POLITECHNIKA POZNAŃSKA
WYDZIAŁ ELEKTRYCZNY
INSTYTUT AUTOMATYKI I INŻYNIERII INFORMATYCZNEJ

Konrad Dysput

PRACA DYPLOMOWA INŻYNIERSKA

Wirtualny system plików zarządzający przechowywaniem i synchronizacją danych w chmurze

Promotor: dr Andrzej Sikorski

Poznań, 2017

Spis treści

1. Wstęp

Systemy archiwizacji danych w związku z coraz większą ilością przetwarzanych informacji zyskały na ogromnej popularności.

1.1. Wirtualny system plików

1.2. Cel i zakres pracy

2. Fukncjonalności systemu

2.1. Porównanie systemów plików NTFS

2.2. Porównanie systemów plików SMB

2.3. Porównanie systemów plików FAT32

2.4. Usługa Blob Azure Storage

3. Projekt wirtualnego systemu plików

3.1. System katalogów

3.2. Menadżer danych

3.3. Udostępniane funkcje

4. Architektura systemu

4.1. Podział projektu

W celu zmniejszenia złożoności oraz kosztów utrzymania projektu zdecydowano się na rozbicie architektury na cztery moduły. Każdy z wyodrębnionych modułów ma za zadanie spełniać określoną czynność w architekturze systemu. Rozbicie projektu pozwala na łatwiejsze zarządzanie kodem oraz, w przypadku dalszego rozwoju przez większą liczbę programistów umożliwia szybkie wdrożenie nowych uczestników projektu. Zastosowanie warstw projektowych bardzo dobrze sprawdza się przy projektach dużej wielkości w przypadkach, gdy programista dąży do rozłożenia odpowiedzialności komponentów w poszczególnych modułach.

W programie w związku z podziałem projektu na pod moduły możemy wyróżnić rozbudowaną architekturę systemu oraz widoczny podział obowiązków poszczególnych pod projektów. W każdym z wydzielonych programów możemy wyróżnić jego odpowiedzialność za pomocą nazw pomocniczych w projekcie tworzonych na podstawie przestrzeni nazw w języku C#. Zbiór oraz opis wszystkich podprojektów znajduje się w tabeli 4.1.ls

4.2. Wzorce architektoniczne

W celu poprawnej integracji modułów skorzystano z wielu wzorców architektonicznych oraz projektowych. Jednym z najpopularniejszych wzorców architektonicznych w aplikacjach internetowych jest wzorzec Model-Widok-Kontroler (eng. Model-View-Controller, MVC) zakładający rozkład podziału obowiązków na trzy główne człony programu. Kontroler we wzorcu pełni funkcję klasy odpowiedzialnej za przyjmowanie żądań użytkownika w celu pobrania strony, danych lub uzyskania dostępu. Jest to część modułu odpowiedzialna za zarządzanie przepływem informacji pomiędzy modelami, a widokami. Kontroler w przypadku aplikacji z tak rozbitą architekturą przekierowuje dane otrzymane od użytkownika do modułu odpowiedzialnego za logikę biznesową. Rezultatem wykonania metod na modułach jest otrzymanie wynikowego modelu bazodanowego, który następnie jest przekazywany do widoku pod postacią Modelu Widoku (eng. ViewModel). Transformacja modelów jest niezbędna ze względu na bardzo dużą ilość danych zwracanych z serwisu, które nie zawsze są potrzebne przy tworzeniu

Tabela 4.1: Podział architektury systemu

Nazwa	Typ	Opis
Moduł aplikacji użytkownika	Aplikacja internetowa ASP.NET MVC	Aplikacja internetowa odpowiedzialna za sterowanie komunikacji pomiędzy interfejsem użytkownika na stronie internetowej, a logiką aplikacji.
Moduł logiki biznesowej	Biblioteka klas	Aplikacja biblioteczna odpowiedzialna za kontrolowanie przepływu informacji pomiędzy aplikacją internetową, do której użytkownik wysyła żądania, a bazą danych na której wykonywane są operacje pobrania danych niezbędnych do stworzenie i wyświetlenia widoku użytkownikowi.
Moduł bazodanowy	Biblioteka klas	Aplikacja zawierające modele struktury bazodanowych potrzebnych do stworzenia bazy danych (eng. Code-First)oraz operacji na nich przy użycia języka zapytań funkcyjnych LINQ.
Moduł testów	Projekt testów jednostkowych	Aplikacja zawierająca scenariusze testowe oraz testy sprawdzające poprawność działania kodu poprzez sprawdzanie oczekiwanego wyjścia z metod.

