AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE
Wydział Fizyki i Informatyki Stosowanej

Praca inżynierska

Damian Łączak

kierunek studiów: **Informatyka Stosowana**

Aplikacja internetowa do nauki języka

Opiekun: **dr hab. inż Tomasz Bołd**

Kraków, styczeń, 2018

Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie i nie korzystałem ze źródeł innych niż wymienione w pracy.

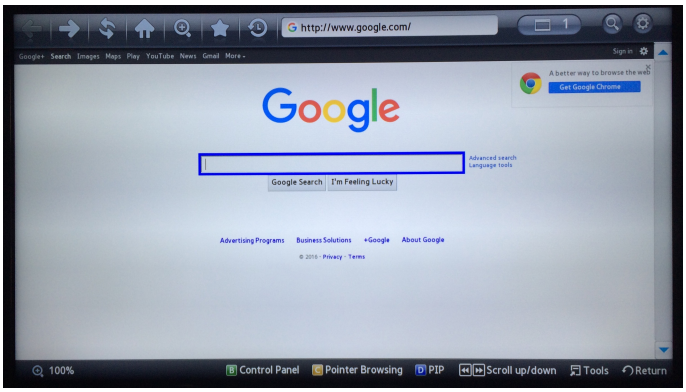
Spis treści

Todo list

1. Wstęp

Tematem pracy inżynierskiej jest opracowanie aplikacji internetowej do nauki języków obcych.

Coraz większą popularność w dzisiejszym świecie zdobywają aplikację internetowe. Ich niewątpliwą zaletą jest możliwość uruchomienia ich na dowolnym urządzeniu posiadającym internet i przeglądarkę internetową. Jeszcze do nie dawna były to jedynie komputery i urządzenia mobilne, lecz wraz ze wzrostem popularności inteligentnych technologii (po ang. smart) coraz więcej produktów codziennego użytku potrafi łączyć się z internetem. Do takich urządzeń zaliczają się mikrofalówki, lodówki, telewizory i wiele innych.



Rys. 1.1. Przeglądarka internetowa wbudowana w telewizor.

TODO WSTEP

2. Założenia projektu

2.0.1. Założenia aplikacji

Zadaniem aplikacji jest umożliwienie nauki wybranych słówek danego języka obcego z pomocą wirtualnych fiszek, które będą prezentowane użytkownikowi w nieregularnych odstępach czasowych obliczanych na podstawie poprawności udzielonych odpowiedzi. Program powinien umożliwiać:

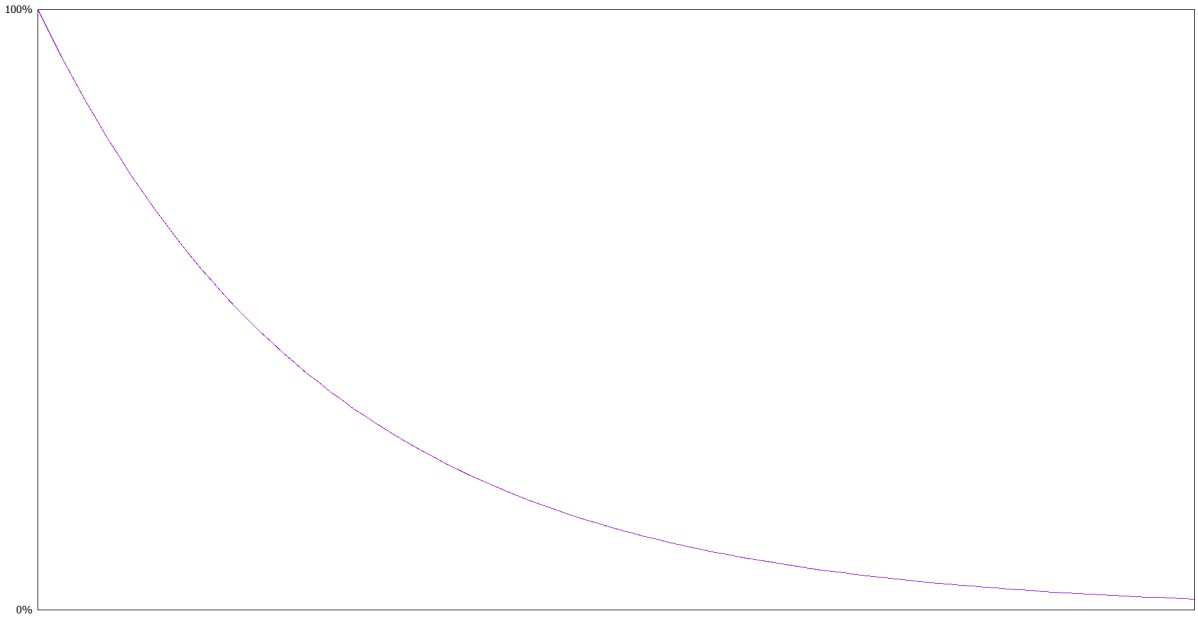
– Dodawanie, modyfikowanie i usuwanie fiszek.

–

Program został stworzony w języku C# z użyciem frameworka ASP.NET MVC. Wszystkie dane są przechowywane na serwerze Microsoft SQL. Po stronie klienta użyto języka HTML wraz z arkuszami styli stworzonymi w oparciu o język SASS. Wszystkie skrypty napisano w języku Scala, który jest potem kompilowany przez narzędzie Scala.js do

3. Przyswajanie wiedzy

Hermann Ebbinghaus[1] (1850-1909) był Niemieckim psychologiem, który jako jeden z pierwszych zajmował się tematyką eksperymentalnej psychologii związanej z przyswajaniem wiedzy. Jednym z jego pierwszych eksperymentów było stworzenie testu, podczas którego uczył się zestawu 20 sylab, które były bezsensowne ze względu na fakt, iż w jego języku nie występowały żadne słowa, które ich używały. Eksperyment ten pozwolił mu skonstruować pierwszą na świecie krzywą zapominania. Miała ona charakter eksponencjalny. Możemy z niej wywnioskować, iż podczas początkowego okresu zapominania tracimy najwięcej zapamiętanych informacji.



Rys. 3.1. Eksponencjalna krzywa zapominania.

Trzeba także zwrócić uwagę na fakt, iż w rzeczywistości[2] mózg nie jest w stanie przyswoić danych informacji w 100% po zakończeniu nauki.

3.1. Interwały potwórzeń

W celu jak najlepszego zapamiętania wyuczonych informacji należy, poza odpowiednim programem nauczania, zwrócić uwagę na fakt, w jakich interwałach czasowych powtarzany jest

przerabiany materiał[2]. Interwał musi być wystarczająco krótki, aby zapobiec zapominaniu i wystarczająco długi, aby zbyt często nie przyswajać powtarzanego materiału.

