O'REILLY®

C# 8.0 w pigułce

Joseph Albahari
Eric Johannsen

Helion

Tytuł oryginału: C# 8.0 in a Nutshell: The Definitive Reference

Tłumaczenie: Łukasz Piwko
z wykorzystaniem fragmentów książki C# 6.0 w pigułce
w przekładzie Roberta Górczyńskiego i Jakuba Hubisza

ISBN: 978-83-283-7281-8

Authorized Polish translation of the English edition of C# 8.0 in a Nutshell
ISBN 9781492051138 © 2020 Joseph Albahari and Eric Johannsen

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/c8wpig
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/c8wpig.zip

Printed in Poland.

Spis treści

18

Zestawy

Zestaw to podstawowa jednostka wdrożeniowa na platformie .NET Core i jednocześnie kontener dla wszystkich typów. Zestaw zawiera skompilowane typy wraz z ich kodem IL (ang. *Intermediate Language*), zasobami środowiska uruchomieniowego oraz informacjami pomagającymi w wersjonowaniu, zapewnianiu bezpieczeństwa i odwoływaniu się do innych zestawów. Ponadto zestaw definiuje granice dla ustalania typu oraz uprawnień. W .NET Core zestaw stanowi pojedynczy plik z rozszerzeniem *.dll*.

Podczas budowy aplikacji na platformie .NET Core powstają dwa pliki: zestaw (*.dll*) i moduł rozruchowy (*.exe*) odpowiedni dla wybranej platformy.

Na platformie .NET Framework wygląda to inaczej, ponieważ generowany jest zestaw PE (ang. *portable executable*). Ma on rozszerzenie *.exe* i pełni funkcję zarazem zestawu i modułu rozruchowego. Jeden PE może być przystosowany do uruchamiania zarówno w 32-, jak i 64-bitowych wersjach systemu Windows.

Ponadto .NET Core umożliwia odwoływanie się do bibliotek WinRT, które mają rozszerzenie *.winmd*. Pod względem strukturalnym są podobne do zestawów, ale nie zawierają kodu IL, lecz jedynie metadane.

Większość typów przedstawionych w tym rozdziale pochodzi z poniższych przestrzeni nazw:

```
System.Reflection
System.Resources
System.Globalization
```

Co znajduje się w zestawie?

Zestaw zawiera cztery wymienione poniżej rodzaje elementów:

Manifest zestawu

Dostarcza informacji CLR, takich jak: nazwa zestawu, jego wersja oraz inne zestawy, do których się odwołuje.

Manifest aplikacji

Dostarcza systemowi operacyjnemu informacje takie jak sposób wdrożenia zestawu oraz wskazuje, czy wymagane są uprawnienia administratora.

Skompilowane typy

Skompilowany kod IL oraz metadane typów zdefiniowanych w zestawie.

Zasoby

Inne dane osadzone wewnątrz zestawu, takie jak obrazy oraz lokalizowane ciągi tekstowe.

Z wymienionych powyżej tylko *manifest zestawu* jest obowiązkowy, choć zestaw niemal zawsze zawiera skompilowane typy (jeżeli nie jest zestawem zasobów. Zobacz podrozdział „Zasoby i zestawy satelickie").

Manifest zestawu

Manifest zestawu służy do dwóch celów:

- opis zestawu w zarządzanym środowisku hostingu;
- działanie w charakterze katalogu dla modułów, typów i zasobów w zestawie.

Dlatego też zestawy są *samoopisujące się*. Konsument może ustalić wszystkie dane, typy i funkcje zestawu bez konieczności użycia jakichkolwiek dodatkowych plików.

Manifest zestawu to nie jest element, który wyraźnie dodajesz do zestawu. Zamiast tego jest osadzany w zestawie podczas kompilacji.

Poniżej przedstawiono podsumowanie funkcjonalnie ważnych danych przechowywanych w manifeście:

- prosta nazwa zestawu;
- numer wersji (`AssemblyVersion`);
- klucz publiczny i podpisana wartość hash zestawu, jeśli jest ściśle typowany;
- lista zestawów, do których się odwołuje, wraz z ich wersjami i kluczami publicznymi;
- lista typów zdefiniowanych w zestawie;
- kultura docelowa w przypadku zestawu satelickiego (`AssemblyCulture`).

Manifest może również zawierać wymienione poniżej dane informacyjne:

- pełny tytuł oraz opis (`AssemblyTitle` i `AssemblyDescription`);
- informacje o firmie i o prawach autorskich (`AssemblyCompany` i `AssemblyCopyright`);
- wyświetlana wersja (`AssemblyInformationVersion`);
- dodatkowe atrybuty dla niestandardowych danych.

Część danych wywodzi się z argumentów przekazywanych kompilatorowi (przykładami mogą być lista zestawów, do których się odwołuje, lub klucz publiczny użyty do podpisania danego zestawu). Pozostałe informacje pochodzą z atrybutów zestawu podanych w nawiasach.

Zawartość manifestu zestawu można wyświetlić za pomocą oferowanego przez .NET narzędzia o nazwie *ildasm.exe*. W rozdziale 19. dowiesz się, jak wykorzystać refleksję do wykonania tego samego zadania, ale w sposób programowy.

Określanie atrybutów zestawu

Często używane atrybuty zestawu można zdefiniować w Visual Studio na stronie właściwości projektu na karcie *Package* (pakiet). Ustawienia z tej karty są dodawane do pliku projektu (*.csproj*).

Jeśli chcesz użyć atrybutów niedostępnych na karcie *Package* lub nie używasz pliku *.csproj*, to możesz je zdefiniować w kodzie źródłowym. Projekty platformy .NET Framework automatycznie tworzą do tego celu plik o nazwie *AssemblyInfo.cs* w folderze *Properties*. Natomiast projekty na platformie .NET Core tego nie robią. Choć atrybuty można definiować w dowolnym pliku z kodem źródłowym należącym do projektu, dla porządku lepiej jest utworzyć specjalny plik *.cs* do ich przechowywania.

Plik przeznaczony do przechowywania atrybutów zawiera tylko instrukcje `using` i deklaracje atrybutów zestawów. Poniżej na przykład udostępniamy typy o wewnętrznym zakresie dostępności projektowi testów jednostkowych:

```
using System.Runtime.CompilerServices;
[assembly:InternalsVisibleTo("MyUnitTestProject")]
```

Manifest aplikacji (Windows)

Manifest aplikacji to plik w formacie XML dostarczający systemowi operacyjnemu informacje o zestawie. Manifest aplikacji zostaje umieszczony w module rozruchowym jako zasób Win32 w trakcie kompilacji. Jeśli istnieje, jest wczytywany i przetwarzany, zanim CLR wczyta zestaw, oraz może mieć wpływ na sposób uruchamiania aplikacji przez system Windows.

Manifest aplikacji .NET ma element główny o nazwie `assembly` zdefiniowany w przestrzeni nazw XML `urn:schemas-microsoft-com:asm.v1`:

```
<?xml version="1.0" encoding="utf-8"?>
<assembly manifestVersion="1.0" xmlns="urn:schemas-microsoft-com:asm.v1">
  <!-- zawartość manifestu -->
</assembly>
```

Przedstawiony poniżej manifest informuje system operacyjny o konieczności żądania uprawnień administratora:

```
<?xml version="1.0" encoding="utf-8"?>
<assembly manifestVersion="1.0" xmlns="urn:schemas-microsoft-com:asm.v1">
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
    <security>
      <requestedPrivileges>
        <requestedExecutionLevel level="requireAdministrator" />
      </requestedPrivileges>
    </security>
  </trustInfo>
</assembly>
```

Konsekwencjami żądania uprawnień administratora dokładnie zajmiemy się w rozdziale 21.

Aplikacje UWP do umieszczenia w sklepie Windows Store mają znacznie bardziej rozbudowany manifest zapisany w pliku *Package.appxmanifest*. Zawiera on m.in. deklaracje możliwości programu, które mają wpływ na ustalenie uprawnień przydzielanych przez system operacyjny. Edycję tego pliku najłatwiej przeprowadzić w środowisku Visual Studio, które wyświetla odpowiedni interfejs użytkownika po dwukrotnym kliknięciu pliku manifestu.

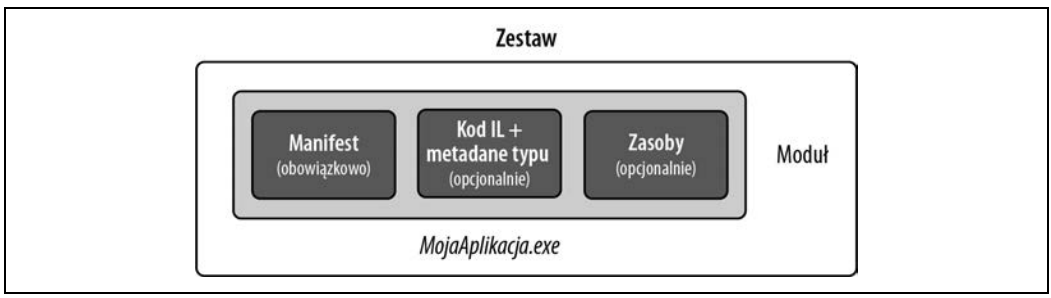
Wdrożenie manifestu aplikacji .NET

Do projektu .NET Core manifest aplikacji można dodać w Visual Studio. Należy kliknąć prawym przyciskiem myszy projekt w oknie *Solution Explorer* (eksplorator rozwiązań), wybrać pozycję *Add* (dodaj), następnie kliknąć opcję *New item* (nowy element) i wybrać *Application Manifest File* (plik manifestu aplikacji). W trakcie kompilacji manifest zostanie osadzony w zestawie wynikowym.

Narzędzie .NET *ildasm.exe* nie wykrywa obecności osadzonego manifestu aplikacji. Jednak po dwukrotnym kliknięciu zestawu w oknie *Eksploratora rozwiązań* Visual Studio wskazuje, czy istnieje osadzony manifest aplikacji.

Moduły

Zawartość zestawu jest tak naprawdę umieszczona wewnątrz kontenera pośredniego, zwanego **modułem**. Moduł odpowiada plikowi zawierającemu treść zestawu. Powodem zastosowania tej dodatkowej warstwy jest możliwość przechowywania zestawu w wielu plikach, co jest możliwe na platformie .NET Framework, ale zabronione na platformie .NET Core. Relację tę przedstawia rysunek 18.1.



Rysunek 18.1. Zestaw przechowywany w pojedynczym pliku

Mimo że .NET Core nie pozwala na tworzenie zestawów przechowywanych w wielu plikach, należy mieć świadomość istnienia dodatkowego poziomu kontenerów związanego z modułami. Najczęściej zdarza się tak w przypadku refleksji (zob. „Refleksje dla zestawów" oraz „Emitowanie zestawów i typów" w rozdziale 19.).

Klasa Assembly

Klasa `Assembly` zdefiniowana w przestrzeni nazw `System.Reflection` jest bramą pozwalającą na uzyskanie dostępu do metadanych zestawu w trakcie działania aplikacji. Istnieje wiele sposobów na uzyskanie dostępu do obiektu zestawu, a najprostszy z nich polega na użyciu właściwości `Type` egzemplarza klasy `Assembly`:

```
Assembly a = typeof (Program).Assembly;
```

Obiekt `Assembly` można również pobrać przez wywołanie jednej z metod statycznych klasy `Assembly`:

`GetExecutingAssembly()`
: Metoda zwraca zestaw typu, który definiuje aktualnie wykonywaną funkcję.

`GetCallingAssembly()`
: Metoda działa tak samo jak `GetExecutingAssembly()`, ale dotyczy funkcji, która wywołała aktualnie wykonywaną funkcję.

`GetEntryAssembly()`
: Metoda zwraca zestaw definiujący pierwotną metodę głównego punktu wejścia aplikacji.

Po utworzeniu egzemplarza klasy `Assembly` jego właściwości i metody można wykorzystać do sprawdzenia metadanych zestawu i odzwierciedlić ich typy. W tabeli 18.1 przedstawiono podsumowanie tych funkcji.

Tabela 18.1. Elementy składowe klasy Assembly

Funkcje	Przeznaczenie	Przejdź do…
`FullName()`, `GetName()`	Zwraca w pełni kwalifikowaną nazwę lub obiekt `AssemblyName`	„Nazwy zestawów” w dalszej części rozdziału
`CodeBase()`, `Location()`	Wskazuje położenie pliku zestawu	„Ładowanie, znajdowanie i izolowanie zestawów” w dalszej części rozdziału
`Load()`, `LoadFrom()`, `LoadFile()`	Pozwala na ręczne wczytanie zestawu do pamięci	„Ładowanie, znajdowanie i izolowanie zestawów” w dalszej części rozdziału
`GetSatelliteAssembly()`	Odszukuje zestaw satelicki w danej kulturze	„Zasoby i zestawy satelickie” w dalszej części rozdziału
`GetType()`, `GetTypes()`	Zwraca typ lub wszystkie typy zdefiniowane w zestawie	„Refleksje i aktywacja typów” w rozdziale 19.
`EntryPoint()`	Zwraca metodę aplikacji, będącą głównym punktem wejścia, jako `MethodInfo`	„Refleksje i wywoływanie składowych” w rozdziale 19.
`GetModule()`, `GetModules()`, `ManifestModule()`	Zwraca wszystkie moduły lub moduł główny zestawu	„Refleksje dla zestawów” w rozdziale 19.
`GetCustomAttribute()`, `GetCustomAttributes()`	Zwraca atrybuty zestawu	„Praca z atrybutami” w rozdziale 19.

Silne nazwy i podpisywanie zestawu

Zestaw o **silnej nazwie** ma unikatową tożsamość. W tym celu konieczne jest dodanie do manifestu dwóch dodatkowych metadanych:

- *unikatowy numer* należący do autorów zestawu;
- *podpisaną wartość hash* zestawu, która podaje unikatowy numer wystawcy odpowiedzialnego za wygenerowanie danego zestawu.

Dodanie powyższych metadanych oznacza konieczność użycia klucza publicznego i klucza prywatnego. **Klucz publiczny** dostarcza unikatowy numer identyfikacyjny, natomiast **klucz prywatny** jest wykorzystywany do podpisania zestawu.

