Visual Studio 自習書シリーズ

Web アプリケーションのテスト

発行日：2013 年 6 月 26 日

最終更新日：2013 年 6 月 26 日

日本マイクロソフト株式会社

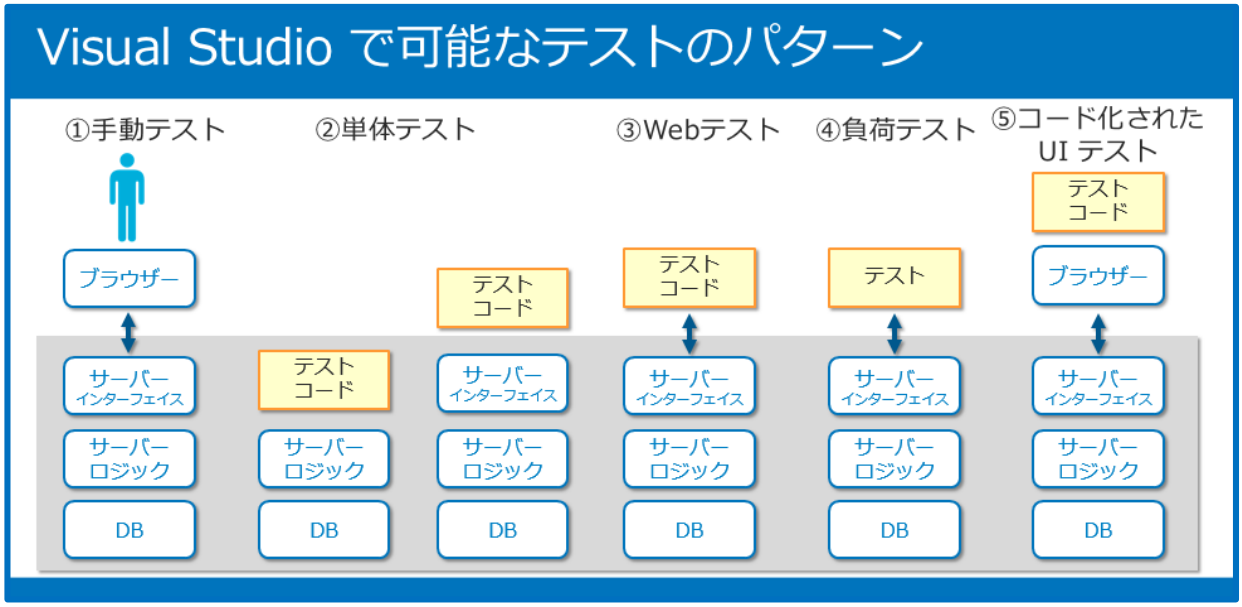
目次

Web アプリケーションのテスト

Web アプリケーションは企業における様々な活動において、ユーザーあるいはパートナー企業と直接的にコンタクトを持つことを可能にするシステムです。そのため Web アプリケーションによるユーザー体験(User Experience)は、直接的にサービスや企業のイメージに結び付きやすい傾向にあります。つまり、高い品質のユーザー体験を提供できればサービスや企業の印象もよくなりますし、その逆も成り立ってしまうということです。

従って Web アプリケーションの開発においてはアプリケーションの品質を十分に確認し、また誤ったデータ操作や顧客対応が行われないようにテストを行うことが重要になります。Web アプリケーションは通常は3階層（ブラウザ、アプリケーションサーバー、データベースサーバー）以上に分割され、また他のシステムやサービスと連携を行うことがほとんどです。そのためテストを用意する場合には、それらの階層関係を考慮しつつテストの目的を明確にしたうえでテストを作成する必要があります。

Visual Studio では主に以下のようなテストシナリオにおいて、ツールの支援を利用したテストの作成、実施を行うことが可能になります。



今回の自習書では、上記のパターンのうち、②の単体テスト、③の Web テスト（正確には “Web パフォーマンス テスト” という名称）、および ④の負荷テストの概要を体験いただきます。

※ Visual Studio 2012 試用版の入手方法

Visual Studio 2012 の試用版は無償で以下のWeb サイトよりダウンロード入手いただけます。

■Visual Studio 2012 試用版

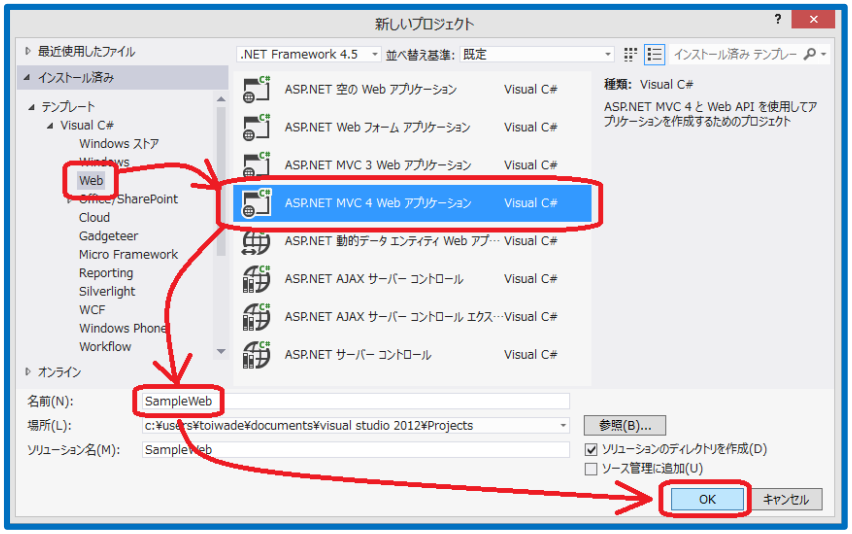
http://www.microsoft.com/visualstudio/jpn/downloads

（本自習書の内容をお試しいただくには Ultimate の環境をご用意ください）

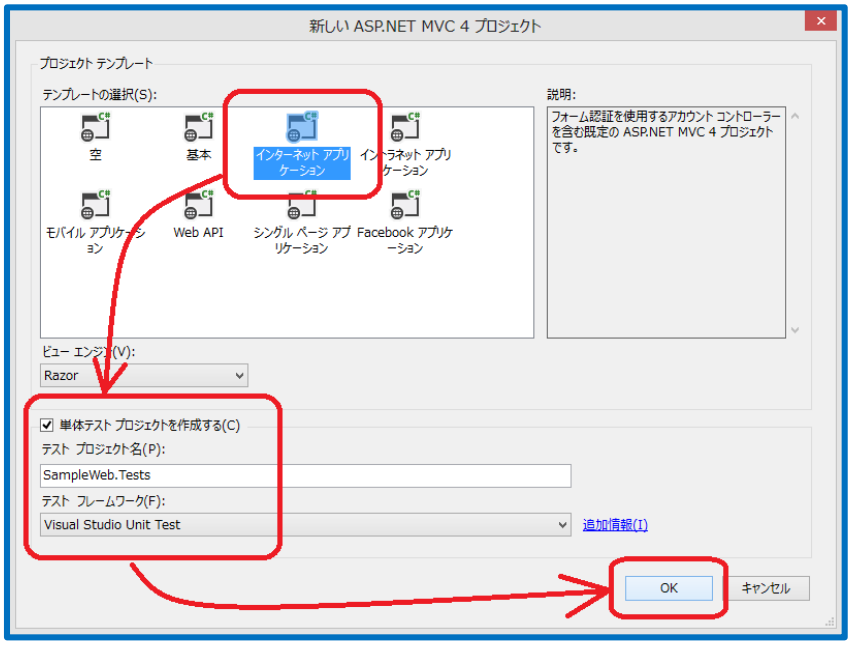
MVC パターンの理解と事前準備

ハンズオンの準備として、ここでは Visual Studio 2012 による MVC フレームワークの概要理解と、テスト用アプリケーションの準備を行います。

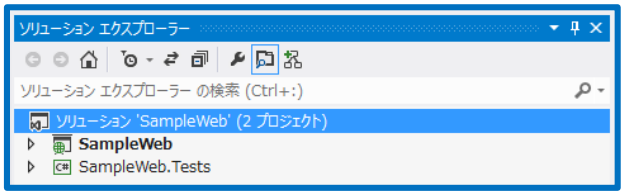
1. 最初に、テスト対象とするアプリケーションを用意します。Visual Studio で新規プロジェクト作成（メニューから「ファイル」->「新規作成」->「プロジェクト」を選択）を行い、表示されたウィザードにおいて、”Web” のプロジェクトテンプレートのなかから “ASP.NET MVC 4 Web アプリケーション” を選択し、プロジェクトの名称として “SampleWeb” を入力し、「OK」をクリックします。



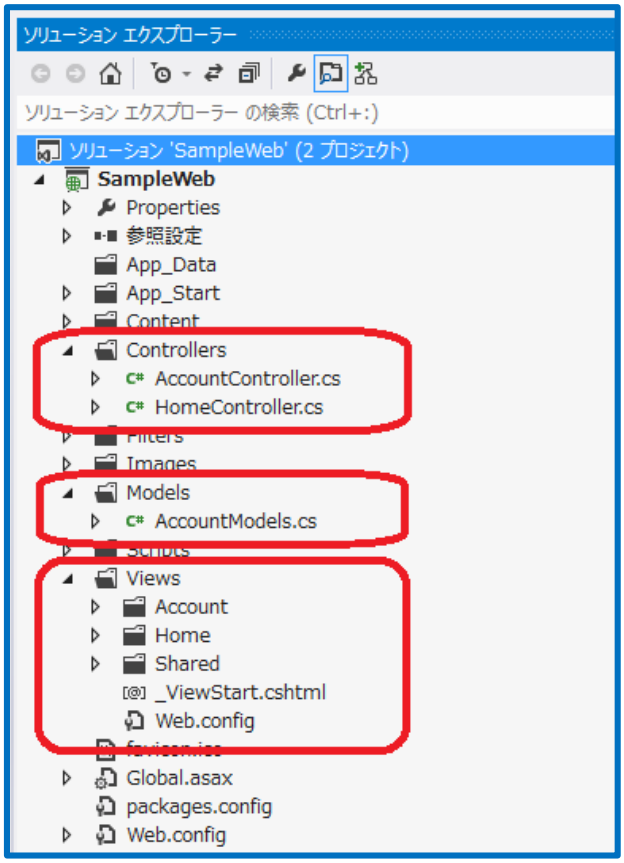
2. 引き続き MVC4 に含まれるテンプレートが表示されますので “インターネット アプリケーション” のテンプレートを選択し、”単体テストプロジェクトを作成する” のチェックボックスを選択したうえで、「OK」をクリックします（ビューエンジンは “Razor” を使用）。



3.　　プロジェクト情報が作成され、”SampelWeb” というソリューションの下に、”SampleWeb” プロジェクトと “SampleWeb.Test” プロジェクトが作成されます。



いくつかのフォルダ、ファイルがありますが、MVC テンプレートを使用する際に最も重要なのは、以下の画面ショットで赤い囲みで囲っているコントローラー(Controllers フォルダの内容)、モデル(Models フォルダの内容)、およびビュー(Views フォルダの内容)です。



ASP.NET MVC フレームワークは、これらの頭文字をとった MVC デザインパターン（Model, View, Controller）をもとにしたテンプレートです。

簡単にそれぞれの役割を記述すると以下のようになります。

Model：モデル

ソフトウェアのロジックの中心となるオブジェクトを集めたもの。MVC パターンでの Model はでは広義の「モデル」を取扱い、純粋なエンティティ オブジェクトだけでなく、エンティティ オブジェクトに対する制御を行うオブジェクトも含みます。

以下は、MVC プロジェクトにおいて Views フォルダの下にある AccountModels.cs に含まれる LocalPasswordModel のコードです。

```
public class LocalPasswordModel
{
    [Required]
    [DataType(DataType.Password)]
    [Display(Name = "現在のパスワード")]
    public string OldPassword { get; set; }

    [Required]
    [StringLength(100, ErrorMessage = "{0} の長さは {2} 文字以上である必要があります。", MinimumLength = 6)]
    [DataType(DataType.Password)]
    [Display(Name = "新しいパスワード")]
    public string NewPassword { get; set; }

    [DataType(DataType.Password)]
    [Display(Name = "新しいパスワードの確認入力")]
    [Compare("NewPassword", ErrorMessage = "新しいパスワードと確認のパスワードが一致しません。")]
    public string ConfirmPassword { get; set; }
}
```

LocalPasswordModel クラスは、MVC プロジェクトにおける下回りの機能（認証関連の機能）を提供するための Model であるため、単純な構造となっていますが、実際に MVC にて開発を進める場合には Model として様々なクラス（オブジェクト）を追加し、ソフトウェアのロジックを作りこむことになります。

View: ビュー

ソフトウェアにおける外観を提供するオブジェクトを集めたもの。今回の MVC テンプレートでは ビューエンジンとして “Razor” (レイザー)記法を利用しています。Razor 記法においては、「@」で開始するキーワードを利用して、HTML の中にコードを埋め込むような形で View を記述します。下記は今回のテンプレートで作成された index.cshtml ファイルの内容です。

Index.cshtml

```
@{
    ViewBag.Title = "Home Page";
}
@section featured {
    <section class="featured">
        <div class="content-wrapper">
            <hgroup class="title">
                <h1>@ViewBag.Title.</h1>
                <h2>@ViewBag.Message</h2>
            </hgroup>
            <p>
                To learn more about ASP.NET MVC visit
                <a href="http://asp.net/mvc" title="ASP.NET MVC Website">http://asp.net/mvc</a>.
                The page features <mark>videos, tutorials, and samples</mark> to help you get the most from ASP.NET MVC.
                If you have any questions about ASP.NET MVC visit
                <a href="http://forums.asp.net/1146.aspx/1?MVC" title="ASP.NET MVC Forum">our forums</a>.
            </p>
        </div>
    </section>
}
```

他方、プロジェクト作成時にビューエンジンとして（これまで同様の） ASPX を使用すると “asp” で始まる独自のタグ（例えば <asp:Content> タグ）を使用し、 View を記述します（下記は、今回作成した SampleWeb とは別に ASPX のビューエンジンを使用した MVC 4 プロジェクトを作成した内容です）。

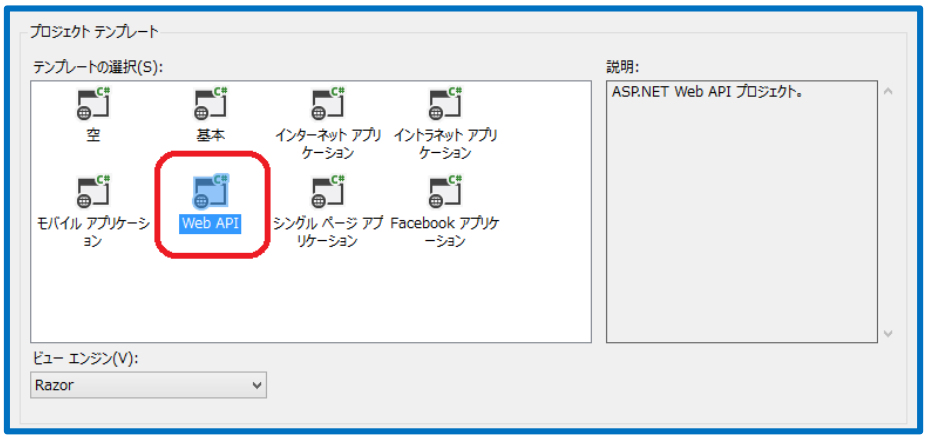
```
Index.aspx
<%@ Page Language="C#" MasterPageFile="~/Views/Shared/Site.Master" Inherits="System.Web.Mvc.ViewPage" %>

<asp:Content ID="indexTitle" ContentPlaceHolderID="TitleContent" runat="server">
    Home Page - My ASP.NET MVC Application
</asp:Content>

<asp:Content ID="indexFeatured" ContentPlaceHolderID="FeaturedContent" runat="server">
    <section class="featured">
        <div class="content-wrapper">
            <hgroup class="title">
                <h1>Home Page.</h1>
                <h2><%: ViewBag.Message %></h2>
            </hgroup>
            <p>
                To learn more about ASP.NET MVC visit
                <a href="http://asp.net/mvc" title="ASP.NET MVC Website">http://asp.net/mvc</a>.
                The page features <mark>videos, tutorials, and samples</mark> to help you get the most from ASP.NET MVC.
                If you have any questions about ASP.NET MVC visit
                <a href="http://forums.asp.net/1146.aspx/1?MVC" title="ASP.NET MVC Forum">our forums</a>.
            </p>
        </div>
    </section>
</asp:Content>
```

MVC のデザインパターンは、Model と View と Controller の相互の依存度を下げるためのパターンで、特に View に関しては Model および Controller **からの**依存度が低くなうようなパターンとなるため、View として Razor と ASPX のいずれのエンジンを使用する場合でも Model や Controller に関しては基本的に共通のロジックで記述することができるようになっています。

Controller: コントローラー

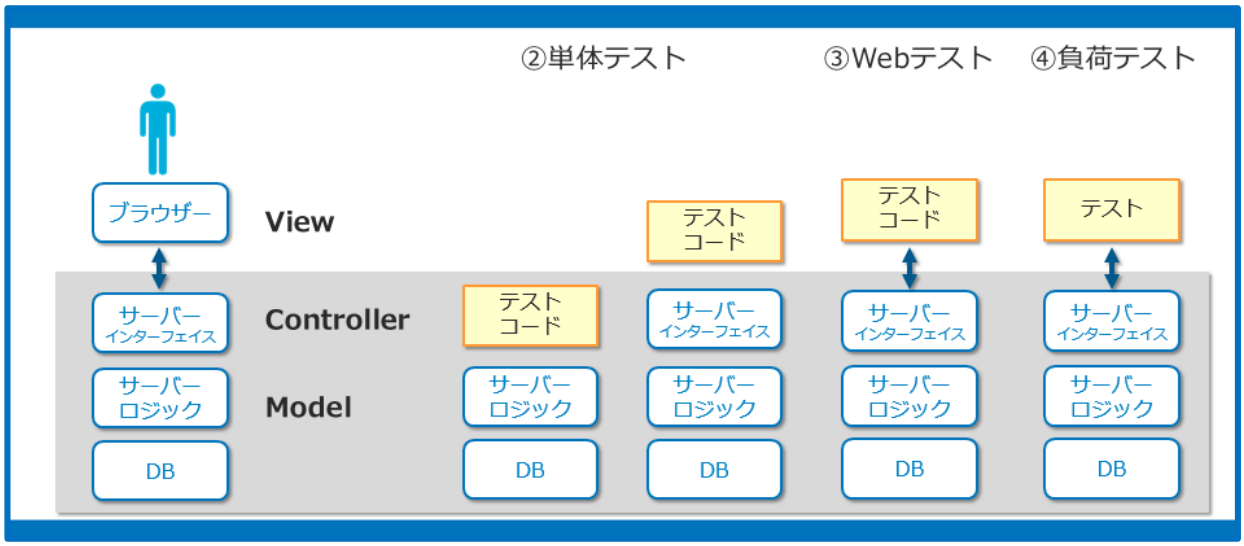
MVC における Controller はソフトウェアと外部のやり取りについて責務を持つオブジェクトです。例えば今回の SampleWeb のように MVC テンプレートにおいて、「インターネット アプリケーション」のテンプレートを選択した際には、URL の一部やフォーム ポスト パラメーターとして渡された情報（引数）を Model に対し渡すことで、適切なビジネスロジックの実行を促します。また、Model によって行われたロジックに基づき、その結果を適切な View に渡す役割も持っています（それを「どのように表示するか」は View の責務になります）。

