ChatGPT

Audyt modułów JavaScript w projekcie TransLogix

`stats.js`

Podsumowanie: Moduł obsługuje statystyki wyświetlane w stopce strony. Po wywołaniu `initFooterStats()` skrypt zbiera elementy `.footer__stats .stat h3` i identyfikuje je po atrybucie `data-stat` jako trzy rodzaje statystyk: *udane dostawy (deliveries)*, *kraje UE (countries)* oraz *pojazdy w ruchu (vehicles)*. Dla pierwszych dwóch wartości jednorazowo formatuje i wyświetla liczby (np. dodając znak „+" dla dostaw miesięcznych, formatując separatorem tysięcy zgodnie z polskim locale, oraz zapewnia domyślną wartość 27 dla krajów UE jeśli brak danych). Trzecia statystyka („pojazdy w ruchu") jest dynamiczna – skrypt pobiera liczbę początkową z `data-value` lub tekstu i następnie losowo zmienia ją co 5–10 sekund w niewielkim zakresie (±9 od wartości bazowej). Animacja zmiany liczby jest płynna (realizowana przez `requestAnimationFrame` z funkcją ease-out cubic). Dla użytkowników z preferencją ograniczonego ruchu (prefers-reduced-motion) animacja jest **wyłączona** – w takim przypadku skrypt wyświetla statyczną wartość i nie uruchamia zmian (respektuje ustawienie systemowe zgodnie z wytycznymi WCAG [1] [2]). Kod kończy działanie, jeśli na stronie brak wymaganych elementów (ochrona przed wywołaniem w nieodpowiednim kontekście).

Zrealizowane na poziomie profesjonalnym:
- Skrypt poprawnie *oddziela logikę dla różnych statystyk*. Każdy typ wartości (dostawy, kraje, pojazdy) jest obsłużony osobno, co ułatwia ewentualną modyfikację lub rozszerzenie. Kod najpierw bezpiecznie sprawdza istnienie elementów (`if (!statHeaders.length) return`), dzięki czemu nie wyrzuci błędów jeśli sekcja statystyk nie występuje.
- **Formatowanie i prezentacja danych** są wykonane starannie: użycie `toLocaleString("pl-PL")` zapewnia poprawne formatowanie liczb według polskich konwencji, a dodawanie znaku „+" sygnalizuje wartości typu „x+" (np. liczba dostaw *ponad* pewien próg) zgodnie z intencją. Dla statystyki krajów wbudowano domyślną wartość 27 (liczba krajów UE) na wypadek braku danych – to pokazuje dbałość o sensowny fallback.
- **Dbałość o dostępność:** Implementacja sprawdza preferencje użytkownika dotyczące animacji. Jeżeli użytkownik w systemie włączył preferencję ograniczenia ruchu, skrypt rezygnuje z animowania liczby pojazdów, co jest zgodne z zaleceniami (unikamy zbędnej animacji dla osób z chorobą symulacyjną itp.) [1] [2].
- **Wydajność i płynność:** Animacja zmiany liczby pojazdów jest realizowana optymalnie. Zamiast wykorzystywać np. setInterval z odświeżaniem DOM, użyto `requestAnimationFrame` do płynnego tweeningu wartości oraz `setTimeout` do wprowadzenia odstępów czasowych między zmianami. Dzięki temu animacja jest płynna, a obciążenie minimalne, bo odświeżanie następuje tylko podczas ~0.6–1.2 sek. trwania animacji co kilka sekund. Pętla animacji jest kontrolowana i **nie powoduje wycieków pamięci** – funkcja rekurencyjna planuje kolejną zmianę dopiero po zakończeniu bieżącej.
- **Bezpieczeństwo manipulacji DOM:** Skrypt **nie używa** niebezpiecznych operacji typu `innerHTML`. Zamiast tego, do zmiany tekstu w elementach używa bezpiecznej właściwości `textContent` (oraz manipulacji atrybutami przez `setAttribute` / `hidden`). Takie podejście eliminuje ryzyko wykonania niepożądanego kodu, nawet jeśli dane wejściowe zawierałyby potencjalnie złośliwe fragmenty HTML – `textContent` traktuje je jako zwykły tekst, chroniąc przed XSS [3]. Ponadto usunięcie wszystkich

znaków poza cyframi (`replace(/[^\d]/g, "")`) przed parsowaniem liczbowym dodatkowo **sanityzuje** wejście (np. jeśli w HTML znalazły się przypadkiem inne znaki, nie zostaną uwzględnione).

