# Introduction and Overview

By Erik Reitan | December 30, 2013

**DOWNLOAD ASSETS: Getting Started with ASP.NET 4.5 Web Forms and Visual Studio 2013 - Wingtip Toys (C#)**

This tutorial series will teach you the basics of building an ASP.NET Web Forms application using ASP.NET 4.5 and Microsoft Visual Studio Express 2013 for Web.

## Introduction

This series of tutorials guides you through the steps required to create an ASP.NET Web Forms application using Visual Studio Express 2013 for Web and ASP.NET 4.5.

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

Comments are welcome, and we'll make every effort to update this tutorial series based on your suggestions.

### Download completed project

You can download a C# project that contains the completed tutorial.

- Getting Started with ASP.NET 4.5 Web Forms and Visual Studio 2013 - Wingtip Toys (C#)

### 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 (OOP) language
- Familiar with Web development concepts (HTML, CSS, JavaScript)
- Familiar with relational database concepts
- Familiar with n-tier architecture concepts

If you are interested in reviewing the areas listed above, consider reviewing the following content:

- Getting Started with Visual C#
- Web Development, HTML, CSS, JavaScript, SQL, PHP, JQuery
- Relational database
- Multitier architecture

### 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
- Bootstrap
- Entity Framework Code First, LocalDB
- Request Validation
- Strongly Typed Data Controls, Model Binding, Data Annotations, and Value Providers
- OAuth and OpenID
- ASP.NET Identity, Configuration, and Authorization
- Unobtrusive Validation
- Routing
- ASP.NET Error Handling

### Application Scenarios and Tasks

Tasks demonstrated in this 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
- Creating a shopping cart
- Adding OpenID support
- Adding a payment method
- Including an administrator role and a user to the application
- Restricting access to specific pages and folder
- Uploading a file to the web application
- Implementing input validation
- Registering routes for the web application
- Implementing error handling and error logging

## 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. If you are unfamiliar with programming concepts and ASP.NET Web Forms, see the additional tutorials provided in the Web Forms Getting Started section on the ASP.NET Web site.

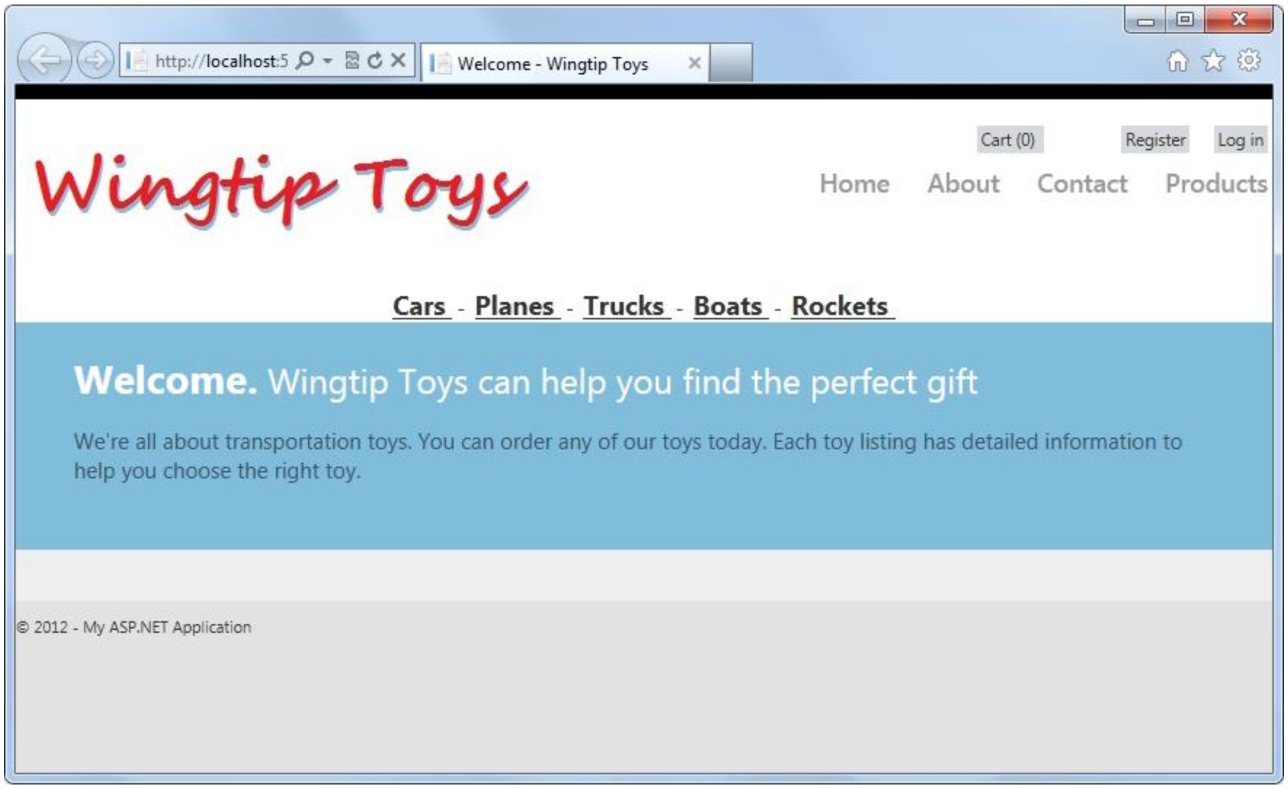
ASP.NET 4.5 Web Forms and Visual Studio 2013 features presented in this Web Forms tutorial series include the following:

- A simple UI for creating projects that offer support for multiple ASP.NET frameworks (Web Forms, MVC, and Web API).
- Bootstrap, a layout and theming framework that provides responsive design and theming capabilities.
- ASP.NET Identity, a new ASP.NET membership system that works the same in all ASP.NET frameworks and works with web hosting software other than IIS.
- Entity Framework 6, an update to the Entity Framework which allows you retrieve and manipulate data as strongly typed objects, access data asynchronous, handle transient connection faults, and log SQL statements.

For a complete list of ASP.NET 4.5 features, see ASP.NET and Web Tools for Visual Studio 2013 Release Notes.

## The Wingtip Toys Sample Application

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 Express 2013 for Web, you will see the following web Home page.



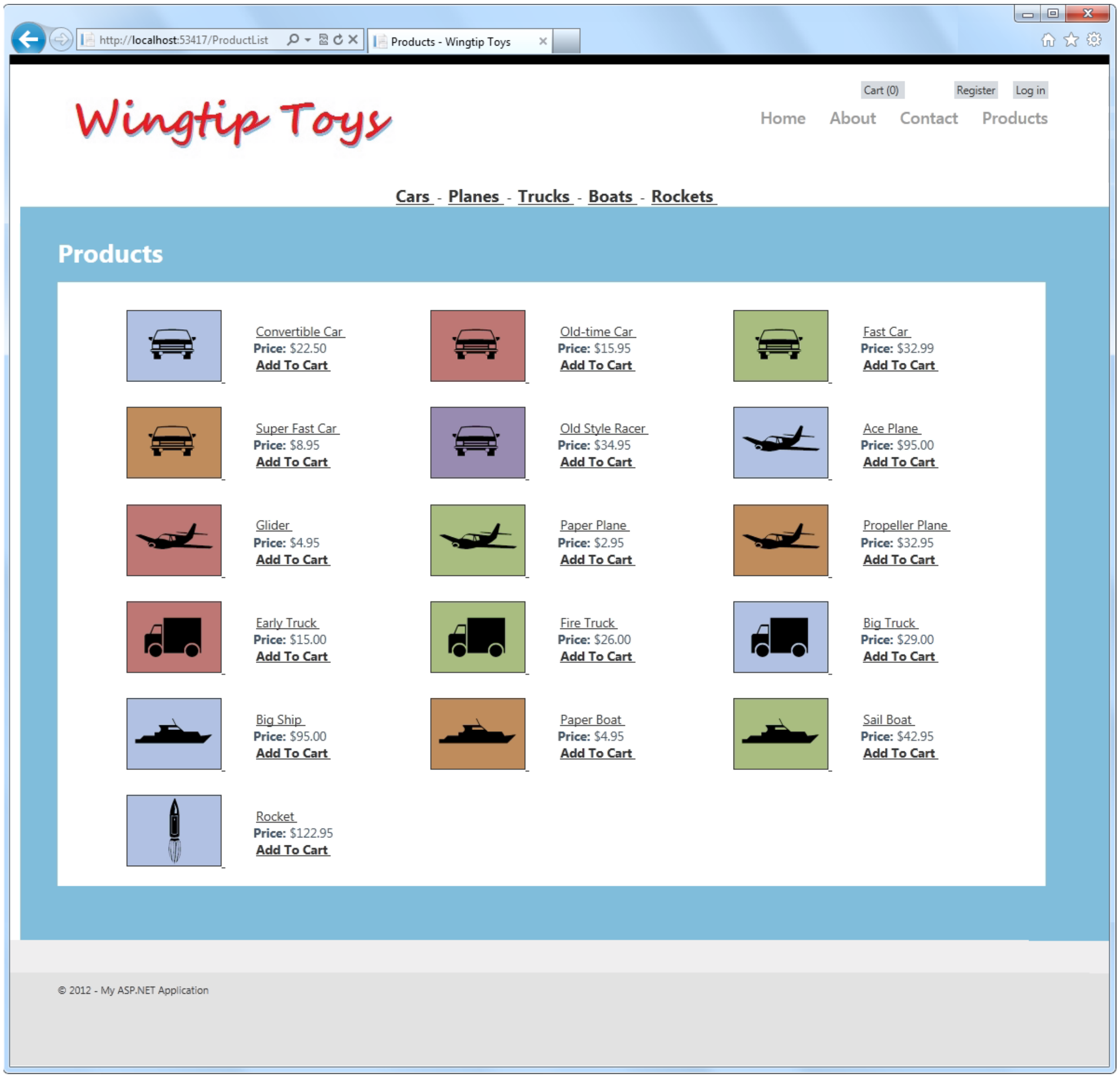
**Welcome.** Wingtip Toys can help you find the perfect gift

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.

© 2012 - My ASP.NET Application

You can register as a new user, or log in as an existing user. Navigation is provided at the top for each product category by retrieving the available products from the database.

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



Products

Convertible Car
**Price:** $22.50
Add To Cart

Old-time Car
**Price:** $15.95
Add To Cart

Fast Car
**Price:** $32.99
Add To Cart

Super Fast Car
**Price:** $8.95
Add To Cart

Old Style Racer
**Price:** $34.95
Add To Cart

Ace Plane
**Price:** $95.00
Add To Cart

Glider
**Price:** $4.95
Add To Cart

Paper Plane
**Price:** $2.95
Add To Cart

Propeller Plane
**Price:** $32.95
Add To Cart

Early Truck
**Price:** $15.00
Add To Cart

Fire Truck
**Price:** $26.00
Add To Cart

Big Truck
**Price:** $29.00
Add To Cart

Big Ship
**Price:** $95.00
Add To Cart

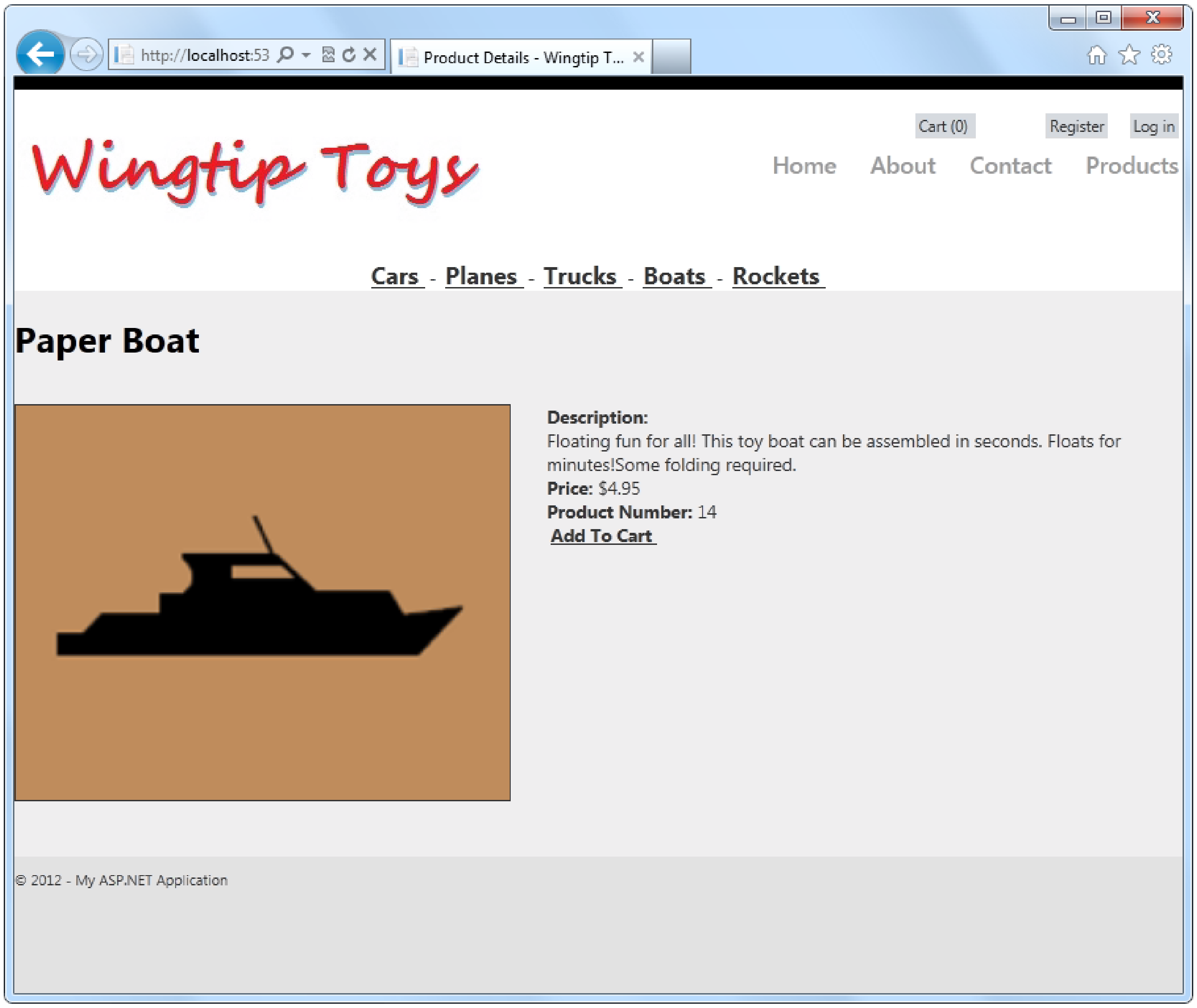
Paper Boat
**Price:** $4.95
Add To Cart

Sail Boat
**Price:** $42.95
Add To Cart

Rocket
**Price:** $122.95
Add To Cart

© 2012 - My ASP.NET Application

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

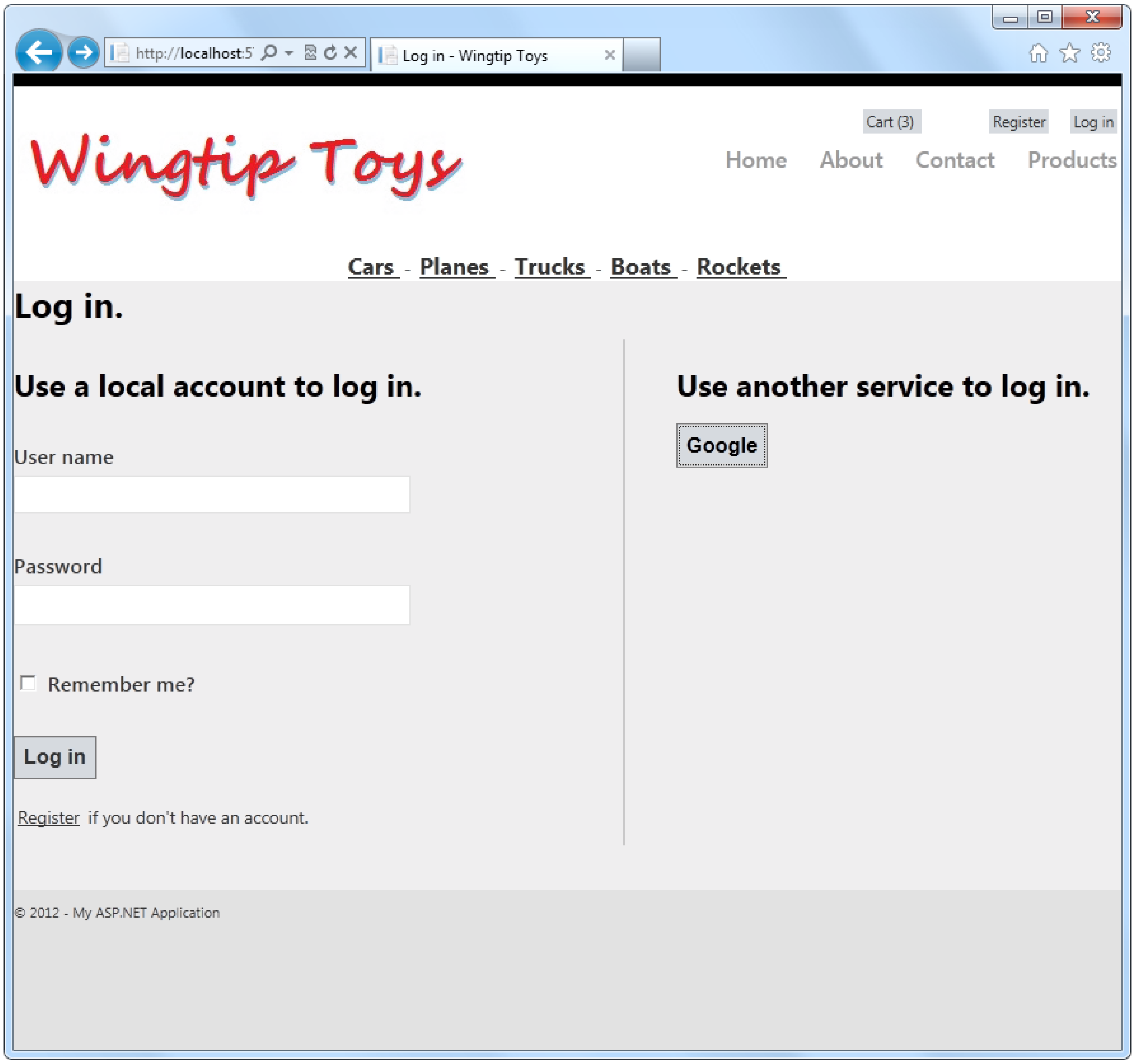


Paper Boat

**Description:**
Floating fun for all! This toy boat can be assembled in seconds. Floats for minutes!Some folding required.
**Price:** $4.95
**Product Number:** 14
Add To Cart

© 2012 - My ASP.NET Application

As a user, you can register and log in using the default functionality of the Web Forms template. This tutorial also explains how to login using an existing gmail account. Additionally, you can login as the administrator to add and remove products from the database.



Log in.

Use a local account to log in.

User name

Password

Remember me?

Log in

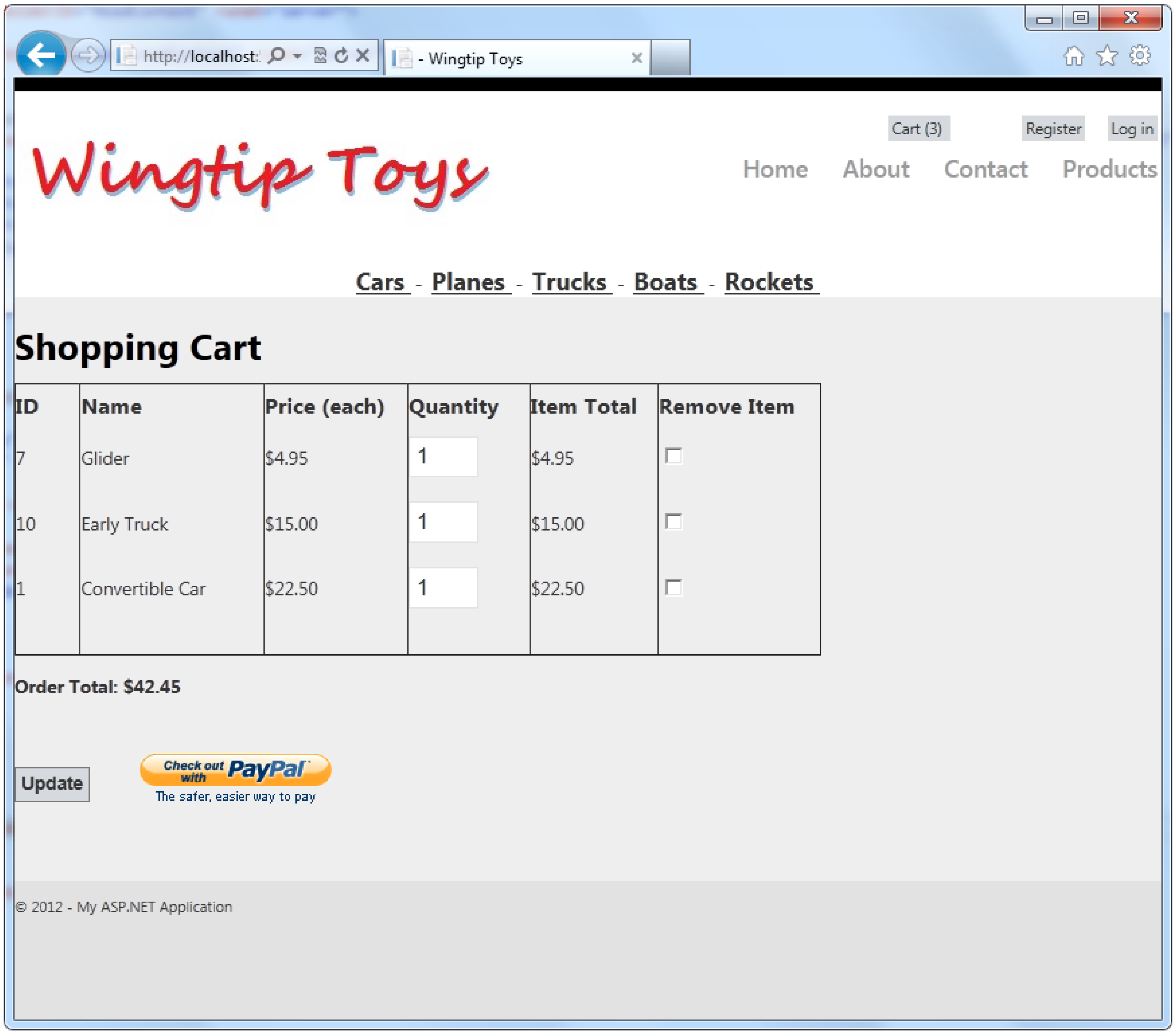
Register if you don't have an account.

Use another service to log in.

Google

© 2012 - My ASP.NET Application

Once you have logged in as a user, you can add products to the shopping cart and checkout with PayPal. Note that this sample application is designed to function with PayPal's developer sandbox. No actual money transaction will take place.

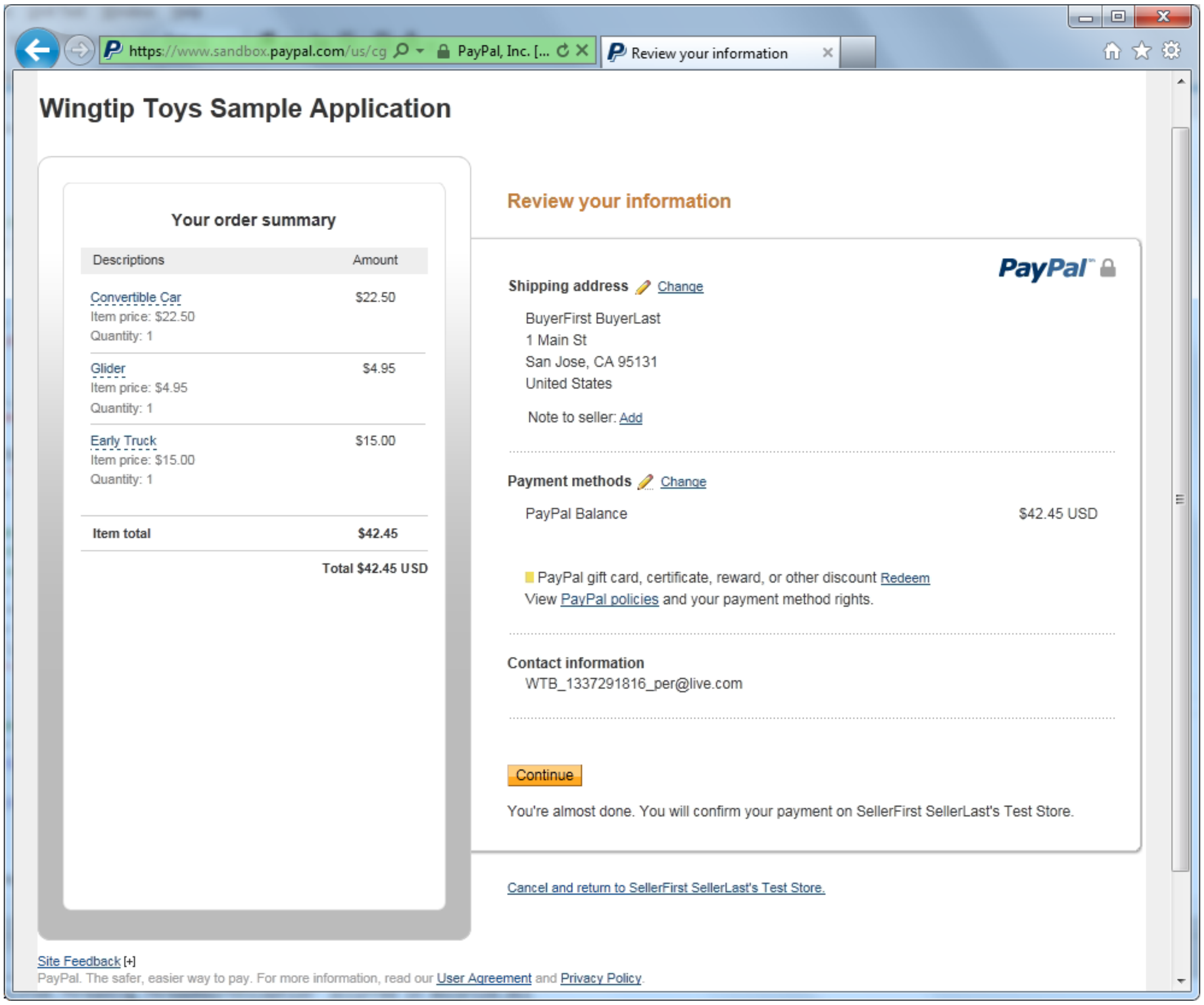


Shopping Cart

| ID | Name | Price (each) | Quantity | Item Total | Remove Item |
|---|---|---|---|---|---|
| 7 | Glider | $4.95 | 1 | $4.95 | ☐ |
| 10 | Early Truck | $15.00 | 1 | $15.00 | ☐ |
| 1 | Convertible Car | $22.50 | 1 | $22.50 | ☐ |

**Order Total: $42.45**

PayPal will confirm your account, order, and payment information.



Wingtip Toys Sample Application

Your order summary

| Descriptions | Amount |
|---|---|
| Convertible Car<br>Item price: $22.50<br>Quantity: 1 | $22.50 |
| Glider<br>Item price: $4.95<br>Quantity: 1 | $4.95 |
| Early Truck<br>Item price: $15.00<br>Quantity: 1 | $15.00 |
| Item total | $42.45 |

Total $42.45 USD

Review your information

Shipping address Change

BuyerFirst BuyerLast
1 Main St
San Jose, CA 95131
United States

Note to seller: Add

Payment methods Change

PayPal Balance $42.45 USD

PayPal gift card, certificate, reward, or other discount Redeem
View PayPal policies and your payment method rights.

Contact information
WTB_1337291816_per@live.com

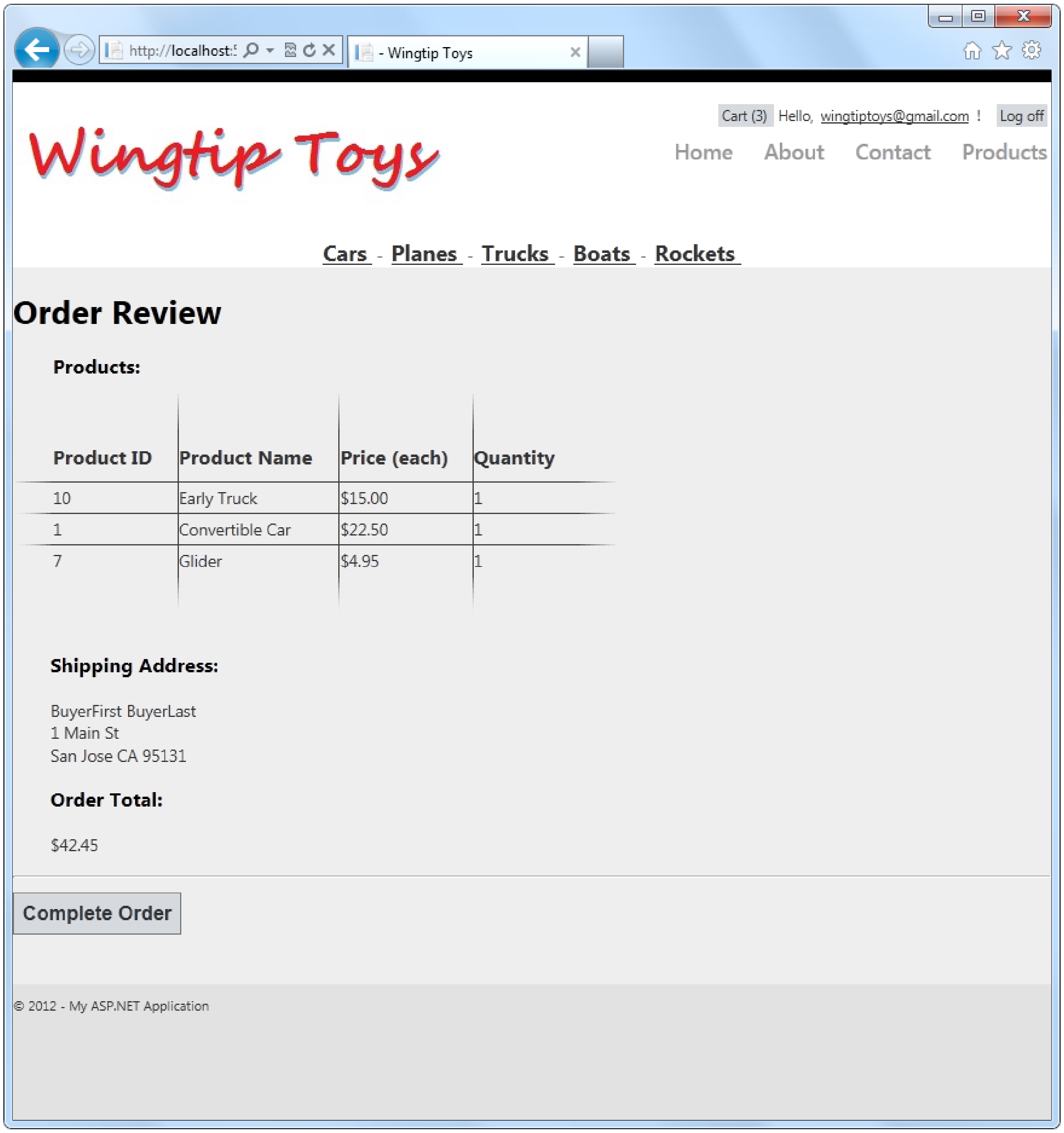
Continue

You're almost done. You will confirm your payment on SellerFirst SellerLast's Test Store.

Cancel and return to SellerFirst SellerLast's Test Store.

Site Feedback [+]
PayPal. The safer, easier way to pay. For more information, read our User Agreement and Privacy Policy.

After returning from PayPal, you can review and complete your order.


http://localhost:
- Wingtip Toys
Cart (3)
Hello, wingtiptoys@gmail.com !
Log off
Wingtip Toys
Home
About
Contact
Products
Cars - Planes - Trucks - Boats - Rockets
Complete Order
© 2012 - My ASP.NET Application

Order Review

**Products:**

| Product ID | Product Name | Price (each) | Quantity |
|---|---|---|---|
| 10 | Early Truck | $15.00 | 1 |
| 1 | Convertible Car | $22.50 | 1 |
| 7 | Glider | $4.95 | 1 |

**Shipping Address:**

BuyerFirst BuyerLast
1 Main St
San Jose CA 95131

**Order Total:**

$42.45

## Prerequisites

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

- Microsoft Visual Studio 2013 or Microsoft Visual Studio Express 2013 for Web. The .NET Framework is installed automatically.

This tutorial series uses Microsoft Visual Studio Express 2013 for Web. You can use either Microsoft Visual Studio Express 2013 for Web or Microsoft Visual Studio 2013 to complete this tutorial series.

Note

Microsoft Visual Studio 2013 and Microsoft Visual Studio Express 2013 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 2013 or Microsoft Visual Studio Express 2013 for Web next to the existing version. Sites that you created in earlier versions can be opened in Visual Studio 2013 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 **optionally** 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 to create the sample application in the *WingtipToys-Assets* folder in the *WingtipToys* folder.

### Download the file from MSDN Samples site:

Getting Started with ASP.NET 4.5 Web Forms and Visual Studio 2013 - Wingtip Toys (C#)

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 2013 projects. By default, the Visual Studio 2013 projects folder is the following:

```
C:\Users\<username>\Documents\Visual Studio 2013\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 and double-click the *WingtipToys.sln* file. Visual Studio 2013 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 4.5 Web Forms and Visual Studio 2013 - Wingtip Toys (C#) 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 4.5 Web Forms and Visual Studio 2013 - Wingtip Toys (C#) 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.

## Create the Project

This tutorial series will teach you the basics of building an ASP.NET Web Forms application using ASP.NET 4.5 and Microsoft Visual Studio Express 2013 for Web. A Visual Studio 2013 project with C# source code is available to accompany this tutorial series.

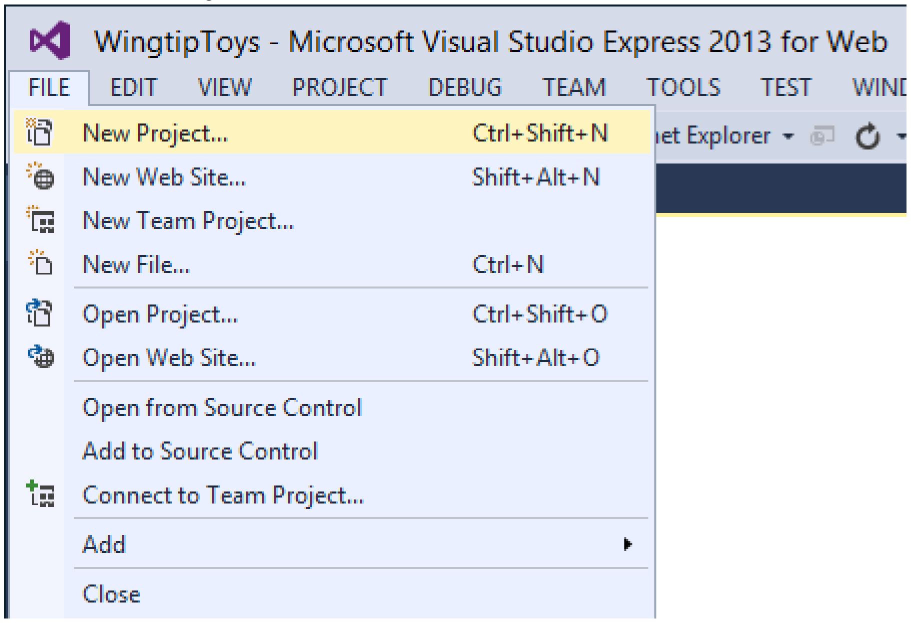
In this tutorial you will create, review, and run the default project in Visual Studio, which will allow you to become familiar with features of ASP.NET. Also, you will review the Visual Studio environment.

## 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.

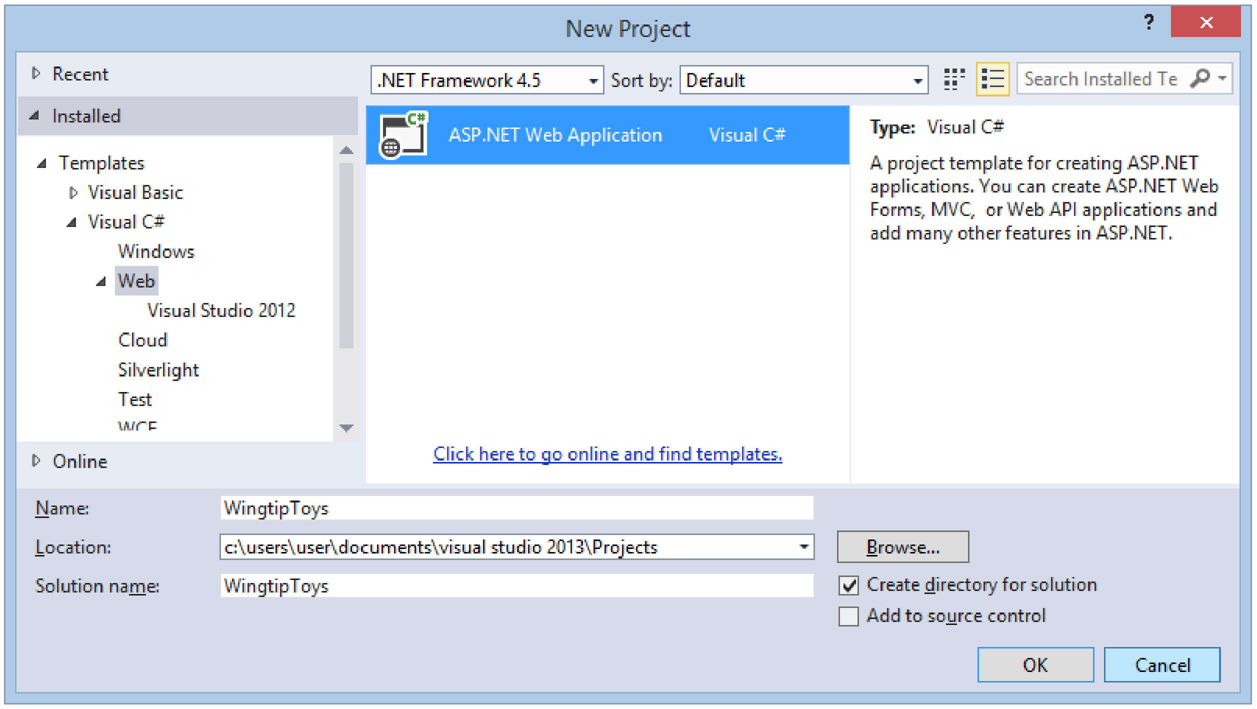
## Creating the Project

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


WingtipToys - Microsoft Visual Studio Express 2013 for Web
FILE EDIT VIEW PROJECT DEBUG TEAM TOOLS TEST WIND
New Project... Ctrl+Shift+N
New Web Site... Shift+Alt+N
New Team Project...
New File... Ctrl+N
Open Project... Ctrl+Shift+O
Open Web Site... Shift+Alt+O
Open from Source Control
Add to Source Control
Connect to Team Project...
Add
Close
et Explorer

3. Select the **Templates** -> **Visual C#** -> **Web** templates group on the left.
4. Choose the **ASP.NET Web Application** template in the center column.

5. Name your project *WingtipToys* and choose the **OK** button.



.NET Framework 4.5 Sort by: Default

ASP.NET Web Application Visual C#

**Type:** Visual C#

A project template for creating ASP.NET applications. You can create ASP.NET Web Forms, MVC, or Web API applications and add many other features in ASP.NET.

Click here to go online and find templates.

Name: WingtipToys

Location: c:\users\user\documents\visual studio 2013\Projects

Solution name: WingtipToys

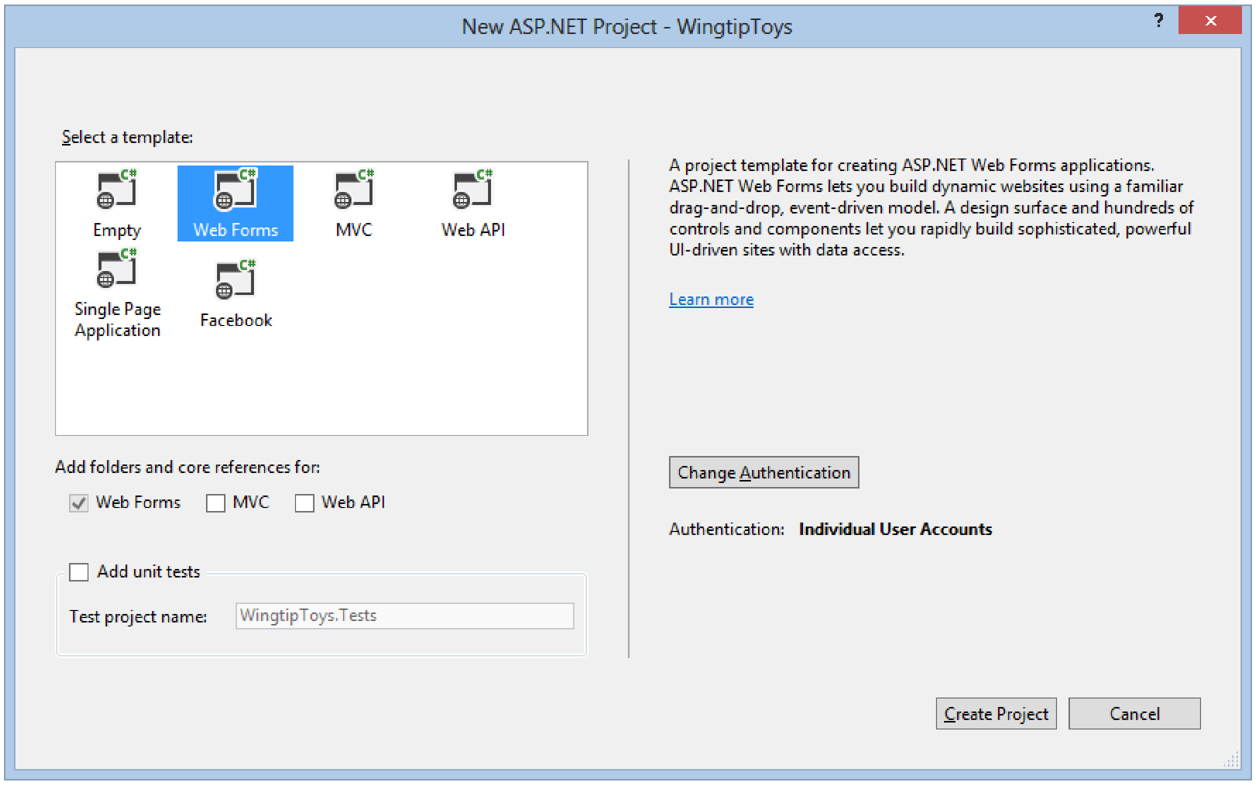
☑ Create directory for solution

☐ Add to source control

**Note**

The name of the project in this tutorial series is **WingtipToys**. It is recommended that you use this exact project name so that the code provided throughout this tutorial series functions as expected.

6. Next, select the **Web Forms** template and chooks the **Create Project** button.


New ASP.NET Project - WingtipToys

Select a template:

A project template for creating ASP.NET Web Forms applications. 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.

Learn more

Add folders and core references for:

[✓] Web Forms [ ] MVC [ ] Web API

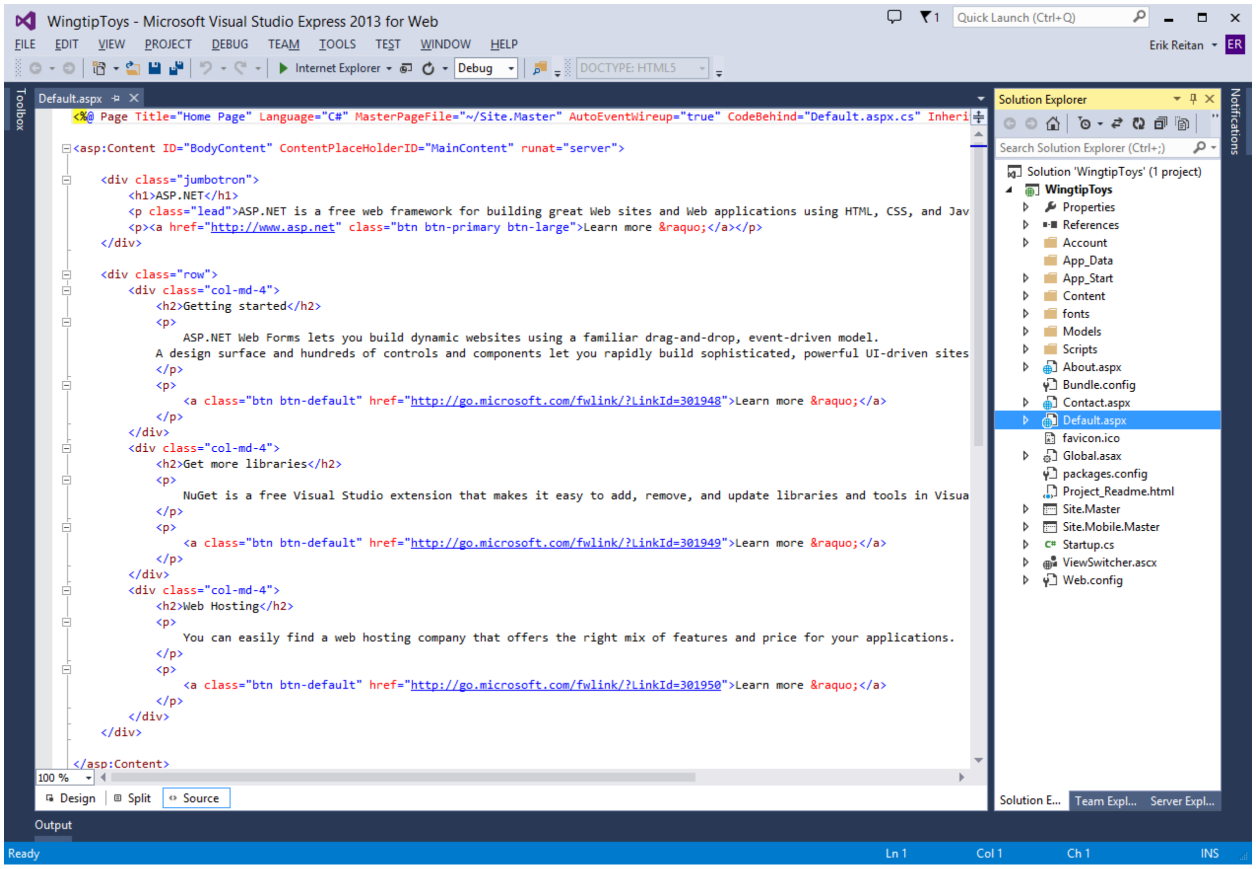
[ ] Add unit tests

Test project name: WingtipToys.Tests

Change Authentication

Authentication: **Individual User Accounts**

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



```
<%@ Page Title="Home Page" Language="C#" MasterPageFile="~/Site.Master" AutoEventWireup="true" CodeBehind="Default.aspx.cs" Inheri

<asp:Content ID="BodyContent" ContentPlaceHolderID="MainContent" runat="server">

    <div class="jumbotron">
        <h1>ASP.NET</h1>
        <p class="lead">ASP.NET is a free web framework for building great Web sites and Web applications using HTML, CSS, and Jav
        <p><a href="http://www.asp.net" class="btn btn-primary btn-large">Learn more &raquo;</a></p>
    </div>

    <div class="row">
        <div class="col-md-4">
            <h2>Getting started</h2>
            <p>
                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
            </p>
            <p>
                <a class="btn btn-default" href="http://go.microsoft.com/fwlink/?LinkId=301948">Learn more &raquo;</a>
            </p>
        </div>
        <div class="col-md-4">
            <h2>Get more libraries</h2>
            <p>
                NuGet is a free Visual Studio extension that makes it easy to add, remove, and update libraries and tools in Visua
            </p>
            <p>
                <a class="btn btn-default" href="http://go.microsoft.com/fwlink/?LinkId=301949">Learn more &raquo;</a>
            </p>
        </div>
        <div class="col-md-4">
            <h2>Web Hosting</h2>
            <p>
                You can easily find a web hosting company that offers the right mix of features and price for your applications.
            </p>
            <p>
                <a class="btn btn-default" href="http://go.microsoft.com/fwlink/?LinkId=301950">Learn more &raquo;</a>
            </p>
        </div>
    </div>

