https://www.pluralsight.com/guides/asp-net-mvc-creating-solutions-with-separate-projects-for-entities-data-access-and-website-functionality

## **ASP.NET MVC - Creating Solutions with Separate Projects for Entities, Data Access, and Website Functionality**

## Introduction

Microsoft Visual Studio facilitates the development of complex software by enabling developers to create solutions consisting of multiple projects of different types. Projects can compile into executables, DLL's, or other forms, and interact with each other by referencing the interfaces, methods, and properties of the other projects in the solution.

Even relatively simple solutions can benefit from separating software layers and components into separate projects.

Separating large, complex solutions into multiple projects can improve developer productivity and make it easier to test, deploy, and maintain the solution over the application lifecycle.

### Skill Levels

This guide is best suited to those with the following skill levels, by topic:

| **Technology** | **Skill Level** |
| --- | --- |
| ASP.NET | Intermediate |
| C# | Intermediate |
| Entity Framework | Beginner |
| MVC | Intermediate |

## Scope

This guide is intended to be an introduction to the concept and techniques, rather than an extensive survey of the possible solution configurations and their advantages and implications. This guide will show how to structure a solution for a simple web application using Entity Framework and the Model-View-ViewModel (MVVM) design pattern along with the repository pattern.

Solutions composed of multiple projects sometimes emerge from single project solutions. This guide will demonstrate how to upgrade a single project solution into a multi-project solution.

This guide focuses on .NET Framework solutions and does not include information on .NET Core solutions.

Deployment considerations will be covered in a separate guide.

## Structure

1. First we'll take a look at the hierarchical solution structure and some suggested naming conventions.
2. Then we'll look at a standard process for setting up a multi-project solution for a data-driven web application.
3. Next we'll follow a case study of converting a single project web application to a multi-project application.
4. We'll conclude with some tips for special situations.

## Prerequisites

You should have:

* An understanding of ASP.NET MVC, and
* An understanding of the Model-View-ViewModel (MVVM) design pattern.

## Hierarchical Solution Structure for ASP.NET MVC with Entity Framework, MVVM, and the Repository Pattern

### Overview of Concepts

The ASP.NET MVC (Model-View-Controller) web application framework provides a design pattern incorporating the principle of separation of concerns (SoC). This is a modular approach to design, isolating the information required to perform a specific function of a computer program within the module responsible for that concern.

The **M**odel portion of MVC is implemented with C# classes that define the properties and methods of the data objects manipulated by the application. These class objects are referred to as entities when they are instantiated (they are also frequently referred to as **POCO**'s, "**plain-old class objects**" because each object is just a C# class).

The Model-View-ViewModel (MVVM) design pattern separates the models which represent the data objects in their conceptual form from the implementation of model classes used to create views used to perform CRUD (create, retrieve, update, delete) operations on data entities. A **V**iew **M**odel reflects the requirements of the view rather than the conceptual objects affected by the view.

Entity Framework (EF) is an object-relational mapper (ORM) that provides another layer of SoC by creating an interface between persistent storage of data in a relational database (RDB) and the model objects of the MVC framework. EF enables code-first database design: the developer can write class objects which are translated into SQL data definition language (DDL) by EF that creates and modifies the underlying database, keeping the data store in sync with the Model through migrations. Entity Framework is added to a project through a NuGet package.

The **repository pattern** of design separates the logic that stores and retrieves data from the business logic that acts on the data. This provides a number of benefits, including eliminating duplication of code and facilitating testing. In web applications, using the repository pattern facilitates the writing of short, well-defined **C**ontroller actions (business logic) that act on strongly typed data objects.

### Structuring the Solution

If all the code for the above was incorporated into one project, the project directory would include folders and files for:

* controllers
* views
* view models
* entity models
* repositories
* the data context
* EF migrations

Plus THE files for application startup, configuration, fonts, routing, scripts, style sheets (CSS), and utility classes. That's a lot, and whenever a change is made in one area the code for the whole has to be redeployed.

Using separate projects within a Visual Studio solution we can create a greater separation of concerns and facilitate development, debugging, testing, and deployment. The various layers of a web application can be divided into projects as follows:

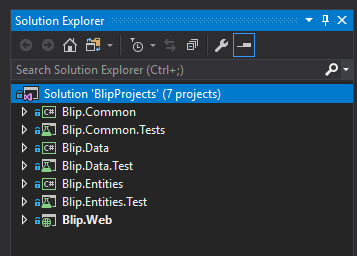
| **Project** | **Components** |
| --- | --- |
| Web | Controllers, Views |
| Entities | Models, View Models |
| Data | Data Context, Repositories |
| Common | Utility functions, Enumerations |

More complex solutions might also have separate projects for logging, financial transactions, and other functions. To facilitate work on solutions with many projects, Visual Studio provides selective loading of projects in solutions with more than 30 projects, by default.

Each of the projects identified above might also be paralleled with a unit test project.

### Example Solution Structure

The Visual Studio **Solution Explorer** view of a web application structured in the fashion shown above, including test projects, might look like this:

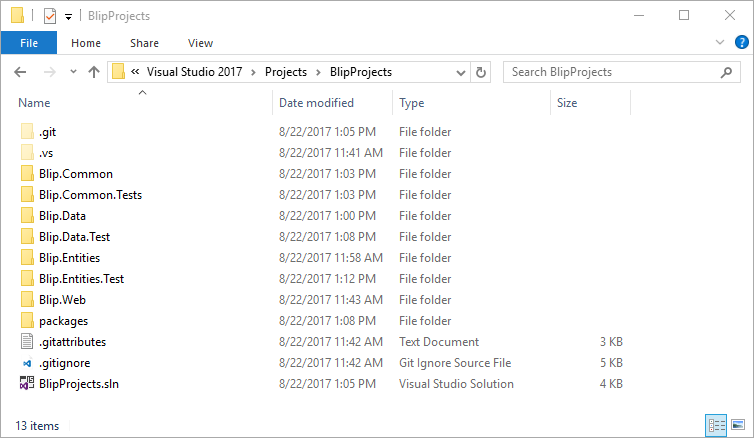


Visual Studio solution explorer: multi-project solution

Note the following attributes of the example shown above:

* **Dots** ("**.**") can be used as separators in *project names* to facilitate grouping related projects together.
* The convention for *unit test projects* is to add "**.Test**" to the name of the project on which the tests will operate.
* Only **Blip.Web** is a .NET Framework MVC Web Application project type (as denoted by the green globe icon), the other projects are plain .NET Framework (empty) projects (as denoted by the green "C#").

The folder structure corresponding to the above solution structure looks like this:



File Explorer: BlipProjects directory

## The BlipProjects Example Project

The example project for this guide, [**BlipProjects**](https://github.com/ajsaulsberry/BlipProjects), can be found on GitHub. You can fork the repository or download the solution in a **.zip** file to follow along with the remainder of this guide.

**BlipProjects** is a multi-project implementation of another sample project, [**BlipDrop**](https://github.com/ajsaulsberry/BlipDrop) that can also be found on GitHub. So you can easily compare the before and after states, there is no difference in functionality or code between the two, aside from the addition of a Common project containing some utility code.

### Creating a Multi-project Solution

The process for creating a multi-project solution is illustrated below using the **BlipProjects** example solution, which is a web application. These steps are common to creating any multi-project solution.

## Create a New Project in Visual Studio

In Visual Studio, select **File** / **New** / **Project** (**Ctrl** + **Shift** + **N**).

