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

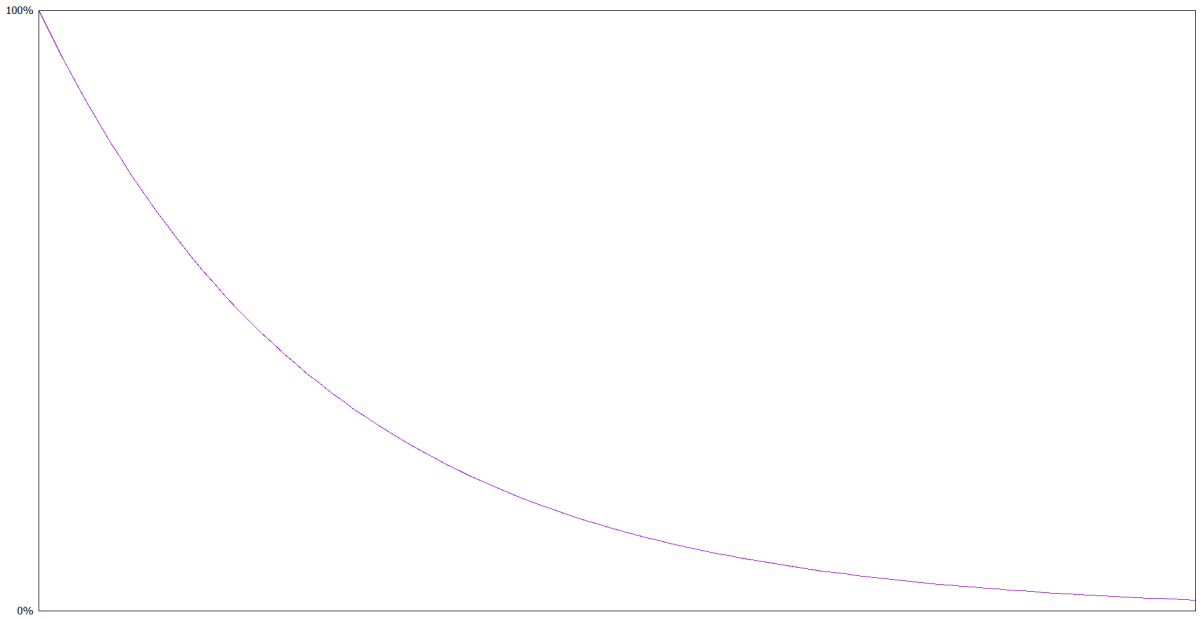
# 1. Wstęp

TODO WSTEP

# 2. Przyswajanie wiedzy

Jeśli opisuje coś na podstawie strony internetowej pana Doktora to czy to jest plagiat?

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. 2.1.** Eksponencjalna krzywa zapominania.

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

## 2.1. Interwały potwórzeń

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

# 3. .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 3.1.** Przykładowy kod aplikacji "Hello World"w języku CIL

## 3.1. 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:

### 3.1.1. .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.

### 3.1.2. .NET Framework

Jest to 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 kod bajtowy skompilowany z języka CIL. Ta implementacja jest używana w tej pracy inżynierskiej.

### 3.1.3. 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 aplikacji w rodzinie języków .NET.

### 3.1.4. 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. 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.

## 4.1. Model-Widok-Kontroler

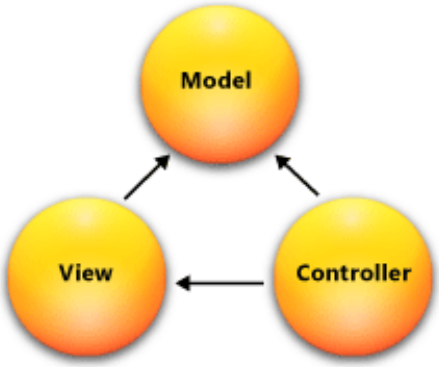
Większość dzisiejszych systemów komputerowych działa na zasadzie wyświetlania danych, które aktualnie znajdują się w bazie danych i ewentualna ich modyfikacja. Z tego też powodu narodził się wzorzec Model-Widok-Kontroler(ang. Model-View-Controller), który rozdziela logikę aplikacji na 3 główne segmenty:

1. Model - Służy do przechowywania danych, pobierania danych i ich ewentualnej zmiany na podstawie zapytań z kontrolera.

2. Kontroler - przetwarza zapytania użytkownika, które przekazuje do modelu, który jest następnie wyświetlany jako widok.

