

**CELEM PRACY JEST PRZEDSTAWIENIE TECHNOLOGII
NODE.JS NA PRZYKŁADZIE ZAPROJEKTOWANEJ
APLIKACJI PRZEZNACZONEJ DO ZARZĄDZANIA
CODZIENNÝMI ZADANIAMI.**

Spis treści

1 Przedstawienie ogólne, ideologii oraz przeznaczenia technologii Node.js	4
1.1 Wstęp	4
1.2 Przeznaczenie	5
1.3 Modułowość	5
2 Omówienie architektury Node.js	7
2.1 Paradygmat	7
2.2 Asynchroniczność	8
2.3 Architektura komunikacji	10
2.4 Alternatywne rozwiązania	13
2.4.1 Vert.x	13
2.4.2 Twisted	13
2.4.3 Ringojs	14
3 Założenia i specyfikacja aplikacji	15
3.1 Specyfikacja problemu	15
3.2 Wymagania funkcjonalne	16
3.3 Wymagania pozafunkcjonalne	17
4 Opracowanie aplikacji	18
4.1 Mean Stack	18
4.1.1 Wstęp	18
4.1.2 MongoDB	18
4.1.3 ExpressJS	18
4.1.4 AngularJS	19
4.2 Komunikacja	19
4.2.1 Rozwiążanie problemu komunikacji	19
4.2.2 Schematy wymiany zasobów	20
4.2.3 Kod po stronie klienta wysyłający żądanie	22

4.2.4	Kod po stronie serwera obsługujący odebranie żądania	24
4.3	Wykorzystane moduły	26
4.3.1	Moduły zewnętrzne	26
4.3.2	Moduły wewnętrzne	27
5	Testy aplikacji	28
5.1	Testy manualne	28
5.1.1	Uzyskanie dostępu do statycznych zasobów serwisu	28
5.1.2	Rejestracja w serwisie	29
5.1.3	Potwierdzenie adresu email	29
5.1.4	Logowanie się w serwisie	30
5.1.5	Resetowanie hasła użytkownika	31
5.1.6	Zmiana danych użytkownika	32
5.1.7	Utworzenie nowej tablicy z zadaniami	33
5.1.8	Wysyłanie zaproszenia do tablicy	33
5.1.9	Akceptacja zaproszenia	34
5.1.10	Odrzucenie zaproszenia	34
5.1.11	Wyrzucenie użytkownika z tablicy	35
5.1.12	Dodanie zadania	35
5.1.13	Dodanie statusu zadania	36
5.1.14	Usuwanie zadania	36
5.1.15	Opuszczanie tablicy	37
5.1.16	Usuwanie tablicy	37
5.2	Testy wydajnościowe	38
5.2.1	Mała ilość danych	38
5.2.2	Średnia ilość danych	38
5.2.3	Duża ilość danych	38
5.2.4	Podsumowanie testu	39
5.3	Testy obciążeniowe serwera	39
5.3.1	Mała ilość danych	39
5.3.2	Średnia ilość danych	39
5.3.3	Duża ilość danych	39
5.3.4	Podsumowanie testu	39
5.4	Testy sprawnościowe serwera	40
5.4.1	Krótki okres czasu	40
5.4.2	Średni okres czasu	40
5.4.3	Duży okres czasu	40
5.4.4	Podsumowanie testu	40

6 Podsumowanie otrzymanych wyników i wnioski na temat środowiska	41
Bibliografia	42
Spis rysunków	43

Rozdział 1

Przedstawienie ogólne, ideologii oraz przeznaczenia technologii Node.js

1.1 Wstęp

Node.js jest cross-platformowym, działającym niezależnie od środowiska językiem programowania, napisanym w językach c/c++ oraz javascript, wydanym 27 marca 2009 roku, zaprojektowanym przez Ryana Dahlę. Pozwala na tworzenie serwerów i narzędzi sieciowych, działających po stronie serwera. Przed powstaniem języka kod javascriptowy był wykonywany głównie przez przeglądarkę internetową po stronie klienta, co pozwalało na bezproblemową manipulację kodem źródłowym strony przez użytkownika, dając możliwość wykonywania złośliwych skryptów, naruszenie bezpieczeństwa baz danych lub uzyskania dostępu do chronionych zasobów servera. Środowisko Node.js może działać niezależnie od środowiska uruchomieniowego. Jest ono zgodne z wieloma systemami operacyjnymi jak Linux, macOS, Microsoft Windows, NonStop, czy serwerami Unix. Język ten cieszy się dużą popularnością oraz pozytywnym odbiorem wśród użytkowników, dzięki czemu, mimo względnie krótkiego okresu życia środowiska, zaowocowało ogromną ilością projektów open-source, tysiącami członków należących do społeczności okołojęzykowej oraz powstaniem wydarzeń poruszających tematy okołosrodowiskowe, takimi jak NodeConf, Node Interactive lub Node Summit. Obecnie wiele największych firm korzysta z serwerów napisanych w języku Node.js. Ich przykładami są między innymi Groupon, IBM, LinkedIn, Microsoft, Netflix, PayPal, Yahoo. Najpopularniejszymi API wspierającymi edycję oraz debugowanie kodu Node.js są Atom, Brackets, JetBrains, Microsoft Visual Studio, NetBeans czy Nodeclipse.

1.2 Przeznaczenie

Node.js zalecany jest do tworzenia aplikacji:

- z dużą liczbą operacji wejścia/wyjścia,
- strumieniowania danych np. video,
- Single Page Applications (SPA),
- udostępniających API w formacie JSON,
- z intensywną wymianą danych w czasie rzeczywistym na wielu urządzeniach, np. portalach społecznościowych.

Ponieważ jest on szybki i lekki, może być stosowany do pisania między innymi bramki API. API to skrót od Application Programming Interface; opisuje, jak poszczególne elementy lub warstwy oprogramowania powinny się komunikować. W praktyce to najczęściej biblioteka oferująca metody, które umożliwiają realizację określonych zadań. Node.js pozwala na zoptymalizowanie pracy oraz uzyskanie skalowalności dzięki asynchronicznemu przetwarzaniu danych dostarczanych do aplikacji, w związku z czym idealnie nadaje się do obsługi komunikacji wymagającej pracy w czasie rzeczywistym. Funkcje napisane w Node.js wykonują się równolegle, korzystając z tak zwanych wywołań zwrotnych (ang. callback), przeciwnie do języków takich jak php, gdzie program jest wykonywany synchronicznie - linia po linii. Dzięki temu nie powstaje problem blokowania określonych funkcjonalności programu w czasie pracy innych niezależnych jego części. Przy pomocy wywołań zwrotnych możemy zapewnić zasygnalizowanie uzyskanych wyników lub zwrócenie, bądź obsługę błędu powstałego w czasie działania bloku kodu.

1.3 Modułowość

Praca z Node.js opiera się głównie o korzystanie ze zbioru zdefiniowanych w ramach modułów funkcji wspierających określone funkcjonalności. Zapewniają one pracę między innymi z plikami systemowymi, z urządzeniami wejścia/wyjścia, protokołami internetowymi (dns, http, tcp, tls/ssl, udp), plikami binarnymi, źródłami danych oraz funkcjami kryptograficznymi. Zmniejszają one złożoność, a co za tym idzie, nakład pracy przy tworzeniu własnej funkcjonalności. Dzięki wsparciu package managera (od roku 2010) nazywanego npm, programiści mogą bez przeszkód

udostępniać napisane przez siebie moduły i biblioteki lub w prosty sposób zaimportowywać ogólnie dostępne moduły i używać ich w swoich projektach. Najpopularniejszymi modułami wykorzystywanyymi w celu poprawy jakości oraz zmniejszenia nakładów pracy przy wytwarzaniu oprogramowania są Express.js, Socket.IO, Hapi.js, Sails.js czy Meteor. Npm jest automatycznie włączony w środowisko Node.js. Jest obsługiwany za pomocą linii komend systemu operacyjnego. Moduły są zapisane w formacie CommonJS oraz zawierają pliki informacyjne w formacie Json. Ilość ogólnodostępnych modułów przekracza obecnie 477000. Jest to spowodowane możliwością przez każdego użytkownika Node.js, bez potrzeby wcześniejszej rejestracji czy przejścia jakiejkolwiek procedury wstępnej, udostępnienia napisanego przez siebie kodu. W związku z tym, wiele dostępnych modułów jest niskiej jakości, może zawierać elementy złośliwego oprogramowania lub nie być bezpiecznym dla naszego systemu operacyjnego. Należy bezwzględnie brać te czynniki pod uwagę w przypadku korzystania z nieznanych modułów i najlepiej najbardziej znacząco ograniczyć korzystanie z nich bez wcześniejszej weryfikacji kodu źródłowego. Zabezpieczeniami w celu ochrony użytkowników, które dostarcza npm, jest usuwanie pakietów, które zostały zgłoszone przez użytkowników jako naruszające ogólne zasady bezpieczeństwa oraz możliwość wglądu w raporty statystyczne odnośnie ilości pobrań lub ilości zależnych od modułu innych pakietów. Kolejnym zagrożeniem, jakie niesie korzystanie z pakietów udostępnionych przez innych użytkowników, jest możliwość usunięcia udostępnionego pakietu z repozytorium npm, w konsekwencji uniemożliwiając naszej aplikacji dalsze korzystanie z pakietu. Sytuacja taka miała miejsce, kiedy skrypt zwany „left-side”, z którego korzystało ponad 2486696 deweloperów, został usunięty z repozytorium, powodując tak zwany efekt domina, będący przyczyną błędów w kolejnych aplikacjach deweloperów. Npm korzysta, tak jak i inne globalnie działające narzędzia JavaScriptowe, z plików zależności w formacie json. Opisuję one wersję wykorzystywanych modułów i pozwalają za pomocą jednoliniowej komendy na szybką i łatwą instalację wszystkich używanych pakietów w lokalnym środowisku.

