

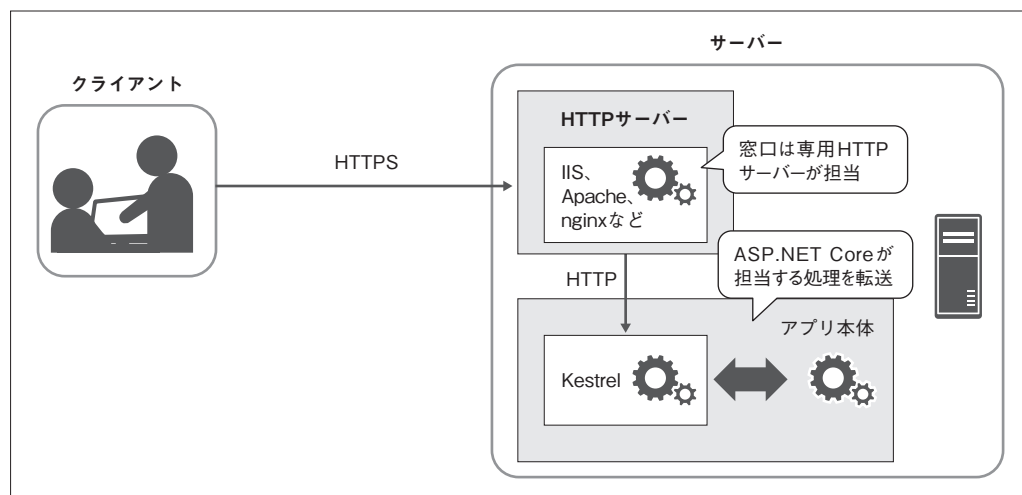
付録

A

本番環境への移行

前章までは、開発したアプリの動作を.NETに同梱されている組み込みサーバーKestrel上で確認してきました。実際、アプリの最低限の動作を確認するだけであれば、.NET + VSCodeをインストールするだけで事足ります。

ただし、KestrelはあくまでASP.NET Coreのためのコンパクトなサーバーです。HTTPサーバーとしては最小限の機能を提供しているにすぎません。一般的な本番環境では、IIS (Internet Informations Services)、nginx、ApacheのようなHTTPサーバーに窓口としての一次的な処理は委ね、Kestrelそのものはアプリとしての処理に徹するのがお勧めです (図App.1)。



❖図App.1 本番環境での一般的な構成

これらの環境は一から準備しても（もちろん）かまいませんが、昨今では前提となる環境をひとまとめに提供してくれるクラウドサービスが充実しています。本節でも、その代表的な1つである**Microsoft Azure**を例に、ASP.NET Coreアプリをデプロイ（配置）する手順を紹介します※¹。

A.1 Microsoft Azureとは？

Microsoft Azure（以降、Azure）とは、マイクロソフトが提供するクラウドプラットフォームです。執筆時点で200以上のサービスが提供されており、これらを組み合わせることで、種々の問題を解決できます。表App.1は、Azureで提供されている主なサービスです。

※1 もちろん、利用可能なデプロイ先はAzureに限りません。同じく、代表的なクラウドサービスであるAWS (Amazon Web Services)、GCP (Google Cloud) にデプロイすることも可能ですし、自らIIS、nginxなどのサーバーを立てて、環境を構築してもかまいません。はたまた、仮想環境の代表格Dockerコンテナも有効な選択肢の1つです。

❖表 App.1 Azureで提供されている主なサービス

サービス	概要
Azure App Service	Webアプリなどをデプロイする基盤となるサービス
Azure SQL Database	リレーショナルデータベースを提供するサービス
Azure Storage	多様な形式のデータを保存できるストレージサービス
Azure Cosmos DB	NoSQL データベースを提供するサービス
Azure Virtual Machines	仮想マシンを構築するサービス
Azure Virtual Network	仮想ネットワークを構築するサービス
Azure Kubernetes Service	Kubernetes が利用可能なサービス
Azure Machine Learning	機械学習のモデル構築や運用を行えるサービス
Azure Cognitive Services	AI サービスをアプリに組み込めるサービス
Azure DevOps	開発（Dev）運用（Ops）において効率的な共同作業を支援するツールをまとめたサービス
Azure Monitor	パフォーマンスやログなどの監視データを収集、分析し管理するサービス

執筆時点では無償評価版も提供されており、一定の利用範囲であれば無償で Azure の各種サービスを導入できます。 .NET + VSCode との親和性も高く、ごく簡単な操作でデプロイできるのもメリットです（前提となる環境があらかじめ用意されているので、 .NET をはじめとした各種ソフトウェアを一からインストールする必要もありません）。

なお、本書では、 Azure そのものについては守備範囲を超えるので、詳細は割愛します。詳しくは、以下のような連載も参考にしてください。

ゼロからはじめる Azure

<https://news.mynavi.jp/techplus/series/zeroazure/>

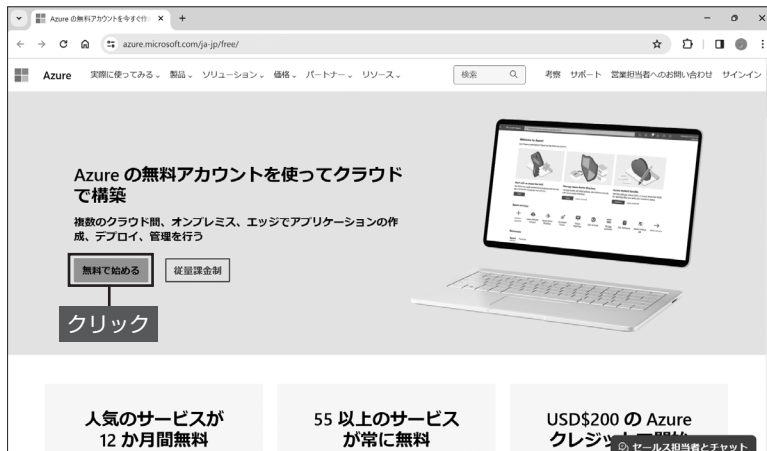
A.2 Azure デプロイの準備

まずは、 Azure にアプリをデプロイするために、 Azure アカウントの登録と、 VSCode 拡張のインストールを済ませておきます。

[1] Azure に登録する

Azure の利用には、 Microsoft アカウントが必要です。「新しい Microsoft アカウントを作成する方法」 (<https://support.microsoft.com/ja-jp/account-billing/a84675c3-3e9e-17cf-2911-3d56b15c0aaf>) などを参考に、あらかじめ取得しておきます。

あとは、「 Microsoft Azure 」 (<https://azure.microsoft.com/ja-jp/free/>) のサイトから「無料で始める」ボタンをクリックし、 Azure アカウントを作成します（図 App.2）。執筆時点での手順については「 Azure の登録から Web アプリケーションの公開 」 (<https://news.mynavi.jp/techplus/article/zeroazure-51/>) などの記事を参考にしてください。

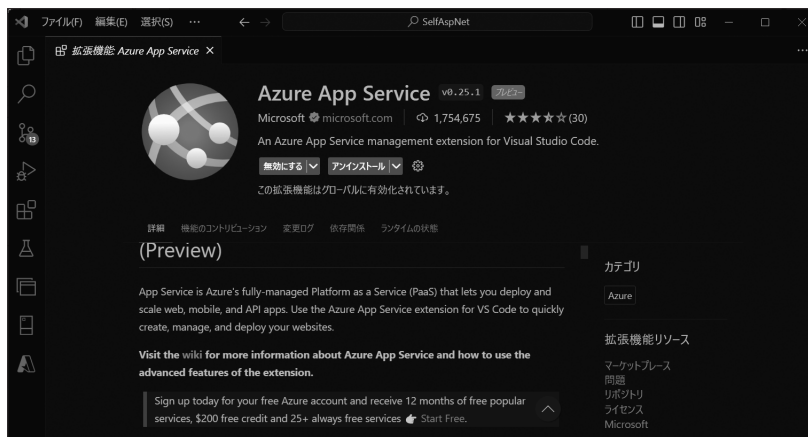


❖図App.2 Microsoft Azureのトップページ

[2] Azure App Service 拡張機能をインストールする

Azure App Service は、Azure の数あるサービスの中でも、Web アプリ、モバイルアプリのホスティングを目的とした PaaS (Platform as a Service)。Azure のリリース当初から提供されている、Azure のコアとも言えるサービスです。アプリを実行するための環境 (Platform) それ自体を提供するサービスなので、アプリ開発者が運用負荷の手間から解放される、アクセスが大きく増減した際にもマシンリソースを即座に調整できる、などのメリットがあります、

そのような Azure App Service へのデプロイ操作を簡単化するのが、Azure App Service 拡張です (図 App.3)。1.2.3 項の手順 [5] を参考に、VSCode に組み込んでおきましょう。




