

Getting Started with ASP.NET 4.5 Web Forms (Beta)

Erik Reitan

Step-by-Step



Ketabton.com

Microsoft®

Getting Started with ASP.NET 4.5 Web Forms (Beta)

Erik Reitan

Summary: This series of tutorials guides you through the steps required to create an ASP.NET Web Forms application using Visual Studio 11 Beta and ASP.NET 4.5 Beta.

Category: Step-By-Step

Applies to: ASP.NET 4.5 Beta, Visual Studio 11 Beta

Source: ASP.NET site ([link to source content](#))

E-book publication date: May 2012

59 pages

Copyright © 2012 by Microsoft Corporation

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

Microsoft and the trademarks listed at

<http://www.microsoft.com/about/legal/en/us/IntellectualProperty/Trademarks/EN-US.aspx> are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

The example companies, organizations, products, domain names, email addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

This book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, Microsoft Corporation, nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

Getting Started with ASP.NET 4.5 Web Forms

By Erik Reitan | April 6th, 2012

Contents

Introduction and Overview	3
Introduction.....	3
Audience	3
Application Features	3
Application Scenarios and Tasks	3
Overview.....	3
Prerequisites.....	7
Download the Sample Application	8
Tutorial Support and Comments	8
After this Tutorial Series.....	9
Create the Project	10
Creating the Project.....	10
Reviewing the Project	12
Running the Default Web Application	13
ASP.NET Web Forms Background.....	13
Web Application Features in the Web Forms Application Template	14
Touring Visual Studio	17
Summary	18
Additional Resources.....	18
Create the Data Access Layer	19
Creating the Data Models	19
Configuring the Application to Use the Data Model	30
Building the Application.....	31
Summary	32
Additional Resources.....	32
UI and Navigation	33
Modifying the UI.....	33

Updating the Master Page.....	36
Adding Image Files	38
Adding Pages	40
Updating the StyleSheet.....	41
Modifying the Default Navigation	42
Adding a Data Control to Display Navigation Data.....	42
Linking the Data Control to the Database	44
Running the Application and Creating the Database	44
Reviewing the Database	45
Summary	48
Additional Resources.....	48
Display Data Items and Details	49
Adding a Data Control to Display Products	49
Displaying Products	49
Adding Code to Display Products	52
Running the Application.....	53
Adding a Data Control to Display Product Details	55
Running the Application.....	57
Summary	58
Conclusion.....	58
Acknowledgements.....	58

Introduction and Overview

Introduction

This series of tutorials guides you through the steps required to create an ASP.NET Web Forms application using Visual Studio 11 Beta and ASP.NET 4.5 Beta.

The application you'll create is named the Wingtip Toys. It's a simplified example of a store front web site that sells items online. This tutorial series highlights several of the new features available in ASP.NET 4.5 Beta.

This tutorial series is the first installment of two. Comments are welcome, and we'll make every effort to update this tutorial series based on your suggestions.

Audience

The intended audience of this tutorial series is experienced developers who are new to ASP.NET Web Forms. A developer interested in this tutorial series should have the following skills:

- Familiar with an object oriented programming language
- Familiar with Web development concepts (HTML, CSS, JavaScript)
- Familiar with relational database concepts
- Familiar with n-tier architecture concepts

Application Features

The ASP.NET Web Form features presented in this series include:

- The Web Application Project (not Web Site Project)
- Web Forms
- Master Pages, Configuration
- Entity Framework Code First, LocalDB
- Request Validation
- Strongly Typed Data Controls, Model Binding, Data Annotations, and Value Providers

Application Scenarios and Tasks

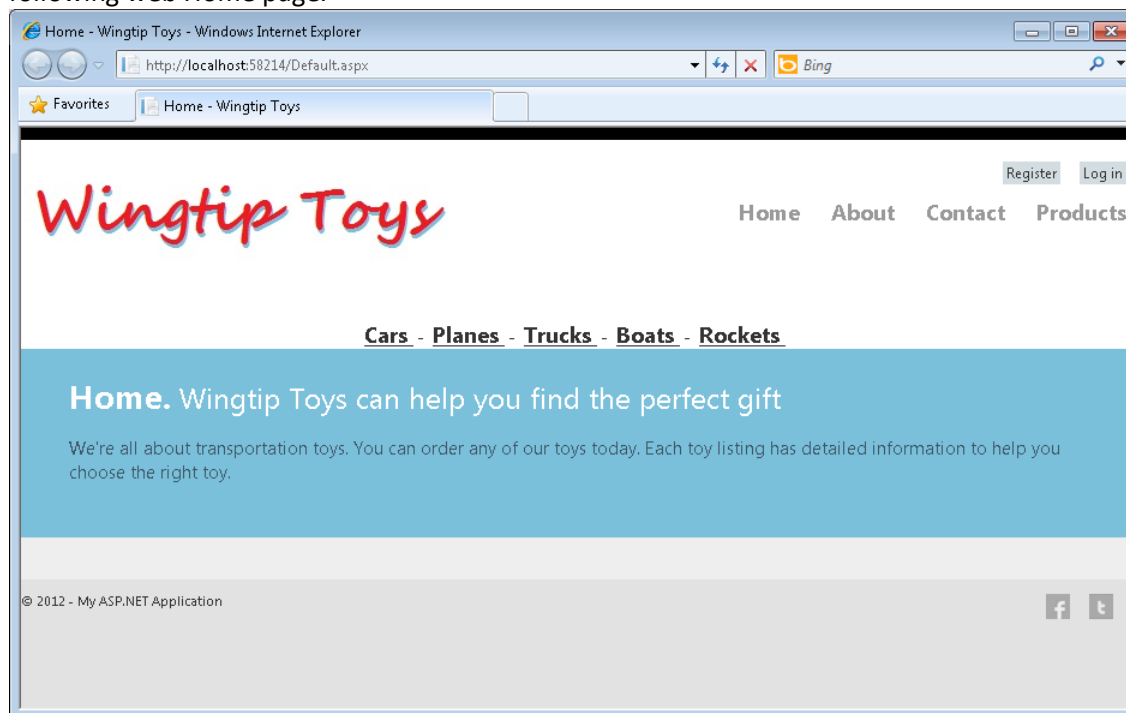
Tasks demonstrated in this first series include:

- Creating, reviewing and running the new project
- Creating the database structure
- Initializing and seeding the database
- Customizing the UI using styles, graphics and a master page
- Adding pages and navigation
- Displaying menu details and product data

Overview

If you are new to ASP.NET Web Forms but have familiarity with programming concepts, you have the right tutorial. If you are already familiar with ASP.NET Web Forms, you can benefit from this tutorial series by the new features available in ASP.NET 4.5 Beta. If you are unfamiliar with programming concepts and ASP.NET Web Forms, see [Getting Started](#) on the ASP.NET Web site.

The following screen shots provide a quick view of the ASP.NET Web forms application that you will create in this tutorial series. When you run the application from Visual Studio 11 Beta, you will see the following web Home page.



You can register as a new user, or log in as an existing user. Navigation is provided at the top for each product category. Each time the Home page is reached, one of the available products from the database will be displayed.

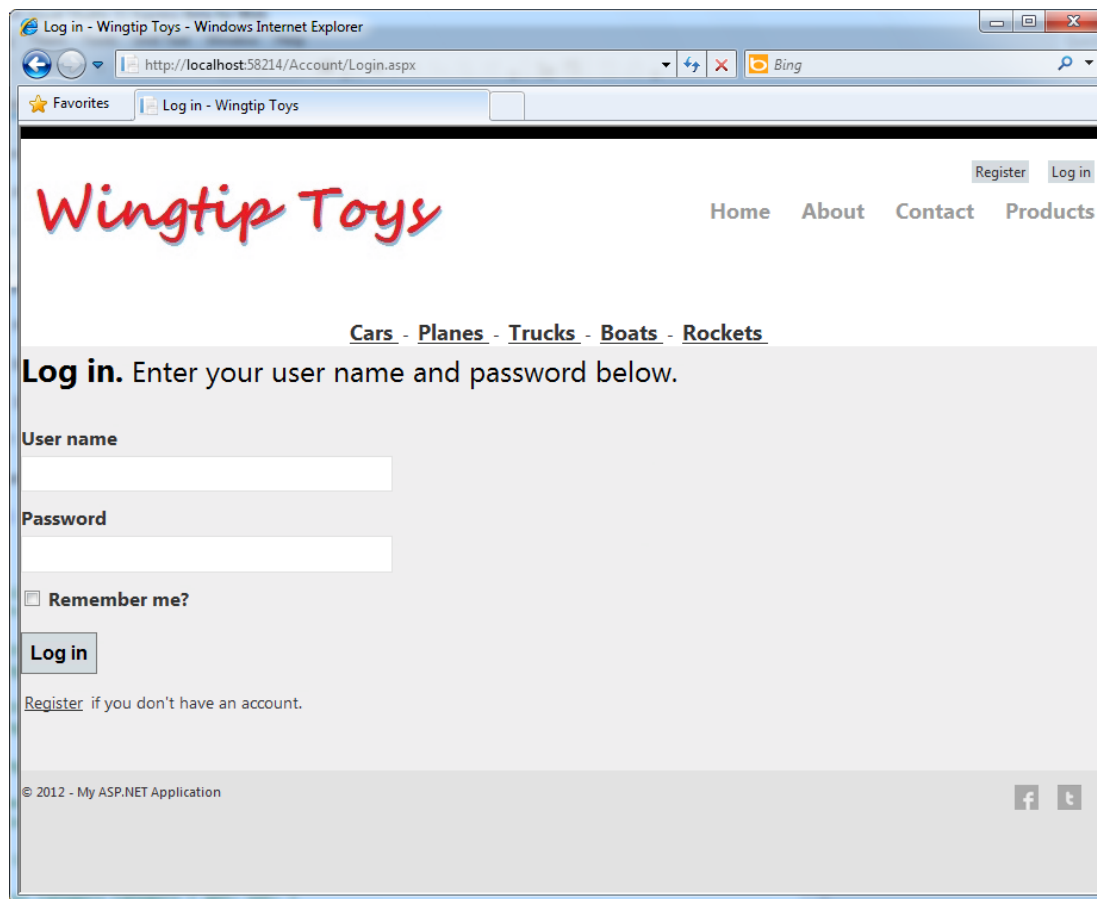
By selecting the **Products** link, you will be able to see a list of all available products.



You can also see individual product details by selecting any of the listed products.



As a user, you can register and login using the default functionality of the Web Forms template.



Prerequisites

Before you start, make sure that you have the following software installed on your computer:

- [Microsoft Visual Studio 11 Beta](#) or [Microsoft Visual Studio 11 Express Beta for Web](#). The .NET Framework is installed automatically.

This tutorial series uses Microsoft Visual Studio 11 Express Beta for Web. You can use either Microsoft Visual Studio 11 Express Beta for Web or Microsoft Visual Studio 11 Beta to complete this tutorial series.

Note

Microsoft Visual Studio 11 Beta and Microsoft Visual Studio 11 Express Beta for Web will often be referred to as Visual Studio throughout this tutorial series.

If you already have a Visual Studio version installed, the installation process will install Visual Studio 11 Beta or Microsoft Visual Studio 11 Express Beta for Web next to the existing version. Sites that you

created in earlier versions can be opened in the Beta version of Visual Studio 11 and continue to open in previous versions.

Note

This walkthrough assumes that you selected the **Web Development** collection of settings the first time that you started Visual Studio. For more information, see [How to: Select Web Development Environment Settings](#).

Download the Sample Application

After installing the prerequisites, you are ready to begin creating the new Web project that is presented in this tutorial series. If you would like to run the sample application that this tutorial series creates, you can download it from the MSDN Samples site. This download contains the following:

- The sample application in the **WingtipToys** folder.
- The resources used when creating the sample application in the **WingtipToys-Assets** folder of the **WingtipToys** folder.
- A PDF file containing this tutorial series in the **WingtipToys** folder.

Download the file from MSDN Samples site:
[Getting Started with ASP.NET Web Forms 4.5](#)

The download is a .zip file. To see the completed project that this tutorial series creates, find and select the **C#** folder in the .zip file. Save the **C#** folder to the folder you use to work with Visual Studio 11 Beta projects. By default this is the following folder:

C:\Users\<username>\Documents\Visual Studio 11\Projects

Rename the **C#** folder to **WingtipToys**.

Note

If you already have a folder named **WingtipToys** in your **Projects** folder, temporarily rename that existing folder before renaming the **C#** folder to **WingtipToys**.

To run the completed project, open the **WingtipToys** folder that you copied to your **Projects** folder and double-click the **WingtipToys.sln** file. Visual Studio 11 Beta will open the project. Next, right-click the **Default.aspx** file in the **Solution Explorer** window and click **View In Browser** from the right-click menu.

Tutorial Support and Comments

Use the Q AND A section included with the [Getting Started with ASP.NET Web Forms 4.5](#) sample for any questions or comments.

Comments on this tutorial series are welcome, and when this tutorial series is updated every effort will be made to take into account corrections or suggestions for improvements that are provided in the tutorial comments.

When an error happens during development, or if the Web site does not run correctly, the error messages may give complex clues to the source of the problem or might not explain how to fix it. To help you with some common problem scenarios, you can also use the [ASP.NET forums](#) or the Q AND A section included with the [Getting Started with ASP.NET Web Forms 4.5](#) sample. If you get an error message or something doesn't work as you go through the tutorials, be sure to check the above locations.

After this Tutorial Series

As previously mentioned, this tutorial series is the first set of two. Tasks that will be presented in the second series include:

- Business logic and shopping cart functionality
- Membership, authorization and checkout functionality
- Exception handling
- Deployment considerations

Create the Project

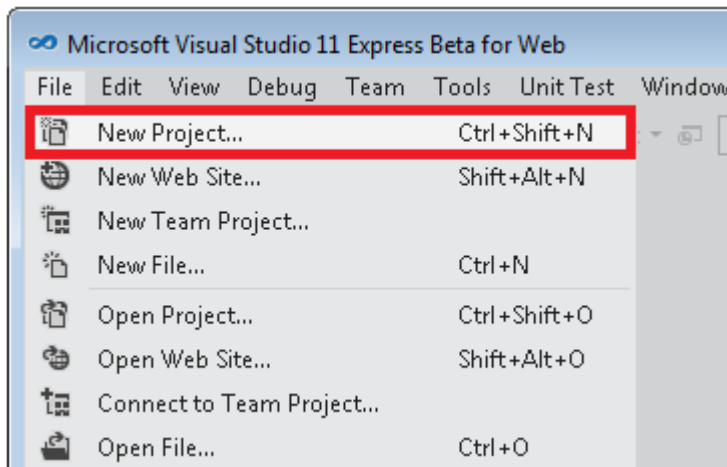
In this tutorial you will create, review, and run the default project in Visual Studio, which will allow you to become familiar with features of ASP.NET. Also, you will review the Visual Studio environment.

What you'll learn:

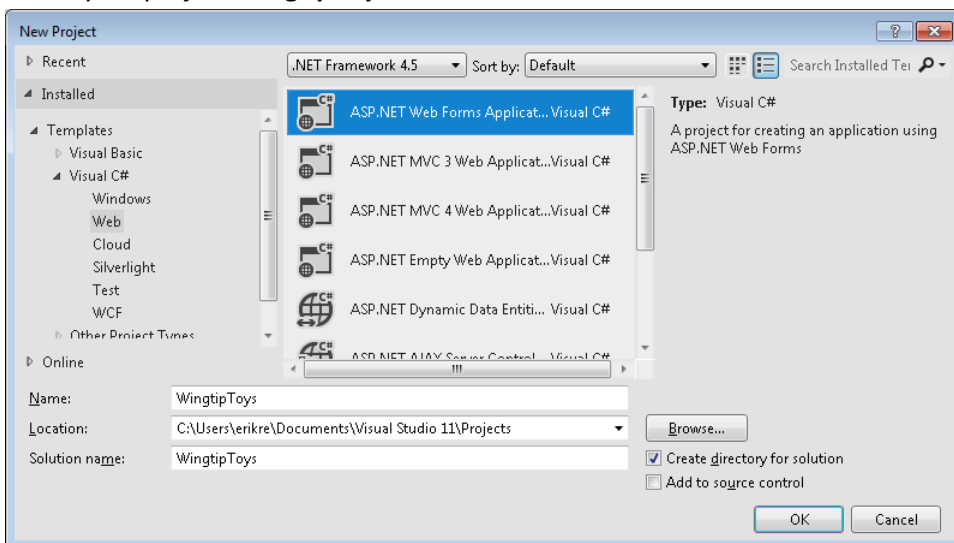
- How to create a new Web Forms project.
- The file structure of the Web Forms project.
- How to run the project in Visual Studio.
- The different features of the default Web forms application.
- Some basics about how to use the Visual Studio environment.