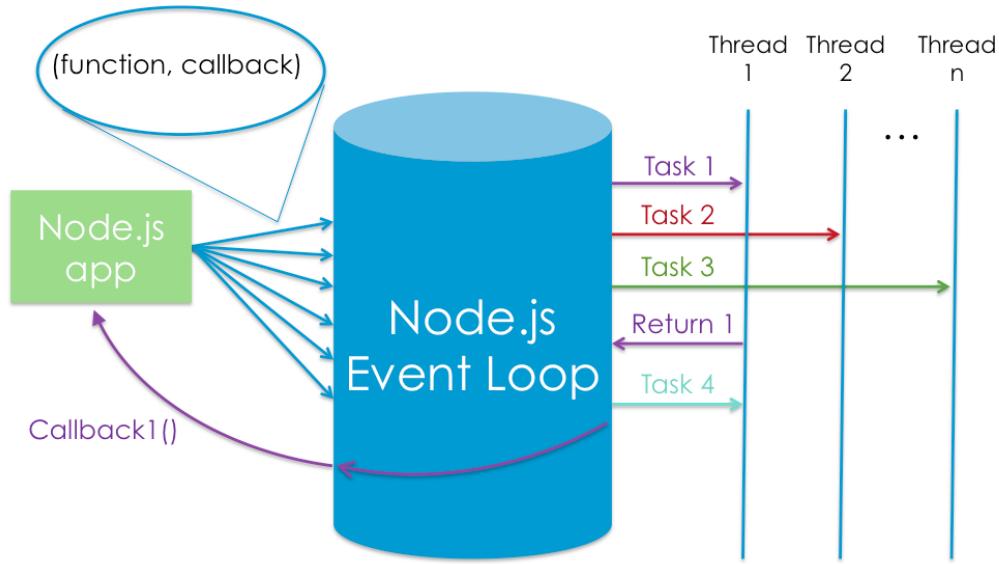
Rozdział 2

Omówienie architektury Node.js

2.1 Paradygmat

Architektura Node.js pozwala na tworzenie oprogramowania sterowanego zdarzeniami (event-driven programming). Jest to paradygmat programowania, w którym kolejność wykonywania kodu zależy od zdarzeń mających miejsce w czasie życia aplikacji (run time), na przykład interakcji użytkownika, czy otrzymania określonych sygnałów. W przypadku języka Node.js, kiedy aplikacja pełni rolę serwera, paradygmat ten najczęściej dotyczy przetwarzania zapytań otrzymywanych ze strony klienta oraz uruchomionych przez nie zdarzeń, tzn. funkcji. W aplikacji sterowanej zdarzeniami należy wyróżnić pętlę główną, która jest odpowiedzialna za obsługę zachodzących w czasie rzeczywistym zdarzeń, czyli wywoływanie triggerów jako wywołań zwrotnych. Są to wyodrębnione części kodu, które po wykonaniu swojego zadania zwracają określoną wartość lub obiekt. W Node.js jest to na przykład funkcja nasłuchująca określonego adresu, pod który klienci kierują określone zapytania. Pozwala to na wykonywanie wielu zadań jednocześnie i niezależnie. Zapewnia przyspieszenie wykonywania skomplikowanych funkcji programu. Daje możliwość przykładowo jednoczesnego zapisywania danych do bazy, przetwarzania innej części zapytania i przygotowywanie odpowiedzi w jednym okresie czasu. Node.js zapewnia w ten sposób działanie asynchroniczne, bez bezpośredniego użycia technologii wielowątkowej. Wychodzi naprzeciw problemowi tworzenia oraz kontrolowania aplikacji współbieżących spełniających zadanie serwera, które są trudne do zaimplementowania w wielu językach programowania oraz często prowadziły do niesatysfakcjonującej wydajności. Język został stworzony na silniku V8 javaScript napisanym w języku C++, wyprodukowanym przez Google, wykorzystywanym w przeglądarkach Google Chrome, który porzuca tradycyjną ideę interpretowania kodu javaScriptowego linia po linii, zapewniając w zamian kompilacje do odpowiednio zoptymalizowanego

kodu maszynowego przed jego wykonaniem, a co za tym idzie, większą wydajność podczas działania programu. Zapewnia on również odpowiednie zarządzanie pamięcią dla obiektów, ściągając z programistów odpowiedzialność alokowania oraz zwalniania zajętych zasobów.

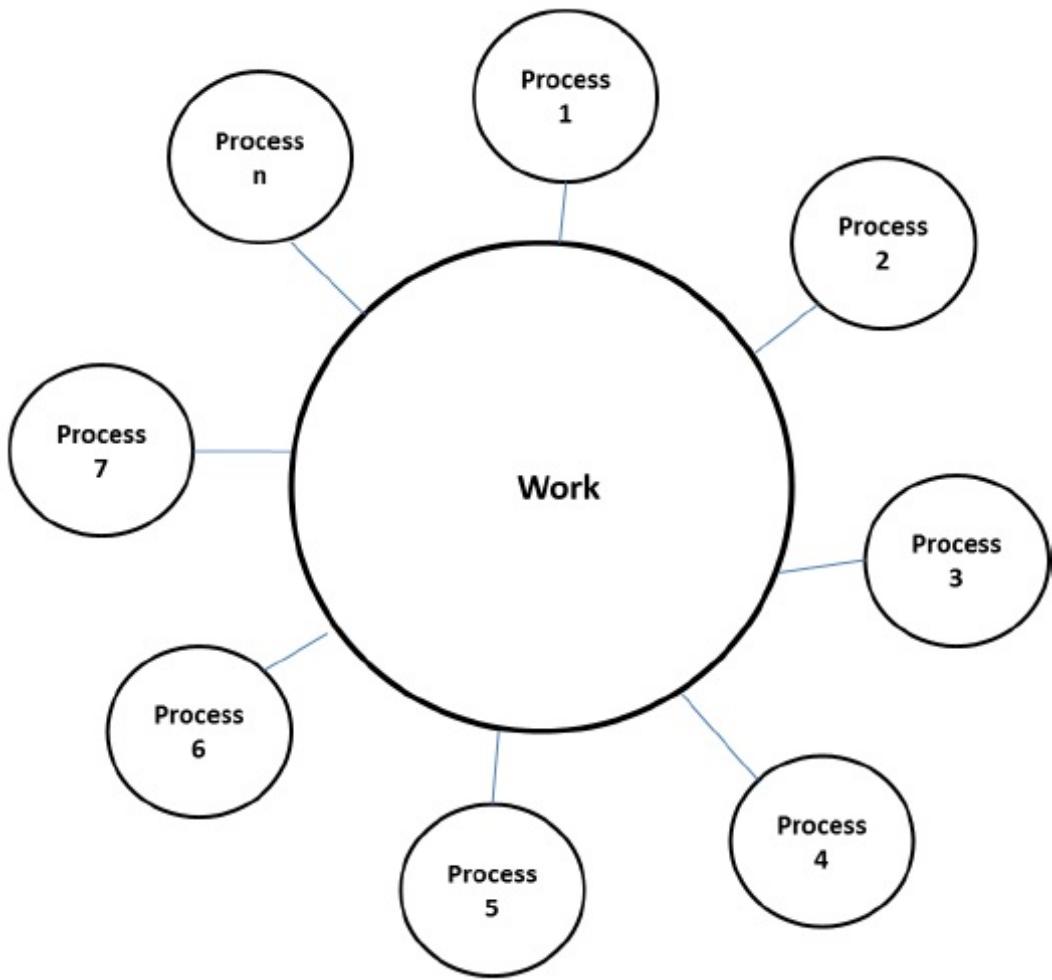


Rysunek 2.1: Pętla zdarzeń w środowisku Node.js - źródło: <https://stackoverflow.com/questions/21607692/understanding-the-event-loop>

2.2 Asynchroniczność

Node.js pracuje wykorzystując tylko jeden wątek, używając technologii asynchronicznego wejścia/wyjścia. Zapewnia to synchronizację wykonywania wielu operacji bez konieczności czekania na zakończenie operacji i zwolnienie dostępu do zasobu poprzez zarządzanie żądaniami wejścia/wyjścia w oderwaniu od wątków wykonywania. Standardowe operacje synchroniczne powodują zablokowanie dalszego wykonywania wątku do czasu zakończenia przetwarzania. W rezultacie określony wątek może zainicjować tylko jedno zadanie wejścia/wyjścia w jednym momencie. Wywołanie asynchronicznej funkcji nie czeka na wyniki, dzięki czemu nie blokujemy wątku wykonawczego. Po zakończonym wywołaniu uruchomiona zostaje funkcja zwrotna lub ogłoszone zostaje zdarzenie w poszczególnych częściach wykonawczych. Pomimo, że Node.js działa na jednym wątku z pętlą zdarzeń, dzięki asynchroniczności potrafi on obsłużyć więcej zapytań niż np. serwer HTTP Apache.

Uzyskanie wielowątkowości, pomimo korzystania tylko z jednego wątku, jest zapewnione dzięki użyciu wzorca projektowego obserwator, w którym jeden obiekt nazywany przedmiotem obserwowanym określa zależności względem obserwujących go innych obiektów, poprzez wywoływanie ich metod dla określonych własnych stanów. W celu obsługi wszystkich asynchronicznych funkcji oraz wątku głównego, Node.js korzysta z biblioteki multiplatformowej języka c - libuv. Biblioteka ta została stworzona specjalnie na potrzebny Node.js, ale jest również wykorzystywana w wielu innych technologiach, np. racer: (obsługa serwerów w języku ruby), czy Trevi: (silnik do obsługi aplikacji internetowych w języku swift). Wykorzystuje ona określony zbiór wątków do równoległej obsługi wielu operacji wejścia/wyjścia bez wzajemnego ich blokowania. Wadą pracy Node.js przy wykorzystaniu tylko jednego wątku jest brak skalowalności pod względem możliwości zwiększenia ilości rdzeni procesorów, na których wykonuje się program, bez użycia dodatkowych modułów, takich jak Cluster (zapewnia możliwość łatwego tworzenia pochodnych procesów, które współdzielą porty serwera), StrongLoop PM (zapewnia zarządzanie procesem produkcyjnym aplikacji Node.js ze wsparciem odpowiedniego zarządzania zasobami, wdrożeniami wielohostowymi oraz interfejsem graficznym), czy PM2 (zarządzanie procesem produkcyjnym, ze wsparciem odpowiedniego zarządzania zasobów, z możliwością nieprzerwanego działania aplikacji przy użyciu przenoszenia jej między domenami, bez potrzeby zatrzymywania pracy). Kolejną możliwością na uniknięcie tego ograniczenia jest zmiana ilości wątków należących do zbioru wykorzystywanego przez bibliotekę libuv. Wątki te działają na wielu rdzeniach systemu, na którym działa serwer. Działanie Node.js jednocześnie na wielu procesach jest zapewnione właśnie dzięki zbiorowi wątków dostarczanych przez bibliotekę libuv. Wątek główny przydziela zadania dla wątków kolejno ze współdzielonej kolejki funkcji, które wątki pochodne mają za zadanie wykonać. Kiedy wątek pochodny zakończy wykonywanie przydzielonego mu zadania, informuje o tym wątek główny poprzez wywołanie określonego wywołania zwrotnego. Z przyczyny odpowiedzialności przez wątek główny do odebrania wszystkich wywołań zwrotnych, funkcja wykonywana przez jeden wątek pochodny może wstrzymać działanie całej asynchronicznej operacji, a co za tym idzie, zmniejszyć jej całkowitą wydajność. Biblioteka libuv zajmuje się odpowiednim podziałem zadań oraz przydzieleniem zasobów tak, aby w jak najlepszy sposób wyważyć nakład pracy między wieloma wątkami.



Rysunek 2.2: Model programowania współbieżnego wykorzystywany przez Node.js - źródło: www.tutorialspoint.com/parallel_algorithm/

2.3 Architektura komunikacji

W tradycyjnym podejściu odpowiedzialne za obsługę przychodzących połączeń są określone wątki lub procesy systemu operacyjnego, które w porównaniu z technologią Node.js wymagają względnie więcej zaalokowanych zasobów. W celu obsługi zapytania przychodzącego do aplikacji Node.js, program rejestruje się w systemie operacyjnym, aby od każdego przychodzącego połączenia otrzymać odpowiednie wywołanie zwrotne, przez co nie potrzebuje procesów czy wątków, aby uzyskać

możliwość obsługi klientów, a tylko własnej pętli głównej, do której wykonywania powraca po przetworzeniu każdego wywołania wstecznego. W czasie pracy każde przychodzące połączenie otrzymuje odpowiednią ilość zasobów na stercie programu, więc nie ma również potrzeby alokowania zasobów dla każdego z wątków czy procesów osobno. Najbardziej popularna metoda na zarządzanie komunikacją między instancjami korzystającymi z serwera używającego technologii Node.js jest wykorzystanie framework'u Restful Api popularnie zwanego Rest. Jest to wzorzec architektury oprogramowania, który opisuje sposób operowania zapytaniami pomiędzy Api, w prosty sposób poprzez obsługę zadań oraz odpowiedzi. Został on następcą protokołu sieciowego SOAP (Simple object Access protocol), stając się wiodącym standardem. Pozwala on na eliminację zbędnej pracy i czasu wymaganego na integrację. Daje możliwość na stworzenie komunikacji bez potrzeby wiedzy na temat instancji korzystających z przesyłanych zasobów - może integrować środowiska napisane w różnych językach, działające na różnych platformach. Wykorzystuje on proste zapytania przy użyciu protokołu http oraz jego popularnie stosowanych metod takich jak post, get, put, delete, a także bardziej złożonych i używanych rzadziej jak options, head, trace oraz connect. Opis typu zasobów, jakie wymienimy, jest określony w nagłówku informacji przez kod statusu http. Poniższa tabelka prezentuje poszczególne typy statusów:

