POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Mateusz Pustułka
Nr albumu 12050

Wsparcie pracy laboratorium poprzez implementację usług w chmurze obliczeniowej

Praca magisterska napisana
pod kierunkiem:

dr inż. Mariusz Trzaska

Warszawa, czerwiec 2015

Streszczenie

Niniejsza praca przedstawia koncept na tworzenie systemów informatycznych w oparciu o chmurę obliczeniową. Dzięki wykorzystaniu odpowiednich wzorców architektonicznych możemy tworzyć aplikacje wydajne i elastyczne, a przeprowadzone badania pokazały, że najnowszych technologii nie należy się obawiać. Cały proces projektowania, zarządzania projektem, ludźmi, sprzętem oraz siecią oparto o usługi chmurowe. Dzięki wykorzystaniu wzorca Domain-Driven Design pokazano jak budować aplikacje, które w przejrzysty sposób modelują zagadnienia biznesowe. Chmura pozwoliła na przeprowadzenie badań oraz testów na środowiskach dostępnych do tej pory tylko dla dużych korporacji.

W poniższej pracy zaprezentowano wzorce architektoniczne oraz narzędzia, które umożliwiły stworzenie generycznego systemu wspomagającego pracę laboratoriów chemicznych. Prototyp został zbudowany przy aktywnym wsparciu konsultantów branżowych oraz platformy Microsoft Azure i zawiera niezbędne moduły do funkcjonowania różnej wielkości laboratoriów, spełniając przy tym rygorystyczne wymagania Polskiego Centrum Akredytacji.

Spis treści

1. Wstęp

Proces zarządzania laboratorium chemicznym jest niezwykle złożony i wymaga znacznych nakładów pracy oraz wkładu finansowego. Z jednej strony trzeba sprostać stale wzrastającym i zmieniającym się wymaganiom klientów, dbać o wysoką jakość obsługi, inwestować w rozwój nowych technologii, z uwagi na rozrastającą się konkurencję oraz zdobyć akredytację Polskiego Centrum Akredytacji, tak aby liczyć się na branżowym rynku. Z drugiej strony, należy pamiętać o efektywności i kosztach prowadzenia takiego przedsiębiorstwa, a przy tym spełnić wymogi dotyczące archiwizacji dokumentów, zatrudniać odpowiednią ilość wykfalifikowanego personelu, przechodzić co roczny audyt oraz stosować odpowiednie procedury bezpieczeństwa.

Wdrożenie systemu informatycznego, wspomagającego zarządzanie laboratorium, nie tylko usprawniłoby cały proces, ale również podniosłoby jakość obsługi klienta poprzez stworzenie dedykowanej platformy do komunikacji.

Sam pomysł, na rozwiązanie powyższych problemów zrealizowało już kilka firm na polskim rynku. Niestety żadna z nich nie brała pod uwagę wieloplatformowości proponowanego rozwiązania, a przede wszystkim rozwoju urządzeń mobilnych i skupiała się jedynie na systemie operacyjnym Microsoft Windows. Niniejsza praca ma na celu pokazanie możliwości optymalizacji czasu i pieniędzy, które oferuje nam chmura obliczeniowa, jak również wykorzystanie dowolnych urządzeń z przeglądarką internetową do zarządzania systemem. Wykorzystanie omówionego rozwiązania ma na celu zwiększenie efektywności pracy laboratoriów chemicznych, jak również ukazanie nowego oblicza aplikacji webowych i pośrednio ich wyższość w czasach internetu rzeczy

1.1. Cel pracy

Celem pracy jest analiza praktycznych aspektów wykorzystania chmury obliczeniowej podczas tworzenia systemów informatycznych oraz jej elastyczność w krytycznych dla biznesu sytuacjach. Zaprezentowane zostało również podejście *Domain-Driven Design*(DDD), do tworzenia oprogramowania, bazującego na współpracy zespołu programistycznego z ekspertami znającymi specyfikę domeny biznesowej, aby stworzyć model konceptualny, opisujący konkretny problem.

Dla lepszego zobrazowania specyfiki DDD oraz wykorzystania chmury obliczeniowej został opracowany prototyp aplikacji, która będzie przykładem współpracy z konsultantami z branży chemicznej. Przygotowane oprogramowanie ma na celu pokazanie generycznego podejścia do tworzenia systemów informatycznych, z łatwo wymienialną, modułową strukturą.

1.2. Rozwiązanie przyjęte w pracy

Prototyp systemu informatycznego jest zbudowany na platformie *Microsoft Azure* jako aplikacja webowa. Część serwerowa systemu została napisana w języku C#, natomiast warstwa prezentacyjna opiera się głównie na *JavaScript* i frameworku *AngularJS*. Środowisko programistyczne *Visual Studio 2013* od firmy *Microsoft*, zostało wykorzystane do implementacji. Przechowywanie danych było zrealizowane, przy użyciu bazy *Microsoft SQL Server 2014*. Maperem pomiędzy obiektami domeny *Plain Old CLR Object*(POCO), a relacyjną bazą danych jest *Entity Framework* w wersji 6. Biblioteka *Castle Windsor* została wykorzystana jako kontener do inwersji zależności i zbudowania systemu modułów. Część raportowa, została przygotowana w oparciu o *Microsoft SQL Server Reporting Services* (SSRS), który dzięki edytorowi *What You See Is What You Get*(WYSIWYG) w prosty sposób pozwala na tworzenie zaawansowanych raportów.

1.3. Rezultaty pracy

Rezultatem pracy jest pokazanie zalet i wad tworzenia systemów informatycznych w oparciu o chmurę obliczeniową oraz podejście *Domain-Driven Design*. Przedstawione zostały funkcjonalne aspekty korzystania z chmury obliczeniowej oraz sposoby na współpracę zespołów programistycznych z ekspertami po stronie biznesu.

Do celów badawczych został stworzony prototyp, który ma szansę wypełnić niszę nowoczesnych systemów informatycznych dla laboratoriów chemicznych na polskim rynku, oraz potwierdzić, iż koszty zarządzania przedsiębiorstwem mogą być z łatwością ograniczone dzięki cyfryzacji.

1.4. Organizacja pracy

Rozdział drugi opisuje najpopularniejsze rozwiązania istniejące na polskim rynku, dotyczące oprogramowania dla laboratoriów. Krótka charakterystyka oraz opis najważniejszych funkcjonalności, pozwoliły na lepsze zaznajomienie się z domeną.

W rozdziale trzecim autor skupił się na przedstawieniu uniwersalnej koncepcji porozumiewania się ekspertów domeny biznesowej z zespołem programistycznym. Ważnym aspektem opisanym w tym rozdziale jest również analiza skalowalności systemu informatycznego, jego modułowej budowy oraz elastyczności zaproponowanej infrastruktury.

W czwartym rozdziale zostały zaprezentowane najpopularniejsze chmury obliczeniowe, wraz z nakreśleniem specyfiki projektowania systemów informatycznych korzystając z podejścia *Domain-Driven Design*. Zamieszczono także krótkie porównanie baz relacyjnych z bazami *NoSQL* i opisano eksperyment oraz procedurę testową wyszukiwania fraz w milionowych zbiorach danych z wykorzystaniem chmury obliczeniowej przy użyciu takich systemów jak *SQL Server* czy *Redis*. Na koniec rozdziału zostały zbadane możliwości zastosowania *Continuous Integration* i publikowania nowych wersji systemu na bazie *Visual Studio Online* i *Microsoft Azure*.

W rozdziale piątym zostały opisane technologie oraz rozwiązania wykorzystane do zrealizowania pracy.

Rozdział szósty poświęcony został implementacji najważniejszych elementów prototypu wraz z opisem możliwości integracji rozwiązania z innymi systemami oraz rozbudowy o nowe moduły.

Rozdział siódmy stanowi podsumowanie całej pracy, zawierające zestawienie wad i zalet prototypu systemu wraz z opisem możliwości jego rozbudowy.

2. Przegląd istniejących rozwiązań

Rynek systemów informatycznych wspomagających prowadzenie laboratoriów chemicznych w Polsce jest niewielki, dodatkowo w większości przypadków rozwój oprogramowania zakończył się kilka lat temu. Dostępne systemy są modułowe i w ograniczonym stopniu pozwalają na personalizację. Wszystkie ściśle wiążą się z systemem operacyjnym Windows. Niestety żadna z firm nie stworzyła wersji demonstracyjnej, co zacznie utrudnia analizę porównawczą, gdyż do użytkowania wymagana jest komercyjna licencja, dostępną jedynie po zakupie oprogramowania. Informacje, które można pozyskać bezpośrednio od producenta posłużyły do stworzenia krótkiej charakterystyki.

2.1. FILAB 2.0

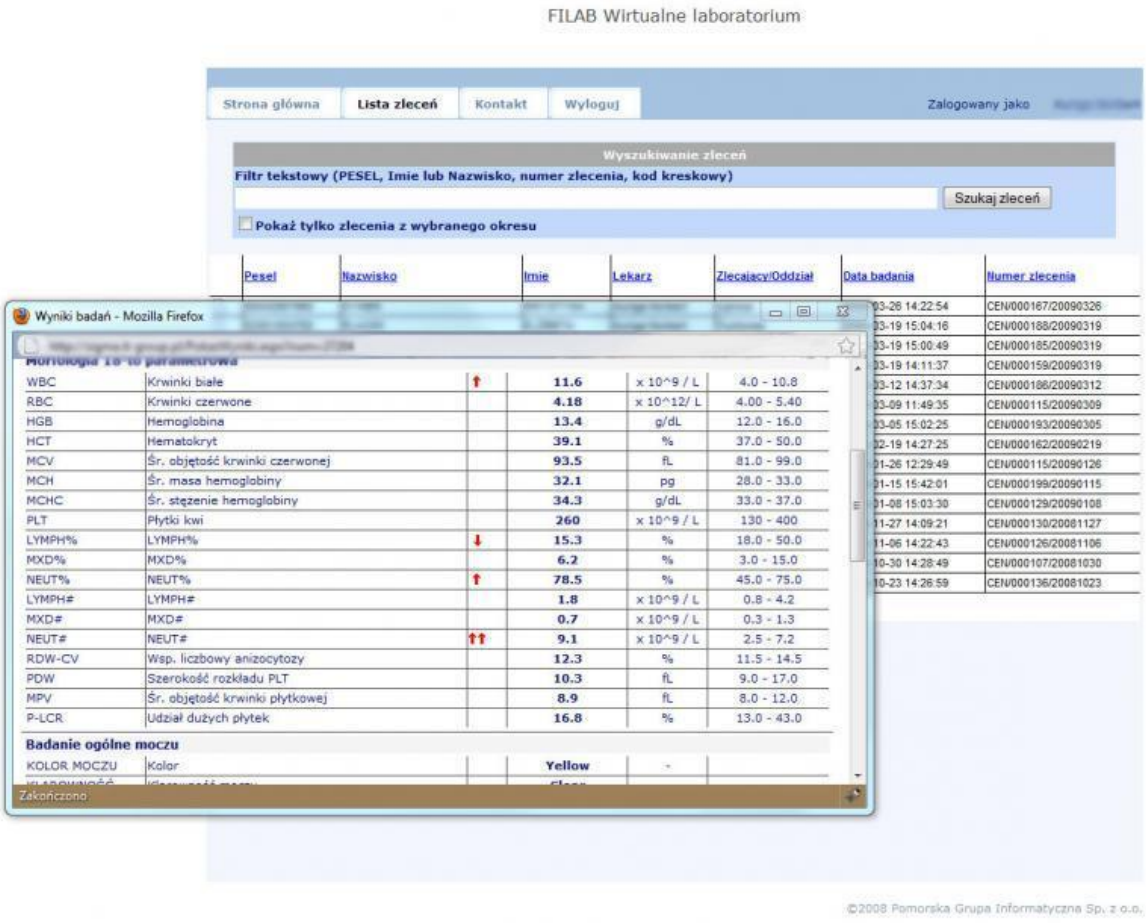
System FILAB 2.0 [1] firmy FIMED Sp. z o.o. jest już drugą odsłoną oprogramowania dla laboratoriów diagnostycznych. Zgodnie z opisem zamieszczonym, na stronie internetowej, w trakcie tworzenia oprogramowania były przeprowadzane konsultacje ze specjalistami diagnostyki laboratoryjnej, przedstawicielami branży IT i firm dostarczających sprzęt diagnostyczny.

Najistotniejsze cechy systemu:

- modułowość,
- profile badań,
- automatyzacja wprowadzania badań zleconych przez punkty pobrań,
- cenniki badań,
- automatyzacja procesu identyfikacji materiału i miejsca w którym się znajduje,
- wsparcie monitorów dotykowych w części procesów,
- personalizacja wydruków i raportów.

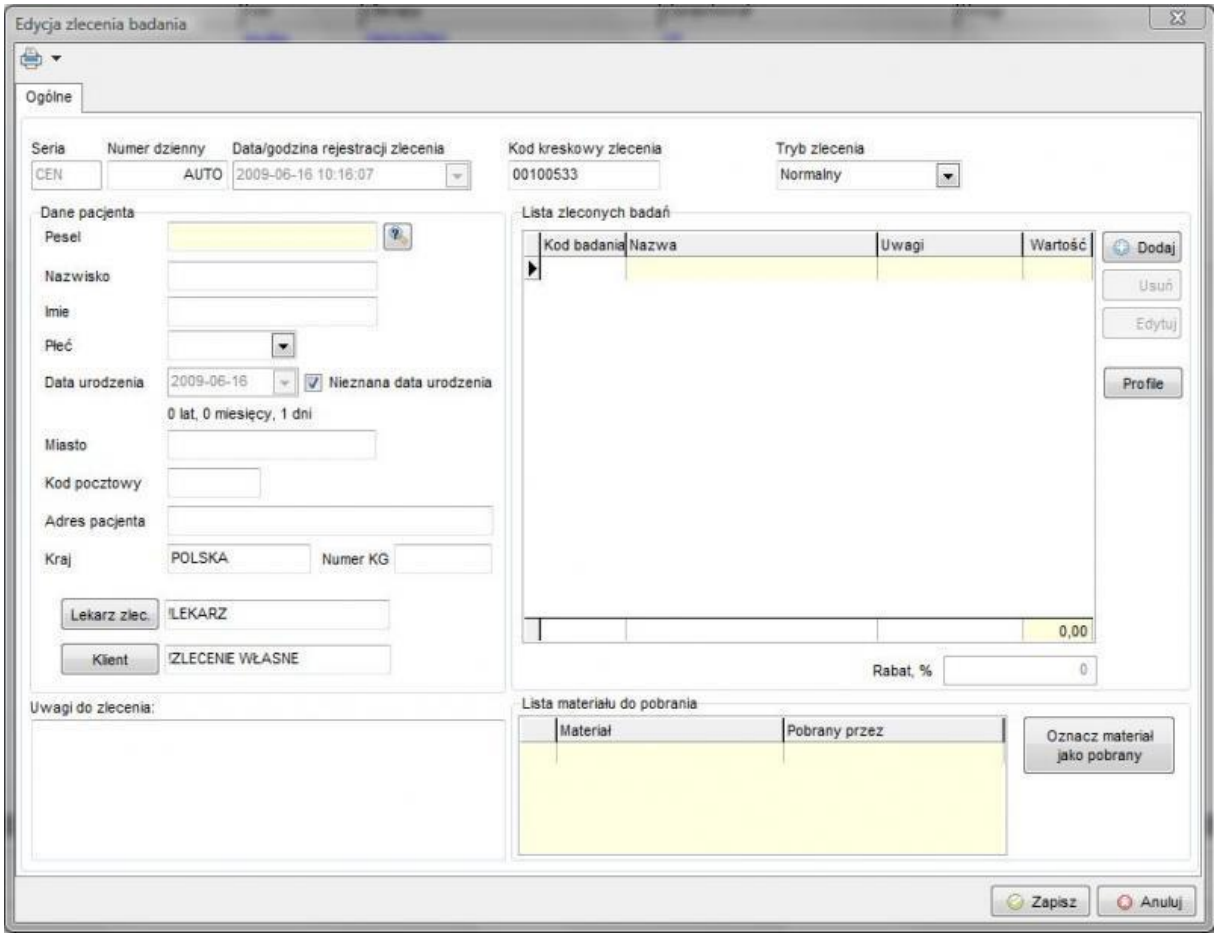
Powyższe cechy oraz dostosowanie komunikacji z systemami zewnętrznymi pozwalają na dokładne monitorowanie całego procesu analizy próbek, tworzenia i obsługi zleceń oraz informowania o wynikach klientów.

Przykładowy widok internetowej prezentacji wyników przedstawiono na rysunku 1.

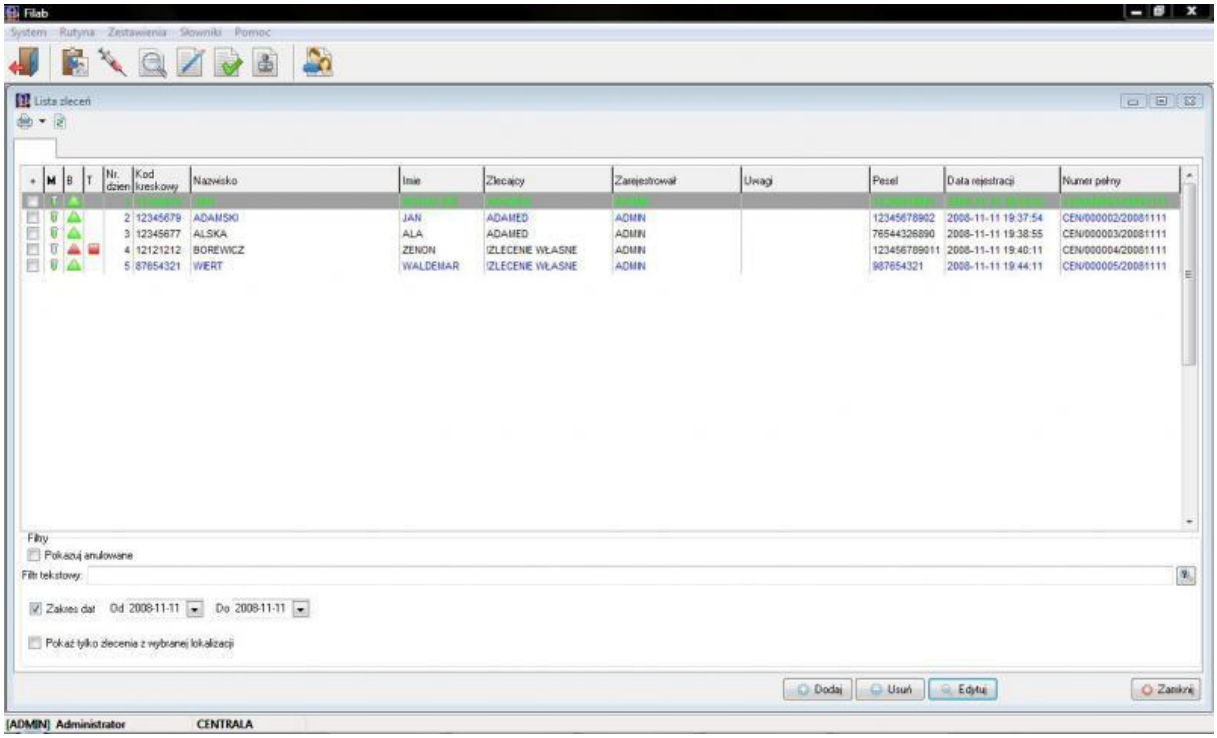


Rysunek 1. Portal z wynikami [1]

Najważniejszym elementem systemu jest rejestracja zleceń. Producent przekonuje o ergonomii, intuicyjności oraz przejrzystości modułu. Przykładowe ekrany prezentuje rysunek 2 oraz rysunek 3.



Rysunek 2. Tworzenie nowego zlecenia [1]



Rysunek 3. Lista zleceń [1]

2.2. KS-SOLAB

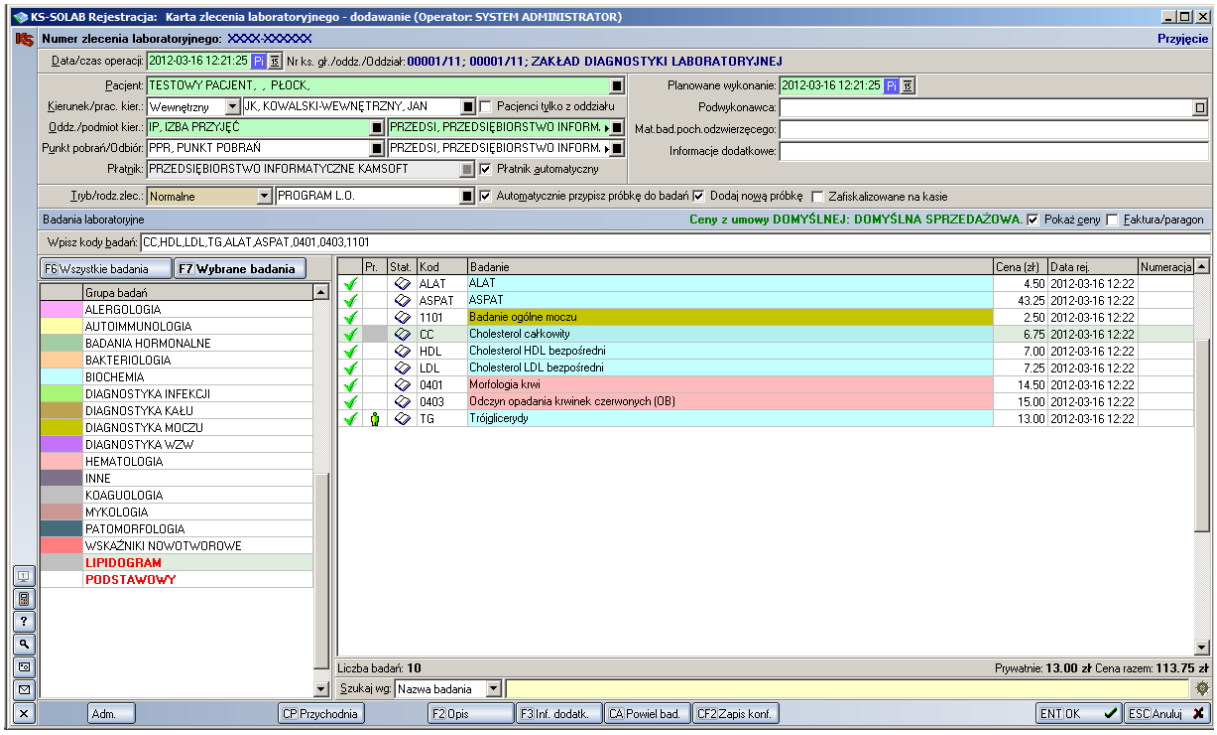
KS-SOLAB[2] firmy Kamsoft S.A. jest programem, który ma za zadanie zintegrowanie wszystkich procesów laboratoryjnych w jednym systemie informatycznym. Producent na swojej stronie prezentuje bazę wiedzy o programie, porady praktyczne, jak również obszerny cennik wraz z opisem konkretnych modułów systemu. Podobnie jak FILAB, system był tworzony we współpracy z pracownikami laboratoriów.

Najistotniejsze cechy systemu:

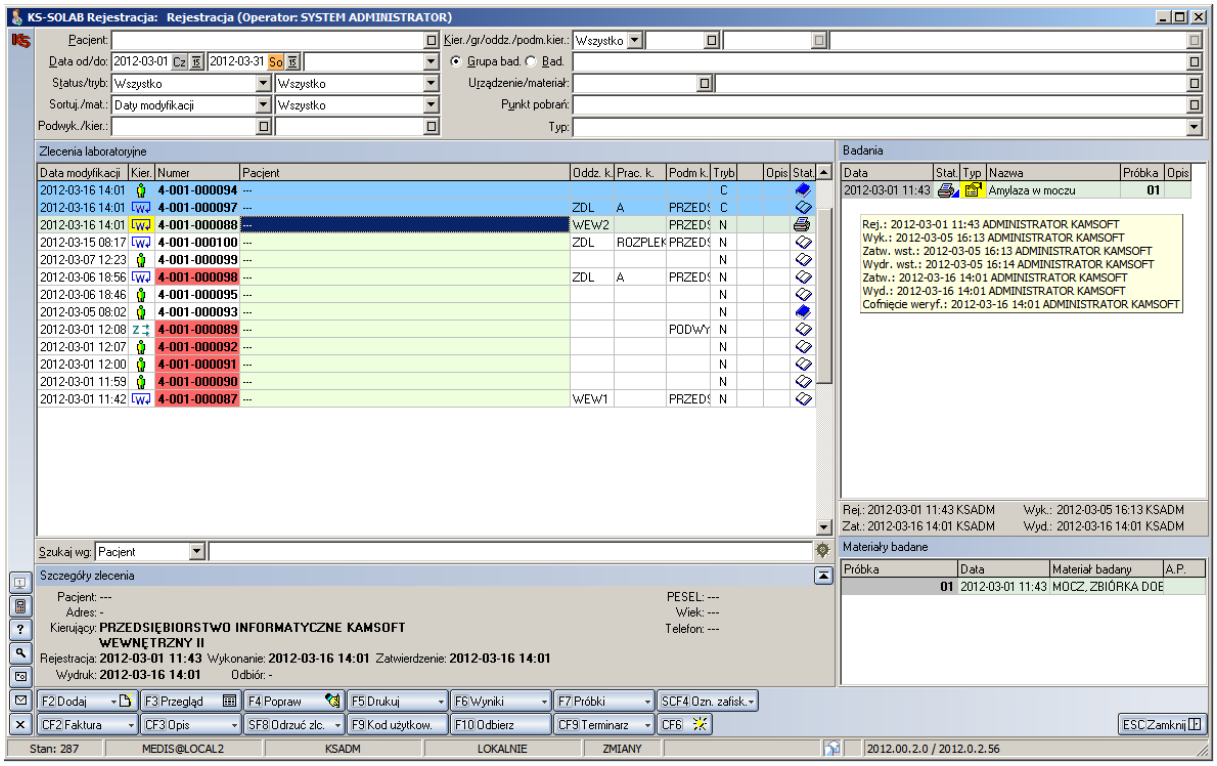
- automatyzacja wykonywania badań,
- współpraca systemu z urządzeniami laboratoryjnymi,
- kontrola jakości,
- pełne rozliczenia finansowe,
- generowanie dokumentacji oraz wielowymiarowe zestawienia statystyczne.