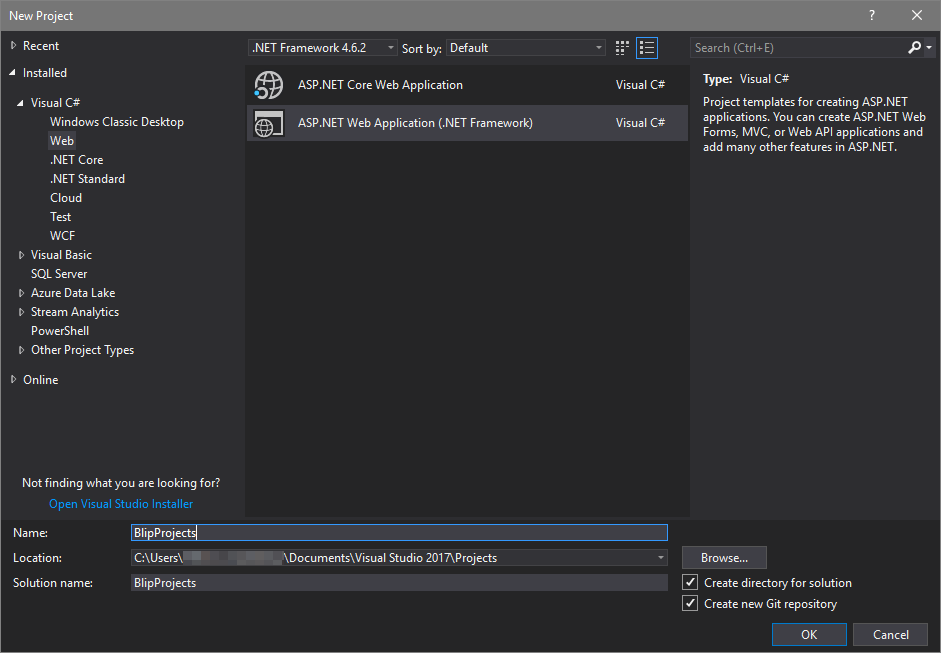
Note that there is no "new solution" option in Visual Studio 2017 or earlier.

***Important Note:****These instructions apply to versions of Visual Studio prior to 15.3.2, released 21 August 2017, which modified the New Project dialog box to add the ability to:*

* *Create a new solution while creating a project*
* *Define the solution name separately from the project name*
* *Add a new project to an existing solution or to a new solution*

These changes make it considerably easier to do the initial setup for a new multi-project solution.

You should see a modal dialog box like this:



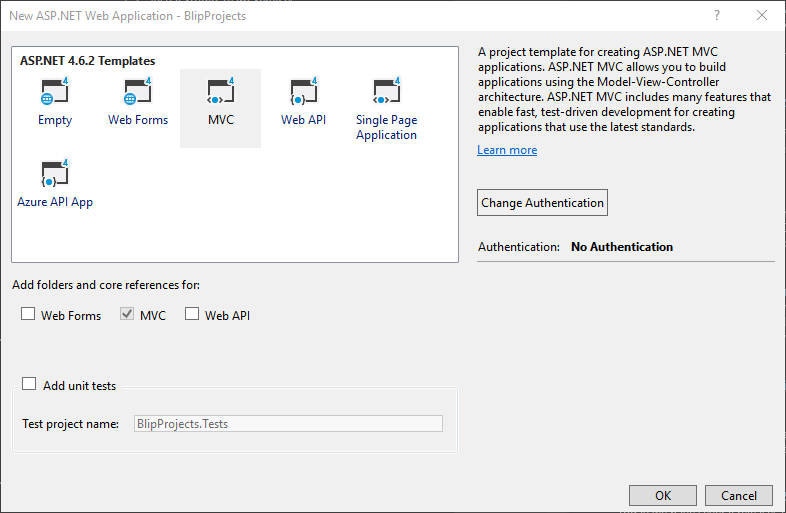
Visual Studio: New Project dialog box

Be sure to do the following:

1. **Name**: give the project the name you would like to assign to the entire solution. This will typically be a name that describes the whole application. You will change the name of the web project in a forthcoming step.
2. Check **Create directory for solution**. This creates the proper directory structure for a solution containing multiple projects.
3. Check **Create new Git repository**. Because life is so much better with source code control.

Click **OK**.

You will be taken to a modal dialog box where you can choose the type of project:



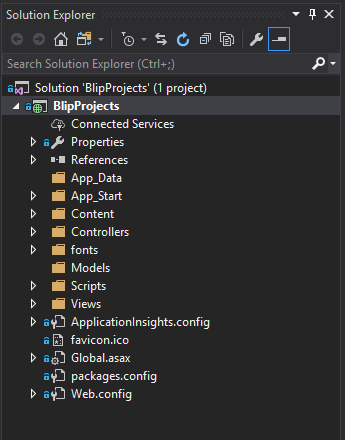
Visual Studio New ASP.NET Web Application dialog box

Be sure to do the following:

1. Select template **MVC**.
2. **Add folders and core references for:**  **MVC**.
3. Leave **Add Unit Tests** unchecked (you can add tests later, and you're going to change the name of the associated project in the next step).

## Rename the New Project

In the Visual Studio **Solution Explorer** you'll see this:



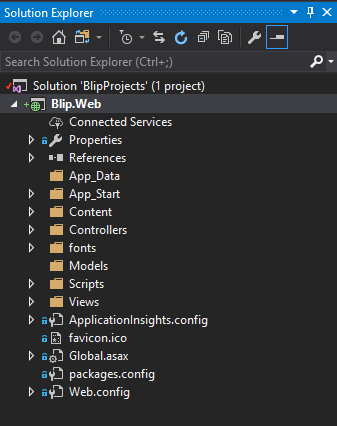
Visual Studio Solution Explorer: single project

*Note the following:*

* *The solution and the project have the same name*
* *The project is a .NET Framework web project*

Now rename the *project* to "**Blip.Web**".

The **Solution Explorer** should now look like this:



Visual Studio Solution Explorer: single project after renaming

*Note the following:*

* *The project is still a web application*
* *The project name is now different than the solution name*

However, the folder structure has not changed to reflect the new naming convention. To correct that:

1. Close Visual Studio.
2. Find your local **BlipProjects** *folder*.
3. In the **BlipProjects** *solution folder*, rename the **BlipProjects** *project folder* to **Blip.Web**.