1xx Informational		
100 Continue	101 Switching Protocols	102 Processing (WebDAV)
2xx Success		
★ 200 OK 203 Non-Authoritative Information 206 Partial Content 226 IM Used	★ 201 Created ★ 204 No Content 207 Multi-Status (WebDAV)	202 Accepted 205 Reset Content 208 Already Reported (WebDAV)
3xx Redirection		
300 Multiple Choices 303 See Other 306 (Unused)	301 Moved Permanently ★ 304 Not Modified 307 Temporary Redirect	302 Found 305 Use Proxy 308 Permanent Redirect (experimental)
4xx Client Error		
★ 400 Bad Request ★ 403 Forbidden 406 Not Acceptable ★ 409 Conflict 412 Precondition Failed 415 Unsupported Media Type 418 I'm a teapot (RFC 2324) 423 Locked (WebDAV) 426 Upgrade Required 431 Request Header Fields Too Large 450 Blocked by Windows Parental Controls (Microsoft)	★ 401 Unauthorized ★ 404 Not Found 407 Proxy Authentication Required 410 Gone 413 Request Entity Too Large 416 Requested Range Not Satisfiable 420 Enhance Your Calm (Twitter) 424 Failed Dependency (WebDAV) 428 Precondition Required 444 No Response (Nginx) 499 Client Closed Request (Nginx)	402 Payment Required 405 Method Not Allowed 408 Request Timeout 411 Length Required 414 Request-URI Too Long 417 Expectation Failed 422 Unprocessable Entity (WebDAV) 425 Reserved for WebDAV 429 Too Many Requests 449 Retry With (Microsoft)
5xx Server Error		
★ 500 Internal Server Error 503 Service Unavailable 506 Variant Also Negotiates (Experimental) 509 Bandwidth Limit Exceeded (Apache) 598 Network read timeout error	501 Not Implemented 504 Gateway Timeout 507 Insufficient Storage (WebDAV) 510 Not Extended 599 Network connect timeout error	502 Bad Gateway 505 HTTP Version Not Supported 508 Loop Detected (WebDAV) 511 Network Authentication Required

Rysunek 2.3: Kody statusu protokołu http - źródło:
<http://mokandra.blogspot.fi/2015/10/http-status-code-definitions.html>

W celu wymiany zasobów framework wykorzystuje najpopularniejsze sposoby opisu struktury służące do opisu obiektów, takie jak najczęściej spotykany, najprostszы w użyciu format json (JavaScript Object Notation) oraz mniej popularny w technologii Rest format xml (eXtensible Markup Language). Różnica pomiędzy dwiema formami jest taka, że xml jest językiem niezdefiniowanych znaczników, a json tylko sposobem na prezentację właściwości obiektu. Porównanie obydwu struktur: Struktura w formacie json opisująca listę pracowników:

```

1 {"employees": [
2   {"firstName":"James", "lastName":"Bond" },
3   {"firstName":"David", "lastName":"Klint" },
4   {"firstName":"Peter", "lastName":"Blom" }
5 ]}
```

Ta sama struktura opisana w formacie xml: <employees>

```
1 <employee>
2   <firstName>James</firstName> <lastName>Bond</lastName>
3 </employee>
4 <employee>
5   <firstName>David</firstName> <lastName>Klint</lastName>
6 </employee>
7 <employee>
8   <firstName>Peter</firstName> <lastName>Blom</lastName>
9 </employee>
10 </employees>
```

2.4 Alternatywne rozwiązania

2.4.1 Vert.x

Pierwszą alternatywą dla technologii Node.js jest framework Vert.x. Pracuje on na środowisku JVM (Java Virtual Machine) i tak jak i Node.js, jest środowiskiem sterowanym zdarzeniami. Został napisany w 2011 roku przez zainspirowanego technologią Node.js Tim Foxa przy użyciu między innymi języków Java, JavaScript, Python, Ruby. Dzięki platformie JVM aplikacje mogą być stworzone dla wielu systemów operacyjnych. Wykorzystuje on niskopoziomową bibliotekę Netty, służącą do asynchronicznej obsługi urządzeń wejścia/wyjścia w celu obsługi aplikacji w architekturze klient-serwer. Do wytwarzania oprogramowania w Vert.x możemy użyć różnych języków programowania - Java, Javascript, Groovy, Ruby, Ceylon, Scala lub Kotlin. Jest więc dobrą alternatywą w przypadku potrzeby użycia języka innego niż javascript. Potrafi on szybciej od technologii Node.js odpowiedzieć na proste żądania, natomiast jest mniej efektywny na przykład przy pracy z gniazdami sieciowymi.

2.4.2 Twisted

Kolejną alternatywą dla technologii Node.js jest framework Twisted. Został stworzony w 2002 roku przez Glyph Lefkowitz'a przy użyciu technologii Python. Wspiera koncepcje środowiska sterowanego zdarzeniami, więc nie wymaga tworzenia osobnych wątków dla obsługi wielu procesów w jednym czasie. Do procesu wytwarzania oprogramowania przy jego użyciu niezbędna jest umiejętność pisania w języku python. Działanie framework'u twisted opiera się głównie na wykorzystaniu

protokołów sieciowych, między innymi HTTP, XMPP, TCP, IMAP, SSH czy Udp. Dzięki długiemu okresu rozwoju framework oferuje wsparcie wielu rozwiązań sieciowych. Nie oferuje tak efektywnej pracy jak technologia Node.js, jednak może okazać się lepszym wyborem, jeśli zachodzi potrzeba wykorzystania specyficznego protokołu sieciowego, który nie został jeszcze zaimplementowany w Node.js.

2.4.3 Ringojs

Ostatnią wymienioną alternatywą jest platforma Ringojs. Napisany został w języku java przez Hannesa Wallnöfer w 2010 roku. Służy do tworzenia aplikacji w języku javascript po stronie serwera działających również w koncepcji środowiska sterowanego zdarzeniami dzięki wykorzystaniu technologii Rhino, która zajmuje się obsługą wielowątkowości. Niewątpliwą zaletą Ringojs jest możliwość wykorzystania kodów w języku java przez serwer, zapewniając integracje frameworka z istniejącymi środowiskami java'owymi. W porównaniu z technologią Node.js, Ringojs jest efektywniejszy dla rozwiązań obsługujących mniejszą ilość jednoczesnych zapytań, oraz w przypadku alokacji dużych plików dzięki pracy w środowisku JVM. Oznacza to, że zyskuje przewagę w przypadku prostych narzędzi, natomiast nie gwarantuje takiej skalowności jak Node.js.

Rozdział 3

Założenia i specyfikacja aplikacji

3.1 Specyfikacja problemu

Potrzebny jest nowoczesny system do zarządzania tablicami zadań, który będzie posiadał autoryzowany dostęp dla użytkowników systemu. Został zauważony problem braku prostej w obsłudze, intuicyjnej aplikacji pozwalającej we współpracy z innymi użytkownikami na zarządzanie zarówno podziałem jak i procesem realizacji wyszczególnionych zadań. W celu reakcji na zapotrzebowanie zostanie zaprojektowany system informatyczny, pracujący w technologii Node.js, ze względu na możliwą do osiągnięcia niezawodność oraz dostępność nawet przy jednoczesnej obsłudze wielu korzystających z aplikacji klientów. Nie powinien on wymagać żadnych procesów wdrożeniowych w celu zrozumienia obsługi narzędzia, ponieważ celem projektu jest stworzenie narzędzia, które wspiera, a nie dodatkowo komplikuje określone zadania. System powinien w przejrzysty sposób prezentować proces realizacji poszczególnych zadań zorganizowanych w tablicach. Właściciel określonej tablicy powinien mieć możliwość zarządzania dostępem do tablicy poprzez udostępnianie lub ograniczanie jej treści pozostałym użytkownikom. Aplikacja zapewni subskrybentom nieprzerwaną możliwość nadzoru, wglądu oraz określania aktualnych statusów dowolnych procesów.

3.2 Wymagania funkcjonalne

Analiza wymagań funkcjonalnych umożliwia zidentyfikowanie i opisanie pożdanego zachowania systemu. Umożliwiają określenie usług oferowanych przez system, reakcji na dane wejściowe oraz zachowania w określonych sytuacjach.

- Wymiana komunikatów w modelu klient-serwer.
- Możliwość utworzenia do 1000000 kont użytkowników.
- Możliwość zmiany danych użytkownika.
- Weryfikacja adresu email przed aktywacją.
- Możliwość resetowania hasła za pomocą adresu email.
- Utrzymanie do 100 tablic jednocześnie dla użytkownika.
- Możliwość usuwania tablic.
- Obsługa do 1000 równoległych członków jednej tablicy.
- Możliwość zarządzania członkami tablicy.
- Obsługa do 10000 równoległych zadań dla tablicy.
- Możliwość usuwania zakończonych zadań.
- Utrzymanie do 1000 statusów do jednego zadania.
- Zapamiętywanie terminu zmiany statusu zadań.
- Walidacja poprawności wprowadzanych przez użytkownika danych zarówno po stronie klienta jak i serwera.
- W razie wystąpienia nieprawidłowych danych system powiadamia o błędach.
- Zapewnienie okna pomocy opisującego korzystanie z aplikacji.
- Możliwość otrzymywania powiadomień o aktualnym statusie tablic oraz poszczególnych zadań.
- System współpracuje z bazą danych.

3.3 Wymagania pozafunkcjonalne

Wymagania niefunkcjonalne opisują kryteria umożliwiające dokonanie oceny działania systemu i elementów mających wpływ na satysfakcję użytkownika. Zdefiniowane zostały następujące wymagania niefunkcjonalne:

- Dostęp do określonych zasobów musi być chroniony poprzez login i hasło użytkownika.
- Budowa interfejsu użytkownika musi być możliwie intuicyjna i prosta w obsłudze.
- Proste analizowanie i diagnozowanie błędów oraz sytuacji problemowych.
- Budowa zapewnia łatwe wprowadzanie koniecznych zmian.
- Jest w stanie obsłużyć jednocześnie do 100.000 użytkowników.
- Czas uruchomienia jest nie dłuższy niż 10 sekund.
- Umożliwia efektywne testowanie wprowadzonych zmian.
- Posiada możliwość obsługi poprzez różne przeglądarki internetowe i środowiska.
- Zrozumiałы dla wszystkich. Czas przyswojenia nie przekracza 15 minut.
- Ma możliwość współistnienia z innymi modułami.
- Spełnia wszystkie normy prawne w naszym kraju.

Rozdział 4

Opracowanie aplikacji

4.1 Mean Stack

4.1.1 Wstęp

Do opracowania rozwiązania zdecydowałem się skorzystać z Mean Stack. Skrót odnosi się do frameworków oraz technologii Mongodb, Express, AngularJS oraz Node.js. Współpracując razem zapewniają bardzo szybkie, efektywne tworzenie skalowalnych aplikacji wieloplatformowych. Do użycia wszystkich wymagany jest tylko jeden język programowania - javaScript, zarówno do obsługi warstwy frontowej aplikacji jak i backendowej. Wszystkie technologie są dostępne w pełni bezpłatnie.

Jako iż Technologia Node.js została już opisana, przejdę do opisu pozostałych technologii.

4.1.2 MongoDB

Napisany w języku c++ system zarządzania bazą danych zorientowany na dokumenty. Operuje na nierelacyjnych bazach danych. Używa struktur json jako schematów budowy bazy danych. Udostępnia całkowitą dowolność w budowie struktury wymaganej bazy danych. Wykorzystuje proste, ale zapewniające szerokie możliwości zapytania bazodanowe.