Można tu jeszcze coś napisać

4. .NET

.NET jest platformą programistyczną umożliwiającą pisanie nowoczesnych aplikacji w językach wysokiego poziomu, do których zalicza się m.in C#, VB oraz F#. Platforma ta wyróżnia się tym iż:

- Pozwala na użycie wielu języków programowania podczas pisania naszych programów.
- Ma zaimplementowane mechanizmy do obsługi operacji asynchronicznych i współbieżnych.
- Można ją stosować na różnych platformach, które posiadają środowisko wykonywalne .NET.

Wszystkie języki używane w platformie .NET kompilowane są do Wspólnego Języka Pośredniego (po ang. *Common Intermediate Language*), który następnie jest tłumaczony na kod bajtowy i wykonywany za pomocą środowiska wykonywalnego danej implementacji .NET.

```
.assembly HelloWorld
.class auto ansi HelloWorldApp
{
     .method public hidebysig static void Main() cil managed
     {
          .entrypoint
          .maxstack 1
          ldstr "Hello world."
          call void [mscorlib]System.Console::WriteLine(string)
          ret
     }
}
```

Listing 4.1. Przykładowy kod aplikacji "Hello World"w języku CIL

4.1. Implementacje .NET

Każda aplikacja .NET jest uruchamiana na jednej z implementacji .NET.
Od roku 2016 wprowadzono .NET Standard - wspólny zestaw API, które każda z implementacji musi posiadać. Pozwala to na pisanie i używanie bibliotek programistycznych w różnych środowiskach .NET.

Istnieją aktualnie 4 główne implementacje .NET:

4.1.1. .NET Core

Został napisany z myślą o tworzeniu aplikacji cross-platformowych, które mogą zostać uruchomione na serwerach, jak i środowiskach chmurowych. Potrafi działać na platformie Windows, macOS oraz Linux. Jest to pierwsza implementacja .NET, która została zaprojektowana przez Microsoft z myślą o wieloplatformowości.

4.1.2. .NET Framework

Pierwsza, oryginalna implementacja .NET, która istnieje od roku 2002. Składa się ze środowiska uruchomieniowego Common Language Runtime (CLR) oraz biblioteki standardowej zwanej jako Framework Class Library (FCL). CLR zapewnia aplikacjom wirtualną maszynę, na której wykonywany jest kod bajtowy skompilowany z języka CIL. Ta implementacja jest używana w tej pracy inżynierskiej.

4.1.3. Mono

Darmowy projekt open-source prowadzony przez firmę Xamarin. Powodem stworzenia tego produktu była możliwość uruchamiania aplikacji napisanych w językach .NET na wielu platformach, jak i dostarczenie użytkownikom Linuxa narzędzi pozwalających na pisanie aplikacji w rodzinie języków .NET.

4.1.4. Universal Windows Platform (*UWP*)

Implementacja, która umożliwia tworzenie aplikacji dla wszystkich platform używających Windows 10, Xboxa , niektórych urządzeń stworzonych przez Microsoft i dostosowanych urządzeń IoT.

4.2. C#

C# jest językiem programowania trzeciej generacji. Został opublikowany w roku 2001 przez firmę Microsoft. Jest on silnie typowanym językiem wielo-paradygmatowym umożliwiającym programowanie imperatywne, deklaratywne, generyczne, funkcjonalne, komponentowe i zoriento-

wane obiektowo. Najnowszy jego standard został wydany w listopadzie 2017 roku i jest oznaczony jako wersja 7.2. Wchodzi on w grupę języków .NET przez co jest kompilowalny do języka CIL. [3]

```
using System;
namespace HelloWorld
{
    class Hello
    {
        static void Main()
        {
            Console.WriteLine("Hello World!");
            Console.ReadKey();
        }
    }
}
```

Listing 4.2. Przykładowy kod aplikacji "Hello World"w języku CIL

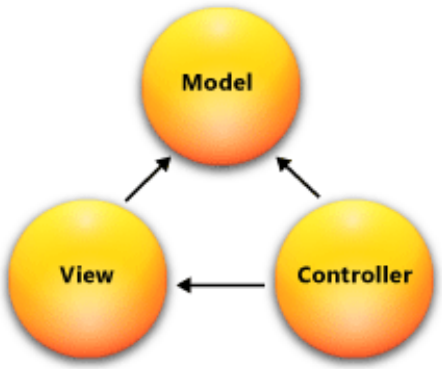
5. ASP.NET MVC

ASP.NET MVC jest frameworkiem do budowania aplikacji internetowych w oparciu o wzorzec architektoniczny Model-View-Controller (MVC). Wykorzystuje implementacje .NET Framework do uruchamiania skompilowanego kodu źródłowego.

5.1. Model-Widok-Kontroler

Większość dzisiejszych systemów komputerowych działa na zasadzie wyświetlania danych, które aktualnie znajdują się w bazie danych i ewentualnie ich modyfikacji. W celu ujednolicenia tych systemów stosowany jest wzorzec Model-Widok-Kontroler(ang. Model-View-Controller), który rozdziela logikę aplikacji na 3 główne segmenty:

1. Model - Służy do pobierania, przechowywania i zamiany danych.
2. Kontroler - przetwarza zapytania użytkownika.
3. Widok - Służy do wyświetlania informacji .



Rys. 5.1. Model MVC. Źródło: msdn.microsoft.com

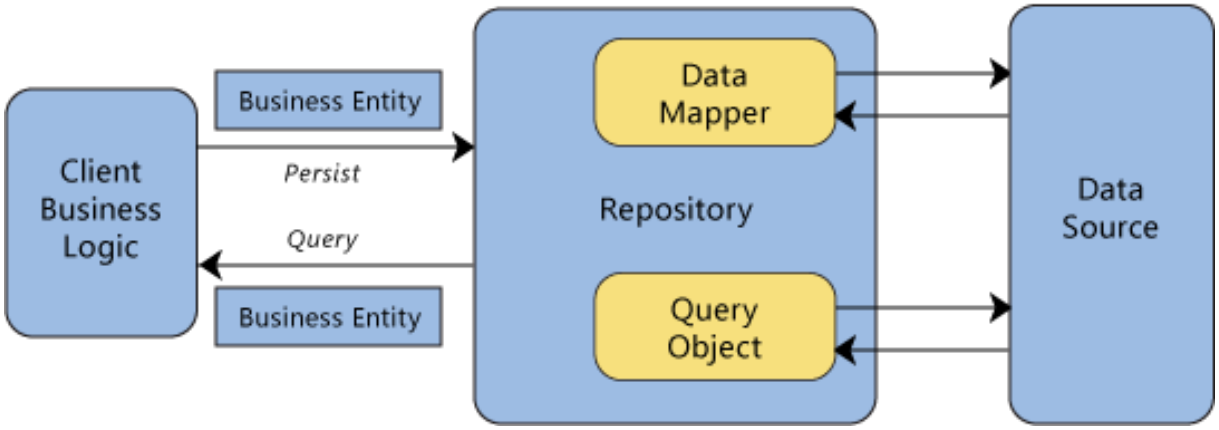
Cała aplikacja została skonstruowana zgodnie z tym wzorcem projektowym. ASP.NET MVC posiada wiele narzędzi, które ułatwiają zastosowanie tego wzorca. Dostarczana jest klasa bazowa **Controller**, która posiada wszystkie podstawowe metody do wyświetlania odpowiednich treści danych i zarządzania zapytaniami.