Before: /BlipProjects/**BlipProjects**  
After: /BlipProjects/**Blip.Web**

1. Open the *solution file*, **BlipProjects.sln**, in a text editor and change the relative path of the **Blip.Web** *project* to "**BlipProjects\Blip.Web.csproj**". When completed, the file should look like this:

Microsoft Visual Studio Solution File, Format Version 12.00

# Visual Studio 15

VisualStudioVersion = 15.0.26730.8

MinimumVisualStudioVersion = 10.0.40219.1

Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") = "**Blip.Web**", "**BlipProjects\Blip.Web.csproj**", "{C822CC2D-7129-46F8-9CA0-A519046EB5A7}"

EndProject

Global

GlobalSection(SolutionConfigurationPlatforms) = preSolution

Debug|Any CPU = Debug|Any CPU

Release|Any CPU = Release|Any CPU

EndGlobalSection

GlobalSection(ProjectConfigurationPlatforms) = postSolution

{C822CC2D-7129-46F8-9CA0-A519046EB5A7}.Debug|Any CPU.ActiveCfg = Debug|Any CPU

{C822CC2D-7129-46F8-9CA0-A519046EB5A7}.Debug|Any CPU.Build.0 = Debug|Any CPU

{C822CC2D-7129-46F8-9CA0-A519046EB5A7}.Release|Any CPU.ActiveCfg = Release|Any CPU

{C822CC2D-7129-46F8-9CA0-A519046EB5A7}.Release|Any CPU.Build.0 = Release|Any CPU

EndGlobalSection

GlobalSection(SolutionProperties) = preSolution

HideSolutionNode = FALSE

EndGlobalSection

GlobalSection(ExtensibilityGlobals) = postSolution

SolutionGuid = {C5D1A472-8E6A-4033-91D9-4F080F9CBFCD}

EndGlobalSection

EndGlobal

1. Save the file, open Visual Studio, and reopen the solution.

## Adding Additional Projects to the Solution

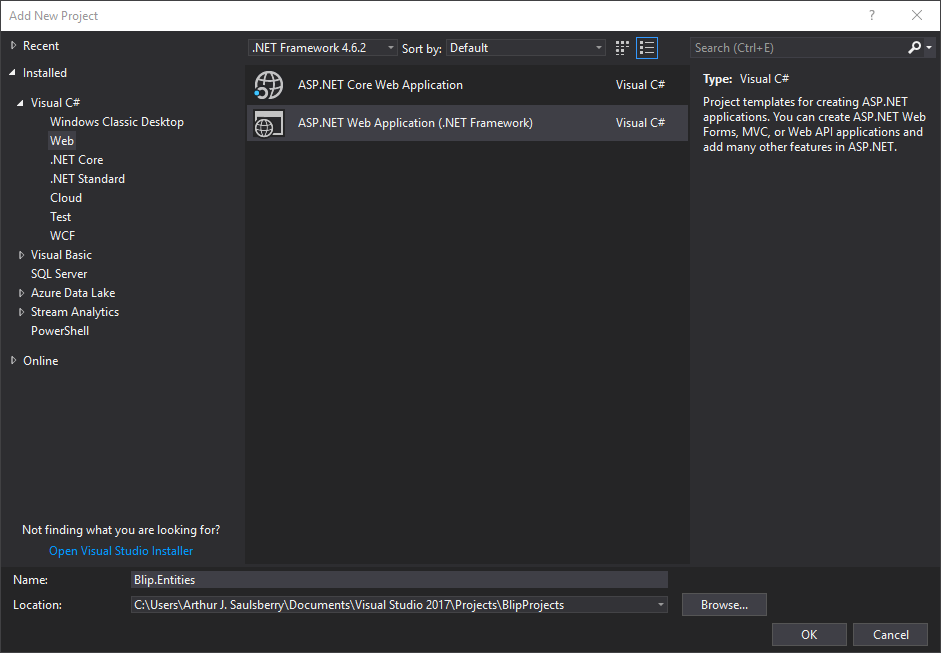
Once you have set up your solution and the initial project correctly, it's easy to add additional projects to the solution. The folder structure for the projects and the solution file (**.sln**) will accurately reflect the names of the new projects.

When you are building a multi-project web application, you only use the MVC Web Application template for the first project. You should create subsequent projects with the Empty template.

The process looks like this as we add the **Blip.Entities** *project* to the **BlipProjects** example solution:

1. Select the solution in the **Solution Explorer**.
2. Select **File** / **New** / **Project** (**Ctrl** + **Shift** + **N**).

You will be taken to a modal dialog box where you can choose the type of project:

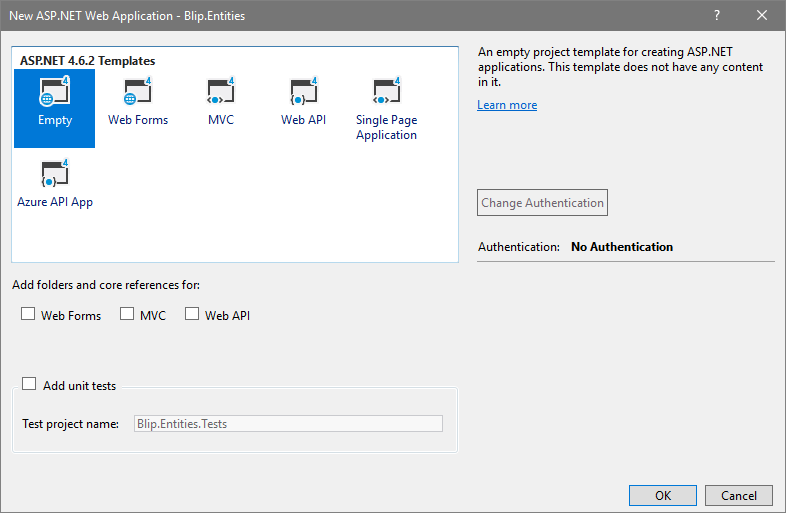


Visual Studio New ASP.NET Web Application dialog box

Do the following:

1. Select **ASP.NET Web Application (.NET Framework)**.
2. **Name:**  **Blip.Entities**.
3. Click **OK**.

You will be taken to the modal dialog box:



Visual Studio: New ASP.NET Web Application dialog box

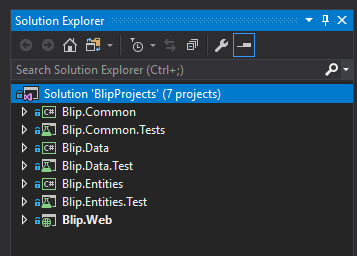
Do the following:

1. Select the **Empty** template.
2. Be sure none of the options for **Add folders and core references for** options are checked.
3. Optionally, add a test project at this time.

Repeat this process for the other projects in the example:

**Blip.Data**  
**Blip.Common**

When you are finished, the **Solution Explorer** should look like this if you have added test projects along with the empty .NET Framework projects:



Visual Studio Solution Explorer

Note that the project icons are different for empty .NET Framework projects.

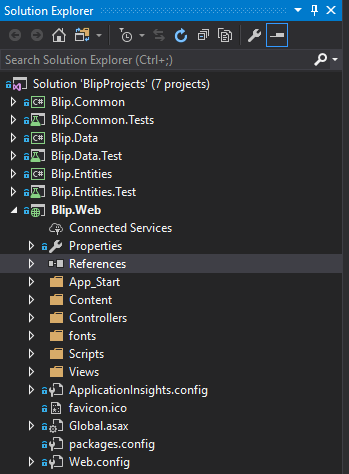
## Remove Unnecessary Folders from the Web Project, Blip.Web

Because some layers of the web application will be handled by other projects in the solution, the associated folders in the web project are unnecessary and we can get rid of them.

