ASP.NET MVC — Detailed Guide (for understanding & building apps)

This document explains the **Model–View–Controller (MVC)** pattern in the context of both **ASP.NET MVC (classic)** and **ASP.NET Core MVC**. It contains conceptual explanations, code examples, configuration recipes, common patterns, and testing notes tailored for .NET C# developers.

> Note: This guide treats ASP.NET Core MVC as the modern recommended platform for new development. Many concepts are the same across classic ASP.NET MVC and ASP.NET Core MVC, but configuration and hosting differ.

Table of contents

---

1. MVC — high-level overview

- **Model:** the domain/data layer — POCO classes, DTOs, ViewModels, and data access logic.
- **View:** presentation layer — Razor views (.cshtml), partials, and client-side assets.
- **Controller:** orchestrates input → model updates → view rendering. Controllers expose action methods that map to HTTP endpoints.

Why MVC? - Clear separation of concerns, testability, and friendly mapping to HTTP semantics (GET/POST/PUT/DELETE).

---

2. Project anatomy (ASP.NET Core MVC)

Typical project structure:

```
/Controllers
  HomeController.cs
/Models
  Product.cs
  ProductViewModel.cs
/Views
  /Home
    Index.cshtml
  /Shared
    _Layout.cshtml
/wwwroot
  /css /js /images
appsettings.json
Program.cs
Startup.cs (optional)
```

Key files: - `Program.cs` / `Startup.cs` — Configure host, services, middleware, and routing. - `appsettings.json` — Configuration. - `wwwroot` — static assets.

---

3. Models — design, binding, validation

- **POCO models**: keep them simple. Use separate ViewModel classes when the shape sent to the view differs from the persistence model.
- **Model binding**: ASP.NET Core automatically binds request data (route, query string, form, JSON body) to action parameters or model properties.
- **Validation**: use DataAnnotations (`[Required]`, `[StringLength]`, `[Range]`) and `IValidatableObject` or custom validators. Controllers check `ModelState.IsValid`.

Example model with validation:

```
public class ProductViewModel
{
    public int Id { get; set; }

    [Required]
    [StringLength(100)]
    public string Name { get; set; }

    [Range(0.01, 10000)]
```

```
    public decimal Price { get; set; }
}
```

---

4. Views — Razor syntax, partials, view components

- Views are .cshtml files using Razor: C# interleaved with HTML.
- Use `_Layout.cshtml` in `/Views/Shared` for consistent shell.
- Partial views (`Partial`) for reuse; View Components for more complex, reusable UI with logic.

Simple Razor view example:

```
@model ProductViewModel
@{
    ViewData["Title"] = "Product";
}
<h1>@Model.Name</h1>
<p>Price: @Model.Price</p>
```

---

5. Controllers — actions, routing, model binding

- Controllers derive from `Controller` or `ControllerBase`.
- Action methods return `IActionResult` (View, Redirect, Json, NotFound, etc.).
- Model binding maps route/query/form/body --> parameters.

Example controller:

```
public class ProductsController : Controller
{
    private readonly IProductService _svc;
    public ProductsController(IProductService svc) { _svc = svc; }

    public IActionResult Index() => View(_svc.GetAll());

    public IActionResult Details(int id)
    {
        var m = _svc.GetById(id);
        if (m == null) return NotFound();
        return View(m);
    }

    [HttpPost]
    public IActionResult Create(ProductViewModel vm)
```

```
    {
        if (!ModelState.IsValid) return View(vm);
        _svc.Create(vm);
        return RedirectToAction("Index");
    }
}
```

---

6. Routing & endpoints

- Classic: attribute routing (`[Route("products/{id}")]`) or conventional routing configured in `UseEndpoints`.
- ASP.NET Core default route: `"{controller=Home}/{action=Index}/{id?}"`.