Creating the Project

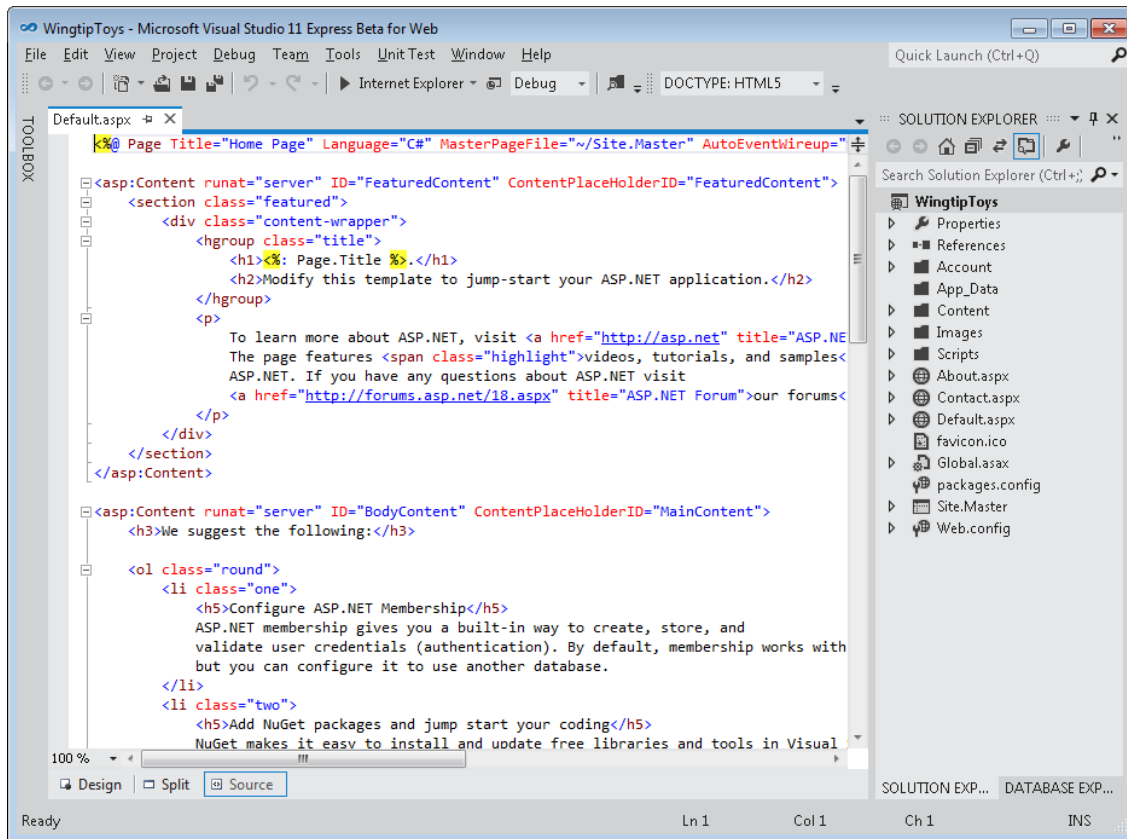
1. Open Visual Studio.
2. Select **New Project** from the **File** menu in Visual Studio.



3. Select the **Templates>Visual C#>Web** templates group on the left.
4. Choose the **ASP.NET Web Forms Application** template in the center column.
5. Name your project **WingtipToys** and choose the **OK** button.



The project will take a little time to create. When it's ready, it shows the **Default.aspx** page.



You can switch between **Design** view and **Source** view by selecting an option at the bottom of the center window. Design view displays ASP.NET Web pages, master pages, content pages, HTML pages, and user controls using a near-WYSIWYG view. Source view displays the HTML markup for your Web page, which you can edit.

Understanding the Project Type

ASP.NET Web Forms lets you build dynamic websites using a familiar drag-and-drop, event-driven model. A design surface and hundreds of controls and components let you rapidly build sophisticated, powerful UI-driven sites with data access. The Wingtip Toys tutorial series is based on ASP.NET Web Forms, but many of the concepts you learn in this tutorial series are applicable to all of ASP.NET.

ASP.NET offers three development frameworks:

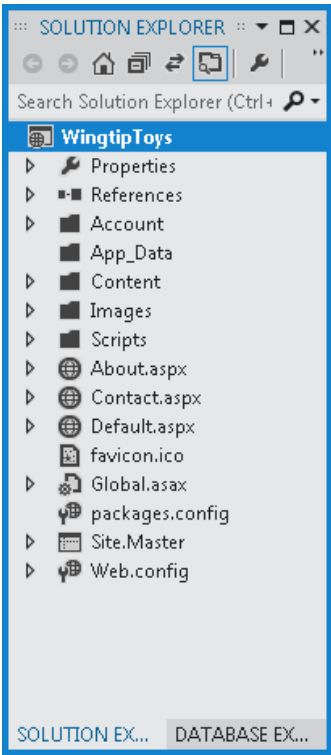
- [ASP.NET Web Forms](#)
The Web Forms framework targets developers who prefer declarative and control-based programming, such as Microsoft Windows Forms (WinForms) and WPF/XAML/Silverlight. It offers a WYSIWYG designer-driven development model, so it's popular with developers looking for a rapid application development (RAD) environment for web development. If you're new to web programming and are familiar with the traditional Microsoft RAD client development tools (for example, for Visual Basic and Visual C#), you can quickly build a web application without having experience in HTML and JavaScript.
- [ASP.NET MVC](#)

ASP.NET MVC targets developers who are interested in patterns and principles like test-driven development, separation of concerns, inversion of control (IoC), and dependency injection (DI). This framework encourages separating the business logic layer of a web application from its presentation layer.

- [ASP.NET Web Pages](#)
ASP.NET Web Pages targets developers who want a simple web development story, along the lines of PHP. In the Web Pages model, you create HTML pages and then add server-based code to the page in order to dynamically control how that markup is rendered. Web Pages is specifically designed to be a lightweight framework, and it's the easiest entry point into ASP.NET for people who know HTML but might not have broad programming experience — for example, students or hobbyists. It's also a good way for web developers who know PHP or similar frameworks to start using ASP.NET.

Reviewing the Project

In Visual Studio, the **Solution Explorer** window lets you manage files for the project. Let's take a look at the folders that have been added to your application in **Solution Explorer**. The web application template adds a basic folder structure:



Visual Studio creates some initial folders and files for your project. The first files that you will be working with later in this tutorial are the following:

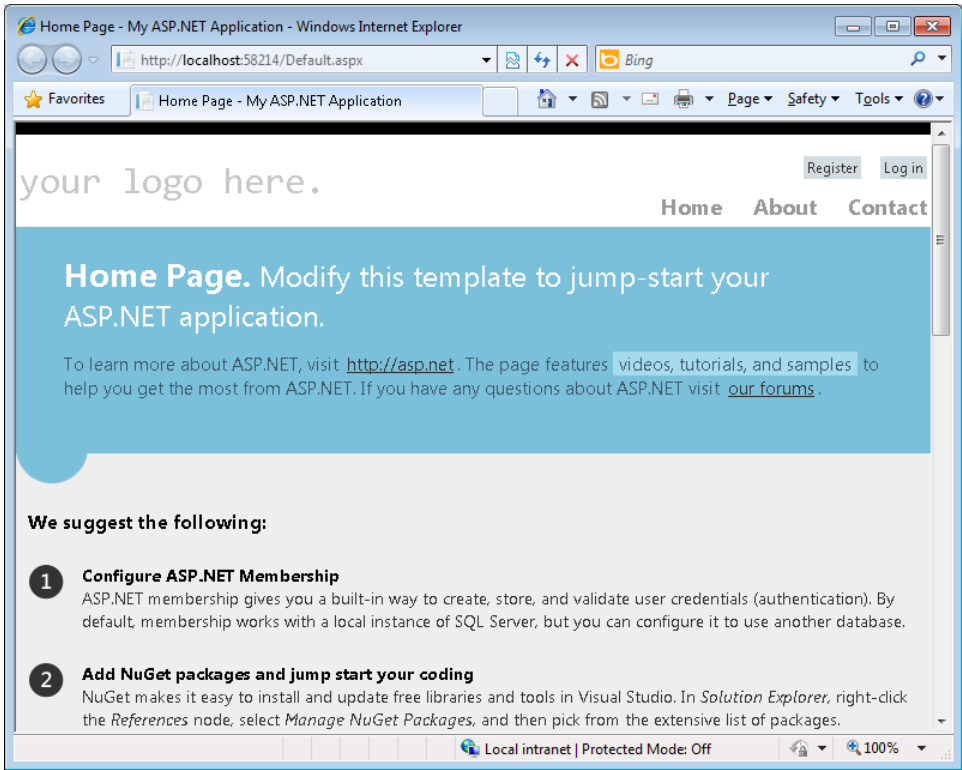
File	Purpose
Default.aspx	Typically the first page displayed when the application is run in a browser.
Site.Master	A page that allows you to create a consistent layout and use standard

	behavior for pages in your application.
Global.asax	An optional file that contains code for responding to application-level and session-level events raised by ASP.NET or by HTTP modules.
Web.config	The configuration data for an application.

Running the Default Web Application

The default Web application provides a rich experience based on built-in functionality and support. Without any changes to the default Web forms project, the application is ready to run on your local Web browser.

1. Press the **Ctrl+F5** key in Visual Studio. The application will build and display in your Web browser.



There are three main pages in this default Web application: Default.aspx (Home), About.aspx, and Contact.aspx. Each of these pages can be reached from the top navigation bar. There are also two additional pages contained in the Account folder, the Register.aspx page and Login.aspx page. These two pages allow you to use the membership capabilities of ASP.NET to create, store, and validate user credentials.

ASP.NET Web Forms Background

ASP.NET Web Forms are pages that are based on Microsoft ASP.NET technology, in which code that runs on the server dynamically generates Web page output to the browser or client device. An ASP.NET Web Forms page automatically renders the correct browser-compliant HTML for features such as styles,

layout, and so on. Web Forms are compatible with any language supported by the .NET common language runtime, such as Microsoft Visual Basic and Microsoft Visual C#. Also, Web Forms are built on the Microsoft .NET Framework, which provides benefits such as a managed environment, type safety, and inheritance.

When an ASP.NET Web Forms page runs, the page goes through a life cycle in which it performs a series of processing steps. These steps include initialization, instantiating controls, restoring and maintaining state, running event handler code, and rendering. As you become more familiar with the power of ASP.NET Web Forms, it is important for you to understand the page life cycle so that you can write code at the appropriate life-cycle stage for the effect you intend.

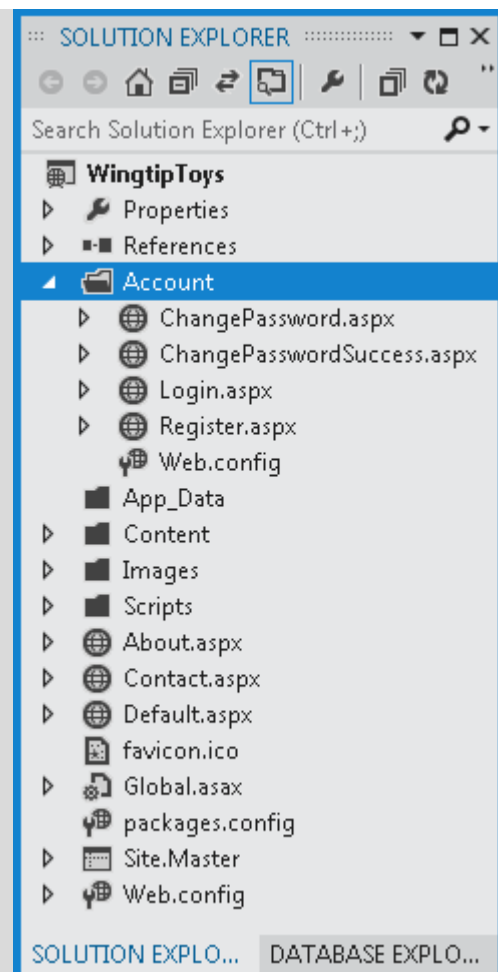
When a Web server receives a request for a page, it finds the page, processes it, sends it to the browser, and then discards all page information. If the user requests the same page again, the server repeats the entire sequence, reprocessing the page from scratch. Put another way, a server has no memory of pages that it has processed—page are stateless. The ASP.NET page framework automatically handles the task of maintaining the state of your page and its controls, and it provides you with explicit ways to maintain the state of application-specific information.

Web Application Features in the Web Forms Application Template

The ASP.NET Web Forms Application template provides a rich set of built-in functionality. It not only provides you with a *Home.aspx* page, an *About.aspx* page, a *Contact.aspx* page, but also includes membership functionality that registers users and saves their credentials so that they can log in to your website. This overview provides more information about some of the features contained in the ASP.NET Web Forms Application template and how they are used in the Wingtip Toys application.

Membership

[ASP.NET membership](#) stores your users' credentials in a database created by the application. When your users log in, the application validates their credentials by reading the database. Your project's Account folder contains the files that implement the various parts of membership: registering, logging in, changing a password, and authorizing access.



By default, the template creates a membership database using a default database name on an instance of SQL Server Express LocalDB, the development database server that comes with Visual Studio 11 Beta.

SQL Server Express LocalDB

SQL Server Express LocalDB is a lightweight version of SQL Server that has many programmability features of a SQL Server database. SQL Server Express LocalDB runs in user mode and has a fast, zero-configuration installation that has a short list of installation prerequisites. In Microsoft SQL Server, any database or Transact-SQL code can be moved from SQL Server Express LocalDB to SQL Server and Windows Azure SQL Database without any upgrade steps. So, SQL Server Express LocalDB can be used as a developer environment for applications targeting all editions of SQL Server. SQL Server Express LocalDB enables features such as stored procedures, user-defined functions and aggregates, .NET Framework integration, spatial types and others that are not available in SQL Server Compact.

Master Pages

An [ASP.NET master page](#) defines a consistent appearance and behavior for all of the pages in your application. The layout of the master page merges with the content from an individual content page to produce the final page that the user sees. In the Wingtip Toys application, you modify the *Site.master* master page so that all the pages in the Wingtip Toys website share the same distinctive logo and navigation bar.

HTML5

The ASP.NET Web Forms Application template uses [HTML5](#), which is the latest version of the HTML

markup language. HTML5 supports new elements and functionality that make it easier to create Web sites. For example, the Wingtip Toys application uses HTML5 in the *Site.Master* master page to create a navigation bar by placing a list of links inside the `<nav>` element. You can easily modify the **nav** element in the *Site.Master* page to create navigation for your own web applications.

