

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	2
1.1. Wirtualny system plików	2
1.2. Cel i zakres pracy	2
2. System plików	3
2.1. Systemów plików FAT32	3
2.2. Systemów plików NTFS	4
2.3. Systemów plików SMB	4
2.4. Blob Azure Storage	5
3. Projekt wirtualnego systemu plików	6
3.1. System katalogów	6
3.2. Menadżer danych	6
3.3. Udostępniane funkcje	6
4. Architektura systemu	7
4.1. Podział projektu	7
4.2. Wzorce architektoniczne	7
4.3. Przepływ danych	10
4.4. Komunikacja z chmurą danych Azure	11
5. Implementacja	12
5.1. Środowisko programistyczne	12
5.2. Biblioteki	13
5.3. Konfiguracja	15
5.4. Atrybuty	17
5.5. Zaimplementowane metody w programie	20
6. Testy, wdrożenia oraz scenariusze użycia	30
7. Zakończenie	31
Bibliografia	32

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. System plików

Celem rozdziału jest przedstawienie istniejących systemów plików w systemach operacyjnych oraz protokołach udostępnienia zasobów. Zastosowane w nich rozwiązania stanowią podstawę do stworzenia alternatywy w postaci strony WWW.

2.1. Systemów plików FAT32

Następca FAT16 [2] wprowadzony do systemów operacyjnych Windows 95 w 1996r. Jego znaczenie podkreśla fakt, że mimo dynamicznego rozwoju informatyki oraz wielu alternatyw, stanowi on nieodłączny element najnowszych systemów operacyjnych (Windows 10). Jest to spowodowane kompatybilnością z ogromną liczbą urządzeń. FAT32 jest obsługiwany przez systemy operacyjne Windows, Linux oraz Mac OS. Jego wykorzystanie znajdziemy również w konsolach do gier takich jak Xbox lub Playstation oraz dyskach przenośnych USB.

System zakładał wykorzystanie 32 bitów na jeden klastery w tablicy alokacji. Umożliwia to zapis plików o maksymalnej wielkości 4GB danych oraz w teorii, możliwość opisu do 268 435 438 klastrów. Wynika z tego że FAT32 można użyć na 16 TB dyskach twardych o sektorach 512-bajtowych. Ze względu na konfigurację sektora uruchomieniowego, w którym zostało ograniczone pole rozmiaru partycji w sektorach, rozmiar ten nie może przekroczyć 2 TB danych dla 512-bajtowych sektorów. FAT32 nie pozwala na zapis plików o nazwie większej niż 255 znaków. Partycja w strukturze FAT składa się z 4 części:

- **Regionu zarezerwowanego**, w którego skład wchodzi część uruchomieniowa systemu operacyjnego oraz informacje o partycji takie jak: wielkość sektora oraz partycji, ilość dostępnych sektorów, czy typ partycji.
- **Tablicy alokacji FAT**, która jest przechowywana zaraz po regionie zarezerwowany. Zawiera ona informacje o relacjach klastrów w systemie operacyjnym. Na jego podstawie możemy zlokalizować plik lub katalog w regionie danych. Każdy element tablicy FAT odpowiada jednemu klastrowi. Na partycji zalecane jest przechowywanie kopii tablicy FAT. Możliwe jest posiadanie więcej niż jednej kopii.

- **Katalog główny**, który jest tworzony automatycznie wraz z uruchomieniem procedury tworzenia systemu plików.
- **Region danych**, w którym znajdują się pliki oraz katalogi użytkownika systemu operacyjnego. Dane te podzielone są na logiczne sektory nazywane klastrami.

FAT32 jest systemem plików podatnym na awarie oraz fragmentację. Do jego wad należy zaliczyć brak systemu uprawnień. Najnowsze systemy informatyczne często korzystają ze zbiorów danych, których wielkość przekracza dopuszczalne 4GB, co powoduje wybór systemu NTFS na rzecz FAT32.

2.2. Systemów plików NTFS

Nowe wymagania stawiane kolejnym wersjom systemów plików sprawiły, że możliwość spełnienia ich przez FAT32 nie była możliwa. W związku z tym stworzono zupełnie nowy system zarządzania plikami. Cele, które postawiono przed NTFS, dotyczyły minimalizacji utraty danych oraz zabezpieczenie ich przed nieautoryzowanym dostępem. Został on stworzony przez firmę Microsoft oraz wdrożony do systemów Windows począwszy od wersji NT 3.1 w 1993r.

NTFS zakłada składowanie wszystkich danych w postaci plików. Zasada ta dotyczy nie tylko ich zawartości, ale również indeksów, bitmap oraz metadanych. Najmniejszą jednostką logiczną na dysku jest klaster. Zastosowanie w systemie wirtualnych oraz logicznych numerów klastrów umożliwia wyznaczenie adresu fizycznego danych.

