Injecting and Resolving Dependencies

Steve Gordon

.NET Engineer and Microsoft MVP

@stevejgordon www.stevejgordon.co.uk

Overview

Constructor injection

Action injection

Middleware injection

Razor view injection

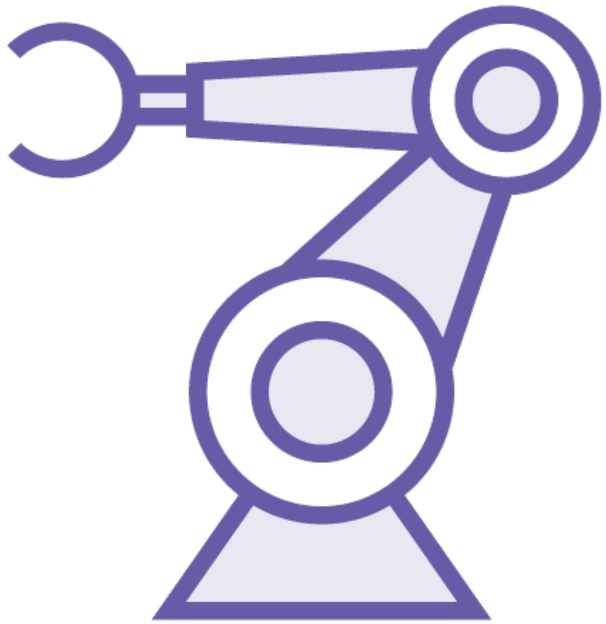
Minimal API handler injection

Background/hosted service injection

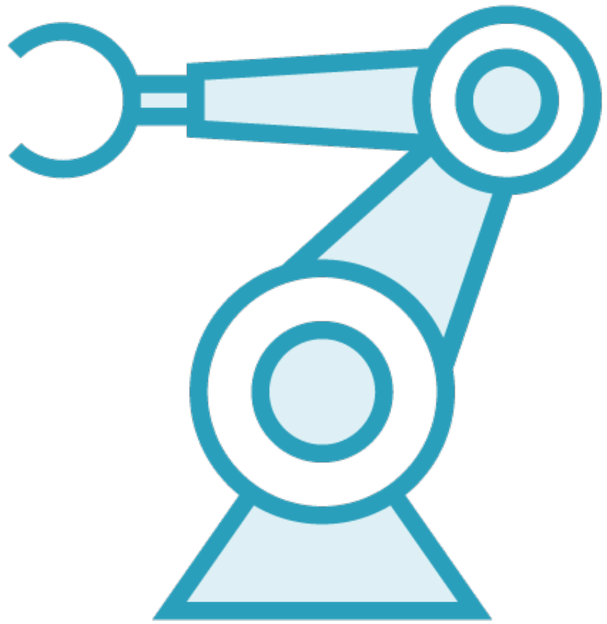
- Manual scope creation

Service Resolution Mechanisms

Service Resolution Mechanisms

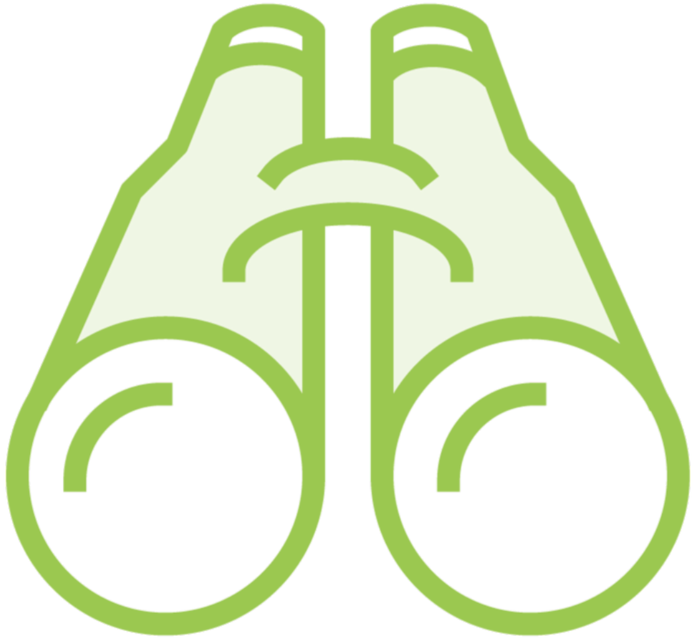


IServiceProvider



ActivatorUtilities

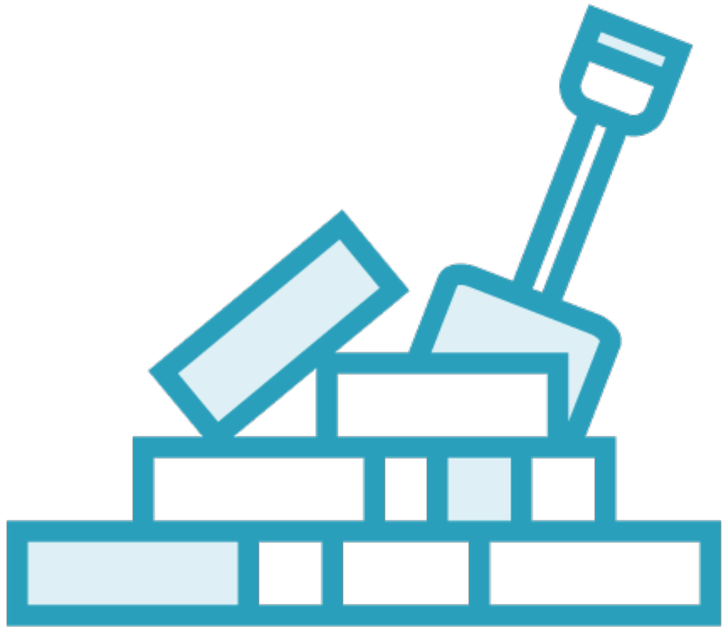
Service Provider Resolution



Services and their dependencies must be registered with the D.I. container

When creating an instance of a service from the container, its dependencies will also be resolved from the container

Activator Utilities



Can create objects that are not registered directly into the D.I. container

Creates an object via its constructor

Arguments can be supplied directly or resolved from an IServiceProvider

Used to activate framework components

- e.g. Controllers, tag helpers, model binders

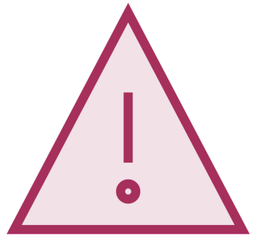
Should not be called by application code

Constructor Injection

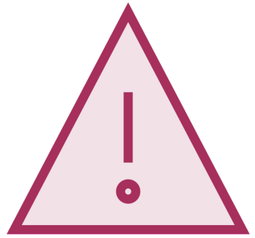
Constructor Injection