3. Widok - Służy do wyświetlania informacji

Cała moja aplikacja została skonstruowana zgodnie z tym wzorcem projektowym. ASP.NET MVC, jak już sama nazwa wskazuje, bardzo mocno opiera się na nim. Dostarczana jest klasa bazowa **Controller**, która dostarcza wszystkie podstawowe metody do wyświetlania odpowiednich treści danych i zarządzania zapytaniami. Całe to rozwiązanie niesie za sobą korzyści związane z warstwą wyświetlania prezentowanych danych. Jest to warstwa aplikacji, w której bardzo często dochodzi do zmian. Dzięki podejściu MVC i odseparowaniu danych od widoku jesteśmy w stanie tworzyć jak i zmieniać widoki, bez wpływu na kod biznesowy aplikacji.

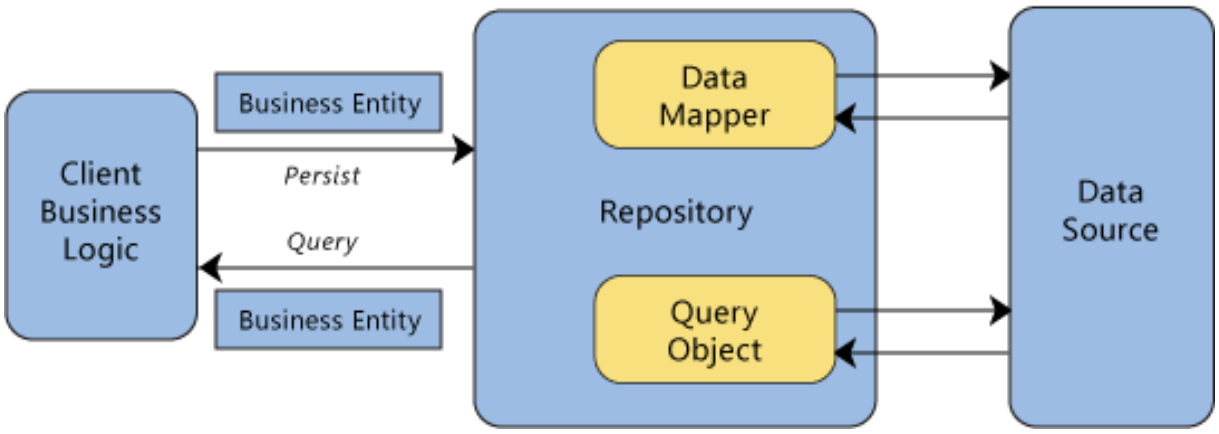



**Rys. 4.1.** Podać źródło

## 4.2. Wzorzec Repozytorium[3]

Wewnątrz naszej aplikacji będziemy używać bazy danych Microsoft SQL. Bardzo często jest tworzony kod, którego celem jest zwrócenie bardzo podobnych danych. W celu zmniejszenia redundancji kodu jak i odseparowania zależności i odpowiedzialności wykorzystałem wzorzec Repozytorium (z ang. Repository Pattern).

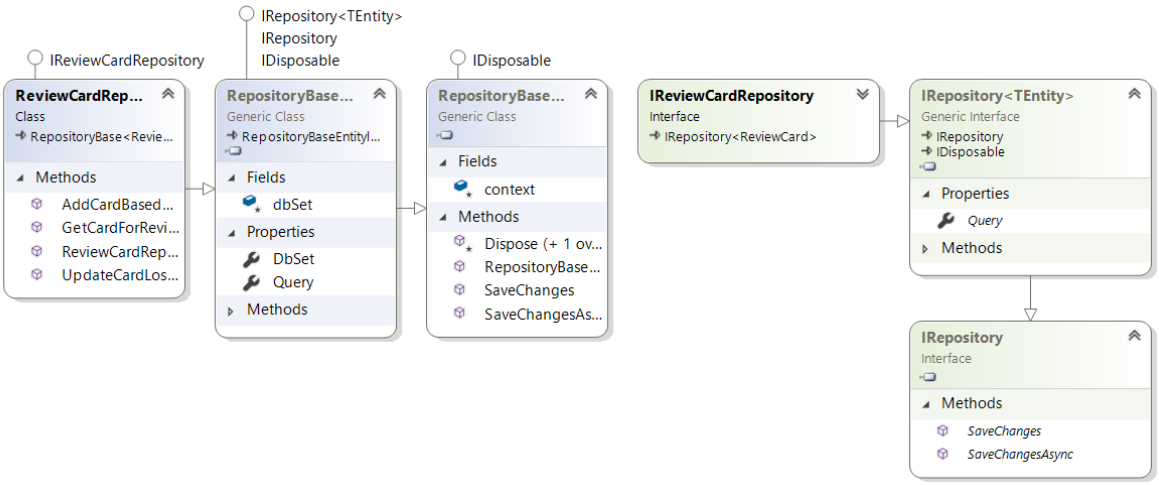
Wzorzec ten wykorzystuje obiekty, zwane repozytoriami, których jedynym zadaniem jest pobieranie i modyfikowanie danych po stronie serwera SQL. Nie zawierają żadnej logiki biznesowej i są niezwiązane z resztą kodu danej aplikacji.