4.1.3 ExpressJS

Framework służący do szybkiego wymagającego jak najmniejszych nakładów pracy wytwarzania zarówno backendu aplikacji internetowych, jak i aplikacji mobilnych.

Dostarcza zbiór określonych klas i metod. Jest to najpopularniejszy framework do tworzenia serwerów w technologii Node.js.

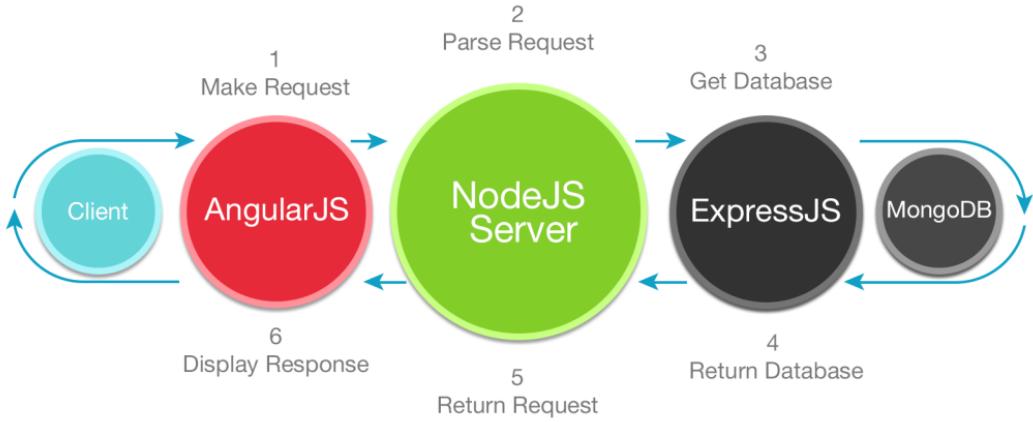
4.1.4 AngularJS

Wykorzystujący wzorzec projektowy MVC (Model View Controller) polegający na oddzieleniu od siebie poszczególnych warstw aplikacji - logiki, widoku oraz modelu komunikacji, framework wykorzystujący dodatkowe tagi w języku html w celu prostego w obsłudze i niewymagającego dodatkowej logiki napisanej w języku javaScript tworzenia dynamicznych stron internetowych.

4.2 Komunikacja

4.2.1 Rozwiążanie problemu komunikacji

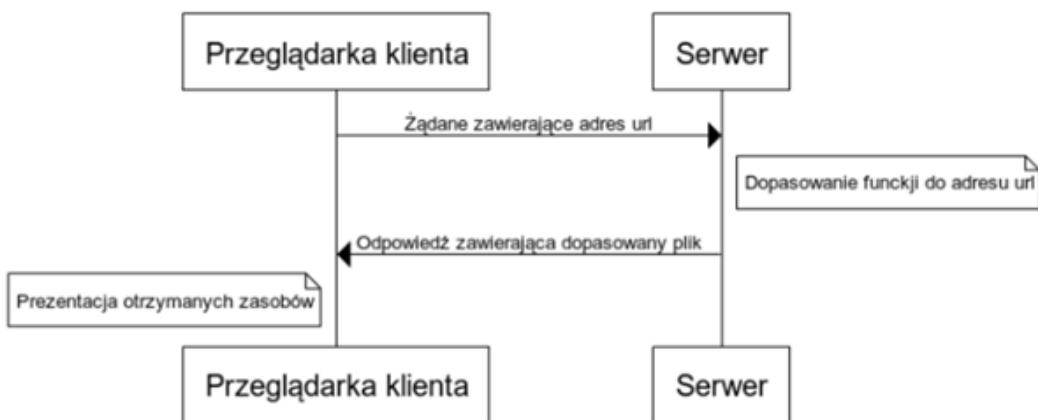
Do komunikacji między warstwą frontendową i backendową po stronie klienta wykorzystałem wysokopoziomową bibliotekę Ajax oraz technologię Restful api. Ajax polega na zarządzaniu asynchroniczną komunikacją. Dzięki temu aplikacja może wykonywać inne funkcje, mimo oczekiwania na odpowiedź ze strony serwera, za sprawą odseparowania warstwy wymiany danych od pozostałych warstw aplikacji. Do opisu wymienianych struktur użyłem standardu json, ponieważ wymaga on mniejszych nakładów pracy od standardu xml. Powyższe frameworki współpracują ze sobą w bardzo intuicyjny i przejrzysty sposób. Angular prezentuje użytkownikowi dynamiczną aplikację internetową, odpowiada za przyjmowanie danych i z pomocą ajaxa wysyła oraz odbiera dane wysyłane do serwera. Serwer Node.js z użyciem technologii express, wykorzystując metodę routingu, dopasowuje zapytanie do odpowiednich funkcji serwisu, przetwarza otrzymane dane, przy współpracy z bazą danych zarządzana przez Mongodb przechowuje informacje użytkowników serwisu oraz w odpowiedzi zwraca odpowiednie dane z powrotem do warstwy frontowej prezentującej dane użytkownikowi.



Rysunek 4.1: Przepływ komunikacji w MEAN Stack źródło: <https://www.dealfuel.com/seller/mean-stack-tutorial/>

4.2.2 Schematy wymiany zasobów

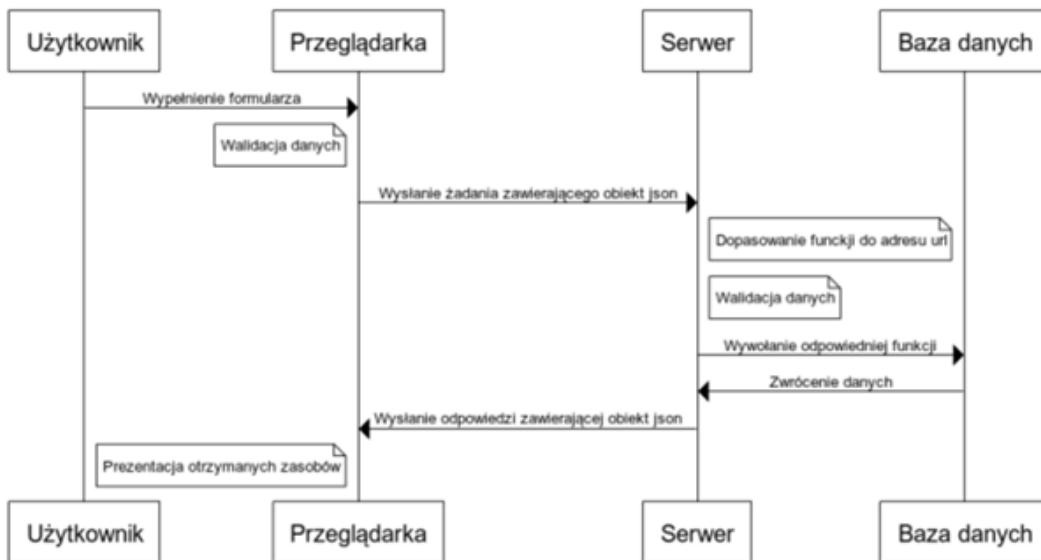
Uniwersalny dla aplikacji schemat procesu pobrania statycznych zasobów z serwera wykorzystuje metodę get. Do zasobów tych należy między innymi plik html, zawierający warstwę prezentacji aplikacji, plik css zawierający styl warstwy prezentacji, czy plik zawierający funkcje wykorzystywane przez aplikacje w języku javaScript. Proces prezentuje się następująco:



Rysunek 4.2: Pobranie statycznych zasobów, źródło: Opracowanie własne

1. Klient przy użyciu przeglądarki wysyła pod określony adres url serwera zapytanie zawierające pożądany zasób.
2. Serwer otrzymuje zapytanie i dopasowuje określoną funkcję po adresie url serwera.
3. Funkcja wybiera dopasowane do zapytania zasoby i wysyła odpowiedź do aplikacji.
4. Aplikacja odbiera i zaczyna korzystać z zasobów.

Uniwersalny dla aplikacji schemat procesu wymiany danych określonych dla specyficznego użytkownika wykorzystuje metodę post. Przykładowe dane wymieniane przez aplikacje to login, hasło, informacje odnośnie żądań użytkownika, komentarze do zadań użytkownika. Przepływ danych wygląda następująco:



Rysunek 4.3: Wymiana zasobów w formacie Json źródło: Opracowanie własne

1. Dane zostają pobrane od użytkownika w określonym formularzu w warstwie frontendowej.
2. Dane zostają zwalidowane pod kątem poprawności po stronie użytkownika.
3. Jesli dane są poprawne, zostają zorganizowane w obiekcie json i wysłane do serwera, w przeciwnym wypadku zostaje zwrócony błąd.

4. Serwer odbiera zapytanie i przekazuje je do odpowiedniej funkcji.
5. Dane zostają zwalidowane pod kątem poprawności po stronie serwera.
6. Jeśli dane są poprawne, serwer realizuje określona funkcję przy pomocy zapytań bazodanowych, w przeciwnym wypadku zostaje przygotowany komunikat o błędzie.
7. Odpowiedź na zapytanie bazodanowe zostaje odebrana i zostają przygotowane dane do wysłania.
8. Przygotowane dane zostają wysłane w określonej odpowiedzi.
9. Warstwa frontendowa otrzymuje odpowiedź i zostaje odświeżona warstwa prezentacji aplikacji.

4.2.3 Kod po stronie klienta wysyłający żądanie

Przykładowy kod wykonywany po stronie klienta, wysyłający żądanie do serwera, żądające określonego zasobu przy użyciu frameworku Angularjs: Plik w języku javaScript zawierający definicje modelu i kontrolera:

```

1 var main = angular.module("main", []);           // create
angular module
2
3 main.controller('SignInController', function($scope) {      // 
create controller for module
4   $scope.SignIn=function(){           // callback function as
field in model
5     if ($scope.login && $scope.password) {      // if both
field are filled
6       var xhr = new XMLHttpRequest();           //create request
object
7       xhr.open("POST", http://site.com/authorization, true); // 
set method type, domain address and is it
asynchronous
8       xhr.setRequestHeader("Content-type", "application/json");
// set header with information of content type
9       req.send(JSON.stringify({ login: $scope.login,
password: $scope.password})); // Send json
object with login and password
10

```

```
11 req.onreadystatechange = function() {           // create
   callback, called on every state change
12 if (req.readyState === 4) {                  // readyState 4 is 4
   when respond is got
13 if (req.status === 200) {                   // http status code
   200 - ok
14 var resObj = JSON.parse(this.responseText); // parse response text to json object
15 $scope.login=resObj.login;                 // set password from
   response
16 $scope.password=$resObj.password;         // set password
   from response
17 alert("Login and password are valid");    // alert
   user
18 }
19 else{                                     // all other http statusses
20   alert(req.status+":"+req.responseText); // alert user
21 }
22 }
23 }
24 }
25 }
26 }
```

Plik w języku html zawierający definicję widoku:

```
1 <!DOCTYPE html>
2 <!-- Apply view to model -->
3 <body ng-app = "main">
4   <!-- Create form for request -->
5   <form class="main" name="form">
6     <!-- Get user login -->
7     <input type = "text" ng-model = "login" name="login"
8       <!-- Set field validation -->
9       required ng-pattern='/[a-zA-Z0-9._-]+$/' ng-maxlength="20
10      " ng-minlength="3">
11     <!-- Get user password -->
12     <input type = "text" ng-model = "password" name="password"
13       <!-- Set field validation -->
14       required ng-pattern='/[a-zA-Z0-9._-]+$/' ng-maxlength="20
15      " ng-minlength="3">
16   </form>
17 </body>
```