Do poprawy (elementy poprawne, ale warte rozważenia przy refaktorze):
- **Modularyzacja i skalowalność:** Kod jest czytelny, ale wszystkie operacje są w jednej funkcji. Gdyby liczba różnych statystyk rosła, można rozważyć wydzielenie części logiki (np. osobne funkcje do formatowania wartości, do obsługi animacji itp.) dla lepszej organizacji. Na obecnym etapie (tylko 3 statystyki) nie jest to jednak konieczne – moduł nie jest przerośnięty.
- **Ujednolicenie pobierania danych:** Dla każdej statystyki powtarza się wzorzec: odczyt `el.dataset.value` lub tekstu, filtracja znaków, `parseInt`. Można by to zrefaktoryzować do wspólnej funkcji pomocniczej, aby uniknąć drobnej duplikacji. Ewentualny refaktor mógłby przyjąć np. strukturę konfiguracyjną (obiekt mapujący typ statystyki na funkcję obsługi), co ułatwi dodanie nowych typów w przyszłości.
- **Obsługa wyjątkowych scenariuszy:** Skrypt zakłada jedno wywołanie na stronę. Gdyby z jakiegoś powodu `initFooterStats()` zostało wywołane ponownie (lub kilka razy), animacja dla pojazdów zostałaby zaplanowana wielokrotnie. Można rozważyć zabezpieczenie przed wielokrotną inicjalizacją (np. poprzez ustawienie atrybutu lub właściwości flagowej przy pierwszym uruchomieniu). To jednak scenariusz mało prawdopodobny w typowym użyciu.
- **Dostępność dynamicznej treści:** Aktualnie zmieniająca się liczba pojazdów jest traktowana jako ozdobnik i nie ma żadnego mechanizmu informowania o zmianie użytkowników czytników ekranu. Jeżeli celem tej animacji jest wyłącznie estetyka, to jest to akceptowalne (lepiej nie zarzucać czytnika ciągłymi aktualizacjami). Jeśli jednak uznano by tę informację za istotną, można by dodać `aria-live="polite"` do kontenera statystyk, by **grzecznie** anonsować zmiany. To do rozważenia w przyszłości – obecnie prawdopodobnie nie jest to krytyczne, zwłaszcza że osoby z włączonym prefer-reduced-motion i tak zobaczą statykę.

Potencjalne ryzyka:
- **Brak krytycznych błędów:** Kod nie wykazuje poważnych bugów ani podatności. Potencjalne ryzyka są **umiarkowane i dotyczą głównie edge-case'ów**. Na przykład, jeśli atrybuty `data-value` lub tekst dla danej statystyki są błędnie sformatowane (nie-numeryczne), moduł po prostu pominie aktualizację tej wartości (`Number.isNaN` zwróci true i blok kodu się pominie). W efekcie dana liczba mogłaby pozostać niezmieniona lub pusta w HTML – nie jest to groźne dla działania strony, choć może zaburzać prezentację. Jest to jednak ryzyko łatwe do wykrycia podczas testów zawartości strony.
- **Wyjątkowe zachowanie localStorage (niskie ryzyko):** (Ten moduł akurat nie używa localStorage – dotyczy to `theme.js` – więc brak bezpośredniego wpływu).
- **Teoretyczne przeciążenie CPU:** Animacja pojazdów uruchamia nieskończoną sekwencję timeoutów. W normalnych warunkach (przeglądarka aktywna, jedna animacja co kilka sekund) obciążenie jest znikome. Gdyby jednak strona z stopką była pozostawiona otwarta na bardzo długo, wykonywałaby się duża liczba iteracji animacji. Nie jest to jednak memory leak – stare timeouty się wykonują i planują kolejne, więc pamięć się nie zwiększa. Przeglądarki dodatkowo spowalniają timeouty w nieaktywnych kartach, więc wpływ na CPU w tle będzie minimalny. Ogółem nie ma tu poważnego ryzyka wydajnościowego.
- **Losowość zakresu animacji:** Skrypt losuje wartości pojazdów w zakresie ±9 od bazowej. W skrajnym przypadku (np. bazowa liczba bardzo mała) te wahania procentowo mogą być duże, a nawet mogą skutkować *zmianą liczby cyfr* (np. 100 może losowo spaść do 92, zmieniając wyświetlanie z trzy- do dwucyfrowego). To może powodować drobne przesunięcia layoutu. Jest to jednak detal estetyczny. Założeniem zapewne jest, że liczby będą na tyle duże, by różnica rzędu 9 nie zmieniała rzędu wielkości.

Rekomendacja: Kod `stats.js` jest napisany solidnie i **bez większych zastrzeżeń nadaje się do wdrożenia produkcyjnego**. Zalecam utrzymanie obecnej architektury, dbając jedynie o poprawne wypełnienie danych w HTML (tak by `data-stat` i `data-value` były prawidłowo ustawione). W

przyszłości, jeśli dodamy więcej dynamicznych statystyk, można rozważyć refaktor na podejście bardziej konfigurowalne. Warto również uwzględniać preferencje użytkowników (co już jest zrobione) i ewentualnie przeprowadzić testy dostępności, aby potwierdzić, że dynamiczna treść nie sprawia problemów. Ogólnie jednak komponent spełnia swoje zadanie w sposób **profesjonalny i bezpieczny**.