**Rys. 4.2.** Wzorzec repozytorium. Źródło: msdn.microsoft.com

Wewnątrz mojego projektu cały wzorzec repozytorium został oparty na interfejsach **IRepository** i **IRepository<T>** (który implementuje **IRepository**). W założeniu te interfejsy mówią jedynie o tym jakie operacje można zrealizować na danym obiekcie klasy 'T'. Informacje o tym jak te informacje są realizowane jak i to, iż są one realizowane za pomocą bazy danych Microsoftu są przechowywane w klasie **RepositoryBase<T>**. Wszystkie kolejne stworzone repozytoria dziedziczą po **RepositoryBase<T>** a ich interfejsy implementują **IRepository<T>**. Takie działanie pozwala nam na brak ścisłego powiązania naszych repozytoriów z bazą danych

Microsoft SQL. Wystarczy iż zmienimy **RepositoryBase<T>** na jakąkolwiek inną klasę implementującą **IRepository<T>** i wykorzystamy ją w naszych repozytoriach. Dzięki temu małym nakładem siły moglibyśmy zastąpić Microsoft SQL danymi zapisywanymi w pamięci RAM naszego komputera wewnątrz wyspecjalizowanych kolekcji.




**Rys. 4.3.** Repozytorium ReviewCardRepository wraz z implementowanymi interfejsami i odziedziczonymi klasami.

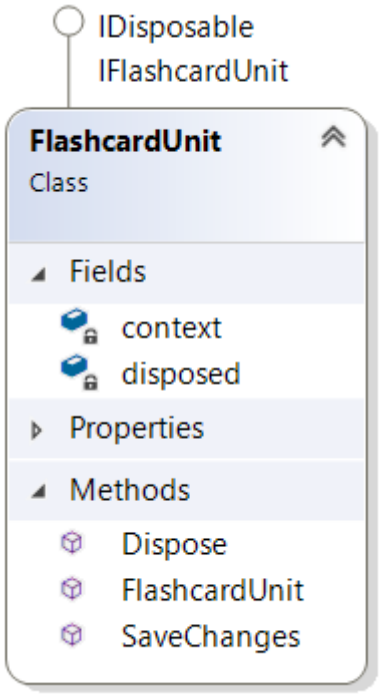
## 4.3. Jednostka Pracy[3]

Wzorzec Jednostki Pracy (z ang. Unit of Work) ma na celu uporządkowanie pracy z repozytoriami za pomocą umieszczenia ich wszystkich w jednej klasie. Dodatkowo wszystkie repozytoria współdzielą kontekst dostępu do bazy danych. Zapisanie danych odbywa się poprzez wywołanie metody **SaveChanges** wewnątrz Jednostki Pracy (SaveChanges jest konwencją wewnątrz mojego projektu).

## 4.4. Mapowanie obiektowo-relacyjne

Wewnątrz mojego projektu wykorzystałem narzędzie Entity Framework, które pozwala na korzystanie z mapowania obiektowo-relacyjnego. Dzięki tej technicę mogę uprzednio zdefiniowane dane tabelaryczne odzworować za pomocą klas, które zostaną stworzone na podstawie definicji tabeli i relacji między nimi. Powstałe klasy są używane wewnątrz kolekcji, implementujących interfejs **IQueryable<T>**, na których możemy wykonywać zapytania z pomocą **LINQ**a.

Takowe rozwiązanie ma wiele zalet :




**Rys. 4.4.** Jednostka pracy wykorzystywana w projekcie.

1. Dane z danej tabeli będziemy mogli otrzymać zawsze w tym samym formacie. Unika się dzięki temu sytuacji gdzie dana tabela ma wiele odpowiadających jej klas w kodzie, które służą jedynie do dostępu do danych.

2. W przypadku zmiany typu w tabeli typ ten możemy bardzo prosto zmienić w kodzie.

3. Zmniejsza to zakres potrzebnych umiejętności w celu używania danych występujących w bazie danych. Dany programista nie musi znać języka SQL, aby w sposób bezproblemowy pobrać interesujące go dane, nawet gdy tworzy bardzo skomplikowane zapytania.

4. Łatwiejsza nawigacja po zależnościach między tabelami. Na danym typie możemy na przykład wykorzystać operacje **Find All References**, która wskaże nam użycia danego typu w naszym kodzie. W przypadku kodu SQL nie mielibyśmy takiej możliwości.

## 4.5. Wstrzykiwanie zależności

Wstrzykiwanie zależności (

Informacja o tym, iż dana klasa ma zamiar używać instancji innej klasy może być zawarta poprzez odpowiednie wywołanie danej funkcji, bądź poprzez użycie listy interfejsów w konstruktorze, która zostanie wypełniona odpowiednimi instancjami.