</asp:Content>
```

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 ASP.NET Frameworks

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 Toy Store 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 four primary 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 are 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.

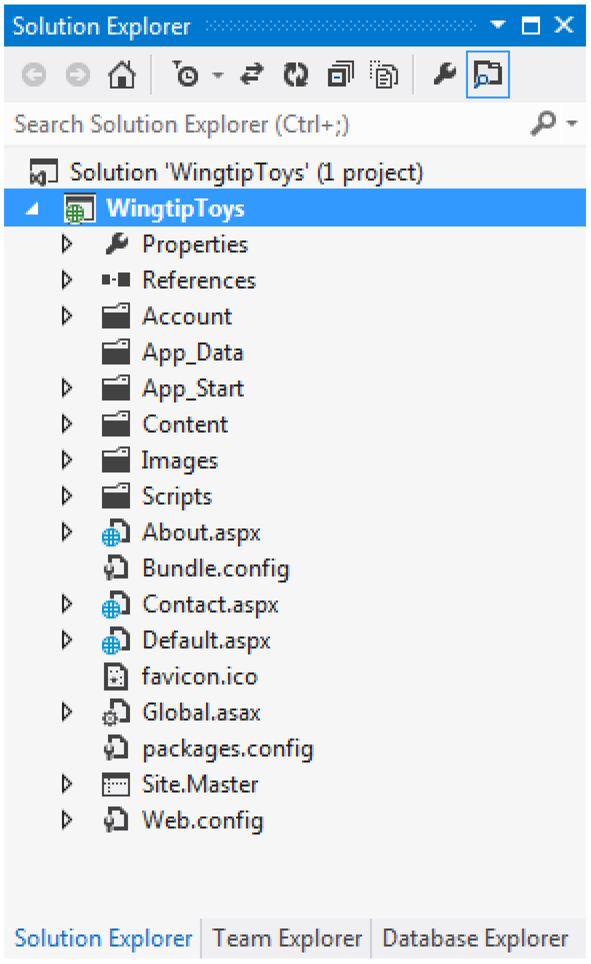
- ASP.NET MVC
  ASP.NET MVC targets developers who are interested in patterns and principles like test-driven development, separation of concerns, inversion of control (IoC), and dependency injection (DI). This framework encourages separating the business logic layer of a web application from its presentation 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.
- ASP.NET Single Page Application
  ASP.NET Single Page Application (SPA) helps you build applications that include significant client-side interactions using HTML 5, CSS 3 and JavaScript. The ASP.NET and Web Tools 2012.2 Update ships a new template for building single page applications using knockout.js and ASP.NET Web API. In addition to the new SPA template, new community-created SPA templates are also available for download.

In addition to the four main development frameworks, ASP.NET also offers additional technologies that are important to be aware of and familiar with, but are not covered in this tutorial series:

- ASP.NET Web API – A framework for building HTTP services that reach a broad range of clients, including browsers and mobile devices.
- ASP.NET SignalR - A library that makes developing real-time web functionality easy.

## 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:

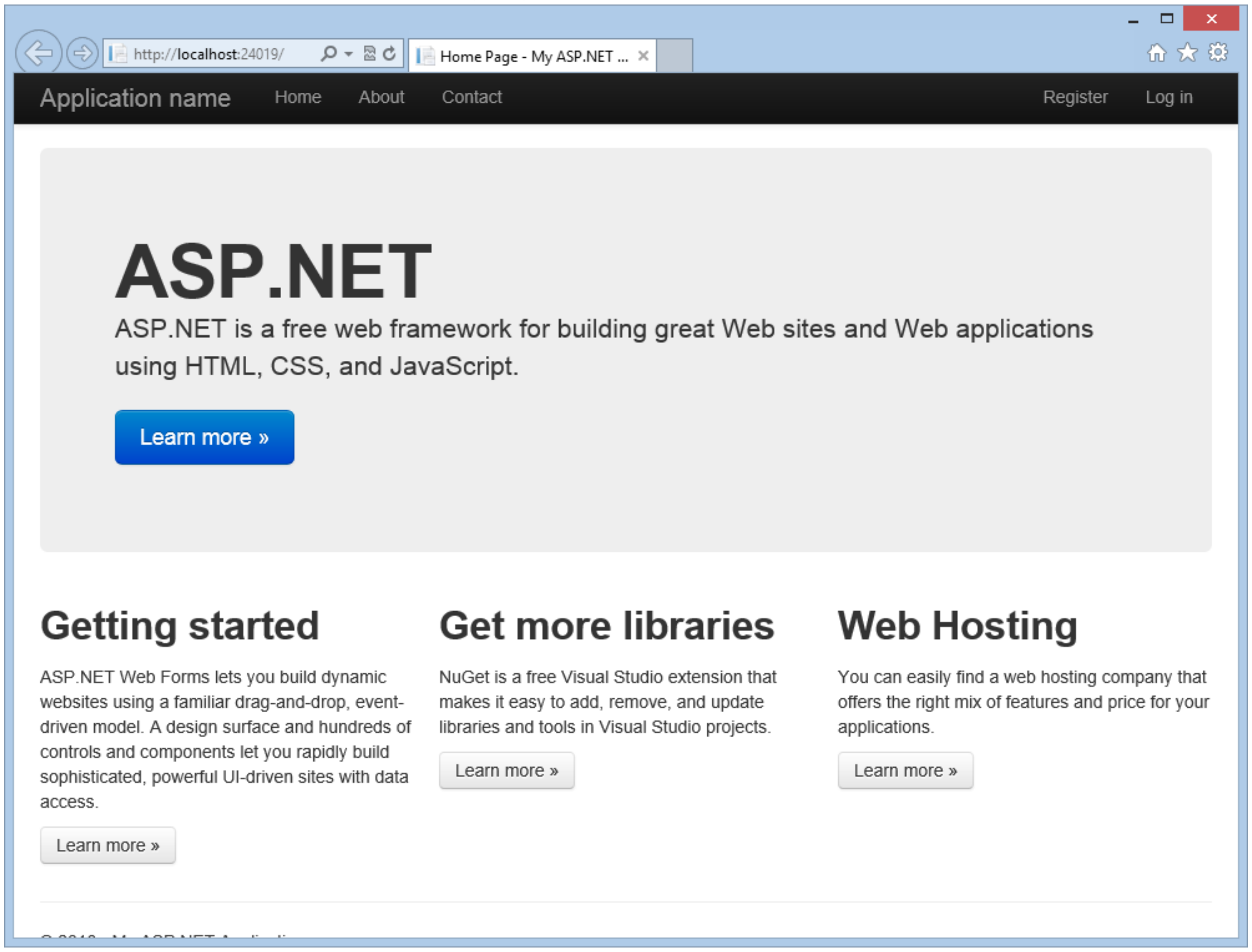


Visual Studio creates some initial folders and files for your project. The first files that you will be working with later in this tutorial are the following:

| **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 ***F5*** key while in Visual Studio.
   The application will build and display in your Web browser.
   

   ASP.NET

   ASP.NET is a free web framework for building great Web sites and Web applications using HTML, CSS, and JavaScript.

   Learn more »

   Getting started

   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.

   Learn more »

   Get more libraries

   NuGet is a free Visual Studio extension that makes it easy to add, remove, and update libraries and tools in Visual Studio projects.

   Learn more »

   Web Hosting

   You can easily find a web hosting company that offers the right mix of features and price for your applications.

   Learn more »
2. Once you have completed review the running application, close the browser window.

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 ASP.NET page life cycle so that you can write code at the appropriate life-cycle stage for the effect you intend.

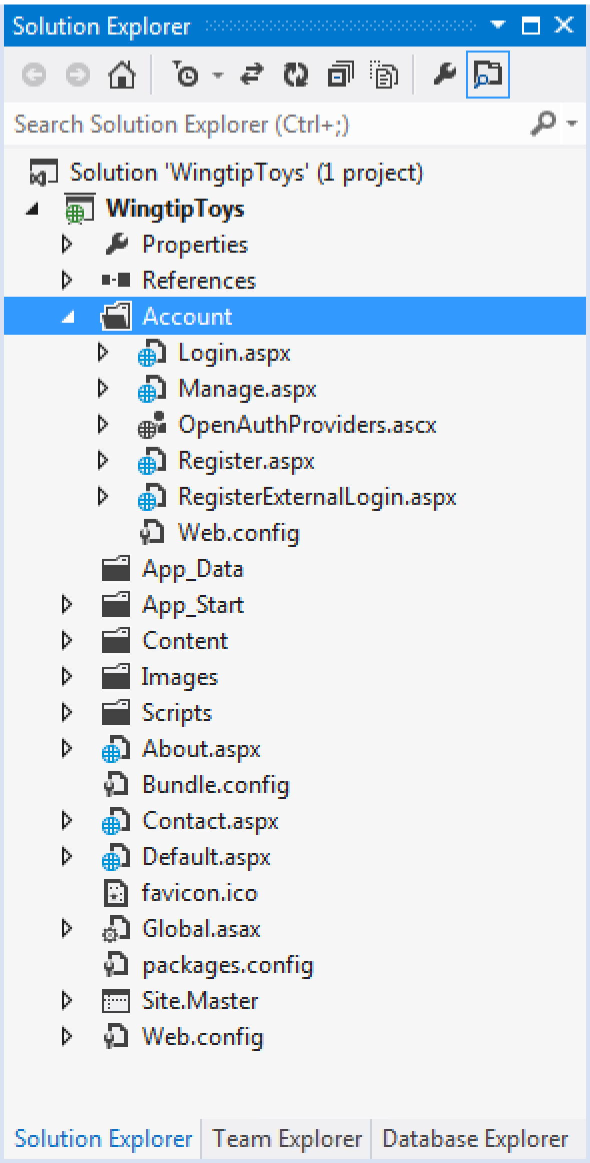
When a Web server receives a request for a page, it finds the page, processes it, sends it to the browser, and then discards all page information. If the user requests the same page again, the server repeats the entire sequence, reprocessing the page from scratch. Put another way, a server has no memory of pages that it has processed—pages 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 Identity 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. Additionally, ASP.NET Web Forms supports OAuth and OpenID. These authentication enhancements allow users to log into your site using existing credentials, from such accounts as Facebook, Twitter, Windows Live, and Google.



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 Express 2013 for Web.

### 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 SQL Azure 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 supports 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.

### Modernizr

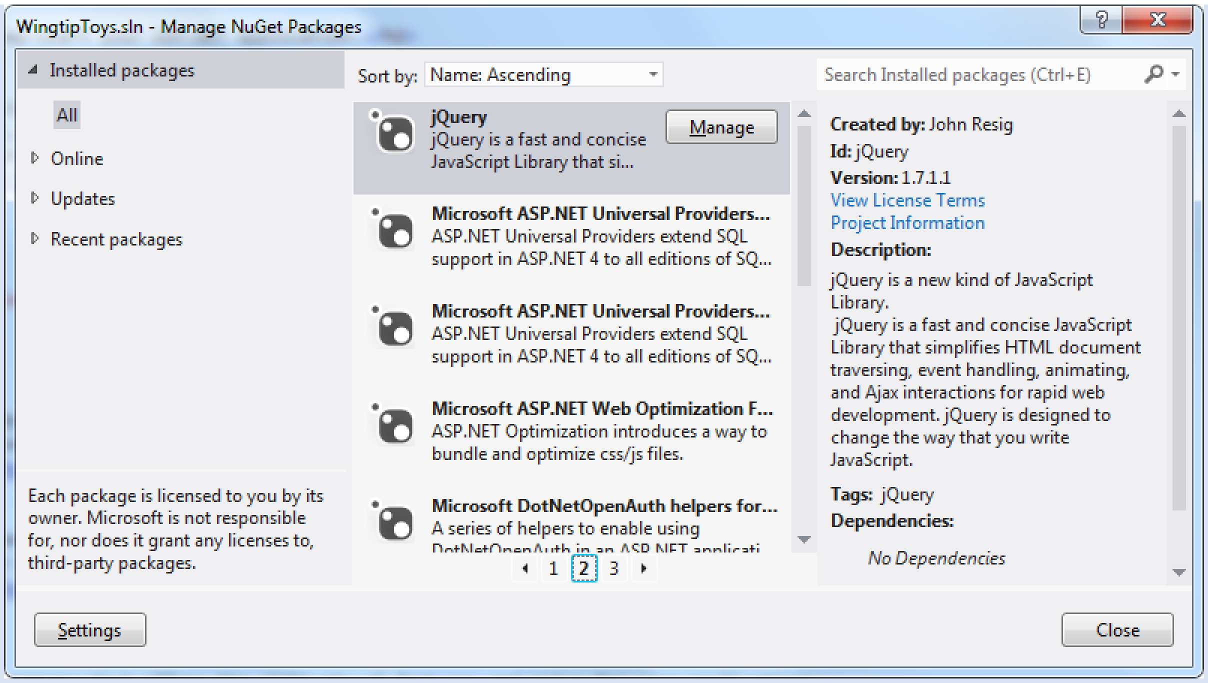
For browsers that do not support HTML5, you can use Modernizr. Modernizr is an open-source JavaScript library that can detect whether a browser supports HTML5 features, and enable them if it does not. In the ASP.NET Web Forms Application template, Modernizr is installed as a NuGet package.

### Bootstrap

The Visual Studio 2013 project templates use Bootstrap, a layout and theming framework created by Twitter. Bootstrap uses CSS3 to provide responsive design, which means layouts can dynamically adapt to different browser window sizes. You can also use Bootstrap's theming feature to easily effect a change in the application's look and feel. By default, the ASP.NET Web Application template in Visual Studio 2013 includes Bootstrap 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.



jQuery
jQuery is a fast and concise JavaScript Library that si...

Microsoft ASP.NET Universal Providers...
ASP.NET Universal Providers extend SQL support in ASP.NET 4 to all editions of SQ...

Microsoft ASP.NET Universal Providers...
ASP.NET Universal Providers extend SQL support in ASP.NET 4 to all editions of SQ...

Microsoft ASP.NET Web Optimization F...
ASP.NET Optimization introduces a way to bundle and optimize css/js files.

Microsoft DotNetOpenAuth helpers for...
A series of helpers to enable using DotNetOpenAuth in an ASP.NET applicati

**Created by:** John Resig
**Id:** jQuery
**Version:** 1.7.1.1
View License Terms
Project Information
**Description:**
jQuery is a new kind of JavaScript Library.
jQuery is a fast and concise JavaScript Library that simplifies HTML document traversing, event handling, animating, and Ajax interactions for rapid web development. jQuery is designed to change the way that you write JavaScript.
**Tags:** jQuery
**Dependencies:**
*No Dependencies*

Each package is licensed to you by its owner. Microsoft is not responsible for, nor does it grant any licenses to, third-party packages.

When you install a package, NuGet copies files to your solution and automatically makes whatever changes are needed, such as adding references and changing you're the configuration associated with your Web application. 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.

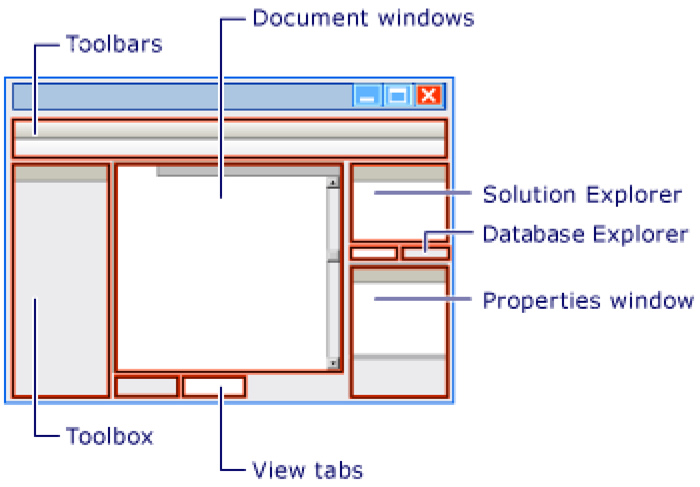
### Unobtrusive Validation
Built-in validator controls have been configured to use unobtrusive JavaScript for client-side validation logic. This significantly reduces the amount of JavaScript rendered inline in the page markup and reduces the overall page size. Unobtrusive validation is added globally 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.

### 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.

For additional information about the installed libraries and packages included with the ASP.NET Web Forms template, see the list of installed NuGet packages. To do this, In Visual Studio create a new Web Forms project, select **Tools** -> **Library Package Manager** -> **Manage NuGet Packages for Solution**, and select **Installed packages** in the **Manage NuGet Packages** dialog box.

### 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 Visual Guide to Visual Web Developer.

## Summary

In this tutorial you have created, reviewed and run 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 series will teach you the basics of building an ASP.NET Web Forms application using ASP.NET 4.5 and Microsoft Visual Studio Express 2013 for Web. A Visual Studio 2013 project with C# source code is available to accompany this tutorial series.

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 will have built a group of data-access classes that are in the *Models folder of the project*.

## 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

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 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 Entity Framework 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. A class is a construct that enables you to create your own custom types by grouping together variables of other types, methods and events. You can map 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 Entity Framework create the database on the fly from these new classes.

You will 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.

## Entity Framework and References

By default, Entity Framework is included when you create a new **ASP.NET Web Application** using the **Web Forms** template. Entity Framework can be installed, uninstalled, and updated as a NuGet package.

This NuGet package includes the following **runtime** assemblies within your project:

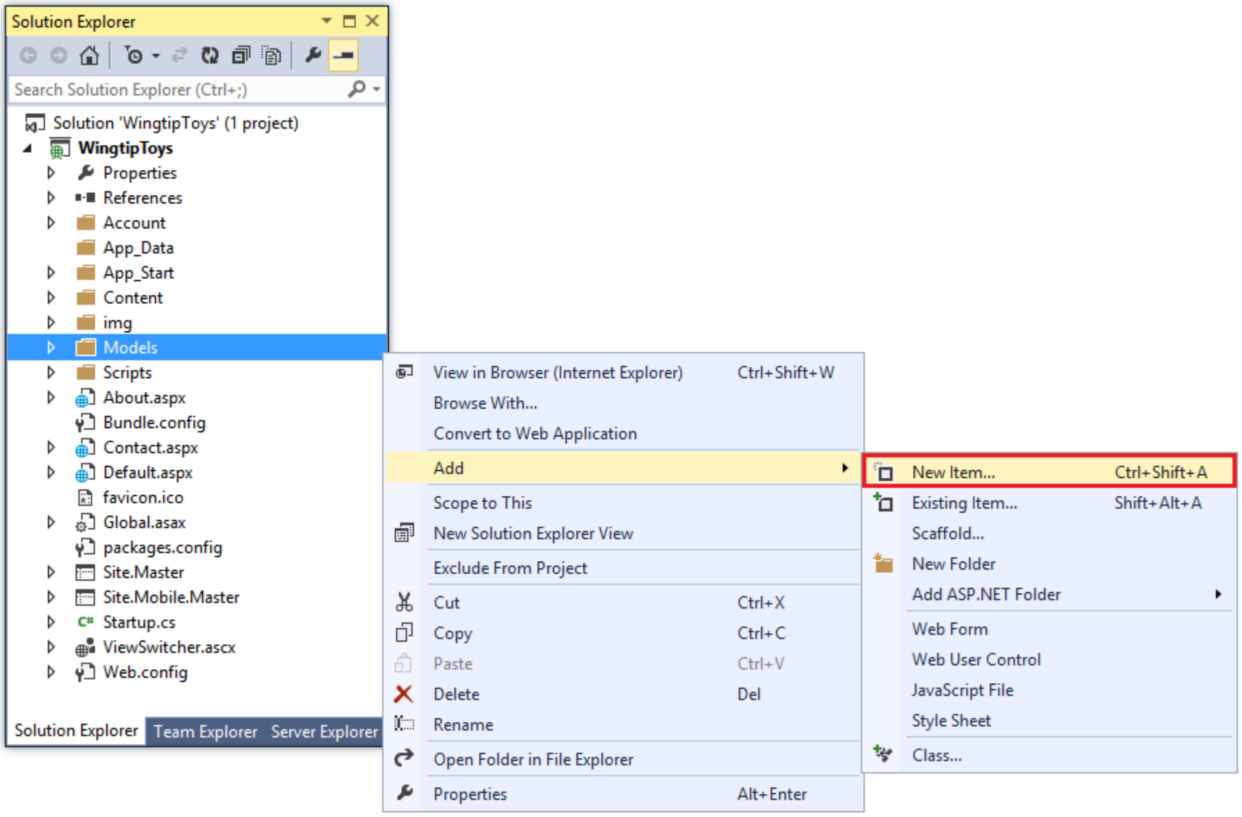
- EntityFramework.dll – All the common runtime code used by Entity Framework
- EntityFramework.SqlServer.dll – The Microsoft SQL Server provider for Entity Framework

## Entity Classes

The classes you create to define the schema of the data are called entity classes. If you're new to database design, think of the entity classes as table definitions of a database. 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.

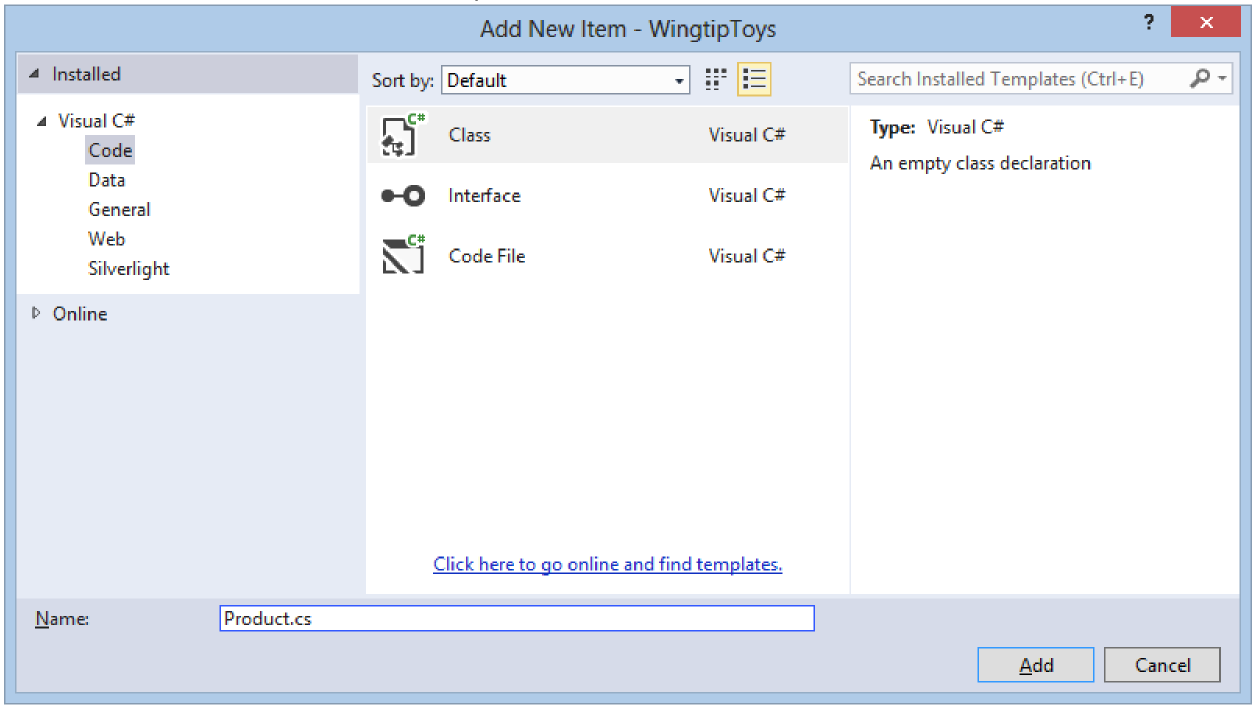
In this tutorial, you'll start out by adding simple entity classes representing the schemas for products and categories. The products class will contain definitions for each product. The name of each of the members of the product class will be `ProductID`, `ProductName`, `Description`, `ImagePath`, `UnitPrice`, `CategoryID`, and `Category`. The category class will contain definitions for each category that a product can belong to, such as Car, Boat, or Plane. The name of each of the members of the category class will be `CategoryID`, `CategoryName`, `Description`, and `Products`. Each product will belong to one of the categories. These entity classes will be added to the project's existing *Models* folder.

1. In **Solution Explorer**, right-click the *Models* folder and then select **Add** -> **New Item**.



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

2. Under **Visual C#** from the **Installed** pane on the left, select **Code**.



3. Select **Class** from the middle pane and name this new class *Product.cs*.
4. Click **Add**.
   The new class file is displayed in the editor.
5. 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; }
  }
}
```

6. Create another class by repeating steps 1 through 4, however, name the new class *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; }
  }
}
```

As previously mentioned, 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 individual products (toys) in the database. Each instance of a `Product` object will correspond to a row within a relational database table, and each property of the

Product class will map to a column in the relational database 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 `[ScaffoldColumn(false)]`. 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 when the database is created.

## 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 (such as the `Product` class and the `Category` class) and provides data access to the database.

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

1. Right-click the *Models* folder and then select **Add** -> **New Item**.
   The **Add New Item** dialog box is displayed.
2. Select **Class** from the middle pane, name it *ProductContext.cs* and click **Add**.
3. 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 Entity Framework, which includes the capability to query, insert, update, and delete data by working with strongly typed objects.

The `ProductContext` class represents 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 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` in the *Models* folder and name it *ProductDatabaseInitializer.cs*.
2. Replace the default code contained in the class with the following code:

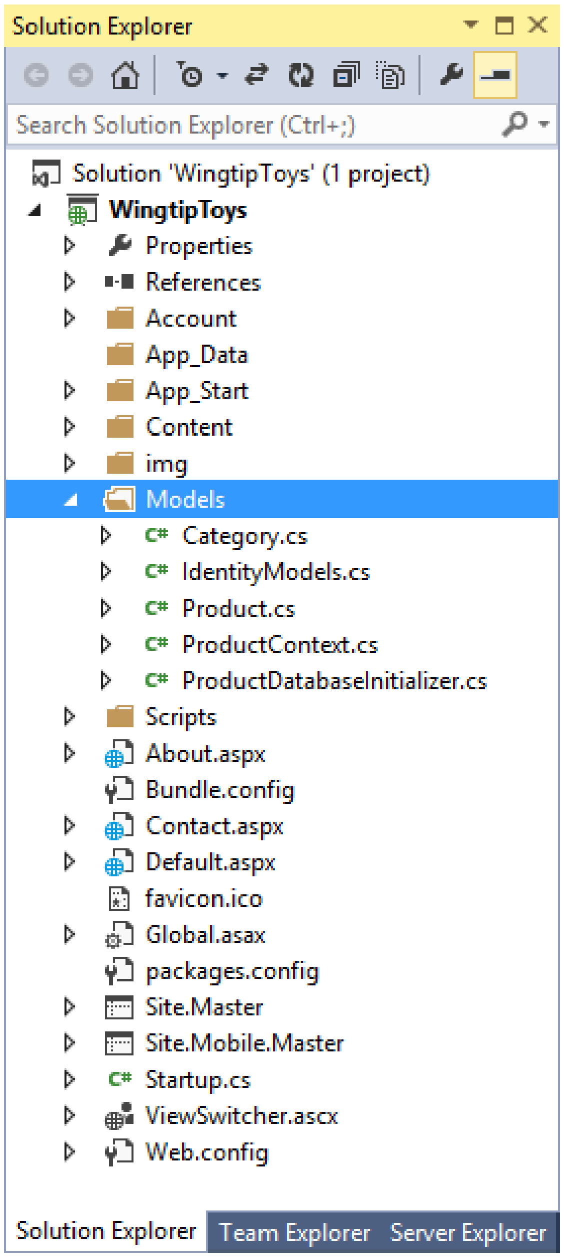
```
using System.Collections.Generic;
using System.Data.Entity;

namespace WingtipToys.Models
{
  public class ProductDatabaseInitializer : 
DropCreateDatabaseAlways<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"
                },
                new Category
                {
                    CategoryID = 2,
                    CategoryName = "Planes"
                },
                new Category
                {
                    CategoryID = 3,
                    CategoryName = "Trucks"
                },
                new Category
                {
                    CategoryID = 4,
                    CategoryName = "Boats"
                },
                new Category
                {
                    CategoryID = 5,
                    CategoryName = "Rockets"
                },
            };

      return categories;
    }

    private static List<Product> GetProducts()
    {
      var products = new List<Product> {
                new Product
                {
                    ProductID = 1,
                    ProductName = "Convertible Car",
                    Description = "This convertible car is fast! The engine is 
powered by a neutrino based battery (not included)." + 
                                  "Power it up and let it go!", 
                    ImagePath="carconvert.png",
                    UnitPrice = 22.50,
                    CategoryID = 1
                },
                new Product
                {
                    ProductID = 2,
                    ProductName = "Old-time Car",
                    Description = "There's nothing old about this toy car,
except it's looks. Compatible with other old toy cars.",
                    ImagePath="carearly.png",
                    UnitPrice = 15.95,
                     CategoryID = 1
                },
                new Product
                {
                    ProductID = 3,
                    ProductName = "Fast Car",
                    Description = "Yes this car is fast, but it also floats in
water.",
                    ImagePath="carfast.png",
                    UnitPrice = 32.99,
                    CategoryID = 1
                },
                new Product
                {
                    ProductID = 4,
                    ProductName = "Super Fast Car",
                    Description = "Use this super fast car to entertain guests.
Lights and doors work!",
                    ImagePath="carfaster.png",
                    UnitPrice = 8.95,
                    CategoryID = 1
                },
                new Product
                {
                    ProductID = 5,
                    ProductName = "Old Style Racer",
                    Description = "This old style racer can fly (with user
assistance). Gravity controls flight duration." +
                                  "No batteries required.",
                    ImagePath="carracer.png",
                    UnitPrice = 34.95,
                    CategoryID = 1
                },
                new Product
                {
                    ProductID = 6,
                    ProductName = "Ace Plane",
                    Description = "Authentic airplane toy. Features realistic
color and details.",
                    ImagePath="planeace.png",
                    UnitPrice = 95.00,
                    CategoryID = 2
                },
                new Product
                {
                    ProductID = 7,
                    ProductName = "Glider",
                    Description = "This fun glider is made from real balsa
wood. Some assembly required.",
                    ImagePath="planeglider.png",
                    UnitPrice = 4.95,
                    CategoryID = 2
                },
                new Product
                {
                    ProductID = 8,
                    ProductName = "Paper Plane",
                    Description = "This paper plane is like no other paper
plane. Some folding required.",
                    ImagePath="planepaper.png",
                    UnitPrice = 2.95,
                    CategoryID = 2
                },
                new Product
                {
                    ProductID = 9,
                    ProductName = "Propeller Plane",
                    Description = "Rubber band powered plane features two
wheels.",
                    ImagePath="planeprop.png",
                    UnitPrice = 32.95,
                    CategoryID = 2
                },
                new Product
                {
                    ProductID = 10,
                    ProductName = "Early Truck",
                    Description = "This toy truck has a real gas powered
engine. Requires regular tune ups.",
                    ImagePath="truckearly.png",
                    UnitPrice = 15.00,
                    CategoryID = 3
                },
                new Product
                {
                    ProductID = 11,
                    ProductName = "Fire Truck",
                    Description = "You will have endless fun with this one
quarter sized fire truck.",
                    ImagePath="truckfire.png",
                    UnitPrice = 26.00,
                    CategoryID = 3
                },
                new Product
                {
                    ProductID = 12,
                    ProductName = "Big Truck",
                    Description = "This fun toy truck can be used to tow other
trucks that are not as big.",
                    ImagePath="truckbig.png",
                    UnitPrice = 29.00,
                    CategoryID = 3
                },
                new Product
                {
                    ProductID = 13,
                    ProductName = "Big Ship",
                    Description = "Is it a boat or a ship. Let this floating
vehicle decide by using its " +
                                  "artifically intelligent computer brain!",
                    ImagePath="boatbig.png",
                    UnitPrice = 95.00,
                    CategoryID = 4
                },
                new Product
                {
                    ProductID = 14,
                    ProductName = "Paper Boat",
                    Description = "Floating fun for all! This toy boat can be 
assembled in seconds. Floats for minutes!" +
                                  "Some folding required.",
                    ImagePath="boatpaper.png",
                    UnitPrice = 4.95,
                    CategoryID = 4
                },
                new Product
                {
                    ProductID = 15,
                    ProductName = "Sail Boat",
                    Description = "Put this fun toy sail boat in the water and 
let it go!",
                    ImagePath="boatsail.png",
                    UnitPrice = 42.95,
                    CategoryID = 4
                },
                new Product
                {
                    ProductID = 16,
                    ProductName = "Rocket",
                    Description = "This fun rocket will travel up to a height 
of 200 feet.",
                    ImagePath="rocket.png",
                    UnitPrice = 122.95,
                    CategoryID = 5
                }
            };

      return products;
    }
  }
}
```

As you can see from the above code, 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.

**Note**

If you wanted the database to be recreated every time you ran the application, you could use the `DropCreateDatabaseAlways` class instead of the `DropCreateDatabaseIfModelChanges` class. However for this tutorial series, use the `DropCreateDatabaseIfModelChanges` class.

At this point in this tutorial, you will have a *Models* folder with four new classes and one default class:



## Configuring the Application to Use the Data Model

Now that you've created the classes that represent the data, you must 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. The *Global.asax* file can be used to handle application events or methods. The *Web.config* file allows you to control the configuration of your ASP.NET web application.

### Updating the Global.asax file

To initialize the data models when the application starts, you will update the `Application_Start` handler in the *Global.asax.cs* file.

**Note**

In Solution Explorer, you can select either the *Global.asax* file or the *Global.asax.cs* file to edit the *Global.asax.cs* file.

1. 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.Optimization;
using System.Web.Routing;
using System.Web.Security;
using System.Web.SessionState;
using System.Data.Entity;
using WingtipToys.Models;

namespace WingtipToys
{
    public class Global : HttpApplication
    {
        void Application_Start(object sender, EventArgs e)
        {
            // Code that runs on application startup
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);

            // Initialize the product database.
            Database.SetInitializer(new ProductDatabaseInitializer());
        }
    }
}
```

**Note**

Your browser **must** support HTML5 to view the code highlighted in yellow when viewing this tutorial series in a browser.

As shown in the above code, when the application starts, the application specifies the initializer that will run during the first time the data is accessed. The two additional namespaces are required to access the `Database` object and the `ProductDatabaseInitializer` object.

## 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. Making this change will allow you to find and inspect the database file later in this tutorial.

1. In **Solution Explorer**, find and open the *Web.config* file.
2. Add the following connection string highlighted in yellow to the `<connectionStrings>` section of the *Web.config* file as follows:

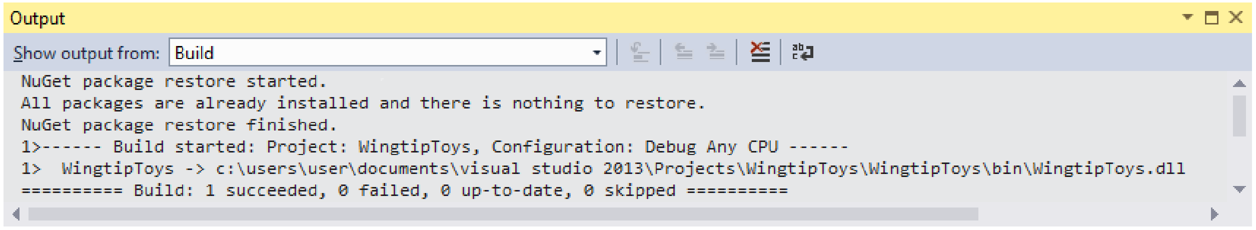
```
  <connectionStrings>
    <add name="DefaultConnection" connectionString="Data
Source=(LocalDb)\v11.0;AttachDbFilename=|DataDirectory|\aspnet-WingtipToys-
20131119102907.mdf;Initial Catalog=aspnet-WingtipToys-20131119102907;Integrated
Security=True"
      providerName="System.Data.SqlClient" />
    <add name="WingtipToys"
         connectionString="Data
Source=(LocalDB)\v11.0;AttachDbFilename=|DataDirectory|\wingtiptoys.mdf;Integra
ted 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 **Debug** menu, select **Build WingtipToys**.
   The **Output** window is displayed, and if all went well, you see a *succeeded* message.



Output

Show output from: Build

```
NuGet package restore started.
All packages are already installed and there is nothing to restore.
NuGet package restore finished.
1>------ Build started: Project: WingtipToys, Configuration: Debug Any CPU ------
1>  WingtipToys -> c:\users\user\documents\visual studio 2013\Projects\WingtipToys\WingtipToys\bin\WingtipToys.dll
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

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 enable you to determine what part of the above steps need to be reviewed and fixed in your project.

## 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

This tutorial series will teach you the basics of building an ASP.NET Web Forms application using ASP.NET 4.5 and Microsoft Visual Studio Express 2013 for Web. A Visual Studio 2013 project with C# source code is available to accompany this tutorial series.

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 highlighted in yellow below.

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

4. Also on the *Default.aspx* page, replace all of the default content contained in the `<asp:Content>` tag so that the markup appears as below.

```
<asp:Content ID="BodyContent" ContentPlaceHolderID="MainContent"
runat="server">
        <h1><%: Title %>.</h1>
        <h2>Wingtip Toys can help you find the perfect gift.</h2>
        <p class="lead">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>
</asp:Content>
```

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

The resulting *Default.aspx* page will appear as follows:

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

<asp:Content ID="BodyContent" ContentPlaceHolderID="MainContent"
runat="server">
        <h1><%: Title %>.</h1>
        <h2>Wingtip Toys can help you find the perfect gift.</h2>
        <p class="lead">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>
</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. The content contained in the *Default.aspx* page will be integrated with the master page content, which will be explained later in this tutorial.

## @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.

**Note**

The *Machine.config* also provides details about all possible configuration settings.

## Web Server Controls

In most ASP.NET Web Forms applications, you will add controls that allow the user to interact with the page, such as 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 add this logo, 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 **modifying or adding** the markup highlighted in yellow:

```
<%@ Master Language="C#" AutoEventWireup="true" CodeBehind="Site.master.cs"
Inherits="WingtipToys.SiteMaster" %>

<!DOCTYPE html>

<html lang="en">
<head runat="server">
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title><%: Page.Title %> - Wingtip Toys</title>

    <asp:PlaceHolder runat="server">
        <%: Scripts.Render("~/bundles/modernizr") %>
    </asp:PlaceHolder>
    <webopt:bundlereference runat="server" path="~/Content/css" />
    <link href="~/favicon.ico" rel="shortcut icon" type="image/x-icon" />

</head>
<body>
    <form runat="server">
        <asp:ScriptManager runat="server">
            <Scripts>
                <%--To learn more about bundling scripts in ScriptManager see
http://go.microsoft.com/fwlink/?LinkID=301884 --%>
                <%--Framework Scripts--%>
                <asp:ScriptReference Name="MsAjaxBundle" />
                <asp:ScriptReference Name="jquery" />
                <asp:ScriptReference Name="bootstrap" />
                <asp:ScriptReference Name="respond" />
                <asp:ScriptReference Name="WebForms.js" Assembly="System.Web"
Path="~/Scripts/WebForms/WebForms.js" />
                <asp:ScriptReference Name="WebUIValidation.js"
Assembly="System.Web" Path="~/Scripts/WebForms/WebUIValidation.js" />
                <asp:ScriptReference Name="MenuStandards.js"
Assembly="System.Web" Path="~/Scripts/WebForms/MenuStandards.js" />
```

```
                <asp:ScriptReference Name="GridView.js" Assembly="System.Web"
Path="~/Scripts/WebForms/GridView.js" />
                <asp:ScriptReference Name="DetailsView.js"
Assembly="System.Web" Path="~/Scripts/WebForms/DetailsView.js" />
                <asp:ScriptReference Name="TreeView.js" Assembly="System.Web"
Path="~/Scripts/WebForms/TreeView.js" />
                <asp:ScriptReference Name="WebParts.js" Assembly="System.Web"
Path="~/Scripts/WebForms/WebParts.js" />
                <asp:ScriptReference Name="Focus.js" Assembly="System.Web"
Path="~/Scripts/WebForms/Focus.js" />
                <asp:ScriptReference Name="WebFormsBundle" />
                <%--Site Scripts--%>
            </Scripts>
        </asp:ScriptManager>

        <div class="navbar navbar-inverse navbar-fixed-top">
            <div class="container">
                <div class="navbar-header">
                    <button type="button" class="navbar-toggle" data-
toggle="collapse" data-target=".navbar-collapse">
                        <span class="icon-bar"></span>
                        <span class="icon-bar"></span>
                        <span class="icon-bar"></span>
                    </button>
                    <a class="navbar-brand" runat="server" href="~/">Wingtip
Toys</a>
                </div>
                <div class="navbar-collapse collapse">
                    <ul class="nav navbar-nav">
                        <li><a runat="server" href="~/">Home</a></li>
                        <li><a runat="server" href="~/About">About</a></li>
                        <li><a runat="server" href="~/Contact">Contact</a></li>
                    </ul>
                    <asp:LoginView runat="server" ViewStateMode="Disabled">
                        <AnonymousTemplate>
                            <ul class="nav navbar-nav navbar-right">
                                <li><a runat="server"
href="~/Account/Register">Register</a></li>
                                <li><a runat="server"
href="~/Account/Login">Log in</a></li>
                            </ul>
                        </AnonymousTemplate>
                        <LoggedInTemplate>
                            <ul class="nav navbar-nav navbar-right">
                                <li><a runat="server" href="~/Account/Manage"
title="Manage your account">Hello, <%: Context.User.Identity.GetUserName()  %>
!</a></li>
                                <li>
                                    <asp:LoginStatus runat="server"
LogoutAction="Redirect" LogoutText="Log off" LogoutPageUrl="~/"
OnLoggingOut="Unnamed_LoggingOut" />
                                </li>
                            </ul>
                        </LoggedInTemplate>
                    </asp:LoginView>
                </div>
            </div>
        </div>
        <div id="TitleContent" style="text-align: center">
            <a runat="server" href="~/">
                <asp:Image  ID="Image1" runat="server"
ImageUrl="~/Images/logo.jpg" BorderStyle="None" />
            </a>
```

```
                <br />
            </div>
            <div class="container body-content">
                <asp:ContentPlaceHolder ID="MainContent" runat="server">
                </asp:ContentPlaceHolder>
                <hr />
                <footer>
                    <p>&copy; <%: DateTime.Now.Year %> - Wingtip Toys</p>
                </footer>
            </div>
        </form>
</body>
</html>
```

This HTML will display the image named *logo.jpg* from the *Images* folder 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

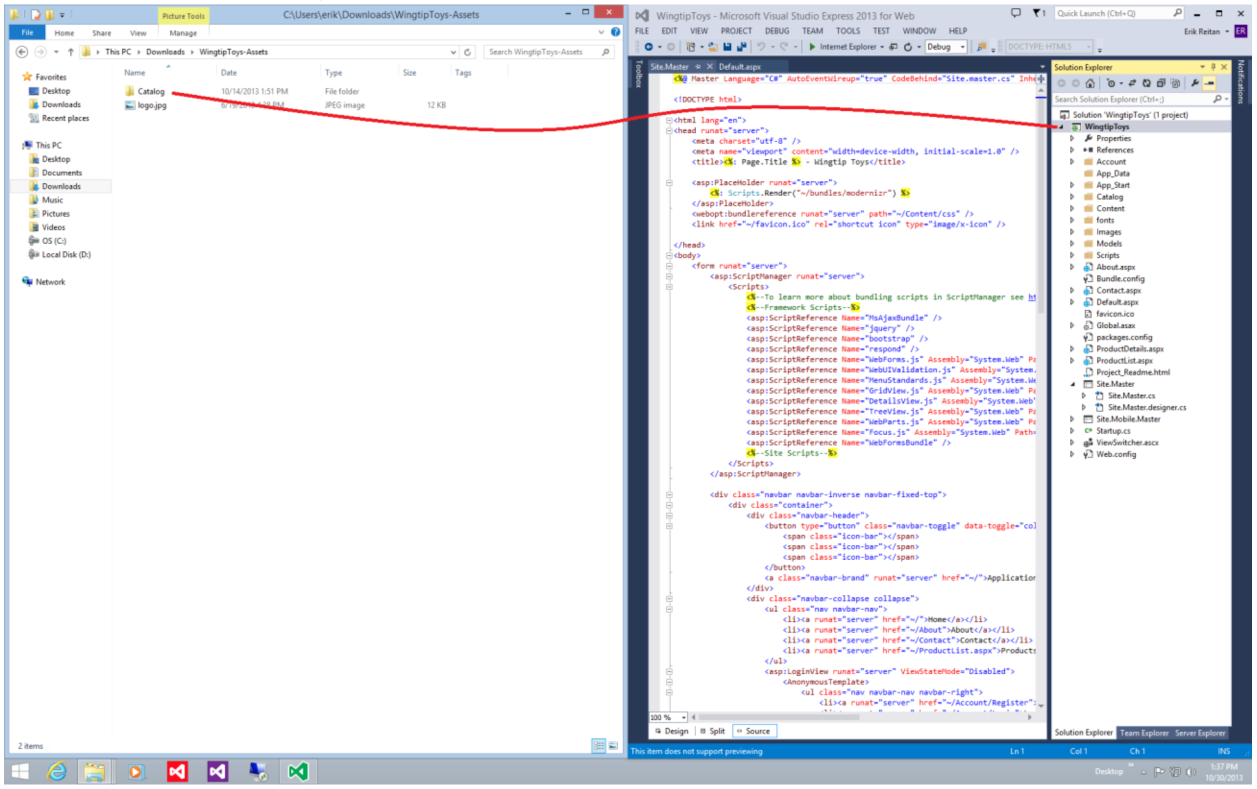
The logo image that is referenced above, along with all the product images, must be added to the Web application so that they can be seen when the project is displayed in a browser.

### Download from MSDN Samples site:

Getting Started with ASP.NET 4.5 Web Forms and Visual Studio 2013 - Wingtip Toys (C#)

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

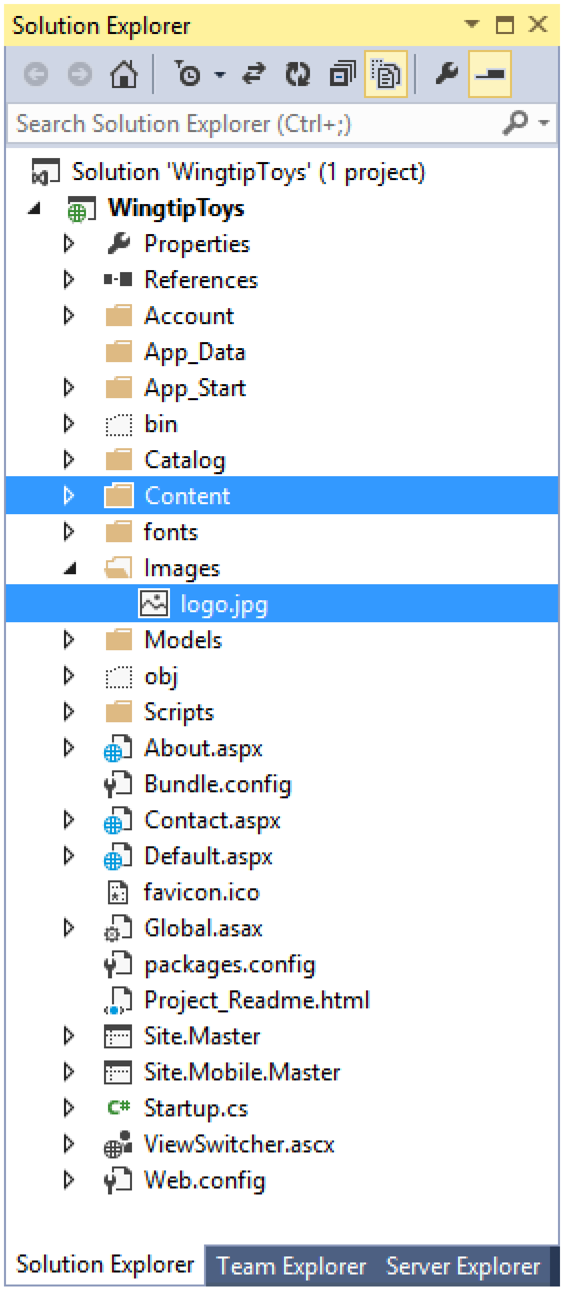
1. If you haven't already done so, download the compressed sample files using the above link from the MSDN Samples site.
2. Once downloaded, open the .zip file and copy the contents to a local folder on your machine.
3. Find and open the *WingtipToys-Assets* folder.
4. By dragging and dropping, copy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



```
<%@ Master Language="C#" AutoEventWireup="true" CodeBehind="Site.master.cs" Inhe

<!DOCTYPE html>

<html lang="en">
<head runat="server">
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title><%: Page.Title %> - Wingtip Toys</title>

    <asp:PlaceHolder runat="server">
        <%: Scripts.Render("~/bundles/modernizr") %>
    </asp:PlaceHolder>
    <webopt:bundlereference runat="server" path="~/Content/css" />
    <link href="~/favicon.ico" rel="shortcut icon" type="image/x-icon" />

</head>
<body>
    <form runat="server">
        <asp:ScriptManager runat="server">
            <Scripts>
                <%--To learn more about bundling scripts in ScriptManager see ht
                <%--Framework Scripts--%>
                <asp:ScriptReference Name="MsAjaxBundle" />
                <asp:ScriptReference Name="jquery" />
                <asp:ScriptReference Name="bootstrap" />
                <asp:ScriptReference Name="respond" />
                <asp:ScriptReference Name="WebForms.js" Assembly="System.Web" Pa
                <asp:ScriptReference Name="WebUIValidation.js" Assembly="System.
                <asp:ScriptReference Name="MenuStandards.js" Assembly="System.We
                <asp:ScriptReference Name="GridView.js" Assembly="System.Web" Pa
                <asp:ScriptReference Name="DetailsView.js" Assembly="System.Web"
                <asp:ScriptReference Name="TreeView.js" Assembly="System.Web" Pa
                <asp:ScriptReference Name="WebParts.js" Assembly="System.Web" Pa
                <asp:ScriptReference Name="Focus.js" Assembly="System.Web" Path=
                <asp:ScriptReference Name="WebFormsBundle" />
                <%--Site Scripts--%>
            </Scripts>
        </asp:ScriptManager>

        <div class="navbar navbar-inverse navbar-fixed-top">
            <div class="container">
                <div class="navbar-header">
                    <button type="button" class="navbar-toggle" data-toggle="col
                        <span class="icon-bar"></span>
                        <span class="icon-bar"></span>
                        <span class="icon-bar"></span>
                    </button>
                    <a class="navbar-brand" runat="server" href="~/">Applicatio
                </div>
                <div class="navbar-collapse collapse">
                    <ul class="nav navbar-nav">
                        <li><a runat="server" href="~/">Home</a></li>
                        <li><a runat="server" href="~/About">About</a></li>
                        <li><a runat="server" href="~/Contact">Contact</a></li>
                        <li><a runat="server" href="~/ProductList.aspx">Products
                    </ul>
                    <asp:LoginView runat="server" ViewStateMode="Disabled">
                        <AnonymousTemplate>
                            <ul class="nav navbar-nav navbar-right">
                                <li><a runat="server" href="~/Account/Register">
```

5. Next, create a new folder named *Images* by right-clicking the **WingtipToys** project and selecting **Add** -> **New Folder**.
6. Copy the *logo.jpg* file from the *WingtipToys-Assets* folder in **File Explorer** to the *Images* folder of the Web application project in **Solution Explorer** of Visual Studio.
7. Click the **Show All Files** option at the top of **Solution Explorer** to update the list of files if you don't see the new files.

**Solution Explorer** now shows the updated project files.



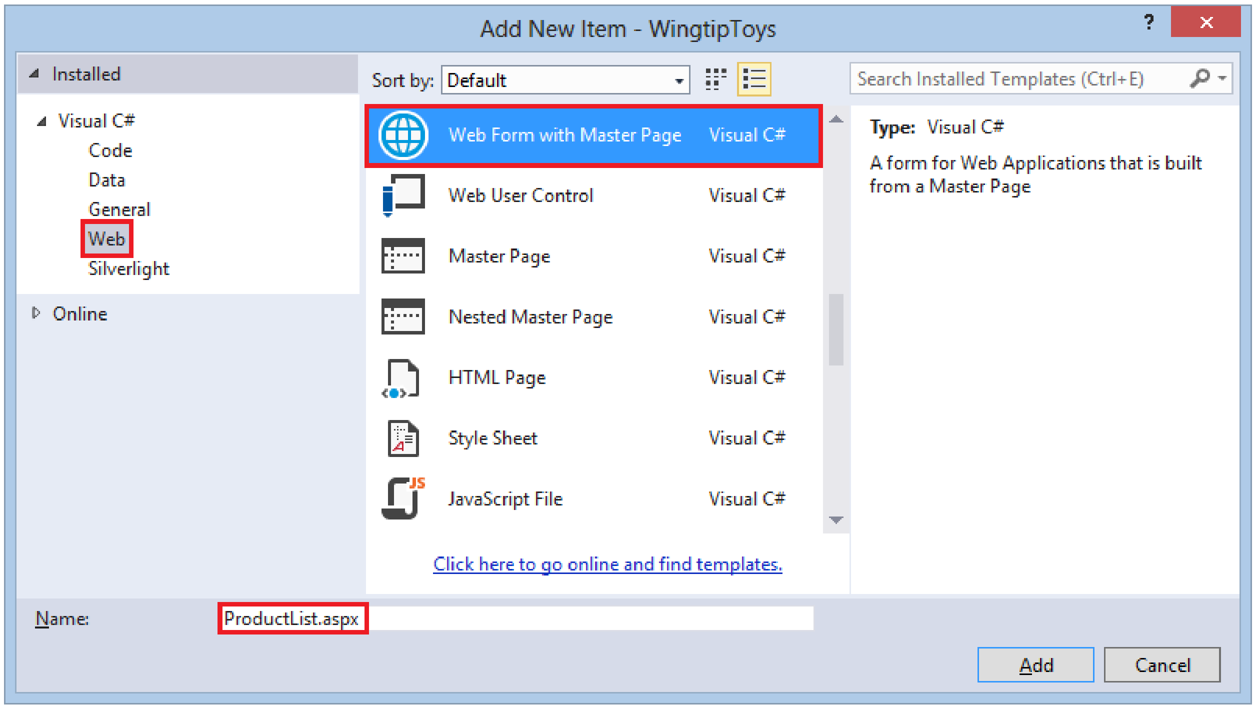
```
Solution 'WingtipToys' (1 project)
◢ WingtipToys
    ▷ Properties
    ▷ References
    ▷ Account
      App_Data
    ▷ App_Start
    ▷ bin
    ▷ Catalog
    ▷ Content
    ▷ fonts
    ◢ Images
        logo.jpg
    ▷ Models
    ▷ obj
    ▷ Scripts
    ▷ About.aspx
      Bundle.config
    ▷ Contact.aspx
    ▷ Default.aspx
      favicon.ico
    ▷ Global.asax
      packages.config
      Project_Readme.html
    ▷ Site.Master
    ▷ Site.Mobile.Master
    ▷ Startup.cs
    ▷ ViewSwitcher.ascx
    ▷ Web.config
```

## Adding Pages

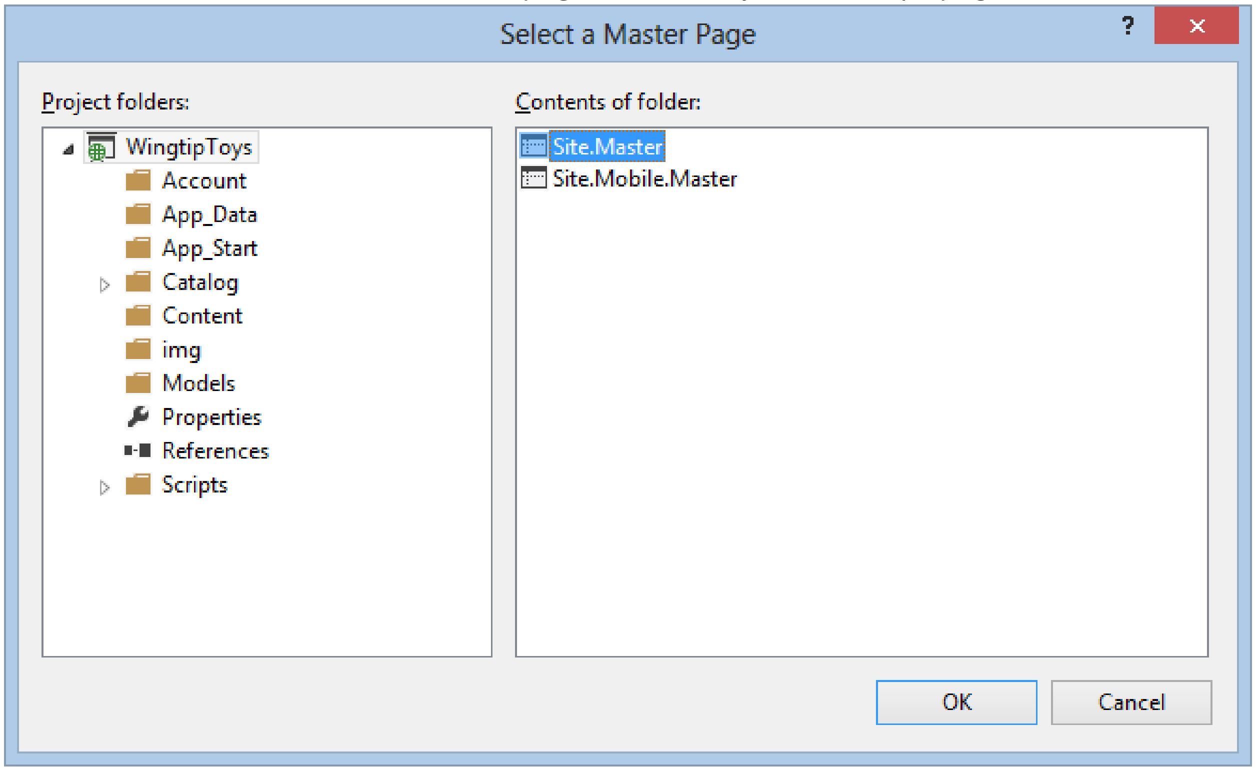
Before adding navigation to the Web application, you'll first add two new pages that you'll navigate to. Later in this tutorial series, you'll display products and product details on these new pages.

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 with Master Page** from the middle list and name it *ProductList.aspx*.


Add New Item - WingtipToys
Installed
Visual C#
Code
Data
General
Web
Silverlight
Online
Sort by: Default
Search Installed Templates (Ctrl+E)
Web Form with Master Page Visual C#
Web User Control Visual C#
Master Page Visual C#
Nested Master Page Visual C#
HTML Page Visual C#
Style Sheet Visual C#
JavaScript File Visual C#
Click here to go online and find templates.
Type: Visual C#
A form for Web Applications that is built from a Master Page
Name: ProductList.aspx
Add
Cancel

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



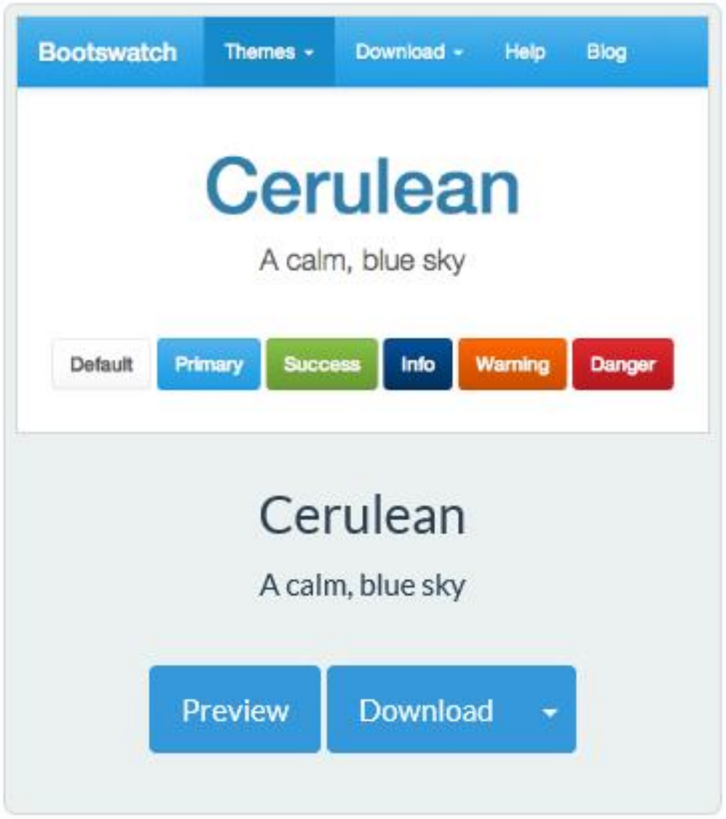
4. Add an additional page named *ProductDetails.aspx* by following these same steps.

## Updating Bootstrap

The Visual Studio 2013 project templates use Bootstrap, a layout and theming framework created by Twitter. Bootstrap uses CSS3 to provide responsive design, which means layouts can dynamically adapt to different browser window sizes. You can also use Bootstrap's theming feature to easily effect a change in the application's look and feel. By default, the ASP.NET Web Application template in Visual Studio 2013 includes Bootstrap as a NuGet package.

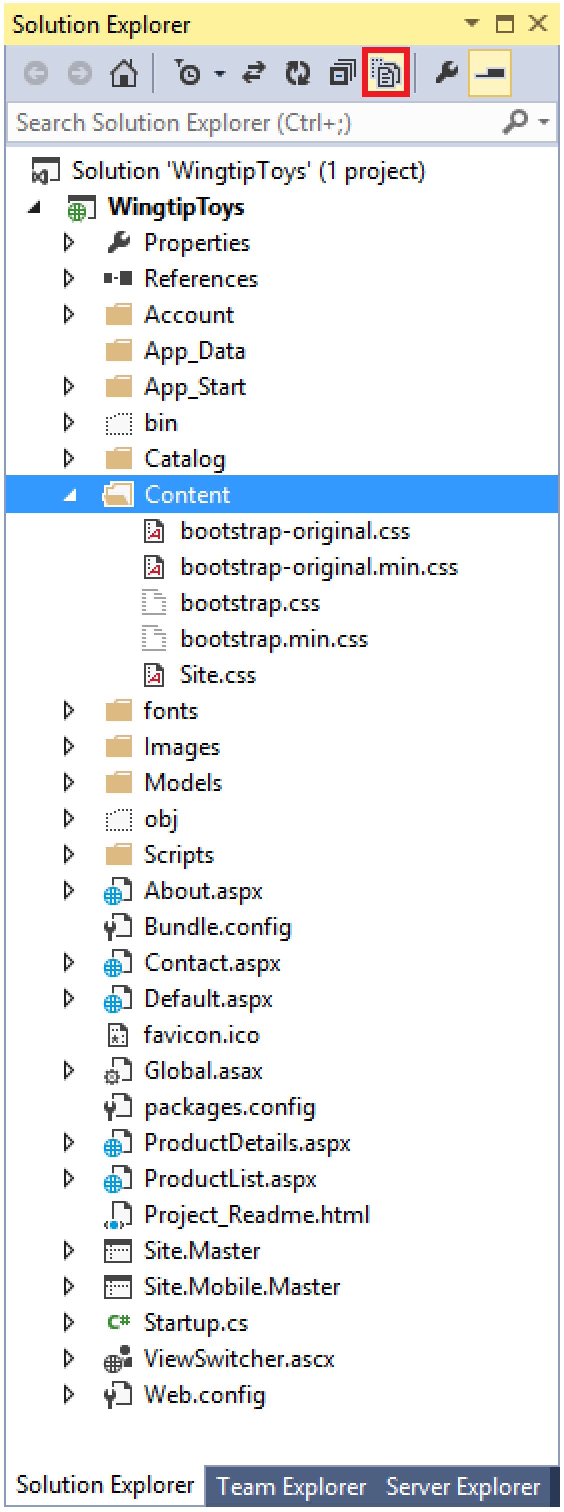
In this tutorial, you will change look and feel of the Wingtip Toys application by replacing the Bootstrap CSS files.

1. In **Solution Explorer**, open the *Content* folder.
2. Right-click the *bootstrap.css* file and rename it to *bootstrap-original.css*.
3. Rename the *bootstrap.min.css* to *bootstrap-original.min.css*.
4. In **Solution Explorer**, right-click the *Content* folder and select **Open Folder in File Explorer**.
   The File Explorer will be displayed. You will save a downloaded bootstrap CSS files to this location.
5. In your browser, go to http://Bootswatch.com.
6. Scroll the browser window until you see the Cerulean theme.



7. Download both the *bootstrap.css* file and the *bootstrap.min.css* file to the *Content* folder. Use the path to the content folder that is displayed in the **File Explorer** window that you previously opened.

8. In **Visual Studio** at the top of **Solution Explorer**, select the **Show All Files** option to display the new files in the Content folder.



You will see the two new CSS files in the **Content** folder, but notice that the icon next to

each file name is grayed out. This means that the file has not yet been added to the project.
9. Right-click the *bootstrap.css* and the *bootstrap.min.css* files and select **Include In Project**.

When you run the Wingtip Toys application later in this tutorial, the new UI will be displayed.

**Note**

The ASP.NET Web Application template uses the *Bundle.config* file at the root of the project to store the path of the Bootstrap CSS files.

## Modifying the Default Navigation

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

1. In **Solution Explorer**, locate and open the *Site.Master* page.
2. Add the additional navigation link highlighted in yellow to the unordered list shown below:

```
<ul class="nav navbar-nav">
    <li><a runat="server" href="~/">Home</a></li>
    <li><a runat="server" href="~/About">About</a></li>
    <li><a runat="server" href="~/Contact">Contact</a></li>
    <li><a runat="server" href="~/ProductList">Products</a></li>
</ul>
```

As you can see in the above HTML, you modified each line item `<li>` 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**). You will run the application at the end of this tutorial.

**Note**

The tilde (~) character is used to specify that the `href` path begins at the root of the project.

## 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, add the following highlighted `<div>` element **after** the `<div>` element containing the `id="TitleContent"` that you added earlier:

```
        <div id="TitleContent" style="text-align: center">
            <a runat="server" href="~/">
                <asp:Image  ID="Image1" runat="server"
ImageUrl="~/img/logo.jpg" BorderStyle="None" />
            </a>
            <br />
        </div>
        <div id="CategoryMenu" style="text-align: center">
            <asp:ListView ID="categoryList"
                ItemType="WingtipToys.Models.Category"
                runat="server"
                SelectMethod="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>
        </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.

**Tip**

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.

In the next step, you will implement the `GetCategories` method to retrieve data.

## 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. Near the beginning of the *Site.Master.cs* file, add two additional namespaces so that all the included namespaces appear as follows:

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

3. Add the highlighted `GetCategories` method after the `Page_Load` event handler as follows:

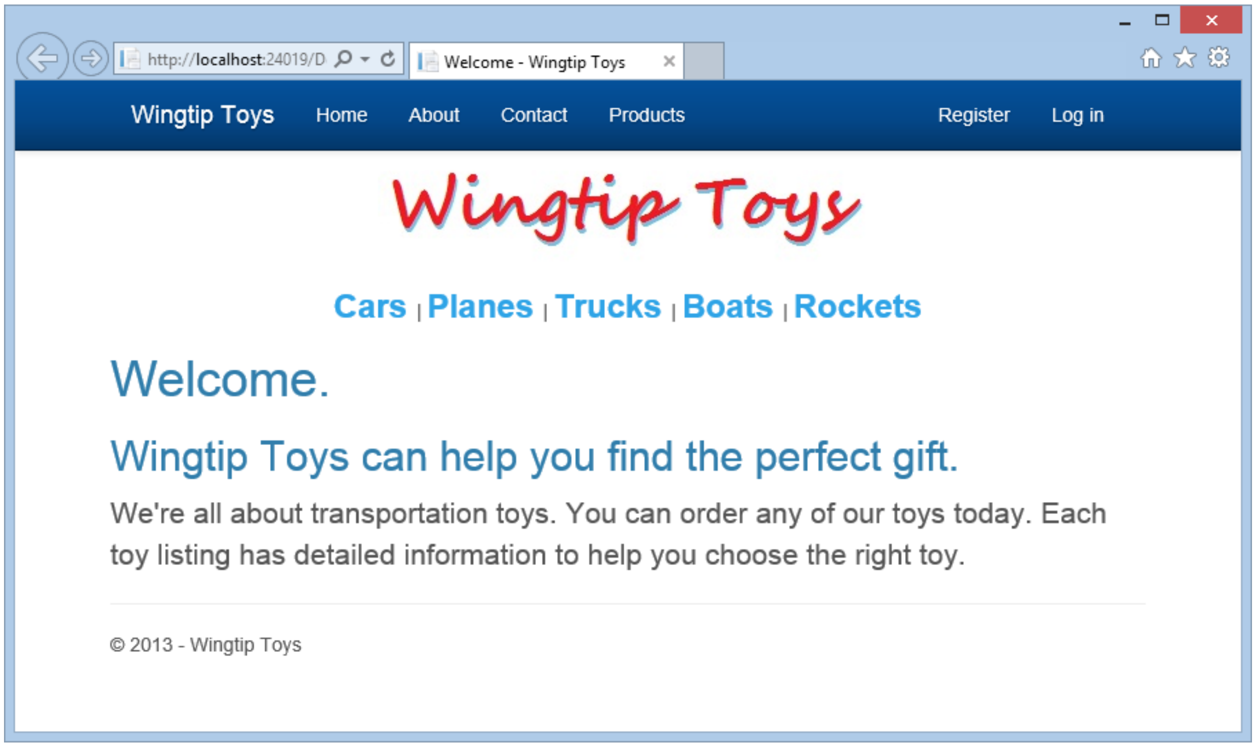
```
protected void Page_Load(object sender, EventArgs e)
{

}

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.cs* 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 **Solution Explorer**, right-click the *Default.aspx* page and select **Set As Start Page**.
2. In Visual Studio press **F5**.
   It will take a little time to set everything up during this first run.