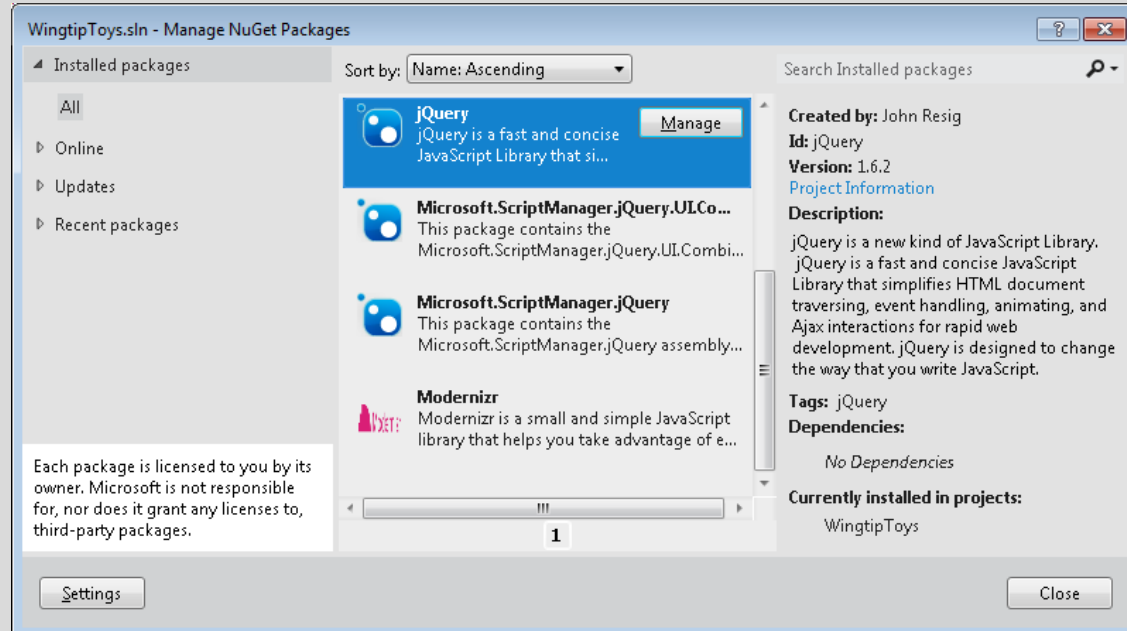
Additional HTML5 elements in the ASP.NET Web Forms Application template include [<header>](#), [<footer>](#), [<article>](#), [<section>](#), and [<hgroup>](#). The `<header>` element encloses a group of navigational aids. The `<footer>` element typically contains information like who authored the section, copyright information, and links to related documents. The `<article>` element encloses content that can stand on its own and potentially be distributed independently of other content on the page. The `<section>` element's role is to enclose a thematic grouping of content, usually with a heading. The `<hgroup>` element is useful when you want a group of a set of `<h1>` to `<h6>` elements that will be considered as a unit within the overall document outline.

Modernizr

For browsers that do not support HTML5, you can use [Modernizr](#). Modernizr is an open-source JavaScript library that can detect whether a browser supports HTML5 features, and enable them if it does not. In the ASP.NET Web Forms Application template, Modernizr is installed as a NuGet package.

NuGet Packages

The ASP.NET Web Forms Application template includes a set of **NuGet** packages. These packages provide componentized functionality in the form of open source libraries and tools. There is a wide variety of packages to help you create and test your applications. Visual Studio makes it easy to add, remove, and update NuGet packages. Developers can create and add packages to NuGet as well.



When you install a package, NuGet copies files to your solution and automatically makes whatever changes are needed, such as adding references and changing your *web.config* file. If you decide to remove the library, NuGet removes files and reverses whatever changes it made in your project so that no clutter is left. NuGet is available from the **Tools** menu in Visual Studio.

jQuery

jQuery is a fast and concise JavaScript Library that simplifies HTML document traversing, event handling, animating, and Ajax interactions for rapid web development. The jQuery JavaScript library is included in the ASP.NET Web Forms Application template as a NuGet package.

Unobtrusive Validation

Built-in validator controls have been configured to use unobtrusive JavaScript for client-side validation logic. This significantly reduces the amount of JavaScript rendered inline in the page markup and reduces the overall page size. Unobtrusive validation is added globally to the ASP.NET Web Forms Application template based on the setting in the <appSettings> element of the *Web.config* file at the root of the application.

Anti-XSS Library

The ASP.NET Web Forms Application template provides encoding routines that help to protect your application against [cross-site scripting \(XSS\)](#) attacks. XSS attacks attempt to inject client-side script into the pages of your web application. The [Anti-XSS](#) library also helps you to protect your application against [LDAP injection attacks](#) that are possible if user input is not properly validated.

Universal Providers

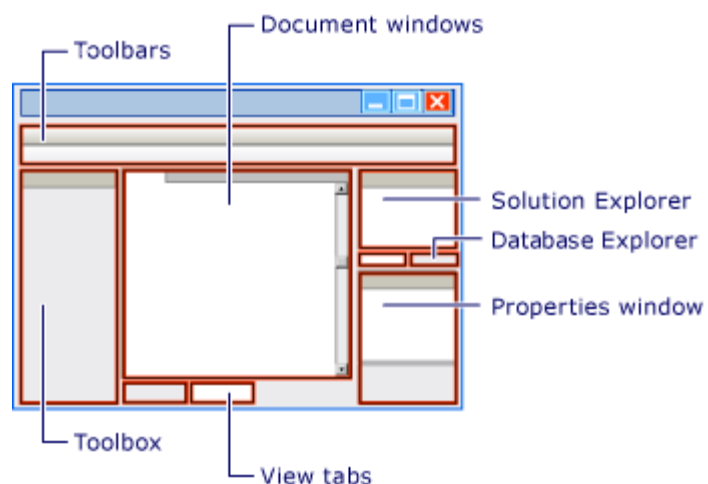
ASP.NET Universal Providers add provider support for all editions of SQL Server 2005 and later, as well as Windows Azure SQL Database. By default, the ASP.NET Web Forms Application template includes the ASP.NET Universal Providers package from NuGet. This means that cloud storage of membership data can quickly be published to SQL Database.

Entity Framework Code First

Besides the features in the ASP.NET Web Forms Application template, the Wingtip Toys application uses [Entity Framework Code First](#), which is a NuGet library that enables code-centric development when you work with data. Put simply, it creates the database portion of your application for you based on the code that you write. Using the Entity Framework, you retrieve and manipulate data as strongly typed objects. This lets you focus on the business logic in your application rather than the details of how data is accessed.

Touring Visual Studio

The primary windows in Visual Studio include the Solution Explorer, the Server Explorer (Database Explorer in Express), the Properties Window, the Toolbox, the Toolbar, and the Document Window.



For more information about Visual Studio, see Dev11 Beta Visual Guide to the Web Development IDE in Visual Studio.

Summary

In this tutorial you have created, reviewed and ran the default Web forms application. You have reviewed the different features of the default Web forms application and learned some basics about how to use the Visual Studio environment. In the following tutorials you'll create the data access layer.

Additional Resources

[Choosing the Right Programming Model](#)

[Web Application Projects versus Web Site Projects](#)

[ASP.NET Web Forms Pages Overview](#)

Create the Data Access Layer

This tutorial describes how to create, access, and review data from a database using ASP.NET Web Forms and Entity Framework Code First. This tutorial builds on the previous tutorial “Create the Project” and is part of the Wingtip Toy Store tutorial series. When you've completed this tutorial, you'll have a new folder named **Models** and you will have built data-access classes in that folder.

What you'll learn:

- How to create the data models.
- How to initialize and seed the database.
- How to update and configure the application to support the database.

These are the features introduced in the tutorial:

- Entity Framework Code First
- LocalDB
- Data Annotations

Creating the Data Models

The [Entity Framework](#) is an object-relational mapping (ORM) framework. It lets you work with relational data as objects, eliminating most of the data-access code that you'd usually need to write. Using the Entity Framework, you can issue queries using [LINQ](#), then retrieve and manipulate data as strongly typed objects. LINQ provides patterns for querying and updating data. Using EF allows you to focus on creating the rest of your application, rather than focusing on the data access fundamentals. Later in this tutorial series, we'll show you how to use the data to populate navigation and product queries.

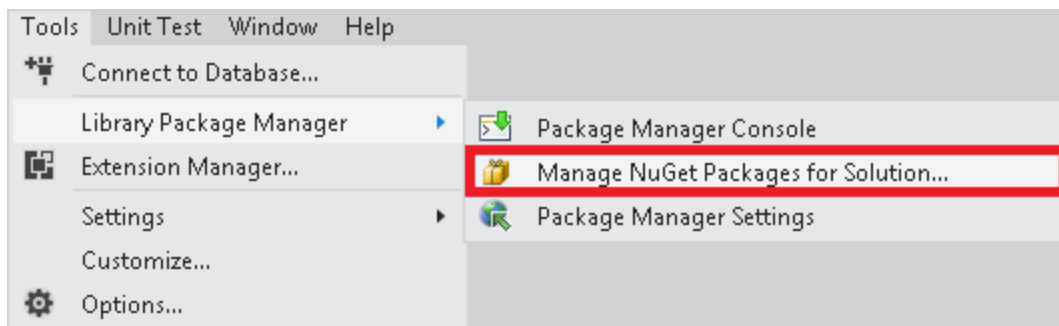
Entity Framework supports a development paradigm called Code First. Code First lets you define your data models using classes; you can then map these classes to an existing database or use them to generate a database. In this tutorial, you'll create the data models by writing data model classes. Then, you'll let the Entity Framework create the database on the fly from these new classes.

You'll begin by creating the entity classes that define the data models for the Web Forms application. Then you will create a context class that manages the entity classes and provides data access to the database. You will also create an initializer class that you will use to populate the database.

Installing the Entity Framework

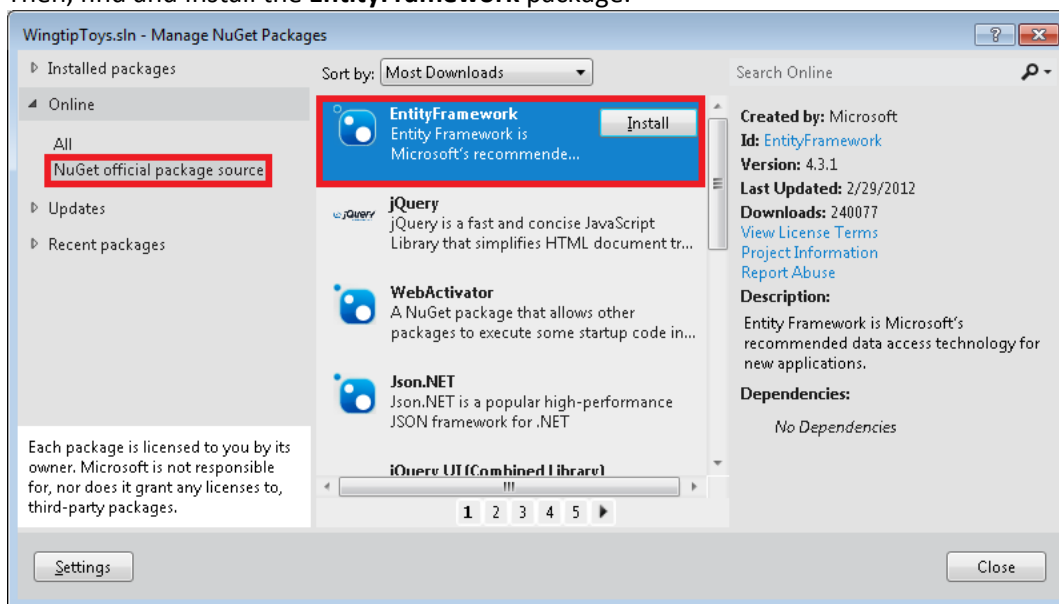
Before you can work with the Entity Framework, you must install it. This is easy using the **NuGet** package installer. NuGet is a Visual Studio extension that makes it easy to install and update open source libraries and tools in Visual Studio.

1. Within Visual Studio, from the **Tools** menu, select **Library Package Manager ->Manage NuGet Packages for Solution**.



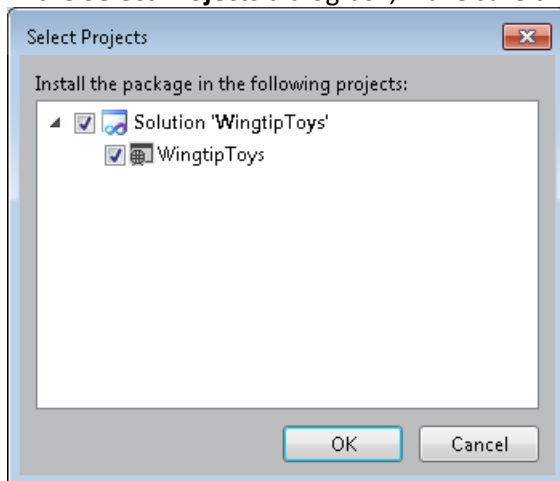
The **Manage NuGet Packages** dialog box is displayed within Visual Studio.

2. In the **Manage NuGet Packages** dialog box, select **NuGet official package source** on the left. Then, find and install the **EntityFramework** package.

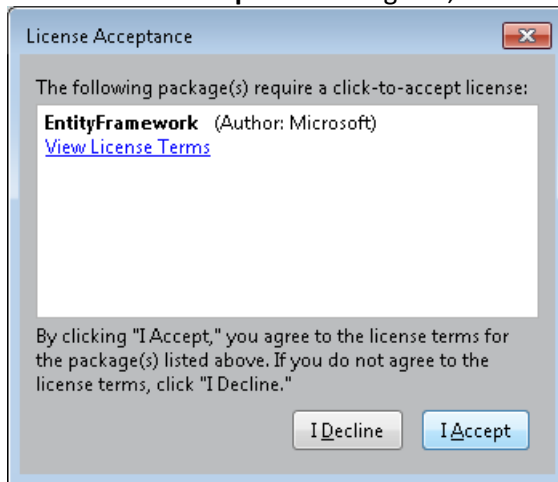


You will need to have an internet connection to download the package.

3. In the **Select Projects** dialog box, make sure the **WingtipToys** selection is selected and click **OK**.



4. In the **License Acceptance** dialog box, select **I Accept** to agree to the license terms.



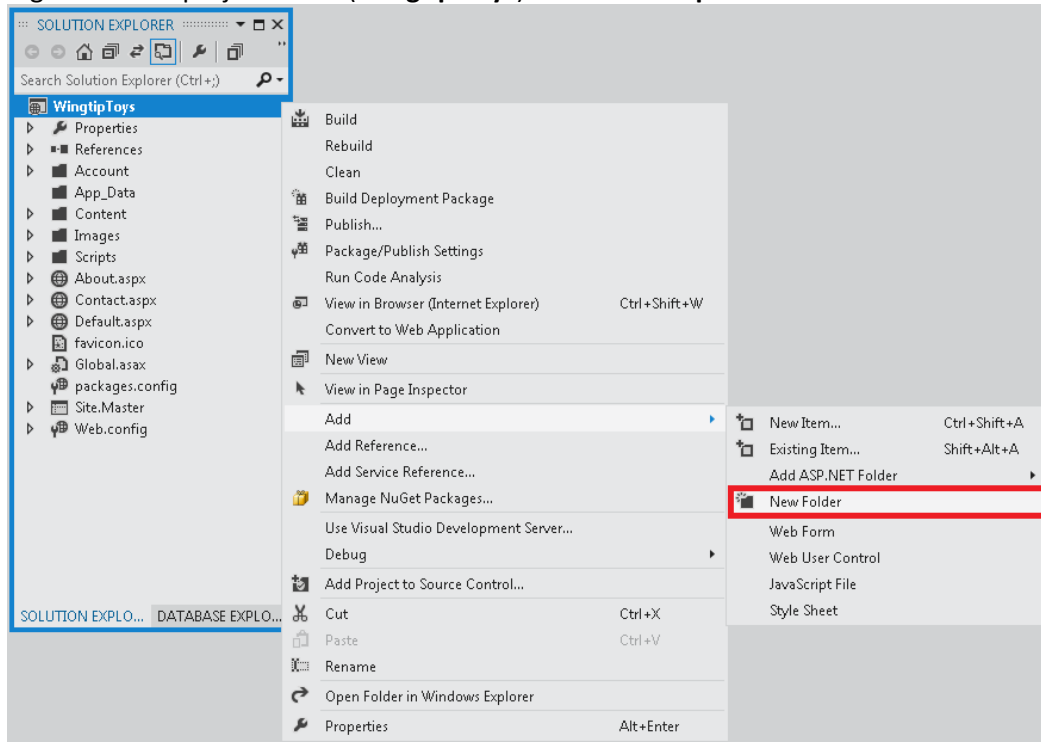
That's it! The Entity Framework is ready to go. Now you need to add the data models using entity classes.