`tabs.js`

Podsumowanie: Ten plik dostarcza funkcję `initTabs()`, która inicjalizuje dostępne **zakładki (tabs)** na stronie. Wyszukuje w DOM wszystkie kontenery z `role="tablist"` i dla każdego z nich ustawia mechanizmy interakcji zgodne z wytycznymi ARIA. Każdy element z `role="tab"` wewnątrz takiego kontenera zostaje skonfigurowany: kliknięcie myszy aktywuje odpowiadającą zakładkę (pokazując powiązany panel treści, ukrywając pozostałe), a obsługa klawiatury umożliwia nawigację między zakładkami **strzałkami** (lewo/prawo – z zawijaniem na końcach, Home – skok do pierwszej zakładki, End – do ostatniej). Gdy zakładka zostaje aktywowana, skrypt ustawia jej atrybut `aria-selected="true"`, jednocześnie oznaczając wszystkie inne jako `aria-selected="false"`, oraz przełącza ich dostępność focusa poprzez `tabIndex` (aktywna zakładka dostaje `tabIndex=0`, reszta `-1`). Powiązane panele treści są show/hide realizowane poprzez ustawianie `hidden=true/false`. Po inicjalizacji, jeśli żadna zakładka nie była zaznaczona w HTML, skrypt aktywuje domyślnie pierwszą zakładkę. Cała ta logika zapewnia, że komponent zakładek jest **dostępny dla użytkowników klawiatury i czytników ekranu**, zgodnie ze standardowym wzorcem ARIA dla tabów.

Zrealizowane na poziomie profesjonalnym:
- **Wdrożenie standardu WAI-ARIA:** Implementacja dokładnie odpowiada zaleceniom dla dostępnych zakładek. Dla każdej grupy tabów kontener ma `role="tablist"`, przyciski mają `role="tab"`, a panele powinny mieć `role="tabpanel"` (co zakładam jest w HTML). Skrypt dynamicznie ustawia atrybuty `aria-selected` oraz właściwość `hidden` tak, by w każdym momencie tylko aktywna zakładka była zaznaczona i tylko odpowiadający jej panel widoczny [4]. Takie powiązanie ról i atrybutów jest wymagane przez specyfikację ARIA – np. zawsze **dokładnie jedna zakładka powinna mieć** `aria-selected="true"` **i być w tab order (**`tabindex="0"`**), pozostałe** `aria-selected="false"` **i** `tabindex="-1"`**, a niewybranym panelom ustawiamy** `hidden` [4]. Kod to zapewnia.
- **Obsługa klawiatury:** Użytkownicy mogą nawigować między zakładkami za pomocą klawiszy strzałek oraz skoków Home/End. Zaimplementowano logikę, że strzałka w prawo przechodzi do następnej zakładki (lub na początek, jeśli jesteśmy na ostatniej), a strzałka w lewo do poprzedniej (lub na koniec, jeśli aktualnie na pierwszej) [5]. Klawisze Home i End szybko przenoszą fokus (i aktywują) odpowiednio pierwszą i ostatnią pozycję. Jest to pełen zakres oczekiwanej funkcjonalności dla komponentu zakładek zgodnie z wzorcami dostępności (Home/End są opcjonalne, ale dobrze że zostały dodane [6]). Co ważne, **aktywacja zakładki następuje natychmiast po zmianie fokusu strzałkami**, bez wymogu dodatkowego Enter – jest to tzw. model *automatic activation*, zalecany gdy zmiana treści nie jest obarczona opóźnieniem [7]. Dzięki temu użytkownik przeglądający klawiaturą może szybko obejrzeć zawartość kolejnych paneli, co poprawia UX.
- **Focus management:** Skrypt zadbał o przeniesienie fokusu na nowo aktywowaną zakładkę (`tab.focus()` w funkcji activateTab). W efekcie, gdy zmieniamy zakładkę klawiaturą lub kliknięciem, fokus zawsze pozostaje na elemencie zakładki, który jest aktywny. To zapobiega sytuacjom pogubienia fokusu i sprawia, że nawigacja jest intuicyjna. W połączeniu z właściwym `tabIndex` (tylko aktywny w tab order), realizuje to tzw. *roving tabindex* pattern.
- **Wiele niezależnych zestawów zakładek:** Użycie `document.querySelectorAll("[role='tablist']")` i iteracja `forEach` oznacza, że na jednej stronie może być wiele komponentów tabów, a inicjalizacja obejmie każdy z osobna. To pokazuje, że kod został napisany z myślą o reużywalności – logika nie jest na sztywno przypisana do jednego ID czy

selektora, tylko **automatycznie wykrywa wszystkie zestawy**. Dzięki temu moduł jest łatwo przenoszalny i skalowalny (np. można mieć sekcję zakładek w różnych częściach strony).
- **Czytelność i prostota:** Implementacja jest zwięzła i czytelna. Użyto nowoczesnej składni (arrow functions, `const/let`, spread/Array.from) co sugeruje, że kod jest świeży i przejrzysty. Nie ma zbędnych zależności – czysty JavaScript. Komentarz na początku pliku jednoznacznie stwierdza cel: *"Accessible tabs component with keyboard navigation"*, co dobrze komunikuje zamiar modułu. Kod jest strukturalnie prosty: wewnętrzna funkcja `activateTab` oraz pętle dodające event listenery. Taka kapsułkowana struktura ułatwia utrzymanie – zmieniając np. sposób aktywacji, wystarczy edytować funkcję `activateTab` w jednym miejscu.