NTFS jest obiektowym systemem plików. Każdy obiekt odwzorowujący plik, składa się z atrybutów takich jak: nazwa, uprawnienia, dane, deskryptor bezpieczeństwa. Na szczególną uwagę zasługuje fakt, że dane

2.3. Systemów plików SMB

SMB[9] jest protokołem o architekturze klient-serwer używany do udostępnienia zasobów w postaci plików lub drukarek. Został on opracowany przez firmę IBM w latach 80. Następnie był rozwijany przez Microsoft, dzięki czemu stał się podstawowym elementem otoczenia sieciowego systemu Windows. Wykorzystuje on dwa inne protokoły niższych rzędów takie jak: NetBIOS w warstwie sesji oraz NetBEUI w warstwie sieci w modelu OSI [6].

2.4. Blob Azure Storage

Usługa chmury Microsoft Azure[10] umożliwia przechowywanie plików w postaci danych niestukturalnych. Magazyn, oferowany w usłudze, umożliwia zarządzanie danymi dowolnego typu takimi jak: dokumenty, pliki binarne lub multimedialne. Dane przechowywane w chmurze mogą być rozmiarów oraz typów nieokreślonych. Usługa charakteryzuje się możliwością dostępu do danych za pośrednictwem protokołu HTTP lub HTTPS. Dane przechowywane w chmurze mogą posłużyć do rozproszonego przetwarzania lub przechowywania ich jako kopii zapasowych. W przypadku internetowego systemu plików usługa może posłużyć jako miejsce fizycznego zapisu danych użytkownika aplikacji.

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 rozbiecie architektury na cztery moduły. Każdy z wyodrębnionych modułów ma za zadanie spełniać określoną czynność w architekturze systemu. Rozbiecie projektu pozwala na prostsze zarządzanie kodem. Zastosowanie warstw projektowych bardzo dobrze sprawdza się w przypadku, gdy programista dąży do rozłożenia odpowiedzialności komponentów w poszczególnych modułach. W związku z podziałem projektu na podmoduły można wyróżnić rozbudowaną architekturę systemu oraz widoczny zakres obowiązków poszczególnych podprojektów. Zbiór oraz opis wszystkich podprojektów znajduje się w tabeli 4.1.1s

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 zajmującej się przyjmowaniem 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 rozbudowaną 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 modeli jest niezbędna ze względu na bardzo dużą ilość danych zwracanych z serwisu, które nie zawsze są potrzebne przy tworzeniu 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.

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 kontrolująca przepływ 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 zawierająca modele bazodanowe potrzebne do stworzenia bazy danych oraz operacji na nich przy użyciu języka zapytań funkcyjnych.
Moduł testów	Projekt testów jednostkowych	Aplikacja zawierająca scenariusze testowe oraz testy sprawdzające poprawność działania kodu poprzez sprawdzanie oczekiwanego wyjścia z metod.

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 zmniejszają 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 liczbę funkcjonalności dzięki zastosowaniu wzorców architektonicznych "odwrócone sterowanie" oraz "wstrzykiwania zależności". Skorzystanie z nich umożliwia uzyskanie funkcjonalności, w której można swobodnie podłączać oraz odłączać kolejne moduły, wpływając na zwiększenie lub ograniczenie funkcjonalności w programie. Dobrą praktyką 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. Zabieg ten zmniejszy złożoność kodu oraz ułatwi jego późniejszą modyfikację bądź rozwój.

Szczególnie wartym uwagi wzorcem architektury jest "wstrzykiwanie zależności". Kontroler mógł zostać uzależniony od pewnego rodzaju klas przy wykorzystaniu biblioteki Unity dla języka C#. Polega ono na przesyłaniu do konstruktora kontrolera obiektów zdefiniowanych przez programistę w konfiguracji biblioteki. Bez mechanizmu Unity, byłoby to zadanie bardzo trudne do zrealizowania. Klasy pełniące funkcje kontrolerów w aplikacjach internetowych są wywoływane poprzez skierowanie żądania pod odpowiedni adres witryny. Czynność ta spowoduje automatyczne uruchomienie konstruktora kontrolera, co uniemożliwi przekazanie parametrów do niego, a jedynie do wywoływanej metody nazywanej akcją. Aby móc wstrzyknąć strukturę danych jako parametr metod inicjalizujących, należy przygotować interfejs określający jego funkcjonalność. Kolejnym krokiem jest powiadomienie biblioteki Unity w konfiguracji o możliwości "wstrzykiwania" podanego typu danych do kontrolera. Operacja w bardzo wielu miejscach zmniejsza złożoność kodu. To rozwiązanie dla programistów niezaznajomionych ze wzorcem Wstrzykiwania zależności może sprawić wiele problemów w przypadku dodawania funkcjonalności do kontrolera uzależnionego od pewnych struktur danych.

