초보 서버 개발자와 함께 보는

ASP.NET Core Best Practices

최은정
Some Other Place

목차

1. 프로젝트 구조

2. 의존성

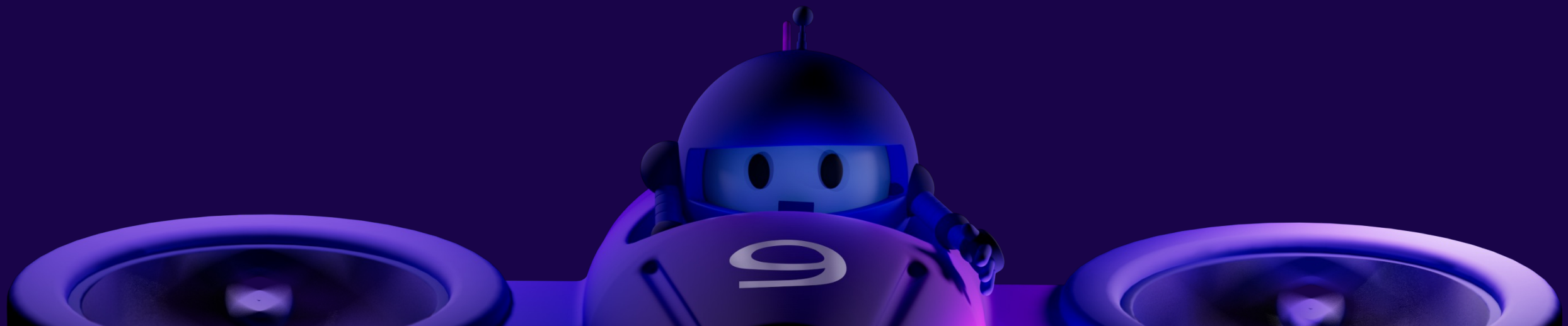
3. 비동기 프로그래밍

4. HttpContext

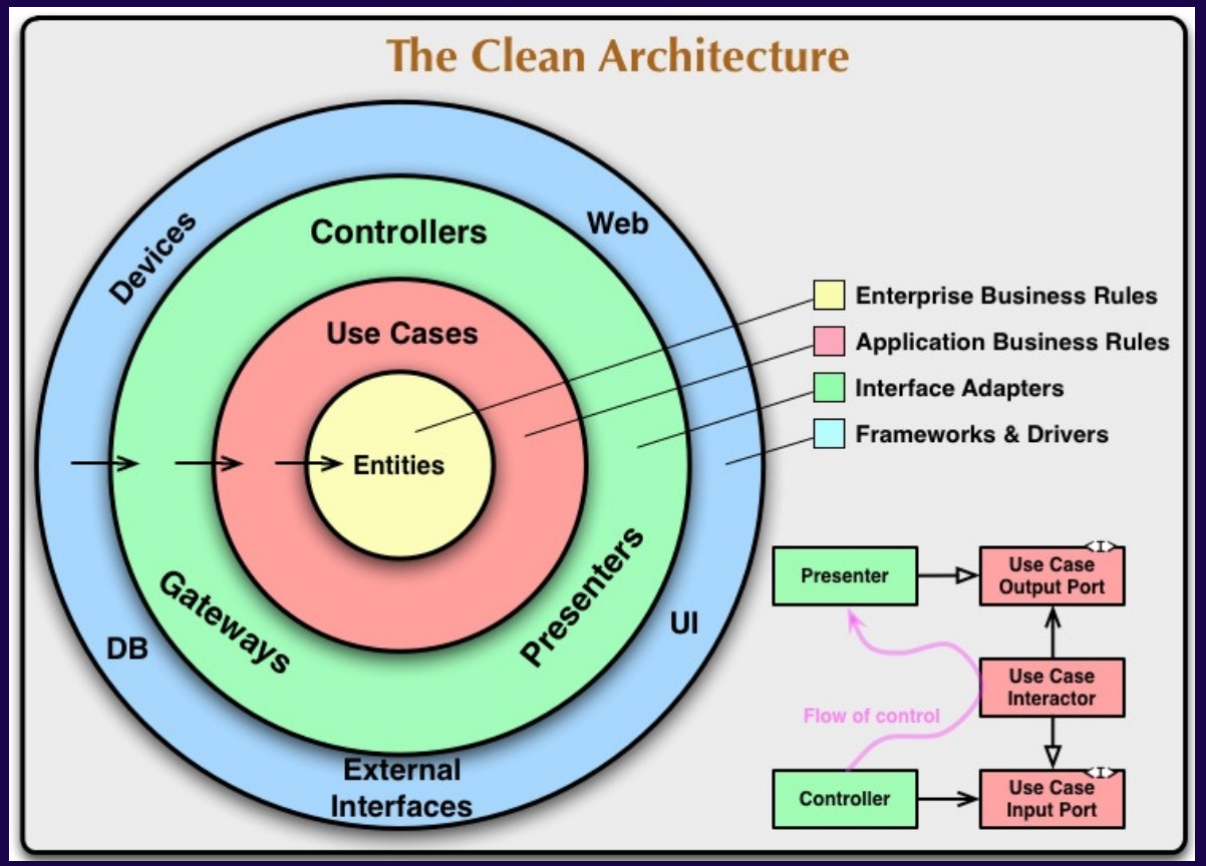
9

01

프로젝트 구조

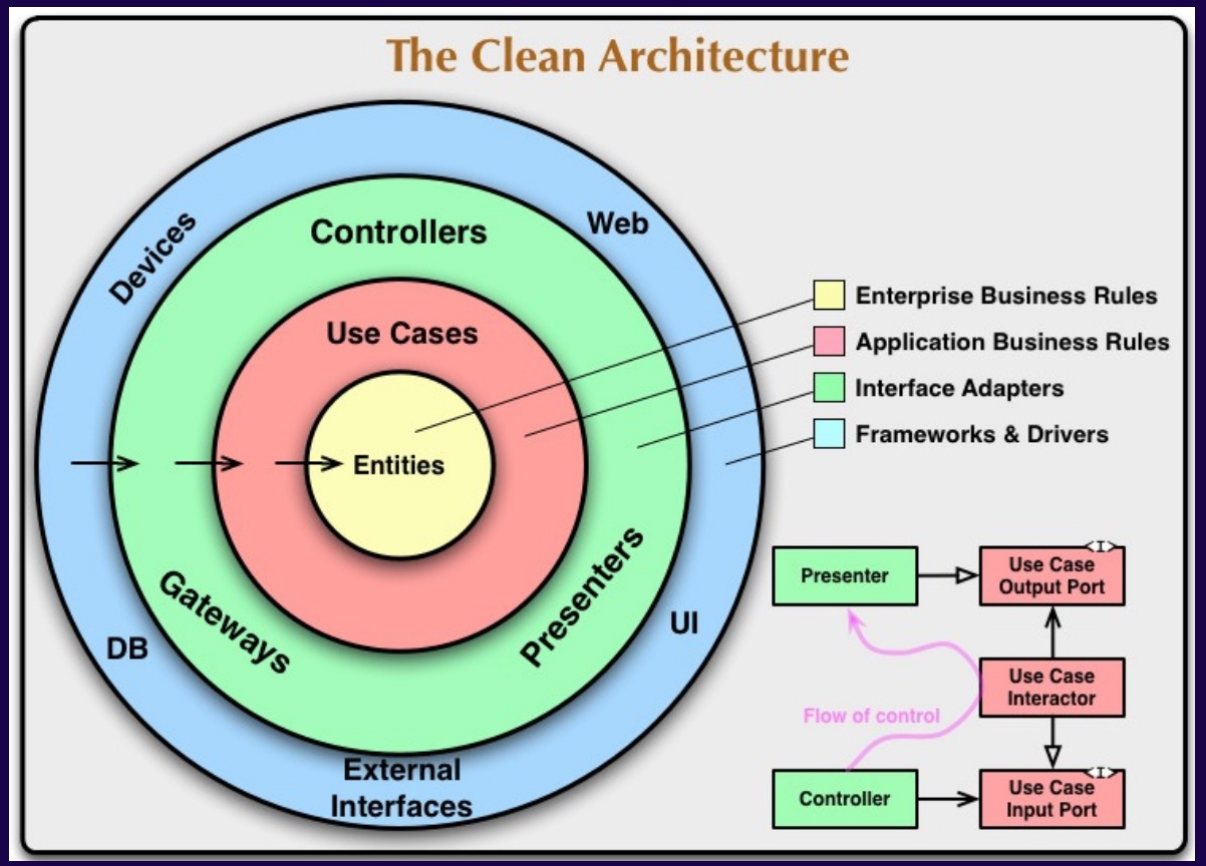


Clean Architecture



1. 변경이 용이하다
2. 테스트가 쉬워진다
3. 코드의 가독성 향상