Klucz publiczny pozwala na zagwarantowanie unikatowości odwołań zestawu, ponieważ zestawy o silnych nazwach wykorzystują ten klucz do identyfikacji.

Nadawanie silnych nazw zestawom na platformie .NET Framework jest bardzo ważne z dwóch powodów:

- Umożliwia załadowanie zestawu do „globalnego bufora zestawów".
- Umożliwia odwoływanie się do zestawu przez inne zestawy z silną nazwą.

Silne nazwy mają znacznie mniejsze znaczenie na platformie .NET Core, ponieważ platforma ta nie ma globalnego bufora zestawów ani nie egzekwuje drugiego ograniczenia.

Na platformie .NET Framework klucz prywatny chroni zestaw przed modyfikacją. Bez niego nikt nie może wydać zmodyfikowanej wersji zestawu, nie niszcząc przy tym podpisu. W praktyce jest to przydatne, gdy ładujemy zestaw do globalnego bufora zestawów platformy .NET Framework. W .NET Core podpis ma niewielkie znaczenie, ponieważ nie jest sprawdzany.

Nadanie silnej nazwy wcześniej „słabemu" zestawowi powoduje zmianę jego tożsamości. Dlatego też zestawom produkcyjnym nadawaj silne nazwy od samego początku, jeśli podejrzewasz, że będą ich potrzebować w przyszłości.

Nadanie silnej nazwy dla wcześniej „słabego" zestawu powoduje zmianę jego tożsamości. Dlatego też zestawom produkcyjnym nadawaj silne nazwy jeszcze przed ich udostępnieniem.

Jak nadać zestawowi silną nazwę?

W celu nadania zestawowi silnej nazwy zaczniemy od wygenerowania pary kluczy publicznego i prywatnego. Do tego celu można wykorzystać narzędzie o nazwie *sn.exe*:

```
sn.exe -k MojaParaKluczy.snk
```

Visual Studio instaluje skrót o nazwie *Developer Command Prompt for VS* (wiersz poleceń programisty dla VS), który uruchamia wiersz poleceń zawierający na ścieżce *PATH* narzędzia programistyczne, np. *sn.exe*.

Powyższe polecenie powoduje wygenerowanie nowej pary kluczy i umieszczenie jej w pliku o nazwie *MojaParaKluczy.snk*. Jeżeli utracimy ten plik, wówczas na zawsze stracimy możliwość ponownej kompilacji zestawu o takiej samej tożsamości.

Aby podpisać zestaw za pomocą tego pliku, możesz zmodyfikować plik projektu. W Visual Studio otwórz okno *Project Properties* (właściwości projektu), przejdź na kartę *Signing* (podpisy), zaznacz pole wyboru *Sign the assembly* (podpisz zestaw) i wybierz swój plik *.snk*.

Tę samą parę kluczy można wykorzystać do podpisania wielu zestawów, które nadal będą miały odmienne tożsamości, pod warunkiem że ich proste nazwy są inne.

Nazwy zestawów

Wielokrotnie wspomniana już „tożsamość” zestawu składa się z czterech fragmentów metadanych w jego manifeście:

- jego prosta nazwa;
- jego numer wersji (w przypadku braku to „0.0.0.0”);
- jego kultura (w przypadku zestawu satelickiego to „neutralny”);
- jego token klucza publicznego (w przypadku silnej nazwy to „null”).

Prosta nazwa nie pochodzi od dowolnego atrybutu, ale od nazwy pliku, w którym zestaw został pierwotnie skompilowany (bez żadnego rozszerzenia). Dlatego też prostą nazwą dla zestawu *System.Xml.dll* jest *System.Xml*. Zmiana nazwy pliku nie powoduje zmiany prostej nazwy zestawu.

Numer wersji jest pobierany z atrybutu `[AssemblyVersion]`. Jest to ciąg tekstowy podzielony w poniższy sposób na cztery części:

```
wersja-główna.wersja-pomocnicza.numer-kompilacji.rewizja
```

Numer wersji można zdefiniować w pokazany poniżej sposób:

```
[assembly: AssemblyVersion ("2.5.6.7")]
```

Kultura jest pobierana z atrybutu `[AssemblyCulture]` i ma zastosowanie dla zestawów satelickich, jak to zostanie omówione w dalszej części rozdziału.

Token klucza publicznego pochodzi od silnej nazwy dostarczonej w trakcie kompilacji, jak pokazano we wcześniejszej części rozdziału.

W pełni kwalifikowana nazwa

W pełni kwalifikowana nazwa zestawu to ciąg tekstowy zawierający wszystkie wymienione wcześniej zestawy identyfikacyjne w poniższym formacie:

```
prosta-nazwa, Version=wersja, Culture=kultura, PublicKeyToken=klucz-publiczny
```

Na przykład w pełni kwalifikowana nazwa zestawu *System.Private.CoreLib.dll to System.Pri-vate.CoreLib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=7cec85d7bea7798e.*

Jeżeli zestaw nie ma atrybutu `[AssemblyVersion]`, wówczas numer wersji jest wyświetlany w postaci „0.0.0.0”. W przypadku zestawu niepodpisanego token klucza jest podawany jako „null”.

Właściwość `FullName` obiektu `Assembly` zwraca w pełni kwalifikowaną nazwę. Kompilator zawsze używa w pełni kwalifikowanej nazwy podczas umieszczania w manifeście odwołań do zestawu.

W pełni kwalifikowana nazwa zestawu nie zawiera nazwy katalogu wskazującej jego położenie na dysku. Wskazanie zestawu znajdującego się w innym katalogu to zupełnie oddzielna kwestia, którą zajmiemy się w sekcji „Ładowanie, znajdowanie i izolowanie zestawów” w dalszej części rozdziału.

Klasa AssemblyName

Klasa `AssemblyName` zawiera typowane właściwości dla każdego z czterech komponentów w pełni kwalifikowanej nazwy zestawu. Omawiana klasa służy do dwóch celów:

- przetwarzanie lub przygotowanie w pełni kwalifikowanej nazwy zestawu;
- przechowywanie pewnych dodatkowych danych pomagających w ustaleniu (znalezieniu) zestawu.

Obiekt `AssemblyName` można otrzymać na dowolny z wymienionych poniżej sposobów:

- utworzenie egzemplarza klasy `AssemblyName` i podanie w pełni kwalifikowanej nazwy;
- wywołanie `GetName()` w istniejącym obiekcie `Assembly`;
- wywołanie `AssemblyName.GetAssemblyName()`, podając ścieżkę dostępu do pliku zestawu na dysku.

Obiekt `AssemblyName` można również utworzyć bez żadnych argumentów, a następnie przypisać wartości jego właściwościom, aby przygotować w pełni kwalifikowaną nazwę. Podczas tworzenia w taki sposób egzemplarz `AssemblyName` jest modyfikowalny.

Poniżej wymieniono najważniejsze właściwości i metody obiektu `AssemblyName`:

```
string      FullName    { get; }            // w pełni kwalifikowana nazwa
string      Name        { get; set; }       // prosta nazwa
Version     Version     { get; set; }       // wersja zestawu
CultureInfo CultureInfo { get; set; }       // dla zestawów satelickich
string      CodeBase    { get; set; }       // położenie

byte[]      GetPublicKey();                 // 160 bajtów
void        SetPublicKey (byte[] key);
byte[]      GetPublicKeyToken();            // wersja 8-bajtowa
void        SetPublicKeyToken (byte[] publicKeyToken);
```

W powyższym fragmencie kodu `Version` to silnie typowana reprezentacja wraz z właściwościami dla poszczególnych liczb składających się na wersję (czyli: wersja główna, wersja pomocnicza, numer kompilacji i rewizja). Metoda `GetPublicKey()` zwraca w pełni kryptograficzny klucz publiczny, natomiast `GetPublicKeyToken()` zwraca ostatnie 8 bajtów używanych podczas przygotowywania tożsamości.

W celu użycia obiektu `AssemblyName` do pobrania prostej nazwy zestawu można użyć wywołania takiego jak pokazane poniżej:

```
Console.WriteLine (typeof (string).Assembly.GetName().Name);  // System.Private.CoreLib
```

Natomiast poniższe wywołanie pobiera wersję zestawu:

```
string v = myAssembly.GetName().Version.ToString();
```

Właściwością `CodeBase` zajmiemy się w dalszej części rozdziału.

Wersja informacyjna zestawu i wersja pliku

Dwa kolejne atrybuty zestawu określają informacje dotyczące wersji. W odróżnieniu od `AssemblyVersion` poniższe dwa atrybuty nie mają wpływu na tożsamość zestawu, w związku z czym nie mają też wpływu na to, co się dzieje w czasie kompilacji lub wykonywania:

`AssemblyInformationalVersion`
: Wersja wyświetlana użytkownikowi końcowemu. Ta wersja będzie widoczna w oknie dialogowym właściwości pliku w systemie Windows jako pozycja *Wersja produktu*. Tu można umieścić dowolny ciąg tekstowy, np. „5.1 Beta 2”. Zwykle wszystkie *zestawy* aplikacji będą miały przypisaną tę samą wersję informacyjną.

`AssemblyFileVersion`
: Wersja odwołująca się do wersji kompilacji danego zestawu. Ta wersja będzie widoczna w oknie dialogowym właściwości pliku w systemie Windows jako pozycja *Wersja pliku*. Podobnie jak `AssemblyVersion` musi zawierać ciąg tekstowy zawierający do czterech liczb rozdzielonych kropkami.

Technologia Authenticode

Technologia *Authenticode* to system podpisywania kodu, którego celem jest dostarczenie tożsamości wydającego. Ta technologia i podpis z użyciem *silnej nazwy* są od siebie niezależne — zestaw można podpisać za pomocą dowolnego z wymienionych lub za pomocą obu systemów.

Wprawdzie podpis silnej nazwy może zagwarantować, że *zestawy* A, B i C pochodzą od tego samego dostawcy (oczywiście przy założeniu, że klucz prywatny nie wyciekł), jednak nie otrzymujemy żadnych informacji o tym dostawcy. Jeżeli chcemy się dowiedzieć, czy dostawcą jest Joe Albahari, czy Microsoft Corporation, potrzebujemy technologii Authenticode.

Technologia Authenticode jest użyteczna podczas pobierania programów z internetu, ponieważ gwarantuje pochodzenie programu ze źródła wymienionego przez urząd certyfikacji oraz potwierdza, że program nie został zmodyfikowany w transporcie. Ponadto chroni przed wyświetleniem ostrzeżenia dotyczącego nieznanego wydawcy podczas uruchamiania po raz pierwszy programu pobranego z internetu. Zastosowanie Authenticode jest również wymagane podczas umieszczania programów w sklepie Windows Store.

Technologia Authenticode działa nie tylko z zestawami .NET, ale również z niezarządzanymi plikami wykonywalnymi oraz binarnymi, takimi jak pliki *.msi*. Oczywiście Authenticode nie gwarantuje, że program jest pozbawiony np. malware, choć prawdopodobieństwo istnienia w nim złośliwego oprogramowania jest znacznie mniejsze. Za plikiem wykonywalnym lub biblioteką stoi konkretna osoba bądź jednostka (potwierdzająca siebie odpowiednio dokumentem tożsamości lub firmowym).

Środowisko uruchomieniowe CLR nie traktuje podpisu w technologii Authenticode jako część tożsamości zestawu. Jednak może odczytywać i weryfikować podpisy Authenticode na żądanie, jak wkrótce zobaczysz.

Podpisanie z użyciem Authenticode wymaga skontaktowania się z **urzędem certyfikacji** (ang. *certificate authority* — CA) i przedstawienia dokumentu potwierdzającego tożsamość osoby lub

firmy (wpis do rejestru firm itd.). Po sprawdzeniu dokumentów przez urząd certyfikacji zostaje wydany certyfikat X.509, którego okres ważności wynosi najczęściej od roku do pięciu lat. Dzięki temu certyfikatowi możesz podpisywać zestawy za pomocą narzędzia *signtool.exe*. Certyfikat można wygenerować samodzielnie, używając do tego narzędzia *makecert.exe*, ale będzie on rozpoznawany wyłącznie w komputerach, w których został zainstalowany.

Możliwość działania certyfikatu (niepodpisanego samodzielnie) w dowolnym komputerze opiera się na architekturze klucza publicznego. Ogólnie rzecz ujmując, certyfikat jest podpisany za pomocą innego certyfikatu należącego do urzędu certyfikacji. Wspomniany urząd certyfikacji jest uznawany za zaufany, ponieważ informacje o tych wszystkich urzędach są zaszyte w systemie operacyjnym. (Jeżeli chcesz je przejrzeć w systemie Windows, przejdź do Panelu sterowania, a następnie w polu wyszukiwania wpisz hasło „certyfikat". W sekcji *Narzędzia administracyjne* kliknij opcję *Zarządzaj certyfikatami komputera*, aby uruchomić menedżera certyfikatów. Rozwiń węzeł *Zaufane główne urzędy certyfikacji* i kliknij pozycję *Certyfikaty*. W przypadku wypłynięcia certyfikatu danego podmiotu urząd certyfikacji może go wycofać. Dlatego też weryfikacja podpisu Authenticode wymaga okresowego pobrania z urzędu certyfikacji uaktualnionej listy certyfikatów, które zostały wycofane.

Ponieważ technologia Authenticode używa podpisu kryptograficznego, ten podpis Authenticode staje się nieprawidłowy, gdy ktokolwiek spróbuje zmodyfikować plik. Tematy kryptografii i wartości hash zostaną poruszone w rozdziale 21.

Jak podpisać zestaw z użyciem Authenticode?

W tej sekcji zajmiemy się tematem podpisywania zestawu za pomocą technologii Authenticode.

Uzyskanie i instalacja certyfikatu

Pierwszym krokiem jest pobranie z urzędu certyfikacji własnego certyfikatu przeznaczonego do podpisywania kodu (zob. ramkę „Gdzie uzyskać certyfikat przeznaczony do podpisywania kodu?"). Możemy pracować z certyfikatem w postaci pliku chronionego hasłem lub też umieścić certyfikat w magazynie certyfikatów znajdującym się w systemie operacyjnym. Zaletą drugiego z wymienionych podejść jest możliwość podpisywania zestawów bez konieczności podawania hasła. Dodatkową korzyścią jest uniknięcie ujawniania hasła w zautomatyzowanych skryptach kompilacji lub plikach wsadowych.