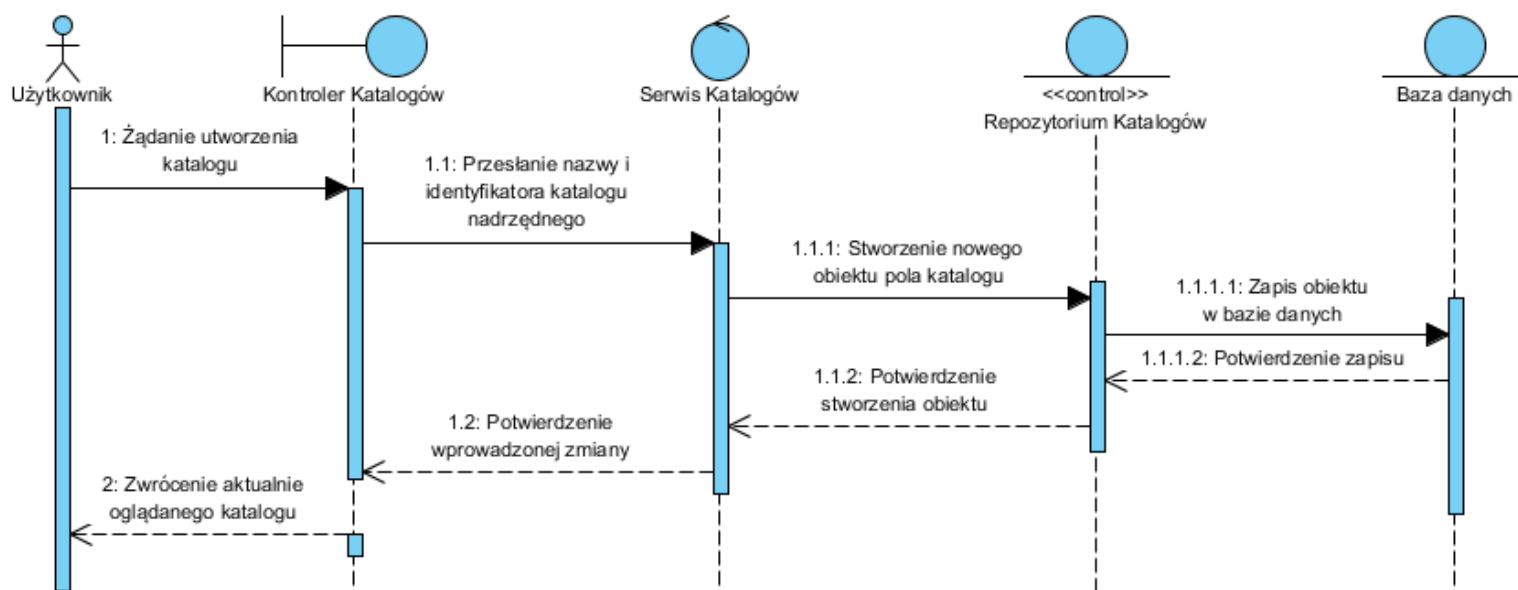
Ostatnim wzorcem architektonicznym użytym w projekcie jest Odwrócone sterowanie. Bardzo częstym błędem programistów, tworzących aplikacje inter-

netowe, jest pisanie zaawansowanej oraz złożonej logiki w klasach kontrolera. Kod napisany w akcji jest niemożliwy do ponownego użycia w innych miejscach aplikacji, ponieważ niezalecane jest tworzenie obiektów kontrolerów w innych kontrolerach. Ponadto kontroler pełni funkcję pośredniczącą pomiędzy logiką bazodanową a użytkownikiem. Tak więc tworzenie złożonych operacji na danych w akcjach w znacznym stopniu narusza dobre praktyki tworzenia 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 to 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 istnieje możliwość zapisu danych. Zarządzanie plikami na serwerze może powodować dużą liczbę problemów między innymi z dostępem do plików oraz katalogów na maszynach, na których aplikacja działałaby bez uprawnień administratorskich. Alternatywnym rozwiązaniem jest magazyn 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 stworzono przy pomocy języków SCSS, JavaScript oraz HTML przy użyciu ASP.NET MVC. Całość oprogramowania powstała z wykorzystaniem programu Visual Studio 2015 w wersji Community oraz systemu automatyzacji zadań gulp[3] w środowisku Node.js, służącemu do minifikacji oraz konkatencji stylów i skryptów. Aplikacja wymaga posiadania na komputerze 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[7] do przeglądarki internetowej Google Chrome lub pakiet Fiddler[8]. Narzędzia 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 można ją uruchomić przy pomocy programu Internet Information Service[4].

5.2. Biblioteki

W celu stworzenia funkcjonalności określonej w pracy użyto 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	<ul style="list-style-type: none">— aplikacji użytkownika— aplikacji biznesowej	Biblioteka umożliwiająca mapowanie pomiędzy obiektami odmiennego typu, dzięki której między innymi dokonywana jest zamiana modelu bazodanowego na model widoku.
Unity	<ul style="list-style-type: none">— 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	<ul style="list-style-type: none">— aplikacji użytkownika— logiki biznesowej— dostępu do danych	Biblioteka umożliwiająca operacje na kolekcjach danych oraz tworzenie struktur bazodanowych poprzez zastosowanie koncepcji utworzenia klas oraz na ich podstawie wygenerowaniu bazy danych.
Microsoft Identity	<ul style="list-style-type: none">— aplikacji użytkownika	Biblioteka zapewniająca aplikacji użytkownika zestaw metody umożliwiających autoryzację oraz uwierzytelnienie. Komponenty zestawu ponadto rozbudowują bazę danych o dodatkowe tabele oraz kolumny przechowujące poufne dane.