Welcome.

Wingtip Toys can help you find the perfect gift.

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.

© 2013 - Wingtip Toys

When you run the application, the application will be compiled and the database named *wingtiptoys.mdf* will be created in the *App_Data* folder. In the browser, you will see a category navigation menu. This menu was generated by retrieving the categories from the database. In the next tutorial, you will implement the navigation.

3. Close the browser to stop the running application.

## Reviewing the Database

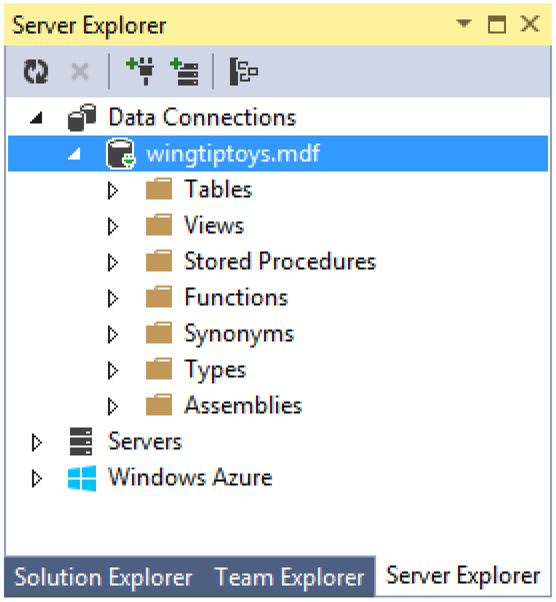
Open the *Web.config* file and look at the connection string section. You can see that the `AttachDbFilename` value in the connection string points to the `DataDirectory` for the Web application project. The value `|DataDirectory|` is a reserved value that represents the *App_Data* folder in the project.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

```
<connectionStrings>
    <add name="DefaultConnection"
         connectionString="Data Source=(LocalDb)\v11.0;Initial Catalog=aspnet-
WingtipToys-20120302100502;Integrated Security=True"
         providerName="System.Data.SqlClient" />
    <add name="WingtipToys"
         connectionString="Data
Source=(LocalDB)\v11.0;AttachDbFilename=|DataDirectory|\wingtiptoys.mdf;Integra
ted Security=True"
         providerName="System.Data.SqlClient " />
  </connectionStrings>
```

**Note**

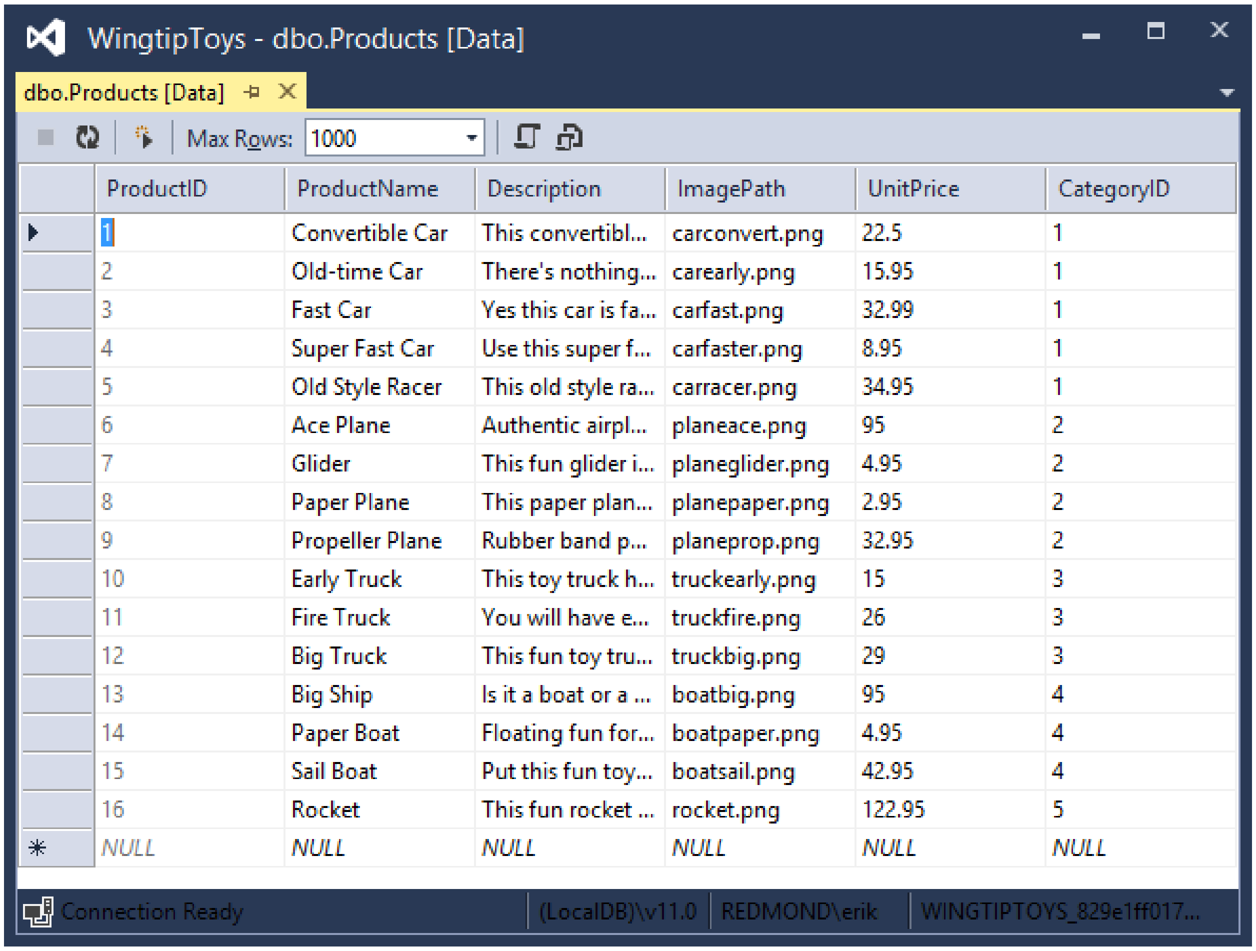
If the *App_Data* folder is not visible or if the folder is empty, select the **Refresh** icon and then the **Show All Files** icon at the top of the **Solution Explorer** window. Expanding the width of the **Solution Explorer** windows may be required to show all available icons.

Now you can inspect the data contained in the *wingtiptoys.mdf* database file by using the **Server Explorer** window.

1. Expand the *App_Data* folder. If the *App_Data* folder is not visible, see the note above.
2. If the *wingtiptoys.mdf* database file is not visible, select the **Refresh** icon and then the **Show All Files** icon at the top of the **Solution Explorer** window.
3. Right-click the *wingtiptoys.mdf* database file and select **Open**.
   **Server Explorer** is displayed.
   

4. Expand the *Tables* folder.

5. Right-click the **Products** table and select **Show Table Data**.
   The **Products** table is displayed.


WingtipToys - dbo.Products [Data]

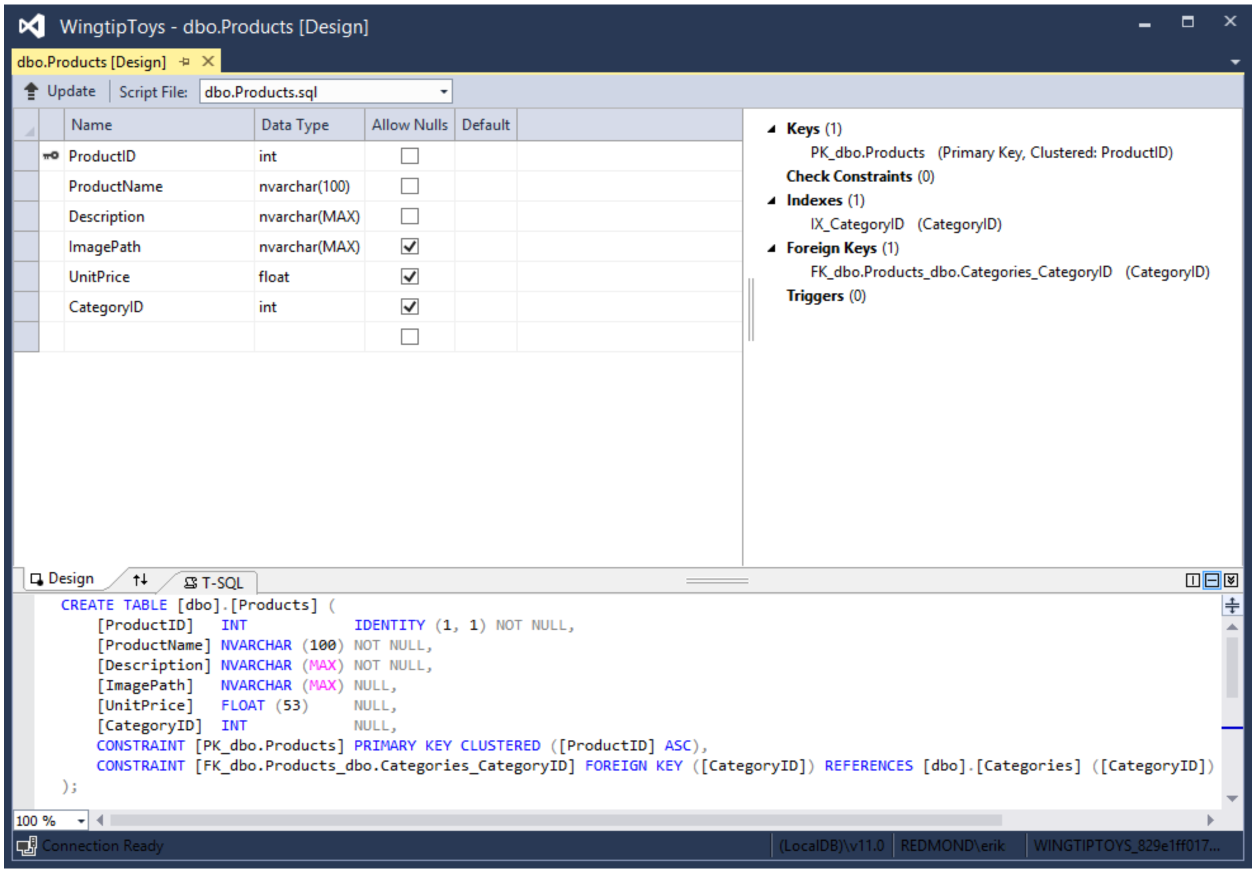
dbo.Products [Data]

| | 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... | planeglider.png | 4.95 | 2 |
| | 8 | Paper Plane | This paper plan... | planepaper.png | 2.95 | 2 |
| | 9 | Propeller Plane | Rubber band p... | planeprop.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\erik | WINGTIPTOYS_829e1ff017...

6. This view lets you see and modify the data in the **Products** table by hand.
7. Close the **Products** table window.
8. In the **Server Explorer**, right-click the **Products** table again and select **Open Table Definition**.

The data design for the **Products** table is displayed.


WingtipToys - dbo.Products [Design]
dbo.Products [Design]
Update
Script File: dbo.Products.sql
Design
T-SQL
100 %
Connection Ready
(LocalDB)\v11.0
REDMOND\erik
WINGTIPTOYS_829e1ff017...

| Name | Data Type | Allow Nulls | Default |
|---|---|---|---|
| ProductID | int | ☐ | |
| ProductName | nvarchar(100) | ☐ | |
| Description | nvarchar(MAX) | ☐ | |
| ImagePath | nvarchar(MAX) | ✓ | |
| UnitPrice | float | ✓ | |
| CategoryID | int | ✓ | |
| | | ☐ | |

**Keys** (1)
PK_dbo.Products (Primary Key, Clustered: ProductID)
**Check Constraints** (0)
**Indexes** (1)
IX_CategoryID (CategoryID)
**Foreign Keys** (1)
FK_dbo.Products_dbo.Categories_CategoryID (CategoryID)
**Triggers** (0)

```
CREATE TABLE [dbo].[Products] (
    [ProductID]   INT            IDENTITY (1, 1) NOT NULL,
    [ProductName] NVARCHAR (100) NOT NULL,
    [Description] NVARCHAR (MAX) NOT NULL,
    [ImagePath]   NVARCHAR (MAX) NULL,
    [UnitPrice]   FLOAT (53)     NULL,
    [CategoryID]  INT            NULL,
    CONSTRAINT [PK_dbo.Products] PRIMARY KEY CLUSTERED ([ProductID] ASC),
    CONSTRAINT [FK_dbo.Products_dbo.Categories_CategoryID] FOREIGN KEY ([CategoryID]) REFERENCES [dbo].[Categories] ([CategoryID])
);
```

9. In the **T-SQL** tab you will see the SQL DDL statement that was used to create the table. You can also use the UI in the **Design** tab to modify the schema.
10. In the **Server Explorer**, right-click **WingtipToys** database and select **Close Connection**. By detaching the database from Visual Studio, the database schema will be able to be modified later in this tutorial series.
11. Return to **Solution Explorer** by selecting the **Solution Explorer** tab at the bottom of the **Server Explorer** window.

## 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 by viewing the database directly. In the next 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 series will teach you the basics of building an ASP.NET Web Forms application using ASP.NET 4.5 and Microsoft Visual Studio Express 2013 for Web. A Visual Studio 2013 project with C# source code is available to accompany this tutorial series.

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:

```
<%@ Page Title="Products" Language="C#" MasterPageFile="~/Site.Master"
AutoEventWireup="true"
         CodeBehind="ProductList.aspx.cs" Inherits="WingtipToys.ProductList" %>
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent" runat="server">
    <section>
        <div>
            <hgroup>
                <h2><%: Page.Title %></h2>
            </hgroup>

            <asp:ListView ID="productList" runat="server"
                DataKeyNames="ProductID" GroupItemCount="4"
                ItemType="WingtipToys.Models.Product" SelectMethod="GetProducts">
                <EmptyDataTemplate>
                    <table >
                        <tr>
                            <td>No data was returned.</td>
                        </tr>
                    </table>
                </EmptyDataTemplate>
                <EmptyItemTemplate>
                    <td/>
                </EmptyItemTemplate>
                <GroupTemplate>
                    <tr id="itemPlaceholderContainer" runat="server">
                        <td id="itemPlaceholder" runat="server"></td>
                    </tr>
                </GroupTemplate>
                <ItemTemplate>
                    <td runat="server">
                        <table>
                            <tr>
                                <td>
                                    <a
href="ProductDetails.aspx?productID=<%#:Item.ProductID%>">
```

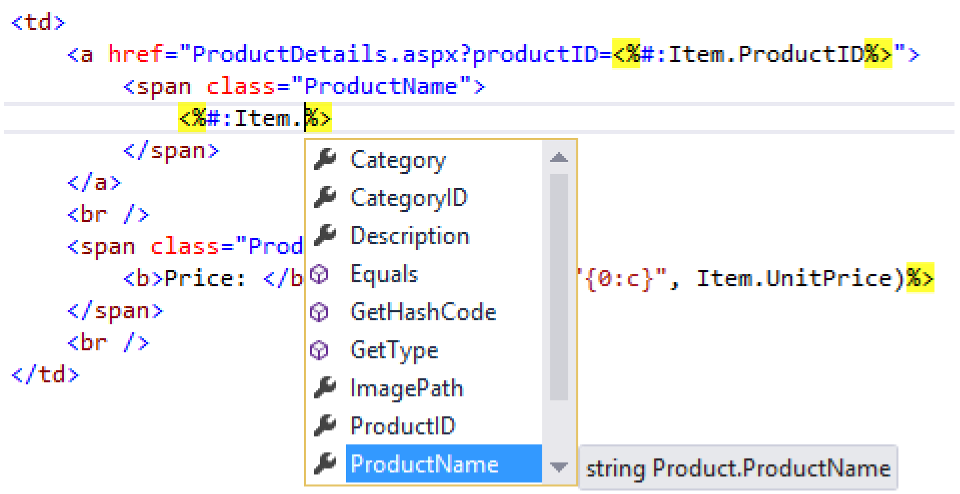
```
                                        <img 
src="/Catalog/Images/Thumbs/<%#:Item.ImagePath%>"
                                            width="100" height="75" style="border: 
solid" /></a>
                                </td>
                            </tr>
                            <tr>
                                <td>
                                    <a 
href="ProductDetails.aspx?productID=<%#:Item.ProductID%>">
                                        <span>
                                            <%#:Item.ProductName%>
                                        </span>
                                    </a>
                                    <br />
                                    <span>
                                        <b>Price: </b><%#:String.Format("{0:c}", 
Item.UnitPrice)%>
                                    </span>
                                    <br />
                                </td>
                            </tr>
                            <tr>
                                <td>&nbsp;</td>
                            </tr>
                        </table>
                        </p>
                    </td>
                </ItemTemplate>
                <LayoutTemplate>
                    <table style="width:100%;">
                        <tbody>
                            <tr>
                                <td>
                                    <table id="groupPlaceholderContainer" 
runat="server" style="width:100%">
                                        <tr id="groupPlaceholder"></tr>
                                    </table>
                                </td>
                            </tr>
                            <tr>
                                <td></td>
                            </tr>
                            <tr></tr>
                        </tbody>
                    </table>
                </LayoutTemplate>
            </asp:ListView>
        </div>
    </section>
</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`:



```
<td>
    <a href="ProductDetails.aspx?productID=<%#:Item.ProductID%>">
        <span class="ProductName">
            <%#:Item.%>
        </span>
    </a>
    <br />
    <span class="Prod
        <b>Price: </b                    '{0:c}", Item.UnitPrice)%>
    </span>
    <br />
</td>
```

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 showing 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.

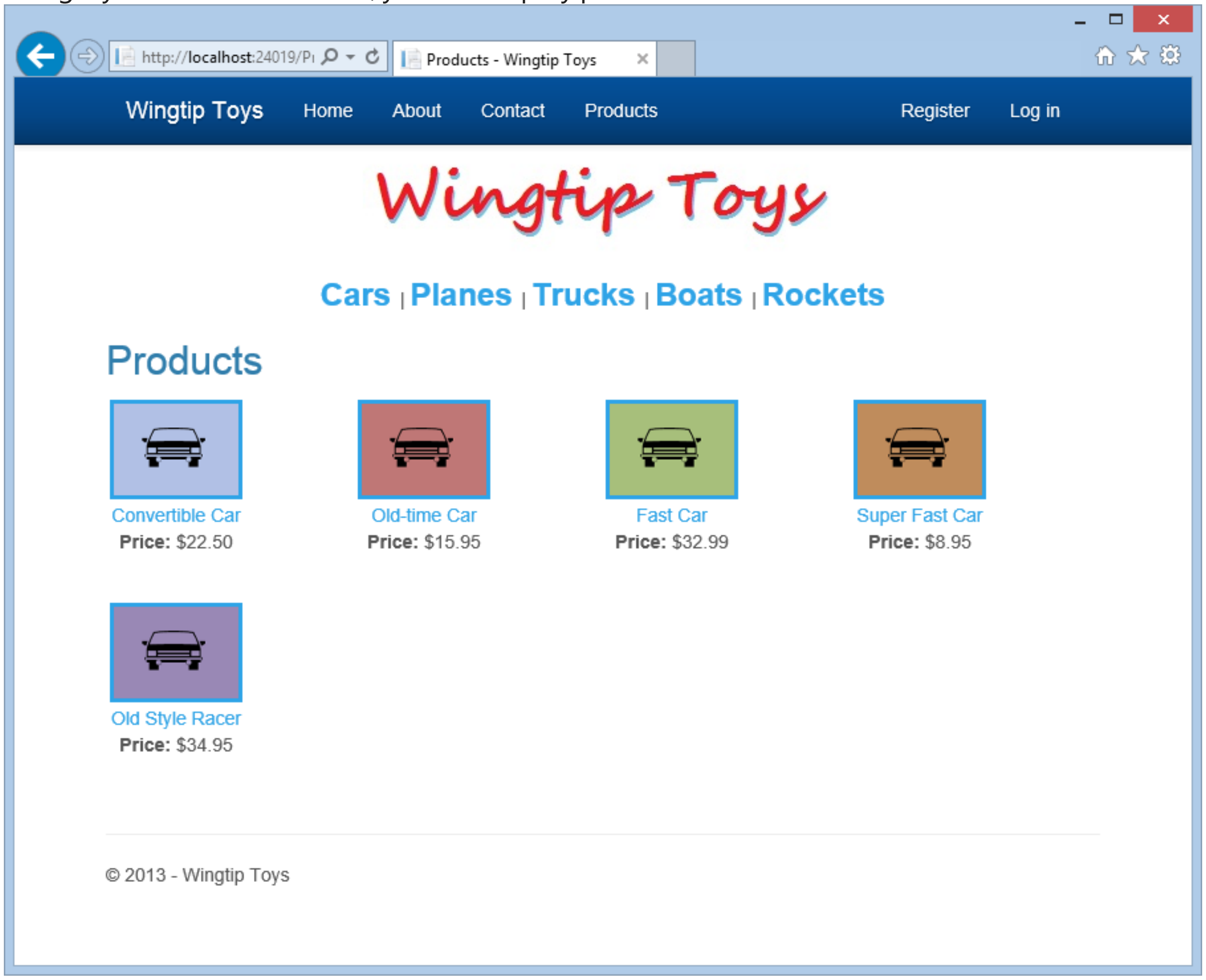
The sources of values for these methods are referred to as *value providers* (such as *QueryString*), and the parameter attributes that indicate which value provider to use are referred to as value provider attributes (such as "`id`"). ASP.NET includes value providers and corresponding attributes for 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.

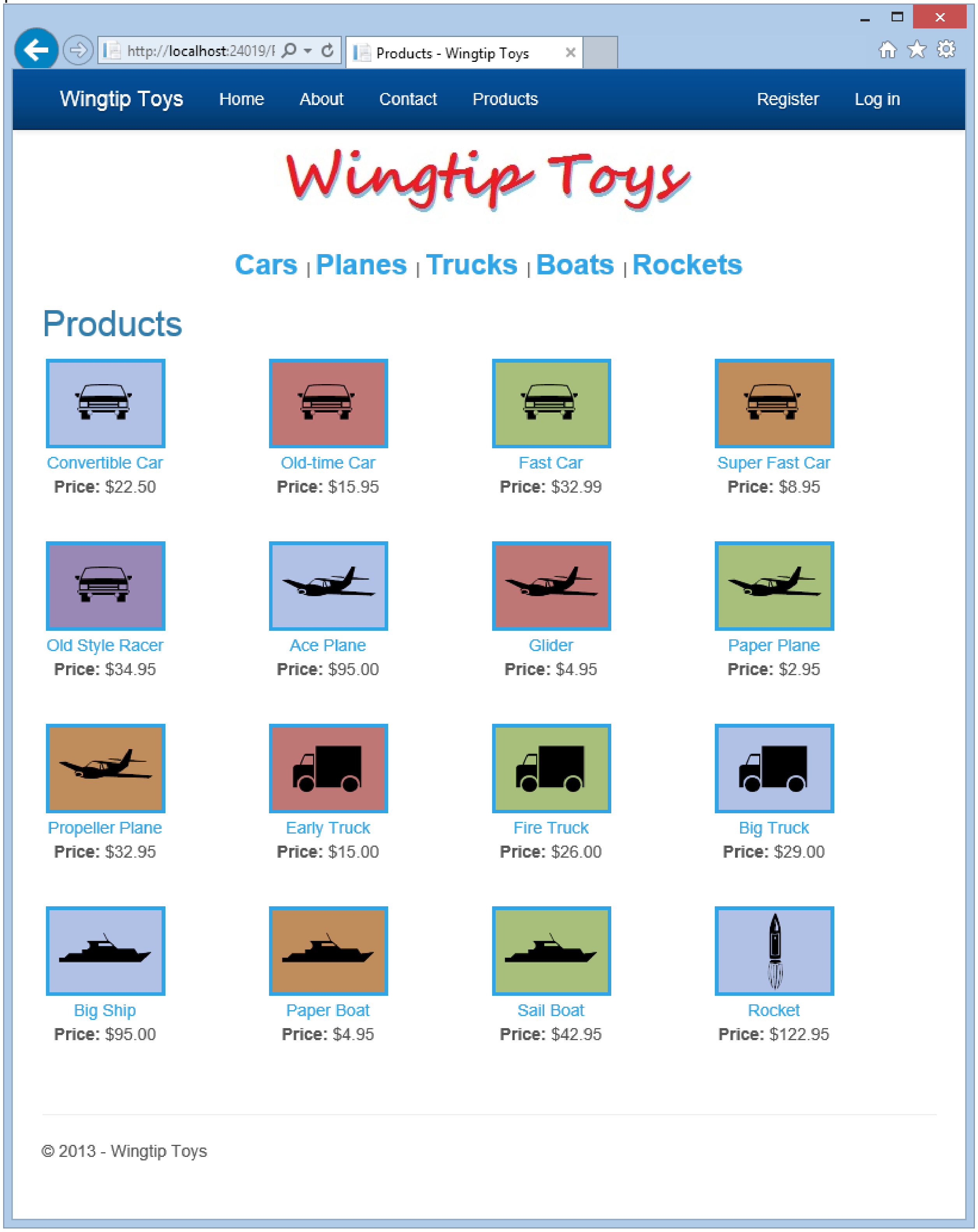
1. In the **Solution Explorer**, right-click the *Default.aspx* page and select **View in Browser**.
   The browser will open and show the *Default.aspx* page.
2. Select **Cars** from the product category navigation menu.
   The *ProductList.aspx* page is displayed showing only products included in the “Cars”

category. Later in this tutorial, you will display product details.



3. Select **Products** from the navigation menu at the top.
   Again, the *ProductList.aspx* page is displayed, however this time it shows the entire list of

products.



http://localhost:24019/P

Products - Wingtip Toys

Wingtip Toys Home About Contact Products Register Log in

Wingtip Toys

Cars | Planes | Trucks | Boats | Rockets

Products

| | | | |
|---|---|---|---|
| Convertible Car<br>**Price:** $22.50 | Old-time Car<br>**Price:** $15.95 | Fast Car<br>**Price:** $32.99 | Super Fast Car<br>**Price:** $8.95 |
| Old Style Racer<br>**Price:** $34.95 | Ace Plane<br>**Price:** $95.00 | Glider<br>**Price:** $4.95 | Paper Plane<br>**Price:** $2.95 |
| Propeller Plane<br>**Price:** $32.95 | Early Truck<br>**Price:** $15.00 | Fire Truck<br>**Price:** $26.00 | Big Truck<br>**Price:** $29.00 |
| Big Ship<br>**Price:** $95.00 | Paper Boat<br>**Price:** $4.95 | Sail Boat<br>**Price:** $42.95 | Rocket<br>**Price:** $122.95 |

© 2013 - Wingtip Toys

4. Close the browser and return to Visual Studio.

## Adding a Data Control to Display Product Details

Next, you'll modify the markup in the *ProductDetails.aspx* page that you added in the previous tutorial 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:

```
<%@ Page Title="Product Details" Language="C#" MasterPageFile="~/Site.Master" 
AutoEventWireup="true"
         CodeBehind="ProductDetails.aspx.cs" 
Inherits="WingtipToys.ProductDetails" %>
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent" runat="server">
    <asp:FormView ID="productDetail" runat="server" 
ItemType="WingtipToys.Models.Product" SelectMethod ="GetProduct" 
RenderOuterTable="false">
        <ItemTemplate>
            <div>
                <h1><%#:Item.ProductName %></h1>
            </div>
            <br />
            <table>
                <tr>
                    <td>
                        <img src="/Catalog/Images/<%#:Item.ImagePath %>" 
style="border:solid; height:300px" alt="<%#:Item.ProductName %>"/>
                    </td>
                    <td>&nbsp;</td>
                    <td style="vertical-align: top; text-align:left;">
                        <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>
```

This code uses a **FormView** control to display details about an individual product. This markup uses methods like those that are used to display data in the *ProductList.aspx* page. 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**.
   The *ProductDetails.aspx.cs* file will be displayed.
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;
        }
    }
}
```

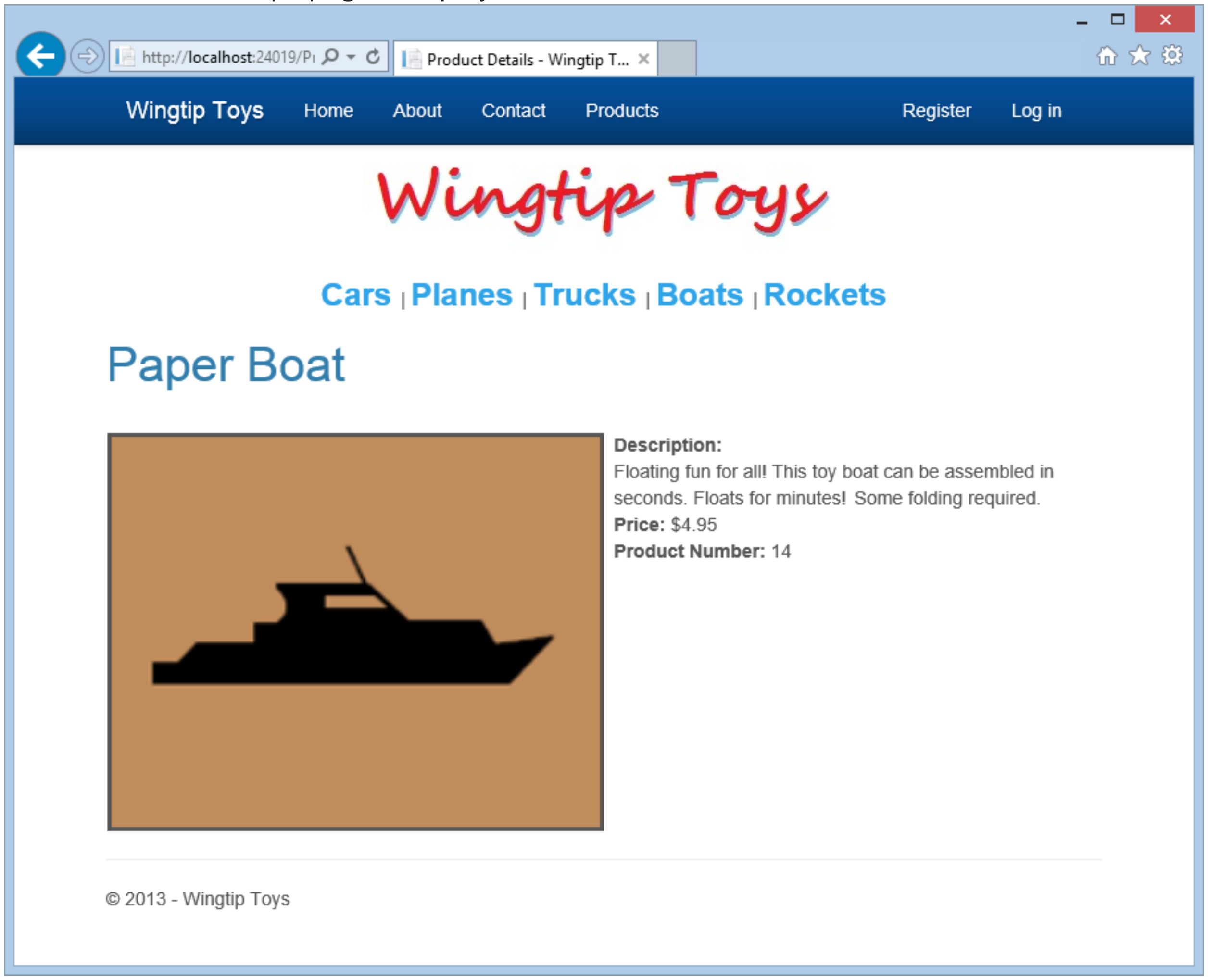
This code checks for a "`productID`" query-string value. If a valid query-string value is found, the matching product is displayed. If no query-string is found, or the query-string value is not valid, no product is displayed on the *ProductDetails.aspx* page.

## Running the Application

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

1. Press **F5** while in Visual Studio to run the application.
   The browser will open and show the *Default.aspx* page.
2. Select "Boats" from the category navigation menu.
   The *ProductList.aspx* page is displayed.

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



Paper Boat

**Description:**
Floating fun for all! This toy boat can be assembled in seconds. Floats for minutes! Some folding required.
**Price:** $4.95
**Product Number:** 14

© 2013 - Wingtip Toys

4. Close the browser.

## 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. In the next tutorial, you'll add a shopping cart to the Wingtip Toys sample application.

## Additional Resources

Retrieving and displaying data with model binding and web forms

# Shopping Cart

This tutorial series will teach you the basics of building an ASP.NET Web Forms application using ASP.NET 4.5 and Microsoft Visual Studio Express 2013 for Web. A Visual Studio 2013 project with C# source code is available to accompany this tutorial series.

This tutorial describes the business logic required to add a shopping cart to the Wingtip Toys sample ASP.NET Web Forms application. This tutorial builds on the previous tutorial "Display Data Items and Details" and is part of the Wingtip Toy Store tutorial series. When you've completed this tutorial, the users of your sample app will be able to add, remove, and modify the products in their shopping cart.

## What you'll learn:

- How to create a shopping cart for the web application.
- How to enable users to add items to the shopping cart.
- How to add a GridView control to display shopping cart details.
- How to calculate and display the order total.
- How to remove and update items in the shopping cart.
- How to include a shopping cart counter.

## Code features in this tutorial:

- Entity Framework Code First
- Data Annotations
- Strongly typed data controls
- Model binding

## Creating a Shopping Cart

Earlier in this tutorial series, you added pages and code to view product data from a database. In this tutorial, you'll create a shopping cart to manage the products that users are interested in buying. Users will be able to browse and add items to the shopping cart even if they are not registered or logged in. To manage shopping cart access, you will assign users a unique `ID` using a globally unique identifier (GUID) when the user accesses the shopping cart for the first time. You'll store this `ID` using the ASP.NET Session state.

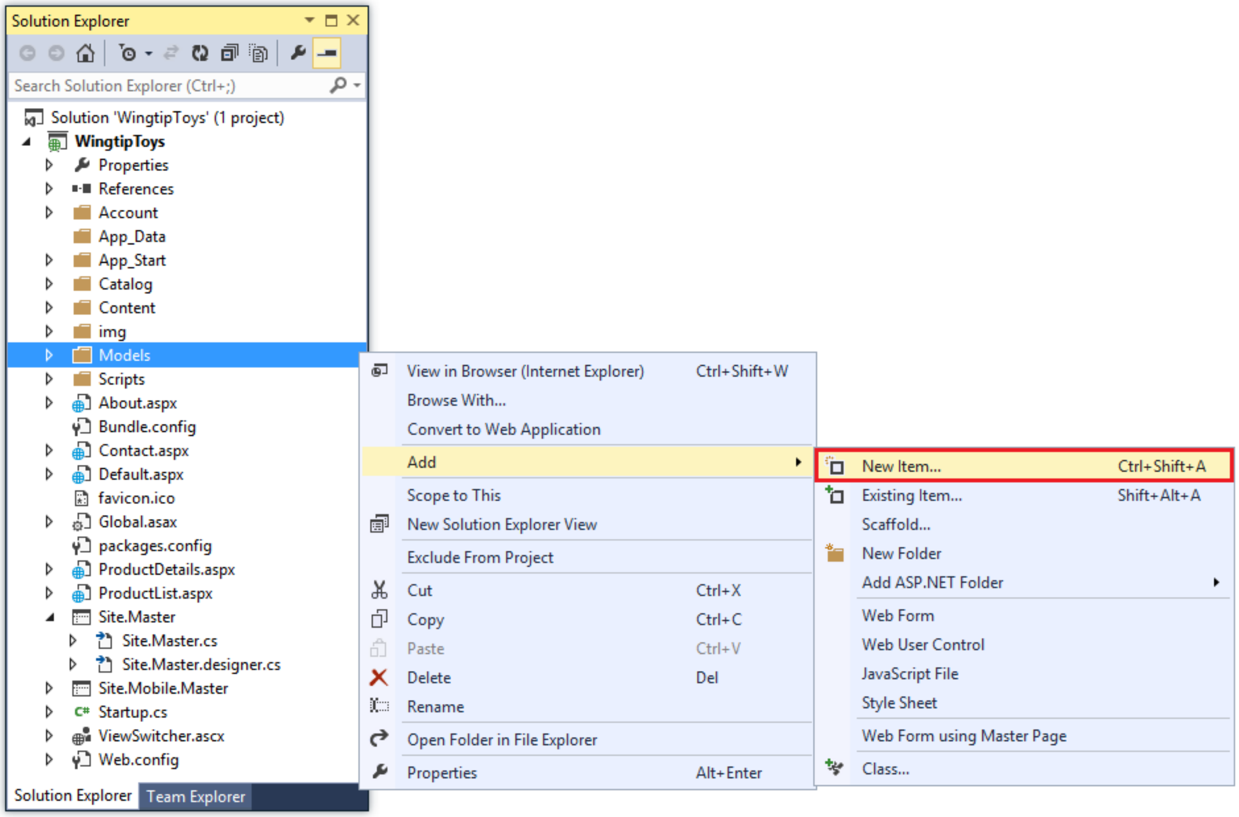
**Note**

The ASP.NET Session state is a convenient place to store user-specific information which will expire after the user leaves the site. While misuse of session state can have performance implications on larger sites, light use of session state works well for demonstration purposes. The Wingtip Toys sample project shows how to use session state without an external provider, where session state is stored in-process on the web server hosting the site. For larger sites that provide multiple instances of an application or for sites that run multiple instances of an application on different servers, consider using **Windows Azure Cache Service**. This Cache Service provides a distributed caching service that is external to the web site and solves the problem of using in-process session state. For more information see, How to Use ASP.NET Session State with Windows Azure Web Sites.

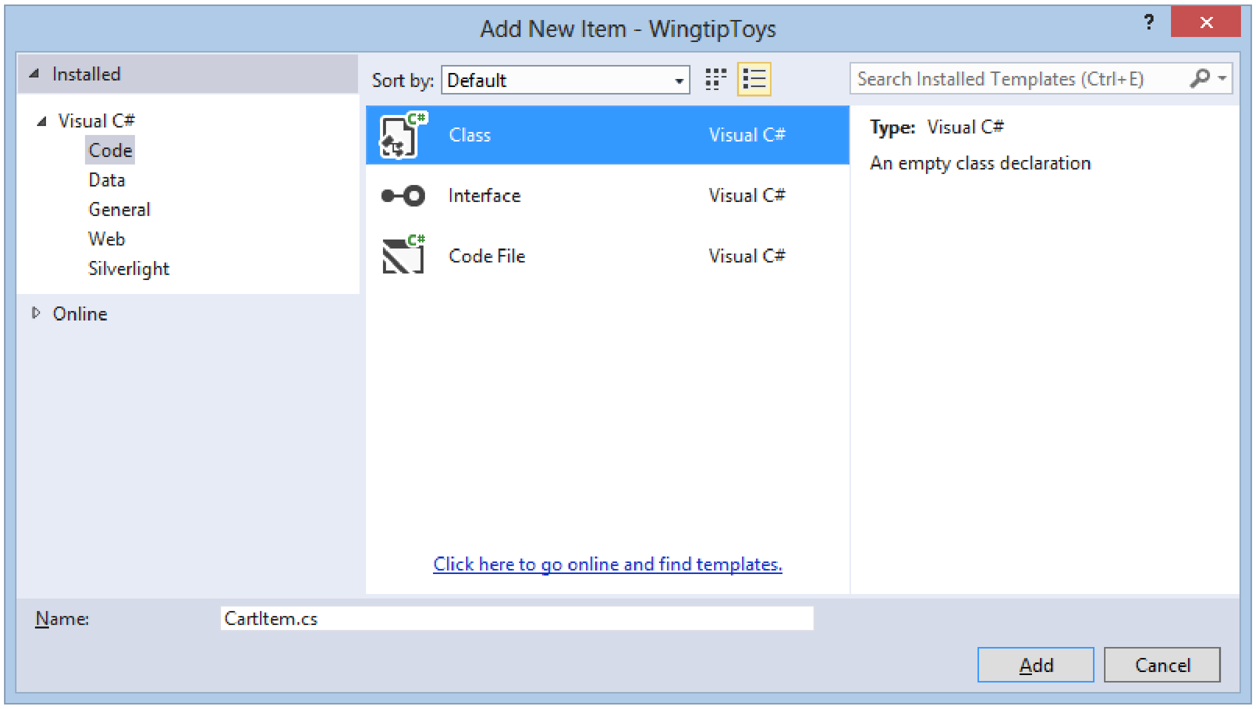
## Add CartItem as a Model Class

Earlier in this tutorial series, you defined the schema for the category and product data by creating the `Category` and `Product` classes in the *Models* folder. Now, add a new class to define the schema for the shopping cart. Later in this tutorial, you will add a class to handle data access to the `CartItem` table. This class will provide the business logic to add, remove, and update items in the shopping cart.

- Right-click the ***Models*** folder and select **Add** -> **New Item**.



- The **Add New Item** dialog box is displayed. Select **Code**, and then select **Class**.



- Name this new class *CartItem.cs*.
- Click **Add**.
  The new class file is displayed in the editor.
- Replace the default code with the following code:

```
using System.ComponentModel.DataAnnotations;

namespace WingtipToys.Models
{
  public class CartItem
  {
    [Key]
    public string ItemId { get; set; }

    public string CartId { get; set; }

    public int Quantity { get; set; }

    public System.DateTime DateCreated { get; set; }

    public int ProductId { get; set; }

    public virtual Product Product { get; set; }

  }
}
```

The `CartItem` class contains the schema that will define each product a user adds to the shopping cart. This class is similar to the other schema classes you created earlier in this tutorial series. By convention, Entity Framework Code First expects that the primary key for the `CartItem` table will be either `CartItemId` or `ID`. However, the code overrides the default behavior by using the data annotation `[Key]` attribute. The `Key` attribute of the ItemId property specifies that the `ItemID` property is the primary key.

The `CartId` property specifies the `ID` of the user that is associated with the item to purchase. You'll add code to create this user `ID` when the user accesses the shopping cart. This `ID` will also be stored as an ASP.NET Session variable.

## Update the Product Context

In addition to adding the `CartItem` class, you will need to update the database context class that manages the entity classes and that provides data access to the database. To do this, you will add the newly created `CartItem` model class to the `ProductContext` class.

1. In **Solution Explorer**, find and open the *ProductContext.cs* file in the *Models* folder.
2. Add the highlighted code to the *ProductContext.cs* file as follows:

```
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; }
        public DbSet<CartItem> ShoppingCartItems { get; set; }
    }
}
```

As mentioned previously in this tutorial series, the code in the *ProductContext.cs* file adds the `System.Data.Entity` namespace so that you have access to all the core functionality of the Entity Framework. This functionality includes the capability to query, insert, update, and delete data by working with strongly typed objects. The `ProductContext` class adds access to the newly added `CartItem` model class.

## Managing the Shopping Cart Business Logic

Next, you'll create the `ShoppingCart` class in a new *Logic* folder. The `ShoppingCart` class handles data access to the `CartItem` table. The class will also include the business logic to add, remove, and update items in the shopping cart.

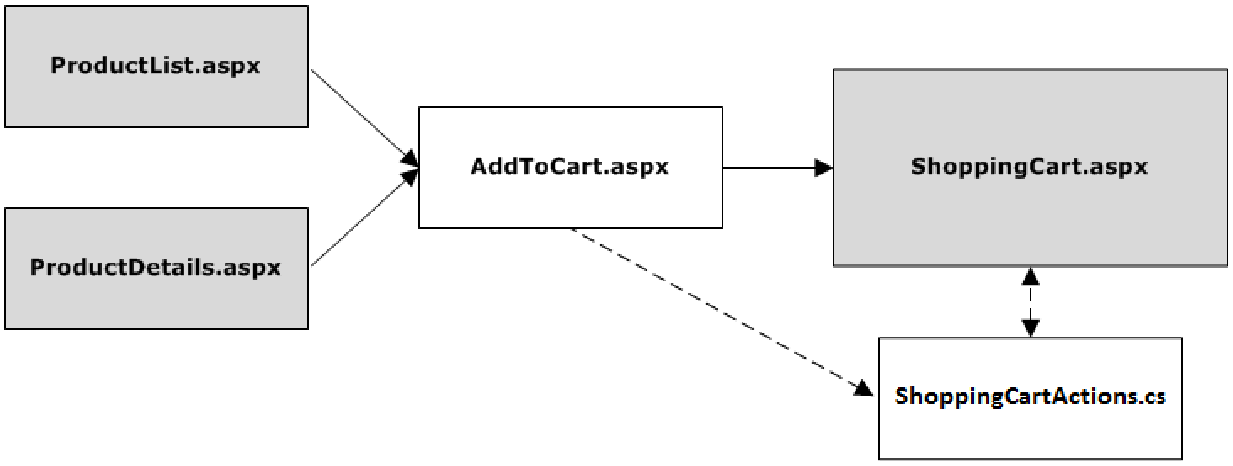
The shopping cart logic that you will add will contain the functionality to manage the following actions:

1. Adding items to the shopping cart
2. Removing items from the shopping cart
3. Getting the shopping cart ID

4. Retrieving items from the shopping cart
5. Totaling the amount of all the shopping cart items
6. Updating the shopping cart data

A shopping cart page (*ShoppingCart.aspx*) and the shopping cart class will be used together to access shopping cart data. The shopping cart page will display all the items the user adds to the shopping cart. Besides the shopping cart page and class, you'll create a page (*AddToCart.aspx*) to add products to the shopping cart. You will also add code to the *ProductList.aspx* page and the *ProductDetails.aspx* page that will provide a link to the *AddToCart.aspx* page, so that the user can add products to the shopping cart.

The following diagram shows the basic process that occurs when the user adds a product to the shopping cart.



When the user clicks the **Add To Cart** link on either the *ProductList.aspx* page or the *ProductDetails.aspx* page, the application will navigate to the *AddToCart.aspx* page and then automatically to the *ShoppingCart.aspx* page. The *AddToCart.aspx* page will add the select product to the shopping cart by calling a method in the ShoppingCart class. The *ShoppingCart.aspx* page will display the products that have been added to the shopping cart.

## Creating the Shopping Cart Class

The `ShoppingCart` class will be added to a separate folder in the application so that there will be a clear distinction between the model (Models folder), the pages (root folder) and the logic (Logic folder).

1. In **Solution Explorer**, right-click the **WingtipToys** project and select **Add** -> **New Folder**. Name the new folder *Logic*.
2. Right-click the *Logic* folder and then select **Add** -> **New Item**.
3. Add a new class file named *ShoppingCartActions.cs*.
4. Replace the default code with the following code:

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

namespace WingtipToys.Logic
{
  public class ShoppingCartActions : IDisposable
  {
    public string ShoppingCartId { get; set; }

    private ProductContext _db = new ProductContext();

    public const string CartSessionKey = "CartId";

    public void AddToCart(int id)
    {
      // Retrieve the product from the database.
      ShoppingCartId = GetCartId();

      var cartItem = _db.ShoppingCartItems.SingleOrDefault(
          c => c.CartId == ShoppingCartId
          && c.ProductId == id);
      if (cartItem == null)
      {
        // Create a new cart item if no cart item exists.
        cartItem = new CartItem
        {
          ItemId = Guid.NewGuid().ToString(),
          ProductId = id,
          CartId = ShoppingCartId,
          Product = _db.Products.SingleOrDefault(
           p => p.ProductID == id),
          Quantity = 1,
          DateCreated = DateTime.Now
        };

        _db.ShoppingCartItems.Add(cartItem);
      }
      else
      {
        // If the item does exist in the cart,
        // then add one to the quantity.
        cartItem.Quantity++;
      }
      _db.SaveChanges();
    }

    public void Dispose()
    {
      if (_db != null)
      {
        _db.Dispose();
        _db = null;
      }
    }

    public string GetCartId()
    {
      if (HttpContext.Current.Session[CartSessionKey] == null)
      {
        if (!string.IsNullOrWhiteSpace(HttpContext.Current.User.Identity.Name))
        {
          HttpContext.Current.Session[CartSessionKey] =
HttpContext.Current.User.Identity.Name;
        }
        else
        {
          // Generate a new random GUID using System.Guid class.
          Guid tempCartId = Guid.NewGuid();
          HttpContext.Current.Session[CartSessionKey] = tempCartId.ToString();
        }
      }
      return HttpContext.Current.Session[CartSessionKey].ToString();
    }

    public List<CartItem> GetCartItems()
    {
      ShoppingCartId = GetCartId();

      return _db.ShoppingCartItems.Where(
          c => c.CartId == ShoppingCartId).ToList();
    }
  }
}
```

The `AddToCart` method enables individual products to be included in the shopping cart based on the product `ID`. The product is added to the cart, or if the cart already contains an item for that product, the quantity is incremented.

The `GetCartId` method returns the cart `ID` for the user. The cart `ID` is used to track the items that a user has in their shopping cart. If the user does not have an existing cart `ID`, a new cart `ID` is created for them. If the user is signed in as a registered user, the cart `ID` is set to their user name. However, if the user is not signed in, the cart `ID` is set to a unique value (a GUID). A GUID ensures that only one cart is created for each user, based on session.

The `GetCartItems` method returns a list of shopping cart items for the user. Later in this tutorial, you will see that model binding is used to display the cart items in the shopping cart using the `GetCartItems` method.

## Creating the Add-To-Cart Functionality

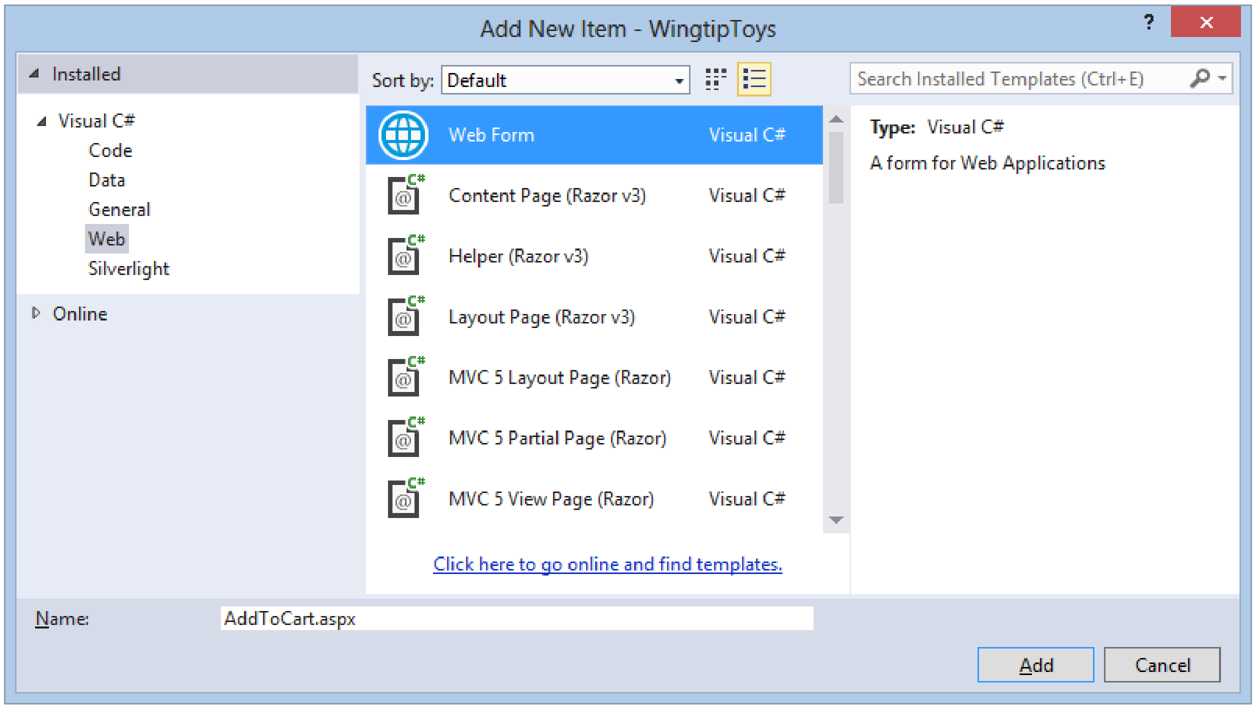
As mentioned earlier, you will create a processing page named *AddToCart.aspx* that will be used to add new products to the shopping cart of the user. This page will call the `AddToCart` method in the `ShoppingCart` class that you just created. The *AddToCart.aspx* page will expect that a product `ID` is passed to it. This product `ID` will be used when calling the `AddToCart` method in the `ShoppingCart` class.

**Note**

You will be modifying the code-behind (*AddToCart.aspx.cs*) for this page, not the page UI (*AddToCart.aspx*).

To create the Add-To-Cart functionality:

1. In **Solution Explorer**, right-click the **WingtipToys** project, click **Add** -> **New Item**. The **Add New Item** dialog box is displayed.
2. Add a standard new page (Web Form) to the application named *AddToCart.aspx*.


Add New Item - WingtipToys
Installed
Visual C#
Code
Data
General
Web
Silverlight
Online
Sort by: Default
Search Installed Templates (Ctrl+E)
Web Form Visual C#
Content Page (Razor v3) Visual C#
Helper (Razor v3) Visual C#
Layout Page (Razor v3) Visual C#
MVC 5 Layout Page (Razor) Visual C#
MVC 5 Partial Page (Razor) Visual C#
MVC 5 View Page (Razor) Visual C#
Click here to go online and find templates.
Type: Visual C#
A form for Web Applications
Add
Cancel

Name: AddToCart.aspx

3. In **Solution Explorer**, right-click the *AddToCart.aspx* page and then click **View Code**. The *AddToCart.aspx.cs* code-behind file is opened in the editor.
4. Replace the existing code in the *AddToCart.aspx.cs* code-behind with the following:

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

namespace WingtipToys
{
  public partial class AddToCart : System.Web.UI.Page
  {
    protected void Page_Load(object sender, EventArgs e)
    {
      string rawId = Request.QueryString["ProductID"];
      int productId;
      if (!String.IsNullOrEmpty(rawId) && int.TryParse(rawId, out productId))
      {
        using (ShoppingCartActions usersShoppingCart = new
ShoppingCartActions())
        {
          usersShoppingCart.AddToCart(Convert.ToInt16(rawId));
        }

      }
      else
```

```
        {
          Debug.Fail("ERROR : We should never get to AddToCart.aspx without a
ProductId.");
          throw new Exception("ERROR : It is illegal to load AddToCart.aspx
without setting a ProductId.");
        }
        Response.Redirect("ShoppingCart.aspx");
      }
    }
}
```

When the *AddToCart.aspx* page is loaded, the product `ID` is retrieved from the query string. Next, an instance of the shopping cart class is created and used to call the `AddToCart` method that you added earlier in this tutorial. The `AddToCart` method, contained in the *ShoppingCartActions.cs* file, includes the logic to add the selected product to the shopping cart or increment the product quantity of the selected product. If the product hasn't been added to the shopping cart, the product is added to the `CartItem` table of the database. If the product has already been added to the shopping cart and the user adds an additional item of the same product, the product quantity is incremented in the `CartItem` table. Finally, the page redirects back to the *ShoppingCart.aspx* page that you'll add in the next step, where the user sees an updated list of items in the cart.

As previously mentioned, a user `ID` is used to identify the products that are associated with a specific user. This `ID` is added to a row in the `CartItem` table each time the user adds a product to the shopping cart.

## Creating the Shopping Cart UI

The *ShoppingCart.aspx* page will display the products that the user has added to their shopping cart. It will also provide the ability to add, remove and update items in the shopping cart.

