładzie, żeby nie przeładowywać tekstu pracy kodem źródłowym. Ale warto powiedzieć, że na przykład DTO dla sekcji `Category` z

³¹ Serwis – klasa oznaczona adnotacją `@Service`, która implementuje logikę biznesową aplikacji.

listingu 18 zawiera na poziomie klasy dwie własnoręcznie napisane adnotacje walidacji, ponieważ logika tej sekcji jest dość skomplikowana i systemowe adnotacje z nią sobie nie radziły.

Natomiast testy do tego kawałka logiki, które były napisane zgodnie z założeniami TDD przed napisaniem rzeczywistego kodu, są pokazane na listingach 23 oraz 24.

Pierwszy test na listingu 23 z linii 25 sprawdza, że ograniczenia na pewno nie są naruszone przy walidacji poprawnego DTO `EmployeeEducationPatchDto`. Natomiast drugi test od linii 32 sprawdza, czy naruszenie nałożone adnotacją `@NotNull` na pole `education` z listingu 21 na pewno zostało złapane. Podczas testów walidacja przeprowadza się ręcznie poprzez wywołanie walidatora wstrzykniętego w linijce 7 za pomocą adnotacji `@Autowired`. Natomiast w rzeczywistym użyciu obiektów walidacja odbywa się automatycznie po oznaczeniu obiektu adnotacją `@Valid`, jak już było powiedziane wyżej.

```
1. /// Unit tests
2. @SpringBootTest
3. @ActiveProfiles("test")
4. public class EmployeeEducationPatchDtoTests {
5.
6.     @Autowired
7.     Validator validator;
8.
9.     private EmployeeEducationPatchDto employeeEducationPatchDto;
10.
11.     @BeforeEach
12.     public void before() {
13.         employeeEducationPatchDto = new EmployeeEducationPatchDto();
14.         employeeEducationPatchDto.setEducation(Education.HIGHER);
15.         employeeEducationPatchDto.setEduName("AGH");
16.         employeeEducationPatchDto.setEduGraduationDate(LocalDate.of(2020, 5,
17. 5));
18.     }
19.
20.     @AfterEach
21.     public void after() {
22.         employeeEducationPatchDto = null;
23.     }
24.
25.     @Test
26.     public void When_Valid_Then_ViolationsEmpty() {
27.         Set<ConstraintViolation<EmployeeEducationPatchDto>> violations =
28.             validator.validate(employeeEducationPatchDto);
29.         assertTrue(violations.isEmpty());
30.     }
31.
32.     @Test
```

```

32.     public void
33.         When_EducationIsNull_Should_ThrowConstraintViolationException() {
34.             employeeEducationPatchDto.setEducation(null);
35.             Set<ConstraintViolation<EmployeeEducationPatchDto>> violations =
36.                 validator.validate(employeeEducationPatchDto);
37.             assertEquals("Education cannot be null",
38.                 violations.iterator().next().getMessage());
39.         }
40.         /// inne testy
41.
42. }

```

Listing 23: Testy jednostkowe dla DTO EmployeeEducationPatchDto

```

1.  /// Unit tests
2.  @ExtendWith(MockitoExtension.class)
3.  @SpringBootTest
4.  @ActiveProfiles("test")
5.  public class EmployeeDataServiceTests {
6.
7.      @Mock
8.      EmployeeRepository employeeRepository;
9.      @Mock
10.     CourseService courseService;
11.     @Mock
12.     EmployeeValidationService employeeValidationService;
13.     @Mock
14.     EmployeeExemptionService employeeExemptionService;
15.     @Mock
16.     EmployeeFilteringService employeeFilteringService;
17.     @Mock
18.     EmployeeCategoryService employeeCategoryService;
19.
20.     private EmployeeDataService employeeDataService;
21.
22.     private Employee employee;
23.
24.     @BeforeEach
25.     public void before() {
26.         employeeDataService = new EmployeeDataService(employeeRepository,
27.             courseService, employeeValidationService,
28.             employeeExemptionService, employeeFilteringService,
29.             employeeCategoryService);
30.
31.         employee = new Employee();
32.         employee.setForeignId(1L);
33.         employee.setFullName("Test");
34.         employee.setHiringDate(LocalDate.of(2020, 5, 5));

```

```

33.         employee.setJobFacility("Apteka 9");
34.         employee.setPosition("Farmaceuta");
35.
36.         Optional<Employee> optionalEmployee = Optional.of(employee);
37.         doReturn(optionalEmployee).when(employeeRepository).findById(any(Long.class))
38.     ;
39.     }
40.     @AfterEach
41.     public void after() {
42.         employee = null;
43.     }
44.
45.     @Test
46.     public void Should_PatchEducation() {
47.         EmployeeEducationPatchDto employeeEducationPatchDto = new
EmployeeEducationPatchDto();
48.         employeeEducationPatchDto.setEducation(Education.HIGHER);
49.         employeeEducationPatchDto.setEduName("AGH");
50.         employeeEducationPatchDto.setEduGraduationDate(LocalDate.of(2020, 5,
51.         5));
52.         BeanUtils.copyProperties(employeeEducationPatchDto, employee);
53.         doReturn(employee).when(employeeRepository).save(any(Employee.class));
54.
55.         Employee newEmployee = employeeDataService.patch(any(Long.class),
employeeEducationPatchDto);
56.
57.         assertNotNull(newEmployee);
58.         assertEquals("AGH", newEmployee.getEduName());
59.     }
60. }

```

Listing 24: Testy jednostkowe dla serwisu EmployeeDataService

Na listingu 24 widać wykorzystanie obiektów fikcyjnych oznaczonych adnotacją `@Mock` frameworka Mockito. W linii 26 tworzy się nowy obiekt serwisu poprzez konstruktor, gdzie wprowadza się te obiekty. Oznacza to, że przeprowadzane w tej klasie testy są testami jednostkowymi, a nie integracyjnymi, które by sprawdzały integrację tego serwisu z innymi warstwami aplikacji. Po oznaczeniu obiektu adnotacją `@Mock` można zdefiniować zachowanie tego obiektu w pewnej sytuacji. W liniach 37 oraz 53 zdefiniowano, co ma się zwrócić po wywołaniu odpowiedniej metody podczas wykonania testu.