Do poprawy (elementy poprawne, ale warte rozważenia przy refaktorze):
- **Obsługa klawiszy Enter/Space:** W obecnej implementacji zakładki są aktywowane przez klik lub od razu przy zmianie fokusu strzałkami. Założeniem jest zapewne, że elementy z `role="tab"` są fizycznymi przyciskami (`<button>`) lub linkami – w takim wypadku naciśnięcie Enter/Spacji wywoła zdarzenie kliknięcia automatycznie. Jeśli jednak w HTML zakładki byłyby np. `div`-ami z rolą tab (czysto semantycznymi, nie fokusowalnymi bez tabindex), to warto byłoby dodać nasłuch na Enter/Space i traktować je jak kliknięcie (tyczy się to trybu *manual activation* zgodnie ze specyfikacją [6]). W większości przypadków użycie `<button role="tab">` rozwiązuje problem, więc obecna implementacja jest poprawna – tę uwagę należy rozważyć tylko, jeśli kiedyś zmienimy strukturę HTML zakładek.
- **Ewentualna obsługa układu pionowego:** Wzorzec ARIA przewiduje, że jeśli zakładki są wyświetlane pionowo (`aria-orientation="vertical"` na tablist), to nawigacja klawiaturą powinna używać klawiszy Up/Down zamiast Left/Right [8]. Aktualny kod zakłada orientację poziomą (co jest domyślne). Jeżeli w przyszłości pojawią się pionowe taby, można dodać warunek sprawdzający `aria-orientation` i odpowiednio interpretujący strzałki góra/dół. Na ten moment prawdopodobnie nie jest to potrzebne (brak takiego przypadku w projekcie).
- **Czyszczenie zdarzeń przy zmianie DOM:** To bardziej uwaga architektoniczna – w obecnym podejściu, przy tradycyjnej nawigacji między stronami, problem nie występuje. Jednak gdyby ten kod był używany w SPA (single-page application) i komponent tabów miałby być niszczony i tworzony ponownie, warto zapewnić usunięcie wcześniej podpiętych listenerów (lub inicjalizować zakładki tylko raz). Rozwiązaniem może być trzymanie referencji do funkcji listenerów i usuwanie ich w razie potrzeby. W kontekście statycznej strony B2B raczej nie jest to konieczne – wspominam jako potencjalny kierunek przy ew. większej refaktoryzacji na framework.
- **Walidacja struktury HTML:** Skrypt zakłada, że każdy element z `role="tab"` posiada atrybut `aria-controls` wskazujący ID elementu panelu. Warto upewnić się, że w HTML każdy panel ma odpowiadający mu `id` oraz `role="tabpanel"`, a także że panele te są rodzeństwem tablisty lub we wspólnej strukturze. Kod pobiera panel przez `document.getElementById()` – to skuteczne i szybkie, ale oznacza że **ID muszą być unikalne w skali całego dokumentu**. To zwykle jest spełnione, lecz przy generowaniu treści należy zachować ostrożność, by np. nie zduplikować ID paneli. W razie gdyby `aria-controls` wskazywało element nieistniejący, bieżąca implementacja po prostu pominie chowanie/pokazywanie takiego panelu (warunek `if (panel) {...}`) – mogłoby to skutkować np. pozostaniem dwóch paneli jednocześnie widocznych. Nie jest to problem kodu per se, a kwestia spójności danych – jednak warto testować komponenty po zmianach w HTML.

Potencjalne ryzyka:
- **Błędna integracja HTML (ryzyko umiarkowane):** Jak wspomniano, największe ryzyko to niepoprawne użycie komponentu w HTML. Jeśli deweloper zapomni np. ustawić `aria-controls` lub da niepasujące ID panelu, funkcjonalność zakładek może częściowo nie zadziałać (np. panel nie będzie się chował). Takie pomyłki powinny wyjść podczas QA – nie wpływają na bezpieczeństwo, a na poprawność działania. Dokumentacja dla integrujących ten moduł mogłaby to podkreślać.
- **Użycie nieoptymalnych elementów interaktywnych:** Jeżeli zakładki nie będą natywnymi elementami

