Praktyczny przewodnik po Web Extension API: tworzenie wtyczek do przeglądarek

Przeglądarki internetowe są narzędziem, którego większość użytkowników komputerów używa na co dzień do przeróżnych celów. Ich funkcjonalność wykracza daleko poza przeglądanie stron internetowych – pozwalają na pobieranie plików, obsługę multimediów, a nawet uruchamianie zaawansowanych aplikacji internetowych. A gdyby tak móc jeszcze bardziej rozbudować ich możliwości? To właśnie umożliwiają rozszerzenia przeglądarki, czyli tzw. wtyczki. W tym artykule pokażemy, jak je tworzyć przy użyciu Web Extension API i co można dzięki nim osiągnąć.

CZYM JEST WEB EXTENSION API?

Web Extension API jest zestawem narzędzi służącym do tworzenia rozszerzeń przeglądarek internetowych (ang. *extensions, add-ons*) umożliwiającym modyfikację i rozszerzanie ich możliwości. Może być wykorzystane we wszystkich przeglądarkach opartych na silniku Chromium (m.in. Google Chrome [1], Microsoft Edge [2] czy Opera [3]), a po drobnych modyfikacjach także w Mozilla Firefox [4]. Każda przeglądarka nieco inaczej implementuje Web Extension API, więc podczas tworzenia projektu konieczne jest zapoznanie się z dokumentacją każdej docelowej platformy i przeprowadzenie odpowiednich testów.

Celem tego artykułu jest wprowadzenie do tworzenia rozszerzeń przeglądarek krok po kroku. Przedstawimy szereg praktycznych informacji i przykładów, które pozwolą na zrozumienie tematu i umożliwią stworzenie własnej wtyczki. Od tego momentu jedynym limitem jest nasza własna inwencja i pomysłowość.

PIERWSZE KROKI Z WEB EXTENSION API

Na początek musimy uzbroić się w umiejętność korzystania z podstawowych technologii wykorzystywanych przy tworzeniu stron internetowych, tj. HTML, CSS i JavaScript lub TypeScript. W tym artykule przedstawimy przykłady z wykorzystaniem JavaScript, ponieważ nie będą one zbyt rozbudowane. Użycie TypeScript nie jest tu niezbędne, lecz ułatwia pracę z większymi projektami, więc jeśli jeszcze się do niego nie przekonaliście, polecam zapoznanie się z dokumentacją tego języka [5].

Zanim zaczniemy implementować logikę naszego rozszerzenia, musimy zacząć od manifestu. Pierwszym elementem, którego przeglądarka szuka, wczytując rozszerzenie, jest plik o nazwie *manifest.json*. Jest to kluczowy element każdego rozszerzenia opartego na Web Extension API zawierający metadane i konfigurację wtyczki. Obecnie możemy spotkać się z dwiema wersjami: Manifest V2 i Manifest V3. Większość przeglądarek korzysta z wersji trzeciej, a przy wersji drugiej pozostał przede wszystkim Mozilla Firefox. W tym artykule skupimy się na nowszej wersji, a przedstawione przykłady dedykowane są Google Chrome. Manifest V2 opisany jest dokładnie w dokumentacji Mozilla Firefox [4], a dokumentacja pozostałych przeglądarek zawiera opis Manifest V3 oraz przedstawia proces migracji z V2 do V3.

Plik *manifest.json* zawierający niezbędne minimum danych przedstawiono w Listingu 1.

Listing 1. Przykładowy manifest Web Extension API

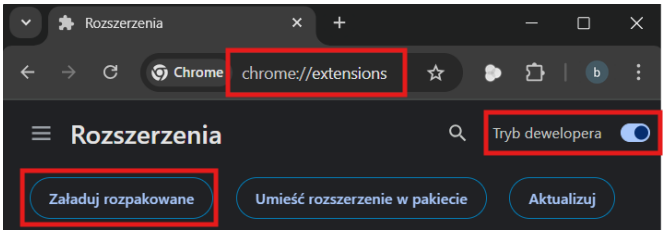
```
{
    "manifest_version": 3,
    "name": "Hello Extension!",
    "version": "1.0.0",
    "description": "Moja pierwsza wtyczka!",
    "icons": {
        "128": "icon-128.png"
    }
}
```