❖図App.3 Azure App Service 拡張のトップページ

A.3 アプリのデプロイ

以上で Azure にアプリをデプロイする準備は完了です。ここからは、実際に新たに作成した Azure 上の領域に、SelfAspNet プロジェクトをデプロイしてみましょう。

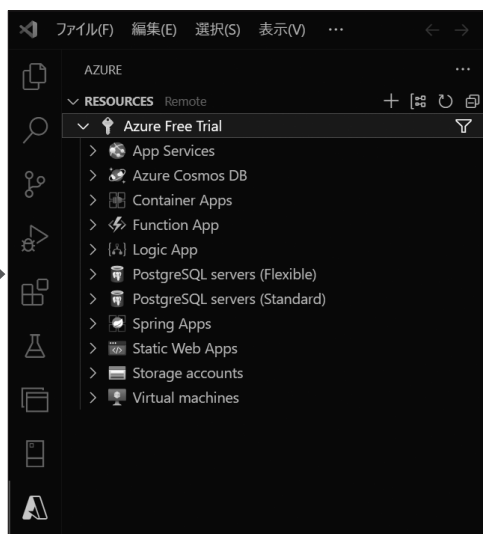
[1] Azure にサインインする

Azure App Service 拡張をインストールすると、VSCode のアクティビティバーには  (Azure) が追加されます。アイコンをクリックし、[RESOURCES] ペインから [Sign in to Azure ...] をクリックしてください (図 App.4)。

[拡張機能 'Azure Resources' が Microsoft を使用してサインインしようとしています] ダイアログが表示されるので、[許可] ボタンをクリックします。ブラウザが起動し、サインイン画面が表示されるので、先ほど取得した Azure アカウントを使ってサインインします。



❖図 App.4 Azure にサインイン



❖図 App.5 利用できるサブスクリプションを表示

サインインに成功すると、Azure で利用できるサブスクリプション※2の一覧が表示されることを確認してください (図 App.5)。

※2 Azure での契約の単位であるとともに、利用できるサービス (リソース) を管理する単位のことです。

[3] Azure App Serviceのインスタンスを作成する

[RESOURCES] ペインの [App Services] を右クリック、表示されたコンテキストメニューから [Create New Web App...] を選択します。VSCodeの上部にウィザード（質問）が表示されるので、表App.2の要領で入力しておきます。

❖表App.2 新規インスタンス作成時の設問

質問	概要	入力／選択値（例）
Enter a globally unique name for the new web app.	アプリの名前	SelfAspNet
Select a runtime stack.	実行環境	.NET8 (LTS)
Select a pricing tier	課金体系	Free (F1)

[4] ローカル環境での動作を確認する

アプリをAzureにデプロイする前に、ローカル環境でも動作を確認しておきましょう。Azureではhttpsプロファイルでの実行が前提となるので、開発証明書（自己署名証明書）も用意しておきます。もちろん、すでに作成済みの場合は重ねて作成する必要はありません。

> cd ./SelfAspNet	SelfAspNetプロジェクトへの移動
> dotnet dev-certs https --trust	開発証明書の作成
> dotnet run --launch-profile https	httpsプロファイルでの起動

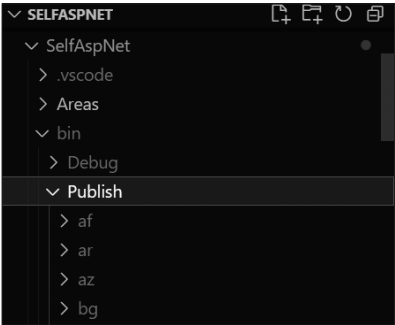
dotnet dev-certs コマンドでは、証明書を作成する旨をダイアログで確認されますが、[はい] ボタンで進めます。

[5] Azureにデプロイする

Azureにデプロイするためのパッケージは、以下のコマンドで作成できます。

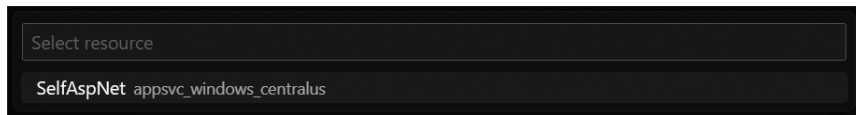
> dotnet publish -c Release -o ./bin/Publish
--

これでbin/Publish フォルダにRelease版のパッケージが作成されます（図App.6）。



❖図App.6 作成されたRelease版のパッケージ

作成されたbin/Publishフォルダーを右クリックし、表示されたコンテキストメニューから[Deploy to Web App...]を選択します。

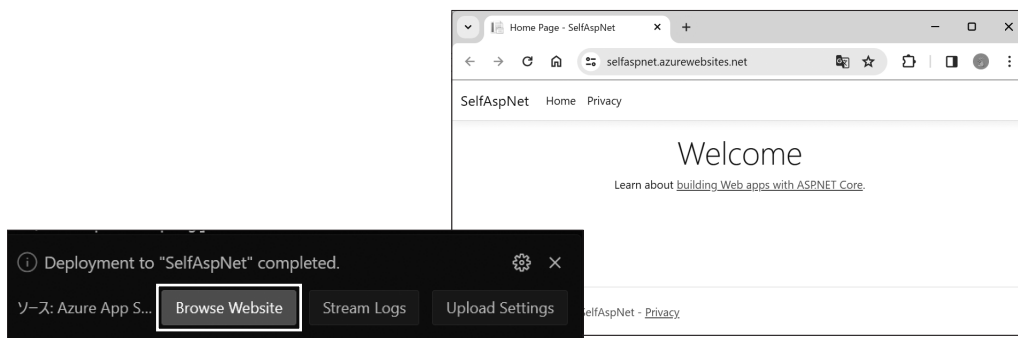


❖図App.7 デプロイ先リソースの選択

[Select resource] で、デプロイ先の Azure App Service リソースを訊ねられるので「SelfAspNet」を選択します(図App.7)。ダイアログでデプロイを確認されるので、[Deploy] ボタンでデプロイを開始します。

[6] デプロイの結果を確認する

デプロイに成功すると、図App.8左のようなトーストが表示されるので、[Browse Website] ボタンをクリックして、アプリにアクセスしてみましょう。図App.8右のような結果が表示されれば、アプリは正しくデプロイできています。



❖図App.8 デプロイの結果確認

A.4 Azure SQL Databaseの利用

Azure SQL Databaseは、言うなればAzure版のSQL Serverです。これまでのローカルのLocalDbで動作していたデータベースを、Azure SQL Databaseに移行してみましょう。

[1] AzureポータルからAzure SQL Databaseを用意する

Azure SQL Databaseを操作するには、以下のアドレスからAzureポータルにアクセスします。

Azureポータル

<https://portal.azure.com/>

「SQLデータベース」－「作成」を選択し、「SQLデータベースの作成」画面（図App.9）にアクセスします。

最初に「Want to try Azure SQL Database for free? (Azure SQL Databaseを無料で試してみませんか?)」というバナー内の「Apply Offer (Preview)」ボタンをクリックして、無料で使えるデータベース枠を適用させておきましょう。

ほとんどは既定のままでかまいません。変更が必要な箇所については、表App.3を参考に入力／選択してください。

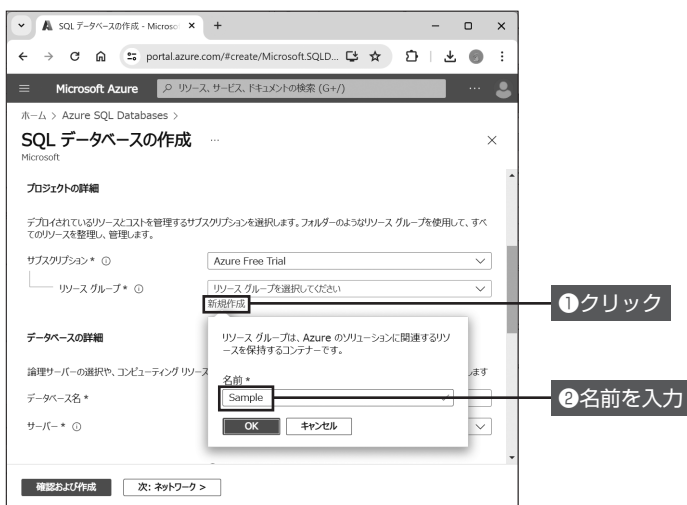
リソースグループは選択ボックス下の「新規作成」リンクから作成できます（図App.10①）。ポップアップが開くので、「名前」欄から任意の名前を入力します（②）。



❖図App.9 「SQLデータベースの作成」画面

❖表App.3 「SQLデータベースの作成」画面の設定値（例）

入力項目	概要	設定値（例）
サブスクリプション	使用するサブスクリプション	Azure Free Trial
リソースグループ	Azureのリソース（アプリ、データベースなど）をまとめるための単位	Sample
データベース名	データベースの名前	SelfAspNet
サーバー	使用するサーバー	wings-server (Japan East)



