Tworzenie mikrofrontendów

Skalowanie zespołów i projektów, nowe możliwości dla twórców oprogramowania

Helion

Luca Mezzalira

Tytuł oryginału: Building Micro-Frontends: Scaling Teams and Projects, Empowering Developers

Tłumaczenie: Anna Mizerska

ISBN: 978-83-283-9318-9

© 2022 ***Helion S.A.***

Authorized Polish translation of the English *Building Micro-Frontends* ISBN 9781492082996 © 2022 Luca Mezzalira.

This translation is published and sold by permissi on of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

Polish edition copyright © 2022 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.

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

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

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści

Przedmowa 9

Wstęp 11

1. Architektury frontendowe 17
Aplikacje mikrofrontendowe 17
Aplikacje jednostronicowe 18
Aplikacje izomorficzne 21
Statyczne strony internetowe 23
Jamstack 23
Podsumowanie 24

2. Zasady mikrofrontendu 25
Od monolitu do mikrousług 26
  Przejście na mikrousługi 27
  Wprowadzenie do mikrofrontendów 28
Zasady mikrousług 30
  Model dla każdej domeny biznesowej 31
  Kultura automatyzacji 32
  Ukrycie szczegółów implementacji 32
  Zdecentralizowane zarządzanie 32
  Niezależne wdrażanie 33
  Izolacja awarii 33
  Łatwa obserwowalność 33
Zastosowanie zasad w mikrofrontendach 33
  Model dla każdej domeny biznesowej 33
  Kultura automatyzacji 34
  Ukrycie szczegółów implementacji 34
  Zdecentralizowane zarządzanie 34

Kup książkę Poleć książkę

Niezależne wdrażanie 34
Izolacja awarii 35
Łatwa obserwowalność 35
Mikrofrontendy to nie panaceum 35
Podsumowanie 36

3. Architektury mikrofrontendowe i ich wyzwania 37
Podstawowe decyzje w architekturze mikrofrontendowej 37
Definiowanie mikrofrontendów 38
Podejście DDD z mikrofrontendami 39
Określanie ograniczonego kontekstu 41
Kompozycje mikrofrontendów 42
Trasowanie mikrofrontendów 45
Komunikacja mikrofrontendów 46
Mikrofrontendy w praktyce 49
Zalando 49
Hello Fresh 49
Allegro 50
Spotify 50
SAP 51
OpenTable 51
DAZN 51
Podsumowanie 52

4. Odkrywanie architektur mikrofrontendowych 53
Podstawowe decyzje związane z mikrofrontendem w praktyce 53
Podział pionowy 54
Podział poziomy 55
Analiza architektury 57
Architektura i kompromisy 58
Architektury podziału pionowego 59
Powłoka aplikacji 59
Wyzwania 61
Implementacja systemu projektowania 68
Komfort pracy programisty (DX) 70
SEO 71
Wydajność a mikrofrontendy 72
Dostępne platformy programistyczne 75
Przypadki użycia 76
Charakterystyka architektury 76
Architektury podziału poziomego 78
Implementacja po stronie klienta 79
Wyzwania 82

Kup ksi k Pole ksi k

SEO 90
Komfort pracy programisty (DX) 90
Przypadki użycia 91
Wtyczka Module Federation 93
Elementy iframe 98
Komponenty sieciowe 105
Kompozycja po stronie serwera 109
Kompozycja po stronie serwera brzegowego 119
Podsumowanie 123

5. Techniczne wdrażanie mikrofrontendów 125
Projekt 125
Module Federation — podstawy 128
Implementacja techniczna 130
Struktura projektu 130
Powłoka aplikacji 132
Mikrofrontend uwierzytelniania 137
Mikrofrontend katalogu 139
Mikrofrontend zarządzania kontem 140
Rozwój projektu 144
Wbudowanie przestarzałej aplikacji 144
Tworzenie interfejsu finalizacji zakupu 146
Implementacja dynamicznych kontenerów zdalnych 148
Przywiązanie do bundlera webpack 148
Podsumowanie 149

6. Tworzenie i wdrażanie mikrofrontendów 150
Zasady automatyzacji 151
Szybka informacja zwrotna 151
Częste uruchamianie zautomatyzowanych procesów 153
Motywacja zespołów 153
Określenie ram 154
Stworzenie strategii testowania 154
Komfort pracy programisty (DX) 155
Podział poziomy a podział pionowy 155
Mikrofrontendowe strategie eliminujące zakłócenia 156
Strategie związane ze środowiskami 157
Kontrola wersji 157
Monorepo 158
Polyrepo 161
Przyszłość systemu kontroli wersji 163

Kup ksi k Pole ksi k

Strategie ciągłej integracji 164
Testowanie mikrofrontendów 165
Funkcje przystosowania 169
Działania szczególne dla mikrofrontendu 171
Strategie wdrażania 171
Wdrażanie metodą blue-green a publikacje kanarkowe 172
Wzorzec Dusiciel 175
Obserwowalność 176
Podsumowanie 177

7. Zautomatyzowany proces dla mikrofrontendów — studium przypadku 179
Informacje wstępne 179
Kontrola wersji 181
Uruchomienie procesu 181
Przegląd jakości kodu 182
Kompilacja 184
Przegląd po kompilacji 184
Wdrożenie 186
Podsumowanie strategii automatyzacji 186
Podsumowanie 187

8. Wzorce projektowe dla mikrofrontendów 188
Integracja API i mikrofrontendy 188
Słownik usług 190
Brama API 196
Wzorzec BFF 201
Warstwa GraphQL z mikrofrontendami 206
Sprawdzone metody 211
Podsumowanie 213

9. Od frontendu monolitycznego do mikrofrontendów — studium przypadku 215
Kontekst 216
Stos technologiczny 216
Platforma i główne sekwencje działań użytkownika 217
Cele techniczne 220
Strategia migracji 220
Podstawowe decyzje związane z mikrofrontendami w praktyce 221
Podział aplikacji jednostronicowej na subdomeny 224
Wybór technologii 228
Szczegóły implementacji 231
Zadania powłoki aplikacji 231
Inicjalizacja aplikacji 231
Komunikacja 232

Kup ksi k Pole ksi k

Integracja z backendem 233
Uwierzytelnianie przez mikrofrontendy 234
Zarządzanie zależnościami 236
Integracja systemu projektowania 238
Wspólne komponenty 238
Implementacja publikacji kanarkowych 239
Lokalizacja 241
Podsumowanie 242

10. Wprowadzenie mikrofrontendów w Twojej organizacji 244
Dlaczego powinniśmy używać mikrofrontendów? 244
Połączenie między organizacjami i architekturą oprogramowania 245
Innowacyjność komitetów 246
Zespoły od funkcjonalności a zespoły od komponentów 248
Zarządzenie przepływami komunikacji 252
Dokument RFC 252
Dokument ADR 253
Techniki ulepszania przepływów komunikacji 254
Praca wstecz 255
Społeczności praktyków i spotkania całej załogi 256
Zarządzanie zewnętrznymi zależnościami 257
Zdecentralizowana organizacja 258
Decentralizacja a mikrofrontendy 260
Podsumowanie 262

A. Co społeczność sądzi o mikrofrontendach? 264

Kup ksi k

Pole ksi k

Kup ksi k Pole ksi k

ROZDZIAŁ 5.

Techniczne wdrażanie mikrofrontendów

W tym rozdziale będziemy się opierać na kluczowych decyzjach związanych z architekturą mikrofrontendową w celu zbudowania strony sklepu internetowego z zastosowaniem jednego z technicznych podejść opisanych w rozdziale 4. Jak już wcześniej wspomnieliśmy, jeśli chodzi o architekturę, nie ma jednego uniwersalnego rozwiązania. Cele projektu, struktura organizacji i komunikacja oraz umiejętności techniczne pracowników są czynnikami, które musimy wziąć pod uwagę, wybierając odpowiednie podejście.

Po zidentyfikowaniu kontekstu, w którym będziemy działać, możemy się posiłkować podstawowymi decyzjami architektury mikrofrontendowej, by określić filary technicznego kierunku naszej architektury. Zamiast tworzyć ten sam przykład w wielu platformach, chcę Ci pomóc stworzyć odpowiedni model myślowy, dzięki któremu opanujesz wszystkie platformy mikrofrontendowe, zamiast zapamiętać tylko jedną lub dwie dostępne możliwości.

Z pewnością przyjrzymy się bliżej kilku liniom kodu, ale główny nacisk będę kładł na zrozumienie, *dlaczego* właśnie taka decyzja została podjęta. W ten sposób — bez względu na to, jakie podejście zastosujesz w swoim następnym projekcie — będziesz potrafił określić prawidłowy kierunek, niezależnie od tego, jak bardzo znasz określoną platformę mikrofrontendową.

Stare przysłowie mówi: „Daj człowiekowi rybę, a nakarmisz go na jeden dzień. Naucz człowieka łowić, a nakarmisz go na całe życie”. Nauczymy się zatem łowić.

Projekt

Nasz projekt zakłada budowę strony sklepu internetowego z drobiazgami dla wewnętrznej organizacji w przedsiębiorstwie. Ta strona ma się składać z kilku subdomen, w tym takich jak:

- logowanie,
- płatność,
- katalog z drobiazgami,
- zarządzanie kontem,
- wsparcie pracownika,
- najczęściej zadawane pytania (FAQ).

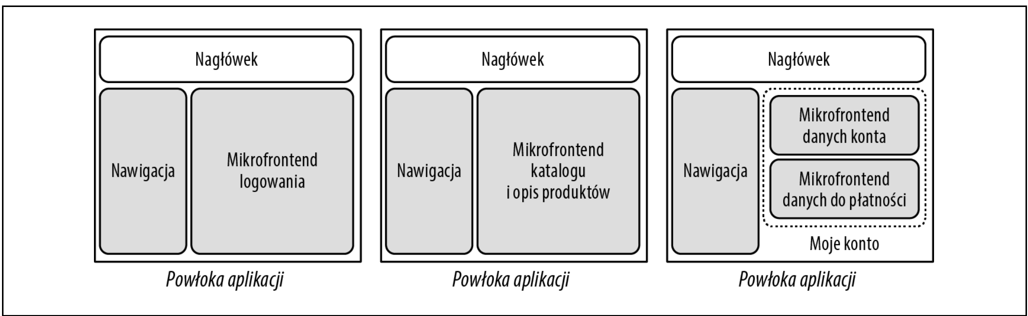
Kup ksi k Pole ksi k

Na potrzeby naszego przykładu w tym rozdziale ograniczymy się tylko do trzech z nich. Będą to: uwierzytelnianie, katalog i zarządzanie kontem. Strona sklepu internetowego musi mieć spójny interfejs użytkownika, by użytkownicy sprawnie poruszali się po stronie w celu zakupu swoich ulubionych drobiazgów. Za ten projekt będzie odpowiedzialnych kilka zespołów. Aby ukończyć projekt na czas, dział techniczny zdecydował się na ponowne wykorzystanie wewnętrznego silnika stworzonego dla rozwiązań handlu internetowego B2C. To jest backend monolityczny, który został przetestowany, był budowany kilka lat i miał okazję pracować w środowiskach produkcyjnych. Jednak dział techniczny chciałby zastosować mikrofrontendy i stworzyć niezależne zespoły odpowiedzialne za subdomeny nowej strony sklepu internetowego. Programiści backendowi zajmą się zmianą architektury backendu z wykorzystaniem mikrousług. Praca zespołów będzie zwinna zarówno na poziomie biznesowym, jak i technicznym.