Ale żeby móc skorzystać z takiego typu fikcyjnych obiektów, należy w klasie, do której planuje się wstawić te obiekty, użyć wstrzykiwania zależności poprzez konstruktor³². W taki sposób zwiększa się hermetyzacja klasy, a także pojawia się możliwość użyć fikcyjnych obiektów podczas testowania, jak było pokazane powyżej. Ta technika jest pokazana na listingu 25, który zawiera kawałek kodu źródłowego serwisu `EmployeeDataService`. W linii 12 znajduje się konstruktor oznaczony adnotacją `@Autowired`, który inicjalizuje zależności klasy na podstawie parametrów.

```
1. @Service
2. public class EmployeeDataService {
3.
4.     private final EmployeeRepository employeeRepository;
5.     private final CourseService courseService;
6.     private final EmployeeValidationService employeeValidationService;
7.     private final EmployeeExemptionService employeeExemptionService;
8.     private final EmployeeFilteringService employeeFilteringService;
9.     private final EmployeeCategoryService employeeCategoryService;
10.
11.     @Autowired
12.     public EmployeeDataService(EmployeeRepository employeeRepository,
13.                               CourseService courseService,
14.                               EmployeeValidationService employeeValidationService,
15.                               EmployeeExemptionService employeeExemptionService,
16.                               EmployeeFilteringService employeeFilteringService,
17.                               EmployeeCategoryService employeeCategoryService) {
18.         this.employeeRepository = employeeRepository;
19.         this.courseService = courseService;
20.         this.employeeValidationService = employeeValidationService;
21.         this.employeeExemptionService = employeeExemptionService;
22.         this.employeeFilteringService = employeeFilteringService;
23.         this.employeeCategoryService = employeeCategoryService;
24.     }
25.     /// inne metody
26. }
```

Listing 25: Kawałek kodu serwisu `EmployeeDataService`, demonstrujący technikę wstrzykiwania zależności poprzez konstruktor

³² Wstrzykiwanie zależności poprzez konstruktor – technika wstrzykiwania zależności frameworku Spring, oznaczająca, że przekazanie wymaganych komponentów do klasy odbywa się w momencie tworzenia instancji. Osiąga się poprzez oznaczenie konstruktora adnotacją `@Autowired`. W przeciwieństwie do wstrzykiwania zależności poprzez pole, gdzie zależności są przypisywane bezpośrednio do pól, które są oznaczone adnotacją `@Autowired`.

5.7 Dodawanie pracowników

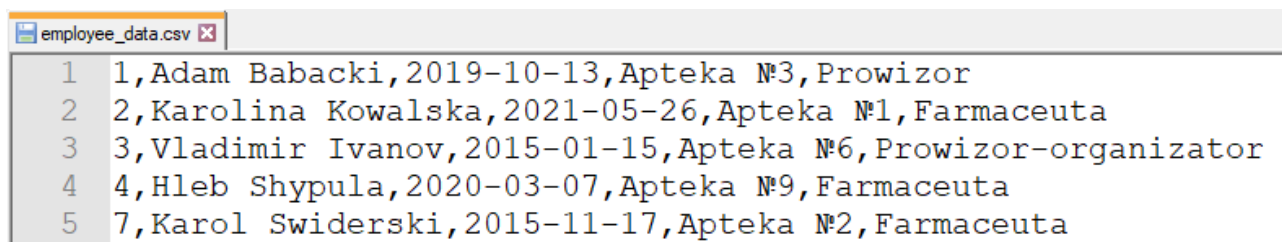
Jak widać na rysunku 6, pod tablicą istnieje przycisk na dodawanie pracowników poprzez wysłanie pliku w formacie *CSV* (ang. *comma-separated values*) na serwer. Według aktualnej wersji aplikacji plik ten powinien być wygenerowany w aplikacji księgowej firmy korzystającej z programu. Dlatego w podrozdziale 5.6 było podane, że pewne pola klasy *Employee* są inicjalizowane na początku życia obiektu pracownika. Są to pola, które przechowują główne informacje dotyczące danego pracownika.

Przy próbie wysłania pliku na serwer należy uważać na format danych, ponieważ na serwerze nie ma żadnej walidacji przesłanego pliku *CSV*. Ma to swoje uzasadnienie – w kolejnych etapach rozwoju aplikacji planuje się automatyzację importu danych do aplikacji poprzez serwerowe połączenie z aplikacją księgową, czyli potrzeba walidacji pliku odpadnie.

Natomiast teraz format pliku powinien być następujący:

Każda kolejna linijka zawiera informacje jednego pracownika oraz składa się z pól oddzielonych przecinkami. Są to kolejno: identyfikator, pełne imię, data zatrudnienia, miejsce pracy, stanowisko.