Entity Classes

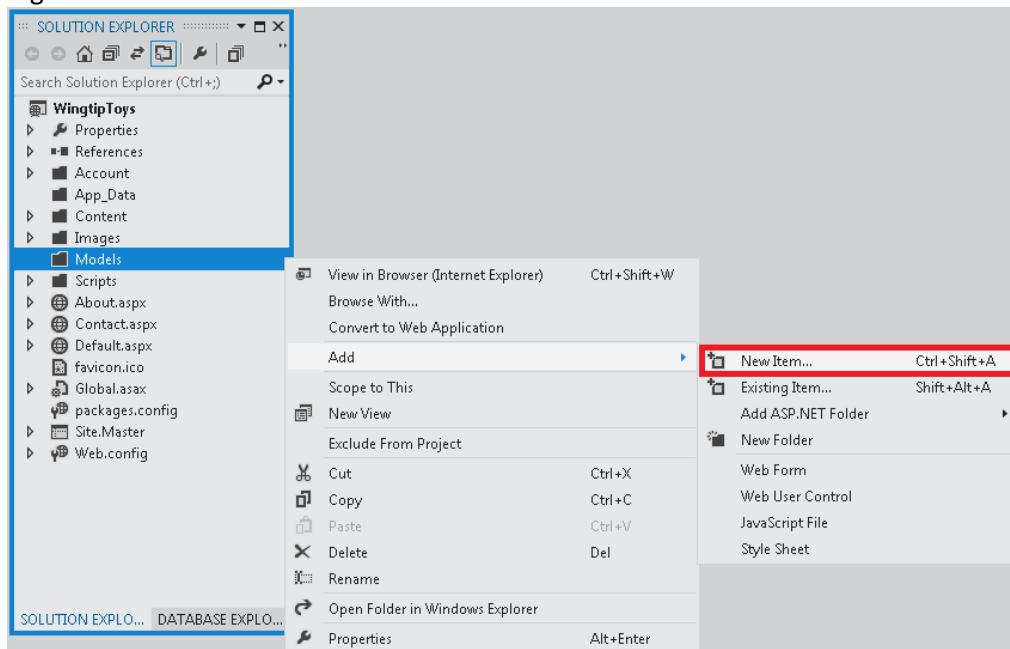
The classes you create to define the schema of the data are called entity classes. If you're familiar with database design, think of the entity classes as defining tables. Each property in the class specifies a column in the table of the database. These classes provide a lightweight, object-relational interface between object-oriented code and the relational table structure of the database.

In this tutorial, we'll start out by adding simple entity classes representing the schemas for products and categories.

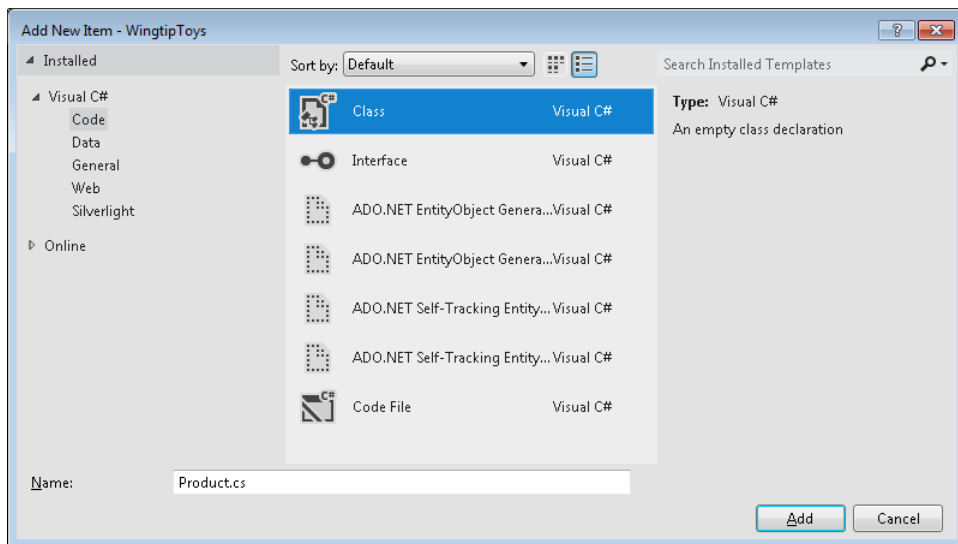
1. Right-click the project name (**Wingtip Toys**) in **Solution Explorer** and select **Add**→**New Folder**.



2. Name the new folder **Models**.
3. Right-click the **Models** folder and then select **Add**→**New Item**.



The **Add New Item** dialog box is displayed.



4. Under **Visual C#** from the **Installed** pane on the left, select **Code**.
5. Select **Class** from the middle pane and name this new class **Product.cs**.
6. Click **Add**.

The new class file is displayed in the editor.

7. Replace the default code with the following code:

```
using System.ComponentModel.DataAnnotations;

namespace WingtipToys.Models
{
    public class Product
    {
        [ScaffoldColumn(false)]
        public int ProductID { get; set; }

        [Required, StringLength(100), Display(Name = "Name")]
        public string ProductName { get; set; }

        [Required, StringLength(10000), Display(Name = "Product Description"),
        DataType(DataType.MultilineText)]
        public string Description { get; set; }

        public string ImagePath { get; set; }

        [Display(Name = "Price")]
        public double? UnitPrice { get; set; }

        public int? CategoryID { get; set; }

        public virtual Category Category { get; set; }
    }
}
```

- Repeat steps 3 through 6, but add a class named **Category.cs** and replace the default code with the following code:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace WingtipToys.Models
{
    public class Category
    {
        [ScaffoldColumn(false)]
        public int CategoryID { get; set; }

        [Required, StringLength(100), Display(Name = "Name")]
        public string CategoryName { get; set; }

        [Display(Name = "Product Description")]
        public string Description { get; set; }

        public virtual ICollection<Product> Products { get; set; }
    }
}
```

The **Category** class represents the type of product that the application is designed to sell (such as "Cars", "Boats", "Rockets", and so on), and the **Product** class represents the products (toys) in the database. Each instance of a **Product** object will correspond to a row within a database table, and each property of the Product class will map to a column in the table. Later in this tutorial, you'll review the product data contained in the database.

Data Annotations

You may have noticed that certain members of the classes have attributes specifying details about the member, such as `[StringLength(15)]` and `[Key]`. These are *data annotations*. The data annotation attributes can describe how to validate user input for that member, to specify formatting for it, and to specify how it is modeled.

Context Class

To start using the classes for data access, you must define a context class. As mentioned previously, the context class manages the entity classes and provides data access to the database.

This procedure adds a new C# context class to the **Models** folder.

- Right-click the **Models** folder and then select **Add ->New Item**. The **Add New Item** dialog box is displayed.
- Select **Code** from the **Installed** pane on the left.
- Select **Class** from the middle pane and name it **ProductContext.cs**.
- Click **Add** at the bottom of the dialog box.
- Replace the default code contained in the class with the following code:

```
using System.Data.Entity;

namespace WingtipToys.Models
{
    public class ProductContext : DbContext
    {
        public ProductContext()
            : base("WingtipToys")
        {
        }

        public DbSet<Category> Categories { get; set; }
        public DbSet<Product> Products { get; set; }
    }
}
```

This code adds the **System.Data.Entity** namespace so that you have access to all the core functionality of the Entity Framework, which includes the capability to query, insert, update, and delete data by working with strongly typed objects.

The **ProductContext** class represents the Entity Framework product database context, which handles fetching, storing, and updating **Product** class instances in the database. The **ProductContext** class derives from the **DbContext** base class provided by the Entity Framework.

Initializer Class

You will need to run some custom logic to initialize the database the first time the context is used. This will allow seed data to be added to the database so that you can immediately display products and categories.

This procedure adds a new C# initializer class to the **Models** folder.

1. Create another **Class** and name it **ProductDatabaseInitializer.cs**.
2. Replace the default code contained in the class with the following code:

```
using System.Collections.Generic;
using System.Data.Entity;

namespace WingtipToys.Model
{
    public class ProductDatabaseInitializer : DropCreateDatabaseIfModelChanges<ProductContext>
    {
        protected override void Seed(ProductContext context)
        {
            GetCategories().ForEach(c => context.Categories.Add(c));
            GetProducts().ForEach(p => context.Products.Add(p));
        }

        private static List<Category> GetCategories()
        {
            var categories = new List<Category> {
                new Category
            }
        }
    }
}
```

```
        {
            CategoryID = 1,
            CategoryName = "Cars"
        },
        newCategory
        {
            CategoryID = 2,
            CategoryName = "Planes"
        },
        newCategory
        {
            CategoryID = 3,
            CategoryName = "Trucks"
        },
        newCategory
        {
            CategoryID = 4,
            CategoryName = "Boats"
        },
        newCategory
        {
            CategoryID = 5,
            CategoryName = "Rockets"
        },
    };

    return categories;
}

private static List<Product> GetProducts()
{
    var products = newList<Product> {
        newProduct
        {
            ProductID = 1,
            ProductName = "Convertible Car",
            Description = "Convertible Car Description.",
            ImagePath="carconvert.jpg",
            UnitPrice = 22.50,
            CategoryID = 1
        },
        newProduct
        {
            ProductID = 2,
            ProductName = "Old-time Car",
            Description = "Old-time Car Description.",
            ImagePath="carearly.jpg",
            UnitPrice = 15.95,
            CategoryID = 1
        },
        newProduct
        {
            ProductID = 3,
            ProductName = "Fast Car",
            Description = "Fast Car Description.",
            ImagePath="carfast.jpg",
            UnitPrice = 32.99,
            CategoryID = 1
        }
    }
}
```

```
    },
    newProduct
    {
        ProductID = 4,
        ProductName = "Super Fast Car",
        Description = "Super Fast Car Description.",
        ImagePath="carfaster.jpg",
        UnitPrice = 8.95,
        CategoryID = 1
    },
    newProduct
    {
        ProductID = 5,
        ProductName = "Old Style Racer",
        Description = "Old Style Racer Description.",
        ImagePath="carracer.jpg",
        UnitPrice = 34.95,
        CategoryID = 1
    },
    newProduct
    {
        ProductID = 6,
        ProductName = "Ace Plane",
        Description = "Ace Plane Description.",
        ImagePath="planeace.jpg",
        UnitPrice = 95.00,
        CategoryID = 2
    },
    newProduct
    {
        ProductID = 7,
        ProductName = "Glider",
        Description = "Description Glider Plane.",
        ImagePath="planeglider.jpg",
        UnitPrice = 4.95,
        CategoryID = 2
    },
    newProduct
    {
        ProductID = 8,
        ProductName = "Paper Plane",
        Description = "Description Paper Plane.",
        ImagePath="planepaper.jpg",
        UnitPrice = 342.95,
        CategoryID = 2
    },
    newProduct
    {
        ProductID = 9,
        ProductName = "Propeller Plane",
        Description = "Description Propeller Plane.",
        ImagePath="planeprop.jpg",
        UnitPrice = 32.95,
        CategoryID = 2
    },
    newProduct
    {
        ProductID = 10,
```

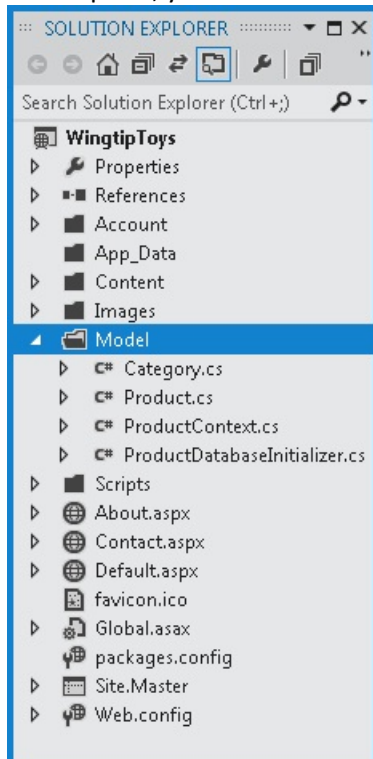
```
ProductName = "Early Truck",
            Description = "Description Early Truck.",
ImagePath="truckearly.jpg",
UnitPrice = 15.00,
CategoryID = 3
        },
newProduct
    {
ProductID = 11,
ProductName = "Fire Truck",
            Description = "Description Fire Truck.",
ImagePath="truckfire.jpg",
UnitPrice = 26.00,
CategoryID = 3
        },
newProduct
    {
ProductID = 12,
ProductName = "Double Decker Bus",
            Description = "Description Double Decker Bus.",
ImagePath="busdouble.jpg",
UnitPrice = 49.95,
CategoryID = 3
        },
newProduct
    {
ProductID = 13,
ProductName = "Big Truck",
            Description = "Description Big Truck.",
ImagePath="truckbig.jpg",
UnitPrice = 29.00,
CategoryID = 3
        },
newProduct
    {
ProductID = 14,
ProductName = "Big Ship",
            Description = "Big Ship Description.",
ImagePath="boatbig.jpg",
UnitPrice = 95.00,
CategoryID = 4
        },
newProduct
    {
ProductID = 15,
ProductName = "Paper Boat",
            Description = "Description Paper Boat.",
ImagePath="boatpaper.jpg",
UnitPrice = 4.95,
CategoryID = 4
        },
newProduct
    {
ProductID = 16,
ProductName = "Sail Boat",
            Description = "Description Sail Boat.",
ImagePath="boatsail.jpg",
UnitPrice = 42.95,
```

```
CategoryID = 4
    },
    newProduct
    {
        ProductID = 17,
        ProductName = "Rocket",
        Description = "Description Rocket.",
        ImagePath="rocket.jpg",
        UnitPrice = 12.95,
        CategoryID = 5
    }
};

return products;
}
}
```

When the database is created and initialized, the **Seed** property is overridden and set. When the **Seed** property is set, the values from the categories and products are used to populate the database. If you attempt to update the seed data by modifying the above code after the database has been created, you won't see any updates when you run the Web application. The reason is the above code uses an implementation of the **DropCreateDatabaseIfModelChanges** class to recognize if the model (schema) has changed before resetting the seed data. If no changes are made to the **Category** and **Product** entity classes, the database will not be reinitialized with the seed data.

At this point, you will have a new **Models** folder with four new classes:



Configuring the Application to Use the Data Model

Now that you've created the classes that represent the data, you have to configure the application to use the classes. In the *Global.asax* file, you add code that initializes the model. In the *Web.config* file you add information that tells the application what database you'll use to store the data that's represented by the new data classes.

Updating the Global.asax file

To initialize the data models when the application starts, add the following code highlighted in yellow to the **Application_Start** method in the **Global.asax.cs** file.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Security;
using System.Web.SessionState;
using System.Data.Entity;
using WingtipToys.Models;

namespace WingtipToys
{
    public class Global : System.Web.HttpApplication
    {
        void Application_Start(object sender, EventArgs e)
        {
            // Code that runs on application startup
            Database.SetInitializer<ProductContext>(new ProductDatabaseInitializer());
        }

        void Application_End(object sender, EventArgs e)
        {
            // Code that runs on application shutdown
        }

        void Application_Error(object sender, EventArgs e)
        {
            // Code that runs when an unhandled error occurs
        }

        void Session_Start(object sender, EventArgs e)
        {
            // Code that runs when a new session is started
        }

        void Session_End(object sender, EventArgs e)
        {
            // Code that runs when a session ends.
            // Note: The Session_End event is raised only when the sessionstate mode
            // is set to InProc in the Web.config file. If session mode is set to StateServer
```

```
// or SQLServer, the event is not raised.  
  
    }  
}  
}
```

In this code, when the application starts, the application specifies the initializer that will run during the first time the data is accessed.

