OŚRODEK PRZETWARZANIA INFORMACJI
PAŃSTWOWY INSTYTUT BADAWCZY

POL-on

System Informacji o Nauce i Szkolnictwie Wyższym

TOM II

redaktor naukowy
Marek Michajłowicz

Systemy informatyczne
wspierające naukę i szkolnictwo wyższe

POL-on:
System Informacji o Nauce i Szkolnictwie Wyższym
TOM II

Redakcja naukowa:
Marek Michajłowicz

OŚRODEK PRZETWARZANIA INFORMACJI
PAŃSTWOWY INSTYTUT BADAWCZY

Systemy informatyczne wspierające naukę i szkolnictwo wyższe
POL-on: System Informacji o Nauce i Szkolnictwie Wyższym
TOM II

Redakcja naukowa:
Marek Michajłowicz

Redakcja techniczna:
Krystyna Fetera, Robert Siewiorek

Autorzy:
Artur Idzi: rozdział 3, 4
dr Jakub Kierzkowski: rozdział 3
Iwona Kucharska: rozdział 2
Marek Michajłowicz: rozdział 1, 2, 3, 4
Małgorzata Paszkowska: rozdział 2
Emil Podwysocki: rozdział 4
dr Piotr Rodzik: rozdział 1, 2
Andrzej Sadłowski: rozdział 3, 4
Antoni Sobkowicz: rozdział 4

Recenzent:
dr hab. Zenon A. Sosnowski, prof. Politechniki Białostockiej

Wydawca:
Ośrodek Przetwarzania Informacji – Państwowy Instytut Badawczy
al. Niepodległości 188b
00-608 Warszawa
e-mail: opi@opi.org.pl
www.opi.org.pl

ISBN 978-83-63060-19-0 (całość)
ISBN 978-83-63060-23-7 (tom drugi)

Projekt okładki i opracowanie graficzne:
Karina Maszewska

Skład i łamanie:
dr Jakub Kierzkowski

Druk:
Drukarnia DSS Szczepan Szymański
ul. Wolska 108, 05-119 Wola Aleksandra

SPIS TREŚCI

ROZDZIAŁ 3

METODY WYTWÓRCZE

Marek Michajłowicz
Artur Idzi
Andrzej Sadłowski
dr Jakub Kierzkowski

W poprzednich rozdziałach przedstawione zostały wszystkie istotne fakty i analizy na gruncie teorii budowy systemów informatycznych. W tym miejscu uwaga autorów skupi się bardziej na elementach eksperymentalnych związanych z wytwarzaniem systemów informatycznych. Wnioski z doświadczeń zaprezentowano w obszarze szeroko rozumianej inżynierii oprogramowania, zarządzania projektem informatycznym w podejściu zwinnym oraz metodami kontroli jakości.

3.1. Metody zwinnego zarządzania projektem

Jedną z pierwszych strategicznych decyzji w momencie rozpoczęcia prac nad projektem POL-on w roku 2011 był wybór zwinnej metodyki scrum. W tym okresie zwinne podejście (ang. *agile*) do zarządzania procesem rozwoju oprogramowania zyskiwało dopiero szerszą popularność, zwłaszcza w Polsce. Dominujące było wciąż podejście kaskadowe lub zorientowane bardziej na sposób zarządzania projektem w ujęciu strategicznym (PRINCE2, RUP), aniżeli skupiające się na metodzie wytwarzania wymiernych przyrostów produktu. Niewątpliwą podstawą takiej decyzji był czas i duży poziom niepewności. Szczególną przesłanką był jednak labilny proces legislacyjny, w którym większość pojęć i wymagań w stosunku do nowego systemu formułowała się wraz ze wzrostem wiedzy i świadomością celów przedstawicieli administracji państwowej.

Zwinny proces tworzenia oprogramowania w oparciu o scrum bazuje na następujących założeniach [25]:

- szybkie (na wczesnym etapie realizacji projektu) dostarczenie działającego (użytecznego) rozwiązania;

- dostarczanie kolejnych działających elementów oprogramowania po każdej iteracji[1];
- akceptowanie zmian w zakresie wymagań, nawet na późnym etapie działania.

3.1.1. Organizacja zespołów wytwórczych

Podczas tworzenia systemu od razu przyjęto zasadę pracy w oparciu o koncepcję samoorganizujących się zespołów wytwórczych o multidyscyplinarnych kompetencjach [31]. Liczebność tych zespołów została określona na 7±2, czyli maksymalnie na dziewięć, a minimalnie na pięć osób. W ramach zespołu powinny znaleźć się wszystkie kompetencje potrzebne do realizacji kompletnego produktu. Do realizacji systemu informatycznego w formie usługi[2] w przyjętej technologii były to umiejętności programowania w warstwach front-end, back-end wraz z bazą danych (SQL) oraz umiejętności w zakresie analizy i projektowania systemów informatycznych z użyciem języków formalnych (UML• i testy[3]).

W początkowej fazie systemu kompetencje w zakresie UX• oraz projektowania graficznego interfejsu użytkownika zostały oddelegowane do innego zespołu funkcyjnego, który z uwagi na swą niewielką liczebność nie był w stanie partycypować w całym cyklu pracy zespołów wytwórczych. W początkowej fazie projektu kompetencje dotyczące administrowania infrastrukturą sprzętową oraz wdrażania były realizowane przez zespół liczący od 2 do 3 osób. Zadania te były częściowo realizowane również przez zespoły programistyczne. Z czasem to podejście zaczęto określać jako Devops•, zgodnie z obowiązującymi trendami [2].

Funkcję właściciela produktu (ang. *product owner)* pełnił analityk/projektant systemu, który formułował nie tylko wymagania co do ogólnej wizji i zastosowania produktu, ale również przygotowywał szczegółową dokumentację analityczną i projektową i był cały czas dostępny dla zespołu podczas trwania iteracji, udzielając szczegółowych odpowiedzi i wyjaśnień na wszystkie pytania[4]. Proces analizy i przygotowania projektu wyprzedzał prace w iteracji, żeby zminimalizować zmienność wymagań i problemy komunikacyjne oraz uwiarygodnić szacowanie. Rodziło to wiele napięć i problemów związanych z koniecznością godzenia pracy w toku (również związanej z utrzymaniem) oraz antycypowaniem i przygotowaniem zakresu do następnej iteracji. Zagadnienia te zostały szerzej omówione w podrozdziale 3.4.

W procesie powstawania i utrzymywania systemu brali udział również administratorzy baz danych oraz aplikacji, którzy zorganizowani byli w oddzielny zespół. W przypadku zespołu kontroli jakości również dominowała organizacja w ramach zespołu funkcyjnego (zespół testów) z oddelegowaniem do poszczególnych zespołów wytwórczych

• Termin wyjaśniony w słowniku
[1] w tym przypadku co dwa tygodnie
[2] *software as a service* (w skrócie SaaS), z ang. oprogramowanie jako usługa
[3] ze szczególnym naciskiem na ich automatyzację
[4] tzw. *proxy product owner*, czyli reprezentant interesów klienta po stronie dostawcy oprogramowania

w ramach iteracji[5]. Umożliwiało to rozwój i wymianę kompetencji oraz tworzenie standardów w ramach zespołu o podobnych zadaniach i kwalifikacjach, i przenoszenie ich na wymierne wsparcie we współpracy z zespołami programistycznymi w ramach iteracji. W takiej formie organizacji pracy nie było wyodrębnionego oddzielnego zespołu odpowiedzialnego za architekturę systemu.

Architekt systemu był równocześnie członkiem jednego z zespołów wytwórczych, co zapewniało mu ciągłą styczność z kodem i doświadczenia, które mógł przekuwać na standardy przenoszone w drodze porozumienia i uzgodnień z poszczególnymi członkami zespołów wytwórczych. Można powiedzieć, że w podejściu do architektury systemu dominował styl kolegialny, demokratyczny.

Miało to swoje wady i zalety. W przypadku technologii opartej na monolitycznej aplikacji z procesem ciągłej integracji do jednej dużej wersji (ang. *release*) udawało się zachować spójność i kontrolę oraz szybkie wprowadzanie nowych rozwiązań w drodze konsensusu. Odbywało się to jednak kosztem dużego zaangażowania architekta w operacje wdrożeniowe i integracyjne, które były oparte w znacznym stopniu o procesy manualne. W przypadku zmiany architektury na model mikrousługowy postawiono na bardzo dużą autonomię poszczególnych zespołów i organizację pracy w duchu kultury Devops [8]. W przypadku zespołów o zróżnicowanym stopniu kompetencji i podejściu do rozumienia architektury i konstrukcji systemu pojawiły się problemy komunikacyjne oraz kwestie związane z brakiem spójności całej aplikacji, jeżeli chodzi o stosowane konwencje.

Nie wpłynęło to jednak w znaczącym stopniu negatywnie na całość systemu, ponieważ bazował on już wtedy na paradygmacie autonomicznych mikroserwisów, powiązanych ze sobą poprzez jasno zdefiniowane kontrakty dla poszczególnych interfejsów.

Natomiast pozytywnie oddziaływało to na dynamikę pracy i zaangażowanie zespołów, które miały poczucie większego wpływu i dopasowania organizacji pracy do swojego stylu, nie odczuwając przy tym zbytnich ograniczeń ścisłymi konwencjami i ramami wyznaczanymi przez architekturę. Elementy rytualizacji pracy zespołowej w oparciu o scrum, przede wszystkim elementy retrospekcji i ciągłych spotkań, mających na celu wymianę doświadczeń i wiedzy o realizowanych zadaniach, wpłynęły pozytywnie na kulturę ciągłego doskonalenia [17]. Z czasem dzięki wspólnym warsztatom i wymianie wiedzy oraz doświadczeń pomiędzy zespołami wytworzyły się wspólne wzorce i konwencje rozwiązań architektonicznych.

W pierwszych latach w proces tworzenia systemu zaangażowanych było około 20-30 informatyków zorganizowanych w 2-4 zespołów wytwórczych dostarczających wymierne przyrosty funkcjonalne w kolejnych iteracjach.

[5] z czasem, wraz ze wzrostem liczebności zespołu oddelegowanie stało się bardziej stabilne i zbliżone do struktury macierzowej

Tabela 3.1. Tabela prezentująca strukturę zatrudnienia osób pracujących nad systemem POL-on w poszczególnych latach

Rok	Liczba zespołów wytwórczych	Liczba programistów	Liczba projektantów i testerów[6]	Liczba administratorów
2011	2	13	7	1
2012	2	13	7	3
2013	2	17	9	3
2014	3	13	9	4
2015	3	12	6	4
2016	3	16	9	5
2017	3	12	10	7
2018	4	12	10	9
2019	5	16	10	7

Tabela 3.1 prezentuje uśrednione wartości dotyczące liczby osób zaangażowanych w pracę nad systemem na przestrzeni wybranych miesięcy w poszczególnych latach. Są to wartości przybliżone na podstawie wewnętrznych i zewnętrznych rozliczeń czasu pracy. Fluktuacje w konkretnych latach wynikają z różnego stopnia oczekiwań ze strony interesariuszy oraz konieczności udziału również w innych projektach dla Ministra Ministerstwo Edukacji i Nauki. Ich przedstawienie ma na celu zobrazowanie skali osób zaangażowanych w utrzymanie i rozbudowę systemu oraz ich wpływu na sposób formowania się zespołów i organizację pracy. Z uwagi na charakter przedstawionego problemu badawczego nie zostały uwzględnione inne zespoły i stanowiska uczestniczące w realizacji systemu, lecz niepartycypujące w sposób bezpośredni w procesie wytwarzania oprogramowania (pracownicy *service desk*, kadra kierownicza, pracownicy wsparcia projektów etc.).

W roku 2014 projekt wszedł w kolejną fazę, co wpłynęło na zmianę form organizacji pracy. Przede wszystkim wzmocnił się podział na prace związane z utrzymywaniem w ruchu systemu i nad jego dalszym rozwojem i rozbudową. Zagadnienie to zostało szerzej przedstawione w kolejnym podrozdziale.

W roku 2018 rozpoczął się proces przebudowy i modernizacji systemu do architektury mikrousługowej z wyodrębnionymi usługami back-end i front-end. Spowodowało to konieczność wyodrębnienia oddzielnych zespołów funkcyjnych i większą specjalizację. Zagadnienia te i ich wpływ na organizację pracy zostały bardziej szczegółowo przedstawione w rozdziale 4, w części opisującej techniczne aspekty wdrażania modelu mikrousługowego.

[6] analityków, projektantów, testerów, programistów testów

3.1.2. Proces pozyskiwania wymagań

W pierwszej fazie budowy systemu POL-on szczególną rolę odegrało dostosowanie założeń metodyk zwinnych do procesu zbierania wymagań [23]. Osoby reprezentujące stronę biznesową (pracownicy Ministerstwa oraz reprezentanci pozostałych interesariuszy) współpracowali na bieżąco z analitykami lub projektantami systemu pełniącymi funkcję Właścicieli Produktu (ang. *Product Owner*), odpowiedzialnymi za realizację ich założeń i celów związanych z funkcjonowaniem systemu.

Funkcję koordynującą i kontrolującą zarządzanie poszczególnymi projektami w ramach modyfikacji i rozbudowy systemu pełniły gremia zorganizowane w ramach Ministerstwa Nauki i Szkolnictwa Wyższego. Dla projektów współfinansowanych przez Unię Europejską[7] za każdym razem stosowano bardzo sformalizowane podejście projektowe, wynikłe z metodyki PRINCE2, gdzie komitet sterujący reprezentował kluczowych interesariuszy projektu i sterował rozwojem systemu na gruncie strategicznym. W przypadku działań nieujętych w formalne ramy projektu z dofinansowania środków zewnętrznych, lecz będącego w zakresie bieżącego finansowania utrzymania i rozbudowy systemu w ramach dotacji celowej Ministra Nauki i Szkolnictwa Wyższego, kluczową rolę odgrywały dwa gremia:

- Zespół ds. POL-on przy Ministrze Nauki i Szkolnictwa Wyższego – w skład którego wchodzili przedstawiciele wszystkich kluczowych departamentów resortu oraz dostawcy oprogramowania;
- Zespół interesariuszy POL-on, powołany zarządzeniem Ministra zespół zrzeszający kluczowych reprezentantów interesariuszy systemu ze środowiska jego użytkowników (wyznaczonych przez przedstawicieli uczelni oraz instytutów naukowych), przedstawicieli wybranych departamentów nauki i szkolnictwa wyższego oraz dostawców oprogramowania [40].

Wykorzystano wszystkie możliwe formy komunikacji, również te mniej sformalizowane[8], które umożliwiały realizację przyjętych założeń:

- działające rozwiązanie jest miarą postępu projektu[25];
- prostota ma kluczowe znaczenie – istotna jest relacja pomiędzy ilością pracy wykonanej i niewykonanej;
- najlepsze wymagania, rozwiązania, metody pracy wyłaniają się w ramach efektywnego współdziałania członków zespołu projektowego;
- w regularnych odstępach czasu zespół projektowy poświęca czas na analizę własnego działania i ustala, w jaki sposób może działać bardziej efektywnie.

[7] w przypadku POL-onu wśród takich projektów można wyróżnić przede wszystkim projekt inicjalny, mający na celu stworzenie systemu oraz kolejny, rozpoczęty w roku 2015, mający na celu jego dostosowanie do potrzeb statystyki publicznej

[8] bezpośrednia rozmowa lub wspólne omówienie prototypu rozwiązania uważane jest za najbardziej efektywny sposób precyzowania wymagań

Kluczowe formy pozyskiwania wymagań to:

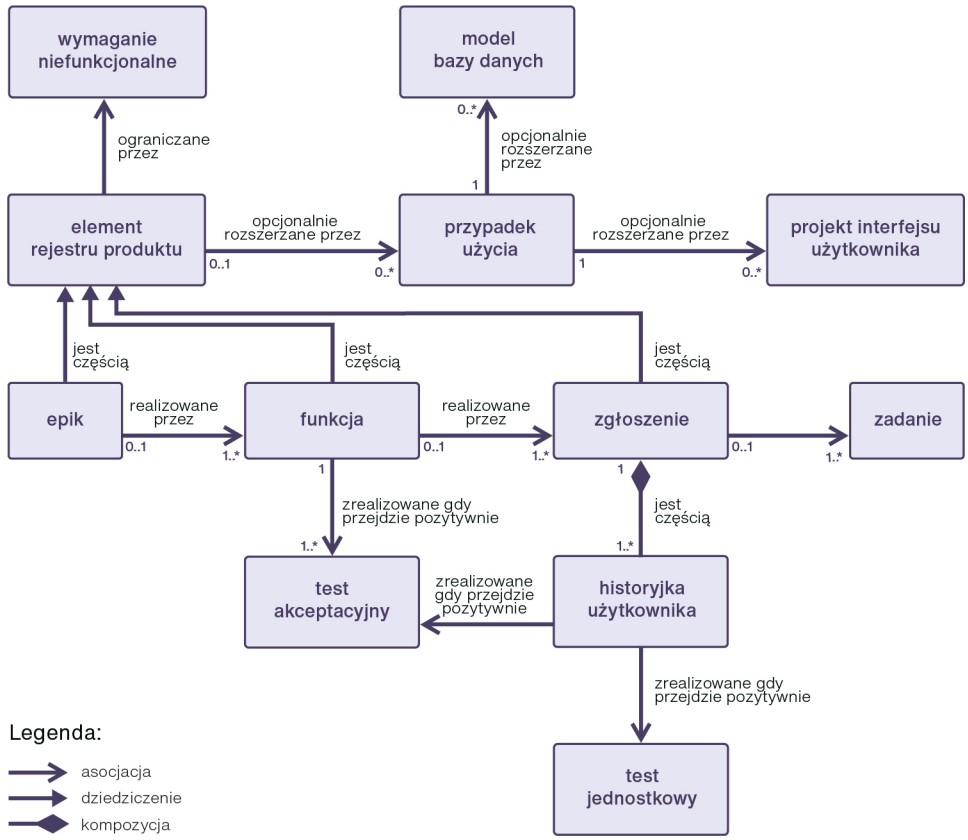
- akty normatywne. W przypadku systemów tworzonych w domenie rządowej kluczowe jest spełnienie wszystkich wymagań wynikających z treści prawa – przede wszystkim zapisów ustaw i rozporządzeń im dedykowanych;
- ustalenia grup i zespołu ds. POL-on;
- zgłoszenia od użytkowników końcowych poprzez elektroniczny system zgłoszenia;
- prototypowanie interfejsu systemu we współpracy z klientem biznesowym;
- warsztaty wymagań, podczas których wypracowywane są koncepcje rozwiązań;
- propozycje zmian do systemu zgłoszone dostępnymi kanałami komunikacyjnymi (e-mail, telefon).

3.1.3. Artefakty analityczne

Przez dłuższy czas w pracy nad wymaganiami szukano optymalnej formuły ich dokumentowania i przełożenia na projekt możliwy do realizacji dla zespołów programistycznych. Szukano również optymalnych artefaktów analitycznych. W pierwszej fazie za najbardziej efektywne formy specyfikowania uchodził projekt graficzny interfejsu użytkownika oraz model bazy danych wraz ze schematem XSD do specyfikowania struktur danych. Powodowało to problemy z zarządzaniem modelem dziedziny, czyli powstawaniem tzw. anemicznego modelu domeny [11].

Trudna do wyspecyfikowania w takiej formie była też logika biznesowa oraz reguły kontroli poprawności danych, dla których tworzono oddzielne słowniki i dokumenty. Największy problem stanowiła jednak szybka utrata trwałości takiej formy dokumentacji oraz brak syntetycznego opisu poszczególnych komponentów systemu. Zaletą była jednak niewątpliwie możliwość szybszego przejścia do prac programistycznych. Wraz ze wzrostem zatrudnienia oraz doświadczeń zespołu, wykuwanych w formie eksperymentów z różnymi typami rozwiązań, opracowano standardy w procesie dokumentowania oraz zarządzania repozytorium artefaktów analitycznych i projektowych.

Kluczowe dla efektywności procesu zarządzania procesem wytwórczym było właściwe zdefiniowanie najważniejszych artefaktów procesu i produktu. Ilustracja 3.1 przedstawia opisujący to metamodel.



Ilustracja 3.1. Model artefaktów procesu wytwórczego

Do najistotniejszych artefaktów modelu zaliczyć należy:

1. elementy rejestru produktu (ang. *backlog item*), na który składają się Epiki, agregujące historyjki użytkownika (ang. *user story*) realizowane na przestrzeni kilku iteracji, definiujące kluczowe cechy produktu *feature*[9] i faktycznie implementowane w formie historyjek użytkownika w klasycznej formie. Z historyjkami użytkownika mogły być powiązane pojedyncze zadania (ang. *tasks*), odpowiadające technicznym aspektom realizacji, lecz niewpływające na zmianę statusu całego przyrostu w procesie[10];
2. artefakty analityczne, przechowywane w oddzielnych repozytoriach. Lista artefaktów była zróżnicowana w zależności od rodzaju zmiany i poziomu złożoności. Za kluczowe artefakty w pierwszej fazie uznawano:

[9] w praktyce porzucone w trakcie realizacji jako zbyt abstrakcyjne elementy opisu pomiędzy *Epic* a *Story*
[10] przez tablicę sprintu przechodzić powinien pełny funkcjonalny przyrost w aplikacji, obejmujący wszystkie warstwy systemu, a nie tylko poszczególne elementy, jak np. baza danych, API, UI

a. listę wymagań niefunkcjonalnych zawierającą ograniczenia co do realizacji poszczególnych *user stories*[11];
b. strukturyzowane przypadki użycia zawierające ścieżkę główną i alternatywne przebiegi[12];
c. artefakty interfejsy użytkownika – projekt UI, wykonywalne makiety, analizy UX.
d. fizyczny model bazy danych[13].

3. artefakty kontroli jakości – składały się na nie głównie scenariusze testowe (ang. *test scenario)*, testy jednostkowe oraz wszelkie skrypty automatyzujące proces kontroli jakości, opisane bardziej szczegółowo w podrozdziale 3.6.

Jako rozwiązania, które okazały się szczególnie efektywne i godne opisania w niniejszej publikacji, zaliczyć należy:

- napisanie i wdrożenie własnego narzędzia do zarządzania słownikiem reguł biznesowych i walidacyjnych o nazwie Ruleski[14]. Jego wdrożenie było motywowane wzrostem świadomości roli reguł biznesowych w projektowaniu systemów informatycznych, opisywanym szczegółowo w literaturze [38]. System stanowił zbiór reguł o charakterze formalnoprawnym lub opisującym logikę biznesową na wyższym poziomie abstrakcji oraz konkretnych reguł walidacji pól bądź kroków procesu;
- jako rozwinięcie motywacji z poprzedniego punktu wykorzystano techniki specyfikowania w formie żyjącej dokumentacji, zwanej inaczej „specyfikacją poprzez przykłady”[15]. Wykorzystano do tego testy automatyczne uruchamiane w różnych warstwach systemu (UI, testowanie logiki aplikacji), które weryfikowały kryteria akceptacyjne zdefiniowane w procesie analizy i opisane w strukturalizowanym języku zbliżonym do naturalnego (Gherkin);
- utworzenie repozytorium artefaktów analitycznych z możliwością automatycznego generowania dokumentacji. Repozytorium powstało w oparciu o popularne oprogramowanie Enterprise Architect. Przypadki użycia oraz wszelkie modele (UML, SysML, BPMN) składowane były jako artefakty w dedykowanych pakietach, do których przygotowano szablony zgodne z firmowymi standardami dokumentowania. Dzięki temu do zbieranych artefaktów analitycznych generowana była szczegółowa, sformalizowana dokumentacja produktu. Repozytorium stosowało też elementy inżynierii wstecznej w obszarze zarządzania strukturą, fizycznym modelem bazy danych;
- wykorzystanie wykonywalnych prototypów jako rozwinięcie formy modelowania interfejsu użytkownika. Do zobrazowania sposobu działania systemu w warstwie UI wykorzystano możliwości narzędzia Axure RP, które odgrywało niepoślednią rolę szczególnie w kontaktach z klientem i analizie złożonych nawigacji i ścieżek interakcji użytkownika z systemem;

[11] głównie prawne, związane z ochroną danych osobowych, rozliczalnością oraz wydajnością
[12] na kolejnych etapach rozwijania systemu ich rola została mocno ograniczona jako mało efektywna w procesie komunikacyjnym
[13] porzucony w momencie odejścia od rozwiązań bazujących na zapisie w oparciu o relacyjną bazę danych
[14] planowane jest opisanie tego rozwiązania bardziej szczegółowo w formie artykułu naukowego poza niniejszą publikacją
[15] ang. *specification by example*

- wdrożenie sesji eventstormingu i formy dokumentowania modelu oraz logiki działania systemu jako mapy poszczególnych pojęć i zagadnień powstałych w wyniku odkrywania dziedziny przez zespół wytwórczy. Rozwiązania te zostały bardziej szczegółowo zaprezentowane w kolejnym rozdziale pracy, przy okazji technicznych aspektów pracy w modelu mikrousługowym z wykorzystaniem podejścia Domain Driven Design.

3.1.4. Planowanie i szacowanie zakresu

Najtrudniejszym elementem tworzenia systemu i zarządzania kolejnymi wydaniami było szacowanie zakresu prac do kolejnych wydań. Przyjęto założenia metodyki scrum, gdzie szacowaniu podlegał poziom złożoności danego wymagania opisanego w ramach *User Story*. Nie była estymowana pracochłonność związana z jej wykonaniem. Stosowano metrykę jednostek względnych (tzw. ang. *story points*). Do oszacowania przyjmowano zadania referencyjne, każdy zespół samodzielnie określał liczbę punktów takiego zadania. Możliwe było stosowanie kilku zadań referencyjnych przez jeden zespół. Zadania referencyjne mogły być zmieniane na wniosek członków zespołu.

Proces szacowania zakresu przebiegał podczas sesji tzw. „pokera planistycznego" w pierwszym dniu iteracji. Każdorazowo w drodze dyskusji i wymiany doświadczeń zespół starał się osiągnąć kompromis w zakresie liczby punktów przyznanych na dane zadanie. W przypadku braku możliwości ustalenia wspólnego wyniku wygrywała punktacja wybrana przez największą liczbę głosujących.

Jeżeli z przeprowadzonego oszacowania wynikało, że dane zadanie nie może zostać ukończone w ramach jednego sprintu, podlegało ono podziałowi na zadania podrzędne, które były szacowane oddzielnie. Jeżeli dana grupa funkcjonalności musiała być wdrożona w całości i jej zakres przekraczał jeden sprint, w porozumieniu z kierownikiem projektu była podejmowana decyzja o wydzieleniu oddzielnej gałęzi w repozytorium kodu i stworzeniu odrębnego środowiska testowego dla tej grupy funkcjonalności. Liczbę zadań możliwych do przyjęcia w ramach iteracji określała ekstrapolowana z poprzednich etapów średnia „prędkość" (ang. *velocity*[16]), pomniejszona o ewentualny bufor na obsługę błędów oraz prac utrzymaniowych, absencje, dni wolne i wszystkie czynniki mogące wpłynąć na efektywność prac.

Oszacowaniu podlegały wyłącznie zmiany funkcjonalne. Na obsługę błędów i prac związanych z utrzymaniem przyjmowany był bufor obniżający nominalną prędkość

[16] *velocity* reprezentuje ilość pracy wykonanej w każdym sprincie wyrażoną w *story points*. Planowana *velocity* zostaje oszacowana przez Zespół Scrumowy na początku każdego sprintu, a *User Stories* zostały przypisane do sprintu, tak aby suma *story points* mieściła się w ramach zdolności określonej przez oszacowanie *velocity*. Rzeczywista prędkość została obliczona na końcu sprintu, podsumowując *story points* dla wszystkich *User Stories* zaakceptowanych przez *Product Ownera*. Planowaną *velocity* należało oszacować, biorąc pod uwagę rzeczywistą prędkość poprzednich sprintów przy umowie braku żadnych zmian w zespole programistycznym w ciągu projektu [24], s. 2

zespołu do określonej liczby punktów możliwych do realizacji w ramach iteracji. Rozmiar bufora określał scrum master przed rozpoczęciem iteracji, sterując nim podczas jej trwania. W zależności od aktualnej liczby błędów, bufor mógł być zwiększany lub zmniejszany. W przypadku gdy jego zwiększenie zagrażało dokonaniu zmian funkcjonalnych zaplanowanych w iteracji scrum master musiał skonsultować swoją decyzję z kierownikiem projektu. Postęp realizacji planu mierzony był empirycznie za pomocą wykresu spalania (ang. *burndown chart*), dostępnego w systemie zarządzania zmianą (JIRA[17]).

3.1.5. Ewolucja w kierunku metody Kanban

Organizacja pracy w ramach scrum przyniosła wymierne korzyści w pracy nad projektem:

- kontrola przyrostu produktu dzięki realizacji zadań w oparciu o zdefiniowane okna czasowe (ang. *time box*);
- kultura ciągłego doskonalenia i uczenia się zespołów dzięki retrospekcjom, sesjom planowania i iteracyjnym przyrostom oprogramowania;
- nabywanie doświadczenia i większej precyzji w procesie szacowania pracochłonności;
- skupienie większości kompetencji i możliwości rozwiązywania określonych problemów w niewielkich, zróżnicowanych zespołach opartych na formie grupowej samoorganizacji.

Konsekwencją było przyjęcie większej tolerancji we wdrażaniu produktu, który miał ograniczony zakres funkcjonalny i uzupełnianie ich w ramach kolejnych iteracji poprzez zbieranie ciągłych uwag od użytkowników końcowych. Można powiedzieć, że w wielu przypadkach proces ten sprowadzał się do prototypowania określonych rozwiązań. Dzięki temu udało się uniknąć typowych problemów związanych z realizacją projektów informatycznych, czyli przekraczania czasu, kosztów oraz dostarczania zakresu funkcjonalnego, który nie jest używany przez użytkowników i nie zaspokaja ich potrzeb. Problemem były jednak koszty związane z poprawą błędów i zwiększoną kontrolą jakości oprogramowania po wdrożeniu (koszty regresji).