Przykładowy plik *CSV* widnieje się na rysunku 10.



```
1 1,Adam Babacki,2019-10-13,Apteka №3,Prowizor
2 2,Karolina Kowalska,2021-05-26,Apteka №1,Farmaceuta
3 3,Vladimir Ivanov,2015-01-15,Apteka №6,Prowizor-organizator
4 4,Hleb Shypula,2020-03-07,Apteka №9,Farmaceuta
5 7,Karol Swiderski,2015-11-17,Apteka №2,Farmaceuta
```

Rysunek 10: Przykładowy plik *CSV*, zawierający dane pięciu pracowników

5.8 Test integracyjny – warstwa serwisu

Celem testów integracyjnych na poziomie serwisu jest sprawdzenie interakcji serwisu z warstwą danych. Czyli w tym przypadku sprawdza się rzeczywiste połączenie z źródłem danych, a nie korzysta się z obiektów fikcyjnych, jak to było pokazane w podrozdziale 5.6 na listingu 24. Akurat w tym przypadku konfiguracja klasy testowej jest prosta. Wystarczy tylko powiedzieć, że będzie to klasa *@SpringBootTest*, żeby móc skorzystać z wstrzykiwanych komponentów. A także podać aktywny profil, z którego należy skorzystać podczas testów, jako adnotację *@ActiveProfiles*. W tym

przypadku będzie to profil testowy. Kod źródłowy klasy testującej serwis EmployeeDataService jest pokazany na listingu 26.

```
1.  /// Integration tests
2.  @SpringBootTest
3.  @ActiveProfiles("test")
4.  public class EmployeeDataServiceIntegrationTests {
5.
6.      @Autowired
7.      EmployeeDataService employeeDataService;
8.
9.      private Employee employee;
10.
11.      @BeforeEach
12.      public void before() {
13.          employee = new Employee();
14.          employee.setForeignId(1L);
15.          employee.setFullName("Test");
16.          employee.setHiringDate(LocalDate.of(2020, 5, 5));
17.          employee.setJobFacility("Apteka 9");
18.          employee.setPosition("Farmaceuta");
19.      }
20.
21.      @AfterEach
22.      public void after() {
23.          employee = null;
24.      }
25.
26.      @Test
27.      public void
Should_NotPatchCategory_When_EmployeeIsExempted_AndEducationIsValid() {
28.          EmployeeCategoryPatchDto employeeCategoryPatchDto = new
EmployeeCategoryPatchDto();
29.          employeeCategoryPatchDto.setCategory(Category.FIRST);
30.
31.          EmployeeEducationPatchDto employeeEducationPatchDto = new
EmployeeEducationPatchDto();
32.          employeeEducationPatchDto.setEducation(Education.HIGHER);
33.          employeeEducationPatchDto.setEduName("AGH");
34.          employeeEducationPatchDto.setEduGraduationDate(LocalDate.of(2020, 5,
5));
35.
36.          BeanUtils.copyProperties(employeeEducationPatchDto, employee);
37.
38.          employee.setExempted(true);
39.
40.          employee = employeeDataService.save(employee);
41.
42.          Employee newEmployee = employeeDataService.patch(employee.getId(),
employeeCategoryPatchDto);
43.
44.          assertNotNull(newEmployee);
45.          assertNull(newEmployee.getCategory());
46.      }
```

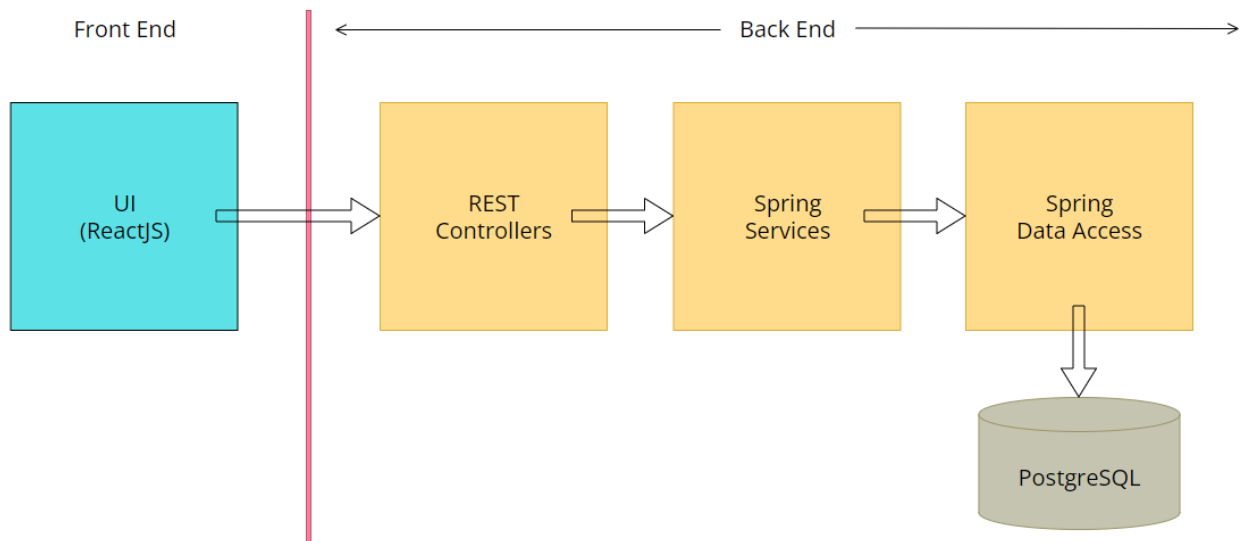
```
47.  
48.    /// inne metody testowe  
49.  
50. }
```

Listing 26: Klasa testowa `EmployeeDataServiceIntegrationTests` z jedną metodą testową, sprawdzającą metodę `patch`

Test pokazany na listingu 26 jest integracyjny. Do liniiki 40 przygotowuje się dane do testowania. Tworzy się obiekt klasy `Employee`, obiekt DTO klasy `EmployeeEducationPatchDto`, obiekt DTO `EmployeeCategoryPatchDto`. Następnie w liniice 40 obiekt `employee` zapisuje się do bazy poprzez wywołanie metody `save` z testowanego serwisu. Dalej w liniice 42 wywołuje się metodę `patch`, mającą za zadanie edytować kategorię pracownika. Ale nie powinno się to stać, ponieważ w liniice 38 założono, że pracownik jest zwolniony (ang. *exemptioned*). A metoda `patch` w serwisie jest tak zaimplementowana, że dla zmiany kategorii pracownik nie powinien być zwolniony. Sprawdza się to tak, że otrzymany obiekt `newEmployee` istnieje (linijka 44), a jego kategoria nie istnieje (linijka 45). W taki sposób podczas tego testu „pod maską” zostało trzy razy nawiązane połączenie z bazą, co jest bardzo dobrym testem integracyjnym. Warto też podkreślić, że wszystkie serwisy zostały zaimplementowane w trybie *test-first*, czyli zgodnie z TDD.