W przypadku mojej aplikacji użyłem do tego biblioteki Ninject[4]. Jest to bardzo popularna biblioteka realizująca wzorzec wstrzykiwania zależności. Dzięki niej możemy zdefiniować listę

potrzebnych interfejsów/klas jako parametry konstruktora naszego kontrolera, a te utworzą się automatycznie podczas żądania użytkownika. Jeśli tworzone klasy także będą wymagały do swojego istnienia pewnych interfejsów bądź klas i zostanie to uściślone w ich konstruktorze to także dla nich zostaną dostarczone odpowiednie instancje.

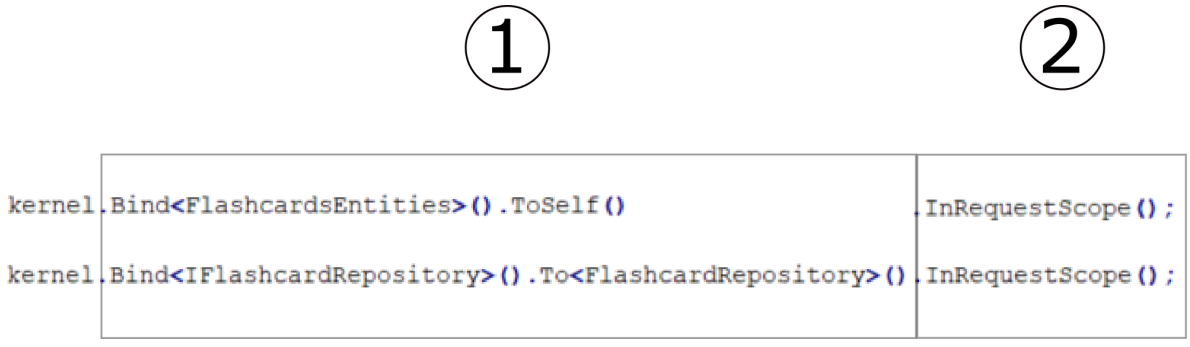
Jednakże sam interfejs nie daje żadnej informacji o tym, jaką klasę należy zinstancjować, ponieważ może być wykorzystywany przez wiele klas pochodnych. W tym celu należy poinformować Ninject o tym, jakie klasy mają zostać zinstancjonowane poprzez zbindowanie interfejsów do odpowiednich klas. To działanie jest przedstawione na poniższym listingu:

```
public static void RegisterServices(IKernel kernel)
{
kernel.Bind<FlashcardsEntities>().ToSelf().InRequestScope();
kernel.Bind<IFlashcardRepository>().To<FlashcardRepository>().InRequestScope();
}
```

**Listing 4.1.** Przykładowy kod bindowania dla biblioteki Ninject

Jak widzimy proces pojedynczego bindowania możemy podzielić na 2 części:

1. Informacja o tym co powinno zostać stworzone przez Ninject gdy jest proszone o daną klasę
2. Informacja o zasięgu współużywania danej instancji.




**Rys. 4.5.** Proces bindowania z wyszczególnionymi częściami składowymi

Wewnątrz pierwszej części określamy jakim typem ma zostać zastąpiony dany typ w procesie wstrzykiwania. Na przykładzie widzimy, że typ może zastąpićsam siebie bądź może zostać użyty inny typ, który będzie jego instancją. Należy pamiętać, iż ten drugi typ musi implementować, dziedziczyć bądź być tym samym typem co pierwszy.
Ninject dodatkowo potrafi zbindować dany typ do **metody** lub klasy implementującej interfejs **Provider<T>**. Taka metoda, badź klasa musi zwrócić w wyniku swojego działania pełnoprawny obiekt danego typu.

Ninject w swoim domyślnym zachowaniu za każdym razem jak poprosimy go obiekt danej klasy to tworzony jest następny obiekt danej klasy. Jednakże często dochodzi do sytuacji, że w danym segmencie kodu chcielibyśmy aby obiekt danej klasy był współdzielony między wszystkimi instacjami wytworzonymi przez Ninject. Wtedy za każdym razem jak poprosimy obiekt klasy A to zawsze w ramach danego zasięgu dostaniemy ten sam obiekt i nie zostanie on utworzony ponownie. Jest to szczególnie przydatne w środowisku serwerowym, gdy chcemy aby dla każdego zapytania klienta wszystkie repozytoria i serwisy były zawsze takie same. Nie tylko nie ponosimy kosztów utworzenia danej instancji, lecz także współdzielone repozytoria mogą szybciej zwracać informacje odnośnie danych po stronie serwera SQL gdy zobaczą iż mają te dane w swoim cache'u.