Kolejny krok polega na przypisaniu poszczególnym zespołom subdomen, za które będą odpowiedzialne:

- *Zespół Sashimi* będzie odpowiedzialny za subdomenę uwierzytelniania. Ponieważ jest to wewnętrzna strona e-handlu, zespół zaimplementuje formularz logowania, używając dostępnego scentralizowanego systemu uwierzytelniana, używanego na co dzień przez wszystkich pracowników do logowania się do systemów wewnątrz organizacji. Ten zespół będzie również odpowiedzialny za uwierzytelnianie użytkownika i za dane osobowe dla mikrofrontendu zarządzania kontem. Jeden członek zespołu będzie programistą full stack, a reszta skupi się na integracji backendu z usługą Microsoft Active Directory (AD) (*https://pl.wikipedia.org/wiki/Active_Directory*).
- *Zespół Maki* zajmie się główną domeną — katalogiem drobiazgów. To największy zespół, który będzie odpowiedzialny za zapewnienie pozytywnych doświadczeń użytkownika podczas korzystania z aplikacji. Zespół będzie podzielony na programistów backendowych i frontendowych.
- *Zespół Nigiri* będzie odpowiedzialny za subdomenę płatności. Będzie integrował różne metody płatności, takie jak karty kredytowe czy PayPal.

Implementowany przez nas przepływ składa się z trzech części. Na rysunku 5.1 pokazano, że mikrofrontendy uwierzytelniania i katalogu będą miały podział pionowy, a zarządzanie kontem będzie złożone z dwóch mikrofrontendów w podziale poziomym. Nad mikrofrontendem uwierzytelniania może pracować tylko jeden programista frontendowy, ponieważ główny ciężar leży po stronie backendu. Jeśli jednak chodzi o katalog, chcemy wywrzeć jak najlepsze wrażenie na użytkowniku, będziemy więc mieli zespół złożony z doświadczonych programistów frontendowych. Ponieważ zarządzanie kontem jest częścią wspólną dla różnych subdomen, dwa zespoły odpowiedzialne za te dwie subdomeny pomogą tworzyć ten widok.



Rysunek 5.1. Części aplikacji zakupowej: logowanie, katalog z opisem produktów i zarządzanie kontem

Kup ksi k   Pole ksi k

W oparciu o podstawowe decyzje i po sprawdzeniu ich założeń zespoły postanowiły zastosować:

Podejście hybrydowe do układu mikrofrontendów

Zamiast używać jednego z dwóch podziałów, pionowego lub poziomego, w całym projekcie, zespoły zdecydowały się na zastosowanie odpowiedniego podejścia dla każdej subdomeny. Podział pionowy lepiej spełnia wymogi biznesowe dla uwierzytelniania i katalogu, a podział poziomy lepiej nadaje się dla subdomeny zarządzania kontem, która musi się składać z wielu subdomen.

Kompozycja po stronie klienta

Kompozycja po stronie klienta wspiera wymagania wewnętrznej strony sklepu internetowego i mieści się w zakresie umiejętności zespołu. Ten typ kompozycji umożliwi także rozwijanie aplikacji w przyszłości pod kątem innych platform, między innymi aplikacji desktopowych, a nawet progresywnych aplikacji webowych.

Trasowanie po stronie klienta

Po podjęciu decyzji o przyjęciu kompozycji po stronie klienta wybór trasowania jest prosty, gdyż trasowanie również musi mieć miejsce po stronie klienta. Musimy także wziąć pod uwagę, że będą występować dwa rodzaje trasowania: trasowanie globalne, obsługiwane przez kontener mikrofrontendu (nazywany również powłoką aplikacji), który będzie odpowiedzialny za przesyłanie informacji między mikrofrontendami, oraz trasowanie lokalne wewnątrz subdomen katalogu, w których zespół Maki stworzy mikrofrontendy z wieloma widokami.

Komunikacja między mikrofrontendami w oparciu o kluczowe decyzje związane z architekturą mikrofrontendową

Zgodnie z podjętymi podstawowymi decyzjami zespół będzie korzystał z magazynu sieciowego w celu dzielenia się sieciowym tokenem JSON (JWT), potrzebnym do korzystania z API uwierzytelniania. Ponieważ widok zarządzania kontem będzie miał dwa mikrofrontendy, a w przyszłości może mieć ich nawet więcej, chcemy utrzymać niezależność zespołów i artefaktów. W rezultacie użyjemy emitera zdarzeń do komunikacji między mikrofrontendami obecnymi w tym samym widoku, z góry definiując zdarzenia wywoływane przez każdy mikrofrontend oraz związane z nimi obciążenie.

Będziemy używać bundlera webpack z wtyczką Module Federation. Po stworzeniu kilku próbnych wersji zespół przekonał się, że wtyczka Module Federation zapewni wszystko, co niezbędne, by z powodzeniem ukończyć ten projekt. Główne powody używania Module Federation zamiast innych rozwiązań to:

Doświadczenie w pracy z bundlerem webpack

Bundler webpack jest powszechnie używany w organizacji. Wielu programistów miało już okazję z nim pracować przy okazji innych projektów, więc nie muszą się uczyć nowego narzędzia. Module Federation świetnie wpisuje się w ich zestaw umiejętności, zważywszy na fakt, że jest to tylko wtyczka do dobrze znanego w organizacji narzędzia.

Kompozycja po stronie klienta

W przypadku mikrofrontendu składanego po stronie klienta wtyczka Module Federation zapewnia łatwy sposób asynchronicznego ładowania paczek JavaScript. Ta wtyczka powstała z myślą o takim

właśnie zastosowaniu, a następnie została rozszerzona o renderowanie po stronie klienta. Jeśli więc w przyszłości wymagania się zmienią, zespoły będą mogły zmienić implementację mikrofrontendów, zostając przy Module Federation jako stabilnym zasobie do rozwijania swojej platformy.

Niezakłócona praca programistów

Zespoły mają duże doświadczenie w pracy z bundlerem webpack. Ponadto implementacja w procesie automatyzacji i lokalne narzędzie programistyczne pozostają takie same, więc zespoły mogą od razu pracować z pełną wydajnością.

Wtyczka Module Federation została wybrana do tego projektu z określonych powodów. Dla innych projektów mogłaby się okazać równie dobrym rozwiązaniem, ale niekoniecznie. Gdy projektujemy architekturę, musimy przemyśleć wszystkie za i przeciw, a nie wybierać technologię na ślepo. Przeanalizuj struktury swoich zespołów, umiejętności programistów, technologie używane w innych projektach prowadzonych w firmie oraz cele biznesowe projektu, zanim wybierzesz odpowiednią architekturę mikrofrontendową. Po przeanalizowaniu kontekstu, w którym pracujesz, podejmij kluczowe decyzje, by stworzyć solidną podstawę dla następnych wyborów, których będziesz musiał dokonać podczas pracy nad projektem.

Module Federation — podstawy

Zanim przejdziemy do implementacji technicznej, musimy zrozumieć kilka podstawowych koncepcji, aby docenić tok myślowy stojący za pewnymi wyborami technicznymi. Wtyczka Module Federation umożliwia aplikacji JavaScript wykonywanie kodu dynamicznie z innej paczki lub budowanie zarówno po stronie klienta, jak i serwera. Ta wtyczka jest dostępna dla bundlera webpack 5 i w pewnym stopniu, z ograniczonymi funkcjonalnościami, dla webpack 4 i Rollup. Module Federation zapewnia dwa elementy, które musimy dobrze poznać:

Host

Kontener, który wgrywa współdzielone biblioteki, mikrofrontendy i komponenty w czasie działania programu.

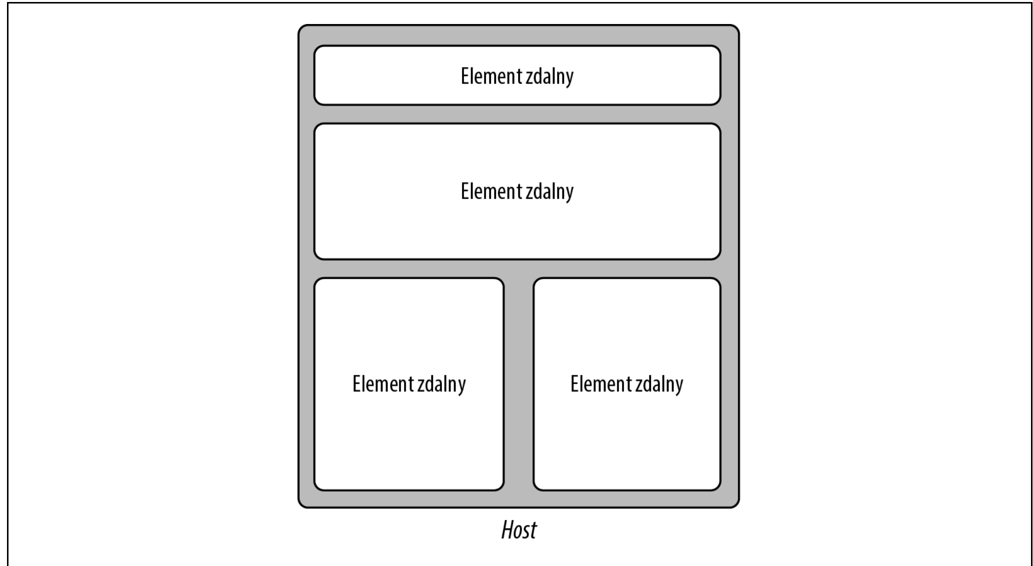
Element zdalny

Paczka JavaScript (ang. *bundle*), którą chcemy wgrać wewnątrz hosta.

Jak widać na rysunku 5.2, host może wgrać wiele elementów zdalnych. W naszym przypadku host to powłoka aplikacji, a element zdalny to mikrofrontend.

Z Module Federation dzielenie się może być dwukierunkowe. Element zdalny może dzielić się częścią lub całą paczką z hostem i na odwrót. Jednak dwukierunkowe dzielenie się może bardzo szybko skomplikować Twoją architekturę. Najlepiej przyjąć dzielenie się jednokierunkowe, by host nie dzielił się niczym z elementami zdalnymi. To sprawia, że znajdowanie i rozwiązywanie problemów jest dużo łatwiejsze, a ryzyko przecieków z hosta do domen będzie mniejsze. To ostatnie zjawisko mogłoby doprowadzić do powiązania między hostami a elementami zdalnymi.

Kup ksi k Pole ksi k



Rysunek 5.2. Wtyczka Module Federation składa się z dwóch kluczowych elementów: hosta, który ma za zadanie wgrywać paczki JavaScript w czasie działania aplikacji, oraz elementu zdalnego, który jest odpowiedzialny za paczki JavaScript, takie jak współdzielone biblioteki, mikrofrontendy, a nawet komponenty

W tle wtyczka Module Federation zarządza dwoma innymi wtyczkami bundlera webpack: Container-Plugin oraz ContainerReferencePlugin. Pierwsza jest odpowiedzialna za tworzenie kontenera w celu asynchronicznego wgrywania i synchronicznego wykonywania modułu. Natomiast druga wtyczka ma za zadanie nadpisywać kontener stworzony jako element zastępczy zdalnym modułem i sprawić, by kod wykonywał się w taki sposób, jakby znajdował się w pierwotnej paczce.