5.9 Test integracyjny – warstwa kontrolera

Celem testów integracyjnych na poziomie kontrolerów jest sprawdzenie poprawności całego *backendu* aplikacji, ponieważ kontroler znajduje się na szczycie hierarchii serwera (patrz: rys. 11).



Rysunek 11: Schemat przepływu danych w aplikacji

Przeprowadzając niezmierną ilość refaktoryzacji na początku tworzenia aplikacji, zauważono pewną regularność w relacjach między komponentami programu. Testując integrację kontrolera z serwisem, który w tym czasie komunikuje się z bazą danych można sprawdzić dosłownie wszystko. Nie ukrywam faktu, że warstwa kontrolerów została zaimplementowana nie w trybie TDD. Testy były przeprowadzone po napisaniu rzeczywistego kodu. Motywacją tego jest to, że testowanie kontrolerów odbywa się z trzech stron – ze strony testów automatycznych, ze strony narzędzia do wysyłania zapytań Postman oraz ze strony testów manualnych. Analizując ten fakt, zdecydowano przyspieszyć napisanie aplikacji, porzucając TDD na tym etapie.

Na listingu 27 zostało zademonstrowane integracyjne testowanie kontrolera `EmployeeController`. Są tu kilka ciekawych rzeczy do omówienia.

```
1. /// Integration tests
2. @RunWith(SpringJUnit4ClassRunner.class)
3. @SpringBootTest(classes = PracaInzLedikomKursyApplication.class)
4. @WebAppConfiguration
5. @ActiveProfiles("test")
6. @WithMockUser(username = "admin", roles = {"USER", "ADMIN"})
```

```

7. public class EmployeeControllerTests {
8.
9.     private MockMvc mockMvc;
10.
11.     @Autowired
12.     private WebApplicationContext webApplicationContext;
13.
14.     @Autowired
15.     private EmployeeDataService employeeDataService;
16.
17.     private Employee employee;
18.
19.     @Before
20.     public void before() {
21.         mockMvc =
MockMvcBuilders.webAppContextSetup(webApplicationContext).build();
22.
23.         employee = new Employee();
24.         employee.setForeignId(1L);
25.         employee.setFullName("Test");
26.         employee.setHiringDate(LocalDate.of(2020, 5, 5));
27.         employee.setJobFacility("Apteka 9");
28.         employee.setPosition("Farmaceuta");
29.     }
30.
31.     @After
32.     public void after() {
33.         employee = null;
34.     }
35.
36.     @Test
37.     public void Should_PatchEmployeeEducation() throws Exception {
38.         employee = employeeDataService.save(employee);
39.
40.         Map<String, String> jsonMap = new HashMap<>();
41.         jsonMap.put("education", "HIGHER");
42.         jsonMap.put("eduName", "AGH");
43.         jsonMap.put("eduGraduationDate", "05.05.2020");
44.
45.         String requestBody = new JSONObject(jsonMap).toString();
46.
47.         MvcResult mvcResult =
mockMvc.perform(MockMvcRequestBuilders.patch("/api/v1/employees/" +
employee.getId() + "/education")
48.             .contentType("application/merge-
patch+json").content(requestBody)).andReturn();
49.
50.         assertEquals(200, mvcResult.getResponse().getStatus());
51.
52.         String jsonResponse = mvcResult.getResponse().getContentAsString();
53.         JSONObject jsonObject = new JSONObject(jsonResponse);
54.
55.         assertEquals("AGH", jsonObject.get("eduName"));
56.     }

```

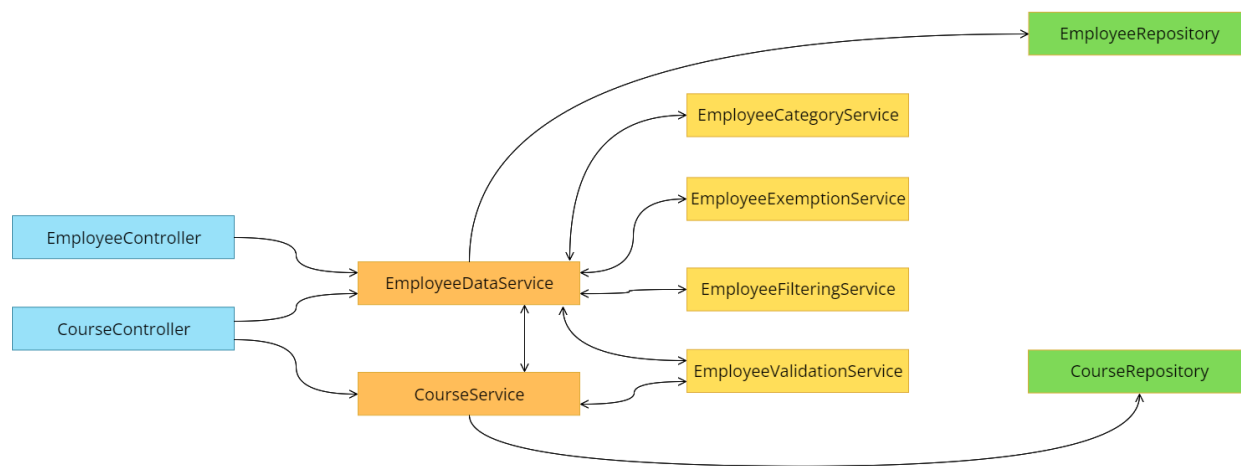
```
57.  
58.    /// inne testy integracyjne  
59.  
60. }
```

Listing 27: Klasa testowa z jedną metodą, testującą punkt końcowy
/api/v1/employees/{id}/education zapytaniem PATCH