# 5. Testy

Bardzo ciężko jest napisać w dzisiejszym świecie aplikację nie zaopatrując się w zestaw testów bądź innych mechanizmów sprawdzających poprawność wykonania kodu. Z tego też powodu napisałem duży zestaw testów jednostkowych, które sprawdzają, czy pojedyncze funkcjonalności danych serwisów są funkcjonalne.

## 5.1. Testy Jednostkowe

Odwołania do innych serwisów/obiektów w większości zastąpiłem za pomocą atrap obiektów (z ang. mock object), które zastępują wywołania danych funkcjonalności poprzez zwrócenie domyślnych wartości. Możemy także zastąpić daną funkcjonalność atrapy jakąś mało skomplikowaną logiką w celach testowych. Taka atrapa nie dość, iż potrafi odłączyć nasz obiekt od reszty systemu, przez co jesteśmy w stanie testować go jednostkowo to dodatkowo zawiera bardzo dużo funkcjonalności pomagających zweryfikować czy dany test przebiegł poprawnie. W celu tworzenia atrap wykorzystałem framework Moq[5].

Większość, jeśli nie wszystkie testy zostały ułożone zgodnie ze wzorcem AAA[6] - Aranżacja (z ang. Arrange) Akcja (z ang. Act) Asercja (z ang. assert). Takie ułożenie testów sprawia, iż testy są konsystentne i czytelne. Potrzebujemy bardzo małej ilości czasu, aby zaznajomić się z danym testem.

```
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 5.1.** Przykładowy test jednostkowy wykorzystujący metodę AAA

## 5.2. Testy Integracyjne

Czy ten fragment jest potrzebny? Omawiam tutaj co prawda część aplikacji, lecz niestety nie jest on de facto wykorzystywany przeze mnie

Dla potrzeb porzuconej części aplikacji został także napisany mały zestaw testów integracyjnych testujących serwis do obsługi zapisu i odczytu plików XML, jak i test tworzenia tymczasowych plików w systemie Windows. Testy te miały za zadanie testować współdziałanie powyższych funkcjonalności z zewnętrznymi komponentami nieobsługiwanymi przez moją aplikację. Z tego też powodu nie został tutaj wykorzystany framework do tworzenia atrap. Wszystkie testy, pomimo tego, iż poprawne to aktualnie nie dowodzą poprawności żadnego elementu aktualnie używanej aplikacji, ponieważ były wykorzystywane przez porzuconą część aplikacji.

# 6. Kaskadowe Arkusze Styli

W moim projekcie wykorzystuję tzw. Kaskadowe Arkusze Styli [**CSSDoc**]. Ich przeznaczeniem jest stylizować wygląd wyświetlanej strony HTML dla użytkownika. Pewne fragmenty strony są odnajdywane dzięki selektorom i są formatowane według reguł zdefiniowanych dla danego selektora. To pozwala nam na ponowne użycie tych samych styli w obrębie naszych aplikacji, dzięki zdefiniowaniu bardziej uogólnionych selektorów.

## 6.1. Syntaktycznie Zarąbiste Arkusze Styli

W dzisiejszym świecie nikt obeznany z technologią tworzenia stron internetowych nie używa już czystego języka CSS do tworzenia styli. Przy tworzeniu styli dla danej strony stykamy się z bardzo dużą redundancją kodu, która jest związana z tym iż pewne fragmenty strony wyglądają zawsze tak samo (np. motyw kolorystyczny dla większości elementów jest ten sam). Brak zmiennych w języku CSS powoduje, iż zmiana kolorystyki strony jest żmudnym i ciężkim zadaniem. Z tego tez powodu wykorzystuję język SASS - Syntactically Awesome Style Sheets (Syntaktycznie Zarąbiste Arkusze Styli)[7]. Jest on kompatybilny ze wszystkimi wersjami CSS. Co powoduje, iż każdy kod napisany w CSS jest zgodny z językiem SASS. Sass oferuje nam bardzo dużo funkcjonalności, między innymi:

1. Zmienne - dane wartości możemy zapisywać jako zmienne i używać ich wielokrotnie w obrębie naszych plików.

2. Zagnieżdżanie - możemy zagnieżdżac selektory wewnątrz siebie. Odwzorowywuje to naturę plików HTML, przez co kod jest bardziej czytelny.

3. Możliwość załączania innych plików. Pozwala to na logiczne podzielenie arkuszy styli.

4. Mixin - Pozwala na tworzenie grup deklaracji, które można ponownie wykorzystać w danym stylu.

5. Dziedziczenie - potrafimy za pomocą słówka kluczowego **@extend** odziedziczyć właściwości danego selektora.