Wzorzec MVC niesie za sobą korzyści związane z warstwą wyświetlania danych. Jest to warstwa aplikacji, w której bardzo często dochodzi do zmian. Dzięki odseparowaniu danych od widoku jesteśmy w stanie tworzyć, jak i zmieniać widoki, bez wpływu na kod biznesowy aplikacji.

5.2. Wzorzec Repozytorium

Wewnątrz aplikacji używana będzie baza danych Microsoft SQL. Do komunikacji z bazą bardzo często jest tworzony kod, który zwraca podobne dane. W celu zmniejszenia redundancji kodu, jak i odseparowania zależności i odpowiedzialności wykorzystany został wzorzec Repozytorium (z ang. Repository Pattern)[4].

Wzorzec ten wykorzystuje obiekty, zwane repozytoriami, których zadaniem jest pobieranie i modyfikowanie danych po stronie serwera SQL. Nie zawierają żadnej logiki biznesowej i są niezwiązane z resztą kodu danej aplikacji.

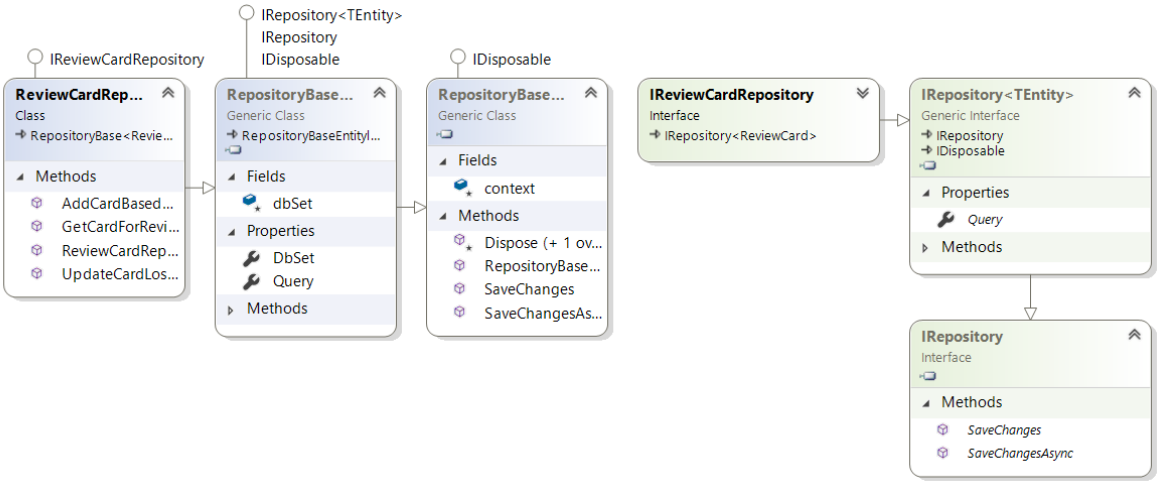


Rys. 5.2. Wzorzec repozytorium. Źródło: msdn.microsoft.com

Wewnątrz projektu cały wzorzec repozytorium został oparty na interfejsach **IRepository** i **IRepository<T>** (który implementuje **IRepository**). W założeniu te interfejsy definiują operacje, które można zrealizować na danym obiekcie klasy 'T'. Informacja o sposobie realizacji operacji jest zdefiniowana w klasie **RepositoryBase<T>**. Wszystkie kolejne stworzone repozytoria dziedziczą po **RepositoryBase<T>** a ich interfejsy implementują **IRepository<T>**. Takie działanie pozwala nam na separację reprezentacji persystentnej (baza danych) i reprezentacji bieżącej (repozytoria). Wystarczy, iż zmienimy **RepositoryBase<T>** na jakąkolwiek inną klasę implementującą **IRepository<T>**. Dzięki temu możemy zastąpić Microsoft SQL inną implementacją.

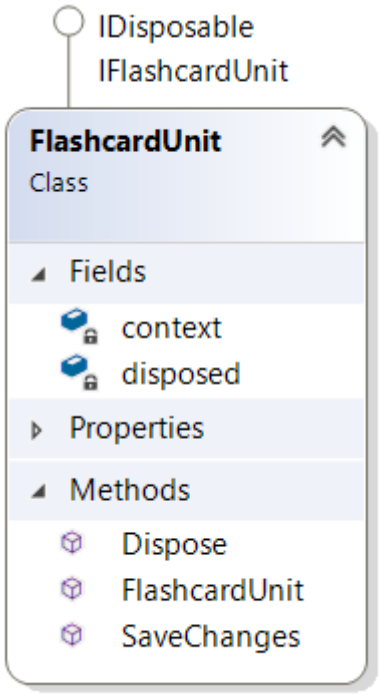
5.3. Jednostka Pracy

Wzorzec Jednostki Pracy (z ang. Unit of Work)[4] ma na celu uporządkowanie pracy z repozytoriami za pomocą umieszczenia ich wszystkich w jednej klasie. Dodatkowo dzięki temu



Rys. 5.3. Repozytorium ReviewCardRepository wraz z implementowanymi interfejsami i odziedziczonymi klasami.

rozwiązaniu wszystkie repozytoria współdzielą kontekst dostępu do bazy Microsoft SQL. Zapisanie danych odbywa się poprzez wywołanie metody **SaveChanges** wewnątrz Jednostki Pracy (SaveChanges jest konwencją wewnątrz projektu).



Rys. 5.4. Jednostka pracy wykorzystywana w projekcie.

5.4. Mapowanie obiektowo-relacyjne

Wewnątrz projektu zostało wykorzystane narzędzie Entity Framework pozwalające na korzystanie z mapowania obiektowo-relacyjnego. Dzięki tej technice można uprzednio zdefiniowane dane tabelaryczne odwzorować za pomocą klas, które zostaną stworzone na podstawie definicji tabel i relacji między nimi. Powstałe klasy są używane wewnątrz kolekcji, implementujących interfejs **IQueryable<T>**, na których możemy wykonywać zapytania z pomocą **LINQ**a, które następnie zostaną przetłumaczone na zapytania SQL.