例えば、MVC テンプレートにおいては「Web API」というテンプレートもありますが、この “Web API” では Model のロジック実行結果を Controller 経由で View に返し表示を行うのではなく、Controller から HTTP Response として返す作りになっています。

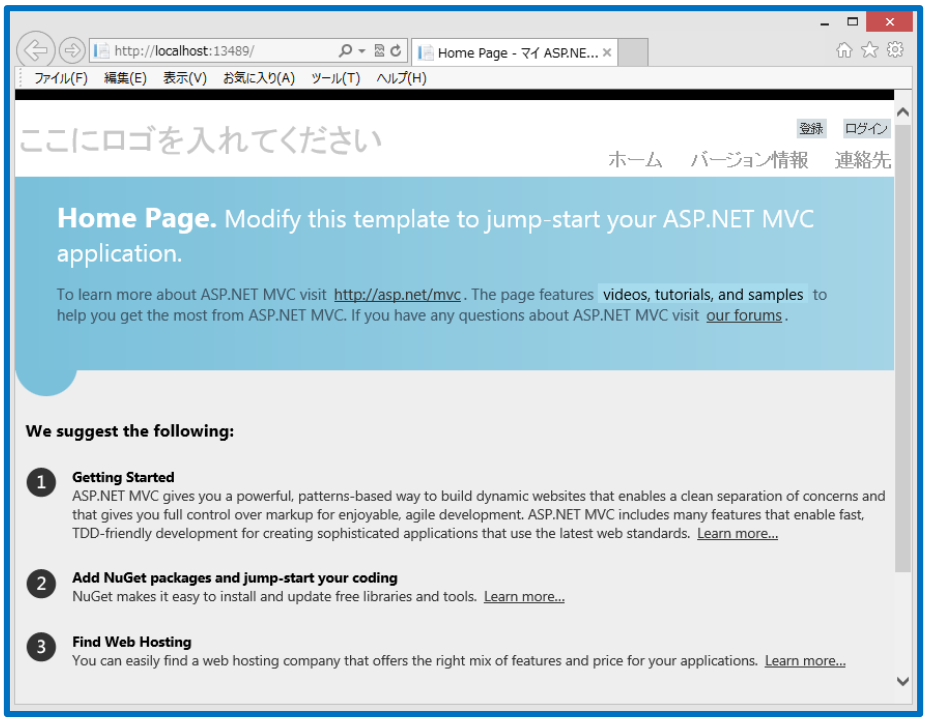


しかし、外部から URL の一部あるいはフォームポスト パラメーターとして渡された情報を Model に引き渡す、そして Model が実行したロジックの結果を外部に返す、という点では同じ責務を持っているといえます。

本自習書冒頭の解説においてテストの分類を行った図を利用して、これら MVC が対応するレイヤと Visual Studio のテスト機能が対象とするレイヤをまとめると以下のようになります。

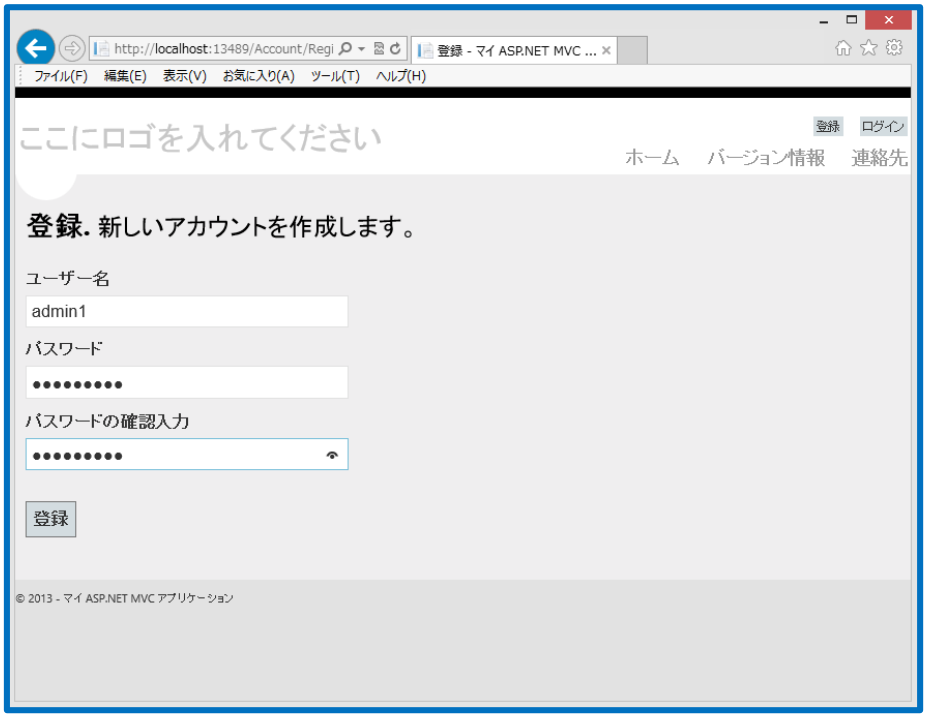


4. 引き続き、この後のテストにて使用するための準備を行います。Visual Studio で、F5 を押し（もしくはメニューから、「デバッグ」->「デバッグの開始」を選択）デバッグを開始します。ローカルPC上で Web アプリケーションがホストされ、以下のような Web が表示されます。



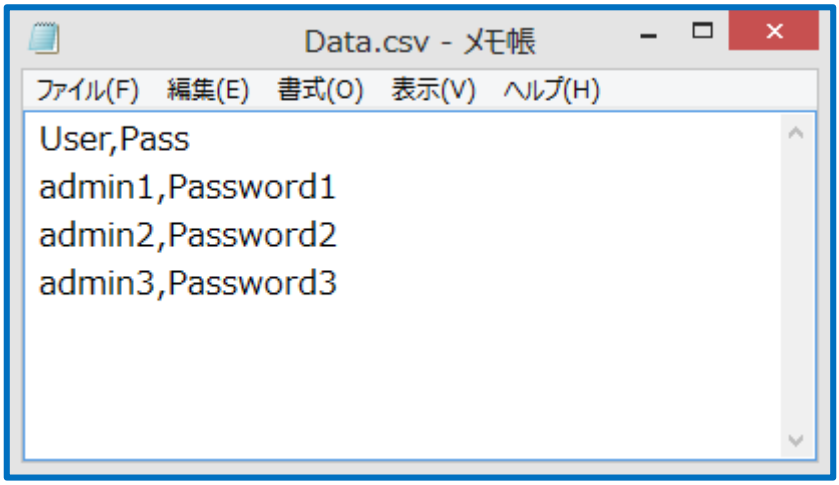
5. 画面右上にある「登録」のリンクをクリックし、ユーザー登録画面で、ユーザー名 “admin1”、パスワー

ド “Password1”、パスワードの確認入力 “Password1” を入力し、「登録」を行います。



6. 登録が成功したら、画面右上の「ログオフ」を押し、ユーザー admin1 でのログイン状態を解除し、同様に “admin2” (パスワードは “Password2”)、および “admin3” (同じく “Password3”) のユーザー登録を行います。これらが完了したら、Web アプリケーションのデバッグを終了してください。

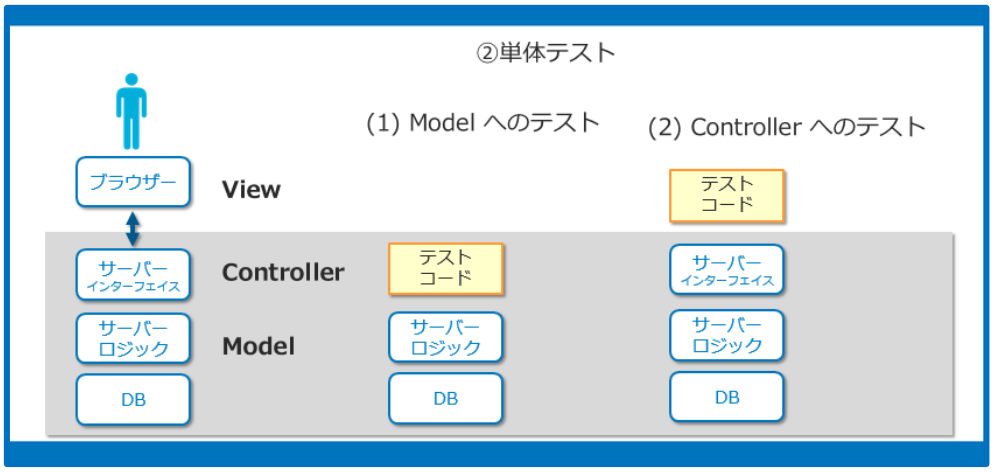
7. “デスクトップ” や “ドキュメント” などのわかりやすいフォルダにて、Data.csv というファイルを作成し、以下のように CSV 形式でデータを入力し、保存してください。



以上で準備は完了です。

Web における単体テスト

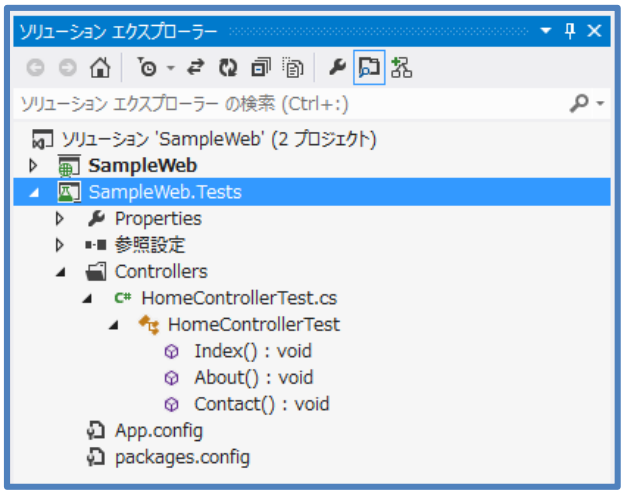
１つ目のハンズオンでは、Visual Studio を使った Web アプリケーションへの単体テストについて学びます。再度整理ですが、MVC パターンにおけるレイヤーと、Visual Studio が提供する単体テスト機能の守備範囲は以下のように分類できます。



つまり上記のように、(1) Model に対する単体テスト、ならびに (2)Controller に対する単体テスト、として単体テストの実施が可能です。

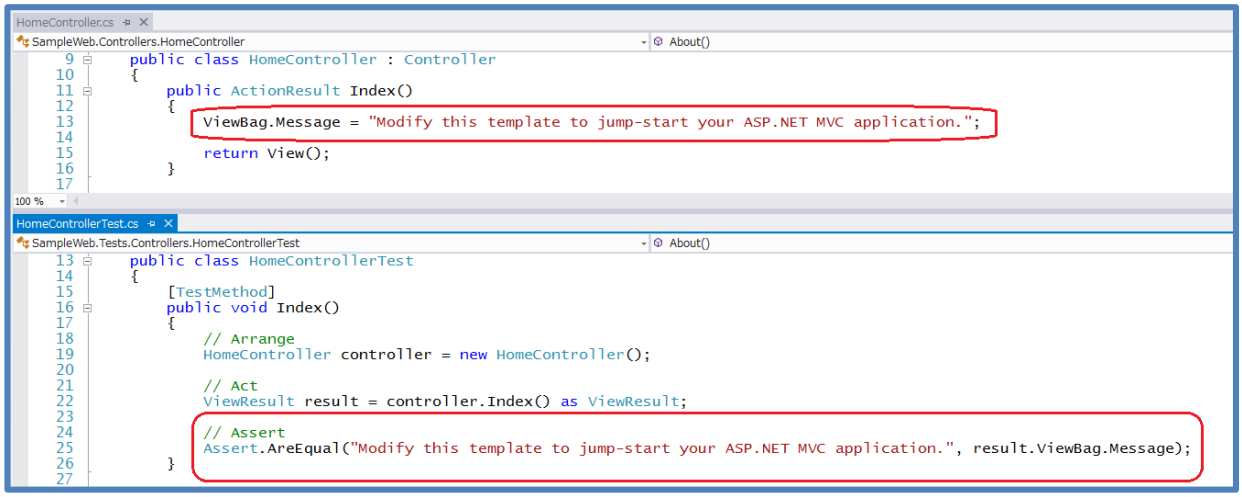
このうち、(1) の Model に対する単体テスト、については基本的に Web 特有の事項はありません。ソフトウェアにおけるロジック全般を担当する Model に対する単体テストの作り込に関しては、本自習書シリーズの初回「Visual Studio 2012 による単体テスト」などを参考にテストをいただくのがよいかと思います。
今回のハンズオンでは、(2) の Controller に対する単体テスト、について確認したいと思います。

1. 今回の自習書では、プロジェクト作成時に単体テストも合わせて作成を行っています。ソリューションエクスプローラーにて、”SampleWeb.Test” プロジェクトを確認すると、”Controllers” フォルダ配下に “HomeController” クラスに対するテストを行う “HomeControllerTest” クラスとそのメソッドが確認できます。



2. 実際にテスト対象となる “HomeController” クラスの “Index” メソッド、および

“HomeControllerTest” クラスにおける “Index” メソッドを比較してみましょう。



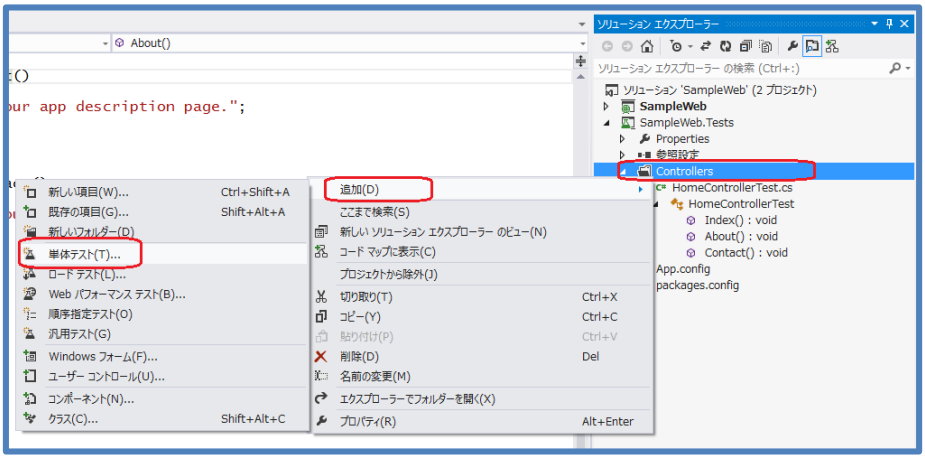
テストクラスである “HomeControllerTest” では、”HomeController” をインスタンス化し、“Index” メソッドを呼び出し、戻り値である ActionResult を取得しています（テスト側ではこれを ViewResult クラスにキャストしています）。
Assert 文としては、「”HomeController” クラスの “Index” を呼び出したのち、ViewBag のメッセージには “Modify this…” という文字列が入っている」という内容で設定し、一つの単体テストとしています。

MVC アプリケーションにおいて Controller に対する単体テストを実施する場合は、プロジェクト作成時に “単体テストプロジェクト” を合わせて作成しておく（事前準備の手順２）ことで、このようなひな形が用意されていますので、これをカスタマイズして単体テストを作りこむことが可能です。

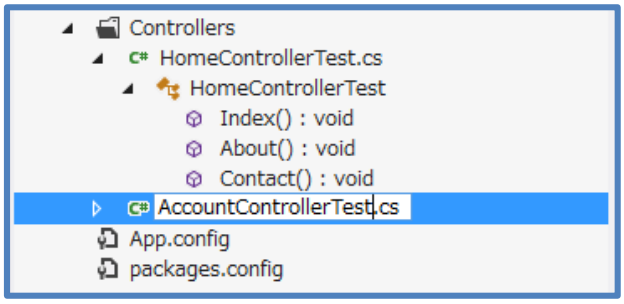
さて、”SampelWeb” プロジェクトにおいては “HomeController” の他に “AccountController” も存在しています。しかしながら、テストプロジェクトである “SampleWeb.Test” プロジェクトには “AccountController” のテスト用コードがありません。

実際のところ、Web アプリケーションの Controller 部分の振る舞いは、他のコンポーネントや MVC Framework の機能に依存している箇所が多く、テスト容易性の観点から考えると、”HomeCotroller” のように単純なテストパターンしか出ない Controller はむしろ少数派になります。Post メソッドが多く、また他のコンポーネントへの依存度の高い “AccountController” のテストクラスがデフォルトで存在しないのはそのような理由もあるかと思います。

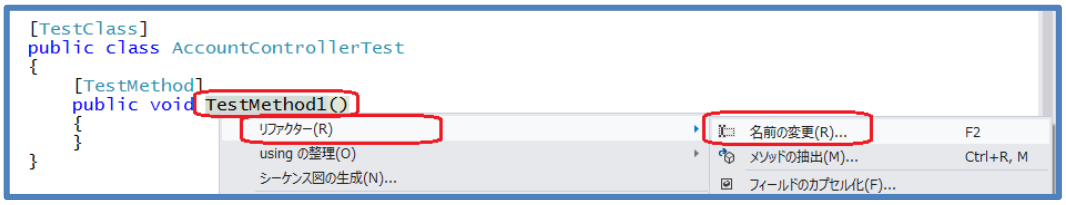
3.　デフォルトでは存在しない “AccountController” のテストクラスですが、今回の “SampleWeb.Test” プロジェクトにおいて作成を行ってみましょう。ソリューション エクスプローラーにおいて、”SampleWeb.Test” プロジェクトの “Controllers” フォルダにて右クリックを行い、表示されたメニューから「追加」→「単体テスト」を選択します。



