O'REILLY®

MODULARNY JAVASCRIPT

DLA ZAAWANSOWANYCH

NICOLÁS BEVACQUA

Helion

Tytuł oryginału: Mastering Modular JavaScript

Tłumaczenie: Tomasz Jakut

ISBN: 978-83-283-5477-7

Helion SA
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/modjsz
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 1.

Myślenie modularne

Jak zostało powiedziane we wstępie, złożoność zdaje się być wszędzie wokół nas, gdy pracujemy nad projektami oprogramowania. Tak samo jak abstrakcje, chowające przed nami złożoność, przygniatając je głazami, pod które boimy się zaglądać. Te głazy to nasze interfejsy, przeznaczone dla reszty świata, dzięki czemu nie musimy praktycznie w ogóle myśleć o złożoności. JavaScript nie jest tutaj wyjątkiem. Wręcz przeciwnie, dzięki potędze dawanej nam przez dynamiczne języki o wiele łatwiejsze — a czasem wręcz kuszące — jest tworzenie skomplikowanych programów.

Na dobry początek omówię, jak w lepszy sposób aplikować do naszej pracy abstrakcje, interfejsy oraz koncepty będące ich fundamentami. Dzięki temu będziemy mogli zminimalizować ilość złożoności, z jaką będziemy musieli się stykać, pracując nad projektem, nową funkcją lub jej częścią, aż do pojedynczych gałęzi w funkcjach.

1.1. Wprowadzenie do myślenia modularnego

Kluczowe dla opanowania *myślenia modularnego* jest zrozumienie, że złożoność ostatecznie jest nieunikniona. Równocześnie ta sama złożoność może zostać zamieciona pod ~~dywan~~ interfejs, by już nigdy nie zostać ujrzana czy zagościć w czyichś myślach. Ale — i tutaj zaczyna się trudna sztuka — interfejs musi być dobrze zaprojektowany, by jego użytkownicy nie poczuli frustracji. Ta frustracja może wręcz popchnąć niektórych do zajrzenia pod maskę i uświadomienia sobie, że implementacja jest dalece bardziej złożona niźli biedny interfejs, na który się wściekają, i jeśli ten interfejs w ogóle by nie istniał, czytelność i utrzymanie kodu byłyby na wyższym poziomie.

Systemy mogą być zorganizowane jako zestaw luźno powiązanych części: możemy podzielić je na projekty, złożone z wielu aplikacji, zawierających z kolei wiele warstw aplikacyjnych, które mogą mieć setki modułów, złożonych z tysiąca funkcji itd. Takie podejście pomaga nam pisać kod, który jest prosty do zrozumienia i utrzymania, ma rozsądny poziom modularności i pozwala nam nie oszaleć. W podrozdziale 1.4, „Modularny podział na części", zaprezentuję sposoby na osiągnięcie sensownego podziału na części w celu tworzenia modularnych aplikacji.

Gdy szkicujemy projekt komponentu, pojawia się potrzeba istnienia publicznego interfejsu, który inne części systemu mogą wykorzystywać do komunikacji z naszym komponentem. Interfejs, czy też API, jest zbiorem metod i atrybutów, który nasz komponent eksponuje światu. Te metody i atrybuty nazwać można także *punktami styku* (ang. *touchpoints*)[1] — elementami interfejsu, z którymi można wejść w interakcję. Im mniej punktów styku, tym mniejsza powierzchnia interfejsu, a tym samym większa jego prostota. Interfejs z dużą powierzchnią jest bardzo elastyczny, ale równocześnie może być zdecydowanie trudniejszy do zrozumienia i użycia, biorąc pod uwagę liczbę funkcji, jakie eksponuje.

Interfejs odgrywa podwójną rolę. Pozwala nam dopisać kolejne kawałki naszego komponentu, eksponując jedynie te funkcje, które są już gotowe do interakcji z resztą świata, i ukrywając równocześnie wszystko to, czym nie powinniśmy się dzielić z innymi komponentami. Jednocześnie pozwala swoim użytkownikom — komponentom lub systemom używającym naszego interfejsu — czerpać korzyści z funkcji, jakie eksponujemy, bez potrzeby zawracania sobie głowy szczegółami implementacyjnymi.

Porządne, udokumentowane interfejsy są jednym z najlepszych sposobów na odizolowanie skomplikowanego kawałku kodu tak, aby inni mogli go używać bez znajomości szczegółów implementacyjnych. Stosowne ułożenie porządnych interfejsów może uformować warstwę aplikacji, taką jak usługa czy warstwa danych w aplikacji biznesowej. Dzięki temu możemy być w stanie odizolować z grubsza logikę do jednej z takich warstw, równocześnie oddzielając prezentację od zadań stricte biznesowych lub związanych z przechowywaniem danych. Taka wymuszona separacja warstw jest efektywna, ponieważ utrzymuje poszczególne

[1] Należę do tych, którzy uważają, że tłumaczenie pojęć programistycznych na język polski ma średni sens. Dlatego w wielu miejscach pojawiać się będą także oryginalne, angielskie zwroty, które mogą być bardziej zrozumiałe niż ich polskie odpowiedniki — *przyp. tłum.*

komponenty w porządku, a warstwy — w jednolitości. Jednolite warstwy, stworzone z komponentów skonstruowanych z podobnych wzorców, dają poczucie ich znajomości, sprawiające, że stają się one prostsze do wykorzystania przez programistę, który w miarę upływu czasu będzie się coraz bardziej przyzwyczajał do określonego kształtu API.