- Controllers
- Razor page models
- ViewComponents
- TagHelpers
- Filters
- Middleware
- Application classes

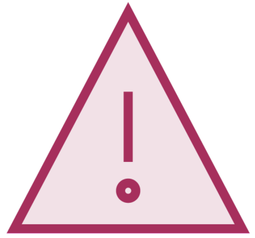
Constructor Rules



Assign default values for arguments not provided by the container



When services are resolved, a public constructor is required



Only a single applicable constructor can exist for services resolved via ActivatorUtilities (framework components such as controllers)

Services resolved from the container can support multiple applicable constructors.

IServiceProvider Constructor Selection

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddSingleton<MyService>();
   services.AddSingleton<AService>();
   services.AddSingleton<AnotherService>();
}

public class MyService
{
    public MyService(AService aService)
    {
    }

    public MyService(AService aService, AnotherService anotherService)
    {
    }
}
```

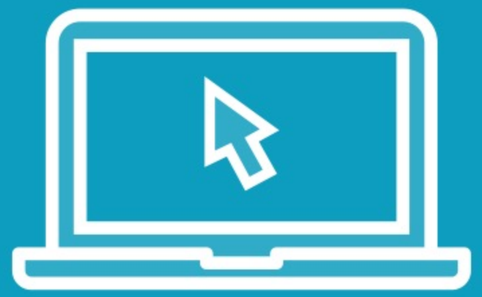
IServiceProvider Constructor Selection

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddSingleton<MyService>();
   services.AddSingleton<AService>();
}

public class MyService
{
    public MyService(AService aService)
    {
    }

    public MyService(AService aService, AnotherService anotherService)
    {
    }
}
```