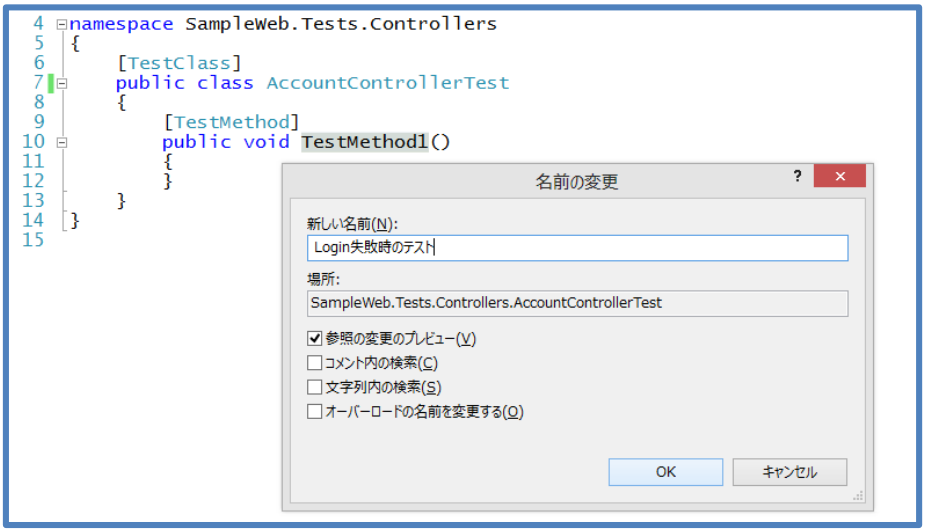
4. 作成されたファイルの名前を “AccountControllerTest.cs” に変更してください。



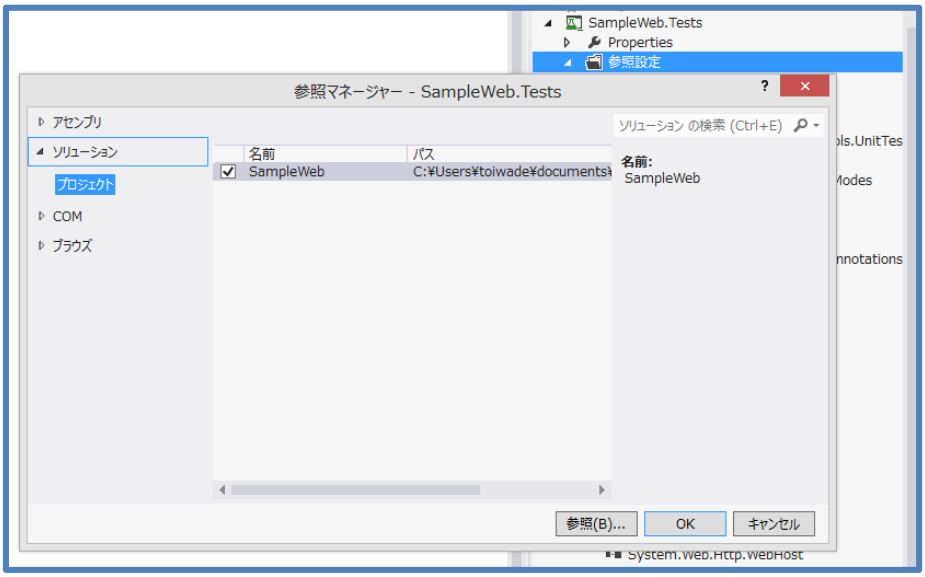
5. “AccountControllerTest” クラスを開き、テストメソッドの名称を変更しましょう。“TestMethod1” を選択した状態で右クリックを押し、表示されたメニューから 「リファクター」→「名前の変更」を選びます。



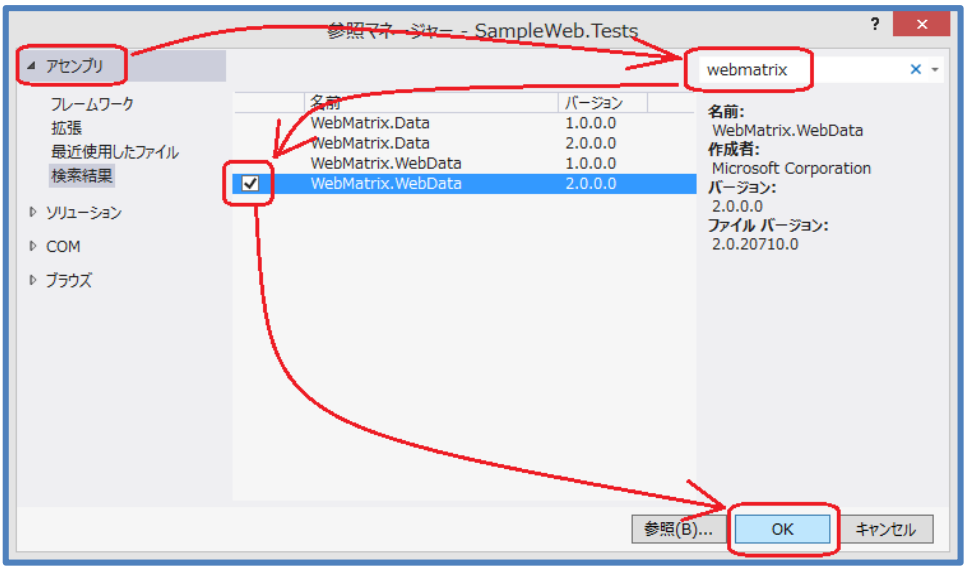
6. 新しい名前として “Login失敗時のテスト” と入力し、「OK」を押してください。引き続き表示されるプレビュー内容を確認し、「適用」を押します。



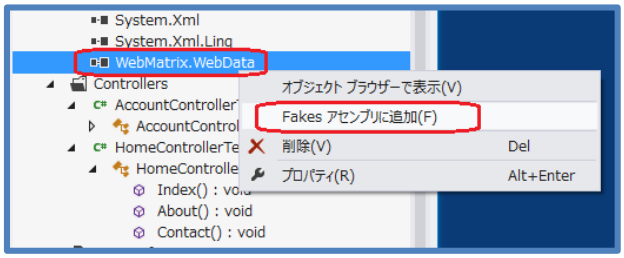
7.　　次に、ソリューションエクスプローラーにて、”SamapleWeb.Test” プロジェクトの参照設定に、同一ソリューション内の “SampleWeb” を追加します（”SampelWeb.Test” の “参照設定“ を右クリックして出てくるメニューにて、”参照の追加”を選択すると下記の画面ショットのようなウィザードが表示されるので、そこで追加を行います）。



8.　　同様に今回のテストにて使用する　WebMatrix.WebData アセンブリを追加します。”参照の追加“　で表示されるウィザードにおいて、”アセンブリ” を選択し、検索ボックスに “webmatrix” と入力します。いくつかのアセンブリが表示されますが、このうち “WebMatrix.WebData” のバージョン “2.0.0.0” を選択し、「OK」を押します。



9.　　“SampleWeb.Test” プロジェクトの “参照設定” の項目に追加され、表示された “WebMatrix.WebData” アセンブリを右クリックし、表示されたメニューから “Fakes アセンブリに追加“ を選択します。



10. AccoountControllerTest を以下のように変更します。

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

using System.Web.Mvc;
using SampleWeb;
using SampleWeb.Models;
using SampleWeb.Controllers;
using Microsoft.QualityTools.Testing.Fakes;

namespace SampleWeb.Tests.Controllers
{
    [TestClass]
    public class AccountControllerTest
    {
        [TestMethod]
        public void Login失敗時のテスト()
        {
            AccountController controller = new AccountController();
            LoginModel model = new LoginModel();
            ViewResult result;

            // (1) 1回目の Assert
            Assert.AreEqual(0, controller.ModelState.Count);

            // (2) Fake による関連コンポーネントの動作の偽装
            using (ShimsContext.Create())
            {

WebMatrix.WebData.Fakes.ShimWebSecurity.LoginStringStringBoolean
                    = (str1, str2, bool1) => false;
                result = controller.Login(model, "") as ViewResult;
            }

            // (3) 2回目の Assert
            Assert.AreEqual(1, controller.ModelState.Count);
        }
    }
}
```

このテストメソッドでは、(1) の一回目の Assert において、AccountController オブジェクトの ModelState においてエラーがない状態であることを確認しています（ModelState に情報がないことを、Count プロパティの値が 0 であることを確認することで確かめています）。

また、(2) のコードにより、AccountController クラスの Login(LoginModel, String) のメソッドにおいて、ログイン情報を確認する WebSecurity クラスの Login(String, String, Boolean) メソッドが必ず失敗するように設定を行っています。ここで、WebSecurity クラスの Login メソッドに “登録された正しいユーザー情報が渡さ

れた際に True が返ること”、あるいは “登録されたユーザー情報に一致しないユーザー名やパスワードが渡った際には False が返ること”、という確認は 「AccountController クラスの Login メソッドの単体テスト」の対象外になることに注意してください。これは AccountController の Login メソッドとしては、「引数として受け取ったユーザー情報を WebSecurity クラスに引き渡し、その結果をもとに画面遷移（次のViewの表示）を行ったり、画面（元のログイン画面の View）にエラーメッセージを渡す」ということを責務にしているためです（シナリオテスト、あるいは結合テストとして、「ログイン画面で正しいユーザー情報が入力された場合に、ログインに成功し次の画面に遷移する」といったテストを用意することはあります。）

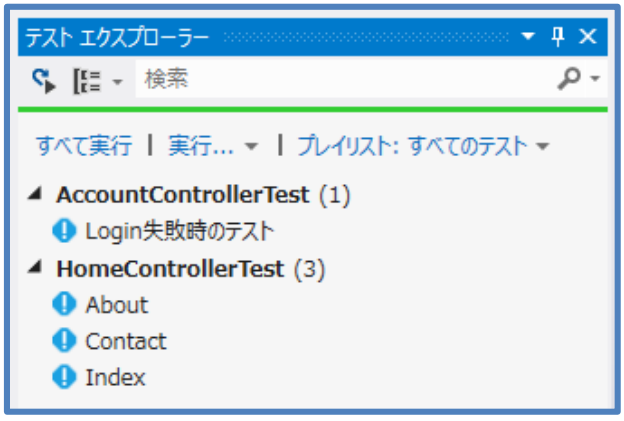
さて、(2)の Fake を使ったコードより、この後に AccountController の Login(LoginModel, String) メソッドが呼ばれた際に、ログイン情報の検証に失敗した（ユーザー名とIDの情報が存在していない）場合の、AccountController の振る舞いをテストできることになります。

```
//
// POST: /Account/Login

[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public ActionResult Login(LoginModel model, string returnUrl)
{
    if (ModelState.IsValid && WebSecurity.Login(model.UserName, model.Password, persistCookie: model.RememberMe))
    {
        return RedirectToLocal(returnUrl);
    }