Delete the following folders from **Blip.Web**:

**App\_Data**  
**Entities**

The revised folder structure for **Blip.Web** should look like this:



Visual Studio Solution Explorer: Blip.Web folder structure

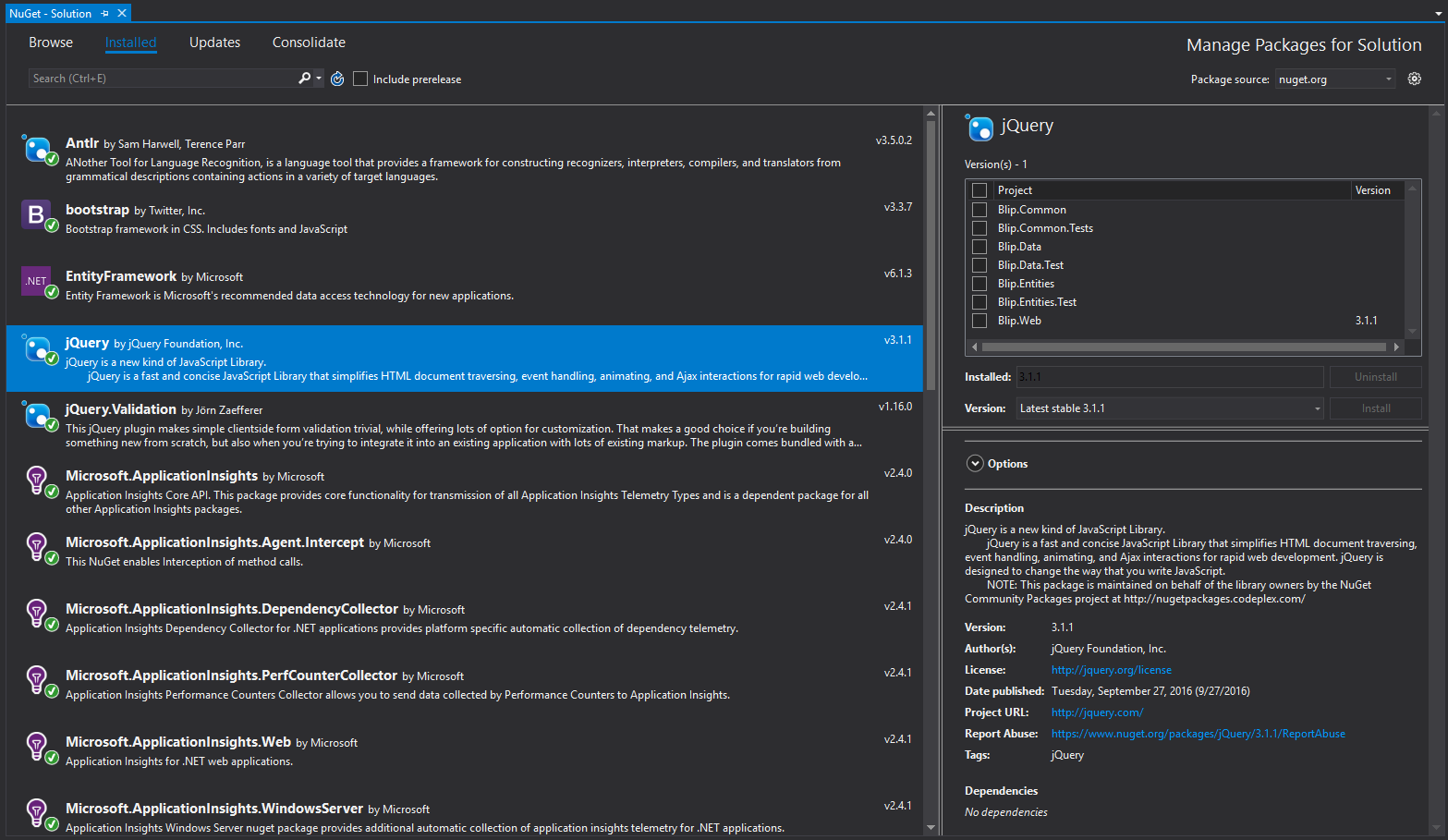
While working with multiple projects adds some complexity on the front end, it makes it easier to focus on the relevant code when developing each tier of the application; folders for other tiers are hidden when the **Solution Explorer** tree is rolled up for those projects.

## Update and Add NuGet Packages for the Solution

Once the projects are step up, it's a good time to update the NuGet packages that are part of each project template. At this point, we can also add the specific NuGet packages needed by the projects that provide the functionality for each level of our web application.

Managing NuGet packages at the solution level allows us to consolidate references for packages that are used by more than one project, helping to ensure we don't have multiple versions of a package in the same solution. That cuts down on interface and functionality confusion in development and reduces the deployment payload.

The solution level view of NuGet packages allows us to clearly see which projects have a given package installed and what version is installed in each package. For example, in the **BlipProjects** sample solution, we can see that jQuery is only installed in the web project:



Visual Studio NuGet package manager for BlipProjects solution

To see the package manager for the solution, be sure to highlight the solution in the Solution Explorer, then right-click and select **Manage NuGet packages for solution...** or select **Tools** / **NuGet Package Manager** / **Manage NuGet Packages for Solution...**

You can also see the package manager for just a project by highlighting the project in the **Solution Explorer** and following either of those steps.

#### Update NuGet Packages

If there are new releases of any of the packages in your solution, the count of updated packages will appear next to the **Updates** tab and the list will appear below. You can choose to update some or all of the packages in your solution.

The NuGet package manager will only update the instances of package associated with projects in which the package is already installed, so you don't have to worry about it installing updates in projects where the packages don't belong.

#### Add NuGet Packages

**Entity Framework** (EF) is used by the **BlipDrop** example project and will be used in our **BlipProject** solution. Since EF isn't part of the standard MVC template, we'll have to add it.

1. Open the **NuGet Package Manager** and select the **Browse** tab. Entity Framework should appear near the top of the list of packages. If it doesn't, be sure **Package source** is set to "**All**" or "**nuget.org**".
2. Highlight **Entity Framework**.
3. In the **right-hand pane** of the package manager window, check the following projects:
   * **Blip.Data**
   * **Blip.Web**
4. Click **Install**.

The package manager will chug through the installation process and present you with a couple of confirmation dialog boxes to accept.

*Although* ***Blip.Data*** *provides the data context for our solution and does the work of transferring data to the database through the repository classes we're going to set up later,* ***Blip.Web*** *still needs the EF libraries so that the state of data objects can be preserved between projects.*

**Microsoft.AspNet.Mvc** provides two of the types we are using in this solution, **SelectListItem** and **SelectList**. These classes need to be available in the projects that use them, so follow the process above to install this package in these projects:

* **Blip.Data**
* **Blip.Entities**
* **Blip.Web**

That takes care of the package management. You can close the **NuGet Package Manager** window.

## Blip.Entities Project

Having taken care of adding and updating the libraries we're going to use, we can begin putting together the code for our solution. Since the demonstration project is to convert a single project solution to a multi-project solution, we'll look at the differences between the two as we go along.

Let's start with the **Blip.Entities** projects, which contains the POCO's (plain-old class objects) that define the data entities for our solution.

### Entity Folders

We'll begin by adding folders to contain the class files.

1. Right-click on the **Blip.Entities** project.
2. Select **Add / New Folder** from the *context menu*.
3. Name the folder "**Customers**".

Repeat the process for the following folders:

* **Customer.ViewModels**
* **Geographies**

Note that you can use **dots** ("**.**") as separators in folder names. This follows the convention that Visual Studio uses by default to name unit test projects.

You can follow a number of different schemes for organizing entity classes. Since classes are generally defined on a one class per file basis, the number of files can be substantial in a large project. Some developers use the dot separator to provide levels of organization; others use a hierarchical folder structure.