Takowe rozwiązanie ma wiele zalet :

1. Dane z danej tabeli można otrzymać zawsze w tym samym formacie.
2. Zmniejsza to zakres potrzebnych umiejętności w celu pobrania danych. Dany programista nie musi znać języka SQL, aby w sposób bezproblemowy pobrać interesujące go dane, nawet gdy tworzy bardzo skomplikowane zapytania.
3. Łatwiejsza nawigacja po zależnościach między tabelami. Na danym typie można wykorzystać operacje **Find All References**, która wskaże miejsca jego użycia w kodzie.
4. Operacje na interfejsie **IQueryable<T>** pozwalają na pobranie danych dopiero, gdy będą nam potrzebne za pomocą techniki lazy loading.

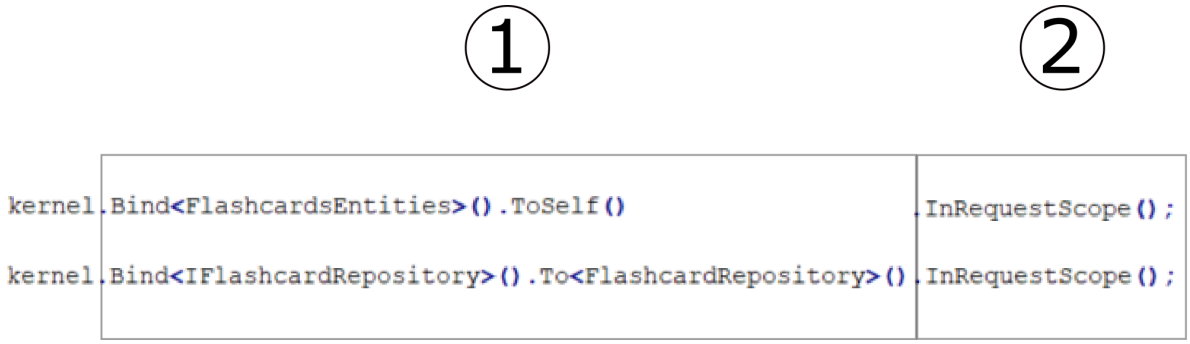
5.5. Wstrzykiwanie zależności

Wstrzykiwanie zależności (po ang. Dependency Injection) jest wzorcem projektowym, którego działanie polega na tym, iż obiekty aplikacji nie muszą tworzyć zależnych od siebie obiektów, lecz są tworzone przez obiekt nadrzędny dla użytku tych klas.

Informacja o tym, jakie obiekty powinny zostać dostarczone danej instancji jest przekazywana najczęściej poprzez listę argumentów konstruktora bądź przy wywołaniu specjalnych metod.

Wewnątrz projektu w celu realizacji tego wzorca wykorzystana została biblioteka Ninject[5]. Z pomocą dodatkowej biblioteki **Ninject.Web.Common** jest ona w stanie w łatwy sposób zintegrować się z aplikacją ASP.NET MVC wykorzystując interfejs **IDependencyResolver**.

Każde utworzenie kontrolera wewnątrz aplikacji ASP.NET MVC wiąże się wpierw z odpytaniem aktualnie używanego IDependencyResolvera o próbę utworzenia takowego kontrolera. W tym miejscu zaczyna działać Ninject, który stara się utworzyć obiekt. Gdy konstruktor danej klasy będzie zawierał jakiekolwiek parametry to Ninject będzie starał się je utworzyć na podstawie zdefiniowanych uprzednio bindingów. Jeśli okaże się, że stworzenie obiektu kontrolera jest niemożliwe, to zwracany jest błąd wewnętrzny 500.
Stworzenie bindingów, które będą informować, na jakich zasadach odbywa się tworzenie żądanych obiektów, jest wykonywane za pomocą metody **Bind** znajdującej się w Kernelu Ninjecta.



Rys. 5.5. Proces bindowania z wyszczególnionymi częściami składowymi.

Pierwsza część bidowania odpowiada za przekazanie informacji o tym jakiego **Providera** należy użyć w celu zwrócenia instancji danego typu. Metody **ToSelf()** i **To<T>()** używają wbudowanego **StandardProvidera**. Pierwsza z metod zawsze będzie informowała o tym, iż należy zwrócić obiekt tego samego typu, gdzie druga konkretnie informuje jaki typ powinien zostać zwrócony w wyniku danego zapytania.

Ninject oferuje także opcje stworzenia własnych providerów. Można to osiągnąć za pomocą stworzenia klasy dziedziczącej po **Provider<T>** lub wykorzystania delegatów wewnątrz metody **ToMethod(Func<IConext, T>)**.

Ninject w swoim domyślnym zachowaniu zawsze tworzy nowy obiekt danej klasy, gdy zostanie poproszony o instancje danego typu. Nie zawsze jest to jednak sytuacja pożądana. Każdy bind

posiada możliwość zdefiniowana osobnego zasięgu w którym będzie on obowiązywał. Umożliwi to współkorzystanie z danego obiektu wewnątrz danego zakresu. W wypadku środowiska webowego bardzo często stosowanym zasięgiem jest **InRequestScope()**, które powoduje, iż obiekty danego typu stworzone przez Ninject są współdzielone w trakcie danego zapytania.

Takie działanie niesie za sobą szereg korzyści. Nie tylko nie ponosimy kosztów tworzenia nowych obiektów, lecz także jesteśmy w stanie współdzielić dane informacje pomiędzy różnymi serwisami. (Np. repozytoria nie muszą wykonywać dodatkowych zapytań SQL, gdyż część wyników znajduje się ich w cache'u)

6. Microsoft SQL Server

Microsoft SQL Server[6] jest aplikacją, umożliwiającą zarządzanie serwerami baz danych. Został stworzony przez firmę Microsoft. Jego pierwsza edycja ukazała się w roku 1989.

Do tworzenia zapytań wykorzystano tutaj język Transact-SQL (T-SQL). Jest to rozszerzenie języka SQL (Structured Query Language), które między innymi wprowadza:

1. Lokalne zmienne.
2. Blok **Try Catch**, wspierający obsługę wyjątków.
3. Dodatkowe funkcjonalności związane z obsługą tekstu, kalendarza i matematyki.
4. Możliwość tworzenia pętli oraz instrukcji warunkowych.
5. Procedury, funkcje składowane i wyzwalacze.

W celu integracji serwera Microsoft SQL z projektem napisanym w języku C# należy pobrać paczkę NuGet[1] o nazwie **System.Data.SqlClient**.