Wykorzystanie w pełni tej architektury umożliwia nam nie tylko określenie elementów zdalnych w bundlerze webpack, ale również wgranie ich za pomocą kodu JavaScript. Na przykład możemy pobrać ścieżki z API i wygenerować dynamiczny widok elementów zdalnych w oparciu o kraj użytkownika.

Ponieważ Module Federation to wtyczka bundlera webapck, możemy korzystać z innych jego funkcjonalności, by jak najlepiej zoptymalizować kod pod kątem naszego projektu. Na przykład Module Federation domyślnie tworzy wiele plików z fragmentami kodu JavaScript, ale może będziemy woleli mniej rozczłonkowaną implementację dla naszego elementu zdalnego i będziemy chcieli wgrywać tylko dwa lub trzy pliki. W takim przypadku moglibyśmy użyć wtyczki MinChunkSizePlugin (*https://webpack.js.org/plugins/min-chunk-size-plugin/*, strona w języku angielskim), która zmusza webpack do podzielenia kodu na pliki o ustalonej minimalnej liczbie kilobajtów. Moglibyśmy również użyć wtyczki DefinePlugin (*https://webpack.js.org/plugins/define-plugin/*, strona w języku angielskim), by podczas kompilacji zastąpić zmienne w kodzie innymi wartościami lub wyrażeniami. Dzięki tej wtyczce możemy bardzo łatwo stworzyć logikę, by dostarczyć odpowiednią ścieżkę bazową, gdy testujemy nasz kod lokalnie lub gdy działa w naszych środowiskach programistycznych. W połączeniu z innymi dostępnymi w ekosystemie webpack wtyczkami Module Federation może mieć potężne możliwości i być wygodnym sposobem na dopracowywanie efektu końcowego.

Kup ksi k Pole ksi k

Społeczność frontendowa zaczęła wykorzystywać Module Federation w swoich projektach, powstają więc nowe narzędzia w celu usprawnienia pracy programistów, w tym zestawienia danych pomagające zrozumieć związki między współdzielonymi zależnościami elementów zdalnych, takimi jak wtyczka Atriom (*https://www.npmjs.com/package/atriom-plugin*, strona w języku angielskim) czy wtyczka Live Reload (*https://www.npmjs.com/package/@module-federation/fmr*, strona w języku angielskim) w połączeniu z Module Federation.

Powinniśmy znać Module Federation na tyle dobrze, by móc zagłębić się w szczegóły implementacji. Projekt, nad którym tutaj pracujemy, ma dużo wspólnych konfiguracji dostępnych dla wtyczki Module Federation w projekcie mikrofrontendowym. Więcej informacji znajdziesz w oficjalnej dokumentacji (*https://webpack.js.org/concepts/module-federation/*, strona w języku angielskim).

Implementacja techniczna

Po analizie kontekstu, w którym będziemy budować naszą aplikację, po podjęciu kluczowych decyzji i wybraniu strategii technicznej nadszedł czas na przyjrzenie się szczegółom implementacji. Repozytorium sklepu internetowego z drobiazgami jest dostępne na GitHubie (*https://github.com/aws-samples/talk-dev-to-me-twitch/tree/main/micro-frontends-module-federation*), więc możesz zobaczyć cały projekt lub go sklonować, by się nim pobawić. Specjalnie stworzyłem przykład bez interakcji z serwerem, byś mógł go uruchomić lokalnie bez żadnych zewnętrznych zależności.

W ramach przypomnienia: aplikacja składa się z powłoki aplikacji dostępnej przez całą sesję użytkownika. Powłoka aplikacji wgrywa różne mikrofrontendy, między innymi uwierzytelnianie, katalog, zarządzenie kontem. Zespół wybrał jako narzędzie platformę React z bundlerem wepack i wtyczką Module Federation, co pozwala wszystkim zespołom na niezależną pracę. Za pomocą Module Federation zespoły mogą się dzielić zależnościami i wgrywać je tylko raz w sesji użytkownika. W ten sposób użytkownik ma zapewnione wygodne korzystanie z aplikacji, a programista może pracować bez zakłóceń. Skupmy się na głównych częściach aplikacji.

Struktura projektu

Zanim przejdziemy do studium przypadku, chciałbym powiedzieć kilka słów na temat struktury, którą stworzyłem na potrzeby tego projektu. Gdy sklonujesz repozytorium, zobaczysz kilka folderów, tak jak to pokazano na rysunku 5.3.

Każdy folder reprezentuje niezależny projekt. W tym samym repozytorium mamy do czynienia z podejściem *monorepo*, ale można by to łatwo przekształcić na podejście z wieloma repozytoriami (*polyrepo*). Więcej informacji na temat tych dwóch podejść do kontroli wersji znajdziesz w rozdziale 6.

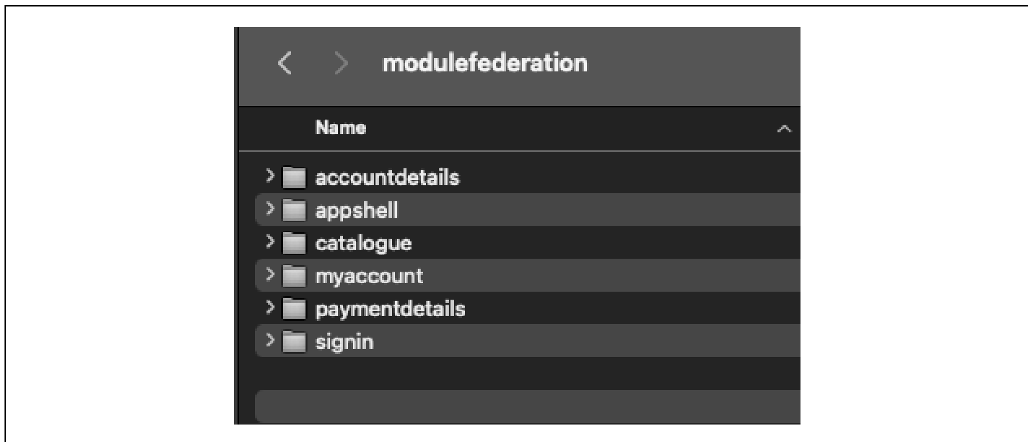
Wszystkie foldery mają taką samą strukturę jak na rysunku 5.4.

Każdy folder mikrofrontendu zawiera:

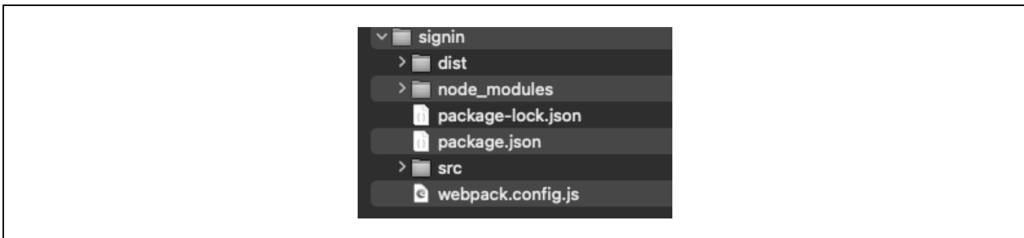
folder dist

Po zbudowaniu mikrofrontendu możemy tutaj znaleźć wszystkie zoptymalizowane pliki do wdrożenia w danym środowisku.

Kup ksi k Pole ksi k



Rysunek 5.3. Struktura folderów projektu wewnętrznej aplikacji sklepu z drobiazgami



Rysunek 5.4. Struktura mikrofrontendu

folder node_modules

Ten folder zawiera wszystkie zależności mikrofrontendów.

package.json

Ten plik zawiera wszystkie metadane dotyczące projektu oraz definicje funkcjonalnych atrybutów projektu, używanych przez menedżer plików npm do instalowania zależności, uruchamiania skryptów i znajdowania punktów wejścia do naszej paczki.

folder src

Ten folder zawiera całą logikę biznesową mikrofrontendu.

webpack.config.js

Ten plik to konfiguracja bundlera webpack, z której pobierane są ustawienia wtyczki Module Federation obsługującej nasze mikrofrontendy.

To typowa struktura folderu projektu. Module Federation nie wymaga zmiany dotychczasowego sposobu pracy. Ta wtyczka zarządza w tle wszystkimi zależnościami oraz logiką wgrywania i usuwania zależności bez konieczności zmiany Twojej ulubionej struktury folderów.

Kup ksi k

Pole ksi k

Powłoka aplikacji

Jak już zostało opisane wcześniej, powłoka aplikacji jest dostępna przez całą sesję użytkownika. Zważywszy na fakt, że jest niezbędna do zarządzania mikrofrontendami i nie wpasowuje się w żadną subdomenę biznesową, zespoły zdecydowały się przypisać implementację powłoki nowemu zespołowi o nazwie Sasazushi, złożonemu z głównych programistów. Ponieważ budowa tej części systemu nie będzie wymagać dużo pracy i dzięki temu, że powłoka aplikacji nie jest odpowiedzialna za logikę biznesową, wysiłek włożony w jej utrzymanie powinien być minimalny, główni programiści będą pracować zarówno w tym zespole, jak i w swoich macierzystych zespołach.

Zespół Sasazushi jest odpowiedzialny za:

- zapobieganie przeciekom domen do powłoki aplikacji,
- implementację globalnego trasowania między mikrofrontendami,
- upewnianie się, że mikrofrontendy są poprawnie wgrywane i usuwane z widoku,
- generowanie wspólnych dla wielu domen zależności w jednym lub wielu plikach JavaScript.

Co więcej, ponieważ zespół składa się z głównych programistów, to będzie odpowiedzialny za ogólne działanie systemu przez ustawienie regularnych spotkań z zespołami w celu dzielenia się najlepszymi metodami optymalizacji lub w celu omówienia określonych wąskich gardeł, które zostały zidentyfikowane podczas oceny wydajności.