widoku użytkownika oraz możliwość zmniejszenia złożoności operowanego modelu. Dane przekazywane do widoków przy pomocy silnika Razor tworzą stronę w formacie HTML z nałożonymi informacjami zwróconymi z bazy danych.

W wielu aplikacjach internetowych w architekturze Model-Widok-Kontroler programista dąży do uzależnienia nowo utworzonych klas kontrolerów od pewnych, już utworzonych w aplikacji składowych, jakimi są serwisy. Umożliwiają one wydzielenie logiki biznesowej do osobnych klas oraz zmniejsza ilość kodu napisanego w kontrolerze, przez co klasa ta pełni funkcję pośrednika pomiędzy widokiem, a modelem bazodanowym. Niepotrzebne operacje bazodanowe w kontrolerze oraz duża ich złożoność powoduje duże trudności w późniejszym rozwoju aplikacji oraz brak możliwości ponownego użycia kodu.

Nowo tworzone kontrolery mogą w bardzo szybkim czasie zyskać dużą liczba funkcjonalności dzięki zastosowaniu wzorców architektonicznych Odwróconego sterowania oraz Wstrzykiwaniem zależności. Skorzystanie z nich umożliwia uzyskanie funkcjonalności, w której możemy swobodnie podłączać oraz odłączać kolejne moduły wpływając na zwiększenie lub ograniczenie funkcjonalności w programie. Dobrymi praktykami programisty jest również ponowne używanie raz napisanego kodu w wielu miejscach. Operacje, które są wykonywane na katalogach oraz plikach są do siebie zbliżone, mimo że korzystają z dwóch różnych kontrolerów odpowiedzialnych za przepływ informacji. Kontrolery te mimo innych zastosowań korzystają z tych samych metod napisanych w serwisie logiki biznesowej. Zastosowanie wzorców spowoduje wzbogacenie kontrolera na początku jego istnienia o stworzone wcześniej funkcjonalności, które mogą zostać wykorzystane ponownie, zmniejszając tym samym złożoność kodu oraz ułatwiając późniejszą modyfikację lub rozwój.

Szczególnie wartym uwagi wzorcem architektury jest Wstrzykiwanie zależności. Kontroler mógł zostać uzależniony od pewnego rodzaju klas przy wykorzystaniu biblioteki Unity, dla języka C#. Wstrzykiwanie zależności polega na przesyłaniu do konstruktora kontrolera obiektów zdefiniowanych przez programistę w konfiguracji biblioteki. Bez mechanizmu Unity, byłoby to zadanie bardzo ciężkie do zrealizowania, ponieważ klasy pełniące funkcje kontrolerów w aplikacjach internetowych są wywoływane przez użytkownika aplikacji poprzez skierowanie żądania pod odpowiedni adres witryny, zatem nie jesteśmy w stanie przekazać do konstruktora kontrolera parametrów, a jedynie do wywoływanej metody, nazywanej akcją. Aby móc wstrzyknąć złożoną strukturę danych jako parametr metod inicjalizujących należy przygotować interfejs określający funkcjonalność przesyłanego modelu oraz powiadomić bibliotekę Unity w konfiguracji o możliwości wstrzykiwania podanego typu danych do kontrolera. Operacja w bardzo wielu miejscach zmniejsza złożoność kodu. Rozwiązanie może dla programistów nie zaznajomionym z wzorcem Wstrzykiwania zależności sprawić wiele problemów

