Plan testów (ISO/IEC/IEEE 29119)

# 1. Obiekt testów

**Shopmost** to platforma e-commerce oparta na GraphQL i React z podstawowymi funkcjami handlowymi. Zbudowany w React, modułowy i w pełni konfigurowalny. Jest projektem open source, a jego twórcy stawiają na transparentność w ciągu całego procesu jego wytwarzania.

# 2. Zakres testów

* **config** - należy sprawdzić, czy różne, czasem dziwnie zestawione ze sobą parametry konfiguracyjne nie powodują wywalenia apki lub wyświetlania jakichś dziwnych/niespójnych rzeczy.
* **module 'auth'** - w zasadzie cały ten moduł należałoby obłożyć testami, z racji tego, że jest on krytyczny dla bezpieczeństwa całej aplikacji, nawet drobne jego wady poddają w wątpliwości jakikolwiek sens używania aplikacji.
* **module 'base/api'** - ponieważ ten komponent odpowiada za poprawną obsługę interakcji przez API, należałoby dużym stopniu go wytestować na różne sposoby, jeżeli niepoprawnie obsługuje on jakiś rodzaj wyjątku, to może to w późniejszym etapie działania aplikacji doprowadzić do błędu, który będzie ekstremalnie ciężki do zlokalizowania.
* **module 'catalog'** - należy sprawdzić, czy wszystkie interakcje z bazą danych przebiegają prawidłowo jak również wyświetlanie odpowiednich rzeczy na stronie. Z tego modułu sensowne wydaje się przetestowanie komponentu 'pages', 'graphql/types', 'services', oraz 'subscribers', a także interakcji zachodzących pomiędzy nimi.
* **module 'promotion'** - również krytyczny, należy sprawdzić, czy nie ma żadnych bledow logicznych/matematycznych, a także sposobów, by nielegalnie naliczyć sobie jakas promocje. Przetestować cały moduł, a także interakcje jego różnych komponentów między sobą.
* **module 'customer'** - sprawdzić komponent api funkcjonalnie, czy usuwanie/dodawanie różnych customerów przebiega zawsze tak jak powinno w różnych scenariuszach, nie pojawia się usunięty klient, czy nie uda się dodać dwóch takich samych itp.
* **module 'cms'** - ewentualny do sprawdzenia.

3. Czynniki Ryzyka

## 3.1 Analiza Ryzyka

Przeprowadziliśmy analizę ryzyka, uwzględniając aktualną wiedzę, aby zidentyfikować potencjalne problemy, które mogą wpłynąć na projektowanie i wykonanie testów. Główne czynniki ryzyka to:

1. **Zaprzestanie rozwijania aplikacji w trakcie trwania semestru**: Jeśli rozwijanie aplikacji zostanie przerwane lub przeniesione do archiwum w trakcie trwania projektu testowego, może to prowadzić do braku dostępu do aktualnych źródeł kodu i dokumentacji, co utrudni testowanie i dostarczenie wyników testów.
2. **Brak przewidywania zmian w trakcie testów**: Jeśli zmiany w aplikacji nie są przewidywane i zarządzane, to może to prowadzić do problemów z utrzymaniem stabilności procesu testowego.
3. **Całkowita zmiana struktury i działania projektu wskutek nowej wersji**: Aktualizacje aplikacji, które znacząco zmieniają jej strukturę i działanie, mogą sprawić, że istniejące testy staną się przestarzałe lub nieprawidłowe. Konieczne jest szybkie dostosowanie testów do nowej wersji.
4. **Projekt zbyt skomplikowany do zlokalizowania defektu**: W przypadku bardzo skomplikowanych projektów testowanie może być wyjątkowo trudne, a lokalizacja defektów może być problematyczna, szczególnie w przypadku niewystarczającego doświadczenia zespołu testowego. Konieczne jest planowanie testów na podstawie odpowiednich strategii i technik testowania.
5. **Błędy w aplikacji**: Istniejące błędy w aplikacji mogą utrudnić pisanie testów i prowadzić do fałszywych wyników testów. Warto śledzić bazy błędów i weryfikować istniejące problemy.
6. **Niezrozumienie celu aplikacji**: Brak zrozumienia, jaki jest główny cel aplikacji i jakie są kluczowe funkcjonalności, może prowadzić do błędnych priorytetów w testach i pominięcia istotnych przypadków testowych.
7. **Niezrozumienie istniejącej architektury**: Brak zrozumienia istniejącej architektury aplikacji może utrudnić identyfikację punktów testowych. Konieczne jest zapoznanie się z architekturą aplikacji.
8. **Ograniczone środowisko testowe**: Brak dostępu do odpowiedniego środowiska testowego, które odzwierciedla produkcję, może prowadzić do problemów z testami wydajnościowymi i integracyjnymi. Warto zadbać o odpowiednie środowisko testowe.