❖図App.10 リソースグループの新規作成

サーバーについても同様です。選択ボックス下の「新規作成」リンクをクリックすると、図App.11の画面に移動するので、必要な情報を入力します。[OK] ボタンで元の画面に戻ります。

サーバー名を入力

場所を選択

ここを選択

ログイン名を入力

パスワードを入力

クリック

❖図App.11 サーバーの新規作成

以上の内容を入力できたら、「確認および作成」ボタンをクリックします。入力内容の確認画面が表示されるので、「作成」ボタンをクリックすると、データベースが作成されます（図App.12）。

❖図App.12 データベースの作成が完了した

[2] ファイアウォール規則を追加する

ただし、Azure SQL Databaseは既定で外部のコンピューターからのアクセスを遮断しています。このままでは、マイグレーションを実行ができなかったり、そもそもアプリからアクセスができなかったりするので、外部からのアクセスを許可しておきましょう。

具体的には、Azure SQL Databaseのファイアウォールに対して、リクエスト元のIPアドレスを登録し、明示的にアクセスを許可します。ファイアウォールを設定するには、先ほどの完了画面から[リソースに移動] ボタンをクリックします※3。

作成したSelfAspNetデータベースの詳細情報が表示されるので (図 App.13)、[サーバーファイアウォールの設定] タブを選択し、[ネットワーク] ページに移動します。



❖図App.13 SelfAspNetデータベースの詳細情報を表示

[パブリックネットワークアクセス] 欄の[選択したネットワーク] を選択すると、ページ下部に[ファイアウォール規則] 欄が表示されるので、図 App.14のように設定を追加しておきましょう。



※3 あるいは、Azure ポータルのトップページの [リソース] 欄から [SelfAspNet (~)] (種類はSQL データベース) を選択してもかまいません。



❖図App.14 ファイアウォールの設定

①は手元のコンピューターからのアクセスを、②はAzure App Serviceからのアクセスを、それぞれ許可するための設定です。現在のローカル環境に基づいて、ClientIPAddress_YYYY-MM-DD_HH-MM-SSのような形式のIPアドレス規則が追加されます。


以上を確認できたら、[保存] ボタンでファイアウォール規則を保存します。

[3] データベース接続文字列を取得する

SelfAspNet データベースの概要ページ (図App.13) に戻り、[データベース接続文字列の表示] リンクをクリックすると、接続文字列が表示されます (図App.15)。



❖図App.15 SelfAspNetデータベースの接続文字列を表示

[ADO.NET (SQL 認証)] から  (クリックボードにコピー) をクリックし、接続文字列をコピー、appsetting.jsonに反映させておきましょう (リストApp.1)。この設定はマイグレーションを実行するために利用します。

```
"ConnectionStrings": {
  "MyContext": "Server=tcp:wings-server.database.windows.net,1433;Initial
Catalog=SelfAspNet;Persist Security Info=False;User ID=wings;Password={your_password};MultipleActiveResultSets=False;Encrypt=True;TrustServerCertificate=False;Connection Timeout=30;"
}
```

サーバー名、ユーザー名／パスワードは自分の環境のもので置き換えてください。そもそもパスワードの部分は {your_password} のように記載されているので、コピーしたものをそのままでは使えません。

[4] データベース接続文字列を環境変数に設定する

Azure App Service 上でデータベースにアクセスするために、こちらにも接続文字列を設定しておきます※4。これには、Azure ポータルのトップページの [リソース] 欄から [SelfAspNet]（種類は App Service）を選択し、左ページのメニューから [環境変数] を選択してください。[環境変数] ページが表示されるので、その [接続文字列] タブに移動します。

ページ上部にある [+ Add connection string] ボタンをクリックすると、[Add/Edit connection string] ページが開くので、図 App.16 の要領で設定情報を入力しておきましょう。



Add/Edit connection string

名前 *
MyContext

値
Server=tcp:wings-server.database.windows.net,1433;Initial Catalog=SelfAspNet;Persist Security Info=False;Us...

種類 *
SQL Azure

デプロイメントの設定
☐

適用

破棄

❖図 App.16 SelfAspNet アプリの環境変数ページ

※4 appsettings.json を書き換えたローカルアプリを再度デプロイしてもかまいません。ただし、本番環境の接続情報は Azure 上に登録しておいたほうが、後々の管理には便利でしょう。

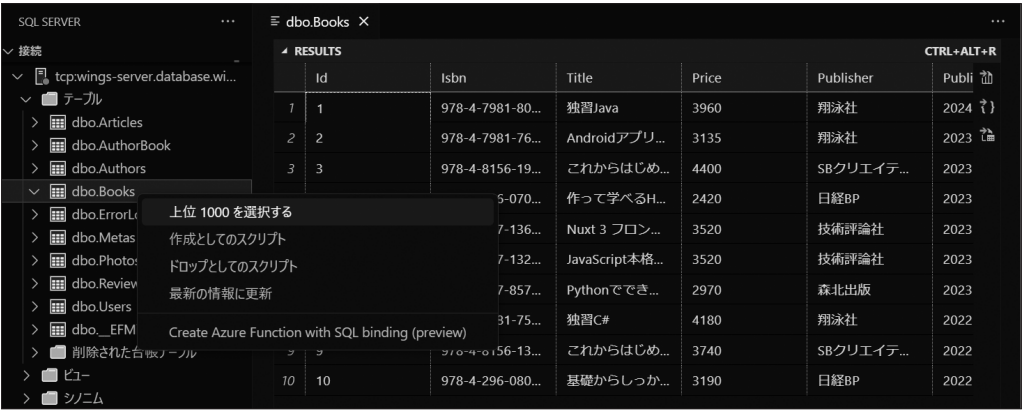
設定できたら、[適用] ボタンをクリックします。元の [環境変数] ページに戻るので、接続文字列が追加されていること確認したうえで、さらに [適用] ボタンをクリックします。[Save changes] ダイアログが表示されたら、[確認] ボタンをクリックして終了です。

[5] マイグレーションを実行する

この状態で、ローカル環境からマイグレーションを実行します。

```
> dotnet ef database update
```

マイグレーションが正しく実行できたら、2.4.4項の手順で [SQL SERVER] ペインに Azure SQL Database を登録し、テーブル／データが展開されていることを確認してください（図 App.17）。

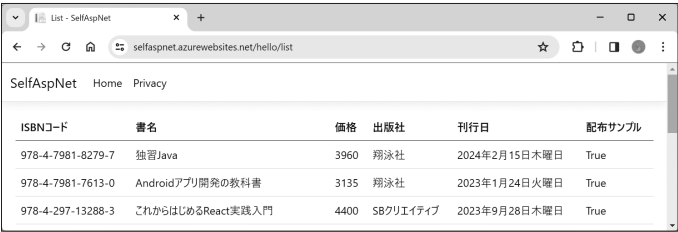


Id	Isbn	Title	Price	Publisher	Publi
1	978-4-7981-80...	独習Java	3960	翔泳社	2024
2	978-4-7981-76...	Androidアプリ...	3135	翔泳社	2023
3	978-4-8156-19...	これからはじめ...	4400	SBクリエイテ...	2023
	5-070...	作って学べるH...	2420	日経BP	2023
	7-136...	Nuxt 3 フロン...	3520	技術評論社	2023
	7-132...	JavaScript本格...	3520	技術評論社	2023
	7-857...	Pythonででき...	2970	森北出版	2023
	31-75...	独習C#	4180	翔泳社	2022
	フォロワー156-13...	これからはじめ...	3740	SBクリエイテ...	2022
10	978-4-296-080...	基礎からしっか...	3190	日経BP	2022

❖図App.17 Azure SQL Databaseに展開されたデータベース

[6] アプリの動作を確認する

最後に、Azureのページからも確認します。「<https://selfaspnet.azurewebsites.net/hello/list>」にアクセスし、図 App.18のように書籍一覧ページが表示されることを確認してください。



ISBNコード	書名	価格	出版社	刊行日	配布サンプル
978-4-7981-8279-7	独習Java	3960	翔泳社	2024年2月15日木曜日	True
978-4-7981-7613-0	Androidアプリ開発の教科書	3135	翔泳社	2023年1月24日火曜日	True
978-4-297-13288-3	これからはじめReact実践入門	4400	SBクリエイティブ	2023年9月28日木曜日	True

❖図App.18 Azure上で書籍一覧ページを表示

App
A

本
番
環
境
へ
の
移
行

Column ➔ **ASP.NET CoreアプリからSQLiteを利用するには？**

本書では、データベースサーバーとしてSQL Serverを利用してきましたが、macOS/Linuxなどの環境ではSQL Serverを別に準備するのは（特に学習環境では）厄介です。そこで、非Windows環境では学習用のデータベースとして、SQLiteを利用することをお勧めします。基本的な手順はそこまで変化しないので、2.4節からの違いを中心に補足しておきます。