interaktywnymi (<button> lub <a>), a np. elementami `<span>` z dodanym `role="tab"`, to brak obsługi Enter/Space może utrudnić aktywację zakładki z klawiatury (trzeba wtedy użyć strzałek). To jednak łatwo naprawić zmieniając znacznik na `<button>` lub dodając odpowiedni listener. Ogólnie to drobne ryzyko kompatybilności z oczekiwaniami użytkownika, nie zagrożenie systemowe.
- **Brak wpływu na bezpieczeństwo:** Kod nie operuje na danych użytkownika, nie tworzy też żadnych węzłów HTML z stringów – nie ma tu wektora XSS czy podobnych zagrożeń. Użycie `setAttribute` do modyfikacji atrybutów (np. `aria-selected`) jest bezpieczne, szczególnie że nazwy atrybutów są stałe i nie pochodzą z zewnętrznego inputu [9]. Nie stwierdzono również ryzyk dla wydajności: zdarzenia są podpinane tylko na elementach tabów (zwykle jest ich kilka na stronie), a operacje przy kliknięciu/keydown są proste (kilka zmian atrybutów/klas). To **bardzo niewielkie obciążenie** nawet na słabszych urządzeniach.
- **Dostępność treści paneli:** Choć sam moduł tego nie reguluje, warto nadmienić, że dla pełnej dostępności paneli zakładek, każdy widoczny panel powinien mieć fokusowalny początek (np. nagłówek lub `tabindex="0"` na elemencie panelu), aby po naciśnięciu Tab użytkownik przeszedł do treści aktywnego panelu [10]. Jeśli treść panelu zaczyna się np. od akapitu (nie fokusowalnego), można dodać `tabindex="0"` do kontenera panelu (zostanie on zdjęty przez `hidden` gdy panel nieaktywny). To niuans, ale zgodny z praktykami MDN [10]. Kod `tabs.js` nie zajmuje się tym aspektem (słusznie, to kwestia statycznego markup), ale integratorzy powinni mieć to na uwadze. Źle ustawione tabindexy mogłyby chwilowo dezorientować użytkownika klawiatury (to nie tyle ryzyko błędu, co **pole do poprawy użyteczności**).

Rekomendacja: `tabs.js` to **dojrzała i dopracowana implementacja** komponentu zakładek. Zapewnia zgodność z ARIA i wygodę obsługi, co wskazuje na wysoki poziom wykonania. Rekomenduję wprowadzić go na produkcję praktycznie bez zmian. Przed wdrożeniem warto jedynie sprawdzić spójność HTML (role, aria-controls, id paneli) we wszystkich miejscach użycia. Dodatkowo można przeprowadzić testy użyteczności z klawiaturą i czytnikiem ekranu, aby upewnić się, że kolejność fokusa i etykiety spełniają oczekiwania. Jeśli kiedyś zajdzie potrzeba obsługi bardziej złożonych scenariuszy (np. zakładki zamykane, dynamicznie dodawane), kod będzie trzeba rozbudować – obecnie jednak w pełni pokrywa wymagany zakres. Ogólnie komponent można ocenić jako **bardzo dobrze przygotowany (production-ready)** w swojej klasie.

`theme.js`

Podsumowanie: Moduł ten odpowiada za mechanizm przełączania motywu kolorystycznego strony (tryb jasny/ciemny). Po wywołaniu `initThemeToggle()` skrypt wyszukuje przycisk przełącznika motywu (selektor `.theme-toggle`). Jeśli go znajdzie, ustala najpierw preferowany motyw startowy: sprawdza, czy w `localStorage` zapisano ostatni wybór użytkownika pod kluczem `"translogix-theme"`. Jeśli tak, używa go; jeśli nie, odczytuje systemowe ustawienie za pomocą `window.matchMedia("(prefers-color-scheme: dark)")` – na tej podstawie wybiera motyw domyślny: ciemny lub jasny [11]. Następnie motyw jest **stosowany globalnie** poprzez dodanie do elementu `<html>` klasy CSS odpowiadającej wybranemu trybowi (`theme-dark` lub `theme-light`), przy jednoczesnym usunięciu klasy przeciwnej. Skrypt aktualizuje również stan przycisku: ustawia atrybut `aria-pressed` (wskazując czy aktywny jest tryb ciemny = przycisk wciśnięty) oraz `aria-label` – dynamicznie zmienia tekst etykiety na „Przełącz na tryb jasny/ciemny” zależnie od obecnego stanu. Wreszcie, podpinany jest event handler na kliknięcie tego przycisku: każde kliknięcie toggluje wartość motywu (jasny <-> ciemny), ponownie stosuje odpowiednią klasę na `<html>`, zapisuje nowy wybór do `localStorage` i aktualizuje atrybuty ARIA przycisku. Dzięki temu użytkownik może ręcznie przełączać motyw, a strona pamięta ten wybór przy kolejnych wizytach. Całość wykonuje się tylko po stronie klienta (brak przeładowania strony przy zmianie motywu).