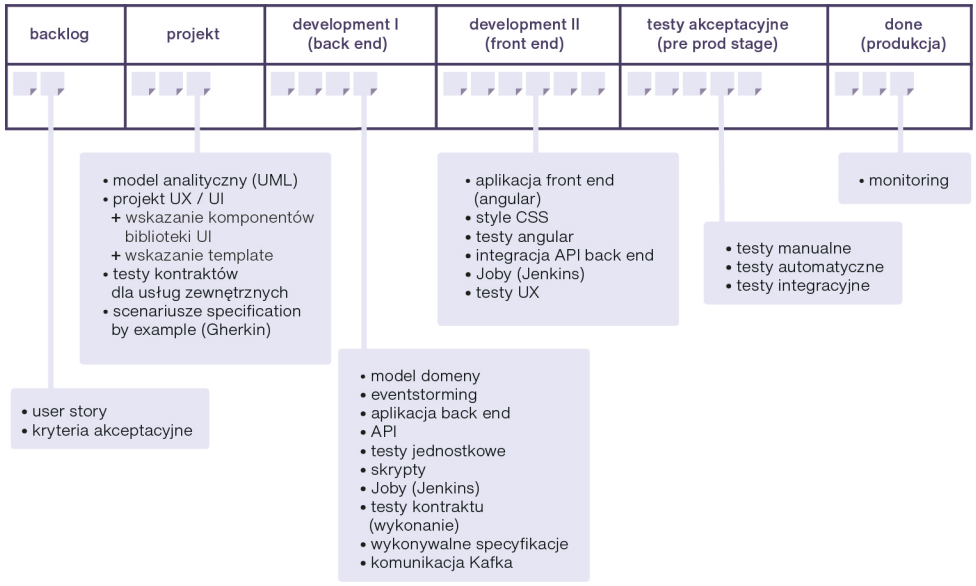
W momencie podjęcia decyzji o zmianie architektury rozwiązania na model mikrousługowy oraz zdefiniowaniu bardziej automatycznych przepływów zadań w ramach CI/CD okazało się, że iteracyjny proces oparty o scrum nie do końca współgrał ze zmianami technologicznymi. Częstokroć nie trzeba było czekać na pełną iterację, by osiągnąć wymierny przyrost produktu, ponieważ zmniejszyło się sprzężenie pomiędzy poszczególnymi modułami systemu i konieczność ich wdrażania w sposób zintegrowany. Oprogramowanie można było wdrażać w sposób bardziej autonomiczny w ramach jednego

[17] atlassian.com/software/jira (dostęp: 18.06.2021)

zespołu, co bardziej szczegółowo opisane zostało w podrozdziale 4.3. Rozpoczęto analizę i dyskusje wewnątrz zespołów, w której punktem wyjścia było oparte bardziej na metodyce kanban• w miejsce scrum. Główne założenia tego podejścia to skupienie na systemie przepływu zadań w ujęciu całościowym, wizualizacja oraz kontrola pracy w toku. Metodyka kanban stanowi formę zarządzania wywodzącą się z tak zwanego „szczupłego zarządzania" (ang. *lean management*), rozwiniętego w japońskich fabrykach[18].

Kluczowa w tym przypadku była zmiana paradygmatu ze skupiającego się na dostarczeniu określonego produktu w przewidzianym czasie podzielonym na iteracyjne interwały na zorientowany na sam proces produkcyjny, eliminację opóźnień, marnotrawstwa zasobów oraz skrócenie czasu dostarczenia produktów do klienta (ang. *lead time*). Takie założenia bardziej pasowały do architektury zorientowanej na autonomiczne, rozproszone usługi dostarczane i integrowane w trybie ciągłym, aniżeli na podejście w pełni iteracyjne.

W ramach działań związanych z analizą nowego procesu skupiono się na określeniu poszczególnych etapów i ich wizualizacji w formie tablicy kanban wraz ze zdefiniowaniem kryteriów sterowania przepływem. Poniżej znajduje się wyjściowa wizualizacja, która była rozwijana w toku dalszych dyskusji i wymiany doświadczeń.



Ilustracja 3.2. Uproszczony projekt tablicy kanban wraz z kryteriami

[18] metoda wywodzi się z formy podejścia do procesów produkcji i zarządzania stosowanych w fabrykach Toyoty

W poszczególnych torach pływackich opisane są czynności, które powinny być wykonane w ramach każdego etapu. Ich spełnienie stanowi formę weryfikacji (ang. *check list)* przed przekazaniem zadania do kolejnego etapu.

Kluczowe było wyodrębnienie etapu projektowania, który wcześniej nie był brany pod uwagę w podejściu scrum, oraz rozbicie fazy programowania na warstwy back-end i front-end. Było to zgodne z założeniem, że każdy moduł zostanie rozbity na dwie oddzielne aplikacje, które będą musiały się ze sobą komunikować. W fazie projektowania przyjęto, że wykorzystane będą komponenty biblioteki interfejsu, żeby można było wprowadzić element standaryzacji i automatyzmu w całej aplikacji. Projekt graficzny miał odpowiadać poszczególnym elementom komponentów biblioteki lub zainicjować potrzebę ich wytworzenia. Docelowo wykorzystane powinny być konkretne elementy szczegółowo opisane i udokumentowane w ramach showcase[19]. W założeniu miało to umożliwić wytwarzanie elementów aplikacji front-end również przez osoby nieposiadające zaawansowanych kompetencji w tym zakresie, będących raczej programistami warstwy back-end (API). Decyzja w tym zakresie, jak również koordynacja pracy oraz finalny wygląd aplikacji powinny być jednak pod kontrolą zespołu odpowiedzialnego za front-end, który weryfikowałby również UX dostarczanych rozwiązań. W praktyce polegało to bardzo często na oddelegowaniu programisty front-end do wsparcia pozostałych osób rozwijających aplikację.

Testy rozwiązań w fazie wytwórczej były rozproszone pomiędzy fazy development (I i II) i na tym etapie nie musiałyby obejmować wszystkich elementów weryfikacji poszczególnych przyrostów aplikacji, zwłaszcza jeżeli chodzi o integrację. Kluczowe testy odbywałyby się na środowisku preprodukcyjnym (testy zewnętrzne), najlepiej w formie przygotowanych wcześniej skryptów automatycznych.

3.1.6. Odpowiedzialności

Zadania w poszczególnych fazach mogą być wykonywane przy współudziale wielu osób i zespołów, natomiast dla lepszego określenia odpowiedzialności proponuję na początek zdefiniować, kto odpowiada za przesuwanie poszczególnych zadań pomiędzy torami pływackimi, a więc:

1. za przejście z backlogu do projektowania odpowiada Lider zespołu analityczno-projektowego;
2. za pobranie z fazy projektu do development I odpowiada Lider zespołu developerskiego (scrum master);
3. o przejściu zadania do fazy development II również decyduje scrum master, po uzgodnieniu z zespołem projektowania informacji (front-end). Samo zadanie może być wykonane przez dowolnego developera z zespołu odpowiedzialnego za dany

[19] zostały one opisane w rozdziale 4 w ramach opisu doświadczeń związanych z wytworzeniem własnej biblioteki komponentów UI

przyrost lub z zespołu projektowania informacji. Ważne, by decyzja co do faktycznego wykonania zapadała na podstawie realnej oceny zgodności i dostępności komponentów UI w bibliotece oraz możliwości wykonania (złożone rozwiązania UI powinny być wykonane bezpośrednio lub koordynowane przez programistę front-end);
4. Przekazanie zadania do fazy testów akceptacyjnych (*pre prod stage*) odbywa się za zgodą i wiedzą Lidera zespołu projektowania informacji (front-end);
5. faza testów akceptacyjnych (*pre prod stage*) to odpowiedzialność Lidera zespołu kontroli jakości, który decyduje, czy dana funkcjonalność może zostać wdrożona w środowisku produkcyjnym;
6. Niektóre zadania przechodziłyby pomiędzy torami w trybie push (przesunięcie do kolejnego etapu), a niektóre pull (pobranie zadania do realizacji, gdy jest to możliwe).

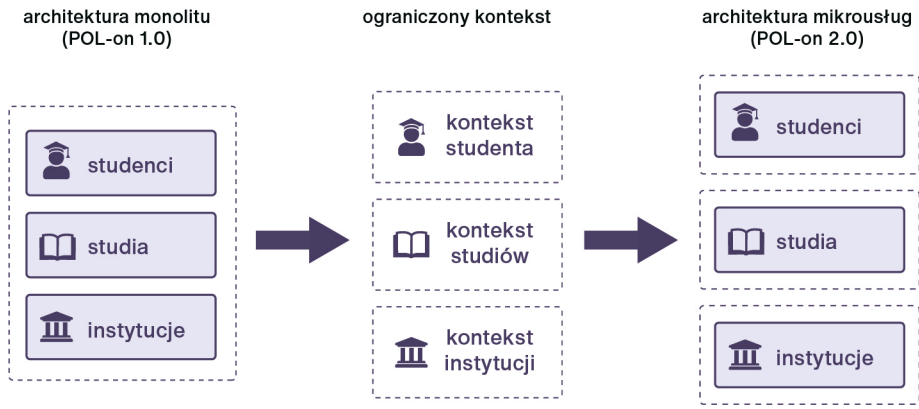
Tabela 3.2 oraz proces oparty o kanban jest aktualnie wprowadzany w sposób eksperymentalny w wybranych elementach w poszczególnych zespołach wytwórczych. Na etapie tworzenia niniejszej publikacji nie jest możliwe opisanie szerzej wniosków empirycznych z dotychczasowych doświadczeń we wdrażaniu kanban w całym procesie wytwórczym.

3.2. Zmiana paradygmatu w projektowaniu oprogramowania

Doświadczenie zdobyte podczas tworzenia poprzedniej wersji systemu pokazało, że model domeny powinien być przystosowany do ewolucji. Zespół programistów zdecydował się zastosować nowe podejście do tworzenia oprogramowania, znane jako Domain Driven Design [9]. Proces tworzenia oprogramowania zakładał ścisłą współpracę między wielofunkcyjnymi zespołami programistycznymi z ekspertami dziedzinowymi. Domena była odkrywana wspólnie przez zespoły i ekspertów domenowych za pomocą metody Event Storming [4]. Event Storming to warsztat umożliwiający eksplorację złożonych domen w sposób bardziej efektywny. W wyniku warsztatów i współpracy programistów z ekspertami dziedzinowymi, zespół deweloperski podzielił aplikację na moduły, które odzwierciedlają realia związane z nauką i szkolnictwem wyższym. Eksperci dziedzinowi byli zaangażowani w proces definiowania granic modułów (ograniczonych kontekstów) i identyfikowania rzeczywistych obiektów, które powinny znaleźć odzwierciedlenie w kodzie źródłowym (Ilustracja 3.3). W ten sposób model domeny był kształtowany przez wszystkich kluczowych uczestników.

Jednym z możliwych zastosowań tego podejścia jest sourcing zdarzeń, w którym zmiany stanu obiektów są zapisywane jako sekwencja zdarzeń [12]. W ten sposób, czytając dziennik zdarzeń, będzie można zrekonstruować obecny lub przeszły stan każdego obiektu. Gdy usługa wykryje zmianę danych, opublikuje obiekt zdarzenia. Każde opublikowane wydarzenie domeny będzie dostępne i gotowe do użycia dla wszystkich zainteresowanych modułów. Na czas reakcji systemu wpływają dwa główne czynniki: jedna relacyjna baza danych w warstwie infrastruktury oraz duża liczba operacji odczytu. Głównym celem tworzenia nowego systemu, oprócz zgodności z nowymi aktami prawnymi, było stworzenie wydajnej i przyjaznej dla użytkownika aplikacji. Aby to

osiągnąć, użyto dwóch różnych modeli danych w każdym module: pierwszy był używany do aktualizacji informacji, a drugi – do czytania informacji. Aby zaimplementować model domeny jako działające oprogramowanie, zespół programistów odwzorowywał ograniczone konteksty na mikrousługi. W tym podejściu każdy ograniczony kontekst jest traktowany jako potencjalny kandydat do wdrożenia jako mikroserwis. Nowa wersja systemu była zbiorem niezależnie wdrażanych usług. Rozwój systemu opierał się na kilku podstawowych założeniach, które są zgodne z filozofią tworzenia mikroserwisów. Oczekuje się, że będzie to system luźno powiązany, w którym mikroserwisy (moduły) będą miały niewielką wiedzę na temat pozostałych komponentów. Główną zaletą będzie możliwość wdrożenia, przetestowania i utrzymania każdego modułu osobno, co skróci czas potrzebny do wejścia na rynek.



Ilustracja 3.3. Schemat przejścia do architektury mikrousługowej

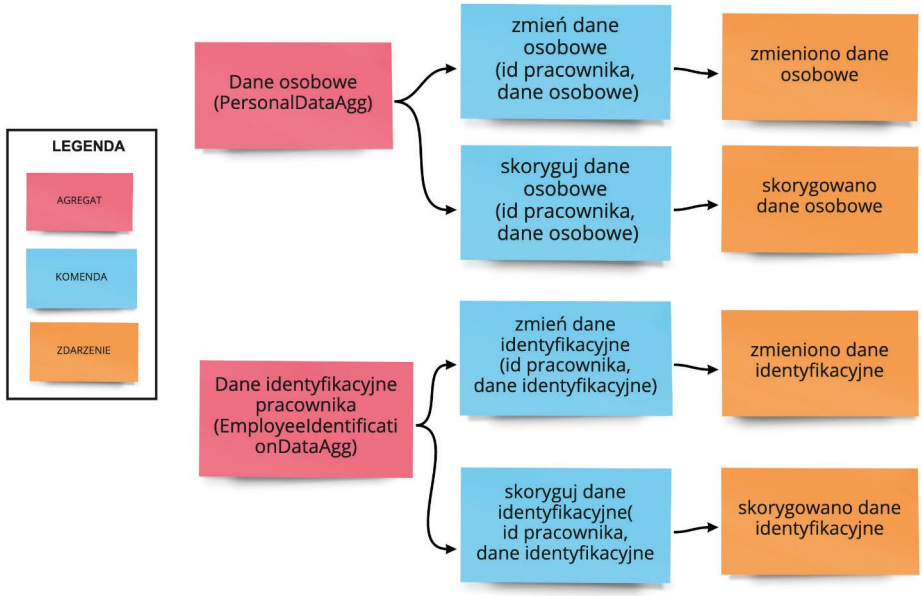
W procesie budowy nowej wersji systemu wykorzystane zostało podejście Domain Driven Design [9], bazujące na następujących założeniach:

1. logika projektu powinna zostać opisana w modelu domenowym;
2. projektowanie systemu powinno być nastawione na współpracę między zespołem technicznym a ekspertami domenowymi, czyli osobami po stronie klienta;
3. wszędobylski język (ang. *unambiguous language*) – baza danych nie jest modelem opisu rzeczywistości. Do prawidłowego opisu używany jest język pochodzący z badanego obszaru[20], który jest zrozumiały dla twórców oprogramowania i implementowany w kodzie, żeby wyeliminować dualizm pojęć i problemy związane z przekładaniem logiki biznesowej na algorytmy;
4. podział logiki na aplikacyjną (separuje warstwę usług związanych z realizacją wymagań funkcjonalnych od elementów technicznych) oraz domenową (model domeny);

[20] Używany przez ekspertów domenowych

5. budulec (ang. *building blocks*) – język standardowych wzorców, z których budujemy modele.

Do zalet DDD należy zaliczyć technikę analizy wymagań za pomocą Event Stormingu[21]. Ta wymyślona przez Alberto Brandoliniego metoda była stosowana za każdym razem gdy zespół zaczynał pracę nad nowym modułem i to z jej pomocą określano najlepszą architekturę dla danego modułu. W porównaniu do np. UML stosowana notacja jest bardzo prosta i intuicyjna, co przyspieszyło przekazywanie programistom wiedzy zgromadzonej przez analityka. Jako że stosowanie Event Stormingu wymusza zaangażowanie wszystkich uczestników, każdy w sposób aktywny zdobywał wiedzę o projektowanym module. Dyskusja nad możliwymi wariantami rozwiązania, już na początkowym etapie, pozwalała na odnalezienie nieścisłości w wymaganiach czy planowanym procesie biznesowym.



Ilustracja 3.4. Przykładowa tablica z sesji Event Stormingu

Należy jednak zwrócić uwagę, szczególnie w niedoświadczonym zespole, że sesje Event Stormingu potrafiły trwać bardzo długo, zbaczając na techniczne szczegóły implementacji, co nie powinno mieć miejsca. Zbyt długie spotkania powodowały, że część uczestników przestawała brać w nich aktywny udział. Dlatego ważne jest aby, szczególnie w początkowym okresie, każda sesja miała swojego moderatora odpowiedzialnego za pilnowanie dyscypliny w tym zakresie.

[21] eventstorming.com (dostęp: 18.06.2021)

3.3. Zarządzanie przepływem pracy dla utrzymania oraz prac rozwojowych

W trakcie prac związanych z budową systemu POL-on ujawniał się coraz bardziej problem planowania i realizacji konkretnych wymagań. Każde oddanie do użytku większych funkcjonalności wpływało na zmniejszenie prędkości (ang. *velocity*) zespołów w następujących po nich iteracjach. W niektórych przypadkach zespoły nie dostarczały żadnych nowych funkcjonalności w ramach sprintu, ponieważ skupione były coraz bardziej na poprawie błędów, pracach stabilizacyjnych oraz monitoringu aplikacji intensywnie eksploatowanej przez użytkowników końcowych. Odwołując się do opisanej w poprzednim podrozdziale taksonomii 3.1 jako prace związane z utrzymaniem systemu uznawano kategorie zgłoszeń typu błąd (ang. *bug*) lub zadanie (ang. *task*) nie wpływające na realizację nowych wymagań funkcjonalnych (ang. *user stories*). Z czasem jako kategorię zgłoszeń, stanowiących element profilowania kodu aplikacji do zgłoszonych przez użytkowników uwag w trybie szybkich modyfikacji, określono kategorię „ulepszenie" (ang. *improvement*). Tak zdefiniowany zakres zadań odpowiadał definicji prac rozumianych jako utrzymanie (ang. *maintenance*) w szeroko rozumianej teorii inżynierii oprogramowania[22].

Analiza tego zjawiska ujawniła problemy w interpretacji podejścia scrum, ponieważ skupiało się ono w większym stopniu na kwestiach związanych z rozwojem, aniżeli utrzymaniem oprogramowania. Można powiedzieć, iż metodyki zwinne zyskały sławę dzięki temu, że skupiały się na produkcji wysokiej jakości oprogramowania, ale nie poruszały w równym stopniu kwestii związanych z jego konserwacją. Zgodnie z [35] konserwacja systemu stanowi prawie połowę całkowitego wysiłku włożonego w dowolny system oprogramowania w okresie jego życia. Kwestia ta stanowi oczywiście główny aspekt problematyki długu technicznego•, czyli jednego z kluczowych pojęć w podejściu scrum do zarządzania jakością oprogramowania. Warto przypomnieć w tym momencie główne elementy decydujące o jego powstawaniu[23]:

- presja na wydanie oprogramowania w zakładanym terminie pomimo braków funkcjonalnych;
- zmiany wymagań zbyt późno, w efekcie nie ma czasu ani budżetu, by należycie wprowadzić je do projektu;
- nieprecyzyjne lub ewoluujące wymagania w trakcie trwania projektu;
- zbyt sztywne, hermetyczne rozwiązania uniemożliwiające łatwe dostosowanie do dodatkowych potrzeb (brak modularności zbyt wiele wzajemnych zależności);
- skupienie na szybkim wypracowywaniu zysków, zamiast odroczyć przychody w początkowych fazach w imię możliwych korzyści osiągniętych później;
- brak dostosowania do standardów i dobrych praktyk w początkowych fazach, dostosowywanie do standardów w późniejszych fazach może być znacznie utrudnione;

[22] utrzymanie oprogramowania (ang. *software maintenance*) to zmiana programu komputerowego po jego dostarczeniu, wdrożeniu i uruchomieniu w środowisku klienta mająca na celu tylko i wyłącznie poprawę błędów oraz dostosowanie oprogramowania do zmieniającego się środowiska teleinformatycznego
[23] mansfeld.pl/programowanie/dlug-techniczny-co-to-jest-przyklady (dostęp: 18.06.2021)

- brak refaktoryzacji w początkowych fazach (późniejsza refaktoryzacja będzie bardziej problematyczna).

Ponieważ terminy realizacji projektu były bardzo często nienegocjowalne z uwagi na zapisanie ich w prawie[24] prowadziło to do istotnych konfliktów w gronie zespołów wytwórczych, wynikających z poczucia, że wdrożenie produktu jest częstokroć przedwczesne i uwarunkowane zewnętrznymi czynnikami. Przenosiło się to na efekty związane z szacowaniem zakresu kolejnych wymagań, gdzie dominować zaczął subiektywny aspekt poczucia przeciążenia pracą związaną z poprawą błędów z poprzednich iteracji oraz tzw. działaniem na skróty. Programiści estymując poziom złożoności bardzo często uwzględniali ten element jako swoisty bufor, zabezpieczający czas na prace niezaplanowane, w konsekwencji obniżając przepustowość całego zespołu. Zamiast odnosić się w sposób zobiektywizowany do oceny złożoności danego wymagania, antycypowali również problemy związane z ewentualną poprawą błędów w przyszłości oraz czas potrzebny na przełączanie kontekstu pomiędzy różnymi kategoriami zadań w trakcie iteracji. Działanie takie było uzasadnione i stanowiło swoisty mechanizm obronny i regulujący rzeczywisty potencjał zespołu[25].

Ta specyfika wymuszała niejako zaciąganie długu technologicznego na poczet realizacji bieżących zobowiązań. W tym zakresie scrum jako metodyka pozostawiał dużą dowolność i pole do interpretacji.

Jako formę rozwiązania tego problemu przygotowano, w formie hipotezy, trzy potencjalne rozwiązania:

- estymowanie zadań związanych z utrzymaniem i prowadzenie dla nich odrębnego rejestru produktu – zarządzanie rejestrem produktu w sposób analogiczny do prac rozwojowych;
- podział kompetencyjny w ramach zespołów wytwórczych – wytypowanie grupy osób lub dedykowanego zespołu, który zajmowałby się tylko rozwiązaniem problemów związanych z konserwacją kodu, poprawą błędów oraz szeroko rozumianym utrzymaniem;
- manipulowanie przepustowością sprintów oraz lepszy podział pracy – retrospektywna analiza stanu zgłoszeń z kategorii utrzymania z ostatnich sprintów oraz podjęcie decyzji na etapie planowania o zmniejszeniu przepustowości zespołu na prace rozwojowe (ang. *velocity*), połączone z jasnymi regułami decydowania o przyjmowaniu tego typu zgłoszeń przez zespół w trakcie iteracji.

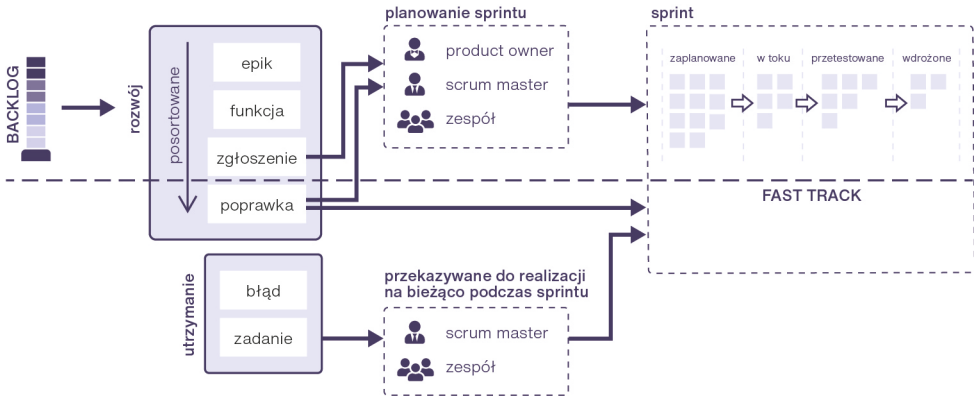
Pierwsze podejście zostało szybko odrzucone na gruncie koncepcyjnym jako niemożliwe do praktycznej realizacji. Prace związane z poprawą aplikacji są z założenia nieplanowane a mieszanie ich z rejestrem produktu w formie *user story* powodowałoby

[24] w praktyce trudno wyobrazić sobie bardziej ograniczające dla procesu wytwarzania oprogramowania okoliczności
[25] presja czasu powodowała w praktyce wydłużenie pracy nad złożonymi elementami funkcjonalnymi

praktyczną niemożliwość określenia rzeczywistego stanu prac nad oprogramowaniem z perspektywy klienta. Skutkowałoby to też podejściem zorientowanym bardziej na minimalizację strat, aniżeli rozwój i dążenie do skuteczności[26].

Drugie podejście zostało szybko odrzucone w związku ze wzrostem poczucia niezadowolenia członków zespołu (poprawa błędów odbierana była jako forma stygmatyzacji kompetencji pracownika w stronę realizacji zadań mało wymagających i kreatywnych obarczonych ogromną presją czasu) oraz brakiem efektywności, wynikającym z faktu, że osoba poprawiająca daną funkcjonalność rzadko kiedy odpowiadała za jej wytworzenie[27].

Wybrano trzecie podejście, które zobrazowane jest na Ilustracji 3.5.



Ilustracja 3.5. Schemat kategoryzacji pracy w ramach iteracji

Tak jak wspomniano wcześniej, to zespół wytwórczy decydował o przepustowości prac rozwojowych w danym sprincie. Nie było to uzależnione od faktycznych danych rejestrowanych w systemie zarządzania zmianą i ekstrapolowanych na kolejne iteracje. Innymi słowy, to ile punktów historyjek użytkownika[28] był w stanie przyjąć zespół do realizacji w danej iteracji stanowiło jego zobowiązanie. Dla zgłoszeń związanych z utrzymaniem przewidziana była szybka ścieżka (ang. *fast track*), czyli realizowane były one w pierwszej kolejności na podstawie statusów i priorytetów. Dopóki nie wpływało to na ryzyko dostarczenia nowych funkcjonalności w ramach iteracji, proces odbywał się w sposób automatyczny. Jednakże w chwili, gdy zespół i pomagający mu w realizacji zadań mistrz młyna (ang. *scrum master*) stwierdzał ryzyko danej nieplanowanej czynności na końcowy efekt prac, rozpoczynały się negocjacje z właścicielem produktu[29]

[26] w podejściu do zarządzania definiuje się to jako kultura porażki
[27] musiała dużo czasu poświęcić na zrozumienie jej sposobu działania, bardzo często angażując w ten proces faktycznego twórcę
[28] skwantyfikowana miara określająca stopień złożoności danych jednostek produktowych
[29] który bardzo często musiał kontaktować się w tej sprawie również z klientem

oraz kierownikiem projektu w celu podjęcia strategicznej decyzji o usunięciu określonych wymagań z zakresu zadań przewidzianych na iterację, by stworzyć warunki do realizacji tego typu zadań.

Dzięki przyjęciu tego typu rozwiązań udało się z czasem w sposób bardziej efektywny i zaplanowany zarządzać kategoriami poszczególnych prac, umiejętnie żonglując nimi w ramach iteracji. Do rozwiązania również tego problemu w sukurs przyszła częściowo zmiana technologiczna, która uprościła i skróciła proces realizacji zadań typu *bug fixing*, o czym opowiada szerzej rozdział 4.3. W praktyce podejście typu *fast track* stanowiło też integrację elementów procesu ciągłego typu kanban z iteracjami. Uznanie jednakże, iż była to faktyczna implementacja tego podejścia jest nieuprawnione i stanowi zbyt duże uproszczenie, gdyż faktycznie poza wizualizacją i realizacją procesu w formie wyciągania do realizacji wymagań z rejestru (ang. *pull*), nie posiada ono jego pozostałych cech.

3.4. CI/CD – punkt wyjścia i ewolucja w toku rozwoju systemu

Pracując jako programiści ludzie stają się dobrze opłacanymi tłumaczami. Tłumaczą wymagania biznesowe na kod maszynowy, który komputer zrozumie i poprawnie wykona. Owocem takiej pracy jest kod źródłowy. Posiadając tłumaczenie wymagań użytkownika na język zrozumiały dla maszyny, stajemy przed wyzwaniem, jak wdrożyć owoc takiej pracy na serwer produkcyjny. Możemy uznać, że wystarczy skopiować skompilowany kod w serwer produkcyjny, ale:

- jak sprawdzić, by świeżo napisany kawałek kodu działał jak miał działać?
- jak sprawdzić, by świeżo napisany kawałek kodu nie popsuł niczego, co do tej pory działało?
- jak sprawdzić, by świeżo napisany kawałek kodu nie spowodował krytycznego problemu w środowisku produkcyjnym?
- jak wdrożyć w środowisku produkcyjnym aplikację ze świeżo napisanym kawałkiem kodu?
- jak wycofać ze środowiska produkcyjnego aplikację, która działa niepoprawnie?
- co zrobić, gdy aplikacja używa bazy danych i zmiana naszego kodu powodowała konieczność zmiany bazy danych?

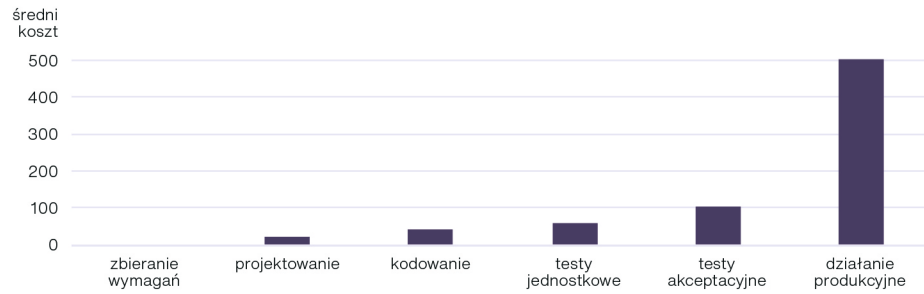
Rozdział ten ma za zadanie przybliżyć problemy przedstawione powyżej oraz przedstawić rozwiązania wypracowane podczas pracy nad systemami wchodzącymi w skład ekosystemu POL-on.

3.4.1. CI-CD

CI z angielskiego oznacza *continuous integration*, czyli ciągłą integrację. W systemach informatycznych jest to proces polegający na łączeniu własnego kodu ze zdal-

nym repozytorium tak, jak szybko jest to tylko możliwe, nawet kilka razy dziennie. Fakt połączenia kodu powinien wyzwolić proces budowania aplikacji, tak aby możliwie w jak najkrótszym czasie autor zmiany dostał informację, że kod się nie kompiluje. W ekosystemie POL-on proces ten nie tylko wyzwala budowanie, ale też i uruchamia testy, oraz przeprowadza statyczną analizę kodu w narzędziu Sonar[30]. Dzięki takiemu podejściu kod zawsze (w rzeczywistości prawie zawsze) jest gotowy do wdrożenia.