[1]NuGet jest menadżerem paczek w środowisku .NET.[7]

7. Testy

Aplikacje tworzone w dzisiejszych czasach wymagają dostarczenia zestawu testów, których zadaniem jest sprawdzenie poprawności działania aplikacji. Z tego też powodu napisany został zestaw testów, które sprawdzają, czy poszczególne komponenty aplikacji działają poprawnie.

7.1. Testy Jednostkowe

Testy jednostkowe mają za zadanie sprawdzić poprawność działania pojedynczego modułu. Wszelkie odwołania do innych komponentów zostają zastąpione przez atrapy obiektów (po ang. mock objects), które symulują działanie ich prawdziwych odpowiedników. Wewnątrz projektu do operacji związanych z atrapami wykorzystany został framework Moq, który zawiera także dodatkowe funkcjonalności pozwalające ocenić czy dany test przebiegł poprawnie. Testy zostały ułożone zgodnie ze wzorcem AAA[8] - Aranżacja (po ang. Arrange) Akcja (po ang. Act) Asercja (po ang. assert). Dzięki temu testy są konsystentne i czytelne, przez co nie potrzeba dużej ilości czasu w celu zaznajomienią się z nimi.

```
public void StopLastTraining_assert_tests()
{
        //Aranżacja - stworzenie wewnętrznej atrapy obiektu dla atrapy serwisu
        mockTraining();
        //Akcja
        trainingReviewService.StopLastTraining();
        //Asercja - sprawdzenie poprawności wykonania
        trainingRepository.Verify(x => x.Remove(It.IsAny<long>()), Times.Once);
        unit.Verify(x => x.SaveChanges(), Times.Once); //wykorzystanie ↩
            funkcjonalności frameworku Moq w celu sprawdzenia czy dana metoda ↩
            została wywołana.
        Assert.AreEqual(null, sessionService.Object.UserInfo.TrainingInfo);
}
```

Listing 7.1. Przykładowy test jednostkowy wykorzystujący wzorzec AAA

7.2. Testy Integracyjne

Testy integracyjne mają za zadanie pokazać poprawną komunikacje pomiędzy modułami w projekcie. Modułem może być każdy komponent zaprogramowany na potrzeby projektu jak i zewnętrzny system (np. system obsługi plików). Wszystkie moduły, które nie są testowane w danym teście powinny zostać zastąpione przez odpowiednie atrapy.

```
[TestMethod]
public void ExistTest()
{
        using (var temp = new WindowsTempFile())
                Assert.IsTrue(File.Exists(temp.Path));
}}
```

Listing 7.2. Test integracyjny wykorzystany w projekcie.

8. Kaskadowe Arkusze Styli

Kaskadowe Arkusze Styli (po ang. Cascading Style Sheets, w skrócie CSS)[**CSSDoc**] wykorzystywane są w celu opisu sposobu wyświetlania elementów stron WWW. Pozwalają zdefiniować między innymi takie właściwości elementów jak: kolory, czcionki, położenie. Dzięki oddzieleniu warstwy danych od warstwy prezentacji możliwe jest wykorzystanie tych samych arkuszów styli na wielu stronach. W celu określenia wyglądu danego elementu należy zdefiniować odpowiednią regułę, na którą składa się :

1. Selektor, który określa dla jakich elementów przeznaczona jest dana reguła.

2. Zestaw właściwości wraz z przypisanymi do nich wartościami

```
p //seletor p
{
//Właściwość color do której przypisano wartość orange
    color: orange;
}
```

Listing 8.1. Przykład prostej reguły, dzięki której wszystkie paragrafy będą miały domyślnie kolor pomarańczowy.

8.1. Syntaktycznie Zarąbiste Arkusze Styli

W dzisiejszym świecie nikt obeznany z technologią tworzenia stron internetowych nie używa już czystego języka CSS do tworzenia arkuszy styli. Wynika to z faktu występowania zbyt dużej redundancji kodu, która jest związana z tym, iż niektóre fragmenty strony są formatowane zawsze w ten sam sposób (np. motyw kolorystyczny dla większości elementów jest ten sam).

Jest to spowodowane brakiem zmiennych, szablonów, oraz innych konstrukcji, które pomogłoby w ograniczeniu powyższego problemu.

Z tego też powodu powstały rozwiązania niedogodności występujących w języku CSS. Jednym z nich jest język SASS - Syntactically Awesome Style Sheets (z ang. Syntaktycznie Zarąbiste Arkusze Styli)[9]. Jest on kompatybilny ze wszystkimi wersjami CSS, przez co każdy kod napisany w CSS jest z nim zgodny. Oferuje bardzo dużo funkcjonalności, między innymi:

1. Zmienne, które pozwalają wielokrotnie używać danych wartości
2. Możliwość zagnieżdżania reguł w innych regułach.
3. Możliwość załączania plików pozwalająca na podział arkuszy styli.
4. Mixin - szablony reguł, które można ponownie wykorzystywać.
5. Dziedziczenie reguł za pomocą słówka kluczowego **@extend**
6. Za pomocą operatorów +, -, *, /, % można wykonywać operacjach na liczbach.

```
@mixin border-radius($radius) {
  -webkit-border-radius: $radius;
     -moz-border-radius: $radius;
      -ms-border-radius: $radius;
          border-radius: $radius;
}

.box { @include border-radius(10px); }
```

Listing 8.2. Przykładowy kod SCSS wykorzystujący mixin.

9. Scala

Scala[10] jest językiem programowania ogólnego zastosowania, który został wprowadzony na rynek przez Laboratorium "École Polytechnique Fédérale de Lausanne". Jest to język kompilowalny bezpośrednio do kodu bajtowego Javy, przez co programy w nim napisane z łatwością uruchamiają się w środowisku wykonywalnym maszyny wirtualnej Javy. Scala jest językiem wielo-paradygmatowym[11]. Korzysta z dobrodziejstw programowania funkcjonalnego i obiektowego.

```
object HelloWorld {
  def main(args: Array[String]): Unit = {
    println("Hello, world!")
  }
}
```

Listing 9.1. Hello world napisany w języku Scala.