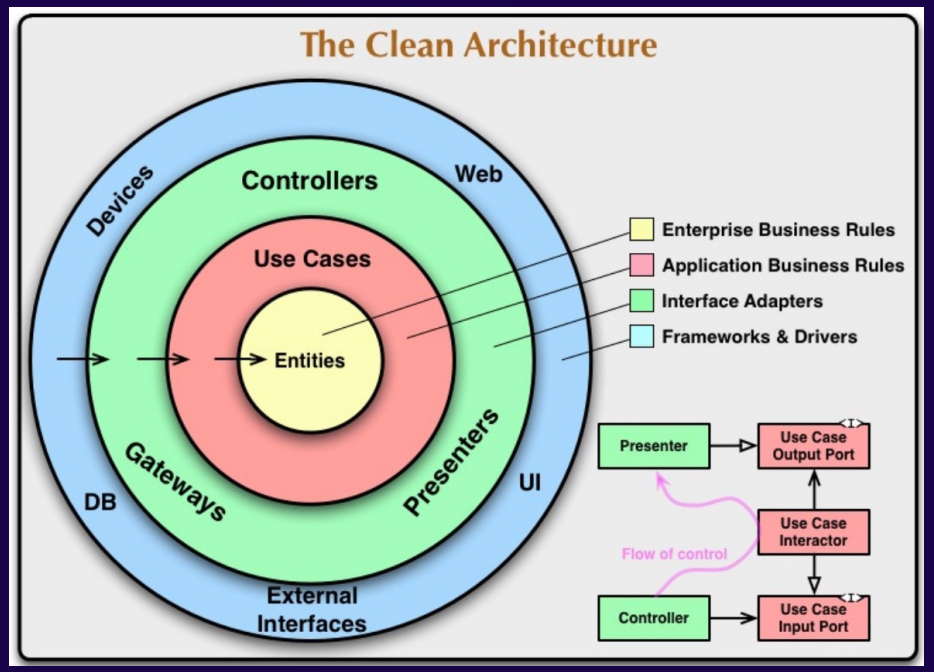
Entities (Domain layer)



비즈니스 규칙

```
// Domain Layer
public class User
{
    public Guid Id { get; private set; }
    public string Name { get; private set; }
    public string Email { get; private set; }
    public string PasswordHash { get; private set; }

    public User(string name, string email, string
passwordHash)
        {
            Id = Guid.NewGuid();
            Name = name;
            Email = email;
            PasswordHash = passwordHash;
    }
}

public interface IUserRepository
{
    Task<bool> EmailExistsAsync(string email);
    Task AddUserAsync(User user);
}
```