Przeanalizujmy konfigurację bundlera webpack. Jako że Module Federation jest wtyczką, musimy jedynie ją zaimportować, jak każdą inną bibliotekę JavaScript.

```
const { ModuleFederationPlugin } = require("webpack").container;
```

Najbardziej podstawowa konfiguracja składa się z pliku wejściowego, folderu wyjściowego i trybu, w jakim chcemy, aby nasz kod się transpilował:

```
module.exports = {
  entry: "./src/index",
  mode: "development",
  output: {
   publicPath: "auto",
  },
// dodatkowa konfiguracja
}
```

W kolejnym kroku zazwyczaj dodajemy reguły do obsługi określonych funkcjonalności języka. Na przykład zanim ES6 stał się dostępny w przeglądarkach, musieliśmy transpilować określone funkcjonalności, takie jak generatory (*https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Generator*, strona w języku angielskim), na kod zrozumiały dla przeglądarki. W naszym przypadku używamy wtyczki Babel z Reactem ze skonfigurowaną obsługą JSX.

```
// ...

module: {
   rules: [
    {
     test: /\.jsx?$/,
     loader: "babel-loader",
```


Kup ksi k   Pole ksi k

```
        exclude: /node_modules/,
        options: {
         presets: ["@babel/preset-react"],
        },
       },
      ],
    }
    // ...
```

Jednak najważniejszą częścią naszej architektury mikrofrontendowej jest konfiguracja wtyczki Module Federation, tak by wgrywała zdalne mikrofrontendy. Z uwagi na fakt, że powłoka aplikacji jest kontenerem dla naszych mikrofrontendów, w nomenklaturze Module Federation jest hostem. Oznacza to, że będziemy musieli skonfigurować wszystkie zdalne mikrofrontendy, które chcemy leniwie wgrywać w powłoce aplikacji. W naszym przykładzie konfiguracja wtyczki wygląda następująco:

```
plugins: [
   new ModuleFederationPlugin({
    name: "AppShell",
    remotes: {
     MyAccount: "MyAccount@http://localhost:3004/remoteEntry.js",
     Catalog: "Catalog@http://localhost:3002/remoteEntry.js",
     SignIn: "SignIn@http://localhost:3003/remoteEntry.js"
     // Możesz również ustawić zdalne wejście z adresu URL serwera sieciowego
     //SignIn:"SignIn@http://www.mysite.com/signin/remoteEntry.js"
    },
    shared: {
      react: {
       singleton: true,
      },
      "react-dom": {
       singleton: true,
      },
      "react-router-dom": {
       singleton: true
      },
      "@material-ui/core": {
       singleton: true
      },
      "@material-ui/icons": {
       singleton: true
      }
   }
  })
 ],
}
```

Po zdefiniowaniu nazwy w hoście, w tym przypadku `AppShell`, podajemy wszystkie elementy zdalne, które chcemy wgrać w naszej aplikacji wewnątrz *zdalnego* obiektu. Każdy element zdalny składa się z ID i adresu URL, z którego zostanie pobrany plik JavaScript zawierający mapę wszystkich fragmentów kodu JavaScript wygenerowanych dla tego mikrofrontendu. Te dwie wartości są oddzielone znakiem @. Patrząc na konfigurację, możemy zauważyć, że mikrofrontend katalogu ma ID *Catalog*, a lokalny adres URL to: *http://localhost:3002/remoteEntry.js*. To oznacza, że kiedy powłoka aplikacji wgra mikrofrontend katalogu, wtyczka Module Federation pobierze plik *remoteEntry.js*, by dowiedzieć się, które fragmenty kodu JavaScript musi wgrać i które zależności są wspólne, a następnie

wgra nasz kod ze zdalnego serwera. W tym przypadku serwer działa w środowisku programistycznym, ale to może być równie dobrze zdalny adres URL, co widać w linijce kodu ze znacznikiem komentarza na początku.

Następna część konfiguracji wtyczki określa, które biblioteki są wspólne dla mikrofrontendów. Gdy pracujemy z innymi mikrofrontendowymi platformami, musimy się zastanowić, w jaki sposób dzielić się wspólnymi bibliotekami.

Często programiści tworzą niezależne repozytorium, gdzie zapisują wspólne biblioteki i zależności. Następnie wprowadzają automatyzację w celu budowy i wdrażania wspólnego kodu oraz wyznaczają osoby odpowiedzialne za aktualizację, rozmiary paczek, wycofanie zmian, wdrażanie i tak dalej. Jednak w przypadku Module Federation musimy jedynie określić zależności, którymi chcemy się dzielić, w każdym elemencie zdalnym i hoście. W naszym przykładzie są to wszystkie mikrofrontendy i powłoka aplikacji. Bundler webpack i Module Federation stworzą wiele plików JavaScript i pobiorą je tylko raz — podczas sesji użytkownika dla wszystkich mikrofrontendów.

To może się wydawać oczywiste, ale uwierz mi, że nie zawsze tak jest. Prostota optymalizacji naszych mikrofrontendów, którą oferuje Module Federation, jest naprawdę fenomenalna.

Określanie wspólnych bibliotek w Module Federation jest tak łatwe jak w kodzie pokazanym poniżej:

```
//...

shared: {
     react: {
       singleton: true,
     },
     "react-dom": {
       singleton: true,
      },
      "react-router-dom": {
        singleton: true
      },
      "@material-ui/core": {
        singleton: true
      },
      "@material-ui/icons": {
        singleton: true
      }
}

//...
```

W obiekcie `shared` wymieniamy wszystkie biblioteki, które mają być wspólne. Możemy również użyć zaawansowanych API wtyczki Module Federation, by wykonać następujące rzeczy:

- wgrywanie biblioteki tylko raz za pomocą właściwości `singleton`, tak jak to widać w naszym przykładzie;
- wgrywanie wszystkich wspólnych bibliotek przed pobraniem kodu aplikacji za pomocą właściwości `eager`;
- określenie wersji bibliotek, które mają być wgrane, za pomocą właściwości `requiredVersion`;

Kup ksi k Pole ksi k

- konfiguracja zmiennej i zakresu, w jakim instancja wspólnej biblioteki powinna być stworzona, za pomocą właściwości `shareKey` dla zmiennej i `shareScope` dla zakresu.

To tylko niektóre konfiguracje, które możemy zastosować dla wspólnych modułów. Biorąc pod uwagę, że API cały czas się rozwija, nie byłbym zaskoczony, gdyby w przyszłości pojawiło się jeszcze więcej możliwości.

Wersje współdzielonych bibliotek

Wtyczka Module Federation domyślnie wgrywa najnowszą wersję biblioteki podaną w konfiguracji. Jeśli mamy zatem powłokę aplikacji wgrywającą bibliotekę React 18 i mikrofrontend, który używa React 17, Module Federation wgra React 18, jeżeli za pomocą właściwości `requiredVersion` (*https://webpack.js.org/concepts/module-federation/#high-level-concepts*, strona w języku angielskim) nie uściślimy wersji, której chcemy użyć. Jeśli nie będziemy na to uważać, mogą wystąpić błędy na produkcji. Za pomocą właściwości `shareKey` i `shareScope` możemy zarządzać różnymi wersjami tej samej biblioteki w tej samej aplikacji przez określenie różnych kontenerów dla naszych zależności.

Module Federation umożliwia nam wgrywanie zależności albo synchronicznie, z zastosowaniem właściwości `eager` w konfiguracji wtyczki, albo asynchronicznie. Zalecane jest wgrywanie asynchroniczne, tak by użytkownik nie musiał wgrywać od razu wszystkich zależności w dużej paczce. Dodatkowo nie pogorszą się takie wskaźniki jak TTFB (ang. *time to first byte* — czas do pierwszego bajta) czy TTI (ang. *time to interactive* — czas, by aplikacja stała się w pełni interaktywna). Aby wgrywać nasze zależności asynchronicznie, musimy podzielić inicjalizację naszej aplikacji na kilka plików. Podzielimy powłokę aplikacji na trzy główne pliki: *index.js*, *bootstrap.js* i *app.js*. Plik *index.js* będzie punktem wejścia naszej aplikacji i wymaga tylko jednej linii kodu:

```
import("./bootstrap");
```

Plik *bootstrap.js* będzie odpowiedzialny za stworzenie instancji powłoki aplikacji i dołączenie aplikacji React do elementu `div`, który znajduje się w szablonie HTML o nazwie *root*.

```
import React from "react";
import ReactDOM from "react-dom";
import App from "./App";
ReactDOM.render(<App />, document.getElementById("root"));
```

Wreszcie, plik *app.js* będzie zawierał dwa ważne elementy. Pierwszy, komponent `Main`, będzie implementował podstawowy interfejs użytkownika z nagłówkiem i bocznym menu. Drugi element, React Router, to jedno z najpopularniejszych rozwiązań trasowania dla aplikacji React do globalnego trasowania.

```
import React from "react";
import {BrowserRouter as Router} from "react-router-dom";
import Main from "./Main";

const App = () => {
  return(
    <Router>
      <Main></Main>
```


Kup ksi k Pole ksi k

```
      </Router>
    )
  }
  export default App;
```

Pamiętaj, że na potrzeby strategii globalnego trasowania możesz użyć innych bibliotek dostępnych dla React, a nawet stworzyć własną, choć wynajdowanie koła na nowo nie jest zbyt produktywne. Ważne jest, aby pamiętać, że nie musimy się ograniczać do określonych bibliotek trasowania w swojej architekturze mikrofrontendowej.

W kodzie komponentu `Main` jest kilka kluczowych koncepcji do odkrycia. Zacznijmy od globalnego trasowania. Przy zastosowaniu powłoki aplikacji trasowanie odbywa się po stronie klienta i to powłoka jest odpowiedzialna za prowadzenie użytkownika od jednego mikrofrontendu do drugiego. Zazwyczaj mamy różne poziomy w adresie URL, na przykład *https://www.mysite.com*. Główny adres (ang. *root*) ma zawsze poziom 0. Gdy użytkownik klika link do innej strony, przechodzimy do poziomu 1., *https://www.mysite.com/catalog*. Każdy dodatkowy link kliknięty przez użytkownika doda kolejne poziomy do adresu URL.

Powłoka aplikacji jest odpowiedzialna tylko za logikę strony głównej i poziomu 1. Zazwyczaj, gdy użytkownik klika link na stronie głównej, przechodzimy do nowego obszaru strony, który odpowiada nowej subdomenie biznesowej, więc wgrywa się mikrofrontend. Strona sklepu, którą właśnie omawiamy, ma następującą strukturę:

```
<main className={clsx(classes.content, {
           [classes.contentShift]: open,
      })}>
      <div className={classes.drawerHeader} />
      <Switch>
          <Route path="/myaccount" render={_ => renderMFE(MyAccount)}/>
          <Route path="/shop" render={_ => renderMFE(Catalog)}/>
          <Route path="" render={_ => renderMFE(SignIn)}/>
      </Switch>
</main>
```

Jak widzimy, obiekt `Route` składa się ze ścieżki (pierwszy poziom) i odpowiadającego jej mikrofrontendu, który ma być wgrany. Wersja numer 16 biblioteki React używała eksperymentalnego API o nazwie Suspense (*https://reactjs.org/docs/concurrent-mode-suspense.html*, strona w języku angielskim), by zawiadamiać bibliotekę, że należy wgrać pewien kod, a w międzyczasie dostarczała komponent, który miał być wyrenderowany jako symbol zastępczy.

```
<Suspense fallback={<Spinner />}>
 <CatalogMFE />
</Suspense>
```

Z komponentem Suspense i możliwością leniwego wgrywania komponentu możemy użyć następującej składni, by wgrać zdalne mikrofrontendy, a Module Federation pobierze dany moduł i udostępni go powłoce aplikacji. Tak naprawdę funkcja `renderMFE` używana w obiekcie `Route` w pełni wykorzystuje tę technikę.

```
const Catalog = React.lazy(() => import("Catalog/Catalog"));
const SignIn = React.lazy(() => import("SignIn/SignIn"));
const MyAccount = React.lazy(() => import("MyAccount/MyAccount"));
 const renderMFE = (MFE) => {
   return(
```


Kup ksi k Pole ksi k

```
      <React.Suspense fallback="Wgrywanie...">
        <MFE />
      </React.Suspense>
    )
}
```

Gdy na stronie domowej użytkownik wybiera nowy obszar sklepu, router wgrywa nowy mikrofrontend w podobny sposób, jakby leniwie wgrywał standardowy komponent biblioteki React w aplikacji jednostronicowej (SPA). Funkcja importu zawiera identyfikator podany w konfiguracji Module Federation, znajdującej się w pliku *webpack.config.js*, dzięki czemu wiadomo, który mikrofrontend ma zostać wgrany. Module Federation zaimportuje zdalny moduł za Ciebie.