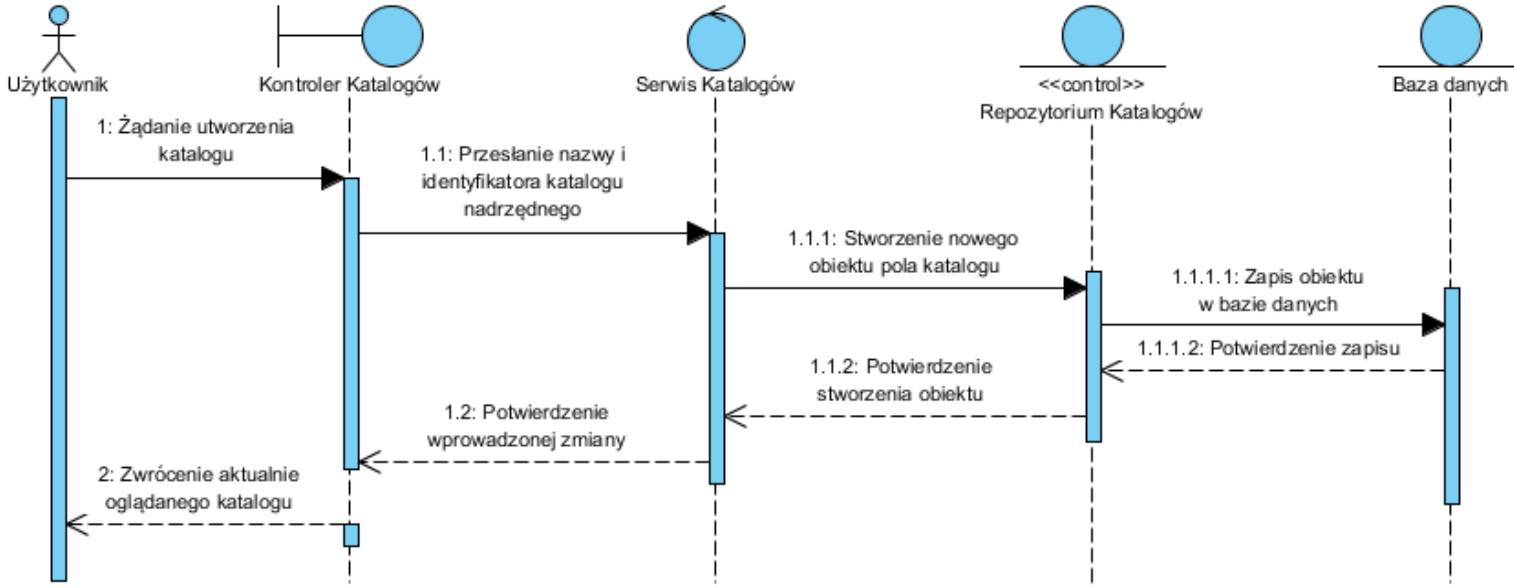
w przypadku dodawania funkcjonalności do kontrolera uzależnionego od pewnych struktur danych.

Ostatnim wzorcem architektonicznym użytym w projekcie jest Odwrócone sterowanie. Bardzo częstym błędem programistów tworzących aplikacje internetowe jest pisanie zaawansowanej oraz złożonej logiki w klasach kontrolera. Kod napisany w akcji jest nie możliwy do ponownego użycia w innych miejscach aplikacji, ponieważ nie zalecane jest tworzenie obiektów kontrolerów w innych kontrolerach. Ponadto kontroler pełni funkcję pośredniczącą pomiędzy logiką bazodanową, a użytkownikiem, w związku z tym tworzenie złożonych operacji na danych w akcjach w znacznym stopniu naruszają dobre praktyki przyjęte przy pisaniu kontrolerów. Stosowanym podejściem w takich sytuacjach jest wydzielanie logiki do osobnych klas odpowiedzialnych za przepływ informacji. W przypadku złożonych aplikacji internetowych zalecane jest wydzielenie klas odpowiedzialnych za przetwarzanie danych do osobnych projektów. Rozwiązanie te powoduje łatwiejszą orientację w kodzie oraz utrzymuje porządek w strukturze projektowej.

