1 Ucieczka z monolitycznego piekła

Niniejszy rozdział dotyczy

- objawów monolitycznego piekła i sposobów na ucieczkę od niego dzięki przyjęciu architektury mikroserwisowej;
- zasadniczych cech architektury mikroserwisowej oraz jej zalet i wad;
- tego, jak mikroserwisy umożliwiają zastosowanie stylu DevOps do tworzenia dużych, złożonych aplikacji;
- języka wzorców architektury mikroserwisowej i powodów, dla których należy go używać.

Był dopiero poniedziałek w porze lunchu, ale Mary, dyrektor ds. technicznych w Food to Go, Inc. (FTGO), była już sfrustrowana. Jej dzień zaczął się naprawdę dobrze. Poprzedni tydzień spędziła z innymi architektami i programistami na doskonałej konferencji, poznając najnowsze techniki tworzenia oprogramowania, w tym ciągłe wdrażanie i architekturę mikroserwisową. Mary spotkała się również ze swoimi byłymi kolegami z informatyki z North Carolina A&T State University i podzieliła się opowieściami związanymi

z technologicznym przywództwem. Konferencja sprawiła, że poczuła się silna i chętna do ulepszania sposobu, w jaki FTGO rozwija oprogramowanie.

Na nieszczęście to uczucie szybko ustąpiło. Właśnie spędziła pierwszy poranek w biurze na kolejnym bolesnym spotkaniu ze starszymi inżynierami i biznesmenami. Dwie godziny poświęcili na dyskusję o tym, dlaczego zespół programistów znowu spóźni się z kolejnym krytycznym wydaniem. Niestety tego rodzaju spotkania stały się coraz powszechniejsze w ciągu ostatnich kilku lat. Mimo zastosowania oprogramowania zwinnego tempo rozwoju spadało, a więc osiągnięcie celów firmy było prawie niemożliwe. I, co gorsza, nie było prostego rozwiązania.

Konferencja uświadomiła Mary, że FTGO cierpi z powodu *monolitycznego piekła* i że lekarstwem byłoby przyjęcie architektury mikroserwisowej. Ale architektura mikroserwisowa i związane z nią najnowocześniejsze praktyki opracowywania oprogramowania przedstawione na konferencji wydawały się nieuchwytnym marzeniem. Dla Mary nie było jasne, jak może walczyć z dzisiejszymi pożarami, poprawiając jednocześnie sposób opracowywania oprogramowania w FTGO.

Na szczęście, jak dowiemy się z tej książki, istnieje wyjście z tej sytuacji. Ale najpierw spójrzmy na problemy, przed którymi stoi FTGO, oraz na to, z czego one wynikają.

1.1. Powolny marsz w kierunku monolitycznego piekła

Od momentu startu pod koniec 2005 r. FTGO rozwijało się błyskawicznie. Dziś jest jedną z głównych internetowych firm dostarczających żywność w Stanach Zjednoczonych. Firma planuje nawet ekspansję zagraniczną, choć plany te są zagrożone z powodu opóźnień we wdrażaniu niezbędnych funkcjonalności.

