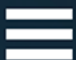


Michael Geers

Mikrofrontendy w akcji

Helion

Tytuł oryginału: Micro Frontends in Action

Tłumaczenie: Karolina Stangel

ISBN: 978-83-283-7781-3

Original edition copyright © 2020 by Manning Publications Co.
All rights reserved.

Polish edition copyright © 2021 by Helion S.A.
All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz wydawca dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/mikrak
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści

Przedmowa 11

Podziękowania 13

O książce 15

O autorze 19

CZĘŚĆ I POCZĄTEK PRZYGODY Z MIKROFRONTENDAMI 21

1. *Czym jest mikrofrontend?* 23

1.1. Szerszy kontekst 24
- *1.1.1. Systemy i zespoły 26*
- *1.1.2. Frontend 28*
- *1.1.3. Integracja frontendu 31*
- *1.1.4. Wspólne tematy 32*

1.2. Jakie problemy rozwiązuje architektura mikrofrontendowa? 34
- *1.2.1. Optymalizacja rozwoju funkcjonalności 34*
- *1.2.2. Precz z frontendowym monolitem 34*
- *1.2.3. Gotowość na zmiany 36*
- *1.2.4. Korzyści z niezależności 38*

1.3. Wady architektury mikrofrontendowej 40
- *1.3.1. Redundancja 40*
- *1.3.2. Spójność 41*
- *1.3.3. Różnorodność technologii 41*
- *1.3.4. Więcej kodu frontendowego 42*

1.4. Kiedy stosować model mikrofrontendowy? 42
- *1.4.1. Dla średnich i dużych projektów 42*
- *1.4.2. Najlepiej działa w sieci 43*
- *1.4.3. Produktywność i koszty 44*
- *1.4.4. Kiedy unikać mikrofrontendów? 44*
- *1.4.5. Kto używa mikrofrontendów? 45*

1.5. Podsumowanie 46

Kup ksi k Pole ksi k

2. Mój pierwszy projekt mikrofrontendowy 49

2.1. Przedstawiamy TraktoryOnline 50

2.1.1. Zaczynamy 51
2.1.2. Uruchamianie aplikacji 52

2.2. Przejścia między stronami za pomocą łączy 54

2.2.1. Własność danych 54
2.2.2. Kontrakt między zespołami 55
2.2.3. Jak to zrobić? 56
2.2.4. Zmiany adresów URL 59
2.2.5. Korzyści 59
2.2.6. Wady 60
2.2.7. Kiedy stosować łącza? 60

2.3. Kompozycja za pomocą ramek iframe 61

2.3.1. Jak to zrobić? 61
2.3.2. Korzyści 63
2.3.3. Wady 63
2.3.4. Kiedy stosować ramki iframe? 64

2.4. Co dalej? 65

2.5. Podsumowanie 66

CZĘŚĆ II ROUTING, KOMPOZYCJA I KOMUNIKACJA 67

3. Kompozycja techniką AJAX i routing po stronie serwera 69

3.1. Kompozycja techniką AJAX 70

3.1.1. Jak to zrobić? 71
3.1.2. Przestrzenie nazw dla stylów i skryptów 73
3.1.3. Deklaratywne ładowanie z biblioteką h-include 76
3.1.4. Korzyści 77
3.1.5. Wady 78
3.1.6. Kiedy stosować technikę AJAX? 79
3.1.7. Podsumowanie 79

3.2. Routing przez współdzielony serwer WWW 80

3.2.1. Jak to zrobić? 82
3.2.2. Przestrzenie nazw dla zasobów 85
3.2.3. Metody konfiguracji przekierowań 86
3.2.4. Odpowiedzialność za infrastrukturę 87
3.2.5. Kiedy stosować? 88

3.3. Podsumowanie 89

4. Kompozycja po stronie serwera 91

4.1. Kompozycja — Nginx i SSI 92

4.1.1. Jak to zrobić? 93
4.1.2. Szybsze ładowanie 95

Kup ksi k Pole ksi k

4.2. Obsługa niepewnych fragmentów 97
4.2.1. Zawodny fragment 97
4.2.2. Integracja fragmentu „W okolicy” 98
4.2.3. Limity czasu i treści zastępcze 99
4.2.4. Treści zastępcze 101
4.3. Analiza wydajności łączenia fragmentów 102
4.3.1. Ładowanie równoległe 102
4.3.2. Zagnieżdżone fragmenty 103
4.3.3. Opóźnione ładowanie 103
4.3.4. Czas do odebrania pierwszego bajta a streaming 104
4.4. Szybki przegląd innych rozwiązań 106
4.4.1. Edge-Side Includes 106
4.4.2. Tailor firmy Zalando 107
4.4.3. Podium 109
4.4.4. Którego rozwiązania użyć? 115
4.5. Zalety i wady kompozycji po stronie serwera 115
4.5.1. Korzyści 115
4.5.2. Wady 116
4.5.3. Kiedy integracja po stronie serwera ma sens? 117
4.6. Podsumowanie 117

5. Kompozycja po stronie klienta 119
5.1. Mikrofrontend w komponencie webowym 120
5.1.1. Jak to zrobić? 122
5.1.2. Framework w komponencie webowym 126
5.2. Izolacja stylów dzięki mechanizmowi Shadow DOM 128
5.2.1. Tworzenie ukrytego elementu głównego 128
5.2.2. Ograniczenie widoczności stylów 129
5.2.3. Kiedy używać mechanizmu Shadow DOM? 131
5.3. Zalety i wady komponentów webowych jako metody kompozycji 132
5.3.1. Korzyści 132
5.3.2. Wady 132
5.3.3. Kiedy integracja po stronie klienta ma sens? 133
5.4. Podsumowanie 134

6. Wzorce komunikacji 135
6.1. Komunikacja interfejsów użytkownika 136
6.1.1. Od komponentu nadrzędnego do podrzędnego 137
6.1.2. Komunikacja od komponentu podrzędnego do nadrzędnego 140
6.1.3. Komunikacja między fragmentami 144

6.1.4. Mechanizm publikacji/subskrypcji interfejsu API Broadcast Channel 148
6.1.5. Kiedy komunikacja między interfejsami użytkownika się sprawdza? 149
6.2. Inne mechanizmy komunikacji 150
6.2.1. Globalny kontekst i autentykacja 151
6.2.2. Zarządzanie stanem 152
6.2.3. Komunikacja między frontendem a backendem 153
6.2.4. Replikacja danych 153
6.3. Podsumowanie 155

7. Routing po stronie klienta i powłoka aplikacji 157

7.1. Powłoka aplikacji z płaskim routingiem 160
7.1.1. Czym jest powłoka aplikacji? 160
7.1.2. Anatomia powłoki aplikacji 160
7.1.3. Routing po stronie klienta 162
7.1.4. Renderowanie stron 164
7.1.5. Kontrakty między powłoką aplikacji a zespołami 167
7.2. Powłoka aplikacji z dwupoziomowym routingiem 168
7.2.1. Router pierwszego poziomu 169
7.2.2. Routing na poziomie zespołu 170
7.2.3. Co się dzieje przy zmianie adresu? 171
7.2.4. Interfejsy API powłoki aplikacji 174
7.3. Rzut oka na metaframework single-spa 175
7.3.1. Jak działa single-spa? 176
7.4. Wady połączonych aplikacji jednostronicowych 181
7.4.1. Tematy do przemyślenia 181
7.4.2. Kiedy stosować połączone aplikacje jednostronicowe? 184
7.5. Podsumowanie 185

8. Kompozycja i uniwersalne renderowanie 187

8.1. Połączenie dwóch kompozycji 188
8.1.1. Mechanizm SSI i komponenty webowe 190
8.1.2. Kontrakt między zespołami 194
8.1.3. Inne rozwiązania 195
8.2. Kiedy stosować uniwersalną kompozycję? 195
8.2.1. Uniwersalne renderowanie z kompozycją wyłącznie po stronie serwera 196
8.2.2. Większa złożoność 196
8.2.3. Uniwersalnie renderowane połączone aplikacje jednostronicowe? 196
8.3. Podsumowanie 197

9. *Który rodzaj architektury pasuje do mojego projektu?* *199*

9.1. Przypomnienie terminologii 200

9.1.1. Routing i przejścia między stronami 201
9.1.2. Metody kompozycji 201
9.1.3. Wysokopoziomowe modele architektoniczne 203

9.2. Porównanie złożoności 206

9.2.1. Architektoniczna niejednorodność 207

9.3. Witryna czy aplikacja? 207

9.3.1. Kontinuum między dokumentem a aplikacją 208
9.3.2. Serwer, klient czy uniwersalne renderowanie? 209

9.4. Wybór odpowiedniej architektury i metody integracji 210

9.4.1. Silna izolacja (przestarzały kod / kod strony trzeciej) 212
9.4.2. Szybkie pierwsze załadowanie / stopniowe ulepszanie 212
9.4.3. Błyskawiczna reakcja 213
9.4.4. Miękka nawigacja 214
9.4.5. Wiele mikrofrontendów na jednej stronie 214

9.5. Podsumowanie 215

CZĘŚĆ III SZYBKOŚĆ, SPÓJNOŚĆ I EFEKTYWNOŚĆ 217

10. *Ładowanie zasobów* *219*

10.1. Strategie odnoszenia się do zasobów 220

10.1.1. Odniesienia bezpośrednie 220
10.1.2. Cache-busting a niezależne wdrożenia 221
10.1.3. Odniesienie przez przekierowanie (klient) 222
10.1.4. Odniesienia w dyrektywie include 225
10.1.5. Synchronizacja wersji kodu i zasobów 227
10.1.6. Zasoby we fragmencie 230
10.1.7. Zintegrowane rozwiązania (Tailor, Podium itp.) 230
10.1.8. Szybkie podsumowanie 232

10.2. Podział na pakiety 233

10.2.1. Protokół HTTP/2 233
10.2.2. Wszystko w jednym pakiecie 234
10.2.3. Pakiet na zespół 235
10.2.4. Pakiet dla każdej strony i fragmentu 235

10.3. Ładowanie na żądanie 235

10.3.1. Komponenty pośredniczące 236
10.3.2. Opóźnione ładowanie stylów 237

10.4. Podsumowanie 237

11. Wydajność to klucz 239

11.1. Projektowanie z myślą o wydajności 240

11.1.1. Różne zespoły, różne pomiary 240
11.1.2. Międzyzespołowy budżet wydajności 242
11.1.3. Odpowiedzialność za spowolnienia 243
11.1.4. Korzyści w obszarze wydajności 244

11.2. Biblioteki zewnętrzne 246

11.2.1. Koszt autonomii 246
11.2.2. Małe jest piękne 247
11.2.3. Jedna globalna wersja 249
11.2.4. Wersjonowane pakiety z bibliotekami zewnętrznymi 250
11.2.5. Unikaj współdzielenia kodu biznesowego 262