1. In **Solution Explorer**, right-click **WingtipToys**, click **Add** -> **New Item**.
   The **Add New Item** dialog box is displayed.
2. Add a new page (Web Form) that includes a master page by selecting **Web Form using Master Page**. Name the new page *ShoppingCart.aspx*.
3. Select **Site.Master** to attach the master page to the newly created *.aspx* page.
4. In the *ShoppingCart.aspx* page, replace the existing markup with the following markup:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Site.Master"
AutoEventWireup="true" CodeBehind="ShoppingCart.aspx.cs"
Inherits="WingtipToys.ShoppingCart" %>
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent" runat="server">
    <div id="ShoppingCartTitle" runat="server" class="ContentHead"><h1>Shopping
Cart</h1></div>
    <asp:GridView ID="CartList" runat="server" AutoGenerateColumns="False"
ShowFooter="True" GridLines="Vertical" CellPadding="4"
        ItemType="WingtipToys.Models.CartItem"
SelectMethod="GetShoppingCartItems"
        CssClass="table table-striped table-bordered" >
        <Columns>
        <asp:BoundField DataField="ProductID" HeaderText="ID"
SortExpression="ProductID" />
        <asp:BoundField DataField="Product.ProductName" HeaderText="Name" />
        <asp:BoundField DataField="Product.UnitPrice" HeaderText="Price (each)"
DataFormatString="{0:c}"/>
        <asp:TemplateField   HeaderText="Quantity">
                <ItemTemplate>
                    <asp:TextBox ID="PurchaseQuantity" Width="40"
runat="server" Text="<%#: Item.Quantity %>"></asp:TextBox>
                </ItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="Item Total">
                <ItemTemplate>
                    <%#: String.Format("{0:c}",
((Convert.ToDouble(Item.Quantity)) *
Convert.ToDouble(Item.Product.UnitPrice)))%>
                </ItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="Remove Item">
                <ItemTemplate>
                    <asp:CheckBox id="Remove" runat="server"></asp:CheckBox>
                </ItemTemplate>
        </asp:TemplateField>
        </Columns>
    </asp:GridView>
    <div>
        <p></p>
        <strong>
            <asp:Label ID="LabelTotalText" runat="server" Text="Order Total:
"></asp:Label>
            <asp:Label ID="lblTotal" runat="server"
EnableViewState="false"></asp:Label>
        </strong>
    </div>
    <br />
</asp:Content>
```

The *ShoppingCart.aspx* page includes a **GridView** control named `CartList`. This control uses model binding to bind the shopping cart data from the database to the **GridView** control. When you set the `ItemType` property of the **GridView** control, the data-binding expression `Item` is available in the markup of the control and the control becomes strongly typed. As mentioned earlier in this tutorial series, you can select details of the `Item` object using IntelliSense. To configure a data control to use model binding to select data, you set the `SelectMethod` property of the control. In the markup above, you set the `SelectMethod` to use the GetShoppingCartItems method which returns a list of `CartItem` objects. The **GridView** data control calls the method at the appropriate time in the page life cycle and automatically binds the returned data. The `GetShoppingCartItems` method must still be added.

## Retrieving the Shopping Cart Items

Next, you add code to the *ShoppingCart.aspx.cs* code-behind to retrieve and populate the Shopping Cart UI.

1. In **Solution Explorer**, right-click the *ShoppingCart.aspx* page and then click **View Code**. The *ShoppingCart.aspx.cs* code-behind file is opened in the editor.

- Replace the existing code with the following:

```
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 WingtipToys.Logic;

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

    }

    public List<CartItem> GetShoppingCartItems()
    {
      ShoppingCartActions actions = new ShoppingCartActions();
      return actions.GetCartItems();
    }
  }
}
```

As mentioned above, the `GridView` data control calls the `GetShoppingCartItems` method at the appropriate time in the page life cycle and automatically binds the returned data. The `GetShoppingCartItems` method creates an instance of the `ShoppingCartActions` object. Then, the code uses that instance to return the items in the cart by calling the `GetCartItems` method.

## Adding Products to the Shopping Cart

When either the *ProductList.aspx* or the *ProductDetails.aspx* page is displayed, the user will be able to add the product to the shopping cart using a link. When they click the link, the application navigates to the processing page named *AddToCart.aspx*. The *AddToCart.aspx* page will call the `AddToCart` method in the `ShoppingCart` class that you added earlier in this tutorial.

Now, you'll add an **Add to Cart** link to both the *ProductList.aspx* page and the *ProductDetails.aspx* page. This link will include the product `ID` that is retrieved from the database.

1. In **Solution Explorer**, find and open the page named *ProductList.aspx*.
2. Add the markup highlighted in yellow to the *ProductList.aspx* page so that the entire page appears as follows:

```
<%@ Page Title="Products" Language="C#" MasterPageFile="~/Site.Master" 
AutoEventWireup="true"
         CodeBehind="ProductList.aspx.cs" Inherits="WingtipToys.ProductList" %>
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent" runat="server">
    <section>
        <div>
            <hgroup>
```

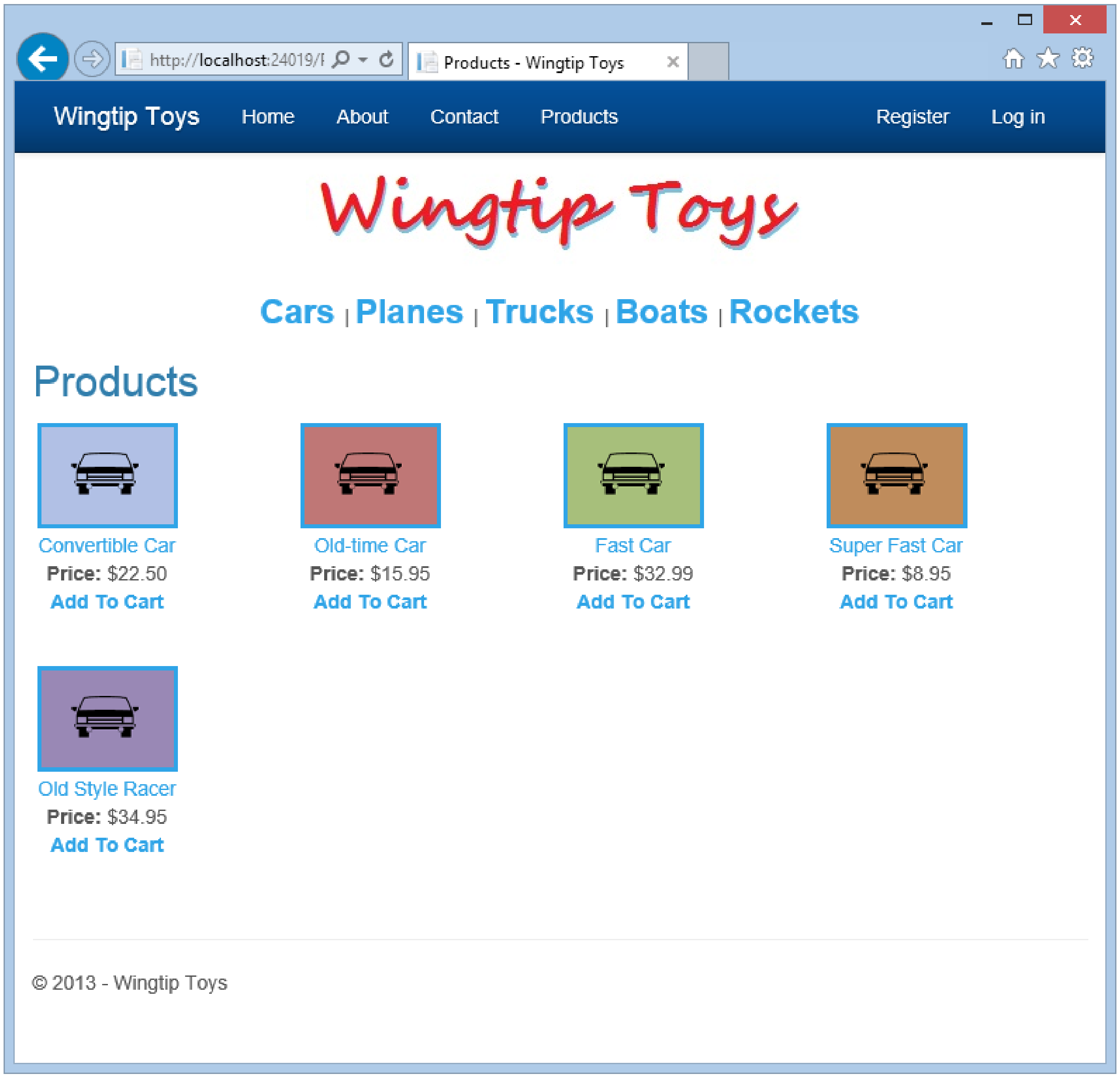
```
                <h2><%: Page.Title %></h2>
            </hgroup>

            <asp:ListView ID="productList" runat="server"
                DataKeyNames="ProductID" GroupItemCount="4"
                ItemType="WingtipToys.Models.Product"
SelectMethod="GetProducts">
                <EmptyDataTemplate>
                    <table runat="server">
                        <tr>
                            <td>No data was returned.</td>
                        </tr>
                    </table>
                </EmptyDataTemplate>
                <EmptyItemTemplate>
                    <td runat="server" />
                </EmptyItemTemplate>
                <GroupTemplate>
                    <tr id="itemPlaceholderContainer" runat="server">
                        <td id="itemPlaceholder" runat="server"></td>
                    </tr>
                </GroupTemplate>
                <ItemTemplate>
                    <td runat="server">
                        <table>
                            <tr>
                                <td>
                                    <a
href="ProductDetails.aspx?productID=<%#:Item.ProductID%>">
                                        <img
src="/Catalog/Images/Thumbs/<%#:Item.ImagePath%>"
                                            width="100" height="75"
style="border: solid" /></a>
                                </td>
                            </tr>
                            <tr>
                                <td>
                                    <a
href="ProductDetails.aspx?productID=<%#:Item.ProductID%>">
                                        <span>
                                            <%#:Item.ProductName%>
                                        </span>
                                    </a>
                                    <br />
                                    <span>
                                        <b>Price:
</b><%#:String.Format("{0:c}", Item.UnitPrice)%>
                                    </span>
                                    <br />
                                    <a
href="/AddToCart.aspx?productID=<%#:Item.ProductID %>">
                                        <span class="ProductListItem">
                                            <b>Add To Cart<b>
                                        </span>
                                    </a>
                                </td>
                            </tr>
                            <tr>
                                <td>&nbsp;</td>
                            </tr>
                        </table>
                        </p>
                    </td>
                </ItemTemplate>
                <LayoutTemplate>
                    <table runat="server" style="width:100%;">
                        <tbody>
                            <tr runat="server">
                                <td runat="server">
                                    <table id="groupPlaceholderContainer" 
runat="server" style="width:100%">
                                        <tr id="groupPlaceholder" 
runat="server"></tr>
                                    </table>
                                </td>
                            </tr>
                            <tr runat="server">
                                <td runat="server"></td>
                            </tr>
                            <tr></tr>
                        </tbody>
                    </table>
                </LayoutTemplate>
            </asp:ListView>
        </div>
    </section>
</asp:Content>
```

## Testing the Shopping Cart

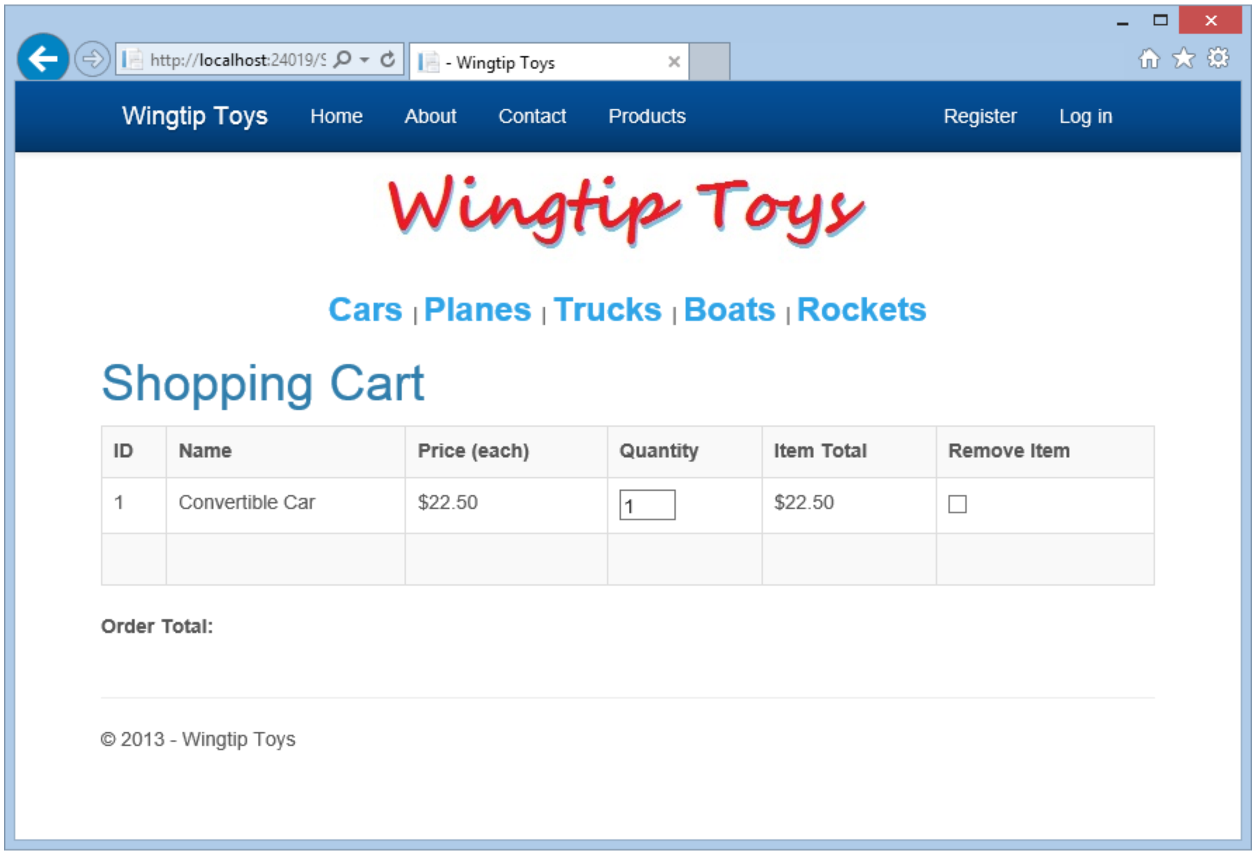
Run the application to see how you add products to the shopping cart.

1. Press **F5** to run the application.
   After the project recreates the database, the browser will open and show the *Default.aspx* page.
2. Select **Cars** from the category navigation menu.
   The *ProductList.aspx* page is displayed showing only products included in the "Cars"

category.


http://localhost:24019/
Products - Wingtip Toys
Wingtip Toys
Home
About
Contact
Products
Register
Log in
Wingtip Toys
Cars | Planes | Trucks | Boats | Rockets
Products
Convertible Car
Price: $22.50
Add To Cart
Old-time Car
Price: $15.95
Add To Cart
Fast Car
Price: $32.99
Add To Cart
Super Fast Car
Price: $8.95
Add To Cart
Old Style Racer
Price: $34.95
Add To Cart
© 2013 - Wingtip Toys

3. Click the **Add to Cart** link next to the first product listed (the convertible car).
   The *ShoppingCart.aspx* page is displayed, showing the selection in your shopping cart.



Shopping Cart

| ID | Name | Price (each) | Quantity | Item Total | Remove Item |
|---|---|---|---|---|---|
| 1 | Convertible Car | $22.50 | 1 | $22.50 | ☐ |
| | | | | | |

Order Total:

© 2013 - Wingtip Toys

4. View additional products by selecting **Planes** from the category navigation menu.
5. Click the **Add to Cart** link next to the first product listed.
   The *ShoppingCart.aspx* page is displayed with the additional item.
6. Close the browser.

## Calculating and Displaying the Order Total

In addition to adding products to the shopping cart, you will add a `GetTotal` method to the `ShoppingCart` class and display the total order amount in the shopping cart page.

1. In **Solution Explorer**, open the *ShoppingCartActions.cs* file in the *Logic* folder.
2. Add the following `GetTotal` method highlighted in yellow to the `ShoppingCart` class, so that the class appears as follows:

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

namespace WingtipToys.Logic
{
  public class ShoppingCartActions : IDisposable
  {
    public string ShoppingCartId { get; set; }

    private ProductContext _db = new ProductContext();

    public const string CartSessionKey = "CartId";

    public void AddToCart(int id)
    {
      // Retrieve the product from the database.
      ShoppingCartId = GetCartId();

      var cartItem = _db.ShoppingCartItems.SingleOrDefault(
          c => c.CartId == ShoppingCartId
          && c.ProductId == id);
      if (cartItem == null)
      {
        // Create a new cart item if no cart item exists.
        cartItem = new CartItem
        {
          ItemId = Guid.NewGuid().ToString(),
          ProductId = id,
          CartId = ShoppingCartId,
          Product = _db.Products.SingleOrDefault(
           p => p.ProductID == id),
          Quantity = 1,
          DateCreated = DateTime.Now
        };

         db.ShoppingCartItems.Add(cartItem);
      }
      else
      {
        // If the item does exist in the cart,
        // then add one to the quantity.
        cartItem.Quantity++;
      }
      _db.SaveChanges();
    }

    public void Dispose()
    {
      if (_db != null)
      {
        _db.Dispose();
        _db = null;
      }
    }

    public string GetCartId()
    {
      if (HttpContext.Current.Session[CartSessionKey] == null)
      {
        if (!string.IsNullOrWhiteSpace(HttpContext.Current.User.Identity.Name))
        {
          HttpContext.Current.Session[CartSessionKey] =
HttpContext.Current.User.Identity.Name;
        }
        else
        {
          // Generate a new random GUID using System.Guid class.
          Guid tempCartId = Guid.NewGuid();
          HttpContext.Current.Session[CartSessionKey] = tempCartId.ToString();
        }
      }
      return HttpContext.Current.Session[CartSessionKey].ToString();
    }

    public List<CartItem> GetCartItems()
    {
      ShoppingCartId = GetCartId();

      return  db.ShoppingCartItems.Where(
          c => c.CartId == ShoppingCartId).ToList();
    }

    public decimal GetTotal()
    {
      ShoppingCartId = GetCartId();
      // Multiply product price by quantity of that product to get
      // the current price for each of those products in the cart.
      // Sum all product price totals to get the cart total.
      decimal? total = decimal.Zero;
      total = (decimal?)(from cartItems in _db.ShoppingCartItems
                         where cartItems.CartId == ShoppingCartId
                         select (int?)cartItems.Quantity *
                         cartItems.Product.UnitPrice).Sum();
      return total ?? decimal.Zero;
    }
  }
}
```

First, the `GetTotal` method gets the ID of the shopping cart for the user. Then the method gets the cart total by multiplying the product price by the product quantity for each product listed in the cart.

**Note**

The above code uses the nullable type "`int?`". Nullable types can represent all the values of an underlying type, and also as a null value. For more information see, Using Nullable Types.

## Modify the Shopping Cart Display

Next you'll modify the code for the *ShoppingCart.aspx* page to call the `GetTotal` method and display that total on the *ShoppingCart.aspx* page when the page loads.

1. In **Solution Explorer**, right-click the *ShoppingCart.aspx* page and select **View Code**.
2. In the *ShoppingCart.aspx.cs* file, update the `Page_Load` handler by adding the following code highlighted in yellow:

```
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 WingtipToys.Logic;

namespace WingtipToys
{
  public partial class ShoppingCart : System.Web.UI.Page
```

```
    {
      protected void Page_Load(object sender, EventArgs e)
      {
        using (ShoppingCartActions usersShoppingCart = new ShoppingCartActions())
        {
          decimal cartTotal = 0;
          cartTotal = usersShoppingCart.GetTotal();
          if (cartTotal > 0)
          {
            // Display Total.
            lblTotal.Text = String.Format("{0:c}", cartTotal);
          }
          else
          {
            LabelTotalText.Text = "";
            lblTotal.Text = "";
            ShoppingCartTitle.InnerText = "Shopping Cart is Empty";
          }
        }
      }

      public List<CartItem> GetShoppingCartItems()
      {
        ShoppingCartActions actions = new ShoppingCartActions();
        return actions.GetCartItems();
      }
    }
}
```

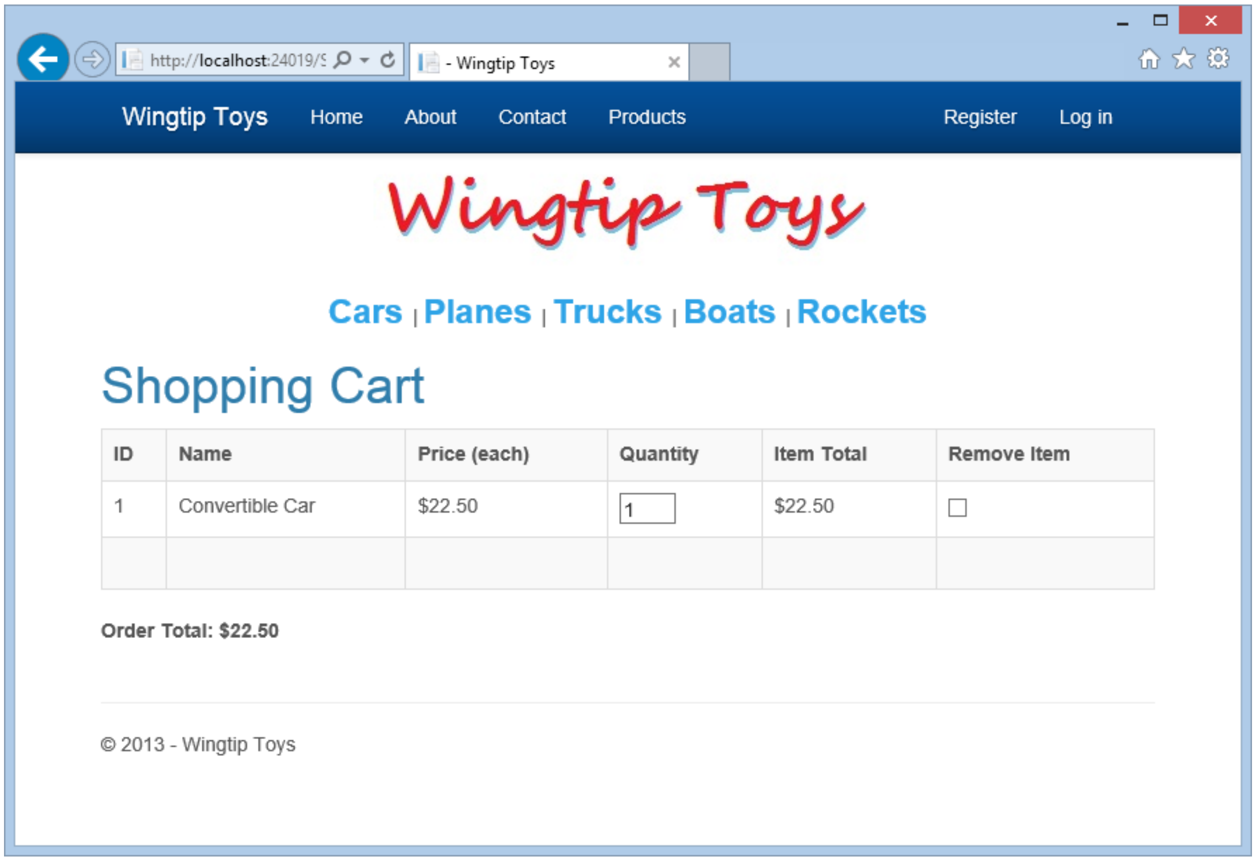
When the *ShoppingCart.aspx* page loads, it loads the shopping cart object and then retrieves the shopping cart total by calling the `GetTotal` method of the `ShoppingCart` class. If the shopping cart is empty, a message to that effect is displayed.

## Testing the Shopping Cart Total

Run the application now to see how you can not only add a product to the shopping cart, but you can see the shopping cart total.

1. Press **F5** to run the application.
   The browser will open and show the *Default.aspx* page.
2. Select **Cars** from the category navigation menu.

3. Click the **Add To Cart** link next to the first product.
   The *ShoppingCart.aspx* page is displayed with the order total.



Shopping Cart

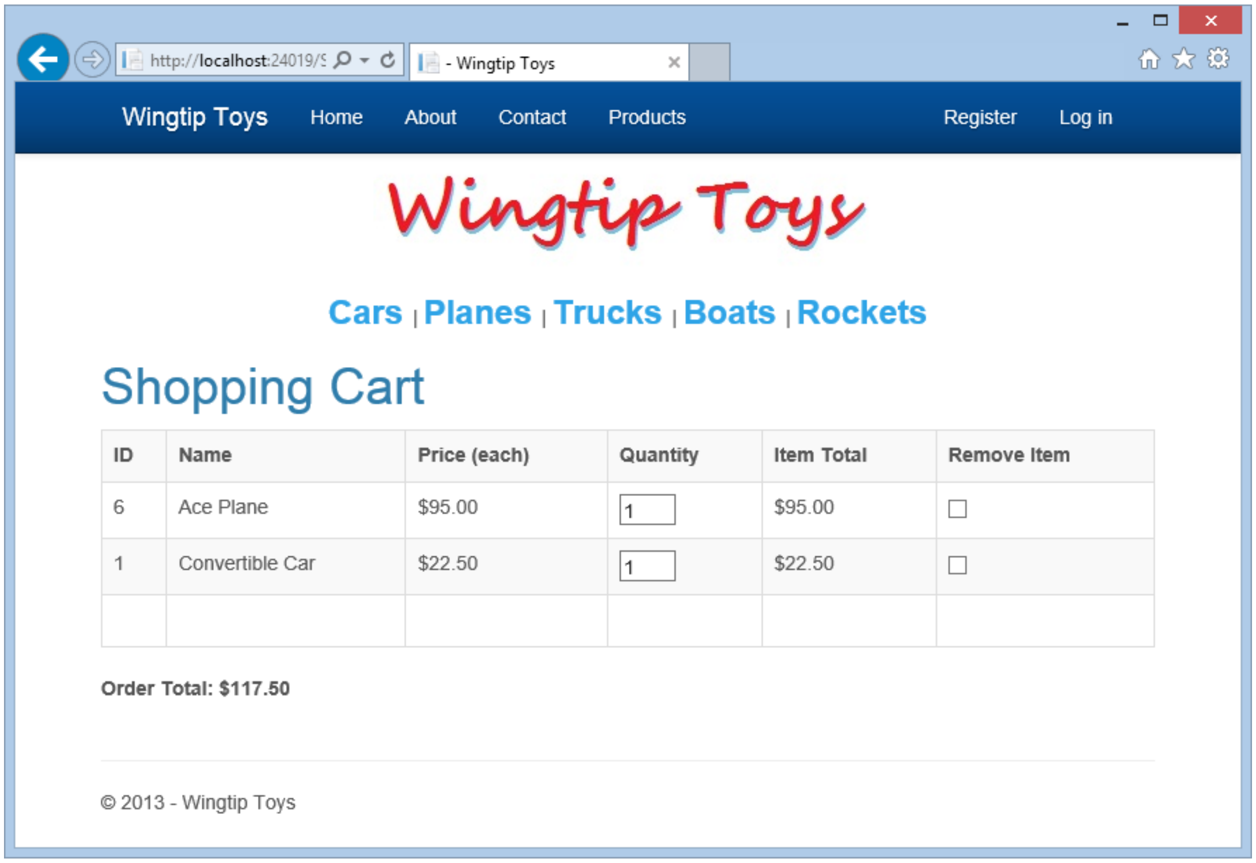
| ID | Name | Price (each) | Quantity | Item Total | Remove Item |
|---|---|---|---|---|---|
| 1 | Convertible Car | $22.50 | 1 | $22.50 | ☐ |
| | | | | | |

Order Total: $22.50

© 2013 - Wingtip Toys

4. Add some other products (for example, a plane) to the cart.

5. The *ShoppingCart.aspx* page is displayed with an updated total for all the products you've added.



Shopping Cart

| ID | Name | Price (each) | Quantity | Item Total | Remove Item |
|---|---|---|---|---|---|
| 6 | Ace Plane | $95.00 | 1 | $95.00 | ☐ |
| 1 | Convertible Car | $22.50 | 1 | $22.50 | ☐ |
| | | | | | |

Order Total: $117.50

© 2013 - Wingtip Toys

6. Stop the running app by closing the browser window.

## Adding Update and Checkout Buttons to the Shopping Cart

To allow the users to modify the shopping cart, you'll add an **Update** button and a **Checkout** button to the shopping cart page. The **Checkout** button is not used until later in this tutorial series.

- In **Solution Explorer**, open the *ShoppingCart.aspx* page in the root of the web application project.
- To add the **Update** button and the **Checkout** button to the *ShoppingCart.aspx* page, add the markup highlighted in yellow to the existing markup, as shown in the following code:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Site.Master"
AutoEventWireup="true" CodeBehind="ShoppingCart.aspx.cs"
Inherits="WingtipToys.ShoppingCart" %>
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent" runat="server">
    <div id="ShoppingCartTitle" runat="server" class="ContentHead"><h1>Shopping
Cart</h1></div>
    <asp:GridView ID="CartList" runat="server" AutoGenerateColumns="False"
ShowFooter="True" GridLines="Vertical" CellPadding="4"
        ItemType="WingtipToys.Models.CartItem"
SelectMethod="GetShoppingCartItems"
        CssClass="table table-striped table-bordered" >
        <Columns>
        <asp:BoundField DataField="ProductID" HeaderText="ID"
SortExpression="ProductID" />
        <asp:BoundField DataField="Product.ProductName" HeaderText="Name" />
        <asp:BoundField DataField="Product.UnitPrice" HeaderText="Price (each)"
DataFormatString="{0:c}"/>
        <asp:TemplateField   HeaderText="Quantity">
                <ItemTemplate>
                    <asp:TextBox ID="PurchaseQuantity" Width="40"
runat="server" Text="<%#: Item.Quantity %>"></asp:TextBox>
                </ItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="Item Total">
                <ItemTemplate>
                    <%#: String.Format("{0:c}",
((Convert.ToDouble(Item.Quantity)) *
Convert.ToDouble(Item.Product.UnitPrice)))%>
                </ItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="Remove Item">
                <ItemTemplate>
                    <asp:CheckBox id="Remove" runat="server"></asp:CheckBox>
                </ItemTemplate>
        </asp:TemplateField>
        </Columns>
    </asp:GridView>
    <div>
        <p></p>
        <strong>
            <asp:Label ID="LabelTotalText" runat="server" Text="Order Total:
"></asp:Label>
            <asp:Label ID="lblTotal" runat="server"
EnableViewState="false"></asp:Label>
        </strong>
    </div>
  <br />
    <table>
    <tr>
      <td>
        <asp:Button ID="UpdateBtn" runat="server" Text="Update"
OnClick="UpdateBtn_Click" />
      </td>
      <td>
        <!--Checkout Placeholder -->
      </td>
    </tr>
    </table>
</asp:Content>
```

When the user clicks the **Update** button, the `UpdateBtn_Click` event handler will be called. This event handler will call the code that you'll add in the next step.

Next, you can update the code contained in the *ShoppingCart.aspx.cs* file to loop through the cart items and call the `RemoveItem` and `UpdateItem` methods.

1. In **Solution Explorer**, open the *ShoppingCart.aspx.cs* file in the root of the web application project.
2. Add the following code sections highlighted in yellow to the *ShoppingCart.aspx.cs* file:

```
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 WingtipToys.Logic;
using System.Collections.Specialized;
using System.Collections;
using System.Web.ModelBinding;

namespace WingtipToys
{
  public partial class ShoppingCart : System.Web.UI.Page
  {
    protected void Page_Load(object sender, EventArgs e)
    {
      using (ShoppingCartActions usersShoppingCart = new ShoppingCartActions())
      {
        decimal cartTotal = 0;
        cartTotal = usersShoppingCart.GetTotal();
        if (cartTotal > 0)
        {
          // Display Total.
          lblTotal.Text = String.Format("{0:c}", cartTotal);
        }
        else
        {
          LabelTotalText.Text = "";
          lblTotal.Text = "";
          ShoppingCartTitle.InnerText = "Shopping Cart is Empty";
          UpdateBtn.Visible = false;
        }
      }
    }

    public List<CartItem> GetShoppingCartItems()
    {
      ShoppingCartActions actions = new ShoppingCartActions();
      return actions.GetCartItems();
    }

    public List<CartItem> UpdateCartItems()
    {
      using (ShoppingCartActions usersShoppingCart = new ShoppingCartActions())
      {
        String cartId = usersShoppingCart.GetCartId();

        ShoppingCartActions.ShoppingCartUpdates[] cartUpdates = new
ShoppingCartActions.ShoppingCartUpdates[CartList.Rows.Count];
        for (int i = 0; i < CartList.Rows.Count; i++)
        {
          IOrderedDictionary rowValues = new OrderedDictionary();
          rowValues = GetValues(CartList.Rows[i]);
          cartUpdates[i].ProductId = Convert.ToInt32(rowValues["ProductID"]);

          CheckBox cbRemove = new CheckBox();
          cbRemove = (CheckBox)CartList.Rows[i].FindControl("Remove");
          cartUpdates[i].RemoveItem = cbRemove.Checked;

          TextBox quantityTextBox = new TextBox();
          quantityTextBox =
(TextBox)CartList.Rows[i].FindControl("PurchaseQuantity");
```

```
            cartUpdates[i].PurchaseQuantity =
Convert.ToInt16(quantityTextBox.Text.ToString());
          }
          usersShoppingCart.UpdateShoppingCartDatabase(cartId, cartUpdates);
          CartList.DataBind();
          lblTotal.Text = String.Format("{0:c}", usersShoppingCart.GetTotal());
          return usersShoppingCart.GetCartItems();
        }
      }

    public static IOrderedDictionary GetValues(GridViewRow row)
    {
      IOrderedDictionary values = new OrderedDictionary();
      foreach (DataControlFieldCell cell in row.Cells)
      {
        if (cell.Visible)
        {
          // Extract values from the cell.
          cell.ContainingField.ExtractValuesFromCell(values, cell,
row.RowState, true);
        }
      }
      return values;
    }

    protected void UpdateBtn_Click(object sender, EventArgs e)
    {
      UpdateCartItems();
    }
  }
}
```

When the user clicks the **Update** button on the *ShoppingCart.aspx* page, the UpdateCartItems method is called. The UpdateCartItems method gets the updated values for each item in the shopping cart. Then, the UpdateCartItems method calls the `UpdateShoppingCartDatabase` method (added and explained in the next step) to either add or remove items from the shopping cart. Once the database has been updated to reflect the updates to the shopping cart, the **GridView** control is updated on the shopping cart page by calling the `DataBind` method for the **GridView**. Also, the total order amount on the shopping cart page is updated to reflect the updated list of items.

## Updating and Removing Shopping Cart Items

On the *ShoppingCart.aspx* page, you can see controls have been added for updating the quantity of an item and removing an item. Now, add the code that will make these controls work.

1. In **Solution Explorer**, open the *ShoppingCartActions.cs* file in the *Logic* folder.
- Add the following code highlighted in yellow to the *ShoppingCartActions.cs* class file:

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

namespace WingtipToys.Logic
```

```
{
  public class ShoppingCartActions : IDisposable
  {
    public string ShoppingCartId { get; set; }

    private ProductContext _db = new ProductContext();

    public const string CartSessionKey = "CartId";

    public void AddToCart(int id)
    {
      // Retrieve the product from the database.
      ShoppingCartId = GetCartId();

      var cartItem = _db.ShoppingCartItems.SingleOrDefault(
          c => c.CartId == ShoppingCartId
          && c.ProductId == id);
      if (cartItem == null)
      {
        // Create a new cart item if no cart item exists.
        cartItem = new CartItem
        {
          ItemId = Guid.NewGuid().ToString(),
          ProductId = id,
          CartId = ShoppingCartId,
          Product = _db.Products.SingleOrDefault(
           p => p.ProductID == id),
          Quantity = 1,
          DateCreated = DateTime.Now
        };

        _db.ShoppingCartItems.Add(cartItem);
      }
      else
      {
        // If the item does exist in the cart,
        // then add one to the quantity.
        cartItem.Quantity++;
      }
      _db.SaveChanges();
    }

    public void Dispose()
    {
      if (_db != null)
      {
        _db.Dispose();
        _db = null;
      }
    }

    public string GetCartId()
    {
      if (HttpContext.Current.Session[CartSessionKey] == null)
      {
        if (!string.IsNullOrWhiteSpace(HttpContext.Current.User.Identity.Name))
        {
          HttpContext.Current.Session[CartSessionKey] =
HttpContext.Current.User.Identity.Name;
        }
        else
        {
          // Generate a new random GUID using System.Guid class.
          Guid tempCartId = Guid.NewGuid();
          HttpContext.Current.Session[CartSessionKey] = tempCartId.ToString();
        }
      }
      return HttpContext.Current.Session[CartSessionKey].ToString();
    }

    public List<CartItem> GetCartItems()
    {
      ShoppingCartId = GetCartId();

      return _db.ShoppingCartItems.Where(
          c => c.CartId == ShoppingCartId).ToList();
    }

    public decimal GetTotal()
    {
      ShoppingCartId = GetCartId();
      // Multiply product price by quantity of that product to get
      // the current price for each of those products in the cart.
      // Sum all product price totals to get the cart total.
      decimal? total = decimal.Zero;
      total = (decimal?)(from cartItems in _db.ShoppingCartItems
                         where cartItems.CartId == ShoppingCartId
                         select (int?)cartItems.Quantity *
                         cartItems.Product.UnitPrice).Sum();
      return total ?? decimal.Zero;
    }

    public ShoppingCartActions GetCart(HttpContext context)
    {
      using (var cart = new ShoppingCartActions())
      {
        cart.ShoppingCartId = cart.GetCartId();
        return cart;
      }
    }

    public void UpdateShoppingCartDatabase(String cartId, ShoppingCartUpdates[]
CartItemUpdates)
    {
      using (var db = new WingtipToys.Models.ProductContext())
      {
        try
        {
          int CartItemCount = CartItemUpdates.Count();
          List<CartItem> myCart = GetCartItems();
          foreach (var cartItem in myCart)
          {
            // Iterate through all rows within shopping cart list
            for (int i = 0; i < CartItemCount; i++)
            {
              if (cartItem.Product.ProductID == CartItemUpdates[i].ProductId)
              {
                if (CartItemUpdates[i].PurchaseQuantity < 1 ||
CartItemUpdates[i].RemoveItem == true)
                {
                  RemoveItem(cartId, cartItem.ProductId);
                }
                else
                {
                  UpdateItem(cartId, cartItem.ProductId,
CartItemUpdates[i].PurchaseQuantity);
```

```
                }
              }
            }
          }
        }
        catch (Exception exp)
        {
          throw new Exception("ERROR: Unable to Update Cart Database - " +
exp.Message.ToString(), exp);
        }
      }
    }

    public void RemoveItem(string removeCartID, int removeProductID)
    {
      using (var _db = new WingtipToys.Models.ProductContext())
      {
        try
        {
          var myItem = (from c in _db.ShoppingCartItems where c.CartId ==
removeCartID && c.Product.ProductID == removeProductID select
c).FirstOrDefault();
          if (myItem != null)
          {
            // Remove Item.
            _db.ShoppingCartItems.Remove(myItem);
            _db.SaveChanges();
          }
        }
        catch (Exception exp)
        {
          throw new Exception("ERROR: Unable to Remove Cart Item - " +
exp.Message.ToString(), exp);
        }
      }
    }

    public void UpdateItem(string updateCartID, int updateProductID, int
quantity)
    {
      using (var _db = new WingtipToys.Models.ProductContext())
      {
        try
        {
          var myItem = (from c in _db.ShoppingCartItems where c.CartId ==
updateCartID && c.Product.ProductID == updateProductID select
c).FirstOrDefault();
          if (myItem != null)
          {
            myItem.Quantity = quantity;
            _db.SaveChanges();
          }
        }
        catch (Exception exp)
        {
          throw new Exception("ERROR: Unable to Update Cart Item - " +
exp.Message.ToString(), exp);
        }
      }
    }

    public void EmptyCart()
    {
      ShoppingCartId = GetCartId();
      var cartItems = _db.ShoppingCartItems.Where(
          c => c.CartId == ShoppingCartId);
      foreach (var cartItem in cartItems)
      {
        _db.ShoppingCartItems.Remove(cartItem);
      }
      // Save changes.
      db.SaveChanges();
    }

    public int GetCount()
    {
      ShoppingCartId = GetCartId();

      // Get the count of each item in the cart and sum them up
      int? count = (from cartItems in _db.ShoppingCartItems
                    where cartItems.CartId == ShoppingCartId
                    select (int?)cartItems.Quantity).Sum();
      // Return 0 if all entries are null
      return count ?? 0;
    }

    public struct ShoppingCartUpdates
    {
      public int ProductId;
      public int PurchaseQuantity;
      public bool RemoveItem;
    }
  }
}
```

The `UpdateShoppingCartDatabase` method, called from the `UpdateCartItems` method on the *ShoppingCart.aspx.cs* page, contains the logic to either update or remove items from the shopping cart. The `UpdateShoppingCartDatabase` method iterates through all the rows within the shopping cart list. If a shopping cart item has been marked to be removed, or the quantity is less than one, the `RemoveItem` method is called. Otherwise, the shopping cart item is checked for updates when the `UpdateItem` method is called. After the shopping cart item has been removed or updated, the database changes are saved.

The `ShoppingCartUpdates` structure is used to hold all the shopping cart items. The `UpdateShoppingCartDatabase` method uses the `ShoppingCartUpdates` structure to determine if any of the items need to be updated or removed.

In the next tutorial, you will use the `EmptyCart` method to clear the shopping cart after purchasing products. But for now, you will use the `GetCount` method that you just added to the *ShoppingCartActions.cs* file to determine how many items are in the shopping cart.

## Adding a Shopping Cart Counter

To allow the user to view the total number of items in the shopping cart, you will add a counter to the *Site.Master* page. This counter will also act as a link to the shopping cart.

1. In **Solution Explorer**, open the *Site.Master* page.

- Modify the markup by adding the shopping cart counter link as shown in yellow to the navigation section so it appears as follows:

```
  <ul class="nav navbar-nav">
      <li><a runat="server" href="~/">Home</a></li>
      <li><a runat="server" href="~/About">About</a></li>
      <li><a runat="server" href="~/Contact">Contact</a></li>
      <li><a runat="server" href="~/ProductList">Products</a></li>
      <li><a runat="server" href="~/ShoppingCart" 
ID="cartCount">&nbsp;</a></li>
  </ul>
```

- Next, update the code-behind of the *Site.Master.cs* file by adding the code highlighted in yellow as follows:

```
using System;
using System.Collections.Generic;
using System.Security.Claims;
using System.Security.Principal;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Linq;
using WingtipToys.Models;
using WingtipToys.Logic;

namespace WingtipToys
{
    public partial class SiteMaster : MasterPage
    {
        private const string AntiXsrfTokenKey = "__AntiXsrfToken";
        private const string AntiXsrfUserNameKey = "__AntiXsrfUserName";
        private string _antiXsrfTokenValue;

        protected void Page_Init(object sender, EventArgs e)
        {
            // The code below helps to protect against XSRF attacks
            var requestCookie = Request.Cookies[AntiXsrfTokenKey];
            Guid requestCookieGuidValue;
            if (requestCookie != null && Guid.TryParse(requestCookie.Value, out 
requestCookieGuidValue))
            {
                // Use the Anti-XSRF token from the cookie
                 antiXsrfTokenValue = requestCookie.Value;
                Page.ViewStateUserKey = _antiXsrfTokenValue;
            }
            else
            {
                // Generate a new Anti-XSRF token and save to the cookie
                _antiXsrfTokenValue = Guid.NewGuid().ToString("N");
                Page.ViewStateUserKey = _antiXsrfTokenValue;

                var responseCookie = new HttpCookie(AntiXsrfTokenKey)
                {
                    HttpOnly = true,
                    Value = _antiXsrfTokenValue
                };
                if (FormsAuthentication.RequireSSL && 
Request.IsSecureConnection)
```

```
                {
                    responseCookie.Secure = true;
                }
                Response.Cookies.Set(responseCookie);
            }

            Page.PreLoad += master_Page_PreLoad;
        }

        protected void master_Page_PreLoad(object sender, EventArgs e)
        {
            if (!IsPostBack)
            {
                // Set Anti-XSRF token
                ViewState[AntiXsrfTokenKey] = Page.ViewStateUserKey;
                ViewState[AntiXsrfUserNameKey] = Context.User.Identity.Name ??
String.Empty;
            }
            else
            {
                // Validate the Anti-XSRF token
                if ((string)ViewState[AntiXsrfTokenKey] != _antiXsrfTokenValue
                    || (string)ViewState[AntiXsrfUserNameKey] !=
(Context.User.Identity.Name ?? String.Empty))
                {
                    throw new InvalidOperationException("Validation of Anti-
XSRF token failed.");
                }
            }
        }

        protected void Page_Load(object sender, EventArgs e)
        {

        }

        protected void Page_PreRender(object sender, EventArgs e)
        {
          using (ShoppingCartActions usersShoppingCart = new
ShoppingCartActions())
          {
            string cartStr = string.Format("Cart ({0})",
usersShoppingCart.GetCount());
            cartCount.InnerText = cartStr;
          }
        }

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

        protected void Unnamed_LoggingOut(object sender, LoginCancelEventArgs
e)
        {
            Context.GetOwinContext().Authentication.SignOut();
        }
    }
}
```

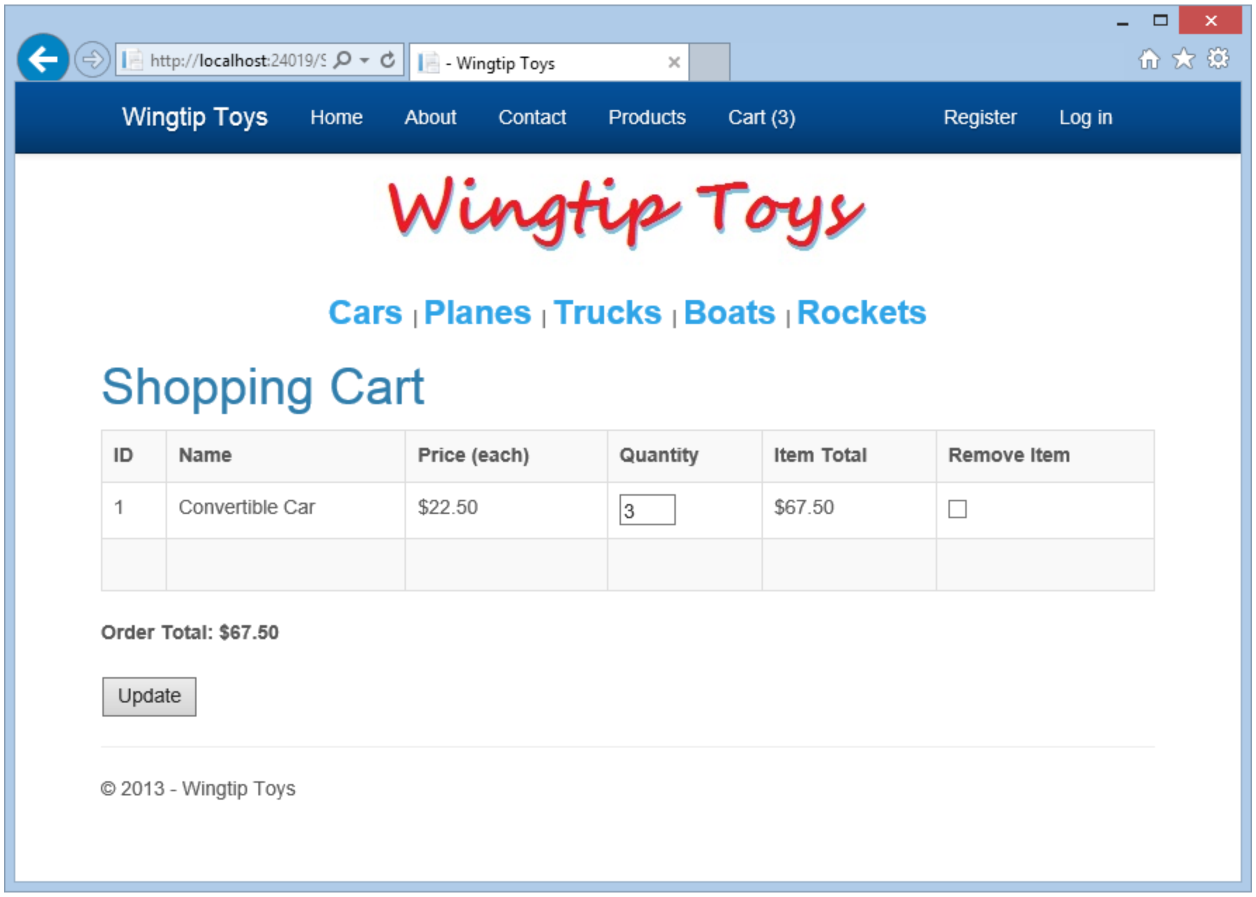
Before the page is rendered as HTML, the `Page_PreRender` event is raised. In the `Page_PreRender` handler, the total count of the shopping cart is determined by calling the `GetCount` method. The returned value is added to the `cartCount` span included in the markup of the *Site.Master* page. The `<span>` tags enables the inner elements to be properly rendered. When any page of the site is displayed, the shopping cart total will be displayed. The user can also click the shopping cart total to display the shopping cart.

## Testing the Completed Shopping Cart

You can run the application now to see how you can add, delete, and update items in the shopping cart. The shopping cart total will reflect the total cost of all items in the shopping cart.

1. Press **F5** to run the application.
   The browser opens and shows the *Default.aspx* page.
2. Select **Cars** from the category navigation menu.
3. Click the **Add To Cart** link next to the first product.
   The *ShoppingCart.aspx* page is displayed with the order total.
4. Select **Planes** from the category navigation menu.
5. Click the **Add To Cart** link next to the first product.
6. Set the quantity of the first item in the shopping cart to 3 and select the **Remove Item** check box of the second item.

7. Click the **Update** button to update the shopping cart page and display the new order total.



Shopping Cart

| ID | Name | Price (each) | Quantity | Item Total | Remove Item |
|---|---|---|---|---|---|
| 1 | Convertible Car | $22.50 | 3 | $67.50 | ☐ |
| | | | | | |

Order Total: $67.50

© 2013 - Wingtip Toys

## Summary

In this tutorial, you have created a shopping cart for the Wingtip Toys Web Forms sample application. During this tutorial you have used Entity Framework Code First, data annotations, strongly typed data controls, and model binding.

The shopping cart supports adding, deleting, and updating items that the user has selected for purchase. In addition to implementing the shopping cart functionality, you have learned how to display shopping cart items in a **GridView** control and calculate the order total.

## Addition Information

- ASP.NET Session State Overview

## Checkout and Payment with PayPal

This tutorial series will teach you the basics of building an ASP.NET Web Forms application using ASP.NET 4.5 and Microsoft Visual Studio Express 2013 for Web. A Visual Studio 2013 project with C# source code is available to accompany this tutorial series.

This tutorial describes how to modify the Wingtip Toys sample application to include user authorization, registration, and payment using PayPal. Only users who are logged in will have authorization to purchase products. The ASP.NET 4.5 Web Forms project template's built-in user registration functionality already includes much of what you need. You will add to this PayPal Express Checkout functionality. In this tutorial you be using the PayPal developer testing environment, so no actual funds will be transferred. At the end of the tutorial, you will test the application by selecting products to add to the shopping cart, clicking the checkout button, and transferring data to the PayPal testing web site. On the PayPal testing web site, you will confirm your shipping and payment information and then return to the local Wingtip Toys sample application to confirm and complete the purchase.

There are several experienced third-party payment processors that specialize in online shopping that address scalability and security. ASP.NET developers should consider the advantages of utilizing a third party payment solution before implementing a shopping and purchasing solution.

**Note**

The Wingtip Toys sample application was designed to shown specific ASP.NET concepts and features available to ASP.NET web developers. This sample application was not optimized for all possible circumstances in regard to scalability and security.

## What you'll learn:

- How to restrict access to specific pages in a folder.
- How to create a known shopping cart from an anonymous shopping cart.
- How to use PayPal to purchase products using the PayPal testing environment.
- How to display details from PayPal in a **DetailsView** control.
- How to update the database of the Wingtip Toys application with details obtained from PayPal.

## Adding Order Tracking

In this tutorial, you'll create two new classes to track data from the order a user has created. The classes will track data regarding shipping information, purchase total, and payment confirmation.

### Add the Order and OrderDetail Model Classes

Earlier in this tutorial series, you defined the schema for categories, products, and shopping cart items by creating the `Category`, `Product`, and `CartItem` classes in the *Models* folder. Now you will add two new classes to define the schema for the product order and the details of the order.