Modifying the Web.Config File

Although Entity Framework Code First will generate a database for you in a default location when the database is populated with seed data, adding your own connection information to your application gives you control of the database location. You specify this database connection using a connection string in the application's *Web.config* file at the root of the project. By adding a new connection string, you can direct the location of the database (*wingtiptoys.mdf*) to be built in the application's data directory (*App_Data*), rather than its default location.

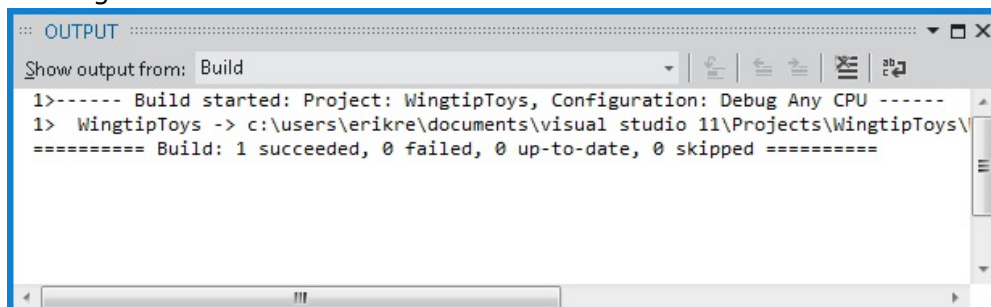
```
<connectionStrings>  
<addname="DefaultConnection"connectionString="Data Source=(LocalDb)\v11.0;Initial  
Catalog=aspnet-WingtipToys-20120110021050;Integrated Security=True"  
providerName="System.Data.SqlClient" />  
<addname="WingtipToys"connectionString="Data  
Source=(LocalDB)\v11.0;AttachDbFilename=|DataDirectory|\wingtiptoys.mdf;Integrated  
Security=True"providerName="System.Data.SqlClient" />  
</connectionStrings>
```

When the application is run for the first time, it will build the database at the location specified by the connection string. But before running the application, let's build it first.

Building the Application

To make sure that all the classes and changes to your Web application work, you should build the application.

1. From the **Build** menu, select **Build WingtipToys**.
The **Output** window is displayed, and if all went well, you see a "succeeded" message.



If you run into an error, re-check the above steps. The information in the **Output** window will indicate which file has a problem and where in the file a change is required. This information will allow you to determine what part of the above steps need to be fixed.

Summary

In this tutorial of the series you have created the data model, as well as, added the code that will be used to initialize and seed the database. You have also configured the application to use the data models when the application is run.

In the next tutorial, you'll update the UI, add navigation, and retrieve data from the database. This will result in the database being automatically created based on the entity classes that you created in this tutorial.

Additional Resources

[Entity Framework Overview](#)

[Beginner's Guide to the ADO.NET Entity Framework](#)

[Code First Development with Entity Framework](#) (video)

[Code First Relationships Fluent API](#)

[Code First Data Annotations](#)

[Productivity Improvements for the Entity Framework](#)

UI and Navigation

In this tutorial, you will modify the UI of the default Web application to support features of the Wingtip Toys store front application. Also, you will add simple and data bound navigation. This tutorial builds on the previous tutorial “Create the Data Access Layer” and is part of the Wingtip Toys tutorial series.

What you'll learn:

- How to change the UI to support features of the Wingtip Toys store front application.
- How to configure an HTML5 element to include page navigation.
- How to create a data-driven control to navigate to specific product data.
- How to display data from a database created using Entity Framework Code First.

ASP.NET Web Forms allow you to create dynamic content for your Web application. Each ASP.NET Web page is created in a manner similar to a static HTML Web page (a page that does not include server-based processing), but ASP.NET Web page includes extra elements that ASP.NET recognizes and processes to generate HTML when the page runs.

With a static HTML page (*.htm* or *.html* file), the server fulfills a Web request by reading the file and sending it as-is to the browser. In contrast, when someone requests an ASP.NET Web page (*.aspx* file), the page runs as a program on the Web server. While the page is running, it can perform any task that your Web site requires, including calculating values, reading or writing database information, or calling other programs. As its output, the page dynamically produces markup (such as elements in HTML) and sends this dynamic output to the browser.

Modifying the UI

You'll continue this tutorial series by modifying the **Default.aspx** page. You will modify the UI that's already established by the default template used to create the application. The type of modifications you'll do are typical when creating any Web Forms application. You'll do this by changing the title, replacing some content, and removing unneeded default content.

1. Open or switch to the **Default.aspx** page.
2. If the page appears in **Design** view, switch to **Source** view.
3. At the top of the page in the **@Page** directive, change the **Title** attribute to “Welcome”, as shown below.

```
<%@PageTitle="Welcome"Language="C#"MasterPageFile="~/Site.Master"AutoEventWireup="true"CodeBehind="Default.aspx.cs"Inherits="WingtipToys._Default"%>
```

4. Replace default content with the content highlighted in yellow below.

```
<asp:Contentrunat="server"ID="FeaturedContent"ContentPlaceHolderID="FeaturedContent">
<sectionclass="featured">
<divclass="content-wrapper">
<hgroupclass="title">
<h1><%=Page.Title%>!</h1>
```

```
<h2>Wingtip Toys can help you find the perfect gift</h2>
</hgroup>
<p>
We're all about transportation toys. You can order
any of our toys today. Each toy listing has detailed
information to help you choose the right toy.
</p>
</div>
</section>
</asp:Content>
```

5. Replace all the content contained in the **MainContent** placeholder , so that this HTML section will now be the following:

```
<asp:Content runat="server" ID="BodyContent" ContentPlaceHolderID="MainContent">
<section style="alignment-adjust: middle">
</section>
</asp:Content>
```

6. Save the **Default.aspx** page by selecting **Save Default.aspx** from the **File** menu.

The resulting page will appear as follows:

```
<%@Page Title="Welcome" Language="C#" MasterPageFile="~/Site.Master" AutoEventWireup="true" CodeBehind="Default.aspx.cs" Inherits="WingtipToys._Default"%>

<asp:Content ID="Content1" ContentPlaceHolderID="HeadContent" runat="server">
</asp:Content>
<asp:Content runat="server" ID="FeaturedContent" ContentPlaceHolderID="FeaturedContent">
<section class="featured">
<div class="content-wrapper">
<hgroup class="title">
<h1><%:Page.Title%>!</h1>
<h2>Wingtip Toys can help you find the perfect gift</h2>
</hgroup>
<p>
We're all about toys and transportation. You can order
any of our toys today. Each toy listing has detailed
information to help you find the right toy.
</p>
</div>
</section>
</asp:Content>
<asp:Content runat="server" ID="BodyContent" ContentPlaceHolderID="MainContent">
<section style="alignment-adjust: middle">
</section>
</asp:Content>
```

In the example, you have set the **Title** attribute of the **@Page** directive. When the HTML is displayed in a browser, the server code `<%:Page.Title%>` resolves to the content contained in the **Title** attribute.

The example page includes the basic elements that constitute an ASP.NET Web page. The page contains static text as you might have in an HTML page, along with elements that are specific to ASP.NET.

@Page Directive

ASP.NET Web Forms usually contain directives that allow you to specify page properties and configuration information for the page. The directives are used by ASP.NET as instructions for how to process the page, but they are not rendered as part of the markup that is sent to the browser.

The most commonly used directive is the @Page directive, which allows you to specify many configuration options for the page, including the following:

- The server programming language for code in the page, such as C#.
- Whether the page is a page with server code directly in the page, which is called a single-file page, or whether it is a page with code in a separate class file, which is called a code-behind page.
- Whether the page has an associated master page and should therefore be treated as a content page.
- Debugging and tracing options.

If you do not include an @Page directive in the page, or if the directive does not include a specific setting, a setting will be inherited from the *Web.config* configuration file or from the *Machine.config* configuration file. The *Machine.config* file provides additional configuration settings to all applications running on a machine.

Web Server Controls

In most ASP.NET Web Forms applications, you will add controls that allow the user to interact with the page, including buttons, text boxes, lists, and so on. These Web server controls are similar to HTML buttons and input elements. However, they are processed on the server, allowing you to use server code to set their properties. These controls also raise events that you can handle in server code.

Server controls use a special syntax that ASP.NET recognizes when the page runs. The tag name for ASP.NET server controls starts with an **asp:** prefix. This allows ASP.NET to recognize and process these server controls. The prefix might be different if the control is not part of the .NET Framework. In addition to the **asp:** prefix, ASP.NET server controls also include the **runat="server"** attribute and an **ID** that you can use to reference the control in server code.

When the page runs, ASP.NET identifies the server controls and runs the code that is associated with those controls. Many controls render some HTML or other markup into the page when it is displayed in a browser.

Server Code

Most ASP.NET Web Forms applications include code that runs on the server when the page is processed. As mentioned above, server code can be used to do a variety of things, such as adding data to a ListView control. ASP.NET supports many languages to run on the server, including C#, Visual Basic, J#, and others.

ASP.NET supports two models for writing server code for a Web page. In the single-file model, the code for the page is in a script element where the opening tag includes the `runat="server"` attribute. Alternatively, you can create the code for the page in a separate class file, which is referred to as the code-behind model. In this case, the ASP.NET Web Forms page generally contains no server code. Instead, the `@Page` directive includes information that links the .aspx page with its associated code-behind file.

The **CodeBehind** attribute contained in the `@Page` directive specifies the name of the separate class file, and the **Inherits** attribute specifies the name of the class within the code-behind file that corresponds to the page.

Updating the Master Page

In ASP.NET Web Forms, master pages allow you to create a consistent layout for the pages in your application. A single master page defines the look and feel and standard behavior that you want for all of the pages (or a group of pages) in your application. You can then create individual content pages that contain the content you want to display, as explained above. When users request the content pages, ASP.NET merges them with the master page to produce output that combines the layout of the master page with the content from the content page.

The new site needs a single logo to display on every page. To do this, you can modify the HTML on the master page.

1. In **Solution Explorer**, find and open the **Site.Master** page.
2. If the page is in **Design** view, switch to **Source** view.
3. Update the master page by replacing the existing `<title>` element with the markup shown below:

```
<title><%:Page.Title%> - Wingtip Toys</title>
```

4. Update the master page again by replacing the existing `<header>` element with the following markup:

```
<header>
<divclass="content-wrapper">
<divclass="float-left">
<pclass="site-title">
<aaid="A2"runat="server" href="~/ ">
<asp:ImageID="Logo"runat="server"
ImageUrl="~/images/logo.jpg"
BorderStyle="None"/>
</a>
</p>
</div>
<divclass="float-right">
<sectionid="login">
<asp:LoginViewrunat="server"ViewStateMode="Disabled">
<AnonymousTemplate>
<ul>
```

```
<li><arunat="server"
href=~ /Account/Register.aspx">Register</a>
</li>
<li><arunat="server"
href=~ /Account/Login.aspx">Log in</a>
</li>
</ul>
</AnonymousTemplate>
<LoggedInTemplate>
<p>
Hello, <arunat="server"
class="username"
href=~ /Account/ChangePassword.aspx"
title="Change password">
<asp:LoginNamerunat="server"
CssClass="username"/>
</a>!
<asp:LoginStatusrunat="server"
LogoutAction="Redirect"
LogoutText="Log off"
LogoutPageUrl=~ /"/>
</p>
</LoggedInTemplate>
</asp:LoginView>
</section>
<nav>
<ulid="menu">
<li><arunat="server"href=~ /">Home</a></li>
<li><arunat="server"href=~ /About.aspx">About</a></li>
<li><arunat="server"href=~ /Contact.aspx">Contact</a></li>
</ul>
</nav>
</div>
</div>
</header>
```

This HTML will display the image named *logo.jpg* from the root of the Web application which you'll add later. When a page that uses the master page is displayed in a browser, the logo will be displayed. If a user clicks on the logo, the user will navigate back to the *Default.aspx* page. The HTML anchor tag `<a>` wraps the image server control and allows the image to be included as part of the link. The **href** attribute for the anchor tag specifies the root `"~/`" of the Web site as the link location. By default, the *Default.aspx* page is displayed when the user navigates to the root of the Web site. The **Image**`<asp:Image>` server control includes addition properties, such as **BorderStyle**, that render as HTML when displayed in a browser.

Master Pages

A master page is an ASP.NET file with the extension `.master` (for example, *Site.Master*) with a predefined layout that can include static text, HTML elements, and server controls. The master page is identified by a special **@Master** directive that replaces the **@Page** directive that is used for ordinary `.aspx` pages.

In addition to the **@Master** directive, the master page also contains all of the top-level HTML elements for a page, such as **html**, **head**, and **form**. For example, on the master page you added above, you use an

HTML **table** for the layout, an **img** element for the company logo, static text, and server controls to handle common membership for your site. You can use any HTML and any ASP.NET elements as part of your master page.

In addition to static text and controls that will appear on all pages, the master page also includes one or more **ContentPlaceHolder** controls. These placeholder controls define regions where replaceable content will appear. In turn, the replaceable content is defined in content pages, such as *Default.aspx*, using the **Content** server control.

Adding Image Files

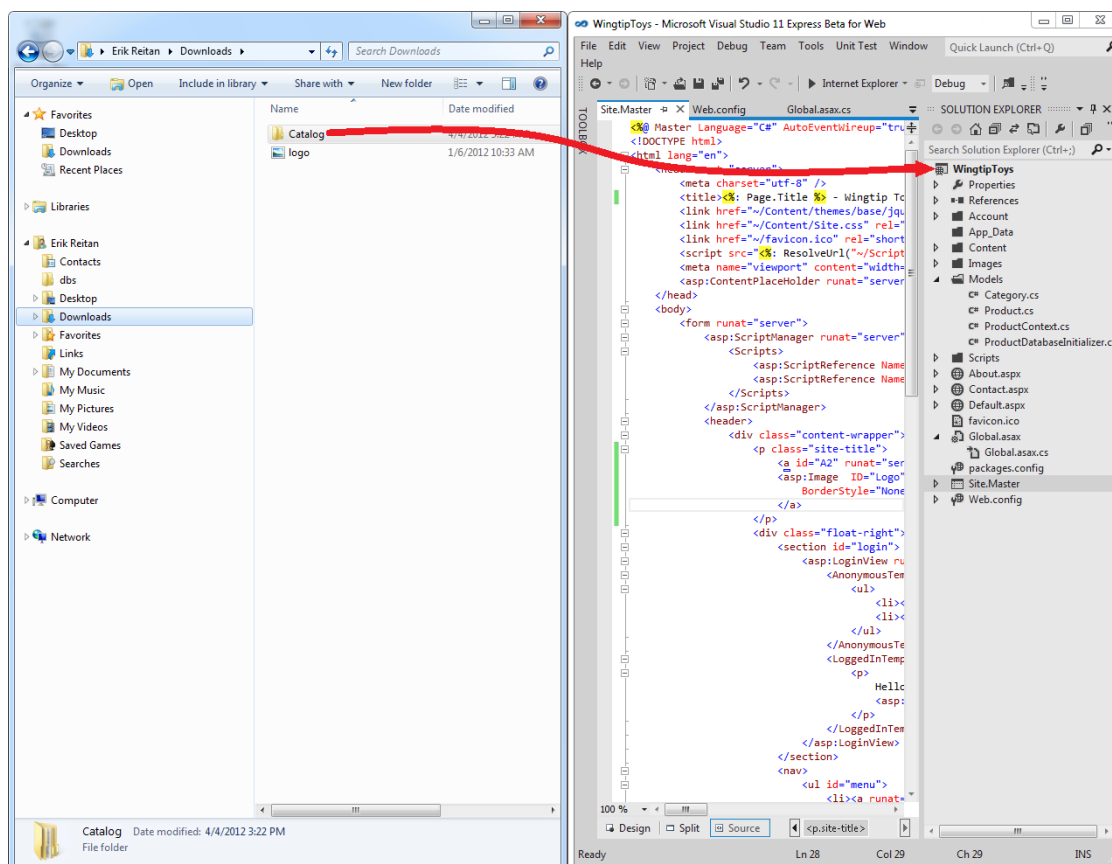
The logo image that is referenced above, along with all the product images, must be added to the Web application so that they can be seen when the project is displayed in a browser.