## 3.2 Plan Zarządzania Ryzykiem

W celu efektywnego zarządzania ryzykiem związanym z testowaniem aplikacji zastosujemy następujące strategie:

1. **Monitorowanie**: Będziemy systematycznie monitorować stan projektu aplikacji i jej źródeł kodu oraz dokumentacji. To pozwoli nam na bieżąco reagować na ewentualne zmiany i dostosowywać nasze strategie testowe w miarę potrzeb.
2. **Dostosowywanie testów**: W przypadku pojawienia się zmian w aplikacji, dostosujemy nasze strategie testowe oraz przypadki testowe tak, aby uwzględniały nowy stan aplikacji i zachowały swoją efektywność.
3. **Kontrola jakości**: Regularnie będziemy weryfikować jakość naprawionych błędów i zmian w aplikacji, aby upewnić się, że testy dostarczają wiarygodne wyniki.

Dzięki tym strategiom będziemy gotowi na skuteczne zarządzanie ryzykiem i dostarczenie wysokiej jakości testów dla aplikacji.

# 4. Strategia Testowania

W sekcji poziomy testów zostaną wymienione poziomy testów wraz z ich oszacowanym ułamkiem całkowitego procesu testowego wyrażonym w procentach [%]. W sekcji techniki testów zostały wymienione poszczególne techniki testów oraz ich orientacyjne zastosowanie. Warto zwrócić uwagę, że większość z tych podpunktów nie jest samodzielną wyłączną techniką i typowy proces testowy może się składać z kilku z nich. Do każdej techniki została przyporządkowana etykieta ważności w systemie moscow.

## 4.1 Poziomy testów

1. **Testy systemowe (system tests)** - sprawdzają, czy moduły działają poprawnie między sobą i czy tworzą spójny system. [40%]
2. **Testy integracyjne (Integration tests)** - odbywają się na poziomie modułów. Sprawdzają, czy poszczególne moduły działają bezproblemowo. [25%]
3. **Testy jednostkowe (unit tests)** - na poziomie pojedynczej jednostki w kodzie (funkcji, klasy). W przypadku znalezienia defektu pomogą zlokalizować dokładne źródło problemu. [25%]
4. **Testy akceptacyjne (acceptance tests)** - sprawdzą, czy system spełnia wymagania biznesowe. [10%]

## 4.2 Techniki testów

1. **Analiza wartości granicznych (Boundary Value Analysis)** - za pomocą tej technik

sprawdzimy, czy system zachowuje się w oczekiwany sposób w przypadkach

granicznych (np. usunięcie przedostatniego elementu z katalogu, ustawienie ceny

produktu na 0 USD itp.). Z zasady takie problemy występują na granicy a nie w

centrum stąd szczególne przywiązanie do tej techniki. [must]

1. **Podział na klasy równoważności (Equivalence Class Partitioning)** - przy testowaniu

podzielimy pewne warunki testowe na klasy równoważności i będziemy testować

jedynie reprezentantów tych klas (np. ustalenie ceny produktu można podzielić na

(ujemna, bliska zeru, dodatnia)). Możemy wykonać takie uproszczenie, ponieważ

rezultat testu wartości w obrębie klasy, którą ustaliliśmy jest taki sam (np ujemną

cenę można wyeliminować warunkiem cena >= 0). [must]

1. **Zgadywanie (error guessing)** - w dużej mierze będziemy zgadywać błędy które

mogą wystąpić w testowanej aplikacji bazując na naszym doświadczeniu (z reguły

deweloperzy zapominają nałożyć górne ograniczenie np. na długość nazwy

produktu). Najbardziej niepowtarzalna technika, która na pewno się pojawi w procesie

testowym. [must]

1. **Testy bezpieczeństwa (security testing)** - sprawdzimy, czy system jest odporny na

ataki oraz nieautoryzowany dostęp (czy hasła są bezpiecznie przechowywane pod

postacią haszy, spróbujemy przeprowadzić ataki XSS, SQL injection, session

stealing itp). [must]