Automatyzacja pracy laboratorium jest zapewniona dzięki kodom kreskowym, które mogą być umieszczone na skierowaniach, probówkach czy wydrukach wyników badań. Aktualnie system jest w stanie zintegrować się z ponad dwustu czterdziestoma urządzeniami. Niestety ostatnia istotna aktualizacja systemu, miała miejsce w 2012 roku, samo oprogramowanie powstało w 2003 roku.

Rysunek 4 prezentuje kartę zlecenia laboratoryjnego, a na rysunku 5 możemy zobaczyć listę zleceń.



Rysunek 4. Karta zlecenie laboratoryjnego [2]



Rysunek 5. Lista zleceń laboratoryjnych [2]

2.3. Podsumowanie

W tym rozdziale przedstawiono dwa najpopularniejsze systemy dostępne na rynku, których zadaniem jest wsparcie laboratoriów różnego rodzaju. Niestety ze względu na brak współpracy ze strony producentów oprogramowania oraz szczątkowe informacje opublikowane na stronach firm konkurencyjnych, trudno było przeprowadzić szczegółową analizę tematu

Z dostępnych danych, wynika że omówione oprogramowanie ma kilka słabych stron:

- ścisłe związanie z jedną platformą – brak wsparcia dla *MacOSX*, *Linuxa* czy urządzeń mobilnych,
- nastawienie na zlecenia z branży medycznej - szczątkowe wsparcie ogólnej branży chemicznej i analiz np. paliw,
- archaiczny interfejs użytkownika - przeładowanie dziesiątkami przycisków oraz ubogie opcje dostosowywania aplikacji do wielkości ekranu, przez co nawigacja pomiędzy funkcjonalnościami systemu jest utrudniona.

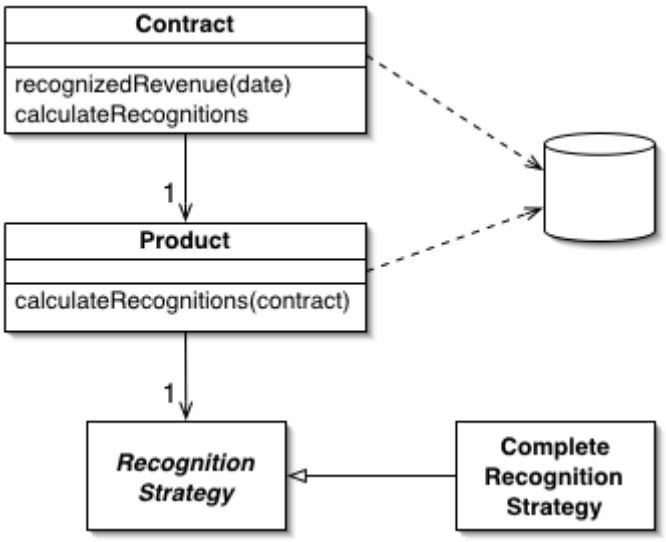
Prototyp stworzony przez autora, ma na celu stworzenie wieloplatformowego rozwiązania, które nie jest ściśle związane z medycyną, a dzięki modułowej budowie może być łatwo przystosowany do każdej branży. Dodatkowo warto wspomnieć, iż prototyp ten, ma znacznie bardziej rozwinięty moduł komunikacji z klientem, w porównaniu do powyżej opisanych systemów, na co składają się: rozbudowany portal dla klienta, a także notyfikacje wysyłane pocztą elektroniczną oraz poprzez krótkie wiadomości tekstowe.

3. Koncepcja rozwiązania

W niniejszym rozdziale opisano główne koncepcje użyte podczas projektowania generycznego systemu dla laboratoriów chemicznych. Skupiono się na najważniejszych założeniach projektowania w oparciu o *Domain-Driven Design*(DDD) oraz chmurę obliczeniową. Autor przybliżył znaczenie uniwersalnego języka wykorzystywanego przy projektowaniu oprogramowania i kontaktach z klientem biznesowym, jak również główne zalety i wady przeniesienia infrastruktury informatycznej do chmury i jej skalowalność w różnych warunkach. Na koniec została poruszona kwestia motywacji do stworzenia systemu opartego o koncepcje DDD oraz chmury obliczeniowej.

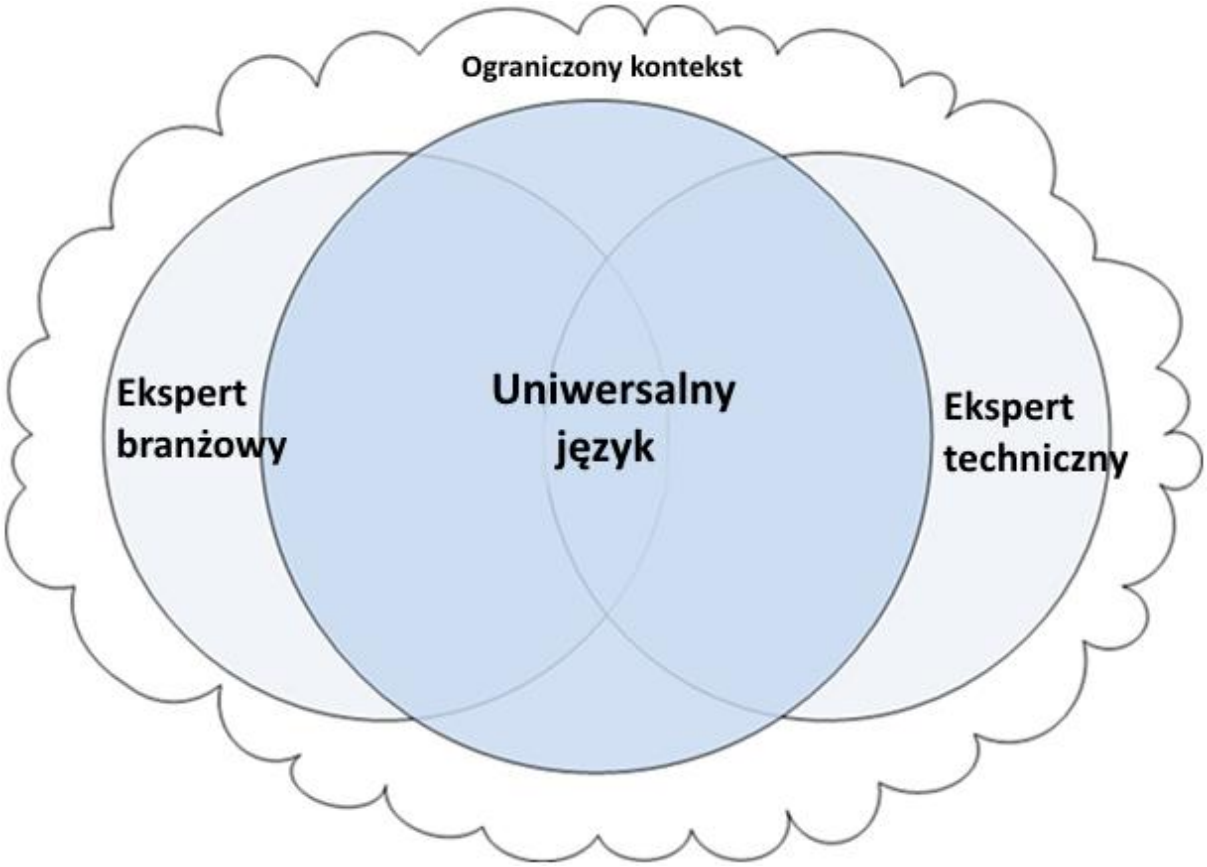
3.1. Uniwersalny język komunikacji z biznesem

Ile razy zdarzało wam się uczestniczyć w spotkaniu z klientem biznesowym i nie rozumieliście o czym on mówi? Ile razy zdarzało wam się opowiadać o wyrafinowanych rozwiązaniach zaimplementowanych w waszych systemach i klient myślał, że mówicie w języku obcym? Czy łatwo było wam się dogadać, znaleźć problem, mówić o tym samym, ale całkowicie w innych słowach? Jeśli tak to z pomocą przychodzi *ubiquitous language*(ang. uniwersalny język). Termin ten zapoczątkował Eric Evans[18] w 2003 roku który chciał zbudować spójny i łatwy w użyciu dla programistów i biznesu język. Uniwersalny język musi podlegać dość rygorystycznym zasadom, a to ze względu na fakt, że powinien bazować na modelu domeny biznesowej(Rysunek 6), która powinna być jednoznacznie określona. Wg Evansa, używanie uniwersalnego języka jest niezwykle ważnym aspektem podczas wytwarzania systemów informatycznych, który powinien być stosowany niezależnie od tego czy prowadzimy konwersację na spotkaniu biznesowym z ekspertami w dziedzinie, czy piszemy dokumentację lub modelujemy elementy procesu biznesowego. Ważnym aspektem używania takiego podejścia rosnące zrozumieniem każdej ze stron oraz ewolucja stosowanego języka.



Rysunek 6. Model obiektowy domeny, zawierający funkcjonalność oraz dane [20]

Na rysunku 7 przedstawiono jak potencjalnie odrębne dziedziny ekspertów domeny biznesowej i ekspertów z zespołu programistycznego powinny się przenikać i tworzyć wspólną część, czyli uniwersalny język, opisujący konkretną branżę w ten sam sposób dla obu stron.



Rysunek 7. Uniwersalny język w obrębie konkretnego kontekstu biznesowego [21]

Podczas tworzenia takiego języka niezwykle istotne jest to, aby to programiści mówili w języku biznesu, a nie odwrotnie. Dla lepszego zrozumienia problemu zasymulujmy następującą sytuację.

Laboratorium chemiczne, posiada specjalistyczny sprzęt do analizowania próbek dostarczanych przez klienta. Takie urządzenia generują nam pliki, które następnie mamy zaimportować do systemu. Programista otwierając ten plik widzi strukturę tego pliku, poszczególne elementy, atrybuty. Zadaniem zespołu technicznego jest stworzenie z danych zawartych w pliku odpowiednich wykresów, które muszą być opisane wcześniej zdefiniowanymi przez klienta symbolami.

Dochodzi do spotkania obu stron i rozmowy pomiędzy programistą i ekspertem biznesowym.

Programista mówi: “Zastanawiamy się jak wyrenderować ten wykres. Pierwszym dzieckiem w pliku jest elementu GPW, ma właściwość R z wartością 3, dodatkowo ma atrybut E. Nie wiemy jak mamy ten plik sparsować.”

Ekspert słysząc taką wypowiedź myśli: “Nie wiem, o czym on mówi, jak zapytam to odpowie jeszcze bardziej niezrozumiałym opisem. Niech zrobią tak jak uważają, przecież mają szkic.”

Takie podejście doprowadza do wielu nieporozumień, a finalnie do nie dotrzymania terminów i straty sporych funduszy, nie można również zapomnieć o nadszarpniętej reputacji obu ze stron.

Prawidłowa konwersacja mogłaby wyglądać następująco:

Programista mówi: “Zastanawiamy się jak wyświetlić ten wykres. Mamy dane o GPW z pola R z wartością 3 i literką E, ale nie wiemy co one oznaczają. Jak powinniśmy narysować wykres z takimi danymi?”

Ekspert odpowiada: “Chodzi o dane z trzeciej pracowni Gazów, a E oznacza poprawny odczyt. Gdyby zamiast E było I to oznaczałoby niepoprawny odczyt. Pokażę, jak coś takiego powinno wyglądać”.

Oczywiście, jest to tylko przykład, ale pokazuje, że nie należy zbytnio zagłębiać się w szczegóły techniczne, dotyczące tworzenia oprogramowanie, gdyż może to deprymować i dezorientować stronę biznesową. Łatwiej nawiązać komunikację, jeżeli zespół deweloperski pozna domenę biznesową i zacznie stosować takie samo nazewnictwo oraz będzie unikał technicznego żargonu. Warto również nazywać klasy oraz metody, tak samo, jak nazwałby je ekspert biznesowy. Jako programiści powinniśmy odzwierciedlać w naszym kodzie to, jak myślą i mówią nasi eksperci biznesowi.

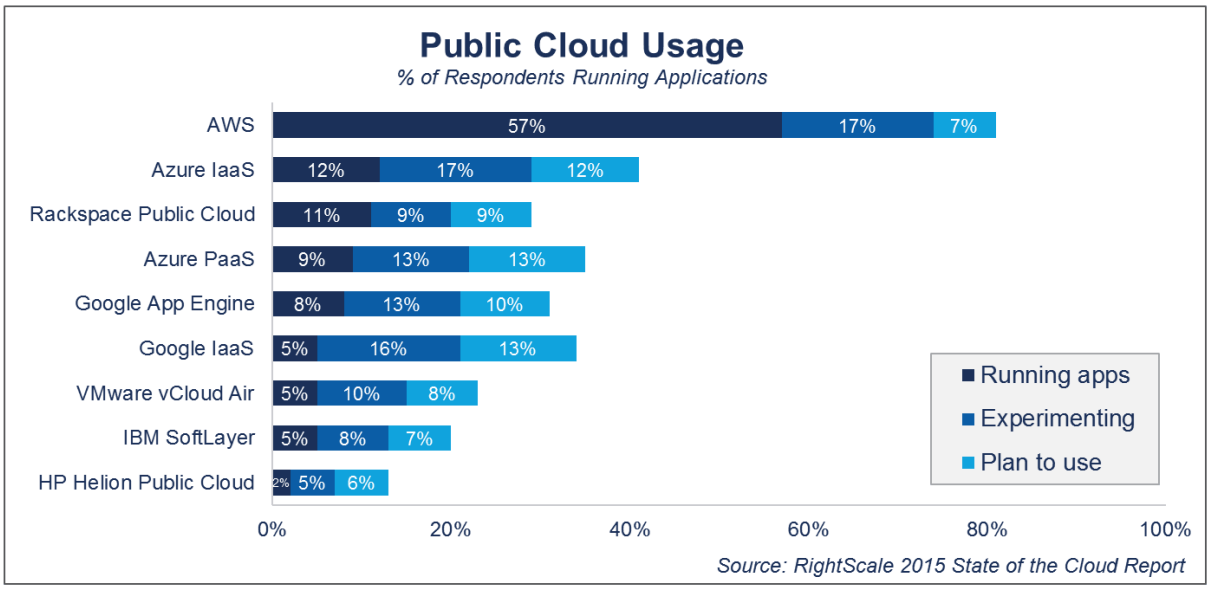
Podsumowując, używając wspólnego języka przez cały proces wytwarzania oprogramowania redukujemy ryzyko wystąpienia nieporozumień. Wytwarzając system w ten właśnie sposób, opracowujemy kod, który jest łatwiejszy do zrozumienia i modyfikacji. Całe podejście nie ma zastosowania w każdym przypadku, np. tworząc nowy kompilator lub parser rzadko potrzebujemy eksperta z dziedziny biznesu. Oczywiście możemy stworzyć uniwersalny język na nasze wewnętrzne potrzeby, ale będzie to język techniczny. Ważnym aspektem w całym procesie, jest przygotowanie zespołu do zmiany podejścia do komunikacji z klientem, gdyż większość programistów nie miała kontaktu z takim sposobem prowadzenia projektu i potrzebuje czasu na dostosowanie się. Warto wówczas mieć doświadczoną osobę, która przejmie rolę mentora zespołu.

3.2. Dlaczego warto przenieść infrastrukturę do chmury

Chmura obliczeniowa jest bardzo modnym tematem w ostatnim czasie, zainteresowanie nie tylko przejawia się wśród małych firm z niskim budżetem, ale również wśród największych rynkowych graczy. Jak możemy przeczytać w realbusiness[37] wg przewidywań firmy *Cisco* do roku 2016 aż 60 procent generowanego nakładu pracy w biznesie będzie zarządzana przez usługi chmurowe. Wg Gartnera, jednej z najbardziej znanych firm analitycznych na świecie, przynajmniej 80 procent organizacji zamierza wykorzystać usługi chmurowe w przeciągu najbliższych dwunastu miesięcy.

Chmura podzieliła środowisko IT na dwa obozy. Z jednej strony mamy wielbicieli, którzy są za przeniesieniem wszystkich usług do chmury niezależnie od potencjalnego ryzyka. Z drugiej strony mamy konserwatystów, według którzy chmura jest tylko chwytem marketingowym, gdyż jest dostępna od dawna, a samo rozwiązanie może przynieść więcej problemów niż korzyści.

Autor tej pracy jest zdania, że skorzystanie z chmury obliczeniowej ma więcej zalet, niż wad, a rozwój samej usługi jest niezagrożony w najbliższych latach. Na rysunku 8 możemy zobaczyć jak wyglądało użycie poszczególnych rozwiązań chmurowych w 2014 roku (badanie zostało przeprowadzone w styczniu 2015 roku), rozbite na trzy kategorie: działające aplikacje, eksperymenty oraz chęć użycia.



Rysunek 8. Wykorzystanie poszczególnych rozwiązań chmurowych w 2014 [22]

Zgodnie z rysunkiem 8 najbardziej popularna jest usługa firmy *Amazon* (AWS), na drugim miejscu, jednak ze znacznie słabszym wynikiem - *Microsoft Azure*. Lider rankingu, wprowadzając na rynek swoje usługi jako pierwszy walczył o klienta ceną i rozbudowaną ofertą.

Poniżej przedstawiono najważniejsze zalety i wady które należy wziąć pod uwagę , decydując się na przeniesienie infrastruktury do chmury.

Zalety:

1. **Dostęp do wyspecjalizowanych urządzeń nie posiadając ogromnego budżetu na jego sfinansowanie.**

Infrastructure as a Service (IaaS), czyli infrastruktura jako usługa, jest typem chmury obliczeniowej, w którym to dostawca wirtualizuje zasoby obliczeniowe poprzez internet. IaaS jest jednym z trzech typów usług chmurowych, wśród pozostałych należy wymienić *Software as a Service*(SaaS) oraz *Platform as a Service*(PaaS). Na potrzeby zrealizowania prototypu i przeprowadzenia doświadczeń skorzystano z pierwszego typu, czyli IaaS, który jest udostępniany dla klientów per jednostka. Wyodrębniamy kilka typów miar: żądanie, bajt, minuta, godzina, miesiąc itp. Taki model pozwala małym firmom na optymalizowanie kosztów, np. poprzez wyłączanie serwerów w godzinach nocnych. System pozwala również na proste tworzenie maszyn testowych, praktycznie zerowym kosztem (w momencie pisania tego tekstu godzina używania najmniejszej maszyny wirtualnej A0 kosztuje jedynie €0.0135). Co najistotniejsze, czas i koszt potrzebny do rozszerzenia naszych serwerów jest nieporównywalnie mniejszy niż aktualizacja sprzętu, który posiadamy fizycznie. Dla przykładu stworzenie nowej maszyny lub zwiększenie mocy obliczeniowej nie trwa dłużej niż 5 minut.

2. **Brak opłat za zużycie energii i utrzymanie sprzętowców odpowiedzialnych za utrzymanie fizycznych serwerów.**

Nawiązując do punktu 1, korzystanie z chmury zwalnia nas z jakichkolwiek opłat za prąd, powierzchnię oraz serwis. Dodatkowo wszelkie awarie sprzętowe są obsługiwane przez usługodawcę i są dla nas transparentne. W zależności od wykupionego pakietu dostawca gwarantuje nam kilka różnych poziomów *Service level agreement*(SLA). Przykładowo, jeżeli nasz dostawca gwarantuje nam 99.95% SLA to oznacza, że w ciągu roku nasza usługa może maksymalnie nie działać przez 4 godziny

i 38 minut. Oczywiście są pakiety, które gwarantują nam większą ochronę np. 99,999%, co gwarantuje nam tylko 5 minut i 26 sekund niedostępności naszej usługi rocznie.

3. IaaS pozwala na przeniesienie wszystkich naszych serwerów do chmury

Niezależnie od tego, czy mamy kilka aplikacji intranetowych postawionych na różnych systemach operacyjnych, czy serwery e-mailowe obsługujące całą komunikację, IaaS zapewnia nam pełną elastyczność. Możemy wybrać, czy chcemy używać darmowych dystrybucji *Linuxa*, czy maszyn z zainstalowanym *Windows Server*. *Microsoft Azure* oferuje również pełną gamę serwerów z preinstalowanym *SQL Serverem*, również w najdroższej wersji *Enterprise*. Zwalnia nas to z zarządzania fizycznymi maszynami, zapewniania bezpieczny dostęp do nich, odpowiednią klimatyzację czy agregaty prądotwórcze. Największą zaletą jest wyspecjalizowana infrastruktura sieciowa, która często wielokrotnie przekracza możliwości lokalnych dostawców internetu oraz koszt zakupu i utrzymania samego sprzętu, routerów, przełączników oraz zapór.

Wady:

4. Niestabilna infrastruktura sieciowa i energetyczna w siedzibie firmy.

Przenosząc całą infrastrukturę do chmury, musimy zapewnić naszym pracownikom odpowiednią jakość połączenia internetowego. Oczywiście nigdy nie osiągniemy takiej wydajności jaką oferuje wewnętrzna infrastruktura, ale duże opóźnienia i mała przepustowość naszego łącza może skutecznie zniechęcić pracowników do korzystania z takiego rozwiązania.

5. Awaria po stronie usługodawcy

Pomimo wysokiego SLA, może wystąpić sytuacja, w której dostawca nie będzie mógł zapewnić dostępu do naszych usług, przez czas dłuższy niż gwarantuje umowa. Usługodawca będzie wówczas zobowiązany zapłaci karę umowną, ale my możemy stracić o wiele więcej. Wyobraźmy sobie sytuację, w której giełda warszawska korzysta tylko z jednego centrum danych i po otwarciu sesji następuje awaria centrum danych. Czy giełda jest przygotowana na taką sytuację, czy dostawca jest odpowiednio zabezpieczony, ile zajmie przywrócenie systemów do działania? *Microsoft* zapewnia, że robi kopie zapasowe swoich centrów danych, na innych centrach, w odległości minimum 500 kilometrów od siebie. Nie ma jednak informacji, co się stanie w przypadku awarii całego centrum danych. Jak wielkie straty może przynieść 5 minut niedostępności usługi w zależności od branży?

Na te pytania i wiele innych musimy odpowiedzieć sobie sami, decydując się na przenosiny usług do chmury obliczeniowej. Pamiętajmy, że obecnie nie wszystkie usługi warto przenieść. Warto wziąć pod uwagę również to, czy jesteśmy gotowi na potencjalne skutki niedostępności naszych usług przez minutę, godzinę, a nawet cały tydzień.

3.3. Motywacja do stworzenia oprogramowania dla laboratoriów

Choć na naszym rynku można kupić oprogramowanie wspomagające zarządzanie laboratoriami o różnych specjalizacjach, to żadne z nich nie jest generyczne i otwarte na inne rynki niż polski. W tym podrozdziale przedstawiono główne wymagania funkcjonalne oraz niefunkcjonalne, które musi spełnić generyczny system informatyczny, aby usprawnić działanie dowolnego laboratorium.

Podczas analizowania rynku, wydzielono najistotniejsze wymagania funkcjonalne, które muszą być spełnione przez system:

- tworzenie zlecenia na wykonanie analizy dostarczonej próbki,
- aktualizacja zlecenia o takie informacje jak miejsce wykonania analizy, dane o sposobie pobrania próbki, komentarz laboratorium,
- możliwość tworzenia dowolnych raportów z przeprowadzonej analizy,
- możliwość utylizacji pobranych próbek w sposób zgodny ze standardami,
- wysłanie kontrolera do próbobrania,
- notyfikacja klienta o zakończonej analizie,
- notyfikacja dyspozytora o przyjęciu zlecenia do analizy,
- dodanie nowego kontrahenta,
- edycja kontrahenta,
- drukowanie kopert,
- zarządzanie metodami badawczymi, które mogą być wykonywane w laboratorium,
- możliwość dowolnego definiowania wymagań, które są niezbędne do spełnienia przez analizowaną próbkę,
- możliwość wystawienia faktury za wykonaną analizę,
- możliwość stworzenia korekty faktury,
- możliwość stworzenia noty korygującej fakturę,
- możliwość dodawania, edycji oraz usuwania użytkowników.

Powyższe wymagania pozwalają na zbudowanie prototypu, który byłby konkurencyjny na obecnym rynku. Dodatkowym atrybutem było wprowadzenie wielojęzyczności, która może zostać wykorzystana przy wprowadzaniu systemu na rynek Unii Europejskiej.

Wymagania niefunkcjonalne, które muszą zostać spełnione, aby system odniósł sukces, to między innymi:

- **użyteczność** - którą rozumiemy jak łatwość korzystania przez użytkownika z nowego rozwiązania, jego przejrzystość i czytelność. W skład użyteczności należy zaliczyć wszystko co widzi użytkownik, od tła aplikacji, poprzez zastosowaną czcionkę, czy dostępną pomoc do systemu.
- **wydajność** - w tej kategorii najważniejsze jest wykorzystanie koncepcji tworzenia oprogramowania z użyciem chmury obliczeniowej i jej skalowalność. System powinien działać tak samo szybko w przypadku użycia przez pięciu użytkowników jak i pięciuset. Z doświadczenia autora wynika, że poszczególne strony systemu nie powinny ładować się dłużej niż pięć sekund i to niezależnie od obciążenia.
- **niezawodność** - system powinien działać bez awarii możliwie jak najdłużej, wszelkie przerwy w działaniu powinny być usuwane w czasie nie dłuższym niż godzina, a pomniejsze błędy w ciągu maksymalnie dwóch dni roboczych. Awarię rozumiemy jako kompletny brak dostępu do systemu. Dzięki wykorzystaniu rozproszonej chmury obliczeniowej możemy zredukować niebezpieczeństwo awarii do minimum.
- **bezpieczeństwo** - do systemu powinni mieć dostęp tylko uprawnieni użytkownicy zastosować należy również komunikację szyfrowaną. Certyfikat bezpieczeństwa

powinien być wystawiony przez zaufanego dostawcę i akceptowany przez wszystkie przeglądarki internetowe.