1. In the **Models** folder, add a new class named *Order.cs*.
   The new class file is displayed in the editor.
2. Replace the default code with the following:

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

namespace WingtipToys.Models
{
  public class Order
  {
    public int OrderId { get; set; }

    public System.DateTime OrderDate { get; set; }

    public string Username { get; set; }

    [Required(ErrorMessage = "First Name is required")]
    [DisplayName("First Name")]
    [StringLength(160)]
    public string FirstName { get; set; }

    [Required(ErrorMessage = "Last Name is required")]
    [DisplayName("Last Name")]
    [StringLength(160)]
    public string LastName { get; set; }

    [Required(ErrorMessage = "Address is required")]
    [StringLength(70)]
    public string Address { get; set; }

    [Required(ErrorMessage = "City is required")]
    [StringLength(40)]
    public string City { get; set; }

    [Required(ErrorMessage = "State is required")]
    [StringLength(40)]
    public string State { get; set; }

    [Required(ErrorMessage = "Postal Code is required")]
    [DisplayName("Postal Code")]
    [StringLength(10)]
    public string PostalCode { get; set; }

    [Required(ErrorMessage = "Country is required")]
    [StringLength(40)]
    public string Country { get; set; }

    [StringLength(24)]
    public string Phone { get; set; }

    [Required(ErrorMessage = "Email Address is required")]
    [DisplayName("Email Address")]
    [RegularExpression(@"[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}",
        ErrorMessage = "Email is is not valid.")]
    [DataType(DataType.EmailAddress)]
    public string Email { get; set; }

    [ScaffoldColumn(false)]
    public decimal Total { get; set; }

    [ScaffoldColumn(false)]
    public string PaymentTransactionId { get; set; }

    [ScaffoldColumn(false)]
    public bool HasBeenShipped { get; set; }

    public List<OrderDetail> OrderDetails { get; set; }
  }
}
```

3. Add an *OrderDetail.cs* class to the *Models* folder.
4. Replace the default code with the following code:

```
using System.ComponentModel.DataAnnotations;

namespace WingtipToys.Models
{
  public class OrderDetail
  {
    public int OrderDetailId { get; set; }

    public int OrderId { get; set; }

    public string Username { get; set; }

    public int ProductId { get; set; }

    public int Quantity { get; set; }

    public double? UnitPrice { get; set; }

  }
}
```

The `Order` and `OrderDetail` classes contain the schema to define the order information used for purchasing and shipping.

In addition, you will need to update the database context class that manages the entity classes and that provides data access to the database. To do this, you will add the newly created Order and `OrderDetail` model classes to `ProductContext` class.

1. In **Solution Explorer**, find and open the *ProductContext.cs* file.
2. Add the highlighted code to the *ProductContext.cs* file as shown below:

```
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; }
        public DbSet<CartItem> ShoppingCartItems { get; set; }
        public DbSet<Order> Orders { get; set; }
        public DbSet<OrderDetail> OrderDetails { get; set; }
    }
}
```

As mentioned previously in this tutorial series, the code in the *ProductContext.cs* file adds the `System.Data.Entity` namespace so that you have access to all the core functionality of the Entity Framework. This functionality includes the capability to query, insert, update, and delete data by working with strongly typed objects. The above code in the `ProductContext` class adds Entity Framework access to the newly added `Order` and `OrderDetail` classes.

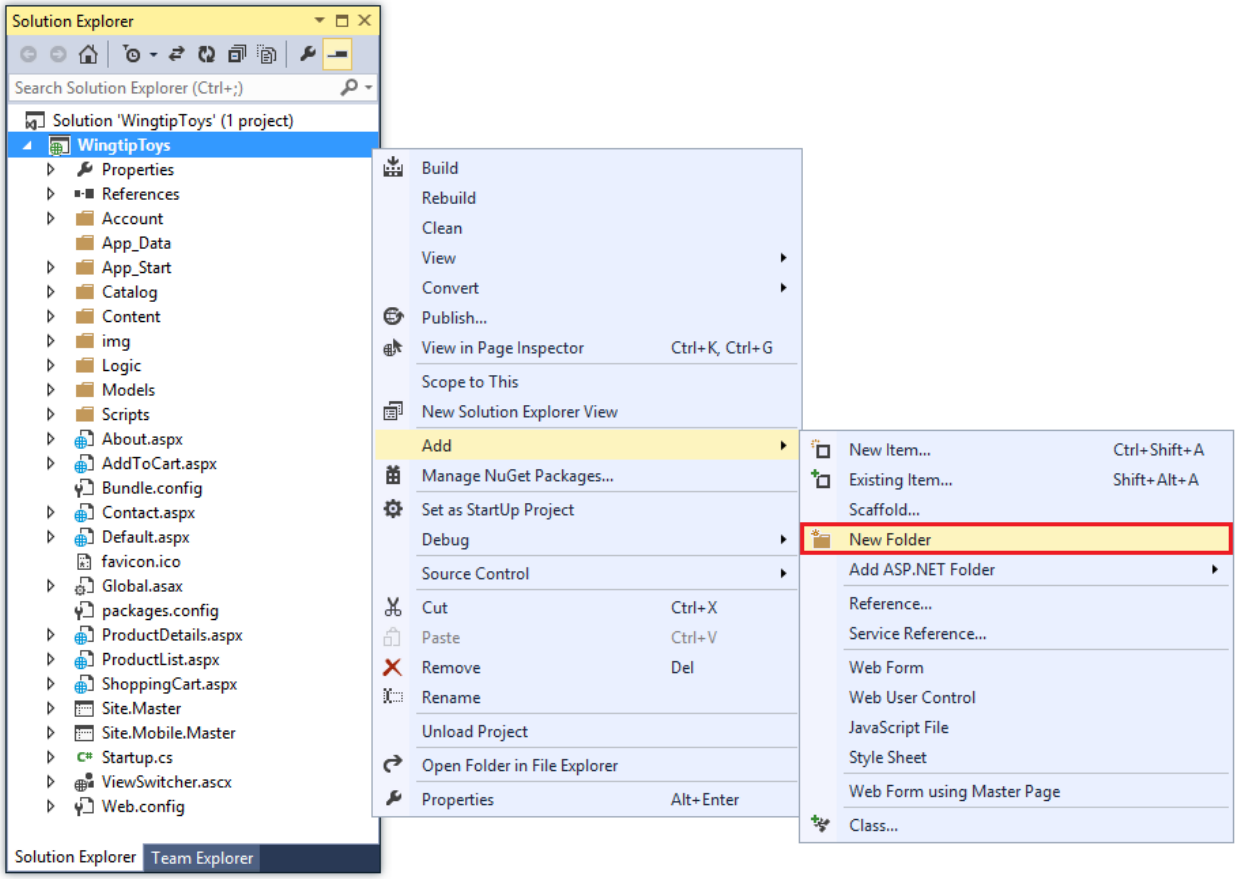
## Adding Checkout Access

The Wingtip Toys sample application allows anonymous users to review and add products to a shopping cart. However, when anonymous users choose to purchase the products they added to the shopping cart, they must log on to the site. Once they have logged on, they can access the restricted pages of the Web application that handle the checkout and purchase process. These restricted pages are contained in the *Checkout* folder of the application.

### Add a Checkout Folder and Pages

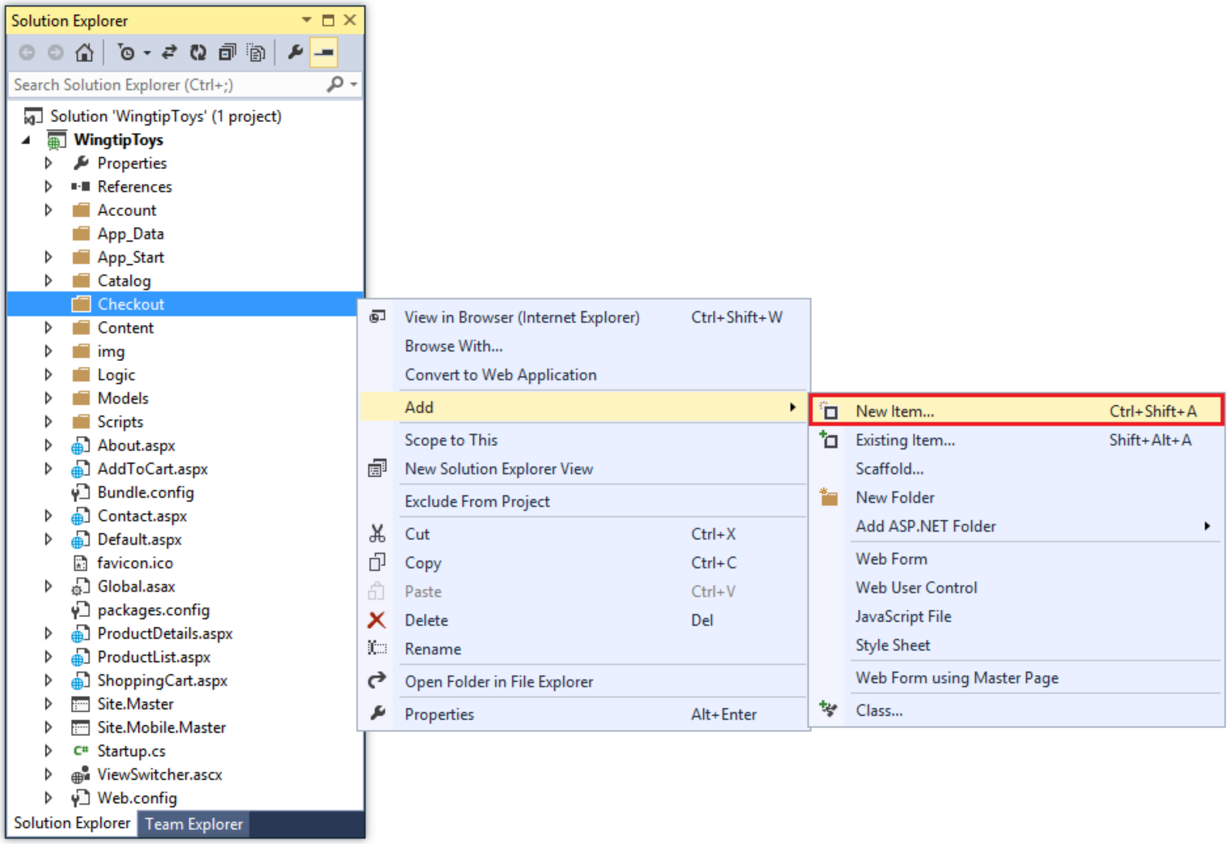
You will now create the *Checkout* folder and the pages in it that the customer will see during the checkout process. You will update these pages later in this tutorial.

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



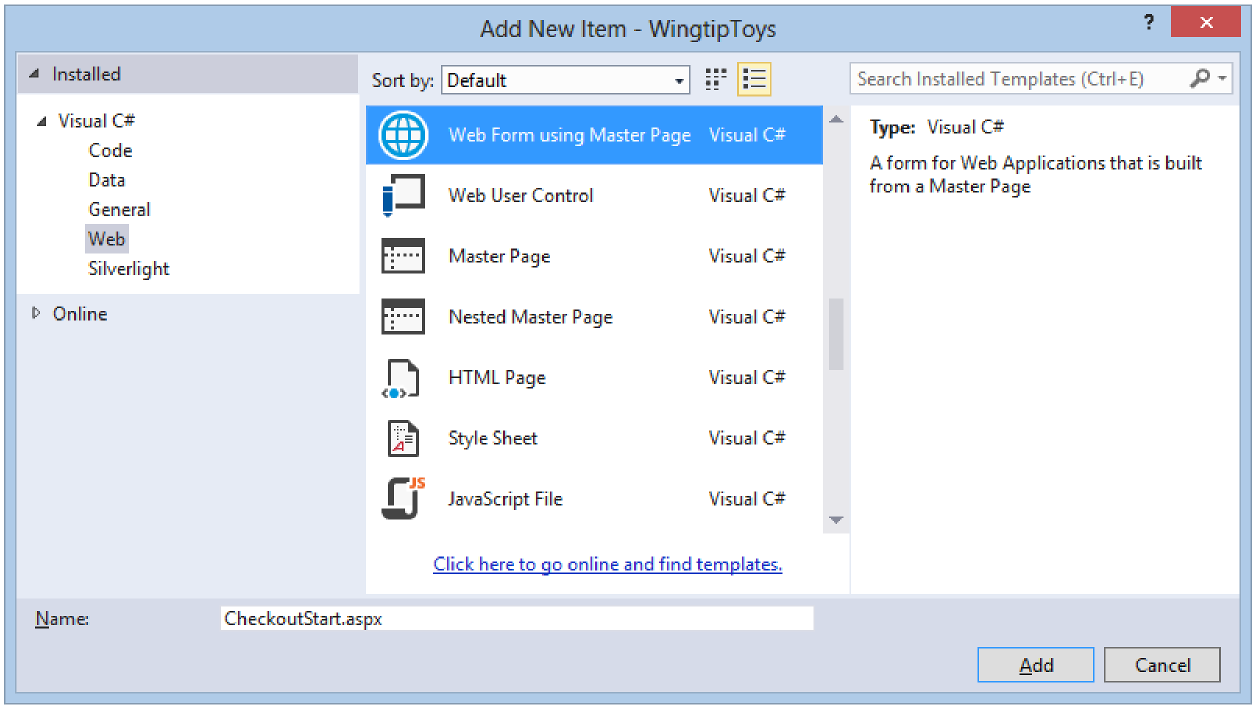
2. Name the new folder *Checkout*.

3. Right-click the *Checkout* folder and then select **Add** -> **New Item**.



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

5. Select the **Visual C#** -> **Web** templates group on the left. Then, from the middle pane, select **Web Form using Master Page** and name it *CheckoutStart.aspx*.


Add New Item - WingtipToys
Installed
Visual C#
Code
Data
General
Web
Silverlight
Online
Sort by: Default
Search Installed Templates (Ctrl+E)
Web Form using Master Page Visual C#
Web User Control Visual C#
Master Page Visual C#
Nested Master Page Visual C#
HTML Page Visual C#
Style Sheet Visual C#
JavaScript File Visual C#
Click here to go online and find templates.
Add
Cancel

**Type:** Visual C#

A form for Web Applications that is built from a Master Page

Name: CheckoutStart.aspx

6. As before, select the *Site.Master* file as the master page.
7. Add the following additional pages to the *Checkout* folder using the same steps above:
   - CheckoutReview.aspx
   - CheckoutComplete.aspx
   - CheckoutCancel.aspx
   - CheckoutError.aspx

## Add a Web.config File

By adding a new *Web.config* file to the *Checkout* folder, you will be able to restrict access to all the pages contained in the folder.

1. Right-click the *Checkout* folder and select **Add** -> **New Item**.
   The **Add New Item** dialog box is displayed.
2. Select the **Visual C#** -> **Web** templates group on the left. Then, from the middle pane, select **Web Configuration File**, accept the default name of *Web.config*, and then select **Add**.
3. Replace the existing XML content in the *Web.config* file with the following:

```
<?xml version="1.0"?>
<configuration>
  <system.web>
    <authorization>
      <deny users="?"/>
    </authorization>
  </system.web>
```

```
</configuration>
```

4. Save the *Web.config* file.

The *Web.config* file specifies that all unknown users of the Web application must be denied access to the pages contained in the *Checkout* folder. However, if the user has registered an account and is logged on, they will be a known user and will have access to the pages in the *Checkout* folder.

It's important to note that ASP.NET configuration follows a hierarchy, where each *Web.config* file applies configuration settings to the folder that it is in and to all of the child directories below it.

## Enabling Logins from Other Sites Using OAuth and OpenID

ASP.NET Web Forms provides enhanced options for membership and authentication. These enhancements include the new **OAuth** and **OpenID** providers. Using these providers, you can let users log into your site using their existing credentials from Facebook, Twitter, Windows Live, and Google. For example, to log in using a Facebook account, users can just choose a Facebook option, which redirects them to the Facebook login page where they enter their user credentials. They can then associate the Facebook login with their account on your site. A related enhancement to the ASP.NET Web Forms membership (ASP.NET Identity) features is that users can associate multiple logins (including logins from social networking sites) with a single account on your website.

When you add an **OAuth** provider (Facebook, Twitter, or Windows Live) to your ASP.NET Web Forms application, you must set the application ID (key) value and an application secret value. You add these values to the *Startup.Auth.cs* file in your Web Forms application. Additionally, you must create an application on the external site (Facebook, Twitter, or Windows Live). When you create the application on the external site you can get the application keys that you'll need in order to invoke the login feature for those sites.

**Note**
Windows Live applications only accept a live URL for a working website, so you cannot use a local website URL for testing logins.

For sites that use an **OpenID** provider (Google), you do not have to create an application on the external site.

1. In **Solution Explorer**, find and open the *App_Start* folder.
2. Open the file named *Startup.Auth.cs*.
3. Uncomment the single line of code highlighted in yellow to allow Google **OpenID** accounts as follows:

```
using Microsoft.AspNet.Identity;
using Microsoft.Owin;
using Microsoft.Owin.Security.Cookies;
using Owin;

namespace WingtipToys
{
    public partial class Startup {

        // For more information on configuring authentication, please visit
http://go.microsoft.com/fwlink/?LinkId=301883
        public void ConfigureAuth(IAppBuilder app)
        {
            // Enable the application to use a cookie to store information for
the signed in user
            // and also store information about a user logging in with a third
party login provider.
            // This is required if your application allows users to login
            app.UseCookieAuthentication(new CookieAuthenticationOptions
            {
                AuthenticationType =
DefaultAuthenticationTypes.ApplicationCookie,
                LoginPath = new PathString("/Account/Login")
            });

app.UseExternalSignInCookie(DefaultAuthenticationTypes.ExternalCookie);

            // Uncomment the following lines to enable logging in with third
party login providers
            //app.UseMicrosoftAccountAuthentication(
            //    clientId: "",
            //    clientSecret: "");

            //app.UseTwitterAuthentication(
            //   consumerKey: "",
            //   consumerSecret: "");

            //app.UseFacebookAuthentication(
            //   appId: "",
            //   appSecret: "");

            app.UseGoogleAuthentication();
        }
    }
}
```

4. Save the *Startup.Auth.cs* file.

When you run the Wingtip Toys sample application, you will have the option to login to your Google account and associate your Wingtip Toys account with the Google account.

## Modifying Login Functionality

As previously mentioned in this tutorial series, much of the user registration functionality has been included in the ASP.NET Web Forms template by default. Now you will modify the default *Login.aspx* and *Register.aspx* pages to call the `MigrateCart` method. The `MigrateCart` method associates a newly logged in user with an anonymous shopping cart. By associating the user and shopping cart, the Wingtip Toys sample application will be able to maintain the shopping cart of the user between visits.

1. In **Solution Explorer**, find and open the *Account* folder.

2. Modify the code-behind page named *Login.aspx.cs* to include the code highlighted in yellow, so that it appears as follows:

```
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.AspNet.Identity.Owin;
using Microsoft.Owin.Security;
using System;
using System.Linq;
using System.Web;
using System.Web.UI;
using WingtipToys.Models;

namespace WingtipToys.Account
{
    public partial class Login : Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            RegisterHyperLink.NavigateUrl = "Register";
            OpenAuthLogin.ReturnUrl = Request.QueryString["ReturnUrl"];
            var returnUrl = 
HttpUtility.UrlEncode(Request.QueryString["ReturnUrl"]);
            if (!String.IsNullOrEmpty(returnUrl))
            {
                RegisterHyperLink.NavigateUrl += "?ReturnUrl=" + returnUrl;
            }
        }

        protected void LogIn(object sender, EventArgs e)
        {
            if (IsValid)
            {
                // Validate the user password
                var manager = new UserManager();
                ApplicationUser user = manager.Find(UserName.Text, 
Password.Text);
                if (user != null)
                {
                    IdentityHelper.SignIn(manager, user, RememberMe.Checked);

                    WingtipToys.Logic.ShoppingCartActions usersShoppingCart = 
new WingtipToys.Logic.ShoppingCartActions();
                    String cartId = usersShoppingCart.GetCartId();
                    usersShoppingCart.MigrateCart(cartId, UserName.Text);


IdentityHelper.RedirectToReturnUrl(Request.QueryString["ReturnUrl"], Response);
                }
                else
                {
                    FailureText.Text = "Invalid username or password.";
                    ErrorMessage.Visible = true;
                }
            }
        }
    }
}
```

3. Save the *Login.aspx.cs* file.

For now, you can ignore the warning that there is no definition for the `MigrateCart` method. You will be adding it a bit later in this tutorial.

The *Login.aspx.cs* code-behind file supports a LogIn method. By inspecting the Login.aspx page, you'll see that this page includes a "Log in" button that when click triggers the `LogIn` handler on the code-behind.

When the `Login` method on the *Login.aspx.cs* is called, a new instance of the shopping cart named `usersShoppingCart` is created. The ID of the shopping cart (a GUID) is retrieved and set to the `cartId` variable. Then, the `MigrateCart` method is called, passing both the `cartId` and the name of the logged-in user to this method. When the shopping cart is migrated, the GUID used to identify the anonymous shopping cart is replaced with the user name.

In addition to modifying the *Login.aspx.cs* code-behind file to migrate the shopping cart when the user logs in, you must also modify the *Register.aspx.cs code-behind file* to migrate the shopping cart when the user creates a new account and logs in.

1. In the *Account* folder, open the code-behind file named *Register.aspx.cs*.
2. Modify the code-behind file by including the code in yellow, so that it appears as follows:

```
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.AspNet.Identity.Owin;
using System;
using System.Linq;
using System.Web;
using System.Web.UI;
using WingtipToys.Models;

namespace WingtipToys.Account
{
    public partial class Register : Page
    {
        protected void CreateUser_Click(object sender, EventArgs e)
        {
            var manager = new UserManager();
            var user = new ApplicationUser() { UserName = UserName.Text };
            IdentityResult result = manager.Create(user, Password.Text);
            if (result.Succeeded)
            {
                IdentityHelper.SignIn(manager, user, isPersistent: false);

                using (WingtipToys.Logic.ShoppingCartActions usersShoppingCart
= new WingtipToys.Logic.ShoppingCartActions())
                {
                  String cartId = usersShoppingCart.GetCartId();
                  usersShoppingCart.MigrateCart(cartId, user.Id);
                }


IdentityHelper.RedirectToReturnUrl(Request.QueryString["ReturnUrl"], Response);
            }
            else
            {
                ErrorMessage.Text = result.Errors.FirstOrDefault();
            }
        }
    }
}
```

3. Save the *Register.aspx.cs* file. Once again, ignore the warning about the `MigrateCart` method.

Notice that the code you used in the `CreateUser_Click` event handler is very similar to the code you used in the `LogIn` method. When the user registers or logs in to the site, a call to the `MigrateCart` method will be made.

## Migrating the Shopping Cart

Now that you have the log-in and registration process updated, you can add the code to migrate the shopping cart—the `MigrateCart` method.

1. In **Solution Explorer**, find the *Logic* folder and open the *ShoppingCartActions.cs* class file.
2. Add the code highlighted in yellow to the existing code in the *ShoppingCartActions.cs* file, so that the code in the *ShoppingCartActions.cs* file appears as follows:

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

namespace WingtipToys.Logic
{
  public class ShoppingCartActions : IDisposable
  {
    public string ShoppingCartId { get; set; }

    private ProductContext _db = new ProductContext();

    public const string CartSessionKey = "CartId";

    public void AddToCart(int id)
    {
      // Retrieve the product from the database.
      ShoppingCartId = GetCartId();

      var cartItem = _db.ShoppingCartItems.SingleOrDefault(
          c => c.CartId == ShoppingCartId
          && c.ProductId == id);
      if (cartItem == null)
      {
        // Create a new cart item if no cart item exists.
        cartItem = new CartItem
        {
          ItemId = Guid.NewGuid().ToString(),
          ProductId = id,
          CartId = ShoppingCartId,
          Product = _db.Products.SingleOrDefault(
           p => p.ProductID == id),
          Quantity = 1,
          DateCreated = DateTime.Now
        };
        _db.ShoppingCartItems.Add(cartItem);
      }
      else
      {
        // If the item does exist in the cart,
        // then add one to the quantity.
        cartItem.Quantity++;
      }
      _db.SaveChanges();
    }

    public void Dispose()
    {
      if (_db != null)
      {
        _db.Dispose();
        _db = null;
      }
    }

    public string GetCartId()
    {
      if (HttpContext.Current.Session[CartSessionKey] == null)
      {
        if (!string.IsNullOrWhiteSpace(HttpContext.Current.User.Identity.Name))
        {
          HttpContext.Current.Session[CartSessionKey] =
HttpContext.Current.User.Identity.Name;
        }
        else
        {
          // Generate a new random GUID using System.Guid class.
          Guid tempCartId = Guid.NewGuid();
          HttpContext.Current.Session[CartSessionKey] = tempCartId.ToString();
        }
      }
      return HttpContext.Current.Session[CartSessionKey].ToString();
    }

    public List<CartItem> GetCartItems()
    {
      ShoppingCartId = GetCartId();

      return _db.ShoppingCartItems.Where(
          c => c.CartId == ShoppingCartId).ToList();
    }

    public decimal GetTotal()
    {
      ShoppingCartId = GetCartId();
      // Multiply product price by quantity of that product to get
      // the current price for each of those products in the cart.
      // Sum all product price totals to get the cart total.
      decimal? total = decimal.Zero;
      total = (decimal?)(from cartItems in _db.ShoppingCartItems
                         where cartItems.CartId == ShoppingCartId
                         select (int?)cartItems.Quantity *
                         cartItems.Product.UnitPrice).Sum();
      return total ?? decimal.Zero;
    }

    public ShoppingCartActions GetCart(HttpContext context)
    {
      using (var cart = new ShoppingCartActions())
      {
        cart.ShoppingCartId = cart.GetCartId();
        return cart;
      }
    }

    public void UpdateShoppingCartDatabase(String cartId, ShoppingCartUpdates[]
CartItemUpdates)
    {
      using (var db = new WingtipToys.Models.ProductContext())
      {
        try
        {
          int CartItemCount = CartItemUpdates.Count();
          List<CartItem> myCart = GetCartItems();
          foreach (var cartItem in myCart)
          {
            // Iterate through all rows within shopping cart list
            for (int i = 0; i < CartItemCount; i++)
            {
              if (cartItem.Product.ProductID == CartItemUpdates[i].ProductId)
              {
                if (CartItemUpdates[i].PurchaseQuantity < 1 ||
CartItemUpdates[i].RemoveItem == true)
                {
                  RemoveItem(cartId, cartItem.ProductId);
                }
                else
                {
                  UpdateItem(cartId, cartItem.ProductId,
CartItemUpdates[i].PurchaseQuantity);
                }
              }
            }
          }
        }
        catch (Exception exp)
        {
          throw new Exception("ERROR: Unable to Update Cart Database - " +
exp.Message.ToString(), exp);
        }
      }
    }

    public void RemoveItem(string removeCartID, int removeProductID)
    {
      using (var _db = new WingtipToys.Models.ProductContext())
      {
        try
        {
          var myItem = (from c in _db.ShoppingCartItems where c.CartId ==
removeCartID && c.Product.ProductID == removeProductID select
c).FirstOrDefault();
          if (myItem != null)
          {
            // Remove Item.
            _db.ShoppingCartItems.Remove(myItem);
            _db.SaveChanges();
          }
        }
        catch (Exception exp)
        {
          throw new Exception("ERROR: Unable to Remove Cart Item - " +
exp.Message.ToString(), exp);
        }
      }
    }

    public void UpdateItem(string updateCartID, int updateProductID, int
quantity)
    {
      using (var _db = new WingtipToys.Models.ProductContext())
      {
        try
        {
          var myItem = (from c in _db.ShoppingCartItems where c.CartId ==
updateCartID && c.Product.ProductID == updateProductID select
c).FirstOrDefault();
          if (myItem != null)
          {
            myItem.Quantity = quantity;
            _db.SaveChanges();
          }
        }
        catch (Exception exp)
        {
          throw new Exception("ERROR: Unable to Update Cart Item - " +
exp.Message.ToString(), exp);
        }
      }
    }

    public void EmptyCart()
    {
      ShoppingCartId = GetCartId();
      var cartItems = _db.ShoppingCartItems.Where(
          c => c.CartId == ShoppingCartId);
      foreach (var cartItem in cartItems)
      {
        _db.ShoppingCartItems.Remove(cartItem);
      }
      // Save changes.
      _db.SaveChanges();
    }

    public int GetCount()
    {
      ShoppingCartId = GetCartId();

      // Get the count of each item in the cart and sum them up
      int? count = (from cartItems in _db.ShoppingCartItems
                    where cartItems.CartId == ShoppingCartId
                    select (int?)cartItems.Quantity).Sum();
      // Return 0 if all entries are null
      return count ?? 0;
    }

    public struct ShoppingCartUpdates
    {
      public int ProductId;
      public int PurchaseQuantity;
      public bool RemoveItem;
    }
    public void MigrateCart(string cartId, string userName)
    {
      var shoppingCart = _db.ShoppingCartItems.Where(c => c.CartId == cartId);
      foreach (CartItem item in shoppingCart)
      {
        item.CartId = userName;
      }
      HttpContext.Current.Session[CartSessionKey] = userName;
       db.SaveChanges();
    }
  }
}
```

The `MigrateCart` method uses the existing cartId to find the shopping cart of the user. Next, the code loops through all the shopping cart items and replaces the `CartId` property (as specified by the `CartItem` schema) with the logged-in user name.

### Updating the Database Connection

If you are following this tutorial using the **prebuilt** Wingtip Toys sample application, you must recreate the default membership database. By modifying the default connection string, the membership database will be created the next time the application runs.

1. Open the *Web.config* file at the root of the project.
2. Update the default connection string so that it appears as follows:

```
    <add name="DefaultConnection" connectionString="Data
Source=(LocalDb)\v11.0;Initial Catalog=aspnet-WingtipToys;Integrated
Security=True" providerName="System.Data.SqlClient" />
```

## Integrating PayPal

PayPal is a web-based billing platform that accepts payments by online merchants. This tutorial next explains how to integrate PayPal's Express Checkout functionality into your application. Express Checkout allows your customers to use PayPal to pay for the items they have added to their shopping cart.

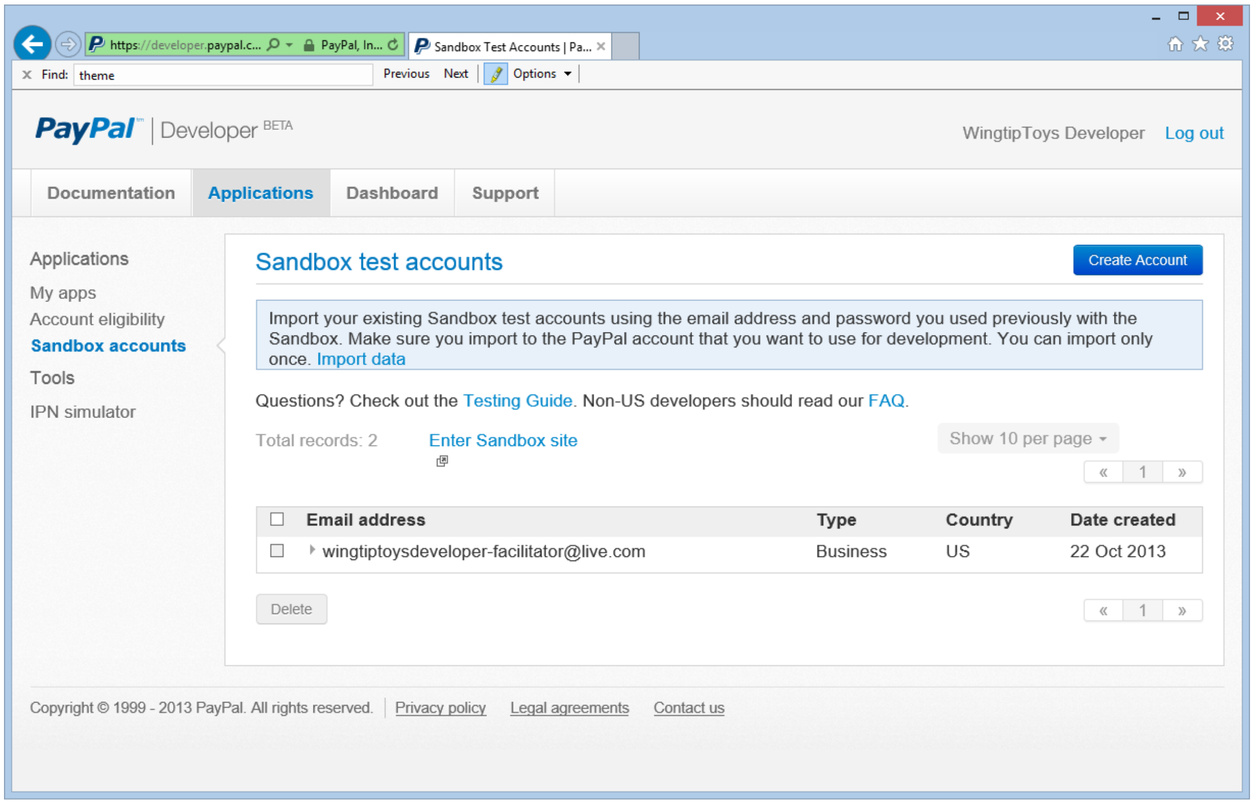
### Create PaylPal Test Accounts

To use the PayPal testing environment, you must create and verify a developer test account. You will use the developer test account to create a buyer test account and a seller test account. The developer test account credentials also will allow the Wingtip Toys sample application to access the PayPal testing environment.

1. In a browser, navigate to the PayPal developer testing site: `https://developer.paypal.com`
2. If you don't have a PayPal developer account, create a new account by clicking **Sign Up** and following the sign up steps. If you have an existing PayPal developer account, sign in by clicking **Log In**. You will need your PayPal developer account to test the Wingtip Toys sample application later in this tutorial.

3. If you have just signed up for your PayPal developer account, you may need to verify your PayPal developer account with PayPal. You can verify your account by following the steps that PayPal sent to your email account. Once you have verified your PayPal developer account, log back into the PayPal developer testing site.
4. After you are logged in to the PayPal developer site with your PayPal developer account you need to create a PayPal buyer test account if you don't already have one. To create a buyer test account, on the PayPal site click the **Applications** tab and then click **Sandbox accounts**.
   The **Sandbox test accounts** page is shown.

   **Note**
   The PayPal Developer site already provides a merchant test account.



Sandbox test accounts

Import your existing Sandbox test accounts using the email address and password you used previously with the Sandbox. Make sure you import to the PayPal account that you want to use for development. You can import only once. Import data

Questions? Check out the Testing Guide. Non-US developers should read our FAQ.

Total records: 2 Enter Sandbox site Show 10 per page

| | Email address | Type | Country | Date created |
|---|---|---|---|---|
| ☐ | wingtiptoysdeveloper-facilitator@live.com | Business | US | 22 Oct 2013 |

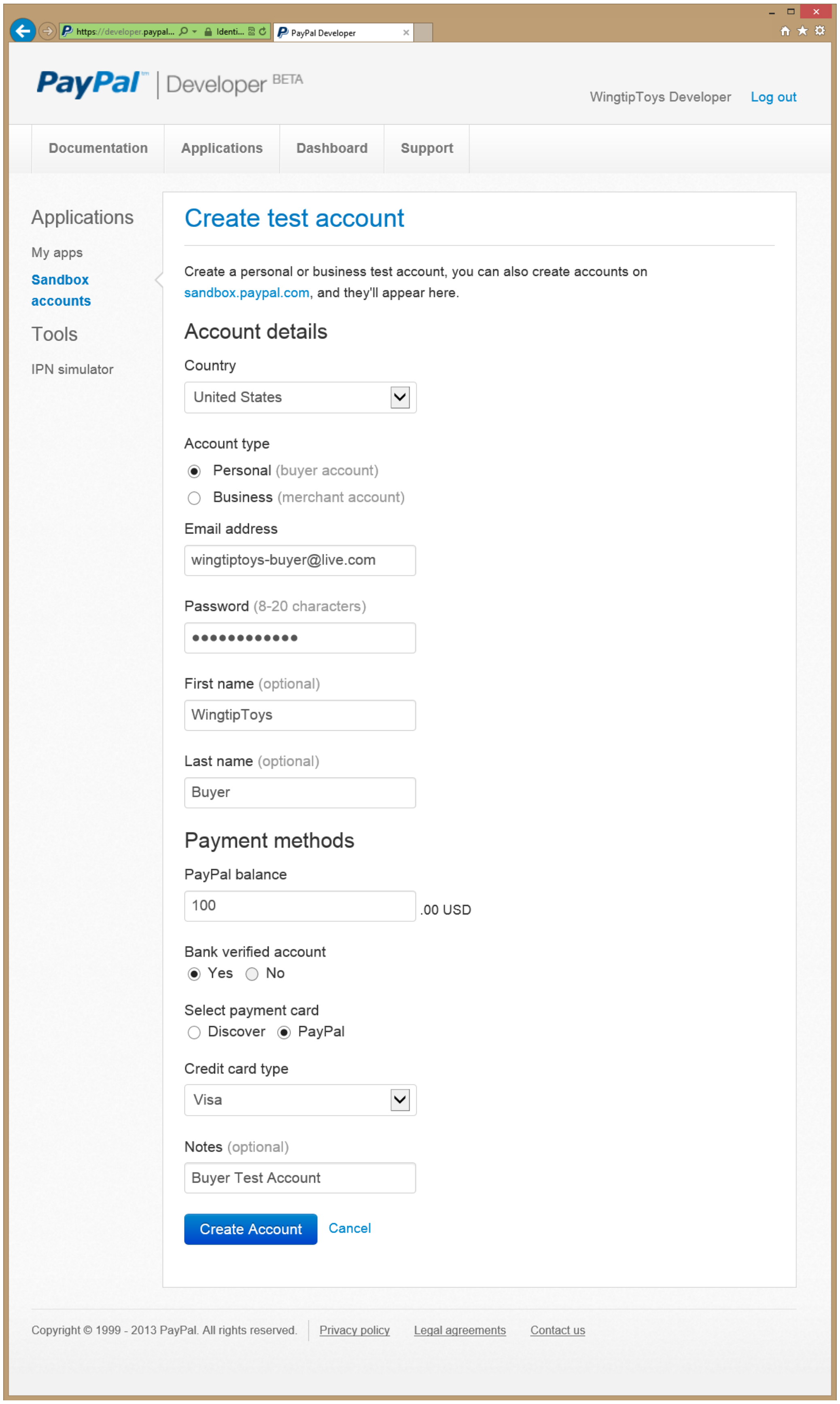
Delete

Copyright © 1999 - 2013 PayPal. All rights reserved. | Privacy policy Legal agreements Contact us

5. On the Sandbox test accounts page, click **Create Account**.
6. On the **Create test account** page choose a buyer test account email and password of your choice.

   **Note**
   You will need the buyer email addresses and password to test the Wingtip Toys sample

application at the end of this tutorial.



## Create test account

Create a personal or business test account, you can also create accounts on sandbox.paypal.com, and they'll appear here.

### Account details

Country: United States

Account type: [● Personal (buyer account)] ○ Business (merchant account)

Email address: wingtiptoys-buyer@live.com

Password (8-20 characters): ●●●●●●●●●●●●

First name (optional): WingtipToys

Last name (optional): Buyer

### Payment methods

PayPal balance: 100 .00 USD

Bank verified account: [● Yes] ○ No

Select payment card: ○ Discover [● PayPal]

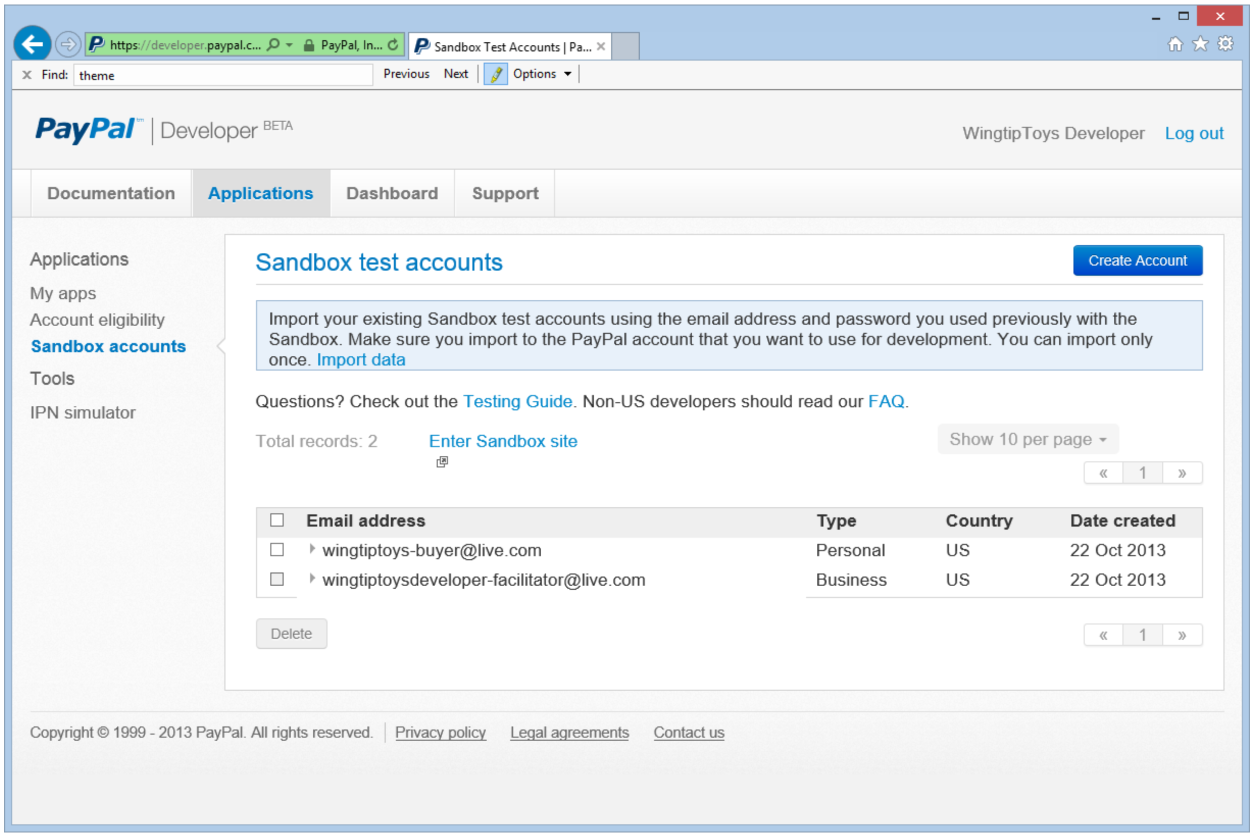
Credit card type: Visa

Notes (optional): Buyer Test Account

Create Account   Cancel

Copyright © 1999 - 2013 PayPal. All rights reserved. | Privacy policy  Legal agreements  Contact us

7. Create the buyer test account by clicking the **Create Account** button.
   The **Sandbox Test accounts** page is displayed.



Sandbox test accounts

Import your existing Sandbox test accounts using the email address and password you used previously with the Sandbox. Make sure you import to the PayPal account that you want to use for development. You can import only once. Import data

Questions? Check out the Testing Guide. Non-US developers should read our FAQ.

Total records: 2 Enter Sandbox site Show 10 per page

| Email address | Type | Country | Date created |
|---|---|---|---|
| wingtiptoys-buyer@live.com | Personal | US | 22 Oct 2013 |
| wingtiptoysdeveloper-facilitator@live.com | Business | US | 22 Oct 2013 |

Delete

Copyright © 1999 - 2013 PayPal. All rights reserved. | Privacy policy Legal agreements Contact us

8. On the **Sandbox test accounts** page, click the **facilitator** email account.
   **Profile** and **Notification** options appear.
9. Select the **Profile** option, then click **API credentials** to view your API credentials for the merchant test account.
10. Copy the TEST API credentials to notepad.

You will need your displayed Classic TEST API credentials (Username, Password, and Signature) to make API calls from the Wingtip Toys sample application to the PayPal testing environment. You will add the credentials in the next step.

## Add PayPal Class and API Credentials

You will place the majority of the PayPal code into a single class. This class contains the methods used to communicate with PayPal. Also, you will add your PayPal credentials to this class.

1. In the Wingtip Toys sample application within Visual Studio, right-click the **Logic** folder and then select **Add** -> **New Item**.
   The **Add New Item** dialog box is displayed.
2. Under **Visual C#** from the **Installed** pane on the left, select **Code**.
3. From the middle pane, select **Class**. Name this new class **PayPalFunctions.cs**.

4. Click **Add**.
   The new class file is displayed in the editor.
5. Replace the default code with the following code:

```
using System;
using System.Collections;
using System.Collections.Specialized;
using System.IO;
using System.Net;
using System.Text;
using System.Data;
using System.Configuration;
using System.Web;
using WingtipToys;
using WingtipToys.Models;
using System.Collections.Generic;
using System.Linq;

public class NVPAPICaller
{
  //Flag that determines the PayPal environment (live or sandbox)
  private const bool bSandbox = true;
  private const string CVV2 = "CVV2";

  // Live strings.
  private string pEndPointURL = "https://api-3t.paypal.com/nvp";
  private string host = "www.paypal.com";

  // Sandbox strings.
  private string pEndPointURL_SB = "https://api-3t.sandbox.paypal.com/nvp";
  private string host_SB = "www.sandbox.paypal.com";

  private const string SIGNATURE = "SIGNATURE";
  private const string PWD = "PWD";
  private const string ACCT = "ACCT";

  //Replace <Your API Username> with your API Username
  //Replace <Your API Password> with your API Password
  //Replace <Your Signature> with your Signature
  public string APIUsername = "<Your API Username>";
  private string APIPassword = "<Your API Password>";
  private string APISignature = "<Your Signature>";
  private string Subject = "";
  private string BNCode = "PP-ECWizard";


  //HttpWebRequest Timeout specified in milliseconds
  private const int Timeout = 15000;
  private static readonly string[] SECURED_NVPS = new string[] { ACCT, CVV2,
SIGNATURE, PWD };

  public void SetCredentials(string Userid, string Pwd, string Signature)
  {
    APIUsername = Userid;
    APIPassword = Pwd;
    APISignature = Signature;
  }

  public bool ShortcutExpressCheckout(string amt, ref string token, ref string
retMsg)
  {
    if (bSandbox)
    {
      pEndPointURL = pEndPointURL_SB;
      host = host_SB;
    }

    string returnURL = "http://localhost:1234/Checkout/CheckoutReview.aspx";
    string cancelURL = "http://localhost:1234/Checkout/CheckoutCancel.aspx";

    NVPCodec encoder = new NVPCodec();
    encoder["METHOD"] = "SetExpressCheckout";
    encoder["RETURNURL"] = returnURL;
    encoder["CANCELURL"] = cancelURL;
    encoder["BRANDNAME"] = "Wingtip Toys Sample Application";
    encoder["PAYMENTREQUEST_0_AMT"] = amt;
    encoder["PAYMENTREQUEST_0_ITEMAMT"] = amt;
    encoder["PAYMENTREQUEST_0_PAYMENTACTION"] = "Sale";
    encoder["PAYMENTREQUEST_0_CURRENCYCODE"] = "USD";

    // Get the Shopping Cart Products
    using (WingtipToys.Logic.ShoppingCartActions myCartOrders = new
WingtipToys.Logic.ShoppingCartActions())
    {
      List<CartItem> myOrderList = myCartOrders.GetCartItems();

      for (int i = 0; i < myOrderList.Count; i++)
      {
        encoder["L_PAYMENTREQUEST_0_NAME" + i] =
myOrderList[i].Product.ProductName.ToString();
        encoder["L_PAYMENTREQUEST_0_AMT" + i] =
myOrderList[i].Product.UnitPrice.ToString();
        encoder["L_PAYMENTREQUEST_0_QTY" + i] =
myOrderList[i].Quantity.ToString();
      }
    }

    string pStrrequestforNvp = encoder.Encode();
    string pStresponsenvp = HttpCall(pStrrequestforNvp);

    NVPCodec decoder = new NVPCodec();
    decoder.Decode(pStresponsenvp);

    string strAck = decoder["ACK"].ToLower();
    if (strAck != null && (strAck == "success" || strAck ==
"successwithwarning"))
    {
      token = decoder["TOKEN"];
      string ECURL = "https://" + host + "/cgi-bin/webscr?cmd=_express-
checkout" + "&token=" + token;
      retMsg = ECURL;
      return true;
    }
    else
    {
      retMsg = "ErrorCode=" + decoder["L_ERRORCODE0"] + "&" +
          "Desc=" + decoder["L_SHORTMESSAGE0"] + "&" +
          "Desc2=" + decoder["L_LONGMESSAGE0"];
      return false;
    }
  }

  public bool GetCheckoutDetails(string token, ref string PayerID, ref NVPCodec
decoder, ref string retMsg)
```

```
  {
    if (bSandbox)
    {
      pEndPointURL = pEndPointURL SB;
    }

    NVPCodec encoder = new NVPCodec();
    encoder["METHOD"] = "GetExpressCheckoutDetails";
    encoder["TOKEN"] = token;

    string pStrrequestforNvp = encoder.Encode();
    string pStresponsenvp = HttpCall(pStrrequestforNvp);

    decoder = new NVPCodec();
    decoder.Decode(pStresponsenvp);

    string strAck = decoder["ACK"].ToLower();
    if (strAck != null && (strAck == "success" || strAck ==
"successwithwarning"))
    {
      PayerID = decoder["PAYERID"];
      return true;
    }
    else
    {
      retMsg = "ErrorCode=" + decoder["L_ERRORCODE0"] + "&" +
          "Desc=" + decoder["L_SHORTMESSAGE0"] + "&" +
          "Desc2=" + decoder["L_LONGMESSAGE0"];

      return false;
    }
  }

  public bool DoCheckoutPayment(string finalPaymentAmount, string token, string
PayerID, ref NVPCodec decoder, ref string retMsg)
  {
    if (bSandbox)
    {
      pEndPointURL = pEndPointURL_SB;
    }

    NVPCodec encoder = new NVPCodec();
    encoder["METHOD"] = "DoExpressCheckoutPayment";
    encoder["TOKEN"] = token;
    encoder["PAYERID"] = PayerID;
    encoder["PAYMENTREQUEST_0_AMT"] = finalPaymentAmount;
    encoder["PAYMENTREQUEST_0_CURRENCYCODE"] = "USD";
    encoder["PAYMENTREQUEST_0_PAYMENTACTION"] = "Sale";

    string pStrrequestforNvp = encoder.Encode();
    string pStresponsenvp = HttpCall(pStrrequestforNvp);

    decoder = new NVPCodec();
    decoder.Decode(pStresponsenvp);

    string strAck = decoder["ACK"].ToLower();
    if (strAck != null && (strAck == "success" || strAck ==
"successwithwarning"))
    {
      return true;
    }
    else
    {
      retMsg = "ErrorCode=" + decoder["L_ERRORCODE0"] + "&" +
          "Desc=" + decoder["L_SHORTMESSAGE0"] + "&" +
          "Desc2=" + decoder["L_LONGMESSAGE0"];

      return false;
    }
  }

  public string HttpCall(string NvpRequest)
  {
    string url = pEndPointURL;

    string strPost = NvpRequest + "&" + buildCredentialsNVPString();
    strPost = strPost + "&BUTTONSOURCE=" + HttpUtility.UrlEncode(BNCode);

    HttpWebRequest objRequest = (HttpWebRequest)WebRequest.Create(url);
    objRequest.Timeout = Timeout;
    objRequest.Method = "POST";
    objRequest.ContentLength = strPost.Length;

    try
    {
      using (StreamWriter myWriter = new
StreamWriter(objRequest.GetRequestStream()))
      {
        myWriter.Write(strPost);
      }
    }
    catch (Exception)
    {
      // No logging for this tutorial.
    }

    //Retrieve the Response returned from the NVP API call to PayPal.
    HttpWebResponse objResponse = (HttpWebResponse)objRequest.GetResponse();
    string result;
    using (StreamReader sr = new StreamReader(objResponse.GetResponseStream()))
    {
      result = sr.ReadToEnd();
    }

    return result;
  }

  private string buildCredentialsNVPString()
  {
    NVPCodec codec = new NVPCodec();

    if (!IsEmpty(APIUsername))
      codec["USER"] = APIUsername;

    if (!IsEmpty(APIPassword))
      codec[PWD] = APIPassword;

    if (!IsEmpty(APISignature))
      codec[SIGNATURE] = APISignature;

    if (!IsEmpty(Subject))
      codec["SUBJECT"] = Subject;

    codec["VERSION"] = "88.0";

    return codec.Encode();
  }

  public static bool IsEmpty(string s)
  {
    return s == null || s.Trim() == string.Empty;
  }
}

public sealed class NVPCodec : NameValueCollection
{
  private const string AMPERSAND = "&";
  private const string EQUALS = "=";
  private static readonly char[] AMPERSAND_CHAR_ARRAY =
AMPERSAND.ToCharArray();
  private static readonly char[] EQUALS_CHAR_ARRAY = EQUALS.ToCharArray();

  public string Encode()
  {
    StringBuilder sb = new StringBuilder();
    bool firstPair = true;
    foreach (string kv in AllKeys)
    {
      string name = HttpUtility.UrlEncode(kv);
      string value = HttpUtility.UrlEncode(this[kv]);
      if (!firstPair)
      {
        sb.Append(AMPERSAND);
      }
      sb.Append(name).Append(EQUALS).Append(value);
      firstPair = false;
    }
    return sb.ToString();
  }

  public void Decode(string nvpstring)
  {
    Clear();
    foreach (string nvp in nvpstring.Split(AMPERSAND_CHAR_ARRAY))
    {
      string[] tokens = nvp.Split(EQUALS_CHAR_ARRAY);
      if (tokens.Length >= 2)
      {
        string name = HttpUtility.UrlDecode(tokens[0]);
        string value = HttpUtility.UrlDecode(tokens[1]);
        Add(name, value);
      }
    }
  }

  public void Add(string name, string value, int index)
  {
    this.Add(GetArrayName(index, name), value);
  }

  public void Remove(string arrayName, int index)
  {
    this.Remove(GetArrayName(index, arrayName));
  }

  public string this[string name, int index]
  {
    get
    {
      return this[GetArrayName(index, name)];
    }
    set
    {
      this[GetArrayName(index, name)] = value;
    }
  }

  private static string GetArrayName(int index, string name)
  {
    if (index < 0)
    {
      throw new ArgumentOutOfRangeException("index", "index cannot be negative
: " + index);
    }
    return name + index;
  }
}
```

6. Add the Merchant API credentials (Username, Password, and Signature) that you displayed earlier in this tutorial so that you can make function calls to the PayPal testing environment.

```
    public string APIUsername = "<Your API Username>";
    private string APIPassword = "<Your API Password>";
    private string APISignature = "<Your Signature>";
```

**Note**

In this sample application you are simply adding credentials to a C# file (.cs). However, in a implemented solution, you should consider encrypting your credentials in a configuration file.

The NVPAPICaller class contains the majority of the PayPal functionality. The code in the class provides the methods needed to make a test purchase from the PayPal testing environment. The following three PayPal functions are used to make purchases:

1. `SetExpressCheckout` function
2. `GetExpressCheckoutDetails` function
3. `DoExpressCheckoutPayment` function

The `ShortcutExpressCheckout` method collects the test purchase information and product details from the shopping cart and calls the `SetExpressCheckout` PayPal function. The `GetCheckoutDetails` method confirms purchase details and calls the `GetExpressCheckoutDetails` PayPal function before making the test purchase. The `DoCheckoutPayment` method completes the test purchase from the testing environment by calling the `DoExpressCheckoutPayment` PayPal function. The remaining code supports the PayPal methods and process, such as encoding strings, decoding strings, processing arrays, and determining credentials.

**Note**

PayPal allows you to include optional purchase details based on PayPal's API specification. By extending the code in the Wingtip Toys sample application, you can include localization details, product descriptions, tax, a customer service number, as well as many other optional fields.

Notice that the return and cancel URLs that are specified in the **ShortcutExpressCheckout** method use a port number.

```
        string returnURL =
"http://localhost:1234/Checkout/CheckoutReview.aspx";
        string cancelURL =
"http://localhost:1234/Checkout/CheckoutCancel.aspx";
```

When Visual Web Developer runs a web project, a random port is used for the web server. As shown above, the port number is 1234. When you run the application, you'll probably see a different port number. Your port number needs to be set in the above code so that you can successful run the Wingtip Toys sample application at the end of this tutorial. The next section of this tutorial explains how to retrieve the local host port number and update the PayPal class.

## Update the LocalHost Port Number in the PayPal Class

The Wingtip Toys sample application purchases products by navigating to the PayPal testing site and returning to your local instance of the Wingtip Toys sample application. In order to have PayPal return to the correct URL, you need to specify the port number of the locally running sample application in the PayPal code mentioned above.

1. Right-click the project name (**WingtipToys**) in **Solution Explorer** and select **Properties**.
2. In the left column, select the **Web** tab.
3. Retrieve the port number from the **Project Url** box.
4. Update the `returnURL` and `cancelURL` in the PayPal class (`NVPAPICaller`) in the *PayPalFunctions.cs* file to use the port number of your web application:

```
    string returnURL = "http://localhost:<Your Port
Number>/Checkout/CheckoutReview.aspx";
    string cancelURL = "http://localhost:<Your Port
Number>/Checkout/CheckoutCancel.aspx";
```

Now the code that you added will match the expected port for your local Web application. PayPal will be able to return to the correct URL on your local machine.

## Add the PayPal Checkout Button

Now that the primary PayPal functions have been added to the sample application, you can begin adding the markup and code needed to call these functions. First, you must add the checkout button that the user will see on the shopping cart page.

1. Open the ShoppingCart.aspx file.
2. Scroll to the bottom of the file and find the `<!--Checkout Placeholder -->` comment.

3. Replace the comment with an `ImageButton` control so that the mark up is replaced as follows:

```
        <asp:ImageButton ID="CheckoutImageBtn" runat="server"

ImageUrl="https://www.paypal.com/en_US/i/btn/btn_xpressCheckout.gif"
                      Width="145" AlternateText="Check out with PayPal"
                      OnClick="CheckoutBtn_Click"
                      BackColor="Transparent" BorderWidth="0" />
```

4. In the *ShoppingCart.aspx.cs* file, after the `UpdateBtn_Click` event handler near the end of the file, add the `CheckOutBtn_Click` event handler:

```
protected void CheckoutBtn_Click(object sender, ImageClickEventArgs e)
{
  using (ShoppingCartActions usersShoppingCart = new ShoppingCartActions())
  {
    Session["payment_amt"] = usersShoppingCart.GetTotal();
  }
  Response.Redirect("Checkout/CheckoutStart.aspx");
}
```

5. Also in the *ShoppingCart.aspx.cs* file, add a reference to the `CheckoutBtn`, so that the new image button is referenced as follows:

```
    protected void Page_Load(object sender, EventArgs e)
    {
      using (ShoppingCartActions usersShoppingCart = new ShoppingCartActions())
      {
        decimal cartTotal = 0;
        cartTotal = usersShoppingCart.GetTotal();
        if (cartTotal > 0)
        {
          // Display Total.
          lblTotal.Text = String.Format("{0:c}", cartTotal);
        }
        else
        {
          LabelTotalText.Text = "";
          lblTotal.Text = "";
          ShoppingCartTitle.InnerText = "Shopping Cart is Empty";
          UpdateBtn.Visible = false;
          CheckoutImageBtn.Visible = false;
        }
      }
    }
```

6. Save your changes to both the *ShoppingCart.aspx* file and the *ShoppingCart.aspx.cs* file.
7. From the menu, select **Debug** -> **Build WingtipToys**.
   The project will be rebuilt with the newly added **ImageButton** control.

## Send Purchase Details to PayPal

When the user clicks the **Checkout** button on the shopping cart page (*ShoppingCart.aspx*), they'll begin the purchase process. The following code calls the first PayPal function needed to purchase products.

1. From the *Checkout* folder, open the code-behind file named *CheckoutStart.aspx.cs*. Be sure to open the code-behind file.
2. Replace the existing code with the following:

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

namespace WingtipToys.Checkout
{
  public partial class CheckoutStart : System.Web.UI.Page
  {
    protected void Page_Load(object sender, EventArgs e)
    {
      NVPAPICaller payPalCaller = new NVPAPICaller();
      string retMsg = "";
      string token = "";

      if (Session["payment_amt"] != null)
      {
        string amt = Session["payment_amt"].ToString();

        bool ret = payPalCaller.ShortcutExpressCheckout(amt, ref token, ref
retMsg);
        if (ret)
        {
          Session["token"] = token;
          Response.Redirect(retMsg);
        }
        else
        {
          Response.Redirect("CheckoutError.aspx?" + retMsg);
        }
      }
      else
      {
        Response.Redirect("CheckoutError.aspx?ErrorCode=AmtMissing");
      }
    }
  }
}
```

When the user of the application clicks the **Checkout** button on the shopping cart page, the browser will navigate to the *CheckoutStart.aspx* page. When the *CheckoutStart.aspx* page loads, the `ShortcutExpressCheckout` method is called. At this point, the user is transferred to the PayPal testing web site. On the PayPal site, the user enters their PayPal credentials, reviews the purchase details, accepts the PayPal agreement and returns to the Wingtip Toys sample application where the `ShortcutExpressCheckout` method completes. When the `ShortcutExpressCheckout` method is complete, it will redirect the user to the *CheckoutReview.aspx* page specified in the `ShortcutExpressCheckout` method. This allows the user to review the order details from within the Wingtip Toys sample application.

## Review Order Details

After returning from PayPal, the *CheckoutReview.aspx* page of the Wingtip Toys sample application displays the order details. This page allows the user to review the order details before purchasing the products. The *CheckoutReview.aspx* page must be created as follows:

1. In the *Checkout* folder, open the page named *CheckoutReview.aspx*.
2. Replace the existing markup with the following:

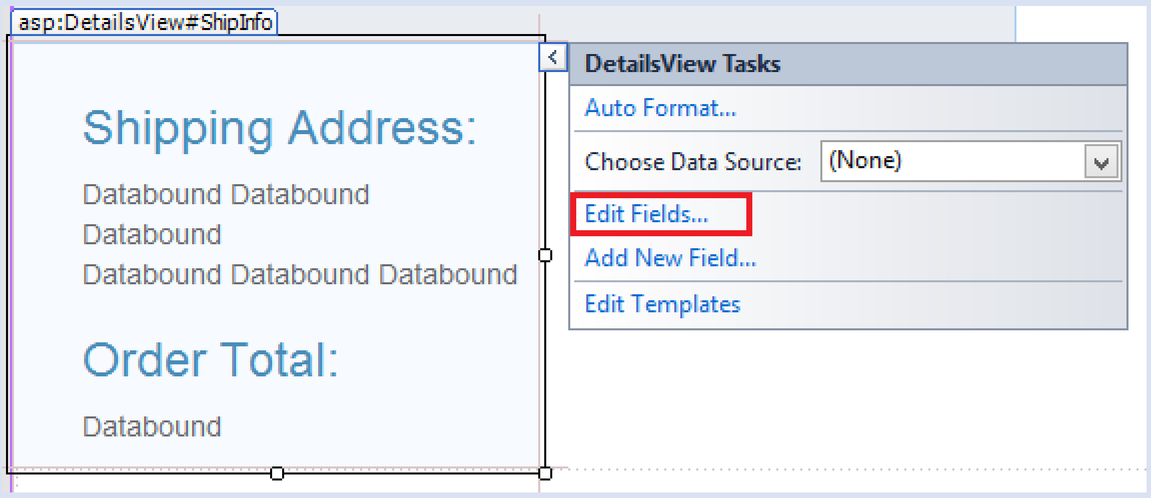
```
<%@ Page Title="" Language="C#" MasterPageFile="~/Site.Master"
AutoEventWireup="true" CodeBehind="CheckoutReview.aspx.cs"
Inherits="WingtipToys.Checkout.CheckoutReview" %>
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent" runat="server">
    <h1>Order Review</h1>
    <p></p>
    <h3 style="padding-left: 33px">Products:</h3>
    <asp:GridView ID="OrderItemList" runat="server" AutoGenerateColumns="False"
GridLines="Both" CellPadding="10" Width="500" BorderColor="#efeeef"
BorderWidth="33">
        <Columns>
            <asp:BoundField DataField="ProductId" HeaderText=" Product ID" />
            <asp:BoundField DataField="Product.ProductName" HeaderText="
Product Name" />
            <asp:BoundField DataField="Product.UnitPrice" HeaderText="Price
(each)" DataFormatString="{0:c}"/>
            <asp:BoundField DataField="Quantity" HeaderText="Quantity" />
        </Columns>
    </asp:GridView>
    <asp:DetailsView ID="ShipInfo" runat="server" AutoGenerateRows="false"
GridLines="None" CellPadding="10" BorderStyle="None" CommandRowStyle-
BorderStyle="None">
        <Fields>
        <asp:TemplateField>
            <ItemTemplate>
                <h3>Shipping Address:</h3>
                <br />
                <asp:Label ID="FirstName" runat="server" Text='<%#:
Eval("FirstName") %>'></asp:Label>
                <asp:Label ID="LastName" runat="server" Text='<%#:
Eval("LastName") %>'></asp:Label>
                <br />
                <asp:Label ID="Address" runat="server" Text='<%#:
Eval("Address") %>'></asp:Label>
                <br />
                <asp:Label ID="City" runat="server" Text='<%#: Eval("City")
%>'></asp:Label>
                <asp:Label ID="State" runat="server" Text='<%#: Eval("State")
%>'></asp:Label>
                <asp:Label ID="PostalCode" runat="server" Text='<%#:
Eval("PostalCode") %>'></asp:Label>
                <p></p>
                <h3>Order Total:</h3>
                <br />
                <asp:Label ID="Total" runat="server" Text='<%#: Eval("Total",
"{0:C}") %>'></asp:Label>
            </ItemTemplate>
            <ItemStyle HorizontalAlign="Left" />
        </asp:TemplateField>
          </Fields>
    </asp:DetailsView>
    <p></p>
    <hr />
    <asp:Button ID="CheckoutConfirm" runat="server" Text="Complete Order"
OnClick="CheckoutConfirm_Click" />
</asp:Content>
```

3. Open the code-behind page named *CheckoutReview.aspx.cs* and replace the existing code with the following:

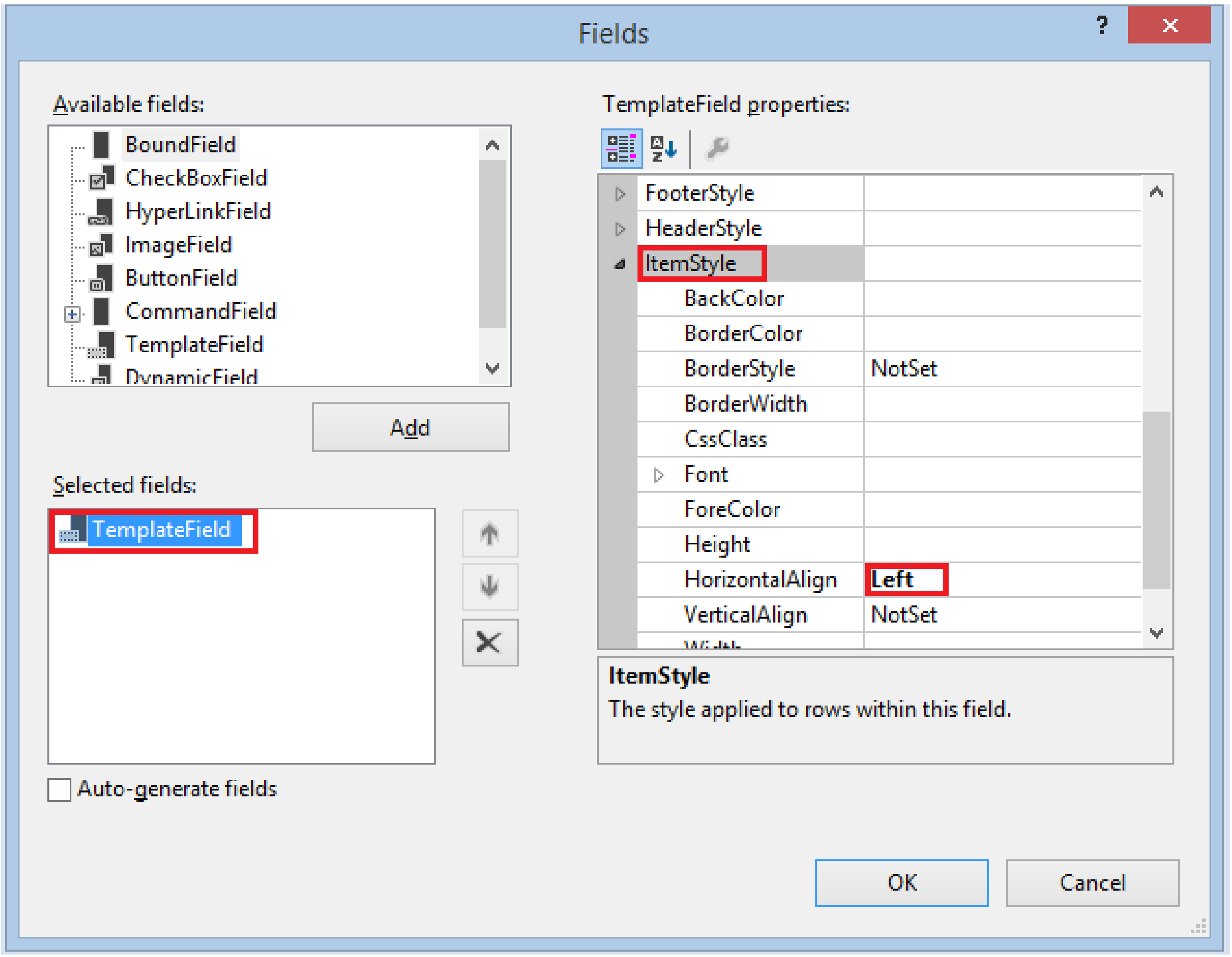
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using WingtipToys.Models;

namespace WingtipToys.Checkout
{
  public partial class CheckoutReview : System.Web.UI.Page
  {
    protected void Page_Load(object sender, EventArgs e)
    {
      if (!IsPostBack)
      {
        NVPAPICaller payPalCaller = new NVPAPICaller();

        string retMsg = "";
        string token = "";
        string PayerID = "";
        NVPCodec decoder = new NVPCodec();
        token = Session["token"].ToString();

        bool ret = payPalCaller.GetCheckoutDetails(token, ref PayerID, ref
decoder, ref retMsg);
        if (ret)
        {
          Session["payerId"] = PayerID;

          var myOrder = new Order();
          myOrder.OrderDate =
Convert.ToDateTime(decoder["TIMESTAMP"].ToString());
          myOrder.Username = User.Identity.Name;
          myOrder.FirstName = decoder["FIRSTNAME"].ToString();
          myOrder.LastName = decoder["LASTNAME"].ToString();
          myOrder.Address = decoder["SHIPTOSTREET"].ToString();
          myOrder.City = decoder["SHIPTOCITY"].ToString();
          myOrder.State = decoder["SHIPTOSTATE"].ToString();
          myOrder.PostalCode = decoder["SHIPTOZIP"].ToString();
          myOrder.Country = decoder["SHIPTOCOUNTRYCODE"].ToString();
          myOrder.Email = decoder["EMAIL"].ToString();
          myOrder.Total = Convert.ToDecimal(decoder["AMT"].ToString());

          // Verify total payment amount as set on CheckoutStart.aspx.
          try
          {
            decimal paymentAmountOnCheckout =
Convert.ToDecimal(Session["payment_amt"].ToString());
            decimal paymentAmoutFromPayPal =
Convert.ToDecimal(decoder["AMT"].ToString());
            if (paymentAmountOnCheckout != paymentAmoutFromPayPal)
            {
              Response.Redirect("CheckoutError.aspx?" +
"Desc=Amount%20total%20mismatch.");
            }
          }
          catch (Exception)
          {
            Response.Redirect("CheckoutError.aspx?" +
"Desc=Amount%20total%20mismatch.");
          }

          // Get DB context.
          ProductContext  db = new ProductContext();

          // Add order to DB.
          _db.Orders.Add(myOrder);
          _db.SaveChanges();

          // Get the shopping cart items and process them.
          using (WingtipToys.Logic.ShoppingCartActions usersShoppingCart = new
WingtipToys.Logic.ShoppingCartActions())
          {
            List<CartItem> myOrderList = usersShoppingCart.GetCartItems();

            // Add OrderDetail information to the DB for each product
purchased.
            for (int i = 0; i < myOrderList.Count; i++)
            {
              // Create a new OrderDetail object.
              var myOrderDetail = new OrderDetail();
              myOrderDetail.OrderId = myOrder.OrderId;
              myOrderDetail.Username = User.Identity.Name;
              myOrderDetail.ProductId = myOrderList[i].ProductId;
              myOrderDetail.Quantity = myOrderList[i].Quantity;
              myOrderDetail.UnitPrice = myOrderList[i].Product.UnitPrice;

              // Add OrderDetail to DB.
              _db.OrderDetails.Add(myOrderDetail);
              _db.SaveChanges();
            }

            // Set OrderId.
            Session["currentOrderId"] = myOrder.OrderId;

            // Display Order information.
            List<Order> orderList = new List<Order>();
            orderList.Add(myOrder);
            ShipInfo.DataSource = orderList;
            ShipInfo.DataBind();

            // Display OrderDetails.
            OrderItemList.DataSource = myOrderList;
            OrderItemList.DataBind();
          }
        }
        else
        {
          Response.Redirect("CheckoutError.aspx?" + retMsg);
        }
      }
    }

    protected void CheckoutConfirm_Click(object sender, EventArgs e)
    {
      Session["userCheckoutCompleted"] = "true";
      Response.Redirect("~/Checkout/CheckoutComplete.aspx");
    }
    }
}
```
The **DetailsView** control is used to display the order details that have been returned from PayPal. Also, the above code saves the order details to the Wingtip Toys database as an `OrderDetail` object. When the user clicks on the **Complete Order** button, they are redirected to the *CheckoutComplete.aspx* page.