6. Operatory - Możemy wykorzystywać znaki +, -, *, /, % do operacji 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 6.1.** Przykładowy kod SCSS wykorzystujący mixin

# 7. Scala[8]

Scala jest językiem programowania ogólnego zastosowania, który został wprowadzony na rynek przez Laboratorium "École Polytechnique Fédérale de Lausanne". Jest to język, który jest kompilowany bezpośrednio do kodu bajtowego Javy, co powoduje, iż programy w nim napisane z łatwością uruchamiają się w środowisku wykonywalnym maszyny wirtualnej Javy. Scala jest językiem wielo-paradygmatowym[9]. Korzysta z dobrodziejstw programowania funkcjonalnego i obiektowego.

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

**Listing 7.1.** Hello world napisany w języku Scala.

## 7.1. Javascript

Mankamentem tworzenia aplikacji webowych jest używanie Javascriptu. Coraz większa rzesza programistów odchodzi od jego używania na rzecz innych języków, które są kompilowalne do niego, takich jak np. Typescript, Coffescript, bądź jak w moim przypadku Scala.JS. Taka zmiana pozwala nam popełniać o wiele mniej błędów pisząc kod. Pomimo tego, iż wynikowym kod nadal będzie w tym samym języku. Wszystko jest spowodowane przez to, iż Javascript posiada:

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

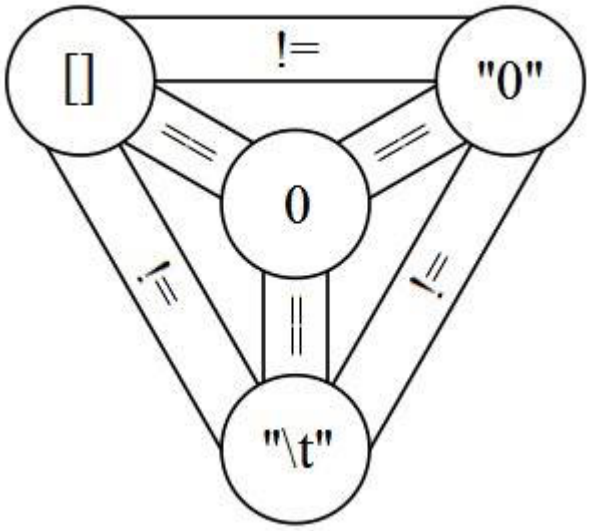
2. Automatyczne wstawianie średników, co może spowodować, iż program działa inaczej niż zamierzał to programista. Czasami potrafi to uratować program napisany przez programistę, lecz często prowadzi do dziwnych i niezrozumiałych błędów, takich jak w przypadku poniższego listingu: 7.2

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

4. Zmienne, którym możemy w każdej chwili zmienić typ przechowywanej wartości. Z tego też powodu bardzo często powstają prozaiczne błędy, gdzie programista oczekując liczby w danym miejscu otrzymał ciąg znakowy co skutkuje błędem w dalszym kodzie programu.

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

**Listing 7.2.** Przykład niepoprawnego kodu Javascript wynikłego z automatycznego wstawienia średnika




**Rys. 7.1.** Przykład dziwnego zachowania języka. Źródło: devrant.com

Chyba jak cytuję obrazek muszę dać pełne źródło jako [x] + na samym dole odnośnik

## 7.2. Scala w wersji JS

Z powodu powyższych problemów zdecydowałem się na użycie scali, która z pomocą projektu scala-js[10] jest językiem kompilowalnym do javascripta. Dzięki temu jestem w stanie wykorzystać wszystkie zalety języka Scala, w tym:

1. Silne typowanie. Dzięki temu mój kod nie zawiera bardzo prostych błędów, które mogą wystąpić w procesie tworzenia oprogramowania.

2. Brak skupiania się na dziwnych i frustrujących aspektach języka Javascripta. Dzięki temu jestem w stanie zająć się pisanym przeze mnie kodem, aniżeli walczyć z głupimi błędami.

### 7.2.1. sbt

sbt (Simple Build Tool) jest narzędziem o otwartym kodzie źródłowym, pozwalającym nam na zarządzanie procesem budowania naszej aplikacji napisanej w języku Scala lub Java. W celu wykorzystania naszej aplikacji należy użyć polecenia **sbt new sbt/scala-seed.g8**, które spowoduje utworzenie minimalnego projektu Scali, który możemy dostosować do własnych potrzeb. Z tego też powodu musimy wykonać pare istotnych zmian w projekcie, aby był on kompilowalny do języka Javascript.

Budowanie dziwnie brzmi. Czy może powinienem tu użyć angielskiej nazwy?

1. Należy dodać plik ./project/plugins.sbt[1], wewnątrz którego znajdzie się instrukcja **addSbtPlugin(órg.scala-js"% śbt-scalajs"% "0.6.20")**, która informuje o tym, iż chcemy użyć Scali.JS

2. W pliku ./project/build.properties musimy określić wersje używanego przez nas narzędzia sbt poprzez dodanie bądź zmodyfikowanie linijki:

```
sbt.version=0.13.16
```

3. Następnie musimy zmodyfikować plik build.sbt znajdujący się w głównym folderze projektu. Przykładowy plik znajduje, który jest używany w projekcie, znajduje się na listingu 7.4.

