POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Michał Sztanderski
Nr albumu s25553

Różne podejścia do tworzenia klienckich aplikacji przeglądarkowych

Praca magisterska napisana pod kierunkiem:

dr inż. Mariusz Trzaska

Warszawa, luty, 2025

Streszczenie

Niniejsza praca dotyczy praktycznego problemu wyboru platformy programistycznej na początkowym etapie tworzenia aplikacji przeglądarkowej. Głównym celem było porównanie dwóch technologii: Angular i Blazor, na podstawie stworzonych w ramach badań aplikacji przeglądarkowych. Praca obejmuje również porównanie z innymi wiodącymi platformami ogólnodostępnymi na rynku.

Wyniki testów zostały przedstawione w rozdziale szóstym, wraz ze szczegółami dotyczącymi: szybkości wytwarzania oprogramowania, prędkości wyświetlenia danych, zużycia pamięci i procesora, kosztów licencji i stabilności.

Zakończeniem pracy jest podsumowanie zawierające opisanie wad i zalet obu aplikacji wytworzonych w zaproponowanych technologiach oraz ogólnymi wnioskami jakie można było wyciągnąć z przeprowadzonych badań.

Słowa kluczowe: Aplikacja przeglądarkowa, Blazor, Angular

Spis treści

1. Wstęp

W ostatniej dekadzie rynek dotyczący programowania osiągnął lawinowy wzrost. Pojawienie się internetu oraz rozwój technologii, w tym zwiększenie użytkowników komputerów i smartfonów zmieniło postrzeganie świata. Coraz to więcej firm rozpoczęło ulepszać swoją infrastrukturę poprzez wytwarzanie dedykowanych aplikacji. Co więcej one same stały się znaczącym towarem na rynku. Świat programowania rozwija się w bardzo szybkim tempem oferując coraz to nowsze programy i sposoby mające na celu usprawnienie pracy. Również pojawienie się w ostatnim czasie technologii AI (ang. *Artificial Intelligence*) jest czynnikiem, który prognozuje się, że w znaczący sposób wpłynie na całą branżę IT (ang. *Information Technology*). Jednym z nowych trendów związanych ze sztuczną inteligencją, jest automatyzacja pisania kodu. Nie jest, więc zaskoczeniem, że każdego dnia programistom oferowane jest wiele możliwości i podejść do tworzenia aplikacji przeglądarkowych, co może wpływać na tworzenie się zjawiska paradoksu wyboru [1]. Obecnie jedną z popularnych technologii do tworzenia aplikacji przeglądarkowych jest Angular, rozwijany przez Google [3]. Blazor, opracowany przez Microsoft, nie zdobył jeszcze tak dużej popularności, ale oferuje innowacyjne podejście, szczególnie dzięki możliwości budowania interfejsów użytkownika przy użyciu języka C# [25] [2]. Pomimo podobnego celu obydwu przedstawionych narzędzi, posiadają one znaczące różnice.

By odpowiednio zrozumieć na czym opiera się praca we wstępie znajduje się krótki opis obu rozwiązań.

Angular to jeden z najpopularniejszych frameworków na rynku, co wynika z jego szerokiego zastosowania i starannego opracowania przez wielu autorów [25]. Wyróżnia się otwartoźródłową licencją, wsparciem dla obiektowego języka TypeScript, dwukierunkowym wiązaniem danych oraz modułową architekturą. Warty uwagi jest również wbudowany router, pozwalający na nawigowanie po widokach bez konieczności przeładowywania całej strony.

Blazor natomiast to platforma, która bazuje na używaniu tylko jednego języka obiektowego C#, do tworzenia kompleksowych aplikacji przeglądarkowych. Dzięki wbudowanemu silnikowi Razor, pozwala na tworzenie dynamicznych widoków z wykorzystaniem języków HTML (ang. *HyperText Markup Language*) i C#. Dostarcza dwa modele hostingu. Blazor Server, gdzie logika aplikacji działa na serwerze oraz Blazor WebAssembly, gdzie cała aplikacja działa w przeglądarce.

Niniejsza praca jest poświęcona porównaniu tych dwóch popularnych rozwiązań, na podstawie aktualnego stanu wiedzy. Przeprowadzone badania mają na celu pokazanie kluczowych różnic pomiędzy wyżej wymienionymi platformami, w zestawieniu do pozostałych dominujących i ogólnodostępnych systemów.

1.1. Cel pracy

Biorąc pod uwagę znaczenie wyboru odpowiedniej technologii, spełniającej wymagania programistów, celem niniejszej pracy jest wsparcie w podjęciu decyzji o wyborze najlepszego rozwiązania. Ze względu na ogromną liczbę dostępnych platform do tworzenia aplikacji przeglądarkowych, praca koncentruje się jedynie na kilku najpopularniejszych z nich, w szczególności na Blazorze i Angularze. Drugorzędnym celem pracy jest analiza technologii Blazor, która, choć rzadziej jest stosowana w środowiskach produkcyjnych, wyróżnia się swoją innowacyjnością i potencjałem. Pozytywna ocena tej technologii może zachęcić czytelników do jej zastosowania w biznesie.

1.2. Rozwiązanie przyjęte w pracy

Aby odpowiedzieć na postawione pytanie badawcze, przeprowadzono testy dwóch popularnych platform programistycznych: Blazor [2] i Angular [3]. Platformy te komunikują się z usługami stworzonymi w ASP.NET Core Web API [4], które obsługują dane z bazy Microsoft SQL Server [5]. Testy przeprowadzono za pomocą biblioteki Playwright [6], umożliwiającej symulację zachowań użytkownika oraz narzędzia Artillery [7], które pozwala na równoczesne uruchomienie wielu scenariuszy testowych. Takie podejście umożliwiło ocenę wydajności obu platform w kontekście liczby równoczesnych użytkowników, dostarczając szczegółowych danych na potrzeby analizy. Alternatywne metody, takie jak dedykowane narzędzia do testów obciążeniowych, odrzucono z powodu ich większej złożoności i mniejszej elastyczności. Wszystkie projekty stworzono w środowisku Visual Studio, co zapewniło spójność i wygodę realizacji pracy.

1.3. Rezultaty pracy

Wynikiem przeprowadzonych badań będzie stworzenie szczegółowego zestawienia wyników testów dwóch technologii: Angular i Blazor. Aby jak najlepiej odpowiedzieć na temat pracy oraz dostarczyć szczegółową wiedzę, analiza obejmie następujące aspekty:

- Szybkość wytwarzania oprogramowania - jest to kluczowy aspekt, ponieważ wpływa na czas realizacji projektów i efektywność zespołu programistycznego. Badanie to pozwoli ocenić, która technologia umożliwia szybsze tworzenie aplikacji.
- Wydajność - ten czynnik ma bezpośredni wpływ na doświadczenia użytkownika końcowego. Porównanie szybkości renderowania danych pozwoli określić, która z technologii lepiej radzi sobie w przypadku obciążenia dużymi zbiorami danych.
- Poziom trudności implementacji - pomoże określić, która technologia jest bardziej przystępna dla programistów o różnym poziomie doświadczenia.
- Zużycie pamięci podręcznej- pozwoli wskazać, która technologia jest bardziej wydajna w zarządzaniu pamięcią.
- Zużycie procesora - wskaże, jak obie technologie obciążają urządzenie użytkownika podczas intensywnych operacji.
- Koszty licencji - która technologia jest bardziej opłacalna w długoterminowym użytkowaniu.
- Stabilność - testy w tym zakresie pozwolą ocenić, jak obie technologie radzą sobie w trudnych warunkach, takich jak duże obciążenie.

1.4. Organizacja pracy

Praca została podzielona na następujące części i rozdziały.

W pierwszej części pracy znajduje się wstęp, który wyjaśnia powody wyboru tematu pracy, wprowadza szczegółowo w podjętą problematykę. Przedstawia również jaki jest cel przeprowadzonych testów, w jak przebiegają badania oraz oczekiwane rezultaty.

Drugi rozdział pracy pod tytułem “Istniejące rozwiązania” omawia szczegółowo rozwiązania istniejące na rynku wraz z wymienieniem ich mocnych i słabych stron.

Trzeci rozdział pracy skupia się wokół technologii zastosowanych w pracy, czyli głównie Angularze i Blazorze. Opisuje również inne narzędzia, które okazały się niezbędne do stworzenia całościowego projektu.

Czwarty rozdział jest poświęcony stworzonemu na potrzeby pracy projektowi. Omawia z jakich modułów się składa oraz tłumaczy kod projektu.

Piąty rozdział pracy powstał w celu przestawienia scenariuszy testowych, które zostały skonstruowane w celu weryfikacji wydajności i wyników przeprowadzonych testów.

Szósty rozdział porównuje obie platformy Angular i Blazor, na podstawie scenariuszów opisanych wcześniej z rozdziału piątego, jak i wiedzy zebranej w celu stworzenia przykładowego projektu.

Ostatni rozdział jest podsumowaniem całej pracy. Opisuje wnioski wyciągnięte z przeprowadzonych testów i porównań technologii.

2. Istniejące rozwiązania

Rozdział ten opisuje istniejące rozwiązania dostępne na rynku, dotyczące aplikacji przeglądarkowych. Zostały one wybrane ze względu na swoją popularność i ciekawe podejście. Przedstawione zostały ich główne cechy, mocne strony, ogólne działanie i daty publikacji.

2.1. React

Pierwszym rozwiązaniem jakie zostało opisane w pracy jest React [8]. To opracowana przez Facebooka biblioteka JavaScript służąca do budowania dynamicznych aplikacji przeglądarkowych. Według danych opublikowanych przez stronę github.com pierwsza wersja powstała już w lipcu 2013 roku. Głównymi cechami, które wyróżniają tą technologię są:

- Deklaratywne podejście – Programiści definiują, jak interfejs powinien wyglądać w danym stanie, a React automatycznie zajmuje się aktualizacją widoku, co upraszcza tworzenie i utrzymanie kodu [8].
- Konstrukcja oparta na komponentach – Aplikacje składają się z modularnych i wielokrotnego użytku jednostek, co zwiększa przejrzystość i ułatwia rozwój [8].
- Wirtualny DOM – React przeprowadza zmiany w wirtualnej wersji DOM (ang. *Document Object Model*), a następnie efektywnie aktualizuje rzeczywisty DOM, co zwiększa wydajność aplikacji [8].
- Jednokierunkowy przepływ danych – Dane przepływają w jednym kierunku, co ułatwia śledzenie zmian w aplikacji i redukuje błędy [8].
- Lekka konstrukcja – React jest biblioteką, a nie pełnym frameworkiem, co oznacza, że dostarcza narzędzi przede wszystkim do budowy interfejsów użytkownika. Dzięki temu jest lekki i można go łatwo integrować z innymi technologiami.

W porównaniu do pierwszej wersji aplikacji powstałej w 2013 roku, obecnie React przeszedł wiele ewolucji i został mocniej rozbudowany [9]. Z powodu swoich zalet, jest używany w wielu firmach takich jak Facebook, Instagram, Airbnb, Twitter czy Pinterest.

2.2. Vue.js

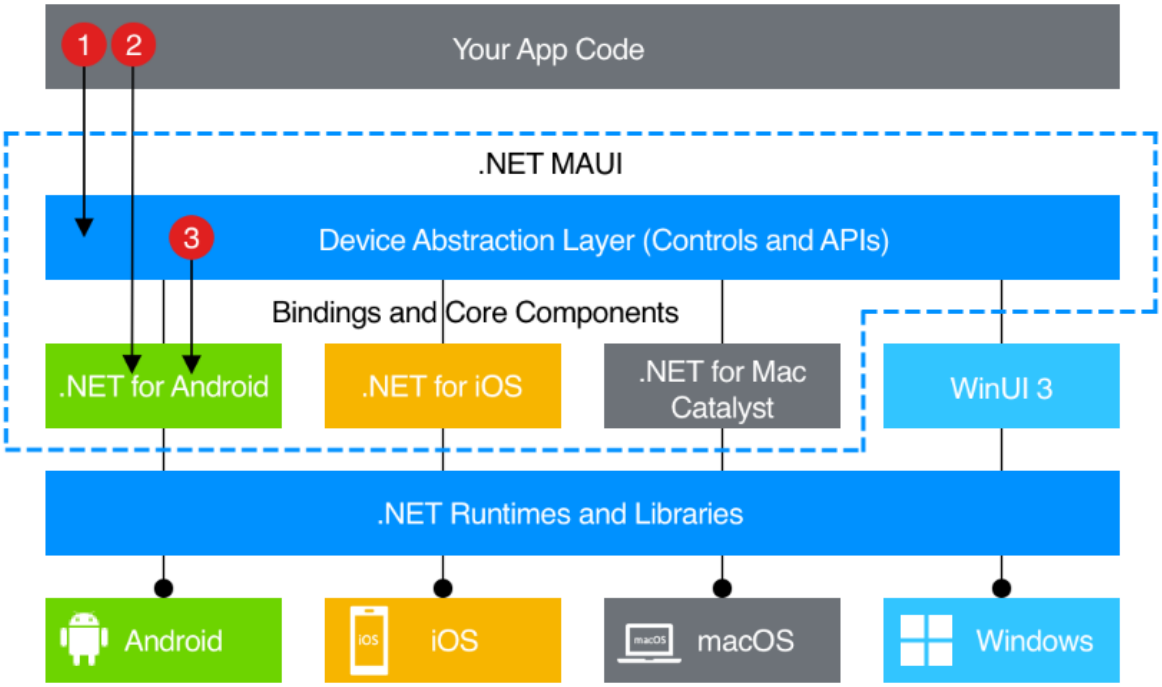
Kolejnym rozwiązaniem przytoczonym w pracy jest Vue.js [10]. Jest to otwartoźródłowa platforma programistyczna oparta na JavaScript, służąca do budowania interfejsów użytkownika. Technologia ta jest ceniona przez deweloperów ze względu na swoją prostotę i wysoką elastyczność. Twórcą Vue.js jest Evan You, który, po zakończeniu pracy w Google, w wywiadzie powiedział: „Pomyślałem, co by było, gdybym mógł po prostu wyodrębnić część, która naprawdę mi się podobała w Angularze, i stworzyć coś naprawdę lekkiego” [11]. Prace nad projektem rozpoczęły się w czerwcu 2013 roku, a pierwsze wydanie miało miejsce w lutym 2014 roku, czyli siedem miesięcy później.

Pierwszą z dwóch głównych cech tej platformy jest deklaratywne renderowanie. Jest to podejście w programowaniu, w którym Vue rozszerza standardowy HTML o składnię szablonów, umożliwiając deklaratywne opisywanie wynikowego kodu HTML na podstawie stanu JavaScript. Drugą ważną cechą tego rozwiązania jest reaktywność. Polega ona na automatycznym śledzeniu zmian w stanie JavaScript oraz optymalnej aktualizacji DOM w momencie, gdy zmiany te zachodzą [10].

2.3. .NET MAUI

.NET Multi-platform App UI (.NET MAUI) to platforma umożliwiająca tworzenie natywnych aplikacji mobilnych i klasycznych z wykorzystaniem języków C# i XAML [12]. Jej główną cechą jest zdolność tworzenia aplikacji na platformy Android, iOS, macOS i Windows przy użyciu jednej, wspólnej bazy kodu [13]. Rozwiązanie to okazuje się szczególnie skuteczne w sytuacjach, gdy konieczne jest dostarczenie produktu dostępnego na różnych urządzeniach. Dzięki współdzieleniu kodu programista może znacznie szybciej stworzyć wymagany program, ponieważ nie musi osobno projektować aplikacji dla każdej platformy. Każdy taki projekt wymagałby bowiem odrębnych narzędzi, języków programowania, specjalistycznej wiedzy i unikalnych umiejętności.

.NET MAUI jest rozwinięciem platformy Xamarin, które zostało opublikowane 23 maja 2022 roku [14]. Jak większość współczesnych technologii, jest ona otwartoźródłowa, co daje użytkownikowi możliwość modyfikowania, analizowania, kopiowania i rozpowszechniania jej kodu. Na rysunku 1 przedstawiono ogólny widok architektury aplikacji .NET MAUI. W aplikacji .NET MAUI kod źródłowy jest tworzony w sposób umożliwiający współpracę przede wszystkim z kontrolkami .NET MAUI oraz warstwą API (1 na rysunku 1). Warstwa ta korzysta bezpośrednio z natywnych interfejsów platformy (3 na rysunku 1). Ponadto, w razie potrzeby, kod aplikacji może bezpośrednio odwoływać się do interfejsów API platformy (2 na rysunku 1), zapewniając większą elastyczność w implementacji funkcjonalności [13]. Warto zauważyć, że mimo ujednolicenia kodu, .NET MAUI pozwala na modyfikację wyglądu lub funkcjonalności aplikacji na wybranej platformie bez wpływu na pozostałe. Dzięki temu programista może dostosować aplikację do specyficznych wymagań, zamiast tworzyć odrębny projekt dla każdej platformy.



Rysunek 1. Ogólny widok architektury aplikacji .NET MAUI od microsoft.com. Źródło: [13]

2.4. Svelte

Svelte [15] to kompilator TypeScript służący do tworzenia aplikacji przeglądarkowych. Został wydany 29 listopada 2016 roku przez Richa Harrisa. Kompilator jest darmowy i dostępny w ramach licencji MIT (ang. *Massachusetts Institute of Technology*). Pierwsza wersja została stworzona przy użyciu JavaScriptu, jednak w 2019 roku zaktualizowano go do wersji opartej na TypeScript.

W porównaniu z technologiami takimi jak Vue.js, Svelte nie wykorzystuje wirtualnego DOM do aktualizacji danych na stronie internetowej. Zamiast tego kompiluje kod bezpośrednio do JavaScriptu, co skutkuje mniejszym rozmiarem projektu. Dodatkowo, dzięki kompilacji, nie ma potrzeby ładowania dodatkowych bibliotek do przeglądarki, ponieważ całość znajduje się w uprzednio skompilowanym kodzie. Podejście polegające na kompilacji kodu w czasie budowania aplikacji przekłada się na szybsze działanie, co sprawia, że aplikacje są bardziej wydajne [16].

Jednym z minusów Svelte jest mniejszy ekosystem i społeczność, co może utrudniać znalezienie wsparcia lub gotowych rozwiązań. Kolejną wadą jest brak wsparcia ze strony dużych firm, co potencjalnie negatywnie wpływa na rozwój i przyszłość tego kompilatora.

3. Zastosowane technologie

Trzecia część pracy przedstawia technologie jakie zostały zastosowane do stworzenia przykładowego projektu.

3.1. Blazor

Pierwszą z dwóch głównych aplikacji, na których skupia się praca, wykonano przy pomocy Blazora. Jest to platforma służąca do budowania aplikacji przeglądarkowych w językach C# i HTML. Pierwsza wersja została opublikowana w 2016 roku przez Microsoft jako część ASP.NET Core Web App. Głównym celem stworzenia Blazora było opracowanie metody tworzenia aplikacji przeglądarkowych wyłącznie za pomocą C#, gdy w większości popularnych platform wykorzystuje się JavaScript jako główny język programowania [25]. Takie rozwiązanie obniża próg wejścia, ponieważ osoba chcąca stworzyć aplikację nie musi uczyć się wielu języków programowania. Wystarczy jej znajomość jednego uniwersalnego języka, jakim jest C#, który należy do najbardziej popularnych języków na świecie [17].

Fundamentem Blazora jest technologia Razor. Razor to silnik szablonów wykorzystujący HTML i C#, który umożliwia płynne przechodzenie między kodem HTML a C# w celu definiowania logiki renderowania komponentów [18]. Razor ogranicza się do dwóch wymienionych języków, dzięki czemu jest bardzo intuicyjny i łatwy do nauki. Umożliwia tworzenie dowolnych i dynamicznych widoków, gdzie można na przykład używać pętli znanych z C# do powielania wybranych obszarów HTML (listing 1).

Listing 1. Pętla for w Razor. Źródło: [2]

```
@for (var i = 0; i < people.Length; i++)
{
        var person = people[i];
        <p>Name: @person.Name</p>
        <p>Age: @person.Age</p>
}
```

Wszystkie technologie .NET historycznie miały jedną cechę wspólną: renderowanie odbywało się po stronie serwera. Przeglądarka wysyłała zapytanie do serwera, gdzie uruchamiany był kod, który zwracał odpowiedź w formie gotowego widoku do wyświetlenia. Cała praca wykonywana była po stronie serwera. Serwer musiał zapewniać generowanie interfejsu użytkownika, uruchamianie logiki biznesowej oraz zarządzanie stanem aplikacji [18]. Aby stworzyć kod, który działałby po stronie klienta, programista najczęściej posługiwał się językiem JavaScript [30]. Do niedawna JavaScript mógł wydawać się niezbędny do tworzenia nowoczesnych aplikacji przeglądarkowych. Sytuacja ta zmieniła się w 2015 roku, kiedy zaprezentowano nowe rozwiązanie — WebAssembly. Umożliwia ono kompilowanie kodu, w tym C#, do postaci binarnej, którą można uruchamiać bezpośrednio w przeglądarkach. Dzięki temu zamiast tradycyjnych skryptów JavaScript możliwe jest wykonywanie kodu niskopoziomowego, co zapewnia wyższą wydajność i szybsze ładowanie aplikacji [2].

Blazor obsługuje zarówno renderowanie po stronie serwera, jak i klienta. Programista może samodzielnie zdecydować, które rozwiązanie chce wykorzystać, a także skorzystać z hybrydy obu sposobów. Obecnie wszystkie popularne przeglądarki internetowe wspierają WebAssembly, które stało się czwartym językiem natywnie obsługiwanym w przeglądarkach (obok HTML, CSS i JavaScriptu) [31].

3.2. Angular

Drugą aplikacją, na której skupia się praca, wykonano przy użyciu Angulara [3]. Platforma ta umożliwia programistom tworzenie szybkich i niezawodnych aplikacji przeglądarkowych. Została stworzona przy użyciu języka TypeScript w maju 2016 roku przez Google. Początkowo miała być nową wersją technologii AngularJS, jednak ze względu na problemy z wsteczną kompatybilnością Google zdecydowało się wydać Angular jako odrębną technologię. AngularJS jest już wersją przestarzałą – wsparcie dla tej platformy zakończyło się w 2022 roku [19].

Za pomocą Angulara można tworzyć zarówno wielostronicowe aplikacje internetowe MPA (ang. *Multi-Page Application*), jak i jednostronicowe aplikacje internetowe SPA (ang. *Single-Page Application*). MPA nie są obecnie popularnym rozwiązaniem ze względu na słabą wydajność. Architektura ta wymaga każdorazowego przeładowania całej strony internetowej. SPA natomiast ładuje stronę jednorazowo, a następnie dynamicznie uzupełnia jej zawartość po stronie przeglądarki za pomocą JavaScript. Z tego powodu programiści zwykle wybierają architekturę SPA do tworzenia aplikacji przeglądarkowych [32].

Jedną z głównych zalet Angulara jest wykorzystanie języka TypeScript do tworzenia aplikacji. Dzięki niemu można stworzyć bezpieczny i zoptymalizowany kod. Język ten jest nadzbiorem JavaScript, co oznacza, że każdy kod napisany w JavaScript jest również poprawnym kodem TypeScript. TypeScript umożliwia statyczne typowanie, co pozwala na wykrywanie błędów na etapie kompilacji, a nie dopiero podczas działania aplikacji. Angular oferuje także kompletny zestaw narzędzi do tworzenia aplikacji przeglądarkowych, w tym wbudowane mechanizmy *routingu*, zarządzania formularzami, walidacji, zarządzania stanem aplikacji (ngRx) oraz obsługi protokołu HTTP [3].

Dodatkową zaletą Angulara jest polityka wsparcia długoterminowego LTS (ang. *Long-Term Support*) realizowana przez Google [3]. Oznacza to, że jedna z największych firm technologicznych stale inwestuje w rozwój i udoskonalanie tej platformy.

3.3. ASP.NET Core Web API

ASP.NET Core Web API [4] to framework opracowany przez Microsoft, będący częścią większej platformy ASP.NET Core. Umożliwia tworzenie nowoczesnych, skalowalnych aplikacji internetowych, mobilnych oraz usług backendowych. ASP.NET Core Web API pozwala na budowanie usług przeglądarkowych (ang. *web services*) i interfejsów API (ang. *Application Programming Interface*) opartych na architekturze REST [20]. REST to styl architektoniczny wykorzystywany do budowy usług sieciowych, w których protokół HTTP służy do komunikacji. Opisywany framework charakteryzuje się kilkoma kluczowymi założeniami:

- Rozdzielenie warstwy klienckiej (aplikacja przeglądarkowa) od serwera (backendu). Dzięki temu obie warstwy mogą być rozwijane niezależnie, co poprawia skalowalność.
- Serwer nie przechowuje sesji ani informacji o kliencie pomiędzy żądaniami. Taka architektura ułatwia utrzymanie usługi, zmniejsza obciążenie serwera i poprawia jego skalowalność.
- Zasoby są przekazywane między klientem a serwerem w formacie JSON lub XML.
- Wykorzystanie standardowych metod HTTP do operacji na zasobach (GET, DELETE, PUT, POST).
- Zwracanie kodów statusu HTTP do informowania klienta o wyniku żądania, np. 200 (OK), 400 (Bad Request).

Typowym zastosowaniem ASP.NET Core Web API jest tworzenie warstwy serwerowej dla aplikacji przeglądarkowych. Framework ten działa na różnych systemach operacyjnych, takich jak

Windows, macOS i Linux, co umożliwia programistom tworzenie aplikacji i usług kompatybilnych z różnymi platformami [4].

ASP.NET Core Web API korzysta z *middleware* do obsługi żądań HTTP [4]. *Middleware* to komponenty, które mogą przetwarzać i modyfikować żądania oraz odpowiedzi przed ich dotarciem do docelowego kontrolera API lub po jego przetworzeniu. Gotowe rozwiązania tego typu są bardzo przydatne dla programistów, oszczędzając czas i redukując ryzyko błędów. Dzięki temu, że ASP.NET Core Web API zostało stworzone przez Microsoft, framework łatwo integruje się z innymi technologiami i narzędziami tego ekosystemu, takimi jak Azure, Entity Framework Core (do obsługi baz danych), SignalR (do komunikacji w czasie rzeczywistym) oraz Visual Studio.

3.4. Microsoft SQL Server

Microsoft SQL Server [5] to relacyjny system zarządzania bazą danych (RDBMS) opracowany przez firmę Microsoft. Opiera się na języku zapytań Transact-SQL (T-SQL), który umożliwia modyfikowanie, dodawanie i usuwanie danych w bazie danych w sposób intuicyjny i wydajny. Pierwsza wersja SQL Server pojawiła się w 1989 roku i została stworzona przez Microsoft dla systemu operacyjnego OS/2.

Microsoft SQL Server wykorzystuje relacyjny model danych, w którym dane są przechowywane w tabelach, a te mogą być powiązane za pomocą kluczy głównych i obcych. Dzięki temu możliwe jest tworzenie zaawansowanych zapytań za pomocą T-SQL, które mogą łączyć dane z różnych tabel. System oferuje także narzędzia do optymalizacji wydajności, takie jak indeksowanie, pamięć podręczna zapytań, partycjonowanie tabel oraz zaawansowane zarządzanie pamięcią [5].

SQL Server jest zdolny do przetwarzania dużych ilości danych i obsługi skomplikowanych zapytań, co czyni go popularnym wyborem dla dużych korporacji. Jednocześnie jest elastyczny i znajduje zastosowanie także w mniejszych projektach. Microsoft oferuje różne wersje SQL Server, dostosowane do specyficznych potrzeb użytkowników [5]. Przykładowo:

- Express: Darmowa wersja, odpowiednia dla małych aplikacji oraz do celów edukacyjnych.
- Enterprise: Zaawansowana wersja, idealna do dużych projektów. Oferuje dodatkowe funkcjonalności, takie jak zaawansowane opcje bezpieczeństwa, skalowalność i mechanizmy wysokiej dostępności.

Dzięki szerokiej gamie funkcji i wszechstronności, Microsoft SQL Server znajduje zastosowanie w różnych środowiskach — od małych aplikacji po zaawansowane systemy dla globalnych przedsiębiorstw.

3.5. Playwright

Playwright [6] to platforma programistyczna służąca do tworzenia testów *end-to-end* dla aplikacji przeglądarkowych. Testy te sprawdzają cały proces działania aplikacji, symulując rzeczywiste scenariusze wykonywane przez użytkowników. Dzięki temu możliwe jest upewnienie się, że wszystkie komponenty aplikacji współpracują prawidłowo.

Prace nad Playwrightem rozpoczął Microsoft w styczniu 2020 roku [21]. Pierwsze wersje tej technologii nie były wystarczająco stabilne i brakowało im wielu funkcjonalności, co ograniczało ich zastosowanie w projektach produkcyjnych. Z najnowszą wersją v1.47 Playwright stał się popularną technologią w automatyzacji testów. Platforma ta wspiera wieloplatformowe testowanie, w tym testy na urządzeniach mobilnych, a także umożliwia jednoczesną obsługę wielu instancji przeglądarek. Playwright oferuje wsparcie dla kilku języków programowania, takich jak JavaScript, TypeScript, Python, C# i Java [6]. Dzięki temu jest dostępny dla szerokiego grona programistów, co obniża próg

wejścia do korzystania z tej technologii. Dodatkowo Playwright umożliwia automatyczne wykonywanie zrzutów ekranu oraz nagrywanie wideo z sesji testowych. Te funkcjonalności sprawiają, że narzędzie to jest niezwykle atrakcyjne dla zespołów QA (ang. *Quality Assurance*).

3.6. Artillery

Artillery [7] to narzędzie służące do przeprowadzania testów obciążeniowych. Zostało stworzone w 2015 roku w języku JavaScript jako oprogramowanie otwartoźródłowe. Umożliwia testowanie wydajności i obciążenia aplikacji przeglądarkowych, dzięki czemu pozwala ocenić, jak system działa przy dużym natężeniu ruchu. Podstawową ideą tego narzędzia jest symulacja dużej liczby użytkowników korzystających z aplikacji przeglądarkowej.