Dodatkowym wymaganiem niefunkcjonalnym jest „wspieralność". Cecha ta, nie dotyczy wyłącznie samego procesu wyboru technologii, ale również niezbędnej infrastruktury, zapewniającej poprawne działania całego systemu. Ważnym aspektem tego wymagania jest łatwość wprowadzania zmian oraz dodawanie nowych funkcjonalności.

Kolejną, istotną kwestią, jest rosnący rynek mobilny, system powinien mieć możliwość łatwej adaptacji serwisów dostarczanych przez aplikację z interfejsem aplikacji mobilnych.

Dzięki zastosowaniu modułowej budowy systemu (Rysunek 9) oraz zastosowaniu skalowalnej infrastruktury *Microsoft Azure*[3], możliwe było stworzenie aplikacji elastycznej i stabilnej w każdych warunkach. Natomiast DDD ułatwiło budowanie nowych modułów, które dzięki zastosowaniu wstrzykiwania zależności, w prosty sposób pozwoliło na rozszerzenie systemu o nowe funkcjonalności. Z uwagi za fakt, iż, jednym z wymagań niefunkcjonalnych była niezawodność, podczas tworzenia oprogramowania, każdy element zawierający logikę biznesową został przetestowany, przy użyciu procesu *Behavior-driven development*(BDD), który najlepiej wpisuje się w projektowanie oprogramowania zorientowanego obiektowo i przy użyciu DDD.

Rysunek 9. Modułowa budowa systemu, opracowanie na bazie platformy mobilnej Telerik

4. Projektowanie aplikacji w oparciu o chmurę obliczeniową

W tym rozdziale zaprezentowano dwie najważniejsze usługi chmurowe na rynku. Rozwinięto również koncept DDD, zaprezentowany po krótce w poprzednich rozdziałach. Skupiono się także na najciekawszych funkcjach chmury obliczeniowej oraz przeprowadzono badania mające na celu porównanie wydajności różnego rodzaju baz danych, dostępnych na *Microsoft Azure*. Na koniec opisano sposób, w który chmura pomogła w prowadzeniu projektu, zarządzaniu kodem oraz automatycznym publikowaniu nowych wersji.

4.1. Przegląd najpopularniejszych chmur

W poniższym rozdziale opisano najpopularniejsze rozwiązania chmurowe na rynku, a także przedstawiono główne funkcjonalności każdej z nich.

4.1.1 Microsoft Azure

Platforma *Microsoft Azure* została pierwotnie zaprezentowana jako *Windows Azure* w 2008 roku, a oddana do komercyjnego zastosowania w 2010 roku. Jest to platforma stworzona przez firmę *Microsoft*, udostępniająca usługi typu *PaaS* oraz *IaaS*, obsługująca różne systemy operacyjne, wiele języków programowania i różnorakie frameworki i narzędzia.

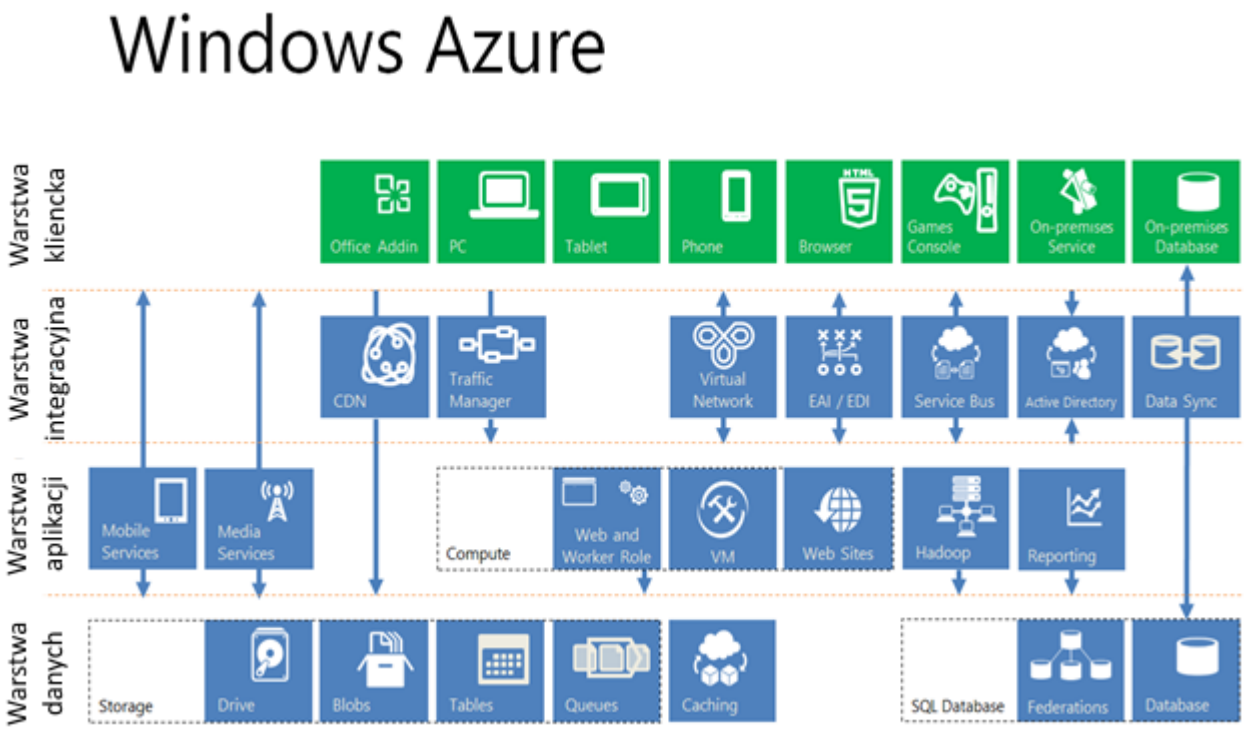
Microsoft udostępnia swoje centra danych na całym globie w siedemnastu regionach, dla samej Europy zostały utworzone dwa centra: w Irlandii oraz Holandii. Warto dodać, że nie wszystkie regiony oferują taką samą funkcjonalność.

Prywatność naszych danych regulują dwa akty prawne, *USA PATRIOT Act (Uniting and Strengthening America by Providing Appropriate Tools Required to Intercept and Obstruct Terrorism Act of 2001)* oraz *E.U. Data Protection Directive (95/46/EC).* Pierwszy z aktów daje pełny dostęp rządowi USA do wglądu w nasze bazy. Regulacje Unii Europejskiej są dużo bardziej restrykcyjne, podmiot, do którego należą dane musi zostać poinformowany, jeśli jego baza ma zostać udostępniona. Aktualny stan prawny może zniechęcić wiele europejskich firm, ponieważ rząd USA ma wgląd do danych znajdujących się na terenie unii, nawet jeżeli sprawa dotyczy firmy, która nie prowadzi swojej działalności na terenie Stanów Zjednoczonych.

Windows Azure oferuje szeroki wachlarz funkcjonalności, najbardziej popularnymi są *Web Sites* umożliwiające hostowanie stron internetowych napisanych między innymi w *PHP, .NET, Node.js*, czy *Pythonie.* Dzięki galerii, możemy tworzyć strony internetowe z predefiniowanych szablonów, czy. zbudować bloga opartego o *WordPress'a* lub *Joomle.* Dzięki integracji z *Visual Studio* możemy za pomocą kilku kliknięć myszki publikować nowe wersje aplikacji bezpośrednio do chmury. Równie ważną usługą jest możliwość tworzenia maszyn wirtualnych, które obsługują zarówno *Linuxa* jak i *Windows'a*, przy użyciu własnych obrazów lub predefiniowanych z galerii. W momencie tworzenie tego tekstu były dostępne trzy typy maszyn wirtualnych:

- **A -** Ekonomiczna wersja, dedykowana serwerom testowym i aplikacjom, które nie wymagają automatycznego skalowania i dużej ilości pamięci RAM,
- **D -** 60% szybsze procesory niż w wersji A, dyski HDD zastąpione dyskami półprzewodnikowymi SSD. Seria dedykowana do aplikacji wymagających wydajnych maszyn,
- **G -** oparta jest o procesory *Intel Xeon E5* w wersji 3, dwukrotnie więcej pamięci operacyjnej i aż czterokrotnie więcej przestrzeni dyskowej niż wersja D. Najwydajniejsze maszyny, ale i najdroższe, kosztujące prawie dwustu krotność najtańszej maszyny serii A.

Najważniejsze funkcje *Azure* z podziałem na warstwy prezentuje rysunek 10.



Rysunek 10. Najważniejsze funkcje platformy Azure [23]

4.1.2 Amazon Web Services

Oficjalnie zaprezentowany w 2006 roku jako zbiór serwisów dostępnych poprzez protokół HTTP. Serwisy używają do komunikacji architektury *Representational State Transfer* (REST) oraz protokołu *Simple Object Access protocol* (SOAP). Rdzeniem AWS jest *Amazon* EC2 (Elastic Compute Cloud), który pozwala na tworzenie wirtualnych maszyn. *Elastic compute units* są jednostkami miary zasobów komputerowych zaproponowanych przez Amazona. Jedna ECU to odpowiednik komputera z procesorem o mocy 1-1.2 GHZ.

Podobnie jak *Microsoft*, *Amazon* ma swoje centra danych na całym świecie, obecnie znajdują się w 11 regionach geograficznych. W Europie jest to tylko Dublin, dla porównania *Microsoft* oferuje dwie lokalizacje w Europie.

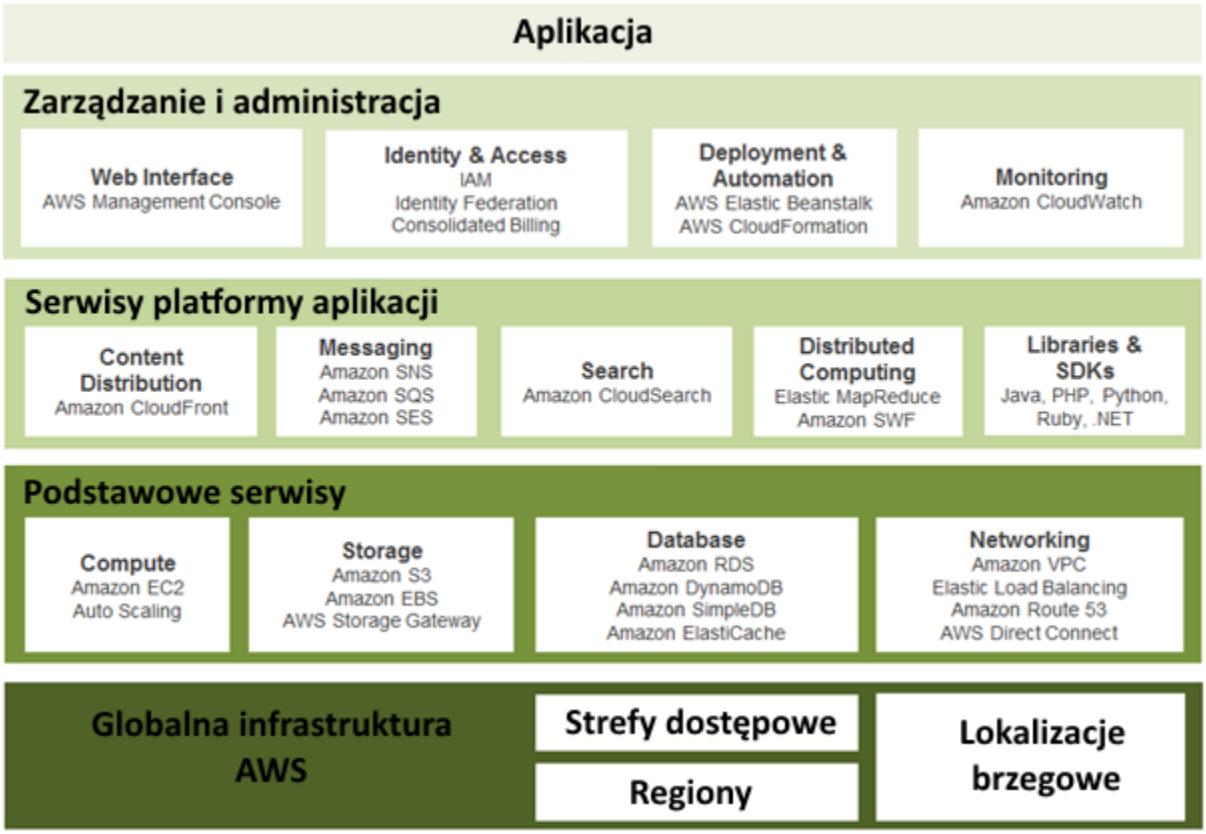
Prywatność danych przechowywanych w chmurze jest zapewniana dzięki programowi *Safe Harbor*, respektowanemu przez Unię Europejską, USA oraz Szwajcarię. Dodatkowo, od 2011 roku *Amazon* posiada specjalną wersję swojej chmury *AWS GovCloud*, która spełnia szereg obostrzeń narzuconych przez rząd USA i departament obrony narodowej.

Amazon udostępnia bogatszą paletę maszyn wirtualnych niż *Azure*. Poniżej zostały przedstawione poszczególne typy:

- **t -** Ekonomiczna wersja, dedykowana serwerom testowym i aplikacjom, które nie wymagają automatycznego skalowania i dużej ilości pamięci RAM,
- **m oraz c -** jest odpowiednikiem wersji **D** platformy *Azure* i może być wykorzystywana tak do aplikacji, które nie wymagają dużej wydajności, jak i do tworzenia klastrów do obsługi wielkich serwisów,
- **g -** specjalna wersja maszyny wyposażona w *NVIDIA Tesla*, do obliczeń wykonywanych na karcie graficznej.

- **h -** najdroższa opcja z maksymalnie 117 GB pamięci operacyjnej, 16 rdzeniami oraz 48000 GB pamięci dyskowej.

Najważniejsze funkcje *Amazon AWS* z podziałem na warstwy prezentuje rysunek 11.



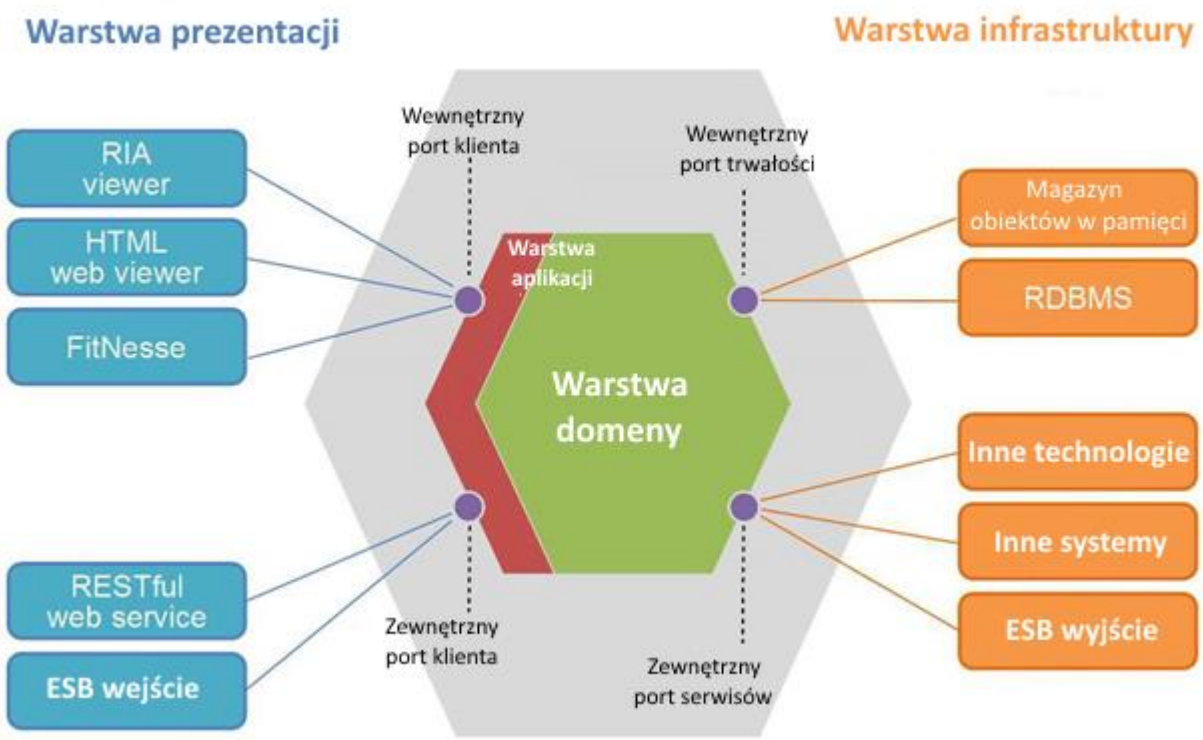
Rysunek 11. Najważniejsze funkcje platformy AWS [24]

4.2. Domain-driven design

Poniższy rozdział ma na celu zaprezentowanie wzorców projektowych, które pomagają w zaprojektowaniu elastycznego systemu.

4.2.1 Architektura cebulowa

W 2005 roku Alistair Cockburn, jeden z inicjatorów ruchu *Agile,* zaprezentował odmienne podejście do tworzenie systemów informatycznych, nazwane architekturą hexagonalną[25]. Cockburn chciał stworzyć systemy, które mogłyby być w takim samym stopniu wykorzystywane przez użytkowników, aplikacje integrujące jak i testy automatyczne. Zaznaczając, że proces tworzenia i testowania powinien być prowadzony w środowisku odizolowanym. Model koncepcyjny prezentuje Rysunek 12.



Rysunek 12. Architektura hexagonalna [25]

Bazując na tej koncepcji Jeffrey Palermo[38], w 2008 roku zaproponował wariację architektury hexagonalnej, czyli architekturę cebulową. Rysunek 13 przedstawia jego koncepcję. Głównym założeniem tego podejścia jest łatwiejsze zarządzanie powiązaniami pomiędzy poszczególnymi warstwami systemu. W architekturze cebulowej korzystamy z reguły schodzenia do środka, a nie odwrotnie. Dzięki temu podejściu najwyższa warstwa interfejsu użytkownika oraz infrastruktury jest w miarę bezboleśnie wymienialna. Daje to duże pole manewru w tworzeniu nowych wersji *UI*, czy zmiany bazy danych w używanej w aplikacji. W podejściu cebulowym nie obudowujemy systemu wokół konkretnych zasobów bazodanowych, serwisów czy plików, a jedynie tworzymy interfejs, który może być wykorzystywany do interakcji z zasobami zewnętrznymi.

Niezwykle istotnym aspektem podejścia cebulowego jest wykorzystanie wstrzykiwania zależności[6] (ang. *Dependency Injection*), ponieważ rdzeń aplikacji zapewnia tylko bazowe interfejsy, bez implementacji. Dopiero tworząc wyższe warstwy wdrażamy poszczególne interfejsy. *DI* pozwala na powiązanie interfejsów rdzenia z klasami z wyższych warstw podczas uruchamiania

aplikacji. Do tego celu możemy użyć popularnych narzędzi takich jak: *Windsor Castle, AutoFac, StructureMap, Unity* i wielu innych dostępnych na rynku.

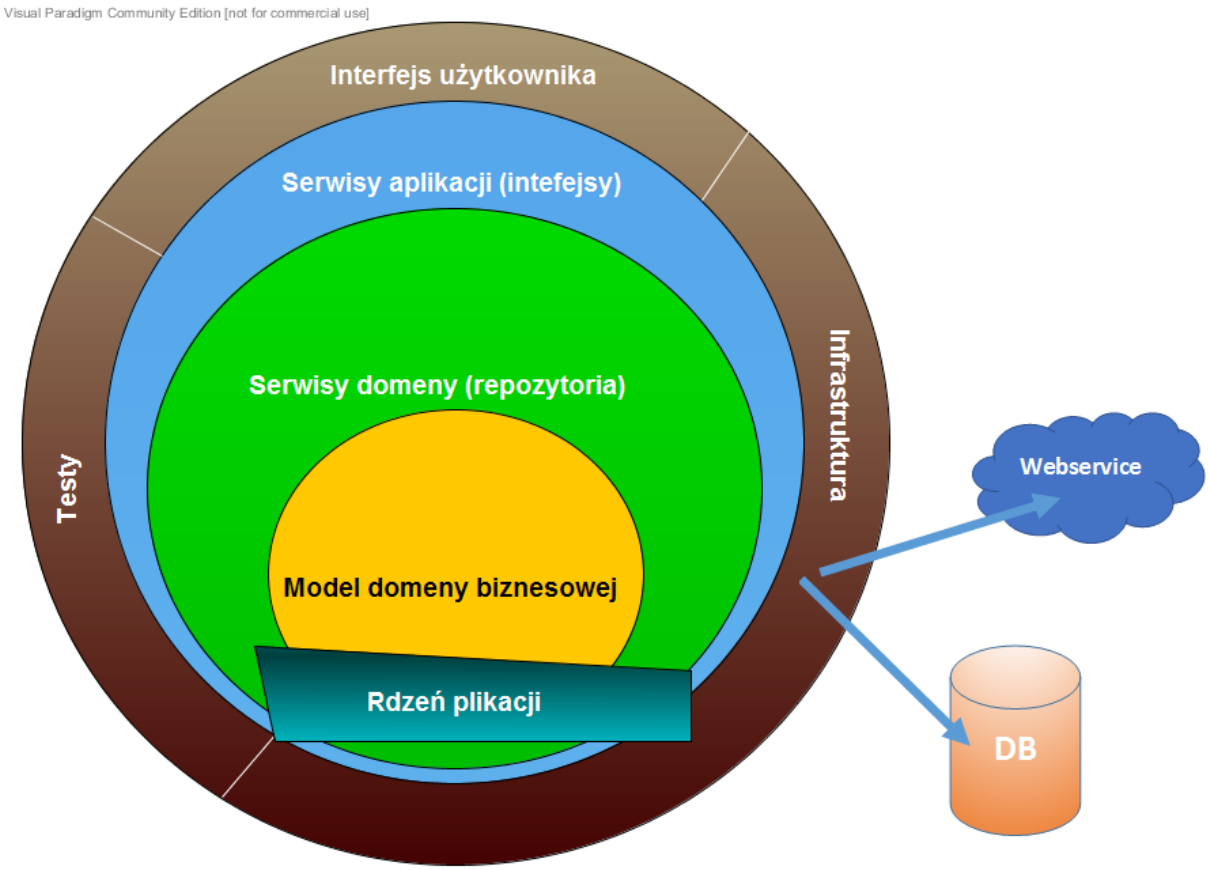
W samym środku diagramu jest *Domain Model*, rdzeń naszej aplikacji, który powinien być niezależny od bibliotek zewnętrznych, baz danych, serwisów zewnętrznych itd. Powinny to być same generyczne obiekty i interfejsy, bez konkretnej implementacji. Rdzeń nie może zawierać referencji do wyższych warstw.

Struktury okalające rdzeń *Domain Services* odpowiadają za logikę biznesową, to w tej warstwie jest wykorzystywany *ubiquitous language*. Dzięki takiemu podejściu, warstwa wyżej *Application Services* kontroluje domenę biznesową, jednocześnie walidując informacje przesyłane przez *API* zanim dotrą do konkretnej części warstwy domenowej. W części *Domain Services* znajdują się również klasy implementujące, interfejsy rdzenia aplikacji odpowiadające za operacje typu *Create-Read-Update-Delete* (CRUD) na poszczególnych obiektach modelu.

Application Services(API) jest punktem wejściowym do domeny biznesowej, przy użyciu *Data transfer object(*DTO). Obiekty DTO mają na celu odizolowanie warstwy API od modelu domenowego i zablokowanie bezpośrednich operacji na nim. Dobrym podejściem jest rozpoczęcie tworzenia nowych metod od tej sekcji, niejako tworząc szkielet z wysokopoziomowymi testami funkcjonalnymi, schodząc następnie niżej, dodając logikę biznesową.

W ostatniej warstwie cebuli mamy elementy, które są najbardziej podatne na zmiany, czyli interfejs użytkownika, testy czy szeroko pojęta infrastruktura. W tej warstwie zawarte są odpowiednie adaptery do komunikacji z bazami danych czy zewnętrznymi serwisami. Warto jeszcze raz podkreślić, że interfejs użytkownika ma dostęp do naszego API, ale już API nie ma dostępu do interfejsu.

Podążając wytycznymi architektury cebulowej oraz DDD tworzymy systemy informatyczne, które mogą mieć bardzo długi czas życia. Pamiętajmy jednak, że takie podejście niesie za sobą szereg utrudnień i jest czasochłonne. Wymaga również doświadczonego zespołu i odpowiedniego projektu. Dla małych aplikacji lub stosunkowo prostych, nie jest zalecane wykorzystywanie tego konceptu.



Rysunek 13. Model architektury cebulowej[38]

4.2.2 Anemiczność modelu

Czym jest anemiczny model domeny? Początkowo może wydawać się zwyczajną klasą zawierającą pola i właściwości. Można zapytać co w tym złego? Patrząc na koncepcję DDD takie klasy są uznawane za antywzorzec[20].

Klasy, które nie modelują zachowań można uznać za kolekcję geterów oraz seterów. Czy to źle, że oddzielamy logikę biznesową od klasy? Wydzielając logikę biznesową do serwisów domenowych tworzymy podział pomiędzy danymi, a metodami, co bezpośrednio łamie jedną z reguł programowania obiektowego jaką jest otwartość na rozszerzanie, ale zamknięcie na modyfikację. Przeciwieństwem anemicznego modelu jest tzw. bogaty model domeny, czyli klasa zawierająca pola, metody oraz logikę biznesową, całkowicie pozbawiając sensu tworzenie serwisów domenowych. Tworzenie takich klas jest zgodne z programowaniem obiektowym i spełnia założenia enkapsulacji i ukrywania informacji.

Anemiczny model rezygnuje z tej reguły, na rzecz większej elastyczności i łatwiejszego utrzymania kodu, pozwalając na tworzenie dedykowanych klas zawierających reguły biznesowe, które udostępniają odpowiednie interfejsy. Takie podejście ułatwia testowanie poszczególnych partii systemu.

Najłatwiej różnice pokazać na przykładzie, jako pierwszy zostanie zaprezentowany bogaty model i proces tworzenia nowego zlecenia przez klienta. Jedyną regułą biznesową jest posiadania odpowiedniej ilości kredytów do stworzenia nowego zamówienia. Listing 1 pokazuje bogaty model.