[1] SQLiteドライバーをインストールする

SQL Serverドライバーの代わりに、SQLiteドライバーをインストールします。

```
> dotnet add package Microsoft.EntityFrameworkCore.Sqlite
```

[2] データベース接続文字列をSQLite向けに変更する

データベース接続文字列をSQLite向けに書き換えます。SQLiteの場合はごく簡単で、Data Sourceパラメーターに対して、データベース名（ファイル名）を引き渡すだけです（リストApp.2）。

▶ リストApp.2 appsettings.json

```
"ConnectionStrings": {  
  "MyContext": "Data Source=SelfAspNet.db"  
}
```

[3] コンテキストを登録する


コンテキストを登録する際にもUseSQLiteメソッドを利用します（リストApp.3）。

▶ リストApp.3 Program.cs

```
builder.Services.AddDbContext<MyContext>(options =>  
    options.UseSqlite(  
        builder.Configuration.GetConnectionString("MyContext")  
    )  
);
```

以上でSQLiteに接続するための準備は完了です。本文の手順に沿って、マイグレーションを実行し、プロジェクトルートの直下にSelfAspNet.dbが生成されることを確認してみましょう。

[4] SQLiteデータベースの内容を確認する

ちなみに、VSCodeからSQLiteに接続するには、SQLTools拡張を利用することをお勧めします。1.2.3項の要領でSQLTools本体とSQLTools SQLiteドライバーをインストールすると、左脇のツールバーに  (SQLTools) ボタンが表示されます。これをクリックすると、SQLToolsを開けるので、その [CONNECTIONS] ペインから [Add New Connection] ボタンをクリックします (図App.18)。



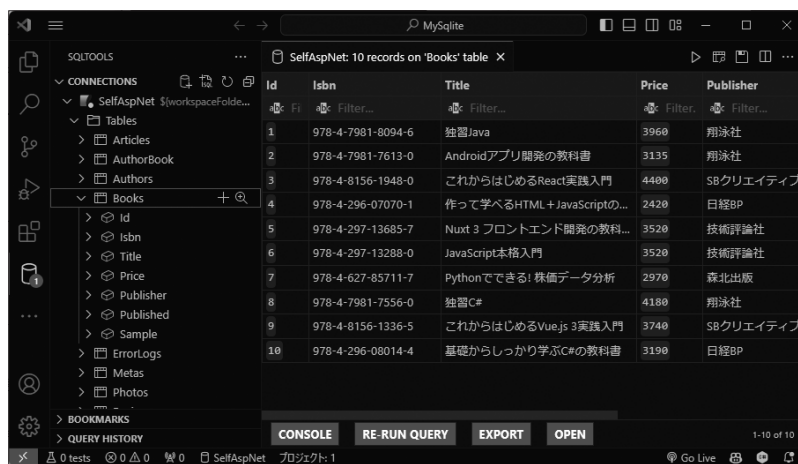
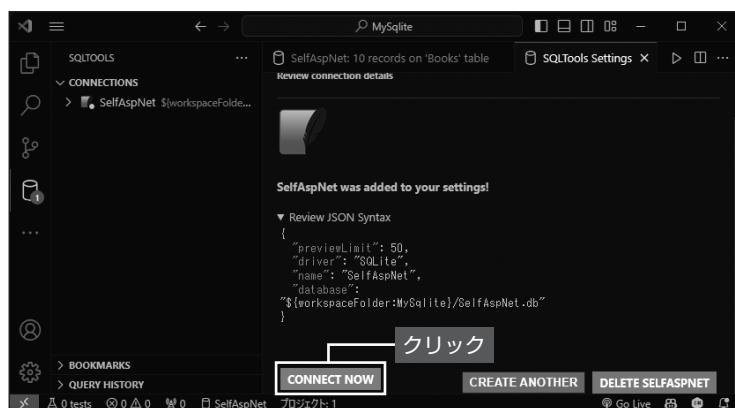
❖図App.18 SQLTools拡張からの接続設定

メイン領域に [Connection Assistant] 画面が開くので、[SQLite (Node)] ボタンをクリックします。接続情報を入力する欄が開くので、表App.4のように値を入力してください（明記のない項目は既定のままでかまいません）。

❖表App.4 SQLiteToolsの接続設定

項目	概要	設定値 (例)
Connection name	接続名	SelfAspNet
Database file	.db ファイルのパス	<プロジェクトのパス>\SelfAspNet.db

入力できたら [SAVE CONNECTION] ボタンをクリックします。確認画面が表示されるので、内容を確認できたら、[CONNECT NOW] ボタンで実際に接続してみましょう（図App.19）。



❖図App.19 データベースへの接続

[CONNECTIONS] ペインに SelfAspNet データベースが追加されているので、(たとえば) [Tables] - [Books] を右クリック、表示されたコンテキストメニューから [Show Table Records] を選択すると、メイン領域にテーブルの内容が表示されます。

アプリから動作を確認するならば、Hello#List、Books#Index などのアクションを利用してください※5。

Column → 単体テストの自動化

1.1.2 項でも触れたように、アプリを Model - View - Controller に分割することのメリットとして、単体テストを自動化しやすいという点が挙げられます。ASP.NET Core (.NET) の世界でもご多分に漏れず、テスト自動化のためのフレームワークとして MSTest が標準で用意されています。

以下では、SelfAspNet プロジェクトをテストするための MSTest プロジェクトを作成し、実行するための手順を示しておきます。

[1] テストプロジェクトを作成する

ソリューションフォルダーで、以下のコマンドを実行します。

```
> dotnet new mstest -o SelfAspNet.Tests — MSTestプロジェクトを作成
> dotnet sln add SelfAspNet.Tests — ソリューションにプロジェクトを追加
```

MSTest プロジェクトを作成するには mstest テンプレートを利用します。プロジェクト名は<テスト対象>.Tests のように命名するのが一般的です。

また、9.1.3 項の手順に沿って、SelfAspNet.Tests から SelfAspNet プロジェクトへの参照を追加しておきましょう。

[2] テストコードを作成する

ここでは Hello#Show アクションをテストするものとします。これには、Controllers フォルダーを作成し、その配下に HelloControllerTest.cs を作成します (リスト App.4)。テストクラスの名前は<テスト対象> Test.cs のように命名するのが一般的です。

※5 ちなみに、自分で一から Scaffold する際には、dotnet aspnet-codegenerator コマンド (3.1 節) に -sqlite オプションを付与する必要があります。

▶ リスト App.4 Controllers/HelloController.cs (P SelfAspNet.Tests)

```
using Microsoft.AspNetCore.Mvc;
using SelfAspNet.Controllers;

namespace SelfAspNet.Tests.Controllers;

[TestClass] _____ ❶
public class HelloControllerTest
{
    [TestMethod] _____ ❷
    public void TestShow()
    {
        var controller = new HelloController(null!); _____ ❸
        var result = controller.Show(); _____
        Assert.IsInstanceOfType(result, typeof(ViewResult)); _____ ❹
        Assert.AreEqual("こんにちは、世界！", controller.ViewBag.Message); _____ ❺
    }
}
```

テストクラスであることを示すのは `TestClass` 属性 (Microsoft.VisualStudio.TestTools.UnitTesting 名前空間❶)、テストメソッドであることを示すのは `TestMethod` 属性 (❷) の役割です。

テストメソッドの中身はカンタン。HelloController クラスをインスタンス化し、Show メソッドを呼び出すだけです (❸)。アクションの戻り値は `ActionResult` 型であるはずなので、その実体が `ViewResult` 型であること (❹)、ViewBag に格納された Message の値が「こんにちは、世界！」であること (❺) を、それぞれ確認しています。



テストメソッドでは、テスト対象のメソッドを実行し、これを `XxxxxAssert` (検証) クラスのメソッドで確認する、という流れが一般的です。

この例であれば、`IsInstanceOfType` (型検証)、`AreEqual` (値検証) をしていますが、その他にも `IsNull` (Null 判定)、`ThrowsException` (例外検証) など、さまざまな検証メソッドが用意されています。詳しくは、以下のページから `XxxxxAssert` クラスを参照してみると良いでしょう。

Microsoft.VisualStudio.TestTools.UnitTesting 名前空間

<https://learn.microsoft.com/ja-jp/dotnet/api/microsoft.visualstudio.testtools.unittesting>

[3] テストを実行する

テストプロジェクトを実行するには、VSCodeの.NET Core Test Explorer拡張を利用するのが便利です。1.2.3項の手順を参考にインストールすると、左脇のツールバーに  (テスト) ボタンが表示されます。これをクリックすると、テストエクスプローラーを開けるので、SelfAspNet.Tests右の  (テストの実行) ボタンをクリックします。