CD z angielskiego *to continuous delivery* lub też *continuous deployment*. Pierwsza definicja oznacza proces wdrożenia potencjalnie każdej wersji aplikacji, która poprawnie się zbuduje i przejdzie przez testy. Dlaczego w poprzednim zdaniu zostało użyte słowo „potencjalnie"? Bo oczywiście nie każdą funkcjonalność chcemy/potrzebujemy od razu wdrożyć – na przykład gdy funkcjonalność jest rozbita na kilka mniejszych i pod względem biznesowym nie ma sensu wdrażać małych fragmentów. Natomiast druga definicja CD *continuous deployment* – oznacza wdrażanie absolutnie każdej poprawnie zbudowanej wersji na środowisko produkcyjne. Jedynym powodem, który może przerwać proces wdrożenia są testy dające wynik negatywny. Ktoś może tu zadać pytanie „Ale po co tak robić?" Dopiero odpowiedź na to pytanie pokazuje dlaczego warto. Autorzy [27] wskazują, co również wynika z naszego doświadczenia (Ilustracja 3.6), zgodnie z którym błąd w kodzie znaleziony szybko, można szybko i tanio poprawić. Im później błąd zostanie wykryty, tym większy będzie koszt jego naprawy. Koszt taki trzeba ponieść, nawet przy założeniu posiadania systemu optymalizującego przypisanie programisty do rozwiązania błędu, jak proponują w [30] (u nas zadania są przypisywane do zespołu, który realizował daną funkcjonalność).



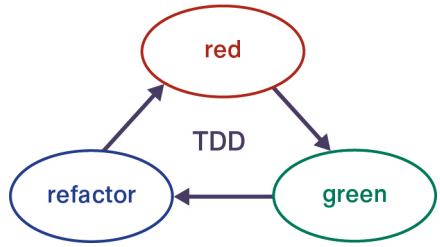
Ilustracja 3.6. Relatywny koszt usunięcia błędu
Źródło: opracowanie własne za [27]

W kontekście rozwoju aplikacji autorzy książek [20] oraz [10] zwracają uwagę na fakt, że wprowadzenie CI/CD potrafi przynieść wymierne korzyści organizacji. Wprowadzanie podejścia CI/CD to coś więcej niż tylko usuwanie granicy między rozwojem aplikacji a jej wdrażaniem i utrzymywaniem. To jest zmiana mentalna dla całej organizacji polegająca na wprowadzeniu kultury przekazywania informacji zwrotnej.

[30] sonarqube.org (dostęp: 18.06.2021)

3.4.2. Testy

W trakcie rozwoju aplikacji z ekosystemu POL-on ustalono, że aplikacje powinny powstawać zgodnie z TDD ([22]), gdyż zapewnia to powstawanie testów równolegle z kodem produkcyjnym.



Ilustracja 3.7. Schemat TDD

Wprowadzanie zmiany w kodzie rozpoczynamy od napisania testu, który musi zwrócić wynik negatywny (stan *Red* na Ilustracji 3.7), w kolejnym kroku tworzymy kod produkcyjny. W takim podejściu powstaje tylko tyle kodu, żeby test zakończył się wynikiem pozytywnym (stan *Green* na Ilustracji 3.7). W kolejnym kroku porządkujemy kod (stan *Refactor* na Ilustracji 3.7). Proces taki powtarzamy do czasu, aż kod produkcyjny spełnia wymagania testu, test w pełni opisuje nasze wymagania oraz nie ma nic do uporządkowania w kodzie. W toku rozwoju aplikacji wyróżniono 2 wyjątki od stosowania TDD.

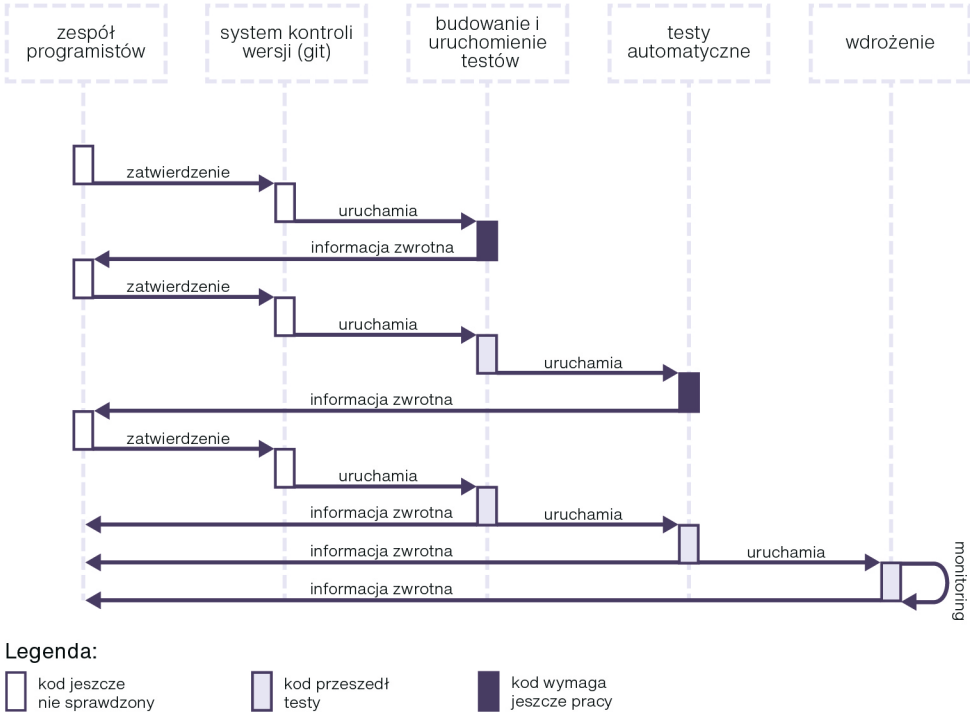
- pracujemy nad prototypem rozwiązania;
- projekt z założenia nie będzie w produkcji dłużej niż 3 miesiące, np. system do głosowania, który z założenia działał do konkretnej daty, po której prezentował tylko statyczne dane a w końcu został wyłączony. (System ten posiadał testy, ale nie w sensie TDD.)

3.4.3. Uruchomienie CI

Odpowiedzialny za ten proces jest system CI (u nas jest to Jenkins[31]). Jedynym sposobem na uruchomienie procesu weryfikacji poprawności budowania aplikacji i przeprowadzania testów jest sposób automatyczny, wyzwalany zaistnieniem zamiany w kodzie. Testowano także podejście ręczne oraz podejście oparte na harmonogramie, czyli uruchamianiu budowania aplikacji codziennie o zadanej godzinie. Ręczne uruchamianie posiada minus tego, że trzeba je uruchomić, o czym w toku prac można zapomnieć, rozwiązanie oparte na harmonogramie działało przez jakiś czas. Tym niemniej zespoły utrzymują obecnie 80 projektów, a zmiany zachodzą w kilku. Utrzymanie wdrożeń opartych

[31] jenkins.io (dostęp: 18.06.2021)

na harmonogramie niepotrzebnie zużywało zasoby na budowanie projektów niezmienianych. Uzupełnieniem procesu automatycznego jest możliwość uruchomienia procesu budowania aplikacji na żądanie. Celem procesu jest osiągnięcie stanu przedstawionego na Ilustracji 3.8.



Ilustracja 3.8. Schemat procesu budowania aplikacji

Posiadanie konfiguracji sposobu budowania aplikacji (przykładowy jenkinsfile przedstawia kod 3.1) tylko w narzędziu Jenkins, okazało się niewystarczające. W toku rozwoju aplikacji koniecznym stało się trzymanie opisu sposobu budowania aplikacji w systemie kontroli wersji. Zastosowanie takiego podejścia umożliwiło opracowanie własnej biblioteki do obsługi procesu wdrożeniowego oraz traktowanie infrastruktury jako serwisu infrastruktura jako usługa• zgodnie z [3] oraz [1]. Opracowana biblioteka oferuje funkcjonalności budowania, testowania i analizy statycznej aplikacji oraz integrację z Jirą (zmiana wersji aplikacji podczas wdrożeń), integrację z komunikatorem (informuje o sukcesie bądź porażce wdrożenia). Aplikacja jest budowana obecnie w jako obraz Docker•, zatem posiadanie jednego miejsca konfiguracji umożliwia także szybką podmianę takiego obrazu aplikacji. Przypadki, które mogą powodować potrzebę podmiany tego obrazu to odkrycie krytycznego błędu w obrazie bazowym lub opracowanie obrazu, który będzie potrzebował mniej zasobów np. zamiast 800 MB dysku, tylko 250 MB. Dodatkowym plusem posiadania własnej biblioteki wdrożeniowej jest możliwość zdefiniowania inicjalnego zadania zasilenia Jenkinsa wszystkimi zadaniami wdrożeniowymi. Ma to umożliwić

w krótkim czasie odtworzenie funkcjonalności Jenkinsa np. w przypadku zmiany wersji Jenkinsa lub możliwej awarii tego systemu. Obecnie posiadamy ponad 80 projektów, a każdy z nich powinien poprawnie wdrożyć się, po takiej operacji. W przypadku braku automatycznego przywrócenia wszystkich zadań wdrożeniowych, istnieje uzasadnione przypuszczenie, że jakieś projekty mogłyby zostać pominięte lub nieuwzględnione w podejściu ręcznym. Co więcej, ręczne odtworzenie wymagałoby więcej czasu niż odtworzenie automatyczne.

Kod 3.1. Przykładowy plik opisujący wdrożenie

```
node {
  stage('pobierz projekt z gita') {
    sh 'git clone https://gitserver/projekt.git'
  }

  stage('zbuduj projekt') {
    sh 'mvn package'
  }

  stage('uruchom testy automatyczne') {
    sh 'runAutomaticTests.sh'
  }

  stage('deploy apliacji na server') {
    sh 'deploy.sh'
  }
}
```

Każde zatwierdzenie do repozytorium kodu wyzwala[32] budowanie, testowanie, analizę statyczną kodu oraz wdrożenie na serwer developerski. W przypadku, gdy projekt się nie buduje lub testy nie przechodzą pozytywnie, osoby, które zatwierdzały zmiany do repozytorium od czasu ostatniego pozytywnego uruchomienia, dostają powiadomienie o tym fakcie na komunikatorze.

3.4.4. Przegląd kodu

Przegląd kodu (z ang. *code review*) – to technika wspomagająca w zespołach programistów, polegająca na tym, iż każdy z programistów (ewentualnie podgrupa zespołu) przegląda zmiany wprowadzane do repozytorium przez innych. Dzięki zaangażowaniu członków zespołu w przeglądanie zmian w kodzie, zespół staje się bardziej świadomy co do zmian w projekcie. Po wprowadzeniu przeglądania kodu[33] okazało się, że każdy z programistów ma pomysł jak przepisać kod, poprawić jego czytelność albo wprowadzić optymalizacje. Kod w ten sposób staje się bardziej własnością całego zespołu niż pojedynczego developera. Zespół staje się bardziej odpowiedzialny za projekt, bo ma większy wpływ na zmiany w nim zachodzące. Kolejną cechą, o której warto wspomnieć

[32] ze względów wydajnościowych proces ten uruchamia się w ciągu 15 minut od zatwierdzenia
[33] używamy GitLaba, który jest zainstalowany na naszych serwerach

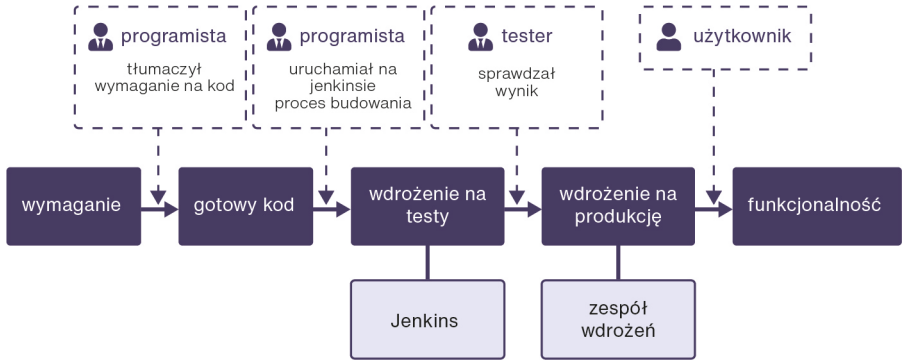
jest wyrównanie poziomu wiedzy w zespole. Każdy z członków zespołu posiada własny styl pisania kodu, większe bądź mniejsze umiejętności w sposobie podziału kodu na komponenty, ale dzięki przeglądaniu kodu lepsi i bardziej zaawansowani pokazują mniej doświadczonym jak programować inaczej/lepiej, bardziej wydajnie, albo przygotować kod do możliwego rozwoju w przyszłości. Technika ta jest stosowana tylko w niektórych zespołach programistów. Zespoły, które zdecydowały się na wprowadzenie przeglądania kodu, zauważyły poprawę jakości kodu. W odniesieniu do zespołów, które nie zdecydowały się na przeglądanie kodu pojawiały się poniższe zastrzeżenia:

- brak potrzeby przeglądania, bo zespół jest zgrany i nie doszło jeszcze w nim do dużej rotacji pracowników;
- obawa przed stratą czasu na przeprowadzanie takiego procesu.

Wydaje się, że w zespołach długo pracujących ze sobą nie ma sensu wprowadzać takiego mechanizmu, tym niemniej przepływ wiedzy na temat rozwiązań stosowanych przez poszczególnych członków zespołu może być zaburzony, zwłaszcza w przypadku pracy zdalnej. Stwierdzenia, że przeglądanie kodu pochłonie dużo czasu to tylko dowód niechęci do zmian procesu wytwarzania oprogramowania. Z naszych doświadczeń wynika, że średni czas przeprowadzania tego procesu przy 5 programistach to 10-15 minut co 2-3 dni.

3.4.5. Czy to potrzebne?

Od początków pracy z ekosystemem POL-on posiadaliśmy CI (Jenkinsa oraz Sonara). Przepływ pracy (Ilustracja 3.9) miał wiele punktów, w których ktoś musiał coś wykonać ręcznie. W związku z tym implementacja wymagania biznesowego potrafiła zająć nawet 50% czasu zanim zmiana taka trafiła na produkcję. Drugie 50% zajmowało wdrożenie i przetestowanie. W przypadku utrzymywania tylko jednego projektu ludzie są w stanie przyzwyczaić się do takiego podejścia i osiągać całkiem rozsądne wyniki. W przypadku posiadania większej ilości projektów, automatyzacja jest jedynym wyjściem. Sam system POL-on osiągnął ponad milion linii kodu, i wdrażanie takiego monolitu zaczęło dostarczać nam dużo wyzwań. Najpoważniejszym problemem była współpraca zespołów programistów, gdyż zespoły zaczęły się blokować wdrożeniami. Jeden nie mógł wdrożyć swojej gotowej funkcjonalności, bo wymagała integracji z inną funkcjonalnością, jeszcze niegotową, z innego zespołu. Zaczęto wprowadzać wymóg pracy na gałęziach, które obejmowały swoim zasięgiem daną funkcjonalność. Czas życia takich gałęzi już podczas ich powstawania był szacowany na tygodnie lub miesiące.



Ilustracja 3.9. Schemat przepływu pracy

Takie rozwiązanie też się u nas nie sprawdzało, bo po połączeniu dużej funkcjonalności testy całego systemu trwały za długo i przeszkadzały we wdrożeniach innych zespołów lub je kompletnie blokowały. W związku z tym dokonano podziału monolitu na mikroserwisy. Decyzja ta miała wpływ na wszystko inne, jako że w pierwszej fazie przechodzenia na podejście mikroserwisowe pojawił się Docker[34], a w kolejnej fazie Kubernetes[35]. Obecnie proces wdrożenia na serwer testowy jest w pełni automatyczny i wyzwalany zmianą kodu źródłowego. Wdrożenie automatyczne do produkcji obecnie jest możliwe, ale wyzwolenie takiego wdrożenia pozostaje w gestii zespołów testowych.

Przejście na pełną automatyzację wdrożeń jest zmianą procesu myślenia całej organizacji. Z punktu widzenia programisty zawsze należy pamiętać o tym, że dopiero co napisany kawałek kodu po zatwierdzeniu do repozytorium zaraz trafi na serwer (być może i produkcyjny). W automatyzacji procesu wdrożenia należy uważać, bo jak wskazuje [39] podczas niego możemy ujawnić dane, których byśmy nie chcieli ujawniać. Podczas analizy kodu publikowanego na GitHubie autor wskazał, że podejście eliminujące człowieka może przyczynić się do ujawnienia haseł z plików konfiguracyjnych, skryptów wdrożeniowych i/lub kodu. Dodatkowo należy zawsze być przygotowanym na przywrócenie kodu w środowisku produkcyjnym sprzed wdrożenia. Bo pomimo usilnych starań i przejściu pozytywnie wszystkich testów aplikacja może okazać się niewydajna. Przywrócenie poprzedniej wersji aplikacji może nastąpić przez mechanizmy Kubernetesa. Do czasu pisania aplikacji ta możliwość nie była wykorzystana. Nasze doświadczenia wskazują, że podczas wykrycia błędu w produkcji kod jest poprawiany i nowa wersja jest wdrażana na serwer produkcyjny.

Zmiana modelu bazy danych w podejściu automatycznym powoduje zmianę podejścia do wprowadzania takich zmian. Używanie systemu kontroli wersji do zarządzania zmianami w bazie np. flyway[36] nie uchroni nas przed potencjalnymi problemami. Problem

[34] docker.com (dostęp: 18.06.2021)
[35] kubernetes.io (dostęp: 18.06.2021)
[36] flywaydb.org (dostęp: 18.06.2021)

z bazą danych pojawia się, w sytuacji gdy usuwamy kolumnę z tabeli, albo zmieniamy jej nazwę. Jeżeli taką zmianę wprowadzimy za szybko i bez uwzględnienia faktu, że obecna wersja aplikacji musi poprawnie współpracować z tak zmienioną bazą, to nie będziemy w stanie wycofać potencjalnie niepoprawnej nowej wersji aplikacji z produkcji. Poprzednia wersja aplikacji po prostu nie będzie w stanie używać tak zmienionej bazy. Rozwiązaniem jest przeprowadzanie zmian struktury bazy w kilku krokach, tak żeby aplikacja w kroku *n* była kompatybilna z bazą danych z kroku *n+1*. Wymaga to przewidzenia, że zmiana w produkcji nie pojawi się wraz z pojedynczym wdrożeniem aplikacji. Zmiana struktury bazy danych będzie osiągnięta w kilku wdrożeniach, które muszą być rozciągnięte w czasie.

3.5. Model wprowadzania zmian w kodzie. Ewolucja czy rewolucja?

Prace nad systemem POL-on rozpoczęły się na przełomie 2010 i 2011 roku. Zespół projektowy ustalił, że repozytorium kodu projektu będzie znajdowało się w systemie kontroli Git, poprzednie projekty były w SVN. Posiadając doświadczenie w łączeniu zmian w SVN, podczas których rozwiązywanie konfliktów trwało po klika dni, zdecydowano się na użycie systemu Git. Podczas analizy dostępnych rozwiązań Git wydawał się najbardziej przyszłościowy, a z racji zmiany sposobu trzymania zmian w stosunku do SVN, łączenie zmian miało zajmować o wiele mniej czasu niż w SVN. Zmiana taka, jak na czas, w którym była podejmowana, i doświadczenie zespołu była obarczona dużym ryzykiem, jako że tylko nieliczni z zespołu eksperymentowali z Git-em. Decyzja ta z obecnego punktu widzenia wydaje się oczywista i jedyna właściwa, tym niemniej w czasie jej podejmowania zakładano możliwość powrotu do SVN. Zespół projektowy poszedł o krok dalej niż tylko wybór samego systemu kontroli wersji kodu źródłowego. Po rozpoznaniu sposobu zarządzania zmianami w systemie Git ustalono również, że będziemy używali nakładki na Git, GitFlow.

Rozdział ten opisuje, jak działa i co nam dał GitFlow, a co zabrał lub wręcz uniemożliwił. Przedstawia również inne podejście do wprowadzania zmian w kodzie, które umożliwia łatwiejszą integrację z CI/CD oraz eliminuje potrzebę łączenia gałęzi kodu.

3.5.1. GitFlow

GitFlow [7] jest najbardziej znanym i najstarszym modelem zarządzania zmianami w kodzie opartym na systemie Git. Był stworzony przez Vincenta Driessena w roku 2010 i jego koncepcja opiera się na dwóch głównych gałęziach: master i develop. Obie gałęzie żyją przez cały czas życia projektu. Gałąź master zawiera zawsze aktualny kod produkcyjny. Funkcjonalności, które są rozwijane (z gałęzi develop), są dołączane do gałęzi master wraz z ich realizacją. Gałąź develop zawiera kod przedprodukcyjny. Kiedy funkcjonalności są kończone, dołącza się je do gałęzi develop. Model ten zakłada istnienie także innych gałęzi (organizujących i porządkujących):

- feature-* gałęzie te są używane do rozwijania nowych funkcjonalności. Rodzicem feature'a zazwyczaj jest develop i do developa jest taka gałąź dołączana;
- hotfix-* gałąź ta jest wymagana, żeby móc natychmiast zareagować na błąd istniejący na gałęzi master. Rodzicem hotfixa jest master, po zakończeniu hotfixa jest on dołączany do gałęzi develop i master;
- release-* gałęzie te wspierają przygotowanie nowej wersji aplikacji przed wdrożeniem produkcyjnym. Rodzicem tej gałęzi jest develop, po zakończeniu prac na gałęzi release jest ona dołączana do gałęzi develop i master.

Podczas rozwijania ekosystemu POL-on nowo tworzony kod był dołączany do gałęzi develop. W odstępach dwutygodniowych z gałęzi develop była tworzona gałąź release, na której następowała stabilizacja kodu oraz przeprowadzanie testów integracyjnych. W związku z faktem, iż czasami pojawiały się problemy na produkcji, z gałęzi master powstawały gałęzie hotfix. Z powyższego opisu wynika, że nawet najprostsza zmiana, która miała być wprowadzona, potrzebowała około 2,5 tygodnia, żeby pojawić się w produkcji. Dodatkowo pojawienie się gałęzi release skutkowało potrzebą integracji tej gałęzi do gałęzi master. Proces wdrożenia był przeciągany przez ponowne testy. Kierowano się przesłanką, że dowolna zmiana kodu na gałęzi master uruchamia ponowne testy. Wraz ze wzrostem ilości zespołów programistycznych zaangażowanych w rozwój ekosystemu POL-on problem przeciągających się przygotowań do fazy wdrożenia pogłębiał się. Zespoły blokowały się wzajemnie, jeden był już gotowy, drugi jeszcze nie. Aplikacja była jedna, zatem do wdrożenia dochodziło, gdy wszystkie zespoły przygotowały się do wdrożenia, a połączony kod wszystkich zespołów przechodził testy. W związku z eliminacją ponownych testów w niektórych projektach pomijano tworzenie gałęzi release i od razu kod z gałęzi develop był dołączany do gałęzi master. To sprawiało, że z uwagi na czas potrzebny na stabilizację kodu z gałęzi master, nie było kodu gotowego do wdrożenia do produkcji. Oznaczało to, że gdyby pojawił się krytyczny błąd w produkcji, w trakcie stabilizacji kodu z gałęzi master, to nie byłoby jak go poprawić i wdrożyć na produkcję. W takiej sytuacji trzeba by było poczekać do czasu, aż kod z gałęzi master przejdzie poprawnie testy. Dopiero po takim procesie możliwe byłoby wprowadzenie poprawki naprawiającej błąd krytyczny z produkcji.

Rozwiązanie przyjęte prawie 10 lat temu wciąż rzutuje na pracę zespołów. Przez ten czas wypracowano narzędzia wspomagające prace i przystosowano cykl pracy zespołów programistów, testerów oraz projektantów. Pomimo zauważalnego problemu w GitFlow – przeciągające się wdrożenia na skutek braku zaufania do wyniku procesu łączenia kodu źródłowego z różnych gałęzi – trudno z takiego podejścia zrezygnować właśnie z powodu poniesionych już kosztów (to jest znane jako tzw. kosztów utopionych) w związku z ewolucją tego rozwiązania.

3.5.2. Celowy brak łączenia kodu z różnych gałęzi

W jednym z zespołów programistów zauważono, iż problemem nie jest posiadanie dużej ilości gałęzi z kodem, ale sam fakt łączenia i przenoszenia kodu między gałęziami (zwłaszcza podczas zmian funkcjonalnych w już istniejącym kodzie). Dopóki do

kodu źródłowego dodajemy nowe funkcjonalności, nie jest to problem, natomiast jakiekolwiek zmiany w kodzie już istniejącym to wyzwanie. Każde łączenie kodu między gałęziami może prowadzić do poważnych problemów (choć nie musi). Zaproponowano podejście polegające na posiadaniu tylko trzech gałęzi głównych: dev, test i prod. Każda z nich odpowiada za środowisko, w którym będzie wdrażany kod z tej gałęzi. Celem takiego podejścia jest potrzeba całkowitej eliminacji łączenia kodu między gałęziami. W proponowanym rozwiązaniu dochodzi do całkowitego zastępowania kodu na gałęzi przez kod z innej gałęzi. Można to porównać do dzieci, rodziców i dziadków, z czasem życia dziadkowie odchodzą, na ich miejsce przychodzą rodzice, których miejsce po jakimś czasie zajmują dzieci. W tym podejściu kod z gałęzi test zastępuje kod z gałęzi prod, natomiast kod z gałęzi dev zastępuje kod z gałęzi test. W przypadku pojawienia się krytycznego błędu w kodzie w produkcji, poprawka funkcjonalności jest realizowana na gałęzi prod. Kolejnym opcjonalnym krokiem jest przeniesienie poprawki z gałęzi prod na gałąź test i dev. Tym niemniej może się okazać, że w toku rozwoju aplikacji kod na gałęzi test/dev na tyle się zmienił, że już nie potrzebuje takiej zmiany. Należy zaznaczyć, że poprawka błędu krytycznego jest dołączana do kodu w odwrotnej kolejności niż nowo powstająca funkcjonalność. Rozwiązanie to ma podstawową zaletę: jest bardzo proste. Niemniej podczas rozwiązywania problemu z produkcji i tak musimy łączyć zmiany między gałęziami z kodem.

3.5.3. Przełącznik funkcji (*feature toggle*)

Problem eliminacji łączenia kodu z różnych gałęzi w procesie rozwoju oprogramowania zainteresował Martina Fowlera. W swoim blogu [14] przedstawił rozwiązanie, które nie posiada wyżej wymienionych wad. Zwrócił uwagę, iż największym problemem jest przenoszenie zmian między różnymi wersjami kodu aplikacji. Postawił śmiały postulat, aby posiadać tylko jedną gałąź z kodem i wszystkie zmiany wprowadzać tylko tam. Jako że nie ma innych gałęzi, nie ma potrzeby łączenia kodu. Skoro nie ma innych gałęzi z kodem, to jak separować kod już skończony do tego wymagającego jeszcze pracy? Wg Fowlera rozwiązaniem są „przełączniki funkcjonalności". Sprowadza się do wstawiania w kodzie warunku logicznego „if", który decyduje o przebiegu logiki aplikacji. Przykład przedstawiono we fragmencie 3.2, w którym warunek „if (newValidation)" jest opisanym przełącznikiem.

Kod 3.2. Pseudokod przedstawiający kod po wprowadzeniu przełącznika – wersja uproszczona

```
//przed wprowadzeniem ,,feature toggle''
passwordValidate(String password) {
  if (password.length < 4) {
    throw new ValidationExcepetion("Password too short")
  }
}

//po wprowadzeniu ,,feature toggle''
passwordValidate(String password) {
  if (newValidation) { // ,,feature toggle''
    if (password.length < 8) {
      throw new ValidationExcepetion("Password too short")
    }
```

```
  } else {
    if (password.length < 4) {
      throw new ValidationExcepetion("Password too short")
    }
  }
}
```

Oczywiście przykład pokazany w Kodzie 3.2 jest dość uproszczony. Podczas wprowadzania takiego przełącznika powinien wyglądać jak Kod 3.3. Na potrzeby zmiany wymagań powinien być powołany nowy komponent do realizacji wymaganej funkcjonalności. W związku z faktem, iż podczas rozwoju oprogramowania posiadamy wiele środowisk (dev, test, prod), w które wdrażamy przygotowany kod, to warunek taki może/powinien być parametryzowany również takim środowiskiem.

Kod 3.3. Pseudokod przedstawiający kod po poprawnym wprowadzeniu przełącznika – wersja docelowa

```
runAllValidation() {
  …
  if (nowaWalidacja) {
    newPasswordValidator.passwordValite(password)
  } else {
    oldPasswordValidator.passwordValite(password)
  }
  …
}
```

Dzięki takiemu podejściu w czasie działania aplikacji możemy decydować, którą ścieżkę aplikacja ma wybrać, przez odpowiednią konfigurację przełącznika, to nawet na środowisku produkcyjnym. Pojawia się pytanie: Dlaczego również w serwerze produkcyjnym chcemy sterować takim przełącznikiem? Bardzo rzadko, a właściwie nigdy, posiadamy dokładnie takie samo środowisko testowe, jak produkcyjne. Może to spowodować, że błąd polegający np. na znacznym spowolnieniu aplikacji i tak pojawi się dopiero w produkcji. Dlatego też warto monitorować zachowanie aplikacji po przestawieniu takiego przełącznika. W przypadku wykrycia ewentualnych błędów, defektów czy też znacznego spowolnienia aplikacji w prosty sposób można przywrócić poprzednie działanie aplikacji, a funkcjonalność poprawić. Oczywiście, jeżeli nowa funkcjonalność działa poprawienie, to kolejnym krokiem, o którym nie możemy zapomnieć, będzie usunięcie z kodu źródłowego starego przebiegu oraz usunięcie samego przełącznika Kod 3.4.

Kod 3.4. Pseudokod przedstawiający kod po usunięciu starego przebiegu aplikacji i przełącznika

```
runAllValidation() {
  …
  newPasswordValidator.passwordValite(password)
  …
}
```

Oczyszczenie kodu ze starych przełączników jest konieczne, jeśli nie chcemy skończyć z ogromną ich ilością. Przy okazji usuwamy martwy kod, którego aplikacja już nie używa. Podejście to nie ma problemów związanych z łączeniem kodu, ale wydłuża cykl implementacji danej funkcjonalności. Dodatkowo wymaga kompatybilności wstecz podczas wprowadzania zmian w modelu bazy danych. Model bazy danych musi umożliwiać wy-

cofanie obecnie wprowadzonej funkcjonalności. Podczas analizy podejścia przy użyciu przełączników funkcjonalności zauważono jeszcze jedną zaletę: premiuje programowanie obiektowe, a wręcz komponentowe. Jako iż ostatnim krokiem w tym podejściu jest usuwanie nieużywanego już przełącznika i starej logiki, zespoły programistów właściwie same starają się umieszczać funkcjonalności w dobrze wydzielonych komponentach.