11.3. Podsumowanie 262

12. Interfejs użytkownika i system projektowania 265

12.1. Po co nam system projektowania? 266

12.1.1. Cel i rola 268
12.1.2. Korzyści 268

12.2. Centralny system projektowania a autonomia zespołów 269

12.2.1. Czy potrzebuję własnego systemu projektowania? 269
12.2.2. Proces, a nie projekt 270
12.2.3. Stały budżet i odpowiedzialność 270
12.2.4. Poparcie zespołów 271
12.2.5. Proces rozwoju — centralny czy federacyjny? 273
12.2.6. Etapy rozwoju 274

12.3. Integracja w czasie wykonania czy wersjonowanie? 276

12.3.1. Integracja w czasie wykonania 276
12.3.2. Wersjonowany pakiet 278

12.4. Artefakty generyczne i specyficzne 281

12.4.1. Wybór formatu komponentów 281
12.4.2. Nieuchronne zmiany 285

12.5. Co powinno wchodzić w skład centralnej biblioteki wzorców? 286

12.5.1. Koszt współdzielenia komponentów 286
12.5.2. Centralny czy lokalny? 286
12.5.3. Centralne i lokalne biblioteki wzorców 289

12.6. Podsumowanie 290

13. Zespoły i granice 293

13.1. Dopasowanie między systemami i zespołami 294

13.1.1. Granice między zespołami 295
13.1.2. Głębia integracji 297
13.1.3. Zmiana kulturowa 300

13.2. Dzielenie się wiedzą 301
13.2.1. Wspólnota praktyków 302
13.2.2. Nauka 303
13.2.3. Zaprezentuj swoją pracę 303
13.3. Globalne problemy 303
13.3.1. Centralna infrastruktura 304
13.3.2. Zespół ds. specjalistycznych komponentów 305
13.3.3. Globalne uzgodnienia i konwencje 305
13.4. Różnorodność technologiczna 306
13.4.1. Zestawy narzędzi i opcje domyślne 306
13.4.2. Szablon frontendowy 306
13.4.3. Odrzuć strach przed kopiowaniem 308
13.4.4. Wartość podobieństw 308
13.5. Podsumowanie 309

14. Migracje, lokalne środowisko rozwojowe i testowanie 311

14.1. Migracja 312
14.1.1. Model koncepcyjny na przetarcie szlaku 312
14.1.2. Strategia nr 1: kawałek po kawałku 314
14.1.3. Strategia nr 2: najpierw frontend 315
14.1.4. Strategia nr 3: od zera do wielkiego wybuchu 317
14.2. Rozwój w środowisku lokalnym 318
14.2.1. Bez kodu pozostałych zespołów 319
14.2.2. Tworzenie atrap fragmentów 319
14.2.3. Fragmenty w izolacji 321
14.2.4. Pobieranie mikrofrontendów innych zespołów ze środowiska testowego lub produkcji 323
14.3. Testowanie 323
14.4. Podsumowanie 325

Czym jest mikrofrontend?

W tym rozdziale:

- Dowiemy się, czym jest mikrofrontend.
- Porównamy podejście mikrofrontendowe z innymi architekturami.
- Zastanowimy się nad znaczeniem skalowalności w pracy nad frontendem.
- Odnotujemy wyzwania, jakie stawia omawiana architektura.

Przez ostatnie 15 lat pracowałem jako programista nad wieloma projektami. W tym czasie miałem sporo okazji, by dostrzec pewną prawidłowość, jeżeli chodzi o pracę w naszej branży: praca z kilkoma osobami nad nowym projektem to wspaniałe doświadczenie. Każdy programista zna z grubsza wszystkie elementy systemu. Nowe funkcjonalności powstają szybko. Omawianie zagadnień ze współpracownikami jest proste. Ten stan rzeczy zmienia się jednak wraz ze zwiększeniem zakresu projektu i wielkości zespołu. Nagle pojedynczy programista nie zna już każdej linijki kodu w systemie. W zespole pojawiają się osoby zwane silosami wiedzy. Rośnie złożoność — zmiana w jednej części systemu może mieć nieoczekiwany wpływ na inną. Rozmowy w ramach zespołu są mniej efektywne. Wcześniej członkowie zespołu podejmowali decyzje przy ekspresie do kawy. Teraz, aby wszyscy byli na bieżąco,

Kup ksi k Pole ksi k

potrzebne są formalne spotkania. Frederick Brooks opisał to zjawisko w książce *Legendarny osobomiesiąc* już w roku 1975. W pewnym momencie zwiększenie liczby programistów w zespole nie zwiększa produktywności.

Aby temu zapobiec, projekty często dzieli się na mniejsze. Modne stało się dzielenie oprogramowania, a w konsekwencji również struktury zespołu na różne technologie. Często stosuje się podział poziomy na jeden zespół frontendowy oraz jeden lub więcej zespołów backendowych. Podejście mikrofrontendowe to alternatywne rozwiązanie. Dzieli ono aplikację pionowo. Każda jej część — od bazy danych po interfejs użytkownika — jest budowana i utrzymywana przez jeden zespół. Tak przygotowane frontendy łączy się, tworząc kompletną stronę widoczną dla klienta. To podejście jest powiązane z architekturą mikrousług. Główną różnicą jest jednak to, że usługa mikrofrontendowa obejmuje również interfejs użytkownika. Takie poszerzenie definicji usługi pozwala uniknąć potrzeby utrzymywania centralnego zespołu frontendowego. Oto trzy główne powody, dla których firmy decydują się na architekturę mikrofrontendową:

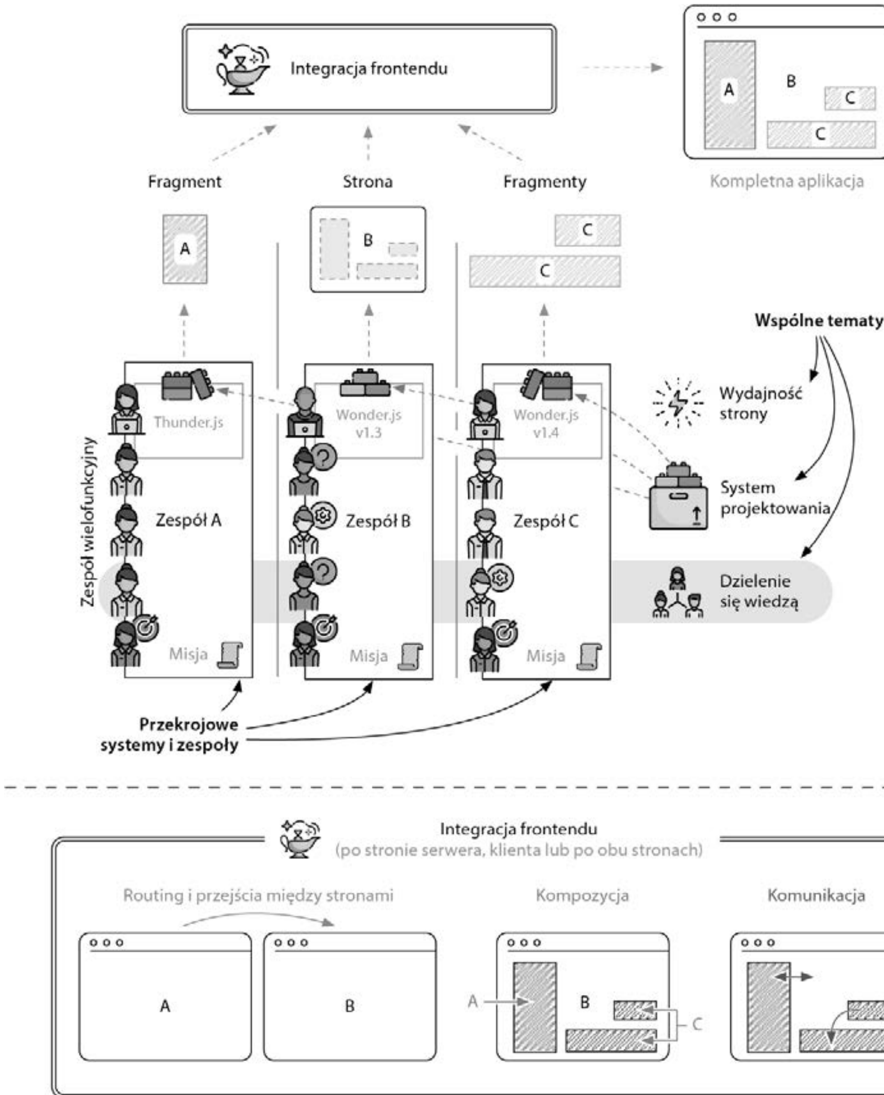
- **Optymalizacja rozwoju funkcjonalności**: Zespół dysponuje wszystkimi kompetencjami, które są potrzebne do dostarczenia funkcjonalności. Nie jest konieczna koordynacja zespołu frontendowego z backendowym.
- **Łatwiejsze aktualizacje na frontendzie**: Każdy zespół dobiera sobie własny zestaw narzędzi od frontendu po bazę danych. Dzięki temu może zdecydować się na aktualizację lub zmianę technologii frontendowej niezależnie od innych.
- **Zwiększenie orientacji na klienta**: Każdy zespół dostarcza swoje funkcjonalności klientowi — bez pośredników. Nie ma zespołów ds. operacji czy infrastruktury.

W tym rozdziale dowiemy się, jakie problemy może rozwiązać architektura mikrofrontendowa i kiedy jej zastosowanie jest uzasadnione.

1.1. Szerszy kontekst

Rysunek 1.1 przedstawia przegląd wszystkich elementów, które mają istotne znaczenie przy wdrażaniu architektury mikrofrontendowej. Mikrofrontendy to nie jakaś konkretna technologia. Jest to raczej odmienne podejście organizacyjne i architektoniczne. Dlatego na diagramie znajduje się tak wiele elementów, włączając w to strukturę zespołów, techniki integracji oraz inne powiązane zagadnienia. Teraz przeanalizujemy wszystkie elementy krok po kroku. Zaczniemy od trzech zespołów, które możemy zobaczyć powyżej przerywanej linii, a następnie omówimy elementy znajdujące się nad nimi. Gdy dotrzemy do ramki z czarodziejską lampą na samej górze, omówimy zagadnienie integracji frontendu. Na dole diagramu widać w powiększeniu trzy elementy, które składają się na to zagadnienie. Są to aspekty, które będziemy

Kup ksi k Pole ksi k



Rysunek 1.1. *Spojrzenie z lotu ptaka na architekturę mikrofrontendową. Podzielone pionowo zespoły (na dole diagramu) stanowią podstawowe założenie tej architektury. Każdy zespół tworzy funkcjonalności w formie stron lub fragmentów. Aby je połączyć i dostarczyć gotową stronę klientowi, można użyć takich technologii jak mechanizm SSI czy komponenty webowe*

musieli uwzględnić, aby zapewnić odpowiednią integrację naszej aplikacji. Podróż zakończymy, zapoznając się z trzema wspólnymi tematami, które znajdują się po prawej stronie diagramu.