Patrząc na listing 27, bardzo uważny czytelnik zauważy, że w odróżnieniu od poprzednich listingów teraz zamiast adnotacji `@BeforeEach` używa się adnotacja `@Before`. Przyczyną tego jest użycie w tej klasie czwartej wersji frameworku JUnit, zamiast piątej. Jak się okazało, konfiguracja klasy testującej kontroler w tej wersji jest znacznie prostsza. W 6 linijce można zauważyć adnotację `@WithMockUser`, odpowiadającą za stworzenie kontekstu z użytkownikiem admin, który ma pewne role. Bez wyspecyfikowania użytkownika testowanie się nie uda, ponieważ wszystkie punktu końcowe są zabezpieczone przez Spring Security i przyjmują zapytania tylko od pewnych użytkowników. W linijce 21 tworzy się obiekt `mockMvc`, który odpowiada za wysyłanie testowego zapytania. W linijce 47 wysyła się zapytanie PATCH do punktu końcowego, a w linijce 48 specyfikuje się typ kontentu oraz ciało zapytania. Metoda `andReturn` zapisuje wynik zapytania do obiektu `mvcResult`. Następnie w linijce 55 sprawdza się, że ciało odpowiedzi zawiera dane, które powinny się były nadpisać w stanie obiektu pracownika.

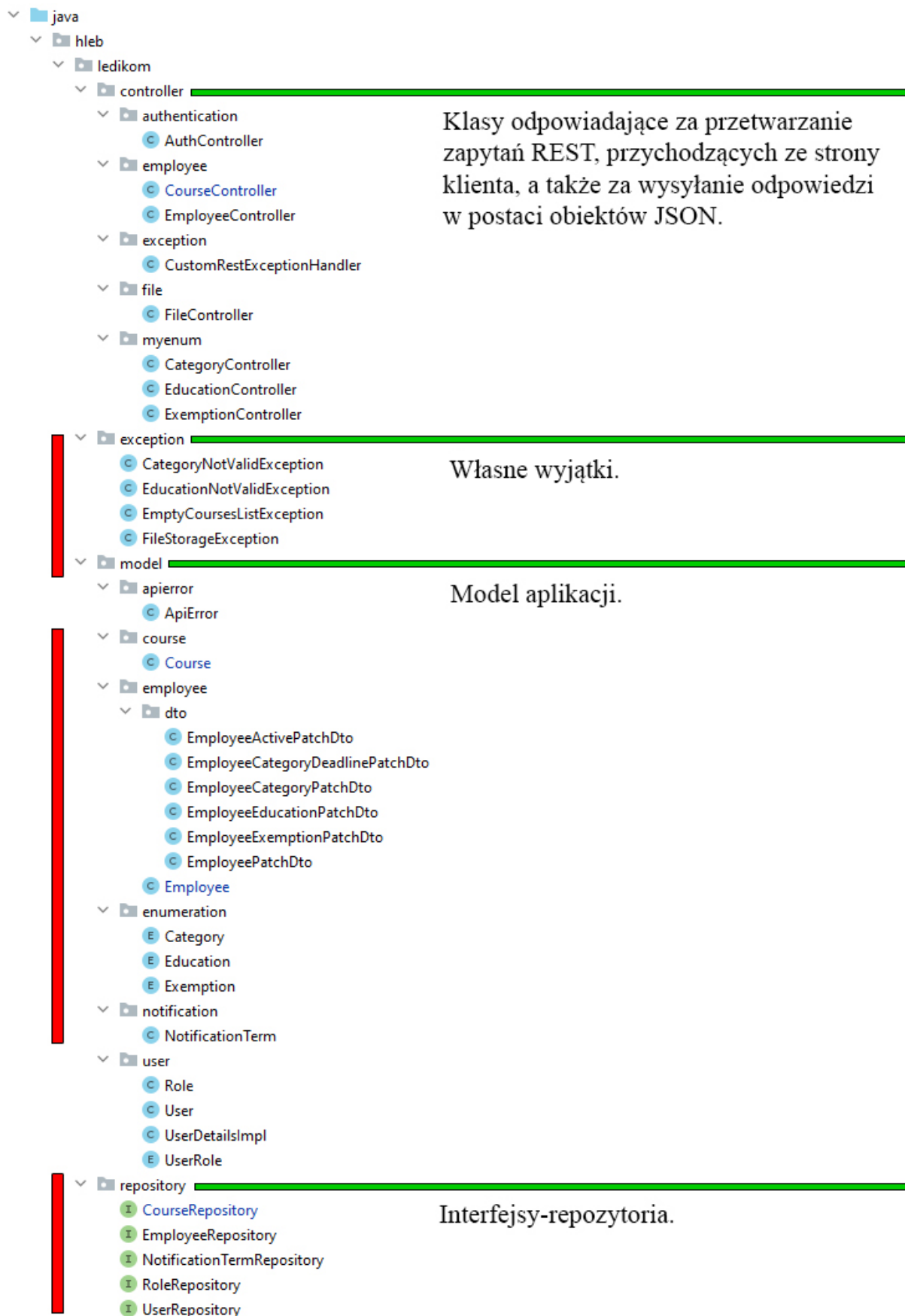
5.10 Ostateczny design

Po wielu kolejnych refaktoryzacjach otrzymano ostateczny design aplikacji (patrz: rys. 12). Na schemacie są pokazane tylko najważniejsze klasy, odpowiadające za najbardziej znaczące zadania logiki biznesowej. Pełną listę klas można zobaczyć na rysunkach 13 oraz 14. Jak już było pokazane na rysunku 11, część serwerowa aplikacji jest rozbita na trzy kawałki: kontroler, serwis oraz dane (*repository*). Widać to także na rysunku 12. Informacje ze strony klienta przyjmują kontrolery, następnie przekazują dane do serwisu, gdzie włącza się logika biznesowa. Dalej dane trafiają do bazy danych za pomocą interfejsu-repozytorium. Także widać, że kontroler `EmployeeController` wywołuje metody tylko z jednej klasy z przyrostkiem `Employee`. Jest to zrobione w celu lepszej czytelności kodu oraz dla zwiększenia hermetyzacji.