1. **Testy wytrzymałości/wydajności (load testing, performance testing)** - przetestujemy

czy system jest wydajny i czy zachowuje wydajność w przypadku dużych obciążeń

(np. gdy wielu użytkowników korzysta z portalu jednocześnie albo gdy baza danych

zawiera dużą ilość rekordów). [must]

1. **Testy bazujące na tabeli decyzyjnej (Decision Table Based Testing)** - użyjemy tej

techniki do funkcji, które przyjmują kombinację warunków (np ujemna cena produktu i

status ukryty). Ta technika zostanie zastosowana w szczególności w modułach

krytycznych dla działania całego systemu, czyli tam, gdzie jest wymagana duża

niezawodność. Tabela decyzyjna pomoże nam się upewnić, że sprawdziliśmy

wszystkie kombinacje warunków. [could]

1. **Testy przejścia pomiędzy stanami (State Transition)** - za pomocą tej techniki

sprawdzimy, czy testowana aplikacja zachowuje spójność w przejściach pomiędzy

wyznaczonymi stanami (np. dodanie produktu do użytkownika, który został usunięty).

[could]

1. **Testy funkcjonalne (Functional tests)** - sprawdzimy, czy aplikacja spełnia

wymagania biznesowe (np. czy dokumentacja jest spójna z faktycznym stanem

aplikacji). [could]

# 5. Zasoby

W kontekście naszego projektu testowego zidentyfikowaliśmy potrzebne zasoby, które są niezbędne do przeprowadzenia testów aplikacji. Obejmuje to kilka sekcji.

## 5.1 Ludzie

W naszym projekcie elastyczny podział ról został wybrany jako strategia dostosowana do stopniowego nabywania kolejnych umiejętności w toku realizowania przedmiotu Testowanie Oprogramowania. Ten model umożliwia każdemu członkowi zespołu rozwijanie się i zdobywanie doświadczenia w różnych obszarach, co może wpłynąć korzystnie na ich umiejętności i efektywność w projekcie.

## 5.2 LLMs (Duże Modele Językowe)

1. **Wsparcie w pisaniu przypadków testowych**: Duże modele językowe mogą być wykorzystane do generowania przypadków testowych na podstawie specyfikacji i wymagań aplikacji. Możemy dostarczyć modelowi ogólny opis funkcji i oczekiwań, a model może wygenerować konkretne przypadki testowe, co znacznie przyspieszy proces tworzenia testów.
2. **Wsparcie w pisaniu kodu**: Ponadto duże modele językowe mogą być wykorzystane do bezpośredniego generowania kodu, co może znacznie przyspieszyć proces tworzenia oprogramowania.
3. **Tworzenie dokumentacji testowej**: Możemy wykorzystać duże modele językowe do tworzenia dokumentacji testowej, w tym opisów przypadków testowych, scenariuszy testowych i opisów wyników testów. To pomoże w utrzymaniu spójnej dokumentacji i ułatwi przyszłe testy.

## 5.3 Narzędzia

W naszym projekcie testowym planujemy wykorzystać różne narzędzia, uwzględniając specyfikę projektu, który opiera się na Node.js. Oto lista narzędzi, które zostaną użyte w projekcie.

1. **Narzędzia do testów jednostkowych**: Do testów jednostkowych wykorzystamy narzędzia takie jak Jest lub Mocha w połączeniu z biblioteką do asercji, np. Chai. Pozwoli to na dokładne testowanie poszczególnych jednostek kodu.
2. **Narzędzia do testów integracyjnych/systemowych**: Dla testów integracyjnych i systemowych wykorzystamy narzędzia, które umożliwią testowanie interakcji między różnymi komponentami naszej aplikacji. Cypress może być przydatny do testów E2E (end-to-end).
3. **Narzędzia do testów wydajnościowych/obciążeniowych**: Jeśli planujemy testy wydajnościowe, obciążeniowe lub stresowe, możemy użyć narzędzi takich jak Apache JMeter lub Artillery do symulowania dużego obciążenia na naszej aplikacji.
4. **Edytor kodu źródłowego**: Do pisania kodu testowego i jego zarządzania potrzebujemy edytora kodu źródłowego. Popularnymi edytorami w ekosystemie Node.js są Visual Studio Code, WebStorm lub Atom.
5. **Narzędzia do wirtualizacji i konteneryzacji**: Wykorzystamy narzędzie Docker do wirtualizacji i konteneryzacji, co umożliwi nam uruchamianie i zarządzanie kontenerem z bazą danych w izolowanym środowisku. To pozwoli na testowanie aplikacji z różnymi konfiguracjami bazy danych.
6. **Narzędzia do zarządzania błędami**: Do monitorowania i śledzenia błędów wykorzystamy narzędzia takie jak Jira. Dzięki Jira będziemy w stanie efektywnie zarządzać zgłoszeniami błędów, śledzić postęp testów oraz komunikować się w ramach zespołu.