Tip

Notice that the `<ItemStyle>` tag is used to change the style of the items within the **DetailsView** control. By viewing the page in **Design View**, selecting the **DetailsView** control, and selecting the **Smart Tag** (the arrow icon at the top right of the control), you will be able to see the **DetailsView Tasks**.



By selecting **Edit Fields**, the **Fields** dialog box will appear. In this dialog box you can easily control the visual properties, such as **ItemStyle**, of the **DetailsView** control.



FooterStyle
HeaderStyle
ItemStyle
BackColor
BorderColor
BorderStyle NotSet
BorderWidth
CssClass
Font
ForeColor
Height
HorizontalAlign Left
VerticalAlign NotSet

**ItemStyle**
The style applied to rows within this field.

## Complete Purchase

*CheckoutComplete.aspx* page makes the purchase from PayPal. As mentioned above, the user must click on the **Complete Order** button before the application will navigate to the *CheckoutComplete.aspx* page.

1. In the *Checkout* folder, open the page named *CheckoutComplete.aspx*.
2. Replace the existing markup with the following:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Site.Master"
AutoEventWireup="true" CodeBehind="CheckoutComplete.aspx.cs"
Inherits="WingtipToys.Checkout.CheckoutComplete" %>
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent" runat="server">
    <h1>Checkout Complete</h1>
    <p></p>
    <h3>Payment Transaction ID:</h3> <asp:Label ID="TransactionId"
runat="server"></asp:Label>
    <p></p>
    <h3>Thank You!</h3>
    <p></p>
    <hr />
    <asp:Button ID="Continue" runat="server" Text="Continue Shopping"
OnClick="Continue_Click" />
</asp:Content>
```

3. Open the code-behind page named *CheckoutComplete.aspx.cs* and replace the existing code with the following:

```
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.Checkout
{
  public partial class CheckoutComplete : System.Web.UI.Page
  {
    protected void Page_Load(object sender, EventArgs e)
    {
      if (!IsPostBack)
      {
        // Verify user has completed the checkout process.
        if ((string)Session["userCheckoutCompleted"] != "true")
        {
          Session["userCheckoutCompleted"] = string.Empty;
          Response.Redirect("CheckoutError.aspx?" +
"Desc=Unvalidated%20Checkout.");
        }

        NVPAPICaller payPalCaller = new NVPAPICaller();

        string retMsg = "";
        string token = "";
        string finalPaymentAmount = "";
        string PayerID = "";
        NVPCodec decoder = new NVPCodec();

        token = Session["token"].ToString();
        PayerID = Session["payerId"].ToString();
        finalPaymentAmount = Session["payment_amt"].ToString();

        bool ret = payPalCaller.DoCheckoutPayment(finalPaymentAmount, token,
PayerID, ref decoder, ref retMsg);
        if (ret)
        {
          // Retrieve PayPal confirmation value.
          string PaymentConfirmation =
decoder["PAYMENTINFO_0_TRANSACTIONID"].ToString();
          TransactionId.Text = PaymentConfirmation;


          ProductContext _db = new ProductContext();
          // Get the current order id.
          int currentOrderId = -1;
          if (Session["currentOrderId"] != string.Empty)
          {
            currentOrderId = Convert.ToInt32(Session["currentOrderID"]);
          }
          Order myCurrentOrder;
          if (currentOrderId >= 0)
          {
            // Get the order based on order id.
            myCurrentOrder = _db.Orders.Single(o => o.OrderId ==
currentOrderId);
            // Update the order to reflect payment has been completed.
            myCurrentOrder.PaymentTransactionId = PaymentConfirmation;
            // Save to DB.
            _db.SaveChanges();
          }

          // Clear shopping cart.
          using (WingtipToys.Logic.ShoppingCartActions usersShoppingCart =
              new WingtipToys.Logic.ShoppingCartActions())
          {
            usersShoppingCart.EmptyCart();
          }

          // Clear order id.
          Session["currentOrderId"] = string.Empty;
        }
        else
        {
          Response.Redirect("CheckoutError.aspx?" + retMsg);
        }
      }
    }

    protected void Continue_Click(object sender, EventArgs e)
    {
      Response.Redirect("~/Default.aspx");
    }
  }
}
```
When the *CheckoutComplete.aspx* page is loaded, the `DoCheckoutPayment` method is called. As mentioned earlier, the `DoCheckoutPayment` method completes the purchase from the PayPal testing environment. Once PayPal has completed the purchase of the order, the *CheckoutComplete.aspx* page displays a payment transaction `ID` to the purchaser.

## Handle Cancel Purchase

If the user decides to cancel the purchase, they will be directed to the *CheckoutCancel.aspx* page where they will see that their order has been cancelled.

1. Open the page named *CheckoutCancel.aspx* in the *Checkout* folder.
2. Replace the existing markup with the following:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Site.Master"
AutoEventWireup="true" CodeBehind="CheckoutCancel.aspx.cs"
Inherits="WingtipToys.Checkout.CheckoutCancel" %>
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent" runat="server">
    <h1>Checkout Cancelled</h1>
    <p></p>
    <h3>Your purchase has been cancelled.</h3>
</asp:Content>
```

## Handle Purchase Errors

Errors during the purchase process will be handled by the *CheckoutError.aspx* page. The code-behind of the *CheckoutStart.aspx* page, the *CheckoutReview.aspx* page, and the *CheckoutComplete.aspx* page will each redirect to the *CheckoutError.aspx* page if an error occurs.

1. Open the page named *CheckoutError.aspx* in the *Checkout* folder.
2. Replace the existing markup with the following:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Site.Master"
AutoEventWireup="true" CodeBehind="CheckoutError.aspx.cs"
Inherits="WingtipToys.Checkout.CheckoutError" %>
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent" runat="server">
    <h1>Checkout Error</h1>
    <p></p>
       <table id="ErrorTable">
              <tr>
                     <td class="field"></td>
                     <td><%=Request.QueryString.Get("ErrorCode")%></td>
              </tr>
              <tr>
                     <td class="field"></td>
                     <td><%=Request.QueryString.Get("Desc")%></td>
              </tr>
              <tr>
                     <td class="field"></td>
                     <td><%=Request.QueryString.Get("Desc2")%></td>
              </tr>
       </table>
    <p></p>
</asp:Content>
```

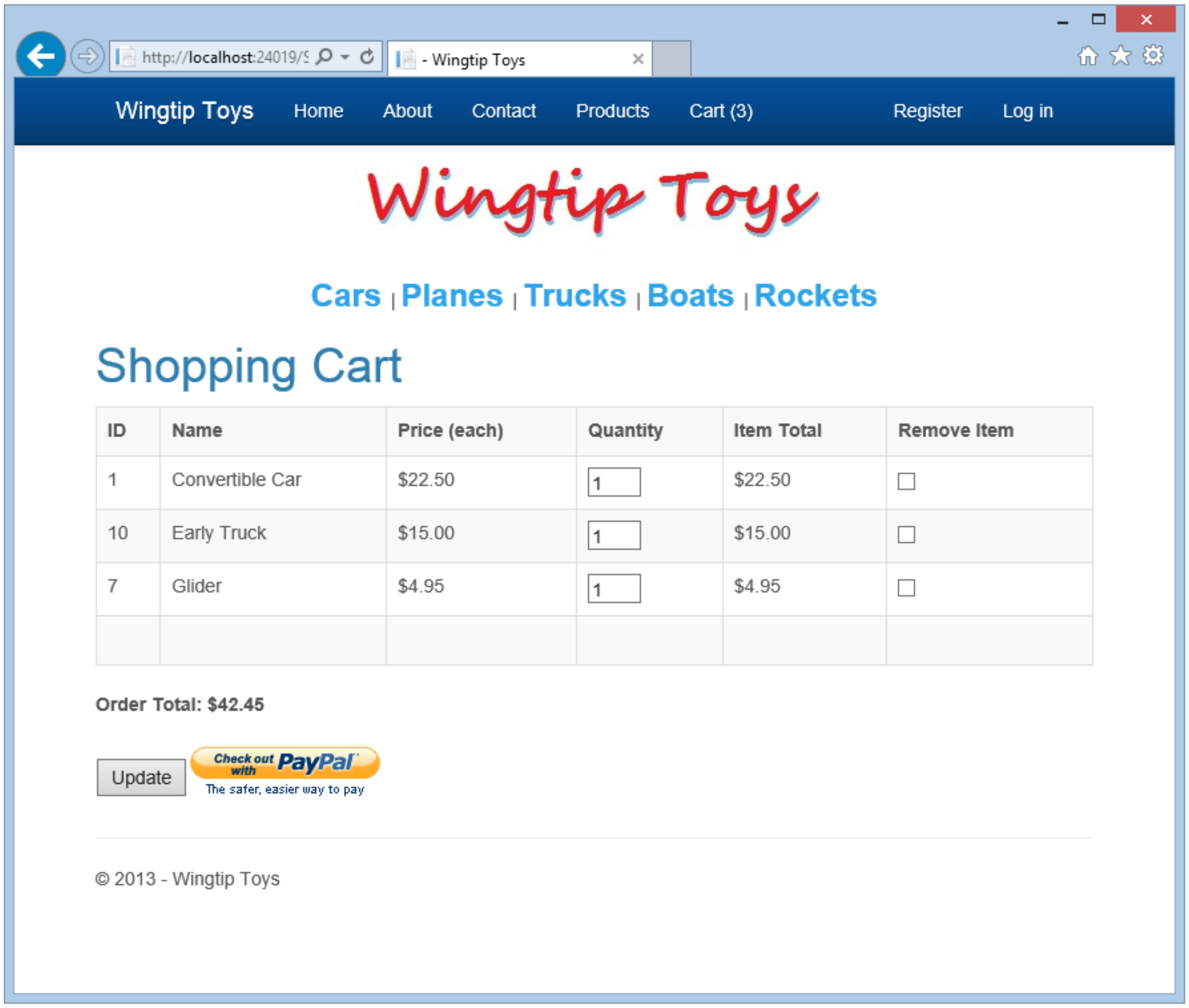
The *CheckoutError.aspx* page is displayed with the error details when an error occurs during the checkout process.

## Running the Application

Run the application to see how to purchase products.

1. Open a Web browser and navigate to https://developer.paypal.com.
2. Login with your PayPal developer account that you created earlier in this tutorial.
   For PayPal's developer sandbox, you need to be logged in at https://developer.paypal.com to test express checkout. This only applies to PayPal's sandbox testing, not to PayPal's live environment.
3. In Visual Studio, press **F5** to run the Wingtip Toys sample application.
   After the database rebuilds, the browser will open and show the *Default.aspx* page.
4. Add three different products to the shopping cart by selecting the product category, such as "Cars" and then clicking **Add to Cart** next to each product.
   The shopping cart will display the product you have selected.

5. Click the **PayPal** button to checkout.



Shopping Cart

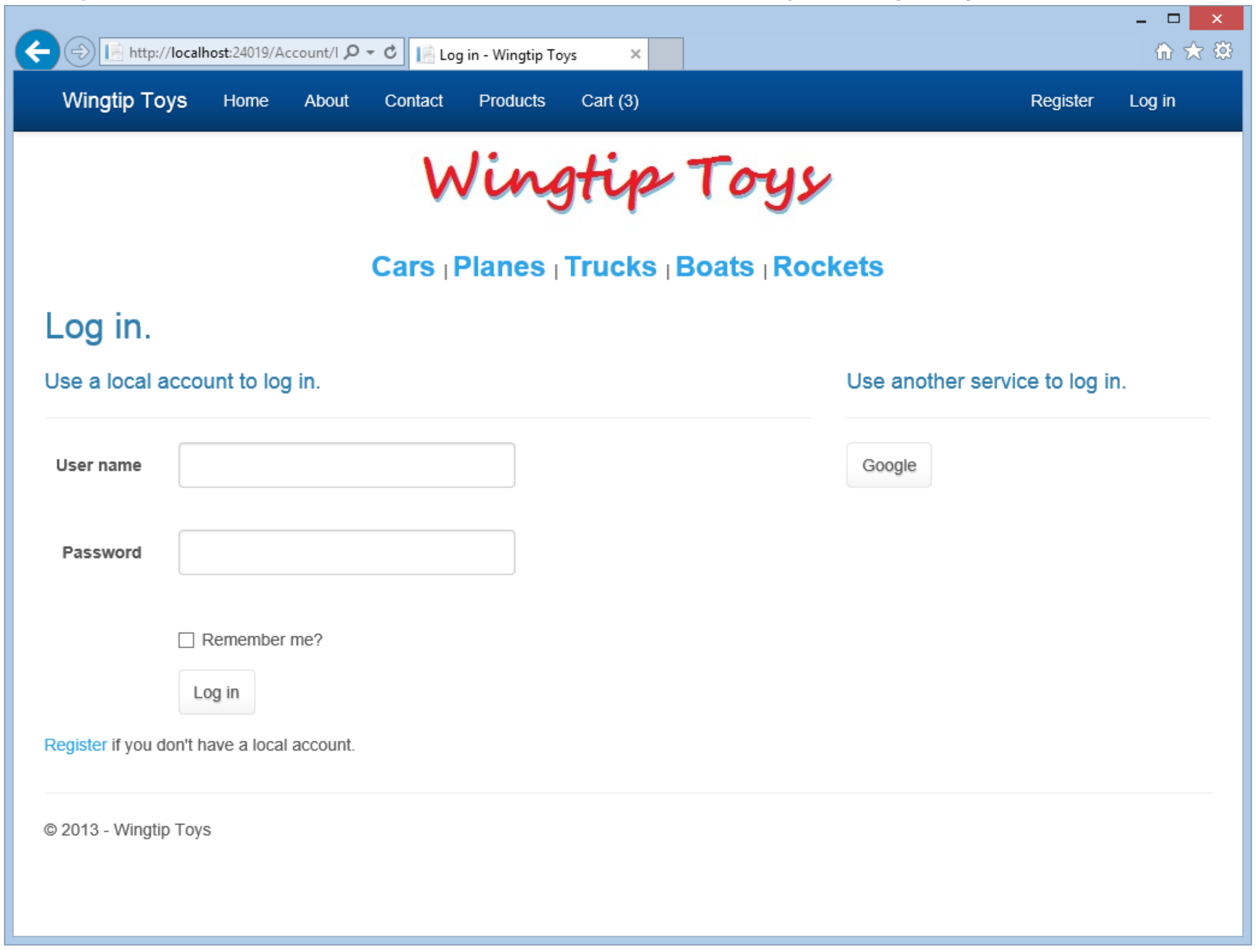
| ID | Name | Price (each) | Quantity | Item Total | Remove Item |
|---|---|---|---|---|---|
| 1 | Convertible Car | $22.50 | 1 | $22.50 | ☐ |
| 10 | Early Truck | $15.00 | 1 | $15.00 | ☐ |
| 7 | Glider | $4.95 | 1 | $4.95 | ☐ |
| | | | | | |

Order Total: $42.45

© 2013 - Wingtip Toys

6. Checking out will require that you have a user account for the Wingtip Toys sample application.
7. Click the **Google** link on the right of the page to log in with an existing gmail.com email account.
   If you do not have a gmail.com account, you can create one for testing purposes at

www.gmail.com. You can also use a standard local account by clicking "Register".



Log in.

Use a local account to log in.

User name

Password

Remember me?

Log in

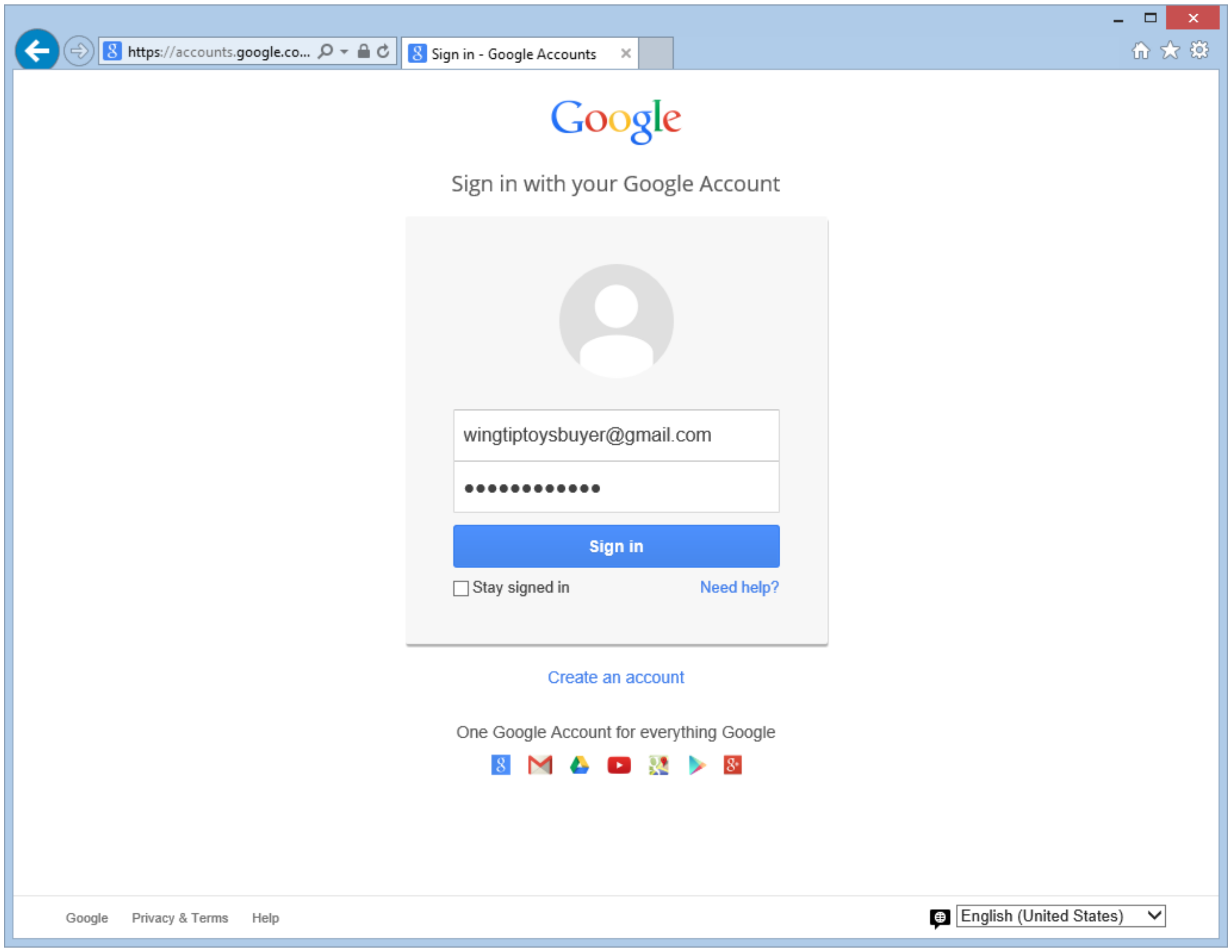
Register if you don't have a local account.

Use another service to log in.

Google

© 2013 - Wingtip Toys

8. Sign in with your gmail account and password.



https://accounts.google.co...

Sign in - Google Accounts

Google

Sign in with your Google Account

wingtiptoysbuyer@gmail.com

●●●●●●●●●●●●

Sign in

Stay signed in

Need help?

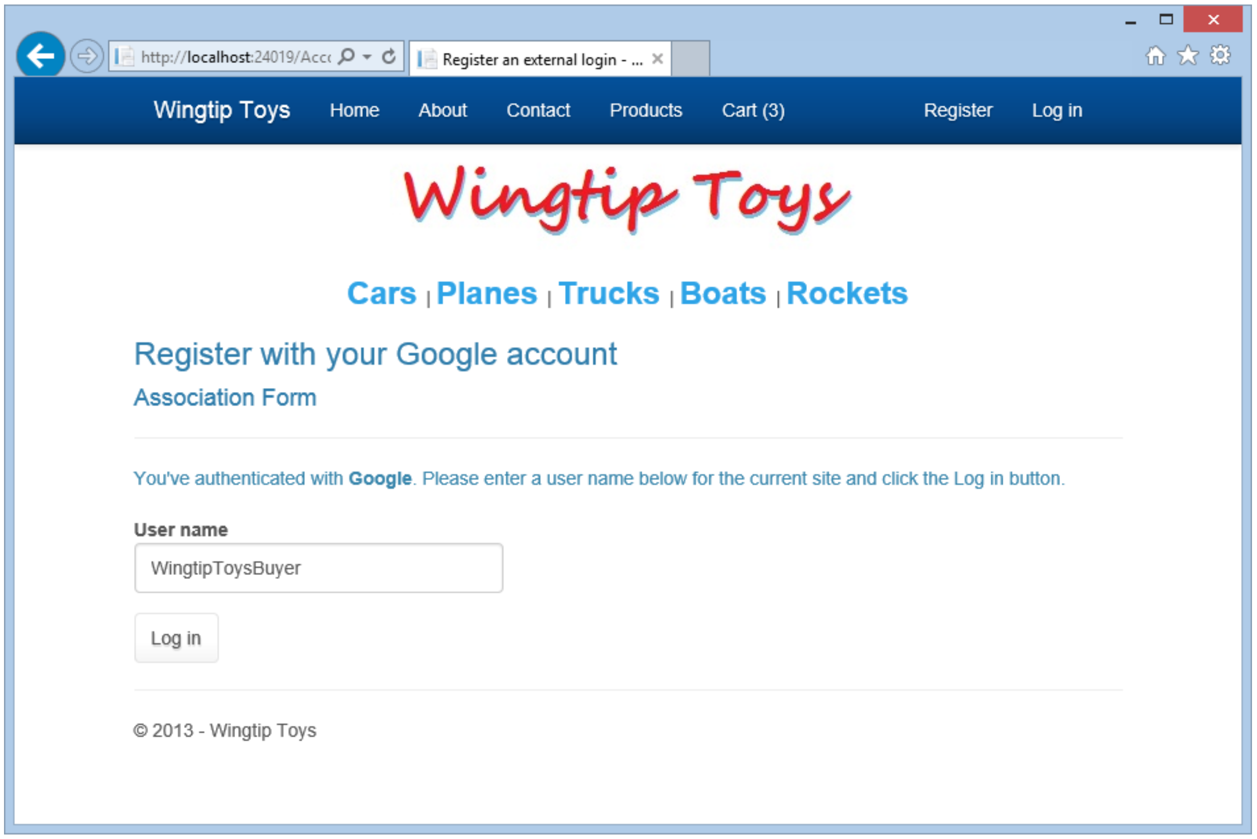
Create an account

One Google Account for everything Google

Google Privacy & Terms Help

English (United States)

9. Click the **Log in** button to register your gmail account with your Wingtip Toys sample application user name.


http://localhost:24019/Acc
Register an external login - ...
Wingtip Toys
Home
About
Contact
Products
Cart (3)
Register
Log in
Wingtip Toys
Cars | Planes | Trucks | Boats | Rockets

Register with your Google account

Association Form

You've authenticated with **Google**. Please enter a user name below for the current site and click the Log in button.

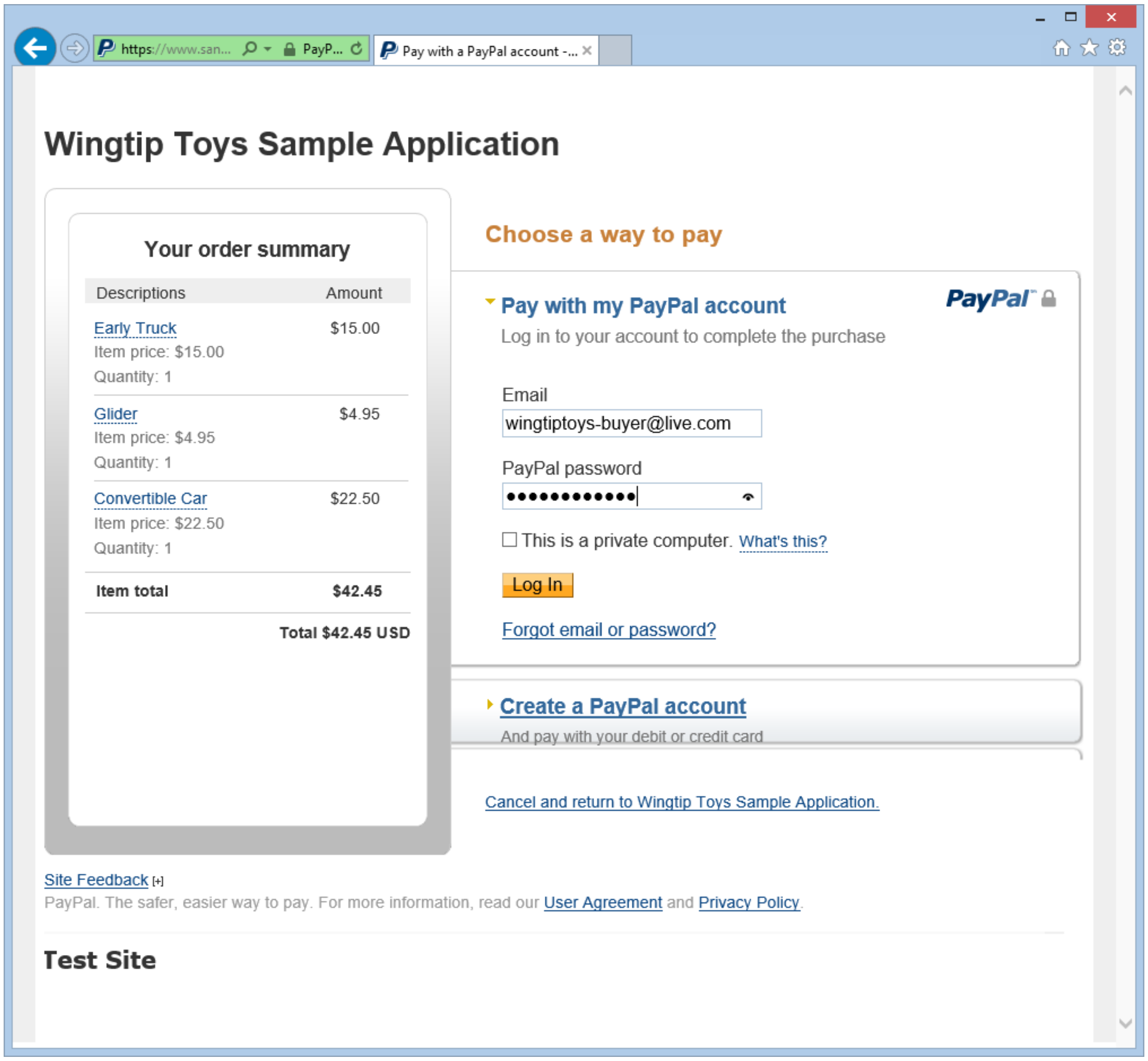
**User name**

WingtipToysBuyer

Log in

© 2013 - Wingtip Toys

10. On the PayPal test site, add your **buyer** email address and password that you created earlier in this tutorial, then click the **Log In** button.



Wingtip Toys Sample Application

Your order summary

| Descriptions | Amount |
|---|---|
| Early Truck<br>Item price: $15.00<br>Quantity: 1 | $15.00 |
| Glider<br>Item price: $4.95<br>Quantity: 1 | $4.95 |
| Convertible Car<br>Item price: $22.50<br>Quantity: 1 | $22.50 |
| **Item total** | **$42.45** |
| | **Total $42.45 USD** |

Choose a way to pay

Pay with my PayPal account

Log in to your account to complete the purchase

Email
wingtiptoys-buyer@live.com

PayPal password
•••••••••••••

☐ This is a private computer. What's this?

Log In

Forgot email or password?

Create a PayPal account

And pay with your debit or credit card

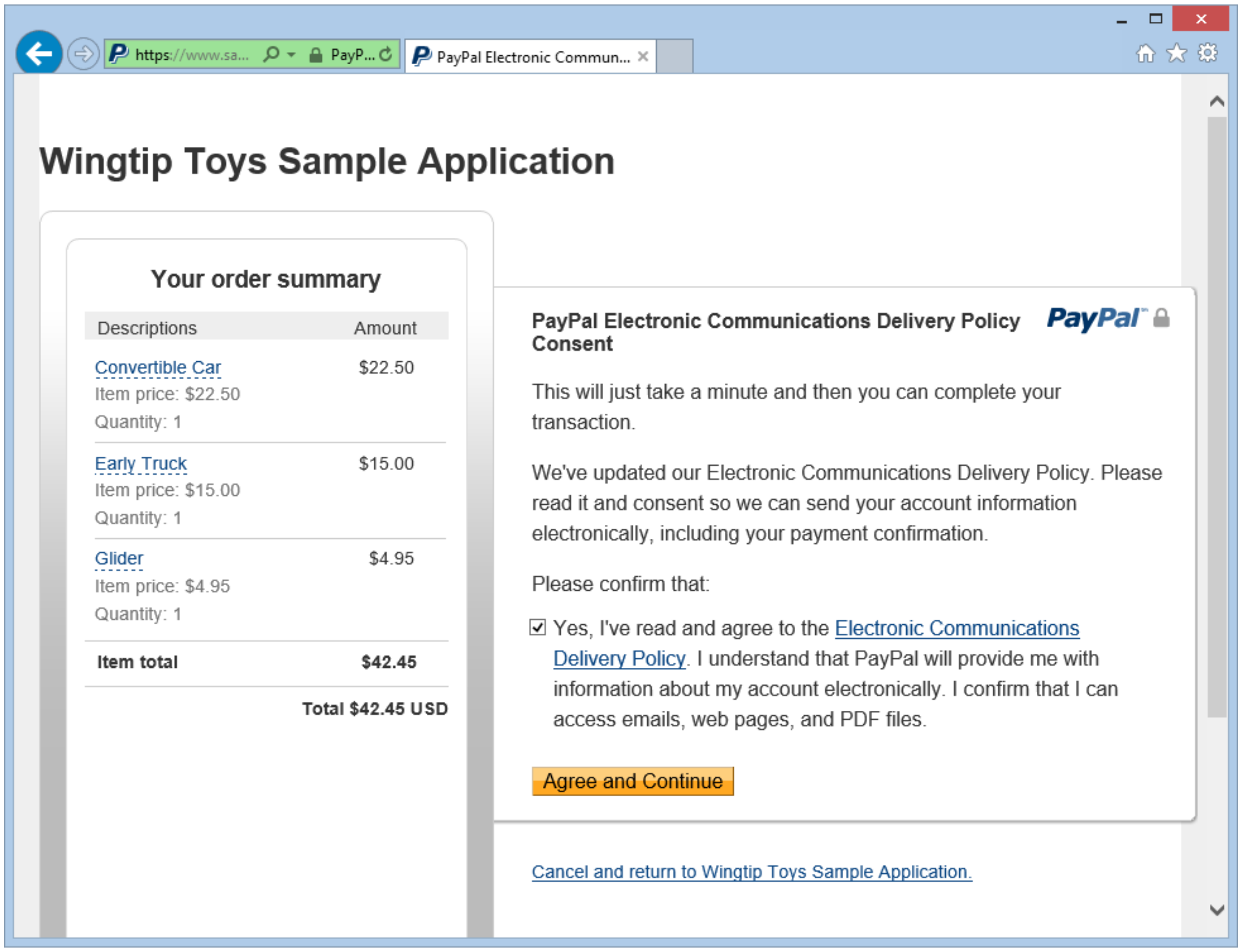
Cancel and return to Wingtip Toys Sample Application.

Site Feedback [+]

PayPal. The safer, easier way to pay. For more information, read our User Agreement and Privacy Policy.

Test Site

11. Agree to the PayPal policy and click the **Agree and Continue** button.
    Note that this page is only displayed the first time you use this PayPal account.



Wingtip Toys Sample Application

Your order summary

| Descriptions | Amount |
|---|---|
| Convertible Car<br>Item price: $22.50<br>Quantity: 1 | $22.50 |
| Early Truck<br>Item price: $15.00<br>Quantity: 1 | $15.00 |
| Glider<br>Item price: $4.95<br>Quantity: 1 | $4.95 |
| **Item total** | **$42.45** |

**Total $42.45 USD**

**PayPal Electronic Communications Delivery Policy Consent**

This will just take a minute and then you can complete your transaction.

We've updated our Electronic Communications Delivery Policy. Please read it and consent so we can send your account information electronically, including your payment confirmation.

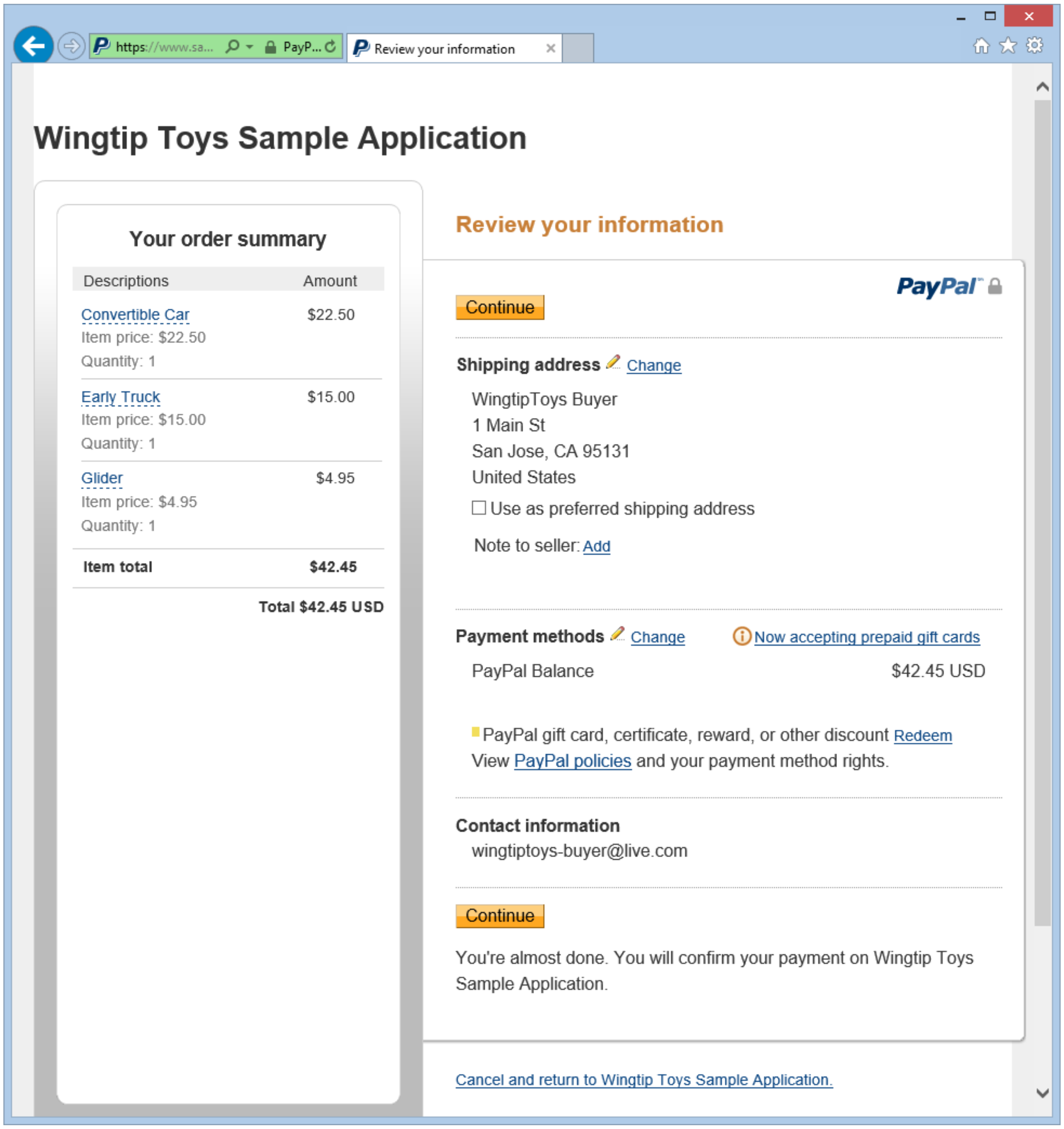
Please confirm that:

☑ Yes, I've read and agree to the Electronic Communications Delivery Policy. I understand that PayPal will provide me with information about my account electronically. I confirm that I can access emails, web pages, and PDF files.

Agree and Continue

Cancel and return to Wingtip Toys Sample Application.

12. Review the order information on the PayPal testing environment review page and click **Continue**.



Wingtip Toys Sample Application

Your order summary

| Descriptions | Amount |
|---|---|
| Convertible Car<br>Item price: $22.50<br>Quantity: 1 | $22.50 |
| Early Truck<br>Item price: $15.00<br>Quantity: 1 | $15.00 |
| Glider<br>Item price: $4.95<br>Quantity: 1 | $4.95 |
| **Item total** | **$42.45** |
| | **Total $42.45 USD** |

Review your information

PayPal

Continue

**Shipping address** Change

WingtipToys Buyer
1 Main St
San Jose, CA 95131
United States

☐ Use as preferred shipping address

Note to seller: Add

**Payment methods** Change   Now accepting prepaid gift cards

PayPal Balance $42.45 USD

PayPal gift card, certificate, reward, or other discount Redeem
View PayPal policies and your payment method rights.

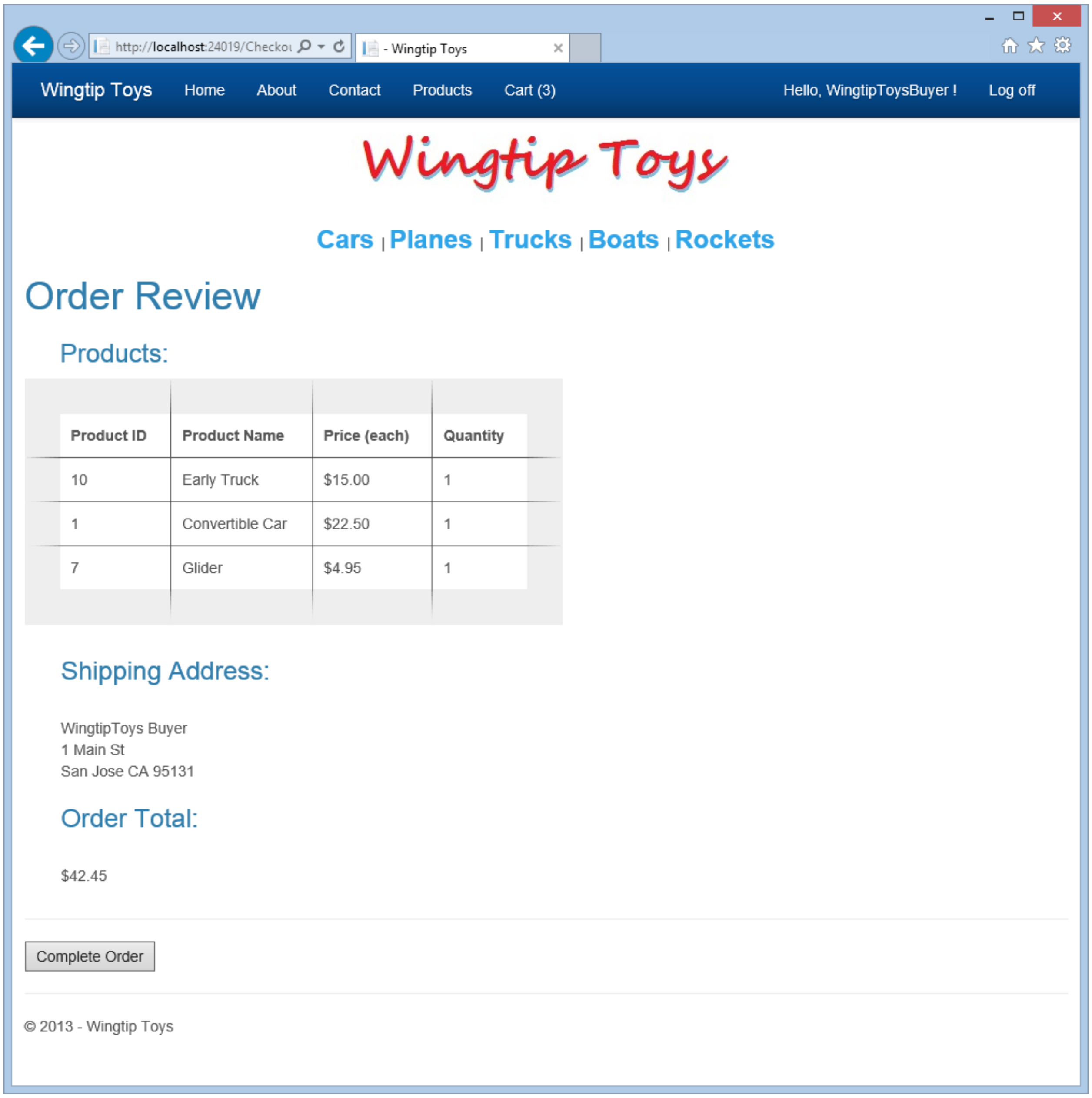
**Contact information**
wingtiptoys-buyer@live.com

Continue

You're almost done. You will confirm your payment on Wingtip Toys Sample Application.

Cancel and return to Wingtip Toys Sample Application.

13. On the *CheckoutReview.aspx* page, verify the order amount and view the generated shipping address. Then, click the **Complete Order** button.


http://localhost:24019/Checkou
- Wingtip Toys
Wingtip Toys
Home
About
Contact
Products
Cart (3)
Hello, WingtipToysBuyer !
Log off
Wingtip Toys
Cars | Planes | Trucks | Boats | Rockets

Order Review

Products:

| Product ID | Product Name | Price (each) | Quantity |
|---|---|---|---|
| 10 | Early Truck | $15.00 | 1 |
| 1 | Convertible Car | $22.50 | 1 |
| 7 | Glider | $4.95 | 1 |

Shipping Address:

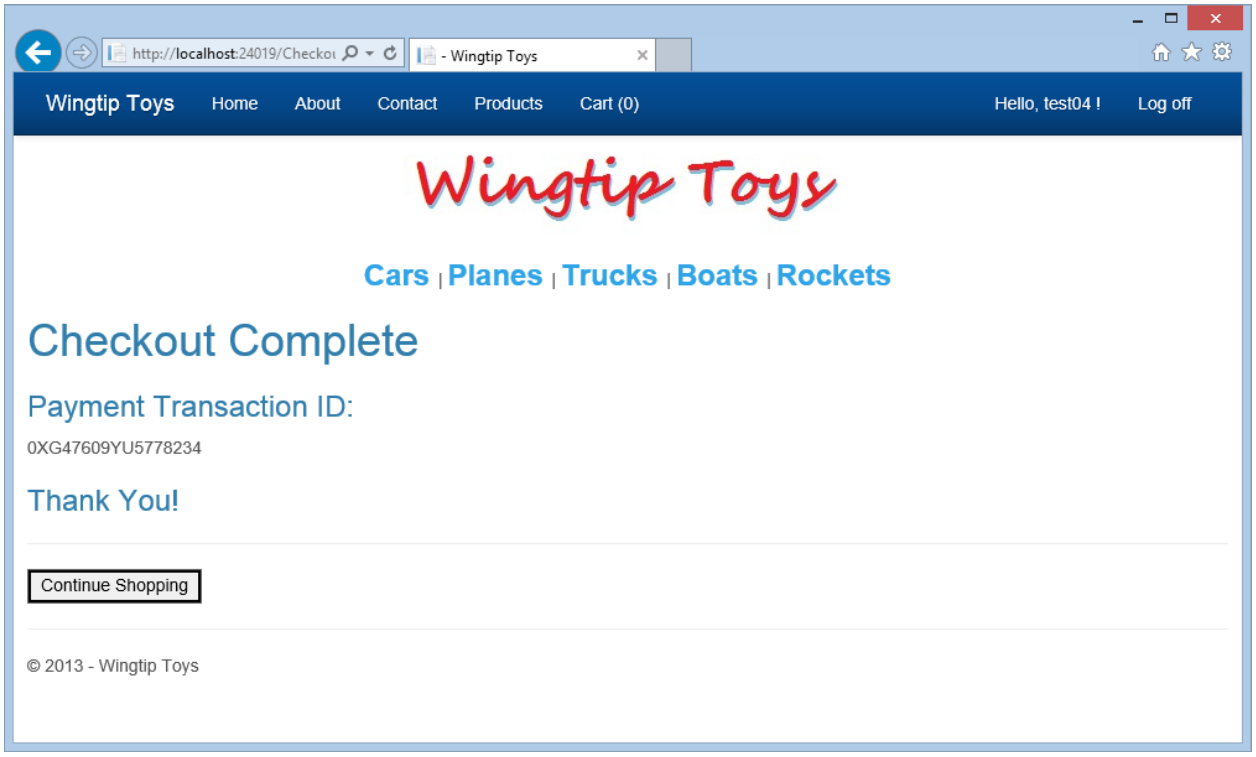
WingtipToys Buyer
1 Main St
San Jose CA 95131

Order Total:

$42.45

© 2013 - Wingtip Toys

14. The **CheckoutComplete.aspx** page is displayed with a payment transaction ID.



Checkout Complete

Payment Transaction ID:

0XG47609YU5778234

Thank You!

Continue Shopping

© 2013 - Wingtip Toys

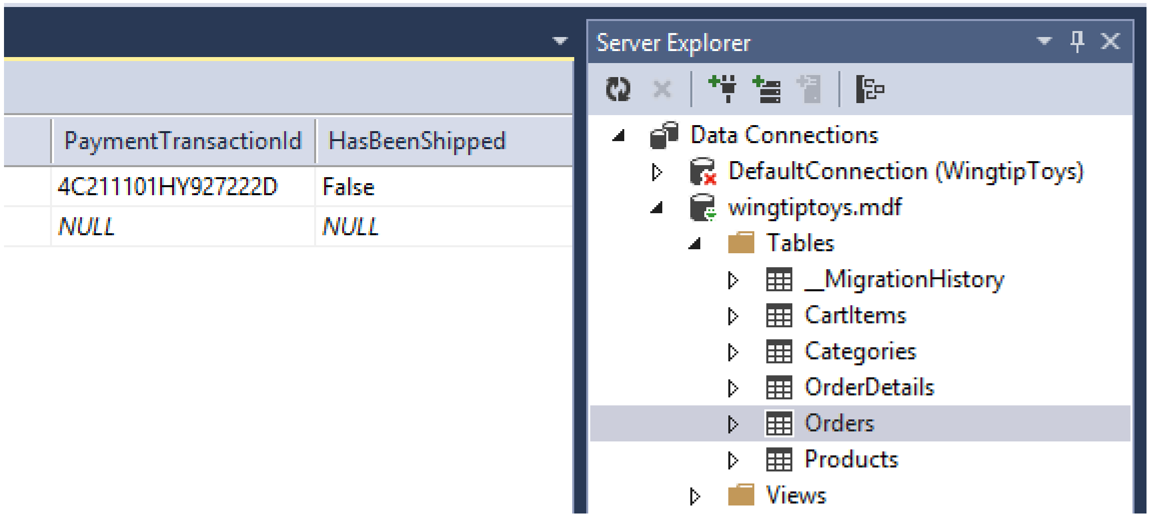
## Reviewing the Database

By reviewing the updated data in the Wingtip Toys sample application database after running the application, you can see that the application successfully recorded the purchase of the products.

You can inspect the data contained in the *Wingtiptoys.mdf* database file by using the **Database Explorer** window (**Server Explorer** window in Visual Studio) as you did earlier in this tutorial series.

1. Close the browser window if it is still open.
2. In Visual Studio, select the **Show All Files** icon at the top of **Solution Explorer** to allow you to expand the **App_Data** folder.
3. Expand the **App_Data** folder.
   You may need to select the **Show All Files** icon for the folder.
4. Right-click the *Wingtiptoys.mdf* database file and select **Open**.
   **Server Explorer** is displayed.
5. Expand the **Tables** folder.
6. Right-click the **Orders** table and select **Show Table Data**.
   The **Orders** table is displayed.

7. Review the **PaymentTransactionID** column to confirm successful transactions.



| | PaymentTransactionId | HasBeenShipped |
|---|---|---|
| | 4C211101HY927222D | False |
| | *NULL* | *NULL* |

8. Close the **Orders** table window.
9. In the Server Explorer, right-click the **OrderDetails** table and select **Show Table Data**.
10. Review the `OrderId` and `Username` values in the **OrderDetails** table. Note that these values match the `OrderId` and `Username` values included in the **Orders** table.
11. Close the **OrderDetails** table window.
12. Right-click the Wingtip Toys database file (*Wingtiptoys.mdf*) and select **Close Connection**.
13. If you do not see the **Solution Explorer** window, click **Solution Explorer** at the bottom of the **Server Explorer** window to show the **Solution Explorer** again.

## Summary

In this tutorial you added order and order detail schemas to track the purchase of products. You also integrated PayPal functionality into the Wingtip Toys sample application.

## Additional Resources

ASP.NET Configuration Overview
Create an ASP.NET MVC 5 App with Facebook and Google OAuth2 and OpenID Sign-on (C#)

## Disclaimer

## Membership and Administration

This tutorial series will teach you the basics of building an ASP.NET Web Forms application using ASP.NET 4.5 and Microsoft Visual Studio Express 2013 for Web. A Visual Studio 2013 project with C# source code is available to accompany this tutorial series.

This tutorial shows you how to update the Wingtip Toys sample application to add an administrator role and use ASP.NET Identity. It also shows you how to implement an administration page from which the administrator can add and remove products from the website.

ASP.NET Identity is the membership system used to build ASP.NET web application and is available in ASP.NET 4.5. ASP.NET Identity is used in the Visual Studio 2013 Web Forms project template, as well as the templates for ASP.NET MVC, ASP.NET Web API, and ASP.NET Single Page Application. You can also specifically install the ASP.NET Identity system using NuGet when you start with an empty Web application. However, in this tutorial series you use the **Web Forms** project template, which includes the ASP.NET Identity system. ASP.NET Identity makes it easy to integrate user-specific profile data with application data. Also, ASP.NET Identity allows you to choose the persistence model for user profiles in your application. You can store the data in a SQL Server database or another data store, including *NoSQL* data stores such as Windows Azure Storage Tables.

This tutorial builds on the previous tutorial titled "Checkout and Payment with PayPal" in the Wingtip Toys tutorial series.

## What you'll learn:

- How to use code to add an administrator role and a user to the application.
- How to restrict access to the administration folder and page.
- How to provide navigation for the administrator role.
- How to use model binding to populate a DropDownList control with product categories.
- How to upload a file to the web application using the FileUpload control.
- How to use validation controls to implement input validation.
- How to add and remove products from the application.

## These features are included in the tutorial:

- ASP.NET Identity
- Configuration and Authorization
- Model Binding
- Unobtrusive Validation

ASP.NET Web Forms provides membership capabilities. By using the default template, you have built-in membership functionality that you can immediately use when the application runs. This tutorial shows you how to use ASP.NET Identity to add an administrator role and assign a user to that role. You will learn how to restrict access to the administration folder. You'll add a page to the administration folder that allows an administrator to add and remove products, and to preview a product after it has been added.

## Adding an Administrator

Using ASP.NET Identity, you can add an administrator role and assign a user to that role using code.

1. In Solution Explorer, right-click on the *Logic* folder and create a new class.
2. Name the new class *RoleActions.cs*.
3. Modify the code so that it appears as follows:

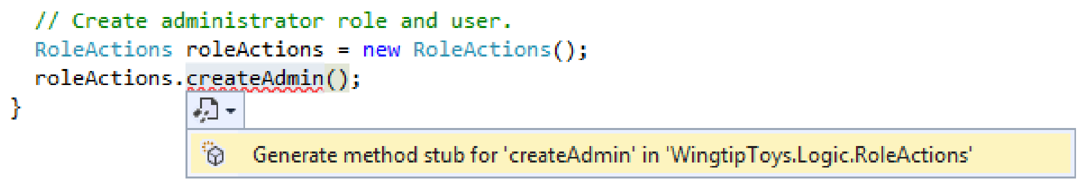
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace WingtipToys.Logic
{
  internal class RoleActions
  {
  }
}
```

4. In Solution Explorer, open the *Global.asax.cs* file.
5. Open and modify the *Global.asax.cs* file by added the code highlighted in yellow so that it appears as follows:

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

namespace WingtipToys
{
    public class Global : HttpApplication
    {
        void Application_Start(object sender, EventArgs e)
        {
          // Code that runs on application startup
          RouteConfig.RegisterRoutes(RouteTable.Routes);
          BundleConfig.RegisterBundles(BundleTable.Bundles);

          // Initialize the product database.
          Database.SetInitializer(new ProductDatabaseInitializer());
```

```
            // Create the administrator role and user.
            RoleActions roleActions = new RoleActions();
            roleActions.createAdmin();
          }
      }
}
```

6. Notice that `createAdmin` is underlined in red. Double-click the `createAdmin` code. The letter "c" in the highlighted method will be underlined.
7. Next, hover over the letter "c" to display the UI that allows you to generate a method stub for the `createAdmin` method.