9.1. Javascript

Javascript jest głównym językiem programowania wykorzystywanym na stronach internetowych. Może być on interpretowany bądź kompilowany metodą Just In Time, która kompiluje kod tuż przed jego wykonaniem. Jest to dynamiczny język wieloparadygmatowy bazujący na prototypach. Wspiera programowanie obiektowe, imperatywne i deklaratywne.[**AboutJS**] Javascript jako język posiada wiele problemów, które mogą doprowadzić do wystąpienia błędów, między innymi takich jak:

1. Niejawne rzutowania, które objawiają się przy użyciu operatora ==. Są przyczyną wielu trudnych do odkrycia błędów

2. Automatyczne wstawianie średników, co może spowodować, iż program działa inaczej niż zamierzał to programista. Czasami potrafi to uratować program napisany przez programistę, lecz często prowadzi do dziwnych i niezrozumiałych błędów, takich jak w przypadku poniższego listingu: 9.2

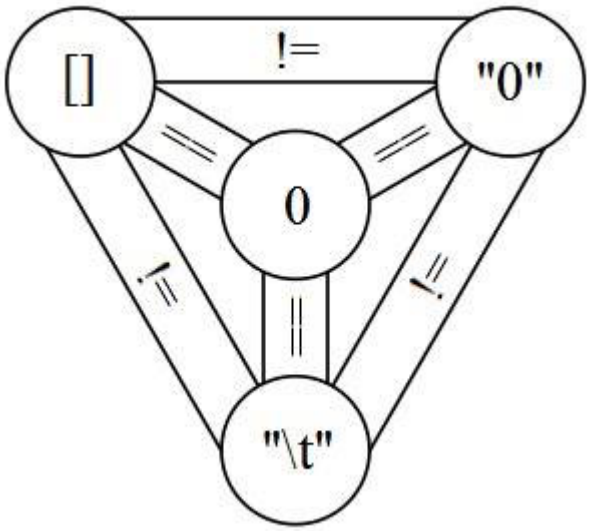
3. Niezrozumiałe zachowana różnych funkcji. Metoda sortująca domyślnie sortuje tablice liczb w porządku alfabetycznym nie zwracając uwagi na fakt, iż w tablicy nie występuje ani jeden ciąg znakowy.

4. Zmienne, które w każdej chwili mogą zmienić swój typ. Pomimo tego, iż jest to błąd programistyczny to w wyniku tej niedogodności powstają trudne do namierzenia błędy w kodzie.

```
function foo() {
    return // Tutaj zostanie wstawiony średnik.
        {
            bar : "test"
        };
}
```

Listing 9.2. Przykład niepoprawnego kodu Javascript wynikłego z automatycznego wstawienia średnika

Z tego też powodu powstały języki, które starają się być lepsze i przyjaźniejsze dla programisty niż Javascript. Jest to między innymi: Typescript, Coffescript, Dart czy Scala.js. Takie języki programowania nie są wykonywane/kompilowane bezpośrednio w przeglądarce, lecz wpierw są kompilowane do kodu Javascript. Spowodowane jest to używaniem przez przeglądarki jedynie języka Javascript do wykonywania kodu[1].



Rys. 9.1. Przykład dziwnego zachowania języka Javascript. Źródło: devrant.com

9.2. Scala w wersji JS

Scala.js jest narzędziem umożliwiającym kompilacje kodu Scali do języka Javascript. Umożliwia to pozbycie się niedogodności wymienionych powyżej. Jest to możliwe dzięki:

1. Silnemu typowaniu, dzięki któremu pisany kod nie zawiera bardzo prostych błędów.

[1]Wyjątkiem jest tutaj przeglądarka Dartium, która posiada maszynę wirtualną języka Dart[12]

2. Nie skupianiu się na dziwnych i frustrujących aspektach języka Javascript.

3. Środowiskom programistycznym wspierającym pracę programisty przez podpowiadanie składni, łatwą nawigację po kodzie źródłowym oraz inne udogodnienia.

4. Możliwości wykrycia błędów w trakcie kompilowania kodu źródłowego.

9.2.1. sbt

sbt (Simple Build Tool) jest narzędziem o otwartym kodzie źródłowym, pozwalającym na zarządzanie procesem budowania aplikacji napisanej w języku Scala lub Java. Jest to narzędzie bardzo rozbudowane, które m.in. umożliwia:

1. Współpracę z wieloma narzędziami testującymi dla scali.

2. Inkrementacyjne kompilowanie i testowanie.

3. Zarządzanie zależnościami przy pomocy menadżera paczek Ivy.[13]

4. Ciągłe wdrażanie, kompilowanie i testowanie kodu.

9.2.2. Proces budowania projektu

Proces kompilacji projektu jest podzielony na 3 etapy[14]:

1. Początkową kompilację

2. Szybką optymalizację

3. Pełną optymalizację (Opcjonalny)

9.2.2.1. Początkowa kompilacja

W trakcie kompilacji pliki .scala są kompilowane do plików .class i .sjsir. Pliki .class nie biorą udziału w tworzeniu kodu javascript. Ich zadaniem jest współpraca z innymi narzędziami, które być może będą ich używać. Przykładem takiego narzędzia może być **IntelliJ** lub **Eclipse**, które plików .class używają w celu wspomagania programisty w trakcie pisania kodu. Pliki .sjsir (Nazwa rozszerzenia jest skrótem od „ScalaJS Intermediate Representation")[15] zawierają kod przejściowy między Scalą a Javascriptem. Większość konstrukcji została zastąpiona przez ekwiwalenty z języka Javascript. Gdybyśmy połączyli wszystkie pliki .sjsir, wyprodukowane przez sbt to ich wynikiem byłby plik większy niż 20 MB. Wynika to z faktu, iż w tym pliku nadal znajduje się wiele niepotrzebnych bibliotek i konstrukcji. Jak na przykład **cała** biblioteka standardowa Scali.

```
module class Ltutorial_webapp_TutorialApp$ extends O {
  def main__AT__V(args: T[]) {
    this.appendPar__Lorg_scalajs_dom_raw_Node__T__V
    (mod:Lorg_scalajs_dom_package$.document__Lorg_scalajs_dom_raw_HTMLDocument
    ()["body"], "Hello World")
  }
  def appendPar__Lorg_scalajs_dom_raw_Node__T__V(targetNode: any, text: T) {
    val parNode: any = ↩
        mod:Lorg_scalajs_dom_package$.document__Lorg_scalajs_dom_raw_HTMLDocument()
    ["createElement"]("p");
    val textNode: any = ↩
        mod:Lorg_scalajs_dom_package$.document__Lorg_scalajs_dom_raw_HTMLDocument()
    ["createTextNode"](text);
    parNode["appendChild"](textNode);
    targetNode["appendChild"](parNode)
  }
  def init___() {
    this.O::init___();
    mod:Ltutorial_webapp_TutorialApp$<-this
  }
}
```

Listing 9.3. Przykładowy plik .sjsir dla projektu wyświetlającego HelloWorld na ekranie.