Kup ksi k Pole ksi k

1.1.1. Systemy i zespoły

Trzy ramki z zespołami A, B i C reprezentują pionowo podzielone systemy oprogramowania, które stanowią fundament tej architektury. Każdy system jest autonomiczny, co oznacza, że może działać nawet wtedy, gdy powiązane systemy mają awarię. Aby zapewnić takie funkcjonowanie, każdy system dysponuje własną bazą danych. Dzięki temu nie polega na synchronicznych zapytaniach do innych systemów, kiedy odpowiada na żądania.

Za każdy system odpowiedzialny jest jeden zespół, który zajmuje się wszystkimi jego elementami od początku do końca. W tej książce nie będziemy omawiać zagadnień backendowych, takich jak replikacja danych między systemami. Mają tutaj zastosowanie sprawdzone rozwiązania z obszaru mikrousług. Skoncentrujemy się na problemach organizacyjnych i integracji frontendu.

MISJA ZESPOŁU

Każdy zespół dysponuje specjalistycznymi kompetencjami, dzięki którym może wygenerować wartość dla klienta. Na rysunku 1.2 możemy zobaczyć projekt sklepu internetowego podzielonego na trzy zespoły.

Rysunek 1.2. *Projekt sklepu internetowego realizowanego przez trzy zespoły. Każdy zespół pracuje nad inną częścią sklepu i ma swoją misję, która określa jego zakres odpowiedzialności*

Każdy zespół powinien mieć opisową nazwę i jasno sprecyzowaną misję skoncentrowaną na kliencie. W projektach, nad którymi pracowałem, dzieliliśmy zespoły tak, aby obejmowały pewne fragmenty ścieżki klienta — etapy jego podróży zmierzającej do realizacji zakupu.

Misją **Zespołu Inspiracji** — jak widać po nazwie — jest inspirowanie klientów przeglądających strony i prezentowanie atrakcyjnych dla nich produktów.

Zespół Decyzji pomaga klientom w dokonywaniu świadomych decyzji zakupowych poprzez prezentowanie doskonałych zdjęć produktów, ich specyfikacji oraz ocen innych klientów, a także udostępnianie narzędzi do porównywania.

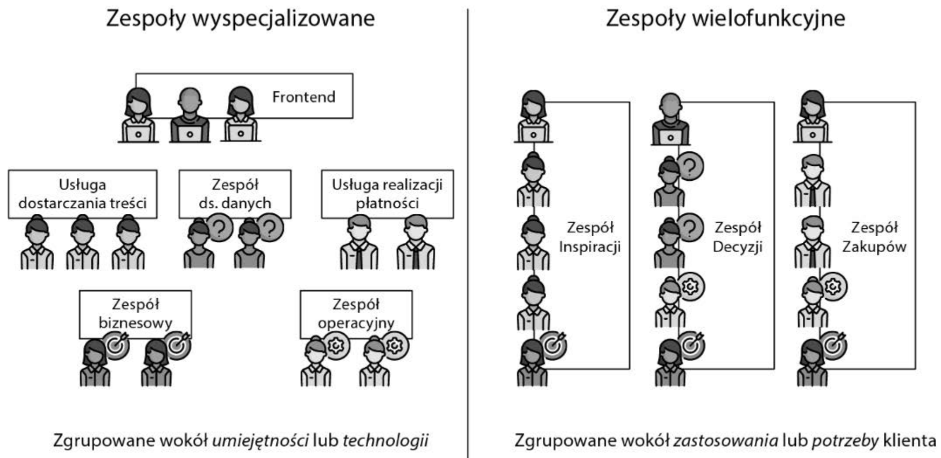
Kup ksi k Pole ksi k

Zespół Zakupów przejmuje klienta w momencie podjęcia przez niego decyzji o zakupie produktu i przeprowadza kupujących przez proces realizacji zamówienia.

Jasno sformułowana misja ma kluczowe znaczenie dla zespołu. Pozwala skupić się na celu i odpowiednio podzielić system.

WIELOFUNKCYJNE ZESPOŁY

Najbardziej istotną różnicą między architekturą mikrofrontendową a innymi podejściami jest struktura zespołów. Po lewej stronie rysunku 1.3 możemy zobaczyć model bazujący na zespołach złożonych ze specjalistów z tej dziedziny. Ludzie są pogrupowani według umiejętności lub technologii. Frontendowcy pracują w zespole frontendowym, a specjaliści ds. obsługi płatności w zespole ds. płatności. Eksperci biznesowi i specjaliści ds. infrastruktury również tworzą odrębne zespoły. Jest to struktura typowa dla mikrousług.



Rysunek 1.3. *Struktura zespołów typowa dla mikrousług (po lewej) w porównaniu do struktury charakterystycznej dla architektury mikrofrontendowej (po prawej). W drugim z tych modeli zespoły skupiają się wokół potrzeb klienta, a nie technologii takich jak frontend czy backend*

Podział na zespoły specjalistów na pierwszy rzut oka wydaje się naturalny, prawda? Frontendowcy lubią pracować z innymi frontendowcami. Mogą omawiać problemy, które próbują naprawić, lub pomysły na usprawnienie jakiejś części kodu. To samo odnosi się do innych zespołów, które specjalizują się w konkretnych obszarach. Tacy specjaliści dążą do perfekcji i starają się tworzyć możliwie najlepsze rozwiązania w swojej dziedzinie. Gdy zespoły doskonale wykonują swoją pracę, końcowy produkt również musi być porywający. Zgadza się?

No cóż, to założenie nie zawsze się sprawdza. Coraz częściej buduje się zespoły interdyscyplinarne, które obejmują zarówno frontendowców, jak i backendowców, ale też specjalistów ds. infrastruktury i biznesu. Dzięki różnorodnym perspektywom są one w stanie stworzyć bardziej kreatywne i skuteczne rozwiązania. Te zespoły

Kup ksi k Pole ksi k

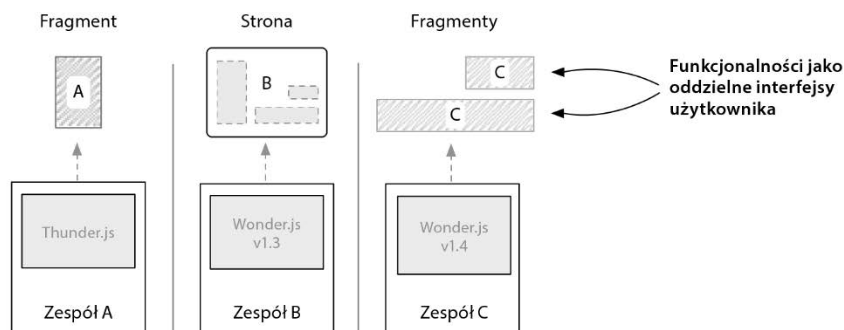
nie budują najlepszej w swoje klasie platformy operacyjnej czy warstwy frontendowej, lecz specjalizują się w realizacji swojej misji. Na przykład starają się stać ekspertami, jeżeli chodzi o *prezentowanie odpowiednich sugestii produktów* lub *zagwarantowanie klientowi bezproblemowej ścieżki zakupowej*. Nie dążą do osiągnięcia mistrzostwa w jakiejś konkretnej technologii, lecz koncentrują się na zapewnieniu możliwie najlepszych wrażeń użytkownika w obszarze, nad którym pracują.

Zespoły wielofunkcyjne mają tę dodatkową zaletę, że wszyscy ich członkowie są bezpośrednio zaangażowani w rozwój danej funkcjonalności. W modelu mikrousług dział operacyjny i działy usługowe nie są w to bezpośrednio zaangażowane. Otrzymują wymagania od zespołów będących ich klientami i nie zawsze dysponują pełnym obrazem. Nie wiedzą, dlaczego te wymagania są istotne. Zespoły wielofunkcyjne sprawiają, że ludziom łatwiej jest się zaangażować, wnosić wkład w projekt i — co najważniejsze — *poczuć się odpowiedzialnym za produkt*.

Zakończyliśmy omawianie podziału na niezależne od siebie systemy oraz zespoły, które są za nie odpowiedzialne. Możemy teraz przejść do kolejnej części diagramu.

1.1.2. Frontend

Przejdziemy teraz do zagadnienia, które sprawia, że podejście mikrofrontendowe różni się od pozostałych modeli architektonicznych. Mówię tu o sposobie myślenia o funkcjonalnościach oraz ich tworzenia. Zespoły mikrofrontendowe są całościowo odpowiedzialne za wybraną funkcjonalność. Dostarczają powiązany interfejs użytkownika jako mikrofrontend, który może być kompletną stroną lub fragmentem do wykorzystania przez inne zespoły. Rysunek 1.4 ilustruje to założenie.



Rysunek 1.4. *Środkowa część przeglądu elementów z rysunku 1.1. Każdy zespół buduje własny interfejs użytkownika jako stronę lub fragment*

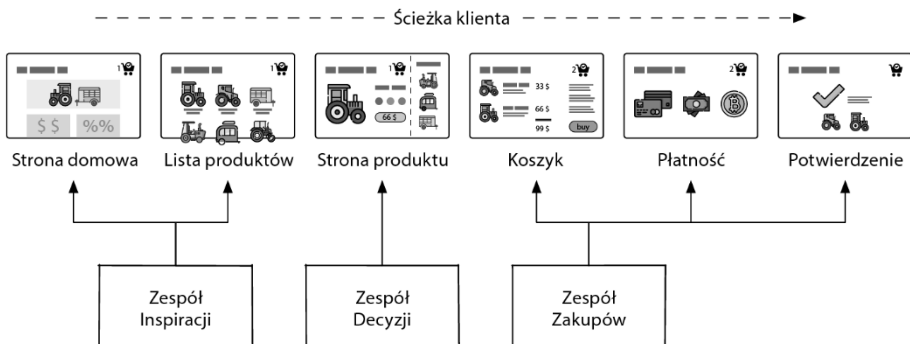
To jest część, w ramach której realizowana jest całość pracy frontendowej. Każdy zespół dostarcza swój *własny frontend*. Jeden zespół tworzy kod HTML, CSS i JavaScript potrzebny do dostarczenia danej funkcjonalności. Aby ułatwić sobie zadanie, zespół może zastosować bibliotekę lub framework JavaScript. Zespoły nie współdzielą bibliotek ani frameworków. Każdy zespół może zdecydować się na

Kup ksi k Pole ksi k

narzędzie, które najlepiej pasuje do postawionego przed nim zadania. Ilustrują to fikcyjne frameworki *Thunder.js* i *Wonder.js*[1]. Zespoły mogą aktualizować zależności niezależnie od siebie. Zespół B korzysta z frameworku Wonder.js w wersji 1.3, podczas gdy Zespół C przeszedł już na kolejną (1.4).

ODPOWIEDZIALNOŚĆ ZA STRONĘ

Porozmawiajmy o stronach. W naszym przykładzie różne zespoły zajmują się różnymi częściami sklepu. Jeżeli podzielimy sklep internetowy na rodzaje stron i spróbujemy przypisać każdy rodzaj do jednego z trzech zespołów, wynik może wyglądać tak, jak pokazano na rysunku 1.5.