```
// Create administrator role and user.
RoleActions roleActions = new RoleActions();
roleActions.createAdmin();
}
```

Generate method stub for 'createAdmin' in 'WingtipToys.Logic.RoleActions'

8. Click the optioned titled:
   `Generate method stub for 'createAdmin' in "WingtipToys.Logic.RoleActions'`
9. Open the *RoleActions.cs* file from the *Logic* folder.
   The `createAdmin` method has been added to the class file.
10. Modify the *RoleActions.cs* file by removing the `NotImplementedeException` and adding the code highlighted in yellow, so that it appears as follows:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using WingtipToys.Models;
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;

namespace WingtipToys.Logic
{
  internal class RoleActions
  {
    internal void createAdmin()
    {
      // Access the application context and create result variables.
      Models.ApplicationDbContext context = new ApplicationDbContext();
      IdentityResult IdRoleResult;
      IdentityResult IdUserResult;

      // Create a RoleStore object by using the ApplicationDbContext object.
      // The RoleStore is only allowed to contain IdentityRole objects.
      var roleStore = new RoleStore<IdentityRole>(context);

      // Create a RoleManager object that is only allowed to contain
IdentityRole objects.
      // When creating the RoleManager object, you pass in (as a parameter) a
new RoleStore object.
      var roleMgr = new RoleManager<IdentityRole>(roleStore);

      // Then, you create the "Administrator" role if it doesn't already exist.
      if (!roleMgr.RoleExists("Administrator"))
      {
        IdRoleResult = roleMgr.Create(new IdentityRole("Administrator"));
        if (!IdRoleResult.Succeeded)
        {
          // Handle the error condition if there's a problem creating the
RoleManager object.
        }
      }

      // Create a UserManager object based on the UserStore object and the
ApplicationDbContext
      // object. Note that you can create new objects and use them as
parameters in
      // a single line of code, rather than using multiple lines of code, as
you did
      // for the RoleManager object.
      var userMgr = new UserManager<ApplicationUser>(new
UserStore<ApplicationUser>(context));
      var appUser = new ApplicationUser()
      {
        UserName = "Admin",
      };
      IdUserResult = userMgr.Create(appUser, "Pa$$word");

      // If the new "Admin" user was successfully created,
      // add the "Admin" user to the "Administrator" role.
      if (IdUserResult.Succeeded)
      {
        IdUserResult = userMgr.AddToRole(appUser.Id, "Administrator");
        if (!IdUserResult.Succeeded)
        {
          // Handle the error condition if there's a problem adding the user to
the role.
        }
      }
      else
      {
        // Handle the error condition if there's a problem creating the new
user.
      }
    }
  }
}
```

The above code works by first establishing a database context for the membership database. The membership database is also stored as an *.mdf* file in the *App_Data* folder. You will be able to view this database once the first user has signed in to this web application.

**Note**

If you wish to store the membership data along with the product data, you can consider using the same `DbContext` that you used to store the product data in the above code.

The *internal* keyword is an access modifier for types (such as classes) and type members (such as methods or properties). Internal types or members are accessible only within files contained in the same assembly (*.dll* file). When you build your application, an assembly file (*.dll*) is created that contains the code that is executed when you run your application.

A `RoleStore` object, which provides role management, is created based on the database context.

**Note**

Notice that when the `RoleStore` object is created it uses a Generic `IdentityRole` type. This means that the `RoleStore` is only allowed to contain `IdentityRole` objects. Also by using Generics, resources in memory are handled better.

Next, the `RoleManager` object, is created based on the `RoleStore` object that you just created. the `RoleManager` object exposes role related API which can be used to automatically save changes to the `RoleStore`. The `RoleManager` is only allowed to contain `IdentityRole` objects because the code uses the `<IdentityRole>` Generic type.

You call the `RoleExists` method to determine if the "Administrator" role is present in the membership database. If it is not, you create the role.

Creating the `UserManager` object appears to be more complicated than the `RoleManager` control, however it is nearly the same. It is just coded on one line rather than several. Here, the parameter that you are passing is instantiating as a new object contained in the parenthesis.

Next you create the "Admin" user by creating a new `ApplicationUser` object. Then, if you successfully create the user, you add the user to the new role.

**Note**

The error handling will be updated during the "ASP.NET Error Handling" tutorial later in this tutorial series.

The next time the application starts, the user named "Admin" will be added as the role named "Administrator" of the application. Later in this tutorial, you will login as the "Admin" user to display additional capabilities that you will added during this tutorial. For API details about ASP.NET Identity, see the Microsoft.AspNet.Identity Namespace. For additional details about initializing the ASP.NET Identity system, see the AspnetIdentitySample.

## Restricting Access to the Administration Page

The Wingtip Toys sample application allows both anonymous users and logged-in users to view and purchase products. However, the logged-in administrator can access a restricted page in order to add and remove products.

### Add an Administration Folder and Page

Next, you will create a folder named *Admin* for the administrator of the Wingtip Toys sample application.

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

2. Name the new folder *Admin*.
3. Right-click the *Admin* folder and then select **Add** -> **New Item**.
   The **Add New Item** dialog box is displayed.
4. Select the **Visual C#** -> **Web** templates group on the left. From the middle list, select **Web Form with Master Page**, name it *AdminPage.aspx***,** and then select **Add**.
5. Select the *Site.Master* file as the master page, and then choose **OK**.

### Add a Web.config File

By adding a *Web.config* file to the *Admin* folder, you can restrict access to the page contained in the folder.

1. Right-click the *Admin* folder and select **Add** -> **New Item**.
   The **Add New Item** dialog box is displayed.
2. From the list of Visual C# web templates, select **Web Configuration File** from the middle list, accept the default name of *Web.config***,** and then select **Add**.
3. Replace the existing XML content in the *Web.config* file with the following:

```
<?xml version="1.0"?>
<configuration>
  <system.web>
    <authorization>
      <allow roles="Administrator"/>
      <deny users="*"/>
    </authorization>
  </system.web>
</configuration>
```

Save the *Web.config* file. The *Web.config* file specifies that only administrators of the application can access the page contained in the *Admin* folder.

## Including Administrator Navigation

To enable the administrator to navigate to the administration section of the application, you must add a link to the *Site.Master* page. Only users that belong to the administrator role will be able to see the **Admin** link and access the administration section.

1. In Solution Explorer, find and open the *Site.Master* page.
2. To create a link for administrators, add the markup highlighted in yellow to the following unordered list `<ul>` element so that the list appears as follows:

```
<ul class="nav navbar-nav">
    <li><a runat="server" id="adminLink" visible="false"
      href="~/Admin/AdminPage">Admin</a></li>
    <li><a runat="server" href="~/">Home</a></li>
    <li><a runat="server" href="~/About">About</a></li>
    <li><a runat="server" href="~/Contact">Contact</a></li>
    <li><a runat="server" href="~/ProductList">Products</a></li>
    <li><a runat="server" href="~/ShoppingCart"
      ID="cartCount">&nbsp;</a></li>
</ul>
```

3. Open the *Site.Master.cs* file. Make the **Admin** link visible only to the "Admin" user by adding the code highlighted in yellow to the `Page_Load` handler. The `Page_Load` handler will appear as follows:

```
protected void Page Load(object sender, EventArgs e)
{
  if (HttpContext.Current.User.IsInRole("Administrator"))
  {
    adminLink.Visible = true;
  }
}
```

When the page loads, the code checks whether the logged-in user has the role of "Administrator". If the user is an administrator, the span element containing the link to the *AdminPage.aspx* page (and consequently the link inside the span) is made visible.

## Enabling Product Administration

So far, you have created the administrator role and added an administrator user, an administration folder, and an administration page. You have set access rights for the administration folder and page, and have added a navigation link for the administrator to the application. Next, you will add markup to the *AdminPage.aspx* page and code to the *AdminPage.aspx.cs* code-behind file that will enable the administrator to add and remove products.

1. In **Solution Explorer**, open the *AdminPage.aspx* file from the *Admin* folder.
2. Replace the existing markup with the following:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Site.Master"
AutoEventWireup="true" CodeBehind="AdminPage.aspx.cs"
Inherits="WingtipToys.Admin.AdminPage" %>
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent" runat="server">
    <h1>Administration</h1>
    <hr />
    <h3>Add Product:</h3>
    <table>
        <tr>
            <td><asp:Label ID="LabelAddCategory"
runat="server">Category:</asp:Label></td>
            <td>
                <asp:DropDownList ID="DropDownAddCategory" runat="server"
                    ItemType="WingtipToys.Models.Category"
                    SelectMethod="GetCategories" DataTextField="CategoryName"
                    DataValueField="CategoryID" >
                </asp:DropDownList>
            </td>
        </tr>
        <tr>
            <td><asp:Label ID="LabelAddName"
runat="server">Name:</asp:Label></td>
            <td>
                <asp:TextBox ID="AddProductName" runat="server"></asp:TextBox>
                <asp:RequiredFieldValidator ID="RequiredFieldValidator1"
runat="server" Text="* Product name required."
ControlToValidate="AddProductName" SetFocusOnError="true"
Display="Dynamic"></asp:RequiredFieldValidator>
```

```
            </td>
        </tr>
        <tr>
            <td><asp:Label ID="LabelAddDescription" 
runat="server">Description:</asp:Label></td>
            <td>
                <asp:TextBox ID="AddProductDescription" 
runat="server"></asp:TextBox>
                <asp:RequiredFieldValidator ID="RequiredFieldValidator2" 
runat="server" Text="* Description required." 
ControlToValidate="AddProductDescription" SetFocusOnError="true" 
Display="Dynamic"></asp:RequiredFieldValidator>
            </td>
        </tr>
        <tr>
            <td><asp:Label ID="LabelAddPrice" 
runat="server">Price:</asp:Label></td>
            <td>
                <asp:TextBox ID="AddProductPrice" runat="server"></asp:TextBox>
                <asp:RequiredFieldValidator ID="RequiredFieldValidator3" 
runat="server" Text="* Price required." ControlToValidate="AddProductPrice" 
SetFocusOnError="true" Display="Dynamic"></asp:RequiredFieldValidator>
                <asp:RegularExpressionValidator 
ID="RegularExpressionValidator1" runat="server" Text="* Must be a valid price 
without $." ControlToValidate="AddProductPrice" SetFocusOnError="True" 
Display="Dynamic" ValidationExpression="^[0-9]*(\.)?[0-9]?[0-
9]?$"></asp:RegularExpressionValidator>
            </td>
        </tr>
        <tr>
            <td><asp:Label ID="LabelAddImageFile" runat="server">Image 
File:</asp:Label></td>
            <td>
                <asp:FileUpload ID="ProductImage" runat="server" />
                <asp:RequiredFieldValidator ID="RequiredFieldValidator4" 
runat="server" Text="* Image path required." ControlToValidate="ProductImage" 
SetFocusOnError="true" Display="Dynamic"></asp:RequiredFieldValidator>
            </td>
        </tr>
    </table>
    <p></p>
    <p></p>
    <asp:Button ID="AddProductButton" runat="server" Text="Add Product" 
OnClick="AddProductButton_Click"  CausesValidation="true"/>
    <asp:Label ID="LabelAddStatus" runat="server" Text=""></asp:Label>
    <p></p>
    <h3>Remove Product:</h3>
    <table>
        <tr>
            <td><asp:Label ID="LabelRemoveProduct" 
runat="server">Product:</asp:Label></td>
            <td><asp:DropDownList ID="DropDownRemoveProduct" runat="server" 
ItemType="WingtipToys.Models.Product" 
                    SelectMethod="GetProducts" AppendDataBoundItems="true" 
                    DataTextField="ProductName" DataValueField="ProductID" >
                </asp:DropDownList>
            </td>
        </tr>
    </table>
    <p></p>
    <asp:Button ID="RemoveProductButton" runat="server" Text="Remove Product" 
OnClick="RemoveProductButton_Click" CausesValidation="false"/>
    <asp:Label ID="LabelRemoveStatus" runat="server" Text=""></asp:Label>
```

```
</asp:Content>
```

3. Next, open the *AdminPage.aspx.cs* code-behind file by right-clicking the *AdminPage.aspx* and clicking **View Code**.
4. Replace the existing code in the *AdminPage.aspx.cs* code-behind 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 WingtipToys.Logic;

namespace WingtipToys.Admin
{
  public partial class AdminPage : System.Web.UI.Page
  {
    protected void Page Load(object sender, EventArgs e)
    {
      string productAction = Request.QueryString["ProductAction"];
      if (productAction == "add")
      {
        LabelAddStatus.Text = "Product added!";
      }

      if (productAction == "remove")
      {
        LabelRemoveStatus.Text = "Product removed!";
      }
    }

    protected void AddProductButton_Click(object sender, EventArgs e)
    {
      Boolean fileOK = false;
      String path = Server.MapPath("~/Catalog/Images/");
      if (ProductImage.HasFile)
      {
        String fileExtension =
System.IO.Path.GetExtension(ProductImage.FileName).ToLower();
        String[] allowedExtensions = { ".gif", ".png", ".jpeg", ".jpg" };
        for (int i = 0; i < allowedExtensions.Length; i++)
        {
          if (fileExtension == allowedExtensions[i])
          {
            fileOK = true;
          }
        }
      }

      if (fileOK)
      {
        try
        {
          // Save to Images folder.
          ProductImage.PostedFile.SaveAs(path + ProductImage.FileName);
          // Save to Images/Thumbs folder.
          ProductImage.PostedFile.SaveAs(path + "Thumbs/" + 
ProductImage.FileName);
        }
        catch (Exception ex)
        {
          LabelAddStatus.Text = ex.Message;
        }

        // Add product data to DB.
        AddProducts products = new AddProducts();
        bool addSuccess = products.AddProduct(AddProductName.Text, 
AddProductDescription.Text,
            AddProductPrice.Text, DropDownAddCategory.SelectedValue, 
ProductImage.FileName);
        if (addSuccess)
        {
          // Reload the page.
          string pageUrl = Request.Url.AbsoluteUri.Substring(0, 
Request.Url.AbsoluteUri.Count() - Request.Url.Query.Count());
          Response.Redirect(pageUrl + "?ProductAction=add");
        }
        else
        {
          LabelAddStatus.Text = "Unable to add new product to database.";
        }
      }
      else
      {
        LabelAddStatus.Text = "Unable to accept file type.";
      }
    }

    public IQueryable GetCategories()
    {
      var _db = new WingtipToys.Models.ProductContext();
      IQueryable query = _db.Categories;
      return query;
    }

    public IQueryable GetProducts()
    {
      var _db = new WingtipToys.Models.ProductContext();
      IQueryable query = _db.Products;
      return query;
    }

    protected void RemoveProductButton_Click(object sender, EventArgs e)
    {
      using (var _db = new WingtipToys.Models.ProductContext())
      {
        int productId = Convert.ToInt16(DropDownRemoveProduct.SelectedValue);
        var myItem = (from c in _db.Products where c.ProductID == productId 
select c).FirstOrDefault();
        if (myItem != null)
        {
          _db.Products.Remove(myItem);
          _db.SaveChanges();

          // Reload the page.
          string pageUrl = Request.Url.AbsoluteUri.Substring(0, 
Request.Url.AbsoluteUri.Count() - Request.Url.Query.Count());
          Response.Redirect(pageUrl + "?ProductAction=remove");
        }
        else
        {
          LabelRemoveStatus.Text = "Unable to locate product.";
        }
      }
    }
  }
}
```

In the code that you entered for the *AdminPage.aspx.cs* code-behind file, a class called `AddProducts` does the actual work of adding products to the database. This class doesn't exist yet, so you will create it now.

1. In **Solution Explorer**, right-click the *Logic* folder and then select **Add** -> **New Item**. The **Add New Item** dialog box is displayed.
2. Select the **Visual C#** -> **Code** templates group on the left. Then, select **Class** from the middle list and name it *AddProducts.cs*.
   The new class file is displayed.
3. Replace the existing code with the following:

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

namespace WingtipToys.Logic
{
  public class AddProducts
  {
    public bool AddProduct(string ProductName, string ProductDesc, string
ProductPrice, string ProductCategory, string ProductImagePath)
    {
      var myProduct = new Product();
      myProduct.ProductName = ProductName;
      myProduct.Description = ProductDesc;
      myProduct.UnitPrice = Convert.ToDouble(ProductPrice);
      myProduct.ImagePath = ProductImagePath;
      myProduct.CategoryID = Convert.ToInt32(ProductCategory);

      using (ProductContext _db = new ProductContext())
      {
        // Add product to DB.
        _db.Products.Add(myProduct);
        _db.SaveChanges();
      }
      // Success.
      return true;
    }
  }
}
```

The *AdminPage.aspx* page allows the administrator to add and remove products. When a new product is added, the details about the product are validated and then entered into the database. The new product is immediately available to all users of the web application.

## Unobtrusive Validation

The product details that the user provides on the *AdminPage.aspx* page are validated using validation controls (`RequiredFieldValidator` and `RegularExpressionValidator`). These controls automatically use unobtrusive validation. Unobtrusive validation allows the validation controls to use JavaScript for client-side validation logic, which means the page does not require a trip to the server to be validated. By default, unobtrusive validation is included in the *Web.config* file based on the following configuration setting:

```
<add key="ValidationSettings:UnobtrusiveValidationMode" value="WebForms" />
```

### Regular Expressions

The product price on the *AdminPage.aspx* page is validated using a **RegularExpressionValidator** control. This control validates whether the value of the associated input control (the "AddProductPrice" TextBox) matches the pattern specified by the regular expression. A regular expression is a pattern-matching notation that enables you to quickly find and match specific character patterns. The **RegularExpressionValidator** control includes a property named `ValidationExpression` that contains the regular expression used to validate price input, as shown below:

```
<asp:RegularExpressionValidator
    ID="RegularExpressionValidator1" runat="server"
    Text="* Must be a valid price without $."
ControlToValidate="AddProductPrice"
    SetFocusOnError="True" Display="Dynamic"
    ValidationExpression="^[0-9]*(\.)?[0-9]?[0-9]?$">
</asp:RegularExpressionValidator>
```

### FileUpload Control

In addition to the input and validation controls, you added the **FileUpload** control to the *AdminPage.aspx* page. This control provides the capability to upload files. In this case, you are only allowing image files to be uploaded. In the code-behind file (*AdminPage.aspx.cs*), when the `AddProductButton` is clicked, the code checks the `HasFile` property of the **FileUpload** control. If the control has a file and if the file type (based on file extension) is allowed, the image is saved to the *Images* folder and the *Images/Thumbs* folder of the application.

### Model Binding

Earlier in this tutorial series you used model binding to populate a **ListView** control, a **FormsView** control, a **GridView** control, and a **DetailView** control. In this tutorial, you use model binding to populate a **DropDownList** control with a list of product categories.

The markup that you added to the *AdminPage.aspx* file contains a **DropDownList** control called `DropDownAddCategory`:

```
    <asp:DropDownList ID="DropDownAddCategory" runat="server"
        ItemType="WingtipToys.Models.Category"
        SelectMethod="GetCategories" DataTextField="CategoryName"
        DataValueField="CategoryID" >
    </asp:DropDownList>
```

You use model binding to populate this **DropDownList** by setting the `ItemType` attribute and the `SelectMethod` attribute. The `ItemType` attribute specifies that you use the

WingtipToys.Models.Category type when populating the control. You defined this type at the beginning of this tutorial series by creating the Category class (shown below). The Category class is in the *Models* folder inside the *Category.cs* file.

```
    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 SelectMethod attribute of the **DropDownList** control specifies that you use the GetCategories method (shown below) that is included in the code-behind file (*AdminPage.aspx.cs*).

```
    public IQueryable GetCategories()
    {
      var _db = new WingtipToys.Models.ProductContext();
      IQueryable query = _db.Categories;
      return query;
    }
```

This method specifies that an IQueryable interface is used to evaluate a query against a Category type. The returned value is used to populate the **DropDownList** in the markup of the page (*AdminPage.aspx*).

The text displayed for each item in the list is specified by setting the DataTextField attribute. The DataTextField attribute uses the CategoryName of the Category class (shown above) to display each category in the **DropDownList** control. The actual value that is passed when an item is selected in the **DropDownList** control is based on the DataValueField attribute. The DataValueField attribute is set to the CategoryID as define in the Category class (shown above).

## How the Application Will Work

When the administrator navigates to the page for the first time, the DropDownAddCategory **DropDownList** control is populated as described above. The DropDownRemoveProduct **DropDownList** control is also populated with products using the same approach. The administrator selects the category type and adds product details (**Name**, **Description**, **Price**, and **Image File**). When the administrator clicks the **Add Product** button, the AddProductButton_Click event handler is triggered. The AddProductButton_Click event handler located in the code-behind file (*AdminPage.aspx.cs*) checks the image file to make sure it matches the allowed file types *(.gif*, *.png*, *.jpeg*, or *.jpg*). Then, the image file is saved into a folder of the Wingtip Toys sample application. Next, the new product is added to the database.

To accomplish adding a new product, a new instance of the `AddProducts` class is created and named products. The `AddProducts` class has a method named `AddProduct`, and the products object calls this method to add products to the database.

```
    // Add product data to DB.
    AddProducts products = new AddProducts();
    bool addSuccess = products.AddProduct(AddProductName.Text,
AddProductDescription.Text,
        AddProductPrice.Text, DropDownAddCategory.SelectedValue,
ProductImage.FileName);
```

If the code successfully adds the new product to the database, the page is reloaded with the query string value `ProductAction=add`.

```
    Response.Redirect(pageUrl + "?ProductAction=add");
```

When the page reloads, the query string is included in the URL. By reloading the page, the administrator can immediately see the updates in the **DropDownList** controls on the *AdminPage.aspx* page. Also, by including the query string with the URL, the page can display a success message to the administrator.

When the *AdminPage.aspx* page reloads, the `Page_Load` event is called.

```
    protected void Page_Load(object sender, EventArgs e)
    {
        string productAction = Request.QueryString["ProductAction"];
        if (productAction == "add")
        {
            LabelAddStatus.Text = "Product added!";
        }

        if (productAction == "remove")
        {
            LabelRemoveStatus.Text = "Product removed!";
        }
    }
```

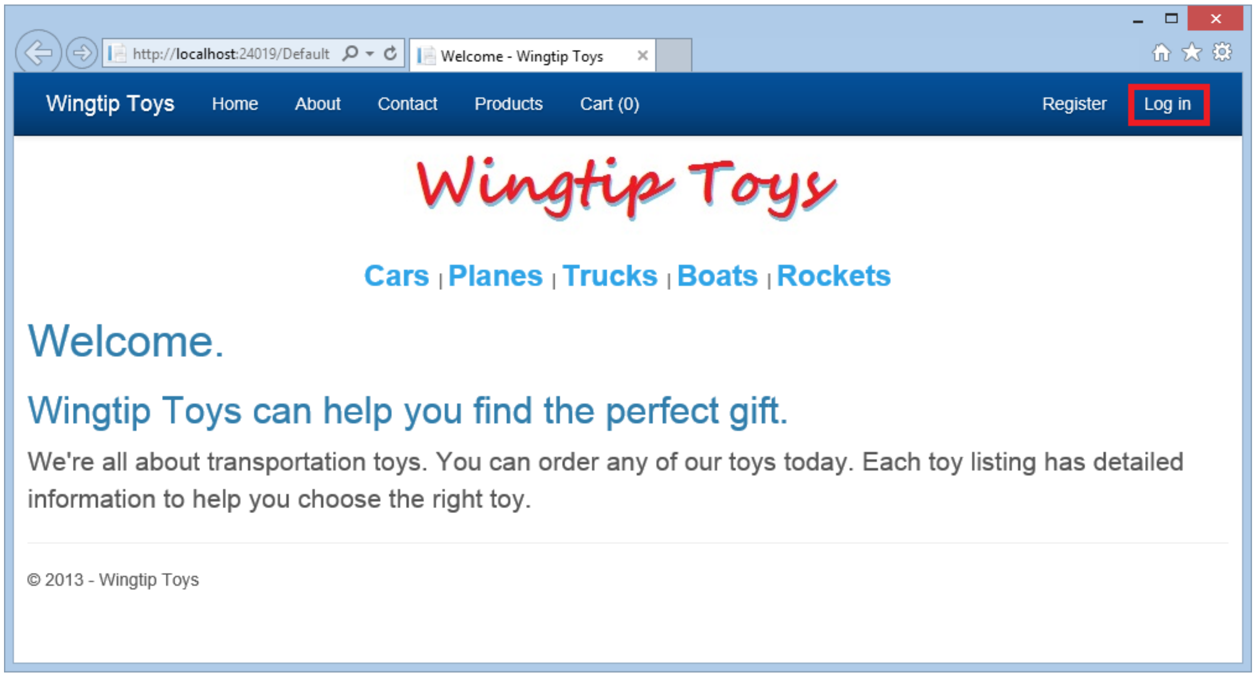
The `Page_Load` event handler checks the query string value and determines whether to show a success message.

## Running the Application

You can run the application now to see how you can add, delete, and update items in the shopping cart. The shopping cart total will reflect the total cost of all items in the shopping cart.

1. In Solution Explorer, press **F5** to run the Wingtip Toys sample application.
   The browser opens and shows the *Default.aspx* page.

2. Click the **Log in** link at the top of the page.



Welcome.

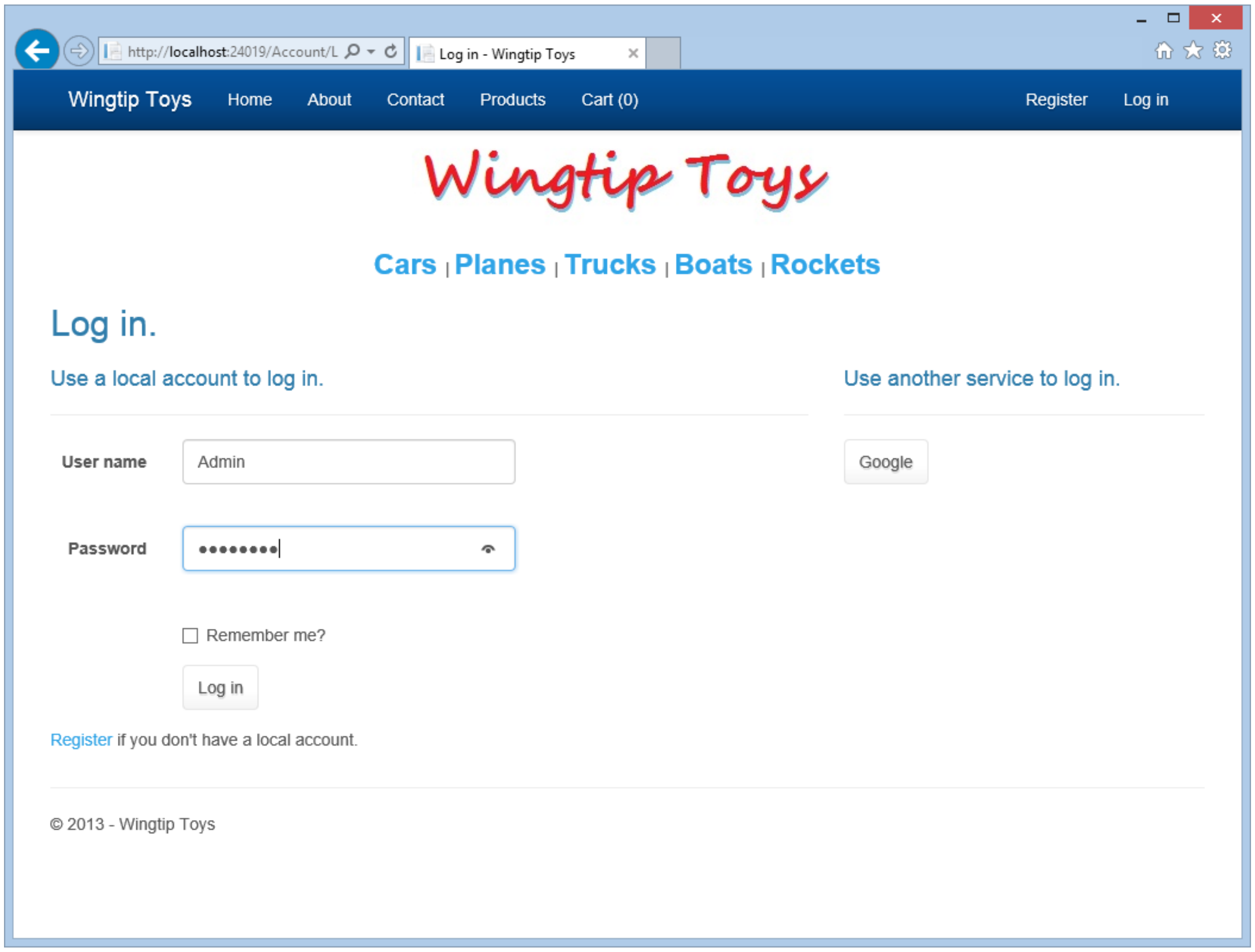
Wingtip Toys can help you find the perfect gift.

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.

© 2013 - Wingtip Toys

The *Login.aspx* page is displayed.

3. Use the following administrator user name and password:
User name: `Admin`
Password: `Pa$$word`



Log in.

Use a local account to log in.

User name: Admin

Password: ••••••••

☐ Remember me?

Log in

Register if you don't have a local account.

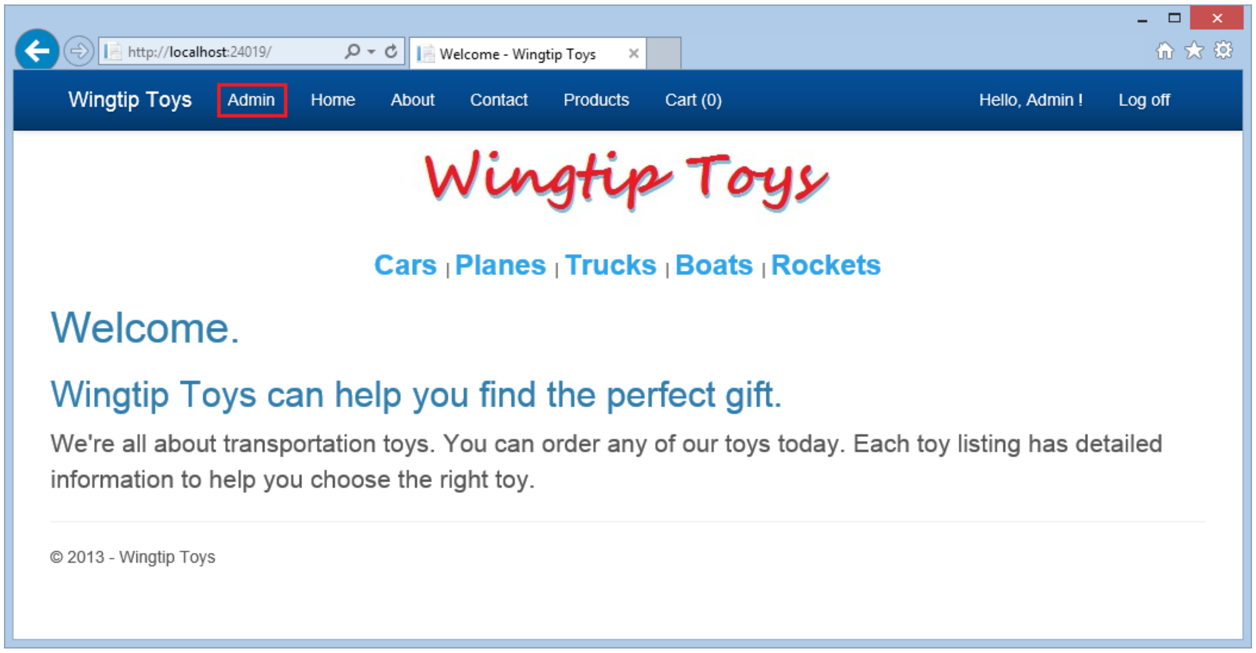
Use another service to log in.

Google

© 2013 - Wingtip Toys

4. Click the **Log in** button near the bottom of the page.

5. At the top of the next page, select the **Admin** link to navigate to the *AdminPage.aspx* page.



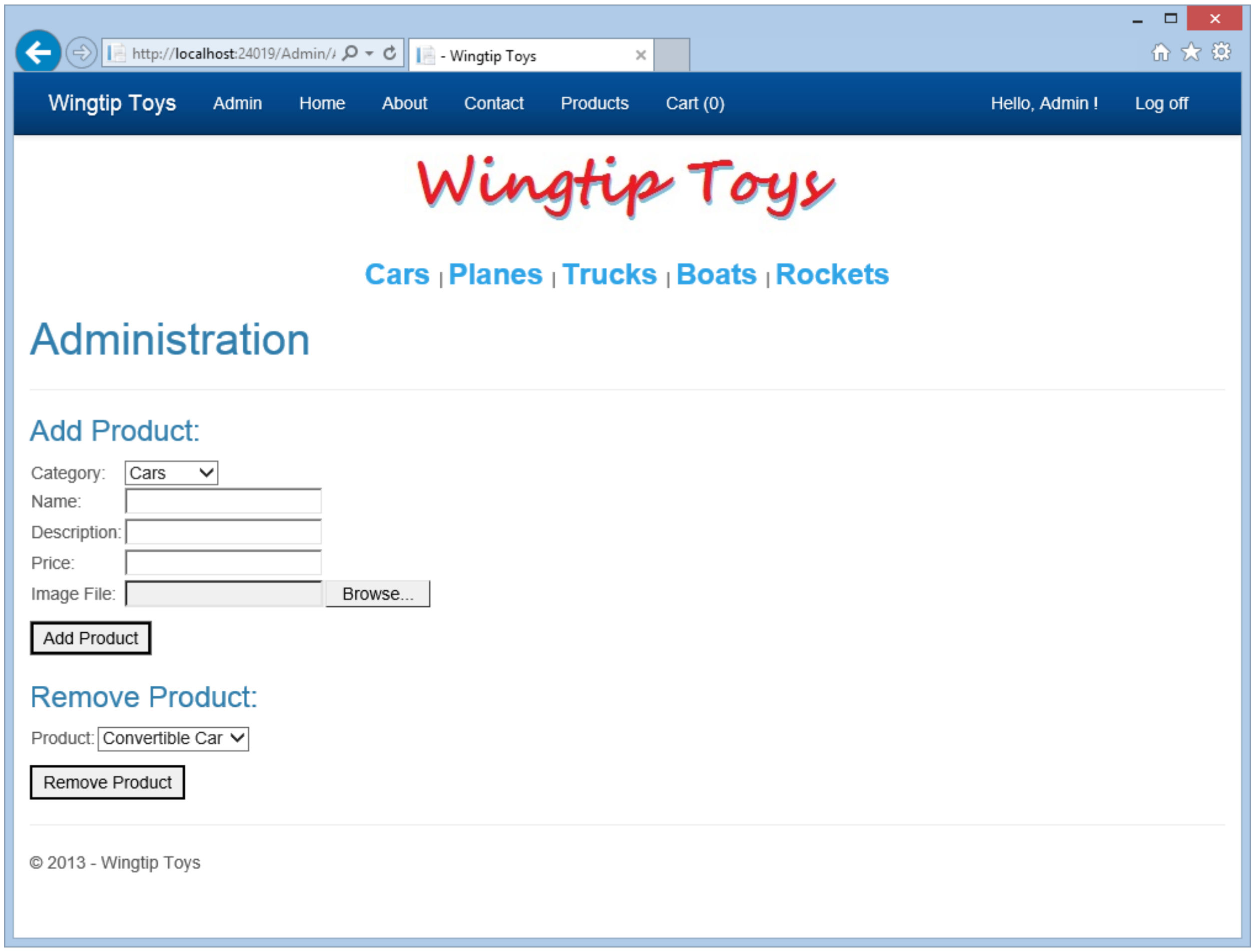
Welcome.

Wingtip Toys can help you find the perfect gift.

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.

© 2013 - Wingtip Toys

6. To test the input validation, click the **Add Product** button without adding any product details.



Administration

Add Product:

Category: Cars

Name:

Description:

Price:

Image File: Browse...

Add Product

Remove Product:

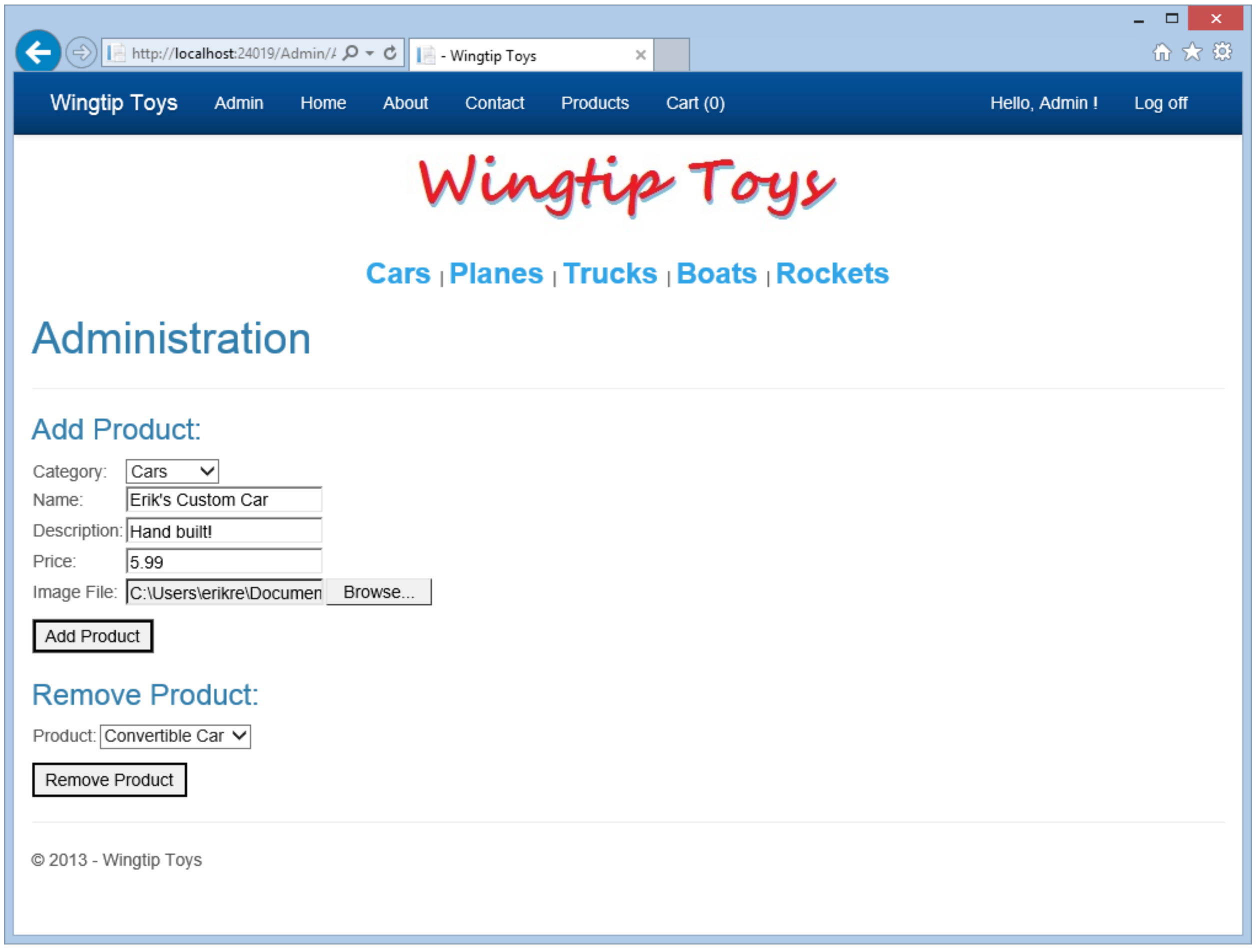
Product: Convertible Car

Remove Product

© 2013 - Wingtip Toys

Notice that the required field messages are displayed.

7. Add the details for a new product, and then click the **Add Product** button.



Administration

Add Product:

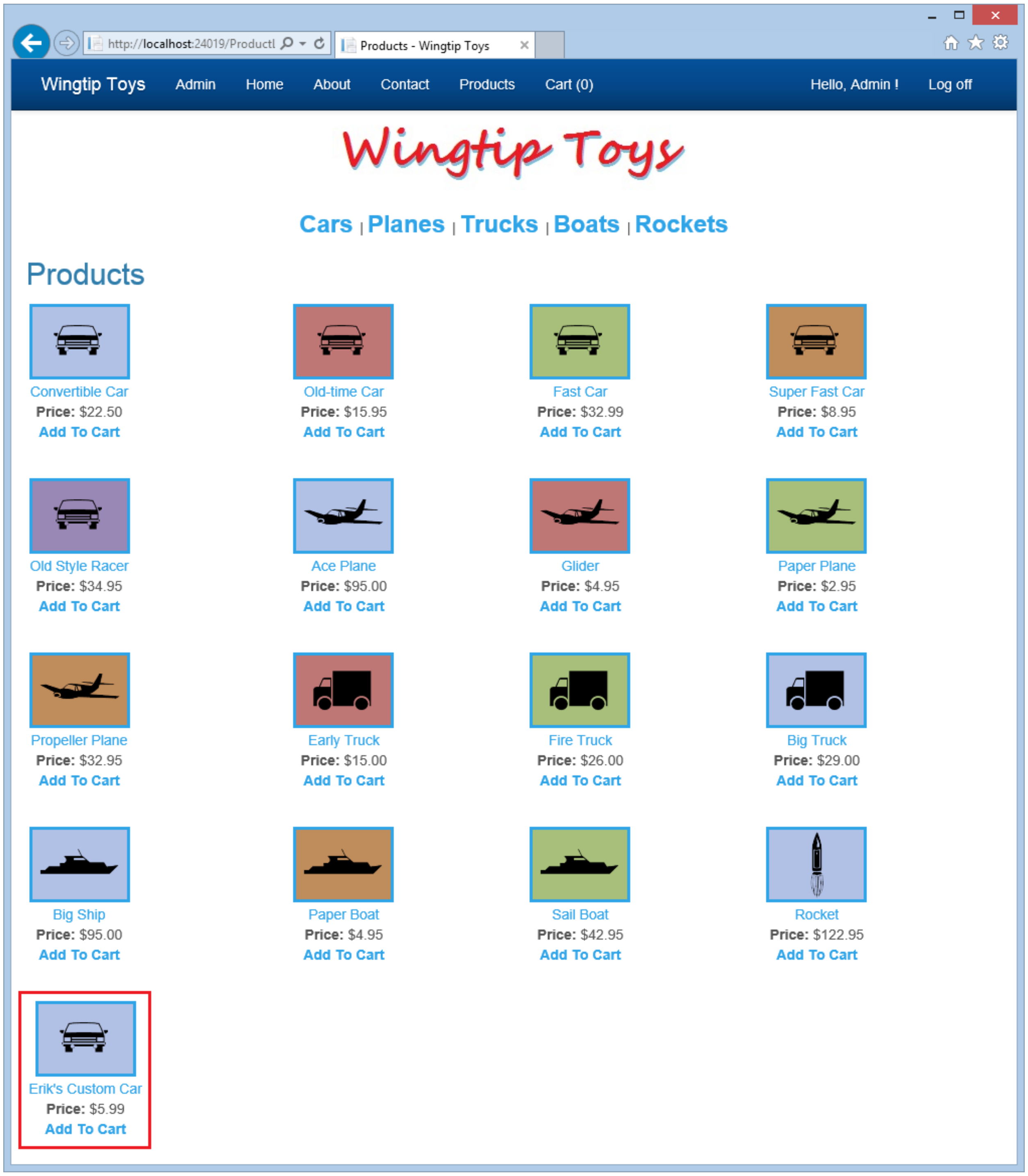
Category: Cars
Name: Erik's Custom Car
Description: Hand built!
Price: 5.99
Image File: C:\Users\erikre\Documen

Remove Product:

Product: Convertible Car

© 2013 - Wingtip Toys

8. Select **Products** from the top navigation menu to view the new product you added.

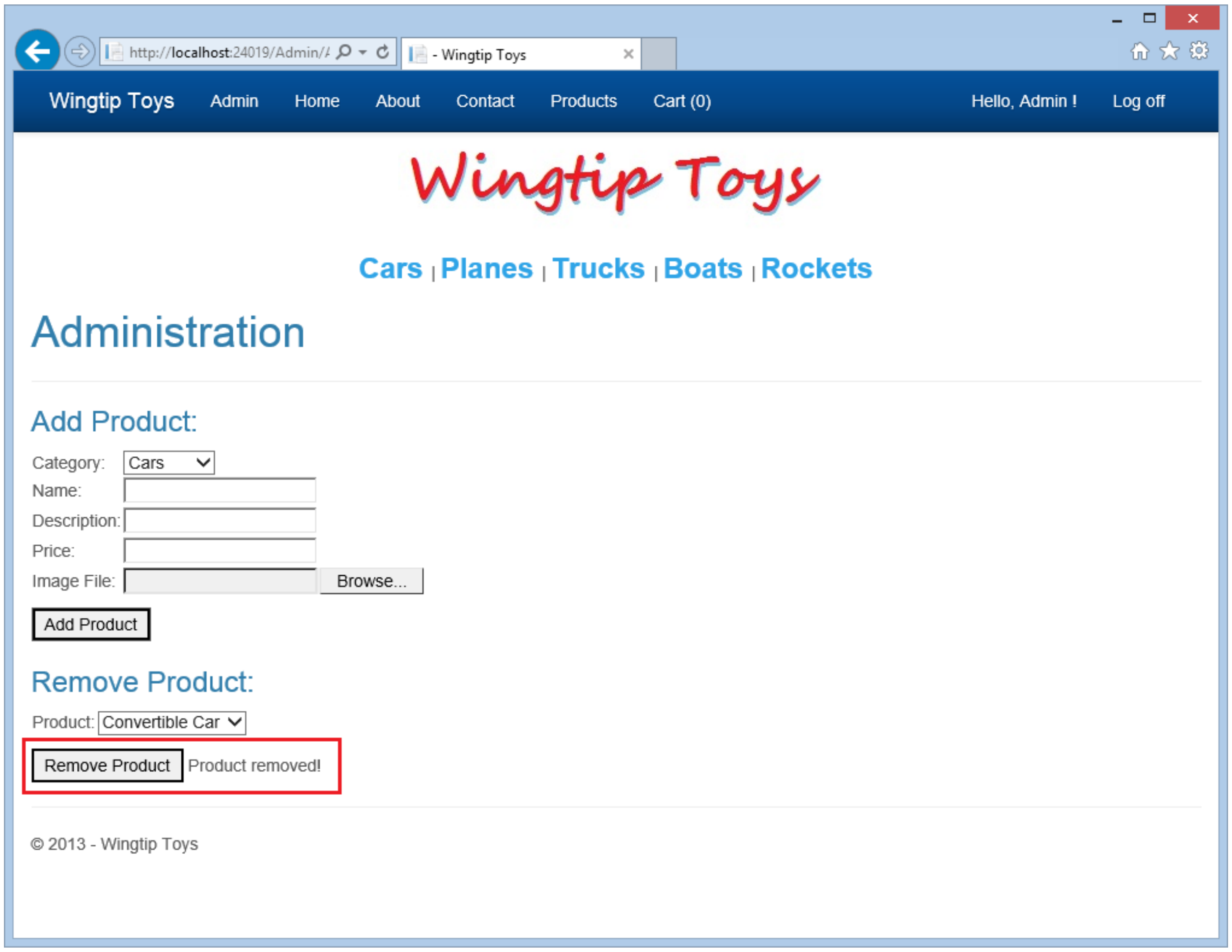


Products

| | | | |
|---|---|---|---|
| Convertible Car<br>**Price:** $22.50<br>Add To Cart | Old-time Car<br>**Price:** $15.95<br>Add To Cart | Fast Car<br>**Price:** $32.99<br>Add To Cart | Super Fast Car<br>**Price:** $8.95<br>Add To Cart |
| Old Style Racer<br>**Price:** $34.95<br>Add To Cart | Ace Plane<br>**Price:** $95.00<br>Add To Cart | Glider<br>**Price:** $4.95<br>Add To Cart | Paper Plane<br>**Price:** $2.95<br>Add To Cart |
| Propeller Plane<br>**Price:** $32.95<br>Add To Cart | Early Truck<br>**Price:** $15.00<br>Add To Cart | Fire Truck<br>**Price:** $26.00<br>Add To Cart | Big Truck<br>**Price:** $29.00<br>Add To Cart |
| Big Ship<br>**Price:** $95.00<br>Add To Cart | Paper Boat<br>**Price:** $4.95<br>Add To Cart | Sail Boat<br>**Price:** $42.95<br>Add To Cart | Rocket<br>**Price:** $122.95<br>Add To Cart |
| Erik's Custom Car<br>**Price:** $5.99<br>Add To Cart | | | |

9. Click the **Admin** link to return to the administration page.
10. In the **Remove Product** section of the page, select the new product you added in the **DropDownListBox**.

11. Click the **Remove Product** button to remove the new product from the application.



Administration

Add Product:

Category: Cars
Name:
Description:
Price:
Image File: Browse...
Add Product

Remove Product:

Product: Convertible Car
Remove Product Product removed!

© 2013 - Wingtip Toys

12. Select **Products** from the top navigation menu to confirm that the product has been removed.
13. Click **Log off** to exist administration mode.
    Notice that the top navigation pane no longer shows the **Admin** menu item.

## Summary

In this tutorial, you added an administrator role and an administrative user, restricted access to the administration folder and page, and provided navigation for the administrator role. You used model binding to populate a **DropDownList** control with data. You implemented the **FileUpload** control and validation controls. Also, you have learned how to add and remove products from a database. In the next tutorial, you'll learn how to implement ASP.NET routing.

## Additional Resources

Web.config - authorization Element
ASP.NET Identity

## URL Routing

This tutorial series will teach you the basics of building an ASP.NET Web Forms application using ASP.NET 4.5 and Microsoft Visual Studio Express 2013 for Web. A Visual Studio 2013 project with C# source code is available to accompany this tutorial series.

In this tutorial, you will modify the Wingtip Toys sample application to customize URL routing. Routing enables your web application to use URLs that are friendly, easier to remember, and better supported by search engines. This tutorial builds on the previous tutorial " Membership and Administration" and is part of the Wingtip Toys tutorial series.

## What you'll learn:

- How to register routes for an ASP.NET Web Forms application.
- How to add routes to a web page.
- How to select data from a database to support routes.

## ASP.NET Routing Overview

URL routing allows you to configure an application to accept request URLs that do not map to physical files. A request URL is simply the URL a user enters into their browser to find a page on your web site. You use routing to define URLs that are semantically meaningful to users and that can help with search-engine optimization (SEO).

By default, the Web Forms template includes ASP.NET Friendly URLs. Much of the basic routing work will be implemented by using *Friendly URLs*. However, in this tutorial you will add customized routing capabilities.

Before customizing URL routing, the Wingtip Toys sample application can link to a product using the following URL:

**http://localhost:1234/ProductDetails.aspx?productID=2**

By customizing URL routing, the Wingtip Toys sample application will link to the same product using an easier to read URL:

**http://localhost:1234/Product/Convertible%20Car**

### Routes

A route is a URL pattern that is mapped to a handler. The handler can be a physical file, such as an .aspx file in a Web Forms application. A handler can also be a class that processes the request. To define a route, you create an instance of the Route class by specifying the URL pattern, the handler, and optionally a name for the route.

You add the route to the application by adding the `Route` object to the static `Routes` property of the `RouteTable` class. The Routes property is a `RouteCollection` object that stores all the routes for the application.

## URL Patterns

A URL pattern can contain literal values and variable placeholders (referred to as URL parameters). The literals and placeholders are located in segments of the URL which are delimited by the slash (`/`) character.

When a request to your web application is made, the URL is parsed into segments and placeholders, and the variable values are provided to the request handler. This process is similar to the way the data in a query string is parsed and passed to the request handler. In both cases, variable information is included in the URL and passed to the handler in the form of key-value pairs. For query strings, both the keys and the values are in the URL. For routes, the keys are the placeholder names defined in the URL pattern, and only the values are in the URL.

In a URL pattern, you define placeholders by enclosing them in braces ( `{` and `}` ). You can define more than one placeholder in a segment, but the placeholders must be separated by a literal value. For example, `{language}-{country}/{action}` is a valid route pattern. However, `{language}{country}/{action}` is not a valid pattern, because there is no literal value or delimiter between the placeholders. Therefore, routing cannot determine where to separate the value for the language placeholder from the value for the country placeholder.

## Mapping and Registering Routes

Before you can include routes to pages of the Wingtip Toys sample application, you must register the routes when the application starts. To register the routes, you will modify the `Application_Start` event handler.

1. In **Solution Explorer** of Visual Studio, find and open the *Global.asax.cs* file.
2. Add the code highlighted in yellow to the *Global.asax.cs* file as follows:

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

namespace WingtipToys
{
    public class Global : HttpApplication
    {
        void Application_Start(object sender, EventArgs e)
        {
          // Code that runs on application startup
          RouteConfig.RegisterRoutes(RouteTable.Routes);
          BundleConfig.RegisterBundles(BundleTable.Bundles);

          // Initialize the product database.
          Database.SetInitializer(new ProductDatabaseInitializer());

          // Create administrator role and user.
          RoleActions roleActions = new RoleActions();
          roleActions.createAdmin();

          // Add Routes.
          RegisterCustomRoutes(RouteTable.Routes);
        }

        void RegisterCustomRoutes(RouteCollection routes)
        {
          routes.MapPageRoute(
              "ProductsByCategoryRoute",
              "Category/{categoryName}",
              "~/ProductList.aspx"
          );
          routes.MapPageRoute(
              "ProductByNameRoute",
              "Product/{productName}",
              "~/ProductDetails.aspx"
          );
        }
    }
}
```

When the Wingtip Toys sample application starts, it calls the `Application_Start` event handler. At the end of this event handler, the `RegisterCustomRoutes` method is called. The `RegisterCustomRoutes` method adds each route by calling the `MapPageRoute` method of the `RouteCollection` object. Routes are defined using a route name, a route URL and a physical URL.

The first parameter ("`ProductsByCategoryRoute`") is the route name. It is used to call the route when it is needed. The second parameter ("`Category/{categoryName}`") defines the friendly replacement URL that can be dynamic based on code. You use this route when you are populating a data control with links that are generated based on data. A route is shown as follows:

```
          routes.MapPageRoute(
              "ProductsByCategoryRoute",
              "Category/{categoryName}",
              "~/ProductList.aspx"
          );
```

The second parameter of the route includes a dynamic value specified by braces (`{ }`). In this case, the `categoryName` is a variable that will be used to determine the proper routing path.

**Optional**

You might find it easier to manage your code by moving the `RegisterCustomRoutes` method to a separate class. In the *Logic* folder, create a separate `RouteActions` class. Move the above `RegisterCustomRoutes` method from the *Global.asax.cs* file into the new `RoutesActions` class. Use the `RoleActions` class and the `createAdmin` method as an example of how to call the `RegisterCustomRoutes` method from the *Global.asax.cs* file.

You may also have noticed the `RegisterRoutes` method call using the `RouteConfig` object at the beginning of the `Application_Start` event handler. This call is made to implement default routing. It was included as default code when you created the application using Visual Studio's Web Forms template.

## Retrieving and Using Route Data

As mentioned above, routes can be defined. The code that you added to the `Application_Start` event handler in the *Global.asax.cs* file loads the definable routes.

### Setting Routes

Routes require you to add additional code. In this tutorial, you will use model binding to retrieve a `RouteValueDictionary` object that is used when generating the routes using data from a data control. The `RouteValueDictionary` object will contain a list of product names that belong to a specific category of products. A link is created for each product based on the data and route.

#### Enable Routes for Categories and Products

Next, you'll update the application to use the `ProductsByCategoryRoute` to determine the correct route to include for each product category link. You'll also update the *ProductList.aspx* page to include a routed link for each product. The links will be displayed as they were before the change, however the links will now use URL routing.

1. In **Solution Explorer**, open the *Site.Master* page if it is not already open.
2. Update the **ListView** control named "`categoryList`" with the changes highlighted in yellow, so the markup appears as follows:

```
<asp:ListView ID="categoryList"
    ItemType="WingtipToys.Models.Category"
    runat="server"
    SelectMethod="GetCategories" >
    <ItemTemplate>
        <b style="font-size: large; font-style: normal">
        <a href="<%#: GetRouteUrl("ProductsByCategoryRoute", new {categoryName
= Item.CategoryName}) %>">
            <%#: Item.CategoryName %>
        </a>
        </b>
    </ItemTemplate>
    <ItemSeparatorTemplate>  |  </ItemSeparatorTemplate>
</asp:ListView>
```

3. In **Solution Explorer**, open the *ProductList.aspx* page.

4. Update the `ItemTemplate` element of the *ProductList.aspx* page with the updates highlighted in yellow, so the markup appears as follows:

```
<ItemTemplate>
  <td runat="server">
    <table>
      <tr>
        <td>
          <a href="<%#: GetRouteUrl("ProductByNameRoute", new {productName = 
Item.ProductName}) %>">
            <image src='/Catalog/Images/Thumbs/<%#:Item.ImagePath%>'
              width="100" height="75" border="1" />
          </a>
        </td>
      </tr>
      <tr>
        <td>
          <a href="<%#: GetRouteUrl("ProductByNameRoute", new {productName = 
Item.ProductName}) %>">
            <%#:Item.ProductName%>
          </a>
          <br />
          <span>
            <b>Price: </b><%#:String.Format("{0:c}", Item.UnitPrice)%>
          </span>
          <br />
          <a href="/AddToCart.aspx?productID=<%#:Item.ProductID %>">
            <span class="ProductListItem">
              <b>Add To Cart<b>
            </span>
          </a>
        </td>
      </tr>
      <tr>
        <td>&nbsp;</td>
      </tr>
    </table>
    </p>
  </td>
</ItemTemplate>
```

5. Open the code-behind of *ProductList.aspx.cs* and add the following namespace as highlighted in yellow:

```
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;
using System.Web.Routing;
```

6. Replace the `GetProducts` method of the code-behind (*ProductList.aspx.cs*) with the following code:

```
public IQueryable<Product> GetProducts(
                    [QueryString("id")] int? categoryId,
                    [RouteData] string categoryName)
{
  var _db = new WingtipToys.Models.ProductContext();
  IQueryable<Product> query = _db.Products;

  if (categoryId.HasValue && categoryId > 0)
  {
    query = query.Where(p => p.CategoryID == categoryId);
  }

  if (!String.IsNullOrEmpty(categoryName))
  {
    query = query.Where(p =>
                        String.Compare(p.Category.CategoryName,
                        categoryName) == 0);
  }
  return query;
}
```

### Add Code for Product Details

Now, update the code-behind (*ProductDetails.aspx.cs*) for the *ProductDetails.aspx* page to use route data. Notice that the new `GetProduct` method also accepts a query string value for the case where the user has a link bookmarked that uses the older non-friendly, non-routed URL.

1. Replace the `GetProduct` method of the code-behind (*ProductDetails.aspx.cs*) with the following code:

```
public IQueryable<Product> GetProduct(
                    [QueryString("ProductID")] int? productId,
                    [RouteData] string productName)
{
  var _db = new WingtipToys.Models.ProductContext();
  IQueryable<Product> query = _db.Products;
  if (productId.HasValue && productId > 0)
  {
    query = query.Where(p => p.ProductID == productId);
  }
  else if (!String.IsNullOrEmpty(productName))
  {
    query = query.Where(p =>
              String.Compare(p.ProductName, productName) == 0);
  }
  else
  {
    query = null;
  }
  return query;
}
```

## Running the Application

You can run the application now to see the updated routes.

1. Press **F5** to run the Wingtip Toys sample application.
   The browser opens and shows the *Default.aspx* page.
2. Click the **Products** link at the top of the page.
   All products are displayed on the *ProductList.aspx* page. The following URL (using your port number) is displayed for the browser:
   **http://localhost:1234/ProductList**

3. Next, click the **Cars** category link near the top of the page.
   Only cars are displayed on the *ProductList.aspx* page. The following URL (using your port number) is displayed for the browser:
   **http://localhost:1234/Category/Cars**
4. Click the link containing the name of the first car listed on the page ("**Convertible Car**") to display the product details.
   The following URL (using your port number) is displayed for the browser:
   **http://localhost:1234/Product/Convertible%20Car**
5. Next, enter the following non-routed URL (using your port number) into the browser:
   **http://localhost:1234/ProductDetails.aspx?productID=2**
   The code still recognizes a URL that includes a query string, for the case where a user has a link bookmarked.

## Summary

In this tutorial, you have added routes for categories and products. You have learned how routes can be integrated with data controls that use model binding. In the next tutorial, you will implement global error handling.

## Additional Resources

ASP.NET Friendly URLs

# ASP.NET Error Handling

This tutorial series will teach you the basics of building an ASP.NET Web Forms application using ASP.NET 4.5 and Microsoft Visual Studio Express 2013 for Web. A Visual Studio 2013 project with C# source code is available to accompany this tutorial series.

In this tutorial, you will modify the Wingtip Toys sample application to include error handling and error logging. Error handling will allow the application to gracefully handle errors and display error messages accordingly. Error logging will allow you to find and fix errors that have occurred. This tutorial builds on the previous tutorial “URL Routing” and is part of the Wingtip Toys tutorial series.

## What you'll learn:

- How to add global error handling to the application’s configuration.
- How to add error handling at the application, page, and code levels.
- How to log errors for later review.
- How to display error messages that do not compromise security.
- How to implement Error Logging Modules and Handlers (ELMAH) error logging.

## Overview

ASP.NET applications must be able to handle errors that occur during execution in a consistent manner. ASP.NET uses the common language runtime (CLR), which provides a way of notifying applications of errors in a uniform way. When an error occurs, an exception is thrown. An exception is any error, condition, or unexpected behavior that an application encounters.

In the .NET Framework, an exception is an object that inherits from the `System.Exception` class. An exception is thrown from an area of code where a problem has occurred. The exception is passed up the call stack to a place where the application provides code to handle the exception. If the application does not handle the exception, the browser is forced to display the error details.

As a best practice, handle errors in at the code level in `Try`/`Catch`/`Finally` blocks within your code. Try to place these blocks so that the user can correct problems in the context in which they occur. If the error handling blocks are too far away from where the error occurred, it becomes more difficult to provide users with the information they need to fix the problem.

### Exception Class

The Exception class is the base class from which exceptions inherit. Most exception objects are instances of some derived class of the Exception class, such as the `SystemException` class, the `IndexOutOfRangeException` class, or the `ArgumentNullException` class. The Exception class has properties, such as the `StackTrace` property, the `InnerException` property, and the `Message` property, that provide specific information about the error that has occurred.

## Exception Inheritance Hierarchy

The runtime has a base set of exceptions deriving from the `SystemException` class that the runtime throws when an exception is encountered. Most of the classes that inherit from the Exception class, such as the `IndexOutOfRangeException` class and the `ArgumentNullException` class, do not implement additional members. Therefore, the most important information for an exception can be found in the hierarchy of exceptions, the exception name, and the information contained in the exception.

## Exception Handling Hierarchy

In an ASP.NET Web Forms application, exceptions can be handled based on a specific handling hierarchy. An exception can be handled at the following levels:

1. Application level
2. Page level
3. Code level

When an application handles exceptions, additional information about the exception that is inherited from the Exception class can often be retrieved and displayed to the user. In addition to application, page, and code level, you can also handle exceptions at the HTTP module level and by using an IIS custom handler.

## Application Level Error Handling

You can handle default errors at the application level either by modifying your application's configuration or by adding an `Application_Error` handler in the *Global.asax* file of your application.

You can handle default errors and HTTP errors by adding a `customErrors` section to the *Web.config* file. The `customErrors` section allows you to specify a default page that users will be redirected to when an error occurs. It also allows you to specify individual pages for specific status code errors.