4.2.4 Kod po stronie serwera obsługujący odebranie żądania

Przykładowy kod, wykonywany po stronie serwera, obsługujący odebrane żądanie od klienta przy użyciu technologii Node.js oraz frameworków express i mongodb:

```
1 var express = require('express');           // load express module
2 var bodyParser = require('body-parser');    // load module for
   parsing json data
3 var DataBase = require('mongodb').MongoClient; // load MongoDB
   database module
4 var Promise = require('promise');          // load asynchronous returns
   from functions module
5
6 var app = express();                     // create express configuration object
7 app.use(bodyParser.json()); // load bodyParser to app object
8
9 // check if database contains given user
10 var Authorization=function (login, password) {
11   return new Promise(function (fulfill, reject) {
```

```

12     dataBase.connect(mongodb://localhost:27017/db, function(
13         err, db).done(function (db) {
14             db.collection("Users").findOne({ login: login,
15                 password: password }, function (err, result) {
16                 if (result != null) {
17                     fulfill(true);
18                 }
19                 else {
20                     fulfill(false);
21                 }
22             });
23         });
24     });
25
26 // check is text format correct
27 function TextValidation(text, min, max) {
28     if (min === undefined) min = 0;
29     if (max === undefined) max = Infinity;
30     return (text !== undefined) && (/^[\w\W]{min,max}/).test(text);
31 }
32
33 // check is user credentials data correct
34 function UserValidation(body) {
35     return new Promise(function(fulfill, reject) {
36         if (!TextValidation(body.login, 3, 20) || !
37             TextValidation(body.password, 3, 20)) {
38             fulfill(false);
39             return;
40         }
41         Authorization(body.login, body.password).done(function (
42             authentication) {
43             fulfill(authentication);
44         });
45     })
}

```

```

46 //function run when server gets post request with url /
  authorization
47 app.post('/authorization', function (req, res) {
48   UserValidation(req.body).done(function(valid) {
49     res.setHeader('Content-Type', "text/html");
50     if (valid) {
51       res.statusCode = 200;
52     }
53     else {
54       res.statusCode = 401; // error has happened
55       res.write("Invalid login or password");
56     }
57     res.end();
58   });
59 });
60
61 //start server
62 var server = app.listen(8081, function () {});

```

4.3 Wykorzystane moduły

4.3.1 Moduły zewnętrzne

Zewnętrzne moduły, zainstalowane za pomocą zbioru repozytorium npm, które użyłem do realizacji rozwiązania, to:

1. express - opisany powyżej
2. fs - file stream, odpowiada za obsługę zapisu oraz wczytywania plików. Wykorzystywany w celu wczytywania plików statycznych aplikacji, takich jak index.htm (strona główna w formacie html), main.js (funkcje w języku javaScript), style.css (plik arkuszu stylów, odpowiadający za styl prezentacji), favicon (ikona serwisu wyświetlana w oknie przeglądarki), czy losowo wybieranych zdjęć wyświetlanych w tle serwisu.
3. path - obsługa różnic między systemami operacyjnymi. Dzięki temu modułowi możemy zapewnić całkowitą sprawność naszego serwisu, niezależnie od środowiska uruchomieniowego. Pozwala niwelować różnice w lokalizacji plików systemowych, czy łącznikach pomiędzy poszczególnymi folderami przy specyfikacji ścieżki do pliku.

4. body-parser - dostarcza możliwość analizowania danych, załączonych do odebranego przez serwer żądania, wykorzystany został w celu odczytu poszczególnych wartości odebranych w formacie json.
5. promise - odpowiada za operowanie wynikami otrzymanymi w wyniku asynchronicznych funkcji. Dzięki temu modułowi nie musimy synchronicznie czekać na otrzymanie zwrotu z kolejnych funkcji, natomiast możemy przy użyciu wywołań zwrotnych zareagować po otrzymaniu określonego wyjścia.
6. cookie-parser - zapewnia możliwość wykorzystania plików cookie dla specyficznego klienta. Dzięki temu modułowi możemy ustawać, usuwać lub edytować wartości plików cookie, które zostaną przydzielone dla konkretnego użytkownika w ramach całej sesji komunikacji z użytkownikiem lub przez określony przez nas czas.

4.3.2 Moduły wewnętrzne

Wewnętrzne moduły, czyli takie, które zostały napisane przeze mnie specjalnie na użytk projektu, to:

1. emails.js (wykorzystujący zewnętrzny moduł mongodb) - moduł zapewnia komunikację z bazą danych w technologii mongodb, poprzez wywoływanie odpowiednich funkcji. Został dostosowany bezpośrednio do obsługi bazy danych dla wykonywanego projektu. Dzięki temu plik główny serwera nie musi znać budowy, ani wykonywać operacji bezpośrednio na dokumentach bazy danych.
2. db.js (wykorzystujący zewnętrzny moduł nodemailer) - dostarcza obsługę wysyłania wiadomości email do użytkowników serwisu w przypadku zjawienia się określonych sytuacji, takich jak na przykład otrzymanie nowego zaproszenia czy zmiana statusu zadania. Moduł dostarcza jedną, prostą w obsłudze funkcję zapewniającą obsługę wiadomości email.

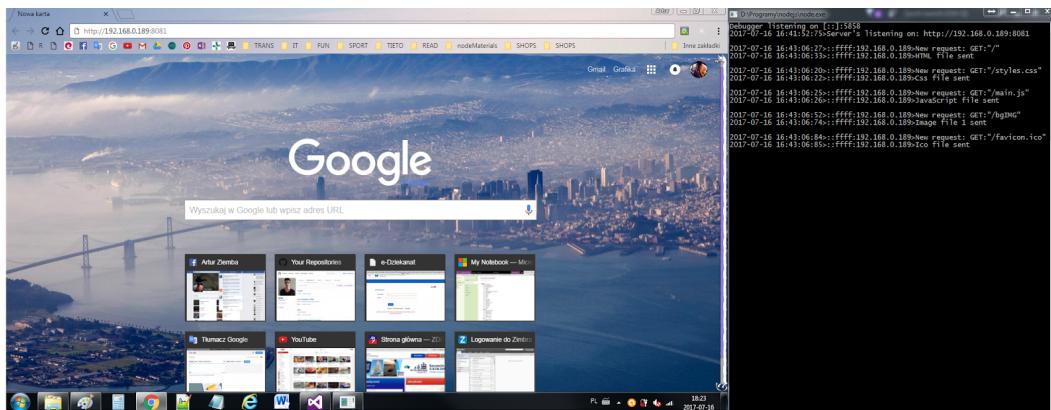
Rozdział 5

Testy aplikacji

5.1 Testy manualne

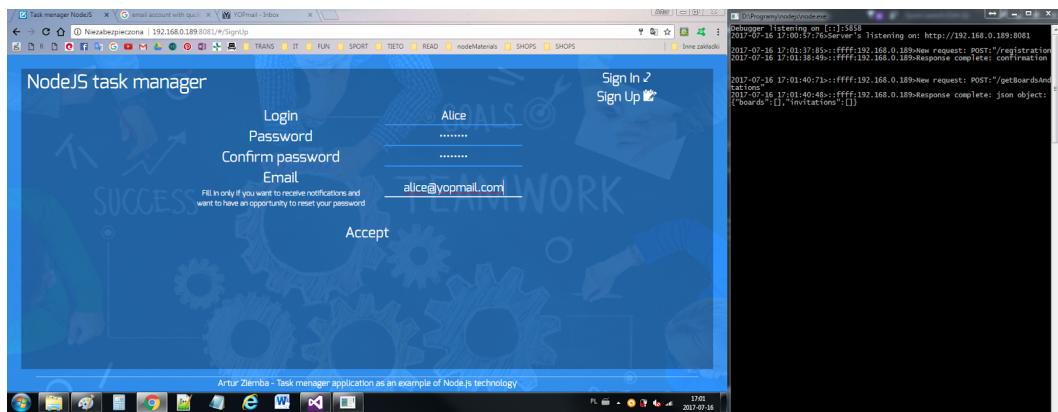
Po wykonaniu aplikacji zostały przeprowadzone testy manualne, sprawdzające poprawność dostarczanej przez serwis funkcjonalności.

5.1.1 Uzyskanie dostępu do statycznych zasobów serwisu



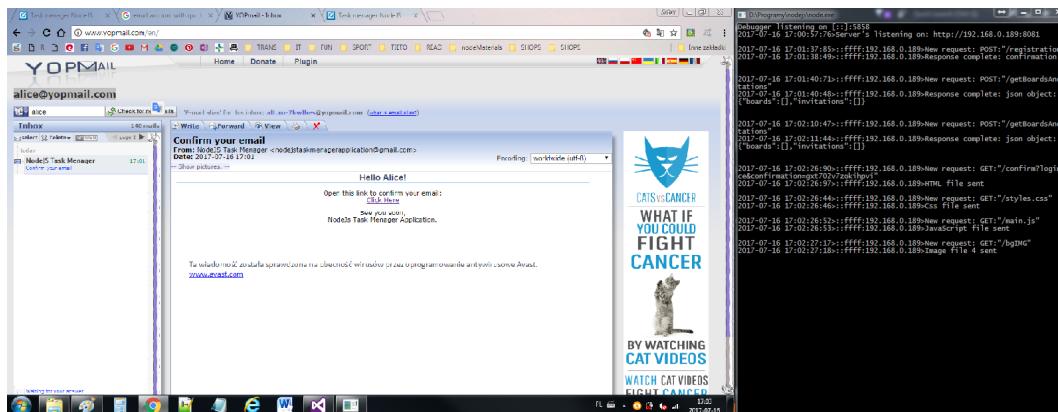
Użytkownik wysyła żądanu pod adres serwera. Otrzymuje odpowiedź zawierającą pliki statyczne serwisu (plik html, css, js, zdjecie w tle i favicon).

5.1.2 Rejestracja w serwisie



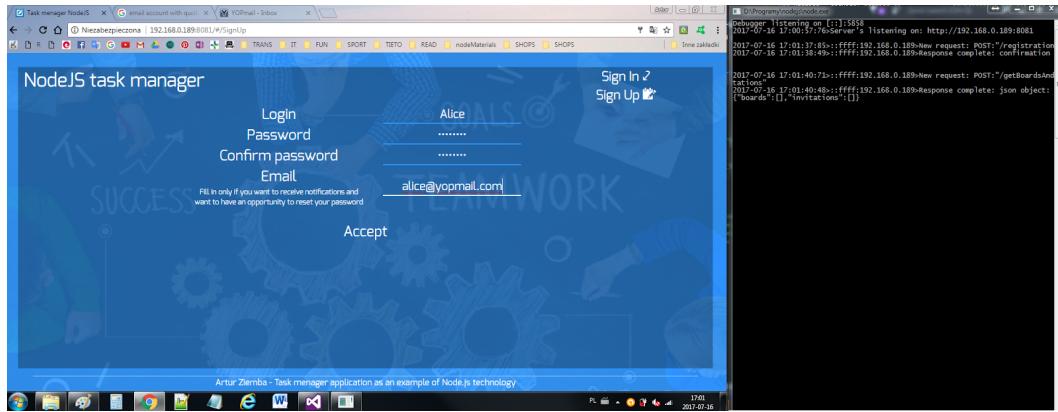
Użytkownik wypełnia formularz rejestracyjny na stronie serwisu i wysyła dane do serwera. Serwer sprawdza poprawność otrzymanych danych i w odpowiedzi przesyła stronę główną aplikacji dostępną po zalogowaniu na konto lub informacje o błędzie.