Keep in mind that the folder structure and naming convention effect the default/automatic namespace structure for the classes. For example:

| **Folder** | **Namespace** |
| --- | --- |
| Blip.Entities\Customers | Blip.Entities.Customer |
| Blip.Entities\Customer.ViewModels | Blip.Entities.Customer.ViewModels |
| Blip.Entities\Customer.ViewModels\Partial | Blip.Entities.Customer.ViewModels.Partial |

Note that in the second example above the dot structure of the folder name provides a fourth tier of the namespace structure while keeping the folder structure flat under the project. Using a well-designed folder and namespace structure helps implement separation of concerns and limits the number and scope of classes referenced by **using** statements elsewhere in the solution. We'll see that in action as we move along.

### Entity Classes

Now we're ready to start adding code.

1. Right-click on the **Customer Folder**.
2. Select **Add... / Class...** from the *context menu*.
3. Name the class "**Customer**". Visual Studio will add the "**.cs**" extension whether or not you include it.

The result should look like this:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Blip.Entities.Customer  
{

class Customer

{

}

}

Note the namespace and default **using** statements.

The complete code for the **Customer** class is as follows:

using System;

using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema;

using Blip.Entities.Geographies;

namespace Blip.Entities.Customer

{

public class Customer

{

[Key]

[Column(Order = 1)]

public Guid CustomerID { get; set; }

[Required]

[MaxLength(128)]

public string CustomerName { get; set; }

[Required]

[MaxLength(3)]

public string CountryIso3 { get; set; }

[MaxLength(3)]

public string RegionCode { get; set; }

public virtual Country Country { get; set; }

public virtual Region Region { get; set; }

}

}

Note that the **using** statements have changed and the namespace reflects the folder structure we defined previously.

### Entities Namespace References

When you look at the **using** statements, you'll see that one of them references another namespace within the **Blip.Entities** project:

using Blip.Entities.Geographies;

Folders and folder names formed with **dot** ("**.**") separators form separate namespaces. Like the .NET Framework, higher tiers of the namespace structure are not supersets of the namespace; they do not contain the classes from the namespaces below.

For example, **Blip.Entities.Customer** does not contain **Blip.Entities.Customer.ViewModels**. We'll see this in action when we look at the controller methods in the **Blip.Web** project. Controller actions will have access to the view models, but not the entities themselves. This prevents an errant controller action from making changes directly to an entity and, thereby, directly to permanent storage. This can have important security benefits.

In the [single project solution](https://github.com/ajsaulsberry/BlipDrop) version of this project, the **Customer** class looks like this:

using System;

using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema;

namespace BlipDrop.Models

{

public class Customer

{

[Key]

[Column(Order = 1)]

public Guid CustomerID { get; set; }

[Required]

[MaxLength(128)]

public string CustomerName { get; set; }

[Required]

[MaxLength(3)]

public string CountryIso3 { get; set; }

[MaxLength(3)]

public string RegionCode { get; set; }

public virtual Country Country { get; set; }

public virtual Region Region { get; set; }

}

}

Since all of our entities in **BlipDrop** are in a single folder, **Models**, no **using** statements are necessary to reference the classes for **Country** and **Region**. This might be convenient for a small project, but it doesn't scale well and can open security holes.

#### Remaining Entity Classes

If you are following along with the **Blip.Projects** example solution and comparing it to the **BlipDrop** solution, you'll see we have the same entities, but they're arranged as follows:

| **Folder/Namespace** | **Class** |
| --- | --- |
| Customer | Customer |
| Customer.ViewModels | CustomersDisplayViewModel |
|  | CustomerEditViewModel |
| Geographies | Country |
|  | Region |

You can see that we're grouping related classes together; view models relating to the **Customer** object are in the same namespace and entities relating to geospatial information are grouped together in the **Blip.Entities.Geographies** namespace. The way you structure your folders, classes, and namespaces will be dependent on the requirements of your solution.

If you're following along with the conversion of **BlipDrop** to **BlipProjects**, continue creating classes and porting the code for the remaining entities and view models.

## Creating References to Other Projects

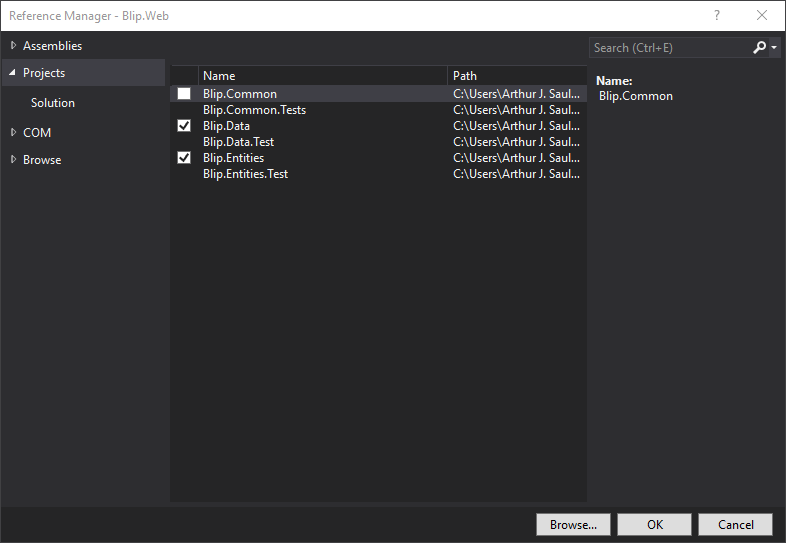
We've created our entities project, but how do we use those classes in other projects? The **using** statements for each class can namespaces from other projects, but to make the namespaces visible to a project we need to add references to the other projects.

### Adding Project References, Step-by-Step

This is how references are created between projects in a solution, using the **BlipProjects** solution as a guide:

1. In the **Solution Explorer**, expand the tree view underneath the **Blip.Web** project.
2. Right-click on the **References** *node*.
3. Select **Add References...** from the *context menu*. This will open the **Reference Manager** *modal dialog box*.
4. In the *left-hand pane* of the **Reference Manager** select **Projects**. You should see a list of projects in the solution.
5. Check the following projects:
   * **Blip.Data**
   * **Blip.Entities**

The result should look like this:



Visual Studio Reference Manager for Blip.Web

1. Click **OK**.

If you expand the *tree* view under the **References** *node* of **Blip.Web**, you should see the other projects among the list of references.

### Avoiding Circular References

It is **essential** to avoid circular references between projects. References only work in one direction. In our example, **Blip.Web** can reference (or "depend on") **Blip.Data**, but **Blip.Data** cannot also reference **Blip.Web**. If your code depends on a circular reference, there is a strong implication the separation of concerns in your project design is incomplete and you should refactor.

### Structural Considerations

"If our web project is exchanging data with the database through the repositories implemented in **Blip.Data**, why do we also need to reference ***Blip.Entities***?

Good question. As our solution is designed, **Blip.Entities** also provide the view models that are bound to the views defined in **Blip.Web** and served by its controller methods.

The view models are the return types for repository methods, so **Blip.Data** needs to know about them, as well as the entities themselves.

It's possible to structure the solution so that view models are in a separate project from the entities. Purists might suggest that is the approach providing the greatest separation of concerns since the web project would not need to reference the entities project, we would have to have a view model for every entity we might want to manipulate through the web interface. For example, if we intended to maintain the list of countries supported in our application with a web page, we'd need a view model for country.

"The line between pragmatism and over-engineering is thin."[*Sean Wildermuth*](https://app.pluralsight.com/profile/author/shawn-wildermuth)*, PluralSight author.*

What makes our Model-View-ViewModel (MVVM) solution design pragmatic is that both the base entities and the view models are accessible to the web tier, so it's possible to maintain data with web forms for CRUD actions using the entity's base class directly. In projects where there are a significant number of database tables containing references, or "look-up" data, this can eliminate a lot of repetitious code that doesn't serve a compelling purpose.