Poleganie na spójnej konstrukcji API jest wspaniałym sposobem na zwiększenie produktywności, biorąc pod uwagę trudność, z jaką przychodzi skonstruowanie odpowiedniego projektu interfejsu. Gdy konsekwentnie korzystamy z podobnych kształtów API, nie musimy za każdym razem projektować ich od nowa, a użytkownicy mogą odetchnąć z ulgą, wiedząc, że nie wymyślamy za każdym razem koła na nowo. Projektowanie API o wiele dokładniej zostanie omówione w kolejnych rozdziałach.

1.2. Krótka historia modularności

Jeśli chodzi o JavaScript, modularność jest nowoczesnym konceptem. W tym podrozdziale szybko zaprezentuję i podsumuję kamienie milowe w ewolucji modularności w świecie JavaScriptu. Ten podrozdział nie ma ambicji, by stać się wyczerpującym temat kompendium, niemniej próbuje pokazać główne zmiany paradygmatu w historii JavaScriptu.

1.2.1. Znaczniki script oraz domknięcia

W zamierzchłych czasach JavaScript był osadzany bezpośrednio w HTML-u przy pomocy znacznika `<script>`. W najlepszym przypadku był wczytywany z osobnych plików, z których wszystkie dzieliły ten sam, globalny zasięg (ang. *global scope*).

Jakakolwiek zmienna lub jakiekolwiek przypisanie zadeklarowane w jednym z tych plików lub wbudowanym skrypcie były dołączane do globalnego obiektu `window`. To powodowało wycieki pomiędzy różnymi, niezwiązanymi ze sobą skryptami i mogło prowadzić do konfliktów lub po prostu zepsucia strony, gdy zmienna z jednego pliku mogła nadpisać globalną zmienną wykorzystywaną przez inny skrypt:

```
<script>
   var initialized = false

   if (!initialized) {
```

```
      init()
   }

   function init() {
      initialized = true
      console.log('init')
   }
</script>

<script>
   if (initialized) {
      console.log('został stworzony!')
   }
   // nawet `init` stało się zmienną globalną
   console.log('init' in window)
</script>
```

Ostatecznie, gdy aplikacje sieciowe zaczęły rosnąć pod względem rozmiaru i złożoności, koncept zasięgu zmiennych oraz zagrożenia wynikające z globalnego zasięgu stały się bardziej ewidentne i znane. Kiedy zostały wynalezione Natychmiastowo Wywołujące się Wyrażenia Funkcyjne (**IIFE** — ang. *Immediately Invoked Function Expressions*), stały się z miejsca jednym z podstawowych sposobów pisania kodu. IIFE działa poprzez otoczenie całego pliku lub jego fragmentu funkcją, która jest wywoływana natychmiast po zadeklarowaniu. Każda funkcja w JavaScripcie tworzy nowy poziom zasięgu, co oznacza, że deklaracje `var` wewnątrz niej nie będą wyciekały na zewnątrz. Nawet mimo tego, że deklaracje zmiennych są wynoszone na początek zawierającego je zasięgu, nigdy nie staną się przypadkowymi zmiennymi globalnymi (ang. *implicit globals*) dzięki otoczeniu przez IIFE, chroniącemu je przed bolączkami zmiennych globalnych.

W następnym przykładzie zostało pokazanych kilka odmian IIFE. Kod wewnątrz każdego IIFE jest odizolowany od globalnego zasięgu; może się do niego odwoływać jedynie poprzez bezpośrednie instrukcje, takie jak `window.fromIIFE = true`:

```
(function() {
   console.log('IIFE używający nawiasów')
})()

~function() {
   console.log('IIFE używający operatora bitowego')
}()

void function() {
   console.log('IIFE używający operatora void')
}()
```

Używając wzorca IIFE, biblioteki zwykle tworzą moduły poprzez eksponowanie, a następnie ponowne używanie pojedynczej zmiennej doczepionej do obiektu `window`; tym samym do minimum ograniczają zanieczyszczenie globalnej przestrzeni nazw. Kolejny przykład pokazuje, jak moglibyśmy stworzyć komponent `mathlib` wraz z metodą `sum` w jednej z takich bibliotek opartych na IIFE. Jeśli chcielibyśmy dodać więcej modułów do `mathlib`, moglibyśmy każdy z nich umieścić w osobnym IIFE — dodawałby on własne metody do publicznego interfejsu `mathlib`, podczas gdy cała reszta kodu pozostawałaby prywatna dla komponentu, który zdefiniował nową funkcję:

```
void function() {
   window.mathlib = window.mathlib || {}
   window.mathlib.sum = sum

   function sum(...values) {
      return values.reduce((a, b) => a + b, 0)
   }
}()

mathlib.sum(1, 2, 3)
// <- 6
```

Ten wzorzec przypadkowo był katalizatorem rozkwitu narzędzi javascriptowych, pozwalając programistom — po raz pierwszy w historii — łączyć każdy moduł otoczony IIFE w jeden wspólny plik. To ograniczyło zużycie sieci, równocześnie oferując prymitywny mechanizm tworzenia paczek kodu (ang. *bundles*), który był w stanie poradzić sobie z mechanizmem automatycznego wstawiania średników i zminifikowanym kodem na tyle, by nie zepsuć logiki aplikacji.