4.3. Przepływ danych

W celu komunikacji pomiędzy użytkownikiem,a bazą danych, klasy pełniące funkcję kontrolerów przekazują odebrane oraz sprawdzone, pod względem poprawności danych, modele do serwisów wzorca Odwrotnego sterowania. Cała logika działania aplikacji oraz sposób wykonywanych operacji znajduje się w klasach pełniących funkcję serwisów. Każda ze struktur danych musi zostać zaimplementowana na podstawie wcześniej utworzonego interfejsu, który jest wymagany przez mechanizmy Wstrzykiwania zależności. Struktury serwisów używają generycznych repozytoriów w celu dostępu do danych znajdujących się w bazie danych.

Każde z repozytoriów inicjalizowane jest w serwisie w momencie potrzeby wykonania operacji bazodanowej na bazie danych. Interfejs, na podstawie którego został stworzony komponent umożliwia podstawowe operacje na bazie danych takie jak: tworzenie, czytanie, aktualizacje oraz usuwanie obiektów. Implementacja repozytorium umożliwia wykonanie wymienionych czynności na dowolnym obiekcie bazodanowym. Raz utworzony obiekt, istnieje przez cały okres czasu potrzebnego na przetwarzanie żądania użytkownika. W przypadku potrzeby implementacji dodatkowych funkcji do generycznego repozytorium dodanie ewentualnego kodu do klasy skutkowałoby rozszerzeniem wszystkich pozostałych repozytoriów. W związku z tym zastosowano mechanizm metod rozszerzonych dla określonego typu klasy odpowiedzialnej za komunikację z bazą danych.



Rysunek 4.1: Diagram sekwencji tworzenia katalogu

4.4. Komunikacja z chmurą danych Azure

Aplikacja w celu przechowywania przesyłanych przez użytkownika plików potrzebuje przestrzeni dyskowej, na której dane mogłyby zostać zapisane. Zarządzanie plikami na serwerze może powodować dużą liczbę problemów między innymi z dostępami do plików oraz katalogów na maszynach, na których aplikacja działałaby bez uprawnień administratorskich. Alternatywnym rozwiązaniem jest magazynu danych oferowany przez usługodawców rozwiązań chmurowych. W celu implementacji przechowywania danych na serwerze skorzystano z chmury Azure z usługi konta magazynu. Wszystkie dane przesłane przez użytkownika zostają zapisane w koncie magazynowym w kontenerze właściciela przestrzeni. Chmura Azure umożliwia podstawowe operacje do zarządzania danymi takie jak dodawanie, usuwanie oraz pobieranie danych. Informacje na temat przechowywanych plików w magazynie znajdują się w bazie danych aplikacji.

5. Implementacja

5.1. Środowisko programistyczne

W celu realizacji poszczególnych założeń projektowych został wykonany złożony program przy użyciu platformy .NET w wersji 4.6. Część serwerowa aplikacji została napisana w języku C# 6.0, a widoki użytkownika zostały stworzone przy pomocy języków SCSS, JavaScript oraz HTML przy użyciu ASP.NET MVC. Całość oprogramowania została stworzona przy wykorzystaniu programu Visual Studio 2015 w wersji Community oraz systemu automatyzacji zadań gulp w środowisku Node.js, służącemu do minifikacji oraz konkatenacji stylów i skryptów. Aplikacja wymaga posiadania na komputerze programisty zainstalowanego silnika bazodanowego MS SQL Server oraz dowolnej przeglądarki internetowej z uruchomioną obsługą języka JavaScript. Środowisko Node.js oraz wszystkie pakiety z nim związane nie są wymagane do dalszej kontynuacji, lecz w znaczny sposób mogą przyspieszyć dalszą pracę. Pomocnymi narzędziami używanymi do analizowania pracy programu jest program rozszerzenie Postman do przeglądarki internetowej Google Chrome lub pakiet Fiddler. Narzędzie te mogą posłużyć do wywoływania akcji kontrolerów poprzez odpowiednie adresy URL. W celu udostępnienia funkcjonalności zapisywania plików w chmurze Azure, należy skorzystać z konta portalu Azure oraz usługi magazynu. Aplikacja została stworzona przy użyciu systemu Windows 10 oraz może zostać uruchomiona przy pomocy programu Internet Information Service (IIS).