Aby utworzyć testy obciążeniowe, należy przygotować plik w języku YAML [22]. YAML to czytelny język serializacji danych, który pozwala na prostą i przejrzystą konfigurację testów. W celu ich uruchomienia należy skorzystać z komendy Artillery, wskazując w parametrach wejściowych wcześniej przygotowany skrypt.

4. Projekty i ich implementacja

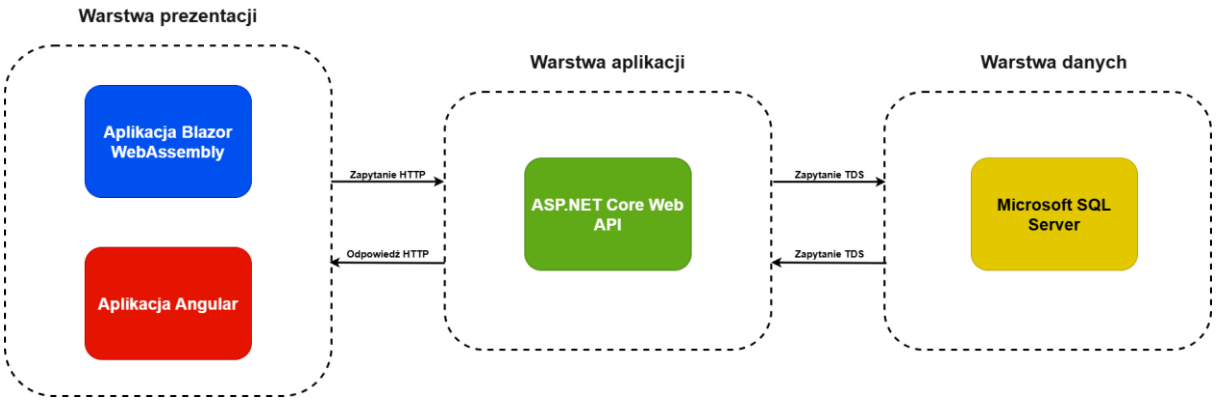
Rozdział opisuje projekty stworzone w ramach niniejszej pracy. Przedstawia schematy, zależności oraz implementację klas i modułów zaimplementowanych w celu porównania dwóch platform służących do tworzenia aplikacji przeglądarkowych. Nie zawiera pełnego kodu, lecz koncentruje się na fragmentach kluczowych, które nie są generyczne ani automatycznie wygenerowane przez technologie uzupełniania kodu.

4.1. Architektura projektu

Architektura projektu to sposób organizacji elementów aplikacji, takich jak moduły, klasy, zależności i przepływ danych, w celu osiągnięcia określonych celów, takich jak skalowalność, łatwość utrzymania oraz wydajność. Definiuje ona strukturę systemu, obejmując zarówno jego elementy składowe, jak i zasady oraz wytyczne, które wpływają na sposób, w jaki te elementy współpracują. Architektura jest kluczowym aspektem w procesie tworzenia oprogramowania, ponieważ odgrywa istotną rolę w zarządzaniu złożonością systemu, pozwalając na jego efektywny rozwój i adaptację do zmieniających się wymagań [23]. Architektura niniejszego projektu opiera się na trzech kluczowych elementach:

- Aplikacja przeglądarkowa,
- API,
- Baza danych

Opisując bardziej szczegółowo proces działania aplikacji przeglądarkowej, należy zauważyć, że sama aplikacja nie przechowuje danych w pamięci podręcznej. Do obsługi danych wykorzystuje API, które wysyła żądania za pomocą protokołu HTTP. API komunikuje się z bazą danych, aby tworzyć, modyfikować, odczytywać i usuwać rekordy zawierające informacje potrzebne do działania aplikacji. Rysunek 2 przedstawia ogólny, uproszczony schemat architektury, pokazując zależności między warstwami oraz sposób ich komunikacji.



Rysunek 2. Wysokopoziomowa architektura projektu. Źródło: Opracowanie własne

Zaprezentowana na rysunku architektura nosi nazwę *three-tier architecture* [24]. Jest to jedna z najpopularniejszych metod projektowania aplikacji klient-serwer. Jej główną zaletą jest możliwość równoległego tworzenia warstw przez niezależne zespoły programistyczne. Dodatkowo umożliwia skalowanie lub aktualizację jednej warstwy bez ingerencji w pozostałe. Wadą tego podejścia mogą

być jednak wysokie koszty infrastrukturalne, ponieważ każda warstwa zwykle działa na osobnym serwerze, co zwiększa wydatki na sprzęt, sieć i utrzymanie.

Obecne trendy w rozwoju aplikacji zmierzają w kierunku rozwiązań chmurowych, które oferują większą elastyczność i skalowalność. Dzięki automatycznemu dostosowywaniu zasobów do obciążenia, łatwości zarządzania infrastrukturą i niższym kosztom operacyjnym rozwiązania chmurowe są coraz bardziej atrakcyjne z perspektywy przedsiębiorców [33].

4.2. Przykładowy projekt

W celu przeprowadzenia porównania między platformami Blazor i Angular stworzono dwa odrębne projekty, z których każdy oparty jest na jednej z tych technologii. Oba projekty zostały zaprojektowane tak, aby oferowały identyczny zestaw funkcjonalności, co zapewnia obiektywność testów oraz umożliwia rzetelną ocenę różnic i podobieństw między tymi rozwiązaniami.

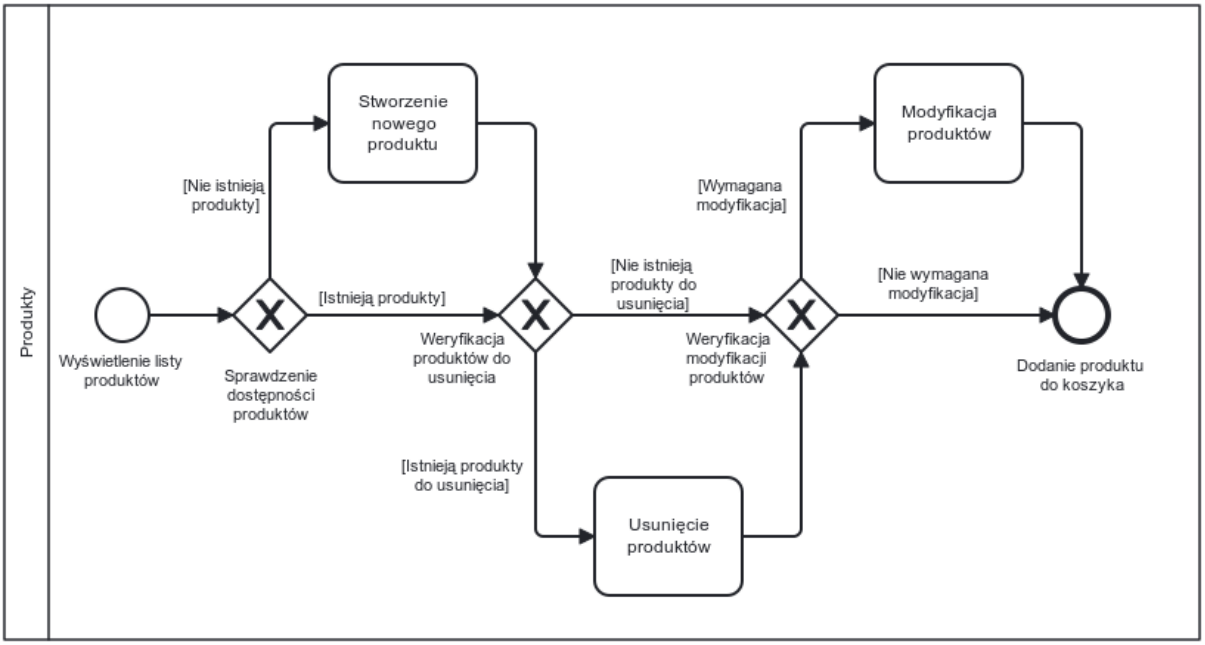
Stworzony projekt to system sklepu internetowego, który posiada następujące funkcjonalności:

- Wyświetlanie wszystkich produktów dostępnych w systemie.
- Otwieranie okna prezentującego szczegółowe dane dotyczące wybranego produktu.
- Dodawanie produktów.
- Modyfikowanie produktów.
- Usuwanie produktów.
- Dodawanie produktów do koszyka.
- Usuwanie produktów z koszyka.
- Czyszczenie koszyka.
- Składanie zamówienia.
- Przeglądanie historii zamówień.

W ramach projektu sklepu internetowego opracowano cztery ekrany:

- Strona główna – umożliwia zapoznanie się z aplikacją oraz nawigację do pozostałych ekranów.
- Logowanie – obsługuje proces autoryzacji użytkownika.
- Produkty – prezentuje listę produktów oraz umożliwia korzystanie z powiązanych funkcji, takich jak dodawanie do koszyka.
- Koszyk – przedstawia produkty dodane do koszyka.
- Zamówienia – wyświetla historię wszystkich zamówień złożonych przez zalogowanego użytkownika.

Diagram aktywności BPMN (ang. *Business Process Model and Notation*) ilustrujący proces dodawania produktu do koszyka został przedstawiony na rysunku 3. W trakcie tego procesu użytkownik może zarządzać listą produktów dostępną dla wszystkich klientów.

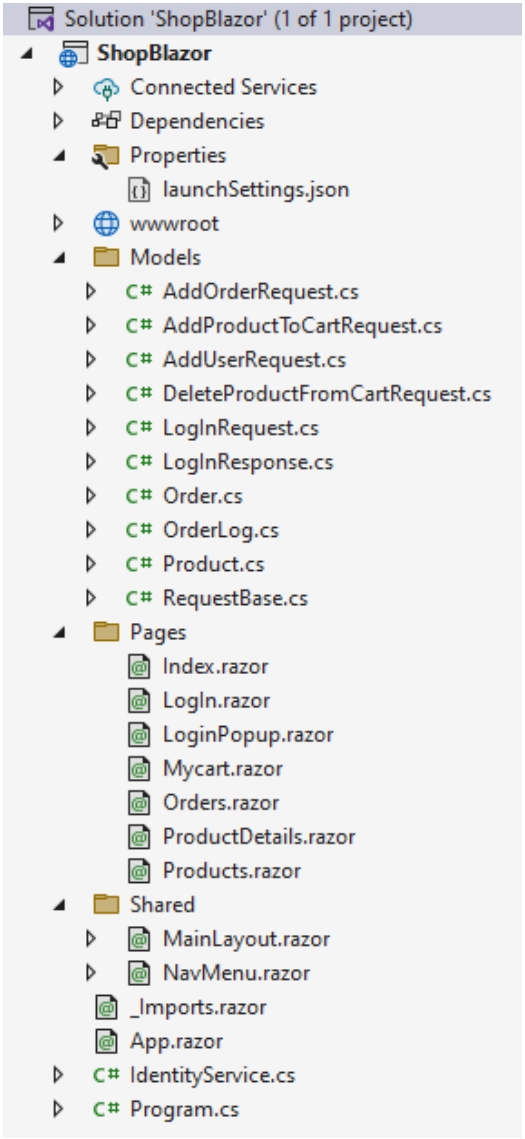


Rysunek 3. Proces biznesowy dodawania produktu do koszyka. Źródło: opracowanie własne

Ekran koszyka oferuje ograniczoną liczbę funkcjonalności. Użytkownik może usunąć wybrany produkt z koszyka, wyczyścić jego zawartość lub złożyć zamówienie. Z kolei ekran zamówień nie zawiera żadnych interaktywnych elementów – prezentuje jedynie listę zamówień złożonych przez klienta.

4.3. Aplikacja Blazor

Do stworzenia aplikacji Blazor wykorzystano środowisko programistyczne IDE (ang. *Integrated Development Environment*) Visual Studio 2022. Rysunek 4 przedstawia strukturę projektu. Aby wyjaśnić zależności w tym projekcie, można podzielić go na dwie główne części: modele oraz widoki Razor. Modele służą do komunikacji między aplikacją a API oraz do wypełniania danymi widoków. Widoki natomiast odpowiadają za prezentowanie danych znajdujących się w modelach, a także za wymianę informacji z API. Kod obsługujący aplikację został umieszczony w widokach, dzięki czemu jest on uruchamiany po stronie klienta, a nie serwera.



Rysunek 4. Solucja aplikacji Blazor. Źródło: opracowanie własne

Punktem wejścia aplikacji Blazor jest plik `Program.cs`, odpowiedzialny za jej budowanie i uruchamianie. W tym pliku konfiguruje się aplikację, definiowane usługi oraz ustawienia bezpieczeństwa. W folderze `wwwroot` znajdują się pliki CSS, które definiują wygląd oraz układ elementów używanych w widokach.

Aby użytkownik mógł korzystać z aplikacji, konieczna jest wcześniejsza autentykacja. Funkcjonalność tę realizuje plik `LogIn.razor`, gdzie użytkownik wpisuje login i hasło, a następnie naciska przycisk `Log in`. Po przesłaniu danych aplikacja wysyła żądanie do API. Przy poprawnych danych API zwraca token, który zostaje zapisany w *cookies* na okres 5 minut. Dzięki temu tokenowi każdorazowe otwarcie widoku pozwala sprawdzić, czy użytkownik jest zalogowany oraz wyświetlić dane przypisane do jego konta. Fragment kodu odpowiedzialny za logowanie został przedstawiony na listingu 2.

Listing 2. Metoda LogInUser w aplikacji Blazor. Źródło: opracowanie własne

```
public async Task LogInUser()
{
       LogInRequest request = new LogInRequest
       {
              Login = Login,
              Password = Password,
              UserToken = ""
       };

       LogInResponse? response = (await Http.PostAsJsonAsync("auth",
       request)).Content.ReadFromJsonAsync<LogInResponse>().Result;

       if (response != null && !String.IsNullOrEmpty(response.Token))
       {
              await
              JSRuntime.InvokeAsync<string>("setCookieValue",response.Token);
              IsSucces = true;
       }
         ShowDialog = true;
}
```

Widok `Index.razor` odpowiada za prezentację strony głównej aplikacji. Jego zadaniem jest powitanie użytkownika oraz umożliwienie przejścia do pozostałych widoków. Widok ten nie korzysta z modeli w celu prezentowania danych.

Strona umożliwiająca dodawanie, usuwanie, modyfikację produktów oraz dodawanie ich do koszyka to `Products.razor`. Produkty są prezentowane w formie tabeli. Po pobraniu listy produktów z API pętla foreach, napisana w języku C#, generuje wiersze dla każdego obiektu.

Koszyk zakupowy również jest przedstawiony w postaci tabeli, analogicznie do widoku `Products.razor`. Funkcjonalności, takie jak składanie zamówień, realizowane są poprzez połączenie z API, co pokazano na przykładzie zamówienia zawartości koszyka na listingu 3.

Listing 3. Metoda zamawiania koszyka w aplikacji Blazor. Źródło: opracowanie własne

```
public async Task CheckOut()
{
       await Http.PostAsJsonAsync("order",
       new AddOrderRequest
       {
              ProductIds = ProductList.Select(x => x.Id).ToList(),
              UserToken = UserToken
       });

       await ClearCart();
}
```

Zamówienia prezentowane są użytkownikowi w formie kart. Każda karta zawiera informacje takie jak data zamówienia, nazwy zamówionych produktów oraz ich łączna cena. Karty są generowane w pętli, a ich kolory przypisywane są losowo, co sprawia, że każda karta wygląda unikalnie i atrakcyjnie. Proces ten zilustrowano na listingu 4.

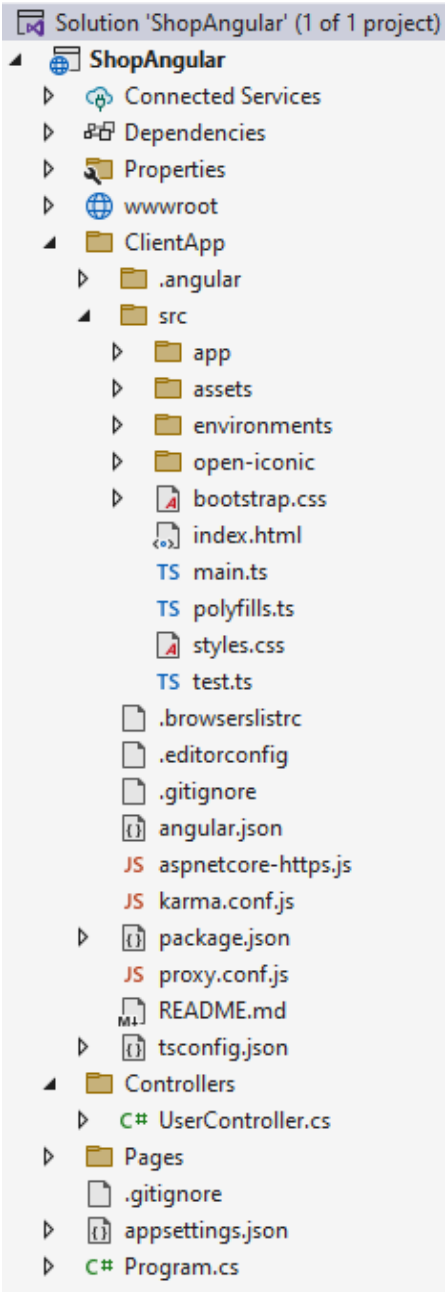
Listing 4. Karta zamówienia w aplikacji Blazor. Źródło: opracowanie własne

```
@foreach (var order in OrderLogList)
{
    if (random.NextDouble() > 0.5)
    {
        HighlightColor = "background-color: rgba(80, 37, 128,0.2)";
```

```
            CardColor = "background-color: rgba(145, 180, 223,0.05)";
        }
        else
        {
            HighlightColor = "background-color: rgba(145,180,223,0.1)";
            CardColor = "background-color: rgba(80, 37, 128,0.2)";
        }

        <div class="card" style="width: 18rem; max-width: fit-content;border-
                radius: 6px;margin:20px;@CardColor">
            <div class="card-body" style="text-align:center">
                <h3 class="card-title" style="@HighlightColor;max-width: fit-
                      content;display: inline;border-radius:16px">
                    @order.OrderDate.ToString("dd-MM-yyyy")
                </h3>
                <h4 class="card-text" style="margin-top: 35px;margin-bottom:
                      15px;">Products:</h4>
                <h5 class="card-text" style="margin:
                      15px;">@order.NameSum</h5>
                <h5 class="card-text" style="margin-top: 45px;margin-bottom:
                      15px;">Total: @order.PriceSum $</h5>
            </div>
        </div>
    }
```

4.4. Aplikacja Angular

Do stworzenia aplikacji przeglądarkowej Angular wykorzystano środowisko programistyczne Visual Studio 2022. Struktura projektu została przedstawiona na rysunku 5.



Rysunek 5. Solucja aplikacji Angular. Źródło: opracowanie własne

Projekt, zaprezentowany na rysunku 5, to klasyczna aplikacja typu SPA. Został on stworzony przy użyciu języków TypeScript oraz C# z wykorzystaniem wersji Angular 14.0.3. W celu lepszego zrozumienia struktury projektu można go podzielić na następujące części:

- `wwwroot` – zawiera zasoby statystyczne takie jak plik CSS, JavaScript, obrazy wykorzystywane przez widoki.
- `ClientApp` – główny folder zawierających aplikację Angulara. Znajdują się w nim komponenty, serwisy, modele oraz pliki konfiguracyjne niezbędne do działania aplikacji.
- `Controllers` – zawiera kontrolery ASP.NET Core, które obsługują żądania HTTP.

- `Program` - główny plik startowy aplikacji ASP.NET Core, odpowiedzialny za konfigurację i uruchomienie serwera.

Kluczowym elementem aplikacji Angular są komponenty, które można opisać jako "bloki budowlane" projektu. Są one odpowiedzialne za zarządzanie logiką i widokami poszczególnych części aplikacji. Każdy komponent składa się z czterech plików:

- *.ts - logika komponentu napisana w TypeScript,
- *.html - szablon HTML odpowiadający za widok,
- *.css: - style komponentu dla widoku HTML,
- *.spec.ts - plik do testów jednostkowych komponentu

Komponenty stworzone w ramach zrealizowania wymagań tego projektu są następujące:

- `cart`,
- `home`,
- `login`,
- `nav-menu`,
- `order`,
- `popup`,
- `product`,
- `product-details`

`Cart` komponent odpowiada za obszar aplikacji związany z koszykiem. Umożliwia przegląd koszyka zalogowanego użytkownika, usuwanie pozycji w nim pozycji, złożenie zamówienia produktów i wyczyszczenie wszystkich rekordów. Każda operacja korzysta z API opisanego w rozdziale 4.5. Na przykładzie metody `checkout`, zaprezentowanej na listingu 5, widzimy, że najpierw tworzony jest request zawierający token użytkownika. W pętli `for` dodawane są identyfikatory produktów, które mają być zamówione. Następnie wysyłany jest `request` typu POST do API. Po udanym złożeniu zamówienia lista produktów w koszyku jest odświeżana za pomocą metody `getProducts`.

Listing 5. Metoda checkout w aplikacji Angular. Źródło: opracowanie własne

```
checkout() {
	const request: AddOrderRequest = {
		ProductIds: [],
		UserToken: this.userToken
	};

	for (var i = 0; i < this.products.length; i++) {
		request.ProductIds.push(this.products[i].id);
	}

	this.http.post<any>(this.baseUrl + 'order', JSON.stringify(request), {
	headers: this.httpOptions.headers }).subscribe(return2 =>
	{
		this.getProducts();
	}, error => console.error(error));
}
```

Komponent `home` odpowiada za stronę główną aplikacji. Wyświetla trzy karty z przyciskami przekierowującymi do modułów koszyka, produktów oraz zamówień. Listing 6 pokazuje strukturę HTML komponentu z opisem modułów i przyciskami nawigacyjnymi.

Listing 6. Karta z ekranu głównego w aplikacji Angular. Źródło: opracowanie własne

```
<div class="card cardHome" style="border-radius: 2vh; background-color:
rgba(30, 37, 38,0.03); box-shadow: 12px 12px 2px 1px rgb(92, 106,
128, 0.1); ">
	<span class="oi oi-list" aria-hidden="true" style="font-size: 6em;
		margin-inline: auto; margin-top: 2vh"></span>
	<div class="card-body" style="text-align: center;">
		<h2 class="card-title" style="background-color: rgba(228,
		161,27,0.2); max-width: fit-content; display: inline;
		border-radius: 6px">Orders</h2>
		<p class="card-text">You will find here all of your's checked
			out carts, list of products that where in the cart and
			order date.</p>
		<a routerLink="/order" class="btn btn-warning">Enter</a>
	</div>
</div>
```

Kolejnym wylistowanym komponentem jest `login`. Umożliwia on użytkownikowi zalogowanie się do portalu, dzięki czemu ma on możliwość skorzystać z jego wszystkich funkcjonalności. `Login` działa na takiej samej zasadzie jak w aplikacji Blazor `LogIn.razor` opisany w rozdziale 4.3. Jednak w tym przypadku jako kod uruchamiany po stornie w przeglądarki używany jest TypeScript. W metodzie `login` zaprezentowanej na listingu 7 widzimy, że tworzy ona `request` na podstawie modelu przypisanego do formularza logowania. Wywoływana jest następnie metoda POST z kontrolera auth znajdującego się w ASP.NET Core Web API opisanym w rozdziale 4.5. Po autentykacji metoda ta uruchamiana jest w usłudze ASP.NET Core Web API działającej po stronie serwera aplikacji przeglądarkowej, której działanie zostanie opisane w kolejnym akapicie. Następnie otwierane jest okienko dialog informujące o udanym logowaniu. Po jego zamknięciu następuje przejście na stronę główną aplikacji. W przypadku negatywnej autentykacji serwera wyświetlane zostaje okienko informujące o błędnym loginie bądź haśle.

Listing 7. Metoda login w aplikacji Angular. Źródło: opracowanie własne

```
login() {
	const request = {
		Login: this.loginForm.controls.login.value,
		Password: this.loginForm.controls.password.value,
		UserToken: ""
	};
	this.http.post<any>(this.baseUrl + 'auth', JSON.stringify(request),
	this.httpOptions).subscribe(result => {
		if (result != null) {
			this.http.post<any>(this.innerServiceUrl + 'user?token='
				+ result.token, null).subscribe(result2 => { });

			const dialogRef = this.dialog.open(PopupComponent, {
				data: 'Login successful!',
				panelClass: 'custom-modalbox',
				position: { top: '25%', left: '45%' }
			});

			dialogRef.afterClosed().subscribe(result5 => {
				this.router.navigate(['']);});
		}
		else {
			const dialogRef = this.dialog.open(PopupComponent, {
				data: 'Wrong login or password',
				panelClass: 'custom-modalbox',
				position: { top: '25%', left: '45%' }});
```

```
            }
        });
}
```

W celu zapisywania otrzymanego tokenu użytkownika z API wykorzystano API hostowane przez serwer w aplikacji Angular. Metoda odpowiedzialna za tę funkcjonalność nosi nazwę `PostToken` i znajduje się w kontrolerze `UserController` (listing 8). Jeśli token nie jest pusty, zostaje zapisany w *cookies* przeglądarki na okres pięciu minut. Mechanizm ten umożliwia walidację użytkownika przy każdym otwarciu komponentu. Metoda wywoływana w celu pobrania tokenu to `GetToken`, która przekazuje zapisany token z *cookies*. W sytuacji, gdy token jest pusty lub został sfabrykowany, użytkownik zostaje przekierowany na stronę logowania.

Listing 8. Kontroler User w solucji Angular. Źródło: opracowanie własne

```
[ApiController]
[Route("[controller]")]
public class UserController : ControllerBase
{
        public UserController(ILogger<UserController> logger, IMemoryCache
                memoryCache)
        {

        }

        [HttpGet]
        public string GetToken()
        {
                return Request.Cookies["token"];
        }

        [HttpPost]
        public void PostToken(string token)
        {
                if (!string.IsNullOrEmpty(token))
                {
                        Response.Cookies.Append("token", token, new
                        CookieOptions
                                { Expires = DateTimeOffset.Now.AddMinutes(5) }
                        );
                }
        }
}
```

`Nav-menu` pełni funkcję paska nawigacji dostępnego na każdym etapie korzystania z aplikacji. Umożliwia swobodne przechodzenie między wszystkimi zaimplementowanymi ekranami. Dodatkowo oferuje funkcjonalność zmiany widoczności w zależności od ustawionej rozdzielczości przeglądarki, co umożliwia jego wygodne użytkowanie również na urządzeniach mobilnych.

Komponent `order` wyświetla wszystkie zamówienia zalogowanego użytkownika. Zawiera jedynie logikę odpowiedzialną za weryfikację autoryzacji oraz pobieranie danych, które są prezentowane w formie tabeli. Kod HTML tej tabeli przedstawiono na listingu 9.

Listing 9. Kod tabeli zamówień w aplikacji Angular. Źródło: opracowanie własne

```
<table class="table table-hover table-borderless  table-responsive" aria-
        labelledby="tableLabel" style="color: black; border-radius: .6rem;
        text-align: center; max-height: 650px; " *ngIf="orders">
        <thead style="background-color: rgba(30, 37, 32,0.01); border-radius:
                .6rem; text-align: center; font-size: 16px;">
```

```
                <tr style="border-radius: .6rem;">
                        <th>Name of products</th>
                        <th>Total cost</th>
                        <th>Order Date</th>
                </tr>
        </thead>
        <tbody style="padding: 1rem;text-align:center">
                <tr *ngFor="let order of orders">
                        <td>{{ order.nameSum }}</td>
                        <td>{{ order.priceSum }} $</td>
                        <td>{{ order.orderDate | date }}</td>
                </tr>
        </tbody>
</table>
```

`Popup` komponent nie zawiera żadnej logiki biznesowej, jego celem jest wyświetlenie w okienku przekazanego tekstu.

`Product` odpowiada za obszar aplikacji związany z produktami. Wyświetla wszystkie produkty znajdujące się w bazie, umożliwia usunięcie produktu, uruchomienie komponentu `product-details` oraz dodanie produktu do koszyka za pomocą metody przedstawionej na listingu 10.

Listing 10. Metoda dodawania produktu w aplikacji Angular. Źródło: opracowanie własne

```
addProductToCart(product: Product) {

        const request: AddProductToCartRequest = {
                ProductId: product.id,
                UserToken: this.userToken
        };

        this.http.post<any>(this.baseUrl + 'cart', JSON.stringify(request), {
        headers: this.httpOptions.headers }).subscribe(error =>
        console.error(error));
}
```

`Product-details` spełnia dwie kluczowe funkcje. Pierwszą z nich jest wyświetlenie pustego formularza po wywołaniu akcji dodania nowego produktu, aby użytkownik mógł wprowadzić dane nowego rekordu. Drugą funkcją jest edycja istniejącego produktu. Po kliknięciu przycisku `Open` dla wybranego wiersza otwierany jest ten sam formularz, jednak wypełniony już danymi wybranego produktu.

Wywołanie API odbywa się w komponencie `products`, a nie w `products-details`. W przedstawionym kodzie, zaprezentowanym na listingu 11, za pomocą metody `dialog.open` otwierane jest okno formularza. Przekazywane są jego wymiary oraz pusty obiekt `Product`, reprezentujący nowy produkt. Następnie metoda `dialogRef.afterClosed`, uruchamiana po zamknięciu okna, dodaje nowy produkt za pomocą API oraz pobiera zaktualizowaną kolekcję. Proces ten zachodzi wyłącznie wtedy, gdy spełniony zostanie warunek zawarty w instrukcji *if*. Zmienna `result`, czyli produkt z formularza, nie może być równa *null* i musi posiadać nazwę. Mechanizm ten stanowi zabezpieczenie przed dodawaniem produktów z pustą wartością w polu nazwa, gdyż pole to jest wymagane.