    // ここで問題が発生した場合はフォームを再表示します
    ModelState.AddModelError("", "指定されたユーザー名またはパスワードが正しくありません。");
    return View(model);
}
```

Fake によりどのような情報が渡されても必ず "false" を返すように偽装されている

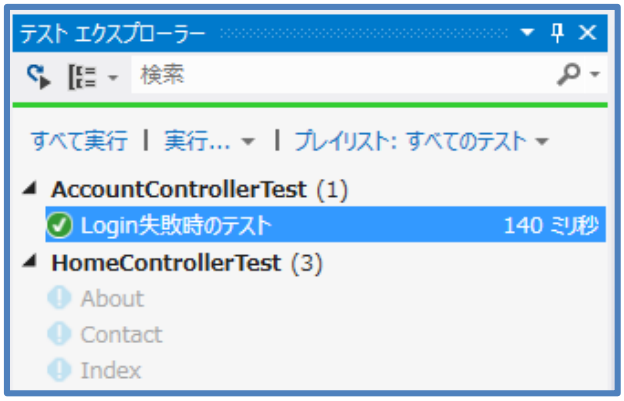
認証失敗時のロジック

11. ソリューションのビルドを行うと、テスト エクスプローラーに以下のように今回作成した “Login失敗時のテスト” のメソッドが表示されます（テスト エクスプローラーが表示されない場合、Visual Studio のトップメニューの「テスト」→「ウィンドウ」→「テスト エクスプローラー」と選択ください）。



12. テストエクスプローラーにおける “Login失敗時のテスト” を右クリックし、表示されたメニューから「選択したテストの実行」を選択し、テストを実行してください。

13.　テストが成功し、以下のように表示されます。これにより 10 の画面ショットにあった (3) の Assert にも成功した（ModelState にエラーが一つ追加された）ことが確認できます。



今回の “Login失敗時のテスト” では、他のコンポーネントとの依存性がそれほど多くないパターン(WebSecurity の偽装のみで動作するパターン)でしたが、Controller はソフトウェアと外部のやり取りについて責務を持つオブジェクトであるため、通常はこれ以外にも様々なコンポーネントとの関連を持っています。
これらを勘案しつつ Controller の単体テストを作成することは場合によってはコストがかかることもあります。

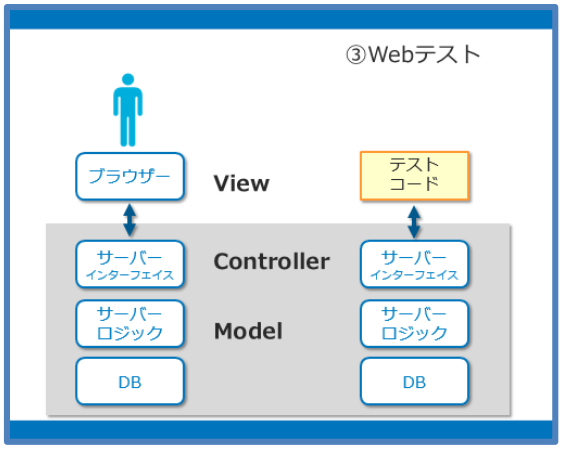
Controller に対する単体テストの実装が難しい場合や、関連するコンポーネントの動作も合わせてテストしたいような場合、次の章で紹介する “Web パフォーマンステスト” を利用してテストを行う、という選択肢もあり得ます。

ここでは以上で 1 つ目のハンズオンを終了します。
1 つ目のハンズオンでは、Visual Studio を使った Web アプリケーションへの単体テストについて学びました。

(1) **単体テストの分類（Model への単体テストと、Controller への単体テスト）**
(2) **Controller に対する単体テスト**
　(ア) **HomeControllerTest の内容の確認**
　(イ) **AccountControllerTest の作成と Fake を使った依存性の排除**

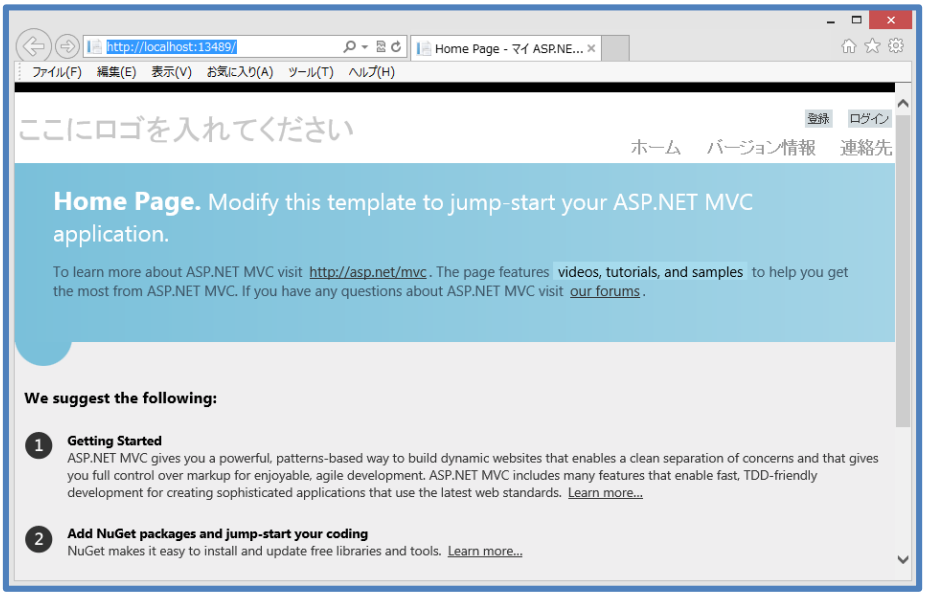
Web パフォーマンス テスト

2つ目のハンズオンでは、Visual Studio を使った Web パフォーマンス テストを体験します。

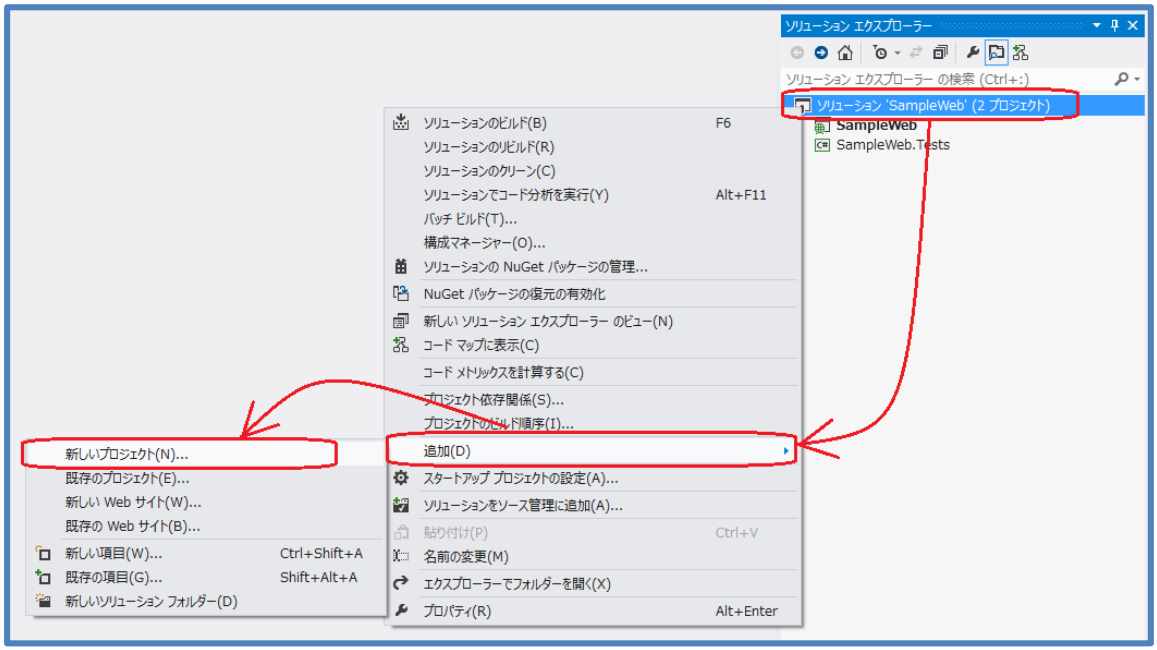


Web パフォーマンステストは、Controller に対するテストとして実施されますが、単体テストとは異なり HTTP Request を通して Controller にアクセスします。つまり、IIS や ASP.NET の基本的な仕組みを経由して Controller へアクセスを行います。
このため、Controller 単体で考えると関連するコンポーネントが増えるため、単体テストとしての純粋性は薄れてしまいますが、ASP.NET MVC 自体がルーティングやリダイレクト、メンバーシップといった機能を活用したフレームワークであるため、これらを包括的に使用した状態での Controller に対するテストが可能になります。

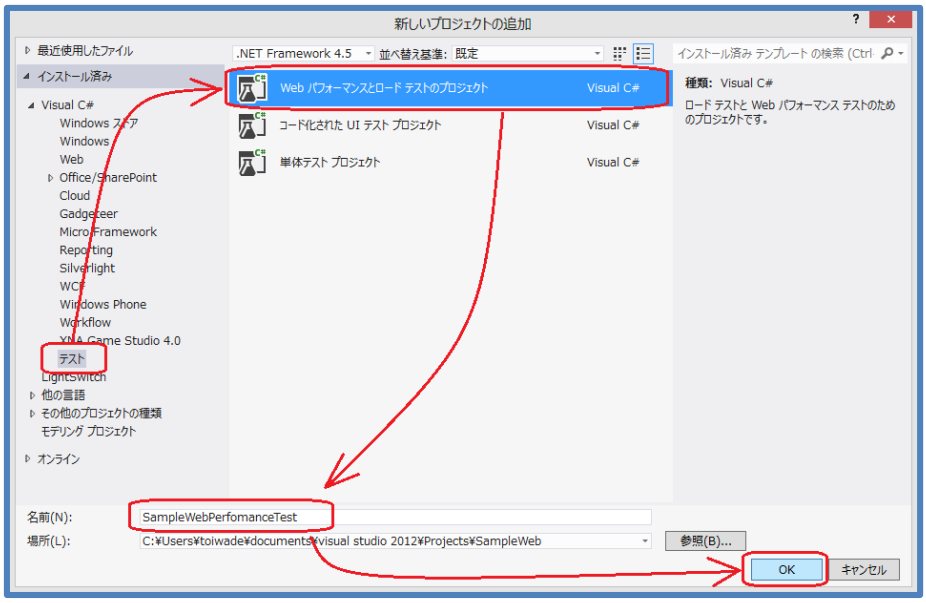
1. Web アプリケーションのホスティング先として、今回はローカルPC上のデバッグ用の環境を利用します。”SampleWeb” プロジェクトをデバッグ実行し、立ち上がったブラウザにおいてアクセスしている URL を確認してください。通常は “http://localhost” にポート番号を加えた URL となります。下記の画面ショットでは http://localhost:13489/ となっています（ポート番号は環境や実行タイミングにより異なります）。URL が確認できたら、デバッグ実行を終了してください。



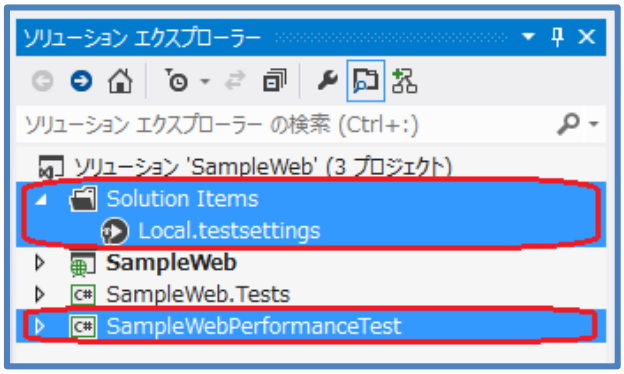
2.　　ソリューション エクスプローラーの “SampleWeb” ソリューションの項目で右クリックを押し、表示されたメニューから「追加」->「新しいプロジェクト」を選択します。



3.　　表示されたウィンドウで「テスト」のカテゴリから「Web パフォーマンスとロードテストのプロジェクト」を選択し、名前として “SampleWebPerformanceTest” と入力し、「OK」をクリックします。



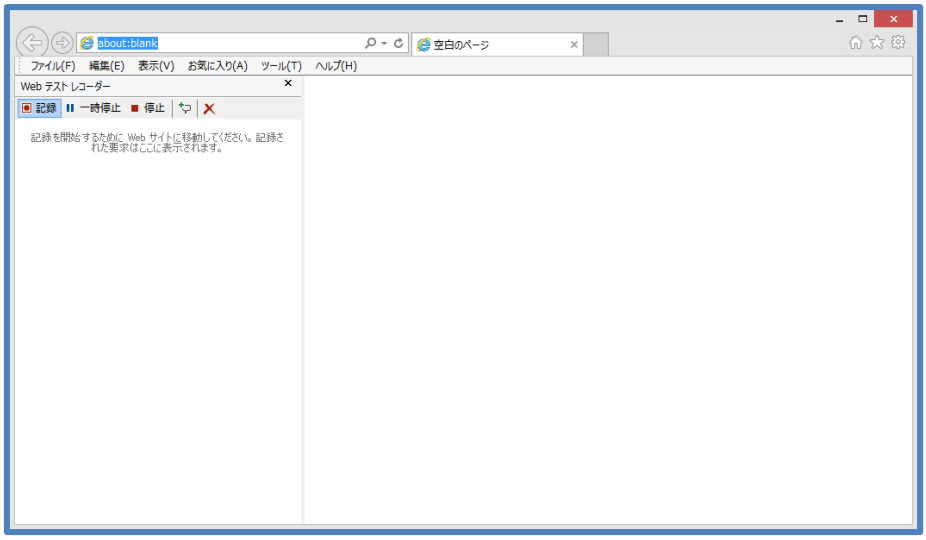
4.　　これにより、ソリューションに “SampleWebPerformanceTest” プロジェクトが追加されるとともに、”Solution Items” として “Local.testsettings” ファイルが追加されます。



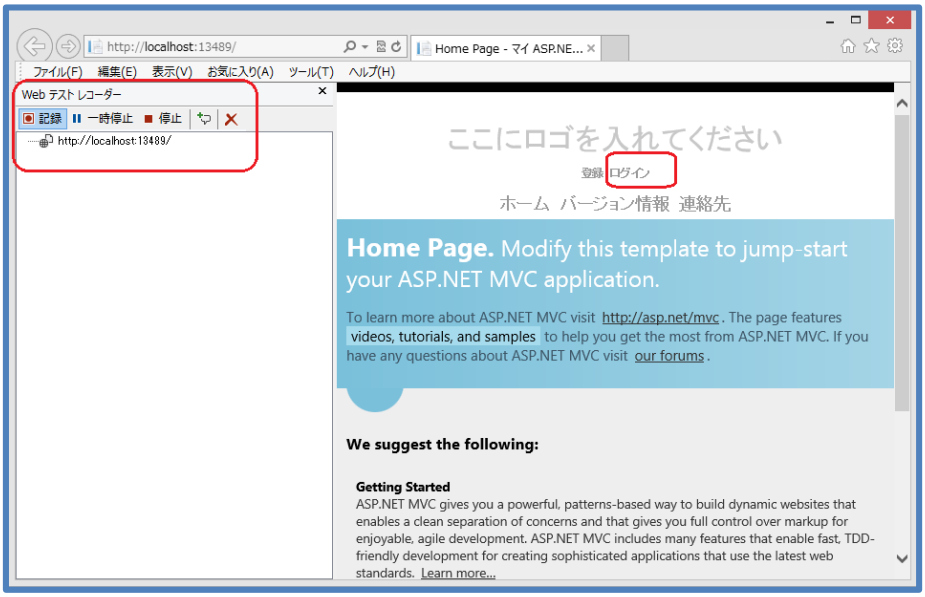
※ Local.testsettings ファイルは、「Web パフォーマンスとロードテストのプロジェクト」の追加により新規に作成されます。既存の単体テストのプロジェクト “SampleWeb.Test” に新しい項目として Web パフォーマンステストや、ロードテストを追加しても作成されませんのでご注意ください（Local.testsettings ファイルはこの後の手順で使用します）。