Zrealizowane na poziomie profesjonalnym:
- **Preferencje użytkownika i fallback:** Implementacja uwzględnia pełen **łańcuch preferencji**: najpierw zapamiętany wybór w `localStorage`, następnie preferencja systemowa, na końcu domyślny jasny motyw. Takie podejście jest zgodne z najlepszymi praktykami tworzenia przełączników motywów [11]. Dzięki temu użytkownik zawsze dostaje motyw zgodny z jego oczekiwaniem już przy pierwszym wejściu (np. tryb ciemny automatycznie, jeśli systemowo woli ciemny, chyba że wcześniej wybrał inaczej).
- **Trwałość ustawień:** Użycie `localStorage` zapewnia, że wybór użytkownika jest zachowany pomiędzy sesjami/przeładowaniami strony [12]. Klucz `"translogix-theme"` jest unikalny dla tej aplikacji. Operacje zapisu/odczytu są wykonane we właściwych miejscach (odczyt raz na inicjację, zapis przy każdym kliknięciu). To standardowy i solidny sposób utrwalania preferencji w aplikacjach web.
- **Bezpośrednia manipulacja klasami CSS:** Zastosowanie `document.documentElement.classList.toggle(...)` do dodawania/usuwania klas `theme-dark/theme-light` jest prostym i wydajnym rozwiązaniem. Unikamy tutaj przebudowy stylów in-line czy przeładowywania arkuszy – strona zapewne w CSS ma zdefiniowane style zależne od tych klas. Toggling klas jest *atomiczny* i łatwy do kontrolowania (tylko dwie możliwości). Ponadto klasa jest dodawana/usuwana **tylko na** `<html>`, więc zmiana motywu wpływa na całą stronę, co jest właściwe (CSS może dalej propagować styl w zależności od przodka).
- **Aspekty dostępności (ARIA):** Skrypt bardzo ładnie dba o dostępność przycisku przełączania. Atrybut `aria-pressed` jest użyty aby oznaczyć stan przełącznika trybu (on/off), co jest zgodne ze wzorcem dla przycisków typu toggle – wciśnięty oznacza stan „dark mode aktywny". Dodatkowo zmienia się etykieta widoczna dla czytników (`aria-label`): np. gdy obecny jest tryb ciemny, etykieta brzmi „Przełącz na tryb jasny". To informuje użytkownika *co akcja zrobi* (a nie tylko w jakim jest stanie, co mogłoby być mylące). Ta praktyka – dynamiczna aktualizacja etykiety – jest rekomendowana, zwłaszcza jeśli na przycisku jest tylko ikona [13]. Realizacja tego w kodzie (funkcja `updateToggleA11y`) jest czytelna i łatwo ją dostosować (np. do zmiany tekstów, lokalizacji).
- **Minimalistyczny i efektywny kod:** Cały moduł jest niewielki, robi tylko to co potrzeba, a jednocześnie niczego nie pomija. Sprawdza obecność elementu toggle i **bezpiecznie kończy działanie, jeśli go brak** (np. strona może nie mieć przełącznika – wtedy nie pojawi się błąd, co świadczy o przewidzeniu takiego scenariusza). Użyto nowoczesnych metod: `matchMedia` do wykrycia trybu systemowego (to lepsze niż np. ręczne wykrywanie godziną czy motywem kolorystycznym w CSS), `classList.toggle` z drugim argumentem (wymuszenie stanu) – to klarownie ustawia odpowiednie klasy. Zdarzenie `click` jest obsłużone poprawnie. Warto dodać, że operacje wykonywane przy kliknięciu są bardzo lekkie (ustawienie klasy i atrybutów, zapis do localStorage), więc **nie występują tu żadne problemy wydajnościowe**.

Do poprawy (elementy poprawne, ale warte rozważenia przy refaktorze):
- **Weryfikacja wartości z localStorage:** Obecnie kod zakłada, że w localStorage będzie wartość `"dark"` lub `"light"` (lub brak klucza). W skrajnie rzadkim wypadku, gdyby tam znalazła się inna wartość (np. wskutek ręcznej manipulacji przez użytkownika lub nietypowego błędu), funkcja `getPreferredTheme()` zwróci tę wartość, a następnie `applyTheme()` spróbuje ją zastosować. Ponieważ w `applyTheme` porównania są tylko do `"dark"` i `"light"`, efekt będzie taki, że żadna z klas nie zostanie dodana (oba porównania dadzą false, więc toggle usunie ewentualne klasy). Strona wtedy pozostanie bez klasy motywu – prawdopodobnie zadziała domyślny styl (zapewne jasny). To nie jest groźny błąd, ale potencjalnie może spowodować niespójny wygląd. Można temu zapobiec czyszcząc niespodziewane wartości – np. jeśli `stored` nie jest `"dark"` ani `"light"`, potraktować jak brak (czyli wybrać ścieżkę `matchMedia`). To drobne usprawnienie dla szczelności logiki.
- **Reagowanie na zmianę preferencji systemowej:** Aktualnie preferencja systemowa jest brana pod uwagę tylko przy pierwszym załadowaniu (gdy brak własnego ustawienia). Nie ma mechanizmu nasłuchiwania zmiany systemowego motywu w locie. Rozważenie tego nie jest konieczne (użytkownik ma przecież toggle i własny wybór przechowany), ale dla pełni nowoczesności można by dodać `matchMedia(...).addEventListener("change", ...)` i np. jeśli użytkownik nie miał własnego