Rysunek 1.5. *Za każdą stronę odpowiedzialny jest jeden zespół*

Ponieważ struktura zespołów odpowiada etapom ścieżki klienta, podział w tym przypadku się sprawdza. Strona domowa ma inspirować do poznawania produktów. Strona produktowa szczegółowo prezentuje produkt i pomaga klientowi podjąć decyzję zakupową.

Jak zastosować ten podział w praktyce? Każdy zespół mógłby oczywiście tworzyć własne strony i udostępniać je w domenie publicznej z odrębnej aplikacji. Te strony można by powiązać łączami, aby użytkownik końcowy mógł się między nimi poruszać. Tyle wystarczy? Zasadniczo tak. Ale w prawdziwym świecie istnieją wymagania, które sprawiają, że jest to bardziej skomplikowane. Dlatego właśnie powstała ta książka. Teraz rozumiesz już jednak *istotę architektury mikrofrontendowej*:

- Zespoły mogą pracować niezależnie od siebie w obszarze swoich kompetencji.
- Mogą wykorzystywać technologie, które najlepiej do tego pasują.

---

[1] Tak, zdaję sobie sprawę z tego, że prawie wszystkie wyrazy ze słownika są już zajęte przez jakiś framework JavaScript zarejestrowany na stronie *https://www.npmjs.com*, włączając w to słowa *thunder* i *wonder*. Ponieważ jednak w ramach żadnego z tych projektów nie zarejestrowano aktywności od około trzech lat, a liczba tygodniowych pobrań jest raczej jednocyfrowa, zostańmy przy tych nazwach. :)

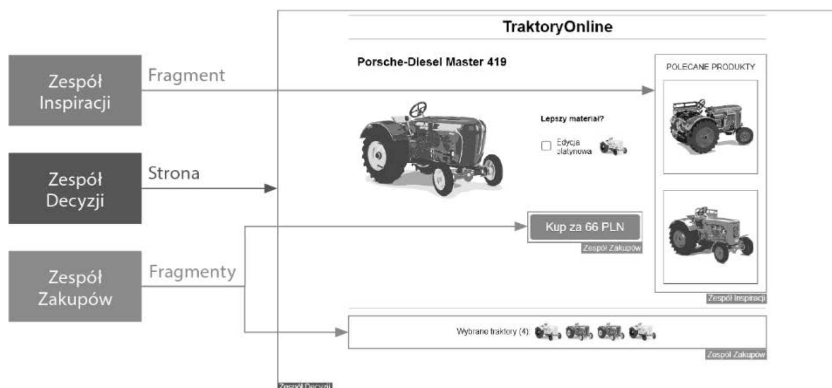
Kup ksi k Pole ksi k

- Ich zależność od wyników pracy innych zespołów jest ograniczona (np. dzięki powiązaniu przez łącza).

FRAGMENTY

Nie zawsze wystarczy mówić o stronach. Często mamy do czynienia z elementami takimi jak nagłówek czy stopka, które pojawiają się na wielu stronach. Nie chcemy, aby każdy zespół tworzył je od początku. W takich przypadkach zastosowanie mają **fragmenty**.

Strona często służy więcej niż jednemu celowi. Może wyświetlać informacje lub udostępniać funkcjonalność, za którą odpowiedzialny jest inny zespół. Na rysunku 1.6 widać stronę produktową witryny TraktoryOnline. Odpowiedzialny za tę stronę jest Zespół Decyzji. Nie wszystkie funkcjonalności i treści są jednak jego autorstwa.



Rysunek 1.6. *Zespoły są odpowiedzialne za strony i fragmenty. Możesz myśleć o fragmentach jako o zagnieżdżonych miniaplikacjach, które są odrębne od reszty strony*

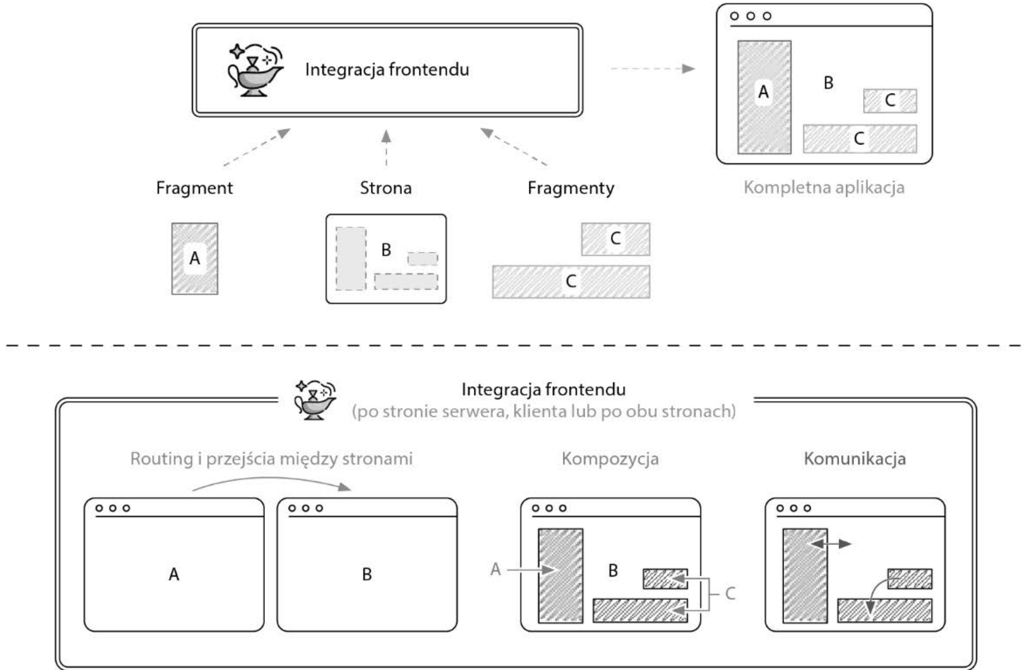
Blok *Polecane produkty* po prawej stronie to element zachęcający klientów do odkrywania oferty sklepu. Tego rodzaju treści są specjalnością Zespołu Inspiracji. Część *Minikoszyk* na dole strony pokazuje wszystkie wybrane modele. Stworzył ją Zespół Zakupów, który posiada informacje na temat zawartości koszyka. Klient może dodać nowy traktor, klikając przycisk *Kup*. Ponieważ ta czynność zmienia zawartość koszyka, również ten przycisk jest udostępniany przez Zespół Zakupów jako fragment.

Zespół może dodać na swojej stronie funkcjonalność przygotowaną przez inny zespół. Niektóre fragmenty mogą potrzebować danych wynikających z kontekstu, takich jak kod produktu dla bloku *Polecane produkty*. Inne fragmenty, w tym minikoszyk, posiadają swój własny stan wewnętrzny. Zespół, który wykorzystuje taki fragment, nie musi jednak znać jego stanu ani szczegółów implementacyjnych.

Kup ksi k Pole ksi k

1.1.3. Integracja frontendu

Rysunek 1.7 pokazuje górną część diagramu prezentującego przegląd elementów architektury mikrofrontendowej. Tutaj wszystko się łączy.



Rysunek 1.7. *Pojęcie integracji frontendu opisuje zbiór technik, których można użyć, aby połączyć elementy interfejsów użytkownika (tj. strony i fragmenty) dostarczane przez zespoły w kompletną aplikację. Możemy wyróżnić trzy kategorie takich technik: routing, kompozycję i komunikację. W zależności od wyborów architektonicznych możemy użyć różnych ich kombinacji*

Termin *integracja frontendu* opisuje zbiór narzędzi i metod, które można wykorzystać, aby połączyć elementy interfejsów użytkownika stworzonych przez różne zespoły w jedną spójną aplikację. Ramka *Integracja frontendu*, powiększona na dole diagramu, pokazuje trzy kategorie technik integracyjnych. Przyjrzyjmy się każdej z nich.

ROUTING I PRZEJŚCIA MIĘDZY STRONAMI

W tym przypadku chodzi o integrację na poziomie strony. Potrzebny jest nam mechanizm pozwalający na przejście ze strony Zespołu A na stronę Zespołu B. Może on być całkiem prosty — tak prosty jak użycie *łączy HTML*. Jeżeli chcesz umożliwić nawigację po stronie klienta, tak aby móc wyświetlić inną treść bez przeładowywania strony, zadanie staje się bardziej skomplikowane. Możemy to osiągnąć dzięki wspólnej **powłoce aplikacji** (ang. *application shell*) lub wykorzystaniu metaframeworku takiego jak **single-spa**. W tej książce przyjrzymy się obu rodzajom rozwiązań.

KOMPOZYCJA

Kompozycja oznacza proces pobierania fragmentów i umieszczania ich w odpowiednich miejscach. Zespół udostępniający stronę zazwyczaj nie pobiera treści fragmentu bezpośrednio, lecz wstawia znacznik lub symbol zastępczy w kodzie HTML w miejscu, gdzie powinien znaleźć się fragment. Oddzielna usługa lub technika kompozycji dokonuje ostatecznego połączenia. Tutaj też istnieją różne sposoby. Możemy je zgrupować w dwie kategorie:

- **kompozycja po stronie serwera** z wykorzystaniem takich technik jak SSI, ESI, Tailor czy Podium,
- **kompozycja po stronie klienta**, np. za pomocą ramek *iframe*, techniki AJAX czy komponentów webowych (Web Components).

W zależności od wymagań, jakim musimy sprostać, możemy wybrać jedną z tych kategorii lub obie.

KOMUNIKACJA

W przypadku aplikacji interaktywnych potrzebujemy również modelu komunikacji. W naszym przykładzie minikoszyk powinien się aktualizować po kliknięciu przycisku *Kup*. Sekcja *Polecane produkty* powinna dostosowywać się do zmiany koloru na stronie produktowej. W jaki sposób strona wpływa na zamieszczone w niej fragmenty? To również zagadnienie z zakresu integracji frontendu.

W drugiej części książki poznamy różne metody integracji. Dowiemy się, jakie mają wady i zalety. W rozdziale 9., „Który rodzaj architektury pasuje do mojego projektu?”, podsumujemy tę część i poznamy pewne wskazówki, które mogą pomóc podjąć dobrą decyzję.

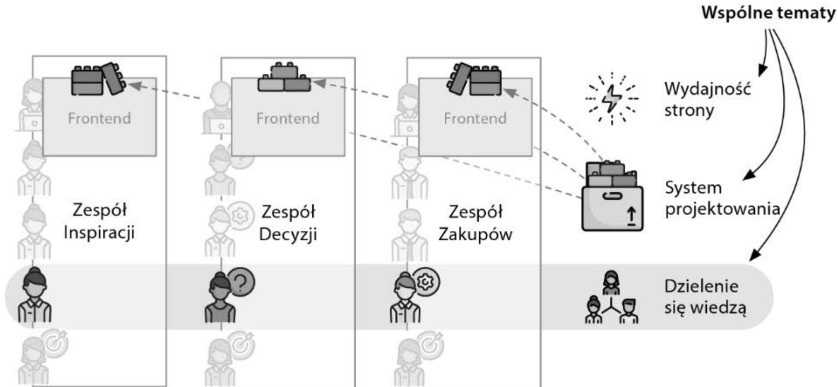
1.1.4. Wspólne tematy

Ideą architektury mikrofrontendowej jest umożliwienie pracy w małych autonomicznych zespołach, które dysponują wszystkimi narzędziami niezbędnymi do tworzenia wartości dla klienta. Nawet w takim modelu istnieje jednak pewien zbiór zagadnień będących przedmiotem zainteresowania wszystkich zespołów. Rysunek 1.8 przedstawia ten rodzaj zagadnień.