5.2. Biblioteki

W celu stworzenia funkcjonalności określonej w pracy wykorzystano liczny zestaw bibliotek platformy .NET oraz interfejsu użytkownika. Wykorzystanie wymienionych poniżej komponentów przyspieszyło w znaczący sposób pracę oraz umożliwiło wykonanie funkcjonalności zgodnie z założoną architekturą systemu.

Tabela 5.1: Użyte biblioteki platformy .NET oraz interfejsu użytkownika

Biblioteka	Moduły	Opis
Automapper	— aplikacji użytkownika — aplikacji biznesowej	biblioteka umożliwiająca mapowanie pomiędzy obiektami odmiennego typu, dzięki której między innymi dokonywana jest zamiana modelu bazodanowego na model widoku.
Unity	— aplikacji użytkownika	Biblioteka umożliwiająca wstrzykiwanie struktur danych. Została użyta w związku z zastosowaniem wzorca architektonicznego odwróconego sterowania, aby wstrzykiwać modele danych do przetwarzania logiki biznesowej aplikacji do konstruktorów kontrolerów. Biblioteka umożliwia również wstrzykiwanie repozytoriów do serwisów danych.
Entity Framework	— aplikacji użytkownika — logiki biznesowej — dostępu do danych	biblioteka umożliwiająca operacje na kolekcjach danych oraz tworzenie struktur bazodanowych poprzez zastosowanie kocepcji Code-First.
Microsoft Identity	— aplikacji użytkownika	Biblioteka zapewniająca aplikacji użytkownika zestaw metody umożliwiających autoryzacje oraz uwierzytelnienie. Komponenty zestawu ponadto rozbudowują bazę danych o dodatkowe tabele oraz kolumny przechowujace poufne dane.

Microsoft Windows Azure Storage	— logiki biznesowej	Biblioteka umożliwiająca komunikację z magazynem danych w chmurze Azure.
Newtonsoft Json.NET	— aplikacji użytkownika	Biblioteka umożliwiająca zamianę obiektu dowolnego typu na tekst w formacie JSON oraz danych w postaci JSON na modele używane w aplikacji.
jQuery	— aplikacji użytkownika	Biblioteka skryptowa operująca na komponentach graficznych oraz definiująca działanie interfejsu użytkownika w dowolnej przeglądarce internetowej z włączoną obsługą języka JavaScript.
Bootstrap Material Design	— testów	Biblioteka zawierająca kaskadowe arkusze stylów oraz skrypty implementuje wygląd i zachowanie komponentów zaprojektowanych przez wzorzec Material Design firmy Google.
PostSharp	— logiki biznesowej	Biblioteka umożliwiająca stworzenie własnych atrybutów dla klas oraz metod w języku C# w projektach biblioteki klas. Kod stworzony w ramach atrybutu w trakcie kompilacji zostanie wstrzyknięty w miejsce określone przy konfiguracji aspektu.
DotNetZip	— logiki biznesowej	Biblioteka umożliwiająca dynamiczne tworzenie plików zip w oparciu o pliki w postaci strumienia lub tablicy bajtów.
MOQ	— testów	Biblioteka umożliwiająca imitowanie struktur danych używanych w testach jednostkowych.
xUnit	— testów	Biblioteka umożliwiająca pisanie metod testujących funkcjonalności napisane w programie.

5.3. Konfiguracja

Aplikacja użytkownika zaimplementowana w ASP.NET MVC charakteryzuje się wykorzystaniem trzech wzorców architektonicznych - Model-Widok-Kontroler, Odwrotnego sterowania oraz wstrzykiwania zależności. Każda z klas pełniących funkcję kontrolerów wykorzystuje zasady zdefiniowane w każdym z wymienionych wzorców. W celu implementacji założeń wykorzystano bibliotekę Unity platformy .NET umożliwiającą wstrzykiwanie złożonych struktur danych - serwisów, do konstruktorów kontrolerów. Konfiguracja zakłada stworzenie kontenera danych oraz zdefiniowanie w nim klas serwisów oraz ich interfejsów, które mogą zostać wykorzystane w argumentach metody inicjalizującej klasę.

```
1  public class IoCConfiguration
2  {
3      public static void ConfigureIocUnityContainer()
4      {
5          IUnityContainer container = new UnityContainer();
6          RegisterServices(container);
7          DependencyResolver.SetResolver(new
8           WebDiskDependencyResolver(container));
9      }
10
11     private static void RegisterServices(IUnityContainer container)
12     {
13         container.RegisterType<ISpaceService, SpaceService>();
14         container.RegisterType<IDirectoryService, DirectoryService>();
15         container.RegisterType<IFieldService, FieldService>();
16     }
17 }
```