W celu umieszczenia certyfikatu w magazynie certyfikatów systemu operacyjnego otwórz menedżera certyfikatów w sposób opisany wcześniej. Następnie otwórz folder *Osobiste*, kliknij prawym przyciskiem myszy folder *Certyfikaty* i wybierz pozycję *Wszystkie zadania /Importuj*. Zostanie uruchomiony kreator importu, który poprowadzi Cię przez cały proces. Po zaimportowaniu certyfikatu klikamy przycisk *Wyświetl* dla danego certyfikatu, przechodzimy do karty *Szczegóły* i kopiujemy *Odcisk palca* tego certyfikatu. Jest to wartość hash typu SHA-1, która będzie później potrzebna do identyfikacji certyfikatu podczas podpisywania.

Jeżeli chcesz podpisać zestaw również za pomocą silnej nazwy, wówczas musisz to zrobić *przed* jego podpisaniem w technologii Authenticode. Wynika to z faktu, że środowisko uruchomieniowe CLR rozpoznaje podpisy Authenticode, ale nie na odwrót. Dlatego też podpisanie zestawu silną nazwą już *po* zastosowaniu podpisu w technologii Authenticode spowoduje uznanie tego drugiego za nieupoważnioną modyfikację zestawu.

Gdzie uzyskać certyfikat przeznaczony do podpisywania kodu?

Pewna liczba urzędów certyfikacji wydających certyfikaty pozwalające na podpisywanie kodu została umieszczona w systemie Windows jako główne urzędy certyfikacji. Zaliczają się do nich Comodo, Go Daddy, GlobalSign, DigiCert, thawte i Symantec.

Istnieją również sprzedawcy tacy jak Ksoftware, którzy oferują certyfikaty do podpisywania kodu od poprzednich organizacji w obniżonej cenie.

Certyfikaty Authenticode wydane przez Ksoftware, Comodo, Go Daddy i GlobalSign są uznawane za mniej restrykcyjne, ponieważ pozwalają także na podpisywanie programów innych niż Microsoft. Pomijając ten aspekt, produkty oferowane przez wszystkich dostawców są identyczne pod względem funkcjonalności.

Zwróć uwagę, że certyfikat SSL, ogólnie rzecz biorąc, nie może być używany do podpisu Authenticode (mimo stosowania tej samej infrastruktury X.509). Po części wynika to z faktu, że certyfikat dla SSL jest potwierdzeniem własności domeny, natomiast Authenticode dotyczy potwierdzenia tożsamości.

Podpisywanie za pomocą signtool.exe

Do podpisywania programu z użyciem technologii Authenticode można wykorzystać narzędzie *signtool.exe* dostarczane wraz z Visual Studio (szukaj w folderze *Program Files* w podfolderze *Microsoft SDKs\ClickOnce\SignTool*). Poniższy kod podpisuje plik o nazwie *LINQPad.exe* za pomocą certyfikatu znajdującego się w katalogu komputerowym o nazwie *Joseph Albahari* przy użyciu bezpiecznego algorytmu haszowania SHA256:

```
signtool sign /n "Joseph Albahari" /fd sha256 LINQPad.exe
```

Istnieje również możliwość podania opisu i adresu URL produktu za pomocą opcji odpowiednio `/d` i `/du`:

```
... /d LINQPad /du http://www.linqpad.net
```

W większości przypadków trzeba również podać *serwer znakowania czasowego*.

Znakowanie czasowe

Po wygaśnięciu certyfikatu nie można już podpisywać programów. Jednak programy podpisane wcześniej, jeszcze *przed* wygaśnięciem certyfikatu, nadal są uznawane za prawidłowe, jeżeli będzie podany tzw. *serwer znakowania czasowego* za pomocą opcji `/tr`. Do tego celu urząd certyfikacji dostarcza adres URI. Przedstawiony poniżej dotyczy certyfikatów wydawanych przez Comodo (i Ksoftware):

```
... /tr http://timestamp.comodoca.com/authenticode /td SHA256
```

Sprawdzenie, czy program został podpisany

Najłatwiejszym sposobem wyświetlenia podpisu Authenticode w pliku będzie przejście do okna dialogowego właściwości danego pliku w *Eksploratorze Windows* (karta *Podpisy cyfrowe*). Narzędzie *signtool.exe* również oferuje w tym celu odpowiednią opcję.

Zasoby i zestawy satelickie

Aplikacja zwykle zawiera nie tylko kod wykonywalny, ale również inne zasoby, takie jak: tekst, obrazy i pliki XML. Tego rodzaju zawartość może zostać przedstawiona w zestawie za pomocą *zasobu*. Mamy dwa nakładające się na siebie przypadki użycia zasobów:

- wykorzystanie danych, które nie mogą być umieszczone w kodzie źródłowym, np. obrazów;
- przechowywanie danych, które mogą wymagać przetłumaczenia w aplikacji wielojęzycznej.

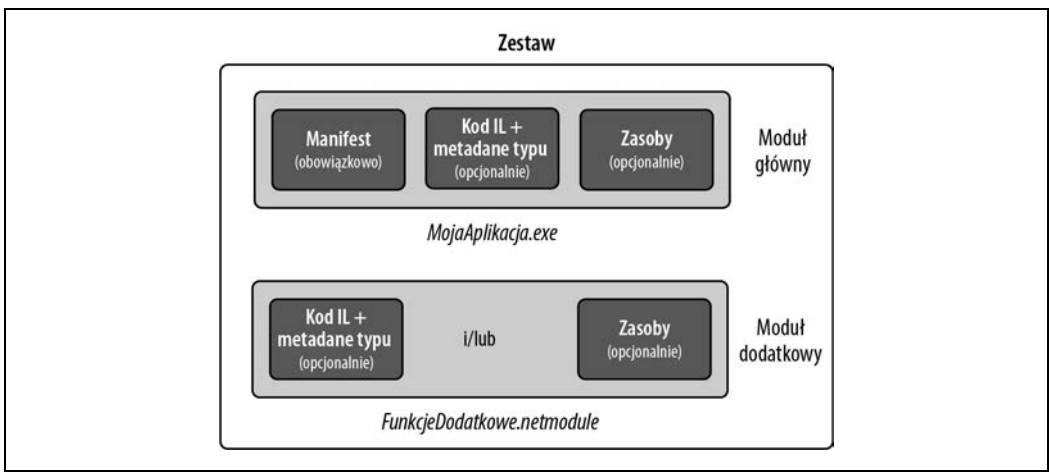
Zasób zestawu to ostatecznie strumień bajtów o pewnej nazwie. Zestaw możemy potraktować jako słownik tablic bajtowych o kluczach w postaci ciągów tekstowych. Taką postać otrzymamy za pomocą narzędzia *ildasm* po przeanalizowaniu zestawu zawierającego zasoby o nazwach *banner.jpg* i *data.xml*, jak widać w poniższym fragmencie kodu:

```
.mresource public banner.jpg
{
  // offset: 0x00000F58 Length: 0x000004F6
}
.mresource public data.xml
{
  // offset: 0x00001458 Length: 0x0000027E
}
```

W omawianym przykładzie pliki *banner.jpg* i *data.xml* zostały umieszczone bezpośrednio w zestawie, każdy z własnym osadzonym zasobem. Jest to rozwiązanie najprostsze.

Platforma .NET Framework pozwala również na dodawanie zawartości za pomocą pośrednich kontenerów *.resources*. Zostały one zaprojektowane do przechowywania treści, która może wymagać przetłumaczenia na inne języki. Zlokalizowane zasoby *.resources* mogą być pakowane jako poszczególne zestawy satelickie albo automatycznie wybierane w trakcie działania aplikacji na podstawie języka systemu operacyjnego użytkownika.

Na rysunku 18.2 pokazaliśmy zestaw zawierający dwa bezpośrednio osadzone zasoby plus kontener *.resources* o nazwie *welcome.resources*, dla którego utworzyliśmy dwa zlokalizowane zestawy satelickie.



Rysunek 18.2. Zasoby

Bezpośrednie osadzanie zasobów

Osadzanie zasobów w zestawach nie jest obsługiwane w aplikacjach przeznaczonych do umieszczenia w sklepie Windows Store. Zamiast tego wszelkie dodatkowe pliki należy umieścić w pakiecie wdrożeniowym i wtedy można uzyskać do nich dostęp za pomocą klasy `StorageFolder`, która została zdefiniowana w przestrzeni nazw `Package.Current.InstalledLocation`.

Aby bezpośrednio osadzić zasoby za pomocą Visual Studio:

- dodaj plik do projektu;
- zdefiniuj jego akcję kompilacji jako *Zasób osadzony*.

Visual Studio zawsze stosuje w nazwach zasobów prefiks w postaci domyślnej przestrzeni nazw projektu plus nazwy podkatalogów, w których znajduje się dany plik. Dlatego też jeśli domyślna przestrzeń nazw projektu to `Westwind.Reports`, a plik nosi nazwę *banner.jpg* i znajduje się w katalogu *pictures*, wówczas nazwa zasobu to *Westwind.Reports.pictures.banner.jpg*.

W nazwach zasobów wielkość liter ma znaczenie. Oznacza to, że nazwy podkatalogów w Visual Studio zawierających zasoby również rozróżniają wielkość liter.

W celu pobrania zestawu można wywołać `GetManifestResourceStream()` w obiekcie zestawu zawierającego zasób. Wartością zwrotną będzie strumień, który można odczytać dokładnie w taki sam sposób jak każdy inny:

```
Assembly a = Assembly.GetEntryAssembly();

using (Stream s = a.GetManifestResourceStream ("TestProject.data.xml"))
using (XmlReader r = XmlReader.Create (s))
  ...

System.Drawing.Image image;
using (Stream s = a.GetManifestResourceStream ("TestProject.banner.jpg"))
  image = System.Drawing.Image.FromStream (s);
```

Otrzymany strumień pozwala na wyszukiwanie danych, więc można w nim użyć poniższego fragmentu kodu:

```
byte[] data;
using (Stream s = a.GetManifestResourceStream ("TestProject.banner.jpg"))
  data = new BinaryReader (s).ReadBytes ((int) s.Length);
```

Jeżeli użyliśmy Visual Studio do osadzenia zasobu, musimy pamiętać o dołączeniu prefiksu opartego na przestrzeni nazw. Aby uniknąć błędu, prefiks można podać w oddzielnym argumencie, stosując *typ*. W poniższym wywołaniu typu jako prefiks została użyta przestrzeń nazw:

```
using (Stream s = a.GetManifestResourceStream (typeof (X), "data.xml"))
```

W powyższym kodzie polecenie `X` może być dowolnym typem wraz z żądaną przestrzenią nazw zasobu (zwykle typ w katalogu tego samego projektu).

Zdefiniowanie akcji kompilacji elementu projektu w Visual Studio jako *Zasób* w aplikacji WPF *nie* jest tym samym co zdefiniowanie akcji kompilacji jako *Zasób osadzony*. Pierwsze z wymienionych w rzeczywistości powoduje dodanie do pliku *.resources* elementu o nazwie *<NazwaZestawu>.g.resources*, a dostęp do zawartości tego elementu odbywa się za pomocą klasy WPF o nazwie `Application` z użyciem adresu URI jako klucza.

Zamieszanie potęguje dodatkowo fakt, że WPF w jeszcze większym stopniu nadużywa pojęcia zasobu. Tutaj *zasoby statyczne* i *zasoby dynamiczne* zupełnie nie mają powiązania z zasobami zestawu.

Wywołanie `GetManifestResourceNames()` zwraca nazwy wszystkich zasobów w zestawie.

Pliki .resources

Pliki *.resources* służą do przechowywania treści, którą potencjalnie można lokalizować. Ostatecznie plik *.resources* staje się zasobem osadzonym w zestawie, podobnie jak plik każdego innego rodzaju. Różnica polega na konieczności wykonania poniższych działań:

- na początku należy w pliku *.resources* umieścić treść;
- następnie trzeba uzyskać dostęp do tej treści za pomocą `ResourceManager` lub adresu *URI* typu *pack*, zamiast używać wywołania `GetManifestResourceStream()`.

Pliki *.resources* mają strukturę binarną, więc nie są możliwe do edycji przez człowieka. Dlatego też podczas pracy z nimi trzeba się opierać na narzędziach dostarczanych przez platformę .NET Framework i środowisko Visual Studio. Standardowe podejście do pracy z ciągami tekstowymi lub typami prostych danych polega na użyciu formatu *.resx*, który może być skonwertowany na plik *.resources* przez Visual Studio lub polecenie `resgen`. Format *.resx* jest również odpowiedni w przypadku obrazów przeznaczonych dla aplikacji Windows Forms lub ASP.NET.

W aplikacji WPF dla obrazów lub podobnej treści, do której odwołujemy się za pomocą adresu URI, konieczne jest użycie oferowanej przez Visual Studio akcji kompilacji o nazwie *Zasób*. Ma to zastosowanie niezależnie od tego, czy wykorzystywana będzie lokalizacja danych.

W poniższych sekcjach znajduje się omówienie wymienionych podejść.

Pliki .resx

Plik *.resx* ma format, na podstawie którego następuje później wygenerowanie pliku typu *.resources*. Plik *.resx* używa formatu XML i ma strukturę w postaci par nazwa-wartość, jak pokazano poniżej:

```
<root>
  <data name="Greeting">
    <value>witaj</value>
  </data>
  <data name="DefaultFontSize" type="System.Int32, mscorlib">
    <value>10</value>
  </data>
</root>
```

W celu utworzenia pliku *.resx* w Visual Studio należy dodać element projektu typu *Plik zasobów*. Pozostałe kroki są wykonywane automatycznie:

- Utworzony zostaje prawidłowy nagłówek.
- Dostarczone jest narzędzie pozwalające na dodawanie ciągów tekstowych, obrazów, plików oraz innego rodzaju danych.
- Plik *.resx* zostaje automatycznie skonwertowany na format *.resources* i osadzony w zestawie podczas kompilacji.
- Utworzona zostaje klasa, która później ma pomóc w uzyskaniu dostępu do danych.