Listing 11. Metoda dodawania nowego produktu w aplikacji Angular. Źródło: opracowanie własne

```
openNewProduct() {
	const dialogRef = this.dialog.open(ProductDetailsComponent, {
		width: '520px',
		height: '440px',
		data: <Product>{},
		disableClose: true
	});

	dialogRef.afterClosed().subscribe(result => {
		if (result != null && result.name != null && result.name != "")
	{
			this.http.post<any>(this.baseUrl + 'product',
			JSON.stringify(result),
			this.httpOptions).subscribe(result3 => {
				this.http.get<Product[]>(this.baseUrl +
				'product').subscribe(result2 => {
					this.products = result2;
				}, error => console.error(error));
			}, error => console.error(error));
		}
	});
}
```

4.5. ASP.NET Core Web API

W celu obsługi danych wykorzystywanych w obu aplikacjach przeglądarkowych zastosowano ASP.NET Core Web API. Komunikacja między aplikacjami a API odbywa się za pośrednictwem protokołu HTTP, z wykorzystaniem formatu JSON do przesyłania danych. Główna logika aplikacji znajduje się w kontrolerach, z których każdy odpowiada za konkretne przypisane zasoby. Kontrolery wykorzystują system *routingu* do automatycznego mapowania adresów URL na odpowiednie metody w klasach. Dzięki atrybutowi `Route` możliwe jest określenie ścieżki konkretnego kontrolera w postaci ciągu znaków, co ułatwia konfigurację całego API. Stworzone zostały następujące cztery kontrolery oparte na architekturze REST:

- `AuthController`,
- `CartController`,
- `OrderController`,
- `ProductController`

Kontrolery korzystają z repozytoriów do wykonywania operacji na bazie danych. W celu ich implementacji wykorzystano bibliotekę ADO.NET, która umożliwia programistom dostęp do relacyjnych baz danych. Dzięki temu stworzono metody nawiązujące połączenie z bazą Microsoft SQL Server oraz wywołujące procedury składowane. Ciąg połączeniowy (`ConnectionString`) wykorzystywany w połączeniach jest pobierany z pliku konfiguracyjnego aplikacji.

`AuthController` odpowiada za proces autentykacji. Udostępnia dwie metody, przedstawione na listingu 12:

- `LogIn` – służy do uwierzytelnienia użytkownika. Na podstawie przesłanych danych logowania (login i hasło) sprawdza, czy takie dane istnieją w bazie. W przypadku pozytywnej weryfikacji generuje token, który następnie zapisuje w pamięci podręcznej za pomocą biblioteki dostępnej w ramach Microsoft.Extensions.Caching. Następnie jest on zwracany w odpowiedzi, chyba że w pamięci znajduje się już wartość przypisana do tego samego użytkownika – wówczas zwracany jest już istniejący.
- `IsLoggedIn` – sprawdza, czy przekazany token został utworzony w API oraz czy wciąż znajduje się w pamięci podręcznej. Dzięki temu można potwierdzić, że użytkownik jest

poprawnie zalogowany i nie próbuje podszywać się pod inną osobę lub obejść mechanizmów autoryzacji.

Tego rodzaju mechanizmy zapewniają wysoki poziom bezpieczeństwa i pewność poprawnego uwierzytelnienia użytkownika.

Listing 12. Kontroler AuthController w API. Źródło: opracowanie własne

```
[ApiController]
[Route("[controller]")]
public class AuthController : ControllerBase
{
	private readonly IMemoryCache memoryCache;
	private readonly UserRepository userRepository;
	public AuthController(ILogger<CartController> logger, IMemoryCache
		memoryCache, IConfiguration configuration)
	{
		this.memoryCache = memoryCache;
		userRepository = new
		UserRepository(configuration.GetConnectionString("AppConString"));
	}

	[HttpPost]
	public LogInResponse LogIn([FromBody] LogInRequest request)
	{
		if(userRepository.DoesUserExists(request.Login, request.Password))
		{
			object? value = memoryCache.Get(request.Login);
			if (value != null && !String.IsNullOrEmpty(value.ToString()))
			{
				return new LogInResponse() { Token = value.ToString() };
			}
			else
			{
				string token = Guid.NewGuid().ToString();
				memoryCache.Set(token, request.Login,
					TimeSpan.FromMinutes(5));
				memoryCache.Set(request.Login, token,
					TimeSpan.FromMinutes(5));
				return new LogInResponse() { Token = token };
			}
		}
		else
		{
			return new LogInResponse() { Token = null };
		}
	}

	[HttpGet]
	public bool IsLoggedIn(string token)
	{
		object? value = memoryCache.Get(token);
		if (value != null && !String.IsNullOrEmpty(value.ToString()))
		{
			return true;
		}
		else
		{
			return false;
		}
	}
}
```

W celu zarządzania koszykiem stworzono kontroler `CartController`. Wszystkie metody zawarte w tym kontrolerze zostały zaprezentowane na listingu 13. Wykorzystują one funkcję `GetUserLogin`, która na podstawie tokenu zalogowanego użytkownika zwraca jego `id`, co umożliwia określenie, dla kogo należy dokonać zmian. Do połączenia z bazą danych zastosowano repozytoria korzystające z wcześniej wspomnianej biblioteki ADO.NET.

Listing 13. Metody w CartController. Źródło: opracowanie własne

```
[HttpGet]
public IEnumerable<Product> Get(string token)
{
	string userId = GetUserLogin(token);
	List<int> prodouctsIds = cartRepository.GetCart(userId);
	if(prodouctsIds != null && prodouctsIds.Any())
	{
		List<Product> products = productRepository.GetProducts();
		return products.Where(x => prodouctsIds.Contains(x.Id));
	}
	else
	{
		return new List<Product>();
	}
}

[HttpPost]
public void Add(AddProductToCartRequest request)
{
	string userId = GetUserLogin(request.UserToken);
	cartRepository.AddProduct(request.ProductId, userId);
}

[HttpPut]
public void Delete(DeleteProductFromCartRequest request)
{
	string userId = GetUserLogin(request.UserToken);
	bool emptyCart = cartRepository.RemoveProduct(request.Id, userId);
	if(emptyCart)
	{
		cartRepository.DeleteCart(userId);
	}
}
```

Kontrolery `OrderController` oraz `ProductController` nie różnią się od `CartController` pod względem zastosowanych mechanizmów i bibliotek. Również wykorzystują funkcję `GetUserLogin` (listing 14) oraz repozytoria stworzone przy użyciu biblioteki ADO.NET. Główna różnica między nimi dotyczy obszaru, który obsługują. Kontroler `OrderController` odpowiada za obsługę zamówień, natomiast `ProductController` zarządza modułem produktów.

Listing 14. Metoda GetUserLogin w API. Źródło: opracowanie własne

```
private string GetUserLogin(string token)
{
	return _memoryCache.Get(token).ToString();
}
```

Repozytoria to klasy odpowiedzialne za nawiązywanie połączenia z bazą danych. W listingu 15, podczas tworzenia obiektu `UserRepository`, przekazywany jest tekst `connectionString`, który zostaje zapisany we właściwości i później wykorzystany przy inicjacji połączenia.

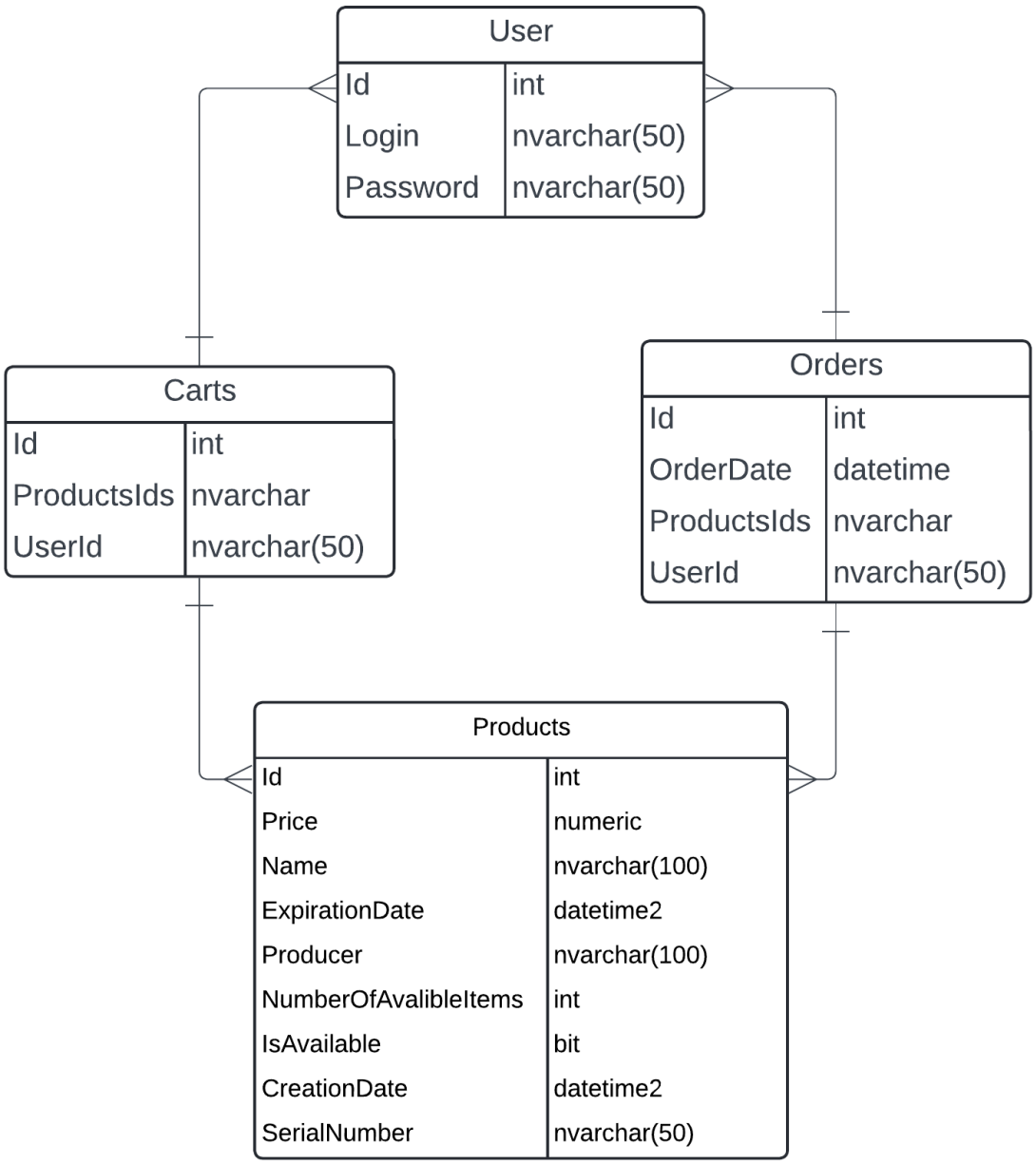
W metodzie `DoesUserExists`, przedstawionej na listingu 15, na początku za pomocą instrukcji `using` tworzony jest obiekt `SqlConnection`. Takie rozwiązanie gwarantuje, że po zakończeniu bloku kodu połączenie zostanie automatycznie zamknięte i zasoby zwolnione, co zapobiega utrzymywaniu niepotrzebnych połączeń z bazą danych. Następnie tworzony jest obiekt `SqlCommand`, do którego przekazywana jest nazwa procedury zapisanej w bazie danych oraz wcześniej utworzony `SqlConnection`. Dzięki temu możliwe jest korzystanie z metod takich jak `ExecuteReader` i `ExecuteNonQuery`, które umożliwiają wywoływanie procedur zapisanych w bazie Microsoft SQL Server. Pozostałe repozytoria zostały zaprojektowane w podobny sposób, wykorzystując tę samą bibliotekę ADO.NET.

Listing 15. Repozytorium użytkownika w API. Źródło: opracowanie własne

```
public class UserRepository
{
	private string ConnectionString { get; set; }

	public UserRepository(string connectionString)
	{
		this.ConnectionString = connectionString;
	}

	public bool DoesUserExists(string login, string password)
	{
		using (SqlConnection connection = new
			SqlConnection(ConnectionString))
		{
			SqlCommand command = new SqlCommand("[dbo].[DoesUserExists]",
				connection);
			command.CommandType = CommandType.StoredProcedure;
			command.Parameters.AddWithValue("@Login", login);
			command.Parameters.AddWithValue("@Password", password);
			connection.Open();

			return command.ExecuteReader().HasRows;
		}

		return false;
	}

	public void AddUser(string login, string password)
	{
		using (SqlConnection connection = new
			SqlConnection(ConnectionString))
		{
			SqlCommand commandInsert = new
				SqlCommand("[dbo].[InsertUser]", connection);
			commandInsert.CommandType = CommandType.StoredProcedure;
			commandInsert.Parameters.AddWithValue("@Login", login);
			commandInsert.Parameters.AddWithValue("@Password", password);

			connection.Open();
			commandInsert.ExecuteNonQuery();
		}
	}
}
```

4.6. Microsoft SQL Server

Miejscem służącym do przechowywania danych aplikacji jest relacyjna baza danych Microsoft SQL Server. Na rysunku 6 znajdują się cztery tabele stworzone na potrzebę przykładowego projektu.



Rysunek 6. Diagram bazy danych. Źródło: opracowanie własne

W tabeli `products`, pomimo obecności wielu kolumn, nie ma żadnego klucza obcego, który odnosiłby się do innej tabeli. Rekordy w tej tabeli zawierają dane globalne, dostępne dla wszystkich użytkowników i są dla każdego z nich identyczne. W przypadku dodania nowego produktu, staje się

on widoczny także dla innych użytkowników. Tabela przechowuje następujące informacje o stworzonych produktach:

- `Id` – unikalny identyfikator,
- `Price` – cena produktu,
- `Name` – nazwa produktu,
- `ExpirationDate` – termin ważności,
- `Producer` – nazwa producenta produktu,
- `NumberOfAvalibleItems` – ilość dostępnych sztuk,
- `IsAvailable` – informacja o dostępności produktu,
- `CreationDate` – data stworzenia rekordu w bazie,
- `SerialNumber` – numer seryjny produktu

Rekordy w tabelach `Carts` i `Orders` są danymi prywatnymi. Każdy użytkownik posiada własny zestaw koszyków i zamówień, które nie są współdzielone z innymi użytkownikami aplikacji. Tabele te wykorzystują powiązanie z tabelą Products w postaci listy identyfikatorów (`Id`), co umożliwia określenie zawartości koszyka lub zamówienia.

W tabeli `User` znajdują się wszyscy użytkownicy aplikacji, którym został nadany dostęp. Jej identyfikator (`Id`) możemy znaleźć w tabelach `Carts` i `Orders`, dzięki czemu aplikacja wie, do którego użytkownika należy dany rekord. Tabela zawiera również dwie kolumny: `Login` i `Password`, w których przechowywane są login oraz hasło użytkownika. Są one wykorzystywane do autentykacji użytkownika logującego się na stronie internetowej.

Procedury znajdujące się w bazie Microsoft SQL Server służą do operacji CRUD (ang. *create, read, update, delete*). Dla przykładu, listing 16 pokazuje procedurę o nazwie `[dbo].[InsertProduct]`, która dodaje nowy produkt do tabeli `Products`. Posiada ona siedem parametrów wejściowych, które zostały uzupełnione przez użytkownika w aplikacji przeglądarkowej. Po zapisaniu danych, za pomocą funkcji `SCOPE_IDENTITY`, zwracany jest klucz główny nowo utworzonego rekordu.

Listing 16. Procedura dodawania produktu. Źródło: opracowanie własne

```
USE [Shop]
GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO

CREATE PROCEDURE [dbo].[InsertProduct]
	@Price numeric(18,2),
	@Name nvarchar(100) NULL,
	@ExpirationDate datetime2(7),
	@Producer nvarchar(100) NULL,
	@NumberOfAvalibleItems int,
	@IsAvailable bit,
	@SerialNumber nvarchar(50)

AS
BEGIN

	SET NOCOUNT ON;
```

```
        INSERT INTO [dbo].[Products]
        VALUES (@Price, @Name, @ExpirationDate, @Producer, @NumberOfAvalibleItems,
        @IsAvailable, GETDATE(), @SerialNumber)


        SELECT SCOPE_IDENTITY();
END
```

4.7. Testy Playwright

Testy *end-to-end* dla aplikacji przeglądarkowych zostały stworzone przy pomocy Playwright w języku JavaScript. Stworzonych zostało osiem scenariuszy testowych, tak, aby w pełni objąć wszystkie funkcjonalności aplikacji. Są one następujące:

- `addProduct` – dodanie nowego produktu,
- `addToCart` – dodanie produktu do koszyka,
- `cart` – wczytanie ekranu koszyka,
- `checkoutCart` – zamówienie zawartości koszyka,
- `deleteProduct` – usunięcie produktu,
- `orders` – wczytanie ekranu zamówienia,
- `products` – wczytanie ekranu produktów,
- `updateProduct` – modyfikacja produktu

Dla każdego scenariusza testowego opracowano odpowiadającą mu funkcję, która go realizuje. Z racji istnienia dwóch aplikacji przeglądarkowych Angular oraz Blazor, dla każdej z nich skonstruowano ich po dwie. Biblioteka Playwright udostępnia szereg metod, dzięki którym można imitować pracę użytkownika. Funkcja `goto` powoduje załadowanie strony internetowej przekazanej w parametrze wejściowym jako URL. Dzięki takim funkcjom jak `locator`, `getByRole`, `getByText` i `getByTitle` wyszukiwane są elementy znajdujące się w aplikacji, co pozwala na interakcję z nimi.

Dla przykładu, listing 17 przedstawia funkcję `addProduct`, która odpowiada scenariuszowi dodawania nowego produktu. Na początku uruchamiana jest strona główna projektu testowego, a następnie przechodzi się na stronę logowania. Po wyszukaniu elementów znajdujących się w formularzu logowania, wypełniane są one, a następnie klikany jest przycisk `Log in`. Po zamknięciu okienka potwierdzającego pozytywną autentykację, program przechodzi do zakładki produktów, otwiera formularz dodawania nowego produktu i uzupełnia w nim dane. Po uzupełnieniu formularza wywoływana jest funkcja zapisania, a program oczekuje, aż zostanie wyświetlony ekran produktów, zaktualizowany o nowy rekord.

Listing 17. Test dodający produkt do bazy. Źródło: opracowanie własne

```
const { expect } = require('@playwright/test');

async function addProduct(page)
{
        await page.goto('https://localhost:7101/');
        await page.goto('https://localhost:7101/login');
        await page.locator('#login').click();
```

```
        await page.locator('#login').fill('test@test.pl');
        await page.locator('#password').click();
        await page.locator('#password').fill('pass');
        await page.getByRole('button', { name: 'Log in' }).click();
        await page.getByText('Close').click();
        await page.getByRole('link', { name: ' Products' }).click();
        await expect(page.getByRole('cell', { name: 'Name', exact: true
        })).toBeVisible();

        await page.getByRole('status').click();
        var number = Math.floor(Math.random() * 100) + 1;
        var name = "TestAddAutomatic" + number.toString();

        await page.getByTitle("product-name").type(name);
        await page.getByTitle("product-price").type(number.toString());
        await page.getByTitle("product-producer").type(name);
        await page.getByTitle("product-
              numberOfAvalibleItems").type(number.toString());
        await page.getByTitle("product-isAvailable").check();
        await page.getByTitle("product-expirationDate").type(new
              Date().toLocaleDateString());
        await page.getByTitle("product-creationDate").type(new
              Date().toLocaleDateString());
        await page.getByTitle("product-creationDate").type(new
              Date().toLocaleDateString());

        await page.getByText("Save").click();
        await expect(page.getByTitle("Name").last()).toHaveText(name);
}

module.exports = {
        addProduct,
}
```

4.8. Testy Artillery

W celu sprawdzenia wydajności aplikacji Angular i Blazor wykorzystano narzędzie Artillery. Pozwala ono na tworzenie wirtualnych użytkowników, którzy wykonują zadany test stworzony przy pomocy Playwright. Na listingu 18 zaprezentowano skrypt YAML, który konfiguruje test obciążeniowy. Parametr target wskazuje URL serwera, na którym będą przeprowadzane testy; w tym przypadku jest to lokalny serwer działający na porcie 7101. `Phases` definiuje trzy różne fazy testu, w których stopniowo zwiększa się obciążenie. Parametr `duration` określa czas trwania danej fazy testu w sekundach. `ArrivalRate` to liczba użytkowników, którzy będą inicjowani na sekundę w danej fazie. Parametr `maxVusers` definiuje maksymalną liczbę wirtualnych użytkowników, którzy mogą być aktywni. `Name` to nazwa fazy, która ułatwia jej identyfikację. Sekcja `engines` definiuje narzędzia wykorzystywane do wykonywania testów. W przypadku skryptu z listingu 18 jest to silnik

Playwright. `Scenarios` określa, co będzie testowane w danym scenariuszu, czyli jakie działania będą podejmowane przez wirtualnych użytkowników. W parametrze `testFunction` zapisano wszystkie funkcje testowe, opisane w rozdziale 4.7. Dzięki temu przetestowano wcześniej zdefiniowane scenariusze.

Listing 18. Skrypt YAML. Źródło: opracowanie własne

```
config:
  target: https://localhost:7101
  phases:
       - duration: 30
         arrivalRate: 1
         maxVusers: 2
         name: ramp up
       - duration: 30
         arrivalRate: 2
         maxVusers: 10
         name: sustain
       - duration: 30
         arrivalRate: 2
         maxVusers: 20
         name: max load
  engines:
       playwright: {}
  processor: "./tests/artillery/blazor.js"
scenarios:
  - engine: playwright
    testFunction: "artilleryProducts"
```

5. Testy porównawcze technologii

Testy opisane w poniższym rozdziale przeprowadzono na dwóch maszynach. Na jednej z nich działa aplikacja, natomiast na drugiej uruchamiano testy za pomocą narzędzia Artillery. Komputer udostępniający aplikacje był wyposażony w procesor Intel i7-9700F oraz 16 GB pamięci RAM. Aplikacje uruchomiono w trybie *release*, z wyłączoną opcją debugowania kodu. Do testów wykorzystano przeglądarkę Chromium. Komputer używany do obsługi Artillery wyposażono w procesor Intel i7-1370P oraz 64 GB pamięci RAM. Każdy test składa się z trzech faz:

- *ramp up* – 3 użytkowników aktywnych przez 60 sekund,
- *sustain* – 10 użytkowników aktywnych przez 60 sekund,
- *max load* – 25 użytkowników aktywnych przez 60 sekund.

Podczas każdego scenariusza testowego monitorowano za pomocą narzędzia Performance Monitor [26] zużycie pamięci RAM i procesora. Wyniki testów zostały przedstawione w formie wykresów, które ilustrują dwie funkcje w czasie. Czerwona linia reprezentuje zużycie pamięci podręcznej, wyrażone w procentach, natomiast zielona linia przedstawia wykorzystanie procesora, również wyrażone w procentach. Wykresy wyników znajdują się na rysunkach: 7, 8, 10, 11, 13, 14, 16, 17, 19, 20, 22, 23, 25, 26, 28, 29. Na podstawie danych uzyskanych za pomocą Performance Monitor w tabelach: 1, 3, 5, 7, 9, 11, 13 i 15 zapisano maksymalne oraz średnie wartości zużycia zasobów dla każdej z testowanych maszyn.

Narzędzie Artillery umożliwia zbieranie i wizualizację danych z przeprowadzanych testów. Na wykresach, przedstawionych na rysunkach 9, 12, 15, 18, 21, 24, 27 i 30, zaprezentowano pięć następujących funkcji, które obrazują wyniki testów:

- `vusers.failed` – liczba użytkowników, dla których test zakończył się niepowodzeniem,
- `vusers.session_length.min` – minimalny czas wykonania danego scenariusza testowego (w sekundach) w określonym przedziale czasowym,
- `vusers.session_length.mean` – średni czas wykonania danego scenariusza testowego (w sekundach) w określonym przedziale czasowym,
- `vusers.session_length.p95` – 95. percentyl czasu wykonania danego scenariusza testowego (w sekundach) w określonym przedziale czasowym,
- `vusers.session_length.max` – maksymalny czas wykonania danego scenariusza testowego (w sekundach) w określonym przedziale czasowym.

Dodatkowo, na podstawie danych zebranych za pomocą narzędzia Artillery, przedstawiono wyniki testów w formie tabel zawierających następujące rekordy:

- Minimalna długość sesji [s] – najkrótszy czas (w sekundach) potrzebny na wykonanie scenariusza testowego,
- Średnia długość sesji [s] – średni czas (w sekundach) wykonania scenariusza testowego,
- Długość sesji p95 [s] – 95. percentyl czasu (w sekundach) wykonania scenariusza testowego,
- Maksymalna długość sesji [s] – najdłuższy czas (w sekundach) wykonania scenariusza testowego,
- Użytkownicy udani – liczba użytkowników, którzy pomyślnie zakończyli scenariusz testowy bez błędów,
- Użytkownicy z błędem – liczba użytkowników, którzy nie ukończyli scenariusza testowego z powodu błędów.

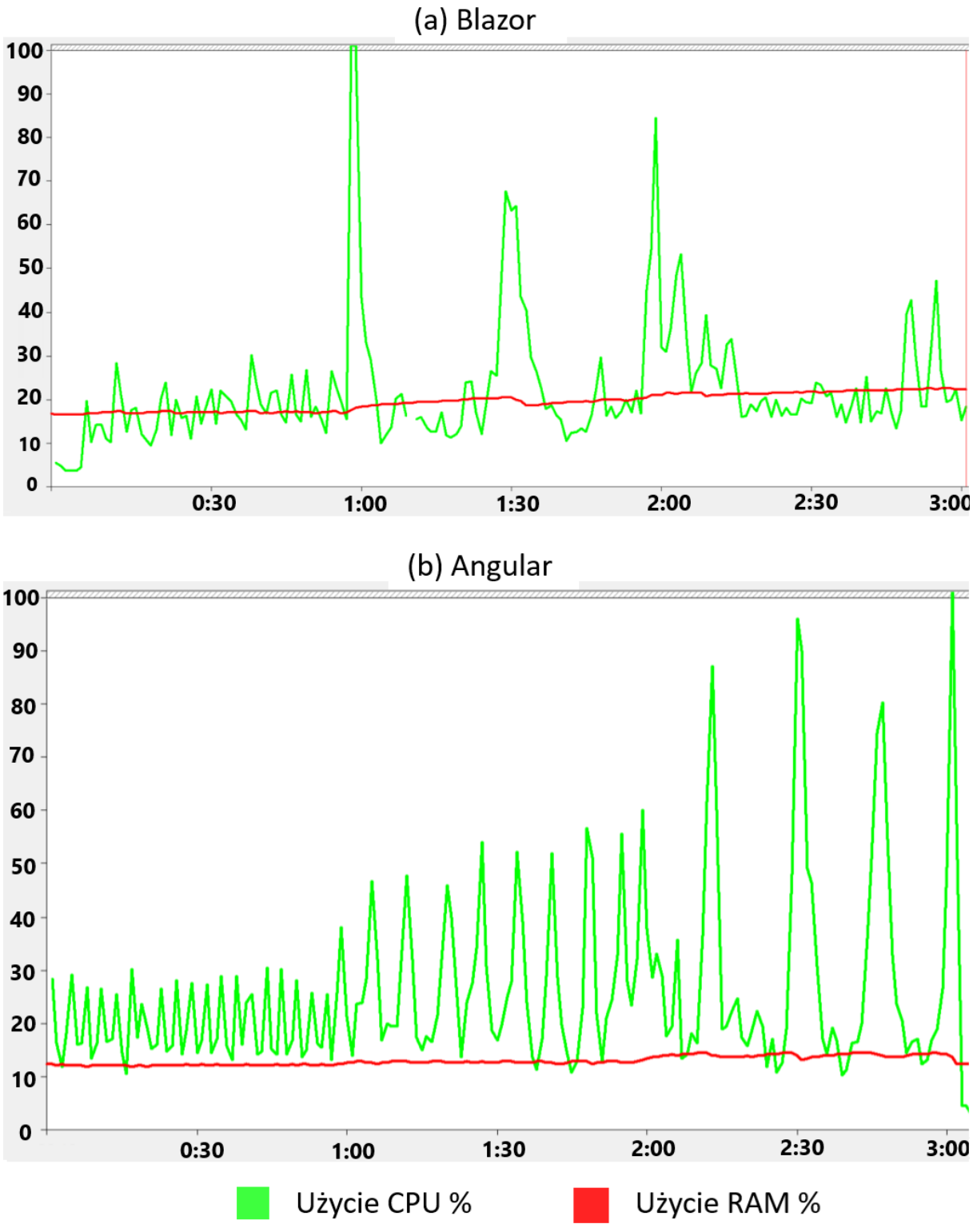
5.1. Wczytanie ekranu produktów

Tabela 1 przedstawia dane dotyczące obciążenia komponentów komputerowych podczas wykonywania testu. Komputer testujący aplikację Blazor wykazuje wyższe zużycie zasobów w porównaniu do komputera testującego aplikację Angular, mimo że dla obu aplikacji maksymalne zużycie procesora osiągnęło 100%. Z kolei komputer hostujący aplikację Blazor zużył mniej zasobów niż komputer hostujący aplikację Angular. Na przykład różnica w średnim zużyciu CPU wyniosła 6,6%.