WYDAJNOŚĆ STRONY

Ponieważ komponujemy stronę z fragmentów przygotowanych przez różne zespoły, finalnie użytkownik często musi pobrać więcej kodu niż w innym przypadku. Od samego początku należy kontrolować wydajność strony. Dlatego poznamy również użyteczne narzędzia pomiarowe i techniki pozwalające na optymalizację pobierania zasobów. Można uniknąć zbędnego pobierania całych frameworków bez narażania autonomii zespołów. W rozdziałach 10., „Ładowanie zasobów”, i 11., „Wydajność to klucz”, zajmiemy się aspektami wydajnościowymi.

Kup ksi k Pole ksi k



Rysunek 1.8. *Aby zapewnić odpowiedni efekt końcowy i uniknąć zbędnej pracy, od samego początku konieczne jest omówienie zagadnień takich jak wydajność strony, system projektowania oraz dzielenie się wiedzą*

SYSTEM PROJEKTOWANIA

Aby zapewnić spójny wygląd i pozytywne wrażenia klienta, warto wdrożyć *wspólny system projektowania*. Możesz go sobie wyobrazić jako duże pudełko klocków LEGO, z którego każdy zespół może wybrać to, czego potrzebuje. Zamiast plastikowych klocków system projektowania na potrzeby aplikacji webowej będzie zawierał takie elementy, jak przyciski, pola formularzy, czcionki czy ikony. Fakt, że każdy zespół wykorzystuje te same podstawowe komponenty, zapewnia znaczną przewagę w obszarze projektowania. W rozdziale 12., „Interfejs użytkownika i system projektowania", nauczymy się różnych sposobów wprowadzania tych rozwiązań w życie.

DZIELENIE SIĘ WIEDZĄ

Autonomia to podstawa, ale powstawanie silosów wiedzy nie jest pożądane. Z punktu widzenia produktywności nie ma sensu, aby każdy zespół budował własną infrastrukturę do rejestrowania błędów. Wybranie wspólnego rozwiązania lub zastosowanie rozwiązania przygotowanego przez inne zespoły pomaga skoncentrować się na realizacji misji. Należy zadbać o przestrzeń oraz regularne okazje do wymiany wiedzy między zespołami.

Kup ksi k Pole ksi k

1.2. Jakie problemy rozwiązuje architektura mikrofrontendowa?

Teraz wiesz już, czym jest mikrofrontend. Przyjrzyjmy się zatem korzyściom organizacyjnym i technologicznym wynikającym z tego modelu. Zapoznamy się również z najczęstszymi wyzwaniami, jakim trzeba sprostać, aby zapewnić produktywność tego podejścia.

1.2.1. Optymalizacja rozwoju funkcjonalności

Pierwszym i najważniejszym powodem, dla którego firmy wybierają model mikrofrontendowy, jest dążenie do skrócenia czasu potrzebnego na dostarczenie funkcjonalności. W architekturze poziomej („warstwowej") w rozwój jednej funkcjonalności zaangażowanych jest wiele zespołów. Oto przykład. Dział marketingu chce stworzyć nowy rodzaj banera reklamowego. Przedstawiciel strony biznesowej rozmawia z zespołem ds. dostarczania treści o rozszerzeniu istniejącej struktury danych. Zespół ds. dostarczania treści rozmawia z kolei z zespołem frontendowym, aby omówić zmiany w API. Zaplanowano spotkania, napisano specyfikacje. Każdy zespół planuje swoją pracę i rezerwuje na nią czas w jednym z kolejnych sprintów. Jeżeli wszystko pójdzie zgodnie z planem, funkcjonalność będzie gotowa, gdy zakończy ją realizować ostatni zespół. Jeżeli nie, konieczne będą dalsze spotkania, aby omówić zmiany.

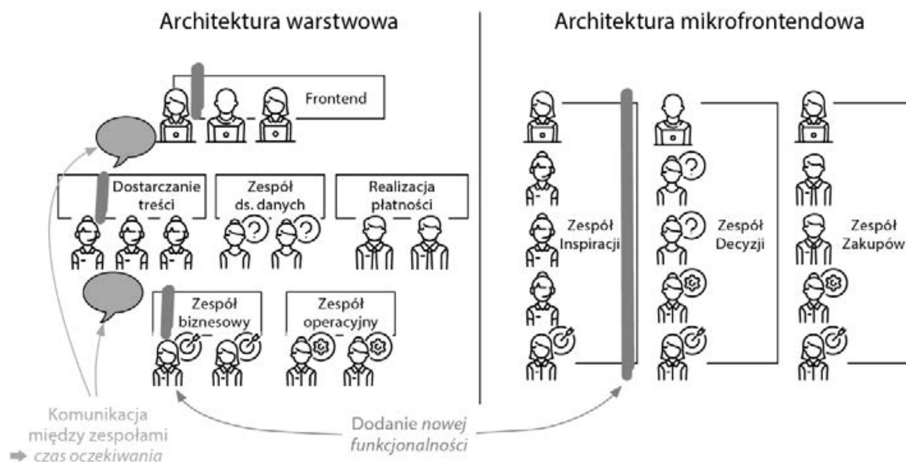
Skrócenie czasu oczekiwania w komunikacji między zespołami to główne założenie architektury mikrofrontendowej.

Dzięki temu modelowi wszystkie osoby zaangażowane w tworzenie danej funkcjonalności należą do tego samego zespołu. Ilość pracy do wykonania się nie zmienia, ale komunikacja wewnątrz zespołu jest błyskawiczna i mniej sformalizowana. Iteracje są szybsze, nie trzeba czekać na pozostałe zespoły, nie ma rozmów na temat priorytetyzacji.

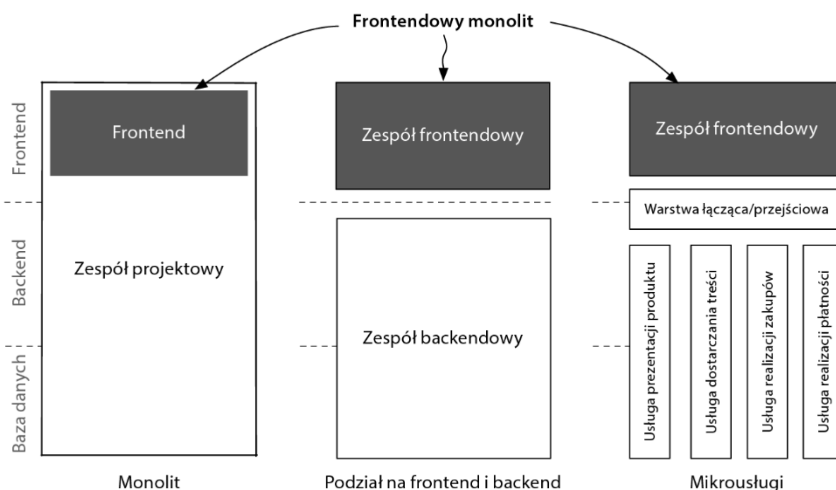
Rysunek 1.9 obrazuje tę różnicę. Architektura mikrofrontendowa pozwala na optymalizację procesu wdrażania funkcjonalności poprzez zebranie wszystkich osób zaangażowanych w jednym miejscu.

1.2.2. Precz z frontendowym monolitem

Większość współczesnych paradygmatów architektonicznych nie oferuje skalowalnej formuły frontendowej. Na rysunku 1.10 widzimy trzy modele: monolit, podział na frontend i backend oraz mikrousługi. We wszystkich przypadkach frontend jest monolitem. Oznacza to, że całość frontendu zdefiniowana jest w jednej bazie kodu i tylko jeden zespół może nad nim wydajnie pracować.



Rysunek 1.9. *Diagram przedstawia proces tworzenia nowej funkcjonalności. W modelu warstwowym (po lewej) funkcjonalność budują trzy zespoły. Muszą koordynować swoją pracę i czekać na siebie nawzajem. Dzięki architekturze mikrofrontendowej (po prawej) całą funkcjonalność tworzy jeden zespół*



Rysunek 1.10. *W większości modeli frontend to system monolityczny*

W modelu mikrofrontendowym cała aplikacja, włączając w to frontend, zostaje podzielona na mniejsze przekrojowe (pionowe) systemy. Każdy zespół dba o własny, relatywnie niewielki frontend. W porównaniu z modelem monolitycznego frontendu budowanie i utrzymanie niewielkich modułów frontendowych niesie istotne korzyści. Moduły mikrofrontendowe:

- można wdrażać niezależnie,
- ograniczają zasięg ewentualnej awarii,
- obejmują mniejszy zakres funkcjonalności i są łatwiejsze do zrozumienia,

- zawierają mniej kodu, co upraszcza ewentualną refaktoryzację i zmiany,
- są bardziej przewidywalne, ponieważ nie współdzielą stanu z innymi systemami.

Przyjrzyjmy się kilku z tych tematów.

1.2.3. Gotowość na zmiany

Ustawiczne uczenie się i wdrażanie nowych technologii to chleb powszedni programisty. Jest to szczególnie prawdziwe w odniesieniu do pracy nad frontendem. Narzędzia i frameworki zmieniają się jak w kalejdoskopie. Era gwałtownego wzrostu złożoności pracy na frontendzie zaczęła się w 2005 roku wraz z koncepcją Web 2.0, frameworkiem Ruby on Rails, biblioteką Prototype.js i techniką AJAX, dzięki którym statyczne strony przekształciły się w interaktywne serwisy.

Od tego czasu wiele się jednak zmieniło. Już od dawna, aby być frontendowcem, nie wystarcza „dobra znajomość HTML i CSS". Teraz frontendowiec to specjalista z dziedziny inżynierii oprogramowania. Musi znać się na takich zagadnieniach, jak responsywność, wrażenia użytkownika, wydajność, komponenty wielokrotnego użytku, testowalność, dostępność, bezpieczeństwo oraz zmiany w standardach webowych i ich wspieraniu przez przeglądarki. Ewolucja frontendowych narzędzi, bibliotek i frameworków umożliwia budowanie aplikacji webowych o wyższej jakości i większych możliwościach, które pozwalają sprostać rosnącym oczekiwaniom użytkowników. Narzędzia takie jak Webpack, Babel, Angular, React, Vue.js, Stencil czy Svelte odgrywają dzisiaj istotną rolę, lecz prawdopodobnie nie jesteśmy jeszcze na końcu drogi. Umiejętność adaptowania nowych technologii w uzasadnionych sytuacjach stanowi istotną przewagę dla zespołu i firmy.