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 klas lub metod. Język C# nie zapewnia takiej możliwości poprzez zastosowanie dostępnych mechanizmów platformy .NET. Kod stworzony w ramach atrybutu w trakcie kompilacji zostanie dodany 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

Biblioteka AutoMapper wymaga zdefiniowania typów konwersji danych źródłowych na dane docelowe. W związku z tym stworzono metodę w klasie statycznej, zawierającą dozwolone ścieżki konwersji. Funkcja, uruchamiana jest jednorazowo wraz ze startem aplikacji. Konwersja przykładowych modeli danych znajduje się w Listingu 5.2.

```

1 public static class MapperConfig
2 {
3     public static void RegisterMaps()
4     {
5         AutoMapper.Mapper.Initialize(n =>
6         {
7             n.CreateMap<Field, FieldViewModel>();
8             n.CreateMap<HttpPostedFileBase, FileViewModel>();
9
10            n.CreateMap<FieldInformation, FieldInformation>()
11              .ForMember(dest => dest.FieldInformationId,
12                        opts => opts.MapFrom(from => Guid.NewGuid()))
13              .ForMember(dest => dest.Field,
14                        opts => opts.Ignore());
15
16            .....
17        }
18    }
19 }

```

Listing 5.2: Konfiguracja biblioteki Automapper

Wszystkie funkcje konfigurujące aplikację ASP.NET MVC wykonywane są w klasie Global.asax podczas uruchamiania aplikacji. Pojedyncze wykonanie zapisuje konfigurację na cały czas działania aplikacji internetowej.

```

1
2 public class MvcApplication : System.Web.HttpApplication
3 {
4     protected void Application_Start()
5     {
6         AreaRegistration.RegisterAllAreas();
7         FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
8         RouteConfig.RegisterRoutes(RouteTable.Routes);
9         BundleConfig.RegisterBundles(BundleTable.Bundles);
10        IoCConfiguration.ConfigureIocUnityContainer();
11        MapperConfig.RegisterMaps();
12    }
13    ...
14 }

```

Listing 5.3: Konfiguracja aplikacji

5.4. Atrybuty

W celu łatwiejszego użytkowania wymienionych bibliotek zostały stworzone pomocnicze atrybuty. Wykorzystanie ich zmniejsza złożoność kodu oraz przyspiesza pracę programisty. Do listy nowo utworzonych atrybutów należy zaliczyć:

- **AutomapAttribute** - służy on do konwersji modeli danych po przetworzeniu logiki znajdującej się w akcji. Umożliwia zamianę modelu bazodanowego na model widoku.

```
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 ten może być wykonywany tylko na akcji znajdującej się w klasie pełniącej rolę 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 akcji tylko poprzez użycie asynchronicznego zapytania AJAX (Asynchronous JavaScript and XML).

```
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

Ma on za zadanie zablokować wysyłanie żądań pobrania danych bezpośrednio z przeglądarki użytkownika lub poprzez narzędzia takie 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. Jest on używany w projekcie biblioteki klas w serwisach odpowiadających za logikę plików oraz katalogów. W celu jego implementacji zastosowano bibliotekę PostSharp. Wykorzystanie jej umożliwia wprowadzanie atrybutów w klasach nie będących kontrolerami w aplikacjach ASP.NET MVC. Atrybut wymaga, aby w argumentach metody przyjmowane były zmienne oznaczające identyfikator użytkownika oraz pola, na których 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.

```

1  [Serializable]
2  public class Permission : MethodInterceptionAspect
3  {
4      public override void OnInvoke(MethodInterceptionArgs args)
5      {
6          Guid userId = args.GetAttributeValue<Guid>("userId");
7          Guid fieldId = args.GetAttributeValue<Guid>("fieldId");
8          var serviceInstance = (ServiceBase)args.Instance;
9          bool hasUserRights = serviceInstance
10             . _authManager
11             . IsUserHasRights(userId, fieldId);
12         if (!hasUserRights)
13         {
14             throw new UnauthorizedAccessException("..");
15         }
16         base.OnInvoke(args);
17     }
18 }

```

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

- **DataChangeAttribute** - atrybut zapisujący dane w metodach wykonujących zmiany w bazie danych. Wywołanie go następuje po zakończeniu całej funkcji. Atrybut jest używany w bibliotece klas dzięki wykorzystaniu biblioteki Postsharp. Aspekt korzysta z metody zaimplementowanej w abstrakcyjnej klasie bazowej serwisu.

```

1
2  [Serializable]
3  public class DataChangeAttribute : OnMethodBoundaryAspect
4  {
5      public override void OnExit(MethodExecutionArgs args)
6      {
7          ((ServiceBase)args.Instance).Save();
8          base.OnExit(args);
9      }
10 }

```

Listing 5.9: Aspekt zapisujący dane w bazie danych

5.5. Zaimplementowane metody w programie

Niniejszy rozdział ma na celu przedstawienie sposobu implementacji funkcjonalności uwzględnionych w wymaganiach programu. Omawiane funkcje pokazują sposób komunikacji poszczególnych modułów aplikacji.