In other solutions, it may be important that every data interaction using the web interface passes through a view model between the controller action in the web project and the repository in the data layer project. In this scenario, only the data project would reference the entities project.

## Blip.Web Project

The controller actions and views are almost the same in the **Blip.Projects** multi-project solution and the **BlipDrop** single project solution. Rather than slog through the process of creating the individual classes, let's look just at where they differ.

### Controller Actions

Because the single project solution also used the repository pattern, the only changes are in the references to other projects and to the namespaces.

Each file below is identified by its full path starting at the solution level.

#### BlipDrop\BlipDrop\Controllers\CustomerController.cs

using System.Collections.Generic;

using System.Web.Mvc;

using BlipDrop.Data;

using BlipDrop.ViewModels;

namespace BlipDrop.Controllers

{

public class CustomerController : Controller

{

// GET: Customer

public ActionResult Index()

{

var repo = new CustomersRepository();

var customerList = repo.GetCustomers();

return View(customerList);

}

...

}

}

Note that the **using** statements refer to other namespaces within **BlipDrop**.

#### BlipProjects\Blip.Web\Controllers\CustomerController.cs

using System.Collections.Generic;

using System.Web.Mvc;

using Blip.Data;

using Blip.Entities.Customer.ViewModels;

namespace Blip.Web.Controllers

{

public class CustomerController : Controller

{

// GET: Customer

public ActionResult Index()

{

var repo = new CustomersRepository();

var customerList = repo.GetCustomers();

return View(customerList);

}

...

}

}

Note the following about the multi-project code shown above:

* The **using** statements refer to two external projects:
  1. **Blip.Data**
  2. **Blip.Entities**
* The reference to **Blip.Entities** only references the **Customer** view models; this controller does not need to know about the entities themselves nor any view models other than those associated with the **Customer** object.
* Instantiation and use of the repository, **CustomersRepository**, is unchanged.

#### BlipDrop\BlipDrop\Views\Customer\Create.cshtml

In the single project solution, the model is the **CustomerEditViewModel** found in the **ViewModels** folder.

@model BlipDrop.ViewModels.CustomerEditViewModel

@model Blip.Entities.Customer.ViewModels.CustomerEditViewModel

@{ ViewBag.Title = "Customer"; }

<h2>Customer</h2>

@using (Html.BeginForm())

{

@Html.AntiForgeryToken()

<div class="form-horizontal">

<h4>Create a new customer by entering the customer name, country, and region.</h4>

...

}

#### BlipProjects\Blip.Web\Views\Customer\Create.cshtml

In the multi-project solution, the view comes from the **Blip.Entities** project.

@model Blip.Entities.Customer.ViewModels.CustomerEditViewModel

@{ ViewBag.Title = "Customer"; }

<h2>Customer</h2>

@using (Html.BeginForm())

{

@Html.AntiForgeryToken()

<div class="form-horizontal">

<h4>Create a new customer by entering the customer name, country, and region.</h4>

...

}

Aside from the **@model** *directive*, the code for the two views is the same.

## Blip.Data Project

The data tier project contains a number of components that translate the class objects in **Blip.Entities** into relational database tables and handle create, retrieve, update, and delete (CRUD) operations on that data:

* The data context, **ApplicationDbContext** in our example.
* Entity Framework data migration configuration code, including the **Seed** method that enables us to populate the database with reference data when the database is created.
* Repository classes that exchange data between the data context and the business logic layer of the application.

We need to perform a number of actions to set up the database, seed it with data, and make it accessible to our web project.

### Add a Reference to the Entities Project

**Blip.Data** relies on the classes we defined in **Blip.Entities** so we need to add a corresponding reference.

Using the same process we used for adding references to **Blip.Web**, open the **Reference Manager** for **Blip.Data**, and add a reference to **Blip.Data**.

### Specifying Database Physical Location

We can put the physical database files in an **App\_Data** folder under the project folder. Alternately, if we don't specify a specific location for the files, Visual Studio will put databases created on the local SQL Server Express development server in the following directory:

**<boot drive>:\Users\<current username>**

If we use the **App\_Data** directory, we'll need to specify the path in the configuration files.

Accordingly, this is an optional step. Not doing it won't affect the operation of the solution.

*Note that it's not a good idea to add the physical database files to the project because Git won't be able to open them to write changes, leaving you unable to commit changes to your Git repository until you remove the database files from the project, which will require removing them from the directory in which they are located.*

### Creating Connection Strings

Our solution requires a *connection string* to identify the data store; in this case, a SQL Server database, associated with the **ApplicationDbContext** data context that will be used by Entity Framework and our repository methods. It is possible for projects to have more than one data context and more than one type of data store. Each requires a *connection string*.

Connection strings are the source of much anguish for developers. While this guide will endeavor to steer you around a number of connection string landmines, your development environment – and therefore the problems you encounter – may vary. The Other Resources section of this guide includes some valuable resources devoted to the murky depths of connection strings.

In our **Blip.Projects** example project, we need to have our *connection string* in two places and it has to be exactly the same in both locations:

| **Project** | **File** |
| --- | --- |
| Blip.Data | App.config |
| Blip.Web | Web.config |

These are XML format files. In both projects the **.config** files are found in the project root, below the folders.

**Important Note:**  If your data tier project (**Blip.Data** in our example) contains a **Web.config** file rather than an **App.config** file, see the Editing the Project Type section below.

For our example project, the connection string section is as follows:

<connectionStrings>

<add name="ApplicationDbContext" connectionString="Data Source=(localdb)\ProjectsV13;Initial Catalog=BlipData;Integrated Security=SSPI;AttachDBFilename=BlipData.mdf" providerName="System.Data.SqlClient" />

</connectionStrings>

Note the following important points to identify some of the ways your connection string might need to differ in order to work:

* If you are using a different version of Visual Studio or a database other than the development version of SQL Server Express (localdb) as your database, the **DataSource** portion of your connection string might be different.
* If you want to place the physical database files in a directory other than the default location, such as the **App\_Data** directory as discussed above, the **AttachDBFilename** portion of the string will need to include the fully qualified path to the directory. This path should exist prior to using the Entity Framework **Update-Database** command.

The **connectionStrings** *section* of the **.config** file is usually placed near the top of the file, within the **<configuration>** *element* (which defines the configuration file).

### Creating the Data Context

The *data context* for our solution is simple and straightforward. There are a few setup requirements and a few portions which need to be maintained manually as the development of the application progresses.

Here's the entire code, followed by notes on configuration and development:

#### Blip.Data\Context\Context.cs

using System.Data.Entity;

using System.Data.Entity.ModelConfiguration.Conventions;

using Blip.Entities.Customer;

using Blip.Entities.Geographies;

namespace Blip.Data

{

public class ApplicationDbContext : DbContext

{

public ApplicationDbContext() : base("ApplicationDbContext")

{

}

public DbSet<Country> Countries { get; set; }

public DbSet<Customer> Customers { get; set; }

public DbSet<Region> Regions { get; set; }

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

}

}

}

#### Setting up the Data Context

* Our public class, **ApplicationDbContext** (which can be called anything), inherits the **DbContext** class from **System.Data.Entity**, a namespace provided by Entity Framework.
* If we add statements to the **OnModelCreating** method, we're likely to use members of **System.Data.Entity.ModelConfiguration.Conventions**. Since we're not doing that in **Blip.Data**, this **using** state isn't required; it's here just to point it out.
* We need references to the namespaces for the entities associated with our application, in this case, **Blip.Entities.Customer** and **Blip.Entitites.Geographies**.
* The constructor needs the exact name of the *connection string* we previously created, "**ApplicationDbContext**".