Examples:

```
// Program.cs / Startup.cs
app.UseRouting();
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

---

7. Middleware, filters, and cross-cutting concerns

- **Middleware** (in the pipeline) handles cross-cutting concerns like static files, routing, authentication, error handling.
- **Filters** (MVC layer) provide action-level hooks: `Authorize`, `ExceptionFilter`, `ActionFilter`, `ResultFilter`.

Use middleware for global concerns; filters for MVC-specific features.

---

8. Dependency Injection & services

- ASP.NET Core has first-class DI. Register services in `Program.cs` / `Startup.ConfigureServices`:

```
builder.Services.AddScoped<IProductService, ProductService>();
builder.Services.AddDbContext<AppDbContext>(options => ...);
```

- Controllers receive services via constructor injection.

---

9. Data access (EF Core) and repositories

- Use `DbContext` with `DbSet<T>` for persistence.
- Consider layering: controllers → services → repositories → DbContext.
- Use migrations (`dotnet ef migrations add`) to manage schema.

Example minimal `DbContext` and usage:

```
public class AppDbContext : DbContext
{
    public DbSet<Product> Products { get; set; }
}

public class ProductService : IProductService
{
    private readonly AppDbContext _db;
    public ProductService(AppDbContext db) { _db = db; }
    public IEnumerable<Product> GetAll() => _db.Products.ToList();
}
```

---

10. Authentication & Authorization

- Use `AddAuthentication` and `AddAuthorization` to configure schemes and policies.
- Protect controllers/actions with `[Authorize]` and allow anonymously with `[AllowAnonymous]`.
- For cookie, JWT, and external providers (OAuth), configure corresponding middleware.

---

11. Areas, tag helpers, and view components

- **Areas**: organize large apps into logical sections (Admin, Identity). Create `Areas/Admin/Controllers`, `Areas/Admin/Views`.
- **Tag Helpers**: server-side helpers for generating HTML (anchor, form, input) with C# attributes.
- **View Components**: reusable chunks of UI + server logic (like partials with DI).

---

12. Testing strategies (unit, integration, functional)

- **Unit tests**: test controllers/services in isolation; mock dependencies (use Moq).
- **Integration tests**: use `WebApplicationFactory<TEntryPoint>` to test endpoints and middleware.

- **UI/functional tests**: Selenium or Playwright for end-to-end browser tests.

Example unit testing controller with xUnit + Moq:

```
var mockSvc = new Mock<IProductService>();
mockSvc.Setup(s => s.GetAll()).Returns(new List<Product>());
var ctrl = new ProductsController(mockSvc.Object);
var result = ctrl.Index() as ViewResult;
Assert.NotNull(result);
```

---

13. Common patterns & best practices

- Use **ViewModels** to avoid passing domain entities to views.
- Keep controllers thin — orchestrate work in services.
- Validate inputs and use `ModelState`.
- Protect against over-posting by binding only allowed properties or using view models.
- Use logging and structured logs (ILogger).
- Properly configure CORS when APIs are consumed from other origins.

---

14. Migration notes (MVC 5 → ASP.NET Core MVC)

- ASP.NET Core is cross-platform, has built-in DI, and a different hosting/startup model.
- Web.config is replaced by `appsettings.json` and environment variables.
- Some APIs moved or renamed; port controllers and views but rework startup, DI, and middleware.

---

15. Appendix: Minimal example

Program.cs (ASP.NET Core 7+ minimal host)

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();
// Add DB, services, etc.

var app = builder.Build();
if (!app.Environment.IsDevelopment()) app.UseExceptionHandler("/Home/Error");
app.UseStaticFiles();
app.UseRouting();
app.UseAuthorization();

app.MapControllerRoute(
```

```
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();
```

HomeController.cs

```
public class HomeController : Controller
{
    public IActionResult Index() => View();
    public IActionResult About() => View();
}
```

Index.cshtml

```
@{
    ViewData["Title"] = "Home";
}
<h1>Welcome</h1>
```

Further reading & official docs

- The official Microsoft documentation contains extensive tutorials and references for both classic ASP.NET MVC and ASP.NET Core MVC.

If you want, I can now: - Generate a **one-page quick reference** (cheat sheet) for controllers, routing, and attributes. - Create a **sample ASP.NET Core MVC project** (full files) you can drop into VS / VS Code. - Produce **unit & integration test templates** wired to the sample project.

Tell me which you want next.