wyboru, automatycznie przełączyć motyw gdy system przełączy (scenario: ktoś ustawia globalny dark mode w OS podczas otwartej strony). To jednak funkcjonalność *nice-to-have*, zależna od wymagań – obecne założenie, że strona ustawia się raz przy load, jest zupełnie akceptowalne.
- **Internacjonalizacja etykiet:** Teksty „Przełącz na tryb jasny/ciemny" są wpisane na sztywno po polsku. Jeśli aplikacja będzie wielojęzyczna, trzeba będzie ten fragment dostosować. To jednak uwaga poza czysto techniczną oceną – sam kod jest łatwy do ewentualnego przerobienia na pobieranie napisów z data-attributes lub globalnej mapy tłumaczeń.
- **Obsługa starszych przeglądarek:** Kod korzysta z `matchMedia` i `localStorage`, co we współczesnych przeglądarkach jest standardem. Bardzo stare przeglądarki (np. IE11) nie obsłużą modułów ES6 ani `prefers-color-scheme`, ale zakładam, że w kontekście projektu B2B nie wspieramy przestarzałych środowisk. Gdyby jednak wymagane było szersze wsparcie, należałoby dodać odpowiednie polyfille lub detekcje. W tej chwili nie wydaje się to potrzebne – to bardziej punkt sprawdzenia zgodności z założeniami projektowymi.

Potencjalne ryzyka:
- **Ryzyko XSS: brak** – Kod nie wprowadza do DOM żadnych danych pochodzących od użytkownika. Operacje to przełączanie klas i atrybutów o z góry znanych nazwach, oraz zapisywanie/odczyt stałych stringów z localStorage. Nie ma tu miejsca na wywołanie niebezpiecznych funkcji czy wstrzyknięcie HTML/JS. Dla ścisłości: nawet gdyby ktoś zewnętrzny manipulował zawartość `localStorage` (co samo w sobie wymaga już dostępu do komputera), najgorsze co może osiągnąć to brak dopasowania dowartej klasy motywu – żadne wykonanie złośliwego kodu nie nastąpi. Użycie `setAttribute` z bezpiecznymi atrybutami (aria, aria-label) i manipulowanie `.classList` to tzw. **safe sinks** – nie stwarzają zagrożenia wg OWASP [9].
- **Ewentualne problemy z dostępnością:** O ile sam mechanizm ARIA jest poprawny, warto upewnić się, że element `.theme-toggle` jest faktycznie przyciskiem lub przynajmniej elementem z `tabindex` i rolą button. Skrypt zakłada, że można na nim wyłapać klik i że jest fokusowalny. Jeśli markup byłby niewłaściwy (np. zwykły `<span>` bez roli), mogłoby to ograniczyć dostęp z klawiatury. To jednak kwestia integracji – sam kod reaguje na `querySelector`, czyli zarówno `<button class="theme-toggle">`, jak i `<div class="theme-toggle" role="button" tabindex="0">` będą działać. Należy po prostu stosować poprawne praktyki HTML (preferowany jest `<button>`).
- **Bardzo rzadkie przypadki Storage Exception:** W przeszłości, w trybie prywatnym Safari, wywołanie `localStorage.setItem` mogło rzucić wyjątek (QuotaExceeded) jeśli storage był niedostępny. Obecnie większość przeglądarek po prostu odmawia zapisu bez wyjątku lub ma osobny storage per tryb. Ryzyko, że zapis się nie powiedzie, jest minimalne. W najgorszym razie – gdyby `setItem` nie zadziałało – użytkownik po odświeżeniu strony straci zapamiętanie wyboru motywu. Strona nadal będzie działać, tylko wróci do domyślnego trybu. Nie jest to scenariusz krytyczny, i raczej ma zasięg ograniczony do specyficznych konfiguracji, niemniej warto być tego świadomym.
- **Brak innych ryzyk:** Zmiana motywu potencjalnie powoduje przebudowę layoutu (przeładowanie stylów z CSS dla innego motywu), co może chwilowo obciążyć rendering, ale to jest nieodczuwalne i spodziewane (użytkownik świadomie kliknął przełącznik). Nie stwierdzono tu żadnych problemów z wydajnością czy logiką – kod jest prosty i trudny do złamania.