Tabela 1. Obciążenie komponentów dla scenariusza 5.1. Źródło: opracowanie własne

	Komputer testujący - Blazor	Komputer hostujący - Blazor	Komputer testujący - Angular	Komputer hostujący - Angular
Pamięć RAM maks. [%]	22,5	60,6	13,2	67,9
Pamięć RAM śr. [%]	19,5	59,8	12,1	67,1
Użycie CPU maks. [%]	100	26,6	100	32,3
Użycie CPU śr. [%]	22,5	4,6	35,2	11,2

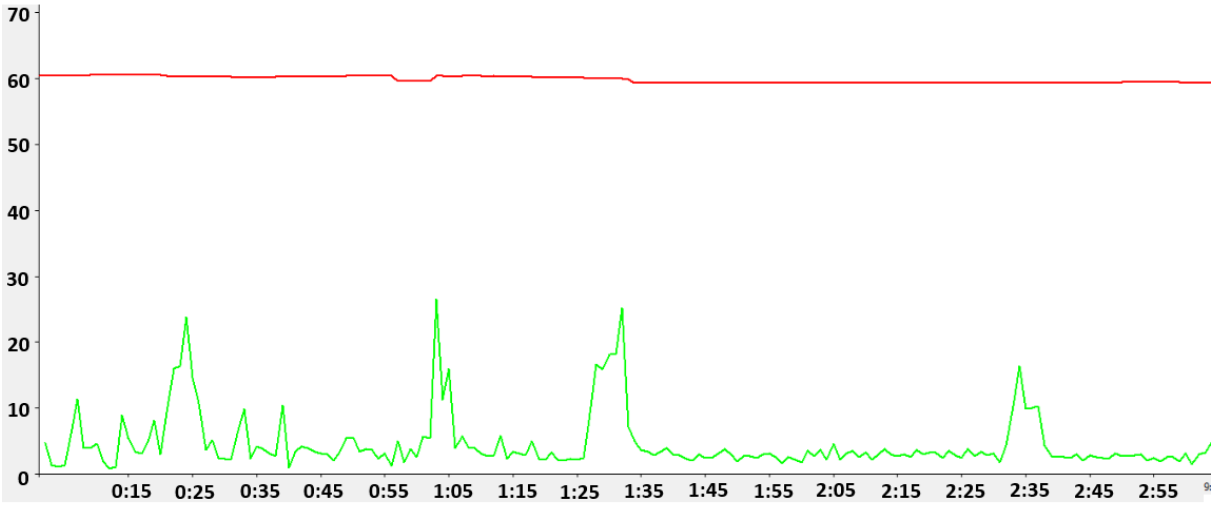
Na rysunku 7 przedstawiono dwa wykresy obrazujące obciążenie komputera testującego aplikacje Blazor i Angular. W przypadku aplikacji Blazor wykres wskazuje na większe zużycie procesora, co widoczne jest jako liczne nagłe wzrosty obciążenia. Zużycie pamięci RAM w obu aplikacjach stopniowo wzrasta, jednak aplikacja Blazor od początku wykorzystuje więcej tego zasobu.



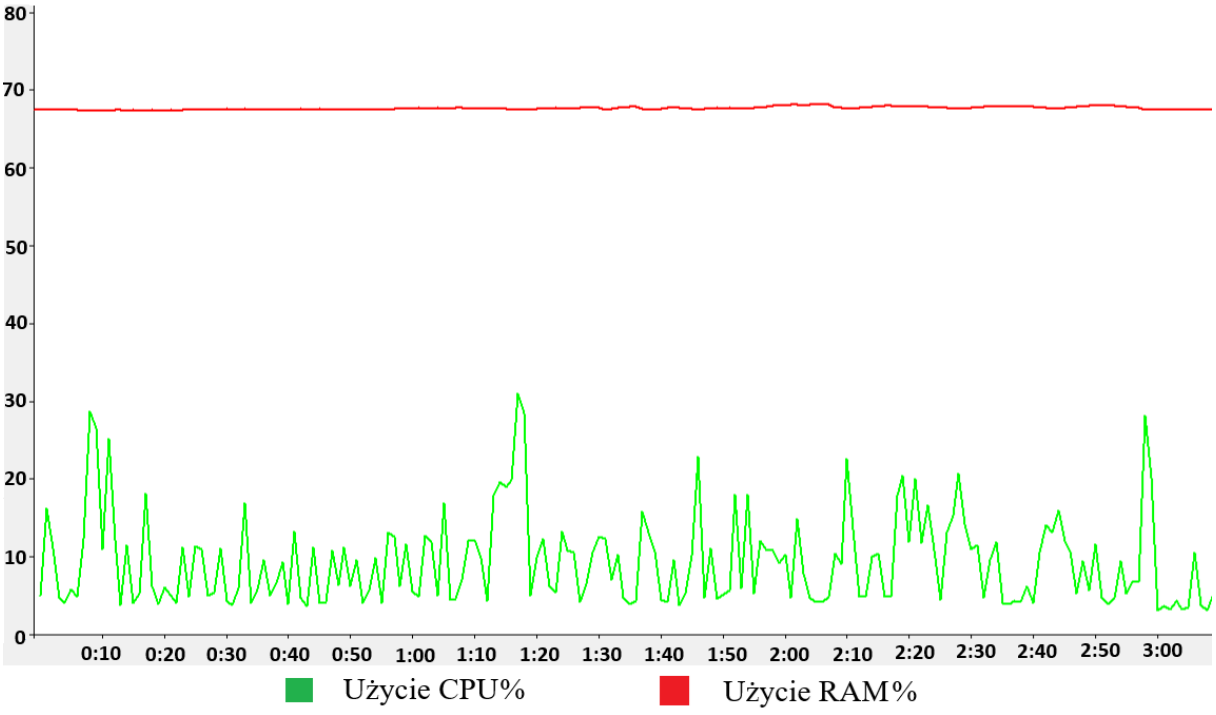
Rysunek 7. Obciążenie komputera testującego dla scenariusza 5.1. Źródło: opracowanie własne

Na rysunku 8 przedstawiono wykres obciążenia komputera hostującego aplikacje Blazor i Angular. Zaobserwowano sporadyczne, niskie skoki zużycia procesora w przypadku aplikacji Blazor, podczas gdy aplikacja Angular charakteryzuje się ich znacznie większą liczbą. Dodatkowo, aplikacja Angular zużywa więcej zasobów pamięci podręcznej.

(a) Blazor



(b) Angular



Rysunek 8 . Obciążenie komputera hosta dla scenariusza 5.1. Źródło: opracowanie własne

W tabeli 2 przedstawiono wyniki testu obciążeniowego. Zaobserwowano, że dla każdego parametru aplikacja Angular osiąga lepsze wyniki. Na przykład minimalna długość sesji użytkownika wynosi 1,8 sekundy, podczas gdy aplikacja Blazor potrzebowała na to dwa razy więcej czasu.

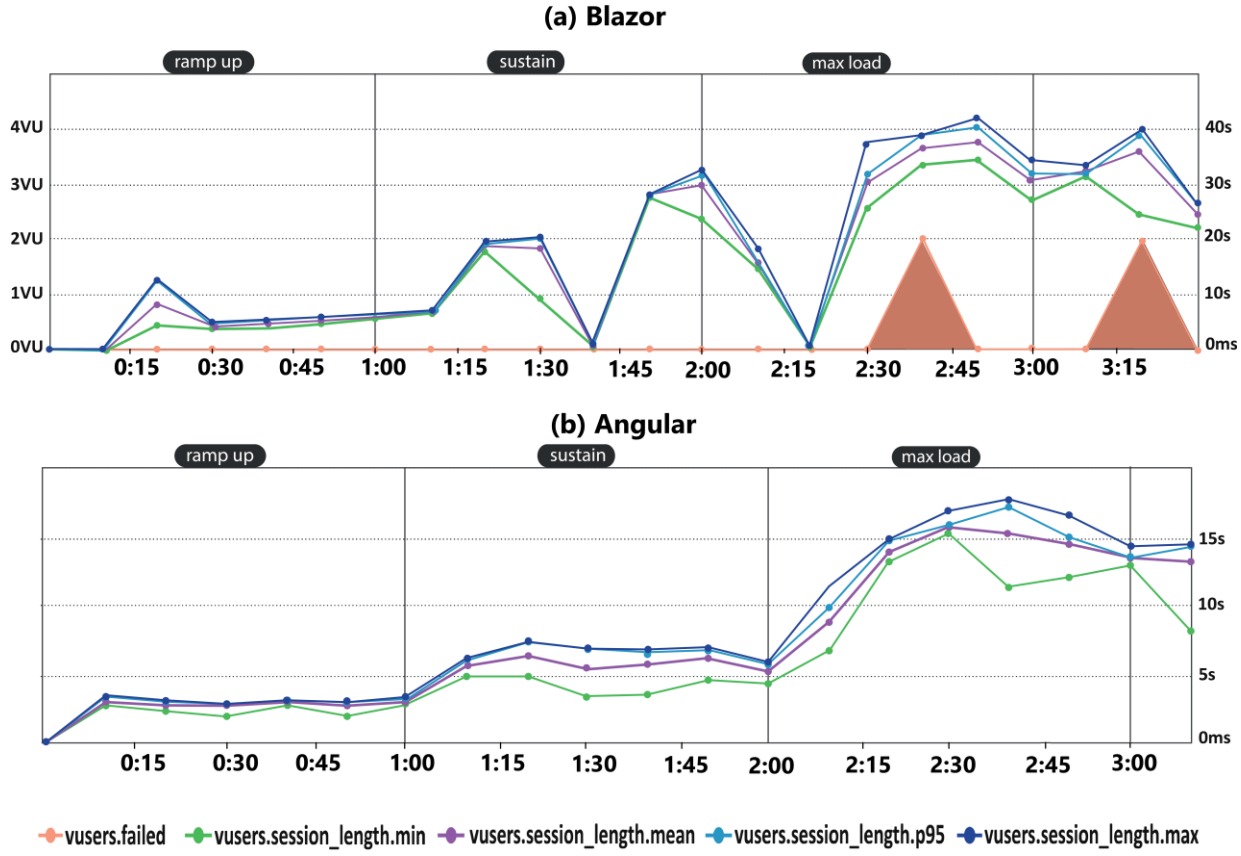
Tabela 2. Wyniki testu obciążającego dla scenariusza 5.1. Źródło: opracowanie własne

	Blazor	Angular
Minimalna długość sesji [s]	3,6	1,8
Średnia długość sesji [s]	23	8,4
Długość sesji p95 [s]	38,9	16,1
Maksymalna długość sesji [s]	42,4	17,7
Użytkownicy udani	109	257
Użytkownicy z błędem	4	0

Rysunek 9 przedstawia wykres wyników testu obciążeniowego aplikacji Blazor i Angular dla scenariusza wczytywania ekranu produktów.

W przypadku aplikacji Blazor, podczas fazy *ramp-up*, wykazywała ona wysoką wydajność, a średnia długość sesji wynosiła poniżej 10 sekund. Przy 15 aktywnych użytkownikach czas odpowiedzi znacząco wzrósł do około 30 sekund. Maksymalne obciążenie spowodowało wystąpienie błędów przekroczenia czasu odpowiedzi u czterech użytkowników. Po zakończeniu fazy maksymalnego obciążenia (*max load*), w której nie dodawano nowych użytkowników, wydajność aplikacji zaczęła się poprawiać.

Dla aplikacji Angular, w fazie *ramp-up*, utrzymywała ona stałą wydajność na poziomie 3 sekund. Przy 15 aktywnych użytkownikach czas odpowiedzi wydłużył się dwukrotnie. Aplikacja, nawet przy maksymalnym obciążeniu, nie generowała błędów przekroczenia czasu odpowiedzi.



Rysunek 9. Wynik testu obciążającego dla scenariusza 5.1. Źródło: opracowanie własne

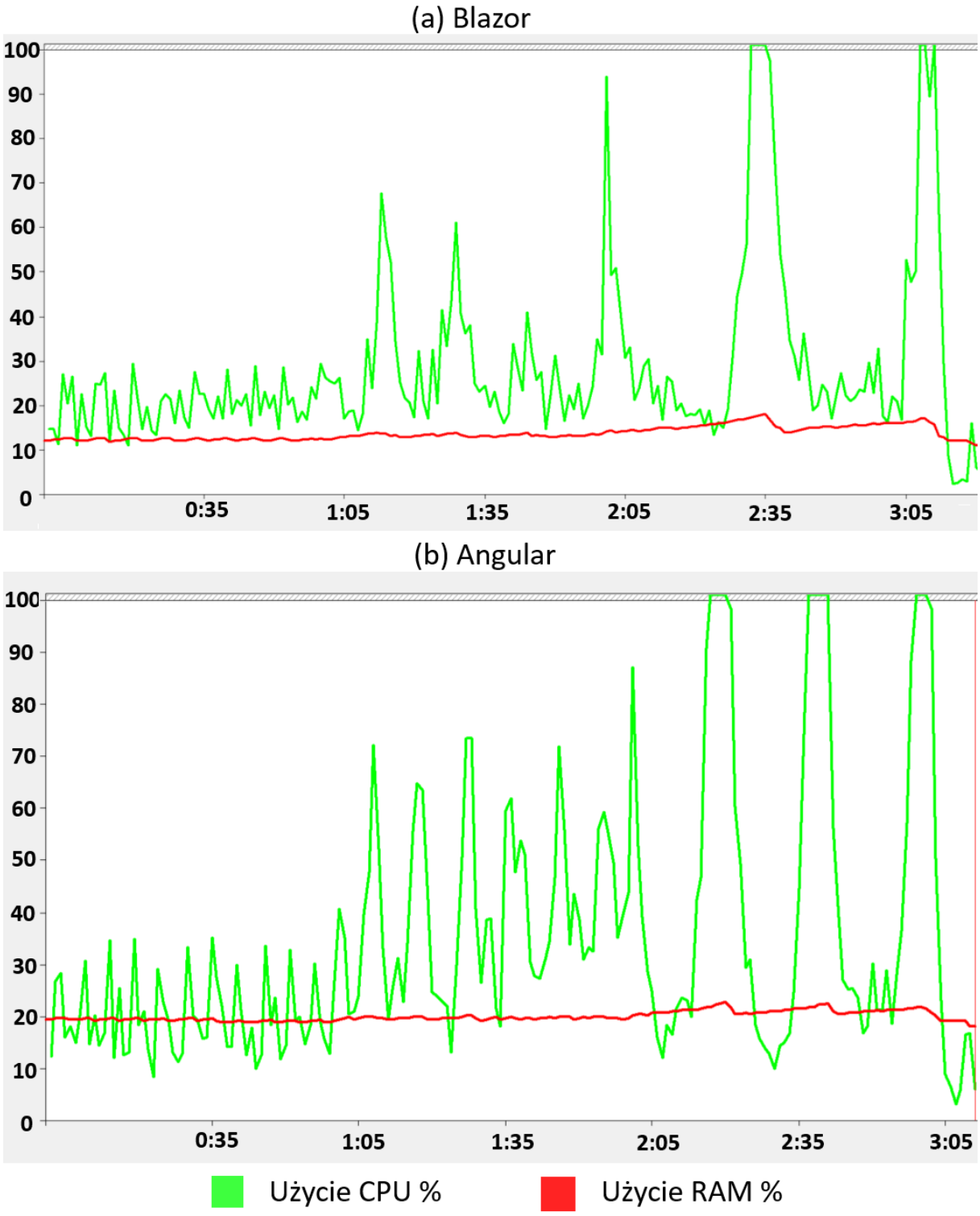
5.2. Dodanie produktu

Tabela 3 przedstawia informacje o obciążeniu komponentów komputerowych podczas testu dodawania nowego produktu. Zużycie pamięci podręcznej przez komputer hostujący aplikację Blazor było aż o 9,3% niższe w porównaniu z aplikacją Angular. Zużycie procesora również okazało się niższe, choć różnica była mniej znacząca. Podobnie komputer testujący aplikację Blazor zużywał mniej zasobów, przy czym największą różnicę zaobserwowano w średnim zużyciu CPU, które wyniosło 6,9%.

Tabela 3. Obciążenie komponentów dla scenariusza 5.2. Źródło: opracowanie własne

	Komputer testujący - Blazor	Komputer hostujący - Blazor	Komputer testujący - Angular	Komputer hostujący - Angular
Pamięć RAM maks. [%]	18	54,6	22,7	64,2
Pamięć RAM śr. [%]	13,7	54,4	20	63,7
Użycie CPU maks. [%]	100	24,9	100	28,4
Użycie CPU śr. [%]	28,7	6,7	35,6	7,6

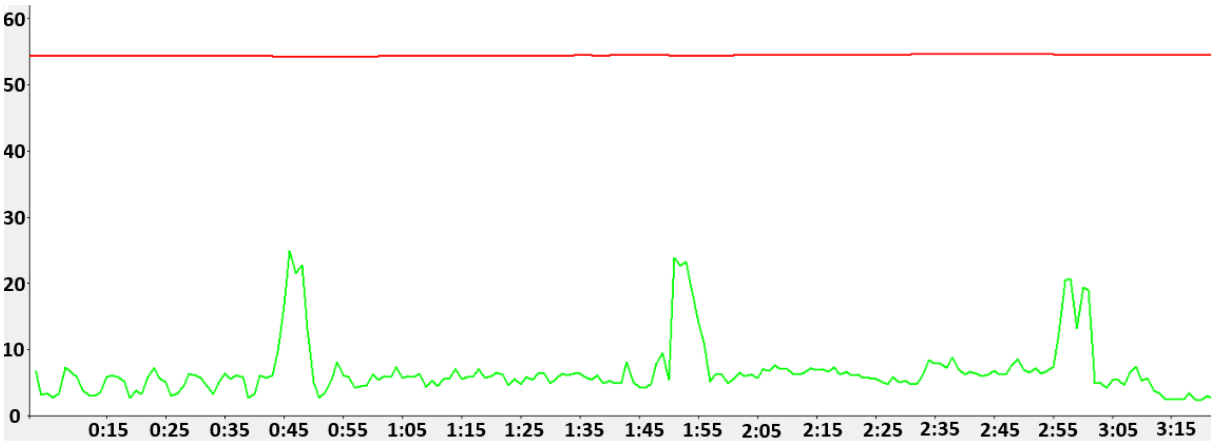
Na rysunku 10 znajdują się dwa wykresy obciążenia komputera testującego aplikacje Angular i Blazor. Aplikacja Blazor spowodowała pięć znaczących nagłych wzrostów zużycia procesora, podczas gdy aplikacja Angular wykazała ich kilkanaście. Bazowe zużycie pamięci podręcznej jest na korzyść aplikacji Blazor.



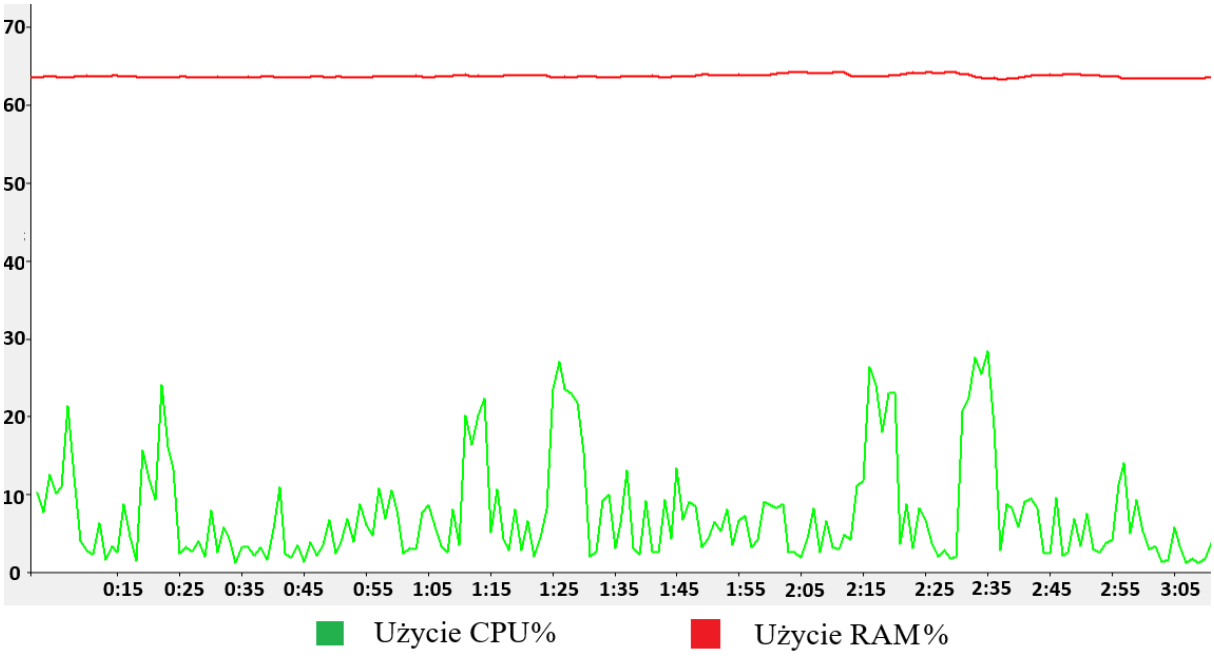
Rysunek 10. Obciążenie komputera testującego Angular dla scenariusza 5.2 (od góry Blazor, Angular). Źródło: opracowanie własne

Na rysunku 11 przedstawiony jest wykres obciążenia komputera hostującego aplikacje Angular i Blazor. Aplikacja Blazor spowodowała trzy mało znaczące skoki wzrostów zużycia procesora, podczas gdy aplikacja Angular w większym stopniu wykorzystywała ten zasób komputera.

(a) Blazor



(b) Angular



Rysunek 11. Obciążenie komputera hosta Angular dla scenariusza 5.2. Źródło: opracowanie własne

W tabeli 4 zaprezentowane zostały wyniki testu obciążeniowego dla scenariusza 5.2. Ponownie zaobserwowano, że aplikacja Angular odnotowuje lepsze wyniki niż Blazor. Różnica jednak nie jest aż tak duża jak w przypadku scenariusza 5.1. Dla minimalnej długości sesji użytkownika aplikacja Angular potrzebowała 4,2 sekundy, podczas gdy aplikacja Blazor wymagała o 0,9 sekundy więcej.

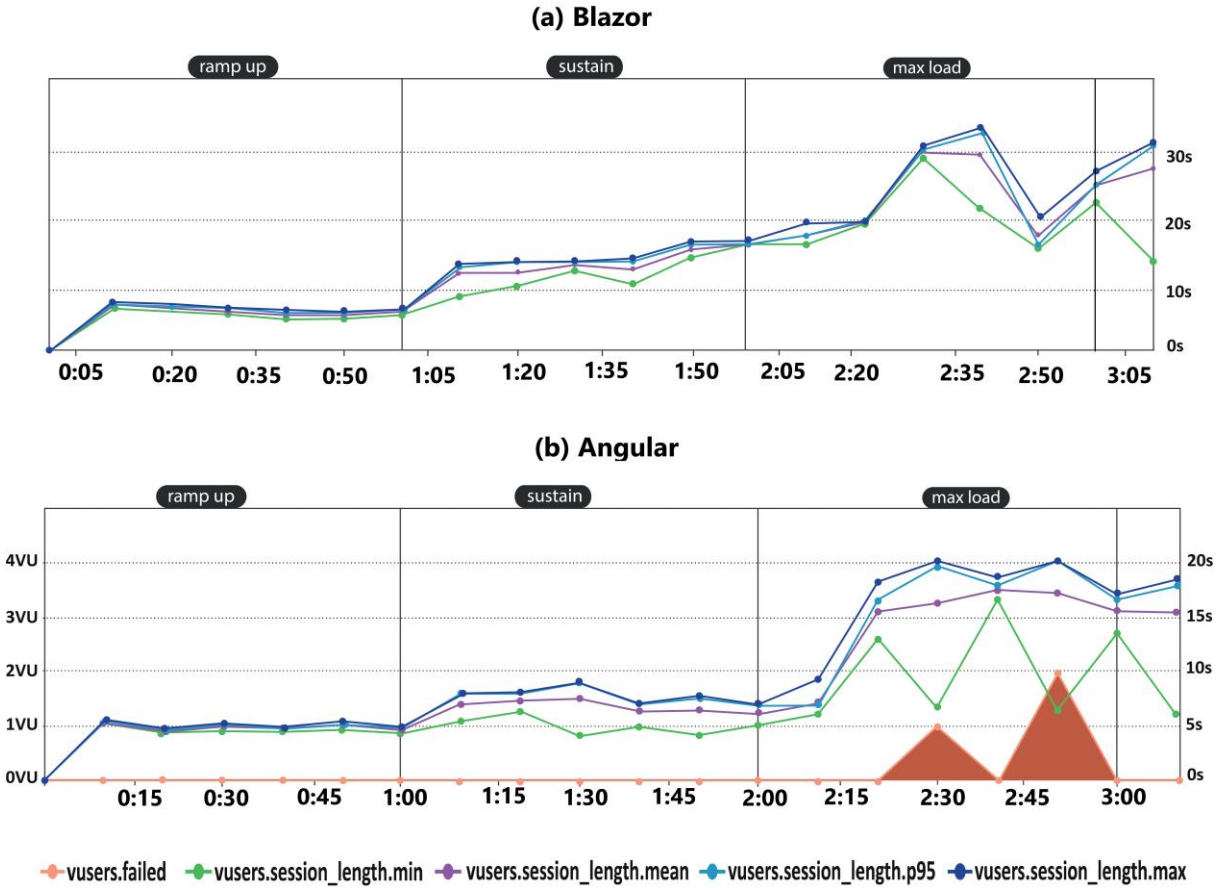
Tabela 4. Wyniki testu obciążającego dla scenariusza 5.2. Źródło: opracowanie własne

	Blazor	Angular
Minimalna długość sesji [s]	5,1	4,2
Średnia długość sesji [s]	19,7	10,5
Długość sesji p95 [s]	33,8	18,9
Maksymalna długość sesji [s]	36,9	20,1
Użytkownicy udani	124	257
Użytkownicy z błędem	0	3

Rysunek 12 przedstawia wykres wyników testu obciążeniowego aplikacji Blazor i Angular dla scenariusza dodawania nowego produktu.

Aplikacja Angular zanotowała trzy przypadki błędów podczas testu, mimo że osiągnęła lepsze czasy odpowiedzi. Fazy *ramp-up* i *sustain* charakteryzowały się stałą, wysoką wydajnością na poziomie około 5 sekund. W końcowej fazie zaobserwowano znaczące wydłużenie czasów trwania scenariuszy oraz wystąpienie błędów.

W przypadku aplikacji Blazor, podczas fazy *ramp-up*, wykazywała ona wysoką wydajność. Czas oczekiwania wzrastał niemal liniowo do połowy fazy *max load*, w której czasy wydłużyły się do poziomu powyżej 30 sekund.



Rysunek 12. Wynik testu obciążającego dla scenariusza 5.2. Źródło: opracowanie własne

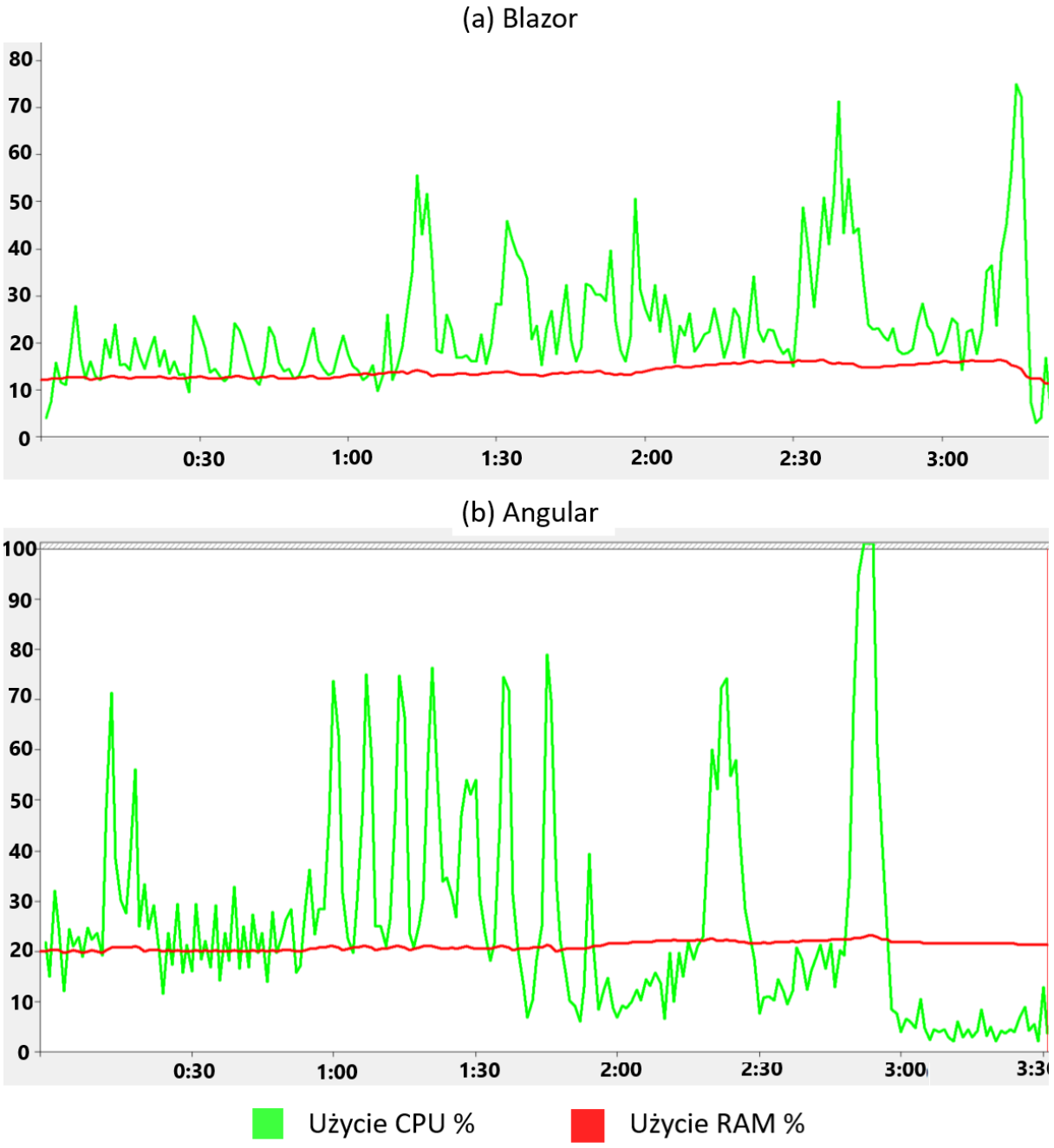
5.3. Modyfikacja produktu

Tabela 5 przedstawia informacje o obciążeniu komponentów komputerowych podczas testu dodawania nowego produktu. W przypadku obu aplikacji komputer testujący charakteryzował się większym zużyciem zasobów procesora niż komputer hostujący. Różnice między maksymalnym a średnim zużyciem pamięci podręcznej dla obu maszyn i obu aplikacji były niewielkie (np. dla komputera testującego aplikację Blazor wyniosły 2,2%).