Rysunek 12: Przepływ danych w najważniejszych klasach części serwerowej aplikacji

Czerwonymi pionowymi kreskami na rysunkach 13 oraz 14 są zaznaczone klasy, które zostały napisane ściśle według zasad TDD. Są to głównie klasy odpowiadające za logikę biznesową aplikacji, czyli klasy z modelu aplikacji oraz klasy-serwisy, a także pomocnicze klasy i interfejsy. W sumie wszystkie klasy kodu rzeczywistego zawierają **2836 linii kodu**.

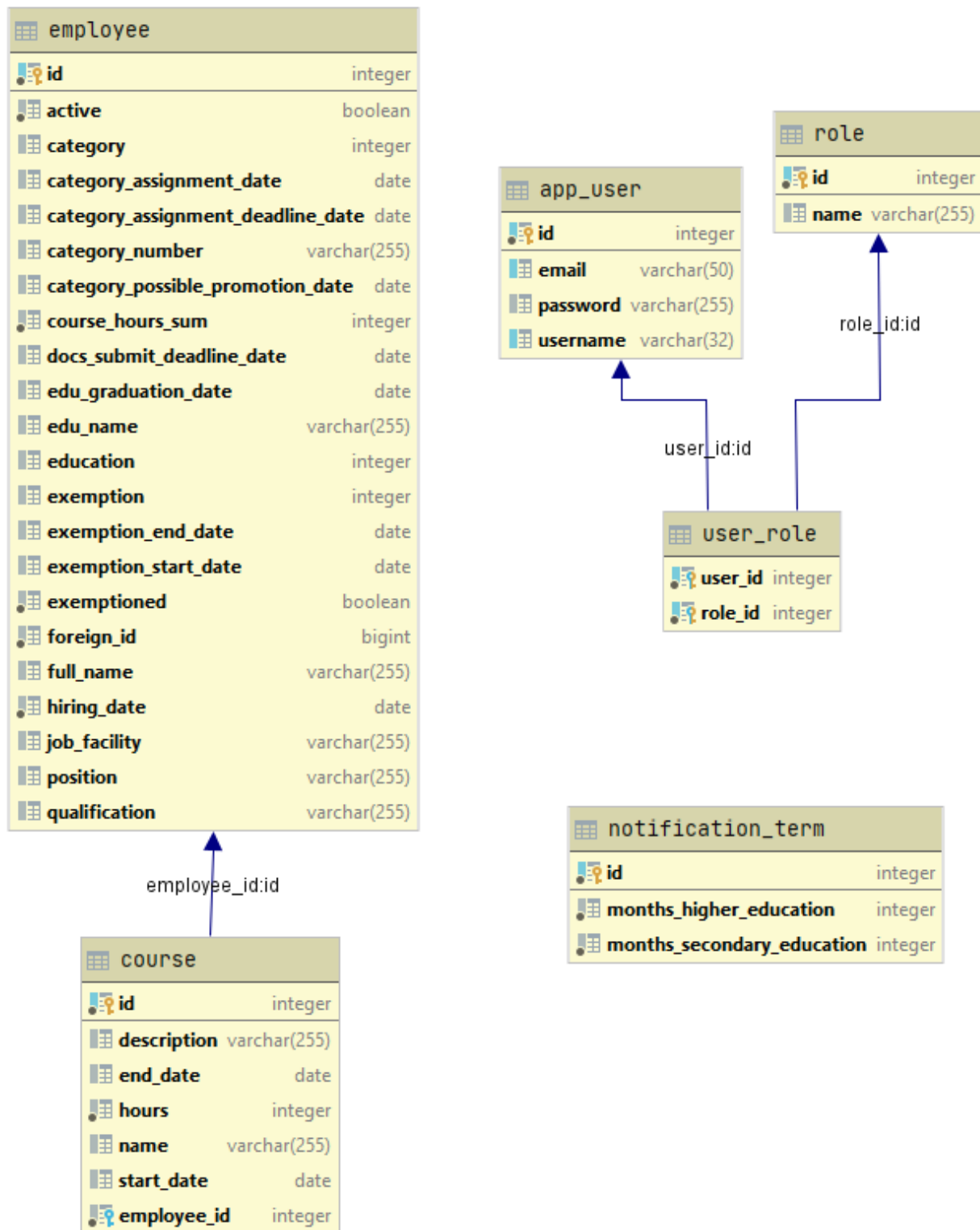


Rysunek 13: Struktura pakietów części serwerowej aplikacji (1)



Rysunek 14: Struktura pakietów części serwerowej aplikacji (2)

Na rysunku 15 widać ERD dla bazy danych. Zawiera on trzy encje na potrzeby logiki biznesowej (`employee`, `course`, `notification_term`) oraz trzy encje na potrzeby bezpieczeństwa aplikacji (`app_user`, `user_role`, `role`).



Rysunek 15: Diagram encji bazodanowych wygenerowany w programie DataGrip

5.11 Testy

Na rysunku 16 można zobaczyć strukturę katalogów, zawierających klasy testowe. W sumie wszystkie te klasy zawierają **114 testów** i **2674 linijki kodu**.



Rysunek 16: Struktura pakietów, zawierających klasy testowe

6. Podsumowanie

6.1 Wnioski

Pisząc projekt dyplomowy, udało się wypróbować TDD z dwóch stron:

- pisać testy w taki sposób, aby każdy test wymagał dodania pojedynczej linii do kodu rzeczywistego oraz niewielkiej refaktoryzacji;
- pisać testy w taki sposób, aby każdy z nich wymagał dodania setek linii kodu oraz ogromnej ilości refaktoryzacji.