```
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 7.3.** Plik .sbt, który jest wykorzystywany w projekcie.

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

### 7.2.2. Proces budowania projektu

Projekt kompilacji projektu jest podzielony na 2 etapy[11]:

1. Początkową kompilację
2. Szybką optymalizację
3. Pełną optymalizację

#### 7.2.2.1. 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 tych plików używa w celu wspomagania pisania kodu Scali. Pliki .sjsir (Nazwa rozszerzenia jest skrótem od ŚcalaJS Intermediate Representation")[12] zawierają kod przejściowy między Scalą a Javascriptem. Większość konstrukcji została zastapiona 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 7.4.** Przykładowy plik .sjsir dla projektu wyświetlającego HelloWorld na ekranie.

Czy tutaj powinienem napisać o mojej obserwacji o tym, iż pliki bibliotek sjsir nie znajdują się w plikach wytworzonych przez kompilator scala.js?

#### 7.2.2.2. Szybka optymalizacja

Z tego też powodu stosuje się drugi krok, który jest optymalizacją kodu .sjsir i stworzeniem kodu wynikowego. W tym celu używamy optymalizatora FastOptJS poprzez wpisanie komendy **FastOptJS** w sbt. Optymalizacja ma na celu:

1. Wyeliminowanie fragmentów kodu, które są nieużywane. Na przykład kod biblioteki standardowej, którego nie użyliśmy.

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.

   Czy termin inline'owanie jest poprawny?

Dzięki tej operacji nasz kod wykonywalny zmniejszy się z 20 MB do 1.5-2.5MB[12].

#### 7.2.2.3. Pełna optymalizacja

Kod, który jest tworzony przez Scala.js jest zgodny z restrykcjami narzuconymi przez kompilator Closure[13]. Jest to narzędzie stworzone przez firmę Google, które potrafi optymalizować kod Javascriptowy.[14]

Ten kompilator jest wykorzystywany jako ostateczny krok optymalizacyjny Scali.js. Stworzony w tym procesie kod wykonywalny będzie miał wielkość między 150 KB do kilkuset KB.[12] W celu użycia pełnej optymalizacji należy wywołać polecenie **FullOptJS** w sbt.