Tabela 5. Obciążenie komponentów dla scenariusza 5.3. Źródło: opracowanie własne

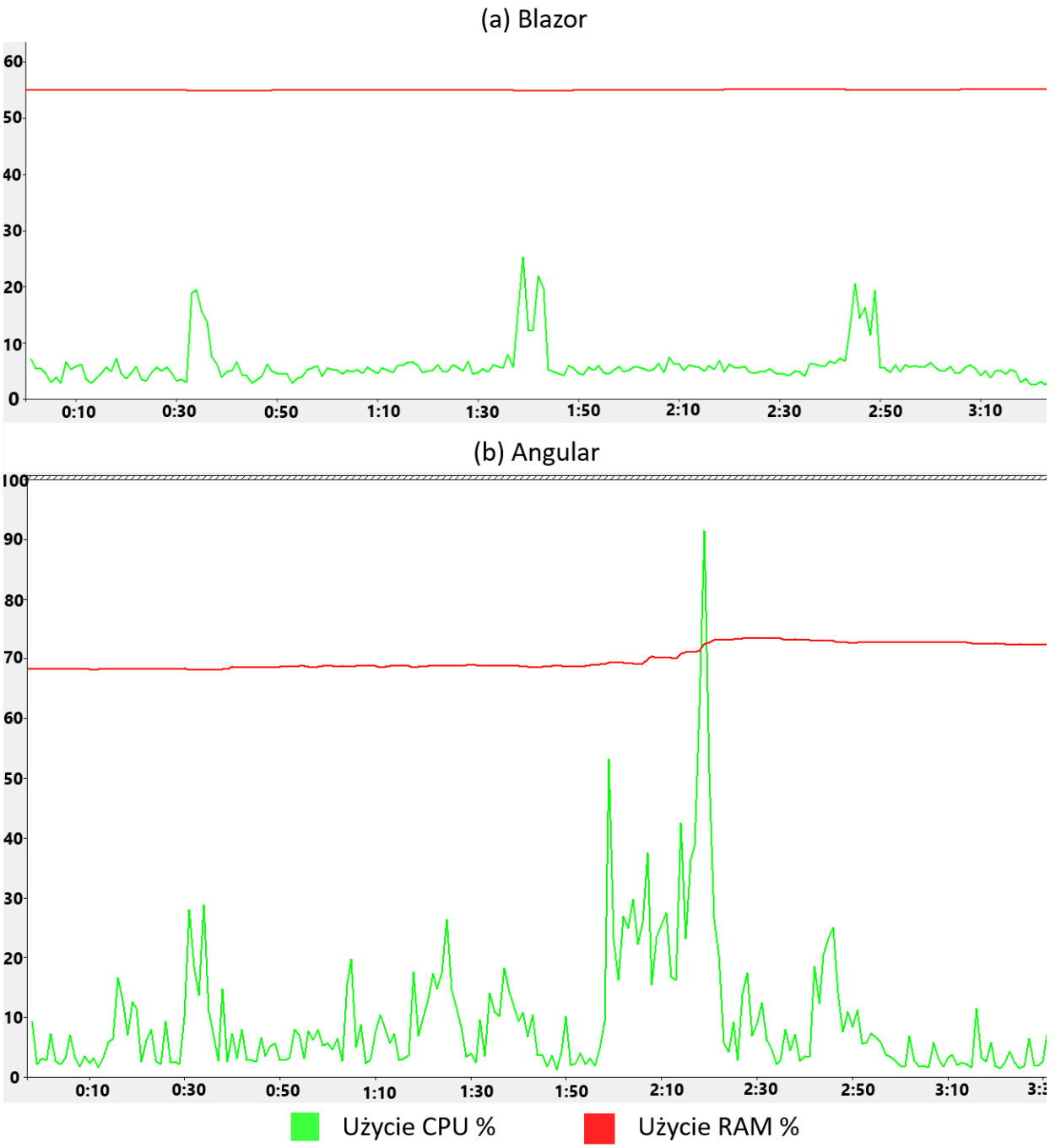
	Komputer testujący - Blazor	Komputer hostujący - Blazor	Komputer testujący - Angular	Komputer hostujący - Angular
Pamięć RAM maks. [%]	16,1	55,2	22,9	73,4
Pamięć RAM śr. [%]	13,9	55	21,1	70,1
Użycie CPU maks. [%]	100	25,3	100	91,4
Użycie CPU śr. [%]	19,9	6	25,9	10,45

Rysunek 13 przedstawia dwa wykresy obciążenia komputera testującego aplikacje Blazor i Angular. W przypadku aplikacji Blazor zużycie zasobów pamięci podręcznej i procesora wzrasta stopniowo wraz z liczbą aktywnych użytkowników. Natomiast wykres dla aplikacji Angular jest bardziej chaotyczny, z nagłymi wysokimi skokami, po których następuje spadek zużycia procesora.



Rysunek 13. Obciążenie komputera testującego dla scenariusza 5.3. Źródło: opracowanie własne

Na rysunku 14 przedstawiono wykres obciążenia komputera hostującego aplikacje Angular i Blazor. Dla wykresu aplikacji Angular można zaobserwować nagły, znaczący wzrost zużycia procesora w okienku 20-sekundowym (między 18:42:20 a 18:42:40).



Rysunek 14. Obciążenie komputera hosta dla scenariusza 5.3. Źródło: opracowanie własne

W tabeli 6 zaprezentowano wyniki testu obciążeniowego dla scenariusza 5.3. Liczba użytkowników z błędem dla obu aplikacji jest porównywalna. Różnica między tymi wartościami wynosi zaledwie dwóch użytkowników na korzyść Blazora, mimo lepszych wyników czasowych Angulara.

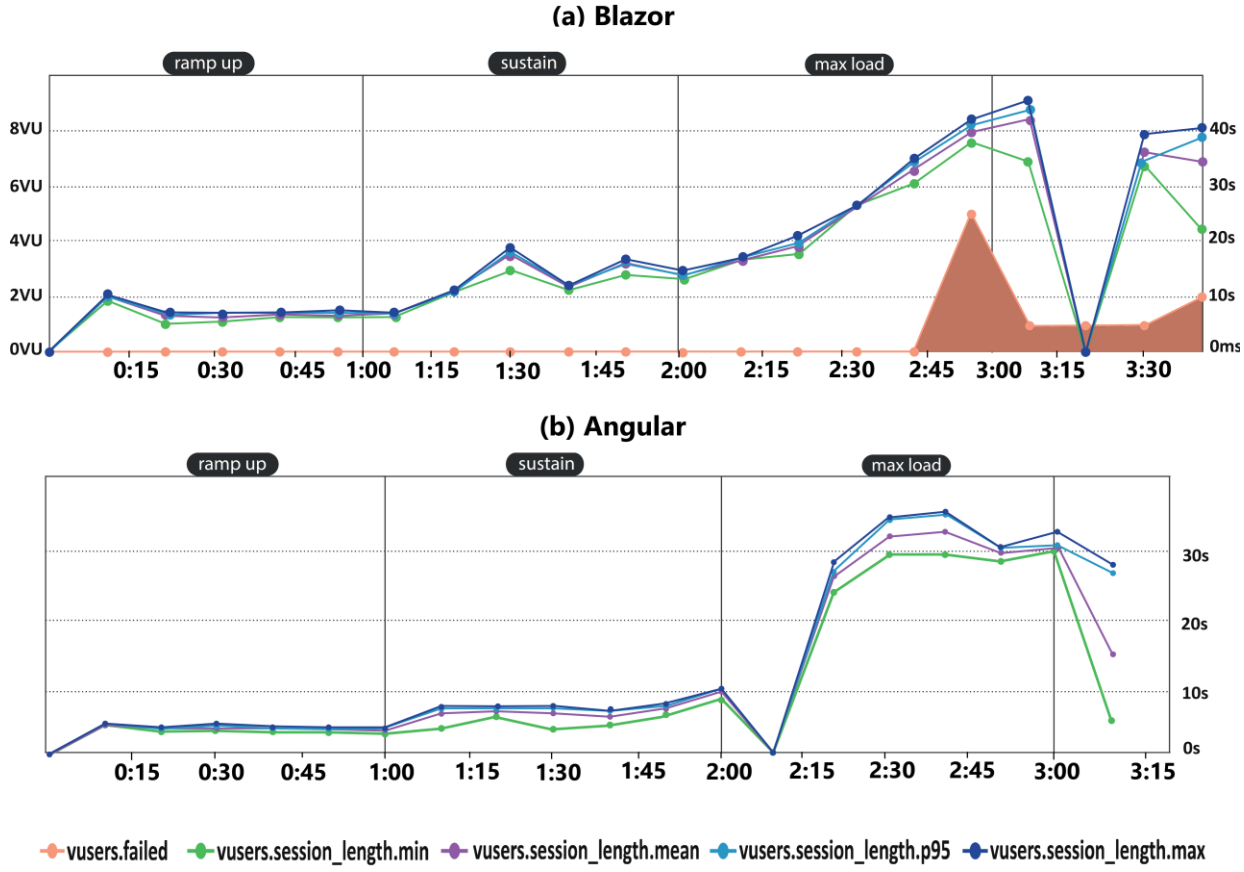
Tabela 6. Wyniki testu obciążającego dla scenariusza 5.3. Źródło: opracowanie własne

	Blazor	Angular
Minimalna długość sesji [s]	5,4	2,9
Średnia długość sesji [s]	22,8	11,3
Długość sesji p95 [s]	43	32,5
Maksymalna długość sesji [s]	45,3	34,7
Użytkownicy udani	99	163
Użytkownicy z błędem	10	12

Na rysunku 15 przedstawiono wykres wyników testu obciążeniowego aplikacji Blazor i Angular dla scenariusza modyfikacji produktu.

W końcowej fazie aplikacja Blazor zaczęła zwracać błędy użytkownikom, co utrzymywało się do końca testu. Pod koniec można zaobserwować spadek czasu trwania sesji do 0 sekund, co wynikało z faktu, że w tym okresie żaden użytkownik nie ukończył testu z powodu zbyt długiego czasu trwania sesji.

Aplikacja Angular, w fazach *ramp-up* i *sustain*, utrzymywała czas trwania sesji poniżej 10 sekund. Obciążenie wynikające z obecności 25 aktywnych użytkowników spowodowało trzykrotne wydłużenie czasu potrzebnego na przejście testu.



Rysunek 15. Wynik testu obciążającego dla scenariusza 5.3. Źródło: opracowanie własne

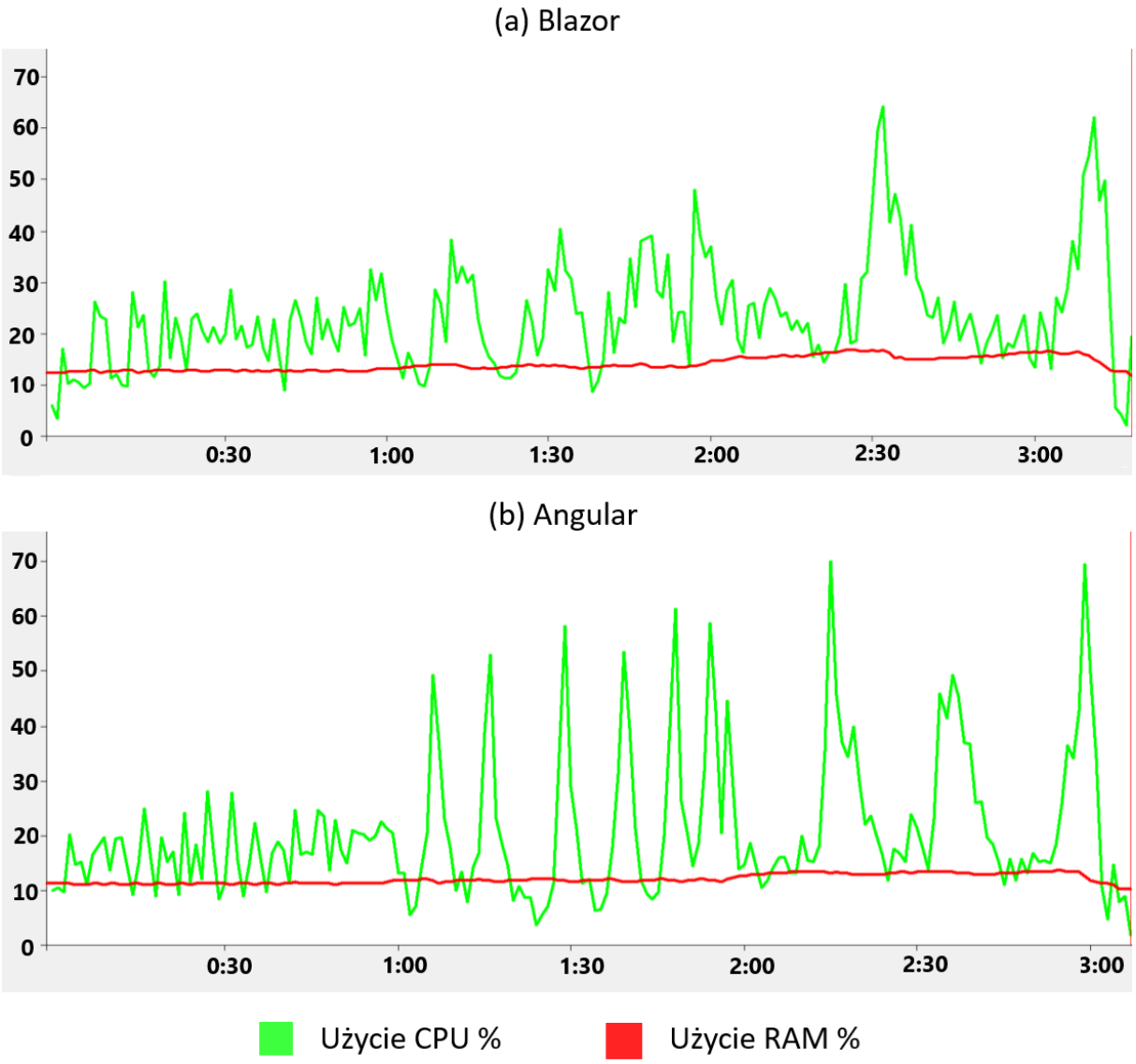
5.4. Usunięcie produktu

W tabeli 7 przedstawiono dane dotyczące obciążenia komponentów komputerowych podczas testu usuwania produktu. Różnica między średnim użyciem pamięci podręcznej komputera testującego a komputera hostującego dla aplikacji Blazor wynosi 41,4%, podczas gdy dla aplikacji Angular jest to 51,3%.

Tabela 7. Obciążenie komponentów dla scenariusza 5.4. Źródło: opracowanie własne

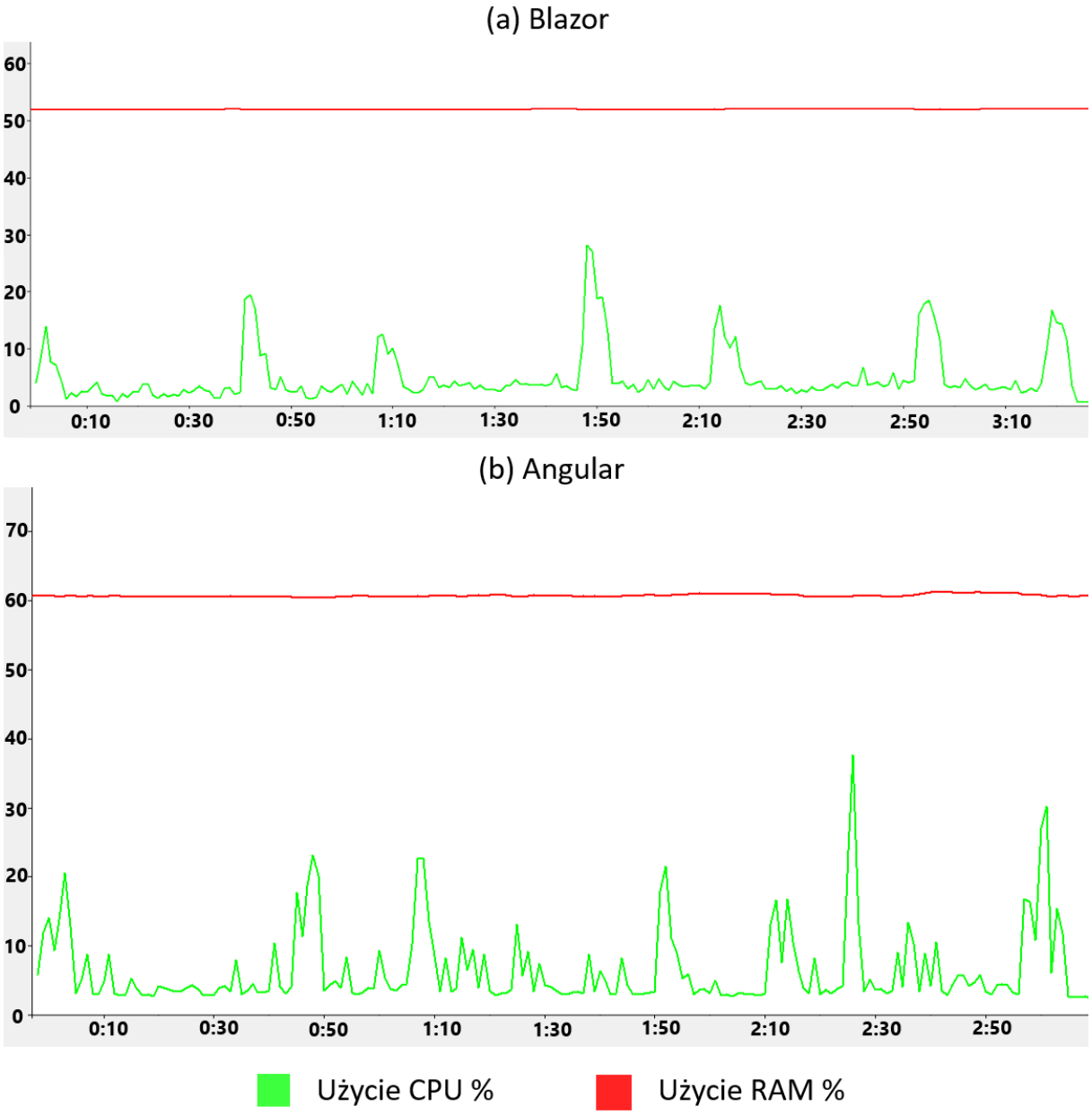
	Komputer testujący - Blazor	Komputer hostujący - Blazor	Komputer testujący - Angular	Komputer hostujący - Angular
Pamięć RAM maks. [%]	16,8	55,6	13,6	63,9
Pamięć RAM śr. [%]	14	55,4	12	63,3
Użycie CPU maks. [%]	64,4	24	70	31,7
Użycie CPU śr. [%]	23,1	6,4	20,9	7,2

Rysunek 16 przedstawia dwa wykresy obciążenia komputera testującego aplikację Blazor i Angular. Dla obu aplikacji widoczny jest stały wzrost procentowego zużycia pamięci podręcznej oraz procesora. Skoki w użyciu CPU są jednak znacznie większe dla aplikacji Angular.



Rysunek 16. Obciążenie komputera testującego dla scenariusza 5.4. Źródło: opracowanie własne

Na rysunku 17 przedstawiono wykresy obciążenia komputera hostującego aplikacje Blazor i Angular. Zużycie pamięci RAM dla obu aplikacji nie odbiega znacząco od początkowych wartości. Oznacza to, że zwiększanie liczby aktywnych użytkowników ma niewielki wpływ na pamięć podręczną.



Rysunek 17. Obciążenie komputera hosta dla scenariusza 5.4. Źródło: opracowanie własne

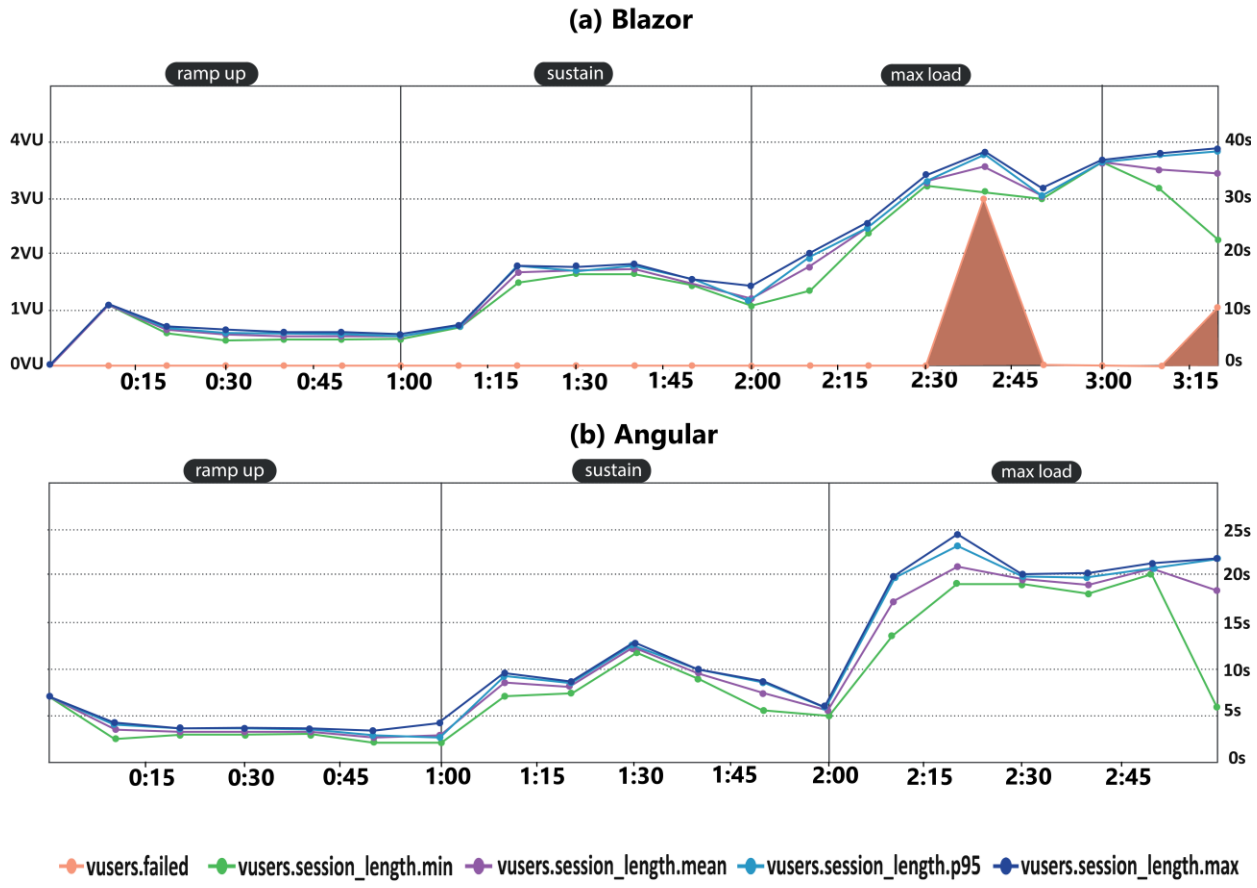
Tabela 8 prezentuje wyniki testu obciążeniowego dla scenariusza usuwania produktu. Różnica w liczbie użytkowników, którym udało się przejść test (70), wskazuje na wyższą wydajność aplikacji Angular. Tezę tę potwierdzają dane dotyczące długości sesji.

Tabela 8. Wyniki testu obciążającego dla scenariusza 5.4. Źródło: opracowanie własne

	Blazor	Angular
Minimalna długość sesji [s]	4,3	2
Średnia długość sesji [s]	20,7	11,8

Długość sesji p95 [s]	37,4	21,3
Maksymalna długość sesji [s]	38,4	24,4
Użytkownicy udani	116	186
Użytkownicy z błędem	4	0

Na rysunku 18 przedstawiono wykresy wyników testu obciążeniowego aplikacji Blazor i Angular dla scenariusza 5.4. Aplikacja Angular zakończyła test w ciągu 180 sekund, po przejściu przez wszystkie trzy fazy. Aplikacja Blazor potrzebowała dodatkowego czasu po zakończeniu fazy *max load*, aby wcześniej utworzone sesje mogły zakończyć scenariusz.



Rysunek 18. Wynik testu obciążającego dla scenariusza 5.4. Źródło: opracowanie własne

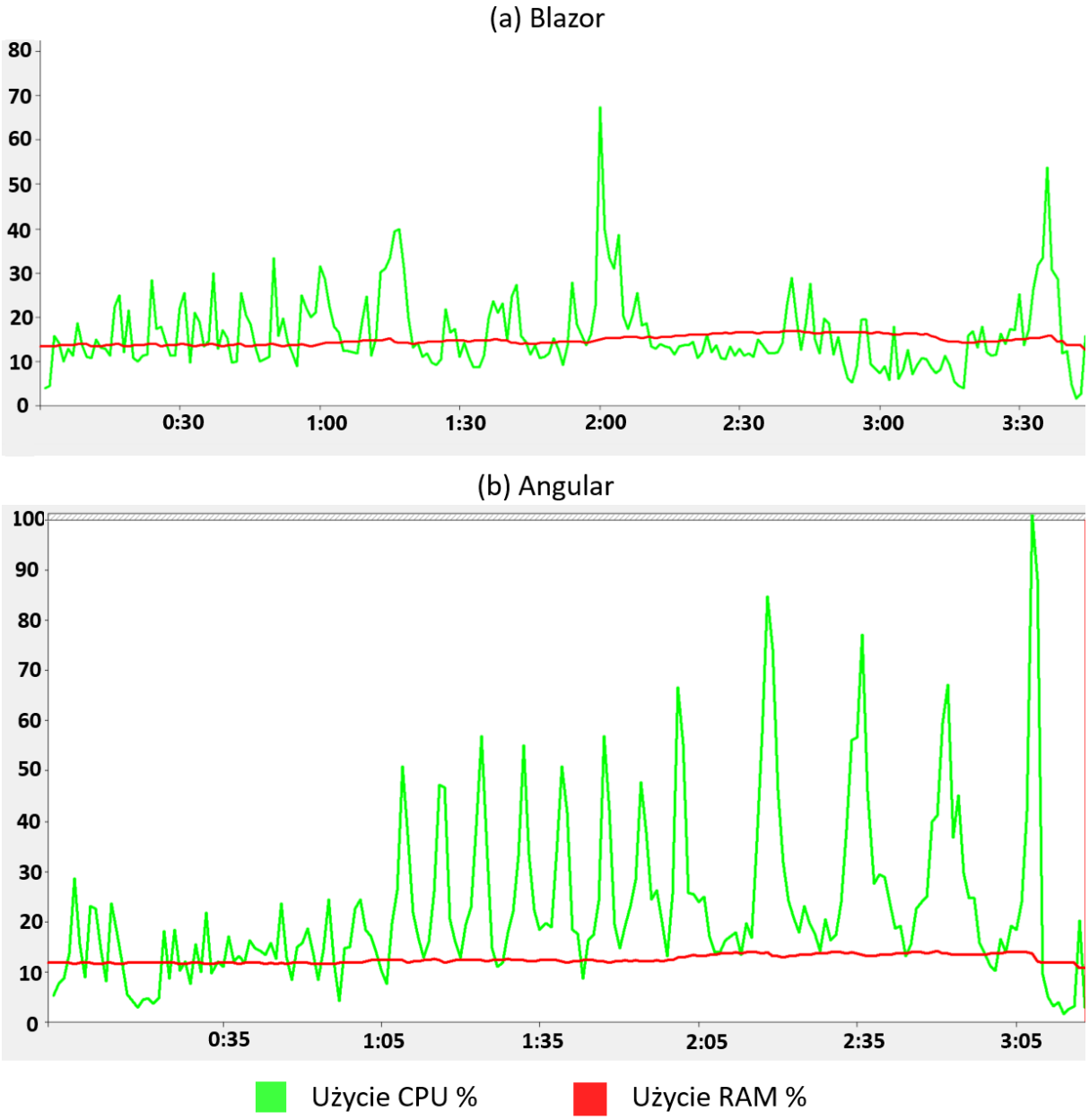
5.5. Dodanie produktu do koszyka

W tabeli 9 przedstawiono dane o obciążeniu komponentów komputerowych podczas testu dodawania produktu do koszyka. Aplikacja Blazor odnotowała lepsze wyniki pod względem wykorzystania procesora na obu maszynach. Dla komputera testującego maksymalne obciążenie CPU wyniosło 67,4%, podczas gdy w przypadku aplikacji Angular wartość ta osiągnęła 100%.