Rekomendacja: `theme.js` jest **gotowy do produkcji** i spełnia swoje zadanie zgodnie z obecnymi standardami. Zalecam przetestowanie go w różnych warunkach (różne przeglądarki, systemowe tryby jasny/ciemny) – tak dla pewności, choć wszystko wskazuje, że będzie działał prawidłowo. W przyszłości, jeśli pojawią się nowe wymagania (np. więcej niż dwa motywy, czy synchronizacja motywu między frontendem a backendem), będzie trzeba rozbudować mechanizm – obecna architektura (prosta funkcja inicjalizująca) jest łatwa do modyfikacji lub przepisania na inny styl (np. wykorzystanie `data-theme` atrybutu zamiast klas, jak sugerują niektóre artykuły [14]). Na ten moment jednak implementacja jest **lekka, czysta i skuteczna**. Wdrożenie jej przyniesie użytkownikom możliwość wygodnego przełączania trybu z zachowaniem ich preferencji, co zwiększa ergonomię strony.

Ocena końcowa jakości paczki modułów

Analizowane trzy moduły – **stats.js**, **tabs.js** i **theme.js** – prezentują **wysoką jakość kodu** oraz dbałość o szczegóły istotne na produkcji (dostępność, wydajność, bezpieczeństwo). Każdy z nich realizuje odrębną funkcjonalność w sposób **klarowny i zgodny z najlepszymi praktykami**:

- Architektura modułowa jest zachowana – skrypty nie korzystają ze zmiennych globalnych (poza standardowymi obiektami `document`, `window` itd.), nie kolidują ze sobą i mogą być ładowane niezależnie. Podział odpowiedzialności jest logiczny.
- We wszystkich plikach widać świadomość **wydajności**: brak obciążających pętli na dużych kolekcjach czy nasłuchiwania globalnych zdarzeń w sposób mogący powodować *lag*. Tam, gdzie to potrzebne (animacja liczby), zastosowano optymalny mechanizm `requestAnimationFrame`. Inicjalizacje są robione jednokrotnie i warunkowo.
- Szczególną zaletą jest uwzględnienie **dostępności (a11y)**: od obsługi klawiatury w tabach, przez aria-pressed/label w toggle motywu, po respektowanie prefers-reduced-motion. Takie elementy często są pomijane, tutaj zaś zostały profesjonalnie zaimplementowane – świadczy to o wysokich kompetencjach autora.
- **Bezpieczeństwo front-end**: Kod nie ma oczywistych wektorów ataku, nie operuje na danych zewnętrznych bez walidacji. Stosowanie bezpiecznych metod manipulacji DOM (textContent, setAttribute z bezpiecznymi parametrami [3] [9]) świadczy o dobrych nawykach. W kontekście publicznego front-endu B2B jest to ważne – te skrypty na pewno nie stanowią słabego punktu aplikacji.
- **Gotowość do produkcji:** Z punktu widzenia audytora, kod wygląda na przetestowany w typowych scenariuszach. Ewentualne uwagi to głównie potencjalne usprawnienia, a nie konieczne poprawki. Nie stwierdzono błędów, które uniemożliwiałyby wdrożenie. Style CSS ani warstwa wizualna nie były przedmiotem oceny, ale czysto od strony JS – moduły powinny bez problemu działać w środowisku produkcyjnym.

Ogólna ocena: Pakiet tych trzech modułów oceniam jako **production-ready** – czyli w pełni gotowy do użycia na produkcji. Jakość kodu odpowiada poziomowi mid-senior+ (w niektórych aspektach wręcz seniorski standard). Świadczy o tym zarówno poprawność techniczna, jak i uwzględnienie aspektów poza samym „działaniem" (a11y, preferencje, edge-case'y). Podsumowując, projekt TransLogix w części front-endowej reprezentowanej przez te moduły stoi na solidnym fundamencie. Zalecam kontynuację tego podejścia w dalszym rozwoju – kod jest czytelny, łatwy w utrzymaniu i powinien służyć bezproblemowo użytkownikom końcowym. Wszystkie trzy moduły można śmiało określić jako **„senior-ready" i gotowe do produkcji**. [7] [14]

---

[1] [2] SCR40: Using the CSS prefers-reduced-motion query in JavaScript to prevent motion | WAI | W3C
https://www.w3.org/WAI/WCAG21/Techniques/client-side-script/SCR40

[3] [9] Cross Site Scripting Prevention - OWASP Cheat Sheet Series
https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

[4] [10] ARIA: tab role - ARIA | MDN
https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/Reference/Roles/tab_role

[5] [6] [7] [8] Tabs Pattern | APG | WAI | W3C
https://www.w3.org/WAI/ARIA/apg/patterns/tabs/

[11] [12] [13] [14] The best light/dark mode theme toggle in JavaScript - DEV Community
https://dev.to/whitep4nth3r/the-best-lightdark-mode-theme-toggle-in-javascript-368f