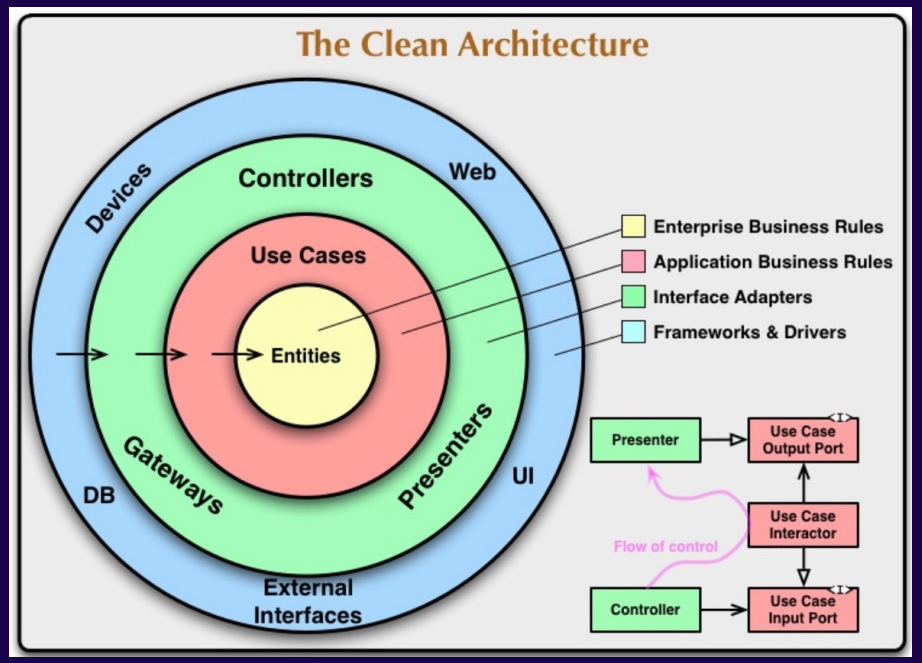
Use cases (Application layer)



비즈니스 규칙을 처리

```
// Application Layer
public class RegisterUserUseCase
{
    private readonly IUserRepository _userRepository;
    private readonly IPasswordHasher _passwordHasher;

    public RegisterUserUseCase(
      IUserRepository userRepository, IPasswordHasher passwordHasher)
    {
        _userRepository = userRepository;
        _passwordHasher = passwordHasher;
    }

    public async Task<RegisterUserResponse> ExecuteAsync(RegisterUserRequest
request)
    {
        if (await _userRepository.EmailExistsAsync(request.Email))
            throw new EmailAlreadyExistException();

        var passwordHash = _passwordHasher.HashPassword(request.Password);
        var user = new User(request.Name, request.Email, passwordHash);

        await _userRepository.AddUserAsync(user);

        return new RegisterUserResponse
        {
            Success = true,
            Message = "User registered successfully."
        };
    }
}
```

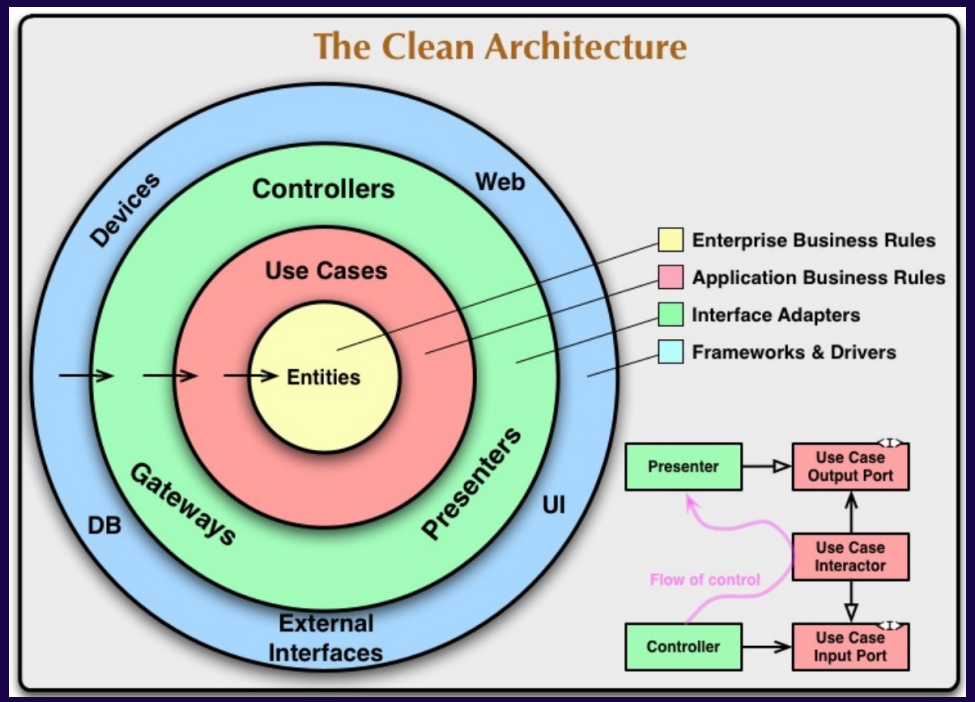
Interface Adapters (Presentation layer)