Tabela 9. Obciążenie komponentów dla scenariusza 5.6. Źródło: opracowanie własne

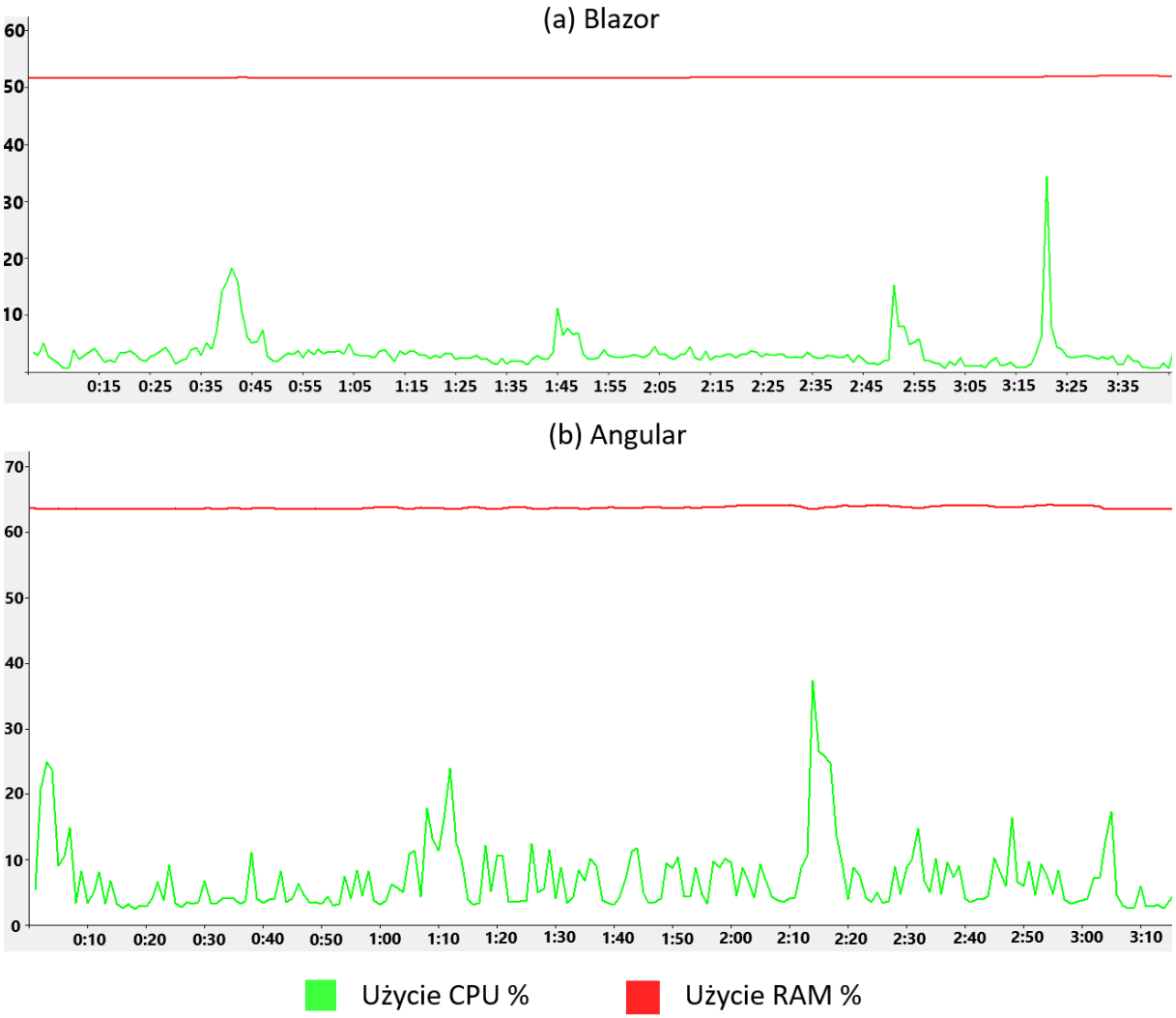
	Komputer testujący - Blazor	Komputer hostujący - Blazor	Komputer testujący - Angular	Komputer hostujący - Angular
Pamięć RAM maks. [%]	16,9	52	14	64,2
Pamięć RAM śr. [%]	14,8	51,7	12,5	63,8
Użycie CPU maks. [%]	67,4	34,4	100	37,4
Użycie CPU śr. [%]	16,2	3,4	23	7

Rysunek 19 przedstawia dwa wykresy obciążenia komputera testującego aplikację Blazor i Angular. Użycie procesora w aplikacji Angular charakteryzuje się wysokimi i częstymi skokami. Blazor wykazuje większą stabilność, mniejszą częstotliwość skoków oraz ich amplitudę.



Rysunek 19. Obciążenie komputera testującego dla scenariusza 5.6. Źródło: opracowanie własne

Na rysunku 20 przedstawiono wykresy obciążenia komputera hostującego aplikację Angular i Blazor. Zarówno obciążenie procesora, jak i pamięci podręcznej jest większe dla aplikacji Angular niż dla Blazora.



Rysunek 20. Obciążenie komputera hosta dla scenariusza 5.6. Źródło: opracowanie własne

Tabela 10 przedstawia wyniki testu obciążeniowego dla scenariusza dodawania produktu do koszyka. Różnica między minimalną długością sesji wynosi 2 sekundy. Mimo że wartość ta nie jest mała, ponieważ stanowi prawie dwukrotność (3 do 5 sekund), to różnica między maksymalną długością sesji jest jeszcze większa. Wynosi ona aż 52,5 sekundy, co oznacza, że Blazor w najgorszym przypadku potrzebował prawie 3 razy więcej czasu na ukończenie scenariusza niż aplikacja Angular.

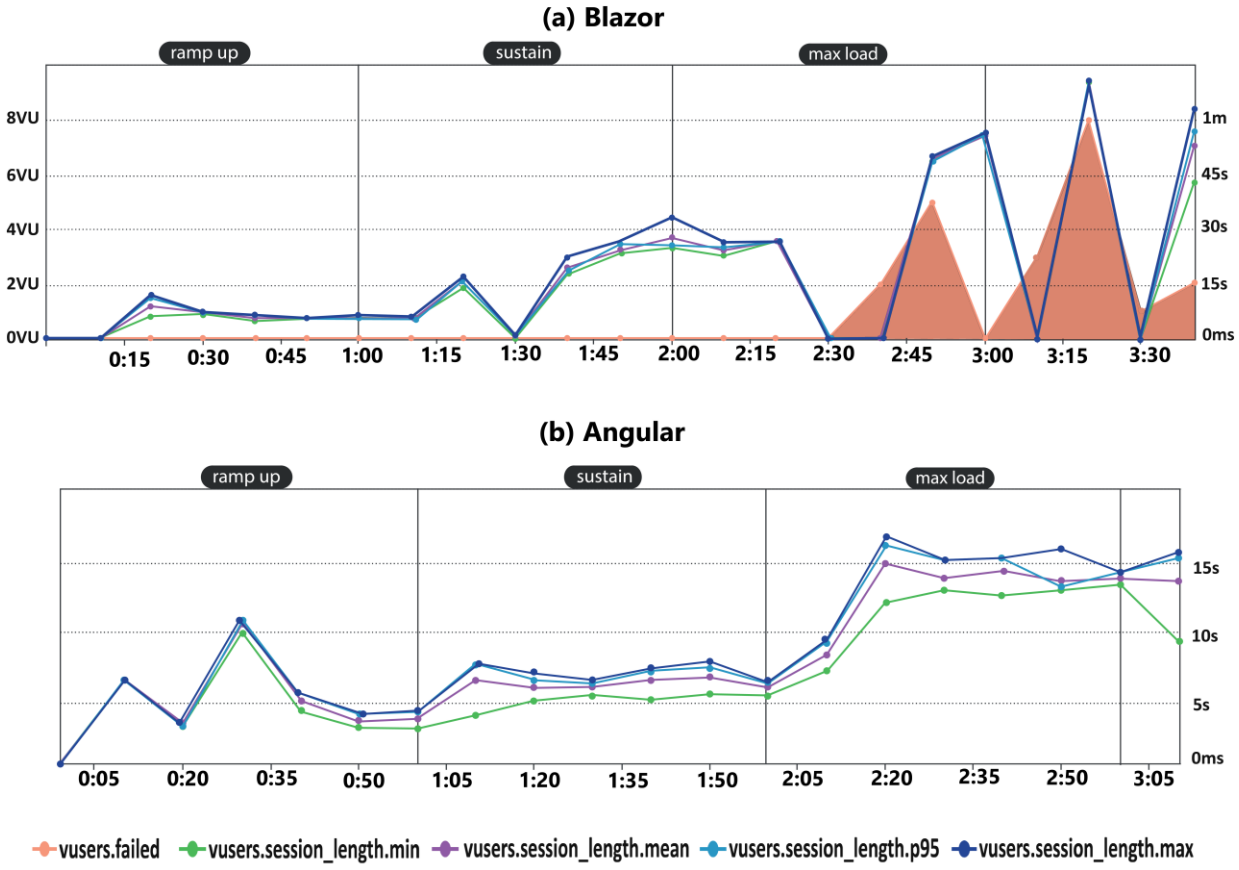
Tabela 10. Wyniki testu obciążającego dla scenariusza 5.6. Źródło: opracowanie własne

	Blazor	Angular
Minimalna długość sesji [s]	5	3
Średnia długość sesji [s]	23,7	10,6
Długość sesji p95 [s]	56,9	16,8

Maksymalna długość sesji [s]	71	18,5
Użytkownicy udani	70	217
Użytkownicy z błędem	21	0

Na rysunku 21 przedstawiono wykresy wyniku testu obciążeniowego aplikacji Angular i Blazor dla scenariusza 5.6. W fazie *ramp-up* można zaobserwować duże wahania czasu dla aplikacji Angular. Blazor wykazał mniejszy rozrzut, lecz mimo to jego czasy w tym okresie nie były znacząco lepsze.

Momentem krytycznym w teście jest mniej więcej połowa trwania fazy *max load*. Aplikacja Angular poradziła sobie z tym wyzwaniem, zwiększając czasy sesji do około 15 sekund. Blazor natomiast zaliczył gigantyczny spadek wydajności, co w rezultacie spowodowało pojawieniem się aż 21 użytkowników, którzy otrzymali błędy.



Rysunek 21. Wynik testu obciążającego dla scenariusza 5.6. Źródło: opracowanie własne

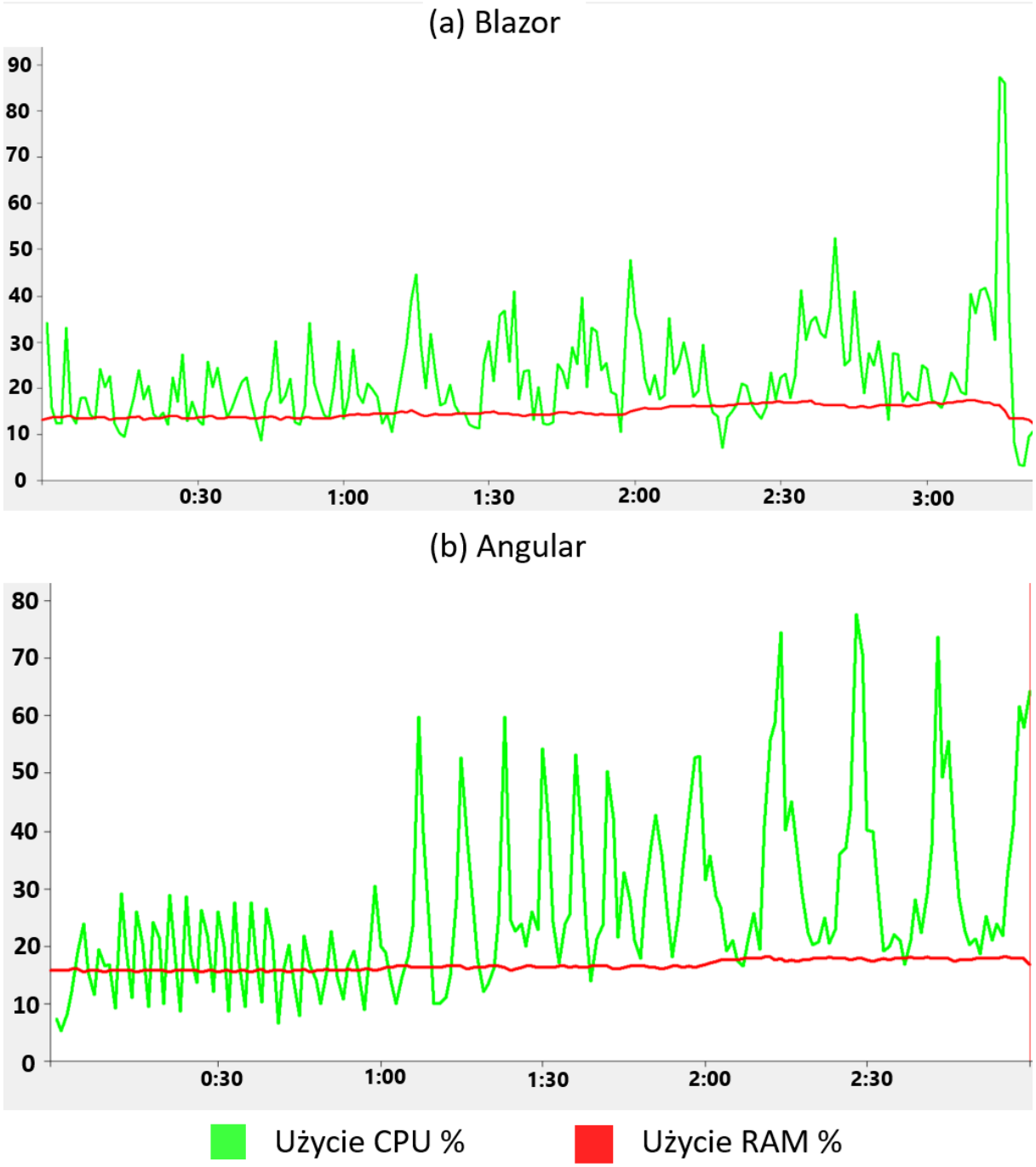
5.6. Wczytanie ekranu koszyk

Tabela 11 przedstawia informacje o obciążeniu komponentów komputerowych podczas testu wczytywania ekranu koszyka. Pomimo niższego średniego zużycia procesora przez komputer testujący aplikację Blazor, odnotowano większe maksymalne wykorzystanie CPU w porównaniu z aplikacją Angular (87,3% w Blazorze wobec 77,6% w Angularze). Różnica w średnim zużyciu procesora dla komputera hostującego wyniosła 4,3% na korzyść Blazora.

Tabela 11. Obciążenie komponentów dla scenariusza 5.5. Źródło: opracowanie własne

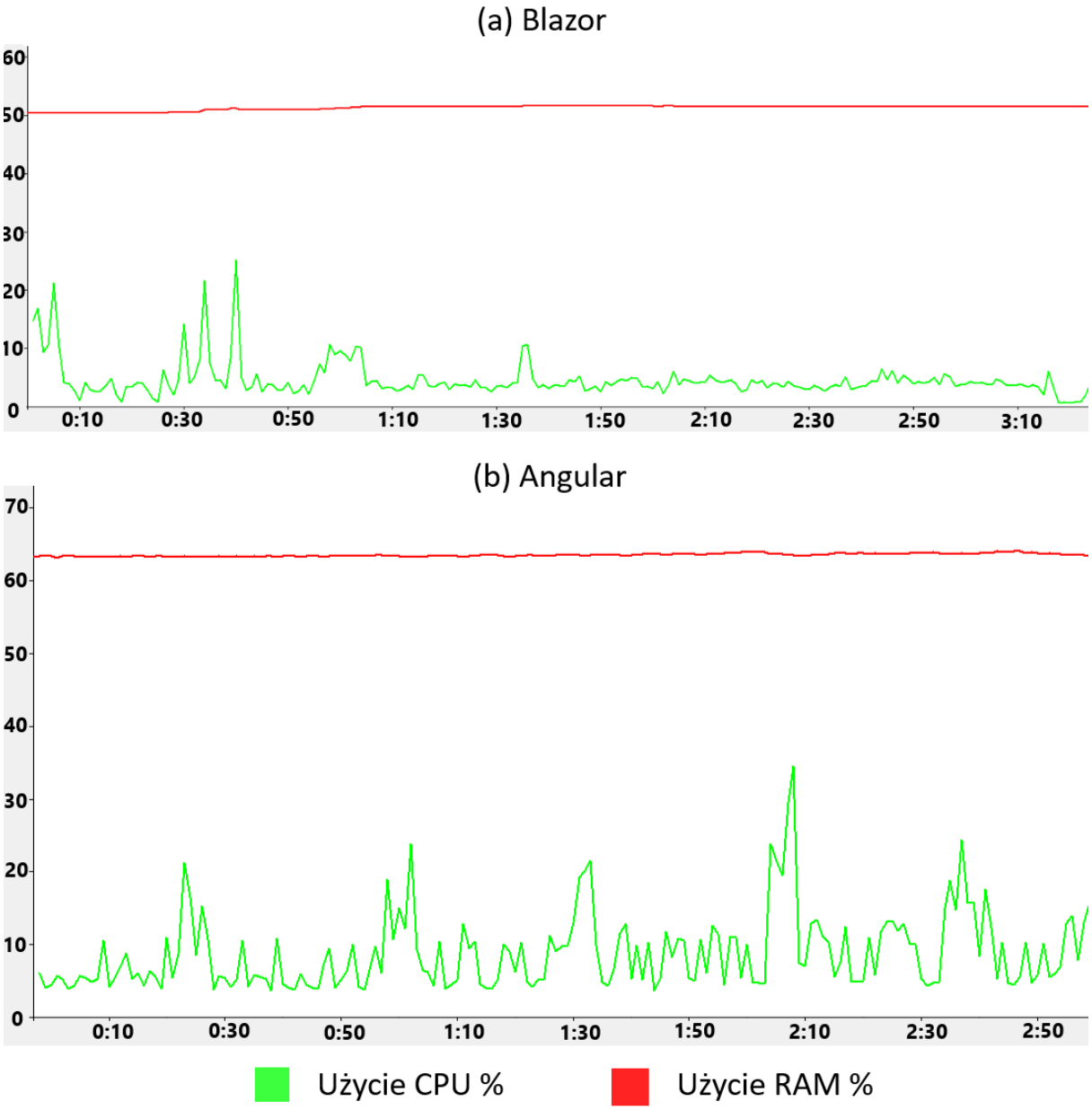
	Komputer testujący - Blazor	Komputer hostujący - Blazor	Komputer testujący - Angular	Komputer hostujący - Angular
Pamięć RAM maks. [%]	17,4	51,6	18	64
Pamięć RAM śr. [%]	14,9	51,3	16,6	63,5
Użycie CPU maks. [%]	87,3	25,2	77,6	34,5
Użycie CPU śr. [%]	22,3	4,6	26,4	8,9

Rysunek 22 przedstawia dwa wykresy obciążenia komputera testującego aplikację Blazor i Angular. Obciążenie zarówno procesora, jak i pamięci podręcznej, jest większe dla aplikacji Angular niż Blazor.



Rysunek 22. Obciążenie komputera testującego dla scenariusza 5.5. Źródło: opracowanie własne

Na rysunku 23 przedstawiono wykresy obciążenia komputera hostującego aplikację Blazor i Angular. Obciążenie procesora dla aplikacji Angular jest większe niż dla Blazora. Dodatkowo aplikacja Angular zużywała o ponad 10% więcej pamięci podręcznej komputera.



Rysunek 23. Obciążenie komputera hosta dla scenariusza 5.5. Źródło: opracowanie własne

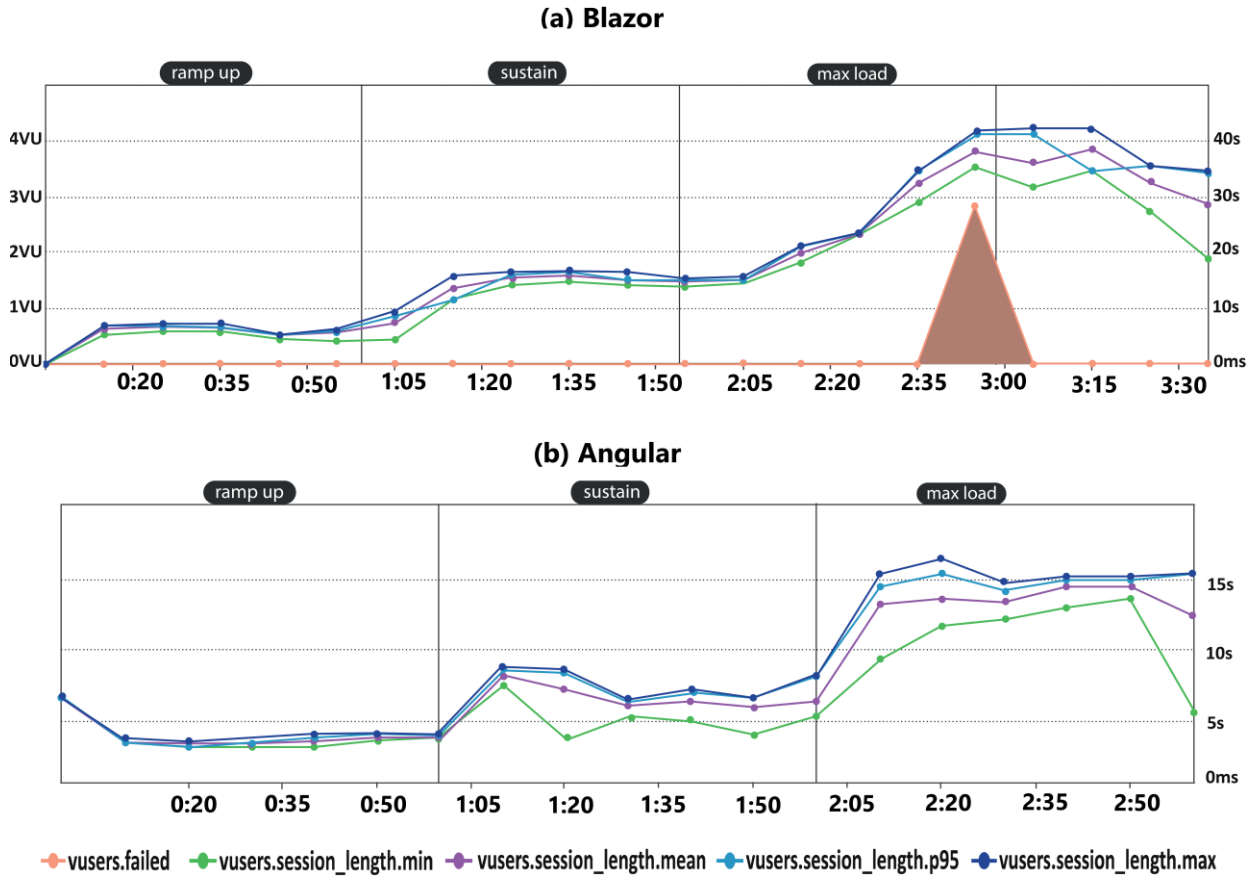
W tabeli 12 przedstawiono wyniki testu obciążeniowego dla scenariusza wczytywania ekranu koszyka. Aplikacja Angular osiągnęła ponad dwukrotnie wyższą liczbę użytkowników, którym udało się zakończyć sesję bez błędów, w porównaniu do aplikacji Blazor. Dodatkowo, Angular nie odnotował żadnych użytkowników z błędami, podczas gdy w przypadku Blazora wystąpiły trzy takie przypadki.

Tabela 12. Wyniki testu obciążającego dla scenariusza 5.5. Źródło: opracowanie własne

	Blazor	Angular
Minimalna długość sesji [s]	4,3	2,6

Średnia długość sesji [s]	22,3	8,5
Długość sesji p95 [s]	42,2	14,6
Maksymalna długość sesji [s]	45,1	16,1
Użytkownicy udani	112	249
Użytkownicy z błędem	3	0

Na rysunku 24 przedstawiono wykresy wyników testu obciążeniowego aplikacji Angular i Blazor dla scenariusza 5.5. W przypadku aplikacji Blazor, w okresie między 2:35 a 3:15, pojawiło się trzech użytkowników, u których wystąpił błąd. Wykres sesji dla Blazora jest bardziej łagodny, mimo wyższych czasów odpowiedzi. Aplikacja Angular zakończyła test w ramach trzech faz, podczas gdy Blazor potrzebował dodatkowych 20 sekund.



Rysunek 24. Wynik testu obciążającego dla scenariusza 5.5. Źródło: opracowanie własne

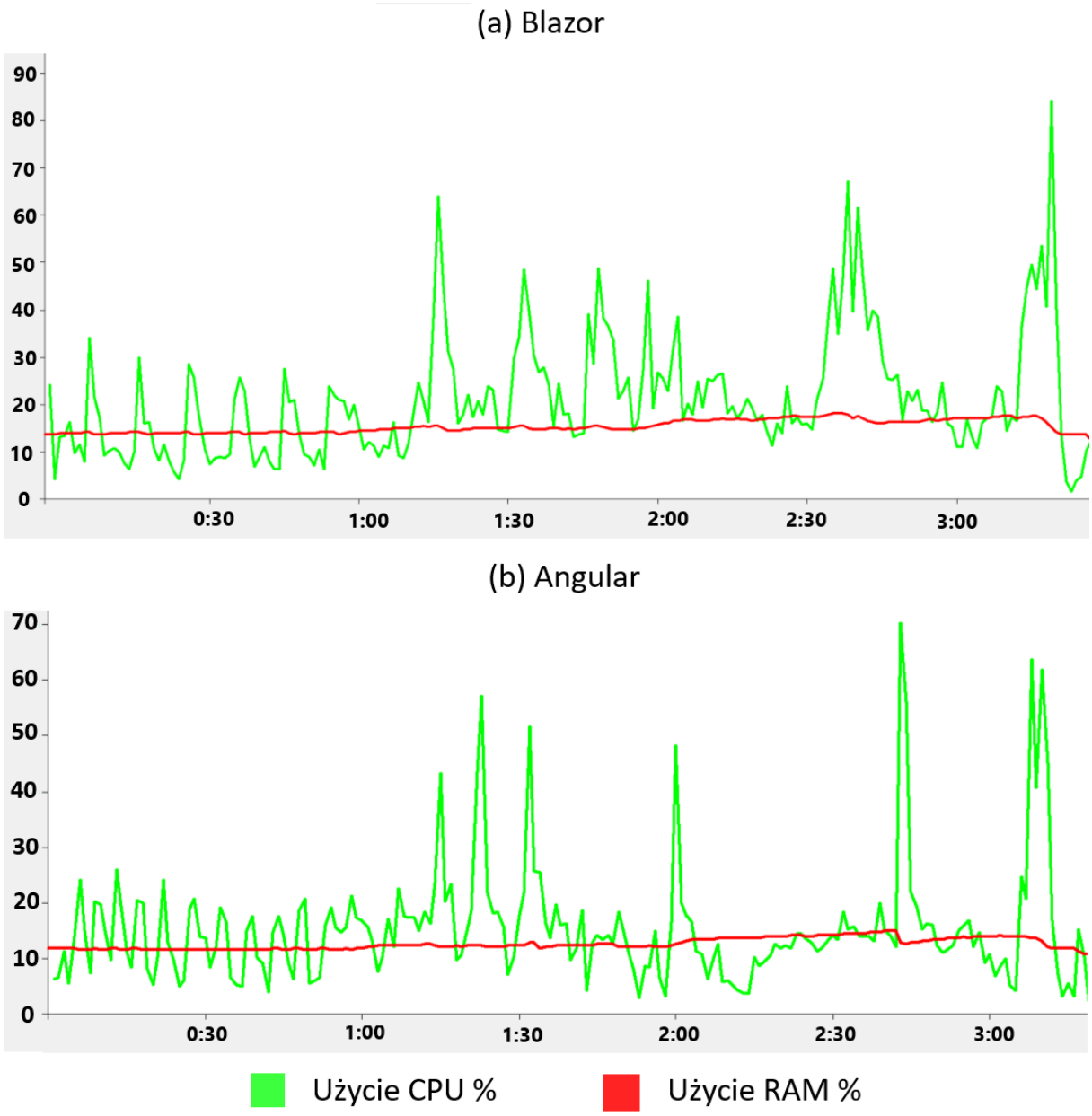
5.7. Zamówienie zawartości koszyka

Tabela 13 przedstawia informacje o obciążeniu komponentów komputerowych podczas wykonywania testu zamawiania zawartości koszyka. Blazor wykorzystał więcej zasobów na komputerze testującym w porównaniu do Angulara, lecz mniej na komputerze hostującym. Różnice są porównywalne do pozostałych scenariuszy.

Tabela 13. Obciążenie komponentów dla scenariusza 5.7. Źródło: opracowanie własne

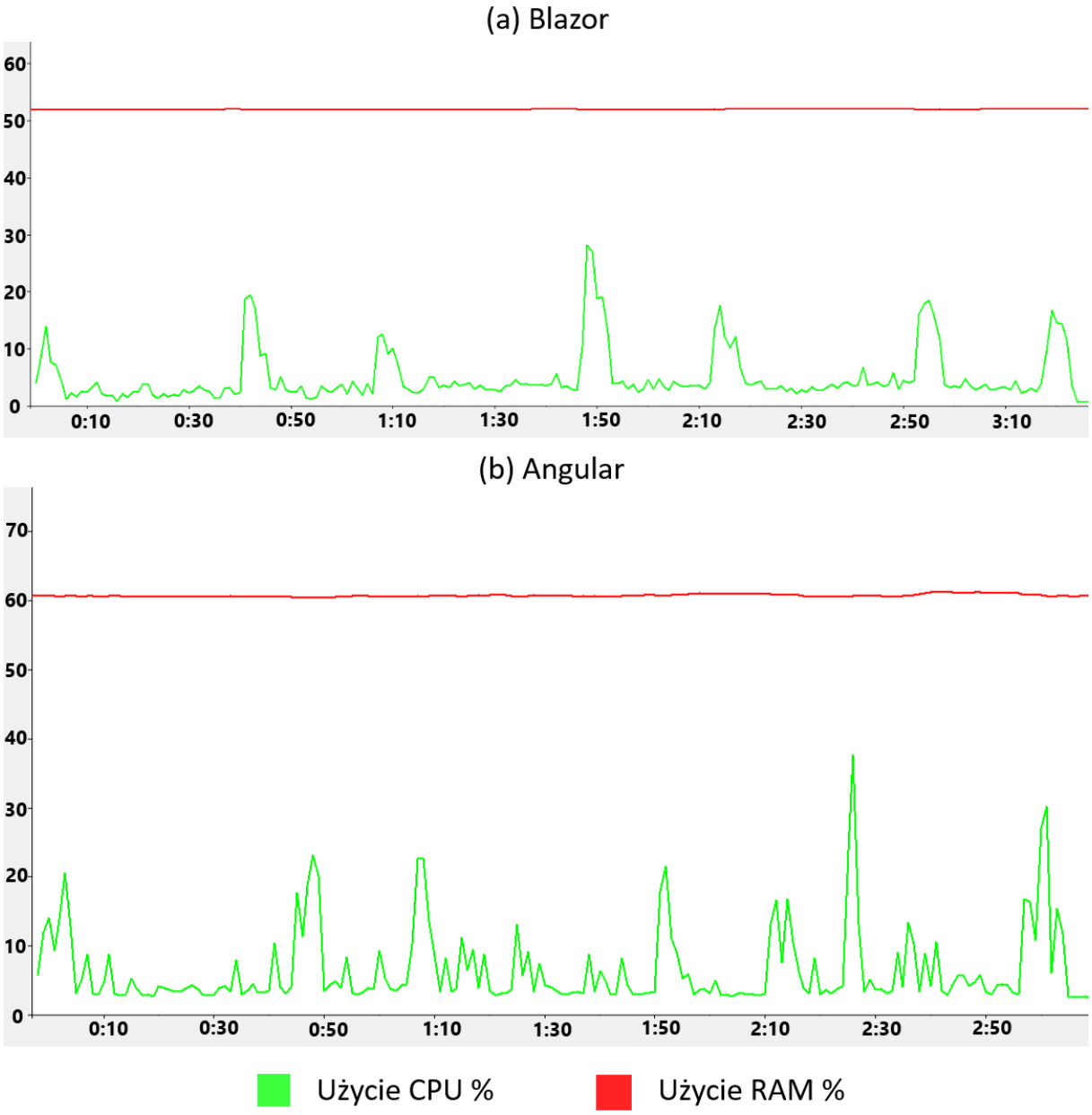
	Komputer testujący - Blazor	Komputer hostujący - Blazor	Komputer testujący - Angular	Komputer hostujący - Angular
Pamięć RAM maks. [%]	18,2	52	15	61,2
Pamięć RAM śr. [%]	15,3	51,9	12,6	60,7
Użycie CPU maks. [%]	84,3	28,1	70,2	37,6
Użycie CPU śr. [%]	21,1	5	15,3	6,7

Rysunek 25 przedstawia dwa wykresy obciążenia komputera testującego aplikację Blazor i Angular. W porównaniu do pozostałych scenariuszy, użycie CPU dla obu aplikacji nie sięgnęło 100% przez cały okres trwania testu.