Narzędzie przeznaczone do obsługi zasobu dodaje obrazy jako obiekty `Image` (*System.Drawing.dll*), a nie jako tablice bajtów, co czyni je nieodpowiednimi dla aplikacji WPF.

Odczyt plików .resources

Jeżeli utworzysz plik *.resx* w Visual Studio, automatycznie zostanie wygenerowana klasa o tej samej nazwie zawierająca właściwości pozwalające na pobieranie poszczególnych elementów.

Klasa `ResourceManager` odczytuje pliki *.resources* osadzone w zestawie:

```
ResourceManager r = new ResourceManager ("welcome",
                                         Assembly.GetExecutingAssembly());
```

(Jeżeli zasób został skompilowany w Visual Studio, pierwszy argument musi zawierać prefiks w postaci przestrzeni nazw).

Następnie istnieje możliwość uzyskania dostępu do zawartości za pomocą wywołań `GetString()` lub `GetObject()` z zastosowaniem rzutowania:

```
string greeting = r.GetString ("Greeting");
int fontSize = (int) r.GetObject ("DefaultFontSize");
Image image = (Image) r.GetObject ("flag.png");
```

Jeżeli chcemy sprawdzić zawartość pliku *.resources*, możemy użyć następującego fragmentu kodu:

```
ResourceManager r = new ResourceManager (...);
ResourceSet set = r.GetResourceSet (CultureInfo.CurrentUICulture,
                                    true, true);
foreach (System.Collections.DictionaryEntry entry in set)
  Console.WriteLine (entry.Key);
```

Utworzenie zasobu pack URI w Visual Studio

W aplikacji WPF pliki XAML muszą mieć możliwość uzyskania dostępu do zasobów za pomocą adresów URI, np.:

```
<Button>
  <Image Height="50" Source="flag.png"/>
</Button>
```

lub (jeśli zasób znajduje się w innym zestawie):

```
<Button>
  <Image Height="50" Source="UtilsAssembly;Component/flag.png"/>
</Button>
```

(W powyższym fragmencie kodu `Component` to dosłowne słowo kluczowe).

W celu utworzenia zasobów przeznaczonych do wczytywania w taki sposób nie można wykorzystać plików *.resx*. Zamiast tego konieczne jest dodanie plików do projektu, a następnie zdefiniowanie ich akcji kompilacji jako *Zasób*, a nie *Zasób osadzony*. Następnie Visual Studio kompiluje je do pliku o nazwie *<NazwaZestawu>.g.resources* — jest to również miejsce dla skompilowanych plików XAML (*.baml*).

W celu programowego wczytania zasobu opartego na URI należy użyć wywołania `Application.GetResourceStream()`:

```
Uri u = new Uri ("flag.png", UriKind.Relative);
using (Stream s = Application.GetResourceStream (u).Stream)
```

Zwróć uwagę na użycie względnego adresu URI. Istnieje również możliwość zastosowania bezwzględnego adresu URI w przedstawionym poniżej formacie (trzy przecinki to nie jest błąd):

```
Uri u = new Uri ("pack://application:,,,/flag.png");
```

Jeżeli zamiast powyższego wskażemy obiekt `Assembly`, wówczas możemy pobrać treść za pomocą klasy `ResourceManager`:

```
Assembly a = Assembly.GetExecutingAssembly();
ResourceManager r = new ResourceManager (a.GetName().Name + ".g", a);
using (Stream s = r.GetStream ("flag.png"))
  ...
```

Klasa `ResourceManager` pozwala również na sprawdzenie zawartości kontenera *.g.resources* w danym zestawie.

Zestawy satelickie

Dane osadzone w plikach *.resources* mogą być lokalizowane.

Lokalizacja zasobu ma znaczenie, gdy aplikacja będzie uruchamiana w wersji systemu Windows przystosowanej do wyświetlania danych w różnych językach. W celu zachowania spójności aplikacja powinna używać tego samego języka.

Typowa konfiguracja przedstawia się następująco:

- zestaw główny zawiera plik *.resources* dla domyślnego lub zapasowego języka;
- oddzielne *zestawy satelickie* zawierają zlokalizowane pliki *.resources* przetłumaczone na różne języki.

Kiedy aplikacja działa, platforma .NET Core analizuje język używany przez system operacyjny (sprawdza wartość `CultureInfo.CurrentUICulture`). W trakcie każdego żądania zasobu za pomocą klasy `ResourceManager` platforma szuka zlokalizowanego zestawu satelickiego. Jeżeli taki jest dostępny — i zawiera żądany klucz zasobu — wówczas będzie użyty zamiast wersji w zestawie głównym.

Oznacza to możliwość rozszerzenia obsługi języka przez dodanie nowych zestawów satelickich, bez zmiany zestawu głównego.

Zestaw satelicki nie może zawierać kodu wykonywalnego, a jedynie zasoby.

Zestawy satelickie są wdrażane w podkatalogach katalogu zestawu zgodnie z przedstawionym poniżej schematem:

```
katalogBazowyProgramu\MójProgram.exe
                      \MojaBiblioteka.exe
                      \XX\MójProgram.resources.dll
                      \XX\MojaBiblioteka.resources.dll
```

W powyższym fragmencie kodu *XX* odwołuje się do dwuznakowego kodu języka (np. „pl” dla języka polskiego) lub do kodu języka i regionu (np. „en-GB” dla języka angielskiego używanego w Wielkiej Brytanii). Taki system nazw pozwala środowisku uruchomieniowemu CLR na automatyczne wyszukanie prawidłowego zestawu satelickiego i jego wczytanie.

Utworzenie zestawu satelickiego

Przypomnij sobie przedstawiony wcześniej przykład pliku *.resx* zawierającego poniższy fragment kodu:

```
<root>
  ...
  <data name="Greeting"
    <value>witaj</value>
  </data>
</root>
```

W trakcie działania aplikacji powitanie pobieramy w następujący sposób:

```
ResourceManager r = new ResourceManager ("welcome",
                                  Assembly.GetExecutingAssembly());
Console.Write (r.GetString ("Greeting"));
```

Przyjmujemy założenie, że po uruchomieniu aplikacji w niemieckiej wersji Windows chcemy wyświetlić powitanie w postaci *Hallo*. Pierwszym krokiem jest dodanie kolejnego pliku *.resx* o nazwie *welcome.de.resx* pozwalającego na zastąpienie słowa *witaj* słowem *hallo*:

```
<root>
  <data name="Greeting">
    <value>hallo<value>
  </data>
</root>
```

Jeżeli pracujemy w Visual Studio, nasza rola się na tym kończy. Podczas ponownej kompilacji w katalogu o nazwie *de* nastąpi automatyczne utworzenie zestawu satelickiego o nazwie *MojaAplikacja.* ↪*resources.dll*.

Testowanie zestawów satelickich

Aby symulować działanie systemu operacyjnego w innym języku, konieczna jest zmiana wartości `CurrentUICulture` za pomocą klasy `Thread`:

```
System.Threading.Thread.CurrentThread.CurrentUICulture
  = new System.Globalization.CultureInfo ("de");
```

Właściwość `CultureInfo.CurrentUICulture` to wersja tylko do odczytu tej samej właściwości.

Użyteczną techniką podczas testowania będzie przeprowadzenie lokalizacji na postać słów, które nadal mogą być uznawane za tekst w języku polskim, ale nie używają standardowych łacińskich znaków Unicode.

Obsługa oferowana przez narzędzia Visual Studio

Narzędzia w Visual Studio oferują rozbudowaną obsługę w zakresie lokalizacji komponentów oraz elementów graficznych. Narzędzie przeznaczone dla WPF oferuje własną procedurę w zakresie lokalizacji. Natomiast inne narzędzia oparte na `Component` zawierają stosowaną jedynie podczas projektowania właściwość powodującą, że w komponencie lub kontrolce Windows Forms pojawia się właściwość `Language`. Aby dostosować element do innego języka, wystarczy po prostu zmienić wartość właściwości `Language`, a następnie rozpocząć modyfikowanie komponentu. Wszystkie właściwości kontrolek oznaczone jako `Localizable` będą zachowane w pliku *.resx* dla danego języka. W każdej chwili można się przełączać między językami — przez zmianę właściwości `Language`.

Kultury i subkultury

Do dyspozycji mamy kultury i subkultury. Kultura przedstawia określony język, natomiast subkultura — region dla danego języka. Na platformie .NET Framework stosowany jest standard RFC1766, zgodnie z którym kultura i subkultura są przedstawiane za pomocą dwuznakowych kodów. Poniżej pokazano kod dla kultur angielskiej i niemieckiej:

```
en
de
```

Poniżej znajdują się kody dla subkultury języków angielskiego w Australii i niemieckiego w Austrii:

```
en-AU
de-AT
```

Na platformie .NET Framework kultura jest przedstawiana za pomocą klasy `System.Globalization.`
`CultureInfo`. Bieżącą kulturę aplikacji można sprawdzić w poniższy sposób:

```
Console.WriteLine (System.Threading.Thread.CurrentThread.CurrentCulture);
Console.WriteLine (System.Threading.Thread.CurrentThread.CurrentUICulture);
```

Wykonanie tego kodu w komputerze przygotowanym dla języka angielskiego używanego w Australii pokazuje różnicę między tymi dwoma wywołaniami:

```
EN-AU
EN-US
```

`CurrentCulture` odzwierciedla ustawienia regionalne w Panelu sterowania Windows, podczas gdy `CurrentUICulture` odzwierciedla język systemu operacyjnego.

Ustawienia regionalne obejmują takie rzeczy jak strefa czasowa oraz formatowanie waluty i daty. Ustawienie `CurrentCulture` określa domyślne zachowanie funkcji takich jak `DateTime.Parse()`. Ustawienia regionalne mogą być modyfikowane do takiego punktu, w którym nie przedstawiają już żadnej konkretnej kultury.

Ustawienie `CurrentUICulture` określa język, w jakim komputer komunikuje się z użytkownikiem. Do tego celu Australia nie potrzebuje oddzielnej wersji angielskiego, może wykorzystywać tę przeznaczoną dla USA. Jeżeli przez kilka miesięcy użytkownik będzie pracował w Austrii, wówczas może przejść do Panelu sterowania i zmienić `CurrentCulture` na język niemiecki używany w Austrii. Jednak gdy nie zna niemieckiego, wówczas będzie chciał pozostawić ustawienie `CurrentUICulture` na wskazujące np. język polski.

Klasa `ResourceManager` domyślnie używa właściwości `CurrentUICulture` bieżącego wątku w celu ustalenia poprawnego zestawu satelickiego do wczytania. W trakcie wczytywania zasobów stosuje mechanizm rozwiązania awaryjnego. Jeżeli została zdefiniowana podkultura zestawu, wówczas będzie użyta. W przeciwnym razie zostanie wykorzystana ogólna kultura. Jeśli takiej nie ma, rozwiązanie awaryjne polega na zastosowaniu w zestawie głównym kultury domyślnej.

Ładowanie, znajdowanie i izolowanie zestawów

Ładowanie zestawu ze znanej lokalizacji to względnie prosta procedura, którą nazywamy po prostu **ładowaniem zestawu**.

Najczęściej jednak programista (lub CLR) zna tylko pełną (lub uproszczoną) nazwę zestawu, który chce załadować. Ta procedura nazywa się **znajdowaniem zestawu** (ang. *assembly resolution*). Od ładowania zestawu różni się tym, że na początku wymaga zlokalizowania miejsca, w którym znajduje się zestaw.

Procedura znajdowania zestawu jest uruchamiana w dwóch sytuacjach:

- kiedy CLR musi znaleźć zależność;
- kiedy programista wywoła specjalną metodę, np. `Assembly.Load(AssemblyName)`.

Jeśli chodzi o przykład pierwszej sytuacji, wyobraź sobie aplikację składającą się z zestawu głównego i zbioru statycznie połączonych zestawów bibliotek (zależności):

```
AdventureGame.dll // główny zestaw
Terrain.dll   // używany zestaw
UIEngine.dll // używany zestaw
```

W określeniu „statycznie połączone” mamy na myśli, że zestaw *AdventureGame.dll* został skompilowany z odwołaniami do zestawów *Terrain.dll* i *UIEngine.dll*. Kompilator nie musi szukać zestawów *Terrain.dll* i *UIEngine.dll*, ponieważ został poinformowany (bezpośrednio lub przez MSBuild) o ich położeniu. W trakcie kompilacji zapisuje **pełne nazwy** zestawów *Terrain* i *UIEngine* w metadanych zestawu *AdventureGame.dll*, ale nie dodaje informacji, gdzie się znajdują. Dlatego po uruchomieniu programu system musi je **znaleźć**.

Ładowanie i znajdowanie zestawów obsługuje kontekst ładowania zestawów (ang. *assembly load context* — ALC), a dokładniej — obiekt klasy `AssemblyLoadContext` z przestrzeni nazw `System.Runtime.Loader`. Ponieważ *AdventureGame.dll* jest głównym zestawem aplikacji, CLR

znajduje jego zależności za pomocą domyślnego ALC (`AssemblyLoadContext.Default`). Ten ALC poszukiwanie zaczyna od sprawdzenia pliku o nazwie *AdventureGame.deps.json* (określającego położenie zależności), a jeśli go nie ma, sprawdza katalog bazowy aplikacji, w którym znajdzie pliki *Terrain.dll* i *UIEngine.dll*. (Domyślny ALC znajduje także zestawy platformy .NET Core).

Programista może dynamicznie ładować dodatkowe zestawy w czasie wykonywania programu. Może w ten sposób np. dołączać nowe funkcje, które są płatne. W takim przypadku te dodatkowe zestawy może dołączać za pomocą wywołania metody `Assembly.Load(AssemblyName)`.