3.5.4. Podsumowanie

Każde z prezentowanych rozwiązań ma cechy zarówno pozytywne, jak negatywne. Jeżeli pozostalibyśmy przy monolitycznej architekturze aplikacji, to podejście „przełączników funkcjonalności" zapewni możliwość przejścia z modelu wdrożeniowego co dwa tygodnie na model wdrażania funkcjonalności, jak tylko są gotowe. Tym niemniej w związku z potrzebą koordynowania prac różnych zespołów i tak musimy pozostać w dwutygodniowym cyklu pracy. Pozostanie przy GitFlow sprawi, że wdrożenie aplikacji będzie możliwe dopiero wtedy, gdy wszystkie funkcjonalności realizowane przez czas jednego sprintu będą gotowe. Z doświadczeń i obserwacji naszego procesu wynika, że w przypadku 3 zespołów programistów uczestniczących w rozwoju aplikacji potrzebowano średnio 4 dni na wdrożenie zakończonych funkcjonalności do produkcji. Na ten czas składały się: sam proces integracji funkcjonalności poszczególnych zespołów (pół dnia), wdrożenie aplikacji na serwer testowy wraz z uruchomieniem skryptów w bazie danych (kolejne pół dnia), uruchomienie i przeprowadzenie testów automatycznych wraz z poprawkami (dwa i pół dnia), wdrożenie na serwer produkcyjny wraz ze skryptami w bazie danych (pół dnia). Przejście między modelami nie będzie ani proste, ani łatwe, tym niemniej taka zmiana wydaje się nieunikniona i może przynieść zysk w postaci skrócenia czasu między zakończeniem implementacji a wdrożeniem do produkcji. Tym niemniej przez wprowadzenie 3.4 oraz podział aplikacji POL-on na mikrousługi, czas potrzebny na wdrożenie gotowej funkcjonalności skrócił się do około 2 godzin. Na czas ten składają się: wdrożenie aplikacji i uruchomienie testów automatycznych, pojedyncze testy manualne głównych funkcjonalności i samo wdrożenie na serwer produkcyjny.

3.6. Automatyzacja metod kontroli jakości

POL-on przez cały okres swojego istnienia rozwijany był w metodyce *scrum*, poprzez implementację historyjek użytkowników w trakcie dwutygodniowych sprintów, po których następowało wdrożenie systemu w środowisku produkcyjnym. Jako młody system, liczący wówczas około kilkunastu modułów, o stosunkowo niskim jeszcze poziomie komplikacji, POL-on testowany był przez zespół testów, który składał się z czterech osób z doświadczeniem w testach manualnych. W środowisku deweloperskim codziennie wdrażana była nowa wersja systemu. W ramach procedury wdrożeniowej automatycznie uruchamiane były w nim testy jednostkowe, tworzone przez członków zespołów

deweloperskich. W tak przygotowanym środowisku testerzy przeprowadzali manualne testy historyjek i retesty•[37] znalezionych defektów systemu.

Po zakończeniu sprintu, po zaimplementowaniu wszystkich historyjek oraz naprawie (i weryfikacji jej poprawności) defektów znalezionych w trakcie wcześniejszych testów w środowisku deweloperskim przygotowywana była wersja przedwdrożeniowa, która wdrażana była w odrębnym środowisku testów przedprodukcyjnych. Tam zespół testów dokonywał kontroli dwóch aspektów.

Pierwszy aspekt to poprawność i kompletność przygotowania nowej wersji systemu. Testerzy weryfikowali, czy wszystkie nowo wytworzone funkcje POL-onu są dostępne wraz ze wszystkimi poprawkami (czyli odpowiadają temu, co miało miejsce w środowisku deweloperskim w momencie zakończenia sprintu).

Drugim aspektem były testy regresji•. Były one wykonywane równolegle przez wszystkich członków zespołu testów, jednocześnie dla funkcji systemu dostępnych dla użytkownika zalogowanego w kontekście jednostki nadzorującej, w kontekście instytucji nauki i szkolnictwa wyższego oraz użytkownika niezalogowanego (zestawienia ogólnodostępne). Testowane były wszystkie główne widoki wyświetlane przez system, przyciski wywołujące podstawowe ścieżki działania i kreatory w systemie. Były to testy głównych przypadków użycia systemu (dodawanie i wyświetlanie danych w POL-onie, bez prób wprowadzenia błędnych danych, bez edycji i usuwania danych), ewentualnie uzupełnione przez testerów o inne przypadki na podstawie wcześniejszych doświadczeń, wykonywane manualnie. Testy regresji w środowisku przedprodukcyjnym trwały do około jednego dnia roboczego.

W przypadku wykrycia istotnych defektów następowała poprawa i wdrożenie poprawionej wersji w środowisku testów przedprodukcyjnych i ponowna kontrola jakości. Pomniejsze usterki naprawiane były w ramach nowego sprintu i retestowane w środowisku deweloperskim. Po pomyślnym zakończeniu testów system wdrażany był w środowisku produkcyjnym.

Równolegle trwały prace nad przygotowaniem automatów testowych. Członkowie zespołów deweloperskich przygotowali framework testowy i testy funkcjonalne pierwszych modułów POL-onu z użyciem narzędzi Cucumber i Selenium. Dalszy rozwój frameworka i tworzenie dalszych testów przejął zespół testów. Stworzone zostało kolejne środowisko testowe, przeznaczone wyłącznie do codziennych automatycznych testów regresji. Użycie automatów testowych zastąpiło też testy w środowisku przedprodukcyjnym – część zautomatyzowanych testów, odpowiadająca wykonywanym wcześniej testom manualnym w tym środowisku, została wytypowana, oznaczona i skonfigurowana tak, aby można ją było uruchomić po wdrożeniu w środowisku przedprodukcyjnym.

[37] definicje w tym podrozdziale pochodzą ze słownika [34]

3.6.1. Narzędzia automatyzacji testów i sposób ich wykorzystania

Framework testowy jest modułem w repozytorium kodu źródłowego POL-onu, zawierającym kod napisany z wykorzystaniem bibliotek Cucumber i Selenium. Zawiera on pliki tekstowe z rozszerzeniem feature napisane w Gherkinie[38], klasy Java, tłumaczące polecenia w Gherkinie na kod w Javie[39], i właściwy kod w Javie wykorzystujący sterowniki Selenium WebDriver[40] do uruchomienia POL-onu i wykonania czynności testowych.

Testy automatyczne uruchamiane są współbieżnie z wykorzystaniem technologii Selenium Grid – sieci maszyn, na których uruchamiane są testy. Na jednej maszynie uruchomiony jest serwer aplikacyjny Selenium w roli koncentratora sieci (*hub*), z którym komunikują się automaty testowe, a na wielu maszynach serwer Selenium w roli jej węzłów (*node*), do których koncentrator kieruje instrukcje przeprowadzenia pojedynczego testu. Każdy węzeł Selenium jest skonfigurowany do uruchomienia określonej liczby instancji sterowników poszczególnych przeglądarek internetowych. Sieć Selenium wykorzystywana do testów POL-onu liczy pięć maszyn wirtualnych z systemami Linux Mint (dwie maszyny, na jednej pierwotnie zainstalowany był system 32-bitowy, a później 64-bitowy, na drugiej od początku 64-bitowy), Windows XP (później zastąpiony przez Windows 10), Windows 7 i Windows 8.1, na każdej sterowniki Selenium WebDriver dla przeglądarek Google Chrome i Mozilla Firefox (na maszynach z systemami z rodziny Windows też Microsoft Internet Explorer, a na Windows 10 przeglądarka Microsoft Edge). Jedna z maszyn z systemem Linux Mint pełni jednocześnie funkcję koncentratora i węzła. W toku prac jako optymalne w istniejących warunkach infrastrukturalnych zespół testów przyjął uruchamianie automatów testowych w 20 wątkach (dla pełnej puli testów; w określonych sytuacjach zdarzało się również wykorzystywanie innej liczby wątków, np. 10, 25, 50 lub 100).

Warto zwrócić tu uwagę na ograniczenie używanych narzędzi[41] – pula testów do przeprowadzenia *a priori* dzielona jest w tym przypadku na 20 części uruchamianych w poszczególnych wątkach. Przy uruchomieniu procesu testów automatycznych do koncentratora Selenium wysyłane jest żądanie przeprowadzenia 20 testów (brany jest pierwszy test z każdej części). Koncentrator rozdziela je po kolei do kolejnych węzłów (tak, aby jak najwięcej z nich wykorzystać, wszystkie w miarę możliwości w równym stopniu) – tu: po 4 instancje WebDrivera (a więc przeglądarki internetowej) na węzeł. Jeśli w którymś z wątków wykonane zostaną wszystkie przewidziane dla niego testy, pozostałe testy działają w 19 wątkach – koncentrator Selenium dostaje mniej żądań. W praktyce w przypadku POL-onu wielokrotnie spotykana jest sytuacja, w której część testów przydzielona do jednego wątku jest bez porównania bardziej czasochłonna od pozostałych, przez co przez dużą część procesu testów automatycznych nie jest fak-

[38] Gherkin to część składowa Cucumbera; zestaw słów kluczowych, przy użyciu których zestawia się przypadki testowe z kroków zdefiniowanych w języku naturalnym. Przykłady zawierają fragmenty kodu 3.5-3.7
[39] metoda, w której zaimplementowany jest krok testowy, ma cucumberową adnotację, zawierającą nazwę kroku użytą w Gherkinie (lub wyrażenie regularne, do którego dopasowywane są nazwy)
[40] część Selenium; sterownik uruchamia i kontroluje wybraną przeglądarkę internetową
[41] a przynajmniej, ówczesnych wersji tych narzędzi, dziś już nie używanych w taki sposób

tycznie wykorzystywana możliwość prowadzenia ich równolegle. Jest to jedno ze zjawisk opisanych w części 3.6.2.

Automatyczne testy regresji we wspomnianym wcześniej osobno wyznaczonym środowisku odbywały się według następującego schematu. Najpierw następowało wdrożenie nowej wersji systemu (kod pochodził z tej samej gałęzi repozytorium, co w przypadku środowiska deweloperskiego). Następnie tworzona była migawka bazy danych POL-onu. Potem uruchamiane były automaty testowe. Po zakończeniu testów baza danych była przywracana z utworzonej migawki do poprzedniego stanu. Cały proces, o ile nie został zakłócony przez czynniki zewnętrzne, trwał od około ośmiu do kilkunastu godzin. Testy w środowisku przedprodukcyjnym uruchamiane były bez tworzenia i przywracania migawek, w mniejszej liczbie wątków i trwały około ośmiu godzin.

3.6.2. Problemy w wykorzystaniu automatów testowych

Monolityczna architektura samego POL-onu 1 wymusiła równie monolityczną architekturę testów automatycznych POL-onu. Automatyczne testy funkcjonalne• zrealizowane zostały jako testy *end-to-end* (e2e) – testy systemu z poziomu interfejsu graficznego użytkownika. Wady takiego podejścia do automatyzacji w połączeniu z innymi czynnikami spowodowały szereg problemów, z którymi zmagał się zespół testów.

Pierwszym z tych problemów była czasochłonność testów. O ile uruchamianie codziennych testów, trwających ponad 8 godzin, w godzinach nocnych nie było samo w sobie uciążliwe, o tyle wielogodzinne testy regresji w środowisku przedprodukcyjnym nie pozwalały na podjęcie decyzji o wdrożeniu, przeprowadzenie testów i wdrożenie POL-onu tego samego dnia. Raport z testów trwających dłużej niż dzień pracy testera w sposób nieunikniony musiał być sprawdzany w następnym dniu roboczym. Gdy automaty testowe (lub trwające równolegle testy manualne) wykryły defekt systemu, informacja o tym docierała do programistów z opóźnieniem. Nawet natychmiastowa naprawa defektów prowadziła do kolejnego wdrożenia i następnych wielogodzinnych testów (chyba że podejmowano decyzję o rezygnacji z użycia automatów testowych, gdy ryzyko powstania błędów regresji wynikających z wprowadzenia poprawki uznawano za niewielkie, a ewentualne nowe błędy za możliwe do znalezienia przez testera manualnie). Czyniło to zestaw testów automatycznych narzędziem niewygodnym.

Drugim problemem była niestabilność testów. Wiele testów automatycznych kończyło się wynikiem fałszywie pozytywnym lub w ogóle nie uruchamiało się. Wynika to bezpośrednio z natury testów *end-to-end* (e2e) – testy takie weryfikują działanie systemu jednocześnie pod wieloma aspektami. Aby test przeprowadzany przez Selenium na interfejsie użytkownika powiódł się, musi być spełniony szereg warunków. Niespełnienie choćby jednego z nich kończy się niepowodzeniem testu, nawet jeżeli testowany komponent nie ma defektów. Wynika to ze zbytniego skupienia testów na interfejsie graficznym. Do czynników powodujących wynik fałszywie pozytywny testu należą m.in.:

- niedostosowanie testu do zmiany w GUI• (nawet jeśli jest to zmiana niedotycząca działania systemu z punktu widzenia użytkownika, jak zmiana identyfikatora komponentu interfejsu);
- przekroczenie czasu, w którym test czeka na zmianę w systemie, przez co następuje przedwczesne stwierdzenie przez test braku zmiany;
- zbyt szybka próba wykonania następnej akcji (i stwierdzenie przez test braku możliwości jej wykonania);
- zakłócenie danych testowych jednego testu przez drugi.

Wynik fałszywie negatywny, nieuruchomienie testu lub wszystkich testów mogą powodować takie czynniki jak:

- niezgodność wersji przeglądarki internetowej i jej sterownika;
- zbyt duże obciążenie maszyn w sieci Selenium (np. przy zbyt dużej liczbie testów uruchomionych jednocześnie);
- niestabilność serwera Selenium (np. przy długotrwałym użyciu);
- zmiany w bazie danych systemu (niemożność zalogowania się na testowe konta, brak lub zmiana obiektów używanych w testach, np. konkretnych instytucji)
- błędy w systemie blokujące przejście do testowanej części.

Niepowodzenie konkretnego testu występowało sporadycznie, losowo. Przy dużej liczbie testów było jednak pewne, że nie każdy test zakończony wynikiem pozytywnym faktycznie wskazywał na błąd w systemie. W efekcie osobną czynnością było weryfikowanie raportu z testów automatycznych i potrzebne było ponowne zweryfikowanie wybranych przypadków testowych (ręcznie lub z użyciem automatu testowego uruchomionego pojedynczo), co jeszcze bardziej wydłużało proces testowy.

Do codziennych zadań testerów należało m.in. tworzenie nowych testów. To, jak wyglądały nowo pisane, testy zależało w dużej mierze od tego, jak przygotowany był framework testowy. Uwidoczniła się tu różnica pomiędzy podejściami osób, które jako pierwsze definiowały kroki testowe w Gherkinie i metodach tłumaczących z Gherkina na Javę. Różnice w możliwych podejściach pokazują poglądowe fragmenty kodu 3.5 i 3.6, oparte na kodzie prawdziwych testów POL-onu.

Kod 3.5. Kod testu przejścia z wyszukiwarki do danych studenta (wersja pierwsza, z roku 2014)

```
Aspekt: STD.UC.004.Przeglądaj dane studiów studenta
  Założenia:
    Zakładając, że Loguję się do polonu z rolą 'INST_PR_WS'
    Oraz Poszerzam okno przeglądarki
    Oraz Przechodzę za pomocą menu "STUDENCI -> Wykaz studentów"
    Oraz Czekam na załadowanie strony '60' sekund

  Szablon scenariusza: Przeglądanie danych studiów studenta z zestawienia: 1 wiersz
        - 1 osoba
    Kiedy Wybieram na filtrze 'Rok akademicki:' wartość '---Wszystkie---'
    Oraz Czekam na załadowanie strony '3' sekund
    Oraz Rozwijam wiersz z napisem 'KRYTERIA DODATKOWE'
    Oraz Wypełniam filtr 'Nazwisko:' wartością '<nazwisko>'
    Oraz Naciskam przycisk z etykietą 'Szukaj' i czekam '60' sekund
```

```
    Oraz Klikam w link o nazwie '<nazwisko>'
    Oraz Klikam w zakładkę o nazwie 'Studia'
    Wtedy Sprawdzam, czy strona zawiera '<kierunek>'
    Oraz Sprawdzam, czy strona zawiera 'Dane dotyczące studiów'
    Oraz Wylogowuję użytkownika

    Przykłady:
      | nazwisko | kierunek  |
      | Kowalski | Etnologia |

  Szablon scenariusza: Przeglądanie danych studiów studenta z zestawienia: 1 wiersz
       = 1 kierunek
    Kiedy Wybieram na filtrze 'Rok akademicki:' wartość '---Wszystkie---'
    Oraz Czekam na załadowanie strony '3' sekund
    Oraz Rozwijam wiersz z napisem 'KRYTERIA DODATKOWE'
    Oraz Wypełniam filtr 'Nazwisko:' wartością '<nazwisko>'
    Oraz Naciskam przycisk z etykietą 'Szukaj' i czekam '60' sekund
    Oraz Zaznaczam opcję '1 wiersz = 1 kierunek'
    Oraz Czekam na załadowanie strony '5' sekund
    Oraz Klikam w link o nazwie '<nazwisko>'
    Oraz Klikam w zakładkę o nazwie 'Studia'
    Wtedy Sprawdzam, czy strona zawiera '<kierunek>'
    Oraz Sprawdzam, czy strona zawiera 'Dane dotyczące studiów'
    Oraz Wylogowuję użytkownika

    Przykłady:
      | nazwisko | kierunek  |
      | Kowalski | Etnologia |
```

Kod 3.6. Kod testu wyszukiwarki konferencji naukowych (wersja pierwsza, z roku 2014)

```
Aspekt: DUN
  Szablon scenariusza: DUN_UC_001_Wyszukaj konferencję
    Kiedy loguję się do polonu z rolą 'INST_KONFERENCJE_ADM'
    Oraz wybieram kontekst instytucji 'Uniwersytet Warszawski'
    Oraz przechodzę za pomocą menu "Działania upowszechniające naukę -> Konferencje
         naukowe"
    Oraz Wprowadzam nazwę konferencji '<nazwa>'
    Oraz Wprowadzam rok organizacji '<rok>'
    Oraz Wybieram rodzaj konferencji '<rodzaj>'
    Oraz Klikam przycisk Szukaj
    Wtedy Wyświetlona zostaje wyszukiwana konferencja '<nazwa>','<rok>'
    Oraz Wylogowuję użytkownika

    Przykłady:
      | nazwa    | rok  | rodzaj          |
      | Nazwa 1 | 2012 | międzynarodowa |
      | Nazwa 2 | 2013 | krajowa         |
```

W przypadku fragmentu 3.5 widać podejście niepoprawne: pojedyncze operacje biznesowe opisane są wieloma krokami, użyte są osobne kroki oczekiwania na zdarzenia w systemie, a czasy oczekiwania są sztywno ustalone. Fragment 3.6 wydaje się być prawidłowy. Istotnie, nie ma wspomnianych wyżej wad, jednak ma inne. Na przykład przycisk `Szukaj` w wyszukiwarce konferencji nie musi być zaimplementowany tak samo, jak przycisk o tej samej nazwie w innej wyszukiwarce, przez co metoda znalezienia tego przycisku w interfejsie graficznym może się nie powieść. Jeśli tak, to każdorazowe

uruchomienie testu z wykorzystaniem kroku `Klikam przycisk Szukaj` w innej wyszukiwarce zakończy się wynikiem negatywnym (test będzie błędny).

Konflikt taki może być rozwiązany na kilka sposobów, w zależności od sytuacji. Można dokonać zmiany nazwy pierwszego kroku, tak aby wskazywała jednoznacznie na kontekst, w jakim można go użyć, i osobno zaimplementować drugi krok, o niekolidującej nazwie – to jednak może prowadzić do rozrostu liczby kroków o podobnych nazwach. Z drugiej strony, może dojść do tego, że taka sama akcja w systemie będzie implementowana niezależnie przez różne osoby w różnych modułach. Można też dokonać zmiany w implementacji kroku (w przykładzie: znajdowanie dowolnego przycisku „Szukaj" w interfejsie), co może skutkować nieprzewidzianym, niewłaściwym zachowaniem (niewykryciem braku żądanego przycisku przez obecność innego o tej samej etykiecie, naciśnięcie niewłaściwego przycisku o tej samej etykiecie, popsuciem pierwszego testu itp.). Można też zmienić podział testu na kroki już w warstwie Gherkina – i w tym przykładzie zaimplementować naciśnięcie właściwego przycisku w kroku użycia wybranej wyszukiwarki. To ostatnie podejście można zaobserwować w kodzie 3.7. Zawiera on ostateczną wersję testu, którego pierwszą wersją jest 3.5, jak również ma wszystkie zalety widoczne we fragmencie 3.6.

Kod 3.7. Kod testu przejścia z wyszukiwarki do danych studenta (wersja ostateczna, z roku 2017)

```
Aspekt: STD.UC 004.Przeglądaj dane studiów studenta
  Szablon scenariusza: Użytkownik przegląda dane studiów studenta
    Zakładając, że Jako 'INST_PR_WS2' w menu "STUDENCI -> Wykaz studentów"
    Oraz Zaznaczam opcję '<opcja>'
    Oraz szukam studentów:
      | Nazwisko:       | <nazwisko>      |
      | Rok akademicki: | ---Wszystkie--- |
    Kiedy przechodzę do danych znalezionego studenta
    Wtedy sprawdzam, czy wyświetlone zostały dane studenta

    Przykłady:
      | nazwisko | opcja                 |
      | Kowalski | 1 wiersz = 1 osoba    |
      | Kowalski | 1 wiersz = 1 kierunek |
```

Różnica w podejściach okazała się być kolejnym problemem w eksploatacji automatów testowych – utrudniała (a więc wydłużała) implementację nowych testów, wymuszała doprowadzenie starych testów do lepszej postaci (czyli zwiększała liczbę czynności do wykonania przez testerów), pogarszała czytelność testów czy utrudniała równomierny podział obowiązków pomiędzy testerów.

Brak czasu na podniesienie jakości testów to problem, który wyrasta z poprzednich. W przypadku każdego z omówionych wcześniej zagadnień jest możliwe znalezienie lepszych rozwiązań, ale wymaga to czasu. A im gorzej napisane były testy, tym więcej czasu zajmowała ich eksploatacja i bieżące utrzymanie, co z kolei powodowało, że członkowie zespołu testów mniej czasu mogli poświęcić poprawie automatów – a to wszystko przy konieczności przeprowadzania testów manualnych bieżących zmian w systemie.

Ostatnim punktem, który należy poruszyć jest niedobór kompetencji w zespole testów, tzn. zbyt mała liczba etatów pracowników doświadczonych w używaniu automatów testowych, mogących poradzić sobie z wcześniej wymienionymi problemami (wiąże się to mocno z kwestią czasu).

3.6.3. Rozwiązania problemów

Opisane w poprzedniej części trudności, częściowo przy okazji zmiany architektury na mikroserwisową w POL-onie 2, a częściowo dzięki rozwojowi kadry, zostały rozwiązane.

Zwiększenie zasobów kompetencji w zespole testów odbyło się w kilku krokach. Stan osobowy zespołu został powiększony o osobę z doświadczeniem w programowaniu automatów testowych, która nie miała przydzielanych zadań związanych z testami manualnymi ani bieżącym utrzymaniem istniejących testów. Osoba ta zajęła się mankamentami całego frameworka testowego, wypracowywaniem dobrych praktyk obowiązujących przy pisaniu nowych automatów, implementacją nowych i poprawą implementacji wcześniejszych kroków testowych. Potem zespół testów był nadal powiększany, a jego członkowie zostali przeszkoleni pod kątem automatyzacji testów z wykorzystaniem Selenium i posiedli stosowne doświadczenie. Na skutek rozwoju kompetencji pracowników z większym stażem oraz rekrutacji zewnętrznej każdemu zespołowi deweloperskiemu odpowiada jeden członek zespołu testów odpowiedzialny za testy manualne nowych funkcji (dostarczanych przez ten zespół) i jeden odpowiedzialny za utrzymanie automatów testowych do starego POL-onu monolitycznego oraz utrzymanie i rozwój testów automatycznych do mikroserwisów nowego POL-onu. Dzięki temu poszczególni pracownicy nie muszą dzielić czasu pracy na testy manualne i automatyczne, co przekłada się na większą ogólną produktywność.

Architektura nowego POL-onu znalazła odzwierciedlenie w organizacji testów automatycznych. Testy e2e zostały ograniczone do roli testów GUI. Testy logiki biznesowej mikrousług POL-onu 2 realizowane są przez testy RESTful API•, pełniące też funkcję testów integracyjnych między tymi mikrousługami. Framework tych testów jest wspólny dla wszystkich modułów. Napisany został w Javie z użyciem Cucumbera i biblioteki REST Assured, co daje wysoki poziom spójności podejścia do tworzenia testów. Kod testów przechowywany jest w repozytoriach kodu poszczególnych mikrousług. Testy są uruchamiane niezależnie od siebie. Mimo dużej ich liczby, czas ich wykonania jest relatywnie krótki (szczegóły w tabeli 3.3).

Testy automatyczne w takim podejściu to nie tylko testy regresji, ale i narzędzie wspierające testy nowych funkcji systemu już nie czysto manualne, a zautomatyzowane. Do testowania warstwy logiki biznesowej wykorzystującej RESTful API konieczne jest narzędzie do wysyłania zapytań HTTP. Narzędzi takich jest wiele. Różnią się m.in. ilością czasu, jaki należy poświęcić na rozpoczęcie testów nowego systemu. Używanie takiego osobnego narzędzia może prowadzić jednak do sytuacji, w której ta sama praca będzie wykonana dwukrotnie: w tym osobnym narzędziu przez testera i w frameworku testowym przez programistę testów. Dlatego jeśli to tylko możliwe, programista testów

implementuje potrzebne kroki w Gherkinie i Javie, a tester manualny wykorzystuje je do swojej bieżącej pracy. Programista testów może też tymczasowo przejąć całość testów funkcjonalnych nowych funkcji modułu, co jest dodatkową korzyścią w niecodziennych sytuacjach, jak choroby i urlopy większej liczby członków zespołu testów jednocześnie.

Interfejs graficzny nowego POL-onu budowany jest z wykorzystaniem frameworka Angular. W tej sytuacji do stworzenia testów automatycznych do niego wybrany został framework Protractor, który korzysta z Selenium dla języka JavaScript i dodaje obsługę komponentów angularowych. Wykorzystany został również Cucumber oraz opisana w części 3.6.1 sieć Selenium. Testy zostały napisane w języku TypeScript. Rolą tych testów jest sprawdzenie poprawności działania samego interfejsu graficznego (warstwy wyższej systemu) i jego komunikacji z warstwą logiki biznesowej (niższą) – nie następuje testowanie warstwy niższej za pośrednictwem wyższej, nie są to więc testy *end-to-end* takie jak w POL-onie monolitycznym. Testy działają w 5 wątkach i trwają około 10 minut.

Testy automatyczne POL-onu 1 nie zostały porzucone. Poszczególne automaty testowe do wyłączonych modułów są wyłączane, ale nie usuwane. Próby poprawy stabilności i szybkości ich działania trwały równolegle z rozwojem testów POL-onu 2. Zakończone zostało używanie osobnego środowiska testowego do codziennego przeprowadzania całej puli testów. Obecnie wykonywane są one raz w tygodniu w środowisku deweloperskim. Dzięki wyłączeniu niepotrzebnych automatów czas testowania systemu w środowisku przedprodukcyjnym skrócił się do około 100 minut. Poprawa komponentów kodu testów związanych z najdłuższym oczekiwaniem i najczęściej kończącymi się niepowodzeniem przypadkami doprowadziła do zmniejszenia czasu trwania procesu testowego do około 40 minut. Dalsze wyłączenia nieużywanych testów poskutkowały skróceniem testów do około 18 minut. Dzięki tym optymalizacjom pełne testy w środowisku deweloperskim trwają około 160 minut.

3.6.4. Statystyki testów i efekty większej automatyzacji

Tabela 3.2 zawiera zestawienie liczb poszczególnych rodzajów artefaktów wytworzonych przez zespół testów POL-onu. Wartości dotyczące POL-onu 1 należy traktować jako przybliżenie liczby testów, która kiedykolwiek była uruchamiana równolegle. Niektóre z nich powstały po wyłączeniu innych, niektóre pojedyncze testy zostały usunięte, z niektórych szablonów scenariuszy usunięte zostały poszczególne przypadki[42]. Niektóre wreszcie testy przeznaczone były do wykonania tylko w jednym ze środowisk testowych. Liczba modułów jest wartością orientacyjną, ponieważ pewna część testów wykonywała operacje w wielu modułach, a niektóre większe moduły składały się z kliku części, które z punktu widzenia rozwoju testów były odrębne i można byłoby też policzyć je osobno. Testy do POL-onu 2 są sukcesywnie dopisywane. Tabela nie zawiera danych

[42] w podanych wcześniej fragmentach kodu 3.5-3.7 obecne były szablony scenariuszy, czyli testy posiadające taką samą strukturę dla wielu różnych danych testowych zestawionych w tabelę oznaczoną słowem kluczowym „Przykłady". Istnieją również proste scenariusze, do których nie podaje się przykładów

o testach innych systemów powiązanych z POL-onem, jak Ogólnopolskie Repozytorium Prac Dyplomowych (ORPD)[43].

Tabela 3.2. Liczba testów e2e

	POL-on 1	POL-on 2
Liczba modułów, do których powstały testy	34	8
Liczba plików z testami	906	16
Liczba scenariuszy	550	19
Liczba szablonów scenariuszy	1329	63
Liczba przykładów	5309	300
Łączna liczba przypadków testowych	5859	319

Zauważyć można, że w testach POL-onu 2 liczba zautomatyzowanych przypadków testowych w przeliczeniu na moduł (prawie 40) jest dużo niższa niż w POL-onie 1 (ponad 170). Pokazuje to wyraźnie, że testów na tym poziomie powstaje teraz zdecydowanie mniej, niż powstało w poprzedniej odsłonie systemu.