Download from MSDN Samples site:

[Getting Started with ASP.NET Web Forms 4.5](#)

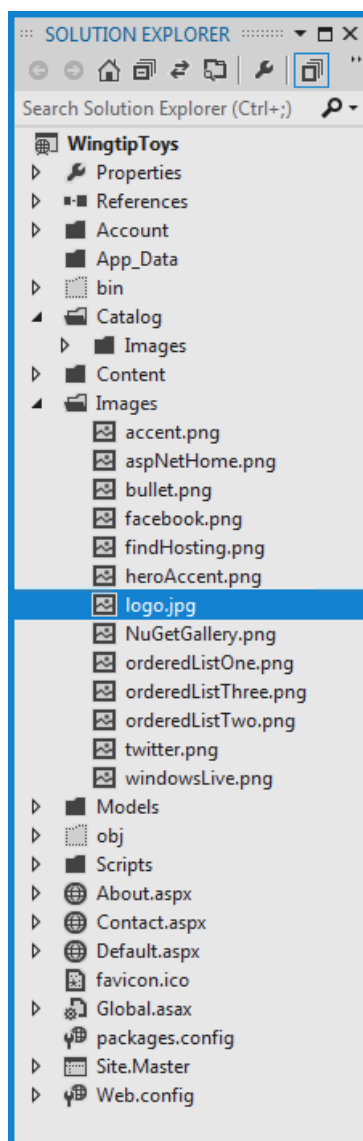
The download includes resources in the **WingtipToys-Assets** folder that are used to create the sample application.

1. If you haven't already done so, download the compressed sample files using the above link from the MSDN Samples site.
2. Once downloaded, open the *.zip* file and copy the contents to a local folder on your machine.
3. Find and open the *WingtipToys-Assets* folder.
4. By dragging and dropping, copy the *Catalog* folder from your local folder to the **root** of the Web application project in the **Solution Explorer** of Visual Studio.



- Next, copy the *logo.jpg* file from the *WingtipToys-Assets* folder to the *Images* folder of the Web application project in the **Solution Explorer** of Visual Studio.
- Click the **Show All Files** option at the top of the **Solution Explorer** to update the list of files if you don't see the new files.

Solution Explorer now shows the updated project files.



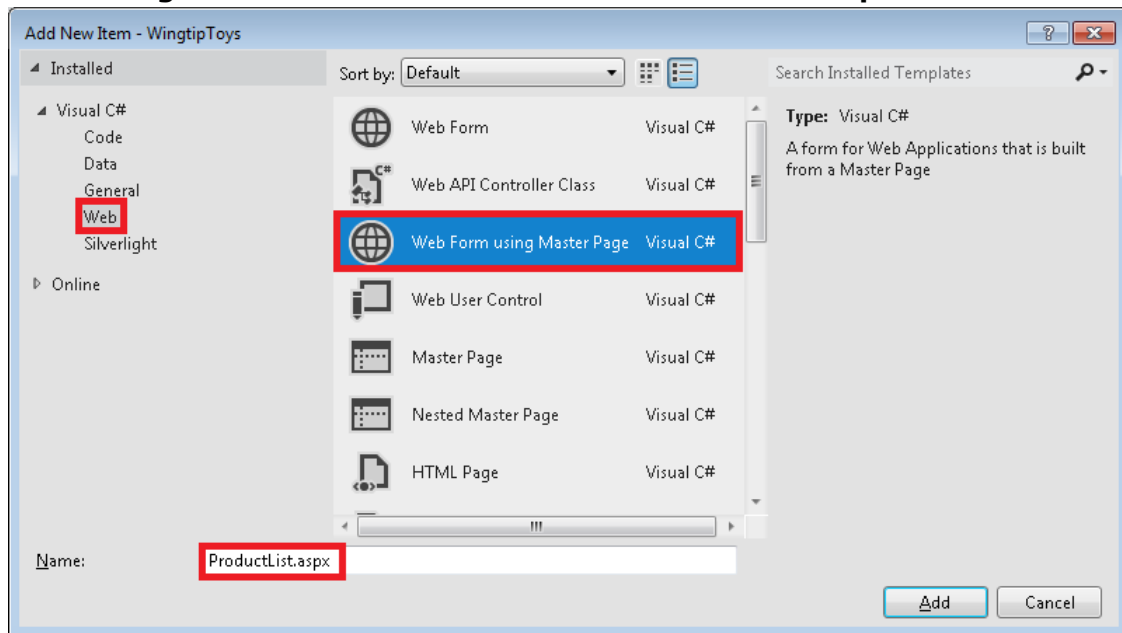
Adding Pages

Before adding navigation to the Web application, you'll first add the new pages that you'll navigate to. Later in this tutorial series, you'll display products and product details on these new pages.

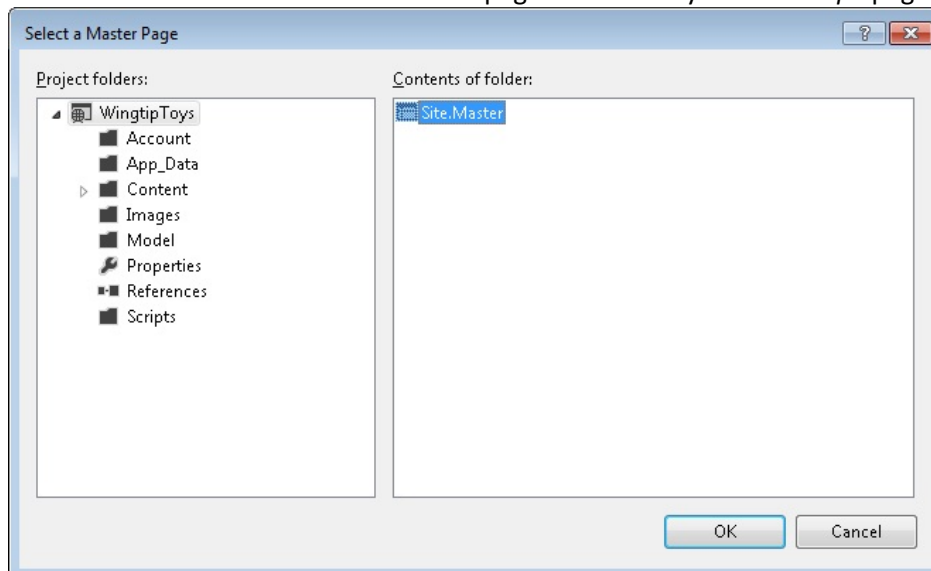
1. In **Solution Explorer**, right-click **WingtipToys**, click **Add**, and then click **New Item**.

The **Add New Item** dialog box is displayed.

2. Select the **Visual C#>Web** templates group on the left. Then, select **Web Form using Master Page** from the middle list and name it **ProductList.aspx**.



3. Select **Site.Master** to attach the master page to the newly created **.aspx** page.



4. Add an additional page named **ProductDetails.aspx** by following these steps.

Updating the StyleSheet

The default project template includes a cascading style sheet (CSS) file named *Site.css*. This file contains style rules that are applied to elements in the Web pages of the site. CSS styles define how elements are displayed and where they are positioned on the page. CSS styles

can be placed inline within a single HTML element, grouped in a style block within the head section of a Web page, or imported from a separate style sheet.

In this tutorial you will comment out two styles to change the look of the Web site.

1. In **Solution Explorer**, find the *Content* folder and open the *Site.css* file.
2. Find the `.main-content` style and comment out the lines that it contains by selecting the lines and clicking the comment out icon on the tool bar.

The styles will appear as shown below:

```
.main-content {  
    /*background: url("../Images/accent.png") no-repeat;  
    padding-left: 10px;  
    padding-top: 30px;*/  
}  
  
.featured + .main-content {  
    /*background: url("../Images/heroAccent.png") no-repeat;*/  
}
```

Modifying the Default Navigation

The default navigation for every page in the application can be modified by changing the **nav** element that's in the **Site.Master** page.

1. In **Solution Explorer**, locate and open the **Site.Master** page.
2. Find the **nav** element and modify the items as shown below:

```
<nav>  
<ulid="menu">  
<li><a href="/">Home</a></li>  
<li><a href="/About.aspx">About</a></li>  
<li><a href="/Contact.aspx">Contact</a></li>  
<li><a href="/ProductList.aspx">Products</a></li>  
</ul>  
</nav>
```

The **nav** element is new with HTML5. It makes adding and displaying navigation easy. As you can see in the above HTML, you modified each line item **** containing an anchor tag **<a>** with a link **href** attribute. Each **href** points to a page in the Web application. In the browser, when a user clicks on one of these links (such as **Products**), they will navigate to the page contained in the **href**(such as **ProductList.aspx**).

Adding a Data Control to Display Navigation Data

Next, you'll add a control to display all of the categories from the database. Each category will act as a link to the *ProductList.aspx* page. When a user clicks on a category link in the browser, they will navigate to the products page and see only the products associated with the selected category.

You'll use a **ListView** control to display all the categories contained in the database.

To add a **ListView** control to the master page:

1. In the **Site.Master** page, update the **<div>** element containing the **id="body"** attribute with the following markup:

```
<div id="body">
<section style="text-align: center">
<asp:ListView ID="categoryList"
Item Type="WingtipToys.Models.Category"
runat="server"
Select Method="GetCategories">
<ItemTemplate>
<b style="font-size: large; font-style: normal">
<a href="ProductList.aspx?id=<#:Item.CategoryID%>">
<#:Item.CategoryName%>
</a>
</b>
</ItemTemplate>
<ItemSeparatorTemplate> - </ItemSeparatorTemplate>
</asp:ListView>
</section>

<asp:ContentPlaceHolder runat="server" ID="FeaturedContent"/>
<section class="content-wrapper main-content clear-fix">
<asp:ContentPlaceHolder runat="server" ID="MainContent"/>
</section>
</div>
```

This code will display all the categories from the database. The **ListView** control displays each category name as link text and includes a link to the *ProductList.aspx* page with a query-string value containing the **ID** of the category. By setting the **ItemType** property in the **ListView** control, the data-binding expression **Item** is available within the **ItemTemplate** node and the control becomes strongly typed. You can select details of the **Item** object using IntelliSense, such as specifying the **CategoryName**. This code is contained inside the container **<#: %>** that marks a data-binding expression. By adding the **(:)** to the end of the **<#** prefix, the result of the data-binding expression is HTML-encoded. When the result is HTML-encoded, your application is better protected against cross-site script injection (XSS) and HTML injection attacks.

Note

When you add code by typing during development, you can be certain that a valid member of an object is found because strongly typed data controls show the available members based on IntelliSense. IntelliSense offers context-appropriate code choices as you type code, such as properties, methods, and objects.

Linking the Data Control to the Database

Before you can display data in the data control, you need to link the data control to the database. To make the link, you can modify the code behind of the *Site.Master.cs* file.

1. In **Solution Explorer**, right-click the *Site.Master* page and then click **View Code**. The *Site.Master.cs* file is opened in the editor.
2. Replace the existing code with the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using WingtipToys.Models;

namespace WingtipToys
{
    public partial class SiteMaster : MasterPage
    {
        protected void Page_Load(object sender, EventArgs e)
        {

        }

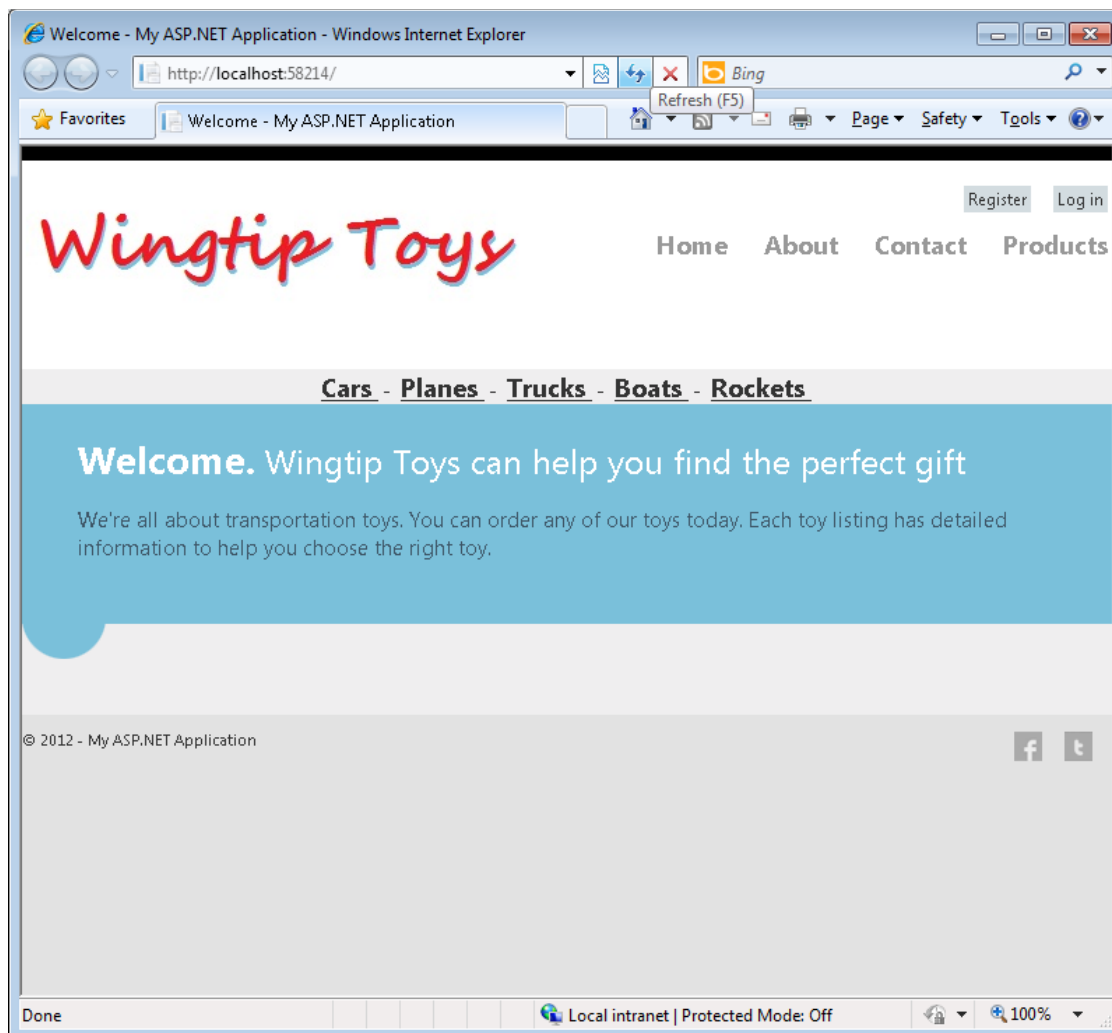
        public IQueryable<Category> GetCategories()
        {
            var db = new WingtipToys.Models.ProductContext();
            IQueryable<Category> query = db.Categories;
            return query;
        }
    }
}
```

The above code is executed when any page that uses the master page is loaded in the browser. The **ListView** control (named “categoryList”) that you added earlier in this tutorial uses model binding to select data. In the markup of the **ListView** control you set the control's **SelectMethod** property to the **GetCategories** method, shown above. The **ListView** control calls the **GetCategories** method at the appropriate time in the page life cycle and automatically binds the returned data. You will learn more about binding data in the next tutorial.

Running the Application and Creating the Database

Earlier in this tutorial series you created an initializer class (named “ProductDatabaseInitializer”) and specified this class in the *global.asax* file. The Entity Framework will generate the database when the application is run the first time because the **Application_Start** method contained in the *global.asax.cs* file will call the initializer class. The initializer class will use the model classes (**Category** and **Product**) that you added earlier in this tutorial series to create the database.

1. In Visual Studio press **Ctrl+F5**.



2.

When the application is run, the application will be compiled and the database named *wingtip toys.mdf* will be created. It will take a little time to set everything up first this first run. In the browser, you will see a category navigation menu. This menu was generated by retrieving the categories from the database. The page navigation text that was rendered from the `<nav>` element is on the right.

3. Close the browser.

Reviewing the Database

Open the *Web.config* file and look at the connection string section. You can see that the **AttachDbFilename** value in the connection string points to the **DataDirectory** for the Web application project. The value **|DataDirectory|** is a reserved value that represents the **App_Data** folder in the project. This folder is where the database is located.