9.2.2.2. Szybka optymalizacja

W celu optymalizacji poprzedniego kroku stosuje się szybką optymalizację kodu .sjsir, która jako rezultat swej pracy stworzy kod Javascript. W tym celu należy użyć optymalizatora FastOptJS poprzez wpisanie komendy **FastOptJS** w sbt. Optymalizacja ma na celu:

1. Wyeliminowanie fragmentów kodu, które są nieużywane. Na przykład kod biblioteki standardowej, który nie zostanie wywołany.
2. Inline'owanie małych funkcji. Zmniejsza to koszt wywołań i wielkość kodu.
3. Zmiana zmiennych na stałe, jeśli ich wartość jest znana w trakcie kompilacji.

Dzięki tej operacji kod wykonywalny zmniejszy się z 20 MB do 1.5-2.5MB[15].

9.2.2.3. Pełna optymalizacja

Kod, który jest tworzony przez Scala.js jest zgodny z restrykcjami narzuconymi przez kompilator Closure[16]. Jest to narzędzie stworzone przez firmę Google, które potrafi optymalizować kod Javascript.[17] Celem tego kroku jest zmniejszenie rozmiaru pliku Javascript i uczynienie konstrukcji w nim występujących bardziej wydajnymi. W wyniku tego procesu otrzymywany jest plik wykonywalny o rozmiarze między 150KB do kilkuset KB[15]. W celu użycia pełnej optymalizacji należy wywołać polecenie **FullOptJS** w sbt.

9.2.2.4. Pliki javascript

W wyniku działań optymalizatorów tworzone są poniższe pliki javascript. Należy mieć na uwadze fakt, iż pliki bibliotek muszą zostać dołączone do kodu strony przed plikiem z kodem wykonywalnym.

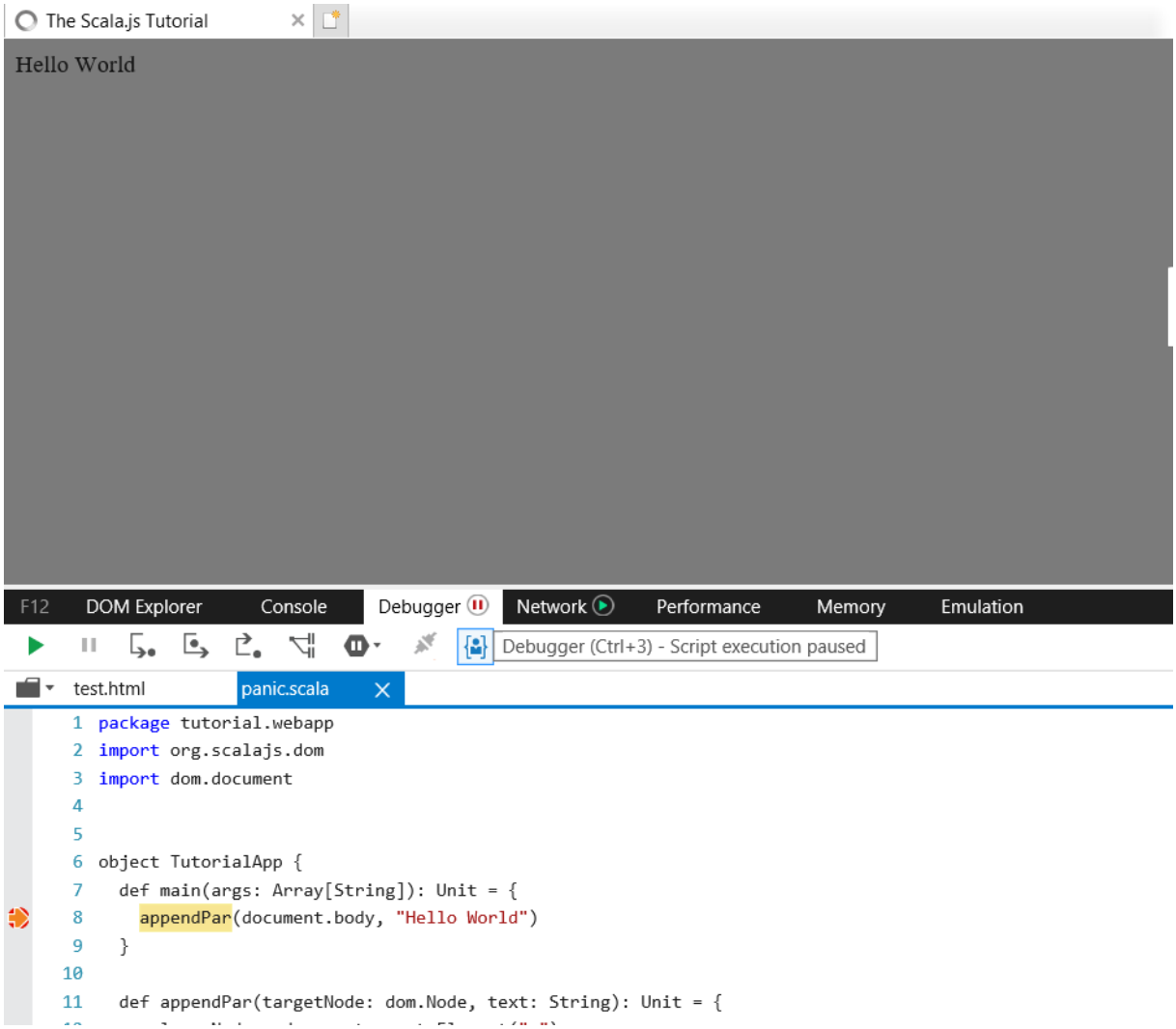
Typ kompilacji	Wytworzony plik	Zawartość pliku
FastOptJS	scala-js-[Nazwa-Projektu]-fastopt.js	Plik z kodem wykonywalnym
FastOptJS	scala-js-[Nazwa-Projektu]-fastopt.js.map	Plik mapujący kod Javascript do kodu Scali.
FastOptJS	scala-js-[Nazwa-Projektu]-jsdeps.js	Plik z kodem zewnętrznych bibliotek użytych w procesie tworzenia aplikacji
FullOptJS	scala-js-[Nazwa-Projektu]-opt.js	Plik z kodem wykonywalnym
FullOptJS	scala-js-[Nazwa-Projektu]-opt.js.map	Plik mapujący kod Javascript do kodu Scali.
FullOptJS	scala-js-[Nazwa-Projektu]-jsdeps.min.js	Plik z kodem zewnętrznych bibliotek użytych w procesie tworzenia aplikacji.

9.2.2.5. Debugowanie

Jednym z najważniejszych procesów, które na celu mają znalezienie błędów w powstałym kodzie jest debugowanie. Proces debugowania aplikacji powinien umożliwiać programiście przynajmniej możliwość zatrzymania kodu w dowolnym miejscu, podejrzenia kodu źródłowego jak i możliwość odczytu i modyfikacji zmiennych.