5.1.3 Potwierdzenie adresu email



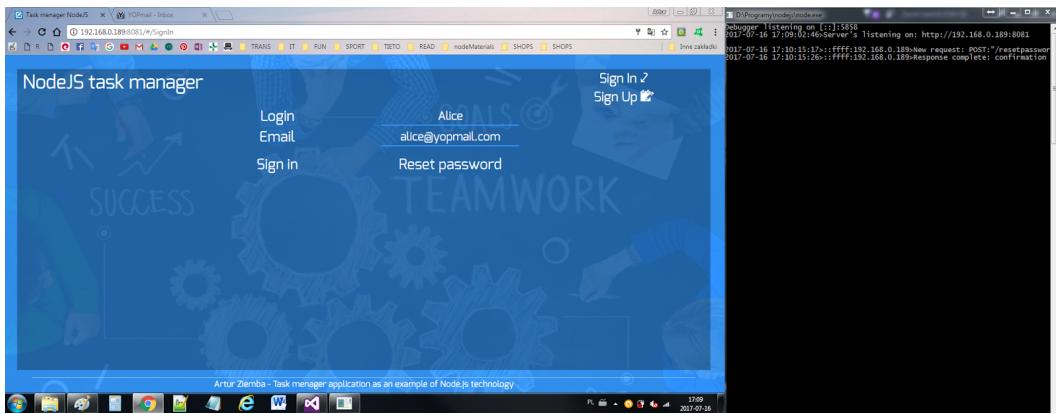
Po podaniu adresu email użytkownik otrzymuje wiadomość email zawierającą link aktywacyjny. Po kliknięciu w link. Serwer sprawdza czy otrzymany w linku kod jest prawidłowy dla podanego użytkownika. Następnie aktywuje adres email dla odpowiedniego użytkownika, wysyła w odpowiedzi stronę główną aplikacji oraz wiadomość email o potwierdzeniu.

5.1.4 Logowanie się w serwisie

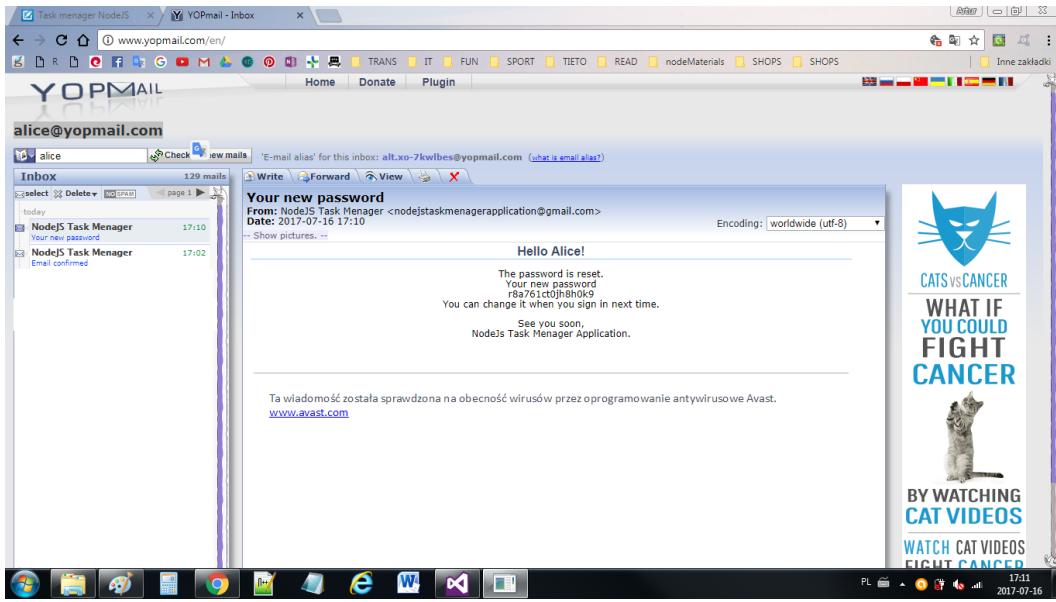


Użytkownik wypełnia formularz na stronie serwisu i przesyła żądanu do serwera. Serwer sprawdza istnienie konta w serwisie. W odpowiedzi użytkownik otrzymuje stronę główną aplikacji lub błąd o niepoprawnych danych. W czasie bycia zalogowanym, co określony czas zostaje wysłane żądanie aktualizacji danych. Aktualizacja następuje również każdorazowo po otrzymaniu pomyślnego potwierdzenia wykonania operacji przez serwer.

5.1.5 Resetowanie hasła użytkownika

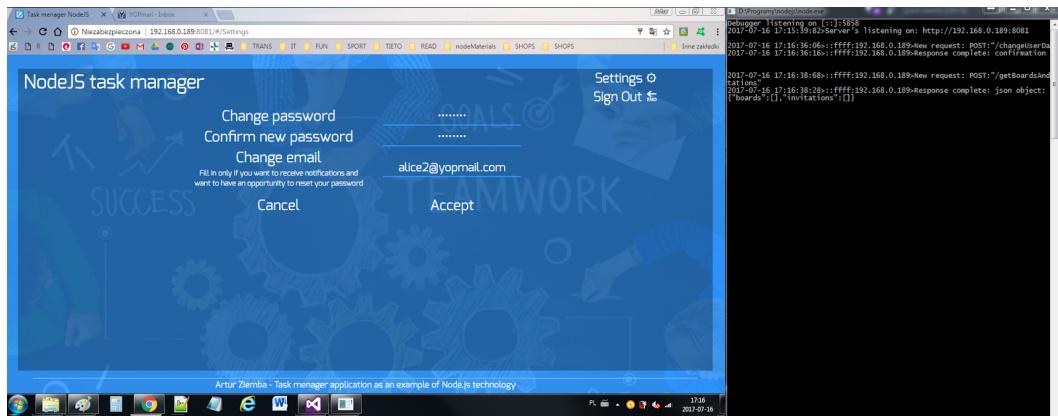


Funkcja jest dostępna po potwierdzeniu adresu email dla konta. Po wypełnieniu odpowiedniego formularza na stronie głównej serwisu, użytkownik wysyła żądanie do serwera. Serwer sprawdza podane dane. W odpowiedzi wysyła informacje o przetworzeniu żądania.

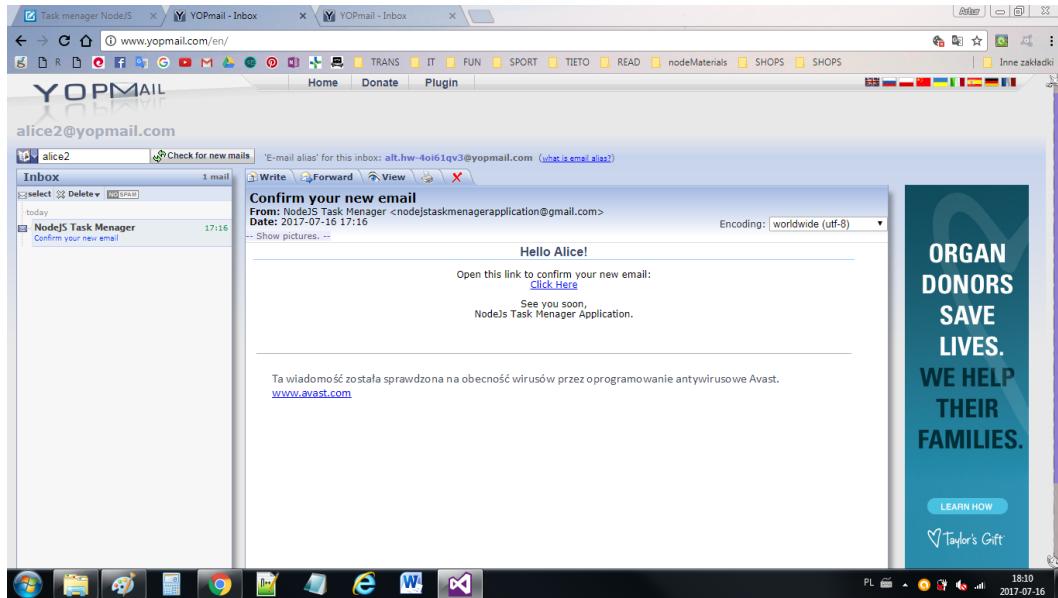


Otrzymujemy także wiadomość email zawierającą nowe hasło do serwisu.

5.1.6 Zmiana danych użytkownika

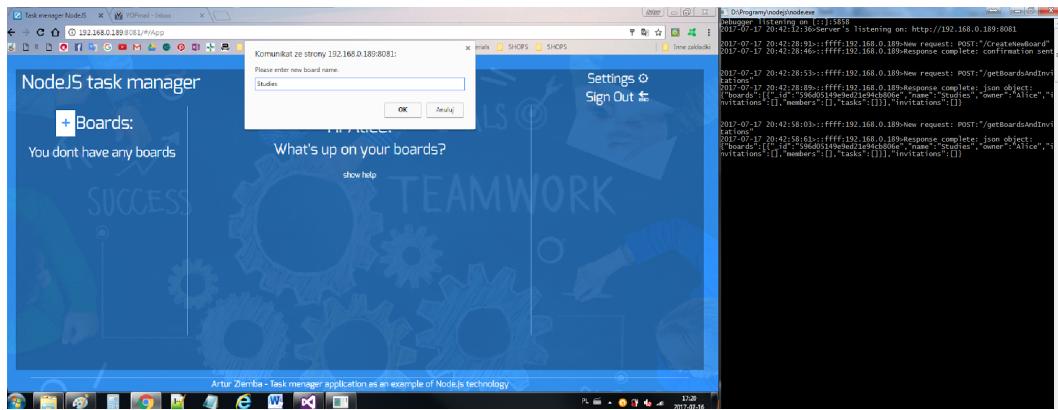


Po zalogowaniu się i wypełnieniu odpowiedniego formularza na stronie serwisu użytkownik wysyła żądanie do serwera. Serwer sprawdza poprawność danych. Jeśli są poprawne, aktualizuje je. W przeciwnym wypadku, zwraca komunikat o błędzie. W odpowiedzi serwer wysyła stronę główną aplikacji.



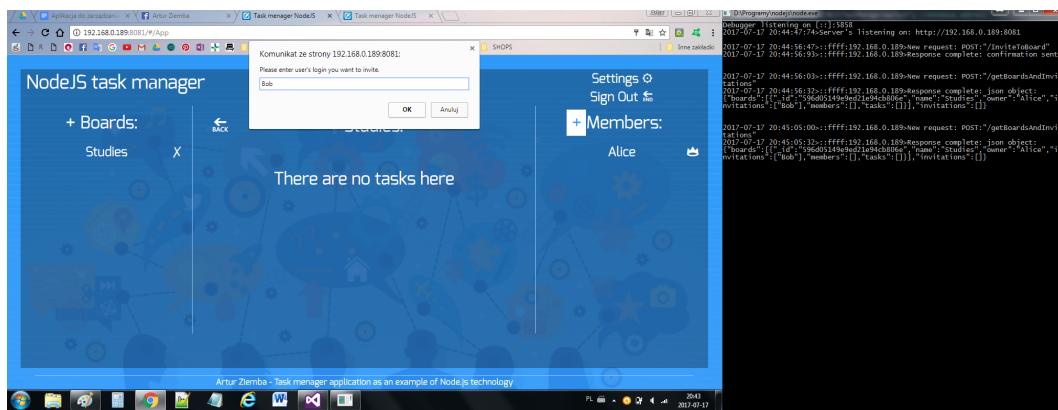
Jeśli zmieni się również adres email, użytkownik musi go ponownie zweryfikować.