図App.20は、テストが成功／失敗したときの表示です。



❖図App.20 上：テストが成功したとき、下：テストが失敗したとき

同じくダウンロードサンプルには、リポジトリクラス (7.1.2項) を利用したデータベーステストの例も収録しています。興味のある人は、BooksControllerTest.csを参照してください。

Column 本家サイトを活用する

本書は、あくまでASP.NET Coreの基本を独学するための入門書です。学習を進めるうえでは、本書を読み進めるだけでなく、一次資料とも言うべき本家ドキュメントをあわせて参照することをお勧めします。

- **ASP.NET ドキュメント**

<https://learn.microsoft.com/ja-jp/aspnet/core/>

- **.NET API browser**

<https://learn.microsoft.com/ja-jp/dotnet/api/>

慣れないうちは、本家ドキュメントの記載は難しく、そもそも目的の箇所にたどり着くのすら苦労するかもしれません。しかし、手間に値するだけの価値が本家ドキュメントにはあります。書籍で知識の骨組みを組み立てつつ、本家ドキュメントでより具体的な（ということは個々のテーマに特化した）知識を得て、肉付けすることで、より効果的に、バランスの取れた知識を得られるはずです。

習熟の度合いが高まってきたら、ASP.NET Coreのソースコードも覗いてみるのもよい勉強になるでしょう。

- **ASP.NET Core ソースコード**

<https://github.com/dotnet/aspnetcore>

ソースコードはそれそのものがコードの見本ですし（実際、本書でもASP.NET Coreのソースコードを何か所かで紹介しています）、ドキュメントだけでは曖昧に見えた仕様を確認するのに役立ちます。

付録

B

「練習問題」
「この章の理解度チェック」
解答

第1章の解答

この章の理解度チェック p.23

- [1] (×) Linux, macOSなどのプラットフォームで動作するマルチプラットフォームフレームワークです。
(○) 正しい記述です。
(×) 親和性を考えれば、いずれかを導入しておくのが望ましいですが、開発環境の導入は必須ではありません。
(×) SQL Serverはもちろん、MySQL、PostgreSQL、SQLite、Oracleなど主要なデータベースに対応しています。
(×) Web APIにアクセス、ではなく、Web APIを作成するためのフレームワークです。
- [2] ① Controller (コントローラー) ② Model (モデル)
③ View (ビュー) ④ ビジネスロジック
⑤ レスポンス (応答)

ASP.NET MVCの基本的な構造を問う問題です。現時点では、まだイメージにないかもしれませんが、早い段階から学習の現在地（全体像）を意識しながら学習を進めることは重要です。

- [3] ① ブラウザー ② HTTPサーバー(Webサーバーでも可)
③ .NET SDK ④ コードエディター
⑤ データベースサーバー

ASP.NET Coreプログラミングに必要なソフトウェアに関する問いです。単に空欄を埋めるだけでなく、互いの関係を今一度、図の中で確認してください。

第2章の解答

練習問題2.1 p.42

- [1] 以下のようなコマンドを実行できていれば正解です。作成されたコードを見ながら、ソリューションとプロジェクトの関係、プロジェクトの構造などを再確認しておきましょう。

```
> cd C:\data ————— ソリューションの作成先に移動
> dotnet new sln --output SelfPractice
                           ソリューションを作成
> cd ./SelfPractice ——— ソリューションフォルダーに移動
> dotnet new mvc -o MyMvc -f net8.0 --no-https
                           プロジェクトを作成
> dotnet sln add MyMvc
                           MyMvcプロジェクトをソリューションに追加
```

- [2] 以下の点を指摘できていれば正解です。
- アクションメソッドはpublicであること
 - 戻り値はActionResult型であること
 - Contentヘルパーで作成したActionResultオブジェクトはreturnで返す

以上を修正したコードは、以下のとおりです。

```
public IActionResult Prac2_1()
{
    return Content("Hello, World!!");
}
```

練習問題2.2 p.50

- [1] Razorはビューエンジンの一種であり、HTMLにRazor固有のコードを埋め込むことにより、動的にページを組み立てることができます。「C#の構文で、条件分岐、ループなどを表現できる」「タグヘルパーを利用することでフォーム／リンクなどをより簡単に組み立てられる」などの特長があります。
- [2] ViewBag, ViewData
前者がビュー変数をプロパティとして表すのに対して、後者はディクショナリのキーとして表します。その性質上、前者の名前には識別子として利用できない文字を含めることはできません（たとえばViewBagではmy-infoのような名前は使えません）。

この章の理解度チェック p.69

- [1] (×) ビジネスロジックはモデルの役割です。コントローラーの役割は、リクエストの受け取りからモデルの呼び出し、最終的な結果のビューへの引き渡しです。
(×) モデルクラスはコントローラークラス（またはアクション）の誤りです。
(×) 検索先のパスは「Views/<コントローラー名>/<アクション名>.cshtml」です。
(×) 一般的なテンプレートと同じく、@...、@{|...}などの式は利用できます。
(×) モデルの型を定義するのは、@modelディレクティブの役割です。
- [2] ① DbSet ② エンティティ
③ プロパティ ④ Id、または<エンティティ名>Id
⑤ Null許容

EF Coreでは、あらかじめ名前付けのルールが決まっており、それに従うことで設定レスでデータベースとエンティティとを紐づけできます。最低限、まずは本間のルールを押さえておきましょう。

- [3] ① RenderBody メソッド ② dotnet add package コマンド
③ appsettings.json
④ MapControllerRoute メソッド、記述先はProgram.cs
⑤ Content
⑥ AddDbContext ⑦ launchSettings.json

メソッド、コマンドなどの名前を丸暗記することは、本間の主目的ではありません（たとえばdotnetコマンドはこのあたにも数多く登場します）。名前を思い出す中で、使い方、周辺の解説を再確認するきっかけとしてください。

- [4] ① Controller ② _db = db ③ IActionResult
④ Books ⑤ @model ⑥ ViewData
⑦ foreach ⑧ Model ⑨ @item

コントローラー、ビュー、モデルの基本構文を問う複合的な問題です。コンストラクター注入（②）、モデルの型付け（⑤）、ループ構文（⑦）など、このあたにも頻出のテーマが盛り込まれているので、間違ってしまったという人は該当の項を再確認しておきましょう。

第3章の解答

練習問題3.1 p.87

[1] 以下の点を指摘できていれば正解です。

- コンテキストは直接インスタンス化するのではなく、コンストラクターの引数から注入する
- 非同期メソッドを呼び出すには、await 演算子を付与する
- 非同期アクションに付与するのは (awaitではなく) async 修飾子
- 非同期アクションの戻り値は Task<ActionResult> 型である

以上を修正した結果については、P.75のリスト3.1を参照してください。

練習問題3.2 p.102

[1] モデルバインドとは、リクエストデータと同名の**引数**を用意することで、対応する値を自動的に割り当てるための仕組みです。**ポストデータ**をはじめ、クエリ情報、ルートパラメーターなど入力元を意識しなくてもよいという特長があります。また、int、stringのような基本型だけでなく、**モデルオブジェクト**のような複合型の個々のプロパティに対して値を割り当てることもできます。

[2] ① RedirectToAction ② Content ③ NotFound

問題にあるものに加えて、View メソッドまでが特によく利用するヘルパーメソッドです。4.2節ではさらにさまざまなヘルパーを紹介するので、そこまで基本ヘルパーを確実に身につけておきましょう。

この章の理解度チェック p.105

[1] (×) 誤った記述です。dotnet コマンドはプロジェクトフォルダーの下直で実行します。

(○) 正しい記述です。

(×) はありません。データ型に応じた値を出力するには、ビューヘルパー DisplayFor を利用します。

(×) 誤った記述です。複数のリクエストを効率的に処理できるので、極力、非同期化すべきです。

(×) 誤った記述です。エンティティの状態を記録するだけで、実際の作成／更新／削除を実行するのは SaveChangesAsync メソッドの役割です。

[2] ① @Html.DisplayNameFor(model => model.Isbn)
② @Html.DisplayFor(model => model.Price)
③ <a asp-action="Edit" asp-route-id="10">Edit
④ <label asp-form="Published" class="control-label"></label>
 <input asp-form="Published" class="form-control" />
⑤ @{|
 ViewData["Title"] = "詳細画面";
 |

モデルを前提とした入出力に関わる基本的なコードです。まずは、これらのコードを自然に書けるようになることが目標です。なお、④の class 属性は Bootstrap で用意されたスタイルクラスです。Scaffold されたコードに従って解答には含めていますが、なくても間違いではありません。