5. 下記のように、WebTest の定義情報（”WebTest1.webtest” ）が表示されるので、メニューから「記録の追加」ボタンをクリックします。

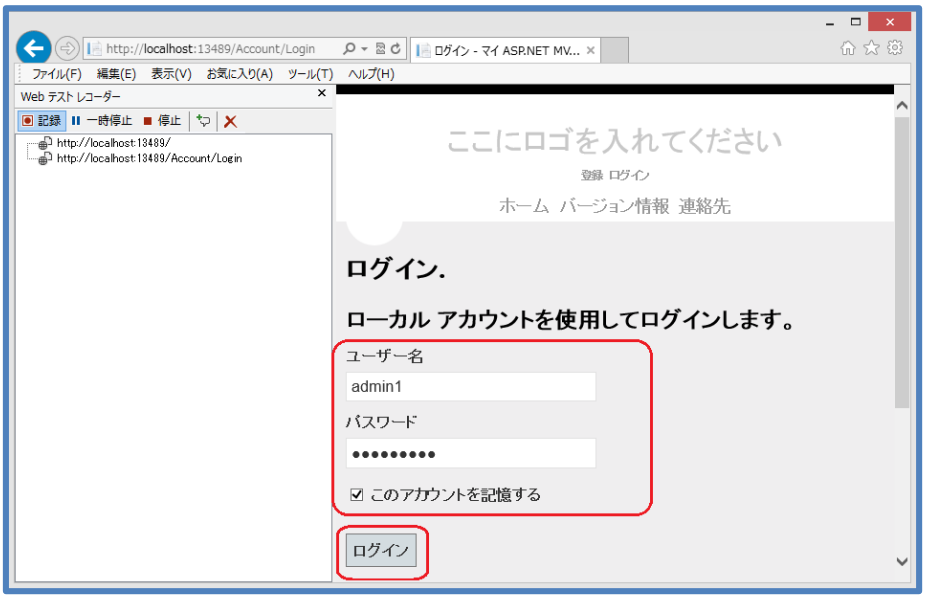
6. ”Web テスト レコーダー” のアドインが表示された状態でブラウザが立ち上がります。



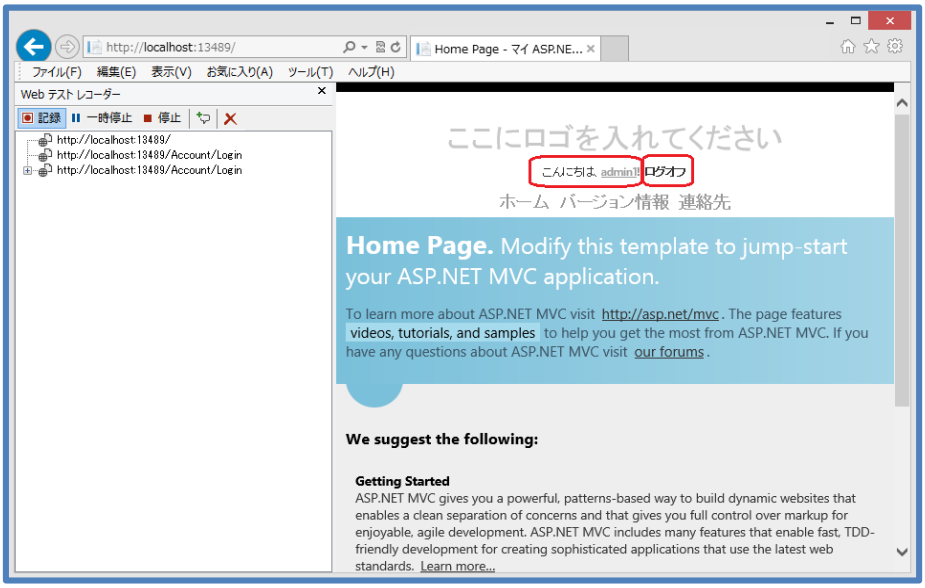
7. 手順1 で確認した URL をブラウザに入力します。ローカルPC上の Web サーバーにアクセスし、画面が表示されます。この際に “Web テスト レコーダー” において、該当する URL にアクセスしたことが記録されています。表示された Web 画面において「ログイン」をクリックしてください。



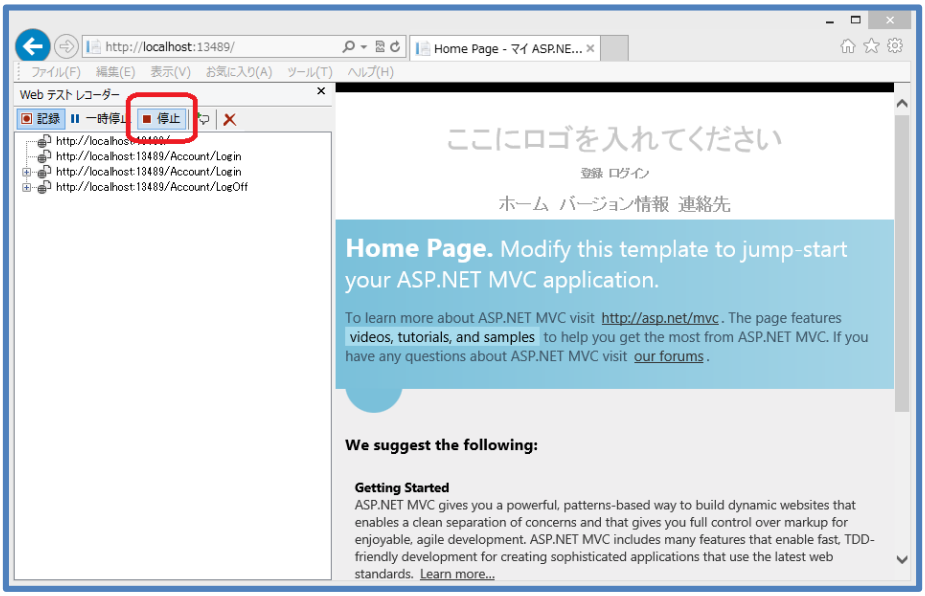
8. ログイン画面が表示されますので、ユーザー名に “admin1”、パスワードに “Password1” を入力し、また、「このアカウントを記録する」にチェックをつけて、「ログイン」をクリックしてください。



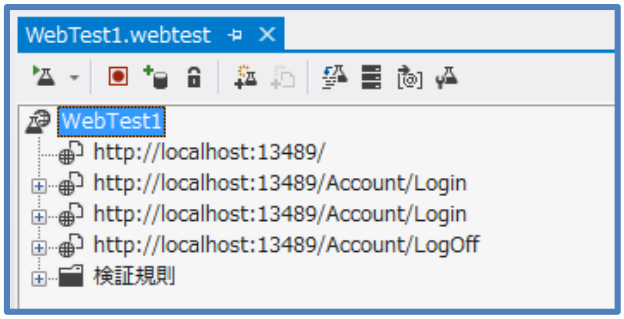
9. 画面上に「こんにちは」＋「ユーザー名」＋「!」、というメッセージが表示されていることを確認し、「ログオフ」をクリックしてください。



10. “Web テスト レコーダー“ の停止を押してください。ブラウザが閉じ、Visual Studio に戻ります。

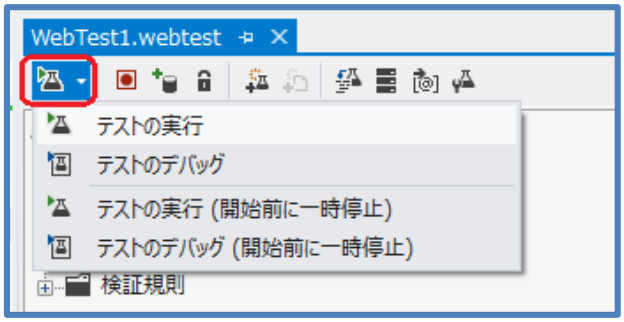


11. Visual Studio において “Web テストレコーダー” にて記録された情報を元に、以下のように WebTest が表示されます。この WebTest ではサーバー（今回の場合はローカルPC）に対して送信された HTTP Request の内容がテストの内容として定義されています。



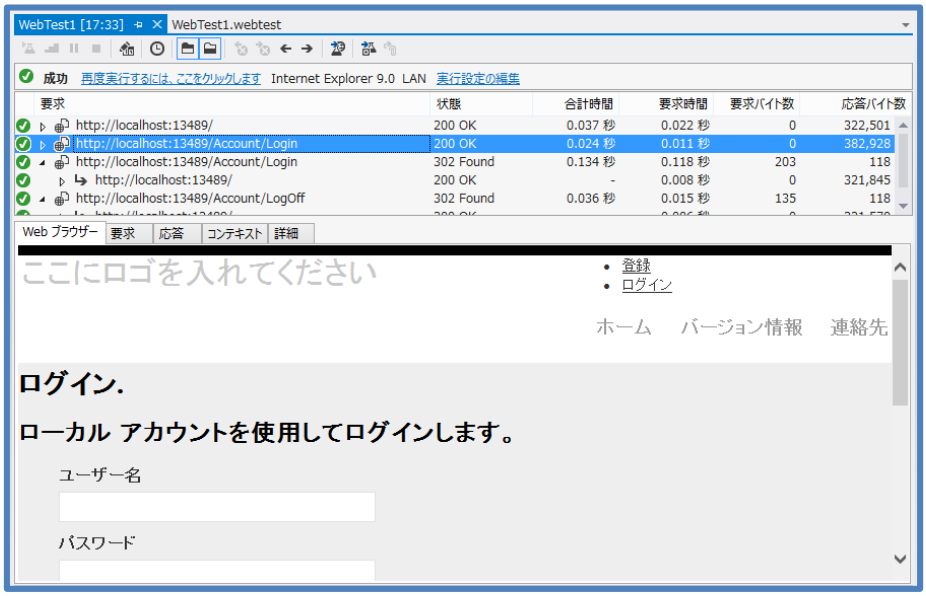
12. 左上のメニュー（三角フラスコのアイコン）をクリックするとメニューが展開されるので、「テストの実

行」を選択します。



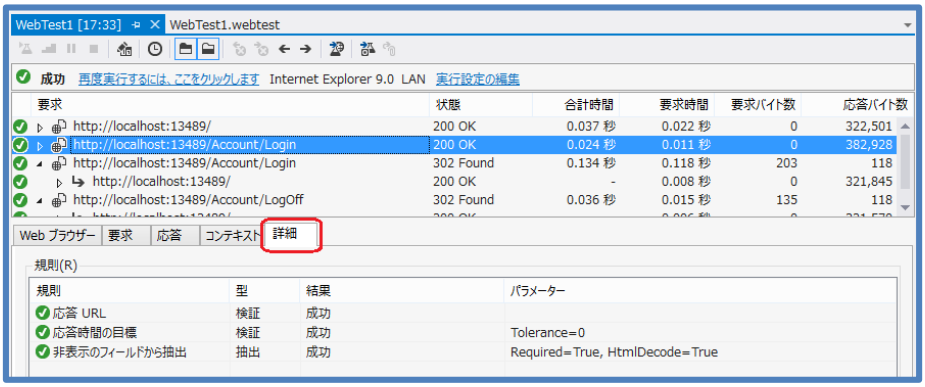
※　Web パフォーマンステストで作成したテストは、「テスト エクスプローラー」で表示されません。

13.　　WebTest に定義されているリクエスト順に Web サーバーへのアクセスが行われ、すべてのリクエストの実行が終了すると、以下のようなテスト結果が表示されます。

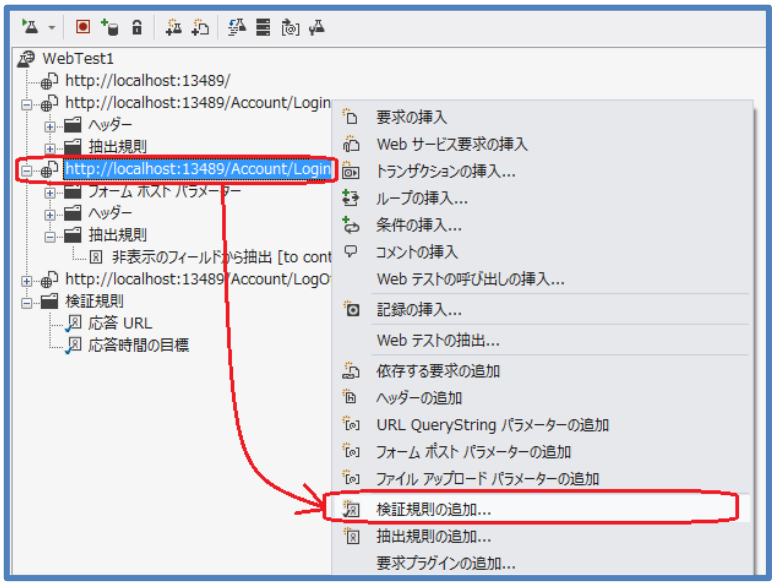


Web パフォーマンステストでは、記録した HTTP Request を連続して再実行することが可能で、その際にサーバーから応答を受けるまでにかかった時間や、受信したデータのバイト数などを確認できます。また、各リクエストに関して、リクエスト（「要求」）の内容やレスポンス（「応答」）の内容を確認できます。

また、「詳細」のタブを利用すると、Web 操作を記録した際に受け取ったいくつかの情報（例えば POST リクエスト後の画面遷移の情報）などから自動的に生成された検証規則に対して、今回のテストの実施結果は合致しているかどうかを確認することが可能です。

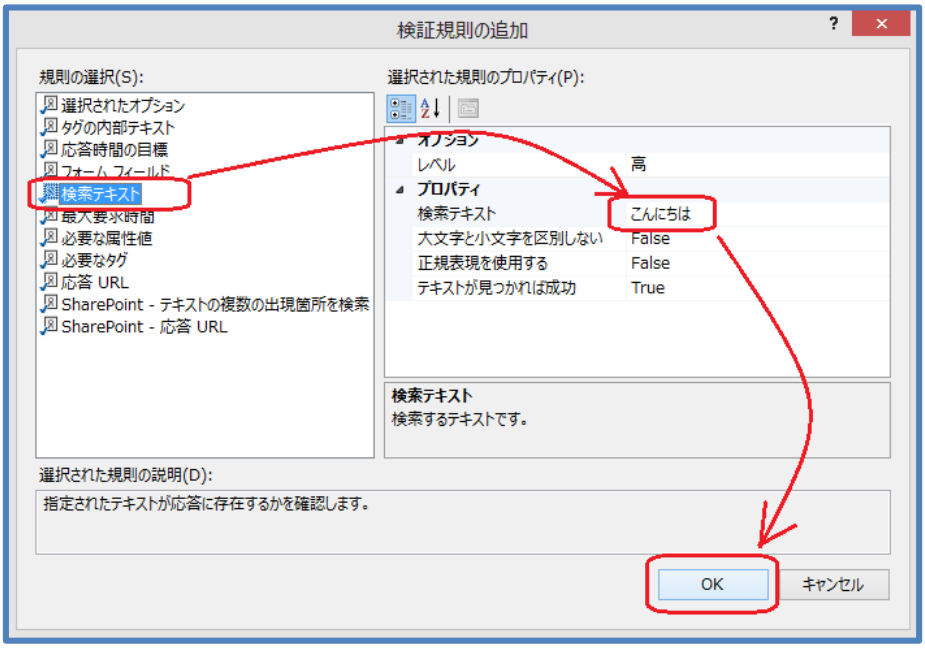


14.　　次に、この Web パフォーマンステストをカスタマイズしてみたいと思います。手順 6 で紹介したように、今回の Web アプリケーションでは、ログイン後に「こんにちは」＋「ユーザー名」＋「!」というメッセージが表示されます。この内容の確認を自動化するための「検証規則」を追加しましょう。
WebTest の定義において、2つ目の “http://<サーバー名>/Account/Login” のリクエスト部分で、右クリックを押し、表示されたメニューから「検証規則の追加」を選択します。

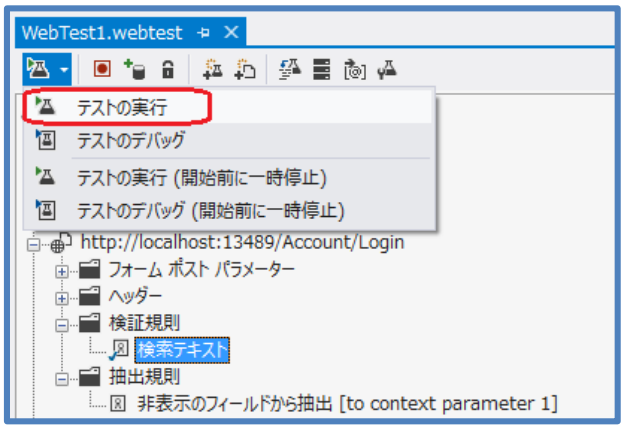


※　今回の自習書の手順では “http://<サーバー名>/Account/Login” のリクエストが 2 つあります。一つ目は Login 画面を表示するための GET リクエストで、二つ目は実際にログインを行うためにユーザー名、パスワードを送信している POST リクエストです。今回の検証規則は、この二つ目のリクエストに対して追加します。

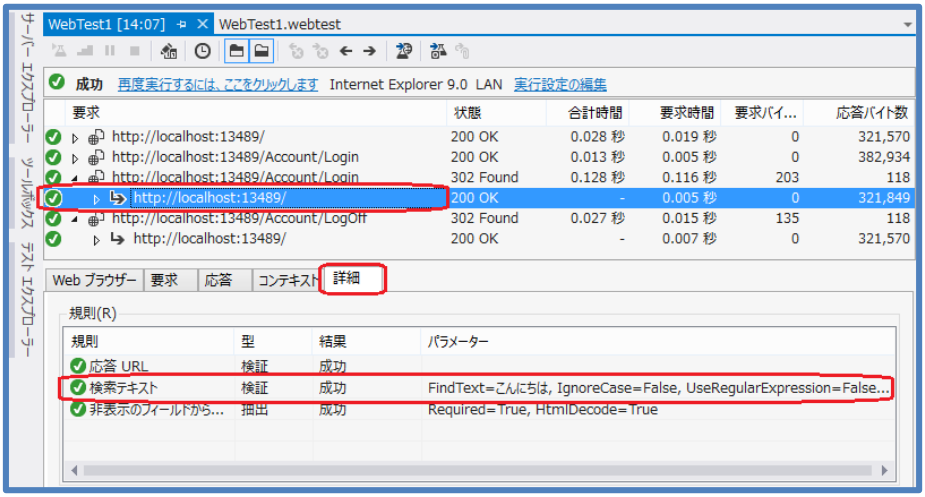
15.　　「検証規則の追加」のウィンドウが表示されますので、左側の一覧から「検索テキスト」の項目を選択し、右側に表示された「選択された規則のプロパティ」の「検索テキスト」のプロパティに、”こんにちは” と入力し、「OK」をクリックしてください。



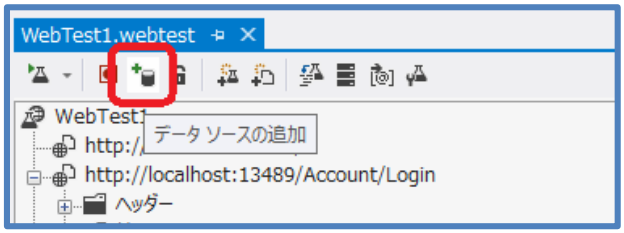
16. WebTest の定義の左上のメニュー（三角フラスコのメニュー）から「テストの実行」を選択します。



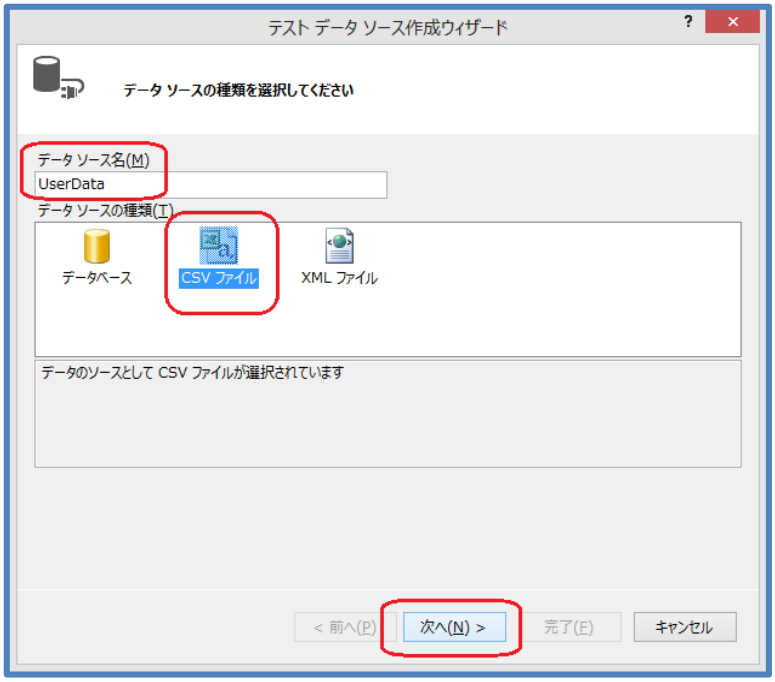
17. テストの結果において、2つ目の “http://<サーバー名>/Account/Login” のリクエストの結果として表示される “http://<サーバー名>” のレスポンス情報において「詳細」タブを確認すると、「検索テキスト」として、「こんにちは」の文字列の検証が行われていることを確認できます。



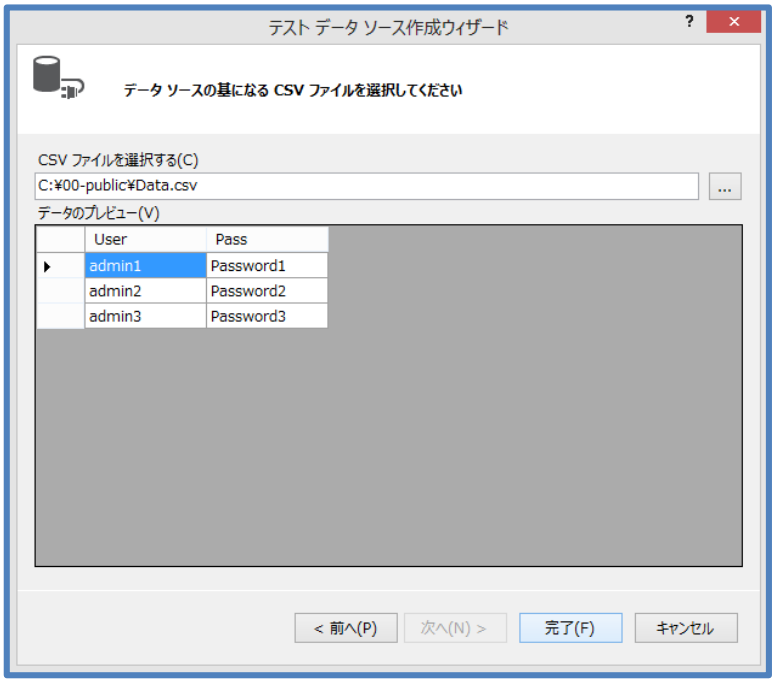
18.　次にフォーム ポスト パラメーターにおいて、外部データを使用する方法を確認します。WebTest の定義のメニューにおいて、「データソースの追加」メニューをクリックします。