Listing 1. Przykład tzw. bogatego modelu.

```
class Customer : Entity//dziedziczy po klasie bazowej
{
    //pola i właściwości
    public bool IsOrderPossible(OrderDetails orderDetails)
    {
        return this.Credits >= orderDetails.Cost;
    }

    public void PlaceOrder(OrderDetails orderDetails)
    {
        if(IsOrderPossible(item))
        {
            Order order = new Order(this, orderDetails);
            order.Update();
            this.Funds -= item.Cost;
            this.Update();
        }
    }
}
```

Klasa *Entity* odpowiada za operacje *CRUD*. Rolą encji domenowej *Customer* jest stworzenie nowego zlecenia oraz komunikacja z warstwą danych. Na Listingu 2 zaprezentowano podejście anemiczne, przy zastosowaniu tej samej logiki.

Listing 2. Przykład anemicznego modelu

```
class Customer { /* właściwości publiczne */ }
class OrderDetails { /* właściwości publiczne */ }
class IsOrderPossibleService : IIsOrderPossibleService
{
    public bool IsOrderPossible(Customer customer, OrderDetails OrderDetails)
    {
        return customer.Credits >= OrderDetails.Cost;
    }
}
class OrderService : IOrderService
{
    ICustomerRepository customers;
    IOrderFactory orderFactory;
    IOrderRepository orders;
    IsOrderPossibleService isOrderPossibleService;

    public void PlaceOrder(Customer customer, OrderDetails OrderDetails)
    {
        if(isOrderPossibleService.IsOrderPossible(customer, OrderDetails))
        {
            Order order = orderFactory.CreateOrder(customer, OrderDetails);
            orders.Insert(order);
            customer.Credits -= OrderDetails.Cost;
            customers.Update(customer);
        }
    }
}
```

Porównując oba przykłady, znacznie więcej kodu jest w modelu anemicznym, który jest bliższy regułom *SOLID[12]*. Takie podejście daje nam luźniejsze powiązanie pomiędzy poszczególnymi elementami, większą spójność, łatwiejszą modyfikację i lepszą testowalność. Zastosowana architektura poza większą elastycznością, pozwala również na łatwiejsze tworzenie testów automatycznych. Dzięki użyciu tego modelu oraz wykorzystaniu automatycznego wstrzykiwania zależności, tworzenie testów jest o wiele przyjemniejsze, bardziej przejrzyste oraz może zaoszczędzić programiście sporo cennego czasu.

Z drugiej strony trzeba pamiętać, że jest sporo ekspertów, między innymi Martin Fowler, którzy uważają, że anemiczny model jest antywzorcem i absolutnie nie powinien być wykorzystywany. Powołują się na podstawową ideę programowania obiektowego, którą jest powiązanie danych z procesem. Twierdzą również, że takie podejście jest równie kosztowne jak bogaty model, równocześnie nie niosąc ze sobą żadnych benefitów. Głównym kosztem ma być mapowanie obiektów oraz wprowadzenie tzw. skryptów transakcyjnych, mających na celu organizację logiki biznesowej w kolekcję procedur, gdzie każda z nich ma za zadanie obsłużenie pojedynczego żądania. Podsumowując, budowanie anemicznego modelu ma tyle samo zwolenników, co przeciwników i do architekta systemu należy decyzja, czy warto z takiego podejścia skorzystać, czy nie.

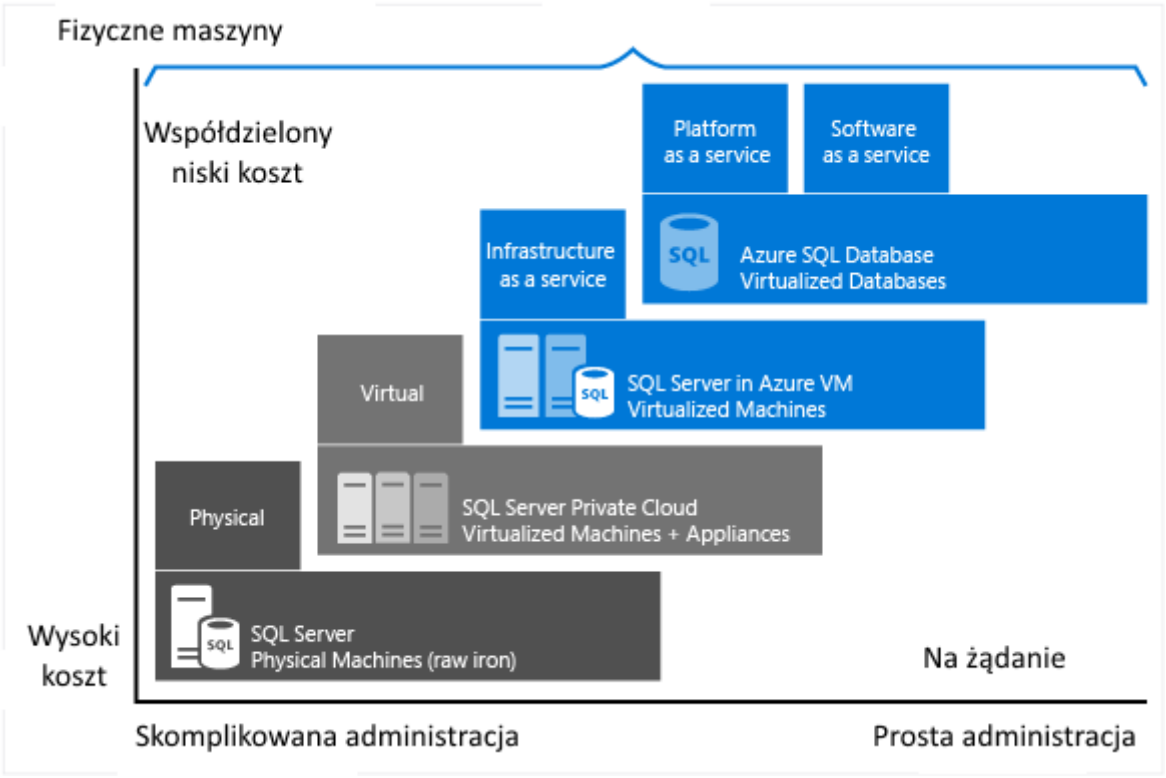
4.3. Przechowywanie danych w chmurze

W poniższym rozdziale opisano dwa typy najbardziej popularnych baz danych. Bazy relacyjne zostały opisane w wersji chmurowej oraz tzw. *stand-alone*. Następnie przedstawiono wady i zalety wykorzystania poszczególnych rozwiązań. Rozdział kończy, krótki test opisanych wcześniej rozwiązań przeprowadzony na platformie *Azure*.

4.3.1 Bazy SQL

Bazy relacyjne są dotychczas najczęściej spotykanym podejściem do przechowywania danych. Dzięki wykorzystaniu *Structured Query Language*(SQL)[15], możemy łatwo operować danymi i generować raporty. Aktualnie są dostępne różne implementacje baz relacyjnych, takie jak *Microsoft SQL Server*, *MySQL* czy *Oracle Database*. Na potrzeby tej pracy wykorzystano *Microsoft SQL Server* w wersji chmurowej oraz tzw. *stand-alone* zainstalowanej na wirtualnej maszynie w wersji 2014.

Czym się różnią od siebie *SQL Server* zainstalowany na wirtualnej maszynie od *Azure SQL*? Podstawą są koszty utrzymania, jeżeli nasza baza jest stosunkowo mała (ważna jest wielkość w GB, a nie liczba rekordów), to w większości wypadków wystarczy nam *Azure SQL*, w każdym innym przypadku zaleca się wykorzystanie pełnoprawnej wersji instalowanej na dedykowanej maszynie. Diagram na rysunku 14 powinien pomóc w podejmowaniu decyzji. Na osi X mamy poziom administracji, który nas interesuje, natomiast na osi Y koszt utrzymania wybranej opcji.



Rysunek 14. Wykres przedstawiający stosunek kosztów do poziomu administracji w zależności od poziomu bazy SQL [27]

Azure SQL Database jest udostępniany jako *database-as-a-service*, czyli *PaaS*. Jest to specyficzna wersja *SQL Servera*, nie wymaga dedykowanej maszyny oraz jest najtańsza w utrzymaniu. *Microsoft* zapewnia, że taka baza danych umożliwia skalowalność i funkcjonalność na najwyższym poziomie, jednocześnie eliminując dodatkowy narzut na zarządzanie serwerem. Najważniejszymi wadami tej wersji są: limit maksymalnej wielkości pliku 500 GB, ograniczone wsparcie dla *SQL Server Management Studio* (domyślnie zarządzanie jest dostępne poprzez dedykowaną witrynę i wymaga wtyczki *Silverlight*), utrudniona migracja danych, brak wsparcia dla *Windows Authentication*, czy wymóg tworzenia kluczy głównych na wszystkich tabelach. Warto również wspomnieć, że nie wszystkie komendy *Transact-SQL* są wspierane przez *Azure SQL*. Przykładem jest komenda USE , która pozwala nam na przełączanie się pomiędzy bazami danych. W wielu zastosowaniach biznesowych problematyczny może być również brak *SQL Agenta*. Największymi zaletami są wysokie SLA na poziomie 99,99%(dopuszczające niedostępność usługi w skali roku przez maksymalnie 52,56 minuty) oraz stosunek ceny do jakości. Ważnym aspektem jest również szybkość wprowadzenia nowych baz danych do życia.

SQL Server zainstalowany jako odrębna instancja, na dedykowanej maszynie nie ma takich ograniczeń jak *Azure SQL*, co wiąże się z dużo wyższymi kosztami i bardziej skomplikowanym zarządzaniem całym serwerem. Taka instancja pozwala na łatwą migrację obecnych baz danych oraz łatwość przenoszenia do chmury, jednocześnie zmniejszając koszt utrzymania fizycznych maszyn. W przypadku posiadania danych po wyżej 500 GB jest jedyną dostępną opcją. *Microsoft* udostępnia odpowiednie szablony nowych wirtualnych maszyn odpowiednio z wersjami *SQL Web*, *Standard* oraz *Enterprise*. Warto nadmienić, że SLA dla tej opcji jest mniejsze niż dla *Azure SQL* i wynosi 99,95% (4,38 godziny niedostępności).

4.3.2 Bazy NoSQL

Bazy *NoSQL*[16] w niezwykle krótkim czasie stały się podstawową platformą dla aplikacji, które generują ogromne ilości danych. Przykładem zastosowania jest np. giełda papierów wartościowych. Na rynku dostępnych jest mnóstwo różnych serwerów *NoSQL*, najpopularniejsze to *MongoDB, RavenDB, CouchDB* czy *HBase*.

Do czego wykorzystywać bazy *NoSQL*? Do zastosowań, gdzie relacyjność bazy danych nie jest potrzebna lub wręcz powoduje problemy. W przypadku, w którym mamy do przechowania bardzo duże ilości danych przekraczających rozmiarem setki petabajtów. W takich zastosowaniach relacyjne bazy skalują się bardzo słabo. Podstawowe zapytania mogą być wykonywane w miarę szybko, ale zapytania z wykorzystaniem funkcji *JOIN* na rozproszonej relacyjnej bazie zupełnie się nie skalują. Innym powodem do użycia *NoSQL* jest przechowywanie danych w formacie *JavaScript Object Notation*(JSON) lub grafów. Oczywiście bazy relacyjne nadają się do przechowywania takich danych, natomiast bazy *NoSQL* są do tego celu stworzone i znacznie ułatwiają np. wyszukiwanie danych w zbiorach. *NoSQL* pozwala na przechowywanie dość specyficznych typów danych, które nie wpasowują się w relacyjny model.

Skalowalność *NoSQL* jest na tyle duża, że w przypadku ogromnej ilości danych, wydajność odczytu jest nawet kilkuset krotnie większa niż w relacyjnej bazie.

Aby łatwiej zrozumieć ten problem, dane zostały podzielone na dwa typy: dane użytkowe, które są zapisywane i odczytywane przez podstawowe funkcjonalności aplikacji oraz dane analityczne, które są najczęściej uzupełniane okresowo, nie muszą być aktualizowane na bieżąco i służą głównie do odczytu. Przykładem danych analitycznych jest raport pokazujący zmianę cen towaru na przestrzeni ostatnich lat. Ze względu na duże przedziały czasowe i często ogromną liczbę rekordów, wydajność w kreowaniu takich raportów odgrywa krytyczną rolę. Przykładem usługi, która idealnie nadaje się do przetwarzania danych analitycznych jest *HDInsight* oparty o *Apache Hadoop*, dzięki skalowalności *Microsoft Azure* potrafi przetwarzać dowolną ilość danych. W prosty sposób integruje się z *Microsoft Excel* i może być oprogramowane przy użyciu najpopularniejszych języków programowania.

Bazy *NoSQL* możemy podzielić na cztery typy:

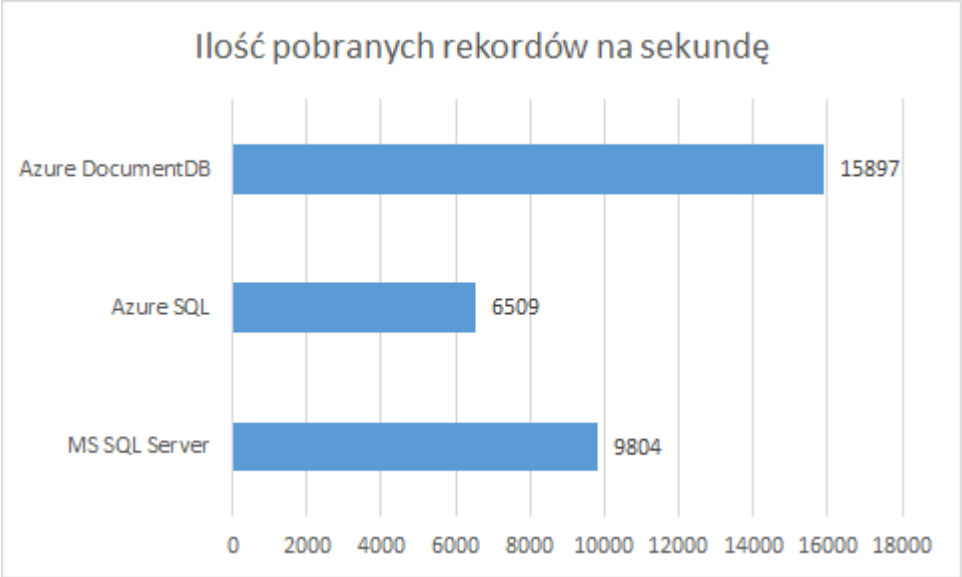
1. Baza typu klucz/wartość - najbliższa słownikom, które przechowują adresy danych w postaci unikatowych kluczy.
2. Baza dokumentowa - najczęściej przechowująca pary klucz/wartość w postaci *JSON, BSON* lub podobnych. W odróżnieniu do baz typu klucz/wartość możemy wykonywać zapytania, w których wykorzystujemy konkretne klucze, które muszą być unikatowe.
3. Baza grafów - wyspecjalizowany typ, stworzony do przechowywania danych o dużej ilości połączeń.
4. Baza typu *Column family* - każdy obiekt w tej bazie jest przechowywany jako klucz/wartość, z tym, że wartość zawsze jest zbiorem kolumn. Analogicznie do relacyjnej bazy danych, *Column family* jest tabelą, a każda para klucz/wartość rekordem.

Żeby lepiej zobrazować różnice w wydajności pomiędzy bazami danych *MS SQL Server* oraz *Azure DocumentDB,* poniżej znajduje się opis krótkiego testu przeprowadzonego na platformie *Microsoft Azure.*

W celach testowych została użyta podstawowa instancja *Azure DocumentDB, Microsoft SQL Server 2014 Standard* (Na wirtualnej maszynie z 32 GB pamięci RAM, 4 rdzeniami i 1 Gb/s łączu internetowym) oraz *Azure SQL*. Wolumen danych wynosił 100 milionów rekordów, klucz główny był typu *GUID*, a każdy rekord zawierał dziesięć pól typu string po maksymalnie 255 bajtów każdy. Wydajność była mierzona podczas odczytu oraz zapisu danych.

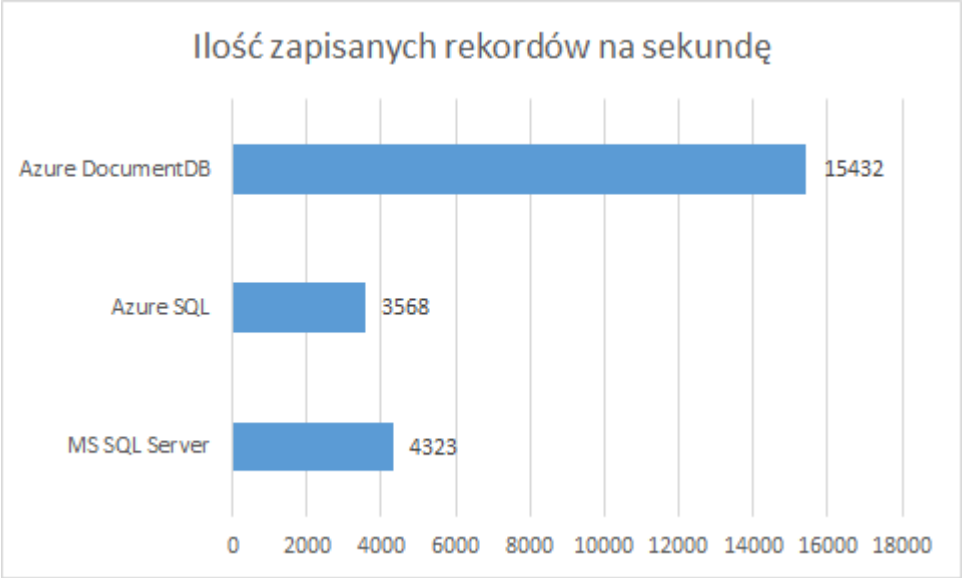
Przetestowano następujące scenariusze (im więcej operacji na sekundę tym lepiej):

1. Pobranie wszystkich danych.



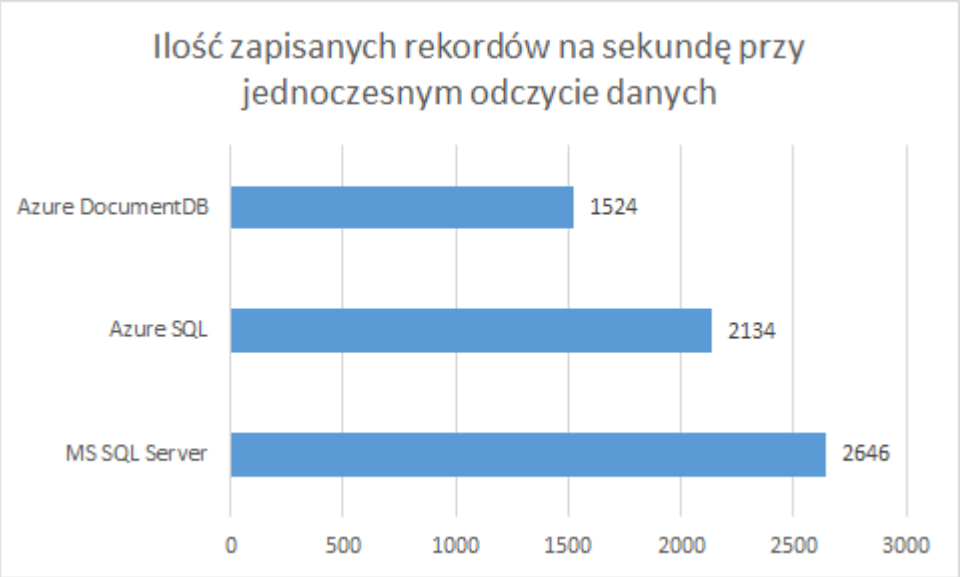
Rysunek 15. Ilość pobranych rekordów na sekundę

2. Załadowanie 100 milionów rekordów do bazy.



Rysunek 16. Ilość zapisanych rekordów na sekundę

3. Zapisywanie rekordów z jednoczesnym odczytem danych.



Rysunek 17. Ilość zapisanych rekordów na sekundę przy jednoczesnym odczycie danych

Podsumowując wyniki testów, *Azure DocumentDB* świetnie sprawdza się podczas zapisywania i odczytu danych, pod warunkiem, że nie odczytujemy danych podczas zapisu i odwrotnie. Zapis danych zdecydowanie przewyższa bazy relacyjne. Jeżeli nasza aplikacja jest typowym systemem biznesowym, np. sprzedażowym, ze zbilansowaną ilością zapisów oraz odczytów, to najlepszym rozwiązaniem jest skorzystanie z *Microsoft SQL Server*. *Azure SQL* jest plasuje się w środku wyników i pomimo dużo niższej ceny nie odbiega znacznie od *MS SQL Server*.

4.4. Pamięć podręczna w chmurze

Pamięć podręczna jest szczególnie przydatna przy pobieraniu tych samych danych wielokrotnie, w krótkich przedziałach czasowych. W tym rozdziale opisano najpopularniejszy sposób przechowywania danych jakim jest *Redis*. Przedstawiono jego wydajność na tle innych rozwiązań oraz dostępne na rynku alternatywy.

4.4.1 Redis, czy warto wykorzystywać jako silnik wyszukiwarki pełno tekstowej?

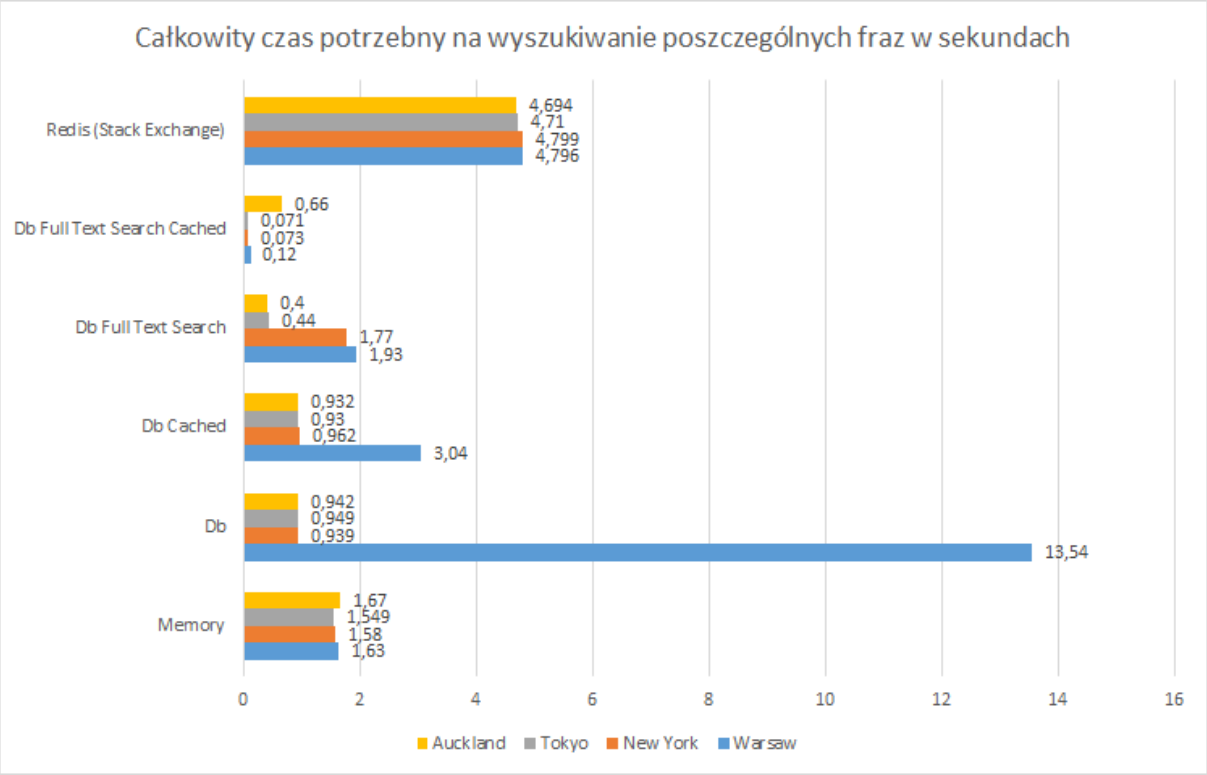
Redis[17] jest odmianą bazy dokumentowej. W większości wypadków jest wykorzystywany jako pamięć podręczna, nietrwała. Wielu ekspertów nie zalicza przez to *Redis'a* do baz danych. Zaletą takiego podejścia jest pominięcie dysku twardego podczas wykonywania operacji *CRUD*, co wielokrotnie zwiększa wydajność. Oczywiście niesie to za sobą ryzyko utraty danych. W większości wypadków *Redis* jest używany tylko i wyłącznie jako pamięć podręczna. Do pomiaru wydajności użyto implementacji *Stack Exchange* dla platformy *.NET*.

Parametry maszyny wirtualnej są następujące: 32 GB pamięci RAM, 4 rdzenie i 1 Gb/s łącze internetowe. Dla zwiększenia skuteczności maszyn, stworzono odrębną maszynę dla *Redis'a* oraz *MS SQL Server*.

Ilość danych wynosiła 10 milionów rekordów i były to dane adresowe miast z całego świata. Testy zostały przeprowadzone z wykorzystaniem odpowiednio: pamięci operacyjnej, *MS SQL Server* oraz *MS SQL Server Full Text Search*. *Redis* był skonfigurowany w sposób domyślny, a zapytanie

zostało zbudowany przy użyciu komendy *SCAN*. *MS SQL Server* był testowany w wersji na zimno (zaraz po starcie serwera) oraz z wykorzystaniem pamięci podręcznej. Zapytanie w każdym przypadku miało znaleźć wszystkie rekordy zawierające frazy: Warsaw, New York, Tokyo oraz Auckland.

Wyniki przedstawiono na rysunku 18.