5.1.7 Utworzenie nowej tablicy z zadaniami



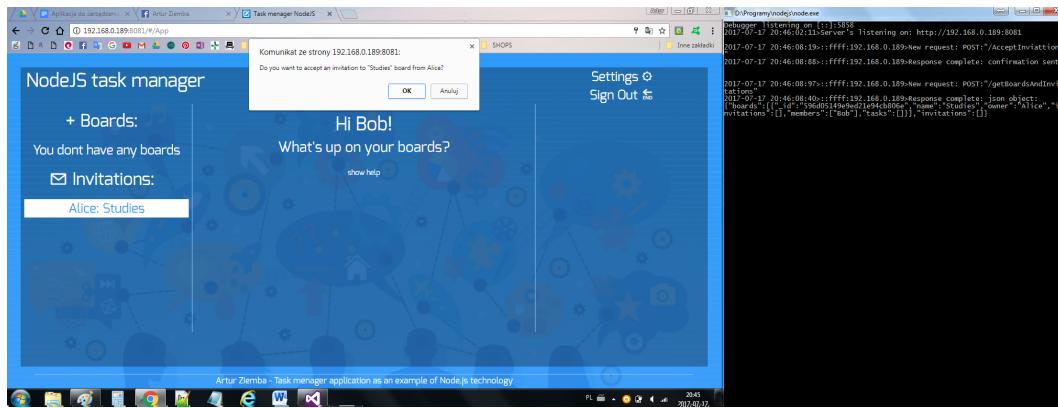
Po zalogowaniu się użytkownik wybiera opcje dodania tablicy, wpisuje wymagane dane i wysyła je do serwera. Serwer sprawdza poprawność danych. Jeśli są poprawne, serwer tworzy nową tablicę dla użytkownika i wysyła potwierdzenie w odpowiedzi. W przeciwnym wypadku odpowiada komunikatem o błędzie.

5.1.8 Wysyłanie zaproszenia do tablicy



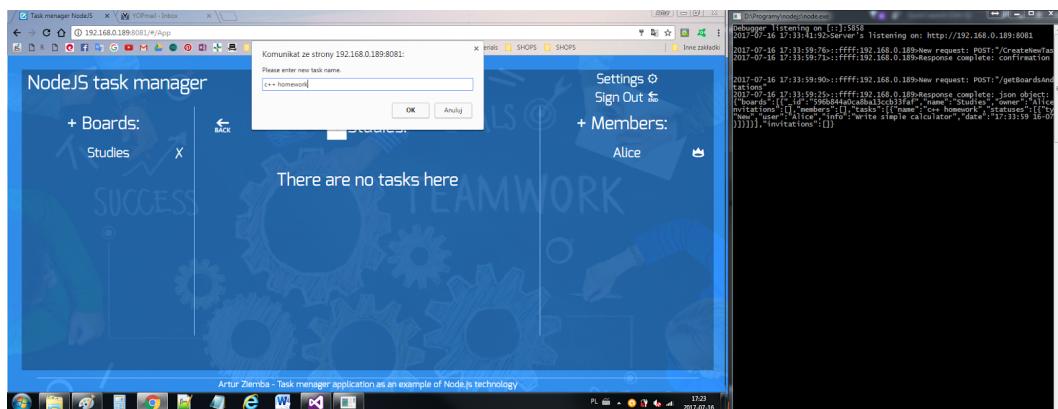
Po zalogowaniu się użytkownik wybiera opcje wysłania zaproszenia do wybranej tablicy, wypełnia dane i wysyła żądanie do serwera. Serwer sprawdza poprawność danych. W odpowiedzi wysyła potwierdzenie o wysłanym zaproszeniu lub komunikat o błędzie.

5.1.9 Akceptacja zaproszenia



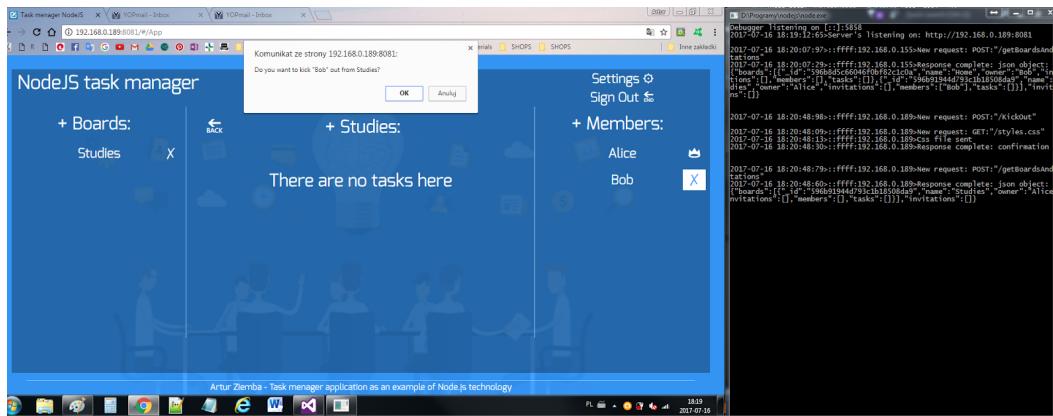
Po zalogowaniu się użytkownik wybiera opcje zaakceptowania oczekującego zaproszenia i wysyła żądanu do serwera. Serwer sprawdza poprawność danych. Odpowiada potwierdzeniem lub komunikatem o błędzie.

5.1.10 Odrzucenie zaproszenia



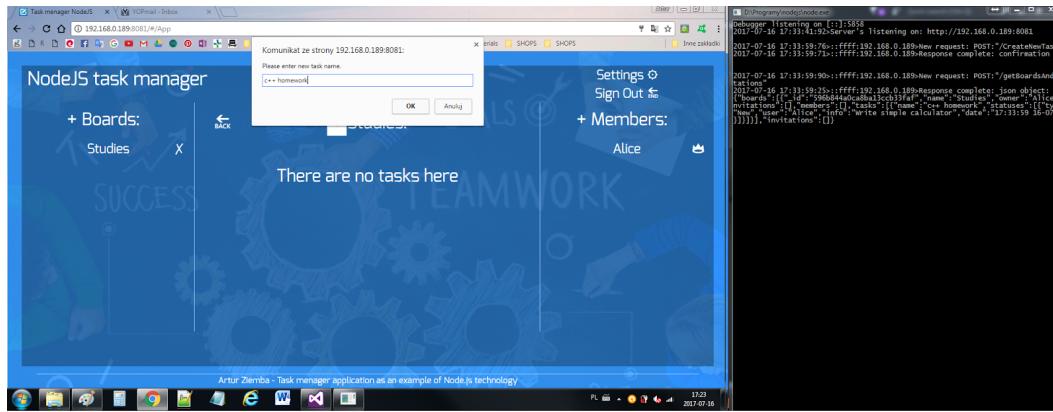
Po zalogowaniu się użytkownik wybiera opcje odrzucenia oczekującego zaproszenia i wysyła żądanu do serwera. Serwer sprawdza poprawność danych. Odpowiada potwierdzeniem lub komunikatem o błędzie.

5.1.11 Wyrzucenie użytkownika z tablicy



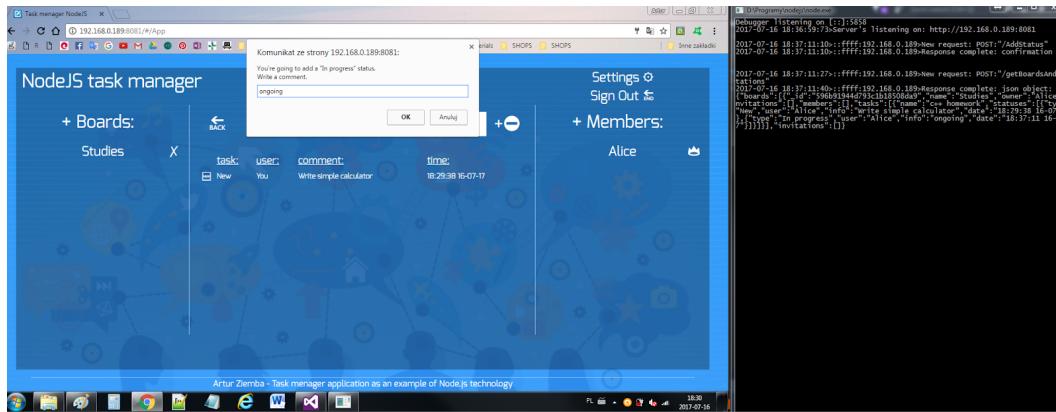
Po zalogowaniu się użytkownik wybiera opcje wyrzucenia użytkownika z określonej tablicy i wysyła żądanu do serwera. Serwer sprawdza poprawność danych. Odpowiada potwierdzeniem lub komunikatem o błędzie.

5.1.12 Dodanie zadania



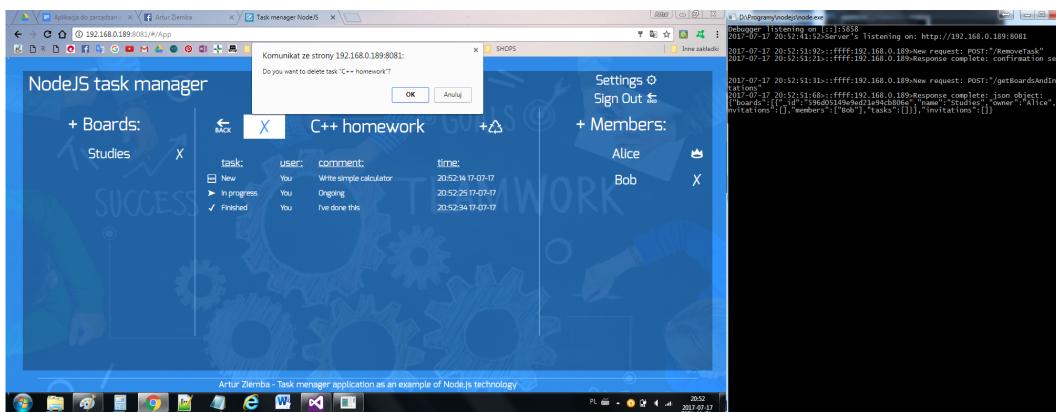
Po zalogowaniu i wybraniu odpowiedniej tablicy użytkownik wybiera opcje dodania zadania, wypełnia dane i przesyła żądanu do serwera. Serwer sprawdza poprawność danych. Odpowiada potwierdzeniem lub komunikatem o błędzie.

5.1.13 Dodanie statusu zadania



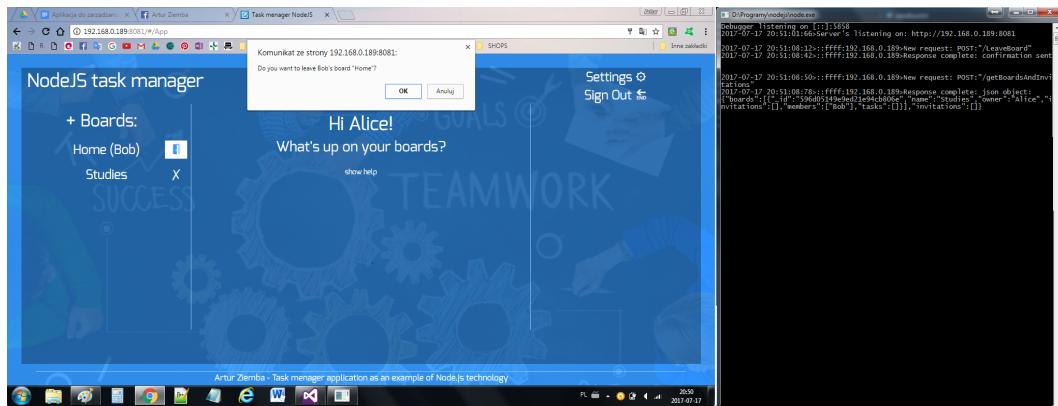
Po zalogowaniu, wybraniu odpowiedniej tablicy i zadania użytkownik wybiera opcje dodania nowego statusu, uzupełnia dane i przesyła żądanie do serwera. Serwer sprawdza poprawność danych. Odpowiada potwierdzeniem lub komunikatem o błędzie.