Tabela 3.3 zawiera zestawienie modułów POL-onu 2 (z RPPD włącznie) z informacją o liczbie zautomatyzowanych przypadków testowych dla testów warstwy RESTful API i przybliżonym czasie ich wykonywania. Testy modułu Pracownicy uruchamiane są w dwóch osobnych procesach z racji długiego czasu ich wykonania i braku konieczności wykonywania całości testów jednocześnie.

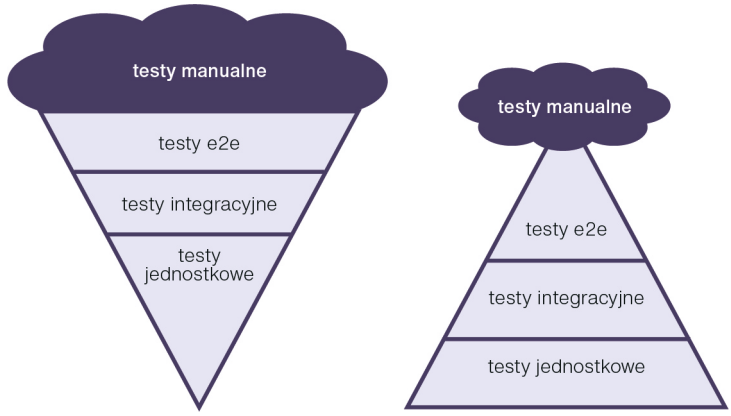
Tabela 3.3. Liczba testów RESTful API

Moduł	Liczba przypadków testowych	Przybliżony czas trwania testów [min]
Doktoranci	188	12
Instytucje	158	2
Kierunki studiów	406	14
Pracownicy	816	36
Pracownicy – import	297	16
Szkoły doktorskie	230	5
Użytkownicy	166	3
RPPD	402	2

[43] testy GUI ORPD nie korzystały z tego samego frameworka i nie używały żadnej z części składowych Cucumbera, natomiast Repozytorium Pisemnych Prac Dyplomowych (RPPD), które zastępuje ORPD, ma interfejs wspólny z POL-onem 2, a testy jego GUI powstają jako część testów interfejsu POL-onu 2

Liczba wszystkich przypadków testowych na tym poziomie testów (z wyłączeniem RPPD) wynosi 2261 przy zaledwie 6 modułach (czyli ponad 375 przypadków na moduł), a więc dużo więcej niż średnio w testach e2e w starym POL-onie. Pokrycie testami mikrousług nowego POL-onu jest zatem większe niż modułów starego POL-onu. Jednocześnie czas, który jest potrzebny na przeprowadzenie testów, jest dużo krótszy – częściowo dzięki niższej czasochłonności testów RESTful API niż e2e, a częściowo dzięki podzieleniu systemu na mikrousługi i wdrażaniu modułów niezależnie od siebie.

Opisane w niniejszym podrozdziale cechy procesu testowego POL-onu 1 i POL-onu 2 pokazują korzyści płynące ze zwiększenia nacisku na poprawną i szeroką automatyzację pracy zespołu testów oraz zmniejszenia liczby testów na interfejsie graficznym za cenę wprowadzenia testów funkcjonalnych na niższym poziomie. Obserwacje te wpisują się w znany w testerskiej literaturze branżowej antywzorzec rożka i wzorzec piramidy testów (artykuły [13, 37], Ilustracja 3.10). Szerokości warstw na ilustracji obrazują wielkość liczby testów automatycznych w danej warstwie. Warstwa testów jednostkowych nie była przedmiotem rozważań w tym podrozdziale. Warstwa testów integracyjnych właściwie nie istniała w przypadku POL-onu monolitycznego. Najwięcej (za dużo) było testów manualnych, wspieranych przez testy *end-to-end* – model ten był własną wersją rożka testowego. Świadome działania zespołu testów i całego zespołu rozwijającego POL-on nakierowane były na dojście do implementacji modelu piramidy, w której testy interfejsu są tylko uzupełnieniem testów w niższych warstwach. Praca ta zakończyła się sukcesem, z zyskiem dla całego projektu.



Ilustracja 3.10. Antywzorzec rożka i wzorzec piramidy testów
Źródło: opracowanie własne za alisterbscott.com/kb/testing-pyramids

ROZDZIAŁ 4

TECHNOLOGIA I ARCHITEKTURA SYSTEMU

Artur Idzi
Marek Michajłowicz
Emil Podwysocki
Andrzej Sadłowski
Antoni Sobkowicz

Poniższy rozdział skupia się wyłącznie na przedstawieniu wniosków z doświadczeń z rozwoju systemu na gruncie technicznym. Motywem przewodnim opisanych eksperymentów jest przejście ze stanu pierwotnej wersji systemu, zbudowanego w oparciu o model architektury monolitycznej do jego nowej wersji, stanowiącej konglomerat rozproszonych usług zgodnych z wzorcem architektury mikrousługowej. Rozdział ten w mniejszym stopniu skupia się na przedstawieniu aspektów teoretycznych, opisujących podobne doświadczenia w bogatej literaturze przedmiotu np. [5, 6, 16, 29], a bardziej bazuje na doświadczeniach autorów z własnych eksperymentów.

4.1. Zastosowana architektura

Obecnie główny element ekosystemu POL-on, sam system POL-on[44], występuje w dwóch wersjach.

1. **POL-on 1** – system budowany w latach 2011-2018 w oparciu o architekturę monolityczną, pełniący obecnie rolę systemu typu *legacy*;
2. **POL-on 2** – system budowany od roku 2018 jako rozwiązanie oparte o wzorzec mikrousługowej architektury rozproszonej, powstający przez wydzielenie funkcjonalności z monolitu.

Każdy z elementów wskazanych powyżej charakteryzuje się diametralnie różną architekturą i technologią. Zmuszone są jednak współpracować ze sobą bardzo ściśle, co uwydatnia rolę warstwy integracyjnej w opisywanym zagadnieniu. Dla oddania skali obu systemów przedstawiono statystyki w Tabelach 4.1, 4.2 oraz 4.3, prezentujące

[44] każda składowa systemu cechuje się innym rozwiązaniem technicznym

liczbę linii napisanego kodu w różnych językach programowania. Nie jest to miara szczególnie wyrafinowana i mówiąca wiele na temat wewnętrznej struktury systemu oraz jego złożoności. Jest to jednak pierwszy sposób zobrazowania rozmiaru systemu plasującego go w gronie największych systemów wspierających procesy na szczeblu całego państwa[45].

Tabela 4.1. Liczba linii kodu w podziale na technologie w POL-on 1

Projekt	Java	JavaScript/TypeScript	jsp	XML	CSS	HTML
nws	992k	5,5k	0	90k	15k	392k
orpd	41k	2,7k	4,1k	907	12k	0
Suma	**1033k**	**8,2k**	**4,1k**	**91k**	**27k**	**392k**

Tabela 4.2. Liczba linii kodu w projektach związanych z interfejsem użytkownika w podziale na technologie w POL-on 2

Projekt	TS	HTML	CSS
polon2-doctoral-studies-gui	3,6k	1,9k	63
polon2-fields-of-study-gui	6,8k	4,1k	120
polon2-gus-gui	2,8k	1k	283
polon2-home-gui	0,279k	0,27	316
polon2-institution-gui	11k	5,1k	15
polon2-opiforms-gui	3,8k	1,1k	245
polon2-patents-gui	3k	3,4k	26
polon2-reports-gui	0,658k	0,231k	27
polon2-rppd-gui	4,2k	1,8k	512
Suma	**36,137k**	**18,901k**	**1,6k**

[45] polon.nauka.gov.pl/siec-polon (dostęp: 18.06.2021)

Tabela 4.3. Liczba linii kodu w projektach związanych z kodem po stronie serwera w podziale na technologie w POL-on 2

Projekt	Java	XML	Python
doctoral-studies-api	19k	374	0
employees-api	65k	907	0
fields-of-study-api	48k	1248	0
gus-api	10k	288	0
opiforms-api	8,6k	332	0
pdat-api	26k	474	0
phd-students-api	14k	432	0
queue-api	0,474k	86	0
users-api	15k	1000	0
mapper	0	0	2605
Suma	**206k**	**5,1k**	**2,6k**

Z analizy samej liczby linii kodu systemów POL-on 1 i POL-on 2 wynika, że drugi z nich ma teraz około 16% objętości systemu pierwszego (pod względem liczby linii kodu). To fakt, bo duża część funkcjonalności nie została jeszcze przepisana do nowego systemu. Co więcej, to z uwagi na duże ryzyko związane z możliwym niedotrzymaniem terminów wynikających z harmonogramu projektu, część funkcjonalności nadal jest rozwijana i utrzymywana w systemie POL-on 1. Jednak dzieje się tak nie dlatego, że w POL-on 1 kod powstaje szybciej, tylko dlatego, że łatwiej zmodyfikować funkcjonalność w POL-on 1, niż napisać ją od nowa w POL-on 2. Dodatkowo należy zwrócić uwagę na fakt, iż podczas przepisywania funkcjonalności z wersji 1 do 2 funkcjonalność w systemie POL-on 1 istnieje nadal, jest tylko niedostępna dla użytkowników. Tu należy bardzo mocno podkreślić, iż przejście z aplikacji stworzonej jako monolit do mikroserwisów nie jest procesem bezkosztowym ani prostym. W naszym przypadku podjęcie decyzji o takim przejściu wynikało z problemów ze stabilizacją i wdrażaniem nowych funkcjonalności rozwijanych przez kilka zespołów, problemami z regresją oraz wydajnością.

4.1.1. Warstwy i składowe systemu

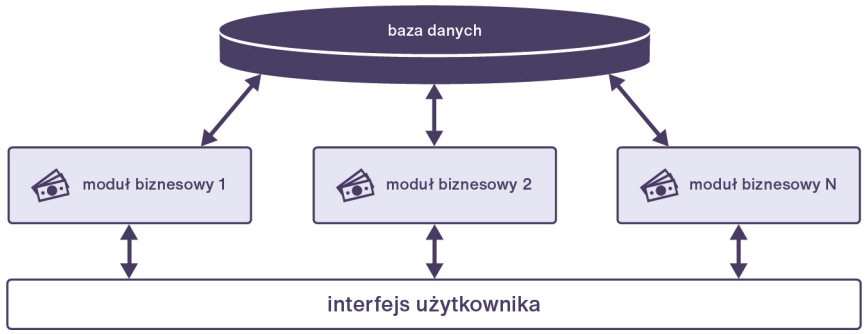
Pomimo tego, że architektury obu wersji systemu POL-on są różne, obie wykorzystują analogiczne komponenty do realizacji swoich celów. Mówiąc w bardzo dużym uproszczeniu, znajdują się w nich:

- baza danych;
- moduły biznesowe odpowiedzialne za realizację poszczególnych wymagań;
- interfejs użytkownika.

Dodatkowo tylko w POL-on 2 znajduje się:

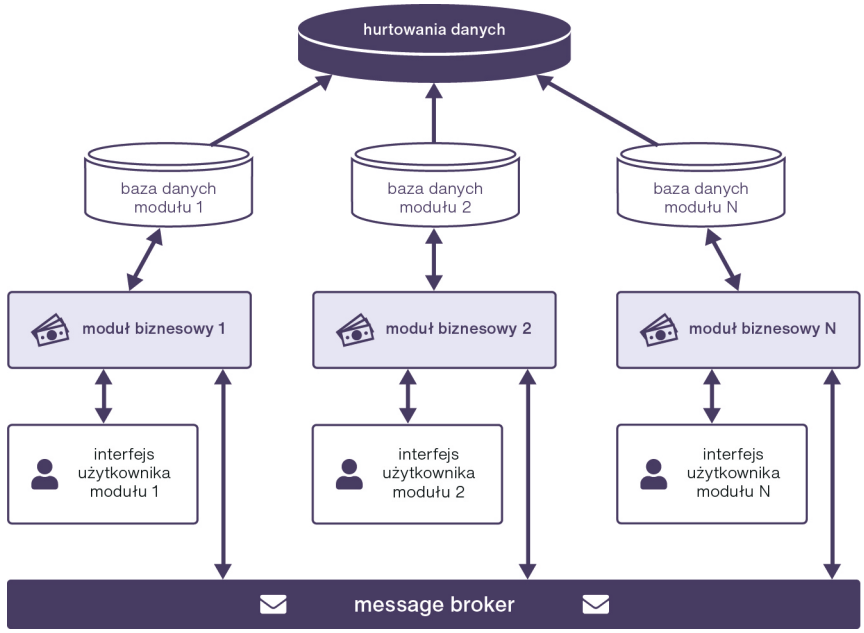
1. dostawca wiadomości (ang. *message broker*);
2. hurtownia danych.

Jednak już sposób organizacji tych elementów jest fundamentalnie odmienny. Na Ilustracji 4.1 schematycznie przedstawiono poszczególne części POL-on 1 oraz sposób ich wzajemnej komunikacji. Należy zwrócić uwagę na wspólną bazę danych i interfejs użytkownika.



Ilustracja 4.1. Składniki POL-on 1

Dla porównania na Ilustracji 4.2 w analogiczny sposób przedstawiono składniki POL-on 2.



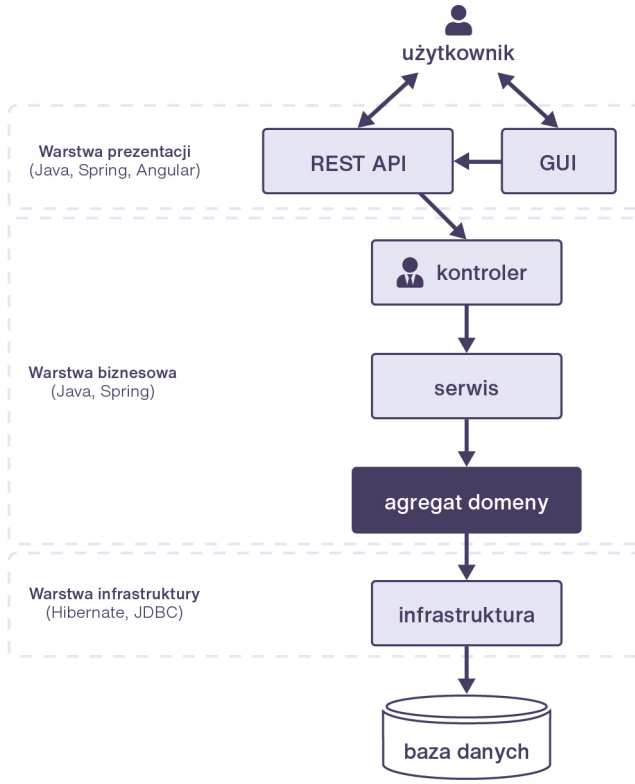
Ilustracja 4.2. Składniki POL-on 2

Zasadniczą różnicą między architekturą POL-on 1 a POL-on 2 jest to, że w wersji aplikacji POL-on 2 każdy z modułów biznesowych ma własną bazę danych, do której żaden inny moduł nie ma dostępu, oraz własny współpracujący tylko z nim interfejs użytkownika. Dodatkowo, choć wszystkie moduły mogą wykorzystywać wzajemnie swoje usługi, został wprowadzony asynchroniczny dostawca wiadomości odpowiedzialny za usprawnienie komunikacji. Hurtownia danych odpowiedzialna jest za agregację i analizę danych zgromadzonych w poszczególnych bazach każdego modułu.

4.1.2. Architektura logiczna

Ponieważ POL-on 2 zrealizowano w architekturze mikroserwisowej, poszczególne mikroserwisy mogą nieznacznie się od siebie różnić – w zależności od zastosowanego rozwiązania technicznego. Warstwa prezentacji danych we wszystkich mikroserwisach została zbudowana za pomocą języka Angular. Na potrzeby implementacji warstwy biznesowej w dużej części modułów zastosowano język Java wraz z frameworkiem Spring, ale w związku z przejściem na mikrousługi jeden z modułów zaimplementowano w języku Python. W przypadku używania bazy relacyjnej warstwę dostępu do danych zaimplementowano z pomocą biblioteki Hibernate lub SpringJDBCTemplate, w przypadku używania bazy nierelacyjnej za pomocą SpringDataJpa. Do przechowywania danych służy baza Oracle, Mongo, ElasticSearch, a także system plików. Użycie określonego sposobu przechowywania danych jest dostosowane do wymogów biznesowych, funkcjonalności systemu oraz szybkości jego działania. Zgodnie z Ilustracją 4.3 użytkownik podczas pracy z systemem może skorzystać zarówno z graficznego interfejsu użytkownika, jak i bezpośrednio wykorzystać udostępnione API[46]. Następnie na poziomie kontrolera sprawdzana jest poprawność struktur danych i użytych formatów. Później sterowanie jest przekazywane do serwisu, gdzie wykonywane są proste walidacje biznesowe, niezależne od już istniejącego stanu w bazie. Gdy ta faza zostanie ukończona, sterowanie przejmuje agregat domeny wykonujący najtrudniejsze walidacje biznesowe, uwzględniające stan danego obiektu oraz obiektów zależnych w bazie danych. Ostatecznie sterowanie przekazywane jest do warstwy infrastruktury odpowiedzialnej za zapisanie informacji w bazie danych.

[46] API jest docelowo skierowane do maszynowej integracji systemów zewnętrznych



Ilustracja 4.3. Architektura logiczna

4.1.3. Komunikacja między POL-on 1 i POL-on 2

Wraz z potrzebą komunikacji między systemami wchodzącymi w skład ekosystemu POL-on 1 zaczęto eksperymenty związane z implementacją szyny ESB (ang. *Enterprise Service Bus*). Miała ona odpowiadać za komunikację między aplikacjami, jednak rozwiązanie to zostało odrzucone z następujących powodów:

- konieczności powołania kanonicznego modelu danych, zarządzania nim, utrzymywania i rozszerzania go, jak zaproponowano w [36]. W pozycji [21] model taki wymaga łączenia danych z różnych źródeł, pozycja [26] przedstawia taki model jako właściwy dla hurtowni danych. Głównym problemem, na który zwrócono uwagę, była duża trudność w jakiejkolwiek modyfikacji takiego modelu, jako iż zmiana taka może pociągać za sobą zmianę we wszystkich systemach już zintegrowanych z szyną;
- wyniesienia logiki konsumenta danych, do warstwy szyny ESB[47]. Zauważono również, że użycie szyny danych sprawia, iż logika konsumenta przenika do szyny;

[47] ang. *Enterprise Service Bus*

- pracochłonności związanej z utrzymaniem kolejnej warstwy systemu, wynikającej z opisanych powyżej dodatkowych elementów powielających logikę biznesową oraz model danych zintegrowanych aplikacji.

Jako rozwiązanie komplementarne względem szyny danych sprawdzono również rozwiązanie oparte na usługach REST. Usługi takie są udostępniane przez moduły, z którymi te usługi powinny być zintegrowane. Moduł taki jest „świadomy" udostępniania swoich danych, w związku z czym przechowuje dane w postaci modelu zoptymalizowanego do odczytu. Dla zmniejszenia ruchu sieciowego wprowadzono również brokera wiadomości. Jego celem jest poinformowanie o zmianie wprowadzonej w systemie źródłowym. Jeżeli konsument wiadomości jest zainteresowany taką zmianą, to może ją pobrać i odłożyć w swojej pamięci podręcznej. W przeciwnym razie może pominąć taką informację. W toku prac sprawdzono warianty wiadomości wysyłanych przez brokera:

- przesyłano tylko zmiany obiektu wraz z identyfikatorem takiego obiektu. Podejście to miało maksymalnie ograniczyć ruch sieciowy. Tym niemniej głównym problemem, jaki się pojawił, była niemożność ustalenia, czy konsument takiej wiadomości posiadał dane na temat poprzedniej wartości obiektu, żeby poprawnie nanieść otrzymaną zmianę;
- przesyłano cały zmieniony obiekt wraz z informacją o typie zmiany. Podejście to miało wyeliminować minus poprzedniego rozwiązania, ale okazało się, że nie wszyscy konsumenci wiadomości potrzebują odbierać wszystkie wiadomości;
- przesyłano tylko informacje na temat zmiany wraz z informacją na temat typu tej zmiany z linkiem, który umożliwiał pobranie nowego stanu.

Obecnie przeprowadzamy testy i oceniamy, które z rozwiązań przedstawionych powyżej lepiej się sprawdzą w naszym środowisku.

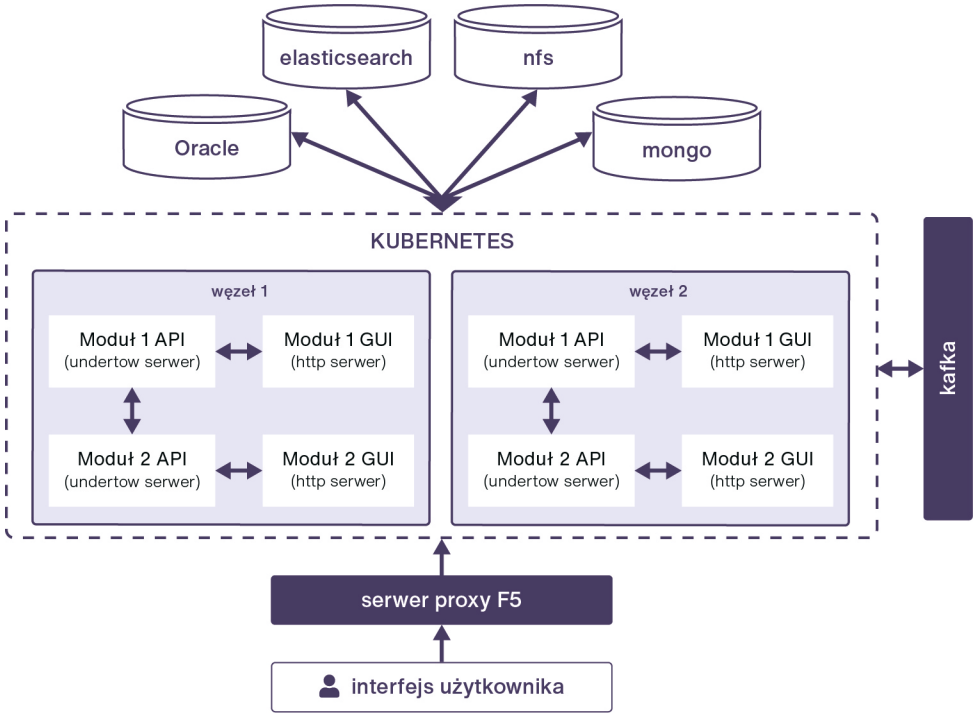
Należy podkreślić, że wymiana informacji między systemami jest operacją skomplikowaną. Podczas analizy sposobów wymiany wiadomości rzadko kto zwraca uwagę na sytuacje awaryjne, które mogą się pojawić podczas działania systemu. Aplikacje komunikują się ze sobą za pomocą usług REST i tak pobrane informacje trzymają w pamięci podręcznej, która z kolei jest aktualizowana przez wiadomość z brokera wiadomości. Zależnie od implementacji pamięć podręczna może być persystentna ciągle lub tylko przez pewien czas. Kluczowym zadaniem jest zapewnienie spójności pamięci podręcznej ze źródłem prawdy, co w zależności od implementacji może opierać się o wiadomości od brokera wiadomości, interwałowe odświeżanie pamięci podręcznej lub połączenie obu tych rozwiązań. System POL-on 2 komunikuje się z POL-on 1 właśnie w opisany powyżej sposób.

By to opracowanie było kompletne, należy również opisać sposób integracji systemów POL-on 2 i POL-on 1 przez bazę danych. Niektóre dane można zmienić tylko w systemie POL-on 2. Do POL-on 1 dane te trafiają przez zmianę bazy danych wyzwoloną przez wyzwalacz w bazie danych. Jest to proces trudny do testowania. W przypadku wykrycia błędu trudno określić, które dane są błędne. Trudna jest także ocena, czy proces związany z migracją danych jest już poprawiony oraz czy znaleziony błąd nadal

występuje. Jest to rozwiązanie tymczasowe do czasu przeniesienia wszystkich modułów do POL-on 2.

4.1.4. Architektura fizyczna

Na Ilustracji 4.4 została schematycznie przedstawiona architektura fizyczna systemu POL-on 2. Żądanie użytkownika trafia na serwer proxy, który ma za zdanie zrównoważyć obciążenie i przekierować żądanie na jeden z dostępnych węzłów Kubernetes. Jak przedstawiono na ilustracji, główną bazą danych POL-on jest Oracle, jednak należy podkreślić, że poszczególne moduły biznesowe posiadają w pełni odseparowane fragmenty bazy danych dostępne tylko dla nich lub też używają, przedstawionych również na ilustracji, bazy Mongo, Elasticsearch lub systemu plików. Ze względu na wysokie koszty licencji i charakter bazy danych Oracle nie zdecydowano się na jej konteneryzację. Również dostawca wiadomości „Kafka", został udostępniony jako zewnętrzna usługa dostępna dla węzłów Kubernetes. Usługi związane z Elasticsearch i bazą Mongo obecnie również są poza klastrem, jednak nie wynika to z ograniczeń technicznych, a z zaszłości historycznych. W niedalekiej przyszłości zaplanowano przeniesienie tych usług do Kubernetes, aby lepiej i bardziej wydajnie zarządzać infrastrukturą.



Ilustracja 4.4. Architektura fizyczna

4.2. Doświadczenia związane z wdrożeniem modelu mikrousługowego

System POL-on służy do gromadzenia bardzo szerokiego zakresu danych związanych z nauką i szkolnictwem wyższym w Polsce. Zakres gromadzenia danych i model danych określają ramy prawne i regulacje, które często się zmieniają. Zmiany w prawie generują wiele modyfikacji w aplikacji i jej modelu danych. Wiele zmian w ramach prawnych może rodzić potrzebę dokonania znaczącej przebudowy systemu, ale bardzo często dotyczą one tylko kilku modułów funkcjonalnych. Nawet jeśli zmiana dotyczy tylko jednej funkcji, nie jest możliwe stworzenie nowej wersji aplikacji bez integracji z innymi modułami, co wydłuża czas dostarczenia na rynek[48].

Źródłem wielu problemów był bardzo duży rozmiar aplikacji, który utrudniał wdrażanie nowych wersji systemu w rozsądnych ramach czasowych, a także powodował opóźnienia w odpowiadaniu na żądania użytkowników. Aplikacja podzielona była na moduły funkcjonalne, które były utrzymywane i rozwijane niezależnie przez różne zespoły wytwórcze. Aby ponownie wdrożyć aplikację w środowisku produkcyjnym, należało zsynchronizować pracę wielu zespołów. Ta metoda zarządzania złożonym systemem oprogramowania wymagała dużego nakładu pracy i częstych testów integracyjnych. W przypadku architektury monolitycznej skalowanie odbywało się poprzez uruchomienie większej liczby kopii aplikacji.

Takie podejście nie rozwiązywało jednak problemu dostępu do bazy danych, ponieważ każda kopia aplikacji korzystała z tej samej instancji. Im więcej było instancji aplikacji, tym większa liczba transakcji musiała być obsługiwana przez jedną relacyjną bazę danych. Przeciążenie bazy danych przez wywołanie zbyt wielu wystąpień aplikacji mogło wpłynąć na niezawodność systemu. W praktyce system nie był skalowalny ze względu na transakcje w bazie danych. Ponadto jedna scentralizowana składnica obsługiwała proste operacje odczytu i zapisu, a także różne zaawansowane raporty używane do wspomagania decyzji i analizy. Występował więc problem użycia systemu do pochłaniających duże zasoby operacji transakcyjnych (zapis/odczyt) oraz do celów analitycznych.

4.3. Techniczne aspekty modelu mikrousługowego

System POL-on 1 został zrealizowany w architekturze monolitycznej, czyli składał się z jednej dużej, centralnej aplikacji. Jak słusznie zauważono w [19], takie podejście ma swoje zalety: szybszy czas dostarczenia oprogramowania do klienta, oraz stosunkowo prosty stos technologiczny. Jednak w miarę upływu czasu i rozwoju systemu stopniowo

[48] ang. *time to market* – jedna z głównych metryk do weryfikacji skuteczności procesu wytwarzania oprogramowania wykorzystywana w formie projektów zwinnych

zaczęły pojawiać się problemy. Wydłużył się czas niezbędny do przeprowadzenia testów integracyjnych i wzrosła liczba przestojów systemu. Stopniowo stało się jasne, że dalsze utrzymywanie takiej architektury będzie coraz trudniejsze. Dlatego też zdecydowano, że POL-on 2 powstanie w oparciu o architekturę mikroserwisów.

Proces podzielenia tak dużego i złożonego zakresu funkcjonalności jaki występował w POL-on (ponad milion linii kodu), był zadaniem skomplikowanym. Choć biznesowo system został podzielony na moduły, to jednak na poziomie implementacyjnym nie były one odseparowane i zachodziła pomiędzy nimi dość intensywna komunikacja, co stoi w sprzeczności z zaleceniem samowystarczalności opisanym np. w [28].

Ideałem byłaby sytuacja, w której moduł mógłby realizować swoją funkcjonalność, pomimo przerwy w działaniu pozostałych części systemu. Takie podejście rodzi pokusę tworzenia bardzo wyspecjalizowanych usług, co z kolei przekłada się na ich dużą ilość. Tego typu architekturę nazywa się potocznie nanoserwisami i jest ona uważana za antywzorzec. Jest to zjawisko niekorzystne ze względu na zwiększony narzut czynności administracyjnych, problemy z utrzymaniem zgodności wersji poszczególnych modułów i zmniejszenie wydajności z powodu komunikacji sieciowej ([19]).

W toku prac przygotowawczych do wydzielenia mikroserwisów z aplikacji POL-on, podjęto kilka decyzji kierunkowych, które stanowiły podwaliny dalszych kroków:

- należało unikać tworzenia zbyt dużej liczby modułów;
- granice modułów miały być wyznaczane tak, aby miały możliwie szeroki zakres funkcjonalności odpowiadający logice biznesowej;
- każdy moduł miał się składać z dwóch fizycznie oddzielnych mikroserwisów – jednego odpowiedzialnego za realizację logiki biznesowej (API•) i drugiego odpowiedzialnego za wyświetlanie interfejsu użytkownika (GUI);
- należało unikać silnych powiązań pomiędzy modułami;
- poszczególne moduły miały się ze sobą komunikować korzystając bezpośrednio ze swojego API, jak i za pomocą brokera wiadomości (np. Kafka).

Przy dużej aplikacji zrealizowanej w architekturze monolitycznej, każde wdrożenie wymuszało zatrzymanie systemu, co było uciążliwe dla użytkowników. Szczególnie w dużych uczelniach, mających dedykowanych pracowników do określonych modułów, taka sytuacja była niezrozumiała – wdrażane funkcjonalności lub poprawki mogły nie dotyczyć zakresu funkcjonalności, za który byli odpowiedzialni. Dlatego troska o zapewnienie jak największej dostępności i zminimalizowanie przestojów systemu była priorytetem od początku prac nad stworzeniem nowej wersji systemu.