Rysunek 18. Całkowity czas na wyszukiwanie poszczególnych fraz, mierzony w sekundach

Najlepszy okazał się *Full Text Search* zaimplementowany w *MS SQL Server*, szczególnie z wykorzystaniem pamięci podręcznej, na drugim miejscu znalazła się podstawowa wersja zapytania bez zainstalowanego *Full Text Search*. Wydajność *Redis'a* nie okazała się zadowalająca i pomimo kilku krotnego testowania nie uległa zmianie. Wyszukiwanie z pamięci operacyjnej było zrównoleglone, jako ciekawostkę warto potraktować czas operacji potrzebny na wyszukiwanie poszczególnych fraz w *Redisie* z wykorzystaniem komendy *KEYS* (nie jest zalecana w środowisku produkcyjnym). Wykorzystując tę komendę Redis przyspieszył i zakończył wyszukiwanie frazy Auckland w czasie 0,27 sekundy, czyli prawie osiemnaście razy szybciej.

Podsumowując, *Redis* doskonale sprawdza się jako pamięć podręczna i w scenariuszach, gdzie dane szybko się zmieniają, ale ich ilość nie przekracza bezpiecznych 53 GB pamięci RAM (obecne maksimum na platformie Azure).

4.4.2 Alternatywy dla Microsoft Azure Cache

Microsoft Azure Cache wykorzystuje *Redisa,* ale na rynku można znaleźć wiele ciekawych alternatyw, które mogą być tańsze oraz wydajniejsze. Podobne funkcjonalności oferuje Cassandra, *MongoDB, ElasticSearch, CouchDB, Riak, Couchbase* i wiele innych. Poniżej przedstawiono trzy najciekawsze alternatywy.

Cassandra została napisana w *Javie* i jest rozprowadzana na licencji *Apache*. Posiada własny język zapytań *CQL3*, które jest bardzo podobny do *SQL*. *CQL3* został stworzony do wykonywania zapytań na serwerach rozproszonych, a co za tym idzie ma kilka istotnych ograniczeń.

Najważniejszymi są: brak *JOINów* oraz funkcji agregujących, ponieważ nie są skalowalne. Wyszukiwanie danych odbywa przy wykorzystaniu kluczy lub ich zakresów, a przechowywane dane mogą mieć ustawiony czas życia. Zaawansowane funkcjonalności są udostępniane dzięki wykorzystaniu *Apache Hadoop. Cassandra* świetnie sprawdza się w systemach opartych o chmurę.

Drugą alternatywą napisaną w Javie jest *ElasticSearch,* stosowany głównie w zaawansowanych wyszukiwarkach. Udostępniony jest na licencji *Apache,* przechowuje dokumenty w formacie *JSON,* zapewnia również wersjonowanie przechowywanych danych. Umożliwia sortowanie w oparciu o punktację, którą sami możemy definiować. Świetnie się sprawdza przy wyszukiwaniu po geo lokalizacji. Główną wadą jest niewielkie wsparcie społeczności.

Ostatnią alternatywą o której warto wspomnieć jest *Riak.* Napisany w *Erlangu, C* oraz JavaScripcie, oferowany na licencji *Apache*. Przechowuje dane w postaci blobów. Zapytania mogą być budowane w oparciu o *Erlanga* i *JavaScript*. Dostępny w dwóch wersjach darmowej i płatnej. *Riak* udostępnia wyszukiwanie pełnotekstowe oraz indeksowanie, świetnie się sprawdza w systemach, które muszą mieć maksymalną odporność na błędy, w fabrykach lub na giełdzie. Co ciekawe, poprzez odpowiednią wtyczkę może integrować się z *Redisem.*

4.5. Ciągła integracja w oparciu o chmurę

Próbując zoptymalizować codzienne, powtarzalne zadanie wielokrotnie mieliśmy okazje tworzyć skrypty automatyzujące naszą pracę. W tym rozdziale pokazano jak podejść do takiej automatyzacji w przypadku wdrażania nowych produktów na środowiska testowe, jak również na ich produkcyjne odpowiedniki. Przedstawiono również ciekawą alternatywę dla płatnych rozwiązań jakim jest *Visual Studio Online*, które ma na celu usprawnienie prowadzenia projektu oraz organizację i wsparcie pracy zespołów deweloperskich. Narzędzia zaprezentowane poniżej są oparty tylko i wyłącznie o rozwiązania chmurowe.

4.5.1 Visual Studio Online

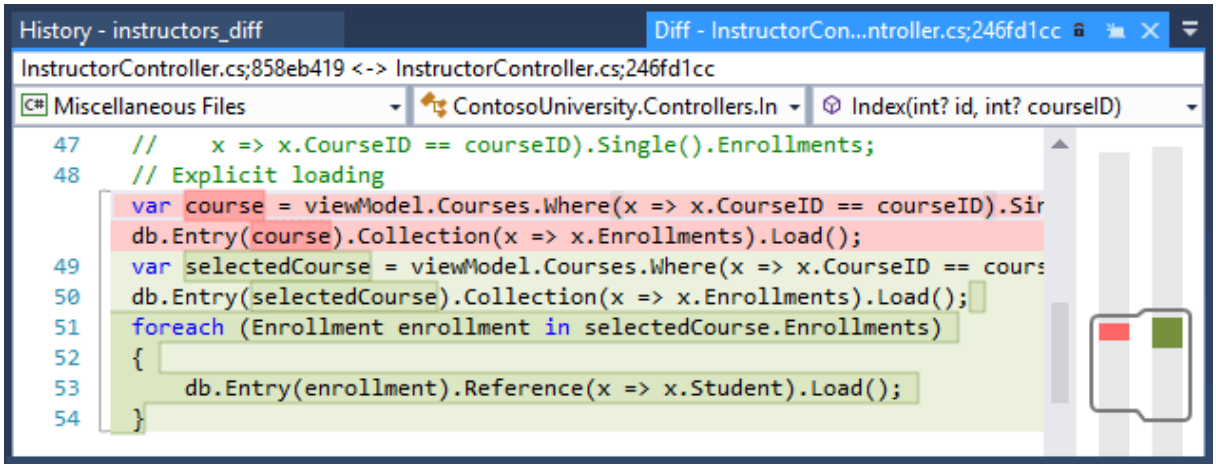
Visual Studio Online(VSO), wcześniej znane jako *Team Foundation Server* zapewnia usługi oparte o chmurę, które mają na celu wspieranie zespołów deweloperskich podczas pracy nad projektem. VSO świetnie integruje się z *Visual Studio*, jak również *Eclipsem*. Zapewnia darmowe repozytoria kodu, wsparcie do obsługi zgłaszanych bugów oraz planowania pracy zgodnie z metodyką *Agile*. Wspiera ciągłą integrację, a dla zespołów maksymalnie pięcioosobowych jest całkowicie darmowy

Wersjonowanie naszego kodu zapewnia *Git* oraz *Team Foundation Version Control*(TFVC). Na potrzeby prototypu został wykorzystany *Git*, ponieważ TFVC jest z założenia bardzo rozbudowany i stworzony do projektów z większą ilością użytkowników. Rysunek 19 przedstawia porównanie kodu wersji bieżącej oraz historycznej pliku bezpośrednio w *IDE Visual Studio*. Dzięki możliwości zautomatyzowania budowania projektu, po każdorazowej zmianie kodu w repozytorium, łatwiej jest utrzymać porządek w projekcie. Automatyzacja uruchamiania testów po pomyślnej kompilacji pozwoli na uniknięcie pomyłek, a dodatkowo odciąży maszynę na której pracujemy. Jest to szczególnie ważna funkcjonalność przy bardzo dużych projektach.

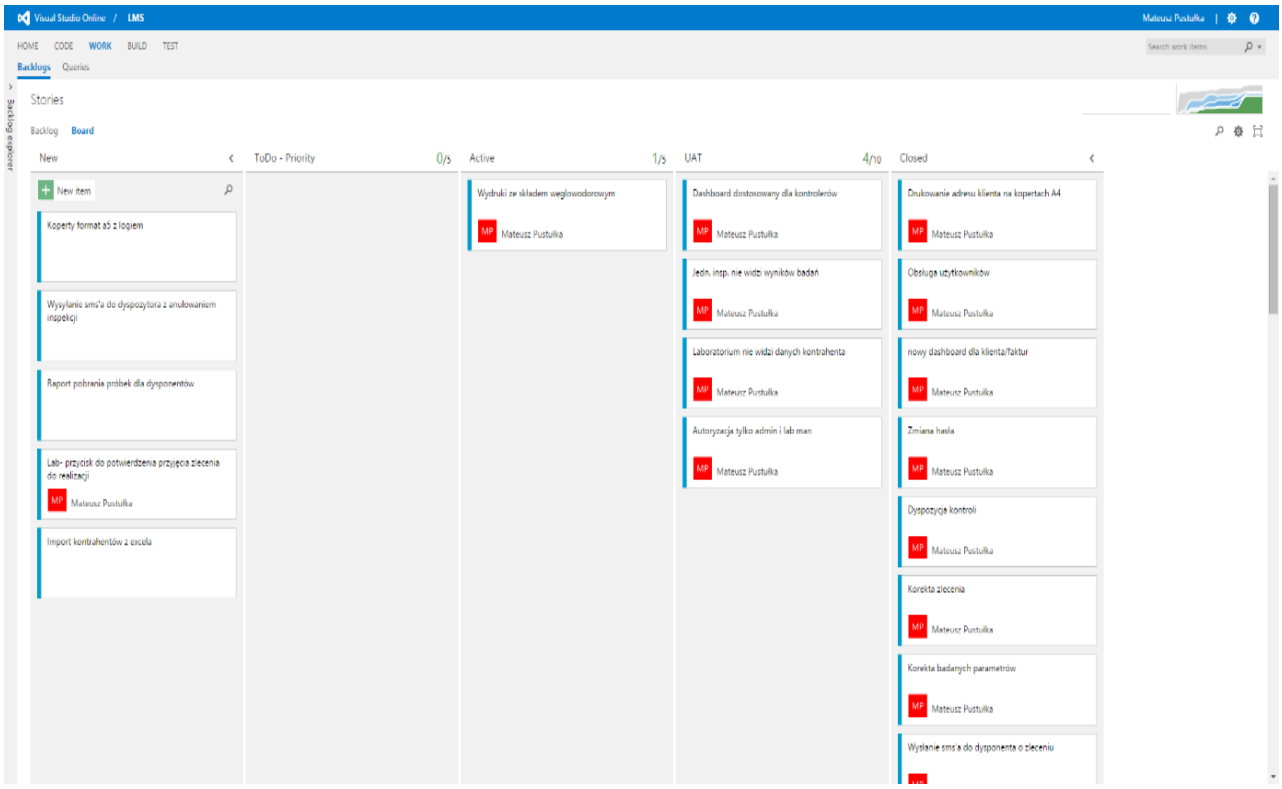
Narzędzia *Agile* udostępniane w ramach VSO pozwalają na tworzenie najprzeróżniejszych tablic, również kanbanowych, jak na rysunku 20. Wszystkie kolumny są edytowalne i łatwo można dostosować wygląd naszej tablicy do naszych potrzeb. Narzędzia *Agile* pozwalają również na łatwe tworzenie nowych zadań, priorytetyzowanie oraz ustawianie takich parametrów zadania jak pracochłonność. Przeciągając i upuszczając poszczególne zadania możemy w łatwy sposób łączyć je w większe grupy funkcjonalne oraz przypisywać je do konkretnych osób. Oczywiście mamy do dyspozycji rozbudowany panel administracyjny oraz raportowy. Dzięki temu możemy zwizualizować na wykresach postępy pracy zespołu i znaleźć potencjalne problemy. Rozbudowany *dashboard* pozwala nam na szybsze przeprowadzanie codziennych spotkań projektowych.

VSO integruje się z większością znanych serwisów wspomagających prowadzenie projektu, takich jak *Attlasian HipChat*, *Trello*, *Jenkins* czy *GitHub*.

Jedyną wadą VSO w stosunku do pełnopłatnej wersji instalowanej na serwerze dedykowanym jest brak zaawansowanego modułu *Business Intelligence* oraz wsparcia dla *SharePointa*.



Rysunek 19. Porównanie kodu bieżącego z historycznym [28]

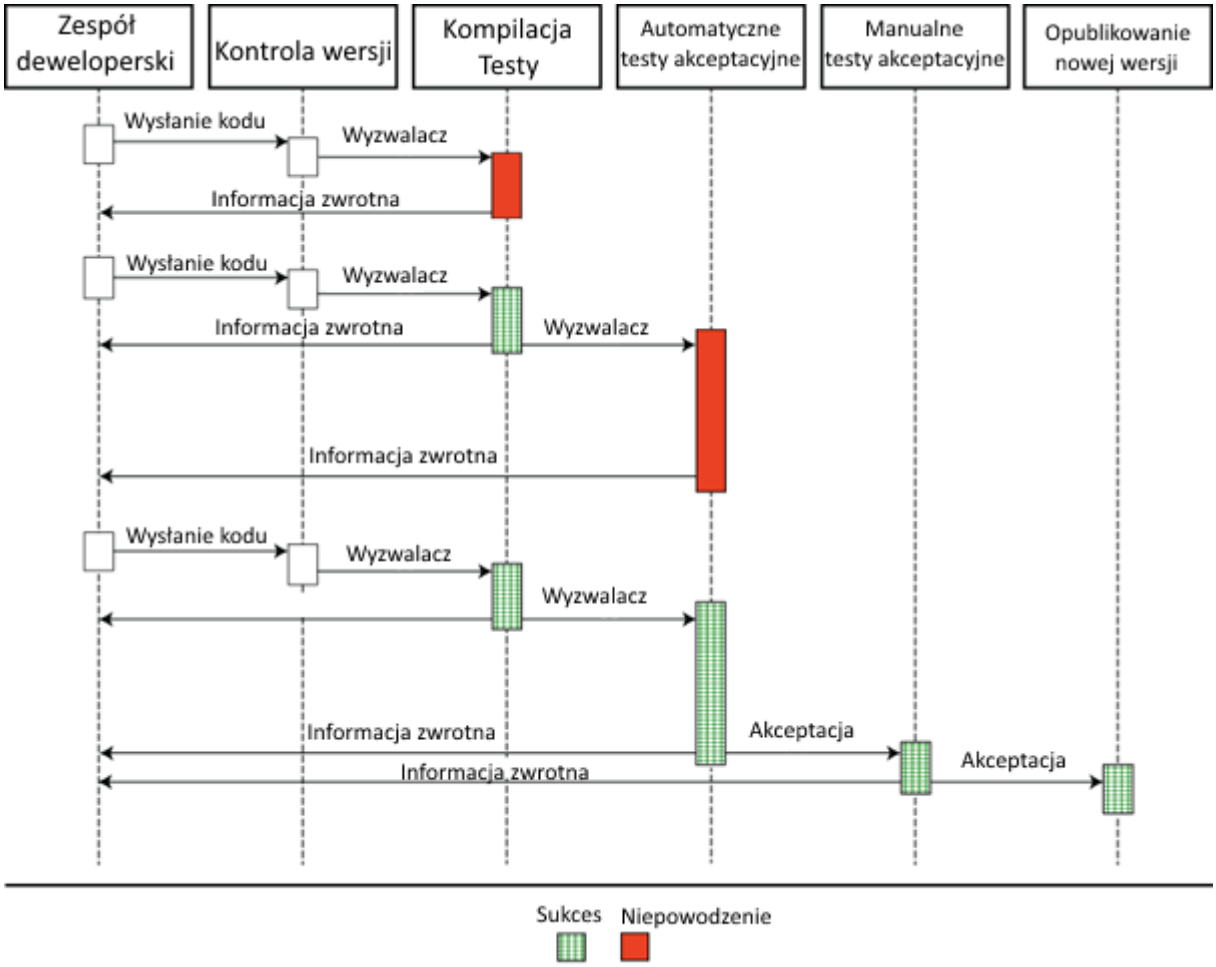


Rysunek 20. Tablica kanbanowa

4.5.2 Automatyzacja budowania i wdrażania projektu

Czym jest ciągła integracja? Jest to podejście do tworzenia oprogramowania, w którym członkowie zespołu integrują swoją pracę bardzo często, przynajmniej raz dziennie. Przekłada się to w dużych zespołach na kilkanaście integracji dziennie w obrębie jednego repozytorium. Każda integracja jest automatycznie weryfikowana wg określonych zasad, najczęściej jest to kompilacja projektu, uruchomienie testów i wdrożenie na odpowiednie środowisko. Dzięki takiemu podejściu szybciej możemy wychwycić potencjalne błędy i zredukować czas potrzebny na wdrażanie zmian[19].

Ciągła integracja została wymuszona przez klienta biznesowego. Zwiększająca się presja na zespołach deweloperskich wymogła stworzenie zautomatyzowanego procesu wdrażania zmian. W większości przypadków stworzenie nowej wersji już działającego systemu oraz wdrożenie na środowisko produkcyjne zajmuje cały dzień, czasami cały weekend i wymaga wsparcia administratorów systemów, dewelopera oraz powoduje problemy z dostępnością usługi. Stała integracja ma na celu zautomatyzowanie tego procesu w maksymalnym stopniu poprzez wersjonowanie kodu, automatyzacje wykonywania testów, a kończąc na tworzeniu maszyn wirtualnych. Na rysunku 21 przedstawiono przykładowy proces wdrażania nowej wersji systemu. Cały proces zaczyna się na integracji nowego kodu źródłowego z już istniejącym na repozytorium. Następnie uruchamiana jest kompilacja systemu oraz testy automatyczne. W międzyczasie mogą być tworzone nowe środowiska testowe, a po pomyślnej kompilacji i przejściu testów automatycznych nowa wersja systemu może być wysłana do testów akceptacyjnych. Po zatwierdzeniu testów środowisko testowe może być podniesione do rangi produkcyjnego (nie musi), po czym następuje przełączenie pomiędzy serwerami i nowa wersja systemu jest wdrożona.

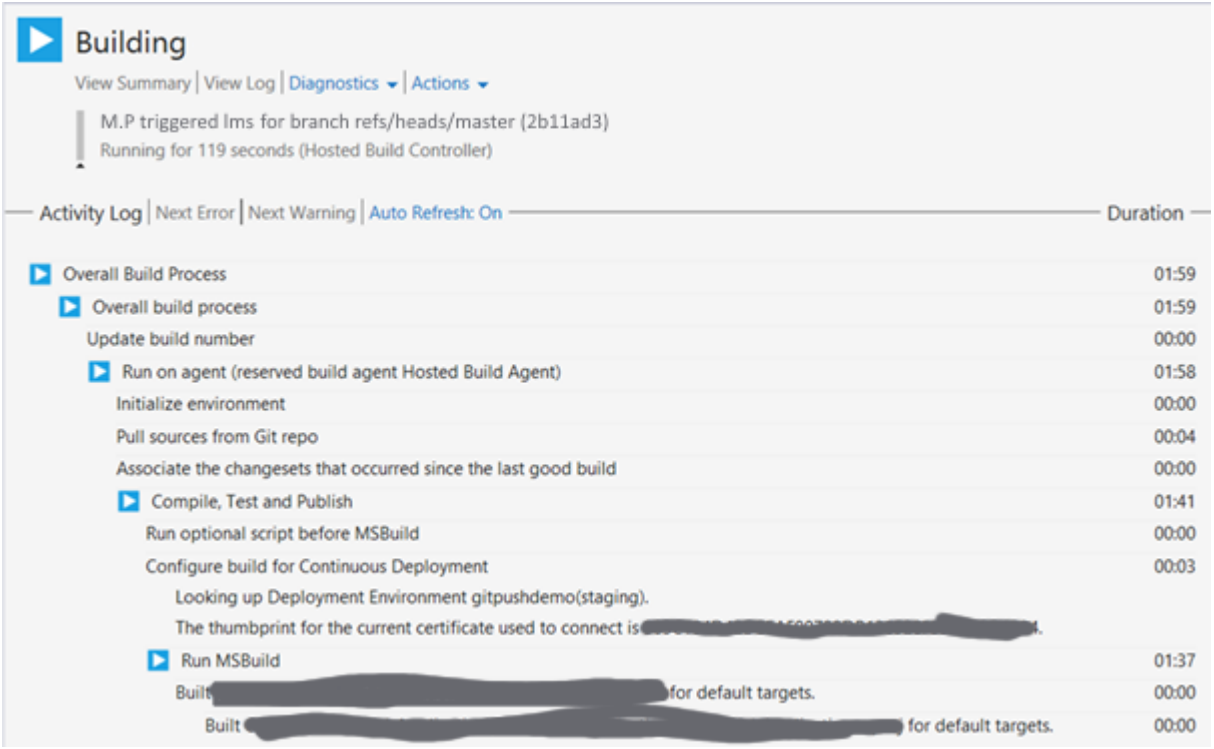


Rysunek 21. Przykładowy proces wdrażania nowej wersji systemu [19]

Ważnym elementem ciągłej integracji są środowiska, na których nasza aplikacja będzie działać. Dzięki zastosowaniu usług chmurowych, niewielkim kosztem możemy tworzyć maszyny testowe, które są odpowiednikiem naszych maszyn produkcyjnych i po pomyślnym wdrożeniu możemy wyłączyć stare wersje systemu oszczędzając czas i pieniądze. Zalecana jest ręczna weryfikacja oraz zapewnienie wsparcia od zespołu deweloperskiego podczas wdrażania nowych wersji produkcyjnych, w przypadku wersji testowych cały proces może odbywać się automatycznie. Oczywiście nic nie stoi na przeszkodzi żeby cały cykl był zautomatyzowany, ale lepiej być przygotowanym na ewentualne problemy.

Dlaczego ciągła integracja lepiej sprawdza się w chmurze? Ponieważ koszt i czas przygotowania odpowiednich serwerów jest wielokrotnie mniejszy niż zorganizowanie odpowiedników fizycznych. Dla przykładu koszt najtańszej maszyny wynosi jednego centa na godzinę, a stworzenie takiej maszyny zajmuje maksymalnie pięć minut. Najważniejsze jest to, że po godzinie możemy taką maszynę wyłączyć bez żadnych konsekwencji. Dodatkowo VSO pozwala na automatyczne tworzenie serwerów kiedy są one potrzebne i automatyczne ich wyłączanie. Wspiera również automatyczne testy obciążeniowe naszej aplikacji, co pozwala nam na szybsze wychwycenie potencjalnych wąskich gardeł. Nic tak nie psuje reputacji witryny jak powolne działanie.

Rysunek 22 przedstawia szczegółowy log stworzony podczas budowania projektu. Widać na nim poszczególne kroki, które wykonuje nasz automat oraz czas potrzebny na ich wykonanie.



Rysunek 22. Log utworzony podczas budowania projektu (zamazano dane wrażliwe)

Wraz z rozwojem ciągłej integracji wypracowano nową rolę w zespole, która ma łączyć w sobie dewelopera oraz administratora systemów. Nazwano ją *DevOps,* co jest połączeniem słów *development* oraz *operations(IT*). Przyjęto, że deweloperzy odpowiadają za wprowadzanie nowych funkcjonalności, a Ops za utrzymanie stabilnego i bezpiecznego środowiska dla aplikacji. Jako, że obie te grupy mają zazwyczaj wiele problemów z dogadaniem się, *DevOps* jest niejako pośrednikiem pomiędzy potencjalnie oddzielnymi działami.

Czy warto wprowadzać ciągłą integrację? W większości wypadków musimy zmienić proces myślenia całej organizacji, pokazać, że taka zmiana to nic złego. Nie jest to łatwy etap ale korzyści, które przynosi są znaczące. Najbardziej opornym środowiskiem są specjaliści IT, czują się zagrożeni, nie chcą tracić kontroli nad serwerami, siecią oraz oprogramowaniem używanym w organizacji. Oczywiście trzeba zrozumieć te obawy, ale automatyzacja ma na celu usprawnienie także ich pracy.

Główne zalety ciągłej integracji:

- Uwolnienie deweloperów oraz administratorów od wykonywania powtarzalnych i monotonnych czynności, co może przełożyć się na wzrost ich zadowolenia i produktywności.
- Automatyzacja pozwala ludziom na poświęcenie większej ilości czasu na samorozwój lub wdrażanie innowacyjnych rozwiązań.
- Automatyzacja zmniejsza czas wdrożenia do niezbędnego minimum.
- Automatyzacja zwiększa jakość procesu wdrażania, ponieważ wymaga ustandaryzowania poszczególnych etapów. Określone zachowania są przewidywalne i mniej podatne na błędy.
- Zmniejszenie kosztów, sprzętu, ludzi itp.

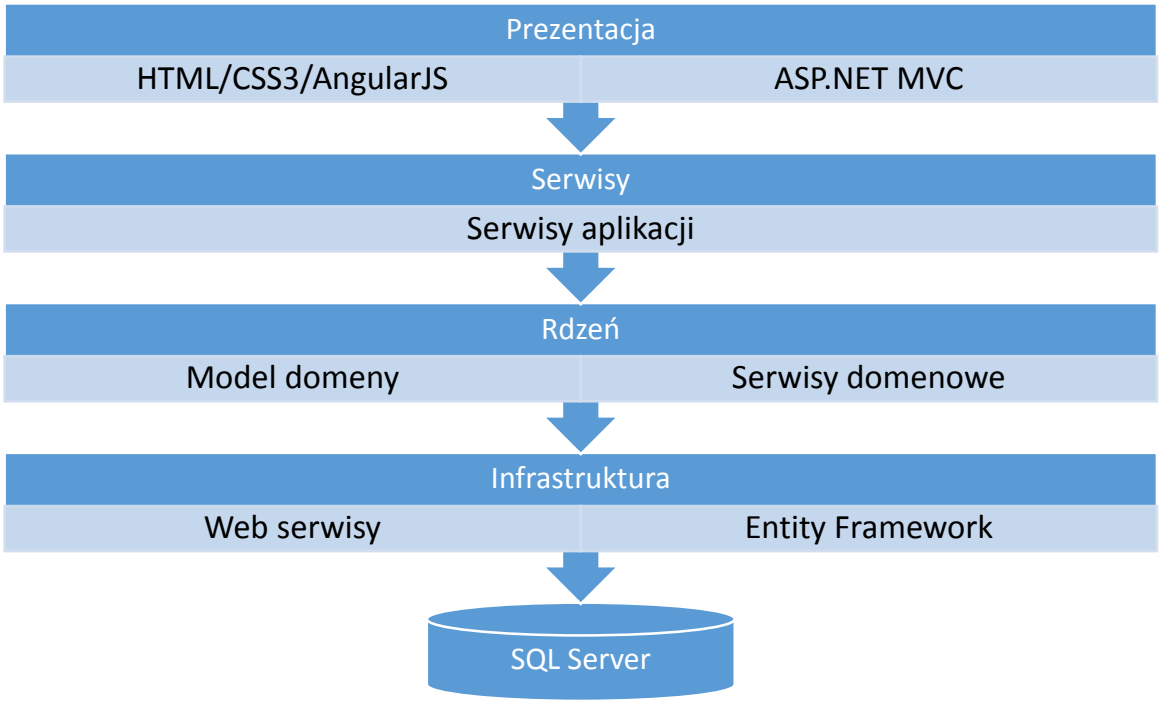
Główne wady:

- Bariera wejścia, wdrożenie całego procesu wymaga czasu i zrozumienia każdej ze stron uczestniczących.
- W bardzo skomplikowanych wdrożeniach, automatyzacja może być problematyczna i wymaga uproszczenia procesu, co może okazać się również zaletą.
- W przypadku błędów w procesie możemy narazić bezpieczeństwo naszych środowisk oraz kodu źródłowego, a dalej idąc danych wrażliwych.