STARY KOD

Obsługa istniejących, starszych systemów również zaczyna być coraz istotniejszym tematem na frontendzie. Duża część czasu pracy programisty może przypadać na refaktoryzację starego kodu oraz opracowywanie strategii migracji. Największe firmy inwestują istotne nakłady pracy w utrzymanie swoich dużych aplikacji. Oto kilka przykładów:

- GitHub wiele lat realizował migrację, która miała na celu usunięcie zależności od biblioteki jQuery[2].
- Firma Trivago, oferująca wyszukiwarkę hoteli, w ramach projektu Ironman podjęła się tytanicznego wysiłku, aby przepisać swój skomplikowany system stylów CSS na modularny system projektowania[3].

---

[2] *Removing jQuery from GitHub.com frontend*, *https://github.blog/2018-09-06-removing-jquery-from-github-frontend*.

[3] Christoph Reinartz, *Large Scale CSS Refactoring at trivago*, *https://medium.com/@pistenprinz/large-scale-css-refactoring-at-trivago-4602113c4a26*.

Kup książkę Poleć książkę

- Etsy jest w trakcie usuwania starego kodu JavaScript. Celem jest ograniczenie rozmiaru pakietu i zwiększenie wydajności strony. Jeden programista nie jest w stanie mieć dobrego oglądu całego systemu, którego kod powstawał latami. Aby zidentyfikować martwy kod, firma zbudowała przeglądarkowe narzędzie do badania pokrycia, które działa na przeglądarkach klientów i przekazuje informacje do serwera[4].

W przypadku aplikacji znacznych rozmiarów, aby utrzymać konkurencyjność, trzeba mieć możliwość wprowadzenia nowych technologii, gdy przynoszą korzyść zespołowi. Ta możliwość nie oznacza jednak, że wskazane jest przepisywanie całego frontendu co kilka lat, aby użyć modnej obecnie technologii.

PODEJMOWANIE DECYZJI NA SZCZEBLU LOKALNYM

Możliwość wprowadzenia technologii i sprawdzenia jej użyteczności w wyizolowanych częściach aplikacji to cenna opcja do wyboru. Nie wymusza konieczności przygotowania planu migracji, który obejmuje dosłownie wszystko.

Podejście mikrofrontendowe umożliwia migracje na poziomie zespołu. Oto przykład. Zespół Zakupów napotkał w ostatnim czasie dużą liczbę błędów czasu wykonania w swoim kodzie JavaScript ze względu na odniesienia do nieokreślonych zmiennych. Ponieważ proces realizacji zakupu powinien być absolutnie pozbawiony błędów, zespół decyduje się użyć języka Elm. Jest to statycznie typowany język kompilowany do JavaScriptu, który zaprojektowano w taki sposób, aby całkowicie zapobiec możliwości powstania błędów czasu wykonania. Nowy pomysł ma swoje wady. Programiści muszą nauczyć się nowego języka i jego zasad. Ekosystem modułów i komponentów o ogólnodostępnym kodzie źródłowym wciąż jest niewielki. Jednak dla Zespołu Zakupów zalety przeważają nad powiązanymi kosztami.

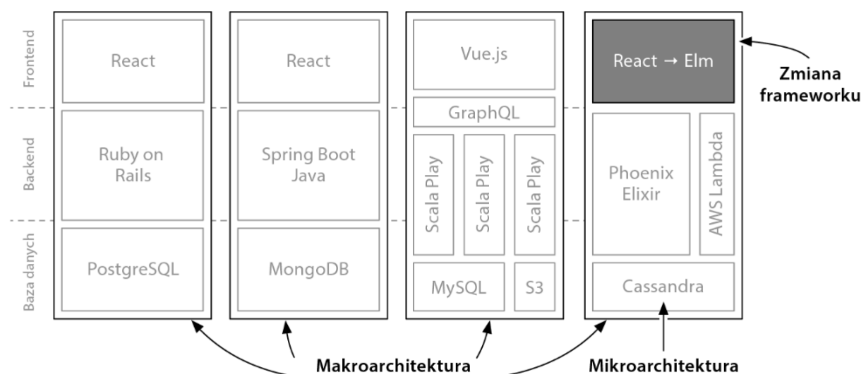
Dzięki podejściu mikrofrontendowemu zespoły mają pełną kontrolę nad wyborem wykorzystywanych technologii (**mikroarchitektura**). Ta autonomia pozwala im podejmować decyzje samodzielnie. Nie muszą konsultować tego z innymi zespołami. Są jedynie zobowiązane zapewnić zgodność ze wspólnie ustalonymi konwencjami (**makroarchitektura**). Może to obejmować stosowanie odpowiednich przestrzeni nazw lub wspieranie wybranej techniki integracji na frontendzie. Więcej o tych konwencjach dowiemy się w dalszej części książki. Na rysunku 1.11 możemy zobaczyć, jak wygląda podział na te dwa poziomy.

W ramach dużej aplikacji o monolitycznym kodzie źródłowym dokonanie takiej zmiany byłoby bardzo poważną decyzją, wymagającą wielu spotkań i dyskusji. Ryzyko byłoby znacznie wyższe, a dla innych części aplikacji lista argumentów za i przeciw mogłaby wyglądać zupełnie inaczej. Proces podejmowania decyzji o tej skali często jest tak trudny, mało produktywny i męczący, że większość programistów nawet nie bierze takich zmian pod uwagę.

---

[4] *Raiders of the Fast Start: Frontend Perf Archeology, https://www.youtube.com/watch?v=qts9gPYoANU.*

Kup ksi k Pole ksi k



Rysunek 1.11. *Zespoły mogą podejmować niezależne decyzje co do architektury stosowanej wewnątrz projektu (mikroarchitektura), ale muszą przestrzegać ustaleń dotyczących makroarchitektury*

Podejście mikrofrontendowe sprawia, że zmienianie aplikacji jest łatwiejsze — zwłaszcza tam, gdzie jest to potrzebne.

1.2.4. Korzyści z niezależności

Autonomia to jedna z najważniejszych korzyści powiązanych z modelem mikrousług i mikrofrontendów. To wygodne, gdy zespoły same podejmują decyzje na temat istotnych zmian takich jak te opisane w poprzednim punkcie. Niesie jednak ze sobą korzyści nawet wtedy, gdy wszyscy pracują w takim samym środowisku, wykorzystując te same narzędzia.

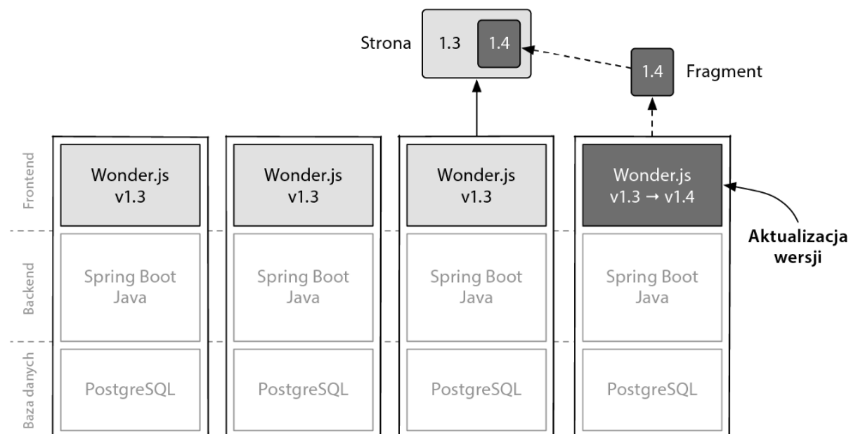
SAMOWYSTARCZALNOŚĆ

Strony i fragmenty są samowystarczalne. Oznacza to, że zawierają cały swój kod HTML, style i skrypty oraz że nie powinny mieć współdzielonych zależności w czasie wykonania. Izolacja sprawia, że zespoły mogą wdrażać nowe funkcjonalności w obrębie fragmentu bez konieczności konsultowania się z innymi zespołami. Aktualizacja może obejmować np. zmianę wersji wykorzystywanego frameworku JavaScript. Ilustruje to rysunek 1.12. Ponieważ każdy fragment jest oddzielną wyspą, zmiana nie jest wielkim wydarzeniem.

Na pierwszy rzut oka pozwolenie, aby każdy zespół korzystał z odrębnych zasobów, może się wydać niegospodarnością — szczególnie w sytuacji, gdy wszystkie zespoły używają tych samych technologii. Takie podejście umożliwia jednak wydajniejszą pracę i szybsze tworzenie funkcjonalności.

DODATKOWE NAKŁADY TECHNICZNE

Z jednej strony mikrousługi na backendzie wprowadzają dodatkowe koszty. Potrzeba więcej zasobów obliczeniowych, np. do tego, aby uruchomić kilka aplikacji Java na osobnej maszynie wirtualnej lub w osobnym kontenerze. Z drugiej jednak strony fakt, że usługi backendowe są o wiele mniejsze niż alternatywny monolit, niesie ze



Rysunek 1.12. *Fragmenty są samowystarczalnymi elementami, które można zmieniać niezależnie od strony, na której są zagnieżdżone*

sobą wiele korzyści. Możemy uruchomić usługę na tańszym, mniej wydajnym sprzęcie. Możemy skalować konkretne usługi poprzez uruchomienie kilku instancji danej aplikacji bez konieczności powielania całego monolitu. Braki zawsze można rozwiązać dodatkowymi nakładami finansowymi na kupno większej liczby serwerów lub serwerów o większej mocy.

Te zalety skalowalności nie mają zastosowania do frontendu. Urządzenia klienta dysponują ograniczoną przepustowością i zasobami. Koszty nie zwiększają się jednak liniowo wraz z liczbą zespołów. W dużym stopniu zależą od sposobu, w jaki zespoły pracują nad swoimi aplikacjami. W rozdziale 11., „Wydajność to klucz", zajmiemy się pomiarami wydajności oraz technikami mającymi na celu przeciwdziałanie niekorzystnym efektom. Można jednak bez dużych wątpliwości powiedzieć, że rozdzielenie zespołów pociąga za sobą dodatkowe koszty.

Dlaczego więc to robimy? Dlaczego nie zbudujemy dużej aplikacji we frameworku React, w ramach której każdy zespół będzie odpowiedzialny za inną część? Jeden zespół mógłby pracować tylko nad komponentami do strony produktowej. Drugi mógłby budować strony pozwalające na zrealizowanie zakupu. Jedno repozytorium kodu źródłowego, jedna aplikacja we frameworku React.

ZERO WSPÓŁDZIELENIA

Wyjaśnieniem jest stwierdzenie faktu, że komunikacja między zespołami jest kosztowna — naprawdę kosztowna. Gdy chcesz zmienić element, który wykorzystują inne zespoły, nawet jeżeli jest to tylko pomocnicza biblioteka, trzeba poinformować wszystkich dookoła, poczekać na odpowiedź i być może omówić wątpliwości. Im więcej osób zaangażowanych, tym bardziej uciążliwy jest to proces.