W każdej aplikacji można zidentyfikować dwie główne przyczyny wdrożenia:

- modyfikacja funkcjonalności biznesowej
- korekta błędów

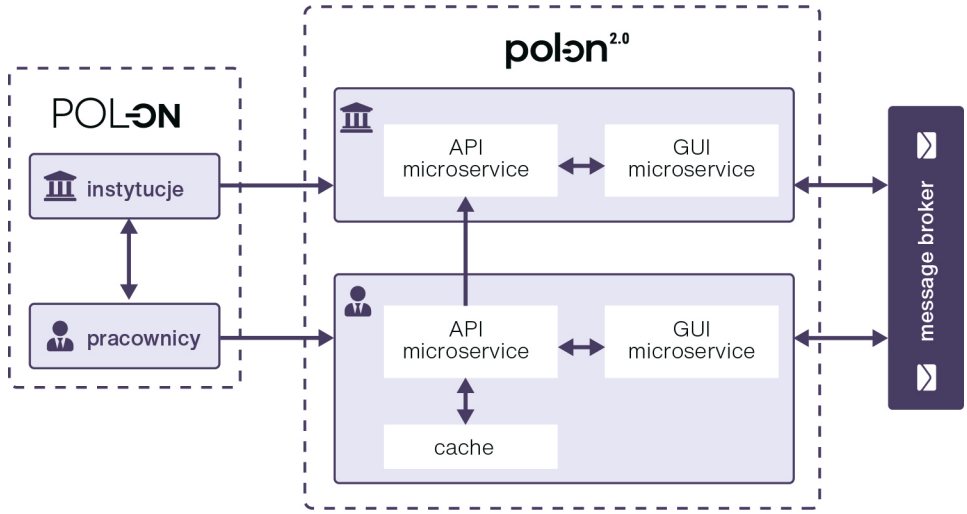
Aby minimalizować liczbę wdrożeń, poprawki błędów, które nie były krytyczne, wdrażane były dopiero wraz z kolejnym zaplanowanym wdrożeniem nowej funkcjonalności. Wydłużało to czas oczekiwania użytkownika na poprawkę oraz budziło niezadowolenie.

Podział na mikroserwisy (przedstawiony schematycznie na Ilustracji 4.5) spowodował, że wdrożenie jednego modułu nie skutkuje zatrzymaniem całej aplikacji – większość pozostałych modułów działa w tym czasie bez przeszkód, co znacząco ogranicza przestoje całej aplikacji. Z samego podziału wynika więc możliwość częstszego wykonywania wdrożeń i skrócenia czasu oczekiwania na pojawienie się poprawki lub nowej funkcjonalności. Dodatkowo w każdym module wydzielono usługi na API i GUI, co podyktowane było doświadczeniem zdobytym podczas utrzymywania POL-on 1, gdzie znaczna część przestojów systemu spowodowana była koniecznością wprowadzenia stosunkowo prostych zmian na interfejsie użytkownika. Dzięki temu zabiegowi taka sytuacja nie miała już miejsca.

W przypadku korzystania z architektury opartej na mikroserwisach zawsze należy się liczyć z sytuacją, gdy jedna usługa nie może uzyskać połączenia do drugiej. Może to być spowodowane awarią lub wdrożeniem. W takim przypadku mamy dwa wyjścia:

- poinformować użytkownika o tymczasowym problemie – licząc na jego wyrozumiałość i szybkie usunięcie usterki;
- zadbać o zwiększenie niezależności usługi – stosując np. buforowanie danych.

Najlepszym tego przykładem są informacje o instytucjach, które są wykorzystywane przez szereg innych modułów. Wiele z nich posiada dopasowaną do własnych potrzeb kopię danych o instytucjach. Zamiast bezpośrednio korzystać z mikroserwisu instytucji, korzystają one z własnego bufora danych, dzięki czemu utrzymana jest wysoka dostępność i wydajność, gdyż takie rozwiązanie jest wielokrotnie szybsze niż komunikacja za pośrednictwem sieci.

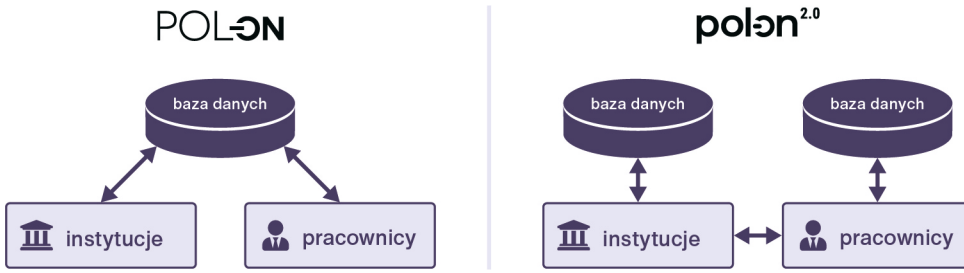


Ilustracja 4.5. Schemat architektury mikrousługowej

Pozostaje problem zapewnienia aktualności danych w buforze, co zostało rozwiązane za pośrednictwem wiadomości wysyłanych z modułu źródłowego do oprogramowania typu „message broker". Za każdym razem gdy dany obiekt, np. instytucja, zostaje zmodyfikowany, moduł wysyła informację o tym zdarzeniu wraz z aktualnym stanem. Tego typu wiadomości nasłuchują wszystkie zależne mikroserwisy i w oparciu o nie aktualizują stan posiadanych informacji. Ponieważ komunikacja tego typu odbywa się w sposób asynchroniczny, ewentualne przestoje poszczególnych modułów nie mają wpływu na ostateczny wynik. Wadą takiego rozwiązania jest wprowadzenie nowego kluczowego elementu do architektury jakim jest oprogramowanie służące do przesyłania wiadomości. Jego ewentualna awaria może doprowadzić do zatrzymania całego systemu. Oczywiście są techniki, które umożliwią zachowanie kolejki wiadomości po stronie nadawcy do momentu usunięcia awarii, ale w praktyce jest to rozwiązanie dość kosztowne i w przypadku dłuższych awarii może doprowadzić do powstania niespójnych danych. Kolejną trudnością jest konieczność zachowania formatu wiadomości – nadawca nie może zmodyfikować nadawanych wiadomości do momentu, w którym wszyscy potencjalni odbiorcy nie będą przygotowani do ich odbioru. Powoduje to powstanie łańcuchów zależności wdrożeń i wymaga odpowiedniego ich planowania. Jest to rozwiązanie dość złożone, dlatego część modułów zdecydowała się na prostsze rozwiązanie bazujące na gotowych bibliotekach buforujących takich jak EHCache[49]. Są one znacznie prostsze w implementacji i pozytywnie wpływają na wydajność, ale nie zapewniają całkowitej niezależności w przypadku awarii systemu źródłowego.

[49] ehcache.org (dostęp: 18.06.2021)

Dodatkową komplikacją jest zapewnienie spójności danych w funkcjonalnościach, które obejmują więcej niż jeden mikroserwis. Nie sprawia to żadnego problemu w architekturze monolitycznej, jednak jest bardzo trudne do osiągnięcia w architekturze mikroserwisowej. Ilustracja 4.6 pozwala lepiej zrozumieć ten problem.



Ilustracja 4.6. Problem zapewnienia spójności danych

Załóżmy, że przykładowa operacja „zatrudnienie pracownika" wymaga sprawdzenia, czy instytucja, w której ma być on zatrudniony, nie została zlikwidowana. W przypadku architektury monolitycznej w POL-on 1 sytuacja jest prosta: oba moduły systemu, zarówno „pracownicy" jak i „instytucje" zapisują dane do wspólnej bazy. Ponieważ czas zapisu jest bardzo krótki, ryzyko, że dojdzie do sytuacji, w której w tej samej chwili instytucja będzie zlikwidowana i pracownik zostanie w niej zatrudniony, jest bardzo małe. Dodatkowo na poziomie jednej bazy danych można wybrać taki poziom izolacji transakcji, który całkowicie wyeliminuje ten problem. Niestety, w przypadku architektury mikroserwisowej, gdzie każdy mikroserwis ma własną bazę danych i poszczególne mikroserwisy komunikują się ze sobą za pomocą REST API, czas potrzebny na komunikację może być znaczący. Mikroserwis „pracownicy" – pyta mikroserwis „instytucje" o status danej instytucji i otrzymuje odpowiedź „instytucja nie jest zlikwidowana". Rozpoczyna więc operację „zatrudnij pracownika". W tym czasie inny użytkownik wprowadza informację o likwidacji instytucji. Ponieważ są to operacje niezależne, może dojść do zatrudnienia pracownika w zlikwidowanej instytucji.

Na podstawie naszych doświadczeń przyjęto, że idealną spójność danych można zapewnić jedynie w ramach pojedynczego mikroserwisu. W przypadku funkcjonalności, które do działania wymagają kilku mikroserwisów należy zapewnić odpowiednie mechanizmy kontrolne i liczyć się z możliwością wystąpienia chwilowych niespójności. To pokazuje, jak ważne jest przemyślane ustalenie granic mikroserwisu.

Posiadanie systemu zbudowanego w architekturze mikroserwisowej daje również możliwość elastycznego skalowania aplikacji. W sytuacji gdy dana usługa jest przeciążona możemy bez problemów powołać do życia jej kolejną instancję, dzięki czemu zwiększamy liczbę operacji, które może ona wykonać. Dodatkowo posiadanie więcej niż jednej instancji znacząco zwiększa odporność systemu na awarie. W przypadku awarii

pierwszej – druga nadal działa. Oprogramowanie takie jak Kubernetes[50] jest w stanie zautomatyzować cały proces, skalując aplikację w zależności od obciążenia i dbając o to, żeby zawsze odpowiednia liczba instancji była dostępna.

Ma to zasadnicze znaczenie w przypadku takiego systemu jak POL-on, gdzie często mamy do czynienia z okresowym wzrostem obciążenia związanym z terminami nałożonymi na instytucje przez prawo. W takiej sytuacji możemy zwiększyć liczbę instancji jednej usługi, zmniejszając jednocześnie nadmiarowość innej. Prowadzi to do optymalizacji kosztów, gdyż dzięki temu można lepiej wykorzystywać istniejącą infrastrukturę i ograniczyć jej zakupy.

4.3.1. Rozwój systemu w architekturze mikrousługowej

W podrozdziale 4.3 zostały przedstawione główne korzyści z zastosowania architektury mikroserwisów w aspekcie utrzymania systemu – takie jak większa dostępność czy wydajność. Należy jednak zwrócić również uwagę na cechy tej architektury także w aspekcie rozwoju aplikacji.

Najważniejszą zaletą jest niewątpliwie możliwość wyboru optymalnego rozwiązania do realizacji danego modułu. W praktyce wygląda to tak jakbyśmy za każdym razem zaczynali pracować nad zupełnie nową aplikacją. W architekturze monolitycznej, cały ciężar wcześniej podjętych decyzji wpływa na obecne prace. Każda zmiana biblioteki, sposobu implementacji musi być za każdym razem dokładnie sprawdzona pod względem kompatybilności z kodem, który powstał na przestrzeni wielu lat. Potencjalnie zmiana w jednej części systemu może niekorzystnie wpłynąć na wydajność w innym miejscu.

Kod aplikacji starzeje się. Programiści pracujący w firmie zaczynają być znużeni ciągłą pracą w ramach tej samej technologii i szybciej pojawia się u nich wypalenie zawodowe. To przekłada się na ich zaangażowanie, a także na decyzje o zmianie pracy. Znalezienie pracownika, który chętnie przyjdzie do takiego projektu, jest trudne. Każdy developer doskonale wie, że najciekawsza, najbardziej rozwijająca, jest praca przy nowym projekcie. Wtedy gdy podejmuje się kluczowe decyzje architektoniczne, kiedy jeszcze wszystkie „drogi" są otwarte.

Wybór architektury mikroserwisowej rozwiązuje te problemy. Pojedynczy mikroserwis ma mniejszą funkcjonalność i jest niezależny od decyzji podjętych w innej usłudze. W związku z tym jego bezwładność jest znacznie mniejsza – ma mniej kodu, więc łatwiej go przerobić i przetestować. W przypadku mniejszych modułów można pokusić się o mniej standardowe rozwiązania. Są one zazwyczaj bardziej ryzykowne i żaden rozsądny architekt nie zgodziłby się na ich użycie w głównym systemie, ale jeżeli mówimy o pojedynczym module, który w dodatku pełni funkcję pomocniczą, to warto

[50] kubernetes.io (dostęp: 18.06.2021)

przeprowadzić taki eksperyment. Być może zakończy się on sukcesem i zostanie zaadaptowany w pozostałych modułach. Jest to szansa na szybsze i bardziej niezawodne oprogramowanie w przyszłości dostosowane do konkretnych wymagań biznesowych. Na przykład w naszym przypadku jedna z usług jest napisana w języku Python, jako że uznano, iż tak będzie bardziej wydajnie, a przede wszystkim usługa powstanie o wiele szybciej niż w przypadku użycia języka Java.

4.3.2. Wybór metodyki dla konkretnego modułu

Doświadczeni programiści wiedzą, że nie ma jednego uniwersalnego sposobu implementacji, który by pasował do każdego wymagania stawianego przez klienta. Niektóre moduły posiadają bardzo złożoną logikę biznesową, z rozbudowanymi procesami i długą listą reguł walidacyjnych. W takich rozwiązaniach dobrze sprawdzi się Domain Driven Design•, ale to samo podejście w przypadku prostego systemu, jak wskazują nasze doświadczenia, będzie nadmiarowe. Niepotrzebnie zajmie więcej czasu i narazi klienta na większe koszty. W systemie POL-on 2 zdecydowano, że najbardziej złożone moduły systemu, oraz takie, które będą narażone na najczęstsze zmiany w przyszłości, zostaną napisane w Domain Driven Design. W pozostałych modułach, w zależności od poziomu ich skomplikowania i preferencji zespołu developerskiego, zostały użyte mniej pracochłonne rozwiązania, takie jak metodyka dokumentowa, trzy- i dwuwarstwowa. Oczywiście nawet krótkie ich opisanie daleko wykracza poza zakres tej publikacji, więc przedstawione zostanie podsumowanie doświadczeń, które zostały zebrane w trakcie implementacji dotychczasowych modułów systemu POL-on. Należy podkreślić, że są to subiektywne odczucia osób biorących udział w tworzeniu systemu i jako takie mogą być obarczone błędami wynikającymi z uwarunkowań w jakich był tworzony dany moduł, oraz osobistych preferencji.

DOMAIN DRIVEN DESIGN

Implementacja systemu w DDD jest złożona, głównie ze względu na konieczność rygorystycznego pilnowania granic kontekstów biznesowych. Dawała jednak programistom poczucie satysfakcji z wykonanej pracy, a dobra organizacja kodu, umożliwiła zarówno pisanie jak i utrzymanie dobrej jakości testów. DDD jest zbiorem zaleceń i zostawia dość dużą swobodę interpretacyjną co w naszym przypadku doprowadzało do długich dyskusji nad wyborem optymalnego rozwiązania. Jednak w miarę kształtowania się wewnętrznych standardów postępowania, ten problem stopniowo zanikał. Należy również zwrócić uwagę, że jest to podejście dość rozbudowane i jego poznanie, tzw. próg wejścia, szczególnie w przypadku mniej biegłych programistów początkowo może stanowić problem.

Domain Driven Design jest metodyką nastawioną na dokładne odzwierciedlenie procesu biznesowego występującego w danym podmiocie. Z uwagi na to, że system POL-on obsługuje bardzo wiele podmiotów, z których każdy charakteryzuje się swoją specyfiką i posiada własny proces biznesowy, więc z założenia nie mógł zostać zamodelo-

wany proces pasujący do każdego z nich. Zaprojektowane operacje biznesowe, mogły nie występować w części podmiotów lub mieć znacząco inną postać. Spowodowało to trudności w stosowaniu zaprojektowanego API i skłoniło część instytucji do rezygnacji z API na rzecz wykorzystania funkcjonalności importowania danych w formacie XML.

Niewątpliwie był to najbardziej pracochłonny sposób implementacji, ale w jego wyniku otrzymaliśmy dobrej jakości produkt przygotowany do dalszych modyfikacji.

TECHNIKA DOKUMENTOWA

W trakcie konsultacji z uczelniami, zwrócono nam uwagę, że dla uczelni pewną trudnością jest osiąganie docelowego stanu synchronizacji systemu uczelnianego z POL-on za pomocą szeregu atomowych operacji typu: „dodaj osobę", „dodaj zatrudnienie", „zmień nazwisko" itp. Zdecydowanie wygodniejsze jest przekazywanie aktualnego stanu danego rekordu. Dlatego też dla wielu z nich korzystanie z możliwości importowania danych w formacie XML jest lepszym rozwiązaniem niż korzystanie z rozbudowanego API. Analogiczny problem pojawiał się u nas, gdy stan danych zapisany w XML musieliśmy przemapować na poszczególne operacje biznesowe, które doprowadzą zapisy w POL-on do tego stanu. Dlatego też, w ramach eksperymentu, część modułów postanowiono zrealizować w sposób dokumentowy – aby sprawdzić na ile takie rozwiązanie będzie się sprawdzało w codziennej praktyce i z jaką reakcją użytkowników to się spotka.

Pod pojęciem „dokumentu" przyjęto dowolny zbiór pól niezbędny do gromadzenia w ramach danego modułu. Był to więc „wirtualny" dokument, który nie miał swojego odpowiednika w rzeczywistości – nie był to np. „wniosek o przyjęcie na studia", czy „umowa o pracę". Podyktowane to było tym, że każda uczelnia może mieć inny wzór i liczbę dokumentów, a system POL-on może gromadzić tylko te dane, które są wymagane przez prawo.

Przy okazji podejścia dokumentowego zdecydowano się również na zmianę sposobu walidacji danych. W systemie POL-on 1 i w dużej liczbie modułów POL-on 2 stosowana była walidacja „restrykcyjna". System nie pozwalał wprowadzić niepoprawnych danych – blokując daną operację. W architekturze dokumentowej POL-on wprowadzono pojęcie „stanu dokumentu", który może być „poprawny" lub „niepoprawny". Umożliwiło to użytkownikowi zapisywanie np. częściowo wypełnionego formularza. Oczywiście niepoprawny dokument nie jest poddany dalszemu przetwarzaniu, stanowi on jedynie formę brudnopisu dla uczelni. Po każdej zmianie dokument jest walidowany i użytkownik natychmiast otrzymuje informację o jego stanie i ewentualną listę błędów. Takie rozwiązanie zostało bardzo przychylnie przyjęte przez użytkowników.

Od strony implementacji ta technika również jest dużo prostsza. Zamiast implementować wiele metod i tworzyć rozbudowane API, powstają jedynie metody „zapisz dokument", „pobierz dokument". Programiści mogą się skoncentrować wyłącznie na zapewnieniu odpowiedniej walidacji reguł biznesowych. Uczelnie, które nie chcą korzystać z interfejsu użytkownika, również docenią proste API, które umożliwia im przesy-

łanie stanu analogicznie do importowania danych za pomocą XML, ale ze wszystkimi zaletami integracji po REST. By utrzymanie historii zmian było możliwe, każda zmiana dokumentu powoduje zapisanie jego nowej wersji – co stwarza możliwość łatwego udostępnienia użytkownikowi funkcjonalności przywrócenia poprzedniej wersji. Oczywiście takie podejście generuje znaczną ilość redundantnych danych, co zauważają również autorzy w [15], ale obecnie, gdy ceny nośników systematycznie spadają, nie jest to duży problem. Kolejną sprawą, na którą warto zwrócić uwagę, jest wysokie prawdopodobieństwo zmian merytorycznych w dokumencie. Wymuszą one dostosowanie kodu do nowej wersji, przy konieczności zachowania obsługi wersji archiwalnych.

POZOSTAŁE SPOSOBY IMPLEMENTACJI

Zastosowanie w tej chwili głównie wspomnianych powyżej architektur nie oznacza, że w POL-on 2 nie znajdą zastosowania także inne sposoby implementacji, takie jak tradycyjna architektura trójwarstwowa oparta o relacyjny model przechowywania danych, jak i nawet dwuwarstwowa. W oparciu o architekturę trójwarstwową powstał cały POL-on 1 i choć ma on swoje problemy, to jednak nie przekreśla to zastosowania tej architektury w ramach jakiegoś mikroserwisu. Architektura dwuwarstwowa zawsze może znaleźć zastosowanie w niektórych modułach wspierających i realizujących mało istotną funkcjonalność biznesową.

Podsumowując, dzięki przejściu na mikroserwisy zyskaliśmy:

- większą dostępność systemu (mniej przestojów);
- możliwość wykonywania częstszych wdrożeń;
- swobodę doboru technologii do usługi;
- optymalizację kosztów sprzętowych;
- szybsze działanie;
- większą niezależność zespołów developerskich;
- możliwość elastycznego skalowania aplikacji.

Stało się to kosztem:

- trudniejszych decyzji na etapie analizy i projektowania
- bardziej skomplikowanej architektury
- utraty „globalnych” transakcji.

Bilans korzyści dla użytkownika końcowego przewyższa potencjalne trudności po stronie wykonawcy. Dotychczasowe doświadczenia wskazują, że przyjęty model rozwoju systemu się sprawdza w praktyce i powinien być kontynuowany.

4.4. Doświadczenia związane ze standaryzacją i unifikacją procesu wytwarzania interfejsu użytkownika. Budowa własnej biblioteki komponentów

System POL-on 1 w warstwie interfejsu użytkownika wykorzystywał framework JavaServer Faces (JSF2), bazujący na języku Java stosowanym w aplikacjach Java EE. Na kluczowym etapie powstawania systemu, w roku 2011, podjęta została strategiczna decyzja o wykorzystaniu biblioteki komponentów RichFaces typu *open source* z obsługą Ajax dla JavaServer Faces, której dostawcą był JBoss, jeden z wiodących w tamtym czasie. W systemie stosowane były wtedy również inne biblioteki, takie jak konkurencyjna biblioteka PrimeFaces[51]. Decyzję podjęto po analizie tych rozwiązań pod względem ich praktyczności i stopnia przydatności do zakładanych celów. Znaczącym czynnikiem był jednak benchmark oparty o liczbę potwierdzonych przypadków użycia, liczbę zapytań w wyszukiwarce Google oraz liczbę wątków dotyczących tych haseł na forach społeczności programistów w internecie. Wybrano dominujące wówczas rozwiązanie.

Ilustracja 4.7. Popularność bibliotek w czasie
Źródło: trends.google.com/trends/explore?date=all&q=richfaces,primefaces,icefaces

Jak się później okazało, RichFaces przestał być rozwijany w czerwcu 2016 roku. Od tego czasu systematycznie tracił popularność i wsparcie społeczności. Jednak w przypadku tak rozbudowanego sytemu jak POL-on modyfikacja nie była już możliwa bez ogromnego nakładu pracy i kosztów, których poniesienie nie było uzasadnione ze względów strategicznych.

Okazją do rozwiązania tego problemu był dopiero projekt przebudowy całego systemu do nowej wersji w związku z nowelizacją prawa, czyli realizacja POL-on w wersji 2.0.

[51] rozważano również nowatorskie podówczas podejście w stylu Single Page Applications, czyli framework Google Web Toolkit (GWT) dostarczany przez Google inc.

Na tym etapie wybór podejścia, jeżeli chodzi o architekturę warstwy interfejsu użytkownika, był już dosyć oczywisty: architektura typu jedna strona (ang. *single page* (SPA)). Obecnie jest to dominujące podejście odnośnie do rozwoju aplikacji, tworzonych w oparciu o język JavaScript i uruchamianych po stronie klienta w dowolnej przeglądarce internetowej. Stanowiło to jednak co najwyżej punkt wyjścia do szeregu dalszych decyzji i doświadczeń.

Przy realizacji warstwy interfejsu użytkownika w ramach POL-on 1 ujawniły się pewne istotne cechy i problemy:

1. brak specjalizacji programistów w warstwie front-end i back-end. To ogromna zaleta, ponieważ możliwe było dostarczanie przekrojowych funkcjonalności nawet przez jedną osobę, która była w stanie tworzyć oprogramowanie przekrojowo w różnych warstwach;
2. standaryzacja komponentów UI polegała w głównej mierze na stworzeniu przykładów i dobrych praktyk do powielania w poszczególnych miejscach aplikacji (zbudowano aplikację *showcase*).

Kwestie te stanowiły kluczowy czynnik przy podejmowaniu dalszych decyzji związanych z rozbudową aplikacji typu SPA. Dla zachowania spójności interfejsu postanowiono utworzyć wspólną część kodu w postaci biblioteki komponentów interfejsu użytkownika, która później była rozbudowana o mechanizmy niezwiązane bezpośrednio z interfejsem (np. kwestie nawigacji, security, obsługi usług REST). Ostateczny rozmiar biblioteki przerósł jednak pierwotne oczekiwania, w związku z czym postanowiono podzielić ją na dwie mniejsze:

1. **OPING** – w tej bibliotece znalazło się wszystko, co w jakikolwiek sposób nie jest biznesowo związane z systemem POL-on, a mogłoby zostać użyte w przyszłości w innych projektach;
2. **POLNG** – w tej bibliotece znalazł się kod stylujący podstawowe elementy biblioteki OPING plus komponenty typowe dla logiki biznesowej POL-on (np.zestawienia, filtry, itp.).

Inspiracją do tworzenia własnej biblioteki był przegląd komponentów w Angular Material, PrimeNG oraz zapoznanie się z zaleceniami projektowania interfejsu użytkownika w projekcie VMWare Clarity. Biblioteki te stanowiły formę hermetyzacji wybranych elementów interfejsu użytkownika, tak by umożliwić ich wykorzystanie bez konieczności szczegółowej implementacji ich logiki działania oraz wyglądu. Ich celem było wprowadzenie unifikacji interfejsu użytkownika oraz automatyzacji procesów implementacyjnych. Kluczowym celem było jednak umożliwienie automatyzacji wdrożeń i aktualizacji interfejsu użytkownika bez konieczności dokonywania modyfikacji w pojedynczych ekranach. Cel ten zyskał na znaczeniu szczególnie w momencie, gdy zespoły wytwórcze musiały przystosować się do pracy w architekturze mikrousługowej.

Kolejnym argumentem motywującym wytworzenie bibliotek była chęć uniezależnienia się od zewnętrznych dostawców komponentów interfejsu użytkownika ze środowiska *open source*, po negatywnych doświadczeniach z biblioteką RichFaces. Potencjalną

korzyścią była też możliwość lepszego dostosowania do norm i standardów dostępności aplikacji dla potrzeb osób niepełnosprawnych, w ramach WAI-ARIA[52] oraz WCAG. O dostępie do aplikacji zgodnie z WCAG można dowiedzieć się więcej z pracy [33].

Na zawartość bibliotek składały się:

- zbiory atomowych, możliwych do ponownego użycia, neutralnych biznesowo komponentów interfejsu użytkownika;
- wspólne funkcjonalności niezwiązane bezpośrednio z warstwą interfejsu użytkownika (np. obsługa security);
- dostarczenie minimalnego zestawu stylów CSS-owych.

Kod 4.1. Przykład użycia kontrolki biblioteki

```
<opi-listbox #previewList="opiPreviewList" opiPreviewList
     [opiCollection]="_persons">
  <opi-option *opiPreviewListItem="let person" [value]="person">{{person.name}}
       {{person.surname}}</opi-option>
</opi-listbox>

<div *ngIf="(previewList.snapshot | async) as _person">
  <h3>{{_person.name}} {{_person.surname}}</h3>

  <dl>
    <dt>E-mail:</dt>
    <dd>{{_person.email}}</dd>

    <dt>Telefon:</dt>
    <dd>{{_person.phone}}</dd>
  </dl>
</div>
```

4.4.1. Rozwiązania techniczne

Fundamentem stworzonych bibliotek był Angular[53] 8+. Zarządcą zależności i narzędziem budującym był npm[54]. Zbudowane artefakty (paczki NPM, obrazy dockerowe aplikacji z dokumentacją) były przechowywane w lokalnym Nexusie[55]. Jako repozytorium kodu źródłowego użyto Git. Z myślą o wpisaniu się w procesy i dostępne narzędzia zdecydowano, że Jenkins będzie narzędziem do pełnej automatyzacji procesów wdrożeniowych (publikacja paczek NPM i obrazów dockerowych, uruchamianie dokumentacji na Kubernetesie) związanych z bibliotekami.

Podczas projektowania architektury bibliotek zastanawiano się nad powołaniem jednej dużej aplikacji obsługującej cały interfejs użytkownika bądź powołaniem wielu

[52] w3.org/WAI/standards-guidelines/aria (dostęp: 18.06.2021)
[53] angular.io (dostęp: 18.06.2021)
[54] npmjs.com (dostęp: 18.06.2021)
[55] sonatype.com/nexus/repository-oss (dostęp: 18.06.2021)

mniejszych aplikacji na moduł funkcjonalny. W końcu wybrano drugą opcję, czyli podejście mikrousługowe (warto nadmienić, że absolutnie nie jest to podejście mikrofrontendowe przedstawione w [18]).

W toku powstawania biblioteki okazało się, że kluczową rolę odgrywa tu dokumentacja projektowa. Dokumentacja ta jest przeznaczona głównie dla programistów. W jej skład wchodzi opis poszczególnych modułów z podziałem na kategorie. Pierwszą kategorią jest opis funkcji danego modułu, przeplatany fragmentami kodu źródłowego lub działającymi komponentami biblioteki. Kolejnym elementem jest API – opis kodu źródłowego z informacjami na temat przeznaczenia poszczególnych jego kawałków (klasy, interfejsu, funkcji, itp.). Wspomniane API jest generowane automatycznie z kodu źródłowego na podstawie komentarzy JSDoc[56]. Ostatnim elementem dokumentacji są przykłady – prezentacja działających komponentów modułu z możliwością podglądu ich kodu źródłowego.

W toku rozwoju bibliotek związanych z interfejsem użytkownika można było zidentyfikować następujące zalety takiego rozwiązania:

1. niezależność wdrożeń – wdrożenie warstwy interfejsu użytkownika jest z punktu widzenia użytkownika końcowego zupełnie niezauważalne;
2. większa niezależność zespołów deweloperskich – zmiany wprowadzane przez jeden zespół nie wpływają na inne zespoły;
3. mniej kodu do utrzymania niż w monolicie – programista zajmuje się zmianami tylko w jednym module, w związku z czym nie widzi innych części;
4. możliwość dynamicznego podłączania i odłączania modułów.