Ostatnią rzeczą, o jakiej warto wspomnieć przy okazji omawiania powłoki aplikacji, jest implementacja systemu projektowania. We wszystkich naszych mikrofrontendach jest używana popularna biblioteka systemu projektowania, Material-UI (*https://mui.com/*, strona w języku angielskim). Nie musimy w pełni rozumieć API tej biblioteki, ale musimy się upewnić, że nasze style nie będą z niczym kolidować po wgraniu mikrofrontendu do powłoki aplikacji. W tym celu użyjemy przedrostków, co zostało opisane w rozdziale 4. W bibliotece Material-UI przedrostek dla każdego mikrofrontendu oraz powłoki aplikacji dodajemy przy użyciu właściwości `seed`:

```
const generateClassName = createGenerateClassName({
 seed:'appshell'
});
```

Przez określenie właściwości `seed` do każdej nazwy klasy CSS dodamy przedrostek podany jako wartość właściwości. W naszym przykładzie wszystkie style w powłoce aplikacji będą miały dodany przedrostek *appshell*. I tak nagłówek o rozmiarze 6 (element `h6`) będzie miał nazwę na wzór:

```
appshell-MuiTypography-h6
```

W ten sposób możemy się upewnić, że każdy zespół może pracować nad własnym mikrofrontendem bez ryzyka konfliktu z innymi mikrofrontendami istniejącymi w tym samym lub różnych widokach. Inne dostępne biblioteki systemu projektowania również stosują to podejście, ale ten system zapewnia bezpieczeństwo równoległej pracy bez deptania sobie po piętach.

Mikrofrontend uwierzytelniania

Ponieważ mikrofrontend uwierzytelniania nie wymaga zbyt wiele pracy po stronie frontendu, zespół Sashimi składa się głównie z programistów backendowych. Tak naprawdę uwierzytelnienie musi być zintegrowane z systemem pojedynczego logowania (ang. *single sign-on*, SSO), używanym w organizacji do logowania się do wewnętrznych aplikacji. Scentralizowany system SSO jest typowym podejściem w dużych przedsiębiorstwach, ale zaleca się go również w mniejszych organizacjach, gdyż zapewnia większą kontrolę nad dostępami do podstawowych systemów firmy.

Mikrofrontend uwierzytelniania jest pierwszym elementem zdalnym wtyczki Module Federation, który opisujemy, więc zaczniemy od konfiguracji. Konfiguracja elementu zdalnego niewiele różni się od tej dla hosta, ale jest kilka dodatkowych pól, co możesz zobaczyć w poniżej przytoczonym kodzie:

```
// kod przed

{
   name: "SignIn",
```


Kup ksi k   Pole ksi k

```
    filename: "remoteEntry.js",
    exposes:{
     "./SignIn": "./src/SignIn"
    },
    shared: {
     // wszystkie wspólne zależności dla tego mikrofrontendu
    },
}

// kod po
```

Pole `filename` (nazwa pliku) służy do określenia punktu wejścia elementu zdalnego. W podanym w tym polu pliku umieściliśmy mapę wszystkich generowanych przez webpack i wgrywanych przez Module Federation fragmentów kodu oraz określiliśmy, że mikrofrontend powinien być renderowany w powłoce aplikacji. W polu `exposes` (eksponowanie) wymieniamy wszystkie moduły, które chcemy eksponować dla hosta w celu zapewnienia integracji wewnątrz aplikacji. W przypadku mikrofrontendów to pole prawdopodobnie będzie miało tylko jeden wpis, ponieważ każdy mikrofrontend jest reprezentowany przez jeden artefakt. Jednak gdy eksponujemy bibliotekę dostępną dla wszystkich, taką jak system projektowania, możemy wylistować wszystkie dostępne punkty wejścia tak, by każdy host korzystający z tej biblioteki mógł wybrać tylko to, czego potrzebuje, i nic więcej. Pole `shared` rządzi się taką samą logiką jak ta opisana przy okazji konfiguracji powłoki aplikacji.

Nie będziemy omawiać konfiguracji dla każdego mikrofrontendu. Są podobne do tej dla mikrofrontendu uwierzytelniania, tylko mają inne wartości w poszczególnych polach, co jest zależne od danej subdomeny reprezentowanej przez mikrofrontend. Jeśli chcesz przyjrzeć się bliżej konfiguracji mikrofrontendów, zajrzyj do repozytorium na GitHubie (*https://github.com/aws-samples/talk-dev-to-me-twitch/tree/main/micro-frontends-module-federation*), gdzie znajdziesz cały przykładowy projekt.

Tym, na czym warto się skupić przy okazji mikrofrontendu uwierzytelniania, jest sposób, w jaki token JWT jest współdzielony między mikrofrontendami. Dla mikrofrontendów, które korzystają z prywatnego API, potrzebujemy mechanizmu szybkiego uzyskiwania kodu dostępu, gdyż to za jego pomocą pobierzemy dane potrzebne do przygotowania interfejsu użytkownika.

Częstą praktyką jest dzielenie się kodem poprzez magazyn sieciowy lub plik cookie, co również zostało opisane przy okazji omawiania podstawowych decyzji, które należy podjąć, projektując architekturę mikrofrontendową. W naszym przykładzie mikrofrontendu uwierzytelniania zastosujemy dokładnie to samo podejście. Tak naprawdę po wywołaniu API logowania otrzymamy token JWT zapisany w sesji:

```
// kod przed

const SignIn = () => {
  let history = useHistory();

  const onSignIn = () => {
    window.sessionStorage.setItem("token", token);
    history.push("/shop");
  }

// kod po

}
```


Kup książkę Poleć książkę

Z zasady wszystkie mikrofrontendy pobierają token zapisany w sesji i przy jego użyciu korzystają z prywatnego API dla swojej domeny. Chyba nie trzeba dodawać, że każdy mikrofrontend, który powinien zostać wyświetlony po uwierzytelnieniu użytkownika, ma za zadanie sprawdzić kod i upewnić się, że użytkownik ma dostęp do danej zawartości. W architekturze podziału poziomego, w której mamy kilka mikrofrontendów na tej samej stronie, kontener mikrofrontendu powinien potwierdzić dostęp użytkownika, a następnie albo wgrać mikrofrontendy określonej strony dla zalogowanych użytkowników, albo pokazać komunikat błędu. W architekturze podziału pionowego przed renderowaniem komponentu każdy mikrofrontend powinien ocenić, czy dany kod jest ważny i czy rola przypisana do użytkownika zezwala na dostęp do danej zawartości.

Mikrofrontend katalogu

Domena katalogu jest najprawdopodobniej najbardziej złożona i największa ze wszystkich mikrofrontendów. To z jej powodu użytkownicy odwiedzają naszą stronę, więc nie tylko musi być prosta, ale musi również zawierać wszystkie interesujące użytkownika informacje. Za ten mikrofrontend odpowiedzialny jest zespół Maki. Jego celem jest implementacja wielu widoków, by użytkownik mógł odkryć zawartość katalogu i dowiedzieć się czegoś więcej o wybranym produkcie. Możliwe, że w przyszłości ten zespół będzie musiał dodać nowe funkcjonalności, takie jak możliwości przesłania komuś zdjęcia produktu lub dodania swojej opinii.

To ten zespół zaimplementuje te funkcjonalności i przygotuje bazę kodu w sposób modułowy, tak by w przyszłości, gdy zajdzie taka potrzeba, można było łatwo przekazać tę część domeny innemu zespołowi. Silna enkapsulacja i modułowość pomogą zespołowi Maki łatwo rozdzielić części domeny i współpracować z innymi zespołami, by zapewnić wygodę użytkownika podczas korzystania z aplikacji.

Osobliwą cechą podziału pionowego jest to, że musimy obsługiwać wiele widoków w tym samym mikrofrontendzie. Taki rodzaj aplikacji jednostronicowej jest szczególny dla domeny katalogu. To nie powinno przeszkodzić w dodawaniu wspólnego lub właściwego dla danej domeny komponentu, takiego jak komponent spersonalizowanych produktów zaimplementowanych przez inne zespoły w tej domenie. Pomimo że powłoka aplikacji jest odpowiedzialna za globalne trasowanie, dla tego mikrofrontendu musimy zastosować lokalne trasowanie (czyli trasowanie zaimplementowane na poziomie mikrofrontendu), które współpracuje z globalnym. W naszym przykładzie lokalne trasowanie nie różni się zbyt wiele od globalnego, co widać poniżej.

// kod przed

```
const Catalog = () => {
  let { path } = useRouteMatch();

  return(
    <div>
      <h1>
        Shop
      </h1>
      <Switch>
        <Route exact path={`${path}`} component={Home}/>
        <Route exact path={`${path}/product/:productId`} component={Details}/>
      </Switch>
    </div>
  )
}
```

// kod po

Kup książkę Poleć książkę

Przy użyciu biblioteki React Router na początku uzyskujemy adres URL pierwszego poziomu. Potem, gdy użytkownik wybierze określony produkt, dołączamy numer identyfikacyjny produktu do bieżącego poziomu. Zaczynając od drugiego poziomu, struktura i zarządzenie pozostają zazwyczaj w gestii mikrofrontendu, więc domena może się rozwijać niezależnie, bez konieczności koordynowania wprowadzania nowych elementów z innymi zespołami. To również zapobiega powtarzaniu się adresów URL na pierwszym poziomie, ponieważ tylko jeden zespół jest odpowiedzialny za globalne trasowanie. Podczas implementacji strony ze szczegółami produktu w żądaniu wysłanym do API w celu uzyskania danych do wyświetlenia możemy użyć numeru identyfikacyjnego produktu.

```
// kod przed

const Details = () => {
  // Uzyskujemy numer identyfikacyjny z URL.
  const {productId} = useParams()
  // Do pobierania logiki produktu z API możemy dodać logikę.
  return(
    <div>
        // Wyświetlamy numer identyfikacyjny produktu.
      {`Details page, product id: ${productId}`}
      <Link to="/shop">Wszystkie produkty</Link>
    </div>
  )
}

// kod po
```

W ten sposób przygotujemy naszą bazę kodu na potencjalne podziały w przyszłości. Łańcuchy zapytań (ang. *query strings*) są dobrą metodą na przekazywanie ulotnych informacji z jednego mikrofrontendu do innego. Jako że ten typ danych jest wykorzystywany od razu przez inną część systemu i nie musi być długo przechowywany, przekazywanie tych informacji w obrębie systemu za pomocą łańcuchów zapytań jest rekomendowanym rozwiązaniem.

Mikrofrontend zarządzania kontem

Zespół Nigiri, odpowiedzialny za subdomenę płatności, oraz zespół Sashimi, odpowiedzialny za uwierzytelnienie, autoryzację i konto użytkownika, muszą wspólnie pracować nad widokiem do zarządzania kontem, który składa się z części z informacjami o płatności i części z danymi użytkownika. Ponieważ te domeny są przypisane do różnych zespołów, potrzebujemy innego podejścia niż w przypadku pozostałych mikrofrontendów. Zamiast podziału pionowego zastosujemy podział poziomy w celu przygotowania końcowego widoku i umożliwimy komunikację między tymi dwoma domenami, by można było wymieniać informacje dotyczące określonych działań użytkownika. W tym celu będziemy musieli stworzyć nowego hosta i dwa elementy zdalne. Wiemy, że każdy element zdalny jest przypisany do zespołu, ale co z nowym hostem? Zespoły Nigiri i Sashimi razem definiują strategię oceny tego wspólnego kontenera dla swoich subdomen. Trasowanie nowego hosta składa się z:

- wgrywania dwóch mikrofrontendów: danych użytkownika i danych płatności,
- sprawdzania, czy użytkownik jest zalogowany, czy nie, oraz wyświetlania komunikatu o błędzie lub przekierowywania użytkownika na stronę logowania.