Podsumowując, nie można obawiać się automatyzacji i jeżeli organizacja jest otwarta na jej wprowadzenie to możemy tylko i wyłącznie na tym zyskać.

5. Narzędzia i koncepcje użyte w pracy

W poniższym rozdziale zostały omówione koncepcje[13] zastosowane w prototypie aplikacji oraz krótki opis zastosowanych technologii i wykorzystanych narzędzi. Na rysunku 23 pokazano wysokopoziomową ideę architektury, którą zaimplementowano w prototypie, a w rozdziale 5.1 opisano ją bardziej szczegółowo.



Rysunek 23. Diagram prezentujący architekturę systemu

5.1. Warstwy aplikacji

W tym rozdziale opisano warstwy, które wydzielono na potrzeby tej pracy, zbudowane w oparciu o architekturę DDD opisaną w rozdziale 4.2.1.

5.1.1 Rdzeń

Rdzeń jest odpowiedzialny za reprezentację logiki biznesowej, aktualnej sytuacji biznesu oraz poszczególnych jej reguł. Nawiązując do DDD, powyższa warstwa mapuje się na tzw. warstwę domeny biznesowej.

W tej warstwie definiujemy poszczególne encje oraz reguły biznesowe z nimi powiązane. Encje leżą u podstaw DDD i są jednoznacznie określane przez swoją tożsamość (Id). Sposób zapisu poszczególnych encji jest tylko nakreślony w tej warstwie i zaimplementowany w części dotyczącej Infrastruktury. Wszystkie encje dziedziczą po klasie **Entity**.

5.1.2 Serwisy

Warstwa aplikacji jest odpowiedzialna za komunikację pomiędzy warstwą prezentacji i domeny. W tej warstwie logika biznesowa powinna być zminimalizowana. W idealnym świecie informacje o domenie nie powinny być zawarte w serwisach.

W prototypie warstwa aplikacji jest oparta o uchwyty wyjścia i wejścia, które definiują kierunek komunikacji. Walidacja informacji wejściowych przebiega przed przekazaniem ich do konkretnej metody serwisu.

Dzięki takiej strukturze każda część tej warstwa może być dowolnie zaimplementowana i nie koliduje z innymi. Takie podejście ułatwia rozszerzanie aplikacji oraz testowanie.

W celu ukrycia domeny biznesowej w warstwie aplikacji prototyp korzysta z obiektów typu ***EntityDto*** *(Data Transfer object)*, które mają na celu izolację warstwy prezentacji od rdzenia aplikacji.

5.1.3 Infrastruktura

W warstwie infrastruktury wyodrębniamy trzy główne moduły: autoryzacji, komunikacji oraz trwałości danych.

Moduł autoryzacji jest związany bezpośrednio z serwisami z warstwy aplikacji i umożliwia tworzenie ról oraz szczegółowych zasad dostępu do poszczególnych serwisów oraz każdej z metod.

Moduł komunikacji został stworzony w celu umożliwiania komunikacji pomiędzy poszczególnymi użytkownikami systemu (e-mail, sms) oraz udostępnia podstawowy system raportowania.

Moduł trwałości danych jest oparty o wzorzec repozytoriów i jest dowolnie rozszerzalny. Na potrzeby prototypu został zaimplementowany ***Entity Framework***, ale może on zostać zastąpiony dowolnym innym frameworkiem.

W tej warstwie przechowujemy konkretną implementację interfejsów domeny biznesowej z warstwy rdzenia aplikacji.

5.1.4 Prezentacja

Warstwa prezentacji jest odpowiedzialna za wyświetlanie informacji dla użytkownika oraz interakcję z nim.

W celu dostosowania się do obecnie panujących trendów interfejs użytkownika został stworzony jako aplikacja webowa. Dla lepszego dostosowania się do poszczególnych urządzeń, na których będzie wykorzystywana aplikacja zastosowane zostało podejście responsywności stron internetowych. Dzięki temu aplikacja powinna być zawsze czytelna, niezależnie od tego czy jest wyświetlana na komputerach stacjonarnych, laptopach czy tabletach. Na obecną chwilę nie jest planowana wersja na smartfony.

Warstwa prezentacji została wykonana przy użyciu takich technologii jak : *AngularJS, HTML 5, CSS 3* oraz *ASP.Net MVC/Web API*.

5.1.5 Modułowość projektowanego systemu

Jednym z głównych założeń systemu jest jego rozszerzalność i elastyczność na zmiany. Każda biblioteka znajdująca się w folderze uruchomieniowym aplikacji jest skanowana podczas startu i dzięki wykorzystaniu wstrzykiwania zależności ładowana do pamięci.

Moduły można łączyć ze sobą kaskadowo, lub tworzyć je zupełnie oddzielnie. Nowe moduły mogą, ale nie muszą nadpisywać obecną już logikę biznesową i poszczególne składowe tak warstwy aplikacji jak i infrastruktury.

W przypadku decyzji o rozszerzeniu warstwy prezentacji o część mobilną, można stworzyć oddzielny moduł, który m. in. może wykorzystywać tylko moduły odpowiedzialne za odczyt danych bez możliwości zapisu.

Dzięki modułowej budowie systemu, w każdej chwili możemy przebudować poszczególne warstwy (poza warstwą rdzenia), jako nowe moduły i wybierać, z której implementacji chcielibyśmy korzystać dla poszczególnych klientów.

5.2. Platforma .Net

W poniższym rozdziale opisano platformę .Net, jej historię oraz sposób działania. Opisano również technologię webową opartą na tej platformie oraz środowisko programistyczne ułatwiające tworzenie, testowanie oraz publikację oprogramowania jakim jest *Visual Studio*.

5.2.1 .Net

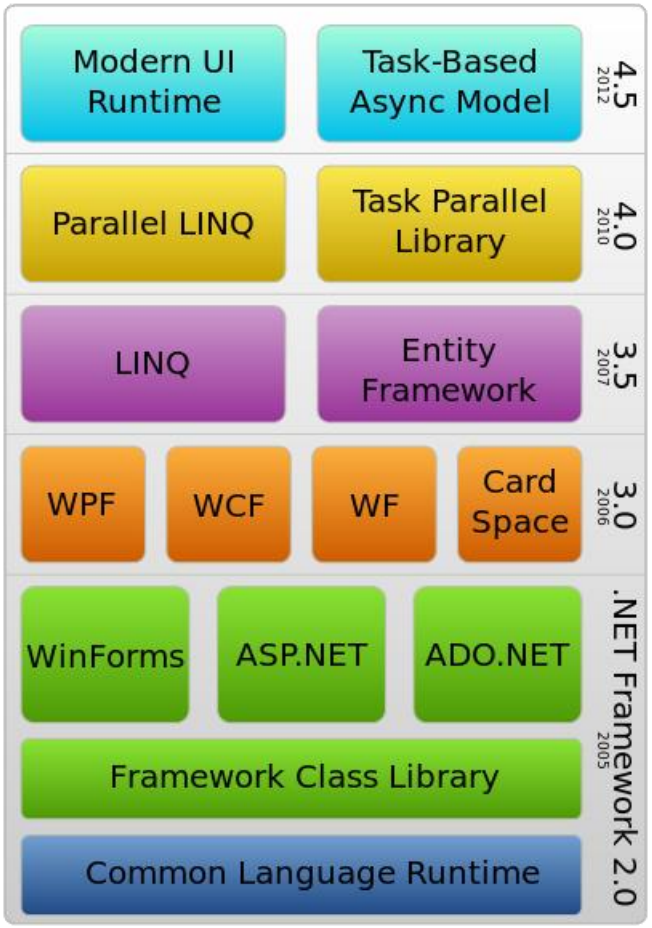
Framework *Net* jest technologią, której korzeni należy się doszukiwać w takich językach jak *C++* oraz *JAVA*. Początkowo *.Net* nazywał się *"Next Generation Windows Services"* i został przemianowany na *.Net* pod koniec roku 2000 wraz z wprowadzeniem pierwszej wersji frameworka.

Głównymi założeniami frameworka są:

- stworzenie środowiska zorientowanego na programowanie obiektowe, które może być uruchamiane niezależnie od sposobu przechowywania kodu aplikacji
- stworzenie środowiska, które zminimalizuje tzw. *dll-hell* związany z wersjonowaniem bibliotek
- stworzenie środowiska, które uprości instalację aplikacji na poszczególnych urządzeniach
- stworzenie środowiska uruchomieniowego, które wyeliminuje problemy z wydajnością aplikacji opartych o języki skryptowe lub interpretowane
- ułatwienie tworzenie różnych typów aplikacji (*web/windows*) oraz zminimalizowanie bariery wejścia podczas nauki

Sam framework może być hostowany przez niezarządzane komponenty korzystające z *CLR (Common Language Runtime)*

Aktualną wersją frameworka w trakcie pisanie tej pracy była 4.5.2, rysunek 24 prezentuje rozwój platformy na przestrzeni lat.

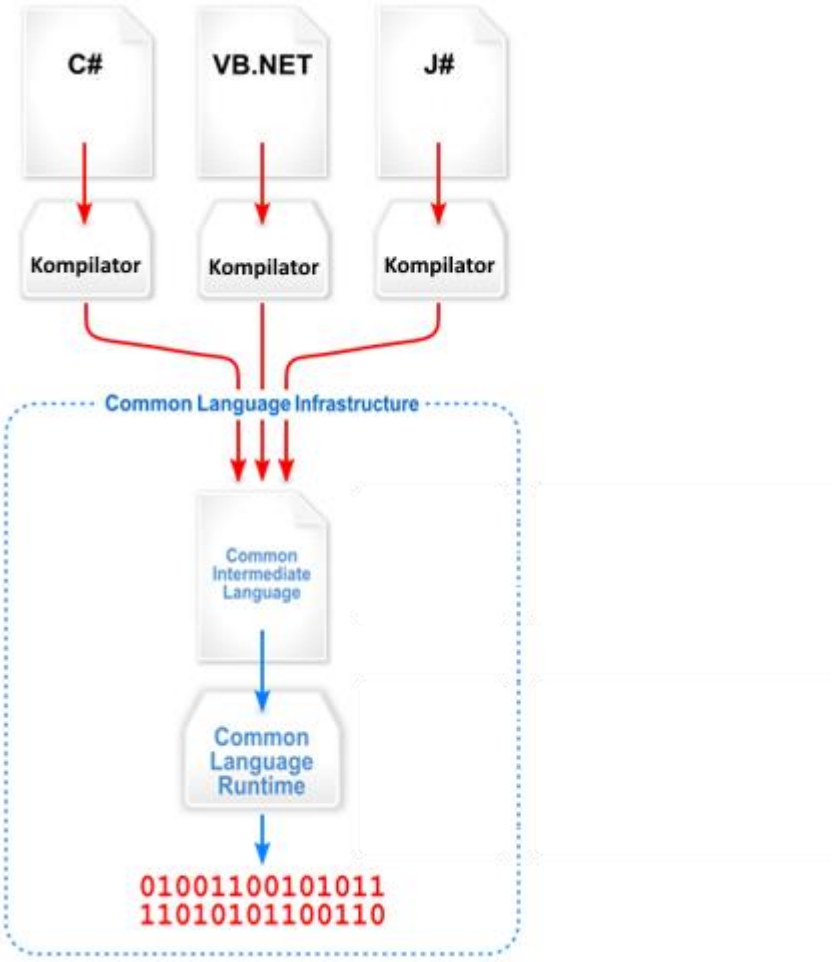


Rysunek 24. Rozwój frameworka na przestrzeni lat [30]

Architekturę .Net można podzielić na kilka pomniejszych składowych:

CLI (Common Language Infrastructure) odpowiadające za udostępnienie niezależnej od języka (*C#, J#, VB. NET*) platformy uruchomieniowej. *CLI* zawiera funkcjonalności związane z obsługą wyjątków, "odśmiecaczem" (*Garbage Collector*), bezpieczeństwem oraz interoperacyjności.

Rysunek 25 prezentuje koncept działania CLI.



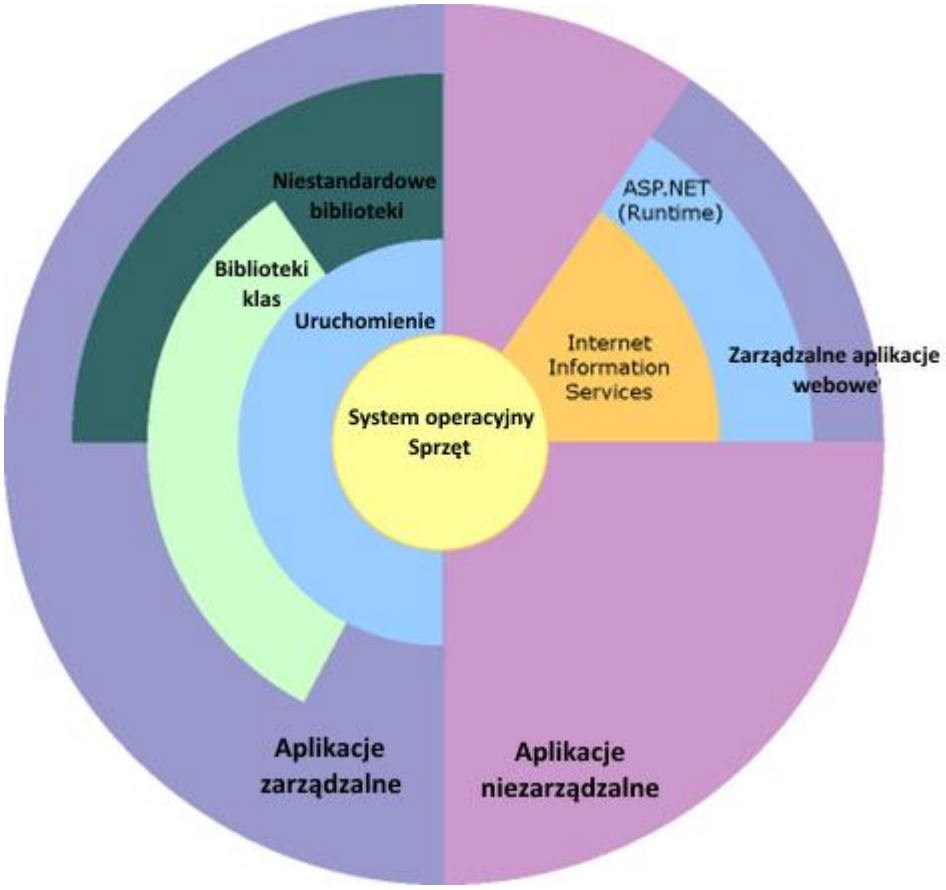
Rysunek 25. Ogólny koncept działania CLI [31]

Biblioteki klas *BCL* (*Base Class Library)* oraz *FCL* (*Framework Class Library) BCL* jest biblioteką bazową i zawiera klasy znajdujące się w przestrzeniach nazw System oraz jej rozszerzeniach takich jak : *Collections*, *Diagnostics*, *IO*, *Security*, *Text* itd. *FCL* rozszerza funkcjonalności dostarczane przez *BCL* o między innymi *LINQ* (*Language integrated query*), *ASP .Net*.

.Net Core od niedawna został udostępniony na zasadach *open-source*, zawiera *CoreCLR* oraz *CoreFX*.

CLR (Common Language Runtime) jest maszyną wirtualną, która zarządza uruchomieniem aplikacji. Podczas uruchamiania aplikacji uruchamia się proces *"just-in-time compilation" (JIT)*, który odpowiada za konwersję skompilowanego kodu do *IL* (*Intermediate Language*). Od wersji 4.5 frameworka, *JIT* może działać na wielu rdzeniach jednocześnie, tym samym przyspieszając start aplikacji. *Microsoft* przemianował w ostatnim czasie *MSIL (Microsoft Intermediate Language*) na *CIL* (*Common Intermediate Language*).

Microsoft, na potrzeby platformy, stworzył dwa pojęcia, kod zarządzalny oraz kod niezarządzalny. Kod zarządzalny nawiązuje do aplikacji napisanych przy użyciu .Net, natomiast kod niezarządzalny odnosi się do aplikacji, które nie wymagają CLR do uruchomienia. Rysunek 26 prezentuje tę koncepcję w relacji do systemu z rozbudowaną architekturą.



Rysunek 26. Relacje pomiędzy aplikacjami zarządzanymi oraz niezarządzanymi [33]

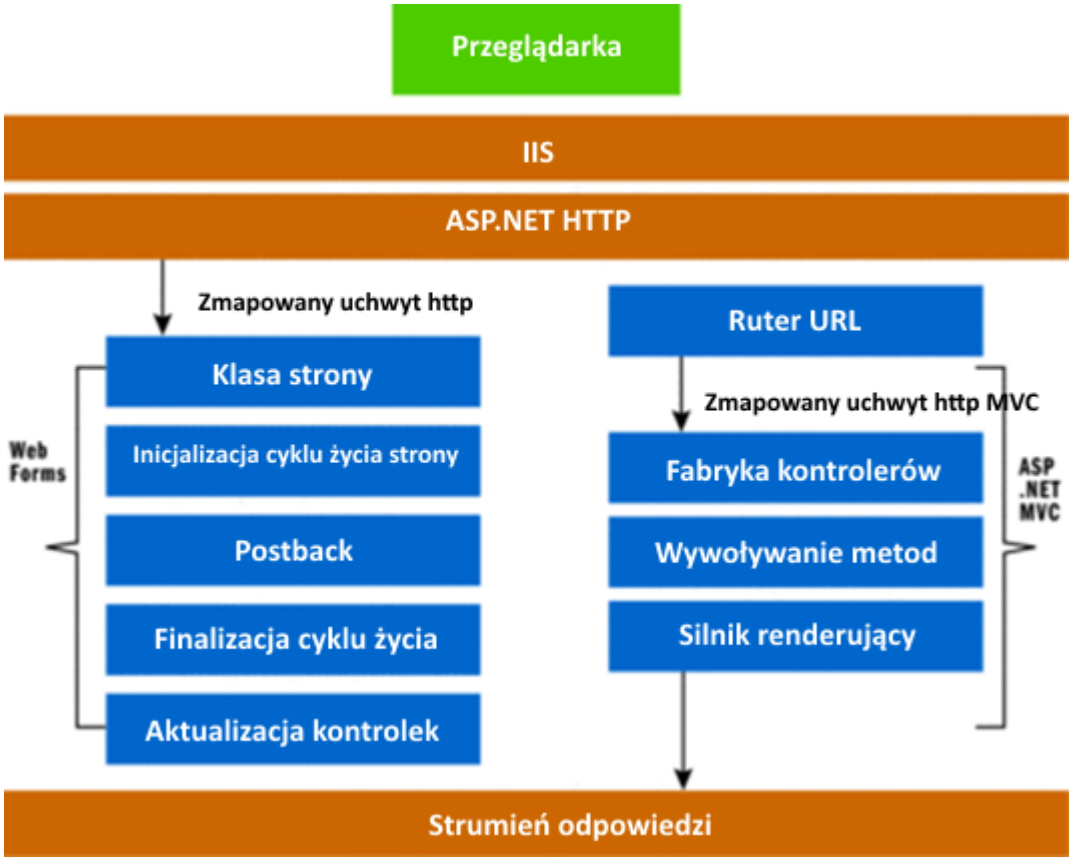
5.2.2 ASP .Net

ASP.Net[5] jest otwartym frameworkiem stworzonym do projektowania aplikacji webowych. Pierwsza wersja została udostępniona w styczniu 2002 wraz z .Net 1.0. Aplikacje w ASP .Net możemy tworzyć w dowolnym języku opartym o CLR. Aktualną wersją platformy jest 4.5, w 2015 ma być udostępniona wersja 5.0 wprowadzająca wiele zmian. Najistotniejszą z nich jest wielo-platformowość oraz korzystanie z nowej platformy "*Roslyn"*. Na obecną chwilę mamy kilka wariacji *ASP .Net*, są to *MVC, Web API* oraz *Web Pages*, które zostaną złączone w jedną platformę nazwaną *"ASP .Net vNext"*.

W początkowych założeniach była prostota przejścia programistów *WinForms* do *WebForms*. Powstała koncepcja *Code-behind*, którą można nazwać koncepcją sterowania aplikacją poprzez zdarzenia, takie jak ładowanie strony, załadowanie strony itp. W założeniu miało to ułatwić oddzielenie warstwy prezentacji od warstwy od logiki systemu.

Rozwinięciem tej koncepcji jest *ASP.Net MVC* (*Model-View-Controller*) zaprezentowane w 2009 roku, które to pozwala na lepsze rozdzielenie warstw aplikacji, jednocześnie usuwając zdarzenia znane z *WebForms*. W tym koncepcie *View* odpowiada za warstwę prezentacji, Model za warstwę logiki biznesowej, a *Controller* jest pośrednikiem pomiędzy nimi. Kontrolery, które zawierają logikę biznesową są nazywane "grubymi" i są uważane za złą praktykę. Początkowo silnikiem renderującym widok był silnik znany z *WebForms*, wraz z wydaniem *MVC 3 Microsoft* wprowadził nowy silnik nazwany *Razor*.

Rysunek 27 prezentuje różnice w renderowaniu stron pomiędzy ASP.Net WebForms, a ASP.Net MVC.



Rysunek 27. Rendering strony, porównanie WebForms z MVC [34]

5.2.3 Visual Studio

Microsoft Visual Studio[14] jest to zintegrowane środowisko programistyczne, tzw. *IDE* (*Integrated development environment*) zbudowane przez *Microsoft.* Obecnie pozwala na tworzenie oprogramowania jedynie na platformy oparte o system operacyjny *Windows*. Wraz z wydaniem wersji 2015 możliwe będzie tworzenie oprogramowania również na inne platformy, a to dzięki decyzji szefostwa firmy, która to uwolniła platformę *.Net.* W chwili pisania tego tekstu firma Microsoft udostępniła wersję otwartą, którą nazwano *.NET Core stack.* Ma to zapoczątkować rozwój aplikacji .Net na innych platformach.

Visual Studio zawiera rozbudowany edytor kodu, który posiada wiele udogodnień. Największym z nich jest *IntelliSense* (*Intelligent sense),* które to ma na celu usprawnić proces tworzenia oprogramowanie poprzez aktywne uzupełnianie składni kodu podczas pisania. Przykład procesu przedstawia Listing 3.

Listing 3. Przykład działania *intellisense*

```
class Test {
       public void it();
       public void that(string what);

      };
```

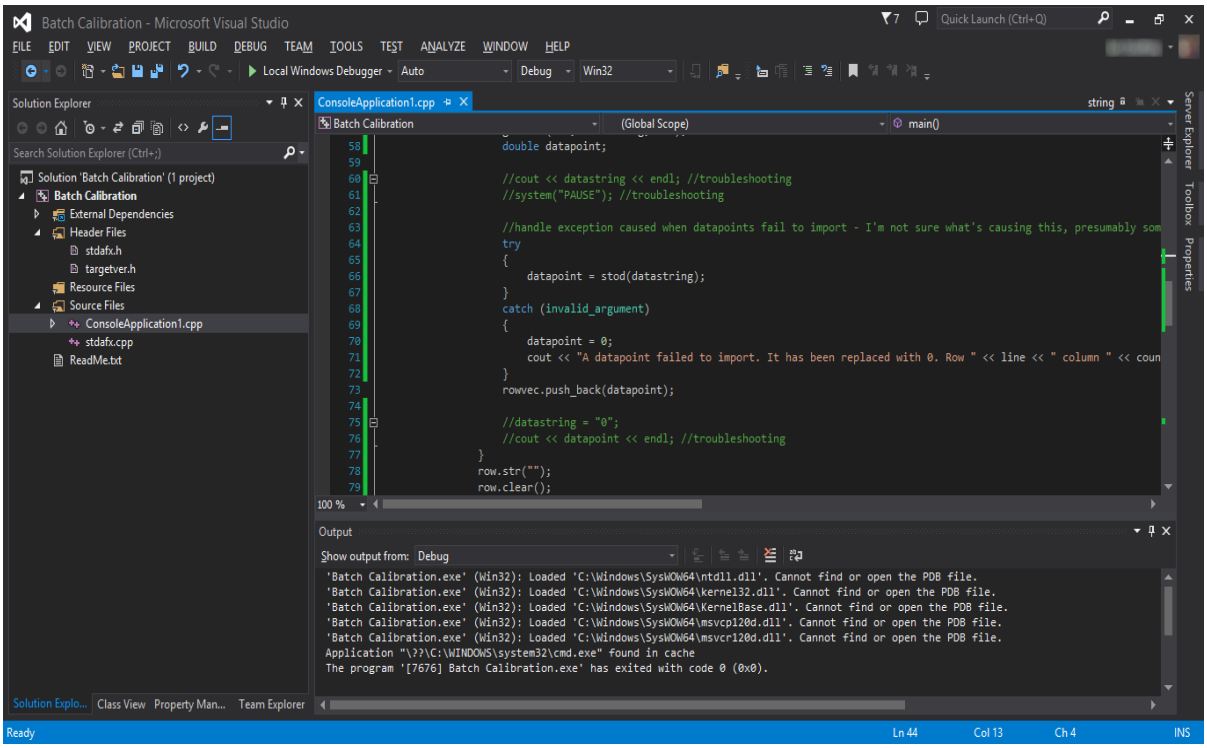
Teraz gdy programista stworzy obiekt klasy Test i zacznie pisać, np. test. to *IntelliSense* automatycznie wylistuje wszystkie dostępne metody oraz atrybuty dla tego obiektu (w tym przypadku metody id, that oraz podstawowe metody dziedziczone z klasy Object).