- **Wykorzystanie wzorca odwrotnego sterowania oraz wstrzykiwania zależności** - wstrzykiwanie serwisów danych do repozytorium odbywa się przy użyciu biblioteki Unity. Konstruktor kontrolera przyjmuje jako argumenty serwisy, od których dana klasa jest zależna. Długość życia wstrzykniętych modeli danych jest określona w konfiguracji biblioteki. Dane przyjęte jako argumenty konstruktora są zapisywane w zmiennych prywatnych kontrolera, a następnie wykorzystywane w poszczególnych akcjach.

```
1 [Authorize]
2 [RoutePrefix("Field")]
3 public class FieldController : Controller
4 {
5     private readonly DirectoryService _directoryService;
6     private readonly FieldService _fieldService;
7     public FieldController(DirectoryService directoryService,
8                           FieldService fieldService)
9     {
10         _directoryService = directoryService;
11         _fieldService = fieldService;
12     }
13 ... }
```

Listing 5.10: Wstrzyknięcie serwisów danych do konstruktora kontrolera

- **Repozytorium dostępu do danych** - klasa odpowiedzialna za wykonywanie operacji na obiekcie bazy danych. Charakteryzuje się ona wykorzystaniem mechanizmu generyczności w celu zastosowania jej do operacji na dowolnej tabeli. Repozytorium implementuje wszystkie podstawowe operacje bazodanowe, określone w interfejsie IRepository.

```
1 public interface IRepository<TEntity>
2 {
3     IEnumerable<TEntity> Get(...);
4     TEntity GetByID(object id);
5     void Insert(TEntity entity);
6     void Delete(object id);
7     void Delete(TEntity entityToDelete);
8     void Update(TEntity entityToUpdate);
9 }
```

Listing 5.11: Interfejs repozytorium dostępu do danych

Implementacja repozytorium zawiera metody wykorzystujące kontekst bazodanowy w celu operacji na danych. Dodanie dodatkowych funkcji do klasy wiąże się z udostępnieniem ich dla wszystkich repozytoriów danych. Klasa posiada dwa konstruktory - parametrowy i bezparametrowy. Wersja bezparametrowa używana jest na potrzeby testów jednostkowych, zaś parametrowa wykorzystywana jest przez serwisy logiki biznesowej.

```
1 public virtual void Delete(object id)
2 {
3     TEntity entityToDelete = dbSet.Find(id);
4     Delete(entityToDelete);
5 }
6 public virtual void Delete(TEntity entityToDelete)
7 {
8     if (_context.Entry(entityToDelete).State == EntityState.Detached)
9     {
10         dbSet.Attach(entityToDelete);
11     }
12     dbSet.Remove(entityToDelete);
13 }
```

Listing 5.12: Metoda usuwania zaimplementowana w repozytorium danych

W celu odseparowania metod zbędnych dla niektórych repozytoriów zdecydowano się wykorzystać metody rozszerzeń w języku C#

```
1 public static class DirectoryExtensions
2 {
3     public static IEnumerable<Field> GetFields(
4         this Repository<Field> source,
5         Guid directoryId)
6     {
7         return source.Get(n => n.ParentDirectoryId == directoryId);
8     }
9     public static Field GetFieldRoot(this Repository<Field> source,
10         Guid fieldId)
11     {
12         Field root = source.GetByID(fieldId);
13         if (root == null)
14         {
15             throw new ArgumentException("There is no root folder");
16         }
17         return root.ParentDirectoryId.HasValue
18             ? GetFieldRoot(source, root.ParentDirectoryId.Value)
19             : root;
20     }
21 }
```

Listing 5.13: Klasa rozszerzająca metody możliwe do wykonania na repozytorium Katalogów

- **Pobranie plików oraz katalogów użytkownika** - akcja odpowiedzialna za stworzenie strony przedstawiającej aktualny stan katalogu, w którym znajduje się użytkownik. Na stronie HTML wygenerowane zostaną ikony przedstawiające pliki i podfoldery. Akcja pobrania plików oraz katalogów jest metodą przeciążoną, możliwą do wykonania poprzez dwa różne adresy URL. Bezparametrowa wersja wygeneruje rezultat dla katalogu domowego użytkownika. Akcja z parametrem stworzy widok zawierający stan dla katalogu o podanym identyfikatorze.

```
1 [HttpGet]
2 [Route("")]
3 [AutoMap(typeof(IEnumerable<Field>),
4           typeof(IEnumerable<FieldViewModel>))]
5 public ActionResult Index()
6 {
7     var userId = Identity.GetUserId(User.Identity);
8     ViewBag.DirectoryId = _directoryService.GetRootField(userId)
9                                           .FieldId;
10    return PartialView("_Directory", _directoryService
11                                .GetAvailableFields(userId));
12 }
13 [HttpGet]
14 [Route("{directoryId}")]
15 [AutoMap(typeof(IEnumerable<Field>),
16           typeof(IEnumerable<FieldViewModel>))]
17 public ActionResult IndexDetails(Guid directoryId)
18 {
19     ...
20    return PartialView("_Directory",
21                        _directoryService
22                          .GetAvailableFields(userId, directoryId));
23 }
```