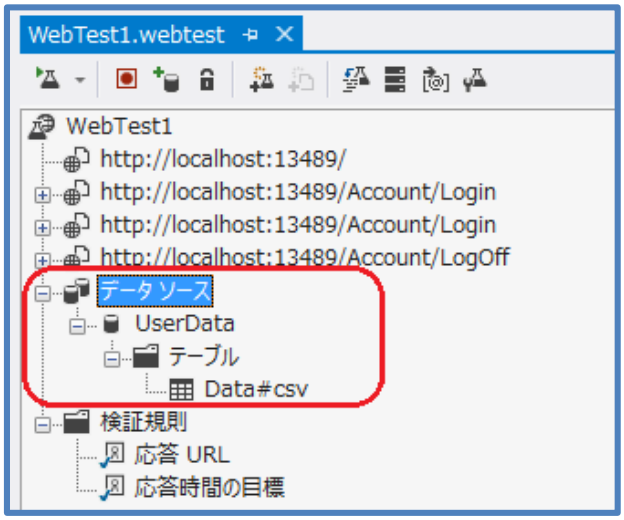
19.　表示された「テスト データ ソース作成ウィザード」にて、データソース名に “UserData” と入力し、種類として “CSVファイル” を選択し、「次へ」をクリックします。



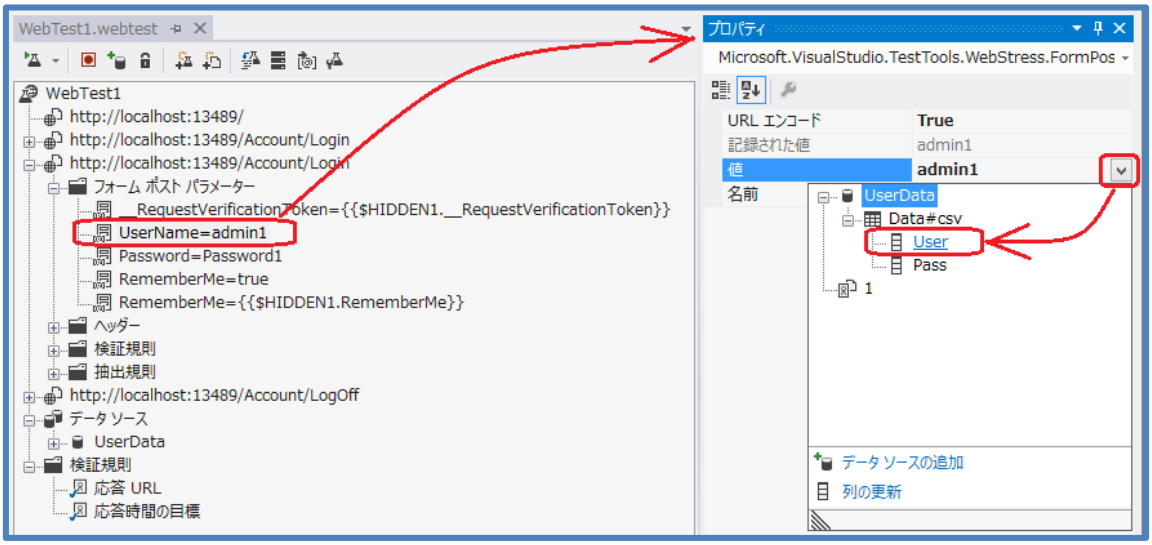
20.　事前準備の「手順７」で作成した “Data.csv” を選択します。選択後、データのプレビューが表示されますので、確認し「完了」をクリックします。（この後、データソースのファイルを現在のプロジェクトに追加してよいか、という確認のダイアログが出ますので、「はい」を押して追加を行ってください）



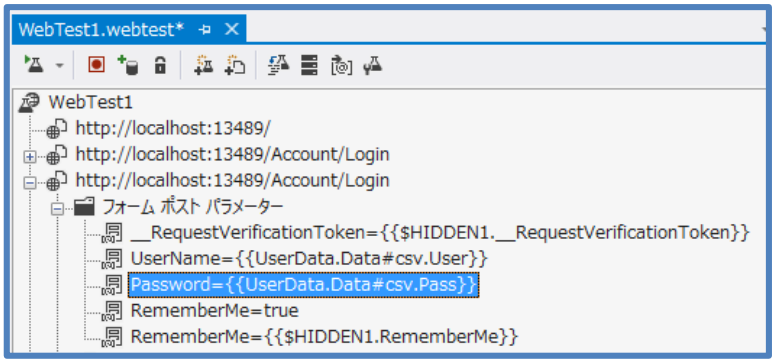
21. WebTest の定義において、以下のように「データ ソース」が追加されていることを確認してください。



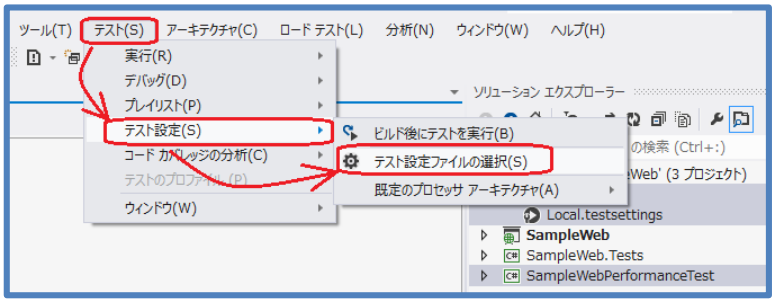
22. 次に、このデータソースをフォームポスト パラメーターで使用するように設定を行います。WebTest の定義で2つ目の “http://<サーバー名>/Account/Login” のリクエストの内容を展開し、”フォーム ポスト パラメーター” の中に記述されている UserName の項目を右クリックし、表示されたメニューから「プロパティ」を選択しプロパティを表示します。「値」の項目を選択した際に右側に表示されるドロップダウンメニューをクリックし、表示されるデータの中から「UserData」を選択し、展開した内容から “User” を選択します。



23.　同様にフォームポストパラメーターの “Password” においても、「UserData」の “Pass” とのバインドを設定します。

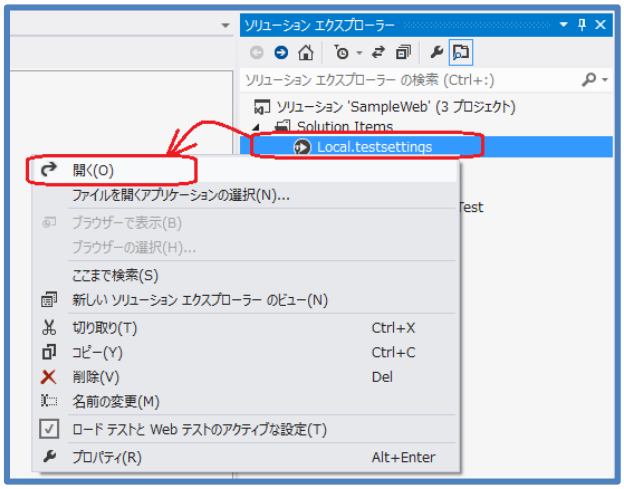


24.　Visual Studio のメニューから、「テスト」を選択し、「テスト設定」の「テスト設定ファイルの選択」をクリックします。

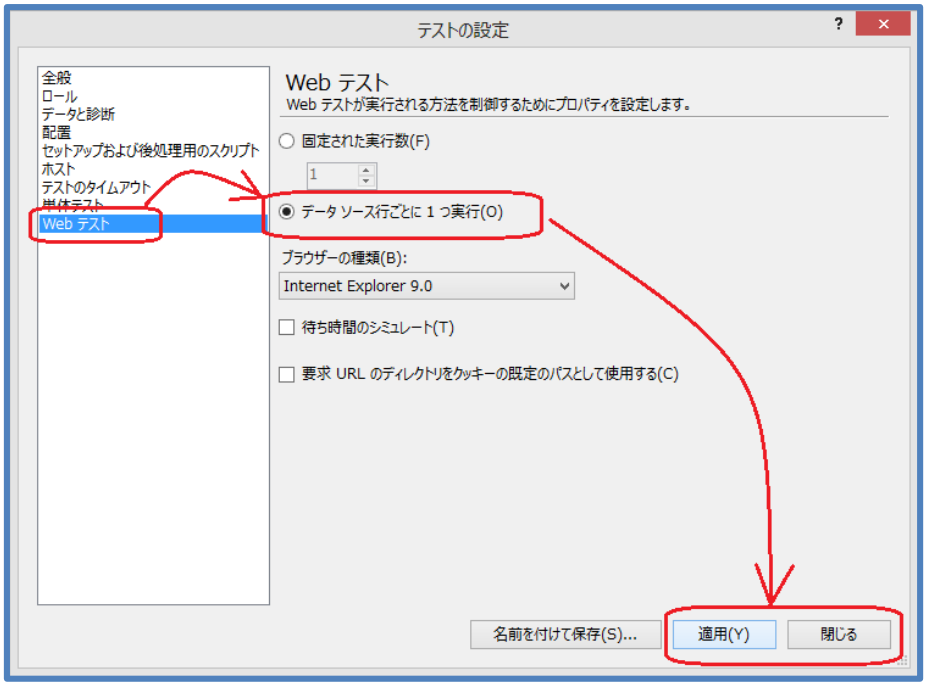


25.　ソリューションのフォルダの直下にある “Local.testsetting” を選択します。

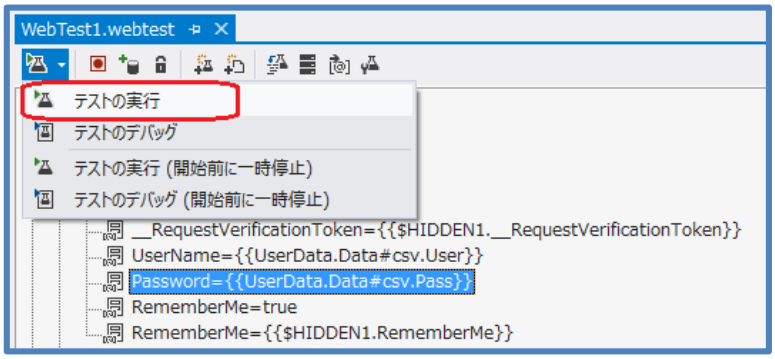
26.　次に、ソリューション エクスプローラー上において、Local.testsetting を右クリックし、表示されたメニューから「開く」を選択します。



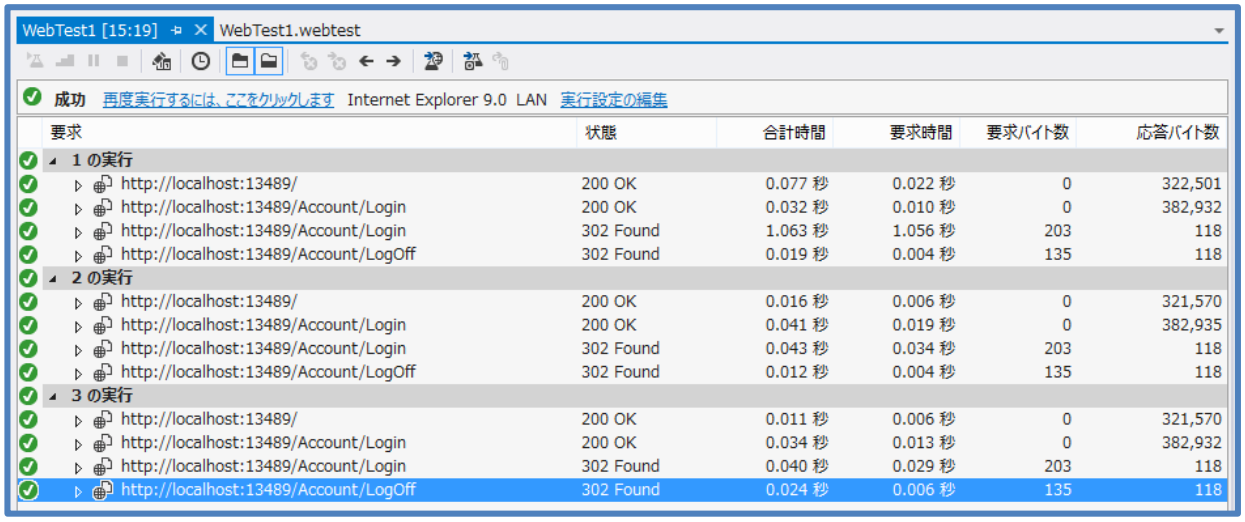
27.　「テストの設定」ウィンドウが表示されるので、「Webテスト」の項目を開き、「データソース行ごとに 1 つ実行」を選択し、「適用」を行ってから、「閉じる」をクリックします。



28.　WebTest の定義の画面から、テストの実行を行います。

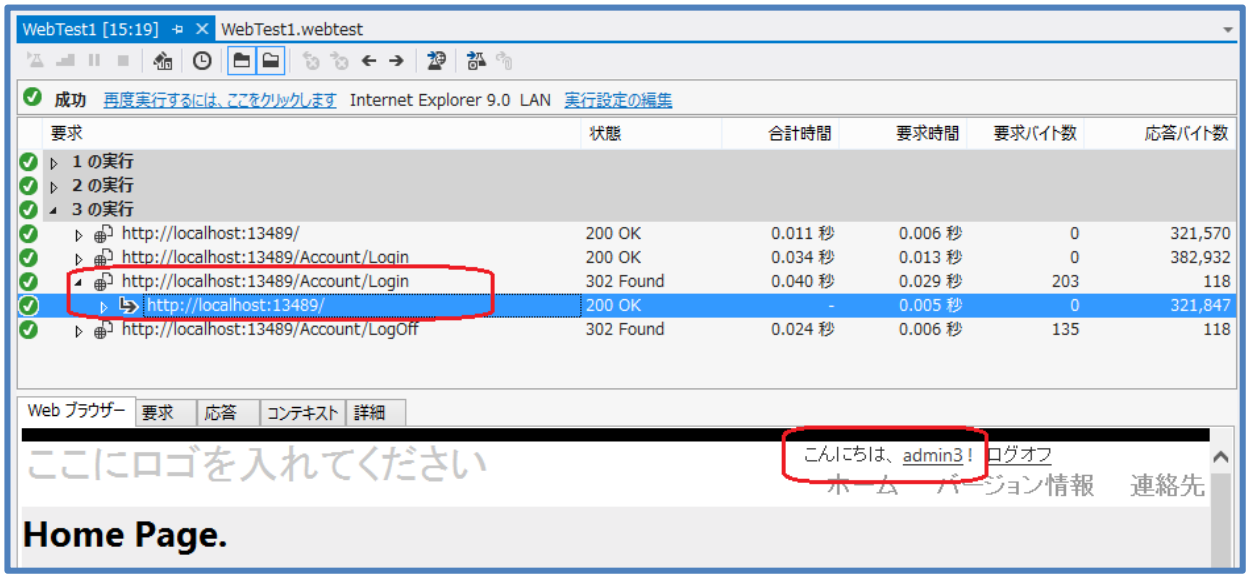


29.　テストが開始され、以下のように３回分（UserData のデータの行数分）テストが実施されます。



「１の実行」「２の実行」「３の実行」それぞれの結果を確認すると、UserData のデータ分の繰り返しが行われたことを確認できます。

例えば、３の実行の２つめの Login リクエストの結果を確認すると、ユーザー名 “admin3” でログインに成功していることを確認することができます。



ここでは以上で２つ目のハンズオンを終了します。

２つ目のハンズオンでは、Visual Studio を使った Web パフォーマンス テストについて学びました。

(1) **Web パフォーマンステストの基礎**

　(ア) Web テストレコーダーによる HTTP Request の記録

　(イ) Web パフォーマンステストの実行

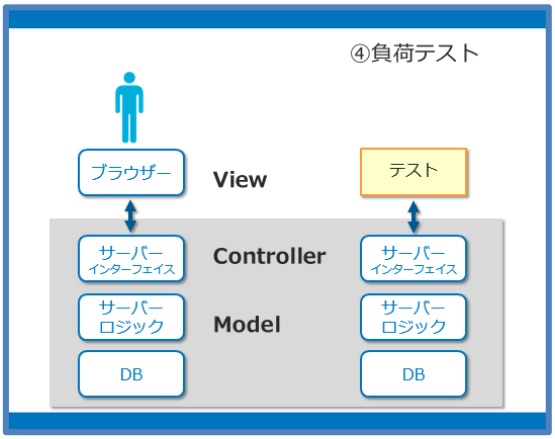
(2) **Web パフォーマンステストのカスタマイズ**

　(ア) 検証規則の追加

　(イ) 外部データを使ったデータ駆動テストの実行

負荷テストの実践

3つ目のハンズオンでは、Visual Studio を使った負荷テスト（ロードテスト）を体験します。



負荷テストはそのテスト目標によって以下の 2 つのパターンに分類されます。

1.（狭義の）負荷テスト

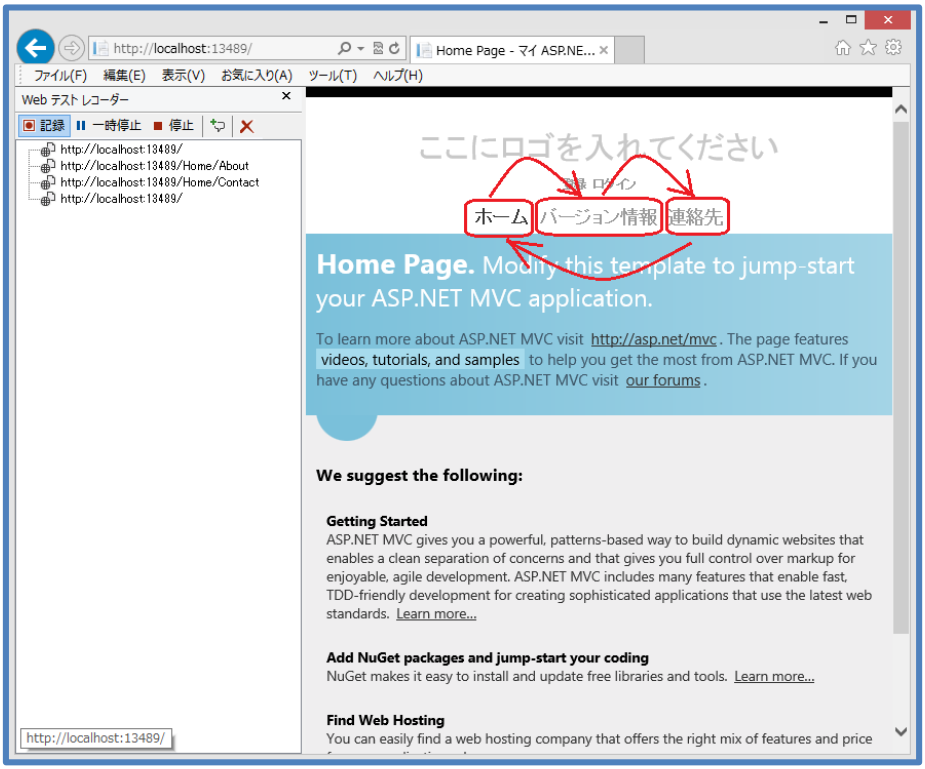
Web アプリケーションの運用中に予測されるワークロードや負荷の量を前提としたときの、テスト対象のシステムやアプリケーションのパフォーマンス特性を判断または検証することに重点を置きます。

2. ストレステスト

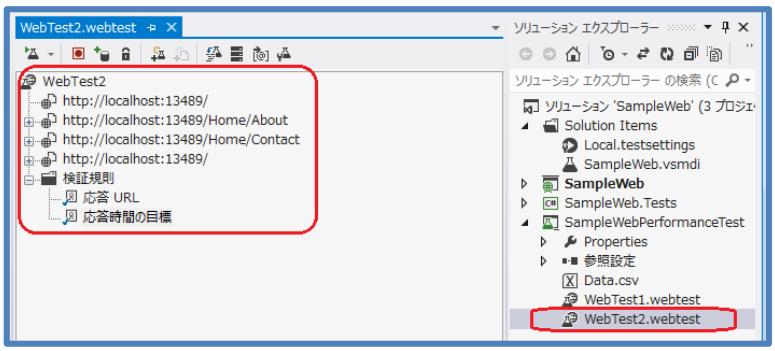
Web アプリケーションの運用中に予測される状態を超えたときの、テスト対象のシステムやアプリケーションのパフォーマンス特性を判断または検証することに重点を置きます。このようなテストでは、アプリケーションがどのような状態で、どのように失敗するか、および障害の発生を警告するためにどのようなインジケーターを監視できるかを判断できるように設計します。

Visual Studio の負荷テスト機能では、このいずれの場合のテストにも利用することができます。

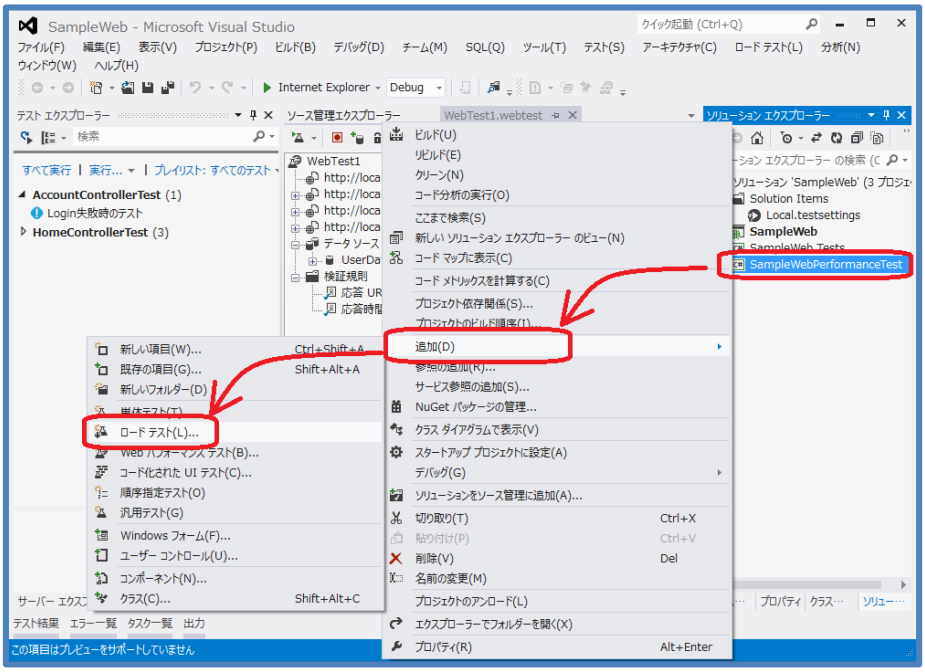