#### Maintaining the Data Context

As you add entity classes to the Entities project of your solutions, you need to keep the list of entities in the context up to date. In the **BlipProjects** solution, we have three entities included in the context:

DbSet<Country> Countries

DbSet<Customer> Customers

DbSet<Region> Regions

Note the following:

* **DbSet** is a *collection class* provided by the Entity Framework namespace **System.Data.Entity**.
* The type of each **DbSet** collection is one of the classes from our entities project.
* In **BlipProjects** our convention is to pluralize the name of the **DbSet** and thereby the name of the database table, but this is not a requirement.
* View model objects like **CustomerEditViewModel** in **Blip.Entities** are not included because they aren't reflected in the database.

### Using Entity Framework Migrations

The next step in bringing our data project to life is enabling EF migrations and applying an initial migration to configure the database. This is done in the **Package Manager Console** (**PMC**) *window*.

#### Enabling EF Migrations

EF migrations have to be "turned on" in the project in which they'll be used after adding the EF NuGet package, as we did above.

1. Depending on your Visual Studio startup configuration, there may be a **PMC** *tab* at the bottom of the Visual Studio window. If there is a **Package Manager Console** *tab*, just click it open.

*If you don't see a* ***PMC*** *tab, select****View / Other Windows / Package Manager Console****from the menu bar. You should see a window like this in the lower third of your Visual Studio window:*

![Package Manager Console](vs-pkg-mgr-cons\_50pct.png)

1. Set **Default Project** to **Blip.Data**. This a hugely important setting you should make a point not to overlook. Visual Studio likes to default this value to the startup or web project, which will lead to errors. You are enabling migrations on the project containing the data context, so this value has to be set accordingly before running an Entity Framework command.
2. At the **command prompt** (**PM>**), type **enable-migrations** and press **Enter**.

Entity Framework should chug away for a while and then inform you that migrations have been enabled. A sign that the process has worked correctly is the presence of a **Migrations** folder under the root of the data project in **Solution Explorer** and a **Configuration.cs** file in that folder. We'll use that file in the next step.

#### Configuring EF Migrations

If you're following along in the sample solution, open the **Configuration.cs** file. It should look like this:

namespace Blip.Data.Migrations

{

// using System;

using System.Collections.Generic;

// using System.Data.Entity;

using System.Data.Entity.Migrations;

// using System.Linq;

using Blip.Entities.Geographies;

internal sealed class Configuration : DbMigrationsConfiguration<Blip.Data.ApplicationDbContext>

{

public Configuration()

{

AutomaticMigrationsEnabled = false;

}

protected override void Seed(Blip.Data.ApplicationDbContext context)

{

// This method will be called after migrating to the latest version. You can use

// the DbSet<T>.AddOrUpdate() helper extension method to avoid

// creating duplicate seed data.

var countries = new List<Country>

{

new Country {

Iso3 = "USA",

CountryNameEnglish = "United States of America"

},

new Country

{

Iso3 = "CAN",

CountryNameEnglish = "Canada"

},

new Country

{

Iso3 = "FRA",

CountryNameEnglish = "France"

}

};

countries.ForEach(s => context.Countries.AddOrUpdate(p => p.Iso3, s));

context.SaveChanges();

...

}

}

}

**Configuration Constructor**

The **Configuration** class is generated automatically when EF migrations are enabled. The constructor enables us to set a variety of configuration options for EF migrations. We're using one important configuration setting that you're likely to want to use in your projects:

**AutomaticMigrationsEnabled = false;**

When automatic migrations are turned on, the EF default, EF will create and apply a migration every time you make a change to your entities that will affect the structure of the database. Early in the development of an application, the entities may undergo a number of changes, leading to a rapid proliferation of migrations. This is to be avoided.

With automatic migrations turned off, we have the opportunity to decide when our modifications to the application's entities are applied to the database.

**Seed Method**

The **Seed** *method* enables us to add data to the database when it is created. This can be a huge time-saver in two common circumstances:

1. During **development**, when the database is often dropped and re-created repeatedly as changes are made to the design, the **Seed** method enables us to populate database tables with data we'll use during development.
2. During application **deployment**, the **Seed** method enables us to ensure the database is initialized with a canonical set of data necessary for the application to run.

In our example data project, **Blip.Data**, we're populating the database with lists of countries and regions.

A few notes about using the **Seed** method:

* The order of data loading is important. Because there is a parent-child relationship between countries and regions, we have to populate the **Countries** table first. If we don't we'll get referential integrity errors during execution of the T-SQL generated by the method.
* Syntax errors in the instantiation of instances of an entity may cause data to be loaded incorrectly or not at all. When loading a large number of records, it can be helpful to test the process with a few records before creating a long list. The **Seed** method also supports loading data from an external file.

#### Creating the Database with EF Code-first Migrations

Now that we've configured our EF migrations, we're ready to create the database. That's done with an initial migration.

Creating the Initial Migration

If you're following along with your own version of the **BlipProjects** sample project, follow these steps:

1. Set the **Default Project** field of the Package Manager Console window to: **Blip.Data** (or whatever you've called your data project).
2. Enter the following *command string* at the **Package Manager Console** *prompt* (**PM>**):

**add-migration InitialMigration**

1. Press **Enter**.

If the command executes correctly, it should create a new file in the **Migrations** directory of the data project (**Blip.Data**) named something like this:

`201708222251192\_InitialMigration.cs`

This file will include a method containing the instructions EF uses to create the database, along with another method containing instructions for reversing the create instructions. Each migration thereafter will have a comparable pair of methods.

Updating the Database

The initial migration will create the database if it doesn't already exist in the path specified in the **connectionString** associated with the data context. To apply the initial migration, do the following:

1. Ensure **Default Project** is still set to **Blip.Data**.
2. Enter the following command string at the **Package Manager Console** prompt (**PM>**):

**update-database -verbose -startupprojectname Blip.Data**

The **-verbose** *argument* will display the full output of the command as it executes.

Setting the **-StartupProjectName** *argument* ensures that the command takes effect on the correct project if the startup project is another project, like the web application (**Blip.Web** in the sample solution).

1. Press **Enter**.

If the process is executing correctly, the first few lines of output in the **PMC** *window* will look something like this:

PM> update-database -verbose -startupprojectname Blip.Data

Using StartUp project 'Blip.Data'.

Using NuGet project 'Blip.Data'.

Specify the '-Verbose' flag to view the SQL statements being applied to the target database.

Target database is: 'BlipData' (DataSource: (localdb)\ProjectsV13, Provider: System.Data.SqlClient, Origin: Configuration).

Applying explicit migrations: [201708222251192\_InitialMigration].

Applying explicit migration: 201708222251192\_InitialMigration.

CREATE TABLE [dbo].[Countries] (

[Iso3] [nvarchar](3) NOT NULL,

[CountryNameEnglish] [nvarchar](50) NOT NULL,

CONSTRAINT [PK\_dbo.Countries] PRIMARY KEY ([Iso3])

)

...

Note that the **Startup** *project* and **NuGet** *project* are both set to the name of the data project: **Blip.Data**. The output also identifies the database server and name. Following the configuration information is the T-SQL DDL generated from the methods in the migration file.