Kup ksi k Pole ksi k

Z tego powodu zespół Nigiri decyduje się wziąć odpowiedzialność za nowy host i współpracować z zespołem Sashimi, by opracować mechanizmy zapewniające płynną pracę programistów, a publikacje nowego hosta nie będą powodować problemów w trakcie pracy zespołu Sashimi.

Możesz się zastanawiać, gdzie technicznie będzie pokazywany użytkownikowi nowy host. Wtyczka Module Federation pozwala na użycie wielu hostów i nie wymaga silnie zhierarchizowanej struktury. Tak naprawdę jest cienka granica między hostem i elementem zdalnym, ponieważ element zdalny może eksponować kilka bibliotek używanych przez host — i na odwrót. Jak już zostało wspomniane w rozdziale 4., musimy zwracać uwagę na tę cienką linię, gdyż jeśli zostanie przekroczona i zaczniemy się dwukierunkowo dzielić zależnościami, stworzymy kod, którym nie będzie się dało zarządzać i który w dłuższej perspektywie czasowej będzie przynosił więcej problemów niż korzyści. Radzę, by wprowadzić hierarchiczną relację między hostem i elementem zdalnym, gdy dzielenie odbywa się w jednym kierunku, dzięki czemu host nie będzie eksponował żadnego modułu swoim elementom zdalnym. Ta prosta, ale skuteczna zasada ułatwia implementację architektury mikrofrontendów, zmniejszając ryzyko wystąpienia potencjalnych błędów. Co więcej, znajdowanie i rozwiązywanie problemów w aplikacji jest łatwiejsze, a powiązania między modułami są luźniejsze, co pozwala uniknąć sytuacji, kiedy wiele modułów jest od siebie zależnych. Gdy w ten sposób zmniejszamy liczbę zależności i zewnętrznych bibliotek, każdy zespół będzie mógł podejmować dobre dla projektu decyzje, biorąc pod uwagę to, że czasami musimy pójść na kompromis, by osiągnąć cele biznesowe organizacji.

Z technicznego punktu widzenia należy wziąć pod uwagę kilka małych zmian, które musimy przeprowadzić na potrzeby podziału poziomego. Przede wszystkim potrzebujemy kontenera dla dwóch mikrofrontendów. Stworzymy host o nazwie *MyAccount* (moje konto), który będzie miał inną niż pozostałe konfigurację wtyczki Module Federation. Kontener musi być hostem, ponieważ dotyczy mikrofrontendów danych użytkownika i płatności, ale jednocześnie musi być elementem zdalnym dla powłoki aplikacji. W tym celu dodajemy obiekty `remotes` i `exposes`, jak widać w następującym fragmencie kodu:

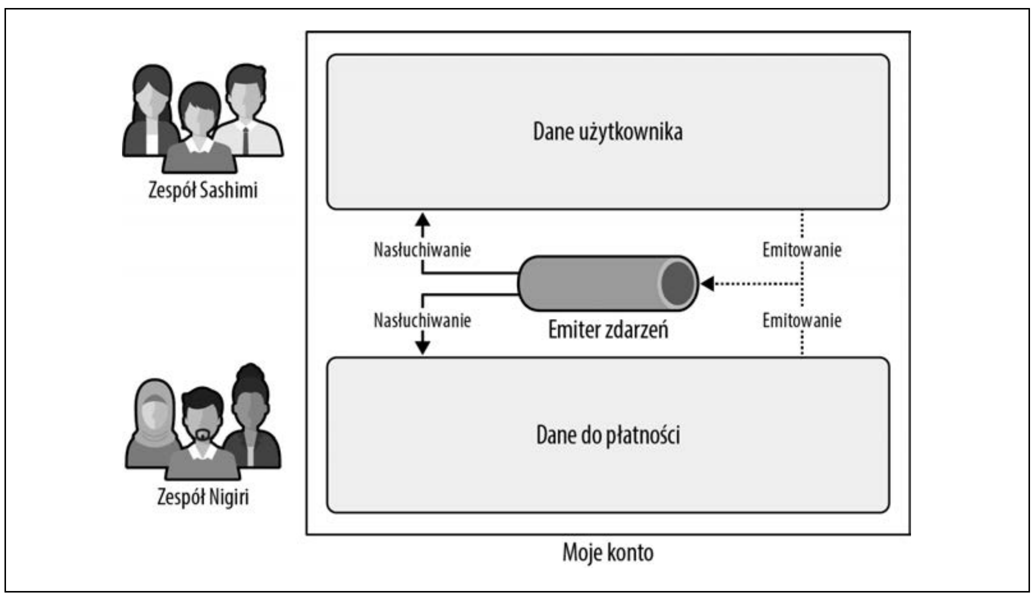
```
// kod przed

name: "MyAccount",
filename: "remoteEntry.js",
exposes:{
   "./MyAccount": "./src/MyAccount"
},
remotes: {
   AccountDetails: "AccountDetails@http://localhost:3005/remoteEntry.js",
   PaymentDetails: "PaymentDetails@http://localhost:3006/remoteEntry.js"
},

// kod po
```

Ta jedna zmiana pozwala, by host był również elementem zdalnym. Nie powinniśmy nadmiernie wykorzystywać tej techniki, ponieważ szybko możesz dojść do dużej liczby małych aplikacji, które reprezentują komponenty, a nie całe domeny biznesowe. Dlatego kluczowe jest, by poprawnie wyznaczyć granice. Jeśli nie jesteśmy pewni, czy dodać nowy mikrofrontend, czy wprowadzić nowe funkcjonalności w jednym z istniejących, musimy wrócić do tablicy z projektem i upewnić się, że nasza decyzja jest zgodna z zasadami mikrofrontendów (w razie potrzeby wróć do rozdziału 2.). Sprawdzona zasada, opisana w rozdziale 4., mówi, że należy zrozumieć, jak bardzo można rozbudować mikrofrontend,

Kup ksi k Pole ksi k

a gdy domena biznesowa przecieka do kontenera, musisz sprawdzić, czy implementujesz mikrofrontend, czy komponent.

W implementacji należy zwrócić uwagę na komunikację między mikrofrontendami. Podstawowe decyzje związane z mikrofrontendami dotyczą dzielenia się informacjami między mikrofrontendami, ale bez dzielenia się kodem między różnymi domenami. W celu utrzymania rozdziału między domenami idealnym rozwiązaniem jest wzorzec *pub/sub*. Zespoły zdecydowały się na wykorzystanie emitera zdarzeń, który rozdziela dwa mikrofrontendy wchodzące w skład widoku zarządzania kontem, ale pozwala się im komunikować ze sobą przez tworzenie i wysyłanie zdarzeń, co pokazano na rysunku 5.5.



Rysunek 5.5. Zespoły Sashimi i Nigiri implementują emiter zdarzeń dzielony między mikrofrontendami w celu komunikacji

W hoście tworzymy instancję emitera zdarzeń i umieszczamy ją w dwóch mikrofrontendach za pomocą właściwości:

```
// kod przed

const AuthenticatedView = (props) => {
  return(
    <React.Suspense fallback="Wgrywanie...">
      <AccountDetails emitter={props.emitter}/>
      <PaymentDetails emitter={props.emitter}/>
    </React.Suspense>
  )
}

//...
let view;
// Proste sprawdzanie kodu dostępu, ale tutaj musi być zastosowana lepsza strategia uwierzytelniania.
if(token){
  const emitter = createNanoEvents();
```


Kup ksi k Pole ksi k

```
    view = <AuthenticatedView emitter={emitter}/>
  } else {
    // Wyświetl inny widok, jeśli użytkownik się nie zalogował.
  }

  // kod po
```

Widok `AuthenticatedView` używa tego samego wzorca co powłoka aplikacji, w której leniwie wgrywamy nasze mikrofrontendy. Korzystamy z React Suspense, dopóki mikrofrontendy nie będą gotowe. Mikrofrontendy są wgrywane tylko wtedy, gdy w obiekcie `sessionStorage` dostępny jest kod dostępu. Aby uprościć ten przykład, nie pokazałem pełniej implementacji uwierzytelniania. W końcu we właściwości `emitter` przypisujemy emiter zdarzeń, który jest dostępny dla obu mikrofrontendów.

Wszystkie mikrofrontendy używają emitera zdarzeń, by powiadomić nas, że coś się zdarzyło w danej domenie biznesowej. W ten sposób mikrofrontend może się rozwijać i jednocześnie zachować niezbędną w całym systemie kompatybilność. Stosując to podejście, ważne jest, aby w dokumentacji umieścić informacje o tym, jakie zdarzenia emituje dany mikrofrontend, a jakich nasłuchuje. Gdy to zrobimy, dodanie nowego mikrofrontendu do widoku będzie niezwykle łatwe, ponieważ w dokumentacji znajdzie się nie tylko nazwa zdarzenia, ale również obciążenie z nim związane. W przypadku reaktywnych strumieni (ang. *reactive streams*) lub niestandardowych zdarzeń (ang. *custom events*) powinniśmy stosować podobne podejście. Implementacja techniczna staje się bardzo łatwa, ponieważ mikrofrontend płatności może emitować zdarzenia, gdy użytkownik zmieni metodę płatności.

```
const onPaymentChanged = () => {
    props.emitter.emit("paymentChanged", "Maj 2021");
}
```

W międzyczasie mikrofrontend obsługujący konto użytkownika nasłuchuje zdarzenia `paymentChanged`, by dokonać zmian w swoim kodzie.

```
const [lastPaymentDate, setPaymentChanged] = useState("Sty 2021")

props.emitter.on("paymentChanged", date => setPaymentChanged(date))

return (
    <div>
      <h3>Dane konta</h3>
      <ul>
        <li><i>imię:</i> Luca</li>
        <li><i>nazwisko:</i> Mezzalira</li>
        <li><i>e-mail:</i> guesswho@acme.com</li>
        <li><i>członek od:</i> Sty 2021</li>
        <li><i>zmiana sposobu płatności: </i>{lastPaymentDate}</li>
        <li><a href="#">Zmień dane konta</a></li>
      </ul>
    </div>
);
```

Gdy mikrofrontend odbiera zdarzanie, biblioteka React zmienia wartość zmiennej `lastPaymentDate`, która pokazuje, kiedy użytkownik po raz ostatni zmieniał metodę płatności. Radzę, by dla zdarzeń używać obiektów o określonym typie. To zmniejsza ryzyko popełnienia literówek w bazie kodu, co jest kluczowe, gdyż dzielimy się łańcuchami znaków z wieloma zespołami. Jeśli nie używasz języka TypeScript, podobny rezultat możesz osiągnąć przez stworzenie obiektu ze stałymi właściwościami, co

Kup ksi k Pole ksi k

pomoże programistom stosować poprawne zasady podczas opracowywania mechanizmu komunikacji zdarzeń w aplikacji.

```
const PaymentEvents = {
    PAYMENT_CHANGED: "paymentMethodChanged";
// inne wydarzenia
}
```

Następnie możesz zamrozić obiekt (*https://developer.mozilla.org/pl/docs/Web/JavaScript/Reference/Global_Objects/Object/freeze*), by zapobiec zmianom w jego budowie oraz dodawaniu i usuwaniu właściwości podczas działania programu.