Rysunek 25. Obciążenie komputera testującego dla scenariusza 5.7. Źródło: opracowanie własne

Na rysunku 26 przedstawiono wykresy obciążenia komputera hostującego aplikację Angular i Blazor. Aplikacja Angular wykazuje większą ilość skoków w użyciu procesora w porównaniu do aplikacji Blazor. Oba wykresy obciążenia RAM są stałe, z niewielkimi zmianami procentowymi.



Rysunek 26. Obciążenie komputera hosta dla scenariusza 5.7. Źródło: opracowanie własne

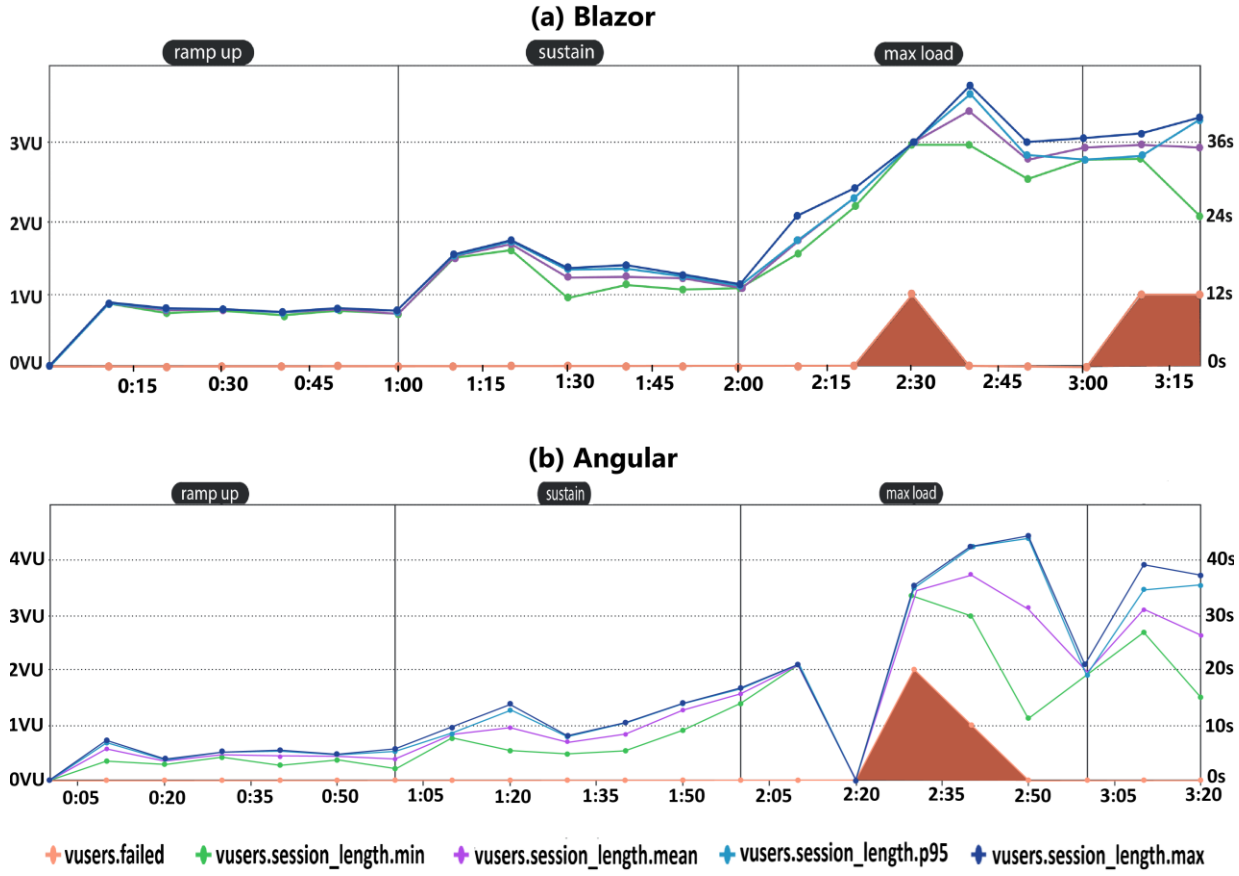
Tabela 14 przedstawia wyniki testu obciążeniowego dla scenariusza zamawiania zawartości koszyka. Obie aplikacje mają równą liczbę użytkowników, u których wystąpił błąd (3). Różnice między maksymalną długością sesji a p95 są niewielkie. Największą różnicę zanotowano w minimalnej długości sesji. Wynik 8,5 sekundy dla Blazora jest prawie czterokrotnością wyniku Angulara, który wynosi 2,2 sekundy.

Tabela 14. Wyniki testu obciążającego dla scenariusza 5.7. Źródło: opracowanie własne

	Blazor	Angular
Minimalna długość sesji [s]	8,5	2,2

Średnia długość sesji [s]	24,8	16,1
Długość sesji p95 [s]	42,2	41,3
Maksymalna długość sesji [s]	45,3	43,9
Użytkownicy udani	105	143
Użytkownicy z błędem	3	3

Na rysunku 27 przedstawiono wykresy wyników testu obciążeniowego aplikacji Angular i Blazor dla scenariusza 5.7. W fazie *max load* w aplikacji Blazor długość sesji stale rosła, a po drodze doszło do jednego użytkownika z błędem, podczas gdy pozostałe dwa przypadki wystąpiły po zakończeniu fazy. Aplikacja Angular wykazała znaczący spadek wydajności w omawianym przedziale. Rezultatem tego był brak zakończonych sesji w okolicach czasu 2:20 oraz pojawienie się błędu u trzech użytkowników.



Rysunek 27. Wynik testu obciążającego dla scenariusza 5.7. Źródło: opracowanie własne

5.8. Wczytanie ekranu zamówienia

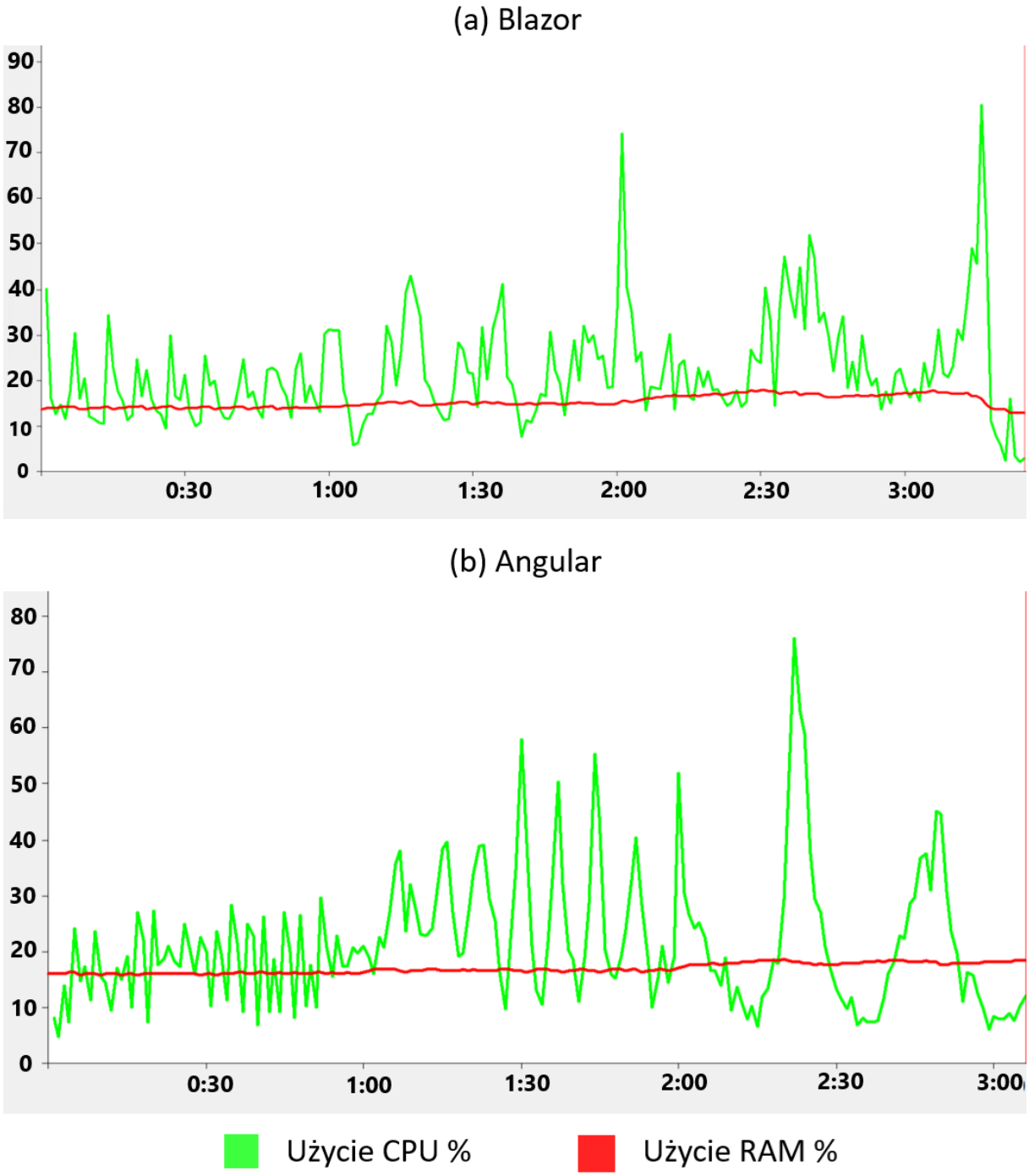
Tabela 15 przedstawia dane o obciążeniu komponentów komputerowych podczas wykonywania scenariusza wczytywania ekranu zamówienia. Różnice dla komputera testującego między dwoma aplikacjami są niewielkie. Zarówno zużycie pamięci podręcznej, jak i procesora nie odbiegały

znacząco od siebie. Większa rozbieżność zanotowano dla komputera hostującego, gdzie to Blazor uzyskał lepszy wynik.

Tabela 15. Obciążenie komponentów dla scenariusza 5.8. Źródło: opracowanie własne

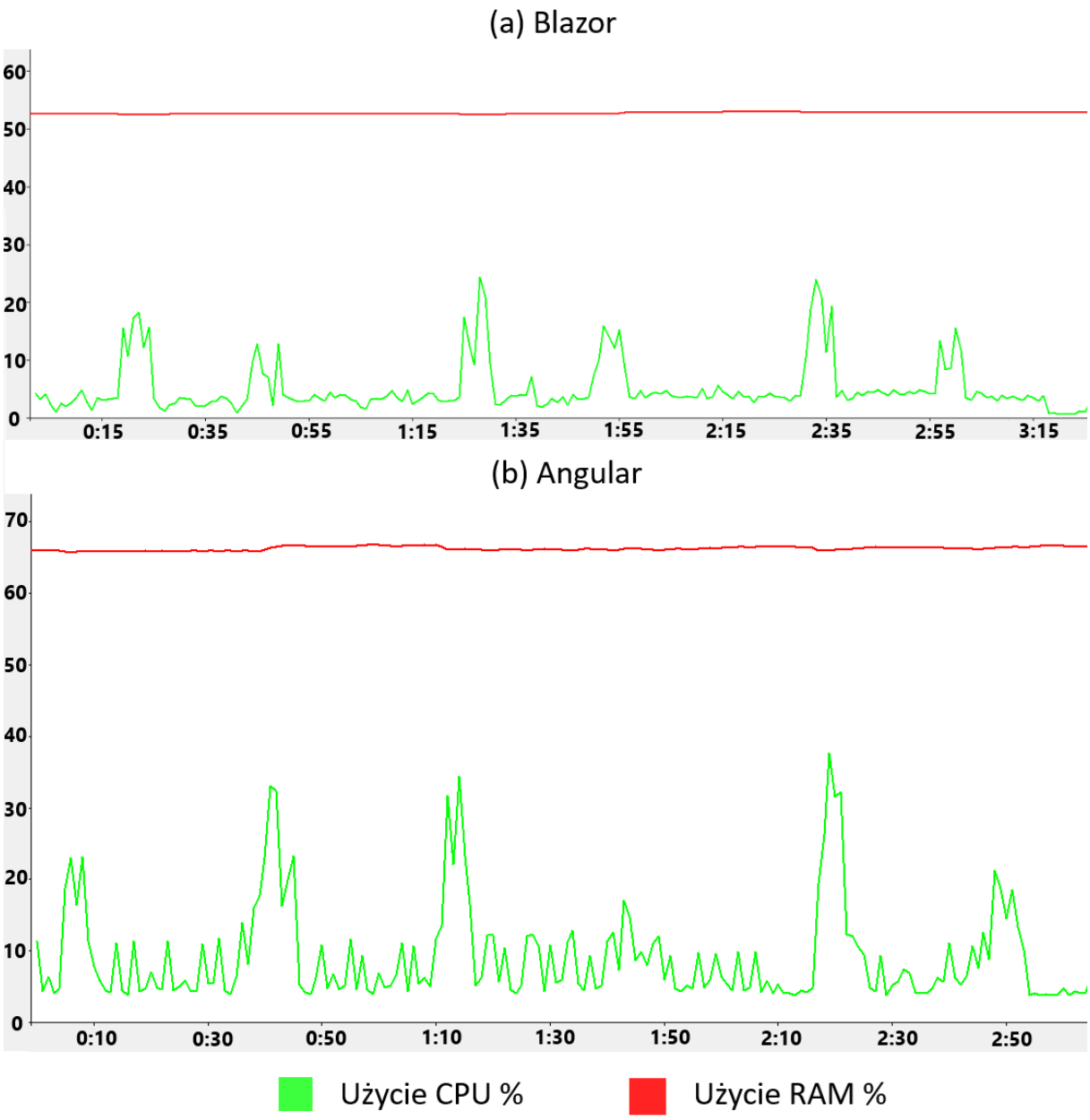
	Komputer testujący - Blazor	Komputer hostujący - Blazor	Komputer testujący - Angular	Komputer hostujący - Angular
Pamięć RAM maks. [%]	17,8	53	18,5	66,8
Pamięć RAM śr. [%]	15,2	52,7	16,9	66,3
Użycie CPU maks. [%]	80,5	24,4	76,1	37,7
Użycie CPU śr. [%]	22	5,1	21,6	9,3

Rysunek 28 przedstawia dwa wykresy obciążenia komputera testującego aplikację Blazor i Angular. Wykresy te są podobne, jednak aplikacja Angular wykazuje większą liczbę skoków w zużyciu CPU.



Rysunek 28. Obciążenie komputera testującego dla scenariusza 5.8. Źródło: opracowanie własne

Na rysunku 29 przedstawiono wykresy obciążenia komputera hostującego aplikacje Angular i Blazor. Liczba skoków jest porównywalna, jednak poziom użycia procesora oraz pamięci podręcznej jest niższy w przypadku aplikacji Blazor.



Rysunek 29. Obciążenie komputera hosta dla scenariusza 5.8 (od góry Blazor, Angular). Źródło: opracowanie własne

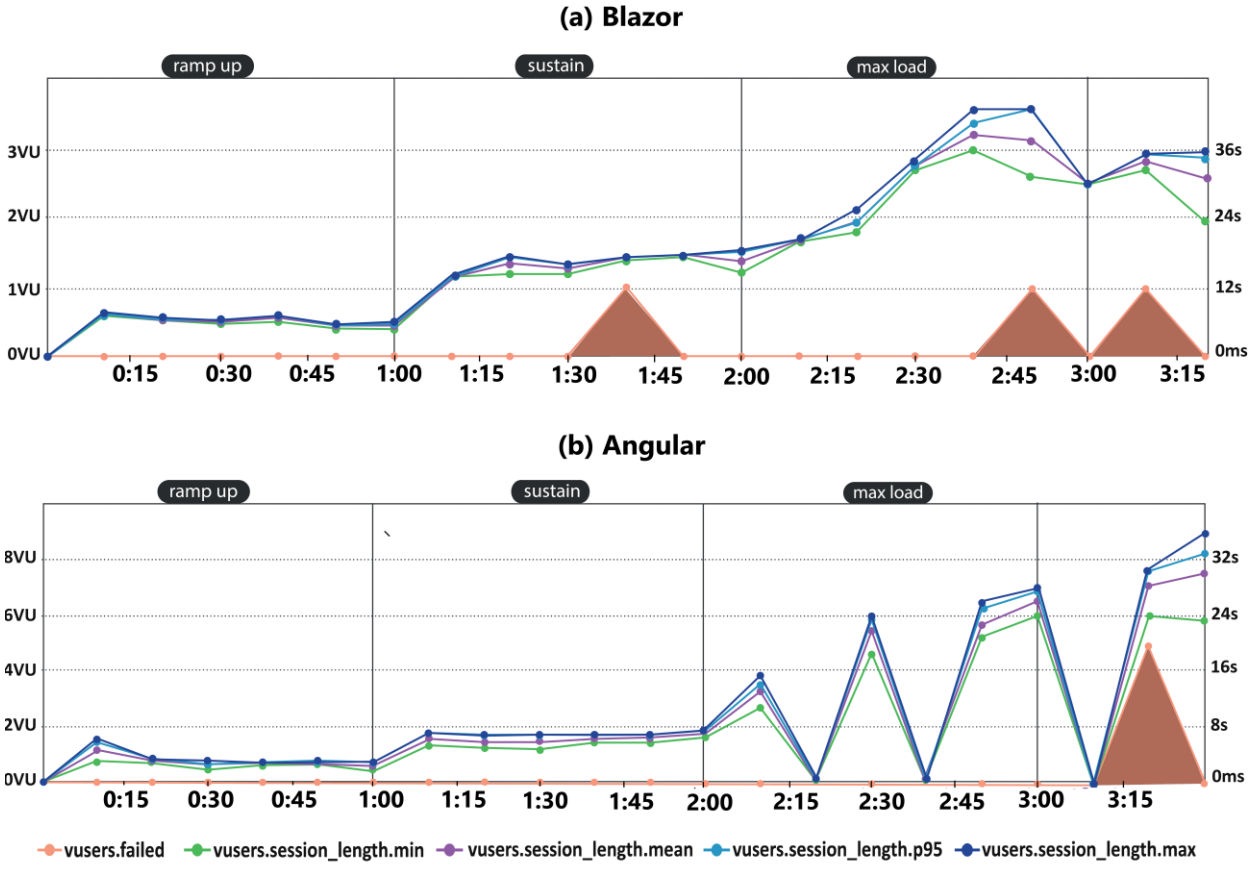
Tabela 16 przedstawia wyniki testu obciążeniowego dla scenariusza wczytywania ekranu zamówienia. Aplikacja Angular wyjątkowo posiada większą liczbę użytkowników, dla których wystąpił błąd. Nie ma to jednak wpływu na długość sesji. Blazor wykazuje ponad dwukrotnie większą minimalną długość sesji.

Tabela 16. Wyniki testu obciążającego dla scenariusza 5.8. Źródło: opracowanie własne

	Blazor	Angular
Minimalna długość sesji [s]	4,7	1,8

Średnia długość sesji [s]	22	11,9
Długość sesji p95 [s]	38,1	28,8
Maksymalna długość sesji [s]	42,9	36
Użytkownicy udani	111	207
Użytkownicy z błędem	3	5

Na rysunku 30 przedstawiono wykresy wyników testu obciążeniowego aplikacji Angular i Blazor dla scenariusza wczytywania ekranu zamówienia. Wykres przebiegu testu dla aplikacji Angular zawiera aż trzy punkty, w których długość sesji wynosi 0. Są to okresy, w których żadna sesja użytkownika nie została zakończona. W aplikacji Blazor pierwszy błąd użytkownika wystąpił już w fazie *sustain*, natomiast w przypadku Angulara miało to miejsce po fazie *max load*.



Rysunek 30. Wynik testu obciążającego dla scenariusza 5.8. Źródło: opracowanie własne

6. Porównanie platform

Platformy do tworzenia aplikacji przeglądarkowych oferują szeroki wachlarz narzędzi, funkcji i rozwiązań. Choć posiadają wiele zalet, żadna z nich nie jest pozbawiona wad. Każda platforma charakteryzuje się unikalnymi mocnymi i słabymi stronami, a ostateczny wybór zależy od ich wzajemnego stosunku, znaczenia przypisywanego poszczególnym aspektom przez programistę oraz, między innymi, od popularności danej technologii. W niniejszym rozdziale omówiono różnice między frameworkami analizowanymi w tej pracy. Szczególną uwagę poświęcono platformom Angular i Blazor, przedstawiając szczegółowe porównanie w zakresie takich aspektów jak proces wytwarzania oprogramowania, wydajność, zużycie pamięci podręcznej oraz obciążenie procesora.

6.1. Główne cechy platform

Frameworki Blazor, Angular, Svelte, Vue.js, .NET MAUI oraz React zostały zestawione w tabelach 17 i 18, które uwzględniają kluczowe cechy pozwalające na ich porównanie. Tabele te dostarczają szczegółowych informacji o różnicach i podobieństwach między nimi, ułatwiając wybór odpowiedniego narzędzia do realizacji konkretnego projektu.

Tabela 17. Cechy frameworków. Źródło: opracowanie własne

Framework	Język programowania	Typ	Łatwość nauki
Blazor	C#	Full-Stack framework	Wysoka
Angular	TypeScript	Frontend framework	Średnia
Svelte	JavaScript	Frontend framework	Wysoka
Vue.js	JavaScript	Progresywny framework	Wysoka
.NET MAUI	C#, XAML	Wieloplatformowy	Średnia
React	JavaScript, JSX	Biblioteka frontend	Średnia

Frameworki różnią się językami programowania, na których są oparte. Blazor i .NET MAUI wykorzystują C#, co czyni je atrakcyjnymi dla programistów z doświadczeniem w ekosystemie .NET. Angular, Svelte, Vue.js oraz React bazują na JavaScript lub jego odmianach, takich jak TypeScript (Angular) czy JSX (React). Wybór języka programowania może być kluczowym czynnikiem przy podejmowaniu decyzji, ponieważ wpływa na dostępność programistów, łatwość nauki oraz integrację z istniejącymi projektami.

Blazor oraz .NET MAUI to frameworki *full-stack*, które mogą być używane zarówno do budowy *frontend,* jak i *backend*, a .NET MAUI dodatkowo obsługuje aplikacje wieloplatformowe. Angular, Svelte, Vue.js oraz React koncentrują się głównie na *frontend,* przy czym React jest bardziej biblioteką niż pełnoprawnym frameworkiem. Typ frameworku decyduje o tym, jakie potrzeby projektowe może spełnić i jakich dodatkowych narzędzi wymaga do realizacji pełnej aplikacji.

Wszystkie analizowane platformy bazują na architekturze komponentowej, co pozwala na modularność i ułatwia zarządzanie kodem. Wyjątkiem jest Vue.js, który jest określany jako

progressive framework, umożliwiający dodawanie funkcji stopniowo, w zależności od potrzeb projektu.

Pod względem łatwości nauki Svelte oraz Vue.js są oceniane wysoko. Ich prostota i przejrzysta dokumentacja sprawiają, że są one prostsze do opanowania, zwłaszcza dla początkujących programistów. Blazor również jest łatwy do nauk dzięki ograniczeniu języków wymaganych do stworzenia aplikacji i intuicyjnych przejściach między językiem C# i HTML w tworzonych stronach. Angular oraz .NET MAUI wymagają bardziej zaawansowanej wiedzy ze względu na złożoną architekturę oraz konieczność znajomości TypeScriptu (Angular) lub XAML (MAUI).

Tabela 18. Pozostałe cechy frameworków. Źródło: opracowanie własne

Framework	Elastyczność	Wsparcie ekosystemu	Przypadki użycia
Blazor	Średnia	Silne	Aplikacje biznesowe
Angular	Wysoka	Silne	Aplikacje korporacyjne
Svelte	Wysoka	Średnie	Lekkie aplikacje
Vue.js	Wysoka	Silne	Uniwersalne aplikacje
.NET MAUI	Średnia	Silne	Mobilne i desktopowe
React	Wysoka	Silne	Dynamiczne aplikacje

Frameworki *frontend*, takie jak Angular, Svelte, Vue.js i React, oferują dużą elastyczność w zakresie konfiguracji i wyboru dodatkowych narzędzi. Blazor oraz .NET MAUI są mniej elastyczne, ponieważ są silnie związane z ekosystemem Microsoftu, co może być zarówno zaletą, jak i ograniczeniem w zależności od wymagań projektu.

Angular, React i Vue.js cieszą się wsparciem dużych społeczności oraz obszernymi ekosystemami narzędzi i bibliotek. Blazor oraz .NET MAUI korzystają ze wsparcia Microsoftu, co zapewnia stabilność, ale może ograniczać różnorodność dostępnych rozwiązań. Svelte, będący stosunkowo nowy, ma mniejsze wsparcie społeczności, co jednak w przyszłości może się zmienić.

Przypadki użycia poszczególnych frameworków są zróżnicowane i zależą od ich charakterystyki oraz wymagań projektu. Blazor najlepiej sprawdza się w aplikacjach biznesowych i korporacyjnych zintegrowanych z ekosystemem Microsoftu, gdzie wykorzystywany jest język C#. Angular, dzięki strukturze opartej na TypeScript, jest często wybierany do dużych aplikacji korporacyjnych i systemów wielostronicowych, wymagających rygorystycznej architektury. Svelte, z uwagi na minimalizm i wysoką wydajność, jest idealny do lekkich i szybkich aplikacji, gdzie priorytetem jest optymalizacja. Vue.js doskonale nadaje się do szerokiego zakresu projektów, od prostych prototypów po bardziej złożone systemy, oferując elastyczność i prostotę integracji. .NET MAUI to rozwiązanie dla aplikacji wieloplatformowych, pozwalające na tworzenie mobilnych i desktopowych aplikacji z wykorzystaniem jednej bazy kodu. React natomiast dominuje w tworzeniu dynamicznych interfejsów użytkownika, sprawdzając się zarówno w małych aplikacjach, jak i dużych systemach wymagających interaktywności i elastyczności.

Wszystkie opisane frameworki, służące do wytwarzania aplikacji są darmowe i otwartoźródłowe. Koszty jakie aplikację generują znajdują się w narzędziach pomocniczych, serwerach, dodatkowych bibliotekach oraz usługach chmurowych. Wszystkie wymienione frameworki są elastyczne i można je uruchamiać na dowolnej platformie obsługującej pliki statyczne. Nie ma wymagań, które powodowałyby brak kompatybilności. Dla przykładu Angular może być uruchamiany na Azure App Services, mimo że *framework* został stworzony przez Google a *hosting* przez Microsoft. Tabela 19 przedstawia informację o kosztach i limitach darmowych planów dla najpopularniejszych platform. Dla stworzonego przykładowego projektu, *hosting* Netlify wydaje się najlepszym rozwiązaniem. Biorąc pod uwagę potencjalny ruch na stronie, limit darmowego planu nie powinien zostać przekroczony. Wybór platformy jest w dużym stopniu uzależniony od wymagań i założeń naszej aplikacji, dlatego ciężko wybrać zwycięzcę tego zestawienia.

Tabela 19. Koszty hostingów. Źródło: opracowanie własne

Platforma	Opis limitów darmowego planu	Cena planów płatnych (od)
Azure App Service	60 minut dziennie, ograniczone zasoby CPU, 1 GB przestrzeni.	~15 USD/miesiąc (Basic Plan)
Firebase Hosting	1 GB przestrzeni, 10 GB transferu na miesiąc.	0,20 USD/GB za transfer poza limitem
Vercel	100 GB transferu, 10 wdrożeń na dzień.	20 USD/miesiąc
Netlify	125 tys. żądań/miesiąc, 100 GB transferu, 1 GB przestrzeni.	19 USD/miesiąc
AWS Amplify	5 GB przestrzeni, 15 GB transferu na miesiąc.	0,15 USD/GB za transfer poza limitem

6.2. Wytwarzanie oprogramowania

Wytwarzanie oprogramowania jest szerokim pojęciem i kluczową cechą przy porównaniu frameworków. Odpowiada on za czas realizacji projektów, efektywności zespołu programistycznego, ilości napotkanych błędów oraz za przyjemność pracy. Badanie to pozwala ocenić, która technologia umożliwia szybsze i łatwiejsze tworzenie aplikacji.

Jedną z kluczowych różnic między Angular a Blazor jest użycie języka programowania. Angular jest oparty na TypeScript. Programowanie w Angular wymaga znajomości nowoczesnych technik front-endowych, takich jak DOM, AJAX, oraz architektury aplikacji opartej na komponentach. Angular zapewnia pełne wsparcie dla jednolitych interfejsów użytkownika i dynamicznego ładowania danych, co pozwala na stworzenie aplikacji w pełni odpowiadającej na zmieniające się potrzeby użytkowników.