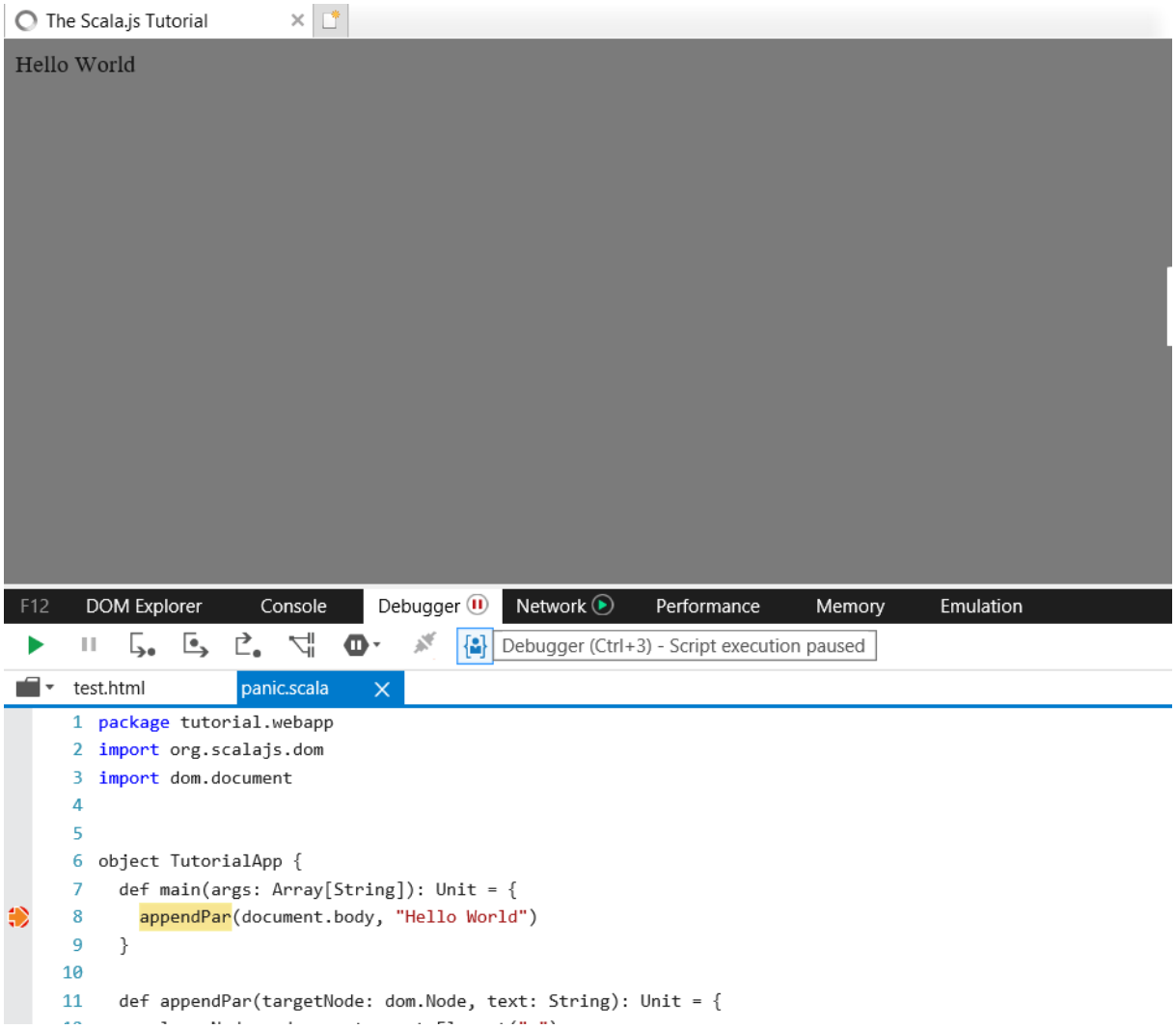
#### 7.2.2.4. Pliki javascript

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

| | | |
|---|---|---|
| FastOptJS | scala-js-[Nazwa-Projektu]-fastopt.js | Plik z kodem wykonywalnym |
| FastOptJS | scala-js-[Nazwa-Projektu]-fastopt.js.map | Plik mapujący kod Scali do kodu Javascript. |
| 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 Scali do kodu Javascript. |
| FullOptJS | scala-js-[Nazwa-Projektu]-jsdeps.min.js | Plik z kodem zewnętrznych bibliotek użytych w procesie tworzenia aplikacji. |

#### 7.2.2.5. Debugowanie

Kod stworzony za pomocą kompilatorów scala.js jest bardzo łatwy w debugowaniu. Użytkownik nie musi analizować bardzo trudnego, zoptymalizowanego kodu javascript w celu analizy programu. Zamiast tego używane są mapy kodów źródłowych[15], które mapują skompilowane pliki do plików źródłowych. Dzięki temu jesteśmy w stanie debugować kod w przeglądarce internetowej analizując kod Scali, pomimo tego, iż pod spodem działa tak naprawdę Javascript. Wszystkie pliki z mapami kodów źródłowych mają format **{SkompilowanyPlik}.map**, gdzie SkompilowanyPlik jest artefaktem naszego procesu kompilacji, np. **scala-js-[Nazwa-Projektu]-fastopt.js**.




**Rys. 7.2.** Przykład procesu debugowania kodu. Nasz program zostaje zatrzymany na linijce przed dodaniem napisu Hello World. [2]

Foot note nie wyświetla - zbadać dlaczego

### 7.2.3. Hello World

Warto jest robić tą sekcję gdzie opisze proces tworzenia hello worlda? Nie będe tego aktualnie pisał bo tylko będe tracił czas jeśli jest to zbedne. Ale z chęcią bym to opisał

# 8. Opis aplikacji

### 8.0.1. Założenia aplikacji

Aplikacja ma za zadanie nauczyć użytkownika słówek z danego języka obcego, który jest wybrany przez użytkownika. Program od strony serwerowej został napisany w języku C# z użyciem frameworka ASP.NET MVC z wykorzystaniem bazy danych Microsoft SQL. Od strony klienta użyty został HTML wraz z arkuszami styli stworzonymi w oparciu o style SASS. Uruchamiany kod Javascript był skompilowanym kodem S

# 9. Podsumowanie oraz wnioski

TODO

# 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] *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).

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

[5] *Moq*. URL: *https://github.com/moq/moq* (term. wiz. 2017-01-01).

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

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

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

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

[10] *Scala.js*. URL: *http://www.scala-js.org/* (term. wiz. 2017-01-02).

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

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

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

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

[15] Nick Fitzgerald John Lenz. *Source Map Revision 3 Proposal*. URL: *https://sourcemaps.info/spec.html* (term. wiz. 2017-01-02).