```
<configuration>
  <system.web>
    <customErrors mode="On"
defaultRedirect="ErrorPage.aspx?handler=customErrors%20section%20-
%20Web.config">
      <error statusCode="404"
redirect="ErrorPage.aspx?msg=404&amp;handler=customErrors%20section%20-
%20Web.config"/>
    </customErrors>
  </system.web>
</configuration>
```

Unfortunately, when you use the configuration to redirect the user to a different page, you do not have the details of the error that occurred.

However, you can trap errors that occur anywhere in your application by adding code to the `Application_Error` handler in the *Global.asax* file.

```
void Application_Error(object sender, EventArgs e)
{
    Exception exc = Server.GetLastError();

    if (exc is HttpUnhandledException)
    {
        // Pass the error on to the error page.
        Server.Transfer("ErrorPage.aspx?handler=Application_Error%20-
%20Global.asax", true);
    }
}
```

## Page Level Error Event Handling

A page-level handler returns the user to the page where the error occurred, but because instances of controls are not maintained, there will no longer be anything on the page. To provide the error details to the user of the application, you must specifically write the error details to the page.

You would typically use a page-level error handler to log unhandled errors or to take the user to a page that can display helpful information.

This code example shows a handler for the Error event in an ASP.NET Web page. This handler catches all exceptions that are not already handled within `try`/`catch` blocks in the page.

```
        private void Page_Error(object sender, EventArgs e)
        {
            Exception exc = Server.GetLastError();

            // Handle specific exception.
            if (exc is HttpUnhandledException)
            {
                ErrorMsgTextBox.Text = "An error occurred on this page. Please
verify your " +
                "information to resolve the issue."
            }
            // Clear the error from the server.
            Server.ClearError();
        }
```

After you handle an error, you must clear it by calling the `ClearError` method of the Server object (`HttpServerUtility` class), otherwise you will see an error that has previously occurred.

## Code Level Error Handling

The try-catch statement consists of a try block followed by one or more catch clauses, which specify handlers for different exceptions. When an exception is thrown, the common language runtime (CLR) looks for the catch statement that handles this exception. If the currently executing method does not contain a catch block, the CLR looks at the method that called the current method, and so on, up the call stack. If no catch block is found, then the CLR displays an unhandled exception message to the user and stops execution of the program.

The following code example shows a common way of using `try`/`catch`/`finally` to handle errors.

```
    try
    {
        file.ReadBlock(buffer, index, buffer.Length);
    }
    catch (FileNotFoundException e)
    {
        Server.Transfer("NoFileErrorPage.aspx", true);
    }
    catch (System.IO.IOException e)
    {
        Server.Transfer("IOErrorPage.aspx", true);
    }

    finally
    {
        if (file != null)
        {
            file.Close();
        }
    }
```

In the above code, the try block contains the code that needs to be guarded against a possible exception. The block is executed until either an exception is thrown or the block is completed successfully. If either a `FileNotFoundException` exception or an `IOException` exception occurs, the execution is transferred to a different page. Then, the code contained in the finally block is executed, whether an error occurred or not.

## Adding Error Logging Support

Before adding error handling to the Wingtip Toys sample application, you will add error logging support by adding an `ExceptionUtility` class to the *Logic* folder. By doing this, each time the application handles an error, the error details will be added to the error log file.

1. Right-click the *Logic* folder and then select **Add** -> **New Item**.
   The **Add New Item** dialog box is displayed.
2. Select the **Visual C#** -> **Code** templates group on the left. Then, select **Class** from the middle list and name it **ExceptionUtility.cs**.
3. Choose **Add**. The new class file is displayed.
4. Replace the existing code with the following:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.IO;
```

```
namespace WingtipToys.Logic
{
  // Create our own utility for exceptions
  public sealed class ExceptionUtility
  {
    // All methods are static, so this can be private
    private ExceptionUtility()
    { }

    // Log an Exception
    public static void LogException(Exception exc, string source)
    {
      // Include logic for logging exceptions
      // Get the absolute path to the log file
      string logFile = "App_Data/ErrorLog.txt";
      logFile = HttpContext.Current.Server.MapPath(logFile);

      // Open the log file for append and write the log
      StreamWriter sw = new StreamWriter(logFile, true);
      sw.WriteLine("********** {0} **********", DateTime.Now);
      if (exc.InnerException != null)
      {
        sw.Write("Inner Exception Type: ");
        sw.WriteLine(exc.InnerException.GetType().ToString());
        sw.Write("Inner Exception: ");
        sw.WriteLine(exc.InnerException.Message);
        sw.Write("Inner Source: ");
        sw.WriteLine(exc.InnerException.Source);
        if (exc.InnerException.StackTrace != null)
        {
          sw.WriteLine("Inner Stack Trace: ");
          sw.WriteLine(exc.InnerException.StackTrace);
        }
      }
      sw.Write("Exception Type: ");
      sw.WriteLine(exc.GetType().ToString());
      sw.WriteLine("Exception: " + exc.Message);
      sw.WriteLine("Source: " + source);
      sw.WriteLine("Stack Trace: ");
      if (exc.StackTrace != null)
      {
        sw.WriteLine(exc.StackTrace);
        sw.WriteLine();
      }
      sw.Close();
    }
  }
}
```

When an exception occurs, the exception can be written to an exception log file by calling the `LogException` method. This method takes two parameters, the exception object and a string containing details about the source of the exception. The exception log is written to the *ErrorLog.txt* file in the *App_Data* folder.

## Adding an Error Page

In the Wingtip Toys sample application, one page will be used to display errors. The error page is designed to show a secure error message to users of the site. However, if the user is a developer making an HTTP request that is being served locally on the machine where the code lives, additional error details will be displayed on the error page.

1. Right-click the project name (**Wingtip Toys**) in **Solution Explorer** and select **Add** -> **New Item**.
   The **Add New Item** dialog box is displayed.
2. Select the **Visual C#** -> **Web** templates group on the left. From the middle list, select **Web Form with Master Page**, and name it **ErrorPage.aspx**.
3. Click **Add**.
4. Select the *Site.Master* file as the master page, and then choose **OK**.
5. Replace the existing markup with the following:

```
<%@ Page Title="" Language="C#" AutoEventWireup="true"
MasterPageFile="~/Site.Master"  CodeBehind="ErrorPage.aspx.cs"
Inherits="WingtipToys.ErrorPage" %>
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent" runat="server">
    <h2>Error:</h2>
    <p></p>
    <asp:Label ID="FriendlyErrorMsg" runat="server" Text="Label" Font-
Size="Large" style="color: red"></asp:Label>

    <asp:Panel ID="DetailedErrorPanel" runat="server" Visible="false">
        <p>&nbsp;</p>
        <h4>Detailed Error:</h4>
        <p>
            <asp:Label ID="ErrorDetailedMsg" runat="server" Font-Size="Small"
/><br />
        </p>

        <h4>Error Handler:</h4>
        <p>
            <asp:Label ID="ErrorHandler" runat="server" Font-Size="Small" /><br
/>
        </p>

        <h4>Detailed Error Message:</h4>
        <p>
            <asp:Label ID="InnerMessage" runat="server" Font-Size="Small" /><br
/>
        </p>
        <p>
            <asp:Label ID="InnerTrace" runat="server"  />
        </p>
    </asp:Panel>
</asp:Content>
```

6. Replace the existing code of the code-behind (*ErrorPage.aspx.cs*) so that it appears as follows:

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

```
namespace WingtipToys
{
  public partial class ErrorPage : System.Web.UI.Page
  {
    protected void Page_Load(object sender, EventArgs e)
    {
      // Create safe error messages.
      string generalErrorMsg = "A problem has occurred on this web site. Please
try again. " +
          "If this error continues, please contact support.";
      string httpErrorMsg = "An HTTP error occurred. Page Not found. Please try
again.";
      string unhandledErrorMsg = "The error was unhandled by application
code.";

      // Display safe error message.
      FriendlyErrorMsg.Text = generalErrorMsg;

      // Determine where error was handled.
      string errorHandler = Request.QueryString["handler"];
      if (errorHandler == null)
      {
        errorHandler = "Error Page";
      }

      // Get the last error from the server.
      Exception ex = Server.GetLastError();

      // Get the error number passed as a querystring value.
      string errorMsg = Request.QueryString["msg"];
      if (errorMsg == "404")
      {
        ex = new HttpException(404, httpErrorMsg, ex);
        FriendlyErrorMsg.Text = ex.Message;
      }

      // If the exception no longer exists, create a generic exception.
      if (ex == null)
      {
        ex = new Exception(unhandledErrorMsg);
      }

      // Show error details to only you (developer). LOCAL ACCESS ONLY.
      if (Request.IsLocal)
      {
        // Detailed Error Message.
        ErrorDetailedMsg.Text = ex.Message;

        // Show where the error was handled.
        ErrorHandler.Text = errorHandler;

        // Show local access details.
        DetailedErrorPanel.Visible = true;

        if (ex.InnerException != null)
        {
          InnerMessage.Text = ex.GetType().ToString() + "<br/>" +
              ex.InnerException.Message;
          InnerTrace.Text = ex.InnerException.StackTrace;
        }
        else
        {
          InnerMessage.Text = ex.GetType().ToString();
```

```
            if (ex.StackTrace != null)
            {
              InnerTrace.Text = ex.StackTrace.ToString().TrimStart();
            }
          }
        }

        // Log the exception.
        ExceptionUtility.LogException(ex, errorHandler);

        // Clear the error from the server.
        Server.ClearError();
      }
    }
  }
```

When the error page is displayed, the `Page_Load` event handler is executed. In the `Page_Load` handler, the location of where the error was first handled is determined. Then, the last error that occurred is determined by call the `GetLastError` method of the Server object. If the exception no longer exists, a generic exception is created. Then, if the HTTP request was made locally, all error details are shown. In this case, only the local machine running the web application will see these error details. After the error information has been displayed, the error is added to the log file and the error is cleared from the server.

## Displaying Unhandled Error Messages for the Application

By adding a `customErrors` section to the *Web.config* file, you can quickly handle simple errors that occur throughout the application. You can also specify how to handle errors based on their status code value, such as 404 – File not found.

### Update the Configuration

Update the configuration by adding a `customErrors` section to the *Web.config* file.

1. In **Solution Explorer**, find and open the *Web.config* file at the root of the Wingtip Toys sample application.
2. Add the `customErrors` section to the *Web.config* file within the `<system.web>` node as follows:

```
<configuration>
  <system.web>
    <customErrors mode="On" 
defaultRedirect="ErrorPage.aspx?handler=customErrors%20section%20-
%20Web.config">
      <error statusCode="404" 
redirect="ErrorPage.aspx?msg=404&amp;handler=customErrors%20section%20-
%20Web.config"/>
    </customErrors>
  </system.web>
</configuration>
```

3. Save the *Web.config* file.

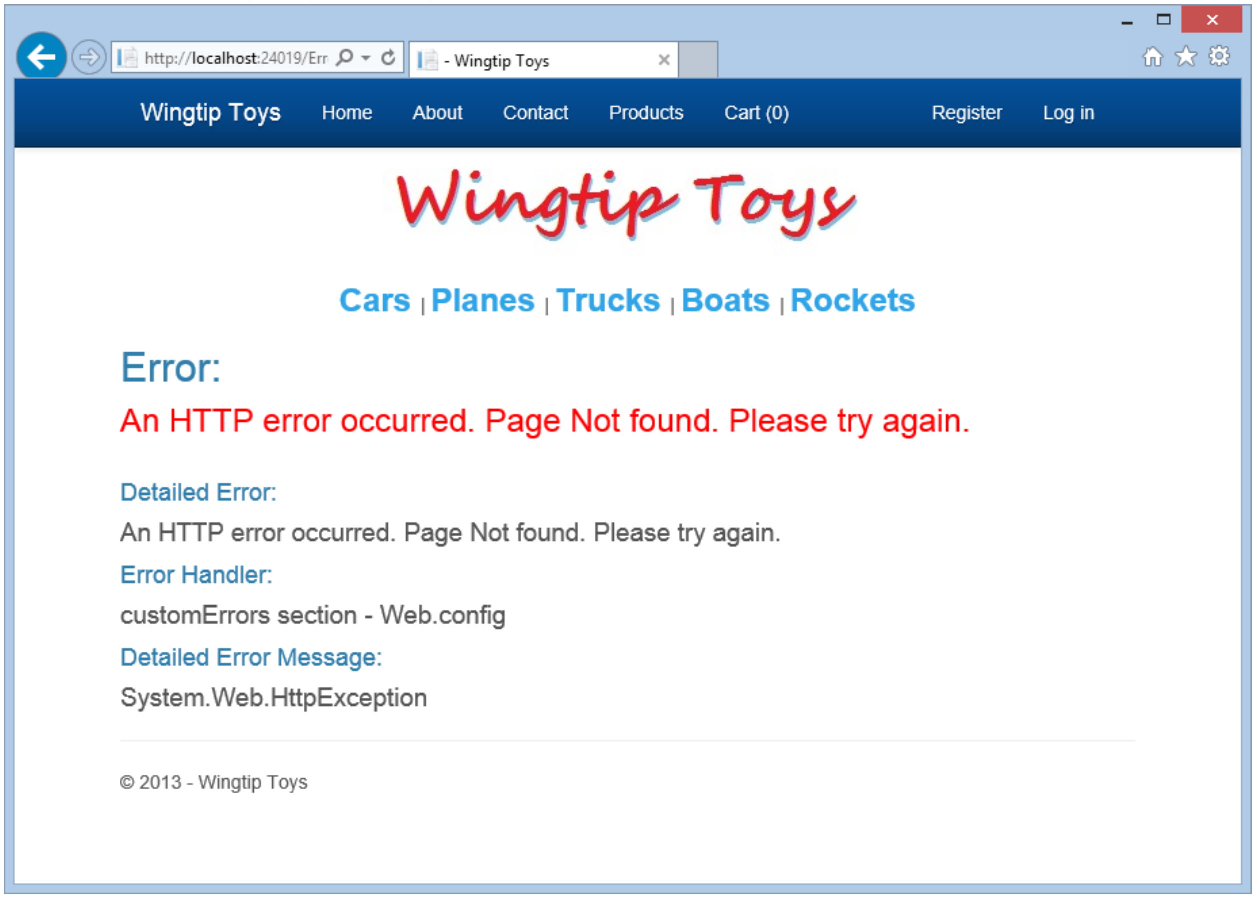
The `customErrors` section specifies the mode, which is set to "On". It also specifies the `defaultRedirect`, which tells the application which page to navigate to when an error occurs.

In addition, you have added a specific error element that specifies how to handle a 404 error when a page is not found. Later in this tutorial, you will add additional error handling that will capture the details of an error at the application level.

## Running the Application

You can run the application now to see the updated routes.

1. Press **F5** to run the Wingtip Toys sample application.
   The browser opens and shows the *Default.aspx* page.
2. Enter the following URL into the browser (be sure to use **your** port number):
   **http://localhost:1234/NoPage.aspx**
3. Review the *ErrorPage.aspx* displayed in the browser.



Error:

An HTTP error occurred. Page Not found. Please try again.

Detailed Error:

An HTTP error occurred. Page Not found. Please try again.

Error Handler:

customErrors section - Web.config

Detailed Error Message:

System.Web.HttpException

© 2013 - Wingtip Toys

When you request the *NoPage.aspx* page, which does not exist, the error page will show the simple error message and the detailed error information if additional details are available. However, if the user requested a non-existent page from a remote location, the error page would only show the error message in red.

## Including an Exception for Testing Purposes

To verify how your application will function when an error occurs, you can deliberately create error conditions in ASP.NET. In the Wingtip Toys sample application, you will throw a test exception when the default page loads to see what happens.

1. Open the code-behind of the *Default.aspx* page in Visual Studio.
   The *Default.aspx.cs* code-behind page will be displayed.
2. In the `Page_Load` handler, add code so that the handler appears as follows:

```
    protected void Page_Load(object sender, EventArgs e)
    {
      throw new InvalidOperationException("An InvalidOperationException " +
      "occurred in the Page_Load handler on the Default.aspx page.");
    }
```

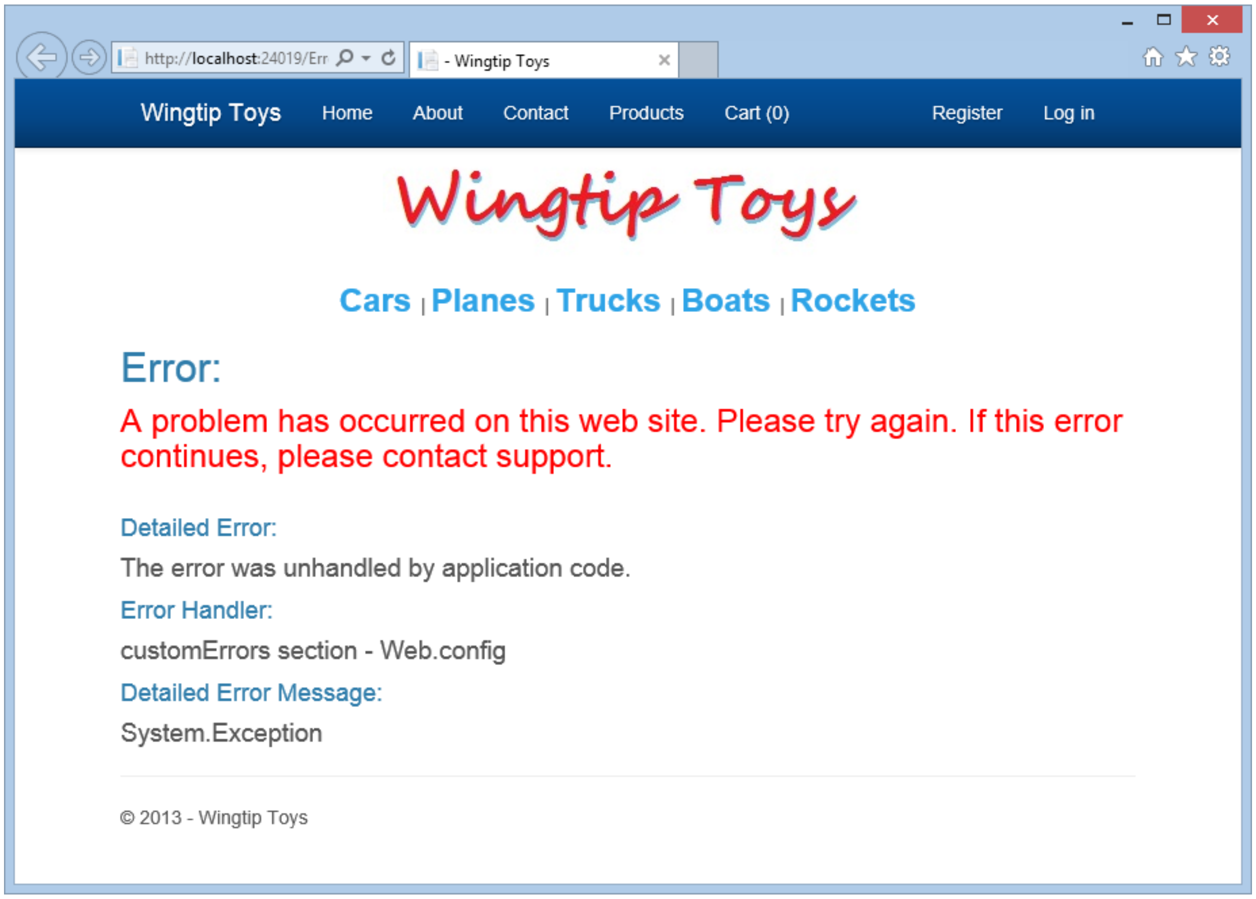
It is possible to create various different types of exceptions. In the above code, you are creating an `InvalidOperationException` when the *Default.aspx* page is loaded.

## Running the Application

You can run the application to see how the application handles the exception.

1. Press **CTRL+F5** to run the Wingtip Toys sample application.
   The application throws the InvalidOperationException.
   **Note**
   You must press **CTRL+F5** to display the page without breaking into the code to view the source of the error in Visual Studio.

2. Review the *ErrorPage.aspx* displayed in the browser.



Error:

A problem has occurred on this web site. Please try again. If this error continues, please contact support.

Detailed Error:

The error was unhandled by application code.

Error Handler:

customErrors section - Web.config

Detailed Error Message:

System.Exception

© 2013 - Wingtip Toys

As you can see in the error details, the exception was trapped by the `customError` section in the *Web.config* file.

## Adding Application-Level Error Handling

Rather than trap the exception using the `customErrors` section in the *Web.config* file, where you gain little information about the exception, you can trap the error at the application level and retrieve error details.

1. In **Solution Explorer**, find and open the *Global.asax.cs* file.
2. Add an **Application_Error** handler so that it appears as follows:

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

  // Get last error from the server
  Exception exc = Server.GetLastError();

  if (exc is HttpUnhandledException)
  {
    if (exc.InnerException != null)
    {
      exc = new Exception(exc.InnerException.Message);
      Server.Transfer("ErrorPage.aspx?handler=Application_Error%20-
%20Global.asax",
          true);
    }
  }
}
```

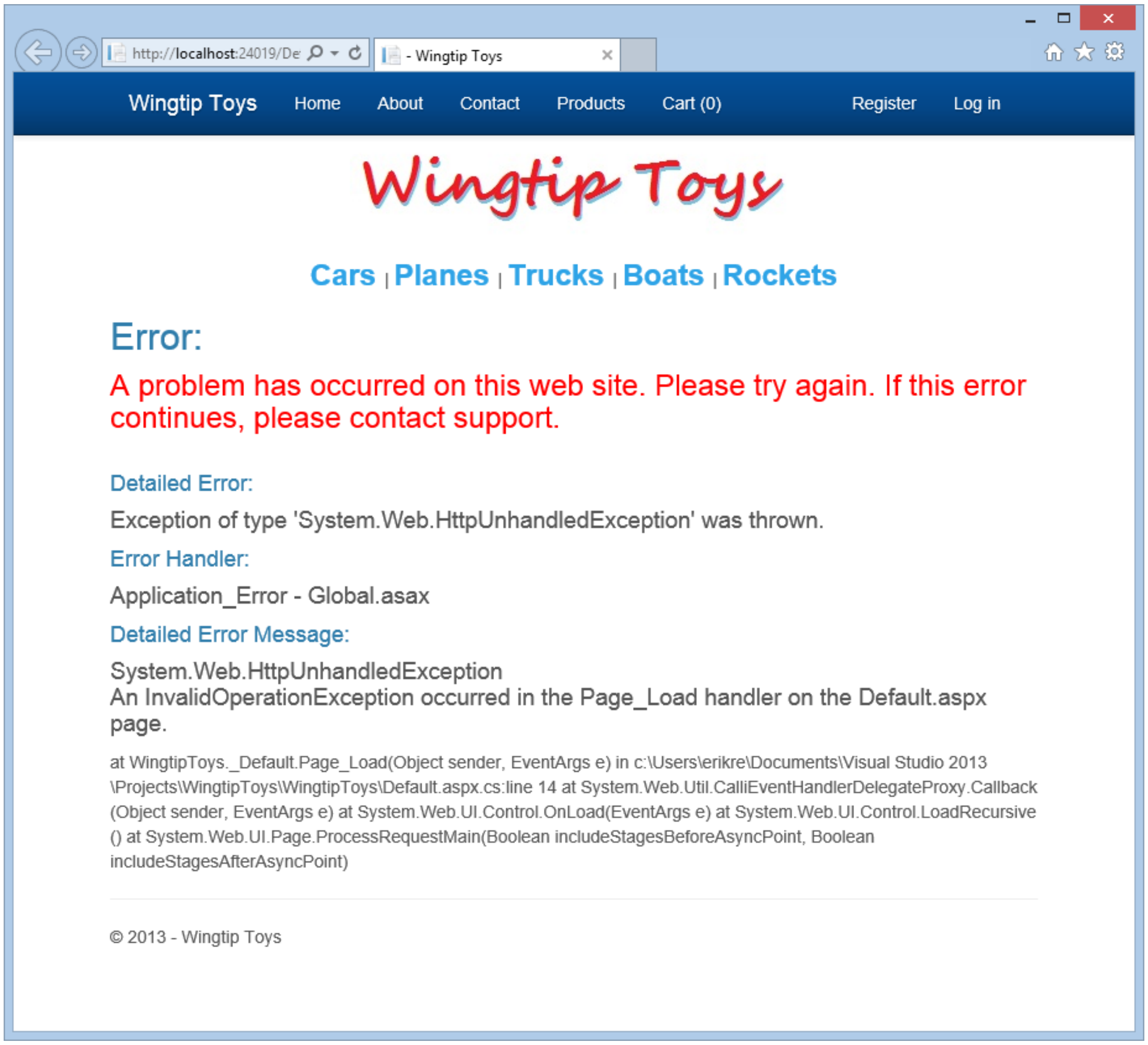
When an error occurs in the application, the `Application_Error` handler is called. In this handler, the last exception is retrieved and reviewed. If the exception was unhandled and the exception contains inner-exception details (that is, `InnerException` is not null), the application transfers execution to the error page where the exception details are displayed.

## Running the Application

You can run the application to see the additional error details provided by handling the exception at the application level.

1. Press **CTRL+F5** to run the Wingtip Toys sample application.
   The application throws the `InvalidOperationException`.

2. Review the *ErrorPage.aspx* displayed in the browser.



Error:

A problem has occurred on this web site. Please try again. If this error continues, please contact support.

Detailed Error:

Exception of type 'System.Web.HttpUnhandledException' was thrown.

Error Handler:

Application_Error - Global.asax

Detailed Error Message:

System.Web.HttpUnhandledException
An InvalidOperationException occurred in the Page_Load handler on the Default.aspx page.

at WingtipToys._Default.Page_Load(Object sender, EventArgs e) in c:\Users\erikre\Documents\Visual Studio 2013 \Projects\WingtipToys\WingtipToys\Default.aspx.cs:line 14 at System.Web.Util.CalliEventHandlerDelegateProxy.Callback (Object sender, EventArgs e) at System.Web.UI.Control.OnLoad(EventArgs e) at System.Web.UI.Control.LoadRecursive () at System.Web.UI.Page.ProcessRequestMain(Boolean includeStagesBeforeAsyncPoint, Boolean includeStagesAfterAsyncPoint)

© 2013 - Wingtip Toys

## Adding Page-Level Error Handling

You can add page-level error handling to a page either by using adding an `ErrorPage` attribute to the `@Page` directive of the page, or by adding a `Page_Error` event handler to the code-behind of a page. In this section, you will add a `Page_Error` event handler that will transfer execution to the *ErrorPage.aspx* page.

1. In **Solution Explorer**, find and open the *Default.aspx.cs* file.
2. Add a `Page_Error` handler so that the code-behind appears as follows:

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

namespace WingtipToys
{
  public partial class _Default : Page
  {
    protected void Page_Load(object sender, EventArgs e)
    {
      throw new InvalidOperationException("An InvalidOperationException " +
      "occurred in the Page_Load handler on the Default.aspx page.");
    }

    private void Page_Error(object sender, EventArgs e)
    {
      // Get last error from the server.
      Exception exc = Server.GetLastError();

      // Handle specific exception.
      if (exc is InvalidOperationException)
      {
        // Pass the error on to the error page.
        Server.Transfer("ErrorPage.aspx?handler=Page_Error%20-%20Default.aspx",
            true);
      }
    }
  }
}
```

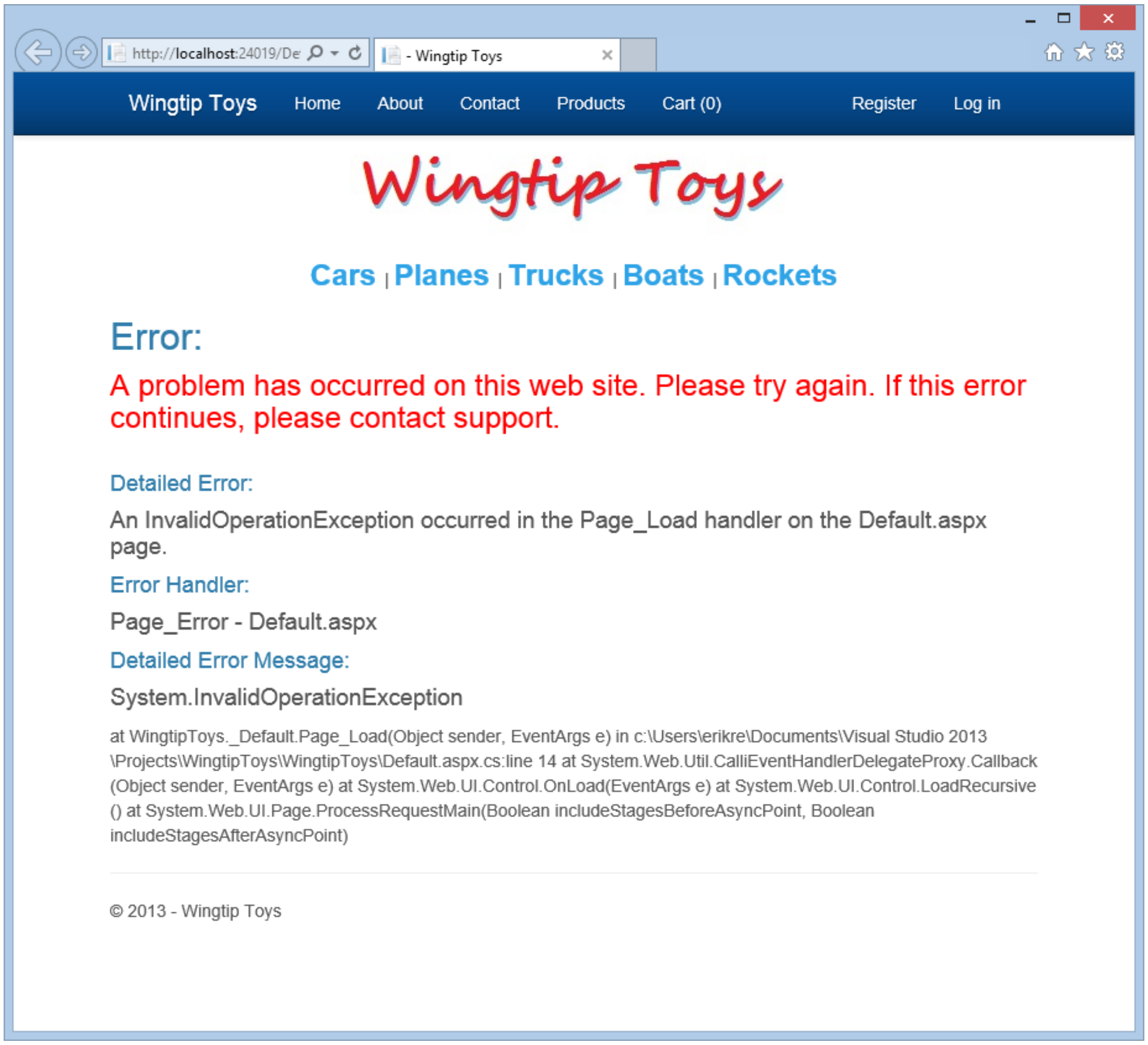
When an error occurs on the page, the `Page_Error` event handler is called. In this handler, the last exception is retrieved and reviewed. If an `InvalidOperationException` occurs, the `Page_Error` event handler transfers execution to the error page where the exception details are displayed.

## Running the Application

You can run the application now to see the updated routes.

1. Press **CTRL+F5** to run the Wingtip Toys sample application.
   The application throws the `InvalidOperationException`.

2. Review the *ErrorPage.aspx* displayed in the browser.



Error:

A problem has occurred on this web site. Please try again. If this error continues, please contact support.

Detailed Error:

An InvalidOperationException occurred in the Page_Load handler on the Default.aspx page.

Error Handler:

Page_Error - Default.aspx

Detailed Error Message:

System.InvalidOperationException

at WingtipToys._Default.Page_Load(Object sender, EventArgs e) in c:\Users\erikre\Documents\Visual Studio 2013 \Projects\WingtipToys\WingtipToys\Default.aspx.cs:line 14 at System.Web.Util.CalliEventHandlerDelegateProxy.Callback (Object sender, EventArgs e) at System.Web.UI.Control.OnLoad(EventArgs e) at System.Web.UI.Control.LoadRecursive () at System.Web.UI.Page.ProcessRequestMain(Boolean includeStagesBeforeAsyncPoint, Boolean includeStagesAfterAsyncPoint)

© 2013 - Wingtip Toys

3. Close your browser window.

## Removing the Exception Used for Testing

To allow the Wingtip Toys sample application to function without throwing the exception you added earlier in this tutorial, remove the exception.

1. Open the code-behind of the *Default.aspx* page.
2. In the `Page_Load` handler, remove the code that throws the exception so that the handler appears as follows:

```
    protected void Page_Load(object sender, EventArgs e)
    {

    }
```

## Adding Code-Level Error Logging

As mentioned earlier in this tutorial, you can add try/catch statements to attempt to run a section of code and handle the first error that occurs. In this example, you will only write the error details to the error log file so that the error can be reviewed later.

1. In **Solution Explorer**, in the *Logic* folder, find and open the *PayPalFunctions.cs* file.
2. Update the `HttpCall` method so that the code appears as follows:

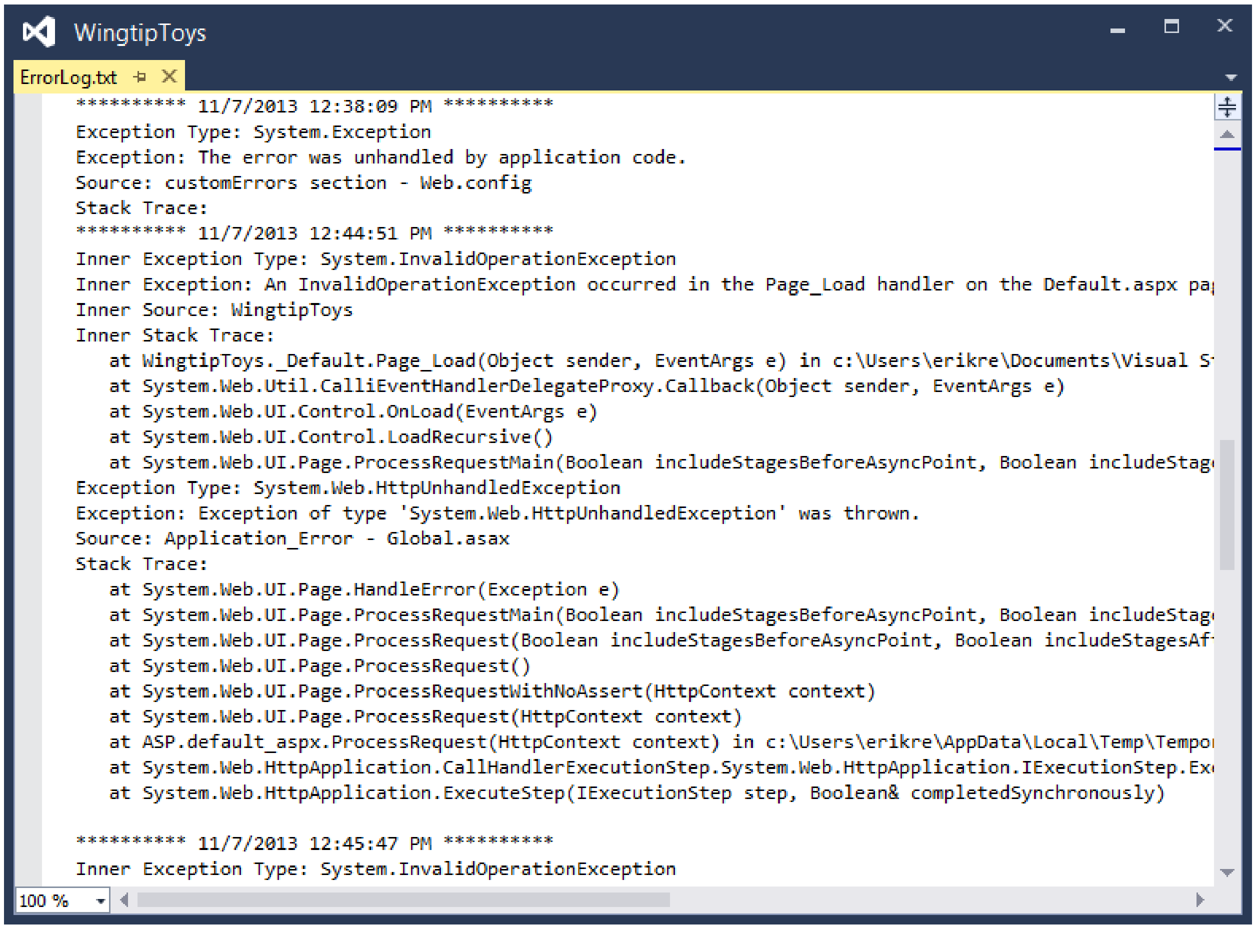
```
public string HttpCall(string NvpRequest)
{
  string url = pEndPointURL;

  string strPost = NvpRequest + "&" + buildCredentialsNVPString();
  strPost = strPost + "&BUTTONSOURCE=" + HttpUtility.UrlEncode(BNCode);

  HttpWebRequest objRequest = (HttpWebRequest)WebRequest.Create(url);
  objRequest.Timeout = Timeout;
  objRequest.Method = "POST";
  objRequest.ContentLength = strPost.Length;

  try
  {
    using (StreamWriter myWriter = new
StreamWriter(objRequest.GetRequestStream()))
    {
      myWriter.Write(strPost);
    }
  }
  catch (Exception e)
  {
    // Log the exception.
    WingtipToys.Logic.ExceptionUtility.LogException(e, "HttpCall in
PayPalFunction.cs");
  }

  //Retrieve the Response returned from the NVP API call to PayPal.
  HttpWebResponse objResponse = (HttpWebResponse)objRequest.GetResponse();
  string result;
  using (StreamReader sr = new StreamReader(objResponse.GetResponseStream()))
  {
    result = sr.ReadToEnd();
  }

  return result;
}
```

The above code calls the `LogException` method that is contained in the `ExceptionUtility` class. You added the *ExceptionUtility.cs* class file to the *Logic* folder earlier in this tutorial. The `LogException` method takes two parameters. The first parameter is the exception object. The second parameter is a string used to recognize the source of the error.

## Inspecting the Error Logging Information

As mentioned previously, you can use the error log to determine which errors in your application should be fixed first. Of course, only errors that have been trapped and written to the error log will be recorded.

1. In **Solution Explorer**, find and open the *ErrorLog.txt* file in the *App_Data* folder.
   You may need to select the "**Show All Files**" option or the "**Refresh**" option from the top of **Solution Explorer** to see the *ErrorLog.txt* file.
2. Review the error log displayed in Visual Studio:



```
********** 11/7/2013 12:38:09 PM **********
Exception Type: System.Exception
Exception: The error was unhandled by application code.
Source: customErrors section - Web.config
Stack Trace:
********** 11/7/2013 12:44:51 PM **********
Inner Exception Type: System.InvalidOperationException
Inner Exception: An InvalidOperationException occurred in the Page_Load handler on the Default.aspx pa
Inner Source: WingtipToys
Inner Stack Trace:
   at WingtipToys._Default.Page_Load(Object sender, EventArgs e) in c:\Users\erikre\Documents\Visual S
   at System.Web.Util.CalliEventHandlerDelegateProxy.Callback(Object sender, EventArgs e)
   at System.Web.UI.Control.OnLoad(EventArgs e)
   at System.Web.UI.Control.LoadRecursive()
   at System.Web.UI.Page.ProcessRequestMain(Boolean includeStagesBeforeAsyncPoint, Boolean includeStag
Exception Type: System.Web.HttpUnhandledException
Exception: Exception of type 'System.Web.HttpUnhandledException' was thrown.
Source: Application_Error - Global.asax
Stack Trace:
   at System.Web.UI.Page.HandleError(Exception e)
   at System.Web.UI.Page.ProcessRequestMain(Boolean includeStagesBeforeAsyncPoint, Boolean includeStag
   at System.Web.UI.Page.ProcessRequest(Boolean includeStagesBeforeAsyncPoint, Boolean includeStagesAf
   at System.Web.UI.Page.ProcessRequest()
   at System.Web.UI.Page.ProcessRequestWithNoAssert(HttpContext context)
   at System.Web.UI.Page.ProcessRequest(HttpContext context)
   at ASP.default_aspx.ProcessRequest(HttpContext context) in c:\Users\erikre\AppData\Local\Temp\Tempo
   at System.Web.HttpApplication.CallHandlerExecutionStep.System.Web.HttpApplication.IExecutionStep.Ex
   at System.Web.HttpApplication.ExecuteStep(IExecutionStep step, Boolean& completedSynchronously)

********** 11/7/2013 12:45:47 PM **********
Inner Exception Type: System.InvalidOperationException
```

### Safe Error Messages

It is **important to note** that when your application displays error messages, it should not give away information that a malicious user might find helpful in attacking your application. For example, if your application unsuccessfully tries to write in to a database, it should not display an error message that includes the user name it is using. For this reason, a generic error message in red is displayed to the user. All additional error details are only displayed to the developer on the local machine.

## Using ELMAH

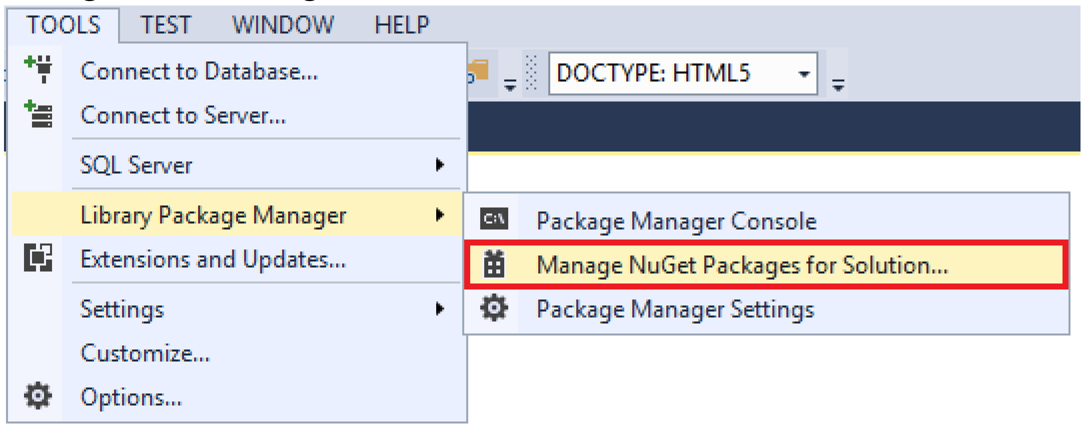
ELMAH (Error Logging Modules and Handlers) is an error logging facility that you plug into your ASP.NET application as a NuGet package. ELMAH provides the following capabilities:

4. Logging of unhandled exceptions.
5. A web page to view the entire log of recoded unhandled exceptions.

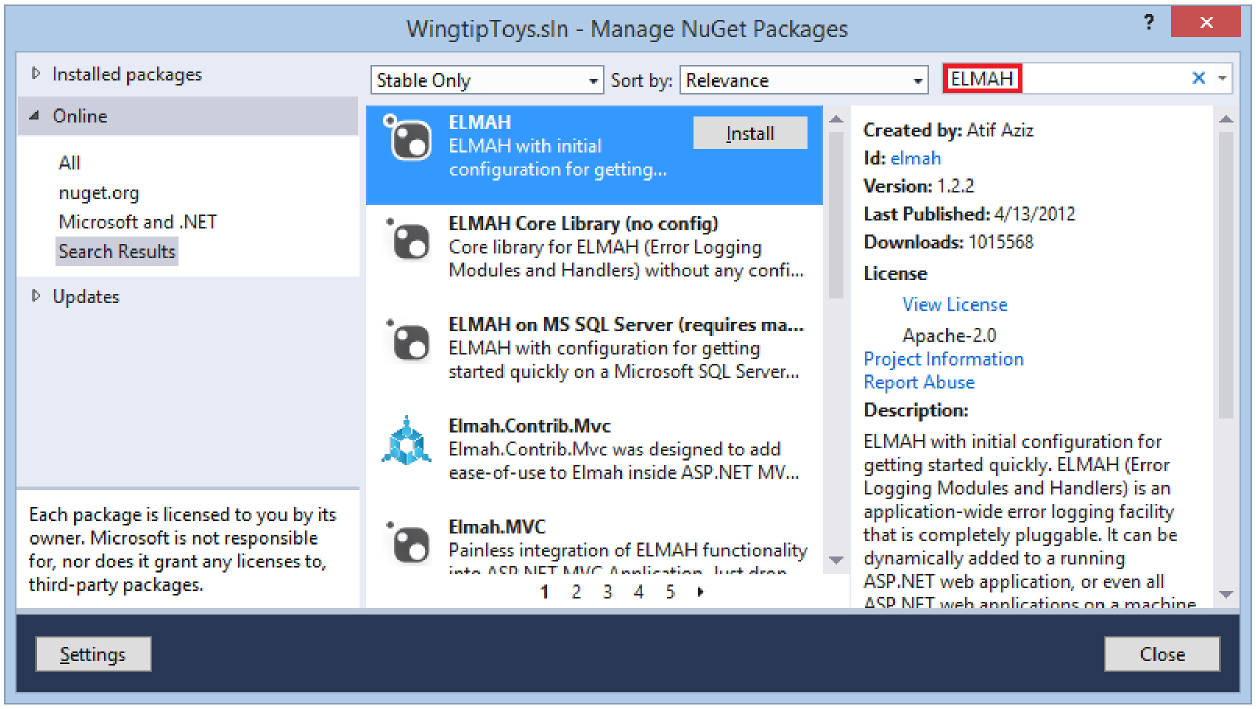
6. A web page to view the full details of each logged exception.
7. An e-mail notification of each error at the time it occurs.
8. An RSS feed of the last 15 errors from the log.

Before you can work with the ELMAH, you must install it. This is easy using the *NuGet* package installer. As mentioned earlier in this tutorial series, 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**.



2. The **Manage NuGet Packages** dialog box is displayed within Visual Studio.
3. In the **Manage NuGet Packages** dialog box, expand **Online** on the left, and then select **nuget.org**. Then, find and install the **ELMAH** package from the list of available packages online.



Each package is licensed to you by its owner. Microsoft is not responsible for, nor does it grant any licenses to, third-party packages.

**ELMAH**
ELMAH with initial configuration for getting...

**ELMAH Core Library (no config)**
Core library for ELMAH (Error Logging Modules and Handlers) without any confi...

**ELMAH on MS SQL Server (requires ma...**
ELMAH with configuration for getting started quickly on a Microsoft SQL Server...

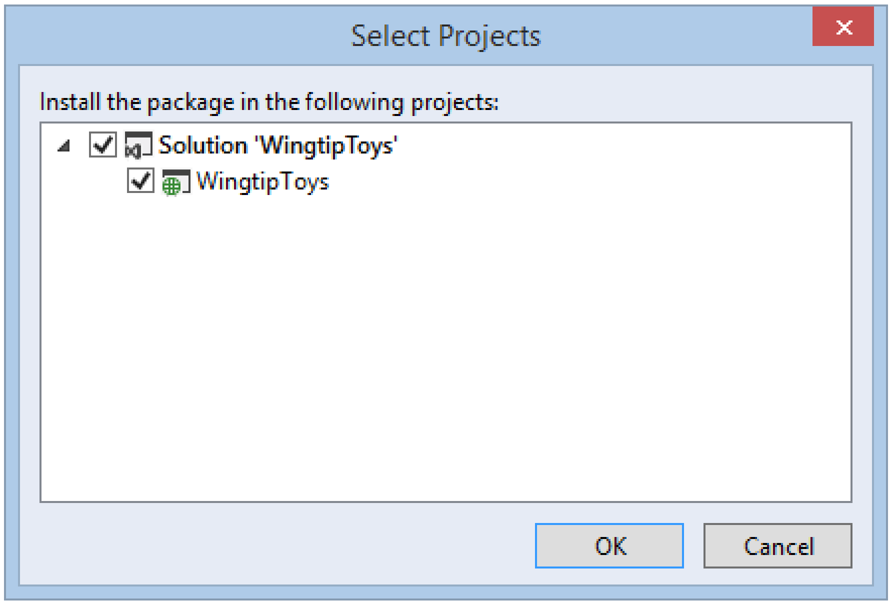
**Elmah.Contrib.Mvc**
Elmah.Contrib.Mvc was designed to add ease-of-use to Elmah inside ASP.NET MV...

**Elmah.MVC**
Painless integration of ELMAH functionality

**Created by:** Atif Aziz
**Id:** elmah
**Version:** 1.2.2
**Last Published:** 4/13/2012
**Downloads:** 1015568
**License**
View License
Apache-2.0
Project Information
Report Abuse
**Description:**
ELMAH with initial configuration for getting started quickly. ELMAH (Error Logging Modules and Handlers) is an application-wide error logging facility that is completely pluggable. It can be dynamically added to a running ASP.NET web application, or even all

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

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



6. Click **Close** in **the Manage NuGet Packages** dialog box if needed.
7. If Visual Studio requests that you reload any open files, select "**Yes to All**".
8. The ELMAH package adds entries for itself in the *Web.config* file at the root of your project. If Visual Studio asks you if you want to reload the modified *Web.config* file, click **Yes**.

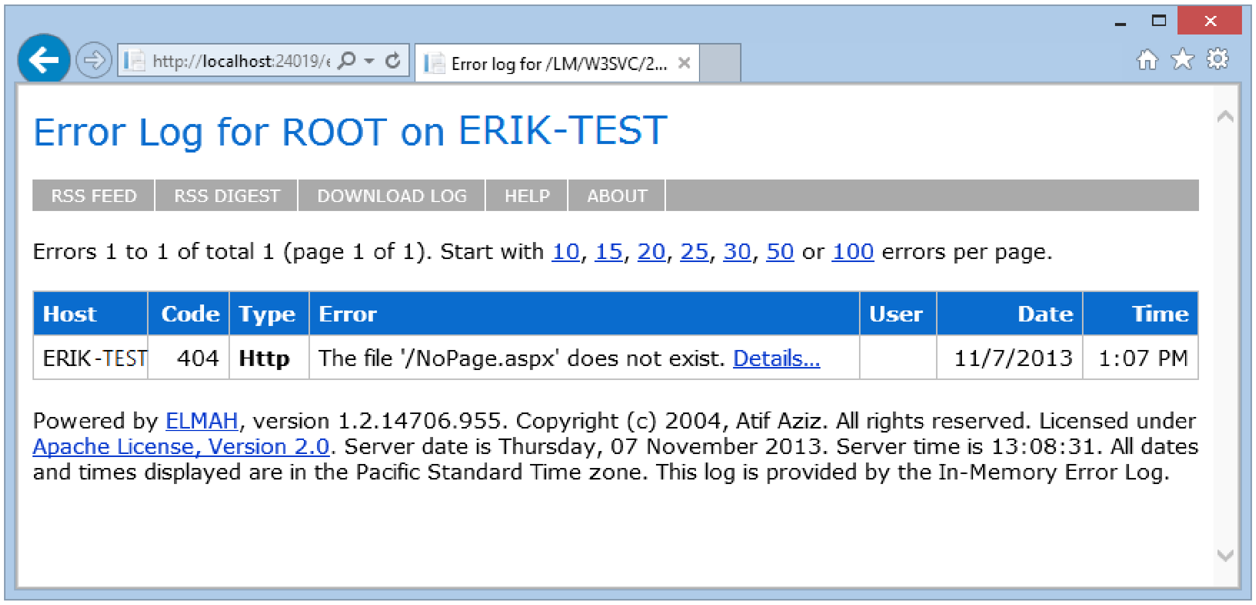
ELMAH is now ready to store any unhandled errors that occur.

## Viewing the ELMAH Log

Viewing the ELMAH log is easy, but first you will create an unhandled exception that will be recorded in the ELMAH log.

1. Press **CTRL+F5** to run the Wingtip Toys sample application.
2. To write an unhandled exception to the ELMAH log, navigate in your browser to the following URL (using your port number):
   **http://localhost:1234/NoPage.aspx**
   The error page will be displayed.
3. To display the ELMAH log, navigate in your browser to the following URL (using your port number):

**http://localhost:1234/elmah.axd**



Error Log for ROOT on ERIK-TEST

Errors 1 to 1 of total 1 (page 1 of 1). Start with 10, 15, 20, 25, 30, 50 or 100 errors per page.

| Host | Code | Type | Error | User | Date | Time |
|---|---|---|---|---|---|---|
| ERIK-TEST | 404 | Http | The file '/NoPage.aspx' does not exist. Details... | | 11/7/2013 | 1:07 PM |

Powered by ELMAH, version 1.2.14706.955. Copyright (c) 2004, Atif Aziz. All rights reserved. Licensed under Apache License, Version 2.0. Server date is Thursday, 07 November 2013. Server time is 13:08:31. All dates and times displayed are in the Pacific Standard Time zone. This log is provided by the In-Memory Error Log.

## Summary

In this tutorial, you have learned about handling errors at the application level, the page level, and the code level. You have also learned how to log handled and unhandled errors for later review. You added the ELMAH utility to provide exception logging and notification to your application using NuGet. Additionally, you have learned about the importance of safe error messages.

## Conclusion

This completes the ASP.NET 4.5 Wingtip Toys tutorial series. For more information about new Web Forms features available in ASP.NET 4.5 and Visual Studio 2013, see ASP.NET and Web Tools for Visual Studio 2013 Release Notes.

## Additional Resources

Logging Error Details with ASP.NET Health Monitoring
ELMAH

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

## Community Contributions

- Graham Mendick (graham.mendick@gmail.com)
  Visual Studio 2012 related code sample on MSDN: Navigation Wingtip Toys
- James Chaney (jchaney@agvance.net)
  Visual Studio 2012 related code sample on MSDN: ASP.NET 4.5 Web Forms Tutorial Series in Visual Basic
- Andrielle Azevedo - Microsoft Technical Audience Contributor (twitter: @driazevedo)
  Visual Studio 2012 translation: Iniciando com ASP.NET Web Forms 4.5 – Parte 1 – Introdução e Visão Geral