[3] ① [DisplayName = "ISBN"]
 public string Isbn { get; set; } = String.Empty;
② return View(await _context.Books.ToListAsync());
③ var book = await _context.Books
 .FirstOrDefaultAsync(m => m.Id == id);
④ return RedirectToAction(nameof(Index));
⑤ public async Task<ActionResult> Create(
 [Bind("Id,Isbn,Title,Price,Publisher,Published,Sample")] [a]
 Book book)
⑥ _context.Update(book);
 await _context.SaveChangesAsync();
⑦ [ActionName("Delete")]

EF Core によるデータベース操作 (②、③、⑥)、モデルバインド (⑤)、ビュー呼び出し／リダイレクトなどのアクション操作 (②、④) などを問う問題です。断片的なコードを答えるだけでなく、ぜひ、周辺のコードを見渡すきっかけにしてください。Scaffold されたコードは、最終的に自分でも何も見なくとも書けるのが目標です。

[4] ① [HttpPost] ② async ③ Bind
④ NotFound ⑤ Update ⑥ SaveChangesAsync
⑦ nameof ⑧ View(book)

モデルバインドから入力値のチェック、データ更新までの流れを、具体的なコードで再確認する問題です。[2] [3] では単体のコードを確認しましたが、一連のコードの中で互いの関連を再確認しましょう。

第4章の解答

練習問題4.1 p.152

[1] Html.DisplayFor メソッド。テンプレートを選択するためのキー情報は、優先順位の低いものから「プロパティのデータ型」「DataType 属性」「UIHint 属性」「DisplayFor メソッドの引数」です。

[2] アプリのHTMLエスケープ漏れを利用した脆弱性です。<script> などのタグを混入させることで、アプリ上で任意のコードを実行できてしまう危険があります。Razor では、これを避けるために @... 式の出力はすべてエスケープ処理されます。意図してエスケープを回避するには、HtmlRaw ヘルパーを利用します。

練習問題4.2 p.192

[1] ① HtmlTargetElement ② TagHelper
③ [HtmlAttributeName("isbn")] ④ override
⑤ Process ⑥ output
⑦ Attributes.SetAttribute

タグヘルパーを定義するには、TagHelper#Process メソッドをオーバーライドするのが基本です。本間には含まれませんが、@addTagHelper ディレクティブに代表されるタグヘルパーの登録についても復習しておきましょう。

[2] ① _ViewStart.cshtml ② セクション
③ Scoped CSS ④ 部分ビュー
⑤ クロスサイトスクリプティング脆弱性

これまでも何度か述べてきましたが、キーワードを覚えることは本質ではありません。キーワードを思い出す中で、関連する概念を復習する手がかかりとしてみてください。

この章の理解度チェック p.207

- [1] ① EditorFor ② Raw
 ③ asp-controller ④ asp-action
 ⑤ asp-route-id ⑥ asp-items
 ⑦ Cache Busting（キャッシュバusting）
 ⑧ asp-fallback-test
 ⑨ asp-fallback-src ⑩ <environment>

ASP.NET Coreでは、さまざまなタグヘルパー／ビューヘルパーが用意されています。属性名などを覚えることは本質ではありませんが、どのようなものが用意されているのかをある程度頭に入れておくことは大切です。

- [2] ① <a href="/book/@@(isbn).png"...
 ② @if (Model.Price < 500) {
 @: [Sale]
 }
 ③ @(@{lock})
 ④ 詳細は「@@IT」の記事を参照
 ⑤ @* この部分はコメントアウト*@

②については、別解としてWriteLiteral、<text>要素を利用してもかまいませんが、1行テキストであれば、まずは「@」で十分です。

③は、C# Razor キーワードです。あまり利用する機会はないかもしれませんが、C#、Razor 双方をエスケープするために `@(@(...))` のように表します。

④は「@」文字を解釈させたくない場合です。「@@」で「@」文字をエスケープできます。

- [3]** ① static ② IHttpContent
 ③ this ④ TagBuilder
 ⑤ MergeAttribute ⑥ AnonymousObjectToHtmlAttributes
 ⑦ RenderSelfClosingTag

タグ文字列を含んだ結果を返すならば、戻り値はIHtmlContent型とするのです。

タグをプログラマ的に組み立てる TagBuilder クラスと、その基本的な用法はぜひ押さえておきましょう。ただし、あまり複雑なものはビューコンポーネントなどを利用したほうが簡単なので、まずは属性込みのタグを生成する方法までをきちんと押さえておきましょう。

- [4]** ① Components/List/Default.cshtml ② ViewComponent
 ③ MyContext ④ Invoke.Async
 ⑤ await ⑥ List
 ⑦ price = 3000

誤っている点は、以下のとおりです。

- メソッドの戻り値はTask<ActionResult>型ではなく、Task<IViewComponentResult>型
- IViewComponentResult型を生成するヘルパーは、PartialViewではなくView
- @modelディレクティブの型は、SelfAspNet.Models.Bookではなく、IEnumerable<SelfAspNet.Models.Book>

データベースアクセスをはじめとしたロジックを伴う、しかも、階層的なタグの組み立てであれば、タグヘルパーや部分ビューよりもビューコンポーネントを利用するのが便利です。ぜひ、基本的な構文をマスターしておきましょう。

第5章の解答

練習問題 5.1 p.259

- 【1】 以下のようなコマンドを書けていれば、正解です。

```
> dotnet ef migrations add AddTelToUser
> dotnet ef database update
```

migrations add→database updateの流れは、マイグレーションの中でも今後最もよく利用するものです。エンティティの修正がどのようにデータベースに反映されるのかを確認しつつ、自然に手が動くまで繰り返しコマンドを実行してください（データベースが汚くなったら、database dropコマンドで一旦データベースを削除すればよいだけです）。

- [2] いずれも EF Core での開発アプローチです。コードファーストが最初にエンティティクラス（コード）を定義してからマイグレーションを利用してデータベースを作成するのに対して、データベースファーストでは既存のデータベースをもとにエンティティ／コンテキストを生成することを言います（dotnet ef dbcontext scaffold コマンドを利用します）。

練習問題 5.2 p.306

- 【1】以下の点が指摘されていれば正解です。

- Filterアクションは同期アクションなので、戻り値もTask<ActionResult>型ではなく、ActionResult型。
- クエリメソッドはIQueryable型を戻り値として返すので、Whereメソッドの戻り値は元の変数bsに書き戻す（2カ所）。
- 部分一致検索なので、Whereメソッド内の検索条件はStartsWithではなくContains。
- Nullable型から値を取り出す場合には、HasValue プロパティでチェックしてからValue プロパティにアクセスする。
- 今日以前の書籍を絞り込むので、「>」演算子は「<=」演算子の誤り。
- Viewメソッドには、取得した結果（モデル）を渡す

これらの点を修正した結果については、P.269のリスト5.40を参照してください。

- [2] 以下のようなコードを書けていれば正解です。

```
await _db.Books.Where(b => b.Price >= 5000)
    .ExecuteUpdateAsync(setters =>
        setters.SetProperty(b => b.Price,
            b => (int)(b.Price * 0.8)));
```

一括更新にはExecuteUpdateAsyncメソッドを利用します。同様に、一括削除にはExecuteDeleteAsyncメソッドを利用します。

この章の理解度チェック p.331

- [1] (×) Null 許容型を無効にした場合、string はnullを許容するという意味になり、生成されるテーブル列にもNULL属性が付与されます。
- (○) 正しい記述です。
- (×) RowVersion属性はTimestamp属性の誤りです。

- (×) エンティティ操作の前後に共通的な処理を差し挟むための仕組みです。本文ではデータ作成／更新日時を記録する例を紹介しました。
- (×) 数値の最小値／最大値を制限するための仕組みです。文字列長の制限にはStringLength属性を利用します。

```
[2] ① public ICollection<Review> Reviews { get; } = new List<
    <Review>();
    ② public int UserId { get; set; }
    public User User { get; set; } = null!;
    ③ [Column(TypeName="CHAR(17)")]
    public string Isbn { get; set; } = String.Empty;
    ④ [Index(nameof(LastName), nameof(FirstName), Name=㉔)
    "Index_FullName")]
    public class User { ... }
    ⑤ builder.Entity<Review>(rev => {
        rev.ToTable("Comments")
        .HasKey(b => b.Code);
    });
```

コレクションナビゲーション (①) の「ゲッターのみ+既定値は空コレクション」、参照ナビゲーション (②) の「外部キープロパティ+既定値はnull (NULL禁止の場合)」はいずれも定型句ですが、理屈も込みで定義の意味を押さえておきましょう。

属性 (③、④) はさまざまに用意されていますが、標準規約を拡張するという意味では、Column、Indexなどが特に重要です。復習しておきましょう。

```
[3] ① var bs = _db.Books
    .Where(b => b.Title.StartsWith("独習"))
    ② var result = _db.Books.AnyAsync(b => b.Price >= 4000);
    ③ var bs = _db.Books
    .OrderBy(b => b.Published)
    .Skip(2).Take(3);
    ④ var bs = _db.Books
    .GroupBy(b => b.Publisher);
    ⑤ var b = _db.Books
    .Include(b => b.Reviews)
    .Include(b => b.Authors)
    .ThenInclude(a => a.User)
    .SingleAsync(b => b.Id == id);
```