## 5.4 Sprzęt

W ramach naszego projektu testowego wykorzystamy standardowe komputery osobiste. Planujemy ograniczyć nasze potrzeby sprzętowe do minimum, aby zminimalizować koszty i uprościć zarządzanie środowiskiem testowym. Komputer będzie głównym narzędziem do pisania kodu testowego, uruchamiania testów i analizy wyników. Standardowy komputer lub laptop jest wystarczający do efektywnego przeprowadzenia testów w naszym projekcie.

# 6. Harmonogram

1. **Plan testów** - Termin: 03-11-2023
2. **Testy systemowe** - Termin: 10-11-2023
3. **Testy integracyjne** - Termin: 17-11-2023
4. **Testy jednostkowe** - Termin: 24-11-2023
5. **Testy wydajnościowe** - Termin: 1-12-2023
6. **Raporty z testów** - Termin: 8-12-2023

# 7. Kryteria

## 7.1 Kryteria wejścia

1. **Warunek przygotowania środowiska testowego:** Zanim rozpocznie się każdą czynność testową, środowisko testowe musi być odpowiednio skonfigurowane i przygotowane, włącznie z odpowiednią wersją oprogramowania, danymi testowymi i dostępem do niezbędnych zasobów.

2. **Dostępność dokumentacji testowej:** Przed rozpoczęciem testów, muszą być dostępne wszystkie niezbędne dokumenty, takie jak plan testów i dane testowe.

3. **Przygotowanie testera:** Tester lub zespół testowy musi być odpowiednio przygotowany, co obejmuje zrozumienie działania obiektu testów, zapoznanie się z celami testów i dostępność niezbędnych narzędzi testowych

## 7.2 Kryteria zakończenia

1.  **Spełnienie warunków sukcesu:** Każda czynność testowa musi być uznana za zakończoną, jeśli wszystkie ustalone kryteria sukcesu, takie jak ilość znalezionych błędów, zgodność z wymaganiami lub poziom pokrycia testami, są osiągnięte.

2. **Dokumentacja wyników:** Po zakończeniu testów, wszystkie wyniki muszą być odpowiednio udokumentowane i opisane, włącznie z informacjami o znalezionych błędach, ich priorytetach i statusach.

## 7.3 Kryteria wstrzymania i wznowienia

1. **Wstrzymanie testów:** Testy mogą zostać wstrzymane w przypadku poważnych problemów ze środowiskiem testowym, narzędziami testowymi lub dostępem do zasobów. Wstrzymanie może również zajść, jeśli identyfikowane są poważne błędy, które wymagają natychmiastowej naprawy.

2. **Warunki wznowienia:** Po wstrzymaniu testów, mogą być wznowione, gdy wszystkie problemy, które spowodowały wstrzymanie, zostaną rozwiązane, a środowisko testowe będzie ponownie gotowe do użycia. Ponadto zespół testowy musi upewnić się, że wszystkie dokumenty i zasoby są nadal dostępne i aktualne.

# 8. Procedury zarządzania

1. **Komunikacja** naszego zespołu opierać się będzie na cotygodniowych spotkaniach w postaci audio-konferencji w celu planowania prac na następny tydzień oraz wiadomościach tekstowych. To wszystko odbywać się będzie przez platformę Discord
2. **Zarządzanie i kontrola projektem** będzie odbywać za pomocą Jiry oraz cotygodniowych spotkań
3. **Procedura przeglądu przypadków testowych** będzie się opierać na ich analizie przez wszystkich członków zespołu w przypadku gdy jest on nowy. Po tym etapie przegląd będzie odbywać się ad hoc przez dowolnego członka zespołu.
4. **Warunkiem akceptacji przypadków testowych** jest brak uwag ze strony zespołu w trakcie procesu przeglądu. Gdy zostaną znalezione braki lub błędu w przypadku testowym zostanie to udokumentowane i przydzielone do naprawy dla odpowiedniego członka zespołu.
5. **Raport z testów** będzie generowany po zakończeniu etapu testowania. Będzie zawierał takie informacje jak nagłówek, wyniki testów, metodologię testów i wnioski