Listing 5.14: Akcje odpowiedzialne za wyświetlenie plików i katalogów na interfejsie użytkownika

Kontroler w akcji Index wywołuje metodę serwisu odpowiedzialną za sterowanie logiką biznesową związaną z zarządzaniem katalogami. Wywoływana funkcja jest funkcją przeciążoną i w przypadku wywołania jej bez identyfikatora folderu, zostaną pobrane dane dla katalogu domowego użytkownika. W innym przypadku zwracane dane zależne są od identyfikatora katalogu podanego jako argument. Metoda bezparametrowa wywołuje wersję z parametrem po otrzymaniu identyfikatora folderu domowego użytkownika.

```

1 public IEnumerable<Field> GetAvailableFields(Guid userId)
2 {
3
4     var defaultDirectoryId = SpaceRepository
5                                     .GetDefaultSpaceDirectory(userId)
6                                     .FieldId;
7
8     return GetAvailableFields(userId, defaultDirectoryId);
9 }
10
11 [Permission]
12 public IEnumerable<Field> GetAvailableFields(Guid userId,
13                                             Guid fieldId)
14 {
15     return FieldRepository.GetFields(fieldId);
16 }

```

Listing 5.15: Akcje odpowiedzialne za wyświetlenie plików i katalogów na interfejsie użytkownika

W celu pobrania danych wykonywana jest operacja na metodzie rozszerzającej repozytorium plików. Funkcja ta odpowiedzialna jest za pobranie wszystkich dzieci (plików i katalogów podrzędnych) dla katalogu podanego jako argument funkcji. Pobrane dane są wynikiem wykonania metody. Na ich podstawie realizowana jest zamiana na model widoku przy pomocy atrybutu Automap.

- **Dodawanie plików** - Internetowy system plików umożliwia zapisywanie danych w aplikacji poprzez skorzystanie z menu kontekstowego strony www. Wybranie opcji "Dodaj" uruchomi dialog wyszukiwania plików na dysku twardego użytkownika. Funkcjonalność umożliwia dodawania wielu plików naraz. Przesłanie ich w sposób asynchroniczny odbywa się przy użyciu biblioteki jQuery oraz operacji AJAX (Asynchronous JavaScript and XML). W celu wysłania danych do serwera należy przygotować obiekt symulujący rolę formularza w języku JavaScript. Elementami, które wchodzi w skład obiektu, są pliki wybrane przez użytkownika w menu dialogowym. Wszystkie informacje o nich znajdują się w ukrytej kontrolce na stronie HTML. Sposób tworzenia obiektu formularza w funkcji języka JavaScript znajduje się w listingu 12.

```

1 var formData = new FormData(),
2 directoryId = getCurrentDirectoryId(),
3 fileLength = document.getElementById("file").files.length;
4 if (!directoryId) {
5     displayToast("Napotkano na problemy. Odśwież stronę",
6         toastType.ERROR);
7 }
8 for (var i = 0; i < fileLength; i++) {
9     formData.append("files[" + i + "]",
10         document.getElementById("file").files[i]);
11 }

```

Listing 5.16: Tworzenie obiektu formularza z plikami wybranymi przez użytkownika

Stworzony formularz jest wykorzystywany w zapytaniu AJAX wysyłającym pliki użytkownika do serwera. Dane wysyłane są na adres URL zależny od identyfikatora katalogu, w którym aktualnie znajduje się użytkownik. Przesłanie plików wiąże się ze stworzeniem zapytania HTTP typu POST. W celu aktualizacji widoku oraz ewentualnym powiadomieniu użytkownika o błędzie w trakcie przetwarzania żądania, stworzone zostały dwie funkcje anonimowe.

```

1 $.ajax({
2     url: 'Field/Update/' + directoryId,
3     type: 'POST',
4     data: formData,
5     success: function (data) {
6         displayToast("Dodano pliki", toastType.SUCCESS);
7         $("#fields").html(data);
8     },
9     error: function (data) {
10        displayToast("Nie dodano danych", toastType.ERROR);
11    },
12    contentType: false,
13    processData: false
14 });