OrderBy メソッド (②) は昇順降順で使用するメソッドも異なるので、要注意です。

また、EF Coreで関連するデータを取得するには、ナビゲーションプロパティを定義するだけでは不十分です。Includeメソッド (⑤) で明示的に取得先のテーブルを指定しなければなりません。「先の先」を取得するThenIncludeについても押さえておきましょう。

```
[4] ① Book book                ② ModelState.IsValid
    ③ Add                        ④ await
    ⑤ RedirectToAction           ⑥ View(book)
    ⑦ asp-action                 ⑧ asp-validation-summary
    ⑨ asp-for                    ⑩ asp-validation-for="Isbn"
```

第3章から何度も見てきたScaffoldされたコードですが、ここで検証処理を学んだことでコードの意味が完全に理解できたはずです。いよいよ本書も中盤、入力値の取得から検証結果の処理、エラーの表示までをひと連なりで整理しておきましょう。

第6章の解答

練習問題6.1 p.368

- [1] ① return View("About", bs);
② return RedirectPermanent("https://wings.msn.to/");
③ return Forbid();
④ return File(p.Content, p.ContentType, p.Name);
⑤ return Content("<p>Hello,World!!</p>",
System.Net.Mime.MediaTypeNames.Text.Html,
Encoding.UTF8);
- ③はStatusCodeヘルパーを使って「return StatusCode(403);」のようにしてもかまいませんが、一般的には専用ヘルパーがあるものはそちらを優先したほうがコードはシンプルとなり、意図も明快です。
- [2] いずれもブラウザーキャッシュを操作するための応答ヘッダーです。Etagヘッダーはリソースに対して付与されたタグで、一般的にはリソースのハッシュ値などがセットされます。ファイルの内容が変わった場合にEtagも変化するので、更新を判定できます。Last-Modifiedヘッダーは、ファイルの最終更新日時を表します。こちらも値が変化したら、ファイルも更新されていることを意味します。更新が検出された場合には新たなデータが送信され、さなければ「304 Not Modified」だけがブラウザーに通知されます。

練習問題6.2 p.430

- [1] それぞれ以下のようなコードが書けていれば正解です。

```
// アプリ単位 (Program.cs)
builder.Services.AddControllersWithViews(options =>
{
    options.Filters.Add<MyLogAttribute>();
});

// コントローラー単位 (FilterController.cs)
[MyLog]
public class FilterController : Controller
```

属性として指定する場合には、接尾辞のAttributeは省略するのが一般的です。アクション単位で適用する場合は、コントローラーの場合と同様に記述できます。

- [2] CSRF (クロスサイトリクエストフォージェリ) 脆弱性とは、サイトに攻撃用のスクリプトを仕込んでおくことで、アクセスしてきたユーザーに意図しない操作を強制できてしまう脆弱性のことです。CSRF脆弱性を含んでいるサイトでは、「掲示板やブログであれば勝手に書き込みされてしまう」「ショッピングサイトであれば自動的に物品を購入されてしまう」などの被害を受ける可能性があります。

この章の理解度チェック p.437

- | | |
|-------------------|-----------------------|
| [1] ① モデルバインド | ② 値プロバイダー |
| ③ ポストデータ | ④ クエリ情報 |
| ⑤ ルートパラメーター | ⑥ Bind |
| ⑦ オーバーポスティング | ⑧ Prefix |
| ⑨ <仮変数名>.<プロパティ名> | ⑩ TryUpdateModelAsync |
| ③～⑤については順不同です。 | |

モデルバインドはなかなか奥深い世界で、時として、魔法のような挙動を見せることもあります。自らモデルバインドをカスタマイズする機会はそのままで多くはないかもしれませんが、内部的な仕組みを知っておくことで、思わぬ挙動に遭遇したときに問題を特定しやすくなるはずです。

- [2] ① View ② Content
③ RedirectToAction ④ バイト配列 ⑤ StatusCode
⑥ NotFound ⑦ PartialView

IActionResult オブジェクトを生成するためのヘルパーはさまざまありますが、まずは本で紹介したものが押さえられていれば、基本的な用途には耐えられます。Redirect.Xxxxx ヘルパーは用途に応じて、さらに細分化されるので、6.1.3 項もあわせて見直しておきましょう。

- [3] ① [ResponseCache(Duration = 60, VaryByQueryKeys = []
new[] { "mode" })]
② [AcceptVerbs("Post", "Put")]
③ builder.WebHost.ConfigureKestrel(opts =>
{
 opts.Limits.MaxRequestBodySize = 1024 * 1024 * 512;
});
④ builder.Services.AddControllersWithViews(options =>
{
 options.CacheProfiles.Add("MyCache", new CacheProfile {
 Duration = 300
 });
});
⑤ [ServiceFilter(typeof(LogExceptionFilter))]

②は、一般的にはHttpPost / HttpPut 属性を列挙してもかまいませんが、ここでは「1つの属性」という制限があるので、AcceptVerbs 属性を使っています。

- [4] ① AttributeUsage ② Attribute, IAuthorizationFilter
③ out ④ throw
⑤ OnAuthorization ⑥ current < Begin || current > End
⑦ context.Result

①、③、④などではC#そのものの知識も問うています。属性クラスの基本的な用法、出力引数などの話題を再確認しておきましょう。また、⑥は構文規則というよりもTimeLimit属性そのもののロジックです。要件を満たすように条件式を組み立てましょう。

第7章の解答

練習問題 7.1 p.471

- [1] AddSingleton、AddScoped、AddTransient、AddSingleton サービスは、アプリで単一であることが保証されます。一方、AddScoped サービスはリクエスト都度に、AddTransient サービスは呼び出し都度に生成されます。
- [2] Use、Map、Run。ミドルウェアパイプラインで指定順に実行すべきミドルウェアを表すのがUseメソッド、パイプラインの終端を表すのがRunメソッド、そして、パスによってパイプラインを分岐するのがMapメソッドです。

練習問題 7.2 p.488

- [1] jsonファイル、xmlファイル、.iniファイル、環境変数、コマンドライン引数、メモリ内のコレクション

以上から3個以上挙げられれば正解です。ただし、.xmlファイル、.iniファイル、メモリ内のコレクションを利用するにはProgram.csで宣言の必要があります。

- [2] Optionsパターンとは、構成情報を自作のクラスにマッピングする手法を言います。マッピングされた値にはOptionsクラスのプロパティとしてより直観的にアクセスでき、しかも、最初から型付けされているので、コードの中でも扱いやすいというメリットがあります。扱える型には、基本型をはじめ、列挙型、DateTime、TimeSpan、および、これらの値の配列／リスト、クラス型などがあります。

この章の理解度チェック p.505

- [1] (×) AddMvcメソッドでは、関連サービスがまとめて有効になります。一般的には過剰なので、AddControllersWithViewsメソッドで十分です。
- (×) AddTransientメソッドの説明です。AddScopedメソッドで登録されたサービスは、リクエスト単位に1つインスタンスを生成します。
- (×) 呼び出し順は追加順によって決まります。
- (×) Runメソッドを終端として、パイプラインは元来た道をさかのぼります。
- (×) フィルターパイプラインとは、EndpointMiddlewareミドルウェアによって実行されるもので、ミドルウェアパイプラインの一部です。

サービスの有効範囲、ミドルウェアの仕組みなどはなんとなく言われるがままに書いても用が足りしまう内容ですが、仕組み、概念を知っておくことで、アプリの挙動がイメージしやすくなります。挙動を知ることとは、デバッグに際しても問題を特定しやすくなることにつながります。

- [2] ① builder.Services.AddSingleton<IMyService, MyService>();
② app.Map("/current", appCur =>
{
 appCur.Run(async context =>
 {
 await context.Response.WriteAsync("<p>current</p>");
 });
});
③ var p = _config.GetValue<string>("MyAppOptions:Projects:0");
④ builder.Configuration.AddXmlFile(
 "appsettings.xml", optional: true, reloadOnChange: true);
⑤ dotnet watch run OpenWeather:ApiKey="hoge"
⑥ _logger.LogWarning(\$"[Path] -> {Current: yyyy年MM月dd日}",
 Request.Path, DateTime.Now);

⑥は、C#標準の文字列補間を利用しても表現できますが、7.4.3 項でも触れた理由からLog.Xxxxxメソッドの機能でバインドすることをお勧めします。

- [3] ① ILogger ② object ③ FileLogger
④ where ⑤ ==> ⑥ lock
⑦ Path.Combine ⑧ ToString ⑨ ?
⑩ AppendAllText