Kod stworzony za pomocą kompilatora scala.js jest bardzo łatwy w debugowaniu. Przeglądarki posiadają możliwość dołączenia mapy kodów źródłowych, które do danych linijek kodu javascript przypisują ich odpowiedniki w pliku źródłowym. Umożliwia to prace z oryginalnym kodem źródłowym podczas używania przeglądarki internetowej.

Pliki z mapami źródłowymi mają format **{SkompilowanyPlik}.map**, gdzie SkompilowanyPlik jest artefaktem naszego procesu kompilacji, np. **scala-js-[Nazwa-Projektu]-fastopt.js**.



Rys. 9.2. Przykład procesu debugowania kodu. Program zostaje zatrzymany na linijce przed dodaniem napisu Hello World. [2]

9.2.3. Hello World

Za pomocą powyższych metod i narzędzi można stworzyć w bardzo prosty sposób przykładową aplikację wykorzystującą język Scala.js. W tym celu należy użyć polecenia **sbt new sbt/scala-seed.g8**, które spowoduje utworzenie minimalnego projektu Scali. Aby projekt wykorzystywał Scale.js należy w nim wykonać parę zmian:

[2] Na obrazku na stronie widnieje już napis Hello World. Jest to artefakt, który został stworzony przy poprzednim uruchomieniu strony i jeszcze się nie odświeżył.

1. Należy dodać plik ./project/plugins.sbt[3], wewnątrz którego znajdzie się instrukcja **addSbtPlugin(órg.scala-js"% śbt-scalajs"% "0.6.20")**, która poinformuje sbt o konieczności użycia pluginu Scali.js.

2. W pliku ./project/build.properties musi zostać określona wersja narzędzia sbt, którym będzie kompilowany projekt.

3. Plik build.sbt, który zawiera ustawienia związane z budową projektu, musi zostać zmodyfikowany w celu użycia plugina Scali.js jak i pomocniczych bibliotek. Przykładowy plik, który jest używany w projekcie można zobaczyć na listingu 9.4.

Po takim zabiegu należy już tylko dodać kod źródłowy Scali do folderu ./src/main/scala. Przykładowy kod można znaleźć na listingu 9.5

```
enablePlugins(ScalaJSPlugin)
libraryDependencies += "org.scala-js" %%% "scalajs-dom" % "0.9.1"
libraryDependencies += "be.doeraene" %%% "scalajs-jquery" % "0.9.1"

name := "Flashcards"
scalaVersion := "2.12.2"

// Informacja o tym, iż chcemy aby nasz kod zawsze miał uruchamianą metodę main ↩
    po wczytaniu witryny.
scalaJSUseMainModuleInitializer := true

skip in packageJSDependencies := false
jsDependencies += "org.webjars" % "jquery" % "2.1.4" / "2.1.4/jquery.js"
jsDependencies += "org.webjars.bower" % "jsrender" % "1.0.0-rc.70" / ↩
    "1.0.0-rc.70/jsrender.js"
```

Listing 9.4. Plik .sbt, który jest wykorzystywany w projekcie.

[3]./ jest katalogiem głównym projektu w tym przypadku.

```
package helloworld
import org.scalajs.dom
import dom.document

object TutorialApp {
  def main(args: Array[String]): Unit = {
    appendPar(document.body, "Hello World")
  }

  def appendPar(targetNode: dom.Node, text: String): Unit = {
    val parNode = document.createElement("p")
    val textNode = document.createTextNode(text)
    parNode.appendChild(textNode)
    targetNode.appendChild(parNode)
  }
}
```

Listing 9.5. Przykładowy projekt w Scali.js wypisujący napis Hello World na stronie.

10. Opis aplikacji

11. Podsumowanie oraz wnioski

TODO

Zauważyłem, że używam tych citów do : -zostawienia odnośnika dla czytelnika, jeśli chciałby o danym temacie poczytać więcej. Na przykład odnośnik do kompilatora Closure, gdy wspominam o tym, że scala.js spełnia wszystkie warunki do użycia kompilatora closure.

Bibliografia

[1] Dr. Bruce Abbott's. *Human Memory*. URL: *http://users.ipfw.edu/abbott/120/Ebbinghaus.html* (term. wiz. 2017-12-31).

[2] Dr. John Wittman. *The Forgetting Curve*. URL: *https://www.csustan.edu/sites/default/files/groups/Writing%20Program/forgetting_curve.pdf* (term. wiz. 2017-12-31).

[3] *Introduction to the C# Language and the .NET Framework*. URL: *https://docs.microsoft.com/en-us/dotnet/csharp/getting-started/introduction-to-the-csharp-language-and-the-net-framework* (term. wiz. 2017-01-03).

[4] *Implementing the Repository and Unit of Work Patterns in an ASP.NET MVC Application (9 of 10)*. URL: *https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application* (term. wiz. 2017-12-31).

[5] *Ninject*. URL: *https://github.com/ninject/Ninject* (term. wiz. 2017-12-31).

[6] *SQL Server 2016*. URL: *https://www.microsoft.com/pl-pl/sql-server/sql-server-2016* (term. wiz. 2017-01-03).

[7] *What is NuGet?* URL: *https://www.nuget.org/* (term. wiz. 2017-01-03).

[8] Microsoft. *Unit Testing: Testing the Inside*. URL: *https://msdn.microsoft.com/en-us/library/jj159340.aspx* (term. wiz. 2017-01-01).

[9] *Syntactically Awesome Style Sheets*. URL: *http://sass-lang.com/* (term. wiz. 2017-01-01).

[10] *Scala (programming language)*. URL: *https://en.wikipedia.org/wiki/Scala_(programming_language)* (term. wiz. 2017-01-02).

[11] *Tour of Scala*. URL: *http://docs.scala-lang.org/tour/tour-of-scala.html)* (term. wiz. 2017-01-02).

[12] *Dartium: Chromium with the Dart VM*. URL: *https://webdev.dartlang.org/tools/dartium* (term. wiz. 2017-01-03).

[13] *Apache Ivy™*. URL: *http://ant.apache.org/ivy/* (term. wiz. 2017-01-03).

[14] *Hands-on Scala.js : The Compilation Pipeline*. URL: *http://www.lihaoyi.com/hands-on-scala-js/#TheCompilationPipeline* (term. wiz. 2017-01-02).

[15] *Compilation and optimization pipeline.* URL: *https://www.scala-js.org/doc/internals/compile-opt-pipeline.html* (term. wiz. 2017-01-02).

[16] *Understanding the Restrictions Imposed by the Closure Compiler.* URL: *https://developers.google.com/closure/compiler/docs/limitations* (term. wiz. 2017-01-02).

[17] *Closure Compiler.* URL: *https://developers.google.com/closure/compiler/* (term. wiz. 2017-01-02).