Problem ze wzorcem IIFE polegał na braku sprecyzowanego drzewka zależności. Programiści musieli tworzyć listy plików z komponentami w odpowiedniej kolejności, tak aby zależności były wczytane przed potrzebującymi ich modułami.

1.2.2. RequireJS, AngularJS i wstrzykiwanie zależności

To problem, o którym prawie przestaliśmy myśleć w chwili, gdy pojawiły się systemy modułów, takie jak RequireJS, czy mechanizm wstrzykiwania zależności (ang. *dependency injection*; **DI**) w AngularJS. Obydwa te rozwiązania pozwoliły nam na bezpośrednie ustalanie zależności dla każdego modułu.

Poniższy przykład pokazuje, jak można zdefiniować bibliotekę znajdującą się w pliku *mathlib/sum.js* przy pomocy funkcji `define` z RequireJS, która została

dodana do globalnego zasięgu. Wartość zwrócona przez funkcję zwrotną (ang. *callback*) `define` jest następnie wykorzystywana jako publiczny interfejs modułu:

```
define(function() {
   return sum

   function sum(...values) {
      return values.reduce((a, b) => a + b, 0)
   }
})
```

Możemy następnie stworzyć moduł *mathlib.js*, który będzie agregował wszystkie funkcje, jakie chcemy zawrzeć w naszej bibliotece. W naszym przykładzie jedyną jej zależnością będzie *mathlib/sum*, ale możemy w analogiczny sposób określić tyle zależności, ile tylko chcemy. Zależności wypisujemy, umieszczając ścieżki ich plików w tablicy, a ich publiczne interfejsy przekazujemy jako parametry do funkcji zwrotnej, w tej samej kolejności co w przypadku ścieżek w tablicy:

```
define(['mathlib/sum'], function(sum) {
   return { sum }
})
```

Skoro już zdefiniowaliśmy bibliotekę, możemy jej użyć przy pomocy funkcji `require`. Zauważmy, że w tym przykładzie łańcuch zależności jest rozwiązywany za nas:

```
require(['mathlib'], function(mathlib) {
   mathlib.sum(1, 2, 3)
   // <- 6
})
```

To zdecydowany plus RequireJS i jego drzewka zależności opartego na dziedziczeniu. Nieważne, czy nasza aplikacja zawiera setki, czy tysiące modułów — RequireJS poradzi sobie z rozwiązaniem drzewka zależności bez konieczności dbania o poprawną kolejność listy zależności. Dzięki temu, że zależności definiujemy tylko tam, gdzie faktycznie są potrzebne, znika potrzeba listowania każdego komponentu oraz jego relacji do innych modułów. Wyeliminowany zostaje także podatny na błędy proces utrzymywania aktualności takiej listy. I pomyśleć, że usunięcie z naszego programu tak sporego źródła złożoności i tak jest jedynie skutkiem ubocznym, a nie główną zaletą używania RequireJS.

Ta bezpośredniość i precyzja w deklarowaniu zależności na poziomie samego modułu pozwalają jasno zdefiniować rolę danego komponentu w kontekście aplikacji. Dodatkowo wprowadzają większy stopień modularności — coś, co

wcześniej było nieopłacalne z powodu trudności w definiowaniu łańcuchów zależności.

RequireJS nie był jednak rozwiązaniem bez wad. Cały wzorzec opierał się na możliwości asynchronicznego ładowania modułów, co było dość szkodliwym rozwiązaniem dla środowisk produkcyjnych ze względu na jego niską wydajność. Używanie asynchronicznego mechanizmu ładowania wymuszało korzystanie z setek żądań sieciowych, jednego po drugim, zanim kod był ostatecznie wykonany. Z tego też powodu do optymalizacji aplikacji przed wdrożeniem jej do produkcji trzeba było używać innego narzędzia. Innym problemem mogła się okazać rozwlekłość; ostatecznie i tak kończyliśmy z długą listą zależności, wywołaniem funkcji RequireJS oraz funkcją zwrotną dla naszego modułu. Dodatkowo było kilka różnych funkcji w tej bibliotece oraz kilka różnych sposobów ich wywoływania, co jeszcze bardziej komplikowało sprawę. API nie należało do najbardziej intuicyjnych, ponieważ istniało mnóstwo sposobów zrobienia tego samego: zadeklarowania modułu wraz z zależnościami.

Z mechanizmem wstrzykiwania zależności w AngularJS związanych było wiele z tych problemów. W swoim czasie był rozwiązaniem eleganckim; opierało swe działanie na sprytnym parsowaniu ciągów tekstowych w celu uniknięcia tablicy zależności, używając zamiast niej nazw parametrów funkcji do rozwiązywania zależności. Ten mechanizm był niekompatybilny z minifikatorami, które zamieniały nazwy parametrów na pojedyncze znaki, równocześnie psując wstrzykiwanie.