Demo



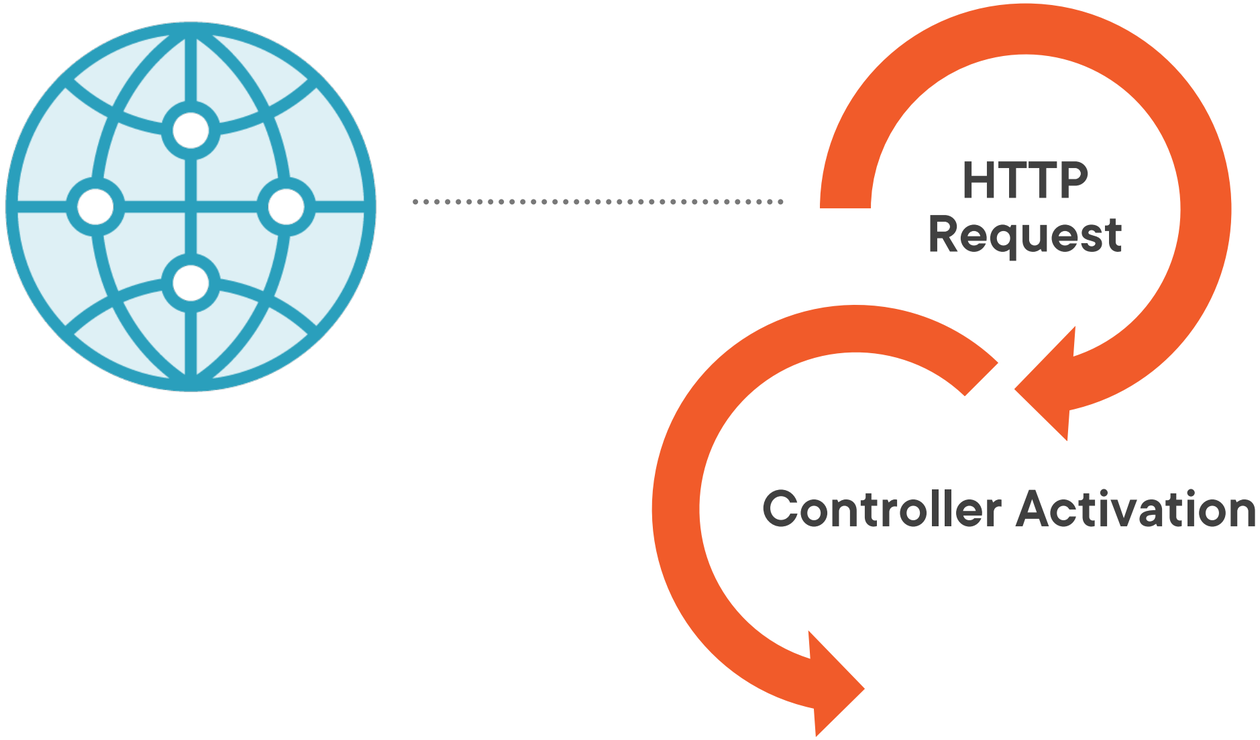
Injecting dependencies into controllers

Injecting into action methods

ASP.NET Core activates (creates) a new controller instance per request.

admin/bookings/upcoming

admin/bookings/upcoming



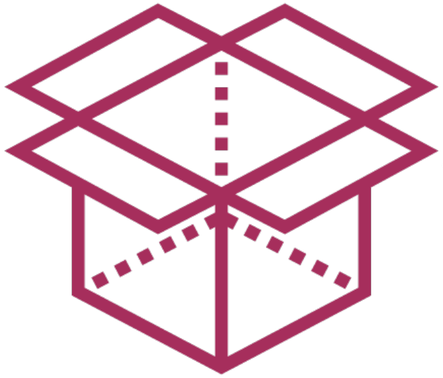
admin/bookings/upcoming

HTTP Request

Controller Activation

Resolve Services

Dependency Injection Container



new ICourtMaintenanceService()