Gdy w tym samym widoku mamy wiele mikrofrontendów, które się ze sobą komunikują przez zdarzenia, musimy się upewnić, że wszystkie zdarzenia zostaną odebrane przez zainteresowane mikrofrontendy. Na przykład wyobraź sobie, że w jednym widoku mamy trzy mikrofrontendy i dwa z nich wgrywają się natychmiast, a trzeci ładuje się wolniej z powodu problemów z siecią. Gdy pierwsze dwa emitują zdarzenia, trzeci, gdy się wgrywa, nie jest w stanie odebrać żadnych informacji. Jednym z możliwych rozwiązań jest stworzenie bufora zdarzeń w kontenerze aplikacji i ponowne odtworzenie zdarzeń, gdy trzeci mikrofrontend zostanie w pełni załadowany. Będzie się to wiązać z pewnym wysiłkiem, gdyż będzie trzeba wprowadzić monitorowanie stanu każdego mikrofrontendu, zebrać wszystkie zdarzenia w tablicy lub podobnej strukturze danych, a następnie odtworzyć je dla jednego lub większej liczby mikrofrontendów bez odtwarzania pełnej listy zdarzeń dla każdego mikrofrontendu (co mogłoby spowodować wewnętrzny konflikt stanów). Wiele mikrofrontendów w jednym widoku wymaga od nas dokładnego przemyślenia, jak obsługiwać niepowodzenia lub częściowe niepowodzenia. Jest to konieczne, gdyż jak mówi Werner Vogels, dyrektor techniczny firmy AWS, „wszystko się cały czas psuje".

Rozwój projektu

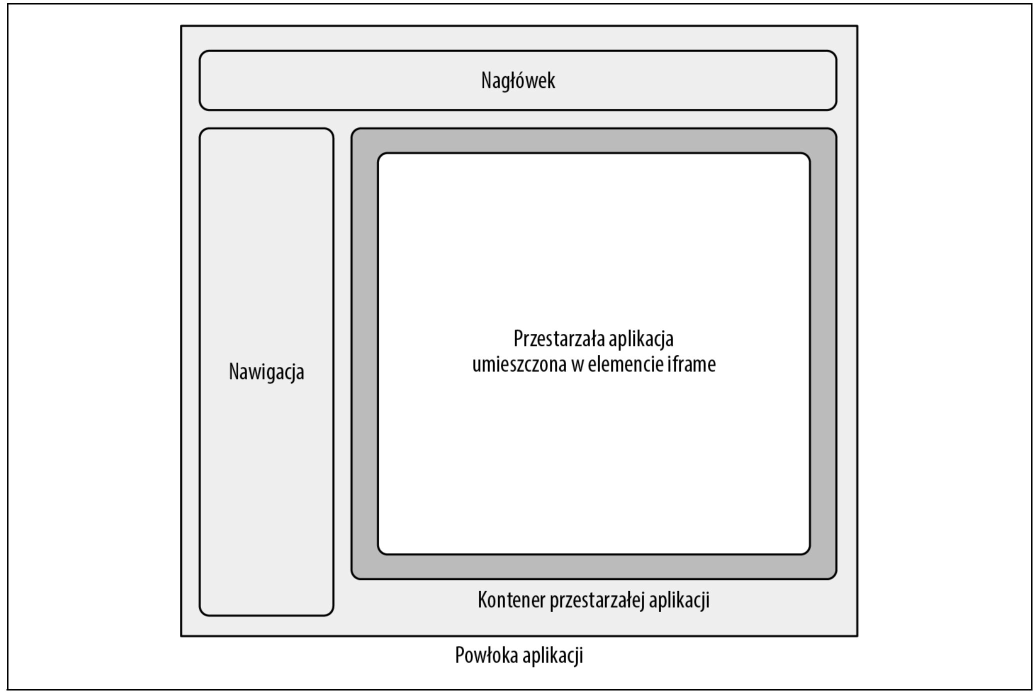
Nie chcemy tworzyć projektu, który będzie działał tylko przez krótki czas. Chcemy stworzyć taki, który może się rozwijać bez potrzeby zaczynania od początku. Zastanówmy się zatem, w jaki sposób nasz projekt mógłby się rozwijać i jak możemy stworzyć spójną implementację dla różnych domen.

Wbudowanie przestarzałej aplikacji

Wyobraź sobie, że do naszego sklepu musimy dodać narzędzie do personalizacji takich produktów jak koszulki, bluzy i kubki. Narzędzie to przestarzała aplikacja, stworzona kilka lat temu przy użyciu starej wersji Angulara. Tylko jedna osoba, która pracowała nad tym narzędziem, wciąż jest zatrudniona w firmie i współpracuje przy tym projekcie, naprawiając błędy i optymalizując bazę kodu tam, gdzie jest to możliwe. By szybciej dodać nową funkcjonalność, działy biznesowy i techniczny postanowiły zintegrować stare narzędzie z istniejącą architekturą mikrofrontendową i dostarczać takie rozwiązanie przez określony czas. Potem nowy zespół przejmie odpowiedzialność za projekt i przekształci go w rozwiązanie mikrofrontendowe. Aplikacja jest dobrze wyizolowana i nie potrzebuje żadnych szczególnych informacji na temat środowiska, w którym działa. Za pomocą łańcucha zapytań możemy przekazać do konfiguratora ustawienia niezbędne do renderowania pliku. Dodatkowo chcemy zmniejszyć ryzyko potencjalnych konfliktów z innymi częściami bazy kodu, między innymi z powłoką aplikacji.

Kup ksi k Pole ksi k

Możemy rozwiązać ten problem przez umieszczenie przestarzałej aplikacji w elemencie `iframe`, co zapobiegnie konfliktom z istniejącymi mikrofrontendami, tak jak to pokazano na rysunku 5.6.



Rysunek 5.6. Powłoka aplikacji wgrywa mikrofrontend, który działa jak pośrednik między nowym i starym światem. Przestarzała aplikacja jest opakowana elementem iframe w celu zminimalizowania oddziaływania na istniejącą bazę kodu mikrofrontendów

Jeśli jednak chcemy komunikować się ze starą aplikacją i na odwrót, na przykład wyświetlać błędy w całym interfejsie, nie tylko w elemencie `iframe`, musimy stworzyć most między starą aplikacją a powłoką aplikacji, aby móc wykorzystać system ostrzeżeń. Moglibyśmy bezpośrednio zintegrować przestarzałą aplikację z powłoką aplikacji, ale to oznaczałoby zanieczyszczenie bazy kodu powłoki. Istnieje lepsza strategia. Zamiast tego możemy zastosować adapter w formie mikrofrontendu jako kontener dla elementu `iframe`, w którym zawarta jest stara aplikacja. Ten mikrofrontend będzie odpowiedzialny za zarządzanie elementem `iframe` za pomocą łańcuchów zapytań oraz za przyjmowanie wszystkich wiadomości ze starej aplikacji i tłumaczenie ich na zdarzenia emitowane w magistrali zdarzeń.

Adapter (ang. *adapter pattern*) to wzorzec projektowy (znany również jako opakowanie, ang. *wrapper*), dzięki któremu możemy użyć istniejącej klasy interfejsu jako innego interfejsu. Jest to często wykorzystywane, aby istniejące klasy współpracowały z innymi bez konieczności zmiany kodu źródłowego.

Dzięki użyciu mikrofrontendu jako adaptera możemy przygotować nasz projekt pod kątem jego dalszego rozwoju w przyszłości. Możemy także zmniejszyć stopień wymaganej refaktoryzacji kodu

powłoki aplikacji, która musi być najpierw przeprowadzona, by dodać starą aplikację, a następnie zastąpić ją nową, opartą na mikrofrontendach. W obrębie powłoki aplikacji utrzymamy logikę niezależną od logiki biznesowej, gdyż komunikacja będzie tłumaczona na zdarzenia. Ten proces działa jak warstwa ochronna między wewnętrznymi a zewnętrznymi systemami. Ten wzorzec staje się również przydatny, gdy chcemy zebrać wiele aplikacji w jednym systemie i wolno, ale stopniowo zastępować przestarzałe aplikacje za pomocą mikrofrontendów implementujących wzorzec *Strangler* (*https://martinfowler.com/bliki/StranglerFigApplication.html*, strona w języku angielskim), który umożliwia aplikacji mikrofrontendowej działanie obok tych wykonanych w starszej technologii.

Tworzenie interfejsu finalizacji zakupu

Powiedzmy, że projekt spotyka się z aprobatą w organizacji i zespól Nigiri pracuje nad procesem finalizacji zakupu. Zespół odpowiedzialny za produkt i zespół od UX zdecydowały się umieścić koszyk w nagłówku powłoki aplikacji. Koszyk powinien być widoczny tylko wtedy, gdy użytkownik jest zalogowany w aplikacji sklepowej, więc ten komponent będzie obecny jedynie w określonych widokach. Po naciśnięciu przycisku koszyka nowy mikrofrontend finalizacji zakupu przeprowadzi użytkownika przez proces zamawiania produktu.

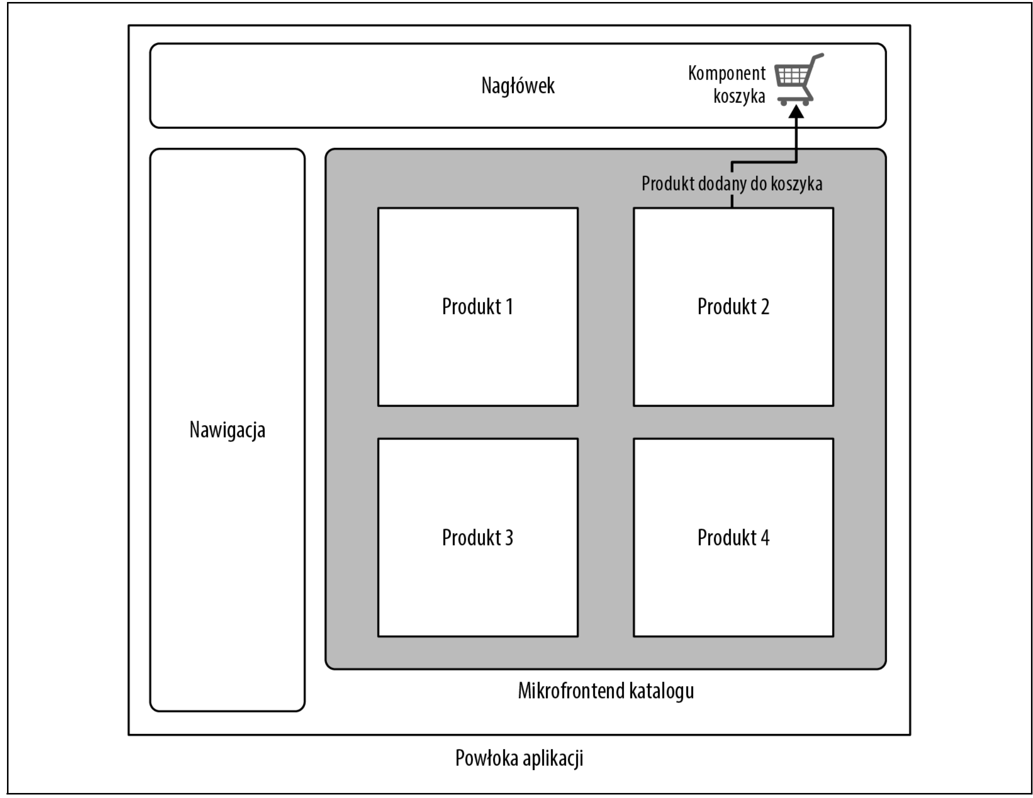
Komponent koszyka (patrz rysunek 5.7) jest odpowiedzialny za:

- ukrywanie i pokazywanie koszyka w zależności od obszaru, do którego przechodzi użytkownik;
- wyświetlanie liczby produktów w koszyku;
- rozpoczęcie procesu finalizacji zakupu.