W późniejszym okresie AngularJS w wersji 1.x wprowadził odpowiednie zadanie w procesie budowania aplikacji (ang. *build task*), które zamieniało taki kod:

```
module.factory('calculator', function(mathlib) {
   // ...
})
```

na format strawny dla minifikatorów ze względu na zawieranie dokładnej listy zależności:

```
module.factory('calculator', ['mathlib', function(mathlib) {
   // ...
}])
```

Nie trzeba chyba dodawać, że późne wprowadzenie tej mało znanej funkcji, połączone z pewnym przeinżynierowaniem wynikającym z istnienia nowego kroku w procesie budowania aplikacji, tylko po to, by naprawić coś, co nie powinno być zepsute, nie zachęcało do korzystania z tego wzorca — pomimo jego

niezaprzeczalnych zalet. Programiści zwykle wybierali pozostanie przy jawnym wypisaniu wszystkich zależności, tak jak robili to w RequireJS.

1.2.3. Node.js i świt żywego CommonJS

Jedną z wielu innowacji, jakie wprowadziło Node.js, był system modułów CommonJS (CJS). Dzięki wykorzystaniu faktu, że programy napisane w Node.js miały bezpośredni dostęp do systemu plików, standard CommonJS przypomina o wiele bardziej tradycyjne mechanizmy ładowania modułów. W CommonJS każdy plik jest modułem ze swoim własnym zasięgiem i kontekstem. Zależności są ładowane przy użyciu synchronicznej funkcji `require`, która może być wywoływana w dowolnym miejscu w module — tak jak to pokazuje następujący przykład:

```
const mathlib = require('./mathlib')
```

Podobnie jak w RequireJS czy AngularJS, w CommonJS do modułów również można się odwoływać przy pomocy ścieżki do ich pliku. Główna różnica między tymi rozwiązaniami polega na tym, że zarówno nadmiarowy kod, jak i tablica zależności zniknęły, a interfejs eksponowany przez moduł może zostać przypisany do zmiennej lub użyty w miejscu, w którym dozwolone jest wyrażenie JavaScriptu.

W przeciwieństwie do Require.js czy AngularJS CommonJS był dość rygorystyczny. W RequireJS i AngularJS mogliśmy mieć wiele modułów zdefiniowanych w jednym pliku, podczas gdy CommonJS wymuszało zależność: jeden plik = jeden moduł. Dodatkowo RequireJS oferował kilka sposobów deklarowania modułów, a AngularJS kilka rodzajów fabryk, usług, dostawców itp. — pomijając to, że sam jego mechanizm wstrzykiwania zależności był ściśle powiązany z tym frameworkiem. CommonJS z kolei zapewniało tylko jeden sposób deklarowania modułów. Dowolny plik JavaScriptu był modułem, a wywołanie `require` wczytywało jego zależności, równocześnie zamieniając wszystko, co było przypisane do `module.exports`, na publiczny interfejs modułu. To pozwoliło na stworzenie lepszych narzędzi, w tym także tych do introspekcji kodu, umożliwiając szybsze zapoznanie się z hierarchią systemu komponentów CommonJS.

W końcu wynaleziono Browserify, które znalazło sposób na zasypanie przepaści pomiędzy modułami CommonJS przeznaczonymi dla serwerów Node.js a przeglądarką. Użycie komendy terminalowej `browserify` oraz podanie jej ścieżki

do głównego modułu aplikacji kończyło się połączeniem wręcz zastraszającej liczby modułów w jeden plik, gotowy do odpalenia w przeglądarce. Niemniej najważniejszą zaletą CommonJS, która zadecydowała o jego sukcesie jako de facto standardu w ładowaniu modułów, był rejestr pakietów npm.

Choć trzeba przyznać, że npm nie jest ograniczone wyłącznie do modułów CommonJS czy wręcz pakietów JavaScript, to wciąż pozostają one najpopularniejsze w rejestrze. Wizja możliwości wybierania spośród tysięcy pakietów (w zasadzie już pół miliona — a liczba ta rośnie w zatrważającym tempie) dla swojej aplikacji, połączonej z możliwością ponownego użycia sporej części systemu zarówno w Node.js, jak i w przeglądarce, była tym, co pozwoliło CommonJS zdobyć palmę pierwszeństwa wśród systemów modułów.

1.2.4. ES6, import, Babel i Webpack

Wraz ze standaryzacją ES6, która nastąpiła w czerwcu 2015 roku, oraz pojawieniem się na długo przed tym Babela, transpilującego ES6 do ES5, dziarskim krokiem nadeszła nowa rewolucja. Specyfikacja ES6 zawierała składnię natywnych modułów JavaScriptu, często nazywanych modułami ECMAScript (ang. *ECMAScript modules*; **ESM**).

W dużej mierze ESM czerpało inspirację z modułów CJS oraz ich poprzedników. Nowe moduły oferują zarówno statyczne, deklaratywne API, jak i dynamiczne, programowalne API oparte na obiecankach (ang. *promise*)[2]:

```
import mathlib from './mathlib'
import('./mathlib').then(mathlib => {
   // ...
})
```

W ESM również każdy plik jest modułem z własnym zasięgiem i kontekstem. Jedną z głównych przewag ESM nad CJS jest to, w jaki sposób ESM zezwala na importowanie statycznie zależności i zachęca do tego. Statyczne importy znacząco poprawiły możliwości introspekcji systemów modułów dzięki temu, że mogą być analizowane statycznie i wyodrębniane leksykalnie z abstrakcyjnego drzewa składniowego (ang. *abstract syntax tree*; **AST**) każdego modułu w systemie. Statyczne importy w ESM są ograniczone do najwyższego poziomu modułu, ułatwiając parsowanie i introspekcję. Inną przewagą ESM nad `require()` z CJS jest to, że ESM pozwala także na asynchronicznie ładowanie modułów, sprawiając,