Visual Studio posiada zintegrowanego debuggera kodu, edytor graficzny dla aplikacji okienkowych oraz webowych, projektanta klas i schematów bazy danych oraz wiele innych. Całe IDE jest pluginowalne i może być z łatwością rozszerzane o nowe funkcjonalności. *Visual Studio* wspiera natywnie repozytoria kodu oparte o *GIT'a* oraz *Team Foundation Server*.

Visual Studio wspiera kilka języków programowania, są to C, C++, C++/CLI, VB.NET, C# oraz język dynamiczny F#. Posiada również wbudowaną obsługę języków front-endowych jak *JavaScript, CSS* czy *HTML/XHTML, XML/XSLT*.

Microsoft wraz z wersją 2013 stworzył specjalną wersję *Community Edition*, która jest odpowiednikiem wersji Professional i jest ona darmowa dla studentów i kontrybutorów otwartego oprogramowania.

Wygląd środowiska jest konfigurowalny i zawiera kilka predefiniowanych ustawień (Web developer, *WinForms* developer itp.), które definiują układ okien. Środowisko posiada również bardzo bogatą kolekcję motywów graficznych tych wbudowanych jak i przygotowanych przez społeczność. Na rysunku 28 przedstawiona jest wersja Dark.



Rysunek 28. MS VS 2013 Dark Theme, widok interfejsu

Visual Studio jest bardzo dojrzałym produktem, wraz z wersją 2010 zostało kompletnie przepisane w technologii *WPF* (*Windows Presentation Foundation*), a rdzeń wzbogacił się o framework *MEF* (*Managed Extensibility Framework*). Pomimo licznych zalet, wiele osób wytyka temu środowisku brak wersji specjalnie dla Mono, pamięciożerność, czy problemy z licencjonowaniem. Wychodząc naprzeciw potrzebom społeczności powstało kilka ciekawych alternatyw. Są to, w kolejności losowej:

- Sharpdeveloper (http://www.icsharpcode.net/),
- Monodevelop (http://www.monodevelop.com/),
- Xamarin Studio (http://xamarin.com/platform).

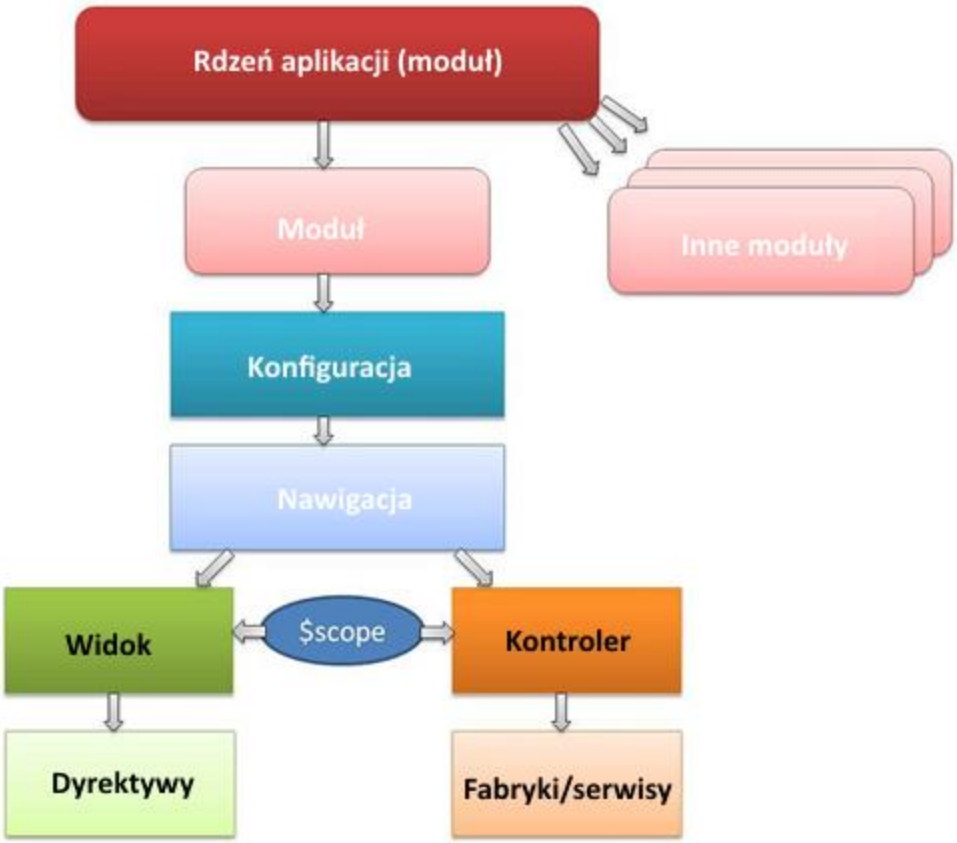
5.3. Single Page Applications

W tym rozdziale opisano dlaczego warto budować aplikacje typu *Single Page*. Zaprezentowano technologie webowe[10], które ułatwiają tworzenie takich aplikacji oraz framework *JavaScript* łączący je wszystkie w całość.

5.3.1 AngularJS

AngularJS (*Superheroic JavaScript MVW Framework*), czyli czym byłby HTML, gdyby został zaprojektowany do budowania aplikacji webowych. Jest to framework *Model-View-Whatever* (MVW) zaprojektowany przez firmę Google, pod przewodnictwem Miško Hevery'ego i Adama Abrons'a w 2009. Projekt został porzucony przez Abrons'a, ale dzięki Hevery'emu i Google zyskał wsparcie Igora Minára i Vojty Jína.

Zasada działania *Angulara* jest prosta i polega na parsowaniu html'a i poszukiwaniu specjalnych tagów (zaczynających się od ng- lub data-ng-). Następnie powstałe po sparsowaniu dyrektywy są wiązane z modelem. Głównym punktem aplikacji jest element opatrzony atrybutem ng-app, który powinien być zadeklarowany tylko raz na aplikację. Na rysunku 29 jest zaprezentowana struktura projektu opartego o Angulara.



Rysunek 29. Struktura projektu [35]

Dlaczego warto korzystać z Angulara? Budując aplikacje webowe, borykamy się z wieloma problemami. Głównym z nich jest statyczność stron *HyperText Markup Language* (HTML), a co za tym idzie manipulowanie elementami *Document Object Model* (DOM), czyli obiektowym modelem

dokumentu, który jest wieloplatformowym sposobem reprezentacji oraz interakcji obiektów w dokumentach HTML, *Extensible Markup Language (*XML*)* oraz *Extensible Hypertext Markup Language* (XHTML*).* AngularJS próbuje sobie z tą niedoskonałością poradzić poprzez:

- wiązanie danych (ang. *Data Binding*) – czyli automatyczna aktualizacja widoku za każdym razem gdy nastąpi zmiana modelu i odwrotnie, gdy zmieni się widok, ma to odwzorowanie na model. Główną zaletą tego rozwiązania jest bezpośrednia eliminacja operowania na elementach DOM.
- Dyrektywy – są to reużywalne szablony zawierające *HTML* i *JavaScript*, które mogą być stworzone specjalnie dla konkretnej aplikacji webowej lub jako biblioteka. Jednym z popularniejszych zestawów dyrektyw jest ***Angular-UI***.
- Kontrolery – jest to obiekt *JavaScriptowy,* definiowany atrybutem ***ng-controller*** lub wstrzykiwany poprzez definicję trasy w serwisie routingowym. Kontroler jest kluczowy w podejściu MVC
- wstrzykiwanie zależności (ang. *Dependency injection*) jest sposobem na deklaratywne opisanie działania naszej aplikacji. Powiązań pomiędzy modułami, serwisami, fabrykami, kontrolerami. DI pomaga również uprościć sposób testowania takich aplikacji.

Angular nie posiada żadnych zależności i jest napisany w czystym *JavaScripcie,* a dzięki modułowej budowie jest dość lekki. Wspiera również minifikację plików *JavaScript.*

Alternatywy*:*

- *DurandalJS* zbudowany w oparciu o *KnockoutJS, Sammy, jQuery* oraz *RequireJS,*
- Backbone.js oparty w głównej mierze o *jQuery* oraz *Underscore,* a został stworzony przez Jeremiego *Ashkenas'a,*
- *EmberJS* stworzony przez *Yehuda Katz.*

5.3.2 HTML 5

Specyfikacja HTML 5[9] została oficjalnie wydana 28 października 2014, a już trwają prace nad wersją 5.1. Specyfikacją HTML zajmuje się *World Wide Web Consortium (W3C),* które to potrzebowało aż siedemnastu lat na wydanie następcy HTML 4.

W założeniach HTML 5 ma być docelowo wybierany do budowy wieloplatformowych aplikacji, by zrealizować te założenia wprowadzono szereg ulepszeń i nowych elementów, takich jak *video, audio* czy *canvas*. Bardzo popularnymi tagami są również *section, article, header, footer* oraz *nav.*

HTML 5 miał zastąpić *Adobe Flash* w aplikacjach webowych dzięki wsparciu strumieniowania audio oraz video, a także wsparciu Skalowalnych Grafik wektorowych (SVG). Nie bez znaczenia był również fakt wsparcia *Steve'a Jobs'a* (*Apple Inc*), który otwarcie krytykował *Flash'a* i wieścił jego rychły koniec. Największy serwis wszelkiej maści filmów *YouTube*, którego właścicielem jest firma Google stopniowo porzuca Flash'a na rzecz HTML 5 i kodeka H.264/MPEG-4 AVC i formatu WebM. Domyślnie w przeglądarce Chrome *YouTube* korzysta z HTML 5.

Krótkie streszczenie nowości HTML 5 prezentuje rysunek 30.

Rysunek 30. HTML 5 cechy[36]

Wdrażanie HTML 5 nie obyło się bez problemów. Twórcy przeglądarek internetowych wielokrotnie inaczej implementowali różne elementy specyfikacji, co doprowadza do sytuacji, gdzie programiści są zmuszeni do dostosowywania jednej aplikacji do różnych przeglądarek. Największe problemy sprawia *Internet Explorer*, dostarczany wraz z systemem operacyjnym i używany w większości korporacji na świecie. W popularnym teście *https://html5test.com/* sprawdzającym kompatybilność przeglądarki ze specyfikacją IE w wersji 9 osiąga jedynie 113 punktów na 555, co czyni go praktycznie bezużytecznym. *Microsoft* chcąc sprostać wymaganiom, postanowił porzucić *Internet Explorera* całkowicie (zostanie tylko dla zachowania wstecznej kompatybilności) i stworzył całkowicie nową przeglądarkę o kodowej nazwie *Spartan.* Najnowsza przeglądarka Chrome(*Google*) osiąga wynik 523 punktów.

5.3.3 Responsywność interfejsu, CSS 3

W parze z rozwojem HTML 5 szedł rozwój arkuszy stylów *(CSS),* który początkowy swój szkic miał już w 1999 roku. CSS jest językiem, który ma zadanie opisać wygląd dokumentów, głównie HTML/XHTML, ale może być także wykorzystywany do opisu dokumentów XML.

CSS został stworzony w celu odseparowania warstwy prezentacyjnej od zawartości dokumentu. Dzięki temu możemy w łatwy sposób tworzyć aplikacje dostosowane do specyficznych upodobań klientów (skórki) lub ułatwiać korzystanie z aplikacji osobom z różnymi schorzeniami psycho-fizycznymi. CSS podobnie jak HTML jest rozwijany przez konsorcjum W3C.

Dzięki CSS 3[11] mamy możliwość tworzenia zaawansowanych grafik oraz animacji. Możemy transformować obiekty i dowolnie je kształtować, nie musimy ładować obrazków na stronie, wystarczy zapisać w arkuszu. Rozwój CSS 3 spowodował spadek popularności Flash'a, a po wycofaniu wsparcia na urządzeniach mobilnych rynek skurczył się do 11,2% wg W3Techs.

Jedną z najciekawszych funkcjonalności CSS 3 są Media queries, używane do wyświetlania zawartości strony dostosowanej do wyświetlacza, na którym to strona jest renderowana. Takie podejście do tworzenia stron nazwano *Responsive web design* (*RWD).* Media queries oficjalnie zostały zarekomendowane przez W3C w 2012 roku.

Przykład użycia przedstawia listing 4 (tablety):

Listing 4. Przykład media query

```
@media (min-width: @screen-sm-min) { ... }
```

Podobnie jak ma to miejsce z HTML 5, nie wszystkie przeglądarki poprawnie wyświetlają elementy zadeklarowane w arkuszu, pomocna okazuje się społeczność i strona *http://caniuse.com/*, na której to znajdziemy odpowiednie informacje dotyczące kompatybilności poszczególnych przeglądarek z deklarowanymi elementami.

6. Prototyp systemu

W niniejszym rozdziale opisany został sposób zrealizowania prototypu oraz rozwiązania, które pozwalają na rozszerzenie konceptu o dodatkowe moduły. Prototyp przyjął nazwę LMS, czyli *Laboratory Management System.*

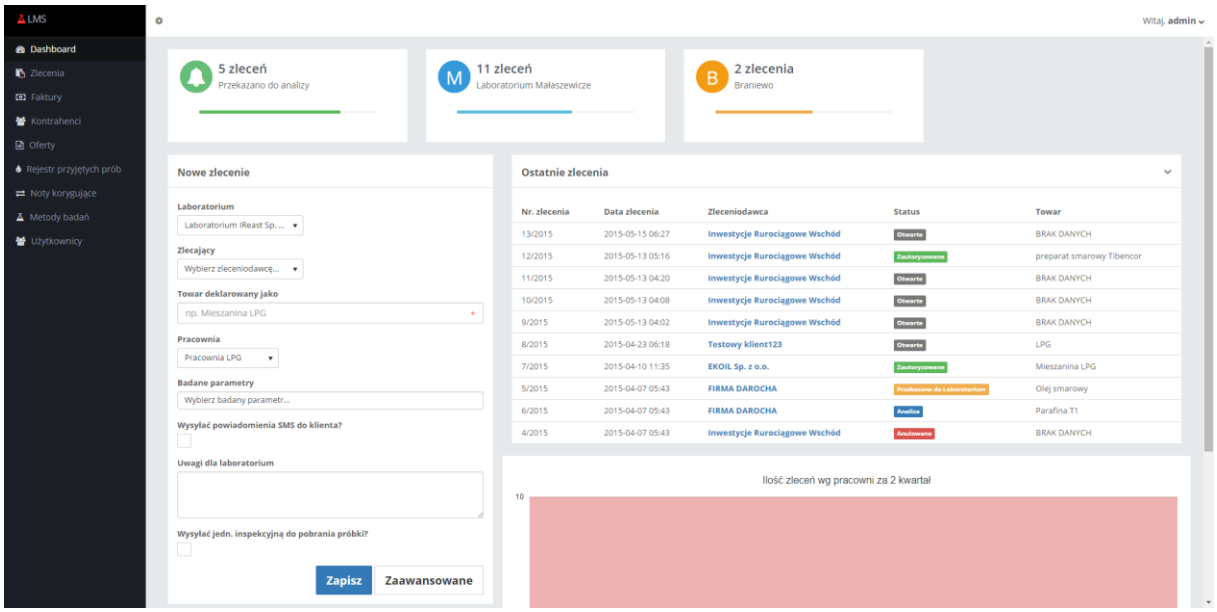
W celach testowych, cały system został opublikowany używając *Microsoft Azure* wykorzystując wirtualne maszyny, *Visual Studio Online* oraz dodatkowe funkcje platformy niezbędne do przeprowadzenia testów.

6.1. Aplikacja webowa

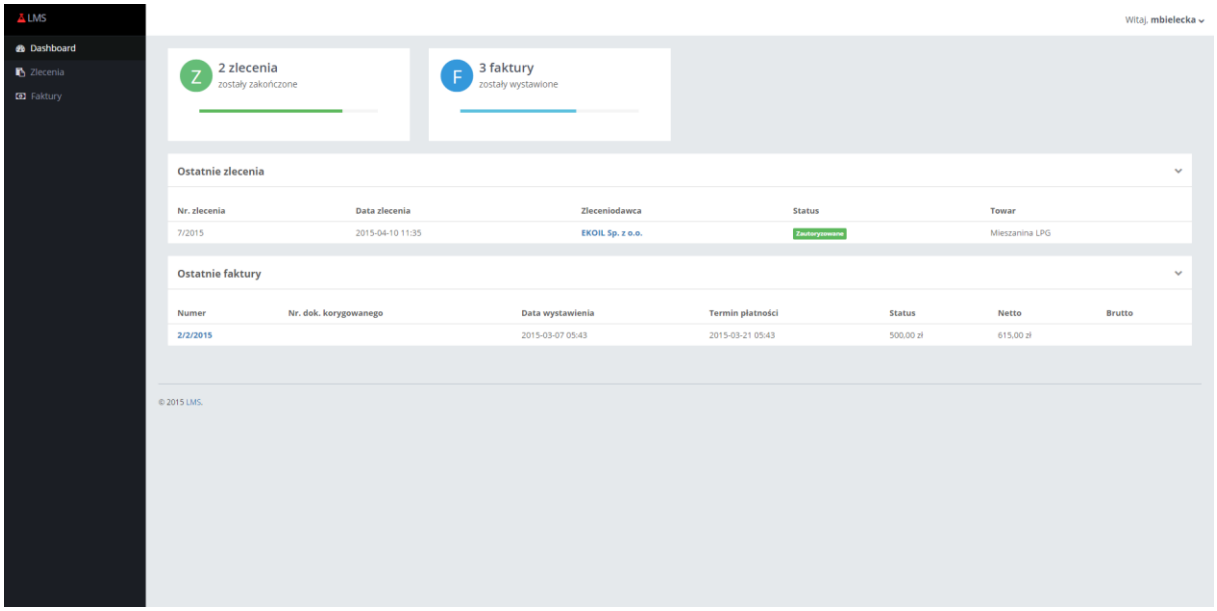
Prototyp powstał na bazie ASP.NET i jego wariantów MVC oraz Web API. Wykorzystanie HTML5, CSS3 oraz AngularJs pozwoliło na zaoszczędzenie dużej ilości czasu potrzebnego do ukończenia prototypu.

Oprogramowanie pozwala na tworzenie zleceń, zarządzanie kontrahentami, fakturami, notami korygującymi, ofertami, rejestrami przyjętych prób, metodami badań oraz użytkownikami. Zawiera również rozbudowany moduł raportów oraz infrastruktury, która zapewnia komunikację z pracownikami oraz klientami firmy.

Głównym obszarem roboczym prototypu jest *dashboard*. W zależności od nadanych uprawnień użytkownik ma dostęp do różnych widoków. Widok najbardziej rozbudowany prezentuje rysunek 31, natomiast rysunek 32 pokazuje jak wygląda ta sama strona dla klienta firmy.

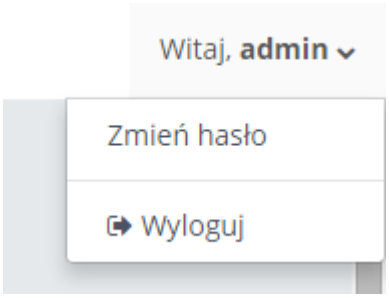


Rysunek 31. Główny ekran aplikacji

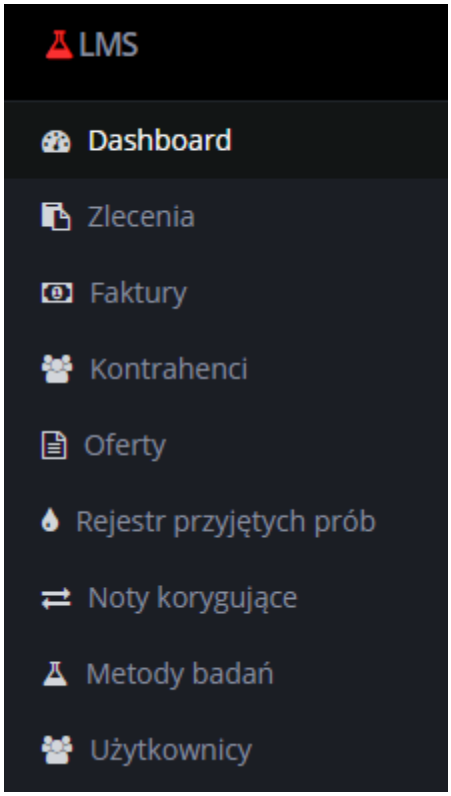


Rysunek 32. Główny ekran aplikacji (wersja dla klienta firmy)

Cały widok możemy podzielić na trzy obszary, główny, prezentujący zawartość poszczególnych funkcji. Belkę górną (rysunek 33), na której wyświetlane są dane użytkownika wraz z menu akcji oraz nawigację (rysunek 34) po lewej stronie ekranu, która jest generowana automatycznie.

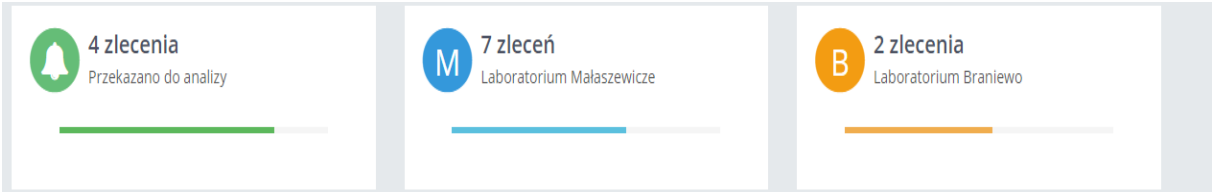


Rysunek 33. Belka górna

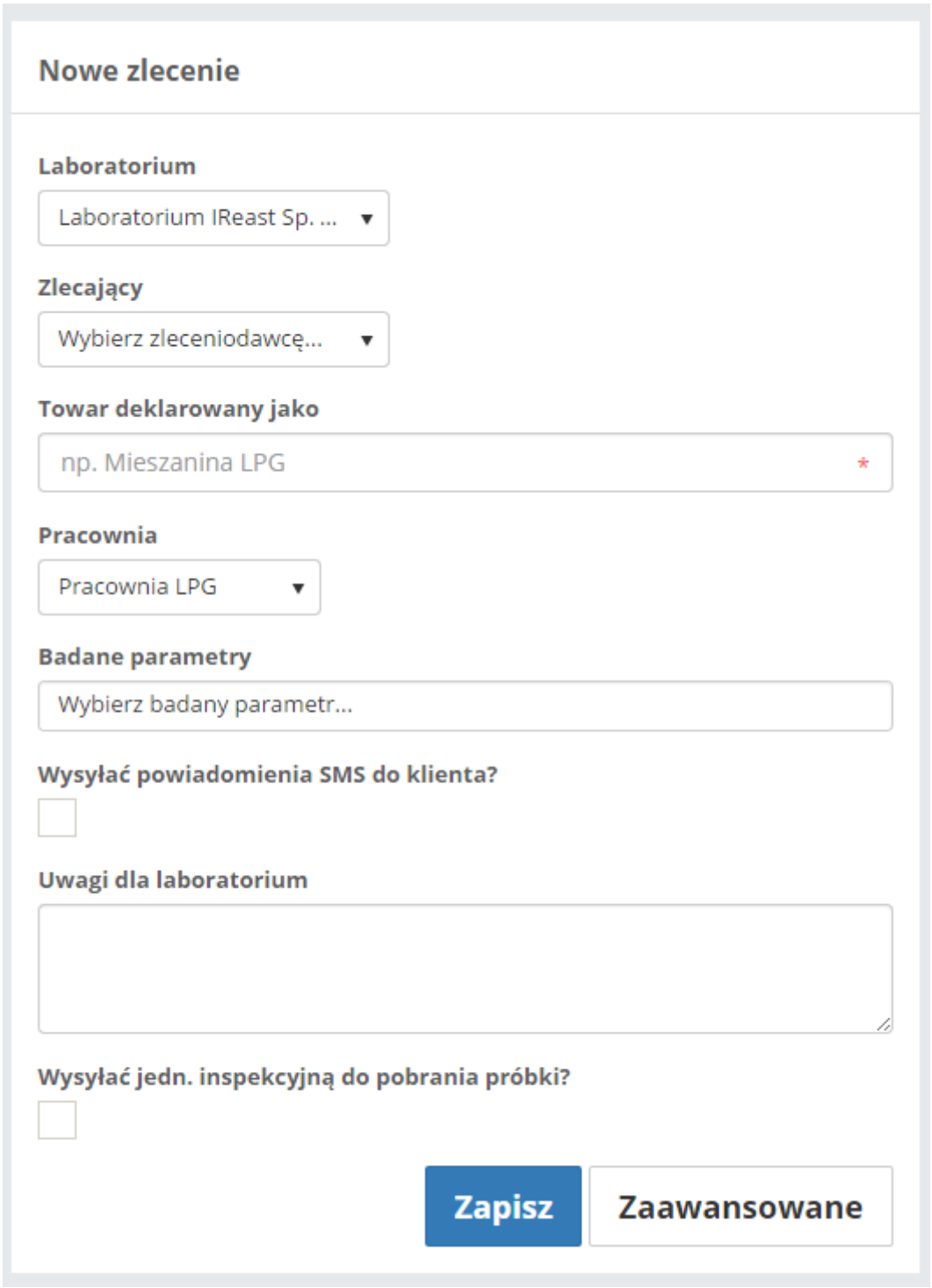


Rysunek 34. Menu nawigacji

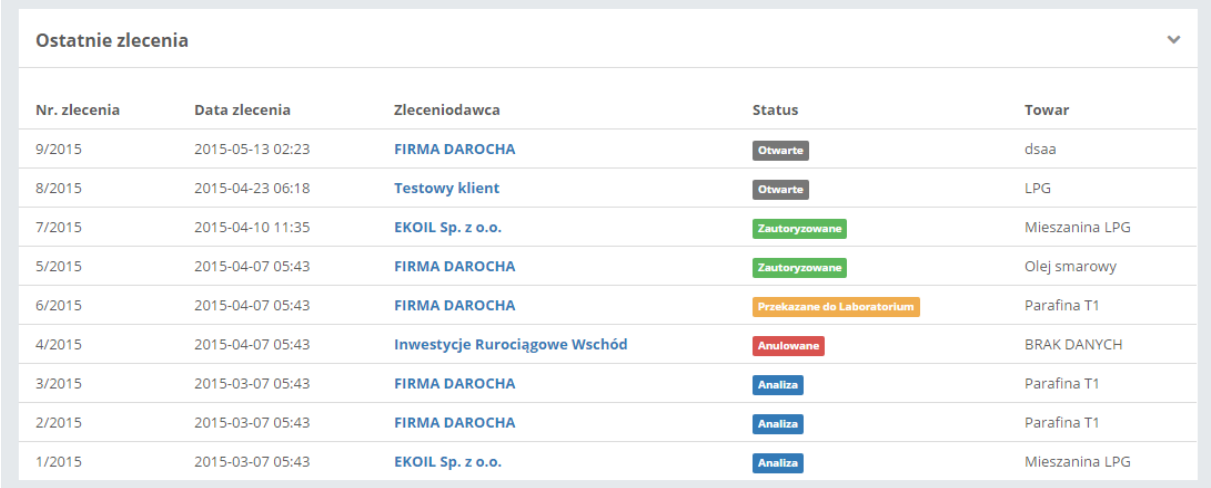
Widok *dashboardu* możemy podzielić na kilka najważniejszych elementów, część górną (rysunek 35) prezentującą aktualne statystyki z podziałem na laboratoria. Część środkową zawierającą po lewej stronie widżet (rysunek 36), odpowiadający za tworzenie nowego zlecenia oraz po prawej stronie dziesięć ostatnich zleceń (rysunek 37). Posiadając uprawnienia administracyjne, widzimy wszystkie zlecenia, dla niższych uprawnień lista zleceń jest ograniczana. Odpowiednio dla laboratorium, wyświetlają się zlecenia aktualnie analizowane, autoryzowane lub oczekujące. Natomiast dla jednostki inspekcyjnej pokazujemy zlecenia, które wymagają przeprowadzenia próbobrania. Widok dashboardu zamyka w prawym dolnym rogu wykres prezentujące przeprowadzoną liczbę analiz w aktualnym kwartale, z podziałem na poszczególne pracownie. Widok dla klienta prezentuje tylko jego faktury oraz zlecenia, bez możliwości wykonywania innych akcji niż podgląd oraz wydruk dokumentów.