If the process completes successfully, you should see:

* The database has been created.
* The **Migrations** *table* in the database will contain a record with the name of the migration.

### Creating Repository Methods

Now that you have a database containing some data to test against, you can begin developing repositories. Repository classes typically contain methods for each of the create, update, retrieve, and delete functions associated with databases and other permanent data stores, depending on the type of data store.

Return types are usually individual objects or collections of objects based on view models or entities from the data project. They can also be types provided by any libraries referenced by the data project and, as necessary, the project invoking the repository method.

Using the **Blip.Data** example project, we'll take a look at a couple of repository methods associated with the **Customer** object.

Our repositories are organized in a folder and namespace structure corresponding to the entities we defined in **Blip.Data**, but many other potential organizational schemes are possible.

In the **Customer** *folder*, the **CustomerRepository.cs** *file* contains a method for getting a list of customers displayed on the **Index.cshtml** *view*. Here's the top part of that file, including this method:

using System;

using System.Collections.Generic;

using System.Data.Entity;

using System.Linq;

using Blip.Entities.Customer;

using Blip.Entities.Customer.ViewModels;

namespace Blip.Data

{

public class CustomersRepository

{

public List<CustomerDisplayViewModel> GetCustomers()

{

using (var context = new ApplicationDbContext())

{

List<Customer> customers = new List<Customer>();

customers = context.Customers.AsNoTracking()

.Include(x => x.Country)

.Include(x => x.Region)

.ToList();

if (customers != null)

{

List<CustomerDisplayViewModel> customersDisplay = new List<CustomerDisplayViewModel>();

foreach (var x in customers)

{

var customerDisplay = new CustomerDisplayViewModel()

{

CustomerID = x.CustomerID,

CustomerName = x.CustomerName,

CountryName = x.Country.CountryNameEnglish,

RegionName = x.Region.RegionNameEnglish

};

customersDisplay.Add(customerDisplay);

}

return customersDisplay;

}

return null;

}

}

...

}

}

Note the following:

* The entities and view models from the **Blip.Entities** *project* are referenced in the **using** *statements*.
* The **GetCustomers()** *method* returns a **List** *collection* of the view model object for the **Index.cshtml** *view*. The **Index()** *method* in the **CustomerController** in the **Blip.Web** *project* passes this collection to the view where the data is bound to the form.
* The *elements* of **List<CustomerDisplayViewModel>** are assembled by using the relationship between the **Customer** object and the **Country** and **Region** *objects* to get the display names for the view, rather than the index values.

Take a look at the repositories in the sample project to explore some other features of repositories. The Related PluralSight Training Classes section below has links to courses with more information about the repository pattern, including how to create repositories with Interfaces, which facilitates testing.

## Run and Debug

That's it! If you've followed along on your own to this point, you should be ready to rumble. The **BlipProjects** sample solution on GitHub is ready to run. (You may need to customize the connection string for your local environment, but the solution will create the database on the first run.)

Keep in mind that if you change your entity model by adding, removing, or modifying entities (not view models), you need to do the following before you can run and debug your solution:

1. Update the data context.
2. Add an EF code-first migration (`Add-Migration' MigrationName).
3. Update the database (**Update-Database**).

Be sure to set the ***Default Project*** in the Package Manager Console to the data project before running EF commands.

## About the Blip.Common and .Test Projects

The **Blip.Common** *project* includes a few examples of code that could be included in a "utility" project that could be included in multiple solutions and referenced by other projects that can use the functions. In the **BlipProjects** solution, it isn't referenced by the other projects.

The various **.Test** *projects* just exist to show the multi-project solution structure; they don't contain any tests. Writing unit tests is beyond the scope of this guide, but there are relevant PluralSight courses referenced below.

## Editing the Project Type

Visual Studio may, for reasons of its own, convert your blank .NET Framework projects to web template projects. When it does this, it renames the **App.config** *file* to **Web.config**. It also changes the project type GUID in the **.csproj** *file*. It does not add the additional NuGet packages associated with web template projects. It also does not change the contents of the **App.config** *file*, it just renames it.

This is the process for converting these projects back to basic .NET Framework projects:

1. In **Solution Explorer**, rename **Web.config** to **App.config**.
2. Exit Visual Studio.
3. In a text editor, open the **.csproj** *file* associated with the project.
4. Find the **ProjectTypeGuids** *element*, which looks like this:

**<ProjectTypeGuids>{349c5851-65df-11da-9384-00065b846f21};{fae04ec0-301f-11d3-bf4b-00c04f79efbc}</ProjectTypeGuids>**

1. Delete the first GUID and the semicolon after it, leaving **{fae04ec0-301f-11d3-bf4b-00c04f79efbc}** as the sole project type GUID.
2. Reopen the solution in Visual Studio. The project type icon should be a green "C#", not a green globe.

Experience shows that once you've converted a project back, Visual Studio does not seem to want to convert them again.

## More Information

If you want to dive deeper into the topics discussed in this guide, the following is a selected list of resources.

### Related PluralSight Training Classes

[PluralSight](https://www.pluralsight.com/) offers a number of courses on the topics mentioned in this guide. The following are some suggestions organized by technology:

| **Technology** | **Course(s)** |
| --- | --- |
| ASP.NET, EF, View Models | [Best Practices in ASP.NET: Entities, Validation, and View Models](https://app.pluralsight.com/library/courses/aspdotnet-bestpractices-models/table-of-contents) |
| ASP.NET | [Building Applications with ASP.NET MVC 4](https://app.pluralsight.com/library/courses/mvc4-building/table-of-contents) |
| C# | [C# Fundamentals with Visual Studio 2015](https://app.pluralsight.com/library/courses/c-sharp-fundamentals-with-visual-studio-2015/table-of-contents) |
| Entity Framework | [Getting Started with Entity Framework 6](https://app.pluralsight.com/library/courses/entity-framework-6-getting-started/table-of-contents) |
| MVC | [ASP.NET MVC 5 Fundamentals](https://app.pluralsight.com/library/courses/aspdotnet-mvc5-fundamentals/table-of-contents) |

### Case Study Code on GitHub

The complete Visual Studio solutions described in this guide are available on GitHub:

| **Project** | **Repo** |
| --- | --- |
| BlipDrop | <https://github.com/ajsaulsberry/BlipDrop> |
| BlipProjects | <https://github.com/ajsaulsberry/BlipProjects> |

You can fork the projects, run the code, and experiment on your own.

Note that the sample project is not intended to be a real-life case study or production code; it exists to illustrate the topics covered in this guide.

### Other Resources

*Disclaimer: Pluralsight and the author of this Guide are not responsible for the content, accuracy, or availability of 3rd party resources.*

Here are some recommended resources for more information:

[ConnectionStrings.com](https://www.connectionstrings.com/)  
This website provides a wealth of information on connection strings for MS SQL Server and many other data stores, including SQL Azure, Oracle, and MySQL. The .NET Framework Data Provider for SQL Server [has its own page](https://www.connectionstrings.com/sqlconnection/).

[Microsoft Entity Framework Documentation](https://msdn.microsoft.com/en-us/library/ee712907(v=vs.113).aspx)  
This page is the jumping off point for a variety of resources on EF.

[Microsoft: The Repository Pattern](https://msdn.microsoft.com/en-us/library/ff649690.aspx)  
This page provides a summary of the benefits of the repository pattern.