Jednak były też wady:

1. rozwój nowej wersji biblioteki oznacza zmianę zależności w każdym module, który używa poprzedniej wersji;
2. utrzymanie spójności w sposobie implementacji wzorców UX (ang. *user experience*), które powinny być jednakowe w całym systemie;
3. zmiana bazowych, krytycznych funkcjonalności systemu (np. dodanie wymaganego elementu na belce aplikacji) wymaga podbicia używanych wersji bibliotek we wszystkich modułach.

4.5. Monitoring aplikacji jako metoda jej optymalizacji

Wdrożenie aplikacji do serwera produkcyjnego nie kończy procesu rozwoju oprogramowania. Dopiero po rozpoczęciu używania aplikacji użytkownicy mogą odkryć te miejsca systemu, które były niepoprawnie zaprojektowane, wykonane bądź zbyt słabo zoptymalizowane. Mając na uwadze to, że nie każdy użytkownik od razu zgłosi taki problem,

[56] jsdoc.app/about-getting-started.html (dostęp: 18.06.2021)

a duża część stwierdzi, że aplikacja nie nadaje się do pracy, więc nie będzie jej używać, warto posiadać mechanizm do wychwytywania takich sytuacji, by wszystko szybko naprawić. Z doświadczenia wynika, że dopóki aplikacja działa poprawnie, monitoring jest niezauważalny, a można by wręcz powiedzieć, że wprost niepotrzebny. Z drugiej strony doświadczenie wskazuje również, że bez systemu monitoringu nie wiemy czy system w ogóle działa i czy jego działanie jest poprawne. Zatem monitoring jest potrzebny przez cały czas życia aplikacji.

4.5.1. Czym jest monitoring?

W słowniku języka polskiego pod pojęciem monitoringu[57] możemy znaleźć dwie definicje. W tym opracowaniu skupiono się na jednej z nich, mianowicie „stała obserwacja i kontrola jakichś procesów lub zjawisk", którą uzupełniono takim oto stwierdzeniem ze słownika angielskiego[58]: „w celu odkrycia czegoś związanego z procesem". Dlatego też pod pojęciem monitoringu możemy rozumieć:

- zbieranie metryk z aplikacji;
- wysyłanie alertów w momencie awarii;
- wizualizację (i agregację) zebranych danych;
- eksploracje zebranych metryk (także w różnych przekrojach);
- szybsze znajdowanie przyczyn awarii;
- predykcje wystąpienia awarii.

Na wyżej wymienione wyzwania względem systemu monitoringu nakłada się też zwiększenie wolumenu przetwarzanych danych wraz z koniecznością ich szybkiego przetworzenia. Należy podkreślić, iż monitorowanie nie może i nie powinno wpływać na wydajność monitorowanej aplikacji. Rozwiązania wspierające procesy monitoringu muszą zapewniać m.in. interoperacyjność, niezawodność oraz wysoką wydajność. We współczesnym świecie narzędzia do monitoringu muszą także zapewniać wysoki poziom automatyzacji, aby sam proces akwizycji metryk, ich prezentacji i wizualizacji był możliwie szybki i łatwy.

Obecnie monitoring jest sposobem radzenia sobie z wszechobecną mnogością usług i skomplikowaniem systemów informatycznych. Monitoring powinien objąć ważne obszary naszego biznesu, począwszy od infrastruktury sieciowej, przez serwery, kontenery, aplikacje, aż do odczuć użytkownika końcowego (wydajność systemu) pracującego w naszej aplikacji. Niemal każdy system monitoringu bazuje na metrykach i wartościach progowych, przy których występują alarmy i wysyłane są powiadomienia. Przykładowe narzędzia, które mogą nam pomóc w monitorowaniu (nie przedstawiono wszystkich narzędzi dostępnych na rynku), to:

[57] sjp.pwn.pl/slowniki/monitoring.html (dostęp: 18.06.2021)
[58] dictionary.cambridge.org/pl/dictionary/english/monitoring (dostęp: 18.06.2021)

- CloudWatch,
- Prometheus,
- Grafana,
- Zabbix,
- Icinga,
- Telegraf,
- InfluxDB,
- Kapacitor,
- Graylog.

Opracowanie to nie stawia sobie za cel opisywania podobieństw czy różnic wśród narzędzi do monitorowania. Zwykle nie stosuje się tylko jednego narzędzia do monitorowania, a pewnej kombinacji narzędzi. Każde z nich pokrywa swoim zakresem jakiś wycinek monitoringu, np. Graylog – zbiera logi z aplikacji i umożliwia ich przeglądanie w jednym miejscu, Grafana potrafi wizualizować zebrane metryki, a Prometheus zbiera metryki. Obecnie w ekosystemie POL-on używa się narzędzi Graylog, Prometheus, Grafana, Zabbix. Użycie takich narzędzi jest podyktowane przyzwyczajeniami i doświadczeniami. Zabbix pojawił się najwcześniej, jako że zespół posiadał już doświadczenie z tym narzędziem, Graylog – bo istniała potrzeba posiadania jednego miejsca do przeglądania logów, Prometheus po to, aby zbierać metryki z aplikacji wdrożonych na Kubernetesie.

4.5.2. Co monitorować?

Jak już wspomniano, monitorujemy biznes, a mówiąc dokładnie za [32] – następujące obszary:

- obszar infrastruktury;
- obszar aplikacji (wraz z wydzielonym w [32] obszarem oprogramowania middleware);
- obszar biznesu.

OBSZAR INFRASTRUKTURY

W tej sferze monitorujemy:

- użycie procesora;
- użycie pamięci RAM (dostępna, użyta, ogólnie);
- ilość wolnego miejsca dysku HDD;
- liczba operacji wejścia/wyjścia na dysku (IOPS);
- przepustowość sieci;
- liczba procesów czekających na CPU;
- liczba zalogowanych użytkowników;
- ilość czasu od uruchomienia (ang.*uptime*);
- liczba otwartych plików.

Analiza tych danych pomaga w wyszukiwaniu wąskich gardeł aplikacji, np. kiedyś zmierzono się z problemem przepustowości łącza internetowego. Sytuacja ta zbiegła się z udostępnieniem jednego z systemów produkcyjnych, w związku z czym błędnie założono duże zainteresowanie nową funkcjonalnością. Dzięki zestawieniu wielkości ruchu przychodzącego do bramy sieciowej z ruchem przychodzącym do aplikacji zauważono, że założenie jest zupełnie nieuprawnione. Z dokładnej analizy wynikało, że nowa funkcjonalność rzeczywiście cieszyła się zainteresowaniem, ale nie tak dużym, żeby powodować problemy z przepustowością łącza. Kolejnym przypadkiem, z jakim się zmierzono, było ograniczenie wynikające z limitu liczby otwartych plików w systemie. Wielu użytkowników chciało dokonać zmian w systemie, aby dotrzymać terminu wynikającego z przepisów prawa. I okazało się, że pula połączeń nie może nawiązać nowego połączenia do bazy danych. Zbieranie danych przedstawionych w 4.5.2 ma pomóc potwierdzić hipotezę, iż problem występujący w aplikacji jest związany ze sprzętem, lub w prosty sposób odrzucić ją.

OBSZAR APLIKACJI

W obszarze tym podstawowym parametrem jest dostępność aplikacji. Zatem interesuje nas:

- liczba zapytań HTTP na sekundę;
- czy aplikacja posiada poprawne połączenia do systemów zewnętrznych np. baza danych, Elasticsearch;
- czy pula połączeń do bazy danych nie wyczerpała się;
- ilość pamięci dostępnej dla aplikacji;
- stosunek poprawnych odpowiedzi HTTP do błędnych;
- percentyle opóźnień aplikacji.

Monitorowanie aplikacji zazwyczaj spoczywa na barkach osób, które nie brały czynnego udziału w procesie ich rozwoju. Dlatego monitorowanie „stosunku poprawnych odpowiedzi HTTP do błędnych" jest bardzo ważne, i to w kontekście każdej usługi (ang. *endpoint*). Każda usługa może wygenerować błędną odpowiedź, tym niemniej posiadając procentowy udział poprawnych odpowiedzi do błędnych jesteśmy w prosty sposób w stanie stwierdzić, czy dana usługa działa poprawnie.

Jednym z najczęstszych wymagań niefunkcjonalnych zgłaszanych przez klienta jest maksymalny czas odpowiedzi na akcję użytkownika. Dlatego monitorowanie opóźnień aplikacji zasługuje na więcej uwagi. Systemy produkcyjne powinny zapewniać odpowiedni poziom opóźnień, by użytkownik nie doświadczał dyskomfortu wynikającego z oczekiwania na działania systemu. Mierzenie czasów odpowiedzi jako średnich nie jest dobrym pomysłem, gdyż ta miara potrafi ukryć istotę możliwych problemów. Obecnie ekosystem POL-on monitoruje percentyle opóźnień. Jest to miara pokazująca poziom opóźnień, których doświadcza dany odsetek użytkowników. Np. p95, p99, p999 – oznacza kolejno poziom opóźnień, których doświadcza 95%, 99% i 99,9% użytkowników. Dla przykładu p95 na poziomie 200 ms i p99 na poziomie 700 ms mówi o tym,

że 95% użytkowników otrzymuje odpowiedź po 200 ms, a 99% czeka 700 ms. Oznacza to, że 4% użytkowników naszej aplikacji doświadcza opóźnień przez wąskie gardła w aplikacji. Dla kompletności tego opracowania należy wspomnieć o mierze Apdex[59]. To miara, która w sposób numeryczny potrafi ocenić stopień zadowolenia użytkowników. Skala tej miary mieści się w przedziale <0;1>, gdzie 0 oznacza brak zadowolonych użytkowników a 1 oznacza pełne zadowolenie wszystkich. Ekosystem POL-on mierzy zadowolenie użytkowników za pomocą tej miary.

W toku prac związanych z monitoringiem aplikacji używano również systemu JavaMelody, zintegrowanego z aplikacją. Przy użyciu tej aplikacji były wyszukiwane zbyt wolne zapytania do bazy danych oraz niepoprawne mapowania hibernate powodujące wycieki pamięci. Obecnie ekosystem POL-on przenosi się do rozwiązań chmurowych w związku z czym głównym narzędziem do zbierania metryk jest Prometheus, a głównym narzędziem służącym do zbierania i analizy logów jest Graylog.

OBSZAR BIZNESU

W obszarze tym powinno się monitorować biznes. Każda aplikacja ma swój specyficzny sposób pracy, dlatego trudno podać tu sensowne reguły. Powinno się zbierać informacje odnośnie do normalnej pracy systemu. Podczas używania zewnętrznego dostarczyciela autoryzacji warto monitorować nieudane próby logowania. W przypadku nieudanego logowania monitoring w obszarze infrastruktury nie wyśle alertu, bo zapomnienie hasła przez użytkownika to typowy przypadek. Aplikacja nie zezwoli takiemu użytkownikowi na dostęp, ale problemem może nie być podawanie błędnego hasła, a błędne działanie usługi autoryzacyjnej. W związku z tym monitorujemy liczbę udanych zalogowań i liczbę nieudanych zalogowań. Dzięki temu możemy odkryć np. próbę przełamania naszych zabezpieczeń przez atak typu *brute force*. Innym przykładem może być monitorowanie tego, jak użytkownicy korzystają z poszczególnych funkcji systemu, np. jakich filtrów w zestawieniu używają, a jakich nie. To źródło wiedzy dla optymalizacji zapytań i zbierania doświadczeń przy projektowaniu innych modułów.

4.6. Integracja z hurtownią danych

W każdym systemie możemy wydzielić dwa główne cele zbieranie danych oraz udostępnianie danych. Poniżej przedstawiono doświadczenia związane z drugim celem, czyli udostępnianiem danych.

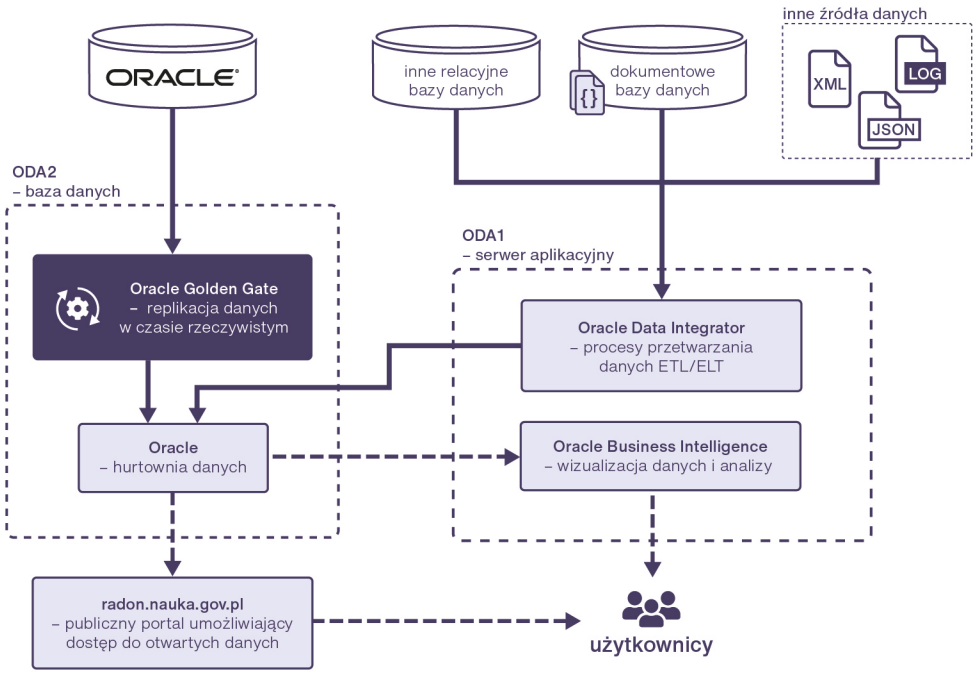
W toku ewolucji systemu POL-on zauważono, iż warstwa udostępniania danych dla użytkowników wymaga o wiele więcej uwagi, a przede wszystkim więcej zasobów niż zakładano na początku projektu. Wymagania co do raportów w takim ekosystemie nie

[59] www.apdex.org (dostęp: 18.06.2021)

były znane przed jego powstaniem, dlatego też duża ich część była tworzona dopiero na konkretne zamówienie interesariuszy bardzo często przez programistów, którzy tworzyli je na podstawie swojej wiedzy, oraz surowych danych z relacyjnej bazy danych. W związku z pracą na surowych danych (ang. *raw data*) raporty prezentujące te same dane, ale w różnych przekrojach dość często się ze sobą nie zgadzały. W pewnym momencie ilość przekrojów raportów i ich wersji zaczęła przekraczać możliwości zespołów programistycznych. Ważnym aspektem technicznym, była konieczność wykonywania wszystkich prac analitycznych typu raporty cykliczne, analizy, raporty dla użytkowników systemu, z poziomu produkcyjnej instancji bazy danych. Powodowało to dodatkowe oraz niepotrzebne obciążenie systemu produkcyjnego, związane z pracami analityków danych. Takie podejście wynikało z braku w ówczesnym czasie hurtowni danych. Główną przewagą hurtowni danych nad systemem transakcyjnym jest, możliwość wstępnego przygotowania danych do analizy z wykorzystaniem procesów przetwarzania danych (ang. *ETL – Extract Transform Load*). Dane w hurtowni danych są już odpowiednio ustrukturyzowane, oczyszczone oraz podzielone na obszary tematyczne (ang. *data marts*). Analiza danych bezpośrednio z poziomu bazy danych systemu transakcyjnego często wiąże się z koniecznością przygotowania indywidualnych procesów przetwarzania danych. Wynika to z konieczności wydobycia odpowiedniej porcji danych, następnie oczyszczenia jej oraz przygotowania do analizy. Są to typowe procesy, które obsługuje hurtownia danych, a analizując dane bezpośrednio w bazie danych systemu transakcyjnego, opisane kroki, analityk danych musiał wykonywać podczas każdej analizy. Z upływem czasu okazało się, że w rozrastającym się ekosystemie systemów informatycznych nadzorowanych przez Ministerstwo Nauki i Szkolnictwa Wyższego konieczne będzie wdrożenie centralnej hurtowni danych, która byłaby w stanie połączyć dane z różnych systemów dziedzinowych, odpowiednio je połączyć, usunąć możliwe duplikaty oraz przygotować dane do sporządzania na ich podstawie poprawnych raportów, w możliwie wszystkich przekrojach.

Analizując dostępne na rynku rozwiązania analityki biznesowej (ang. *Business Intelligence*). Do implementacji hurtowni danych wykorzystaliśmy dwa serwery Oracle Database Appliance, na których zostało zainstalowane i skonfigurowane oprogramowanie bazy danych Oracle wraz z dodatkowymi narzędziami takimi jak Oracle Golden Gate oraz Oracle Data Integrator. Do budowy raportów oraz jako narzędzie do analiz *ad hoc* wybraliśmy Oracle Business Intelligence. Całość wykorzystywanego przez nas oprogramowania jest w wersji Enterprise Edition.

Hurtownia danych miała zapewnić dostęp do zintegrowanych danych z całego ekosystemu POL-on, który posiada wysoce heterogeniczne dane, składowane głównie w bazach danych Oracle, MongoDB, PostgreSQL, MS SQL. Systemy transakcyjne oraz bazy danych projektowane były na przestrzeni 10 lat, dlatego można zauważyć różne, skrajne podejścia do przechowywania danych. Z jednej strony dysponujemy systemami z wysoko znormalizowaną strukturą danych, a na przeciwległym biegunie mamy dane przechowywane w postaci obiektowej w formacie JSON. Różne formaty i struktury danych generowały olbrzymie problemy podczas przekrojowych, interdyscyplinarnych analiz danych.



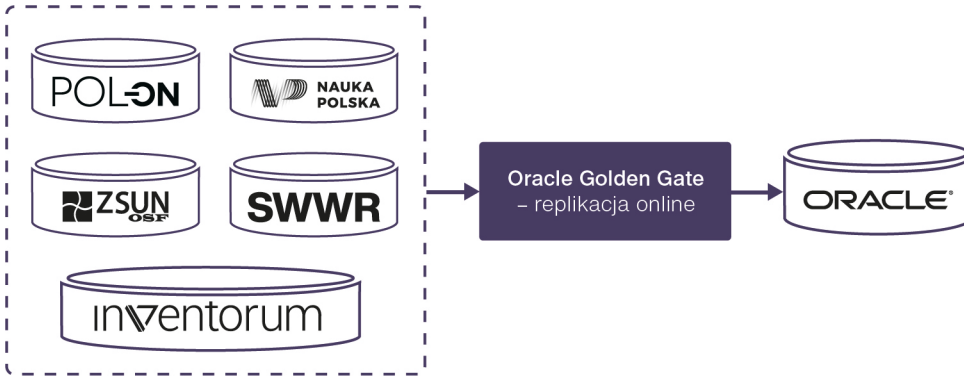
Ilustracja 4.8. Architektura hurtowni danych

4.6.1. Jezioro danych – miejsce, gdzie spływają wszystkie dane

W architekturze naszej hurtowni danych przewidzieliśmy dedykowane miejsce na przechowywanie nieprzetworzonej kopii danych z integrowanych systemów dziedzinowych – jest to tzw. jezioro danych (ang. *data lake*). Implementując tę funkcjonalność otrzymaliśmy w jednym miejscu w organizacji kopię danych z kluczowych systemów, z których można czerpać wiedzę bez obciążania systemów dziedzinowych. W znacznym stopniu zmniejszyło to obciążenie systemów produkcyjnych zadaniami niezwiązanymi z głównym celem istnienia danego systemu. Gromadzone dane wykorzystywane są do integracji pomiędzy systemami wewnętrznymi, zewnętrznymi oraz są źródłem do dalszych procesów przetwarzania w hurtowni danych.

Przetwarzając dane z heterogenicznych źródeł musieliśmy wdrożyć kilka sposobów replikacji danych z systemów produkcyjnych. Do replikacji systemów opartych o bazę danych Oracle wykorzystujemy dedykowane narzędzie Oracle Golden Gate. Dzięki wykorzystaniu tego oprogramowania mamy replikację w czasie rzeczywistym, bez obciążania systemów produkcyjnych. Dane wprowadzane przez użytkowników systemów

domenowych są dostępne w hurtowni danych z opóźnieniem rzędu milisekund. Dzięki temu możemy wykonywać analizy na aktualnych danych, wdrażać algorytmy weryfikacji oraz wykrywania anomalii w danych.

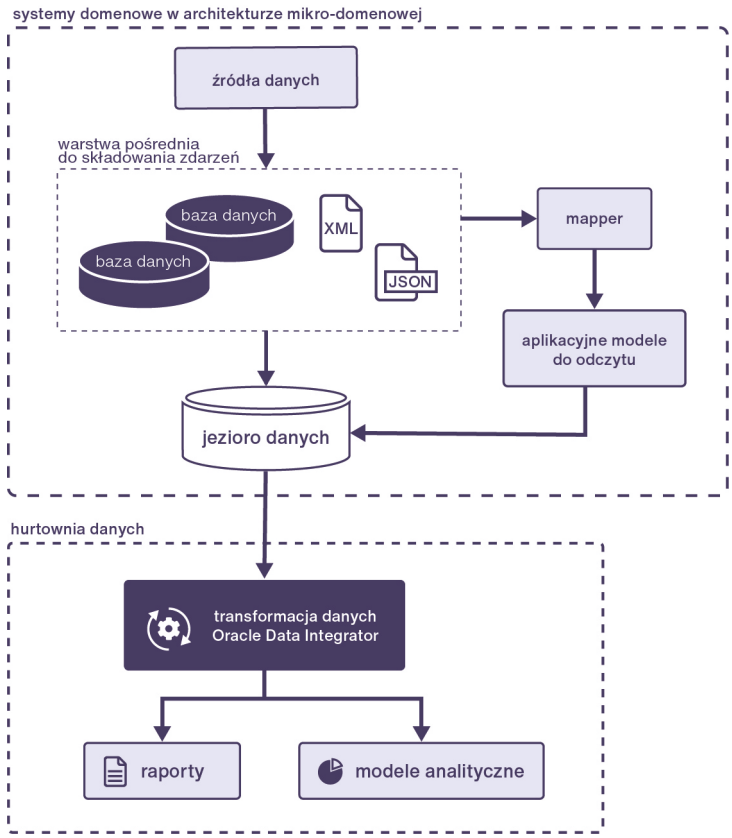


Ilustracja 4.9. Integracja baz danych Oracle

Podążając za światowymi trendami w branży IT część systemów częściowo bądź całkowicie zrezygnowała ze składowania danych w tradycyjnej relacyjnej bazie danych[60]. Posiadając duże zbiory danych w bazach relacyjnych oraz wykształcone przez lata kompetencje SQL-owe zdecydowaliśmy się na transformację danych z modelu obiektowego na relacyjny. Dzięki zastosowaniu transformacji danych przechowywanych w różnych formatach obiektowych np. JSON, XML uzyskaliśmy możliwość przekrojowego analizowania danych z całego ekosystemu z wykorzystaniem już posiadanych narzędzi i kompetencji.

Zaimplementowaliśmy dwa scenariusze przetwarzania tego typu danych, pierwszym z nich są dedykowane tylko do odczytu modele danych implementowane i utrzymywane przez zespoły deweloperskie systemów transakcyjnych. W tym przypadku cały ciężar analizy i przetwarzania danych przeniesiony został na programistów systemów domenowych, a efektem ich pracy są obiekty w bazie relacyjnej będące odwzorowaniem logiki biznesowej realizowanej w aplikacji. Utworzone w ten sposób modele danych są integralną częścią aplikacji, jednak są całkowicie odseparowane od części transakcyjnej. Drugim wykorzystanym w naszym przypadku rozwiązaniem jest zapisywanie modelu danych w postaci dokumentu JSON, a następnie przetwarzanie przez mechanizmy ETL w hurtowni danych. W tym przypadku ciężar analizy i przetwarzania dokumentu w całości odbywa się po stronie hurtowni danych.

[60] takim rozwiązaniem były opisane w poprzednich podrozdziałach podejścia typu eventsourcing oraz cqrs



Ilustracja 4.10. Integracja z systemami w architekturze mikrousługowej

Pozostałe źródła danych takie jak inne bazy relacyjne (MySQL, PostgreSQL) lub pliki z danymi integrujemy z wykorzystaniem narzędzia Oracle Data Integrator, które umożliwia implementację pełnego procesu ETL/ELT z dowolnego źródła danych do naszej hurtowni danych.

4.6.2. Wpływ wdrożenia hurtowni danych na obciążenie systemów produkcyjnych

Opisany w Rozdziale 2 proces sprawozdawczości statystycznej GUS realizowany jest w zintegrowanym środowisku POL-on od 2016 roku. Wtedy to uczelnie oraz instytuty rozpoczęły uzupełnianie sprawozdań S-10, S-11, S-12, Sprawozdania Mobilności w systemie POL-on. Wcześniej formularze te były składane albo w formie papierowej, albo poprzez portal statystyki publicznej Głównego Urzędu Statystycznego. Wprowadzenie rozwiązania informatycznego znacznie usprawniło i uprościło cały proces z punktu widzenia użytkownika końcowego. Elektroniczne formularze były już wstępnie wypełnio-

ne, a rola użytkownika końcowego ograniczała się do uzupełnienia danych, których w systemie POL-on nie było. Jednak zaimplementowane rozwiązanie było bardzo kosztowne z punktu widzenia rozwoju oraz utrzymania całego procesu, a co gorsza w znaczny sposób obciążało system produkcyjny POL-on, czyli finalnie utrudniało pracę pozostałym użytkownikom systemu. Wszystkie problemy związane z wydajnością wynikały ze złożoności procesu, wykorzystanych technologii oraz ograniczeń jakie występowały w ówczesnym czasie w środowisku POL-on (m.in. brak centralnej hurtowni danych, która powstała w 2019 roku).

W 2016 roku implementacja całego procesu została oparta o opisany w następnym podrozdziale mechanizm loga technicznego Envers. Jego działanie polega na odkładaniu każdej zmiany na tabeli w bazie relacyjnej do osobnej tabeli jako kopii danych z pełną datą zmiany oraz informacją o rodzaju zmiany (tj. dodanie danych, modyfikacja danych oraz usunięcie danych). Powstaje w ten sposób pełna historia zmian obiektu, jednak w związku z dużą ilością danych powstałe obiekty są bardzo duże (nawet do 2-3 razy więcej danych), a co za tym idzie ich odczyt jest kosztowny pomimo zastosowania indeksów.

Tabela 4.4. Wzrost liczby przechowywanych danych w systemie produkcyjnym oraz logu technicznym na podstawie liczby wierszy

Encja danych	Aktualna	Log techniczny	Zmiana %
ECTS	22 821 784	42 037 540	184,20 %
SEMESTR	21 628 582	25 287 052	116,91 %
POMOC MATERIALNA	20 210 641	47 637 016	235,70 %
STUDIA	6 245 055	24 772 108	396,67 %
OSOBA	4 157 370	9 039 691	217,44 %

Zaimplementowane rozwiązanie miało działać w trybie asynchronicznym, czyli użytkownicy końcowi zlecali wygenerowanie przez system POL-on wstępnie uzupełnionych sprawozdań. Takie żądanie trafiało na kolejkę raportową w systemie POL-on i było obsługiwane zgodnie z kolejnością zleceń oraz bieżącym obciążeniem systemu. Następnym krokiem było dodanie przez użytkowników danych, których brakowało w systemie POL-on (np. studenci z wymian w ramach umów międzynarodowych, takich jak np. Erasmus). Ostatnim etapem procesu była walidacja wprowadzonych przez użytkowników danych np. czy liczba studentów kobiet nie jest większa od całkowitej liczby studentów. Takich walidacji zostało zaimplementowanych kilkaset. W przypadku braku niezgodności pomiędzy wprowadzonymi danymi oraz wyliczonymi przez system POL-on, użytkownik końcowy mógł zatwierdzić sprawozdanie i przesłać do akceptacji do odpowiedniej komórki w Ministerstwie Nauki i Szkolnictwa Wyższego.

Rozwiązanie asynchroniczne zostało wybrane głównie ze względu na pojawiające się podczas implementacji problemy. Głównym problemem była złożoność obliczeniowa procedur do wyliczania danych, które były źródłem do wstępnego wypełnienia formularzy statystyki szkolnictwa wyższego (w nomenklaturze GUS określanych jako S*).

Ze względu na bardzo dużą liczbę danych oraz skomplikowane algorytmy, całość procesu trwała, w zależności od bieżącego obciążenia systemu, od kilku do kilkudziesięciu minut. Kolejnym powodem do wprowadzenia przetwarzania asynchronicznego, bezpośrednio powiązanym z opisanym wcześniej problemem, była ograniczona możliwość równoczesnego przetwarzania danych. Produkcyjna baza POL-on, mogła jednocześnie obsługiwać do 12 procesów równocześnie, wynikało to z liczby posiadanych licencji Oracle na produkcyjną bazę POL-on. Wykorzystywana konfiguracja systemu produkcyjnego umożliwiała przetwarzanie 12 formularzy w tym samym czasie, przy założeniu, że system jest przeznaczony tylko do sprawozdawczości GUS. Dlatego, by umożliwić normalną pracę pozostałym użytkownikom systemu, zdecydowano się na wydzielenie dedykowanej puli połączeń do bazy danych dla modułu GUS, co było pewnego rodzaju kompromisem pomiędzy umożliwieniem pracy z systemem pozostałym użytkownikom, a zapewnieniem efektywnego przetwarzania formularzy S.

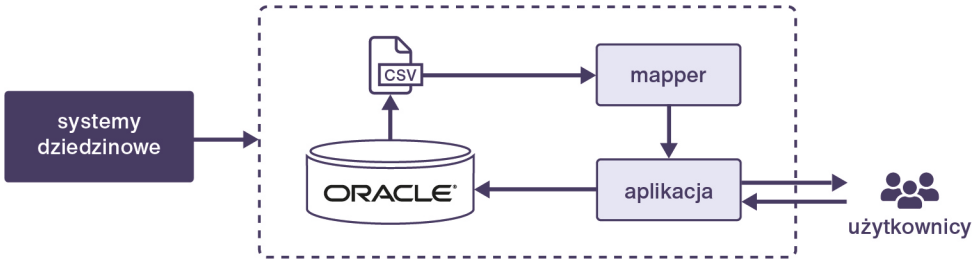
Pomimo zastosowanych prób optymalizacji procesów przetwarzania danych, podczas sprawozdawania danych przez uczelnie i instytucje występowały problemy związanie z wydajnością systemu POL-on. W okresie od stycznia do lutego kiedy trwa opisywany proces sprawozdawczy, wielokrotnie pojawiały się problemy związane z czasem oczekiwania na odpowiedz systemu. W tym czasie obciążenie bazy danych bardzo często oscylowało w okolicach 100%. Tak wysokie obciążenie wiązało się również z procesem masowego importu danych studentów i pracowników. Oba procesy rywalizowały o zasoby bazodanowe takie jak procesor, czy dostęp do dysków (zapis/odczyt).