Bardziej złożonym przykładem jest implementacja systemu wtyczek, w ramach którego użytkownik może dostarczać zewnętrzne zestawy, które są rozpoznawane i ładowane w trakcie działania programu w celu rozszerzenia jego funkcjonalności. Komplikacje wiążą się z tym, że każdy zestaw wtyczki może mieć własne zależności, które również trzeba odnaleźć.

Jeśli utworzysz podklasę klasy `AssemblyLoadContext` i przesłonisz jej metodę znajdującą (`Load`), możesz kontrolować sposób wyszukiwania zależności przez wtyczkę. Możesz np. zdecydować, że każda wtyczka powinna się znajdować w osobnym folderze, zawierającym także jej zależności.

Konteksty ALC pełnią jeszcze jedną funkcję. Przez utworzenie osobnego obiektu `AssemblyLoadContext` dla każdego z nich (wtyczka + zależności) możesz je od siebie odizolować, co sprawi, że ich zależności będą ładowane równolegle i nie będą przeszkadzać sobie wzajemnie (ani aplikacji głównej). Każdy z nich może np. mieć własną wersję JSON.NET. Dlatego oprócz *ładowania* i *znajdowania* konteksty ALC zapewniają mechanizm *izolacji*. W pewnych warunkach można nawet *usunąć* załadowany ALC, aby zwolnić zajmowaną przez niego pamięć.

W tej sekcji bardziej szczegółowo opisujemy wymienione mechanizmy i zagadnienia:

- Jak ALC obsługują ładowanie i znajdowanie zestawów.
- Rola domyślnego ALC.
- Metoda `Assembly.Load` i kontekstowe ALC.
- Jak posługiwać się klasą `AssemblyDependencyResolver`.
- Jak ładować i znajdować biblioteki niezarządzane.
- Usuwanie załadowanych ALC.
- Stare metody ładowania zestawów.

Następnie pokazujemy praktyczny przykład ilustrujący tworzenie systemu wtyczek wykorzystującego izolację ALC.

Klasa `AssemblyLoadContext` jest nowością w .NET Core. Na platformie .NET Framework też były konteksty ALC, ale miały ograniczoną funkcjonalność i były ukryte: jedynym sposobem, aby je tworzyć i ich używać, było wywołanie metod statycznych `LoadFile(string)`, `LoadFrom(string)` i `Load(byte[])` klasy `Assembly`. W porównaniu z API ALC te metody są sztywne, a ich obecność w kodzie może wywoływać zaskakujące efekty (w szczególności w odniesieniu do obsługi zależności). Dlatego lepiej jest używać API `AssemblyLoadContext` platformy .NET Core.

Konteksty ładowania zestawów

Jak napisaliśmy wcześniej, klasa `AssemblyLoadContext` odpowiada za ładowanie i znajdowanie zestawów oraz stanowi mechanizm izolacji.

Każdy obiekt klasy .NET `Assembly` należy do dokładnie jednego kontekstu `AssemblyLoadContext`. Kontekst ALC zestawu można pobrać tak:

```
Assembly assem = Assembly.GetExecutingAssembly();
AssemblyLoadContext context = AssemblyLoadContext.GetLoadContext (assem);
Console.WriteLine (context.Name);
```

Inaczej można powiedzieć, że ALC *zawiera* lub *ma* zestawy, do których dostęp można uzyskać przez jego własność `Assemblies`. Kontynuując poprzedni przykład:

```
foreach (Assembly a in context.Assemblies)
  Console.WriteLine (a.FullName);
```

Ponadto klasa `AssemblyLoadContext` zawiera statyczną własność `All`, która pozwala uzyskać listę wszystkich kontekstów ALC.

Aby utworzyć nowy kontekst ALC, należy utworzyć obiekt klasy `AssemblyLoadContext`, podając nazwę (przydaje się w trakcie debugowania), choć częściej najpierw tworzy się jej podklasę, aby zaimplementować logikę *znajdowania* zależności, czyli ładowania ich na podstawie *nazwy*.

Ładowanie zestawów

Klasa `AssemblyLoadContext` zawiera następujące metody do bezpośredniego ładowania zestawów do kontekstu:

```
public Assembly LoadFromAssemblyPath (string assemblyPath);
public Assembly LoadFromStream (Stream assembly, Stream assemblySymbols);
```

Pierwsza ładuje zestaw ze ścieżki do pliku, a druga — ze strumienia (`Stream`), który może pochodzić bezpośrednio z pamięci. Drugi parametr dotyczy zawartości pliku debugowania projektu (*.pdb*) i jest opcjonalny. Umożliwia dodanie do danych stosu informacji na temat kodu źródłowego po jego wykonaniu (przydatne do raportowania wyjątków).

Żadna z tych metod nie wykonuje *wyszukiwania* zestawu. Poniższy kod ładuje zestaw *c:\temp\foo.dll* do osobnego kontekstu ALC:

```
var alc = new AssemblyLoadContext ("Test");
Assembly assem = alc.LoadFromAssemblyPath (@"c:\temp\foo.dll");
```

Jeśli zestaw jest poprawny, zawsze uda się go załadować pod jednym ważnym warunkiem: jego uproszczona nazwa nie powtarza się w obrębie ALC. To znaczy, że nie można załadować wielu wersji jednego zestawu do jednego ALC — w takim przypadku należy utworzyć dodatkowe konteksty ALC. Dodatkowy egzemplarz zestawu *foo.dll* można załadować następująco:

```
var alc2 = new AssemblyLoadContext ("Test 2");
Assembly assem2 = alc2.LoadFromAssemblyPath (@"c:\temp\foo.dll");
```

Typy pochodzące z różnych obiektów `Assembly` są niekompatybilne, nawet jeśli poza tym zestawy są identyczne. W naszym przykładzie typy w `assem` są niezgodne z typami w `assem2`.

Załadowanego zestawu nie można usunąć inaczej niż przez usunięcie jego kontekstu ALC (zobacz podrozdział „Usuwanie kontekstów ALC”). CLR blokuje plik na czas, kiedy jest załadowany.

Unikaj blokowania pliku przez załadowanie go przez tablicę bajtów:

```
bytes[] bytes = File.ReadAllBytes (@"c:\temp\foo.dll");
var ms = new MemoryStream (bytes);
var assem = alc.LoadFromStream (ms);
```

Ma to dwie wady:

- Własność `Location` zestawu będzie pusta. Czasami dobrze jest wiedzieć, skąd został załadowany dany zestaw (a niektóre API wykorzystują tę informację do jego napełnienia).
- Natychmiast musi wzrosnąć zużycie pamięci prywatnej, aby zmieścił się w niej cały zestaw. Gdyby został załadowany na podstawie nazwy pliku, system CLR użyłby pliku mapowanego w pamięci, co pozwoliłoby na leniwe ładowanie i współdzielenie procesu. Ponadto, gdyby dostępnej pamięci zrobiło się mało, system operacyjny mógłby zwolnić pamięć zajmowaną przez zestaw i załadować go ponownie w razie potrzeby, aby uniknąć zapisu w pliku stronicowania.

Metoda LoadFromAssemblyName

Klasa `AssemblyLoadContext` zawiera także poniższą metodę ładującą zestaw po nazwie:

```
public Assembly LoadFromAssemblyName (AssemblyName assemblyName);
```

W odróżnieniu od dwóch poprzednio opisanych metod tej nie należy przekazywać żadnych informacji określających lokalizację zestawu, ponieważ instruuje ona ALC, że ma wyszukać zestaw.

Znajdowanie zestawów

Poprzednia metoda uruchamia *wyszukiwanie zestawu*. CLR także uruchamia tę procedurę podczas ładowania zależności. Powiedzmy, że zestaw *A* statycznie odwołuje się do zestawu *B*. Aby rozwiązać odwołanie do *B*, CLR uruchamia proces znajdowania zestawu *w kontekście ALC, w którym jest załadowany zestaw A*.

CLR znajduje zależności za pomocą procedury znajdowania zestawów — niezależnie od tego, czy uruchamiający ją zestaw znajduje się w domyślnym czy indywidualnym kontekście ALC. Różnica polega na tym, że w przypadku domyślnego ALC zasady znajdowania są sztywne, natomiast w indywidualnym ALC programista sam je tworzy.

Oto co potem następuje:

1. Najpierw CLR sprawdza, czy w danym kontekście ALC nie została już wykonana identyczna procedura znajdowania zestawu (z taką samą pełną nazwą zestawu). Jeśli tak, zwraca obiekt `Assembly`, który został zwrócony poprzednio.
2. W przeciwnym razie wywołuje wirtualną chronioną metodę `Load` kontekstu ALC, która znajduje i ładuje zestaw. Domyślna metoda `Load` kontekstu ALC stosuje zasady, które opisaliśmy w podrozdziale „Domyślny kontekst ALC”. W przypadku indywidualnego kontekstu ALC

sposób wyszukiwania zestawu leży w gestii programisty. Może on np. przeszukać wybrany folder i wywołać metodę `LoadFromAssemblyPath` po znalezieniu zestawu. Możliwe jest też zwrócenie już załadowanego zestawu z tego samego lub innego kontekstu ALC (pokazujemy to w podrozdziale „Pisanie systemu wtyczek").

3. Jeśli w drugim kroku zostanie zwrócone `null`, CLR wywołuje metodę `Load` domyślnego kontekstu ALC (jest to praktyczne narzędzie „awaryjne" do znajdowania zestawów platformy i aplikacji).
4. Jeśli wynik trzeciego kroku jest `null`, CLR ogłasza zdarzenia `Resolving` w obu kontekstach ALC — najpierw w domyślnym, a potem w oryginalnym.
5. Ze względu na zgodność z .NET Framework, jeśli zestaw nadal nie został znaleziony, zostaje zgłoszone zdarzenie `AppDomain.CurrentDomain.AssemblyResolve`.

Po zakończeniu tego procesu CLR sprawdza, czy nazwa załadowanego zestawu zgadza się z zażądaną nazwą. Uproszczona nazwa musi się zgadzać, podobnie jak token klucza publicznego, jeśli został określony. Wersja nie musi być taka sama — może być niższa lub wyższa od zażądanej.

Widzimy więc, że są dwa sposoby implementacji algorytmu znajdowania zestawów w niestandardowym kontekście ALC:

Przesłonięcie metody `Load` kontekstu ALC

Wybór tego sposobu pozwala w pełni kontrolować bieg wydarzeń w ALC, co zazwyczaj jest bardzo pożądane (i niezbędne, jeśli potrzebujesz izolacji).

Obsługa zdarzenia `Resolving` kontekstu ALC

Ta procedura jest uruchamiana tylko, jeśli domyślny kontekst ALC nie znajdzie zestawu.

Jeśli ze zdarzeniem `Resolving` zostanie powiązanych kilka procedur obsługi, wygrywa ta, która zwróci wartość inną niż `null`.

W ramach przykładu założymy, że do naszej aplikacji chcemy załadować zestaw o nazwie *foo.dll* z katalogu *c:\temp* (innego niż folder instalacji programu), który w trakcie kompilowania aplikacji był jej nieznany. Ponadto przyjmiemy, że *foo.dll* ma prywatną zależność *bar.dll*. Chcemy dopilnować, aby po załadowaniu i wykonaniu zestawu *c:\temp\foo.dll* program znalazł też *c:\temp\bar.dll*. Ponadto nie chcemy, aby zestaw *foo* i jego prywatna zależność *bar* przeszkadzały aplikacji głównej.

Zaczniemy od napisania własnego kontekstu ALC przesłaniającego metodę `Load`:

```
using System.IO;
using System.Runtime.Loader;

class FolderBasedALC : AssemblyLoadContext
{
  readonly string _folder;
  public FolderBasedALC (string folder) => _folder = folder;
```

```
  protected override Assembly Load (AssemblyName assemblyName)
  {
    // próba znalezienia zestawu
    string targetPath = Path.Combine (_folder, assemblyName.Name + ".dll");

    if (File.Exists (targetPath))
      return LoadFromAssemblyPath (targetPath); // Load the assembly

    return null; // nie udało się znaleźć zestawu — to może być zestaw platformy
  }
}
```

Zwróć uwagę, że w przypadku braku szukanego pliku zestawu metoda `Load` zwraca `null`. To ważne, ponieważ *foo.dll* będzie korzystać także z zestawów platformy .NET Core, w związku z czym metoda `Load` będzie wywoływana na takich zestawach jak `System.Runtime`. Zwrot `null` umożliwia systemowi CLR zwrot do domyślnego ALC, który znajdzie te zestawy.

Zauważ, że nie próbujemy załadować zestawów .NET Core do własnego kontekstu ALC. Zestawy platformy nie są przeznaczone do działania poza domyślnym kontekstem i próba ich załadowania do innego kontekstu może spowodować wadliwe działanie, pogorszenie wydajności i niespodziewaną niezgodność typów.

Oto przykład użycia naszego kontekstu ALC do załadowania zestawu *foo.dll* z folderu *c:\temp*:

```
var alc = new FolderBasedALC (@"c:\temp");
Assembly foo = alc.LoadFromAssemblyPath (@"c:\temp\foo.dll");
...
```

Kiedy potem zaczniemy wywoływać kod zestawu *foo*, system CLR w pewnym momencie będzie musiał znaleźć zależność *bar.dll*. W tym momencie uruchomi się metoda `Load` naszego kontekstu ALC, która znajdzie ten zestaw w folderze *c:\temp*.