TDD zakłada to, że należy umieć pracować i tak, i tak. Ale ogólny trend jest taki, że im mniejsze kroki, tym lepiej. Oczywiście, na początku programowania należy robić tylko i wyłącznie małe kroki, żeby odczuć rytm TDD. Dalej po uzyskaniu pewności siebie można zwiększyć te kroki, robić bardziej agresywne ruchy podczas refaktoryzacji. W ten sposób może być lepiej widać, która struktura kodu bardziej pasuje do sytuacji.

Podczas pisania aplikacji potwierdziłem na własnych doświadczeniach ogólnie wiadomy fakt, że testować należy tylko kod zawierający instrukcje warunkowe, cykle oraz operacje. Natomiast testowanie np. takich kawałków kodu, jak gettery/settery jest zbędne. Te rzeczy zostaną dobrze przetestowane podczas testowania większych kawałków.

Uważam, że testy są dobrze zaprojektowane i odpowiadają standardom dobrego testu. Wskaźnik „linijek kodu testowego na test” wynosi 23.5. Mówi to o tym, że test nie wymaga dużej ilości kodu inicjalizacyjnego. Także starałem się unikać duplikowania kodu inicjalizacji. Ten kod należy umieszczać w specjalnie wydzielonym miejscu, np. w metodzie oznaczonej adnotacją `@BeforeEach`, jak to było pokazane w kilku podrozdziałach dotyczących implementacji. Znacznie zwiększy to czytelność kody testowego. Także podczas wielokrotnego uruchomienia testów (przynajmniej 2 razy dziennie) nie wykryto niespodziewanych awarii.

Wskaźnik „linijek kodu testowego na linie kodu rzeczywistego” wynosi 0.94. Jest to bardzo wysoka wartość. Dla gigantycznego projektu systemu LifeWare³³, napisanego zgodnie z TDD, ten wskaźnik wynosi 1.

³³ K. Beck, TDD. Sztuka tworzenia dobrego kodu, Helion 2014, s. 211.

Ogólnie podsumowując pracę nad projektem, można powiedzieć, że udało się zrealizować prawie wszystkie cele. Prawie wszystkie, ponieważ zabrakło mi czasu na zaimplementowanie jednej dodatkowej funkcjonalności, której realizacja została przeniesiona na kolejny etap rozwoju aplikacji.

Odnosząc się do głównych celów niniejszej pracy, można powiedzieć, że udało się zdobyć cenne doświadczenie nie tylko w TDD, ale również w całym szeregu innych technologii. Odpowiadając na pytanie: „Czy poziom wejścia do tej metodyki jest akceptowalny dla programisty niezbyt doświadczonego w jakimkolwiek testowaniu?”, mogę powiedzieć, że napotkane problemy były związane nie tyle z napisaniem testów, ile z konfiguracją klas testowych, ponieważ testowanie każdej warstwy aplikacji wymaga różnych ustawień. Także w pełni udało się przedstawić zasadnicze cechy oraz wzorce TDD.

6.2 Możliwości rozwoju

Aplikacja została napisana zgodnie z zasadami TDD, w związku z tym jest bardzo dobrze pokryta testami. TDD wymusiło też, aby kod spełniał ogólnie przyjęte reguły programowania obiektowego. Na podstawie tych dwóch faktów można powiedzieć, że dalszy rozwój aplikacji nie będzie problemem. Po pierwsze w planach jest jedna funkcjonalność, na implementowanie której zabrakło czasu. Jest to generowanie raportów w postaci plików PDF zawierających plan certyfikacji pracowników, a także generowanie dokumentacji związanej z certyfikacją poszczególnych pracowników. Od strony bezpieczeństwa zostanie zaimplementowane automatyczne odświeżanie tokenu JWT, ponieważ teraz ma on nieskończony czas ważności, co grozi bezpieczeństwu aplikacji. Zostaną także dodane testy automatyczne dla poziomu bezpieczeństwa, ponieważ teraz są to testy manualne. Następnie wskazana będzie refaktoryzacja kodu źródłowego interfejsu użytkownika, ponieważ jest on dość „brudny”. W planach jest dostosowanie aplikacji do obsługi wielu użytkowników, ponieważ w przyszłości może chcieć z niej skorzystać wiele firm medycznych. Ale także istnieje możliwość rozwoju aplikacji w takim kierunku, że będzie się wdrażać ją dla każdej firmy osobno.

Załączniki

Kod źródłowy aplikacji: <https://github.com/hlebshypulahub/AGH-Hleb-Shypula>

Link do aplikacji: <https://master.d2iyvc7w20vpow.amplifyapp.com>

Przykładowe dane do logowania:

Nazwa użytkownika: **username**

Hasło: **password**

Bibliografia

1. K. Beck, *TDD. Sztuka tworzenia dobrego kodu*, Helion 2014.
2. V. Farcic, A. Garcia, *Test-Driven Java Development*, Packt Publishing Ltd., Birmingham 2015.
3. K. Beck, *Test-Driven Development By Example*, Addison Wesley 2002.
4. <https://bit.ly/3xRUGCE>, *Czym jest TDD*, [dostęp: 14.08.2021].
5. <https://bit.ly/3m94iqK>, *Trzy kroki TDD*, [dostęp: 15.08.2021].
6. <https://bit.ly/3z4S7hW>, *O testach jednostkowych i TDD*, [dostęp: 18.08.2021].
7. <https://bit.ly/3qt6Os3>, *Klucze do udanego testowania jednostkowego – jak programiści testują swój własny kod*, [dostęp: 20.12.2021].
8. <https://bit.ly/3sust5M>, *Czym jest Testowanie integracyjne*, [dostęp: 20.12.2021].
9. <https://bit.ly/3pJeKqn>, *Zaprojektuj warstwę danych*, [dostęp: 27.10.2021].
10. <https://bit.ly/3sv52JH>, *Czym jest Java? Definicja, znaczenie i funkcje platform Java*, [dostęp: 23.12.2021].
11. <https://bit.ly/3poE6JI>, *Co to jest Spring Framework? Niekonwencjonalny przewodnik*, [dostęp: 23.12.2021].
12. <https://bit.ly/3yXYYLc>, *Podstawy Java: Czym jest Spring Boot*, [dostęp: 23.12.2021].
13. <https://bit.ly/32hpzaa>, *JUnit 5 User Guide*, [dostęp: 24.12.2021].
14. <https://bit.ly/3yReuIE>, *Mockito i jak go używać*, [dostęp: 24.12.2021].
15. <https://bit.ly/3ms25pm>, *Tech 101: What Is React JS*, [dostęp: 24.12.2021].
16. <https://bit.ly/3Epzl6M>, *Biblioteka React UI, której zawsze chciałeś*, [dostęp: 26.12.2021].
17. <https://bit.ly/2ZeP7Ty>, *Rozporządzenie ministerstwa zdrowia Białorusi z dnia 28 maja 2021 r. №70, o profesjonalnej certyfikacji pracowników medycznych, farmaceutycznych oraz innych pracowników służby zdrowia*, [dostęp: 20.12.2021].
18. <https://bit.ly/30Z26tf>, *1С:Бухгалтерия 8*, [dostęp: 27.12.2021].