1. 2 つ目のハンズオン（Web パフォーマンステスト）で作成した Web パフォーマンス テストのプロジェクトにおいて、追加でもう一つの Web パフォーマンステストを作成します。ソリューション エクスプローラーにおいて、”SampleWebPerformanceTest” のプロジェクトを右クリックし、表示されたメニューから「追加」→「Web パフォーマンステスト」を選択します。Web テスト レコーダーのアドインが起動した状態でブラウザが立ち上がりましたら、2 つ目のハンズオンでアクセスしたローカルPCの Web サーバーにアクセスします。最初に「ホーム」のページが表示されるので、ナビゲーションから「バージョン情報」を選択し(http://<ホスト名>/Home/About)、次に「連絡先」（http://<ホスト名>/Home/Contact）を選択し、最後に「ホーム」(http://<ホスト名>)をクリックします。これが終わりましたら、Web テスト レコーダーで「停止」をクリックし、記録を終了します。



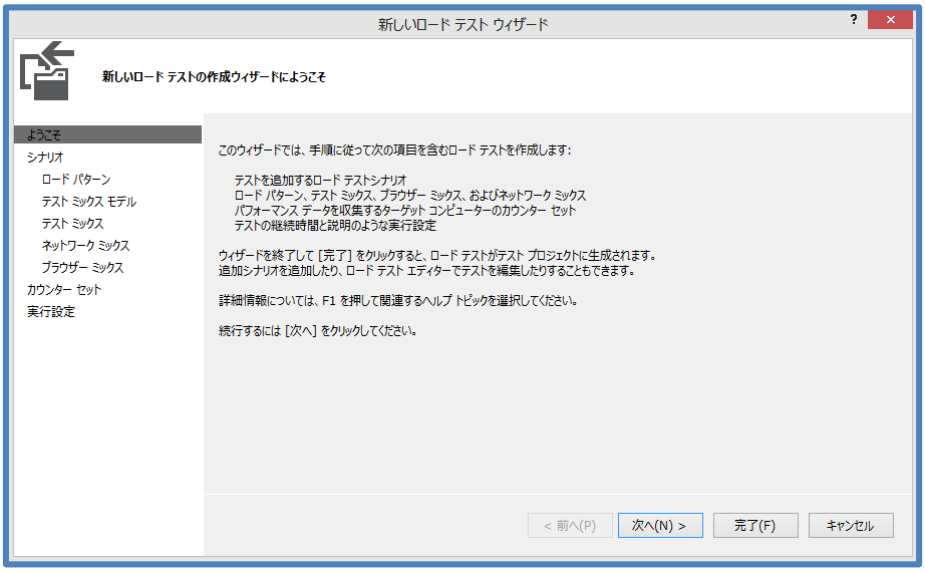
2. ソリューション エクスプローラーにて、ファイル ”WebTest2.webtest” が追加されたことを確認してください。



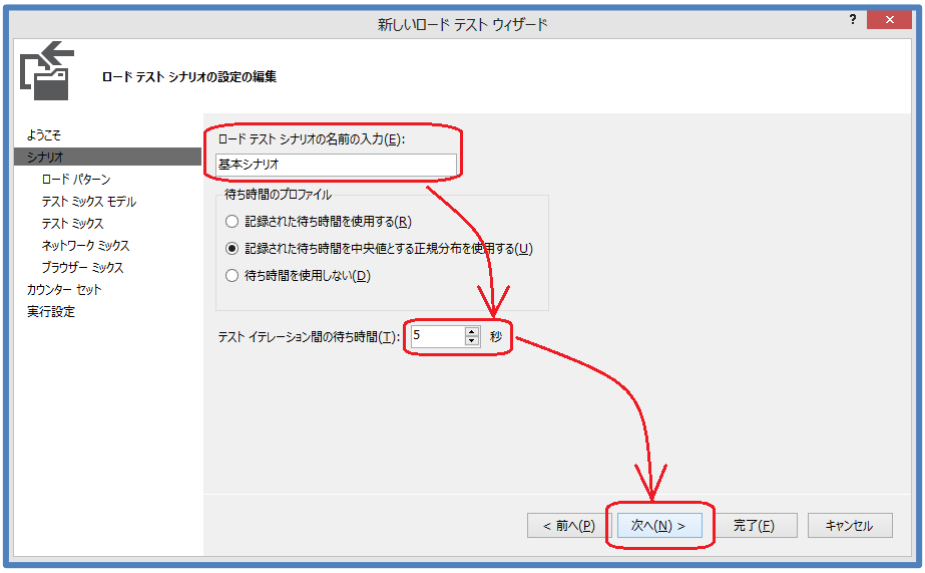
3. 次にロードテストの項目を追加します。ソリューション エクスプローラーにおいて、”SampleWebPerformanceTest” のプロジェクトを右クリックし、表示されたメニューから「追加」→「ロードテスト」を選択します。



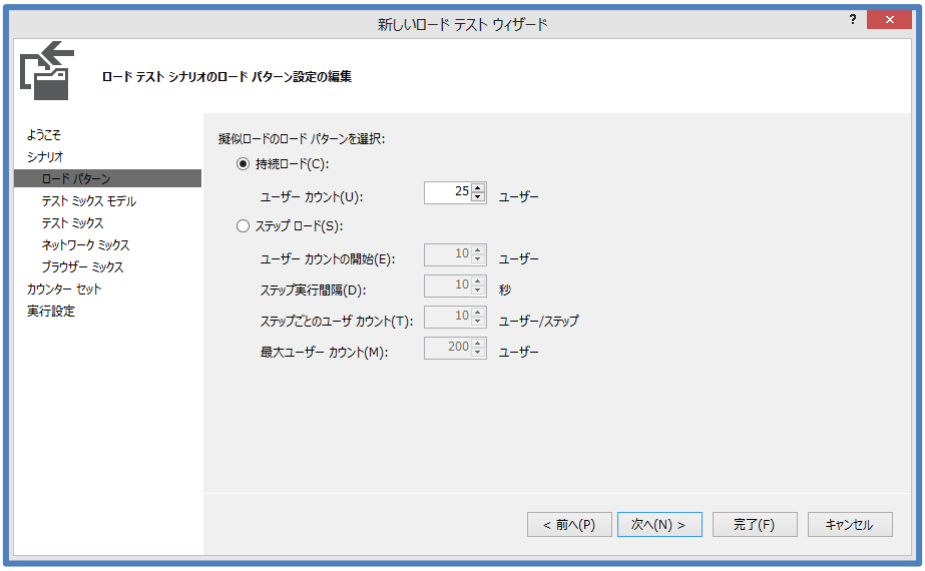
4. ロードテスト追加用のウィザードが表示されますので、「次へ」をクリックします。



5. 最初の設定画面では、ロードテストのシナリオの概要を決定します。シナリオの名前として「基本シナリオ」と入力し、テスト イテレーションの待ち時間を「５」秒に設定しておきます。また、待ち時間のプロファイルに関しては（デフォルトで設定されている）正規分布を選択します。すべて設定が確認できたら「次へ」をクリックします。

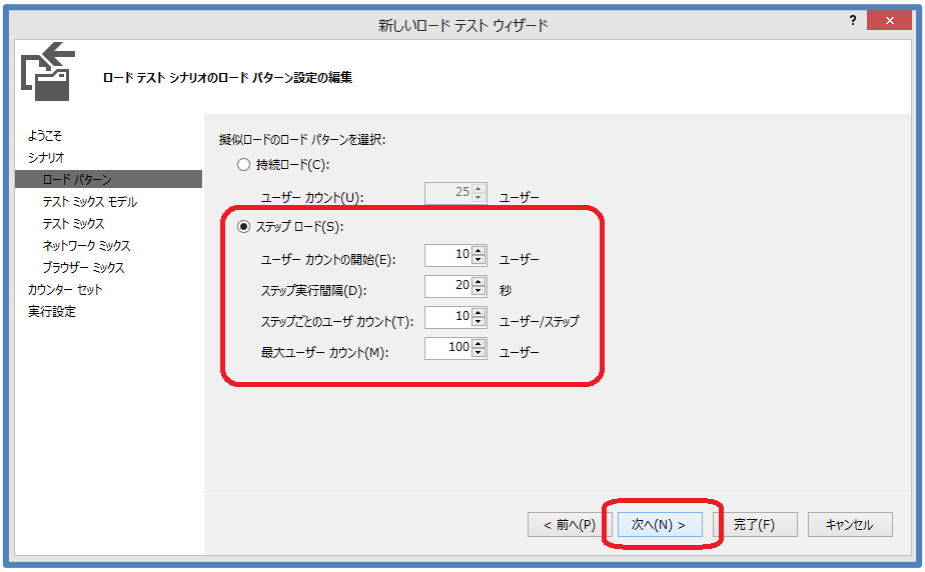


6. 次にロード（負荷）のパターンを決めます。Visual Studio のロードテストでは、用意されたテストシナリオ（今回は２つ目のハンズオンで作成した Web パフォーマンス テストを使用します）をもとに、仮想的なユーザーが一斉にそのシナリオに基づいて Web アクセスを行った場合の振る舞いをテストできます。

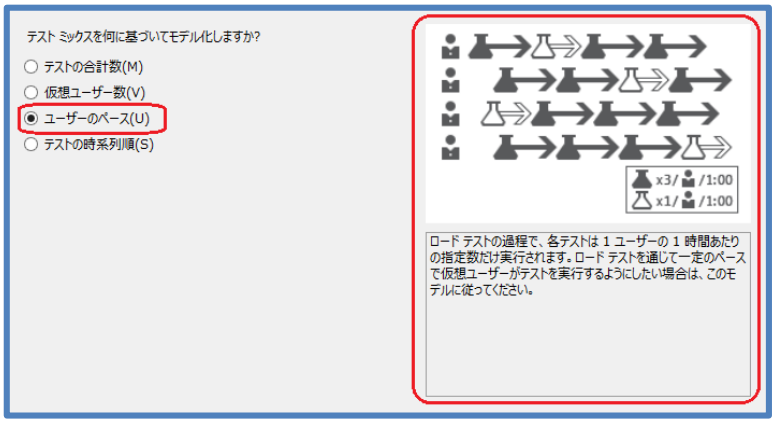


上記のように表示されている画面において、２つの選択肢のうち上の選択肢（「持続ロード」）は固定数の仮想ユーザーによって Web アプリケーションへのアクセスを行う場合に利用します。この３つ目のハンズオンの冒頭において分類した負荷テストのパターンのうち、狭義の負荷テスト、として分類されるテストを行う際などに利用します。
また、下の選択肢（「ステップ ロード」）は一定間隔ごとに仮想ユーザーを増やしながら、アプリケーションの振る舞いをテストする際に利用します。

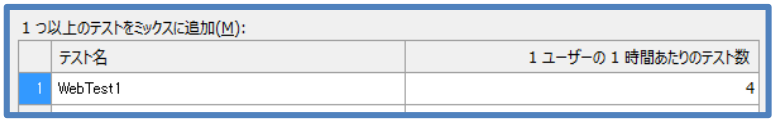
今回はステップロードにて、「10人」の仮想ユーザーで開始し、「20秒毎」に仮想ユーザーを「10人」ずつ増やす、という設定にします。また、最大のユーザー数を「100人」と設定します。設定が確認できたら「次へ」をクリックします。



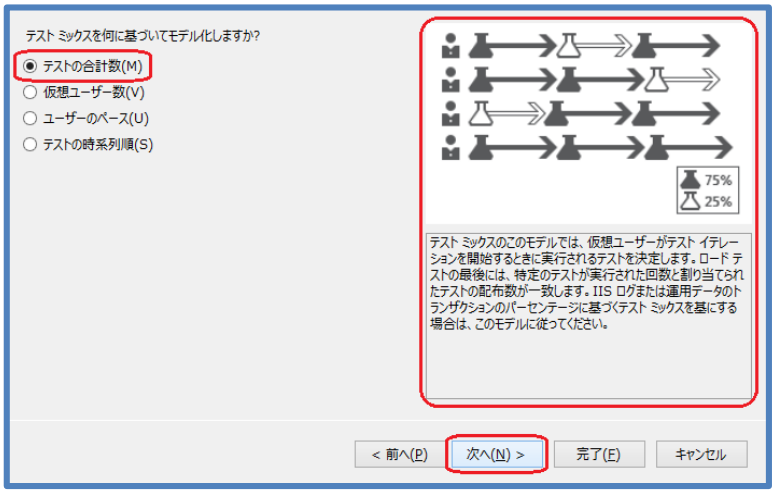
7. 次の画面ではロードテストを実施する際に、どのようにシナリオを実行するかを選択します。例えば、「ユーザーのペース」を選択すると、各仮想ユーザーが一時間当たりに実行するテストの数を設定できます。



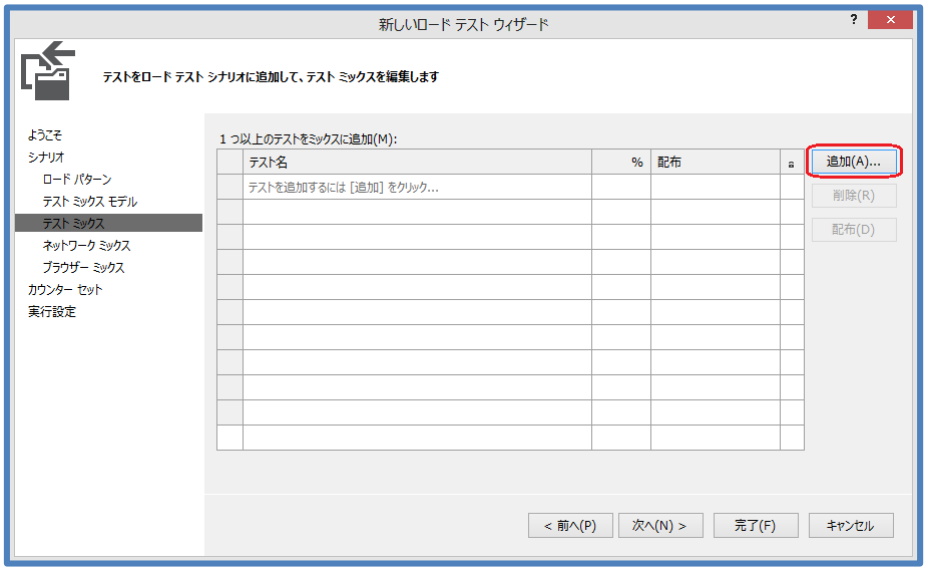
“社員数2000名の企業において運用する社内の情報共有のサーバーにおいて、15分に１回程度の割合でユーザーがアクセスを行う。” といったシナリオの場合に、手順6 で設定しているロードパターンの最大ユーザー数を2000ユーザーとしたうえで、この「ユーザーのペース」を選択し、次の画面にて利用するテストシナリオとその１時間あたりの実施回数（この場合は１時間で４回の実施）を設定することで、このようなシナリオに基づく負荷テストが可能になります。



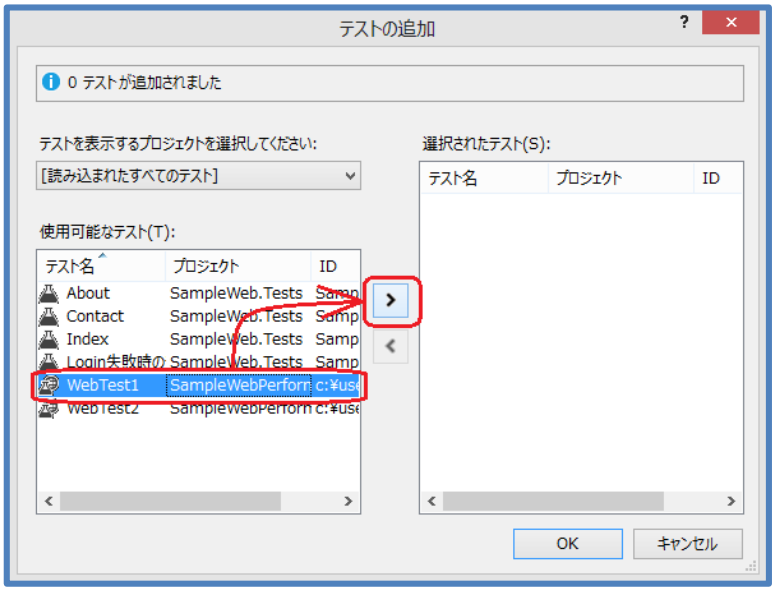
今回はよりランダムな状況で Web アプリケーションの性能上限を確認する、ということを目的に負荷テストを行う、と仮定し「テストの合計数」を選択し、「次へ」をクリックしてください。



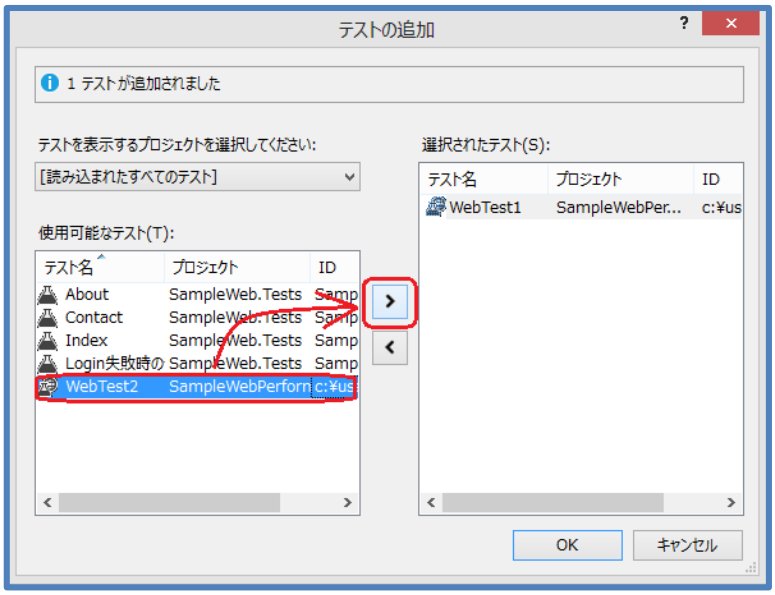
8.　　次に、テストのシナリオ選択の画面が表示されますので、「追加」を押してください。



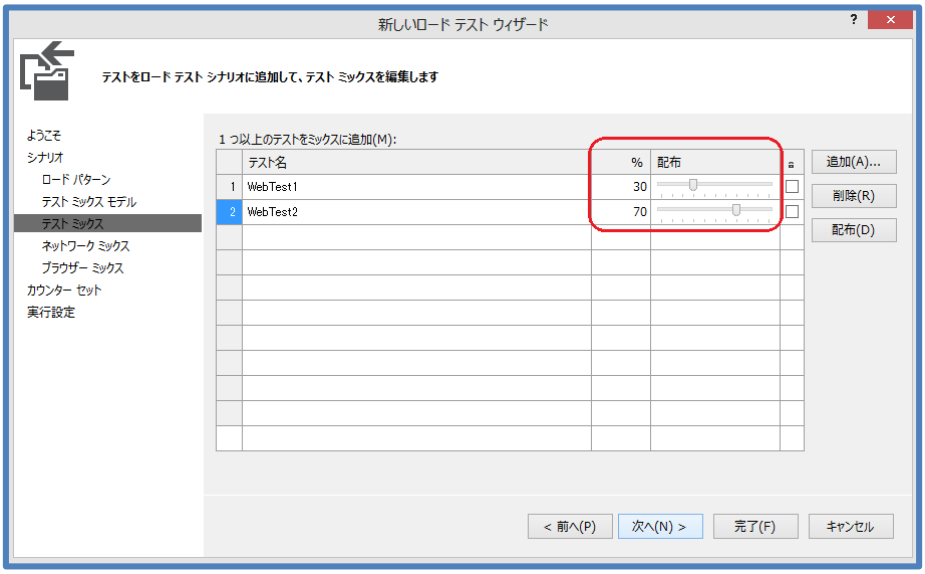
9.　　新しく立ち上がった「テストの追加」画面で、”WebTest1” を選択し、真ん中の矢印（＞）をクリックしてください。



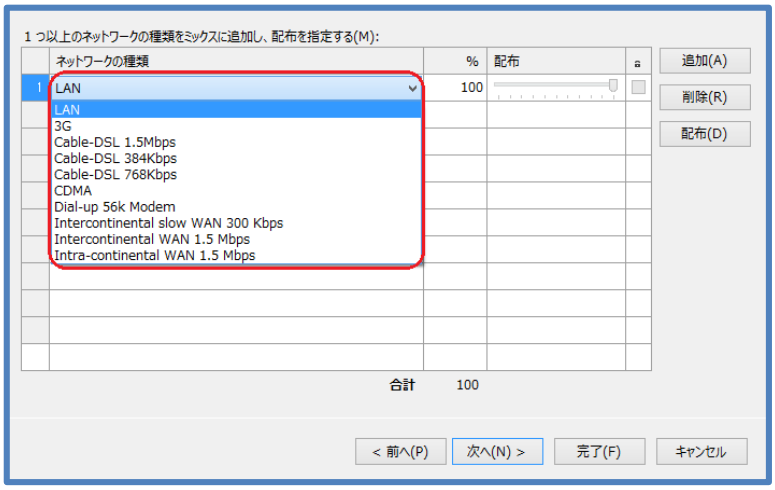
10.　　引き続き ”WebTest2” を選択し、真ん中の矢印（＞）をクリックします。右側の「選択されたテスト」に “WebTest1” と “WebTest2” が表示されているのを確認したら「OK」をクリックして「テストの追加」画面を閉じてください。



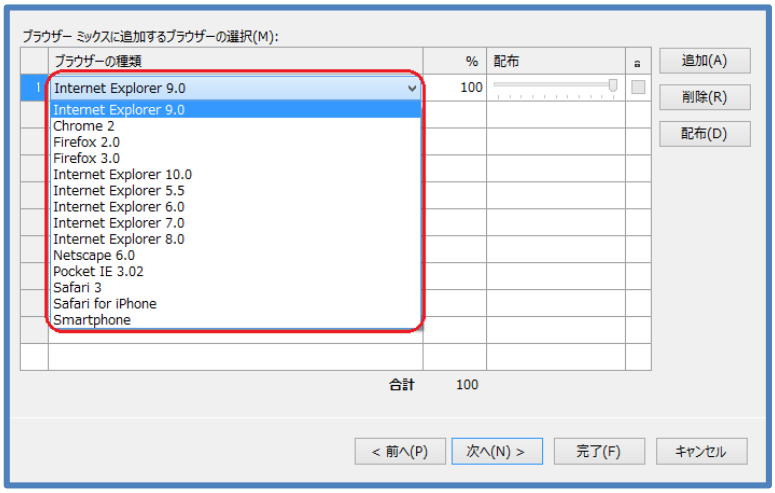
11.　　次に選択した “WebTest1” と “WebTest2” のシナリオを、どの程度の割合で実施するかを入力します。今回 “WebTest1” は「ログインを行いその後ログアウトを行う」シナリオで、”WebTest2” は「ページを順に閲覧する」シナリオとなっていますので、「ログインを行うユーザーのほうが少ない Web アプリケーション」と仮定し、”WebTest1” の割合を 30%、”WebTest2” の割合を 70% とします。今回は手順7 においてテスト ミックスを「テストの合計数」でモデル化しているので、それぞれの仮想ユーザーは 30% の確率で “WebTest1” を実施し、70% の確率で “WebTest2” を実行します。また、その実行が終了すると、同じ確率でシナリオの選択を行いテストを実行する、といった動作となります。