```

Listing 5.17: Tworzenie obiektu formularza z plikami wybranymi przez użytkownika

Dane wybrane przez użytkownika zostają odebrane w akcji Tworzenia plików w kontrolerze. Argumentem wywoływanej funkcji jest kolekcja danych w formacie `HttpPostedFileBase` oraz identyfikator katalogu, w którym aktualnie znajduje się użytkownik. Zastosowanie własnej ścieżki wywołania umożliwiło, na podstawie adresu URL, otrzymanie identyfikatora w argumencie funkcji.

```

1 [HttpPost]
2 [Route("Update/{directoryId}")]
3 public ActionResult Create(IEnumerable<HttpPostedFileBase> files ,
4                             Guid directoryId)
5 {
6     var fileModel = Mapper.Map<IEnumerable<FileViewModel>>(files);
7     Guid userId = Identity.GetUserId(User.Identity);
8     _fieldService.CreateField(userId, directoryId, fileModel);
9     return RedirectToAction("IndexDetails", "Directory",
10                             new { directoryId });
11 }

```

Listing 5.18: Akcja otrzymująca plik użytkownika

Dane plików zostają przetworzone przy pomocy biblioteki AutoMapper. Operacja ta umożliwi interpretację obiektu przez bibliotekę logiki biznesowej, w której zdefiniowany jest model pośredni. Dane oraz informacje o identyfikatorach aktualnie zalogowanego użytkownika i bieżącego katalogu służą jako argumenty metody zapisu plików w logice biznesowej aplikacji.

```

1 public void CreateField(Guid userId ,
2                         Guid fieldId ,
3                         IEnumerable<FileViewModel> fileViewModels)
4 {
5     foreach (var fileViewModel in fileViewModels)
6     {
7         CreateField(userId, fieldId, fileViewModel);
8     }
9 }
10 [DataChangeAttribute]
11 [Permission]
12 public void CreateField(Guid userId ,
13                         Guid fieldId ,
14                         FileViewModel fileViewModel)
15 {
16     var field = AutoMapper.Mapper.Map<Field>(fileViewModel);
17     var fileStream = fileViewModel.InputStream;
18     FieldRepository
19         .CreateField(userId, fieldId, field, fileStream);
20 }

```

Listing 5.19: Przetwarzanie pliku w bibliotece logiki biznesowej

Każdy element modelu pośredniego znajdujący się w argumencie metody, wykorzystywany jest w operacji zapisu. Wykonywana operacja zabezpieczona jest przed nieuprawnionym dostępem do katalogów poprzez wykorzystanie atrybutu uprawnień. Do bazy danych dodane zostaną rekordy plików dzięki skorzystaniu z atrybutu zapisu danych. Operacja dodania odbywa się w metodzie rozszerzeń repozytorium plików. W jej ciele wykonywane jest przesyłanie danych do chmury Azure przy pomocy klasy pomocniczej. Pomyślne odebranie informacji w kontenerze plików umożliwi utworzenie adresu URL przechowywanego w bazie danych aplikacji, który pozwoli na pobranie pliku. Rezultatem wykonanej operacji dodawania jest zapis danych do bazy danych.

```
1 public static void CreateField(this Repository<Field> fieldRepository ,
2                               Guid userId ,
3                               Guid parentDirectoryId ,
4                               Field field ,
5                               Stream inputStream)
6 {
7     string pathToAzureFile = new AzureManager()
8     .UploadFile(inputStream);
9
10    field.FieldInformation.Localisation = pathToAzureFile;
11    field.ParentDirectoryId = parentDirectoryId;
12    field.LastModifiedById = userId;
13
14    fieldRepository.Insert(field);
15 }
```

Listing 5.20: Zapis danych w chmurze oraz bazie danych

Zapis plików w chmurze Azure wykonywany jest przy wykorzystaniu bibliotek WindowsAzure.Storage. Dane do magazynu danych zdefiniowane są w portalu. Informacje te należy użyć do stworzenia bezpiecznego połączenia pomiędzy aplikacją kliencką, a serwerem.

```
1 private readonly CloudBlobClient _blobClient;
2 public AzureManager()
3 {
4     CloudStorageAccount storageAccount =
5     CloudStorageAccount.Parse(AzureKeys.CloudStorageAccountConn3String);
6     _blobClient = storageAccount.CreateCloudBlobClient();
7     CloudBlobContainer container = _blobClient
8     .GetContainerReference(AzureKeys.ContainerName);
9     ...
10 }
```

Listing 5.21: Konfiguracja połączenia z chmurą Azure

W celu przesłania danych należy połączyć się z kontem magazynu. Dzięki niemu możemy wybrać kontener danych, w którym znajdują się pliki użytkownika.

```
1 public string UploadFile(Stream content)
2 {
3     CloudBlobContainer container =
4         _blobClient.GetContainerReference(AzureKeys.ContainerName);
5     var blobId = $"{Guid.NewGuid()} {Guid.NewGuid()}";
6     CloudBlockBlob blockBlob = container
7         .GetBlockBlobReference(blobId);
8     blockBlob.UploadFromStream(content);
9     return blobId;
10 }
```