Listing 5.1: Konfiguracja kontenera Odwrotnego sterowania

Aplikacja w przypadku użycia biblioteki Automapper wymaga zdefiniowania możliwych ścieżek zamiany modeli. W związku z tym stworzono klasę konfigurującą dozwolone ścieżki o nazwie MapperConfig, która jest uruchamiana jednorazowo wraz ze startem aplikacji. Metoda konfigurującą przykładowe modele danych znajduje się w Listingu 5.2.

```
public static class MapperConfig
{
    public static void RegisterMaps()
    {
        AutoMapper.Mapper.Initialize(n =>
        {
            n.CreateMap<Field, FieldViewModel>();
            n.CreateMap<HttpPostedFileBase, FileViewModel>();

            n.CreateMap<FieldInformation, FieldInformation>()
            .ForMember(dest => dest.FieldInformationId,
                    opts => opts.MapFrom(from => Guid.NewGuid()))
            .ForMember(dest => dest.Field,
                      opts => opts.Ignore());

            .....
        }
    }
}
```

Listing 5.2: Konfiguracja biblioteki Automapper

Wszystkie funkcje konfigurujące aplikację ASP.NET MVC oraz te zdefiniowane dla używanych bibliotek wywoływane są w klasie Global.asax, w metodzie wywoływanej podczas uruchamiania aplikacji - Application_Start. Pojedyncze wykonanie zapisuje konfigurację na cały żywot istnienia aplikacji internetowej.

```

 public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        AreaRegistration.RegisterAllAreas();
        FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
        RouteConfig.RegisterRoutes(RouteTable.Routes);
        BundleConfig.RegisterBundles(BundleTable.Bundles);
        IoCConfiguration.ConfigureIocUnityContainer();
        MapperConfig.RegisterMaps();
    }
    ...
}
```

Listing 5.3: Konfiguracja aplikacji

5.4. Atrybuty

W celu łatwiejszego użytkowania wymienionych bibliotek, zostały stworzone pomocnicze atrybuty, umożliwiające w szybki sposób wykonanie funkcji bardzo często wykonywanych. Do listy nowo utworzonych atrybutów należy zaliczyć:

— **AutomapAttribute** - atrybut stworzony w celu zamiany zwracanego z kontrolera, modelu bazodanowego, na model widoku, zaraz po przetworzeniu logiki znajdującej się w akcji.

```
1  [AttributeUsage(AttributeTargets.Method, AllowMultiple = false)]
2  public class AutoMapAttribute : ActionFilterAttribute
3  {
4   private readonly Type _sourceType;
5   private readonly Type _destType;
6
7   public AutoMapAttribute(Type sourceType, Type destType)
8   {
9       _sourceType = sourceType;
10      _destType = destType;
11  }
12
13  public override void OnActionExecuted
14                          (ActionExecutedContext filterContext)
15  {
16      var filter = new AutoMapFilter(SourceType, DestType);
17
18      filter.OnActionExecuted(filterContext);
19  }
```

Listing 5.4: Kod atrybutu Automap

Atrybut może być wykonywany tylko na akcji znajdującej się w klasie pełniącej funkcję kontrolera. W widoku dla danej akcji, należy pamiętać o zadeklarowaniu modelu dla widoku, ponieważ w innym przypadku otrzymamy błąd niezgodności typów przy generowaniu strony. Dane zwracane z akcji powinny być typu określonego w atrybucie Automap.

```
1  [AutoMap(typeof(IEnumerable<Field>),
2           typeof(IEnumerable<FieldViewModel>))]
3  public ActionResult Index()
4  {
5   ....
6   return PartialView("_Directory", availableFields);
7  }
```