Action Injection

Choosing Action Injection

Dependant on how widely used a dependency is within a controller

Constructor injector is most common

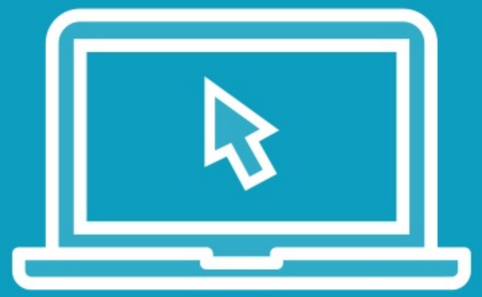
Action injection may be more efficient if a dependency is only needed by a single action

Such cases may indicate that a controller has too many responsibilities

- Consider splitting actions across more focused controllers

Dependency Injection with Minimal APIs

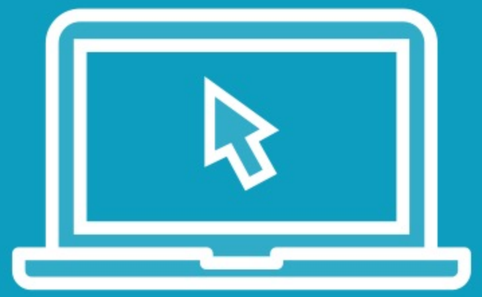
Demo



Create an ASP.NET Core minimal API to return weather forecasts

- Injecting dependencies into minimal API handlers

Demo



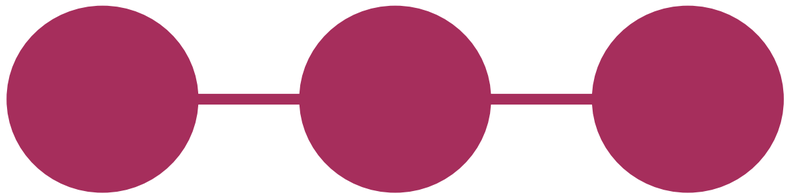
Injecting services into middleware

Understand the behavior of each option

Choosing the correct point of injection for dependencies

Middleware activation differs from framework components such as controllers.

Middleware Activation



Each middleware component is constructed once when the application starts

- Constructor dependencies are therefore resolved from the root container
- Singletons within the application

Constructor dependencies are captured for the life of the application

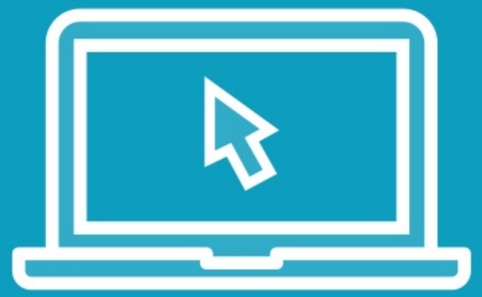
Avoid injecting scoped or transient services via constructor injection

The Invoke/InvokeAsync method is invoked once per request. Parameters are resolved from the request scope.

Comparing Middleware Dependency Injection

Constructor	Invoke/InvokeAsync
Called once when the application starts	Runs once per request
Supports only singleton services	Services are resolved from the request scope
Scoped or transient services will be captured and may not behave correctly	Supports all service lifetimes