W tym przypadku nasza metoda `Load` może także znaleźć zestaw *foo.dll*, więc moglibyśmy uprościć kod:

```
var alc = new FolderBasedALC (@"c:\temp");
Assembly foo = alc.LoadFromAssemblyName (new AssemblyName ("foo"));
...
```

Teraz zastanowimy się nad innym rozwiązaniem: zamiast tworzyć podklasę klasy `AssemblyLoadContext` i przesłaniać metodę `Load`, możemy utworzyć obiekt samej klasy `AssemblyLoadContext` i obsłużyć jej zdarzenie `Resolving`:

```
var alc = new AssemblyLoadContext ("test");
alc.Resolving += (loadContext, assemblyName) =>
{
  string targetPath = Path.Combine (@"c:\temp", assemblyName.Name + ".dll");
  return alc.LoadFromAssemblyPath (targetPath); // ładowanie zestawu
};
Assembly foo = alc.LoadFromAssemblyName (new AssemblyName ("foo"));
```

Zauważ, że nie musimy sprawdzać, czy zestaw istnieje. Zdarzenie `Resolving` pojawia się **po** próbie znalezienia zestawu przez domyślny kontekst ALC (i tylko w razie jego niepowodzenia), więc nasza procedura obsługi nie zostanie uruchomiona dla zestawów platformy. Dzięki temu rozwiązanie jest prostsze, choć ma pewną wadę. Przypomnij sobie, że nasza aplikacja główna w czasie

kompilacji nic nie wie o zestawach *foo.dll* i *bar.dll*. To znaczy, że aplikacja ta też może mieć zależności o nazwach *foo.dll* lub *bar.dll*. W takim przypadku zdarzenie `Resolving` nie zostałoby wyzwolone i zostałyby załadowane zestawy *foo* i *bar* aplikacji. Innymi słowy, nie udałoby nam się osiągnąć **izolacji**.

Nasza klasa `FolderBasedALC` dobrze się nadaje do zaprezentowania koncepcji znajdowania zestawów, ale w prawdziwym programie nie byłoby z niej zbyt wiele pożytku, ponieważ nie obsługuje zależności platformy i (w przypadku projektów bibliotek) NuGet. W punkcie „Klasa AssemblyDependencyResolver" opisujemy rozwiązanie tego problemu, a w punkcie „Tworzenie systemu wtyczek" pokazujemy szczegółowy przykład.

Domyślny kontekst ALC

W chwili uruchomienia aplikacji CLR przypisuje specjalny kontekst ALC statycznej własności `AssemblyLoadContext.Default`. Domyślny kontekst ALC to miejsce, w którym jest ładowany zestaw rozruchowy wraz z jego statycznymi zależnościami i zestawami platformy .NET Core.

Domyślny kontekst ALC zaczyna szukanie zestawów od *sondowania ścieżek domyślnych*, aby automatycznie załadować potrzebne zestawy (zobacz podrozdział „Domyślne sondowanie"). W typowej sytuacji przegląda lokalizacje określone w plikach *.deps.json* i *.runtimeconfig.json*.

Jeśli ALC nie znajdzie zestawu na domyślnych ścieżkach sondowania, następuje zgłoszenie jego zdarzenia `Resolving`. Obsługując je, możemy załadować zestaw z innych lokalizacji, co oznacza, że zależności aplikacji możemy zapisać w innych katalogach, np. podfolderach, folderach współdzielonych, a nawet w postaci zasobu binarnego w zestawie hostowym:

```
AssemblyLoadContext.Default.Resolving += (loadContext, assemblyName) =>
{
  // Próbuje znaleźć assemblyName i zwraca obiekt Assembly lub null.
  // Po znalezieniu pliku zazwyczaj wywołuje się LoadFromAssemblyPath.
  // ...
};
```

Zdarzenie `Resolving` w domyślnym kontekście zostaje zgłoszone także wtedy, gdy niestandardowy kontekst ALC nie zdoła znaleźć zasobu (czyli kiedy jego metoda `Load` zwróci `null`), po czym ta sztuka nie uda się również kontekstowi domyślnemu.

Zestawy do domyślnego kontekstu ALC można też ładować poza zdarzeniem `Resolving`. Zanim jednak skorzystasz z tej możliwości, dokładnie sprawdź, czy nie ma lepszego rozwiązania, polegającego na użyciu osobnego kontekstu ALC lub technik opisanych w następnej sekcji (z wykorzystaniem *wykonawczych* i *kontekstowych* kontekstów ALC). Sztywne zakodowanie domyślnego ALC czyni program wątłym, ponieważ zabiera możliwość jego izolacji jako całości (np. przez systemy testów jednostkowych lub LINQPad).

Jeśli chcesz kontynuować mimo to, lepiej wywołaj *metodę znajdującą* (czyli `LoadFromAssemblyName`) zamiast *ładującej* (np. `LoadFromAssemblyPath`) — w szczególności jeśli zestaw jest dołączany statycznie. Wynika to z faktu, że dany zestaw może być już załadowany, w którym to przypadku metoda `LoadFromAssemblyName` zwróci ten już załadowany zestaw, a `LoadFromAssemblyPath` zgłosi wyjątek.

(Użycie metody `LoadFromAssemblyPath` niesie ze sobą ryzyko załadowania zestawu z innego miejsca niż to, w którym znalazłby go domyślny mechanizm znajdowania zestawów kontekstu ALC).

Jeśli zestaw znajduje się w miejscu, w którym nie zostanie automatycznie znaleziony przez ALC, to nadal możesz postępować zgodnie z tą procedurą i dodać procedurę obsługi zdarzenia `Resolving` kontekstu ALC.

Pamiętaj, że w wywołaniu metody `LoadFromAssemblyName` nie musisz określać pełnej nazwy, wystarczy prosta (nawet gdy zestaw ma silną nazwę):

```
AssemblyLoadContext.Default.LoadFromAssemblyName ("System.Xml");
```

Jeśli jednak dodasz do nazwy token klucza publicznego, to musi się zgadzać z załadowanym.

Domyślne sondowanie

Domyślne ścieżki sondowania to:

- Ścieżki określone w pliku *NazwaAplikacji.deps.json* (*NazwaAplikacji* to nazwa głównego zestawu Twojej aplikacji). Jeśli nie ma tego pliku, zostaje użyty folder bazowy aplikacji.
- Foldery zawierające zestawy platformy .NET Core (jeśli aplikacja z nich korzysta).

MSBuild automatycznie generuje plik o nazwie *NazwaAplikacji.deps.json*, zawierający informacje na temat tego, gdzie należy szukać wszystkich zależności. Dotyczy to zestawów niezależnych od platformy, które znajdują się w folderze bazowym aplikacji, i zestawów platformy, które znajdują się w podkatalogu *runtimes*\, np. w podfolderze *win* albo *unix*.

Ścieżki określone w wygenerowanym pliku *.deps.json* są względne w odniesieniu do folderu bazowego aplikacji — lub dowolnych innych folderów określonych w sekcji `additionalProbingPaths` w plikach konfiguracyjnych *AppName.runtimeconfig.json* i/lub *AppName.runtimeconfig.dev.json* (drugi jest przeznaczony tylko dla środowiska roboczego).

Bieżący kontekst ALC

W poprzedniej sekcji ostrzegaliśmy przed bezpośrednim ładowaniem zestawów do domyślnego kontekstu ALC. Zamiast tego zazwyczaj powinno się ładować/znajdować zestawy do bieżącego kontekstu ALC.

W większości przypadków bieżący kontekst ALC to ten, który zawiera obecnie wykonywany zestaw:

```
var executingAssem = Assembly.GetExecutingAssembly();
var alc = AssemblyLoadContext.GetLoadContext (executingAssem);

Assembly assem = alc.LoadFromAssemblyName (...); // znajdowanie wg nazwy
                 // LUB: = alc.LoadFromAssemblyPath (...); // znajdowanie po ścieżce
```

A oto bardziej elastyczny i wyrażony wprost sposób uzyskania ALC:

```
var myAssem = typeof (SomeTypeInMyAssembly).Assembly;
var alc = AssemblyLoadContext.GetLoadContext (myAssem);
...
```

Czasami nie da się określić bieżącego kontekstu ALC. Powiedzmy, że kazano Ci napisać serializator binarny .NET Core, który opisaliśmy w rozdziale 17. Taki serializator zapisuje pełne nazwy typów, które serializuje (włącznie z nazwami ich zestawów) i które muszą zostać rozwiązane w trakcie deserializacji. Powstaje pytanie, którego ALC użyć. Problem z wykonywanym zestawem polega na tym, że zwróci zestaw zawierający deserializator, a nie ten, który **wywołał** deserializator.

Najlepiej nie zgadywać, tylko zapytać:

```
public object Deserialize (Stream stream, AssemblyLoadContext alc)
{
  ...
}
```

Bezpośrednie wyrażenie intencji zwiększa elastyczność i minimalizuje ryzyko popełnienia błędu. Teraz wywołujący może zdecydować, co ma być traktowane jako „bieżący" kontekst ALC:

```
var assem = typeof (SomeTypeThatIWillBeDeserializing).Assembly;
var alc = AssemblyLoadContext.GetLoadContext (assem);
var object = Deserialize (someStream, alc);
```

Metoda Assembly.Load i kontekstowe ALC

Do pomocy w często wykonywanej czynności ładowania zestawu do bieżącego ALC, tzn.:

```
var executingAssem = Assembly.GetExecutingAssembly();
var alc = AssemblyLoadContext.GetLoadContext (executingAssem);
Assembly assem = alc.LoadFromAssemblyName (...);
```

firma Microsoft zdefiniowała następującą metodę w klasie `Assembly`:

```
public static Assembly Load (string assemblyString);
```

jak również identyczną pod względem funkcjonalnym wersję przyjmującą obiekt `AssemblyName`:

```
public static Assembly Load (AssemblyName assemblyRef);
```

(Nie myl tych metod ze starą metodą `Load(byte[])`, która zachowuje się całkiem inaczej — zobacz podrozdział „Stare metody ładujące").

Tak samo jak w przypadku metody `LoadFromAssemblyName`, możesz podać prostą, częściową lub pełną nazwę zestawu:

```
Assembly a = Assembly.Load ("System.Private.Xml");
```

To spowoduje załadowanie zestawu `System.Private.Xml` do kontekstu ALC, w którym jest załadowany **zestaw wykonywanego kodu**.

W tym przypadku podaliśmy prostą nazwę, choć w .NET Core 3 równie dobre byłyby wszystkie poniższe łańcuchy:

```
"System.Private.Xml, PublicKeyToken=cc7b13ffcd2ddd51"
"System.Private.Xml, Version=4.0.1.0"
"System.Private.Xml, Version=4.0.1.0, PublicKeyToken=cc7b13ffcd2ddd51"
```

Jeśli podasz token klucza publicznego, musi on odpowiadać temu, który jest załadowany.

Microsoft Developer Network (MSDN) ostrzega przed ładowaniem zestawów za pomocą częściowych nazw oraz zaleca określanie dokładnego numeru wersji i tokena klucza publicznego. Argumentacja wspierająca to zalecenie dotyczy czynników związanych z platformą .NET Framework, takich jak skutek dla globalnego bufora zestawów i bezpieczeństwa dostępu do kodu. W .NET Core nie mają one znaczenia i ładowanie za pomocą prostych i częściowych nazw jest generalnie bezpieczne.

Obie te metody służą ściśle do znajdowania zasobów, a więc nie można im podać ścieżki. (Jeśli nadasz wartość własności `CodeBase` obiektu `AssemblyName`, to zostanie zignorowana).

Nie wpadnij w pułapkę polegającą na użyciu metody `Assembly.Load` do ładowania statycznie dołączanego zestawu. W takim przypadku wystarczy się odwołać do typu znajdującego się w tym zestawie i w ten sposób uzyskać zestaw:

```
Assembly a = typeof (System.Xml.Formatting).Assembly;
```

Możesz to nawet zrobić tak:

```
Assembly a = System.Xml.Formatting.Indented.GetType().Assembly;
```

To pozwala uniknąć wpisywania nazwy zestawu (która w przyszłości może się zmienić) przy jednoczesnym uruchomieniu procedury znajdowania zestawu w kontekście ALC *kodu, który jest wykonywany* (co miałoby miejsce w razie użycia metody `Assembly.Load`).

Gdybyśmy mieli samodzielnie napisać metodę `Assembly.Load`, wyglądałaby (prawie) tak:

```
[MethodImpl(MethodImplOptions.NoInlining)]
Assembly Load (string name)
{
  Assembly callingAssembly = Assembly.GetCallingAssembly();
  var callingAlc = AssemblyLoadContext.GetLoadContext (callingAssembly);
  return callingAlc.LoadFromAssemblyName (new AssemblyName (name));
}
```

Metoda EnterContextualReflection

Strategia metody `Assembly.Load`, która polega na wykorzystaniu kontekstu ALC wywołującego zestawu, zawodzi, gdy metoda ta zostaje wywołana przez jakiegoś pośrednika, np. deserializator lub system wykonawczy testów jednostkowych. Jeśli ten pośrednik jest zdefiniowany w innym zestawie, to zostanie użyty jego kontekst ładowania zamiast kontekstu wywołującego.

Tę sytuację już opisaliśmy w części dotyczącej pisania deserializatora. W takich przypadkach najlepszym wyjściem jest zmusić wywołującego do określenia kontekstu ALC, zamiast dedukować go za pomocą metody `Assembly.Load(string)`.

Ponieważ jednak platforma .NET Core wyewoluowała z .NET Framework — na której izolację uzyskiwano dzięki domenom aplikacji, a nie kontekstom ALC — idealne rozwiązanie nie jest powszechne i czasami metoda `Assembly.Load(string)` jest używana nieprawidłowo w sytuacjach, w których ALC nie można niezawodnie wydedukować. Jako przykład może posłużyć serializator binarny platformy .NET Core.

Aby umożliwić używanie metody `Assembly.Load` także w takich sytuacjach, firma Microsoft dodała do klasy `AssemblyLoadContext` metodę o nazwie `EnterContextualReflection`, która przypisuje kontekst ALC do własności `AssemblyLoadContext.CurrentContextualReflectionContext`. Choć jest to własność statyczna, jej wartość jest przechowywana w zmiennej `AsyncLocal`, a więc może przechowywać różne wartości w różnych wątkach (ale być zachowana w operacjach asynchronicznych).