Poszczególne klucze oznaczają:

- `manifest_version` – liczba całkowita reprezentująca wersję manifestu (standardowo `3`, dla Firefox `2`).
- `name` – nazwa rozszerzenia widoczna w Chrome Web Store, w oknie dialogowym instalatora i w widoku zarządzania rozszerzeniami (`chrome://extensions/`); maksymalna długość to 75 znaków.
- `version` – string reprezentujący wersję wtyczki, np. `1`, `1.0.3`, `1.0 beta` lub `build rc1`.
- `description` – opis rozszerzenia widoczny w Chrome Web Store i w widoku zarządzania rozszerzeniami; maksymalna długość to 132 znaki.
- `icons` – ikony naszego rozszerzenia (w różnych rozmiarach, najlepiej w formacie PNG).

Plik *manifest.json* z zawartością jak w Listingu 1 i ikonę o wymiarach 128x128 pikseli w pliku *icon-128.png* należy umieścić w nowym folderze i gotowe. Napisaliśmy pierwsze rozszerzenie do Google Chrome!

Rozszerzenia przeglądarek standardowo skompresowane są w pojedynczym pliku ZIP, lecz podczas tworzenia i testowania projektów możliwe jest uruchomienie nieskompresowanej wtyczki (Rysunek 1). W tym celu w nowej zakładce Google Chrome należy przejść do adresu `chrome://extensions/` (lub kliknąć na ikonę puzzla po prawej stronie od paska adresu i wybrać „Zarządzaj rozszerzeniami”), a następnie w prawym górnym rogu włączyć „Tryb dewelopera”. Dostępna stanie się opcja „Załaduj rozpakowane”. Po kliknięciu w nią wybieramy folder naszego rozszerzenia i klikamy „Wybierz folder”. Poniżej powinien pojawić się kafelek reprezentujący nasze rozszerzenie.



Rysunek 1. Ładowanie rozpakowanego rozszerzenia w trybie deweloperskim

Voila! Stworzyliśmy rozszerzenie do Google Chrome, które jest rozpoznawane przez przeglądarkę i nie robi nic. Skoro wykonaliśmy pierwszy krok, możemy przejść do czegoś bardziej ambitnego.

SKRYPTY TREŚCI (CONTENT SCTIPTS)

Każda zakładka (ang. *tab*) działa w odrębnym wątku przeglądarki i jest odizolowana od pozostałych zakładek, które mogą mieć całkowicie inną zawartość. Zakładki mogą komunikować się z zewnętrznymi serwerami i pobierać z nich zawartość, która jest wyświetlana jako strona internetowa. Rozszerzenie przeglądarki działa jak serwer, który może komunikować się z zakładkami w naszej przeglądarce.

Możemy też wchodzić w interakcję z treścią stron wyświetlanych w poszczególnych zakładkach z poziomu rozszerzenia przeglądarki. Służą do tego skrypty treści (ang. *content scripts*). Z poziomu skryptu treści mamy dostęp do DOM (Document Object Model), dzięki czemu możemy odczytywać zawartość strony lub ją modyfikować. Należy jedynie pamiętać, że działają one w izolowanym środowisku, więc nie mają dostępu do zmiennych i funkcji zdefiniowanych w skryptach strony ani w innych rozszerzeniach przeglądarki.

Żeby uniknąć długiego omawiania teorii, przejdźmy do działania. Stworzymy rozszerzenie, które będzie realizowało dwa zadania: dodanie czerwonej ramki do znacznika `body` oraz wyświetlenia przywitania w konsoli. Kod takiego skryptu przedstawiono w Listingu 2.

Listing 2. content.js – przykładowy skrypt treści

```
const body = document.querySelector('body');
if (body) {
    body.style.border = '5px solid red';
}

console.log('Pozdrowienia od content scripta!');
```

Żeby rozszerzenie wykrywało nasz skrypt i wstrzykiwało go do każdej zakładki, musimy go dodatkowo zarejestrować w naszym manifeście, co przedstawiono w Listingu 3.