Blazor, w przeciwieństwie do Angular, używa C# do tworzenia aplikacji przeglądarkowych. To oznacza, że programiści mogą korzystać z tej samej technologii (C#) zarówno po stronie serwera, jak i po stronie klienta. W Blazor WebAssembly cały kod C# jest kompilowany do formatu WebAssembly i uruchamiany bezpośrednio w przeglądarkach, co pozwala na utrzymanie jednolitej bazy kodu. Programowanie w Blazor jest bardziej zbliżone do tradycyjnego podejścia w aplikacjach desktopowych lub serwerowych, co może być dużą zaletą dla deweloperów z doświadczeniem w .NET i C#.

Istotnym aspektem przy wyborze technologii jest poziom trudności implementacji. Zbyt wysoki poziom może powodować frustrację, wydłużenie czasu tworzenia oprogramowania, a w ostateczności prowadzić do decyzji o zmianie frameworka na inny. Implementacja aplikacji przeglądarkowej w Angularze i Blazorze różni się stopniem skomplikowania głównie ze względu na stosowane technologie i podejście do tworzenia aplikacji. Angular, oparty na TypeScript, wymaga od dewelopera znajomości tego języka oraz zrozumienia bardziej złożonych koncepcji, takich jak modularność, hierarchia komponentów i szczegółowe zarządzanie stanem. Ponadto Angular dostarcza bogaty zestaw narzędzi, takich jak CLI (ang. *Command Line Interface*), które ułatwiają pracę, ale wymagają od użytkownika przyswojenia dodatkowej wiedzy [3]. Z kolei Blazor, bazujący na języku C#, jest bardziej intuicyjny dla programistów zaznajomionych z ekosystemem .NET. Daje możliwość używania tych samych języków i narzędzi w całym stosie aplikacji, co obniża próg wejścia w projekty dla zespołów .NET. Niemniej jednak Blazor wymaga zrozumienia specyfiki działania modeli Server i WebAssembly, co może stanowić wyzwanie dla nowych użytkowników [2].

Wykorzystanie tradycyjnych narzędzi do testowania aplikacji przeglądarkowych w kontekście Blazor WebAssembly może napotkać szereg trudności wynikających z różnic w architekturze oraz technologii używanej w tym frameworku. Aplikacje Blazor WebAssembly są uruchamiane w przeglądarkach za pomocą WebAssembly, co pozwala na wykonywanie kodu C# po stronie klienta, zamiast korzystania z JavaScript, jak w przypadku większości tradycyjnych aplikacji webowych. W związku z tym, narzędzia takie jak Selenium [27], które są przeznaczone do testowania aplikacji JavaScriptowych, nie są w pełni kompatybilne z Blazor. Selenium, które operuje na poziomie DOM, ma trudności w interakcjach z elementami Blazor, które są tworzone dynamicznie przy użyciu C#. Z tego powodu w niniejszej pracy zdecydowano się na użycie Playwright, które jest w pełni kompatybilne z obiema aplikacjami.

Jednym z problemów napotkanych podczas tworzenia przykładowego projektu w framework*u* Blazor było zarządzanie ciasteczkami. W aplikacjach WebAssembly manipulacja *cookies* jest utrudniona, ponieważ platforma nie oferuje natywnego API do bezpośredniego zarządzania ciasteczkami. Aby zapisać lub odczytać dane z *cookies*, należy skorzystać z mechanizmu JavaScript Interop, który umożliwia wywoływanie funkcji JavaScript z poziomu Blazora. Dzięki temu można wykonywać operacje na ciasteczkach, takie jak ich zapis czy odczyt. Alternatywnie dostępna jest zewnętrzna biblioteka Blazor.Cookies [28], która upraszcza tę funkcjonalność, oferując prostszy interfejs do ich manipulacji bez konieczności pisania własnych funkcji JavaScript. Mimo dostępnych rozwiązań brak natywnego wsparcia w Blazorze wymaga dodatkowego wysiłku przy integracji z JavaScript.

6.3. Wydajność

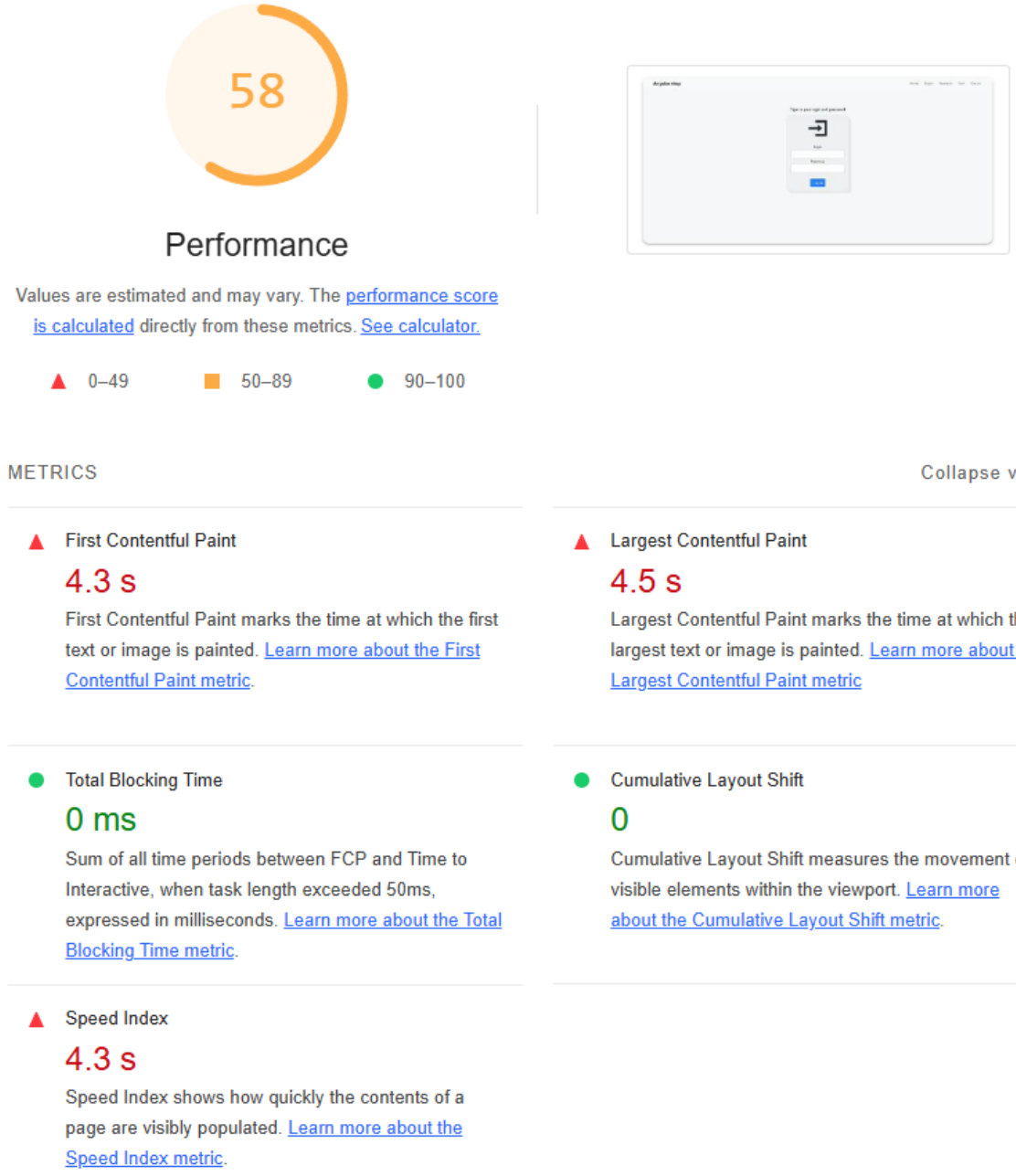
Wydajność aplikacji Angular opiera się na efektywnym zarządzaniu cyklem życia komponentów oraz asynchronicznej obsłudze danych. Dzięki narzędziom takim jak wykrywanie zmian oraz optymalizacji renderowania, Angular jest w stanie skutecznie zarządzać złożonymi aplikacjami. Warto jednak zauważyć, że aplikacje napisane w Angularze mogą być mniej wydajne w przypadku dużych aplikacji z licznymi komponentami, zwłaszcza jeśli chodzi o zarządzanie stanem aplikacji oraz czas ładowania [3].

W przypadku Blazora wydajność aplikacji zależy od wybranego trybu uruchomienia – Blazor WebAssembly wymaga załadowania całego środowiska uruchomieniowego (ang. *runtime*) oraz aplikacji do przeglądarki, co może prowadzić do dłuższego czasu ładowania, szczególnie w przypadku aplikacji z dużą liczbą zależności. Z kolei w Blazor Server, gdzie logika aplikacji jest wykonywana na serwerze, tylko interfejs użytkownika jest tworzony po stronie klienta, co skutkuje mniejszym zużyciem zasobów po jego stronie, ale może powodować opóźnienia wynikające z konieczności przesyłania danych między klientem a serwerem [2]. W niniejszej pracy wykorzystano tryb uruchomienia WebAssembly, dla którego przeprowadzono testy.

Jednym z narzędzi użytych do analizy wydajności został Google Lighthouse [29] to narzędzie otwartoźródłowe służące do analizy wydajności, dostępności, SEO (ang. *Search Engine Optimization*) i ogólnej jakości aplikacji przeglądarkowych. Dostarcza ono szeregu metryk, które pozwalają ocenić efektywność działania aplikacji w różnych warunkach. Kluczowe z nich obejmują:

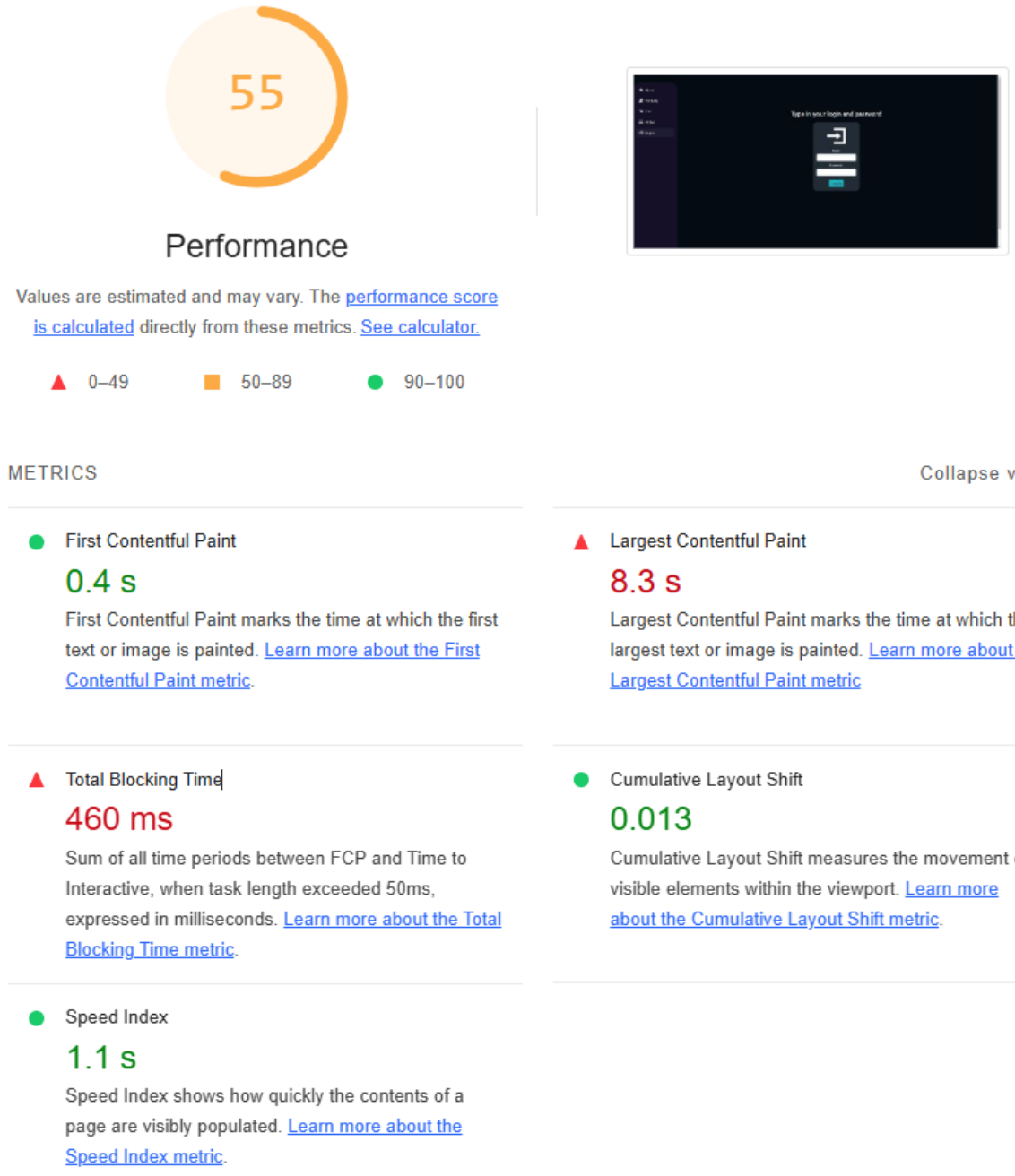
- *First Contentful Paint* - mierzy czas, w jakim użytkownik widzi pierwszą treść na ekranie.
- *Largest Contentful Paint* - wskazuje, jak szybko załadował się największy widoczny element strony.
- *Time to Interactive* - określa czas, po jakim strona staje się w pełni interaktywna, tj. wszystkie zasoby są załadowane, a interfejs reaguje płynnie na działania użytkownika
- *Total Blocking Time* - suma wszystkich okresów między First Contentful Paint a Time to Interactive, w których długość pojedynczego zadania przekraczała 50 ms, wyrażoną w milisekundach.
- *Cumulative Layout Shift* - ocenia stabilność wizualną strony poprzez mierzenie niespodziewanych przesunięć treści.
- *Speed Index* - mierzy, jak szybko treść strony jest widoczna dla użytkownika, uwzględniając stopniowe ładowanie elementów wizualnych

Na rysunku 31 przedstawiono wynik analizy przykładowego projektu stworzonego za pomocą platformy Angular. Wynik wydajności 58 na 100 został obliczony na podstawie metryk opisanych w poprzednim akapicie. Warto zauważyć, że *Cumulative Layout Shift* osiągnęło wartość 0, co jest najlepszym możliwym wynikiem. Oznacza to, że nie doszło do żadnych niespodziewanych przesunięć treści, a zatem możemy przyznać wysoką wizualną stabilność. Drugą metryką, dla której strona osiągnęła wynik 0, jest *Total Blocking Time*, która odzwierciedla, jak bardzo główny wątek przeglądarki jest obciążony w trakcie ładowania strony, co może wpływać na płynność jej działania. Dla aplikacji Angular nie zanotowano okresów, które powodowały blokowanie przeglądarki.



Rysunek 31. Wynik Google Lighthouse dla aplikacji Angular. Źródło: opracowanie własne

Wynik analizy Google Lighthouse dla przykładowego projektu stworzonego za pomocą platformy Blazor został przedstawiony na rysunku 32. Dla metryki *First Contentful Paint* strona osiągnęła wynik 0,4 sekundy. Oznacza to, że pierwszy element został prawie o 4 sekundy szybciej załadowany niż w aplikacji Angular. Słabe wyniki zaprezentowały dwie metryki: *Largest Contentful Paint* oraz *Total Blocking Time*, które wyniosły odpowiednio 8,3 sekundy i 460 ms. *Speed Index* natomiast jest niższy niż dla Angulara o ponad 3 sekundy. Ogólny wynik, jaki osiągnęła aplikacja Blazor w analizie Google Lighthouse, to 55. Wartość ta jest tylko o 3 punkty mniejsza od wyniku Angulara, co wskazuje na małą różnicę wydajnościową, mimo dużej rozbieżności w poszczególnych metrykach.



Rysunek 32. Wynik Google Lighthouse dla aplikacji Blazor. Źródło: opracowanie własne

Na tabeli 20 przedstawiono wyniki testów wydajnościowych aplikacji Angular i Blazor, przeprowadzonych w rozdziale 5. Można zaobserwować, iż framework Angular wygrywa w każdym parametrze o znaczną wartość. Wyniki czasowe aplikacji Blazor są prawie dwukrotnie wyższe niż wyniki dla aplikacji Angular. Różnica w czasie odbiła się zarówno na ilości udanych użytkowników. Dzięki temu, że testy dla aplikacji Angular wykonywały się w przybliżeniu dwa razy szybciej niż te dla aplikacji Blazor, ilość użytkowników, którzy ukończyli test bez błędów, jest prawie dwa razy większa. Ilość użytkowników, którzy napotkali błąd przekroczenia czasu oczekiwania, również wskazuje na większą wydajność aplikacji Angular.

Tabela 20. Wyniki testów wydajnościowych aplikacji Angular i Blazor. Źródło: opracowanie własne

	Blazor	Angular
Śr. minimalna długość sesji [s]	5,1	2,6
Śr. długość sesji [s]	22,4	11,1
Śr. długość sesji p95 [s]	41,6	23,8
Śr. maksymalna długość sesji [s]	45,9	26,4
Śr. ilość użytkowników udanych	105,8	209,9
Śr. ilość użytkowników z błędem	6	2,9

6.4. Zużycie pamięci podręcznej

W teorii różnice w zużyciu pamięci RAM między Angular a Blazor WebAssembly wynikają głównie z architektury obu platform oraz sposobu ich działania w przeglądarce. Angular, oparty na TypeScript, korzysta z dynamicznej interpretacji kodu, co pozwala na efektywne zarządzanie zasobami podczas działania aplikacji. Dzięki temu zużywa on mniej pamięci RAM, szczególnie w przypadku mniejszych aplikacji lub takich, które intensywnie optymalizują wykorzystanie bibliotek [3]. Z kolei Blazor WebAssembly, jako technologia oparta na WebAssembly i środowisku .NET, wymaga załadowania pełnego środowiska uruchomieniowego .NET w przeglądarce, co zwiększa początkowe zużycie pamięci [2].

Na tabeli 21 przedstawiono statystyki związane z zużyciem pamięci podręcznej, zanotowane podczas wykonania testów z rozdziału 5. Wartości te są średnią wyników, jakie aplikacje osiągnęły we wszystkich ośmiu testach. Kluczowy wniosek, jaki można wyciągnąć na podstawie tabeli 21, jest taki, że Blazor WebAssembly zużywa mniej pamięci podręcznej niż Angular, niezależnie od tego, na jaką maszynę patrzymy. Dla komputera testującego średnie zużycie RAM wyrażone w procentach jest zaledwie o 0,4% mniejsze niż dla aplikacji Angular. Większą różnicę można zaobserwować dla komputera hostującego, gdzie różnica ta wyniosła aż 10,8%. Jedynym parametrem, dla którego Blazor osiągnął słabszy wynik od Angulara (o 0,8%), była średnia z maksymalnego zanotowanego zużycia pamięci dla komputera testującego. Oznacza to większe wahania w zużyciu pamięci z racji niższej średniej.

Choć Blazor powoduje dodatkowe wstępne obciążenie pamięci wynikające z infrastruktury .NET, to na podstawie przeprowadzonych testów można zaobserwować, że nie czyni go to bardziej wymagającym pod tym względem w porównaniu do Angulara. Dla użytkownika końcowego wykazuje on większe wahania w zużyciu RAM, lecz nie są one na tyle duże, ani nie jest to na tyle istotny czynnik, aby odmówić przewagi Blazorowi pod względem zużycia pamięci podręcznej.

Tabela 21. Zużycie pamięci RAM dla wszystkich testów obciążających. Źródło: opracowanie własne

	Śr. pamięć RAM maks. [%]	Pamięć RAM śr. [%]
Komputer testujący - Blazor	18	15,1
Komputer testujący - Angular	17,2	15,5
Komputer hostujący - Blazor	54,3	54
Komputer hostujący - Angular	65,7	64,8

6.5. Zużycie procesora

Różnice w zużyciu procesora między Angular a Blazor WebAssembly wynikają z architektury obu technologii oraz sposobu wykonywania kodu zarówno po stronie klienta, jak i serwera. W przypadku Angulara aplikacja działa po stronie klienta, a procesor urządzenia użytkownika jest odpowiedzialny za interpretację i wykonanie kodu TypeScript. W zależności od złożoności aplikacji oraz jej optymalizacji, obciążenie procesora może wzrosnąć, szczególnie podczas pierwszego ładowania lub w trakcie intensywnych operacji, takich jak przetwarzanie danych czy animacje. Maszyna hostująca w Angularze obsługuje jedynie statyczne pliki (HTML, CSS, JavaScript), co minimalizuje jej obciążenie procesora. Blazor WebAssembly natomiast przenosi część obciążenia na maszynę klienta, ponieważ musi załadować środowisko .NET i wykonywać kod w środowisku WebAssembly w przeglądarce. Ten proces, szczególnie podczas początkowego ładowania aplikacji, może znacząco obciążyć procesor urządzenia użytkownika. W porównaniu do Angulara, obciążenie procesora po stronie klienta w Blazor WebAssembly może być większe, zwłaszcza na urządzeniach o słabszych parametrach. Z kolei maszyna hostująca w przypadku Blazor WebAssembly również ma ograniczone obciążenie procesora, ponieważ serwuje głównie statyczne pliki.

Tabela 22 przedstawia statystyki obciążenia procesora podczas wykonania testów z rozdziału 5. Obejmuje informacje o średnim użyciu oraz średnią z maksymalnego odnotowanego użycia CPU dla wszystkich ośmiu scenariuszy. Podobnie jak w przypadku pamięci podręcznej, aplikacja stworzona za pomocą platformy Blazor WebAssembly osiąga lepsze wyniki zarówno dla komputera testującego, jak i hostującego. Największą różnicę między stronami zaobserwowano na komputerze hostującym dla średniej z maksymalnego użycia CPU (14,8%). Parametr średniego użycia procesora dla komputera testującego, który jest o wiele istotniejszy z punktu widzenia użytkownika, nie wykazał aż tak dużych różnic (3,5%). Warto odnotować rozbieżność między średnim użyciem procesora dla komputera hostującego. Co prawda różnica wynosi zaledwie 3,3%, lecz biorąc pod uwagę fakt, że dla obu wyników jest to zaledwie kilka procent, aplikacja Angular wymaga około 64% więcej zasobów CPU od Blazora.

Blazor WebAssembly, choć wymaga załadowania środowiska .NET w przeglądarce klienta, okazał się w testach mniej obciążający dla CPU, co może być związane z bardziej zoptymalizowanym wykonywaniem kodu WebAssembly, mniejszym narzutem na procesor podczas działania aplikacji lub mniejszą liczbą stworzonych użytkowników na potrzeby testów. Maszyna hostująca Blazor, podobnie jak w przypadku Angulara, obsługuje głównie statyczne pliki, dzięki czemu jej zużycie CPU pozostaje niewielkie.

Tabela 22. Obciążenie procesora dla wszystkich testów obciążających. Źródło: opracowanie własne

	Śr. użycie CPU maks. [%]	Użycie CPU śr. [%]
Komputer testujący - Blazor	85,5	22
Komputer testujący - Angular	86,7	25,5
Komputer hostujący - Blazor	26,6	5,2
Komputer hostujący - Angular	41,4	8,5

7. Podsumowanie

Celem niniejszej pracy było wsparcie programisty w podjęciu decyzji o wyborze platformy do tworzenia aplikacji przeglądarkowych. Przedstawiono najpopularniejsze dostępne rozwiązania, opisując ich najważniejsze cechy, silne i słabe strony. Następnie stworzono przykładowy projekt za pomocą dwóch platform Angular i Blazor, w celu ich głębszego przetestowania. Opisano różnice w wydajności, trudności przy implementacji oraz ich wpływu na obciążenie komponentów serwera i klienta.

W ramach badań przeprowadzono testy obciążeniowe aplikacji Angular i Blazor, w których Angular osiągnął znacznie lepsze wyniki wydajnościowe. Może on być bardziej odpowiedni w przypadku aplikacji wymagających wysokiej wydajności i dynamicznej interakcji z użytkownikiem, kosztem większego obciążenia zasobów systemowych. Z kolei Blazor WebAssembly sprawdzi się w projektach, gdzie programiści są ograniczeni brakiem znajomości JavaScriptu i gdzie kluczowa jest integracja z technologiami Microsoftu. Jego plusem jest niskie obciążenie maszyny serwerowej, co pozwala na ograniczenie kosztów utrzymania aplikacji.

Wszystkie opisane w niniejszej pracy frameworki są dobrymi narzędziami do tworzenia aplikacji przeglądarkowych. Dostępne są do nich dokumentacje, liczne poradniki stworzone przez fanów oraz wiele opracowanych rozwiązań błędów, które można napotkać podczas programowania. Jednym z najważniejszych czynników wyboru frameworku jest język programowania, na którym jest on oparty. Różnice, jakie istnieją między nimi, nie są na tyle duże, aby programista musiał uczyć się nowego języka tylko na potrzeby stworzenia aplikacji przeglądarkowej. Zakładając jednak, że nie jesteśmy ograniczeni brakiem znajomości któregoś z języków, wybór ten staje się trudniejszy.

Dla prostej, mało uczęszczanej aplikacji przeglądarkowej najlepszym wyborem mogą być Svelte lub Vue.js. Oba frameworki oferują prostą konfigurację, mały narzut na wydajność i łatwość tworzenia niewielkich aplikacji. Svelte wyróżnia się brakiem *runtime*, co skutkuje mniejszym rozmiarem końcowego pliku, natomiast Vue.js zapewnia świetną dokumentację i wspiera szybki rozwój z wykorzystaniem gotowych bibliotek. Platformy takie jak React czy Angular mogą być nadmiarowe dla takich zastosowań, podczas gdy Blazor i .NET MAUI lepiej sprawdzą się w scenariuszach integracji z ekosystemem Microsoftu.

Wnioski z pracy mogą dostarczyć praktycznych wskazówek dotyczących wyboru frameworka w zależności od potrzeb projektu, pozwalając na świadome podjęcie decyzji technologicznych. Przeprowadzone analizy oraz wyniki testów wydajnościowych mogą stanowić cenne źródło wiedzy dla osób planujących budowę nowoczesnych aplikacji przeglądarkowych.

Bibliografia

[1] Barry Schwartz. The Paradox of Choice. Why More is Less, Harper Perennial 2004. ISBN: 0-06-000568-8

[2] Blazor, https://learn.microsoft.com/en-us/aspnet/core/blazor/, dostęp 2024-12-03

[3] Angular, https://angular.dev/, dostęp 2025-01-05

[4] ASP.NET Core Web API , https://dotnet.microsoft.com/en-us/apps/aspnet/apis, dostęp 2025-01-05

[5] Microsoft SQL Server, https://learn.microsoft.com/en-us/sql/sql-server, dostęp 2025-01-05

[6] Playwright, https://playwright.dev/java/docs/intro, dostęp 2025-01-05

[7] Artillery, https://www.artillery.io/, dostęp 2025-01-05

[8] React, https://react.dev/, dostęp 2025-01-05

[9] React GitHub, https://github.com/facebook/react/releases?page=11/, dostęp 2025-01-05

[10] Vue.js, https://vuejs.org/, dostęp 2025-01-05

[11] Vivian Cromwell, “Evan You”, Between the Wires 2016-11-03, https://web.archive.org/web/20170603052649/https://betweenthewires.org/2016/11/03/evan-you/

[12] .NET MAUI, https://learn.microsoft.com/en-us/dotnet/maui, dostęp 2025-01-05

[13] What is .NET MAUI?, https://learn.microsoft.com/en-us/dotnet/maui/what-is-maui?view=net-maui-9.0, dostęp 2025-01-05

[14] GitHub .NET MAUI, https://github.com/dotnet/maui, dostęp 2025-01-05

[15] Svelte, https://svelte.dev, dostęp 2025-01-05

[16] Vue vs Svelte: A Comprehensive Comparison of Features, https://tildeloop.com/blog/vue-vs-svelte-a-comprehensive-comparison-of-features/, dostęp 2025-01-05

[17] Blazor Usage Statistics, https://trends.builtwith.com/framework/Blazor, dostęp 2025-01-05

[18] Daniel Roth, Jeff Fritz, Taylor Southwick, Scott Addie, Steve Smith. Blazor for ASP.NET Web Forms Developers, Microsoft 2020

[19] AngularJS, https://angularjs.org/, dostęp 2025-01-05

[20] Roy Thomas Fielding, Architectural styles and the design of network-based software architectures. University of California, Irvine 2000. ISBN: 978-0-599-87118-2

[21] Playwight GitHub, https://github.com/microsoft/playwright, dostęp 2025-01-05

[22] YAML, https://yaml.org/, dostęp 2025-01-05

[23] David Garlan and Mary Shaw, An Introduction to Software Architecture, Carnegie Mellon University, Pittsburgh 1994

[24] Three-tier architecture, https://www.ibm.com/topics/three-tier-architecture, dostęp 2025-01-05

[25] Most used frameworks, https://www.statista.com/statistics/1124699/worldwide-developer-survey-most-used-frameworks-web/, dostęp 2025-01-05

[26] Windows Performance Monitor, https://techcommunity.microsoft.com/blog/askperf/windows-performance-monitor-overview/375481, dostęp 2025-01-05dostep 2025-01-05

[27] Selenium, https://www.selenium.dev/, dostęp 2025-01-05

[28] Blazor.Cookies, https://github.com/BitzArt/Blazor.Cookies, dostęp 2025-01-05

[29] Google Lighthouse, https://developer.chrome.com/docs/lighthouse/overview?hl=pl, dostęp 2025-01-05

[30] Usage statistics of JavaScript as client-side programming language on websites, https://w3techs.com/technologies/details/cp-javascript, dostęp 2025-01-05

[31] World Wide Web Consortium (W3C) brings a new language to the Web as WebAssembly becomes a W3C Recommendation, https://www.w3.org/press-releases/2019/wasm/, dostęp 2025-01-05

[32] SPA vs MPA — na czym polega różnica w działaniu tych aplikacji?, https://mindboxgroup.com/pl/spa-vs-mpa-na-czym-polega-roznica-w-dzialaniu-tych-aplikacji/, dostęp 2025-01-05

[33] Czym jest Cloud Computing (Chmura Obliczeniowa)?, https://webixa.pl/blog/czym-jest-cloud-computing-chmura-obliczeniowa/, dostęp 2025-01-05

Spis tabel

Spis rysunków

Spis listingów