Jeśli ta własność nie ma wartości `null`, metoda `Assembly.Load` automatycznie preferuje ją zamiast wywołującego kontekstu ALC:

```
Method1();

var myALC = new AssemblyLoadContext ("test");
using (myALC.EnterContextualReflection())
{
    Console.WriteLine (
        AssemblyLoadContext.CurrentContextualReflectionContext.Name); // test
    Method2();
}
// po usunięciu EnterContextualReflection() nie wywołuje efektu
Method3();

void Method1() => Assembly.Load ("..."); // użyje wywołującego ALC
void Method2() => Assembly.Load ("..."); // użyje myALC
void Method3() => Assembly.Load ("..."); // użyje wywołującego ALC
```

Wcześniej pokazaliśmy, jak napisać metodę o funkcjonalności podobnej do metody `Assembly.Load`. Poniżej znajduje się ulepszona wersja, uwzględniająca refleksję kontekstową:

```
[MethodImpl(MethodImplOptions.NoInlining)]
Assembly Load (string name)
{
  var alc = AssemblyLoadContext.CurrentContextualReflectionContext
      ?? AssemblyLoadContext.GetLoadContext (Assembly.GetCallingAssembly());
  return alc.LoadFromAssemblyName (new AssemblyName (name));
}
```

Choć kontekst refleksji kontekstowej może umożliwić wykonywanie starego kodu, bardziej niezawodne rozwiązanie (zgodnie z tym, co napisaliśmy wcześniej) polega na modyfikacji kodu wywołującego metodę `Assembly.Load`, aby wywoływał metodę `LoadFromAssemblyName` na kontekście ALC przekazanym przez wywołującego.

Na platformie .NET Framework nie ma odpowiednika metody `EnterContextualRe`
`↪flection` — i nie jest on potrzebny — mimo że są te same metody `Assembly.Load`. Wynika to z tego, że na platformie .NET Framework izolację uzyskuje się przede wszystkim przy użyciu domen aplikacji, a nie kontekstów ALC. Domeny aplikacji stanowią silniejszy model izolacji, w ramach którego każda domena ma własny domyślny kontekst ładowania, dzięki czemu izolacja działa nawet wtedy, gdy używany jest domyślny kontekst ładowania.

Ładowanie i znajdowanie bibliotek niezarządzanych

Konteksty ALC mogą ładować i znajdować biblioteki macierzyste. Taka procedura znajdowania jest uruchamiana przez wywołanie zewnętrznej metody z atrybutem `[DllImport]`:

```
[DllImport ("SomeNativeLibrary.dll")]
static extern int SomeNativeMethod (string text);
```

Dzięki temu, że w atrybucie `[DllImport]` nie podaliśmy pełnej ścieżki, wywołanie metody `SomeNativeMethod` uruchomi procedurę znajdowania w kontekście ALC zawierającym zestaw, w którym jest zdefiniowana ta metoda.

Wirtualna metoda *znajdująca* nazywa się `LoadUnmanagedDll`, a metoda ładująca ma nazwę `LoadUnmanagedDllFromPath`:

```
protected override IntPtr LoadUnmanagedDll (string unmanagedDllName)
{
  // znajduje pełną ścieżkę do unmanagedDllName...
  string fullPath = ...
  return LoadUnmanagedDllFromPath (fullPath); // ładuje DLL
}
```

Jeśli nie możesz znaleźć pliku, możesz zwrócić `IntPtr.Zero`. Wówczas CLR zgłosi zdarzenie ALC `ResolvingUnmanagedDll`.

Co ciekawe, metoda `LoadUnmanagedDllFromPath` jest chroniona, a więc w większości przypadków nie będziesz mieć możliwości jej wywołania w procedurze obsługi zdarzeń `ResolvingUnmanagedDll`. Ten sam efekt uzyskasz jednak przez wywołanie statycznej metody `NativeLibrary.Load`:

```
someALC.ResolvingUnmanagedDll += (requestingAssembly, unmanagedDllName) =>
{
  return NativeLibrary.Load ("(full path to unmanaged DLL)");
};
```

Choć biblioteki macierzyste są z reguły znajdowane i ładowane przez konteksty ALC, nie „należą” one do ALC. Załadowana biblioteka macierzysta pozostaje samodzielna i sama odpowiada za znajdowanie wszystkich swoich zależności. Ponadto biblioteki macierzyste mają globalny zakres dostępności w procesie, dzięki czemu nie ma możliwości załadowania dwóch różnych wersji takiej biblioteki, jeśli mają taką samą nazwę pliku.

Klasa AssemblyDependencyResolver

W podrozdziale „Domyślne sondowanie” napisaliśmy, że domyślny kontekst ALC wczytuje pliki *.deps.json* i *.runtimeconfig.json*, jeśli są obecne, aby się dowiedzieć, gdzie szukać zależności specyficznych dla platformy i roboczych pakietów NuGet.

Jeśli chcesz załadować zestaw do własnego kontekstu ALC mającego zależności specyficzne dla platformy lub w postaci pakietów NuGet, musisz w jakiś sposób odtworzyć tę logikę. W tym celu możesz np. parsować pliki konfiguracyjne i ściśle przestrzegać obowiązujących na platformie zasad dotyczących nazw. To jednak jest trudne i na dodatek wystarczy zmiana zasad w przyszłej wersji platformy .NET Core, aby kod przestał działać.

Rozwiązaniem tego problemu jest klasa `AssemblyDependencyResolver`. Należy utworzyć jej obiekt przy użyciu ścieżki do zestawu, którego zależności chce się wysondować:

```
var resolver = new AssemblyDependencyResolver (@"c:\temp\foo.dll");
```

Następnie, aby znaleźć ścieżkę zależności, wywołujemy metodę `ResolveAssemblyToPath`:

```
string path = resolver.ResolveAssemblyToPath (new AssemblyName ("bar"));
```

Przy braku pliku *.deps.json* (lub jeśli ten plik istnieje, ale nie zawiera żadnych informacji dotyczących *bar.dll*) ten kod da nam ścieżkę *c:\temp\bar.dll*.

Podobnie można znajdować niezarządzane zależności za pomocą metody `ResolveUnmanagedDllToPath`.

Doskonałym przykładem bardziej złożonego przypadku jest utworzenie nowego projektu konsolowego o nazwie *ClientApp* i dodanie odwołania NuGet do *Microsoft.Data.SqlClient*. Dodaj poniższą klasę:

```
using Microsoft.Data.SqlClient;

namespace ClientApp
{
  public class Program
  {
    public static SqlConnection GetConnection() => new SqlConnection();
    static void Main() => GetConnection(); // sprawdzenie skuteczności znajdowania
  }
}
```

Teraz skompiluj aplikację i zajrzyj do folderu wynikowego — znajdziesz w nim plik o nazwie *Microsoft.Data.SqlClient.dll*. Jednak *nigdy się on nie załaduje* po uruchomieniu, a próba ręcznego załadowania spowoduje zgłoszenie wyjątku. Zestaw, który go ładuje, znajduje się w podfolderze *runtimes\win* (lub *runtimes/unix*). Domyślny kontekst ALC wie, że trzeba go załadować, ponieważ wczytuje plik *ClientApp.deps.json*.

Gdybyśmy chcieli załadować plik *ClientApp.dll* z innej aplikacji, musielibyśmy napisać kontekst ALC, który potrafiłby znaleźć jego zależność *Microsoft.Data.SqlClient.dll*. Do tego nie wystarczyłoby zajrzenie do folderu, w którym znajduje się plik *ClientApp.dll* (jak to zrobiliśmy w podrozdziale „Znajdowanie zestawów”). Musielibyśmy użyć klasy `AssemblyDependencyResolver`, aby określić położenie tego pliku dla używanej platformy:

```
string path = @"C:\source\ClientApp\bin\Debug\netcoreapp3.0\ClientApp.dll";
var resolver = new AssemblyDependencyResolver (path);
var sqlClient = new AssemblyName ("Microsoft.Data.SqlClient");
Console.WriteLine (resolver.ResolveAssemblyToPath (sqlClient));
```

W systemie Windows wynik będzie następujący:

```
C:\source\ClientApp\bin\Debug\netcoreapp3.0\runtimes\win\lib\netcoreapp2
\Microsoft.Data.SqlClient.dll
```

Kompletny przykład przedstawiamy w podrozdziale „Tworzenie systemu wtyczek”.

Usuwanie załadowanych kontekstów ALC

W prostych przypadkach można usunąć załadowany niedomyślny kontekst `AssemblyLoadContext`, aby zwolnić pamięć i blokady plików załadowanych przez niego zestawów. Aby się to udało, obiekt danego kontekstu musi być utworzony z parametrem `isCollectible` ustawionym na wartość `true`:

```
var alc = new AssemblyLoadContext ("test", isCollectible:true);
```

Wtedy można wywołać metodę `Unload` na ALC, aby rozpocząć proces usuwania.

Model usuwania załadowanego kontekstu ma charakter kooperacyjny, tzn. jeśli w którymkolwiek jego zestawie będzie wykonywana jakakolwiek metoda, to usunięcie nastąpi dopiero po zakończeniu jej działania.

Rzeczywiste usunięcie następuje dopiero podczas usuwania nieużytków. Nie powiedzie się, jeśli cokolwiek poza danym kontekstem ALC będzie miało jakiekolwiek (inne niż słabe) odwołanie do czegokolwiek znajdującego się w tym kontekście (wliczając obiekty, typy i zestawy). Interfejsy API (także należące do platformy .NET Core) często buforują obiekty w statycznych polach lub słownikach — albo subskrybują zdarzenia — co stwarza warunki, które sprzyjają powstawaniu odwołań uniemożliwiających usunięcie kontekstu, szczególnie gdy kod w tym kontekście używa interfejsów API spoza niego w zaawansowany sposób. Znalezienie przyczyny niepowodzenia w usuwaniu kontekstu jest trudne i wymaga użycia specjalnych narzędzi, takich jak WinDbg.

Stare metody ładujące

Jeśli wciąż używasz .NET Framework (albo piszesz bibliotekę pod kątem .NET Standard i chcesz obsługiwać .NET Framework), to nie możesz korzystać z klasy `AssemblyLoadContext`. W takiej sytuacji do ładowania należy używać następujących metod:

```
public static Assembly LoadFrom (string assemblyFile);
public static Assembly LoadFile (string path);
public static Assembly Load (byte[] rawAssembly);
```

Metody `LoadFile` i `Load(byte[])` zapewniają izolację, natomiast `LoadFrom` — nie.

Do znajdowania zależności służy zdarzenie domeny aplikacji `AssemblyResolve`, które funkcjonuje podobnie do zdarzenia `Resolving` domyślnego ALC.

Dostępna jest też metoda `Assembly.Load(string)`, uruchamiająca procedurę znajdowania, która działa w podobny sposób.

Metoda LoadFrom

Metoda `LoadFrom` ładuje zestaw z określonej ścieżki do domyślnego ALC. Działaniem przypomina metodę `AssemblyLoadContext.Default.LoadFromAssemblyPath`, ale różni się od niej w następujących kwestiach:

- Jeśli zestaw o takiej samej prostej nazwie znajduje się już w domyślnym kontekście ALC, metoda `LoadFrom` zwraca ten zestaw, zamiast zgłosić wyjątek.
- Jeśli w domyślnym kontekście ALC nie ma zestawu o takiej samej prostej nazwie i nastąpi załadowanie, zestaw ten otrzymuje specjalny status `LoadFrom`. Ma on wpływ na logikę znajdowania domyślnego kontekstu ALC polegający na tym, że jeśli dany zestaw ma jakiekolwiek zależności w tym samym folderze, to zostaną one automatycznie znalezione.

Platforma .NET Framework ma *globalny bufor zestawów* (ang. Global Assembly Cache — GAC). Jeśli zestaw znajduje się w tym buforze, CLR zawsze ładuje go stamtąd. Dotyczy to wszystkich trzech metod ładujących.

Funkcja automatycznego znajdowania zależności znajdujących się w tym samym folderze metody `LoadFrom` może być bardzo przydatna, dopóki nie załaduje zestawu, którego nie powinna. Znalezienie takiego błędu często jest bardzo trudne, dlatego lepiej jest posługiwać się metodami `Load(string)` i `LoadFile`, a do znajdowania zależności używać zdarzenia `AssemblyResolve` domeny aplikacji. To daje nam możliwość decydowania o sposobie znalezienia każdego zestawu i umożliwia debugowanie (przez utworzenie punktu wstrzymania w procedurze obsługi zdarzenia).

Metody LoadFile i Load(byte[])

Metody `LoadFile` i `Load(byte[])` ładują zestaw z podanej ścieżki lub tablicy bajtów do nowego kontekstu ALC. W odróżnieniu od `LoadFrom` te metody zapewniają izolację i umożliwiają załadowanie wielu wersji jednego zestawu. Są jednak dwa haczyki:

- Wywołanie metody `LoadFile` po raz drugi z identyczną ścieżką spowoduje zwrócenie wcześniej załadowanego zestawu.
- Na platformie .NET Framework obie metody najpierw sprawdzają GAC i jeśli tam znajdą zestaw, ładują go z bufora.

Metody `LoadFile` i `Load(byte[])` tworzą osobny kontekst dla każdego zestawu. Z jednej strony umożliwia to izolację, a z drugiej może utrudniać zarządzanie.

Do znajdowania zależności służy zdarzenie `Resolving` `AppDomain`, które jest zgłaszane we wszystkich kontekstach ALC:

```
AppDomain.CurrentDomain.AssemblyResolve += (sender, args) =>
{
  string fullAssemblyName = args.Name;
  // zwraca obiekt Assembly lub null
  ...
};
```

Zmienna `args` zawiera też własność o nazwie `RequestingAssembly`, informującą, który zestaw uruchomił procedurę znajdowania.

Po znalezieniu zestawu można go załadować za pomocą metody `Assembly.LoadFile`.

Za pomocą metody `AppDomain.CurrentDomain.GetAssemblies()` można uzyskać listę wszystkich zestawów załadowanych do bieżącej domeny aplikacji. Taka sama możliwość istnieje na platformie .NET Core, na której należy zastosować taki kod:

```
AssemblyLoadContext.All.SelectMany (a => a.Assemblies)
```

Tworzenie systemu wtyczek

Aby dokładnie zademonstrować koncepcje przedstawione w tym podrozdziale, napiszemy system wtyczek wykorzystujący nie dające się usunąć konteksty do izolacji wtyczek.

Nasz system demonstracyjny początkowo będzie składał się z trzech projektów .NET Core:

Plugin.Common (biblioteka)

Definiuje interfejs, który będą implementować wtyczki.

Capitalizer (biblioteka)

Wtyczka zamieniająca litery na wielkie.