Celem jest dzielenie się możliwie najmniejszą ilością informacji, aby zapewnić szybsze opracowywanie funkcjonalności. Każdy współdzielony element kodu lub

Kup ksi k   Pole ksi k

infrastruktury może być przyczyną niebagatelnych kosztów związanych z zarządzaniem. Dlatego mówimy tutaj o **architekturze zerowego współdzielenia** (ang. *shared nothing architecture*). Brzmi dość radykalnie i oczywiście rzeczywistość nie jest nigdy czarno-biała, ale ogólnie rzecz biorąc, projekty mikrofrontendowe z założenia akceptują redundancję, jeżeli przynosi to zwiększenie autonomii i przyspiesza rozwój funkcjonalności. Zaobserwujemy tę zasadę w działaniu w wielu miejscach tej książki.

1.3. Wady architektury mikrofrontendowej

Jak wspomnieliśmy wcześniej, model mikrofrontendowy polega przede wszystkim na zapewnieniu autonomicznym zespołom wszystkiego, czego potrzebują, aby przygotować funkcjonalności istotne z punktu widzenia klienta. Autonomia to potężne narzędzie, ale nie ma nic za darmo.

1.3.1. Redundancja

Każdy, kto uczył się informatyki, wie, że należy unikać niepotrzebnego powielania elementów systemu. Może to wymagać normalizacji danych w relacyjnej bazie danych albo zdefiniowania funkcji i zastąpienia nią podobnych fragmentów kodu. Celem jest zwiększenie wydajności i spójności. Wyszkoliliśmy nasze oczy i umysł do wyszukiwania powtarzającego się kodu i znajdowania sposobów jego eliminacji.

Gdy mamy wiele zespołów, które niezależnie od siebie tworzą aplikacje, używając własnego zestawu technologii, jesteśmy wręcz skazani na redundancję. Każdy zespół musi skonfigurować i utrzymywać swój własny serwer aplikacji, proces budowania projektu, określić procedury z zakresu ciągłej integracji. Kod docierający do przeglądarki z dużym prawdopodobieństwem zawiera zduplikowane fragmenty kodu JavaScript lub reguły CSS. Oto kilka przypadków, w których może to być problemem:

- Nie ma jednego miejsca, z którego można by naprawić istotny błąd w powszechnie używanej bibliotece. Wszystkie zespoły, które z niej korzystają, muszą samodzielnie zainstalować i wdrożyć kod naprawiający ten problem.
- Gdy jeden zespół zainwestuje dużo pracy i przyspieszy dwukrotnie proces budowania swojej aplikacji, pozostałe zespoły nie skorzystają automatycznie na tej zmianie. Zespół musi podzielić się tą informacją z innymi, a inne zespoły muszą wdrożyć tę samą optymalizację samodzielnie.

Kup ksi k Pole ksi k

Uzasadnieniem architektury zerowego współdzielenia jest jednak założenie, że koszty wynikające z redundancji są mniejsze niż negatywny wpływ zależności między zespołami.

1.3.2. Spójność

Architektura mikrofrontendowa wymaga, aby wszystkie zespoły miały własną, w pełni niezależną bazę danych. Czasami jednak jeden zespół potrzebuje danych, które należą do drugiego zespołu. W przypadku sklepu internetowego dobrym przykładem są dane na temat produktu. Wszystkie zespoły muszą wiedzieć, jakie produkty oferuje sklep. Najczęstszym rozwiązaniem jest replikacja danych z wykorzystaniem szyny zdarzeń lub systemu doprowadzającego dane. Jeden zespół jest właścicielem danych na temat produktu, a pozostałe regularnie je replikują. Gdy jeden zespół ma awarię, nie wpływa to na resztę, gdyż nadal ma dostęp do lokalnej wersji danych. Mechanizm replikacji jest jednak czasochłonny i wprowadza opóźnienia. W związku z tym zmiany w cenach lub dostępności mogą na krótki okres powodować niespójność danych. Produkt objęty promocją na stronie głównej może być wyświetlany bez zniżki w koszyku zakupowym. Gdy wszystko działa poprawnie, opóźnienia są rzędu milisekund lub sekund, ale w przypadku problemów mogą trwać dłużej.

Podsumowując, za stabilność systemu płaci się pewną cenę. Trzeba zaakceptować ryzyko czasowej niespójności danych.

1.3.3. Różnorodność technologii

Swobodny wybór technologii to jedna z najistotniejszych zalet architektury mikrofrontendowej, ale jednocześnie jeden z punktów, który wzbudza wiele kontrowersji. Czy naprawdę chcemy, aby każdy zespół wykorzystywał zupełnie odmienne technologie? Utrudni to przepływ programistów między zespołami oraz wymianę najlepszych praktyk.

Możliwość wybrania odmiennych technologii nie oznacza jednak, że trzeba z niej skorzystać. Nawet gdy wszystkie zespoły zdecydują się korzystać z tych samych rozwiązań, nadal odnosimy najważniejsze korzyści w postaci możliwości przeprowadzania niezależnych aktualizacji oraz mniejszej potrzeby konsultowania się między zespołami.

W projektach, w ramach których pracowałem, miałem do czynienia z różnymi poziomami zróżnicowania technologii. W niektórych obowiązywały te same technologie, a w innych zespoły mogły wybrać z listy sprawdzonych narzędzi te, które im najbardziej pasowały. Aby zapewnić spójną wizję projektu, już na samym początku należy omówić, jaki poziom autonomii zespołów i zróżnicowania technologicznego jest do przyjęcia w ramach projektu i dla firmy.

Kup ksi k Pole ksi k

1.3.4. Więcej kodu frontendowego

Jak wspomniałem wcześniej, strony budowane zgodnie z architekturą mikrofrontendową zazwyczaj wymagają większej ilości kodu JavaScript i definicji stylów CSS. Budowanie fragmentów, które mogą działać niezależnie od innych, wprowadza pewną nadmiarowość. Ilość niezbędnego kodu nie rośnie liniowo wraz z liczbą zespołów lub fragmentów. Niemniej jednak od samego początku absolutnie niezbędne jest kontrolowanie wydajności strony.

1.4. Kiedy stosować model mikrofrontendowy?

Podobnie jak inne modele, również architektura mikrofrontendowa nie stanowi panaceum na wszystkie programistyczne bolączki. Należy zrozumieć zarówno korzyści, jak i ograniczenia, jakie ze sobą niesie.

1.4.1. Dla średnich i dużych projektów

Model mikrofrontendowy to technika, która ułatwia skalowanie projektów. Kiedy pracujesz nad aplikacją w kilkuosobowym zespole, skalowanie systemu prawdopodobnie nie jest największym wyzwaniem, z jakim się borykasz. Niezłym wyznacznikiem odpowiedniej wielkości zespołu jest zasada dwóch pizz spopularyzowana przez Jeffa Bezosa, prezesa firmy Amazon[5]. Zasada ta mówi, że zespół jest za duży, jeśli nie można go nakarmić dwoma dużymi pizzami. W większych grupach spada wydajność komunikacji, a podejmowanie decyzji staje się skomplikowanym procesem. W praktyce możemy przyjąć, że idealna wielkość zespołu waha się od 5 do 10 osób.

Gdy zespół jest większy, warto rozważyć jego podzielenie. Jedną z opcji, które należy wziąć pod uwagę, jest podział w stylu mikrofrontendowym — na zespoły o pionowych zakresach odpowiedzialności. Pracowałem w różnych projektach mikrofrontendowych w obszarze handlu elektronicznego. Składało się na nie od 2 do 6 zespołów, liczących w sumie od 10 do 50 osób. Dla projektów o tej skali model mikrofrontendowy sprawdza się całkiem nieźle. Nie jest to jednak jedyny scenariusz.

Firmy takie jak Zalando, IKEA czy DANZ wykorzystują podział na przekrojowe obszary odpowiedzialności na dużo większą skalę. Każdy zespół jest odpowiedzialny za wąski zbiór funkcjonalności. Aby zwiększyć wydajność pracy, oprócz zespołów skoncentrowanych na opracowywaniu funkcjonalności Spotify wprowadziło np.

---

[5] Janet Choi, *Why Jeff Bezos' Two-Pizza Team Rule Still Holds True in 2018, http://blog.idonethis.com/two-pizza-team.*

zespoły ds. infrastruktury, zwane tam *Infrastructure Squads*, które stanowią wsparcie dla zespołów zajmujących się funkcjonalnościami i budują dla nich narzędzia takie jak testy A/B. W rozdziale 13., „Zespoły i granice”, zgłębimy ten i powiązane z nim tematy.

1.4.2. Najlepiej działa w sieci

Chociaż założenia architektury mikrofrontendowej nie są ograniczone do konkretnej platformy, działają najlepiej w odniesieniu do aplikacji webowych. W tym przypadku można wykorzystać otwartość sieci.

NATYWNY MONOLIT

Aplikacje natywne na platformy kontrolowane, takie jak iOS czy Android, z założenia mają monolityczny charakter. Przeprowadzenie kompozycji i podmienienie funkcjonalności w locie nie jest możliwe. Aby zaktualizować aplikację natywną, trzeba dostarczyć pojedynczy pakiet aplikacji, który jest poddawany procesowi przeglądu Apple lub Google. Sposobem na obejście tego jest pobieranie części aplikacji z sieci. Zagnieżdżone przeglądarki i obiekty WebView mogą pomóc ograniczyć części aplikacji napisane w natywnym kodzie do minimum. Gdy potrzebne jest zaimplementowanie natywnego interfejsu użytkownika, trudno zagwarantować, że zespoły nie będą wchodziły sobie w paradę.

Oczywiście w przypadku pionowego podziału zespołów każdy zespół może mieć swój webowy frontend i udostępniać funkcjonalność przez interfejs REST. Można utworzyć inne interfejsy użytkownika jako aplikacje natywne bazując na tych interfejsach. Aplikacja natywna mogłaby wykorzystywać istniejącą logikę biznesową opracowaną przez zespoły. W dalszym ciągu mielibyśmy jednak do czynienia z monolityczną warstwą obejmującą całość aplikacji. Podsumowując, jeżeli docelową platformą jest przeglądarka, architektura mikrofrontendowa ma duże szanse się sprawdzić. Jeżeli jednym z celów jest zaoferowanie natywnej aplikacji mobilnej, musimy być gotowi na pewne ustępstwa. W tej książce skoncentrujemy się na aplikacjach webowych i nie będziemy omawiać strategii, które mogłyby mieć zastosowanie w przypadku użycia modelu mikrofrontendowego do tworzenia aplikacji mobilnych.

JEDEN ZESPÓŁ, WIELE FRONTENDÓW

Nie musi być też tak, że jeden zespół jest odpowiedzialny tylko za jeden frontend. W przypadku sklepów internetowych aplikacje często składają się z dwóch części: jednej obsługującej klientów i drugiej przeznaczonej dla pracowników. Zespół tworzący moduł zakupów dla użytkownika końcowego może np. zaimplementować powiązaną funkcjonalność wsparcia technicznego dla pracowników infolinii. Może też zbudować wersję modułu zakupów wykorzystującą obiekty WebView, które można zagnieździć w aplikacji natywnej.