Spis ilustracji

Rysunek 1: Cykl TDD, składający się z pięciu kroków.....	6
Rysunek 2: Odpowiedź serwera przy próbie rejestracji użytkownika z błędną nazwą użytkownika...	35
Rysunek 3: Poprawna rejestracja użytkownika.....	36
Rysunek 4: Okno logowania wraz z komunikatem o błędzie.....	37
Rysunek 5: Dane użytkownika zapisane do Local Storage przeglądarki.....	37
Rysunek 6: Odpowiedź serwera przy próbie zapytania z błędnym tokenem.....	38
Rysunek 7: Strona z listą wszystkich pracowników.....	40
Rysunek 8: Strona z danymi pracownika oraz oznaczony na czerwono przycisk do edycji wykształcenia	41
Rysunek 9: Formularz do edycji wykształcenia pracownika.....	42
Rysunek 10: Przykładowy plik CSV, zawierający dane pięciu pracowników.....	49
Rysunek 11: Schemat przepływu danych w aplikacji.....	52
Rysunek 12: Przepływ danych w najważniejszych klasach części serwerowej aplikacji.....	55
Rysunek 13: Struktura pakietów części serwerowej aplikacji (1).....	56
Rysunek 14: Struktura pakietów części serwerowej aplikacji (2).....	57
Rysunek 15: Diagram encji bazodanowych wygenerowany w programie DataGrip.....	58
Rysunek 16: Struktura pakietów, zawierających klasy testowe.....	59

Spis listingów

Listing 1: Metoda testowa oznaczona adnotacją frameworku testowego JUnit.....	24
Listing 2: Do metody testowej została dołączona metoda statyczna, porównująca dwie wartości.....	24
Listing 3: W metodzie testowej został stworzony obiekt klasy Employee.....	25
Listing 4: Klasa Employee, zawierająca publiczne pole typu String – category.....	25
Listing 5: Przyczyna nieudania się testu oraz oczekiwany i rzeczywisty parametry metody statycznej assertEquals.....	25
Listing 6: Publiczne pole category w klasie Employee zostało zainicjalizowane.....	26
Listing 7: Dodanie konstruktora z parametrem category, gettera do tego pola, które teraz już jest prywatne.....	26
Listing 8: Metoda testowa, w której obiekt jest tworzony za pomocą konstruktora z parametrem, a wartość pola category pobiera się za pomocą gettera.....	27
Listing 9: Publiczne niezmiennicze pole statyczne typu LocalDate, znajdujące się w klasie Employee i przechowujące datę wejścia rozporządzenia w życie.....	27
Listing 10: Test sprawdzający, czy ilość dni między datami jest zgodna z obiektem notificationTerm.....	28
Listing 11: Klasa testowa skonfigurowana na potrzeby testów integracyjnych z bazą danych.....	29
Listing 12: Kod klasy NotificationTerm, która jest encją odwzorowania obiektowo-relacyjnego (skrót ang. ORM).....	30
Listing 13: Metoda testowa, sprawdzająca, czy pracownik jest aktywny.....	31
Listing 14: Metoda isActive w klasie Employee zwracająca stałą.....	31
Listing 15: Metody before oraz after, a także dwie metody testowe, sprawdzające, czy pracownik jest aktywny.....	31
Listing 16: Część kodu klasy Employee, zawierająca prywatne pole active oraz odpowiedni getter oraz setter.....	32
Listing 17: Typ wyliczeniowy Category oraz jego zastosowanie w klasie Employee.....	33
Listing 18: Pola klasy Employee oznaczone adnotacjami walidacji oraz adnotacjami frameworku Hibernate.....	38
Listing 19: Funkcje w języku JavaScript odpowiadające za wysyłanie zapytania PATCH w celu edycji wykształcenia pracownika.....	43
Listing 20: Metoda w klasie REST kontrolera, przyjmująca zapytanie PATCH.....	43
Listing 21: Kod DTO EmployeeEducationPatchDto wraz z adnotacjami walidacji.....	44

Listing 22: Kod metody patch serwisu EmployeeDataService, odpowiadający za kopiowanie wartości DTO do obiektu klasy Employee.....	44
Listing 23: Testy jednostkowe dla DTO EmployeeEducationPatchDto.....	45
Listing 24: Testy jednostkowe dla serwisu EmployeeDataService.....	46
Listing 25: Kawałek kodu serwisu EmployeeDataService, demonstrujący technikę wstrzykiwania zależności poprzez konstruktor.....	48
Listing 26: Klasa testowa EmployeeDataServiceIntegrationTests z jedną metodą testową, sprawdzającą metodę patch.....	50
Listing 27: Klasa testowa z jedną metodą, testującą punkt końcowy /api/v1/employees/{id}/education zapytaniem PATCH.....	52