<https://programmingwithmosh.com/csharp/should-you-split-your-asp-net-mvc-project-into-multiple-projects/>

 April 18th, 2017

[Comments](https://programmingwithmosh.com/asp-net/layered-architecture/#disqus_thread)

# [Layered Architecture in ASP.NET Core Applications](https://programmingwithmosh.com/asp-net/layered-architecture/)

One of the viewers of my [YouTube channel](https://www.youtube.com/user/programmingwithmosh) asked me an interesting question. He mentioned in a typical layered architecture, he sees ASP.NET MVC building blocks (Controller, View, and Model) as part of the presentation layer. These days, however, a lot of modern applications are built with Angular/React on the client and ASP.NET Core (Web API) on the server. So, what is the presentation layer in this kind of architecture? Let’s see!

With this stack, we have the following layers:

* Presentation
* Service
* Business Logic/Application Core
* Data Access/Persistence

## Presentation Layer

Your Angular components, their templates, and the models you define in your Angular app are all presentation layer artifacts.

## Service Layer

The confusing thing about this layer is that the term “service” is overloaded and it means different things to different people. In the context of a layered architecture, it wraps an application and exposes the application functionality in terms of a simple API that the user interface can talk to. This is the classic definition. Think of it as the glue between the presentation and business logic layers.

Now, in our modern stack, our logical service layer is physically composed of two parts: one part is on the client (Angular HTTP services) and the other part is on the server (ASP.NET Core controllers). These Angular services and ASP.NET Core controllers are very cohesive. The methods on these services (eg CourseService.getCourses()) talk directly to the endpoints exposed by your ASP.NET Core controllers.

## Business Logic Layer

In your ASP.NET Core controllers, you often use repository interfaces (**ICourseRepository**), domain classes (**Course**) and services (**PhotoService**). All these are part of the business logic layer. They represent the core of an application irrespective of any presentation or persistence frameworks.

Note that here I’m talking about repository interfaces and not their implementations. These implementations are part of the data access/persistence layer.

Also, note that the services we have here are responsible for orchestration. For example, when adding a photo to a course, first, you need to store that photo in the file system (or some other kind of storage), and then you need to add it to the database (using a repository interface). Here is an example:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17 | // Store the file first  var uploadsPath = Path.Combine(host.WebRoot, "uploads");  if (!Directory.Exists(uploadsPath))      Directory.CreateDirectory(uploadsPath);    var fileName = Guid.NewGuid().ToString() + Path.GetExtension(file.FileName);  var filePath = Path.Combine(uploadsPath, fileName);    using (var stream = new FileStream(filePath, FileMode.Create))  {      file.Copyto(stream);  }    // Add a record to the database  var photo = new Photo { FileName = fileName };  repository.Add(photo);  unitOfWork.Complete(); |

You wouldn’t write all this logic inside an ASP.NET Core Controller. Imagine, tomorrow you decide to use a different framework. You want to re-use as much code as possible. By encapsulating this code in a service (**PhotoService.AddPhoto()**), you can switch from ASP.NET Core to a different framework with less effort.

### But wait for a second…

Now, that strongly-opinionated developer comes and says: “But who does replace ASP.NET Core with something else? How often does that happen?” Let’s say never! By moving all this logic from a controller into a service, you put the responsibility where it really belongs. The result is cleaner, slimmer, easier to read and easier to test controllers.

Imagine a restaurant where the chef does it all. He’s at the door, welcoming guests, giving them a table, taking their order, then going in the kitchen, chopping the vegetables, cooking, washing the dishes, then coming out and giving the bill to the guests. Would you go to that restaurant? I hope not!

In a good and organized restaurant, there are a few people each focusing on only one job. The waiters/waitresses are purely responsible for welcoming the guests and giving them the bill. The chef is purely responsible for cooking. He or she doesn’t wash the dishes! By the same token, you should have classes that do only one thing and do it well. This is what we call **separation of concerns**. You should put the responsibility where it really belongs, even if you’re never going to change the presentation or persistence framework of your application.

So, once again, all your domain classes (**Course**), repository interfaces (**ICourseRepository**) and application services (**PhotoService**) are part of the business logic layer. They represent the core of your application completely decoupled from any presentation and persistence frameworks. This is what Uncle Bob defines as [Clean Architecture](https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html).

## Data Access Layer

This layer is all about persistence. Here we have implementations tightly coupled to Entity Framework (or other frameworks) for persisting and retrieving data. If you’re using Entity Framework, your **DbContext** belongs in this layer. So do **UnitofWork** and **Repository**implementations.

## Splitting a Project

Now, a common (and bad) practice I’ve seen some developers do, is that they blindly split an ASP.NET project into multiple class libraries, one for each layer. And with this, they assume just because they have a class library called **MyProject.BLL**or **MyProject.DAL**, they have properly layered their application. But that’s not necessarily right.