ロガーを例にはしていますが、問いの内容はほとんどがC#の基本的

な文法知識に関するものです。④、⑤、⑥などは改めて聞かれると悩むものもあったかもしれません。そんな内容については、今一度、「独習C# 第5版」(翔泳社)に戻って文法理解を固めることをお勧めします。C#(言語)とASP.NET Core(フレームワーク)と相互に行き来しながら学習を進めることで、飽きることなく、双方の理解を深められるはずです。

第8章の解答

練習問題8.1 p.545

- [1] エリアは、アプリを構成するコントローラー／ビューを分離するための仕組みです。「Areas/<エリア名>」のようなフォルダーで管理されます。ただし、コントローラーではエリアを識別するためにArea属性を付与しなければなりません。また、_ViewStart.cshtml / _ViewImports.cshtmlはそのままではエリアに適用されないで、プロジェクト既定のものをプロジェクトルートに移動します(もしくはエリア配下のViewsフォルダーにコピーしてもかまいません)。
- [2] クッキーはブラウザーに保存される小さなテキストです。設定によってはアプリを閉じたあとでも継続して保持できます。一方、セッションはサーバーで管理されるデータで、キーだけをクッキーで管理します。一般的な保存期間は、アプリを終了する(=ブラウザーが閉じられる)までです。最後の TempData は一時的なメッセージを管理するための仕組みで、一度参照するまでが生存期間です。保存先は、クッキー、サーバー上のメモリなどから選択可能です。

練習問題8.2 p.571

- [1] リソースファイルの作成方法については、8.5.1項で扱っています。ResXpress 拡張を利用する前に、リソースの枠組みを用意しなければならない点に注意してください。作成したリソースは Resources フォルダー配下に Views.View.I18n.fr.resx という名前で保存します。

この章の理解度チェック p.579

- [1] (×) セキュリティ的な脆弱性の原因ともなるので、エラー情報は最小限にします(開発者向けにはログとして別に記録すべきです)。
- (×) コンテキスト変数は、現在のリクエストでだけ保持されます。
- (○) 正しい記述です。
- (×) サポート外のカルチャが要求された場合には、SetDefault Culture メソッドで指定されたカルチャが適用されます。既定のリソースはサポート内のカルチャが要求され、そのリソースが存在しない場合に適用されます。
- (×) 最終的なアクセスのタイミングから有効期限が換算されます。

4番目の設問がわかりにくかったかもしれません。間違った人、自信のなかった人は、P.566の図8.23から、カルチャ、リソースの選択がそれぞれ別ものであることを再確認しておきましょう。

- [2] ① app.MapControllerRoute(
 name: "article",
 pattern: @"article/{aid:regex(^d{1,3}|\$)}",
 defaults: new {
 controller = "Route",
 action = "Param"
 }
);

```
};  
② [Route("chap08")]  
public class RouteController : Controller  
{  
    [Route("d/{id:int=13}")]  
    public IActionResult Details(int id) { ... }  
}  
③ app.MapControllerRoute(  
    name: "area-default",  
    pattern: "{area:exists}/{controller=Home}/{action=Index}/{id?}");
```

- ①では正規表現制約を constraints パラメーターで表してもかまいません。ただし、リクエストパスの中に埋め込んだほうがシンプルなので、まずは解答の例が理想です。
- ②は/chap08がコントローラー共通のパスなので、クラスレベルでRoute属性を指定している点に注目です。
- ③でエリア付きのルートを定義する際、{area}に制約条件 exists を付けるのは、意図しないマッチが発生しないためです。

- [3] ① HttpContext.Response.Cookies.Append("email",
 "hoge@example.com",
 new CookieOptions
 {
 Expires = DateTime.Now.AddDays(7),
 HttpOnly = true
 });
② dotnet watch run --environment Production
③ app.UseExceptionHandler("/Home/Error");
④ app.UseStaticFiles(new StaticFileOptions
 {
 FileProvider = new PhysicalFileProvider(
 Path.Combine(builder.Environment.ContentRootPath,
 Path, "MyStorage")),
 RequestPath = "/storage"
 });
⑤ @inject IHtmlLocalizer<SharedResource> SharedLocalizer
 <p>Common キー : @SharedLocalizer["Common"]</p>
⑥ [Display(Name = "Book_Title")]
 [Required(ErrorMessage = "RequiredError")]
 public string Title { get; set; } = String.Empty;

④についてはそらで解答するのは難しいかもしれません。そんな人は無理せず、VSCode上でインテリセンスを利用してコードを書いてみましょう。実務の世界では、クラス名/メンバー名を丸暗記することは本質ではありません。大雑把に機能を把握しておき、あとはツールを活用し、ドキュメントを調べられる力が大切です。

- [4] ① static ② <T> ③ this ISession session
④ T ⑤ JsonSerializer.Serialize ⑥ T?
⑦ default

セッションを例にはしていますが、問いの内容は拡張メソッド、ジェネリックメソッドと、ほとんどがC#の基本的な文法知識に関するものです。いずれもASP.NET Coreを扱うようになると、よく利用するテーマなので、迷ってしまった人は改めて復習しておくことをお勧めします。

第9章の解答

練習問題 9.1 p.601

- [1] ① async ② IActionResult
 ③ NotFound() ④ Books.FindAsync
 ⑤ book != null ⑥ Remove
 ⑦ await ⑧ RedirectToPage

紙面上では掲載していないコードですが、ASP.NET MVC の削除機能 (3.6.1 項) を覚えていれば、ほとんど同じ要領で表せることがわかるはずです。差分を意識しながら (ということは共通部分を意識しながら) 学び、Razor Pages を効率的に習得していきましょう。

練習問題 9.2 p.637

- [1] 以下のようなものが答えられていれば正解です。

- a. 特定の型
b. IActionResult
c. ActionResult<T>
d. IResult

a. が最もシンプルなパターンで、具体的に返す型を表します。ステータスコードだけを返すような状況では b. を、実データ+ステータスコードのパターンでは c. を、利用します。一般的なコントローラーベースの API ではそこまで十分ですが、d. は、Minimal API との相互運用を意図する際に利用します。

- [2] CoreApi.http に、以下のようなコードが書けていれば正解です (もちろん、id 値、データの中身は異なってもかまいません)。

```
PUT {{CoreApi_HostAddress}}/api/Books/1
Content-Type: application/json

{
  "id": 1,
  "isbn": "978-4-7981-8094-6",
  "title": "独習Java 第6版",
  "price": 3278,
  "publisher": "翔泳社",
  "published": "2024-02-15T00:00:00",
  "sample": true
}
```

この章の理解度チェック p.646

- [1] (×) OnGet / OnPost の誤りです (非同期ハンドラーでは OnGetAsync / OnPostAsync)。
(×) asp-action は asp-page です。IActionResult を生成するヘルパーもそうですが、Razor Pages ではほとんどが MVC と共通したコードを利用しますが、一部に異なるものがあるので注意してください。
(×) 利用できます。利用できないのはアクションフィルターです (代わりにページフィルターを利用します)。
(○) 正しい記述です。ただし、HttpGet / HttpPost 属性を付与することで、命名規則に反した名前を付けることもできます。
(×) 逆の記述です。

- [2] ① dotnet sln add CoreRazor
 ② public async Task OnGetAsync() {
 ③ @page "id=1"
 ④ /// <response code="204"> 新しく書籍情報が更新された Ⓐ
 </response>
 /// <response code="404"> 更新時に対象の書籍が削除されて Ⓐ
 いた </response>
 [HttpPut("{id}")]
 [ProducesResponseType(StatusCodes.Status204NoContent)]
 [ProducesResponseType(StatusCodes.Status404NotFound)]
 public async Task<IActionResult> PutBook(int id, Book book)
 ⑤ [FormatFilter]
 [HttpGet("{id},{format?}")]
 public async Task<ActionResult<Book>> GetBook(int id)

③ は絶対パスで「@page "/books/edit/{id}」のように表すこともできますが、パラメーターを追加するだけであれば差分のみを追加するほうが自然です。

応答形式の変更 (⑤) には、FormatFilter で機能を有効にし、Http.Xxxx 属性で format パラメーター付きのパスを指定します。FormatFilter 属性はコントローラーレベルで指定してもかまいません。

- [3] ① PageModel ② Page()
 ③ [BindProperty] ④ async
 ⑤ IActionResult ⑥ OnPostAsync
 ⑦ ModelState.IsValid ⑧ Books.Add
 ⑨ SaveChangesAsync ⑩ RedirectToPage

① は、OnGet メソッドが IActionResult 型を返すことから、単にページを表示するための Page メソッドを呼び出します (一般的には、戻り値を void としてもかまいません)。

⑦ は、検証に失敗した場合にページを再描画するので「!」(でない) を付けるのを忘れないようにしてください。

警告の原因となるのは、Book プロパティです。Null 禁止型なので、明示的に既定値を表します。

```
public Book Book { get; set; } = default!;
```