5.1.14 Usuwanie zadania



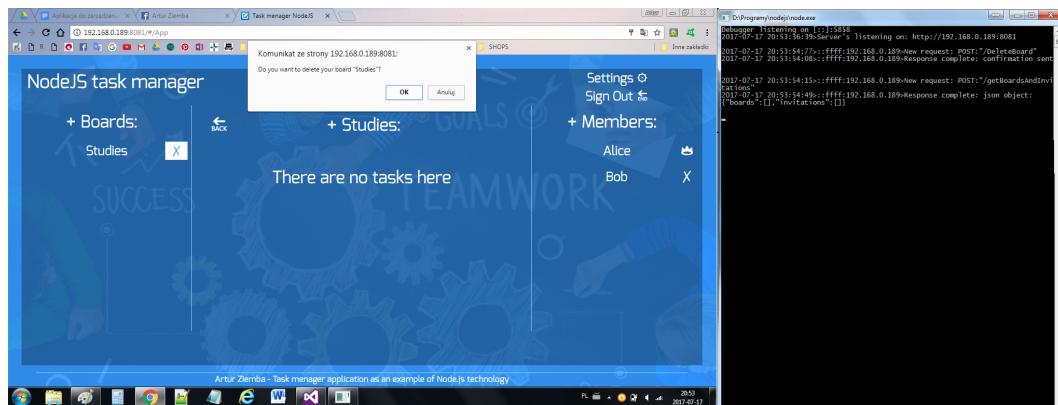
Po zalogowaniu, wybraniu odpowiedniej tablicy i zakończonego zadania użytkownik wybiera opcje usuwania zadania i przesyła żądanie do serwera. Serwer sprawdza poprawność danych. Odpowiada potwierdzeniem lub komunikatem o błędzie.

5.1.15 Opuszczanie tablicy



Po zalogowaniu użytkownik wybiera opcje opuszczenia tablicy, której jest członkiem i wysyła żądanie do serwera. Serwer sprawdza poprawność danych. Odpowiada potwierdzeniem lub komunikatem o błędzie.

5.1.16 Usuwanie tablicy



Po zalogowaniu użytkownik wybiera opcje usunięcia własnej tablicy i wysyła żądanie do serwera. Serwer sprawdza poprawność danych. Odpowiada potwierdzeniem lub komunikatem o błędzie.

5.2 Testy wydajnościowe

Przeprowadzone testy wydajnościowe pozwalają sprawdzić zachowanie serwisu przy różnym obciążeniu bazy danych. Zostały wykonane za pomocą technologii Node.js oraz modułu request, pozwalającego na wysyłanie zapytań. Każdy test został wykonyany 5 razy, a podane wartości przedstawiają średnią czasu pracy.

5.2.1 Mała ilość danych

Czas otrzymania odpowiedzi dla poszczególnych funkcjonalności od aplikacji z bazą danych zawierającą 5 użytkowników, każdy po 2 tablice po 3 zadania po 2 komentarze:

logowanie - 22ms
otrzymanie informacji o tablicach i zaproszeniach - 29ms
dodanie nowej tablicy - 20ms
dodanie nowego zadania - 27ms
dodanie komentarza - 26ms

5.2.2 Średnia ilość danych

Czas otrzymania odpowiedzi dla poszczególnych funkcjonalności od aplikacji z bazą danych zawierającą 50 użytkowników, każdy po 4 tablice po 6 zadań po 5 komentarzy:

logowanie - 26ms
otrzymanie informacji o tablicach i zaproszeniach - 33ms
dodanie nowej tablicy - 27ms
dodanie nowego zadania - 35ms
dodanie komentarza - 30ms

5.2.3 Duża ilość danych

Czas otrzymania odpowiedzi dla poszczególnych funkcjonalności od aplikacji z bazą danych zawierającą 500 użytkowników, każdy po 8 tablic po 12 zadań po 11 komentarzy:

logowanie - 32ms
otrzymanie informacji o tablicach i zaproszeniach - 39ms
dodanie nowej tablicy - 31ms
dodanie nowego zadania - 37ms
dodanie komentarza - 32ms

5.2.4 Podsumowanie testu

Serwis oferuje pełną funkcjonalność przy różnych obciążeniach bazy danych. Mimo dużej ilości posiadanych informacji korzystanie z serwisu jest komfortowe, ponieważ czas obsługi zapytania nie zostaje znaczowo wydłużony.

5.3 Testy obciążeniowe serwera

Przeprowadzone testy pozwalają określić zachowanie aplikacji w przypadku równoczesnej obsługi wielu przychodzących zapytań. Zostały wykonane za pomocą technologii Node.js oraz modułu request, pozwalającego na wysyłanie zapytań. Każdy test został wykonany 5 razy, a podane wartości przedstawiają średnią czasu pracy.

5.3.1 Mała ilość danych

Czas otrzymania odpowiedzi na 50 jednocześnie wysłanych zapytań - 45ms.

5.3.2 Średnia ilość danych

Czas otrzymania odpowiedzi na 300 jednocześnie wysłanych zapytań - 121ms.

5.3.3 Duża ilość danych

Czas otrzymania odpowiedzi na 1500 jednocześnie wysłanych zapytań - 674ms.

5.3.4 Podsumowanie testu

Serwis oferuje pełną funkcjonalność przy różnym stopniu obciążenia serwera. Przy dużej liczbie przychodzących zapytań użytkowanie serwisu jest wciąż komfortowe. Aplikacja wykazuje się skalowalnością.

5.4 Testy sprawnościowe serwera

Przeprowadzone testy pozwalają określić ilość kolejno wysłanych po sobie zapytań, które zostały obsłużone przez serwer w określonym czasie. Zostały wykonane za pomocą technologii Node.js oraz modułu request, pozwalającego na tworzenie zapytań. Każdy test został wykonany 5 razy, a podane wartości przedstawiają średnią czasu pracy.

5.4.1 Krótki okres czasu

Otrzymane odpowiedzi w czasie 1 sekundy - 139.

5.4.2 Średni okres czasu

Otrzymane odpowiedzi w czasie 30 sekund - 4723.

5.4.3 Duży okres czasu

Otrzymane odpowiedzi w czasie 5 minut - 48031.

5.4.4 Podsumowanie testu

Przedstawione wyniki prezentują możliwości obsługi kolejnych zapytań serwera Node.js. Osiągnięte wyniki są zadowalające, jednak łatwo zauważyc, że technologia ta wykazuje się największą sprawnością przy obsłudze równoległych zapytań. Średni uzyskany czas dla obsługi 1500 zapytań równoległych to 674 ms, natomiast dla wysyłanych kolejno wynosi 940 ms.

Rozdział 6

Podsumowanie otrzymanych wyników i wnioski na temat środowiska

Wykonana aplikacja spełnia zadaną w warunkach funkcjonalność. Zostały osiągnięte wszystkie założenia oraz wymagania aplikacji przy użyciu możliwości dostarczanej przez technologię Node.js. Stworzony projekt doskonale nadaje się do użytkowania przez zorganizowane zespoły do wykonywania określonych zadań. Mógliby być na przykład używany do zarządzania pracą zespołów deweloperskich pracujących w różnych metodykach takich jak scrum czy kanban. Dostarcza przejrzysty interfejs oraz nieskomplikowaną prezentację danych dla końcowych użytkowników, przez co nie wymaga praktycznie żadnych szkoleń lub procesów wdrożeniowych w celu zdobycia umiejętności biegłej obsługi narzędzia. Środowisko Node.js wydaje się być bardzo nowoczesnym oraz przyszłościowym środowiskiem. Niewątpliwie największymi zaletami tej technologii jest łatwość budowania wymagających serwisów internetowych poprzez użycie asynchronicznej obsługi wejścia/wyjścia, które pozwala na przetwarzanie wielu funkcji w jednym czasie, tworząc wyjątkowo szybkie w działaniu aplikacje. Zagadnienie to wymaga jednak umiejętności w projektowaniu i analizowaniu funkcji programowania asynchronicznego. Kod pisany jest w szeroko używanym języku JavaScript, co zbudowało ogromną społeczność zainteresowaną technologią Node.js. Dzięki temu możemy z łatwością zdobyć materiały przydatne w procesie poznanowczym środowiska. Globalne repozytorium npm gwarantuje obsługę wielu funkcjonalności, bez potrzeby długiego szukania możliwych rozwiązań. Wszystko jest dostępne bezpłatnie, gotowe do wdrożenia w tworzonej aplikacji. Mimo, że jest to technologia stosunkowo młoda, obecnie Node.js jest używany, a co za tym idzie, sprawdzony przez największe firmy IT w celu

obsługi ich aplikacji i serwerów. Należy być świadomym wszystkich zalet i wad przy wyborze określonej technologii. Node.js nie nadaje się do każdego typu projektu. Nie jest efektywnym środowiskiem w korzystaniu z obliczeń intensywnie wykorzystujących procesor. Jest on natomiast idealnym rozwiązaniem w kwestii pracy nad wieloma rozwiązaniami webowymi. Osobiście praca w Node.js sprawiła mi wielką radość i stała się moją ulubioną technologią dzięki zapewnionej prostocie i możliwych do uzyskania bez większego wysiłku ogromnych możliwościach.

Bibliografia

- [1] Ethan Brown *Web Development with Node and Express: Leveraging the JavaScript Stack 1st Edition* ISBN: 9781491949306
- [2] Robert Onodi *MEAN Blueprints* ISBN: 9781783553945
- [3] Paweł Kozłowski, Peter Bacon Darwin *Mastering Web Application Development with AngularJS* ISBN: 9781782161820
- [4] Dokumentacja języka programowania Node.js. Stan na dzień: 2017-07-20
<https://Node.js.org/en/docs>
- [5] Artykuł poświęcony technologii Node.js w serwisie Wikipedia. Stan na dzień: 2017-07-20 <https://en.wikipedia.org/wiki/Node.js>
- [6] Dokumentacja biblioteki libuv. Stan na dzień: 2017-07-20
<https://github.com/libuv/libuv>
- [7] Dokumentacja MongoDB. Stan na dzień: 2017-07-20
<https://docs.mongodb.com/>
- [8] Artykuł poświęcony technologii Vert.x w serwisie Wikipedia. Stan na dzień: 2017-07-20 <https://en.wikipedia.org/wiki/Vert.x>
- [9] Artykuł poświęcony technologii Twisted w serwisie Wikipedia. Stan na dzień: 2017-07-20 [https://en.wikipedia.org/wiki/Twisted_\(software\)](https://en.wikipedia.org/wiki/Twisted_(software))
- [10] Dokumentacja Ringojs. Stan na dzień: 2017-07-20 <https://ringojs.org/>
- [11] Porównanie technologii Node.js i Ringojs. Stan na dzień: 2017-07-20
<http://hns.github.io/2010/09/21/benchmark.html>

Spis rysunków

2.1	Pętla zdarzeń w środowisku Node.js - źródło: https://stackoverflow.com/questions/21607692/understanding-the-event-loop	8
2.2	Model programowania współbieżnego wykorzystywany przez Node.js - źródło: www.tutorialspoint.com/parallel_algorithm/	10
2.3	Kody statusu protokołu http - źródło: http://mokandra.blogspot.fi/2015/10/http-status-code-definitions.html	12
4.1	Przepływ komunikacji w MEAN Stack źródło: https://www.dealfuel.com/seller/mean-stack-tutorial/	20
4.2	Pobranie statycznych zasobów, źródło: Opracowanie własne	20
4.3	Wymiana zasobów w formacie Json źródło: Opracowanie własne	21