Kup ksi k Pole ksi k

1.4.3. Produktywność i koszty

Podzielenie aplikacji na autonomiczne systemy przynosi wiele korzyści. Nie jest jednak pozbawione pewnych wad.

PRZYGOTOWANIA

Gdy zaczynamy nowy projekt, konieczne jest znalezienie odpowiednich granic między zespołami, początkowa konfiguracja systemów oraz przygotowanie strategii integracji. Musimy również ustalić pewne zasady obowiązujące wszystkie zespoły, takie jak stosowanie przestrzeni nazw. Ważne jest też, aby zapewnić możliwość dzielenia się wiedzą między zespołami.

ZŁOŻONOŚĆ ORGANIZACYJNA

Z jednej strony niewielkie, przekrojowe systemy charakteryzują się mniejszą złożonością techniczną. Z drugiej — system rozproszony sam w sobie dodaje pewien poziom złożoności.

W porównaniu z monolityczną aplikacją pojawia się kilka nowych kategorii problemów, które trzeba uwzględnić. Który zespół będziemy nękać telefonami w weekend, gdy nie będzie można dodać produktu do koszyka? Przeglądarka to współdzielone środowisko operacyjne. Zmiana wprowadzona przez jeden zespół może mieć negatywny wpływ na całą stronę. Nie zawsze łatwo wskazać winnych.

Prawdopodobnie konieczne też będzie wprowadzenie dodatkowej usługi do integracji frontendu. Gdy dokonamy dobrych wyborów, może się okazać, że nie wymaga ona dużych nakładów pracy na utrzymanie. Jest to jednak dodatkowy element układanki, o którym trzeba pamiętać.

Uwzględniając powyższe, należy jednak stwierdzić, że korzyści wynikające z poprawnie zaimplementowanej architektury mikrofrontendowej powinny przewyższać problemy związane z dodatkową złożonością organizacyjną.

1.4.4. Kiedy unikać mikrofrontendów?

To oczywiste, że model mikrofrontendowy nie współgra z każdym projektem. Jak wspomniałem wcześniej, jest to dobre rozwiązanie, gdy musimy zapewnić skalowalność. Jeżeli mamy garstkę programistów i problem utrudnień w komunikacji nie istnieje, wprowadzenie rozwiązań mikrofrontendowych nie przyniesie wielu korzyści.

Aby móc dokonać właściwego podziału systemu, niezwykle ważna jest dobra znajomość dziedziny, w której się pracuje. W idealnych warunkach to, który zespół jest odpowiedzialny za daną funkcjonalność, powinno być zrozumiałe samo przez się. Gdy misje zespołów nie są jasno sprecyzowane lub nakładają się na siebie, może to powodować wiele niejasności i niekończące się dyskusje.

Kup ksi k Pole ksi k

Rozmawiałem z osobami pracującymi w start-upach, które wypróbowywały ten model. Wszystko działało świetnie, dopóki firma nie zdecydowała się diametralnie zmienić modelu biznesowego. Można oczywiście zreorganizować zespoły i powiązane oprogramowanie, ale powoduje to dużo tarć i dodatkowej pracy. Inne rozwiązania organizacyjne są bardziej elastyczne.

Jeżeli potrzeba wiele różnych aplikacji oraz natywnych interfejsów użytkownika, które będą działać na każdym urządzeniu, jeden zespół może nie podołać. Netflix słynie z tego, że ma aplikację na prawie każdą istniejącą platformę, w tym na telewizory, dekodery SBT, konsole do gier, telefony i tablety. Dla każdej z tych platform wyznaczono specjalny zespół zajmujący się interfejsem użytkownika. Z drugiej strony przeglądarki stają się coraz potężniejsze i popularniejsze jako platforma dla aplikacji, dzięki czemu można obsłużyć wiele platform, korzystając z tego samego kodu.

1.4.5. Kto używa mikrofrontendów?

Opisane tutaj koncepcje i pomysły nie są nowe. Amazon nie mówi wiele o wewnętrznej strukturze swoich zespołów programistycznych. Pojedyncze historie sugerują jednak, że już od dłuższego czasu strona sklepu jest tworzona w taki właśnie sposób. Amazon stosuje technikę integracji interfejsów użytkownika, która łączy różne części strony, zanim dotrą do klienta.

Architektura mikrofrontendowa jest dość popularna w sektorze handlu elektronicznego. W 2012 roku Otto Group[6], niemiecka firma zajmująca się sprzedażą wysyłkową i jeden z największych graczy w dziedzinie handlu elektronicznego, rozpoczęła dzielenie swojego monolitycznego systemu. Na ten model przeszła również IKEA[7], szwedzka firma meblowa, oraz Zalando[8], jeden z największych sprzedawców odzieży w Europie. Niemiecka sieć księgarni Thalia[9] przebudowała swój sklep z e-bookami, dzieląc go na przekrojowe części w celu zwiększenia szybkości pracy.

Inne branże też nie stronią od architektury mikrofrontendowej. Spotify[10] dzieli swoich pracowników na autonomiczne zespoły (ang. *squads*) o przekrojowych zakresach odpowiedzialności. Firma SAP[11] opublikowała framework do integracji

[6] Guido Steinacker, *On Monoliths and Microservices*, *https://www.otto.de/jobs/technology/techblog/artikel/on-monoliths-and-microservices_2015-09-30.php*.

[7] Jan Stenberg, *Experiences Using Micro Frontends at IKEA*, *https://www.infoq.com/news/2018/08/experiences-micro-frontends*.

[8] *Project Mosaic | Microservices for the Frontend*, *https://www.mosaic9.org*.

[9] Marcus Gruber, *Another One Bites the Dust — Wie ein Monolith kontrolliert gesprengt wird… Teil I* (w języku niemieckim), *http://tech.thalia.de/another-one-bites-the-dust-wie-ein-monolith-kontrolliert-gesprengt-wird-teil-i*.

[10] *Spotify engineering culture*, *https://engineering.atspotify.com/2014/03/27/spotify-engineering-culture-part-1*.

[11] *Luigi — The Enterprise-Ready Micro Frontend Framework*, *https://luigi-project.io*.

Kup ksi k Pole ksi k

różnych aplikacji. DAZN, platforma streamingowa oferująca transmisje sportowe, również przebudowuje swój monolityczny frontend zgodnie z założeniami architektury mikrofrontendowej[12].

1.5. Podsumowanie

- Model mikrofrontendowy to podejście architektoniczne, a nie konkretna technika.
- W tym modelu granice zespołów nie pokrywają się z granicą między frontendem a backendem. Zespoły są wielofunkcyjne.
- W podejściu mikrofrontendowym aplikacja zostaje podzielona na wiele pionowych części, które rozciągają się od bazy danych po interfejs użytkownika.
- Każdy taki autonomiczny i kompletny system jest mniejszy i bardziej skupiony wokół swojej misji, a przez to łatwiejszy do zrozumienia, testowania i refaktoryzacji niż monolit.
- Technologie frontendowe szybko się zmieniają. Dysponowanie łatwym sposobem na wprowadzanie gruntownych zmian na frontendzie zapewnia cenną przewagę.
- Dobrą strategią jest wyznaczenie granic między zespołami w taki sposób, aby pokrywały się ze ścieżką zakupową i potrzebami klienta.
- Zespoły powinny mieć jasno sformułowaną misję, np. „Pomóc klientowi znaleźć produkty, których szuka”.
- Zespół może być odpowiedzialny za całą stronę lub fragment zawierający daną funkcjonalność.
- Fragment to samowystarczalna miniaplikacja, która zawiera w sobie wszystko, co jest potrzebne do jej działania.
- Model mikrofrontendowy zazwyczaj skutkuje przesyłaniem do przeglądarki większej ilości kodu. Od samego początku absolutnie niezbędne jest monitorowanie wydajności strony.
- Istnieje wiele technik integracji frontendu, które działają po stronie klienta (przeglądarki) lub serwera.

[12] *Micro Frontend Architecture — Luca Mezzalira, DAZN, https://www.youtube.com/watch?v=BuRB3djraeM.*

- Wspólny system projektowania pomaga zapewnić spójny wygląd interfejsów użytkownika różnych zespołów.
- Dobry pionowy podział systemu wymaga wiedzy na temat specyfiki biznesowej firmy. Zmienianie zakresów odpowiedzialności poniewczasie generuje tarcia.

Kup ksi k Pole ksi k

Kup ksi k Pole ksi k

PROGRAM PARTNERSKI

GRUPY HELION

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl

Kilka lat temu świat uznał aplikacje monolityczne za przestarzałe i nie dość elastyczne, jednak do niedawna pojęcie mikrousług dotyczyło wyłącznie backendu. Naturalną konsekwencją ich rozwoju stało się budowanie na podobnych zasadach architektury frontendu. Mikrofrontendy zapewniają elastyczność i łatwość utrzymania. Pozwalają na zaprojektowanie systemu jako zbioru samowystarczalnych komponentów obejmujących własne interfejsy, logikę i bazy danych. Połączenie tych niezależnie rozwijanych elementów następuje w przeglądarce użytkownika. Rozwiązanie takie jest z powodzeniem wykorzystywane przez najważniejszych graczy na rynku.

To książka przeznaczona dla programistów aplikacji internetowych, architektów oprogramowania i inżynierów. Wyjaśniono w niej ideę podziału monolitu na komponenty i pokazano, w jaki sposób zastosować z powodzeniem architekturę mikrousług do frontendu aplikacji. Omówiono też takie zagadnienia jak kompozycja po stronie klienta i po stronie serwera, routing czy zapewnienie spójnego wyglądu. Nie zabrakło wartościowych uwag na temat organizacji pracy zespołów programistów służącej zwiększeniu korzyści z zastosowania architektury mikrofrontendów. Książka prezentuje praktyczne podejście: w kolejnych rozdziałach pokazano poszczególne etapy pracy nad w pełni funkcjonalną aplikacją internetową.

Najważniejsze zagadnienia:

- **tworzenie aplikacji internetowych złożonych z komponentów**
- **strategie integracji: AJAX, SSI i inne**
- **zasadność wyboru architektury mikrofrontendowej**
- **zapewnienie spójności wszystkim interfejsom użytkownika w całej aplikacji**
- **budżet wydajności i strategie ładowania zasobów**

Architektura mikrofrontendowa: niezawodność po stronie frontendu!

Michael Geers jest doświadczonym projektantem aplikacji, specjalizuje się w budowie interfejsów użytkownika. Pierwsze aplikacje sieciowe napisał w wieku kilkunastu lat. Ostatnio tworzy klienty o architekturze wertykalnej. Często występuje na konferencjach branżowych, pisze też artykuły do czasopism technicznych.

helion.pl

HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

Sprawdź nasze szkolenia!

HELIONSZKOLENIA.PL

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-283-7781-3

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 79,00 zł