Listing 5.22: Przesłanie plików do chmury Azure

- **Pobieranie danych** - Użytkownik korzystając z menu aplikacji może pobrać pliki oraz katalogi. Opcja pobierz kieruje żądanie do serwera pod adres URL zależny od typu pobieranych danych. Operacja pobrania danych z serwera wykonywana jest w sposób asynchroniczny. Żądanie odebrane w kontrolerze, przekierowuje identyfikator użytkownika oraz pola, na którym ma wykonywana jest operacja. Biblioteka logiki biznesowej wykonuje metodę pobrania danych na obiekcie pola. Ze względu na funkcję obiekt bazodanowego, wbrew dobrym praktykom jest tworzenie dodatkowych metod wewnątrz tej klasy. W związku z tym dla metod wykonywanych na tego typu obiektach wykorzystano metody rozszerzeń. Funkcja pobrania, sprawdza jakiego rodzaju dane użytkownik zamierza pobrać. W przypadku pobrania pliku metoda przekazuje jako rezultat funkcji, dane znajdujące się w chmurze Azure pod adresem zachowanym w bazie danych.

```
1 private static FileViewModel DownloadFile(this Field source)
2 {
3     if (source.Type != FieldType.File)
4     {
5         throw new InvalidOperationException();
6     }
7     var file = AzureManager
8         .Download(source.FieldInformation.Localisation);
9     var fileStream = ByteHelper.ByteArrayToStream(file);
10    var result = AutoMapper.Mapper.Map<FileViewModel>(source);
11    result.InputStream = fileStream;
12    return result;
13 }
```

Listing 5.23: Pobranie pliku z chmury Azure

Dane z chmury Azure pobierane są za pomocą klasy pomocniczej. Jej konstruktor znajduje się w listingu 5.21. Metoda jako argument przyjmuje identyfikator w postaci napisu. Na jego podstawie określone jest położenie oraz pobierana jest zawartość pliku. Dane z kontenera Azure możemy wczytać przy pomocy strumienia danych. Wynikiem wykonania funkcji jest tablica bajtów wczytanych z kontenera chmury.

```
1 public byte[] Download(string blobId)
2 {
3     CloudBlobContainer container =
4         _blobClient.GetContainerReference(AzureKeys.ContainerName);
5
6     CloudBlockBlob blockBlob = container.GetBlockBlobReference(blobId);
7
8     using (var memoryStream = new MemoryStream())
9     {
10         blockBlob.DownloadToStream(memoryStream);
11         return ByteHelper.ReadToEnd(memoryStream);
12     }
13 }
```

Listing 5.24: Metoda pobierania danych z chmury Azure

Chęć pobrania katalogu wiąże się z pobraniem wszystkich plików znajdujących się w nim oraz spakowaniu danych do formatu .zip. W związku z tym zastosowano bibliotekę DotNetZip oraz funkcję pobrania danych z chmury Azure. Wszystkie pliki, znajdujące się w folderze umieszczane są w dynamicznie stworzonym archiwum danych. Napotkanie katalogu wymusza stworzenie podfolderów w utworzonym archiwum oraz umieszczenie w nim plików znajdujących się w tym katalogu.

```
1 private static FileViewModel DownloadDirectory(this Field source)
2 {
3     var outputStream = new MemoryStream();
4     using (var zip = new ZipFile())
5     {
6         ZipDirectory(source, zip, string.Empty);
7         zip.Save(outputStream);
8     }
9
10    outputStream.Position = 0;
11    var result = AutoMapper.Mapper.Map<FileViewModel>(source);
12    result.InputStream = outputStream;
13    result.FileName = result.FileName + ".zip";
14    return result;
15 }
```

Listing 5.25: Pobranie folderu z danymi

Strumień danych oraz informację o pliku przekazywane są do kontrolera po-
przez model pośredni, a następnie zwracane jako rezultat.

— **Kopiowanie danych** -

6. Testy, wdrożenia oraz scenariusze użycia

7. Zakończenie

Bibliografia

- [1] Chmura Microsoft Azure
<https://azure.microsoft.com/pl-pl/overview/what-is-azure/>
- [2] FAT16:
https://en.wikipedia.org/wiki/File_Allocation_Table#FAT16
- [3] Gulp jako system automatyzacji zadań:
<https://www.npmjs.com/package/gulp>
- [4] Internet Information Services:
<https://www.iis.net/>
- [5] Ograniczenia systemu pliku FAT32:
<https://support.microsoft.com/pl-pl/help/184006/limitations-of-fat32-file-system>
- [6] Model ISO OSI
<http://www.man.poznan.pl/~pawelw/dyplom/layers.html>
- [7] Program Postman
<https://www.getpostman.com/docs/>
- [8] Program Fiddler
<http://www.telerik.com/fiddler>
- [9] Protokół Server Message Block (SMB):
https://en.wikipedia.org/wiki/Server_Message_Block
- [10] Zarządzanie danymi w chmurze *Microsoft Azure*:
<https://docs.microsoft.com/pl-pl/azure/storage/storage-dotnet-how-to-use-blobs>