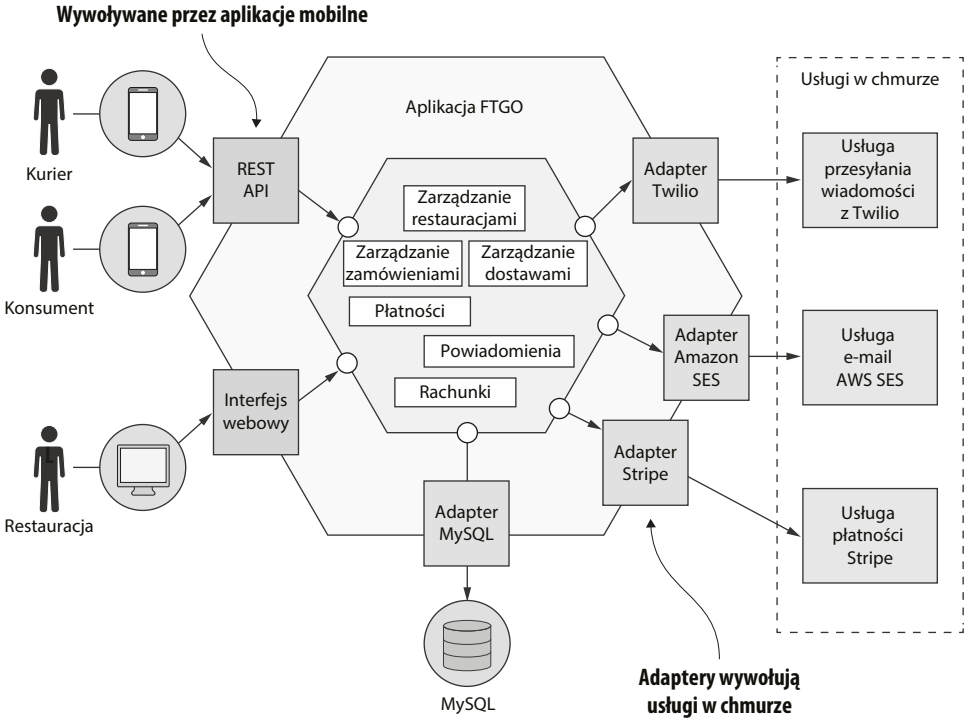
W swojej istocie aplikacja FTGO jest dość prosta. Konsumenci korzystają ze strony internetowej lub aplikacji mobilnej do składania zamówień na jedzenie w lokalnych restauracjach. FTGO koordynuje sieć kurierów, którzy dostarczają zamówienia. Odpowiada również za płacenie kurierom i restauracjom. Restauracje używają strony internetowej FTGO do edycji menu i zarządzania zamówieniami. Aplikacja korzysta z różnych usług sieciowych, w tym ze Stripe'a – do płatności, Twilio – do przesyłania wiadomości oraz Amazon Simple Email Service (SES) – do poczty e-mail.

Podobnie jak wiele innych starzejących się aplikacji korporacyjnych, aplikacja FTGO jest monolitem składającym się z pojedynczego pliku Java Web Application Archive (WAR). Z biegiem lat stała się dużą, złożoną aplikacją. Mimo wysiłków zespołu programistów stała się ona przykładem wzorca wielkiej kuli błota (www.laputan.org/mud/). Cytując Foote'a i Yodera, autorów tego wzorca, jest to „przypadkowo zbudowana, rozwlekła, niechlujna, połączona taśmą klejącą i drutem, dżungla kodu spaghetti". Tempo dostarczania oprogramowania spowolniło. Co gorsza, aplikacja FTGO została napisana z użyciem jeszcze bardziej przestarzałych frameworków. Aplikacja FTGO wykazuje wszystkie objawy monolitycznego piekła.

Kolejny punkt opisuje architekturę aplikacji FTGO. Następnie jest mowa o tym, dlaczego monolityczna architektura początkowo działała dobrze. Zobaczymy, jak aplikacja FTGO przerosła swoją architekturę i jak doprowadziło to do monolitycznego piekła.

1.1.1. Architektura aplikacji FTGO

FTGO jest typową korporacyjną aplikacją Java. Rysunek 1.1 pokazuje jej architekturę. Aplikacja FTGO ma architekturę heksagonalną, która jest stylem architektonicznym opisanym bardziej szczegółowo w rozdziale 2. W architekturze heksagonalnej rdzeń aplikacji składa się z logiki biznesowej. Logikę biznesową otaczają różne adaptery, które implementują interfejsy użytkownika i integrują się z systemami zewnętrznymi.