[2] Terminologia za *JavaScript. Programowanie zaawansowane*, Helion, 2016 — *przyp. tłum.*

że część grafu zależności aplikacji może być wczytywana dopiero w odpowiedzi na konkretne zdarzenia, równolegle lub leniwie w chwili, gdy są potrzebne. Mimo że w momencie powstawania tej książki funkcja ta nie jest zaimplementowana jeszcze w większości środowisk, są wyraźne przesłanki, że Node.js będzie to rozwiązanie wspierać[3].

W Node.js w wersji 8.5.0 pojawiło się wsparcie ESM ukryte za flagą `--experimental-modules`, wymuszając przy tym rozszerzenie *.mjs* dla plików modułów. Większość automatycznie aktualizujących się przeglądarek wspiera już ESM bez żadnych flag.

Webpack z kolei jest następcą Browserify, który w większości przejął rolę uniwersalnego twórcy paczek dzięki swoim licznym funkcjom. Podobnie jak w przypadku Babela i ES6, Webpack od dawna wspiera ESM zarówno w wersji statycznej (`import`, `export`), jak i dynamicznej (`import()`). Dzięki Webpackowi ESM doczekało się ciepłego przyjęcia w środowisku. Dużą rolę odegrał tutaj wprowadzony mechanizm „podziału kodu" (ang. *code-splitting*), dzięki któremu możemy podzielić kod naszej aplikacji na kilka mniejszych paczek, by poprawić wydajność w czasie pierwszego wczytywania aplikacji[4].

Dzięki temu, że ESM jest integralną częścią języka (w przeciwieństwie do CJS), można się spodziewać, że w ciągu kilku lat całkowicie przejmie ekosystem modułów.

1.3. Zalety modularnego projektowania

Wiemy już, że modularność, w przeciwieństwie do współdzielonego zasięgu globalnego, pozwala uniknąć niespodziewanych konfliktów nazw zmiennych dzięki wymuszeniu osobnego zasięgu dla każdego modułu. Poza tym modularność, poprzez rozproszenie na wiele plików, pozwala nam uniknąć sporych dawek złożoności w trakcie pracy nad konkretną funkcją. W rezultacie nasz zespół jest zdolny skupić się na jednym zadaniu i zwiększyć swoją produktywność.

---

[3] Więcej szczegółów poznasz, czytając artykuł *The Current State of Implementation and Planning for ESModules* (*https://mjavascript.com/out/esm-node*), napisany przez członka zespołu Node.js, Mylesa Borinsa.

[4] Dzielenie kodu (*https://mjavascript.com/out/code-splitting*) pozwala na podzielenie aplikacji na kilka paczek w zależności od punktu wejścia (ang. *entry point*) oraz na zebranie zależności wykorzystywanych przez kilka paczek w jednej, osobnej.

Łatwość utrzymania (ang. *maintainability*), czy też możliwość efektywnej zmiany kodu, również została zwiększona dzięki modularności. Kiedy kod jest prosty i modularny, jest łatwiejszy w rozbudowie i rozszerzaniu. Łatwość utrzymania jest cenna niezależnie od wielkości zespołu; nawet w jednoosobowym zespole, jeśli zostawimy kawałek kodu na kilka miesięcy, by po tym do niego wrócić, może być trudno go poprawić czy wręcz zrozumieć — wszystko dlatego, że od samego początku nie pisaliśmy kodu łatwego w utrzymaniu.

Modularny kod z samej swej natury jest prosty w utrzymaniu. Poprzez pisanie prostych fragmentów kodu — podzielonego zgodnie z zasadą jednej odpowiedzialności (ang. *single responsibility principle*; **SRP**), mówiącą o tym, że każdy moduł spełnia jedno zadanie — a następnie łączenie tych fragmentów w bardziej wysublimowane komponenty, jesteśmy w stanie komponować daleko większe komponenty, a ostatecznie — całe aplikacje. Kiedy każdy kawałek kodu programu jest modularny, kod wygląda na prostszy, gdy skupiamy się na konkretnym komponencie, a równocześnie, jako całość, jest zdolny do przejawiania skomplikowanych zachowań. Dokładnie tak jak proces publikowania książki, który został omówiony wcześniej.

Komponenty w modularnych aplikacjach są definiowane przez ich interfejsy. Implementacja tych komponentów nie jest ich esencją, bo tą są ich interfejsy. Kiedy interfejsy są dobrze zaprojektowane, mogą się rozrastać w sposób, który nie psuje reszty kodu oraz dotychczasowego działania komponentu, równocześnie pokrywając coraz więcej przypadków. Kiedy mamy przemyślany interfejs, skrywająca się za nim implementacja staje się łatwa do modyfikacji lub wręcz całkowitego zastąpienia. Silne interfejsy potrafią ukryć słabe implementacje, które z biegiem czasu są refaktoryzowane w bardziej porządny kod. Silne interfejsy są także doskonałe do pisania testów jednostkowych, ponieważ nie musimy się martwić implementacją, testując wyłącznie interfejs — dane wejściowe i wyjściowe komponentu czy funkcji. Jeśli interfejs jest dobrze przetestowany i porządny, możemy spokojnie zepchnąć jego implementację na dalszy plan.