```
<connectionStrings>
<addname="DefaultConnection"connectionString="Data Source=(LocalDb)\v11.0;Initial
Catalog=aspnet-WingtipToys-20120110021050;Integrated Security=True"
```

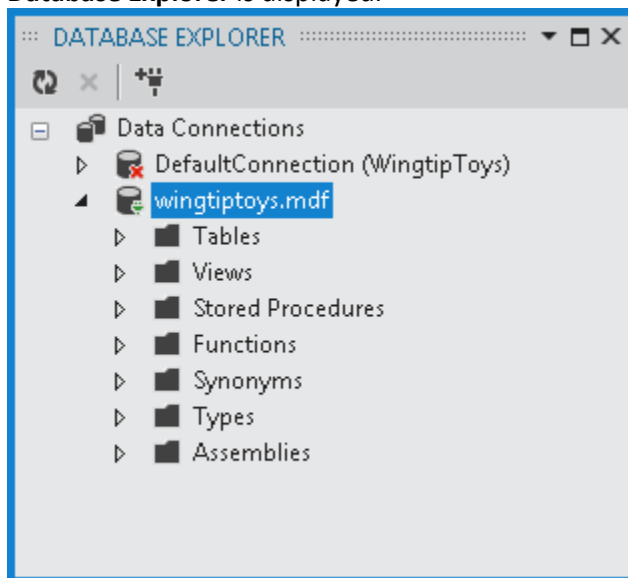
```
providerName="System.Data.SqlClient" />
<addname="WingtipToys"connectionString="Data
Source=(LocalDB)\v11.0;AttachDbFilename=|DataDirectory|\wingtip toys.mdf;Integrated
Security=True"providerName="System.Data.SqlClient" />
</connectionStrings>
```

Note

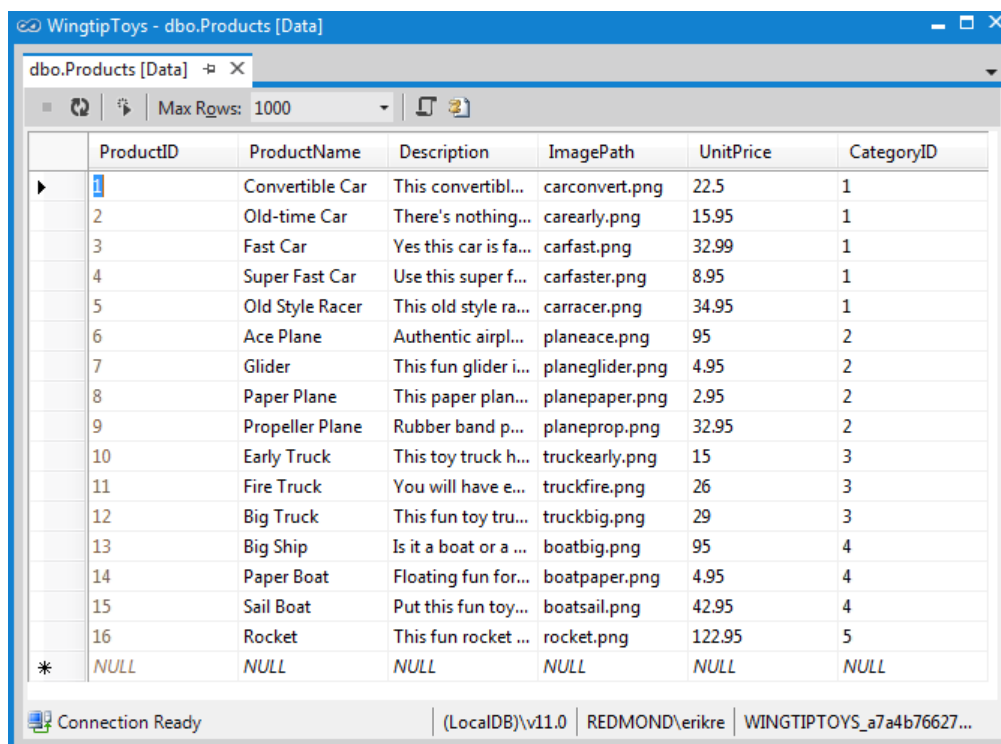
If the **App_Data** folder is not visible or if the folder is empty, select the **Show All Files** icon and the **Refresh** icon at the top of the **Solution Explorer** window. Expanding the width of the **Solution Explorer** windows may be required to show all available icons.

Now you can inspect the data contained in the **wingtip toys.mdf** database file by using the Database Explorer window (Server Explorer window in Visual Studio).

1. Expand the **App_Data** folder.
2. Right-click the **wingtip toys.mdf** database file and select **Open**.
Database Explorer is displayed.



3. Expand the **Tables** folder.
4. Right-click the **Products** table and select **Show Table Data**.
The **Products** table is displayed.

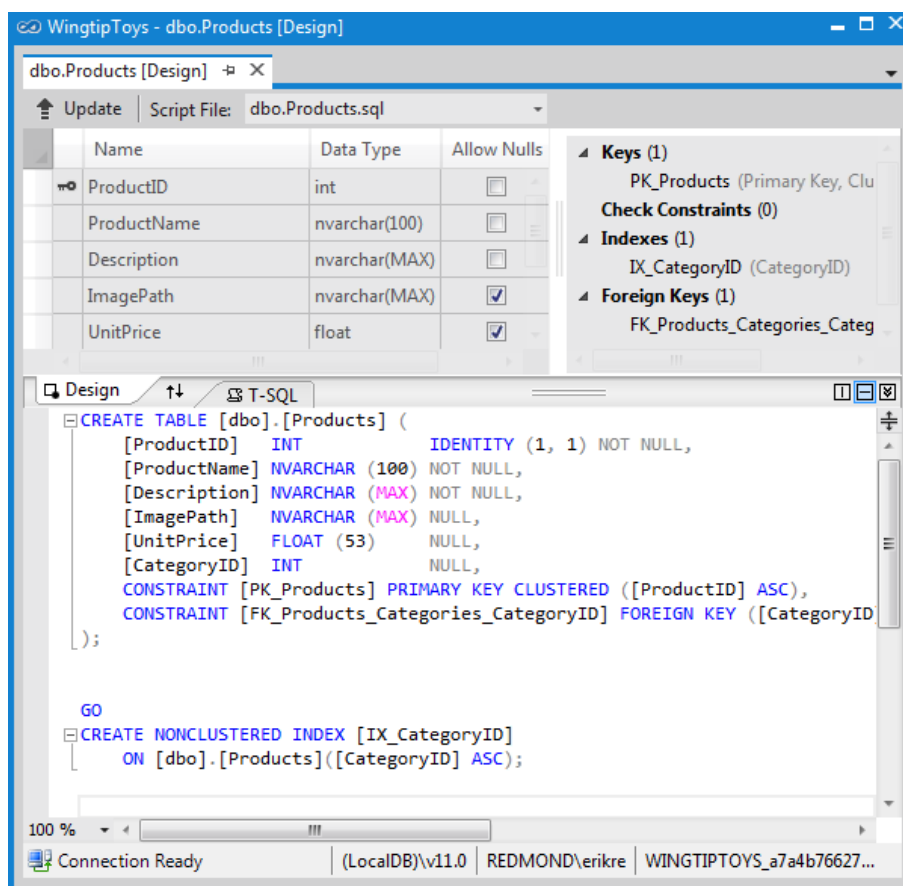


	ProductID	ProductName	Description	ImagePath	UnitPrice	CategoryID
▶	1	Convertible Car	This convertibl...	carconvert.png	22.5	1
	2	Old-time Car	There's nothing...	carearly.png	15.95	1
	3	Fast Car	Yes this car is fa...	carfast.png	32.99	1
	4	Super Fast Car	Use this super f...	carfaster.png	8.95	1
	5	Old Style Racer	This old style ra...	carracer.png	34.95	1
	6	Ace Plane	Authentic airpl...	planeace.png	95	2
	7	Glider	This fun glider i...	plane glider.png	4.95	2
	8	Paper Plane	This paper plan...	plane paper.png	2.95	2
	9	Propeller Plane	Rubber band p...	plane prop.png	32.95	2
	10	Early Truck	This toy truck h...	truckearly.png	15	3
	11	Fire Truck	You will have e...	truckfire.png	26	3
	12	Big Truck	This fun toy tru...	truckbig.png	29	3
	13	Big Ship	Is it a boat or a ...	boatbig.png	95	4
	14	Paper Boat	Floating fun for...	boatpaper.png	4.95	4
	15	Sail Boat	Put this fun toy...	boatsail.png	42.95	4
	16	Rocket	This fun rocket ...	rocket.png	122.95	5
*	NULL	NULL	NULL	NULL	NULL	NULL

Connection Ready | (LocalDB)\v11.0 | REDMOND\erikre | WINGTIPTOYS_a7a4b76627...

This view lets you see and modify the data in the **Products** table by hand.

5. Close the **Products** table window.
6. In the **Database Explorer**, right-click the **Products** table and select **Open Table Definition**. The data design for the **Products** table is displayed.



Here you see the SQL DDL statement that was used to create the table. You can also use the UI to modify the schema. However, defining the schema as you did using entity classes in the previous tutorial may be quicker and easier.

Summary

In this tutorial of the series you have added some basic UI, graphics, pages, and navigation. Additionally, you ran the Web application, which created the database from the data classes that you added in the previous tutorial. You also viewed the contents of the **Products** table of the database. In the following tutorial, you'll display data items and details from the database.

Additional Resources

[Introduction to Programming ASP.NET Web Pages](#)

[ASP.NET Web Server Controls Overview](#)

[CSS Tutorial](#)

Display Data Items and Details

This tutorial describes how to display data items and data item details using ASP.NET Web Forms and Entity Framework Code First. This tutorial builds on the previous tutorial “UI and Navigation” and is part of the Wingtip Toy Store tutorial series. When you've completed this tutorial, you'll be able to see products on the *ProductsList.aspx* page and details about an individual product on the *ProductDetails.aspx* page.

What you'll learn:

- How to add a data control to display products from the database.
- How to connect a data control to the selected data.
- How to add a data control to display product details from the database.
- How to retrieve a value from the query string and use that value to limit the data that's retrieved from the database.

These are the features introduced in the tutorial:

- Model Binding
- Value providers

Adding a Data Control to Display Products

When binding data to a server control, there are a few different options you can use. The most common options include adding a data source control, adding code by hand, or using model binding.

Using a Data Source Control to Bind Data

Adding a data source control allows you to link the data source control to the control that displays the data. This approach allows you to declaratively connect server-side controls directly to data sources, rather than using a programmatic approach.

Coding By Hand to Bind Data

Adding code by hand involves reading the value, checking for a null value, attempting to convert it to the appropriate type, checking whether the conversion was successful, and finally, using the value in the query. You would use this approach when you need to retain full control over your data-access logic.

Using Model Binding to Bind Data

Using model binding allows you to bind results using far less code and gives you the ability to reuse the functionality throughout your application. Model binding aims to simplify working with code-focused data-access logic while still retaining the benefits of a rich, data-binding framework.

Displaying Products

In this tutorial, you'll use model binding to bind data. To configure a data control to use model binding to select data, you set the control's **SelectMethod** property to the name of a method in the page's code. The data control calls the method at the appropriate time in the page life cycle and automatically binds the returned data. There's no need to explicitly call the **DataBind** method.

Using the steps below, you'll modify the markup in the *ProductList.aspx* page so that the page can display products.

1. In **Solution Explorer**, open the *ProductList.aspx* page.
2. Replace the existing markup with the following markup:

```
<%@PageTitle="Products"Language="C#"MasterPageFile="~/Site.Master"AutoEventWireup="true"
CodeBehind="ProductList.aspx.cs"Inherits="WingtipToys.ProductList"%>
<asp:ContentID="Content1"ContentPlaceHolderID="HeadContent"runat="server">
</asp:Content>
<asp:ContentID="Content2"ContentPlaceHolderID="FeaturedContent"runat="server">
<sectionclass="featured">
<divclass="content-wrapper">
<hgroupclass="title">
<h1><%:Page.Title%></h1>
</hgroup>

<sectionclass="featured">
<ul>
<asp:ListViewID="productList"runat="server"
DataKeyNames="ProductID"
GroupItemCount="3"ItemType="WingtipToys.Models.Product"SelectMethod="GetProducts">
<EmptyDataTemplate>
<tablerunat="server">
<tr>
<td>No data was returned.</td>
</tr>
</table>
</EmptyDataTemplate>
<EmptyItemTemplate>
<tdrunat="server"/>
</EmptyItemTemplate>
<GroupTemplate>
<trID="itemPlaceholderContainer"runat="server">
<tdID="itemPlaceholder"runat="server"></td>
</tr>
</GroupTemplate>
<ItemTemplate>
<tdrunat="server">
<table>
<tr>
<td>&nbsp;</td>
<td>
<a href='ProductDetails.aspx?productID=<%#:Item.ProductID%>'>
<imgsrc='Catalog/Images/Thumbs/<%#:Item.ImagePath%>'
width="100"height="75"border="1"/></a>
</td>
<td>
<a href='ProductDetails.aspx?productID=<%#:Item.ProductID%>'>
<spanclass="ProductName">
<%#:Item.ProductName%>
</span>
</a>
<br/>
<spanclass="ProductPrice">
<b>Price: </b><%#:String.Format("{0:c}", Item.UnitPrice)%>
</span>
<br/>
</td>
```

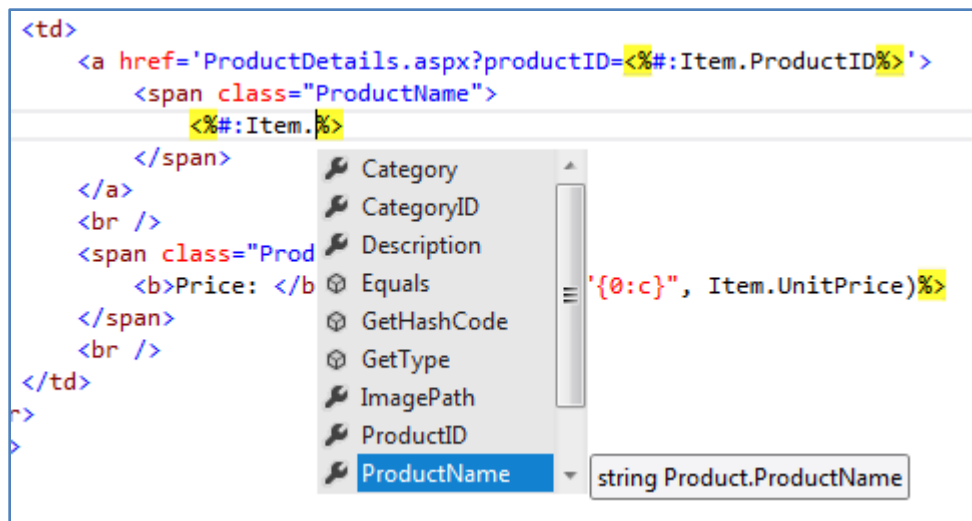
```
</tr>
</table>
</td>
</ItemTemplate>
<LayoutTemplate>
<table runat="server">
<tr runat="server">
<td runat="server">
<table ID="groupPlaceholderContainer" runat="server">
<tr ID="groupPlaceholder" runat="server"></tr>
</table>
</td>
</tr>
<tr runat="server"><td runat="server"></td></tr>
</table>
</LayoutTemplate>
</asp:ListView>
</ul>
</section>
</div>
</section>
</asp:Content>
<asp:Content ID="Content3" ContentPlaceHolderID="MainContent" runat="server">
</asp:Content>
```

This code uses a **ListView** control named "productList" to display the products.