Ponieważ komponent koszyka będzie obecny w powłoce aplikacji, musimy stworzyć logikę, która będzie go ukrywać dla niezalogowanego użytkownika, a pokazywać, gdy użytkownik pomyślnie przejdzie weryfikację. To powłoka aplikacji mogłaby zarządzać widocznością tego komponentu, ale logika domeny finalizacji zakupu przeciekałaby w ten sposób do powłoki, co zanieczyściłoby bazę kodu. Dodatkowo za każdym razem, kiedy chcielibyśmy zmienić logikę widoczności, musielibyśmy publikować nową wersję powłoki aplikacji. A ponieważ za finalizację zakupu i za powłokę aplikacji są odpowiedzialne różne zespoły, tworzenie takich zależności sprawiłoby więcej problemów, niż przyniosło korzyści.

Lepszym rozwiązaniem jest poproszenie zespołu odpowiedzialnego za powłokę aplikacji o dodanie komponentu, a komponent sam będzie obsługiwał swoją widoczność w oparciu o zestaw warunków, na przykład w oparciu o adres URL strony. W ten sposób zmiana logiki będzie miała miejsce wewnątrz komponentu i te szczegóły związane z implementacją nie będą przeciekać do powłoki aplikacji. Zespół odpowiedzialny za powłokę aplikacji będzie musiał aktualizować bibliotekę używaną do wgrywania komponentu, jeśli członkowie zespołu wybrali implementację podczas kompilacji. W przypadku wtyczki Module Federation nie będą musieli nic więcej robić, ponieważ nowy komponent będzie wgrywany podczas działania aplikacji.

Kup ksi k Pole ksi k



Rysunek 5.7. Za komponent koszyka jest odpowiedzialny zespół Nigiri. Komponent jest wgrywany wewnątrz powłoki aplikacji i używa emitera zdarzeń, aby nasłuchiwać sygnału, kiedy produkt powinien być dodany do koszyka, oraz pokazuje całkowitą liczbę produktów dodanych do koszyka

Aby wyświetlić liczbę produktów w koszyku, musimy najpierw dodać przedmiot do koszyka za pomocą API eksponowanego przez backend, a następnie powiadomić komponent koszyka, by zaktualizował liczbę wyświetlaną w interfejsie. W tym celu najlepiej emitować zdarzenie za pomocą instancji klasy emitera zdarzeń. Gdy komponent koszyka odbierze zdarzenie, za pomocą API uzyska liczbę aktualnie znajdujących się w koszyku produktów (co zostało omówione w rozdziale 4.).

Wreszcie, gdy użytkownik przez naciśnięcie komponentu koszyka rozpocznie proces finalizacji zakupu, musi być zmieniony jedynie adres URL, by powiadomić powłokę aplikacji o przejściu do mikrofrontendu finalizacji zakupu.

Jak widzisz, poświęcenie trochę czasu na początku na przemyślenie implementacji tak prostego elementu, jakim jest komponent koszyka, może przynieść wiele korzyści w dłuższej perspektywie czasowej. Ten komponent koszyka zachowa silną enkapsulację, bez względu na to, że działa w innej domenie (w powłoce aplikacji). Przez emiter zdarzeń będzie otrzymywał zdarzenia od innych części systemu i przekierowywał użytkownika do interfejsu finalizacji zakupu. Zgodnie z zasadami mikrofrontendów musimy dbać o komfort pracy naszych programistów i zapobiegać przeciekom domen do innych obszarów aplikacji.

Kup ksi k Pole ksi k

Implementacja dynamicznych kontenerów zdalnych

W tym przykładzie widzieliśmy, w jaki sposób implementować elementy zdalne przez określenie ich ścieżek w konfiguracji bundlera webpack. Jednak wtyczka Module Federation pozwala nam dynamicznie wgrywać elementy zdalne bezpośrednio z kodu JavaScript, bez konieczności wymienienia każdego mikrofrontendu dostępnego w aplikacji podczas kompilacji. Ciekawym aspektem tego podejścia jest możliwość łatwej rozbudowy naszej aplikacji bez potrzeby ponownej kompilacji powłoki aplikacji, która zawiera listę wszystkich mikrofrontendów. Na przykład możemy korzystać z API lub statycznego pliku JSON do uzyskiwania wszystkich dostępnych mikrofrontendów i składać logikę systemu globalnego trasowania w powłoce aplikacji.

Możemy również przekierować ruch w stronę określonej wersji mikrofrontendu, by zminimalizować ryzyko, że nowa wersja spowoduje poważne błędy widoczne dla szerszej grupy odbiorców. Możliwości dynamicznych kontenerów zdalnych są niesamowite i mogą naprawdę pomóc we wprowadzaniu zaawansowanej logiki do Twojej architektury. By zobaczyć tę implementację w praktyce, sprawdź repozytorium na GitHubie (*https://github.com/module-federation/module-federation-examples/tree/master/advanced-api/dynamic-remotes*) prowadzone przez zespół webpack pracujący nad wtyczką Module Federation.

Przywiązanie do bundlera webpack

Założę się, że niektórzy z Was teraz myślą, że opieranie wszystkiego na Module Federation może być ryzykowne, biorąc pod uwagę fakt, iż ta wtyczka jest obecnie w pełni obsługiwana tylko przez bundler webpack 5. Przyjrzyjmy się temu bliżej. Wtyczka Module Federation została opublikowana jesienią 2020 roku i od tego czasu zyskała wielu zwolenników wśród programistów. Łatwo się jej nauczyć, a jej implementacja w nowych lub istniejących projektach przebiega bez większych problemów. Ponadto jest odpowiedzią na złożone wyzwania, takie jak zarządzanie zależnościami czy kompozycja mikrofrontendów, co spotkało się z bardzo pozytywną reakcją społeczności programistów.

Dostępna jest również platforma programistyczna o nazwie Fronts (*https://github.com/unadlib/fronts*), która jest używana do budowy mikrofrontendów z zastosowaniem wtyczki Module Federation lub pewnych jej funkcjonalności bez korzystania z bundlera webpack. Wiele organizacji używa Module Federation na produkcji, a platformy frontendowe, takie jak Angular, zaczęły pokazywać ciekawe przykłady z użyciem tej wtyczki. Ważne jest, aby zrozumieć, czy nasza obawa związana z poleganiem na bundlerze webpack z wtyczką Module Federation jest uzasadniona, czy nie. Ile czasu zajmie Ci refaktoryzacja kodu, gdy zdecydujesz się zastosować inne rozwiązanie? Jak długo Twój produkt będzie działał na produkcji bez refaktoryzacji? Jak kluczowy jest projekt, nad którym pracujesz? Jak bardzo jest prawdopodobne, że w ciągu kolejnych trzech do pięciu lat zmieni się technologia dla tego projektu?

Jedną z głównych zalet mikrofrontendów jest możliwość stopniowej refaktoryzacji kodu. Mikrofrontendy są doskonałym rozwiązaniem pod kątem rozwijania projektów zarówno od strony biznesowej, jak i technicznej i pozwalają Ci na wypróbowywanie nowych pomysłów i rozwiązań, a następnie stopniowe ulepszanie bazy kodu. Gdy będziesz chciał unikać biblioteki lub technologii, która w pewnym stopniu Cię blokuje, musisz się zastanowić, co stracisz. A gdy zdecydujesz się rozwinąć własne narzędzia, musisz oszacować związany z tym wysiłek, a co ważniejsze, musisz przewidzieć również

Kup ksi k Pole ksi k

koszty z utrzymywaniem własnego rozwiązania, podczas gdy korzystanie z rozwiązania open source tego nie wymaga, gdyż tym zajmuje się społeczność. Czasami obawa przed związaniem się z daną technologią znika, gdy dokładnie przeanalizujesz dostępne alternatywy.

Podsumowanie

W tym rozdziale zobaczyliśmy, jak podstawowe decyzje związane z architekturą mikrofrontendową przekładają się na rzeczywistość. Każdy zespół wybrał odpowiednie podejście, by sprostać wymaganiom postawionym przez zespoły odpowiedzialne za produkt. Programiści dbali o utrzymanie rozdziału między mikrofrontendami, wiedząc, że ta droga zagwarantuje im niezależność i możliwość szybszej reakcji na każdą zmianę biznesową. W czasie tej podróży mieliśmy okazję zobaczyć również techniczną implementację architektury mikrofrontendowej z wykorzystaniem bundlera webpack i wtyczki Module Federation.

To jedno z wielu podejść opisanych w rozdziale 4. Z każdą platformą programistyczną i technologią będą związane pewne wyzwania. W tym rozdziale pokazałem Ci sposób myślenia i podpowiedziałem, jak należy podejmować pewne decyzje, co jest o wiele cenniejsze niż ocena każdej implementacji z osobna. Dzięki temu będziesz miał model myślowy, który pozwoli Ci łatwo przechodzić z jednej platformy na inną — wystarczy, że będziesz się kierował tym, czego się tutaj dowiedziałeś.

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

Mikrofrontendy: nowy wymiar niezawodności frontendu!

Mikrousługi stały się odpowiedzią na rosnącą złożoność aplikacji internetowych. Do niedawna pojęcie mikrousług dotyczyło wyłącznie backendu, jednak idea ta zainspirowała projektantów do budowania na podobnych zasadach architektury frontendu. Dzięki temu interfejs użytkownika można podzielić na osobne funkcjonalności zarządzane w odrębny sposób przez różne zespoły programistów. Mikrofrontendy zapewniają elastyczność i skalowalność aplikacji — a to zalety doceniane przez najważniejszych dostawców oprogramowania na rynku.

Ta książka jest praktycznym przewodnikiem dla programistów aplikacji internetowych, architektów oprogramowania, menedżerów technicznych i inżynierów. Wyjaśniono w niej, w jaki sposób stosować architekturę mikrousług do frontendu aplikacji. Pokazano najważniejsze zalety mikrofrontendów, takie jak elastyczność, skalowalność i swoboda w dobieraniu bibliotek i platform programistycznych. Omówiono też takie zagadnienia jak wzorce projektowe dla mikrofrontendów, zasady przeprowadzania migracji z frontendu monolitycznego do mikrofrontendów, a także praktyczne aspekty wdrażania architektury mikrofrontendowej w organizacji. Ciekawą kwestią jest również prezentacja dobrych praktyk, na przykład sprawdzonych strategii automatyzacji i wdrażania mikrofrontendów w środowisku produkcyjnym.

Najważniejsze zagadnienia:

- czym się charakteryzują architektury frontendowe
- jak stosować ideę mikrousług podczas tworzenia frontendu
- cztery filary tworzenia architektury mikrofrontendowej
- zasady i najlepsze praktyki ustalania strategii automatyzacji
- wzorce integracji architektury mikrofrontendowej

Luca Mezzalira jest głównym projektantem rozwiązań dla Amazon Web Services. Tworzeniem oprogramowania zajmuje się od prawie 20 lat, specjalizuje się w dostosowywaniu jego architektury do konkretnych zadań. Jest autorem książek i artykułów w czasopismach technicznych, często występuje na konferencjach branżowych.

ISBN 978-83-283-9318-9