What matters is the direction of dependency and coupling between classes, not folders or projects. You can easily organize your classes into folders and projects but these classes can be poorly coupled to each other, which results in spaghetti architecture. Read my blog post on the topic:

[Should you split your ASP.NET MVC/Core projects?](https://programmingwithmosh.com/csharp/should-you-split-your-asp-net-mvc-project-into-multiple-projects/)

 April 18th, 2017

[Comments](https://programmingwithmosh.com/asp-net/layered-architecture/#disqus_thread)

# [Layered Architecture in ASP.NET Core Applications](https://programmingwithmosh.com/asp-net/layered-architecture/)

One of the viewers of my [YouTube channel](https://www.youtube.com/user/programmingwithmosh) asked me an interesting question. He mentioned in a typical layered architecture, he sees ASP.NET MVC building blocks (Controller, View, and Model) as part of the presentation layer. These days, however, a lot of modern applications are built with Angular/React on the client and ASP.NET Core (Web API) on the server. So, what is the presentation layer in this kind of architecture? Let’s see!

With this stack, we have the following layers:

* Presentation
* Service
* Business Logic/Application Core
* Data Access/Persistence

## Presentation Layer

Your Angular components, their templates, and the models you define in your Angular app are all presentation layer artifacts.

## Service Layer

The confusing thing about this layer is that the term “service” is overloaded and it means different things to different people. In the context of a layered architecture, it wraps an application and exposes the application functionality in terms of a simple API that the user interface can talk to. This is the classic definition. Think of it as the glue between the presentation and business logic layers.

Now, in our modern stack, our logical service layer is physically composed of two parts: one part is on the client (Angular HTTP services) and the other part is on the server (ASP.NET Core controllers). These Angular services and ASP.NET Core controllers are very cohesive. The methods on these services (eg CourseService.getCourses()) talk directly to the endpoints exposed by your ASP.NET Core controllers.

## Business Logic Layer

In your ASP.NET Core controllers, you often use repository interfaces (**ICourseRepository**), domain classes (**Course**) and services (**PhotoService**). All these are part of the business logic layer. They represent the core of an application irrespective of any presentation or persistence frameworks.

Note that here I’m talking about repository interfaces and not their implementations. These implementations are part of the data access/persistence layer.

Also, note that the services we have here are responsible for orchestration. For example, when adding a photo to a course, first, you need to store that photo in the file system (or some other kind of storage), and then you need to add it to the database (using a repository interface). Here is an example:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17 | // Store the file first  var uploadsPath = Path.Combine(host.WebRoot, "uploads");  if (!Directory.Exists(uploadsPath))      Directory.CreateDirectory(uploadsPath);    var fileName = Guid.NewGuid().ToString() + Path.GetExtension(file.FileName);  var filePath = Path.Combine(uploadsPath, fileName);    using (var stream = new FileStream(filePath, FileMode.Create))  {      file.Copyto(stream);  }    // Add a record to the database  var photo = new Photo { FileName = fileName };  repository.Add(photo);  unitOfWork.Complete(); |

You wouldn’t write all this logic inside an ASP.NET Core Controller. Imagine, tomorrow you decide to use a different framework. You want to re-use as much code as possible. By encapsulating this code in a service (**PhotoService.AddPhoto()**), you can switch from ASP.NET Core to a different framework with less effort.

### But wait for a second…

Now, that strongly-opinionated developer comes and says: “But who does replace ASP.NET Core with something else? How often does that happen?” Let’s say never! By moving all this logic from a controller into a service, you put the responsibility where it really belongs. The result is cleaner, slimmer, easier to read and easier to test controllers.

Imagine a restaurant where the chef does it all. He’s at the door, welcoming guests, giving them a table, taking their order, then going in the kitchen, chopping the vegetables, cooking, washing the dishes, then coming out and giving the bill to the guests. Would you go to that restaurant? I hope not!

In a good and organized restaurant, there are a few people each focusing on only one job. The waiters/waitresses are purely responsible for welcoming the guests and giving them the bill. The chef is purely responsible for cooking. He or she doesn’t wash the dishes! By the same token, you should have classes that do only one thing and do it well. This is what we call **separation of concerns**. You should put the responsibility where it really belongs, even if you’re never going to change the presentation or persistence framework of your application.

So, once again, all your domain classes (**Course**), repository interfaces (**ICourseRepository**) and application services (**PhotoService**) are part of the business logic layer. They represent the core of your application completely decoupled from any presentation and persistence frameworks. This is what Uncle Bob defines as [Clean Architecture](https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html).

## Data Access Layer

This layer is all about persistence. Here we have implementations tightly coupled to Entity Framework (or other frameworks) for persisting and retrieving data. If you’re using Entity Framework, your **DbContext** belongs in this layer. So do **UnitofWork** and **Repository**implementations.

## Splitting a Project

Now, a common (and bad) practice I’ve seen some developers do, is that they blindly split an ASP.NET project into multiple class libraries, one for each layer. And with this, they assume just because they have a class library called **MyProject.BLL**or **MyProject.DAL**, they have properly layered their application. But that’s not necessarily right.

What matters is the direction of dependency and coupling between classes, not folders or projects. You can easily organize your classes into folders and projects but these classes can be poorly coupled to each other, which results in spaghetti architecture. Read my blog post on the topic:

[Should you split your ASP.NET MVC/Core projects?](https://programmingwithmosh.com/csharp/should-you-split-your-asp-net-mvc-project-into-multiple-projects/)