Biorąc pod uwagę to, że implementacje są mniej istotne od intuicyjnych interfejsów (które nie są ściśle powiązane ze swoimi implementacjami), możemy rozważyć kompromis pomiędzy elastycznością i prostotą. Elastyczność bezsprzecznie wiąże się ze zwiększeniem złożoności, co samo w sobie jest dobrym powodem, by nie tworzyć elastycznych interfejsów. Równocześnie elastyczność jest często wymogiem, dlatego mimo wszystko należy znaleźć złoty środek pomiędzy interfejsem całkowicie pozbawionym kości a interfejsem totalnie sztywnym. Ten balans

oznacza interfejs, który będzie przyjemny dla użytkownika dzięki swojej prostocie, a równocześnie będzie umożliwiał wykonanie w razie potrzeby bardziej skomplikowanych lub niecodziennych czynności. Wszystko to musi pozostać bez znaczącego wpływu na łatwość użycia lub sporego skoku złożoności implementacji.

Kompromisy pomiędzy elastycznością, łatwością, kompozycyjnością i odpowiednią liczbą zabezpieczeń na przyszłość zostaną omówione w kolejnych rozdziałach.

1.4. Modularny podział na części

Możemy zaaplikować koncepty modularnego projektowania na każdym poziomie konkretnego systemu. Jeśli wymagania projektu przerosną jego początkowe założenia, może powinniśmy rozważyć podzielenie projektu na kilka mniejszych, z mniejszymi zespołami, prostszymi w zarządzaniu. To samo można powiedzieć o aplikacjach: kiedy stają się duże lub mocno złożone, możemy chcieć je podzielić na kilka mniejszych produktów.

Kiedy chcemy uczynić aplikację łatwiejszą w utrzymaniu, powinniśmy rozważyć stworzenie jawnie zdefiniowanych warstw kodu, dzięki czemu będziemy mogli rozwijać każdą z warstw horyzontalnie, równocześnie powstrzymując napływ złożoności lub innych dodatków w innych, niepowiązanych warstwach. Taki sam proces myślowy można zaaplikować do pojedynczych komponentów, dzieląc je na dwa lub więcej mniejszych, które są następnie łączone przez kolejny mały komponent, odgrywający tutaj rolę warstwy kompozycji. Jej jedynym zadaniem będzie zszycie razem kilku innych komponentów.

Na poziomie modułów powinniśmy dbać o to, by funkcje były proste i ekspresywne, miały opisowe nazwy i nie wykonywały zbyt dużej liczby zadań naraz. Być może będziemy mieli funkcję, której wyłącznym obowiązkiem będzie odpytywanie grupy asynchronicznych zadań, z których każde jedno będzie miało swoją własną funkcję. Funkcja, która będzie dbała o ten asynchroniczny przepływ, może stać się następnie publicznym interfejsem metody naszego modułu. Równocześnie trzeba zauważyć, że na ten interfejs składają się jedynie parametry tej funkcji, będące danymi wejściowymi, oraz rezultaty zwracane przez poszczególne zadania, stanowiące dane wyjściowe. Wszystko inne stanowi wyłącznie szczegół implementacyjny i może się w każdej chwili zmienić.

Wewnętrzne funkcje modułu nie muszą być tak sztywne jak interfejs; dopóki on się nie zmieni, możemy dowolnie zmieniać implementację — włączając w to

nawet interfejsy składających się na nią funkcji. Niemniej nie można traktować tych interfejsów po macoszemu. Kluczem do poprawnego projektowania modularnego jest bezgraniczne poszanowanie dla wszystkich interfejsów, włączając w to interfejsy wewnętrznych funkcji.

Wewnątrz funkcji również powinniśmy przenosić do komponentu (ang. *componentize*) pewne aspekty implementacji, zamykając je w funkcjach o odpowiednich nazwach; usuniemy tym samym złożoność, która będzie potrzebna dopiero później, z głównego ciała funkcji i przesuniemy ją na sam koniec funkcji. Piszemy programy, które mają być czytelne i łatwe w modyfikacji dla ludzi — w tym dla nas z przyszłości. Praktycznie każdy, kto choć trochę liznął programowania, odczuł frustrację w trakcie patrzenia świeżym okiem na fragment kodu, który sam napisał kilka miesięcy wcześniej, uświadamiając sobie, że jego projekt wcale nie był tak dobry, jak wtedy zakładał.

Pamiętajmy, że tworzenie programów komputerowych jest zadaniem przeznaczonym dla ludzi — i to zwykle współpracujących ze sobą. Nie optymalizujemy kodu dla komputerów, by uruchamiały programy tak szybko, jak tylko się da. Gdyby tak było, pisalibyśmy w kodzie binarnym lub ręcznie ustawiali odpowiednie komendy na stykach. Zamiast tego powinniśmy się skupiać na optymalizacji dla ludzi, aby programiści pozostali produktywni i szybko rozumieli, a nawet modyfikowali, kawałki kodu, z którymi wcześniej nie mieli żadnej styczności. Pracując w zgodzie z konwencjami i praktykami zrozumiałymi dla wszystkich programistów, upewniamy się, że w przyszłości rozwój aplikacji będzie przebiegał w ten sam sposób jak do tej pory.