Rysunek 35. Część górna dashboardu prezentujące ilość zleceń przekazanych do analizy oraz ich całkowitą liczbę z podziałem na laboratoria



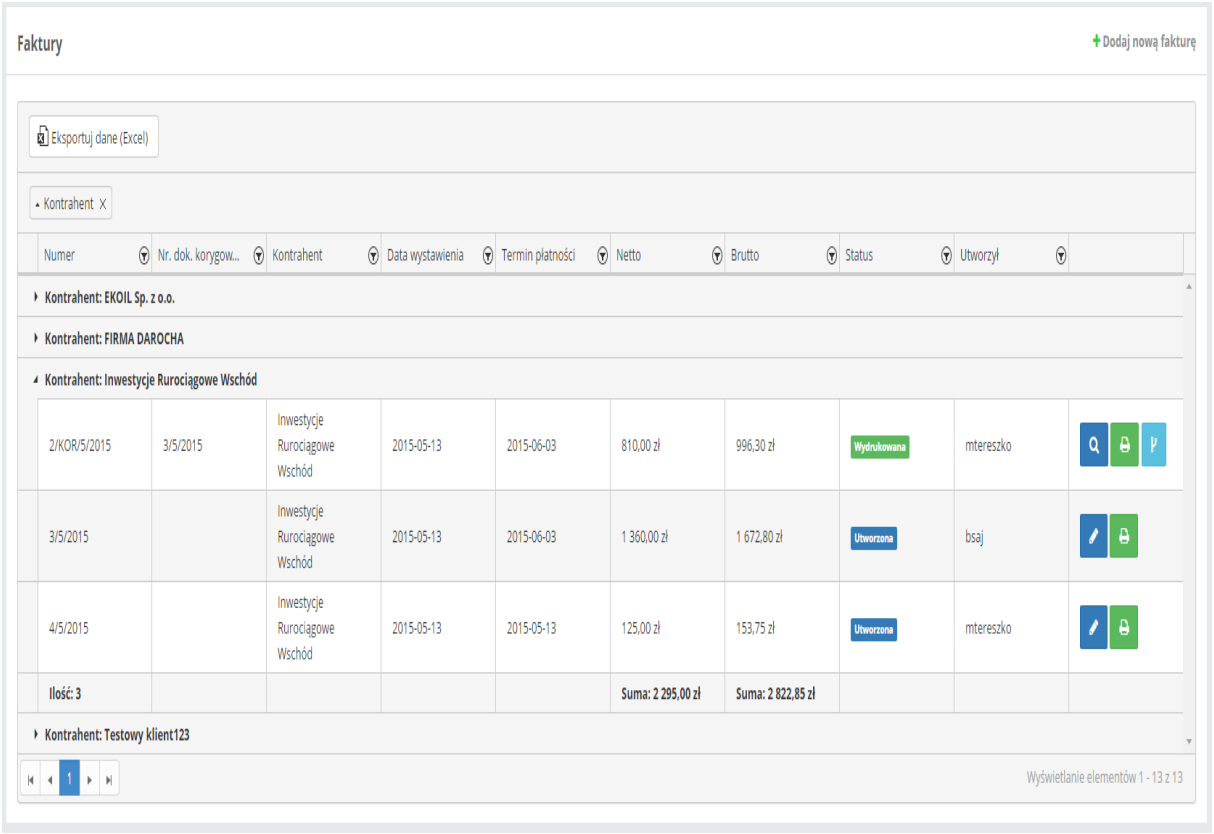
Rysunek 36. Widżet do tworzenia nowego zlecenia



Rysunek 37. Lista zleceń (widok admina - *dashboard*)

Dokumenty w prototypie są generowane na dwa różne sposoby i są w pełni elastyczne na zmiany, poprzez dodanie nowych modułów. Metody generujące raporty zawsze zwracają tablicę bajtów, a więc możemy generować nie tylko pliki w formacie pdf, ale również xlsx czy docx. Dla lepszego zobrazowania elastyczności rozwiązania, prototyp wykorzystuje do generowania dokumentów *Microsoft SQL Server Reporting Services* oraz kontrolkę *XtraReports* z pakietu *DevExpress*. System oferuje wydruk faktur, korekt, not, raportów z przeprowadzonej kontroli oraz sprawozdań z badań. Sprawozdania z badań spełniają wymagania Polskiego Centrum Akredytacyjnego i są opatrzone odpowiednim logiem.

Głównymi funkcjonalnościami oferowanymi przez system są: przeprowadzenie analizy próbki chemicznej oraz fakturowanie. Analiza próbki zaczyna się od przyjęcia zlecenia przez biuro obsługi klienta, które to generuje odpowiednią notyfikację do dyspozytora jednostki inspekcyjnej. Dyspozytor po przyjęcia zlecenia, wysyła inspektora na miejsce wykonania analizy. Po przeprowadzeniu próbobrania, próbka jest zabierana do odpowiedniego oddziału, opisywana i wysyłana do analizy. Po przeprowadzonych badaniach wyniki analizy muszą być zatwierdzone przez kierownika laboratorium. Zatwierdzenie wyników badań automatycznie generuje wiadomość email z opisem badań oraz załączonym raportem z kontroli i wiadomość sms, z krótkim komunikatem informującym o dostępności wyników w biurze obsługi klienta oraz panelu klienta. Cały proces przyjmowania zlecenia i jego obsługi został opisany przy aktywnym udziale ekspertów z branży chemicznej i zatwierdzony przez Polskie Centrum Akredytacji. Część księgowa była tworzona pod nadzorem zaprzyjaźnionego biura rachunkowego i oferuje wszystkie funkcjonalności niezbędne do prowadzenia małej firmy. Przykładową listę faktur prezentuje rysunek 38.



Rysunek 38. Lista faktur z przyciskiem do eksportu danych

Pobocznymi funkcjonalnościami są: ofertowanie, zarządzanie rejestrem prób oraz część administracyjna systemu. Ofertowanie pozwala na podpinanie dowolnych dokumentów z indywidualną ofertą dla poszczególnych klientów. Oferty mogą być aktywowane na żądanie oraz mogą mieć datę ważności. Zarządzanie rejestrem prób jest widokiem tabelarycznym, który udostępnia możliwość określenia momentu utylizacji próbki. Istotnym aspektem tej funkcjonalności jest to, że nie wszystkie pracownie mogą utylizować określone próbki.

Część administracyjna pozwala na zarządzanie użytkownikami, przypisywanie im uprawnień, edycję danych oraz dodawanie specjalnych kont dla klientów. Dodatkowo mamy możliwość zarządzania metodami badań, które mogą wykonywać laboratoria, poprzez określenie czy metody są akredytowane, tworzyć grupy i przypisywać je do konkretnych pracowni.

6.2. System modułów

Prototyp systemu udostępnia niezbędną infrastrukturę do tworzenia nowych modułów, które możemy łączyć w grupy oraz tworzyć zależności pomiędzy nimi. Dodanie nowego modułu jest proste dzięki wstrzykiwaniu zależności. Każdy moduł jest klasą, która dziedziczy z klasy *LmsModule*. Przykład modułu infrastruktury prezentuje listing 5.

Listing 5. Moduł infrastruktury

```
 [DependsOn(typeof(LMSCoreModule))]
public class LMSInfrastructureModule : LmsModule
{
        public override void Initialize()
        {
            IocManager.RegisterAssemblyByConvention(Assembly.GetExecutingAssembly());
            IocManager.IocContainer.Register(
Component.For<IEmailService>().ImplementedBy<EmailService>().LifestyleTransient());
        }

    }
```

Podczas startu aplikacji, system skanuje główny katalog bin w poszukiwaniu modułów do załadowania. Jak widać na powyższym listingu, dzięki wykorzystaniu atrybutu *DependsOn* definiujemy od jakich modułów będzie zależny nasz moduł. W tym przypadku jest to moduł rdzenia prototypu *LMSCoreModule.* W metodzie *Initialize* ustawiamy dodatkowe zależności, które chcielibyśmy dodać do kontenera DI. Tutaj również możemy dodać kod konfiguracyjny modułu. Warto pamiętać, że w zależności od zastosowanego kontenera DI, moduły są rejestrowane w różnej kolejności. Przykładowo, w przypadku zarejestrowania interfejsu w module A, a następnie w module B, zastosowany w prototypie *Castle Windsor* zawsze wybiera pierwszy wpis z kontenera, natomiast *AutoFac* ostatni.

6.3. API

API znajduje się w warstwie aplikacji, która pośredniczy w komunikacji pomiędzy warstwą prezentacji oraz domeny. Warstwa aplikacji ma za zadanie dyrygować obiektami biznesowymi, które mają wykonywać określone zadania. Najważniejszą jej częścią są serwisy, które udostępniają logikę domenową do warstwy prezentacji. Serwisy są wywoływane z wykorzystaniem obiektów DTO jako parametrów, następnie wykonują wcześniej zdefiniowane zagadnienia biznesowe i zwracają odpowiedź jako inny obiekt DTO. Warstwa prezentacyjna nigdy nie powinna mieć bezpośredniego dostępu do warstwy domeny.

Serwisy są definiowane poprzez interfejsy, które implementują interfejs *IApplicationService.* Listing 6 prezentuje interfejs, który udostępnia metody niezbędne do zarządzania klientami firmy.

Listing 6. Interfejs do zarządzania klientami

```
public interface ICustomerAppService : IApplicationService
    {
        GetCustomersOutput GetCustomers();
        GetCustomerOutput GetCustomer(GetCustomerInput input);
        void UpdateCustomer(UpdateCustomerInput input);
        void CreateCustomer(CreateCustomerInput input);

    }
```

Powyższy listing wymaga dodatkowego komentarza, dotyczącego obiektów typu *GetCustomersOutput* oraz *GetCustomerInput*. Klasy zakończone frazą *Output* definiują odpowiedź wysyłaną przez serwis. Jak można się domyśleć klasy zakończone frazą *Input* odpowiadają za parametry wejściowe. Listing 7 prezentuje implementację takich klas.

Listing 7. Klasa wyjściowa oraz wejściowa dla metody GetCustomer

```
public class GetCustomerOutput : IOutputDto
{
    public CustomerDto Customer { get; set; }
}
public class GetCustomerInput : IInputDto
{
    public int CustomerId { get; set; }
}
```

Parametry wejściowe możemy dodatkowo walidować zanim trafią do konkretnej metody serwisu. Listing 8 prezentuje niestandardowy walidator oraz wykorzystanie atrybutu Required opisującego jedną z właściwości.

Listing 8. Walidacja parametrów wejściowych

```
public class UpdateOrderInput : IInputDto, ICustomValidate
    {
        public bool SendOrderToAuthorization { get; set; }
        public OrderDto Order { get; set; }

        public void AddValidationErrors(List<ValidationResult> results)
        {
            if (Order.SendNotification &&
string.IsNullOrWhiteSpace(Order.SendNotificationNumber))
                results.Add(new ValidationResult("Do wysłania notyfikacji niezbędny
jest poprawny nr. telefonu"));
            if (Order.SendInspectionUnit)
            {
                if (Order.Sample.SampleCollectionTime.Year < 2015) results.Add(new
ValidationResult("Podaj poprawny czas kontroli"));
                if (Order.Sample.SampleCollectingMethodId < 1) results.Add(new
ValidationResult("Wybierz sposób pobrania próby"));
                if (string.IsNullOrWhiteSpace(Order.Sample.SampleCollectedFrom))
results.Add(new ValidationResult("Wpisz Miejsce kontroli"));
            }
        }
    }
```

Po zdefiniowaniu odpowiednich interfejsów, podczas uruchamiania aplikacji generowane są tożsame web serwisy korzystające z *ASP.NET Web API*. Dzięki temu mamy dostęp do serwisów poprzez dowolną implementację zgodną z architekturą *REST*. Listing 9 prezentuje automatycznie wygenerowany *JavaScript* kompatybilny z AngularJS, który w łatwy sposób pozwala nam na komunikację z serwisami.

Listing 9. Wygenerowa fabryka AngularJs z interfejsu zaprezentowanego w listingu 6

```
abpModule.factory('abp.services.lms.customer', [
        '$http', function ($http) {
            return new function () {
                this.getCustomers = function (httpParams) {
                    return $http(angular.extend({
                        abp: true,
                        url: abp.appPath + 'api/services/lms/customer/GetCustomers',
                        method: 'POST',
                        data: JSON.stringify({})
                    }, httpParams));
                };

                this.getCustomer = function (input, httpParams) {
```

```
                return $http(angular.extend({
                    abp: true,
                    url: abp.appPath + 'api/services/lms/customer/GetCustomer',
                    method: 'POST',
                    data: JSON.stringify(input)
                }, httpParams));
            };

            this.updateCustomer = function (input, httpParams) {
                return $http(angular.extend({
                    abp: true,
                    url: abp.appPath + 'api/services/lms/customer/UpdateCustomer',
                    method: 'POST',
                    data: JSON.stringify(input)
                }, httpParams));
            };

            this.createCustomer = function (input, httpParams) {
                return $http(angular.extend({
                    abp: true,
                    url: abp.appPath + 'api/services/lms/customer/CreateCustomer',
                    method: 'POST',
                    data: JSON.stringify(input)
                }, httpParams));
            };

        };
    }
]);
```

Wykorzystanie takich metod jest proste i sprowadza się do kodu zaprezentowanego w listingu 10.

Listing 10. Wywołanie metody getCustomer z listingu 6

```
customerService.getCustomer({
    customerId: val
}).success(function (data) {
    vm.customer = data.customer;
}).error(function () {
    abp.notify.error('Nie znaleziono kontrahenta...');
});
```

7. Podsumowanie

Usługi chmurowe zapewniają szereg ułatwień, które usprawniają pracę zespołów programistycznych, zarówno poprzez repozytoria kodu, tablice przydzielonych zadań, jak również poprzez automatyczną integrację. Łatwość tworzenia nowego środowiska pozwala na lepsze testowanie oprogramowania i zwiększa końcową jakość produktu. Warto również pamiętać o minimalizacji kosztów, podczas prowadzenia projektu oraz w okresie jego utrzymania. Wykorzystując bazy *NoSQL* możemy stworzyć systemy, które są skalowalne i elastyczne, a dzięki szerokiej gamie dostępnych maszyn wirtualnych zwiększające moc obliczeniową, co bezpośrednio przekłada się na obsługę większej liczby klientów.

Projektowanie systemów informatycznych w oparciu o chmurę obliczeniową wymusza zmianę podejścia architektonicznego, poprzez odejście od sprawdzonych rozwiązań na rzecz potencjalnych korzyści w przyszłości. Zmianie musi również ulec sposób komunikacji z klientem biznesowym, należy poświęcić czas na wypracowanie wspólnego języka komunikacji od początku projektu, w szczególności przy współpracy z doświadczonymi fachowcami z danej branży.

W ramach pracy, został stworzony prototyp systemu usprawniającego działanie laboratorium chemicznego. Podczas tworzenia oprogramowania, zbierania wymagań i samego testowania, usługi chmurowe znacznie usprawniły cały proces. Dzięki rosnącej integracji *Visual Studio* z *Microsoft Azure*, wdrażanie nowych wersji aplikacji webowych, aktualizacja schematów baz danych, czy testowanie jest niezwykle proste. Zbudowany prototyp pozwala na zrealizowanie pełnego procesu tworzenia zlecenia badania laboratoryjnego, począwszy od analizy i pobierania próbek, kończąc na fakturowaniu oraz raportowaniu. Dodatkowo, użycie nowych technologii webowych zapewniło wsparcie dla urządzeń mobilnych bez pracochłonnego dostosowania interfejsu użytkownika do poszczególnych wielkości wyświetlaczy.

7.1. Zalety i wady przyjętych rozwiązań

Stworzony prototyp systemu spełnia wymagania DDD oraz zapewnia odpowiednią elastyczność i skalowalność, a dzięki modułowej budowie jest generyczny i umożliwia dostosowanie jego działania do potrzeb dowolnego klienta. Dzięki systemowi modułów, możemy łatwo dodawać lub modyfikować dostępne funkcjonalności. Rozbudowany interfejs użytkownika pozwala na generowanie dowolnych raportów, a wykorzystanie SSRS udostępnia edytor WYSIWYG do edycji bardziej skomplikowanych szablonów raportów. Zastosowanie chmury obliczeniowej redukuje koszty związane z zakupem sprzętu, administracją systemem oraz zmniejsza potencjalne ryzyko przestojów w działaniu systemu.

Wadą zaprojektowanego systemu jest niewątpliwie brak wsparcia dla urządzeń zewnętrznych używanych w laboratoriach. Ponadto pomimo udostępnienia rozbudowanego edytora raportów, wymaga on dodatkowej nauki. Samo podejście DDD niesie za sobą szereg problemów, narzut na wprowadzenie nowych funkcjonalności jest dość duży, a ilość testów może okazać się przytłaczająca. Warto również wspomnieć o głównej wadzie prototypu, czyli szczątkowej kompatybilności z najbardziej popularną przeglądarką używaną w biznesie, czyli Internet Explorerem. Ze względu na bardzo słabe wsparcie HTML5 oraz CSS3 aplikacja nie działa poprawnie oraz ma problemy z odpowiednim wyświetlaniem treści.

7.2. Proponowany plan rozwoju

Zaprezentowany prototyp jest punktem wyjścia do stworzenia bardziej rozbudowanego systemu. Dzięki implementacji obsługi wielu języków, możemy stworzyć oprogramowanie, które ma szansę zaistnieć nie tylko na polskim , ale również na zagranicznym. rynku System modułów pozwala na łatwe przygotowywanie raportów, chociaż stworzenie odrębnej aplikacji byłoby dużym

ułatwieniem dla biznesu. Dodanie obsługi SignalR mogłoby usprawnić pracę użytkowników poprzez aktualizację danych w czasie rzeczywistym, a rozbudowanie systemu notyfikacji ułatwiłoby komunikację pomiędzy poszczególnymi działami oraz samym klientem. Warto również wziąć pod uwagę rozwój w kierunku obsługi laboratoriów medycznych i wykorzystanie internetu rzeczy. Obecnie, na rynku dostępne są i cieszą się co raz większą popularnością zegarki, które monitorują nasze podstawowe parametry życiowe i przesyłają je bezpośrednio do analizy. To samo dotyczy sensorów, które monitorują poziom zanieczyszczenia powietrza ,czy hałasu w naszym otoczeniu, jak również jakość pobieranych próbek.

8. Bibliografia

[1]. Dokumentacja FILAB 2.0 [Online] http://www.filab.pl/ [data dostępu: 14.05.2015].

[2]. Dokumentacja KS-SOLAB [Online] http://www.kssa.pl/ [data dostępu: 14.05.2015].

[3]. Guthrie Scott, Simms Mark, Dykstra Tom, Anderson Rick, Wasson Mike. Building Cloud Apps with Microsoft Azure: Best Practices for DevOps, Data Storage, High Availability, and More. : Microsoft Press, 2014.

[4]. Wilder Bill. Cloud Architecture Patterns: Using Microsoft Azure. O'Reilly Media, 2012.

[5]. Galloway Jon, Wilson Brad, Allen Scott K., Matson David. Professional ASP.NET MVC 5. Wrox, 2014.

[6]. Seemann Mark. Dependency Injection in .NET. Manning Publications, 2011.

[7]. Kurtz Jamie. ASP.NET Web API 2: Building a REST Service from Start to Finish. Apress, 2014.

[8]. Lakshmiraghavan Badrinarayanan. Pro ASP.NET Web API Security: Securing ASP.NET Web API. Apress, 2013.

[9]. Cameron Dane. A Software Engineer Learns HTML5, JavaScript and jQuery. CreateSpace Independent Publishing Platform, 2013.

[10]. Duckett Jon. HTML and CSS: Design and Build Websites. Wiley, 2011.

[11]. Gasston Peter. The Book of CSS3: A Developer's Guide to the Future of Web Design. No Starch Press, 2014.

[12]. Hall Gary McLean. Adaptive Code via C#: Agile coding with design patterns and SOLID principles. Microsoft Press, 2014.

[13]. Erl Thomas. Cloud Computing: Concepts, Technology & Architecture. Prentice Hall, 2013.

[14]. Johnson Bruce. Professional Visual Studio 2013. Wrox, 2014.

[15]. Ben-Gan Itzik. Microsoft SQL Server 2012 T-SQL Fundamentals. Microsoft Press, 2012.

[16]. Sadalage Pramod J. . NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley Professional, 2012.

[17]. Carlson Josiah L. . Redis in Action. Manning Publications, 2013.

[18]. Evans Eric. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional, 2003.

[19]. Brader Larry, Leibovitz Roberta, Teruel Jose Luis Soria. Building a Release Pipeline with Team Foundation Server 2012. Microsoft patterns & practices, 2013.

[20]. Domain-Driven design Martin Fowler blog [Online] http://www.kssa.pl/ [data dostępu: 14.05.2015].

[21]. User in experience Brandon Satron blog [Online] http://userinexperience.com/?p=308 [data dostępu: 14.05.2015].

[22]. Right scale Cloud Computing Trends: 2015 State of the Cloud Survey [Online] http://www.rightscale.com/blog/cloud-industry-insights/cloud-computing-trends-2015-state-cloud-survey [data dostępu: 14.05.2015].

[23]. Jim O'Neil Windows Azure Interactive Feature Map [Online] http://blogs.msdn.com/b/jimoneil/archive/2012/09/21/windows-azure-feature-map.aspx [data dostępu: 14.05.2015].

[24]. Jackie Chen's IT Workshop Kick off my AWS journey [Online] http://jackiechen.org/2013/04/29/kick-off-my-aws-journey/ [data dostępu: 14.05.2015].

[25]. The Pragmatic Bookshelf [Online] https://pragprog.com/magazines/2009-12/going-naked [data dostępu: 14.05.2015].

[26]. Epic dominant domains [Online] http://epic.tesio.it/doc/manual/the_bellis_perennis.html [data dostępu: 14.05.2015].

[27]. Microsoft Azure [Online] https://azure.microsoft.com/pl-pl/documentation/articles/data-management-azure-sql-database-and-sql-server-iaas/ [data dostępu: 14.05.2015].

[28]. Visual Studio Online [Online] https://www.visualstudio.com/en-us/version-control-vs [data dostępu: 14.05.2015].

[29]. Visual Studio Online [Online] https://www.visualstudio.com/en-us/products/what-is-visual-studio-online-vs.aspx [data dostępu: 14.05.2015].

[30]. .Net Framework Wikipedia [Online] http://en.wikipedia.org/wiki/.NET_Framework [data dostępu: 14.05.2015].

[31]. Common Language Infrastructure Wikipedia [Online] http://en.wikipedia.org/wiki/Common_Language_Infrastructure [data dostępu: 14.05.2015].

[32]. Common Language Runtime Wikipedia [Online] http://en.wikipedia.org/wiki/Common_Language_Runtime [data dostępu: 14.05.2015].

[33]. MSDN Microsoft [Online] https://msdn.microsoft.com/en-us/library/zw4w595w%28v=vs.110%29.aspx [data dostępu: 14.05.2015].

[34]. MSDN Microsoft [Online] https://msdn.microsoft.com/en-us/magazine/dd942833.aspx [data dostępu: 14.05.2015].

[35]. Entwicklertagebuch diary of a web developer [Online] http://entwicklertagebuch.com/blog/2013/10/how-to-structure-large-angularjs-applications/ [data dostępu: 14.05.2015].

[36]. XMCL [Online] http://www.xmcl.org/html-features.html [data dostępu: 14.05.2015].

[37]. Real business [Online] http://realbusiness.co.uk/article/25710-is-it-the-right-time-to-move-your-business-to-the-cloud [data dostępu: 14.05.2015].

[38]. Jeffrey Palermo blog [Online] http://jeffreypalermo.com/blog/the-onion-architecture-part-1/ [data dostępu: 14.05.2015].

9. Listingi

10. Rysunki