Demo

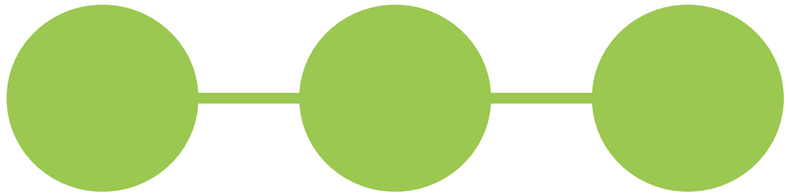


Injecting services into factory-based middleware

Understand the difference between the two middleware approaches

- Affects on registering and resolving dependencies

Factory-based Middleware



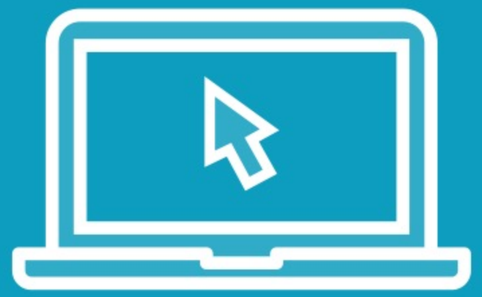
Implement the IMiddleware interface

Resolved by an IMiddlewareFactory on a per-request basis

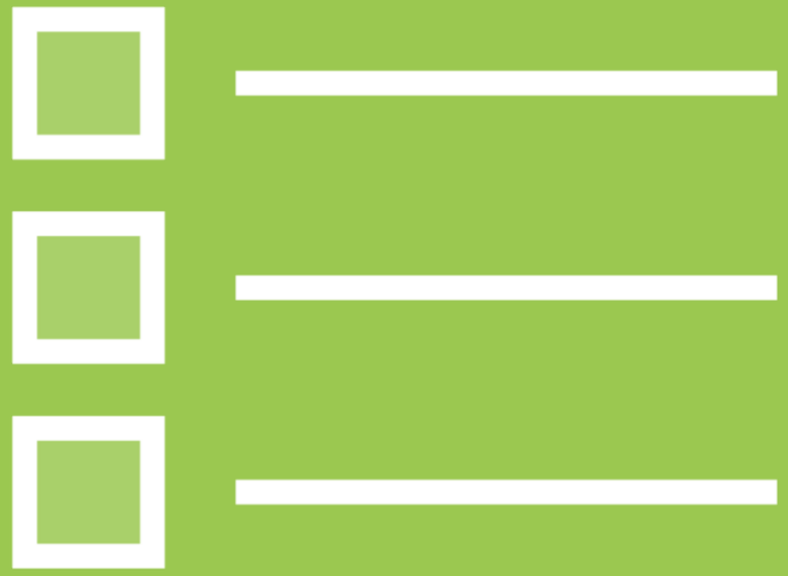
Transient and scoped services may be injected via their constructors

Rarely used in most applications

Demo



Injecting dependencies into Razor views



Requirement

Improve the user experience by showing the maximum court booking lengths on the find available courts page.

View Injection Considerations

Take care to avoid overuse of view injection

Avoid mixing concerns by including business logic in Razor views

Accessing injected static configuration data inside views can be convenient

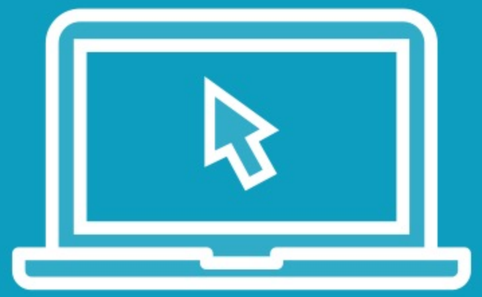
Ensure injected services are specific to the concern of view rendering

- Localizing content
- Populating lists

Pass state via page models, injecting dependencies into models, rather than views

Injecting Dependencies into Hosted Services

Demo



Using dependency injection with ASP.NET Core hosted services

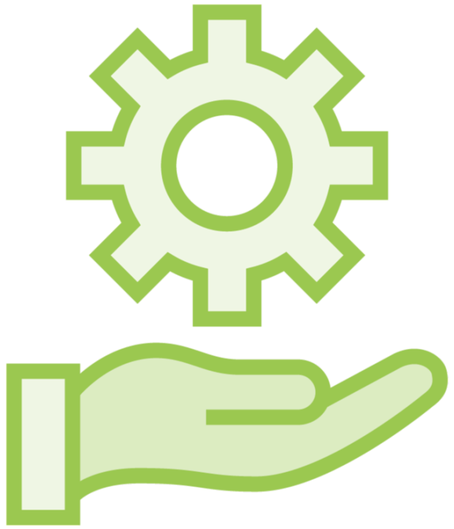
Manually creating scopes

Building ASP.NET Core Hosted Services and .NET Core Worker Services

Steve Gordon

app.pluralsight.com/library/courses/building-aspnet-core-hosted-services-net-core-worker-services

Hosted Services



At startup, the StartAsync method is called on all registered IHostedService implementations

Implementations of IHostedService are created when the application starts

Instances live until the application exits

Services injected into their constructors are captured for the life of the application

- Only singleton services may be safely injected

Service Locator Pattern

Manual Scope Creation

Only required outside of the ASP.NET Core request lifecycle

Hosted services are the main example where this is necessary

Scopes must be disposed once they are no longer required to release scoped resources

Up Next:
Beyond the Built-in Container