Listing 5.5: Wykorzystanie atrybutu AutoMap

— **AjaxActionAttribute** - atrybut umożliwiający dostęp do danej akcji tylko poprzez użycie asynchronicznego zapytania AJAX (Asynchronous JavaScript and XML). Użycie atrybutu spowoduje zablokowanie wykonania się logiki zawartej w akcji poprzez zapytanie inne niż AJAX.

```
1  public class AjaxActionAttribute : ActionMethodSelectorAttribute
2  {
3   public override bool IsValidForRequest(
4               ControllerContext controllerContext,
5               System.Reflection.MethodInfo methodInfo)
6   {
7       return controllerContext.RequestContext
8               .HttpContext.Request.IsAjaxRequest();
9   }
10 }
```

Listing 5.6: Kod atrybutu Automap

Atrybut ma za zadanie zablokować wysyłanie żądań pobrania danych bezpośrednio z przeglądarki użytkownika lub poprzez narzędzia takiej jak Postman lub Fiddler.

```
1  [AjaxAction]
2   public ActionResult Create(Guid rootId, string directoryName)
3  {
4       ...
5       return IndexDetails(rootId);
6  }
```

Listing 5.7: Wykorzystanie atrybutu AjaxAction

— **PermissionAttribute** - atrybut sprawdzający, czy aktualnie zalogowana osoba ma dostęp do wykonywania operacji na pliku lub katalogu. Atrybut używany jest w projekcie biblioteki klas w serwisach odpowiadających za logikę plików oraz katalogów. W celu jego implementacji zastosowano bibliotekę PostSharp, która umożliwia wprowadzanie atrybutów w klasach nie będących kontrolerami w aplikacjach ASP.NET MVC. Atrybut wymaga, aby w argumentach metody przyjmowane były zmienne typu Guid oznaczające identyfikator użytkownika oraz pola, na którym ma zostać wykonana operacja. W celu implementacji aspektu, nowo utworzona klasa musi być oznaczona atrybutem [Serializable]. Jest to jedno z wymagań biblioteki Postsharp. Kod znajdujący się w atrybucie wykonywany jest przed wywołaniem metody docelowej.

```
[Serializable]
public class Permission : MethodInterceptionAspect
{
 public override void OnInvoke(MethodInterceptionArgs args)
 {
     Guid userId = args.GetAttributeValue<Guid>("userId");
     Guid fieldId = args.GetAttributeValue<Guid>("fieldId");
     var serviceInstance = (ServiceBase)args.Instance;
     bool hasUserRights = serviceInstance
                          ._authManager
                          .IsUserHasRights(userId,fieldId);
     if (!hasUserRights)
     {
        throw new UnauthorizedAccessException("..");
     }
     base.OnInvoke(args);
 }
}
```

Listing 5.8: Aspekt potwierdzający uprawnienia zalogowanego użytkownik do wykonywania operacji na pliku lub katalogu

— **DataChangeAttribute** - atrybut zapisujące dane w metodach wykonujących zmiany na modelach bazodanowych, zaraz po wykonaniu się całej funkcji.Atrybut jest używany w bibliotece klas dzięki wykorzystaniu biblioteki Postsharp. Aspekt korzysta z metody zaimplementowanej w abstrakcyjnej klasie bazowej serwisu.

```

[Serializable]
public class DataChangeAttribute : OnMethodBoundaryAspect
{
 public override void OnExit(MethodExecutionArgs args)
 {
     ((ServiceBase)args.Instance).Save();
     base.OnExit(args);
 }
}
```

Listing 5.9: Aspekt zapisujący dane w bazie danych

5.5. Zaimplementowane metody w programie

— **Wykorzystanie wzorca odwrotnego sterowania oraz wstrzykiwania zależności -**

6. Testy, wdrożenia oraz scenariusze użycia

7. Zakończenie

Bibliografia

[1] Ksi??ka