Listing 3. Manifest.json z deklaracją skryptu treści

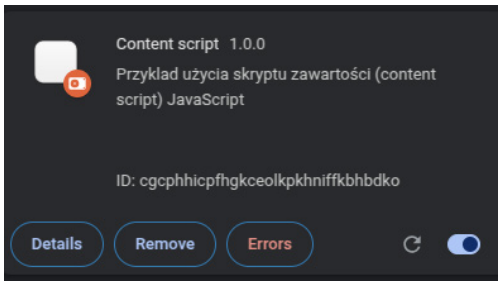
```
{
    "manifest_version": 3,
    "name": "Content script",
    "version": "1.0.0",
    "description": "Przykład użycia skryptu treści
        (content script) JavaScript",
    "icons": {
        "128": "icon-128.png"
    },
    "content_scripts": [
        {
            "matches": ["<all_urls>"],
            "js": ["content.js"]
        }
    ]
}
```

Manifest, w przeciwieństwie do poprzedniego przykładu, zawiera dodatkowy klucz `content_scripts`, w którym zdefiniowano:

- `matches` – tablica adresów lub wzorców adresów, dla których dany skrypt ma być wstrzykiwany. Można podać dokładny URL lub wzorzec z wykorzystaniem znaków `?` lub `*` oraz predefiniowanych wyrażeń. Należy ostrożnie używać `<all_urls>`, ponieważ może to niekorzystnie wpłynąć na wydajność i bezpieczeństwo.
- `js` – tablica nazw plików JS ze skryptami treści. Wszystkie nazwy podane są wraz ze ścieżką względem głównego katalogu rozszerzenia.

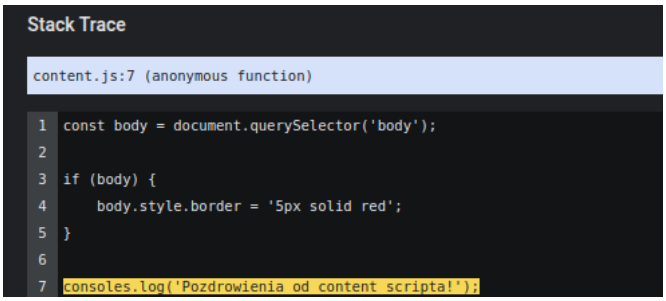
Po załadowaniu naszego rozszerzenia i przejściu na dowolną stronę widzimy, że zgodnie z naszymi oczekiwaniami wokół zawartości strony pojawiła się czerwona ramka. Gdy wejdziemy w konsolę (klawisz F12), to zobaczymy, że wśród komunikatów, które wysyła przeglądana strona, pojawi się nasz tekst `Pozdrowienia od content scripta!`. W tym samym miejscu pojawią się ewentualne błędy, które zgłosi nasz skrypt treści. Proponuję wprowadzić jakąś literówkę w kodzie, aby samemu się o tym przekonać.

Jest jeszcze jedno miejsce, w którym możemy sprawdzić komunikaty błędów, które pojawiły się podczas wykonywania naszego kodu. W widoku zarządzania rozszerzeniami (`chrome://extensions/`) w kafelku reprezentującym nasze rozszerzenie zobaczymy informację o błędzie (Rysunek 2).



Rysunek 2. Odnajdywanie komunikatów błędów

Po kliknięciu przycisku „Errors” zobaczymy szczegółowy komunikat błędu (Rysunek 3).



Rysunek 3. Przykładowy komunikat błędu skryptu treści

APLIKACJE SAMOCHODOWE ANDROID AUTO · Z ARCHIWUM CVE

INDEX: 285358

www.programistamag.pl

Magazyn programistów i liderów zespołów IT

programista

3/2025 (118)

Cena 32.90 zł (w tym VAT 8%)

CZY LLM TO SZTUCZNA INTELIGENCJA?

Praktyczny przewodnik po Web Extension API

Uwierzytelnianie dwuskładnikowe na przykładzie Google Authenticator

„Rust to rak", czyli rzecz o software maintainability

MEGA65 – komputer o wielu obliczach

Nowy numer juz w empikach