Rysunek 1.1. Aplikacja FTGO ma architekturę heksagonalną. Składa się z logiki biznesowej otoczonej adapterami, które implementują interfejsy użytkownika i współpracują z systemami zewnętrznymi, takimi jak aplikacje mobilne i usługi w chmurze, w celu realizacji płatności, przesyłania wiadomości i e-maili

Logika biznesowa składa się z modułów, z których każdy jest zbiorem obiektów domenowych. Przykładami modułów są: `Order Management`, `Delivery Management`, `Billing` i `Payments`. Istnieje kilka adapterów, które współpracują z systemami zewnętrznymi. Niektóre z nich to adaptery *wejściowe*, które obsługują żądania, wywołując logikę biznesową. Przykładami są adaptery `REST API` i `Web UI`. Istnieją także adaptery *wyjściowe*, które umożliwiają logice biznesowej dostęp do bazy danych MySQL i wywoływanie usług w chmurze, takich jak Twilio i Stripe.

Mimo logicznej modułowej architektury aplikacja FTGO została spakowana jako pojedynczy plik WAR. Aplikacja jest przykładem powszechnie stosowanego *monolitycznego* stylu architektury oprogramowania, który tworzy system jako pojedynczy wykonywalny lub możliwy do wdrożenia komponent. Gdyby aplikacja FTGO została napisana

w języku Go (GoLang), byłby to pojedynczy plik wykonywalny. Wersja aplikacji w Ruby lub NodeJS byłaby pojedynczym katalogiem z hierarchią kodu źródłowego. Monolityczna architektura nie jest z natury zła. Programiści FTGO podjęli dobrą decyzję, wybierając do swoich zastosowań architekturę monolityczną.

1.1.2. Zalety architektury monolitycznej

W początkowym okresie, gdy aplikacja FTGO była niewielka, monolityczna architektura aplikacji miała wiele zalet:

- *Prostota w rozwijaniu* – IDE i inne narzędzia programistyczne koncentrują się na budowaniu pojedynczej aplikacji.
- *Łatwość we wprowadzaniu radykalnych zmian w aplikacji* – można zmienić kod i schemat bazy danych, a następnie to zbudować i wdrożyć.
- *Prostota w testowaniu* – programiści napisali kompleksowe testy, które uruchomiły aplikację, wywołały interfejs REST API i przetestowały interfejs użytkownika za pomocą Selenium.
- *Prostota w instalacji* – programista musiał tylko skopiować plik WAR na serwer, na którym zainstalowano Tomcat.
- *Łatwość skalowania* – FTGO umożliwiała uruchomianie wielu instancji aplikacji z użyciem równoważenia obciążenia.

Z czasem jednak projektowanie, testowanie, wdrażanie i skalowanie stały się znacznie trudniejsze. Zobaczmy dlaczego.

1.1.3. Życie w monolitycznym piekle

Niestety, jak mogli się przekonać twórcy FTGO, architektura monolityczna ma ogromne ograniczenia. Popularne aplikacje, takie jak FTGO, mają zwyczaj przerastać architekturę monolityczną. W każdym sprincie zespół programistów FTGO wdrażał kilka kolejnych wymagań i z tego powodu baza kodu stawała się coraz większa. Ponadto, w miarę jak firma odnosiła coraz większe sukcesy, stale rosła liczebność zespołu programistów. Nie tylko zwiększyło to tempo wzrostu bazy kodu, ale także ogólne koszty zarządzania.

Jak pokazuje rysunek 1.2, niegdyś mała, prosta aplikacja FTGO z biegiem lat przekształciła się w monstrualny monolit. Podobnie, mały zespół programistów stał się teraz wieloma zespołami scrumowymi, z których każdy pracuje w określonym obszarze funkcjonalnym. W wyniku przerośnięcia swojej architektury FTGO znajduje się w monolitycznym piekle. Rozwój jest powolny i bolesny. Zwinne opracowywanie i wdrażanie jest niemożliwe. Zobaczmy, dlaczego tak się stało.

KOMPLEKSOWOŚĆ PRZERAŻA DEWELOPERÓW

Głównym problemem związanym z aplikacją FTGO jest to, że jest zbyt skomplikowana. Jest za duża, aby mógł ją w pełni zrozumieć jakikolwiek programista. W rezultacie naprawianie błędów i prawidłowe wdrażanie nowych funkcjonalności stało się trudne i czasochłonne. Przekraczane są terminy.