Plugin.Host (aplikacja konsolowa)

Znajduje i wywołuje wtyczki.

Projekty te znajdują się w następujących katalogach:

```
c:\source\PluginDemo\Plugin.Common
c:\source\PluginDemo\Capitalizer
c:\source\PluginDemo\Plugin.Host
```

Wszystkie odwołują się do biblioteki `Plugin.Common` i poza tym nie będzie żadnych innych odwołań między projektami.

Gdyby aplikacja Plugin.Host odwoływała się do biblioteki Capitalizer, nie tworzylibyśmy systemu wtyczek. Podstawową ideą jest przecież to, że wtyczki są pisane przez strony trzecie już po opublikowaniu Plugin.Host i Plugin.Common.

Jeśli używasz Visual Studio, to dla wygody wszystkie trzy projekty możesz umieścić w jednym rozwiązaniu. Jeśli to zrobisz, kliknij prawym przyciskiem myszy projekt *Plugin.Host*, wybierz opcję *Build Dependencies/Project Dependencies* (zależności kompilacji/zależności projektu), a następnie kliknij pozycję *Capitalizer*. To wymusi kompilację wtyczki Capitalizer po uruchomieniu projektu *Plugin.Host* bez dodawania odwołania.

Plugin.Common

Zaczniemy od biblioteki Plugin.Common. Nasze wtyczki będą wykonywać bardzo proste zadanie polegające na przekształcaniu łańcucha. Poniżej znajduje się definicja naszego interfejsu:

```
namespace Plugin.Common
{
  public interface ITextPlugin
  {
    string TransformText (string input);
  }
}
```

To wszystko, jeśli chodzi o Plugin.Common.

Capitalizer (wtyczka)

Nasza wtyczka Capitalizer będzie się odwoływać do Plugin.Common i zawiera jedną klasę. Na razie jej logika będzie prosta i pozbawiona dodatkowych zależności:

```
public class CapitalizerPlugin : Plugin.Common.ITextPlugin
{
  public string TransformText (string input) => input.ToUpper();
}
```

Jeśli skompilujesz oba projekty i zajrzysz do folderu wynikowego projektu *Capitalizer*, znajdziesz w nim dwa następujące zestawy:

```
Capitalizer.dll // zestaw wtyczki
Plugin.Common.dll // dołączany zestaw
```

Plugin.Host

Plugin.Host to aplikacja konsolowa składająca się z dwóch klas. Pierwsza klasa to niestandardowy kontekst ALC ładujący wtyczki:

```
class PluginLoadContext : AssemblyLoadContext
{
  AssemblyDependencyResolver _resolver;

  public PluginLoadContext (string pluginPath, bool collectible)
    // wybierz łatwą do zapamiętania nazwę, aby ułatwić sobie debugowanie:
    : base (name: Path.GetFileName (pluginPath), collectible)
  {
    // resolver pomocny w znajdowanie zależności
    _resolver = new AssemblyDependencyResolver (pluginPath);
  }

  protected override Assembly Load (AssemblyName assemblyName)
  {
    // patrz niżej
    if (assemblyName.Name == typeof (ITextPlugin).Assembly.GetName().Name)
      return null;

    string target = _resolver.ResolveAssemblyToPath (assemblyName);

  if (target != null)
    return LoadFromAssemblyPath (target);

  // może być zestaw platformy; pozwala na znajdowanie kontekstowi domyślnemu
  return null;
}

protected override IntPtr LoadUnmanagedDll (string unmanagedDllName)
{
  string path = _resolver.ResolveUnmanagedDllToPath (unmanagedDllName);

  return path == null
    ? IntPtr.Zero
    : LoadUnmanagedDllFromPath (path);
  }
}
```

W konstruktorze przekazujemy ścieżkę do głównego zestawu wtyczki i flagę określającą, czy kontekst ALC ma podlegać systemowi usuwania nieużytków (aby można go było usunąć).

Metoda `Load` zawiera logikę znajdowania zależności. Wszystkie wtyczki muszą odwoływać się do Plugin.Common, aby implementować interfejs `ITextPlugin`. To znaczy, że metoda `Load` zostanie w pewnym momencie uruchomiona w celu znalezienia Plugin.Common. Musimy być ostrożni, ponieważ folder wynikowy wtyczki może zawierać nie tylko plik *Capitalizer.dll*, ale również własną kopię pliku *Plugin.Common.dll*. Gdybyśmy ją załadowali do kontekstu `PluginLoadContext`,

mielibyśmy dwie kopie tego zestawu: jedną w domyślnym kontekście hosta i drugą w kontekście `PluginLoadContext` wtyczki. Te zestawy byłyby niekompatybilne ze sobą i host zgłaszałby wyjątek, informując, że wtyczka nie implementuje interfejsu `ITextPlugin`!

Aby uniknąć tego problemu, dodajemy następujący warunek:

```
if (assemblyName.Name == typeof (ITextPlugin).Assembly.GetName().Name)
return null;
```

Zwrot wartości `null` sprawia, że wyszukaniem zestawu zajmie się domyślny kontekst ALC hosta.

Zamiast `null` możemy zwrócić `typeof(ITextPlugin).Assembly` i też będzie dobrze. Skąd pewność, że wyszukiwanie `ITextPlugin` odbędzie się w kontekście ALC hosta, a nie w naszym kontekście `PluginLoadContext`? Przypomnijmy, że klasę `PluginLoadContext` zdefiniowaliśmy w zestawie *Plugin.Host*. W związku z tym wszystkie typy, do których odwołujemy się statycznie z tej klasy, będą uruchamiać procedurę znajdowania zestawu w kontekście ALC, do którego został załadowany jej *zestaw — Plugin.Host*.

Po sprawdzeniu wspólnego zestawu za pomocą metody `AssemblyDependencyResolver` szukamy prywatnych zależności, jakie może mieć wtyczka. (Na razie nie ma żadnych).

Zwróć uwagę, że przesłoniliśmy też metodę `LoadUnmanagedDll`, aby mieć pewność, że w razie gdyby wtyczka miała jakieś niezarządzane zależności, to one także zostaną poprawnie załadowane.

Druga klasa w zestawie *Plugin.Host* to program główny. Dla uproszczenia wpiszemy ścieżkę do wtyczki Capitalizer wprost do kodu (w prawdziwym programie moglibyśmy znajdować ścieżki do wtyczek przez wyszukanie plików DLL w znanych lokalizacjach lub wczytanie pliku konfiguracyjnego):

```
class Program
{
  const bool UseCollectibleContexts = true;

  static void Main()
  {
    const string captializer = @"C:\source\PluginDemo\"
      + @"Capitalizer\bin\Debug\netcoreapp3.0\Capitalizer.dll";

    Console.WriteLine (TransformText ("big apple", captializer));
  }

  static string TransformText (string text, string pluginPath)
  {
    var alc = new PluginLoadContext (pluginPath, UseCollectibleContexts);
    try
    {
      Assembly assem = alc.LoadFromAssemblyPath (pluginPath);

      // lokalizuje typ w zestawie implementującym ITextPlugin
      Type pluginType = assem.ExportedTypes.Single (t =>
                                  typeof (ITextPlugin).IsAssignableFrom (t));

      // tworzy egzemplarz implementacji ITextPlugin
      var plugin = (ITextPlugin)Activator.CreateInstance (pluginType);
```

```
            // wywołanie metody TransformText
            return plugin.TransformText (text);
        }
        finally
        {
            if (UseCollectibleContexts) alc.Unload(); // usuwa ALC
        }
    }
}
```

Przyjrzyjmy się metodzie `TransformText`. Najpierw tworzymy nowy kontekst ALC dla naszej wtyczki, a następnie nakazujemy mu załadowanie głównego zestawu wtyczki. Potem za pomocą refleksji znajdujemy typ implementujący interfejs `ITextPlugin` (ten temat szczegółowo opisujemy w rozdziale 19.). Później tworzymy egzemplarz wtyczki, wywołujemy metodę `TransformText` i usuwamy ALC.

Jeśli chcesz kilka razy wywołać metodę `TransformText`, to lepiej wyślij kontekst ALC do bufora, zamiast go usuwać po każdym wywołaniu.

Wynik:

```
BIG APPLE
```

Dodawanie zależności

Nasz kod jest w pełni zdolny do rozpoznawania i izolowania zależności. Aby to zilustrować, najpierw dodamy odwołanie NuGet do pakietu Humanizer.Core w wersji 2.6.2. Możesz to zrobić w Visual Studio lub przez dodanie poniższego elementu do pliku *Capitalizer.csproj*:

```
<ItemGroup>
  <PackageReference Include="Humanizer.Core" Version="2.6.2" />
</ItemGroup>
```

Teraz zmodyfikujemy klasę `CapitalizerPlugin`:

```
using Humanizer;
namespace Capitalizer
{
  public class CapitalizerPlugin : Plugin.Common.ITextPlugin
  {
    public string TransformText (string input) => input.Pascalize();
  }
}
```

Teraz wynik programu byłby następujący:

```
BigApple
```

Następnie utworzymy kolejną wtyczkę o nazwie Pluralizer. Utwórz nowy projekt biblioteki .NET Core i dodaj odwołanie NuGet do pakietu Humanizer.Core w wersji 2.7.9:

```
<ItemGroup>
  <PackageReference Include="Humanizer.Core" Version="2.7.9" />
</ItemGroup>
```

Teraz dodaj klasę o nazwie `PluralizerPlugin`. Będzie podobna do klasy `CapitalizerPlugin`, tylko będzie wywoływała metodę `Pluralize`:

```
using Humanizer;
namespace Pluralizer
{
  public class PluralizerPlugin : Plugin.Common.ITextPlugin
  {
    public string TransformText (string input) => input.Pluralize();
  }
}
```

Na koniec dodajemy do metody `Main` zestawu Plugin.Host kod ładujący i uruchamiający wtyczkę Pluralizer:

```
static void Main()
{
  const string captializer = @"C:\source\PluginDemo\"
    + @"Capitalizer\bin\Debug\netcoreapp3.0\Capitalizer.dll";

  Console.WriteLine (TransformText ("big apple ", captializer));

  const string pluralizer = @"C:\source\PluginDemo\"
    + @"Pluralizer\bin\Debug\netcoreapp3.0\Pluralizer.dll";
  Console.WriteLine (TransformText ("big apple", pluralizer));
}
```

Teraz wynik będzie następujący:

```
BigApple
big apples
```

Aby w pełni zrozumieć, co się dzieje, zmień wartość stałej `UseCollectibleContexts` na `false` i dodaj poniższy kod do metody `Main`, aby wydrukować listę kontekstów ALC i ich zestawów:

```
foreach (var context in AssemblyLoadContext.All)
{
  Console.WriteLine ($"Kontekst: {context.GetType().Name} {context.Name}");

foreach (var assembly in context.Assemblies)
      Console.WriteLine ($" Zestaw: {assembly.FullName}");
}
```

W wyniku widać dwie wersje pakietu Humanizer, każda załadowana we własnym kontekście ALC:

```
Context: PluginLoadContext Capitalizer.dll
Assembly: Capitalizer, Version=1.0.0.0, Culture=neutral, PublicKeyToken=...
Assembly: Humanizer, Version=2.6.0.0, Culture=neutral, PublicKeyToken=...
Context: PluginLoadContext Pluralizer.dll
Assembly: Pluralizer, Version=1.0.0.0, Culture=neutral, PublicKeyToken=...
Assembly: Humanizer, Version=2.7.0.0, Culture=neutral, PublicKeyToken=...
Context: DefaultAssemblyLoadContext Default
Assembly: System.Private.CoreLib, Version=4.0.0.0, Culture=neutral,...
Assembly: Host, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
...
```

Nawet gdyby obie wtyczki używały tej samej wersji pakietu Humanizer, izolacja zestawów i tak byłaby korzystna, ponieważ każdy ma własne zmienne statyczne.

Sprawdź, jak w C# pracują najlepsi programiści!

C# jest od początku rozwijany w konsekwentny, przemyślany sposób, a każda nowa funkcjonalność idealnie integruje się z resztą języka. W efekcie łączy on nowoczesność i bezpieczeństwo. Wersja 8.0 jest kolejną poważną aktualizacją tego języka. Zapewnia wysokopoziomowe abstrakcje, między innymi wyrażenia, zapytania i kontynuacje asynchroniczne, ale także udostępnia niskopoziomowe mechanizmy pozwalające osiągnąć maksymalną wydajność aplikacji dzięki wykorzystaniu takich konstrukcji jak własne typy wartościowe programisty czy opcjonalne wskaźniki. Ceną, jaką programista płaci za ten rozwój, jest konieczność ciągłej nauki.

To kolejne, przejrzane i zaktualizowane wydanie doskonałego podręcznika dla programistów. Znalazły się tu zwięzłe i dokładne informacje na temat języka C#, Common Language Runtime (CLR) oraz platformy .NET Core. Zaprezentowano precyzyjne opisy pojęć i konkretne przypadki użycia, a poszczególne zagadnienia potraktowano dogłębnie i od strony praktycznej. Sporo uwagi poświęcono dość trudnym tematom, jak współbieżność, bezpieczeństwo i dostęp do funkcji systemu operacyjnego, przy czym nie zaniedbano kwestii czytelności i nie spłycono informacji. Nowe składniki języka C# 8.0 i związanej z nim platformy specjalnie oznaczono, dzięki czemu to wydanie może też służyć jako podręcznik do C# 7.0.

W książce między innymi:

- składnia C#, definiowanie zmiennych, wskaźniki, domknięcia i wzorce
- tajniki LINQ i praca na danych
- programowanie współbieżne i asynchroniczne
- praca z wątkami i programowanie równoległe
- narzędzia platformy .NET oraz kompilator Roslyn

Joseph Albahari jest autorem lubianych książek o C#. Napisał też popularny program dla programistów przeznaczony do roboczego wypróbowywania zapytań LINQ — LINQPad.

Eric Johannsen jest programistą i specjalistą w zakresie uczenia maszynowego. Ma duże doświadczenie w dziedzinie projektowania rozwiązań programistycznych dla nowych potrzeb i okazji biznesowych.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-283-7281-8
9 788328 372818

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 149,00 zł