W 2019 roku podjęto próby przeniesienia części obciążenia systemu związanego z przetwarzaniem sprawozdań do zapasowego centrum danych. Wydzielony został moduł sprawozdawczy GUS, który swoje prace związane z obliczeniami na bazie danych wykonywał w zapasowym centrum danych, czyli zupełnie niezależnym środowisku, które utrzymywane jest na wypadek awarii systemu produkcyjnego i umożliwia przełączenie całego produkcyjnego ruchu użytkowników na lustrzaną kopię systemu. Dzięki synchronizacji w czasie rzeczywistym danych pomiędzy bazą produkcyjną a centrum zapasowym dane były w obu miejscach identyczne. Wdrożona innowacja rozwiązała problem z wydajnością pozostałych modułów. Nie rozwiązało to jednak problemu z czasem potrzebnym na wyliczenie pojedynczego sprawozdania oraz łącznego czasu potrzebnego na przygotowanie danych dla wszystkich instytucji, które były zobligowane do sprawozdania swoich danych. Większość instytucji wielokrotnie poprawiała i przeliczała ponownie dane do sprawozdań, co powodowało, że w kolejce do obsłużenia czekało cały czas kilkanaście żądań. Finalnym efektem opisanego problemu było oczekiwanie od kilku do kilkunastu godzin na wygenerowanie sprawozdania. Dopiero w 2020 roku przeniesienie całego procesu do hurtowni danych rozwiązało problemy związane z wydajnością systemu. Poniżej zaprezentowany został zestaw podstawowych statystyk opisujących efektywność ówczesnego procesu przetwarzania danych.

Tabela 4.5. Statystyki liczby wygenerowanych sprawozdań, czasu przetwarzania oraz oczekiwania na przetworzenie w podziale na lata

Rok	Liczba przeliczeń	Czas przetwarzania (sek.)	Czas oczekiwania (min.)
2017	4680	583	122
2018	5205	1429	141
2019	4759	1554	89

Ważnym etapem w ewolucji procesu sprawozdawczości GUS było rozpoczęcie w 2018 roku projektu „Zintegrowany system usług dla Nauki – etap II" (ZSUN II), którego jednym z kluczowych produktów miała być centralna hurtownia danych. Pod koniec 2019 roku rozpoczęła się migracja procesów sprawozdawczych z systemu POL-on do hurtowni danych. Dla powodzenia migracji kluczowa była zmiana architektury systemu. Zdecydowano się na przetwarzanie całości danych dla wszystkich raportujących instytucji kilka razy dziennie. W zależności od potrzeb było to od 1 do 5 razy na dobę oraz generowanie gotowych do pobrania sprawozdań S. Spowodowało to, że użytkownik końcowy nie musiał, „zlecać" wygenerowania danych, tylko pobierał gotowy raport, zmiany które wprowadzał w trakcie dnia były cyklicznie przetwarzane, walidowane i finalnie aktualizowały gotowe do pobrania raporty. Całość procesu trwała kilkanaście minut, w tym czasie przetwarzane było prawie 20 milionów wierszy, które zagregowane zostały do 869 wskaźników, dla 864 instytucji na pięciu poziomach agregacji. Następnie wyliczone dane zostały zwalidowane z danymi wprowadzonymi przez użytkowników, a produktem końcowym było prawie 15 tysięcy gotowych do pobrania raportów końcowych.



Ilustracja 4.11. Architektura zastosowana w raportach dla GUS

Dodatkową korzyścią wynikającą z przeniesienia procesu do hurtowni danych było uproszczenie modelu danych, który w systemie POL-on był typowym generycznym modelem typu EAV (ang. *Entity-Attribute-Value*), dla każdego roku generowany był „nowy" model, który był optymalny dla aplikacji napisanej w języku Java, jednak już z punktu widzenia analityka danych, był dość zawiły i trudny do użycia. W hurtowni danych zaimplementowano prosty w użyciu model gwiazdy, który został zastosowany w aplikacji Oracle Business Intelligence jako interaktywny pulpit informacyjny, umożliwiający analizę danych z poziomu aplikacji, bez znajomości języków dostępu do danych typu SQL, R, Python.

Implementacja modelu danych dla sprawozdań GUS w aplikacji analitycznej klasy (ang. *business intelligence*), umożliwia w prosty sposób dowolną agregację danych oraz przeglądanie danych w różnych przekrojach.

W aplikacji Oracle BI została również udostępniona funkcjonalność umożliwiająca pracownikom Ministerstwa Nauki i Szkolnictwa Wyższego bieżącą analizę realizacji procesu sprawozdawczego przez instytucje.

Wdrożenie centralnej hurtowni danych było bardzo ważne dla uproszczenia procesów związanych ze sprawozdawczością GUS. W znacznym stopniu proces został zoptymalizowany, skrócił się czas dostępu do danych, drastycznie spadło obciążenie głównego komponentu dla całego ekosystemu nauki i szkolnictwa wyższego tj. systemu POL-on. Ważne są również wskaźniki związane z zarządzaniem projektem oraz wykorzystaniem zespołów developerskich. W znacznym stopniu skróceniu uległ czas związany z analizą problemów z danymi oraz ich naprawą. Kluczowym wskaźnikiem było również zmniejszenie zaangażowania programistów Java w utrzymanie i rozwój całego procesu. Do 2020 roku praktycznie przez 4 miesiące jeden zespół developerski zaangażowany był tylko w proces sprawozdawczości GUS, aktualnie to obciążenie zostało zminimalizowane i ogranicza się do implementacji pojedynczych zmian po stronie aplikacji POL-on.

4.7. Zarządzanie obiektami współdzielonymi na przykładzie PBN 2.0

Nowa wersja Polskiej Bibliografii Naukowej – PBN 2.0 – pomimo tego, że z prawnego punktu widzenia jest częścią POL-on, była wytworzona przez odrębny zespół oraz posiadała inne założenia technologiczne. Było to podyktowane przede wszystkim doświadczeniami zebranymi podczas pracy nad pierwszą wersją systemu, jak i problemami specyficznymi dla dziedziny publikacji naukowych, za które w środowisku POL-on odpowiedzialny jest PBN.

4.7.1. Wersjonowanie obiektów w środowisku współdzielonym

Jednym z istotnych problemów w systemach, w których dane są współdzielone pomiędzy użytkownikami, jest konieczność zapewnienia wielu użytkownikom możliwości raportowania danych odzwierciedlających ich rozumienie stanu rzeczywistości.

W pierwszej wersji systemu PBN zastosowano prosty mechanizm polegający na całkowitej separacji danych pomiędzy kontekstami użytkowników, a także na próbach łączenia ich w widokach tam, gdzie było to niezbędne. Na przykład, publikacje różnych jednostek były utrzymywane jako osobne obiekty – „zdarzenia ewaluacyjne", każde z pełnym zapisem informacji – jednak były łączone podczas wyszukiwania. Tyle że to jednak nie było niezawodne i powodowało powstawanie dużej liczby duplikatów. Powodowało to też trudność w policzeniu rzeczywistej liczby publikacji – pewna była tylko liczba zdarzeń ewaluacyjnych.

Druga wersja systemu została zaprojektowana tak, aby sprostać temu zadaniu, pozostawiając jednak pełną kontrolę nad danymi, które będą podlegały ewaluacji. Dwa podstawowe założenia, jakie zostały poczynione w fazie projektowania, były następujące:

1. publikacja, jeżeli jest identyfikowalna przez unikalny numer biznesowy, taki jak ISBN albo DOI, występuje w bazie raz;
2. dane używane w procesie ewaluacji są odrębne dla każdego podmiotu naukowego i podlegają jego pełnej kontroli.

Te dwa założenia, choć mogą wydawać się sprzeczne, są w istocie bardzo łatwe do pogodzenia – bez negatywnych skutków dla jakości danych. Zaproponowany oraz wdrożony z sukcesem w PBN 2.0 mechanizm opiera się na *Wersjonowanych Obiektach Współdzielonych* oraz *Migawkach Obiektów*.

Wersjonowany Obiekt Współdzielony to podstawowy obiekt opisujący publikację w PBN 2.0. Zawiera wszystkie informacje potrzebne do pełnego, zgodnego z rozporządzeniem opisu publikacji. Obiekt ten przechowuje zarówno same metadane publikacji (dane takie jak Tytuł czy Język publikacji), połączenia z innymi *Wersjonowanymi Obiektami Współdzielonymi* – opisującymi m.in: *Autorów* czy *Konferencje*, oraz informacje o zmianach (w postaci historycznych wersji metadanych). Obiekty te przetrzymywane są w bazie dokumentowej Mongo i połączone ze sobą poprzez relacje w relacyjnej bazie danych PostgreSQL.

Migawki Obiektów zawierają zamrożony w czasie obraz *Wersjonowanego Obiektu Współdzielonego* wraz ze wszystkimi obiektami z nim połączonymi. Migawki te są niezmieniane i powiązane z konkretnym użytkownikiem lub kontem konkretnej instytucji. Przetrzymywane są wraz z informacjami o powiązaniu w bazie relacyjnej PostgreSQL jako dane zserializowane. Taki rozdział pozwala na spełnienie naraz założeń o obiektach współdzielonych – które edytowane są przez wszystkich, co doprowadzi w ostateczności do poprawy jakości danych – jak i o pełnym rozgraniczeniu danych, które uczelnia sprawozdaje w ramach procesu oceny jednostek.

Wstępna ocena przydatności tego rozwiązania po pół roku od wdrożenia produkcyjnego jest bardzo pozytywna. Użytkownicy pracują na wspólnych danych, jednak mają pewność, że dane, które poprawili, są w stanie dokładnie takim, w jakim chcieli, a procesy biznesowe związane z analizą danych znajdujących się w systemie są znacznie ułatwione.

4.7.2. Edycja obiektów a uprawnienia użytkowników

Jedną z cech odróżniających PBN 2.0 od pozostałych części systemu POL-on jest fakt, że PBN pozwala na pracę w nim nie tylko osobom związanym z jednostkami naukowymi, ale także każdemu zainteresowanemu – zarówno naukowcowi obecnemu w bazie danych POL-on, jak i przeciętnemu Kowalskiemu. System ten pod tym względem

jest bardziej podobny do Wikipedii niż do reszty środowiska POL-on – i podobnie jak we wspomnianej internetowej encyklopedii konieczne było rozgraniczenie uprawnień do edycji poszczególnych obiektów – *Publikacji*, *Czasopism* czy *Konferencji* znajdujących się w systemie.

Projektujący system zdecydowali się rozdzielić uprawnienia na 3 poziomy, wyznaczone jasno przez cele, do których dąży użytkownik systemu PBN:

- użytkownik niebędący autorem – osoba, która chce poprawiać dane, ale nie jest ani autorem, ani pracownikiem instytucji;
- autor – osoba będąca autorem publikacji;
- moderator/pracownik podmiotu naukowego – pracownik podmiotu naukowego odpowiedzialny za wprowadzanie danych i przygotowanie ich do procesu ewaluacji.

Użytkownicy na poziomie autor lub moderator otrzymali uprawnienie do „weryfikacji" prac i połączeń. Połączenia zweryfikowane mogą być edytowane jedynie przez innych użytkowników z odpowiednim poziomem uprawnień (lub przez administratorów systemu). Dodatkowo praca „dotknięta" przez osobę z uprawnieniami moderatora automatycznie staje się przez niego zweryfikowaną – co jest związane z faktem, że moderatorami są w systemie osoby, które w sposób prawny odpowiadają za jakość wprowadzonych danych i muszą mieć pewność, że ich zmiany nie zostaną nadpisane przez użytkownika o niższych uprawnieniach.

Tabela 4.6. Macierz uprawnień do edycji w systemie PBN. Tak w komórce oznacza możliwość edycji, nie jej brak. Uznanie autorstwa oznacza jedynie możliwość wskazania siebie w publikacji przy spełnieniu odpowiednich warunków

	Użytkownik	Autor	Moderator
Brak weryfikacji	Tak	Tak	Tak
Weryfikacja autora	Nie	Tak	Tak
Weryfikacja moderatora	Nie	Uznanie autorstwa	Tak

Rozwiązanie to po pół roku w naszej ocenie sprawdza się bardzo dobrze. Pozwala na pełną kontrolę dostępu do edycji do poszczególnych obiektów bez komplikowania systemu uprawnień, jednocześnie umożliwiając pracę użytkownikom o niskich uprawnieniach w roli osób dodających dane, które następnie mogą być poprawione przez osoby wyznaczone do tego przez instytucje.

4.7.3. Import danych z systemów historycznych do PBN 2.0

Jednym z największych wyzwań jakie związane były z wdrożeniem systemu PBN 2.0 była ogromna ilość danych przetrzymywanych w systemach historycznych – PBN-R oraz PBN-MS. PBN-R był systemem stworzonym jako ogólnodostępne repozytorium,

PBN-MS natomiast zapewniał możliwość spełniania ustawowego obowiązku sprawozdawczości dla jednostek.

Dane zebrane w obu systemach stanowiły jednak bardzo dużą wartość zarówno historyczną – w przypadku danych z systemu PBN-R – jak i związaną z procesem oceny jednostek naukowych, planowanym na rok 2021 w systemie PBN-MS. Pominięcie ich nie wchodziło więc w grę.

Proces importu został zaprojektowany tak, aby w jak najmniejszym stopniu utracić pracę użytkowników. Dane z systemu repozytoryjnego – o mniejszej wartości – zostały zmigrowane w sposób w pełni automatyczny, natomiast aby ułatwić użytkownikom przejście z ich aktualnymi danymi sprawozdawczymi (za które byli odpowiedzialni) do nowego systemu, utworzony został dodatkowy serwis pozwalający na przygotowanie danych do migracji poprzez poprawienie ich oraz wskazanie, które dane są ważniejsze. Było to niezbędne, ponieważ PBN-MS[61], z racji swojej budowy oraz uwarunkowań prawnych oparty był głównie na sprawozdawczości na poziomie wydziałów – PBN 2.0 trzyma dane związane z procesem oceny na poziomie podmiotów głównych (takich jak uczelnie).

System przygotowania do migracji pozwalał na pełną weryfikację tego, czy dane w starym systemie są zgodne z aktualnym rozporządzeniem (w przeciwnym wypadku blokował migrację), jak i pozwalał na wskazanie, które publikacje zostaną przeniesione – zarówno z grupy publikacji o tym samym DOI (jako że system docelowy zapewnia unikalność numeru DOI w bazie), jak z dowolnej innej publikacji.

Sam proces migracji przebiegał następująco:

1. system wybierał publikację do przeniesienia w sposób losowy, potem sprawdzał, czy publikacja o danym DOI znajduje się w bazie;
   a. jeśli publikacji tam nie było, tworzył publikację w bazie i oznaczał ją jako „utworzona w procesie importu", dodawał identyfikator z systemu historycznego do publikacji oraz tworzył obiekty powiązane;
   b. jeśli publikacja była w bazie, jedynie dodawał identyfikator z systemu historycznego do publikacji;
2. ponadto system tworzył *migawkę obiektu* dla publikacji i powiązał ją z kontem instytucji;
3. tworzył też wpisy oświadczeń dla publikacji.

Proces ten – zarówno przygotowanie jak i sam import – okazał się bardzo skuteczny. Dane użytkowników – co wynika z komunikacji z nimi – zostały przeniesione do systemu bez utraty wartości.

[61] Polska Bibliografia Naukowa – moduł sprawozdawczy

ROZDZIAŁ 5

ZAKOŃCZENIE

Opis systemu POL-on nie może być kompletny nie tylko z uwagi na jego rozmiar, ale przede wszystkim ciągłą zmienność. Przedstawione w niniejszym opracowaniu zagadnienia stanowią istotny, ale jednak niepełny obraz systemu również z uwagi na skupienie na jego wybranych elementach. Osobnego potraktowania wymagałoby chociażby przedstawienie nowych aplikacji, takich jak portal RADON oraz System Ewaluacji Dorobku Naukowego (SEDN). Nowa wersja repozytorium publikacji, czyli system PBN 2.0, na etapie pisania pracy został dopiero oddany do użytku. Jego ewolucja powinna być przedmiotem równie pogłębionych analiz z uwagi na ogromne zapotrzebowanie na tego typu rozwiązania przez środowisko naukowe oraz instytucje monitorujące i ewaluujące polską naukę. Większość zagadnień przedstawionych w pracy wymaga bardziej szczegółowego przedstawienia w ramach oddzielnych publikacji. Planowane jest ich wydanie w niedługim czasie.

Od początku roku 2021 nastąpiła istotna zmiana w strukturze instytucji, będącej właścicielem systemu. Ministerstwo Nauki i Szkolnictwa Wyższego zostało przekształcone, poprzez połączenie z Ministerstwem Edukacji Narodowej, w Ministerstwo Edukacji i Nauki. Zmiana ta może mieć istotny wpływ na ogólny kształt i cele systemu. Naturalną tendencją w takiej sytuacji jest dążenie do integracji i efektu synergii[62]. Na obecnym etapie trudno antycypować szczegółowe cele związane z dalszym rozwojem systemu, lecz pewne zdarzenia wskazujące dalszy kierunek rozwoju miały miejsce już na etapie końcowych prac nad wydaniem niniejszej pracy. Do zdarzeń takich można zaliczyć wykorzystanie systemu jako głównego źródła do rejestracji nauczycieli akademickich oraz innych osób prowadzących zajęcia na uczelniach na szczepienia przeciwko COVID-19. System odegrał kluczową rolę w tym procesie. Wymagało to bardzo szybkiego dostosowania[63] oraz zwiększenia skali jego użycia.

Szczególną wartość pracy stanowią praktyczne rozwiązania techniczne oraz metodyczne w zakresie zarządzania tak dużym projektem informatycznym. Ewolucja systemu w kierunku architektury mikrousługowej nie została jeszcze w pełni zakończona na etapie pisania pracy. Wypracowane w drodze empirycznych doświadczeń koncepcje architektoniczne, bazujące na metodzie Domain Driven Desing zostały znacząco zmodyfikowane w trakcie zdobywania kolejnych doświadczeń przez zespoły wytwórcze wraz z bardziej dogłębnym poznaniem specyfiki domeny biznesowej. Dzięki autonomii

[62] przykładem jest chociażby chęć połączenia inicjatywy badania losów absolwentów (ELA) z badaniami niższego szczebla edukacji
[63] w zaledwie kilka dni

usług udało się podejść do wielu zagadnień w sposób heterogoniczny przy jednoczesnym zachowaniu spójności integracji całego systemu na poziomie mechanizmów wymiany danych, obsługi logiki biznesowej oraz interfejsu użytkownika. Z monolitycznego systemu powstał rozległy konglomerat usług, które współpracują ze sobą i są rozwijane zupełnie niezależnie. Daje to duży potencjał na przyszłość, by system był w stanie nadążać za najnowszymi trendami i zdobyczami branży IT.

SŁOWNIK POJĘĆ

A

API <ang. Application Programming Interface> Interfejs programistyczny aplikacji. Zestaw reguł definiujący komunikację pomiędzy systemami komputerowymi oraz pomiędzy systemem komputerowym a człowiekiem

D

Devops <ang. Development and Operations> Metodyka zespolenia rozwoju i eksploatacji oraz zapewnienia jakości. Metodyka ta kładzie nacisk na ścisłą współpracę i komunikację profesjonalistów z zakresu utrzymania IT (administratorów) oraz specjalistów od rozwoju oprogramowania (programistów). Uwzględnia współzależność rozwoju i utrzymania IT

Docker Otwarte oprogramowanie służące do realizacji wirtualizacji na poziomie systemu operacyjnego – konteneryzacja. Działające jako „platforma dla programistów i administratorów do tworzenia, wdrażania i uruchamiania aplikacji rozproszonych"

Domain Driven Design <ang.Domain Driven Design> Podejście do tworzenia oprogramowania kładące nacisk na takie definiowanie obiektów i komponentów systemu oraz ich zachowań, aby wiernie odzwierciedlały rzeczywistość

Dług techniczny Pojęcie związane z realizacją projektów informatycznych. Jest to zbiór wielu różnych problemów technicznych, które pojawiają się podczas realizacji projektu informatycznego, a ich rozwiązanie (lub nie) ma wpływ na dalszy proces realizacji projektu

G

GUI <ang. Graphical User Interface> Graficzny interfejs użytkownika

I

Infrastruktura jako usługa Natychmiastowa infrastruktura udostępniana i zarządzana za pośrednictwem internetu. Usługa ta polega na dostarczeniu przez dostawcę całej infrastruktury informatycznej, takiej jak np. wirtualizowany sprzęt, skalowany w zależności od potrzeb użytkownika

K

Kanban Opracowana w Japonii w latach 50. ubiegłego stulecia metoda zarządzania produkcją. Słowo Kanban pochodzi z języka japońskiego i oznacza kartkę papieru. W wolnym tłumaczeniu znaczy „widoczny opis" <jap. Kan - widoczny, Ban - kartka papieru>

R

RESTful API <ang. Representational State Transfer, Application Programming Interface> styl architektury oprogramowania, opierający się o zbiór wcześniej określonych reguł opisujących jak definiowane są zasoby, a także umożliwiających dostęp do nich (REST). Połączony z zestawem reguł definiujących komunikację pomiędzy systemami komputerowymi oraz pomiędzy systemem komputerowym a człowiekiem (API)

Retesty Testowanie potwierdzające (retestowanie) – rodzaj testu związanego ze zmianą przeprowadzonego po naprawieniu defektu w celu potwierdzenia, że awaria spowodowana przez ten defekt już nie występuje

T

Testy funkcjonalne Testowanie wykonywane, by ocenić czy moduł lub system spełnia wymagania funkcjonalne (w odróżnieniu od testów niefunkcjonalnych, tj. testów cech takich jak użyteczność, wydajność czy bezpieczeństwo)

Testy regresji Rodzaj testowania mający na celu wykrycie, czy defekty zostały wprowadzone lub odkryte w niezmienionych obszarach oprogramowania

U

UML Notacja UML <ang. Unified Modeling Language> umożliwia tworzenie diagramów przedstawiających różne punkty widzenia systemu. Jest przeznaczona do specyfikowania modelu obiektowego systemu informatycznego oraz kontekstu, w jakim system będzie używany na wybranym poziomie szczegółowości. Według definicji UML jest językiem służącym do zapisywania projektu systemu i może być stosowany do graficznego opracowania, tworzenia, dokumentowania artefaktów podczas procesu budowy oprogramowania

UX <ang. User Experience> Projektowanie systemów w oparciu o doświadczenia użytkownika

SPIS ILUSTRACJI

SPIS TABEL

SPIS FRAGMENTÓW KODU

BIBLIOGRAFIA

[1] M. Artač, T. Borovssak, E. Di Nitto, M. Guerriero, D. A. Tamburri, *DevOps: Introducing Infrastructure-as-Code*, *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, IEEE, 2017, s. 497–498.

[2] L. Bass, I. Weber, L. Zhu, *DevOps: A software architect's perspective*, Addison-Wesley Professional, 2015, ISBN 978-0134049847.

[3] S. Bhardwaj, L. Jain, S. Jain, *Cloud computing: A study of infrastructure as a service (IAAS)*, *International Journal of Engineering and Information Technology*, 2010, 2(1), s. 60–63.

[4] A. Brandolini, *Introducing EventStorming: An act of Deliberate Collective Learning*.

[5] R. Chen, S. Li, Z. Li, *From Monolith to Microservices: A Dataflow-Driven Approach*, *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, 2017. s. 466–475.

[6] S. De Santis, L. Florez, D. V. Nguyen, E. Rosa, *Evolve the Monolith to Microservices with Java and Node*, IBM Redbooks, 2016, ISBN 978-0783442112.

[7] V. Driessen, *A successful Git branching model*, nvie.com/posts/a-successful-git-branching-model (dostęp: 18.06.2021).

[8] C. Ebert, G. Gallardo, J. Hernantes, N. Serrano, *DevOps*, *IEEE Software*, 2016, 33(3), s. 94–100, ISSN 0740-7459, DOI: 10.1109/MS.2016.68.

[9] E. Evans, *Domain-driven Design: Tackling complexity in the heart of software*, Addison-Wesley Professional, 2004, ISBN 978-0321125217.

[10] N. Forsgren, G. Kim, J. Humble, *Accelerate: The science of lean software and devops: Building and scaling high performing technology organizations*, IT Revolution Press, wyd. 1, 2018, ISBN 978-1942788331.

[11] M. Fowler, *AnemicDomainModel*, martinfowler.com/bliki/AnemicDomainModel.html (dostęp: 18.06.2021).

[12] M. Fowler, *Event sourcing*, martinfowler.com/eaaDev/EventSourcing.html (dostęp: 18.06.2021).

[13] M. Fowler, *TestPyramid*, martinfowler.com/bliki/TestPyramid.html (dostęp: 18.06.2021).

[14] M. Fowler, *Feature Toggles (aka Feature Flags)*, martinfowler.com/articles/feature-toggles.html (dostęp: 18.06.2021).

[15] P. Gómez, C. Roncancio, R. Casallas, *Towards quality analysis for document oriented bases*, *International Conference on Conceptual Modeling*, Springer, 2018. s. 200–216.

[16] J.-P. Gouigoux, D. Tamzalit, *From Monolith to Microservices: Lessons Learned on an Industrial Migration to a Web Oriented Architecture*, *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, 2017, s. 62–65.

[17] C. Hibbs, S. Jewett, M. Sullivan, *The art of lean software development: a practical and incremental approach*, O'Reilly Media, 2009, ISBN 978-0596554439.

[18] C. Jackson, *Micro Frontends*, martinfowler.com/articles/micro-frontends.html (dostęp: 18.06.2021).

[19] M. Kalske, N. Mäkitalo, T. Mikkonen, *Challenges when moving from monolith to microservice architecture*, *International Conference on Web Engineering*, Springer, 2017, s. 32–47.

[20] G. Kim, J. Humble, P. Debois, J. Willis, *The devops handbook: How to Create World-Class Agility, Reliability, & Security in Technology Organizations*, IT Revolution Press, wyd. 1, 2016, ISBN 978-1942788003.

[21] V. Klyuchikov, *Language Integrated Query as a Canonical Data Model for Virtual Data Integration*, *DAMDID/RCDL*, 2019, s. 381–394.

[22] L. Koskela, *Test Driven: TDD and Acceptance TDD for Java Developers*, Manning Publications, 2007.

[23] D. Leffingwell, *Agile Software Requirements: Lean Requirements Practices for Teams, Programs, and the Enterprise*, Addison-Wesley Professional, wyd. 1, 2011, ISBN 978-0321635841.

[24] V. Mahnic, N. Zabkar, *Measuring progress of scrum-based software projects*, *Elektronika ir Elektrotechnika*, 2012, 18(8), s. 73–76, DOI: 10.5755/j01.eee.18.8.2630.

[25] *Manifest Agile*, agilemanifesto.org (dostęp: 18.06.2021).

[26] M. Manukyan, G. Gevorgyan, *Canonical data model for data warehouse*, *East European Conference on Advances in Databases and Information Systems*, Springer, 2016, s. 72–79.

[27] T. Menzies, W. Nichols, F. Shull, L. Layman, *Are delayed issues harder to resolve? Revisiting cost-to-fix of defects throughout the lifecycle*, *Empirical Software Engineering*, 2017, 22(4), s. 1903–1935, DOI: 10.1007/s10664-016-9469-x.

[28] I. Nadareishvili, R. Mitra, M. McLarty, M. Amundsen, *Microservice architecture: aligning principles, practices, and culture*, O'Reilly Media, Inc., 2016, ISBN 978-1491956342.

[29] S. Newman, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*, O'Reilly Media, 2019, ISBN 978-1492047810.

[30] J.-w. Park, M.-W. Lee, J. Kim, S.-w. Hwang, S. Kim, *Cost-aware triage ranking algorithms for bug reporting systems*, *Knowledge and Information Systems*, 2016, 48(3), s. 679–705, DOI: 10.1007/s10115-015-0893-9.

[31] K. Schwaber, J. Sutherland, *The Scrum Guide: The Definitive Guide to Scrum: The Rules of the Game*.

[32] J. Shao, H. Wei, Q. Wang, H. Mei, *A runtime model based monitoring approach for cloud*, *2010 IEEE 3rd International Conference on Cloud Computing*, IEEE, 2010, s. 313–320.

[33] M. Skierniewska, K. Skroban, *Dostępność oprogramowania użytkowego dla osób niepełnosprawnych i starszych w świetle standardów WCAG*, *Roczniki Kolegium Analiz Ekonomicznych/Szkoła Główna Handlowa*, 2018, 50, s. 141–152.

[34] Stowarzyszenie Jakości Systemów Informatycznych, *Słownik terminów testowych ISTQB® (wersja 3.4)*, sjsi.org/download/9147 (dostęp: 18.06.2021).

[35] F. ur Rehman, B. Maqbool, M. Q. Riaz, U. Qamar, M. Abbas, *Scrum Software Maintenance Model: Efficient Software Maintenance in Agile Methodology*, *2018 21st Saudi Computer Society National Computer Conference (NCC)*, 2018, s. 1–5.

[36] C. Viegas, A. Vasconcelos, J. Borbinha, Z. Chora, *A Canonical Data Model for Records Management in the Portuguese Public Administration*, *International Conference on Enterprise Information Systems*, Springer, 2019, s. 210–249.

[37] M. Wacker, *Just Say No to More End-to-End Tests*, testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html (dostęp: 18.06.2021).

[38] K. Wiegers, J. Beatty, *Software requirements*, Developer Best Practices, Pearson Education, 2013, ISBN 978-0735679627.

[39] H. Yasar, *Experiment: sizing exposed credentials in GitHub public repositories for CI/CD*, *2018 IEEE Cybersecurity Development (SecDev)*, IEEE, 2018, s. 143.

[40] *Zarządzenie Ministra Nauki i Szkolnictwa Wyższego z dnia 21 listopada 2017 r. zmieniające zarządzenie w sprawie powołania Zespołu do spraw Zintegrowanego Systemu Informacji o Nauce i Szkolnictwie Wyższym „POL-on"*, Dz. Urz. MNiSW poz. 63.