```
<asp:ListView ID="productList" runat="server">
```

The **ListView** control displays data in a format that you define by using templates and styles. It is useful for data in any repeating structure. This **ListView** example simply shows data from the database, however you can enable users to edit, insert, and delete data, and to sort and page data, all without code.

By setting the **ItemType** property in the **ListView** control, the data-binding expression **Item** is available and the control becomes strongly typed. As mentioned in the previous tutorial, you can select details of the **Item** object using IntelliSense, such as specifying the **ProductName**:



In addition, you are using model binding to specify a **SelectMethod** value. This value (**GetProducts**) will correspond to the method that you will add to the code behind to display products in the next step.

Adding Code to Display Products

In this step, you'll add code to populate the **ListView** control with product data from the database. The code will support showing products by individual category, as well as all products.

1. In **Solution Explorer**, right-click *ProductList.aspx* and then click **View Code**.
2. Replace the existing code in the *ProductList.aspx.cs* file with the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using WingtipToys.Models;
using System.Web.ModelBinding;

namespace WingtipToys
{
    public partial class ProductList : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {

        }

        public IQueryable<Product> GetProducts([QueryString("id")] int? categoryId)
        {
            var _db = new WingtipToys.Models.ProductContext();
            IQueryable<Product> query = _db.Products;
            if (categoryId.HasValue && categoryId > 0)
            {
                query = query.Where(p => p.CategoryID == categoryId);
            }
        }
    }
}
```

```
return query;
    }
}
```

This code shows the **GetProducts** method that's referenced by the **ItemType** property of the **ListView** control in the *ProductList.aspx* page. To limit the results to a specific category in the database, the code sets the **categoryId** value from the query string value passed to the *ProductList.aspx* page when the *ProductList.aspx* page is navigated to. The **QueryStringAttribute** class in the **System.Web.ModelBinding** namespace is used to retrieve the value of the query string variable **id**. This instructs model binding to try to bind a value from the query string to the **categoryId** parameter at run time.

When a valid category is passed as a query string to the page, the results of the query are limited to those products in the database that match the **categoryId** value. For instance, if the URL to the *ProductsList.aspx* page is the following:

<http://localhost/ProductList.aspx?id=1>

The page displays only the products where the category equals 1.

If no query string is included when navigating to the *ProductList.aspx* page, all products will be displayed.

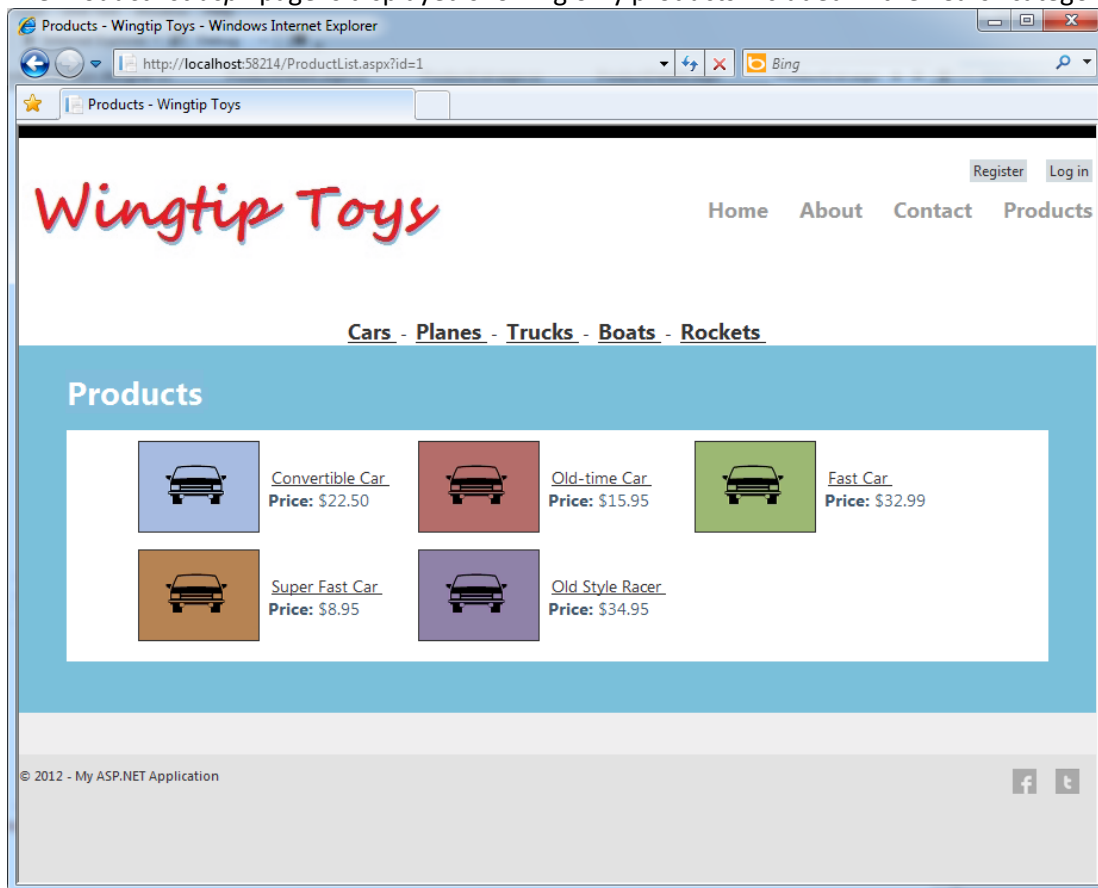
The sources of values for these methods are referred to as **value providers** (such as [QueryString](#)), and the parameter attributes that indicate which value provider to use are referred to as value provider attributes (such as "**id**"). ASP.NET includes value providers and corresponding attributes for all of the typical sources of user input in a Web Forms application, such as the query string, cookies, form values, controls, view state, session state, and profile properties. You can also write custom value providers.

Running the Application

Run the application now to see how you can view all of the products or just a set of products limited by category.

1. In the Solution Explorer, right-click the *Default.aspx* page and select **View in Browser**. The browser will open and show the *Default.aspx* page.

2. Select "Cars" from the category navigation menu on the left.
The *ProductList.aspx* page is displayed showing only products included in the "Cars" category.



3. Select **Products** from the navigation menu at the top.

Again, the *ProductList.aspx* page is displayed with the entire list of products.



Adding a Data Control to Display Product Details

Next, you'll modify the markup in the *ProductDetails.aspx* page so that the page can display information about an individual product.

1. In **Solution Explorer**, open the *ProductDetails.aspx* page.
2. Replace the existing markup with the following markup:

```

<%@PageTitle="Product Details"Language="C#"MasterPageFile=~\Site.Master"AutoEventWireup="true"
CodeBehind="ProductDetails.aspx.cs"Inherits="WingtipToys.ProductDetails"%>
<asp:ContentID="Content1"ContentPlaceHolderID="HeadContent"runat="server">
</asp:Content>
<asp:ContentID="Content2"ContentPlaceHolderID="FeaturedContent"runat="server">
<asp:FormViewID="productDetails"runat="server"ItemType="WingtipToys.Models.Product"SelectMethod="GetProduct"
RenderOuterTable="false">
<ItemTemplate>
<div>
<h1><%#:Item.ProductName%></h1>
</div>
<br/>
<table>
<tr>
<td>
<imgsrc='Catalog/Images/<%#:Item.ImagePath%>'border="1"alt='<%#:Item.ProductName%>'height="300"/>
</td>
<tdstyle="vertical-align: top">
<b>Description:</b><br/><%#:Item.Description%>
<br/>
<span><b>Price:</b>&nbsp;<%#:String.Format("{0:c}", Item.UnitPrice) %></span>
<br/>
<span><b>Product Number:</b>&nbsp;<%#:Item.ProductID%></span>
<br/>
</td>
</tr>
</table>
</ItemTemplate>
</asp:FormView>
</asp:Content>
<asp:ContentID="Content3"ContentPlaceHolderID="MainContent"runat="server">
</asp:Content>

```

This code uses a **FormView** control to display details about an individual product. The **FormView** control is used to display a single record at a time from a data source. When you use the **FormView** control, you create templates to display and edit data-bound values. The templates contain controls, binding expressions, and formatting that define the look and functionality of the form.

To connect the above markup to the database, you must add additional code to the *ProductDetails.aspx* code.

1. In **Solution Explorer**, right-click *ProductDetails.aspx* and then click **View Code**.
2. Replace the existing code with the following code:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using WingtipToys.Models;

```

```
using System.Web.ModelBinding;

namespace WingtipToys
{
    public partial class ProductDetails : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {

        }

        public IQueryable<Product> GetProduct([QueryString("productID")]int? productId)
        {
            var _db = new WingtipToys.Models.ProductContext();
            IQueryable<Product> query = _db.Products;
            if (productId.HasValue && productId > 0)
            {
                query = query.Where(p => p.ProductID == productId);
            }
            else
            {
                query = null;
            }
            return query;
        }
    }
}
```

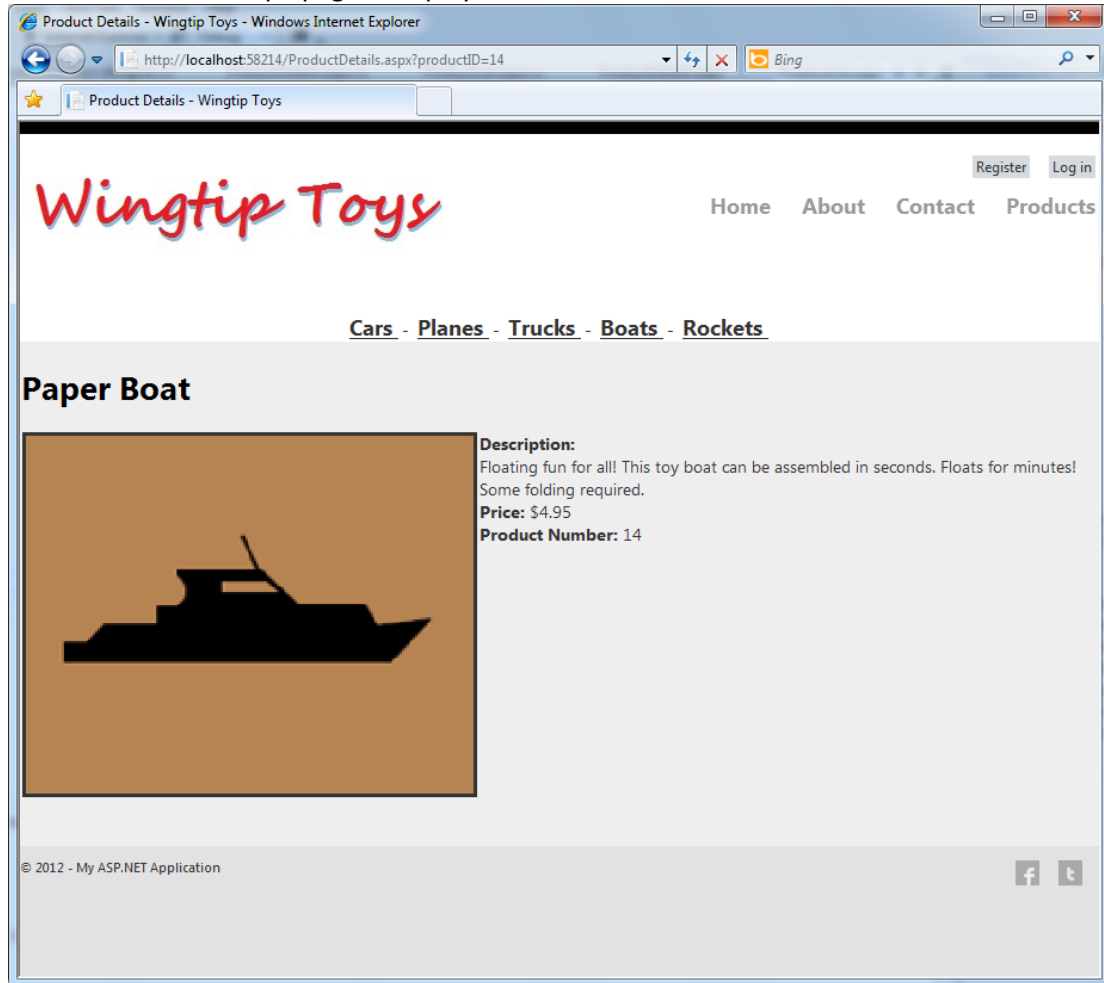
This code checks for a "productID" query-string value. If a valid query-string value is found, the matching product is displayed. If no query-string is found, or the query-string value is not valid, no product is displayed on the *ProductDetails.aspx* page.

Running the Application

Now you can run the application to see an individual product displayed based on the id of the product.

1. In the Solution Explorer, right-click the *Default.aspx* page and select **View in Browser**. The browser will open and show the *Default.aspx* page.
2. Select "Boats" from the category navigation menu on the left. The *ProductList.aspx* page is displayed.

3. Select the "Paper Boat" product from the product list.
The *ProductDetails.aspx* page is displayed.



Summary

In this tutorial of the series you have add markup and code to display a product list and to display product details. During this process you have learned about strongly typed data controls, model binding, and value providers.

Conclusion

This completes part 1 of the ASP.NET 4.5 Web Forms tutorial series. For more information about new Web Forms features available in ASP.NET 4.5 Beta and Visual Studio 11 Beta, see [What's New in ASP.NET 4.5 and Visual Studio 11 Beta](#).

Acknowledgements

I would like to thank the following people who made significant contributions to the content of this tutorial series:

- [Alberto Poblacion](#), MVP & MCT, Spain
- [Alex Thissen, Netherlands](#)(twitter:[@alexthissen](#))
- [Andre Tournier, USA](#)
- Apurva Joshi, Microsoft

- [BojanVrhovnik, Slovenia](#)
- [Bruno Sonnino, Brazil](#) (twitter: [@bsonnino](#))
- [Carlos dos Santos, Brazil](#)
- [Dave Campbell, USA](#)(twitter:[@windowsdevnews](#))
- [Michael Sharps, USA](#)(twitter:[@mrsharps](#))
- Mike Pope, Microsoft
- [Mitchel Sellers, USA](#)(twitter:[@MitchelSellers](#))
- [Paul Cociuba, Microsoft](#)
- [Paulo Morgado, Portugal](#)
- [Pranav Rastogi, Microsoft](#)
- [Tim Ammann, Microsoft](#)
- [Tom Dykstra, Microsoft](#)



By [Erik Reitan](#), Erik Reitan is a Senior Programming Writer on Microsoft's Web Platform & Tools Content Team. During his spare time he enjoys developing Windows Phone and Windows 8 apps.

Get more e-books from www.ketabton.com
Ketabton.com: The Digital Library