Jeśli zaś mowa o wydajności, powinniśmy ją traktować jako funkcję produktu i przez większość czasu nie przywiązywać do niej większej wagi niż do innych funkcji. Dopóki wydajność nie ma być wyróżnikiem naszego produktu z powodów biznesowych, nie powinniśmy się martwić tym, czy system działa najszybciej na świecie. Inaczej skończymy z aplikacją, która będzie wysoce złożona, trudna w utrzymaniu, naprawianiu błędów, rozwijaniu i uzasadnianiu swego dalszego istnienia.

My, programiści, często przedobrzamy także architekturę, którą powinniśmy traktować podobnie jak optymalizację wydajności. Stworzenie architektury przewidującej wszystkie możliwe przypadki, która ma nas uratować w przyszłości, gdy już będziemy zmuszeni skalować naszą aplikację i zmagać się z miliardami żądań na sekundę, prawdopodobnie będzie nas kosztowało sporo czasu poświęconego na jej wymyślenie i zamknie nas w świecie abstrakcji — trudnych do

zrozumienia i nieprzynoszących żadnych korzyści w najbliższej przyszłości. O wiele lepiej skupić się na bieżących problemach lub takich, na które natrafić można już jutro, zamiast planować 50 lat naprzód, równocześnie kombinując, jak taki kierunek rozwoju uzasadnić.

Gdy nie planujemy tak długoterminowo, możemy zauważyć ciekawą rzecz: nasze systemy rosną w sposób bardziej naturalny, adaptując się do wymagań, jakie pojawią się w najbliższej przyszłości, a równocześnie stopniowo stając się przystosowanymi do o wiele większych wymagań. Gdy ten rozwój jest stopniowy, zauważamy, że poprawiamy lub zmieniamy abstrakcję, gdy pojawi się taka potrzeba. Jeśli wprowadzimy abstrakcje zbyt wcześnie, a okażą się one błędnie dobrane, słono zapłacimy za ten błąd. Złe abstrakcje zmuszają nas do naginania aplikacji do ich woli. Gdy odkryjemy, że jakaś abstrakcja jest zła i powinna zostać usunięta, może siedzieć już tak głęboko, że jej usunięcie będzie bolało. Może się też pojawić fałszywe poczucie utopienia zainwestowanych pieniędzy, które sprawi, że będziemy chcieli tę abstrakcję pozostawić, bo zbyt dużo krwi i potu kosztowało nas jej tworzenie.

Spora część tej książki jest poświęcona wyjaśnieniu, jak zidentyfikować oraz wykorzystać poprawnie dobrane abstrakcje w odpowiednim czasie, tak aby zminimalizować ryzyko powikłań.

1.5. Modularny JavaScript: konieczność

Biorąc pod uwagę jego historię, JavaScript jest mocno interesującym przypadkiem, jeśli rozważamy projektowanie modularne. W zamierzchłych czasach początków sieci — i to przez długi okres — nie istniały żadne ogólnie uznane praktyki. Tylko kilka osób wiedziało o języku stojącym za denerwującymi komunikatami. JavaScript, jako młody i niedojrzały, wysoce dynamiczny język, wyglądał jak dziwadło wśród statycznie typowanych języków, takich jak Java czy C#, oraz szerzej wykorzystywanych języków dynamicznych, na przykład Pythona czy PHP.

Brak natywnej modularności sieci — z powodu tego, jak wczytywane były skrypty jako fragmenty rozrzucone w różnych znacznikach `<script>` — stoi w jaskrawym kontraście do innych środowisk uruchomieniowych, w których programy są stworzone z dowolnej liczby plików, a modularne architektury są wspierane natywnie przez język, jego kompilator i środowisko oparte na systemie plików. W przypadku sieci z kolei dopiero teraz powoli przyswajamy sobie temat

natywnych modułów — coś, co inne języki programowania miały od zarania swoich dziejów. Jak zostało to omówione w podrozdziale 1.2, „Krótka historia modularności", brak natywnego mechanizmu wczytywania modułów, połączony z brakiem natywnych modułów — pomijając podział na pliki, które współdzielą globalny zasięg — zmusił społeczeństwo programistów sieciowych do stworzenia kreatywnych rozwiązań tego problemu.

Specyfikacja natywnych modułów JavaScript, która ostatecznie powstała, mocno inspiruje się wcześniejszymi pracami społeczności. Nawet w chwili pisania tej książki wciąż jesteśmy jeszcze z dwa lub trzy lata od momentu, w którym będziemy mogli w pełni efektywnie używać natywnego systemu modułów. Wzorce, które były uniwersalne i adaptowane wszędzie indziej, jak na przykład architektury warstwowe lub komponentowe, w przypadku sieci często dotąd nawet nie były rozważane.

JavaScript przez długi czas nie był uważany za poważną, nowoczesną platformę programistyczną. Zmieniło się to dopiero po uruchomieniu w kwietniu 2004 roku bety Gmaila, która pokazała moc drzemiącą w asynchronicznych żądaniach HTTP wysyłanych przez JS i działającą koncepcję aplikacji sieciowej w obrębie jednej strony (ang. *single-page application*), a później po mającej miejsce w 2006 roku premierze jQuery, które oferowało proste narzędzie do tworzenia aplikacji działającej w różnych przeglądarkach.