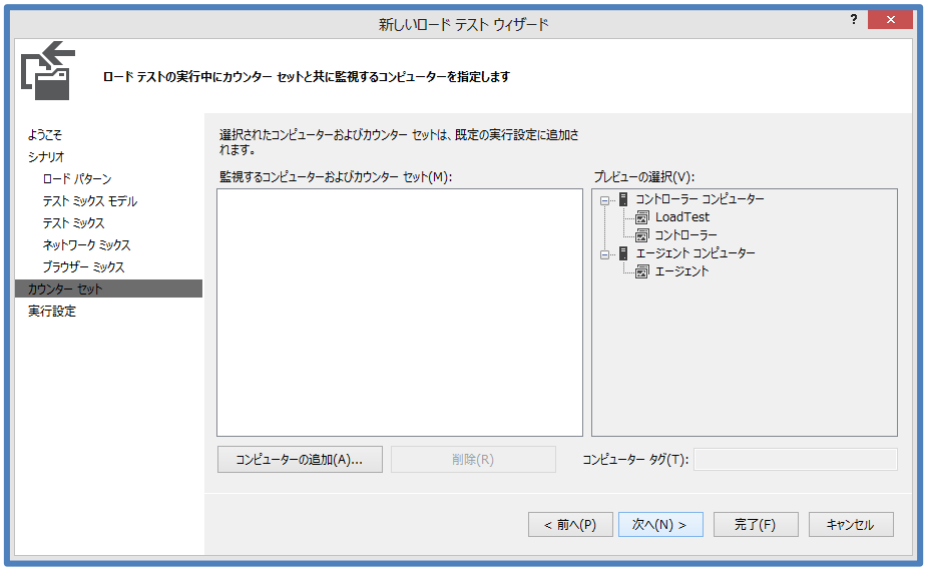
12.　次に実際のユーザーが使用するであろうネットワーク環境を元に、仮想ユーザーとサーバーとの間のネットワーク帯域の選択を行います。今回はデフォルトのまま（100%のユーザーが LAN でアクセスする）で結構ですので「次へ」をクリックしてください。



13.　次にユーザーが使用するブラウザ環境の選択を行います。こちらも今回はデフォルトのまま（100%のユーザーが IE 9 でアクセスする）で結構ですので「次へ」をクリックしてください。



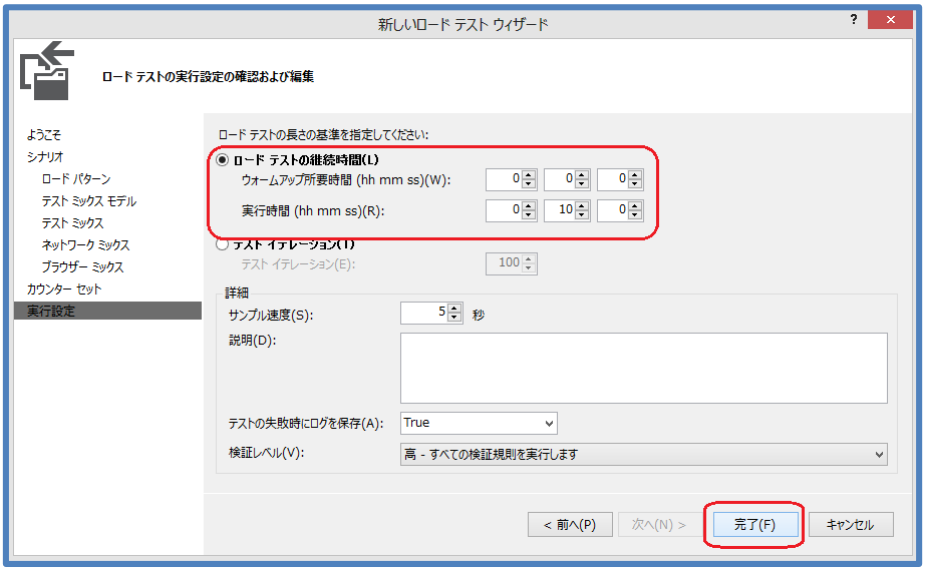
14.　　次にロードテスト実行中に監視を行うコンピューターの指定を行います。デフォルトで、「コントローラーコンピューター」および「エージェントコンピューター」の項目があります。これらは実際に負荷テストを行う際に、仮想ユーザーを発生させるためのコンピューターとなります。通常はこれらのテストを行う側のサーバーとは別に、テスト対象となるサーバー（Web アプリケーションを稼働させるサーバー）が存在することになりますので、そのサーバーの情報を追加します。今回はハンズオンラボの実施のため、テスト対象となるサーバー（Web アプリケーションサーバー）とテストを行う側のサーバー（仮想ユーザーにより負荷を発生させるサーバー）を同一 PC で行っているため、監視対象の追加を行わず、「次へ」をクリックします。



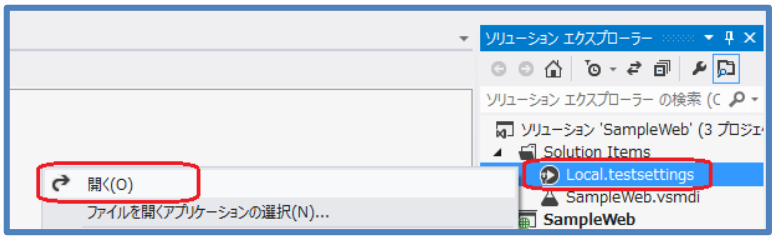
※　開発プロジェクトにおいて、実際の運用環境を想定してのテストを行う場合などは、テスト対象となるサーバーと、テストを行う側のサーバーは必ず分離するようにしてください。負荷テスト実施時にはテストを行う側のサーバー（仮想ユーザーとして、Web アプリケーションに HTTP Request を送る側のサーバー）においてもテスト実行のためのリソースが必要となりますので、同一のサーバーで実施してしまうとパフォーマンス上のボトルネックが現出された際に、開発を行っている Web アプリケーションの問題なのか、テストを行う側の問題なのかの切り分けが難しくなります。テスト対象とテストを行う側のサーバーを分離することで、ボトルネック発生時の問題切り分けが容易になり、精度の高いテスト実施が可能になります。

15.　　最後にロードテストの実行設定として、どのくらいの時間の長さで負荷テストを行うか（あるいは何回行

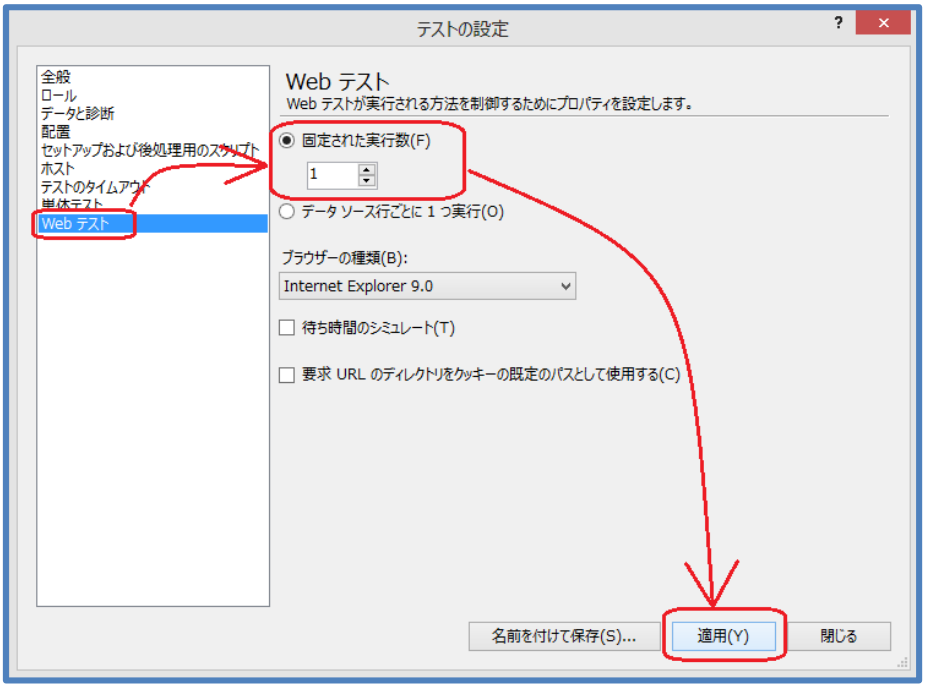
うか）を指定します。今回はデフォルトのまま（１０分間持続する）テストを実行する、として「完了」を押します。



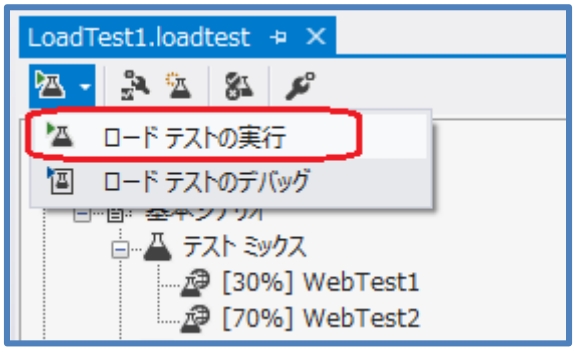
16.　以上でロードテストの作成は終了です。次にロードテストの実行を行う前に、２つ目のハンズオンで行った Web パフォーマンステストに関する設定を元に戻したいとおもいます。ソリューション エクスプローラーにおいて Local.testsetting ファイルを右クリックし、表示されたメニューから「開く」を選択します。



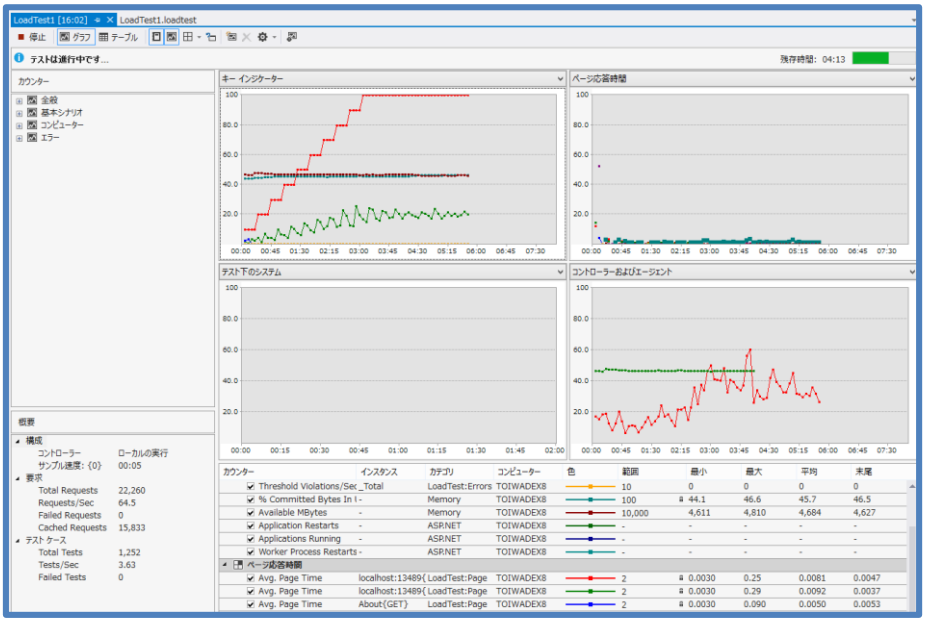
17.　表示されたメニューにおいて「Webテスト」のタブを開き、実行のプロパティを「固定された実行数」に変更し、「適用」を行った後に「閉じる」をクリックします。



18.　　手順15 までの作業で作成されたロードテストの定義 LoadTest1.loadtest において、三角フラスコのメニューを展開し、「ロードテストの実行」を行います。

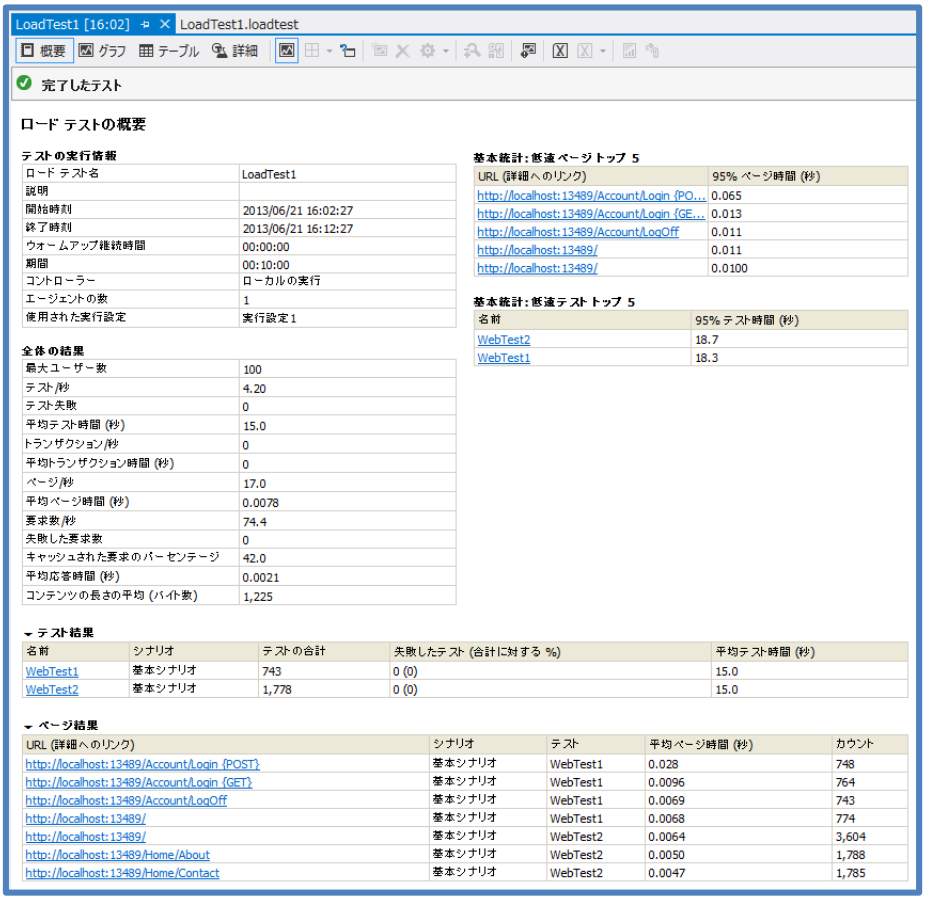


19.　　以下のようにロードテストによる負荷状況が表示されます。今回は「ステップロード」を選択しているので、”User Load”（４つあるグラフのうち左上のグラフにおいて右上がりになっている赤色のグラフ）が徐々に高まっているのがわかります。一方でページの応答時間のデータ（右上のグラフ）はほぼゼロに近いエリアを横ばい状態でキープしていることが確認できます。

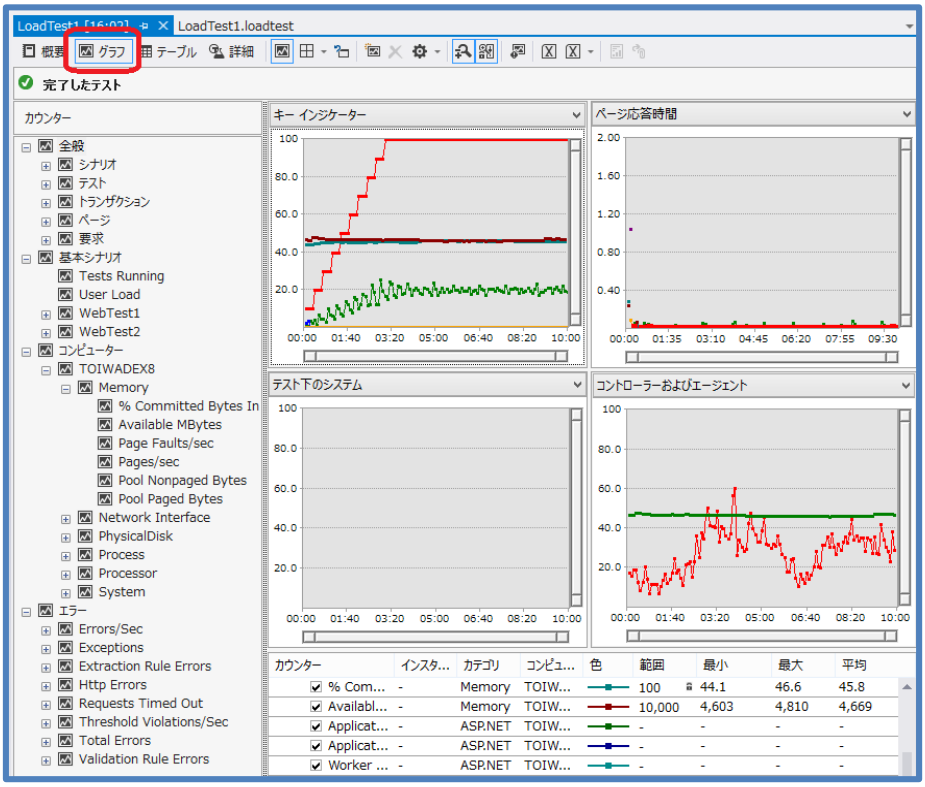


※　テストによる実行状況、結果は、テストを実施いただく環境により異なります。

20.　　テストが終了すると、各ページ毎の平均応答時間やテスト実施においてアクセスされた回数等、以下のようなテスト結果の概要が表示されます。



21.　　また上部のタブにおいて、「グラフ」を選択することで、カウンターごとの詳細な情報を確認することが可能です。



負荷テスト実施時にエラーが起こった際には、そのエラー情報やこれらのカウンター情報により、ボトルネックの発見に役立てることが可能です。

このように Visual Studio の負荷テスト機能を利用すると、多数のユーザーによりアクセスが行われた際にもアプリケーションが正しく動くのか、期待通りの応答時間で返答を行うことができるのか、何名程度のユーザーアクセスに耐えることができるのか、といったことを検証することが可能です。

ここでは以上で３つ目のハンズオンを終了します。
３つ目のハンズオンでは、Visual Studio を使った負荷テストについて学びました。

(1) **負荷テストの作成**
    (ア) 負荷テストの分類（狭義の負荷テストとストレステスト）
    (イ) ウィザードによるロードテストの作成
(2) **ロードテストの実行と結果の確認**