Wraz z pojawieniem się takich frameworków jak Backbone.js, AngularJS, Ember.js czy React nastąpił przełom i nowe techniki zaczęły być wykorzystywane także w odniesieniu do sieci:

- pisanie kodu w ES6 lub późniejszej wersji, a następnie transpilowanie go do kodu w wersji ES5, żeby zapewnić lepsze wsparcie przeglądarek;
- współdzielone renderowanie poprzez wykorzystywanie tego samego kodu na serwerze i w przeglądarce do szybkiego renderowania strony na serwerze przy pierwszym wczytaniu oraz do szybszego wczytywania stron w trakcie nawigacji;
- zautomatyzowane dzielenie kodu na paczki, produkujące jedną paczkę z całego kodu aplikacji w celu optymalizacji dostarczania kodu do przeglądarki;
- dzielenie paczek ze względu na ścieżki w aplikacji, tak aby powstało kilka paczek, z których każda jest zoptymalizowana pod inną stronę odwiedzaną w pierwszej kolejności; CSS jest pakowany razem z kodem modułu JS,

dzięki czemu CSS (nieposiadający własnego systemu modułów) również może być podzielony na paczki;

- mnóstwo sposobów optymalizowania statycznych zasobów, takich jak obrazki, w czasie kompilacji, co poprawiało produktywność w trakcie tworzenia oprogramowania przy równoczesnym dbaniu o to, aby do środowiska produkcyjnego dostał się wysoce zoptymalizowany kod.

Tak oto działa stopniowa natura innowacji w przypadku sieci.

Ta eksplozja innowacji nie wydaje się wynikać tylko z czystej kreatywności, ale także z konieczności. Aplikacje sieciowe stają się coraz bardziej złożone, tak samo jak ich zakres, przeznaczenie oraz wymagania. Staje się zatem logiczne, że ekosystem wokół nich zacznie ewoluować, by sprostać nowym wymaganiom, oferując lepsze narzędzia, lepsze biblioteki, lepsze dobre praktyki, architektury, standardy, wzorce, a przede wszystkim — większy wybór.

W kolejnym rozdziale przyjrzymy się znaczeniu *złożoności* i zaczniemy w naszych programach budować umocnienia obronne, by zapobiec jej inwazji. Dzięki stosowaniu kilku zasad izolowania logiki pomiędzy warstwami komponentów rozpoczniemy naszą wyprawę po złote runo projektowania prostszych aplikacji.

Skorowidz

D

E

F

G

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

Y

Z

Ż


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
GRUPA
Helion

ARCHITEKTURA MODULARNA I NOWOCZESNY JAVASCRIPT — PRZEPIS NA SUKCES APLIKACJI!

JavaScript jest jednym z najpopularniejszych języków programowania, wykorzystywanym na wiele sposobów. Pozwala na wdrażanie różnych paradygmatów w zależności od potrzeb i preferencji programisty. Warto wypróbować programowanie modularne w JavaScripcie choćby dlatego, że dzięki temu kod staje się czytelny, łatwy w utrzymaniu i skalowaniu. Moduł grupuje funkcjonalnie związane ze sobą dane oraz procedury. Architektura modularna, o ile tylko została poprawnie zaimplementowana, pozwala na ograniczenie złożoności kodu i ułatwia pracę nad rozwojem aplikacji. Możliwość pisania niezależnego kodu na każdym poziomie systemu daje zespołom projektowym duże korzyści!

To książka przeznaczona dla osób, które używają JavaScriptu i chcą się nauczyć pisania modularnego kodu. Wyjaśniono tu, na jakich fundamentach opiera się architektura modularna, i krótko opisano jej historię w JavaScripcie. Przedstawiono warunki, jakie musi spełniać moduł, aby dać programistom konkretne korzyści, zaprezentowano zasady refaktoryzacji kodu i omówiono znaczenie doboru właściwej struktury danych. Czytelnik dowie się również, które wzorce projektowe będą odpowiednie w danej sytuacji, a także w jaki sposób podejście modułowe może pomóc w zapewnieniu bezpieczeństwa systemu, wspierać zarządzanie zależnościami czy też procesy budowania i integracji interfejsów oraz abstrakcji. Książka zawiera sporo znakomitych wskazówek i prezentuje najlepsze praktyki w zakresie projektowania i wdrażania modularnej architektury aplikacji.

Najważniejsze zagadnienia:

- czym jest modularność i jak ewoluowała w JavaScripcie
- jak powinien wyglądać moduł i do czego służy API
- obsługa błędów i refaktoryzacja kodu
- nowoczesny JavaScript i jego funkcje przydatne do tworzenia modularnego kodu
- rozwiązywanie problemów i najlepsze praktyki

NICOLÁS BEVACQUA jest inżynierem tworzącym interfejsy użytkownika. Jest też niekwestionowanym ekspertem programowania i niestrudzonym piewcą idei open source. Doskonale zna JavaScript i chętnie dzieli się swoją wiedzą z innymi pasjonatami tego języka. Pisze książki o kodowaniu i publikuje artykuły na Ponyfoo.com. Mieszka w Buenos Aires w Argentynie.

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

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-283-5477-7

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 39,90 zł