- 외부시스템과 use-case 를 연결
- DTO, Controllers
- 데이터를 필요한 형태로 변환

```
// Presentation Layer (Controller or API Endpoint)
public class UserController
{
    private readonly RegisterUserUseCase _registerUserUseCase;

    public UserController(RegisterUserUseCase registerUserUseCase)
    {
        _registerUserUseCase = registerUserUseCase;
    }

    public async Task<IActionResult> RegisterUser(RegisterUserRequest
request)
    {
        var response = await _registerUserUseCase.ExecuteAsync(request);

        if (response.Success)
        {
            return new OkObjectResult(response.Message);
        }

        return new BadRequestObjectResult(response.Message);
    }
}
```

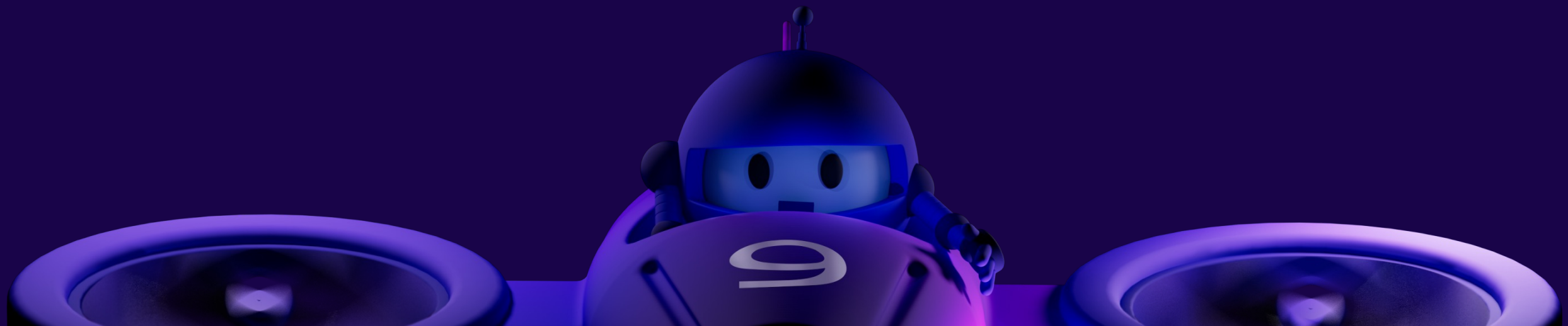
Framework & Drivers (Infrastructure layer)



데이터 베이스와 외부 API 구현과 같은 세부 사항

```
// Infrastructure Layer
public class UserRepository : IUserRepository
{
    // Example in-memory storage
    private readonly List<User> _users = new List<User>();

    public Task<bool> EmailExistsAsync(string email)
    {
        return Task.FromResult(_users.Any(u => u.Email ==
email));
    }

    public Task AddAsync(User user)
    {
        _users.Add(user);
        return Task.CompletedTask;
    }
}
```

02
의존성



의존성 역전 원칙

상위 모듈이 하위 모듈에 의존하지 않고, 추상화에 의존한다는 원칙

```
public class MyDependency
{
    public void WriteMessage(string message)
    {
        Console.WriteLine($"MyDependency.WriteMessage called. Message:
{message}");
}
```

```
public class IndexModel : PageModel
{
    private readonly MyDependency _dependency = new MyDependency();

    public void OnGet()
    {
        _dependency.WriteMessage("IndexModel.OnGet");
    }
}
```

의존성 역전 원칙

상위 모듈이 하위 모듈에 의존하지 않고, 추상화에 의존한다는 원칙

```
public interface IMyDependency
{
    void WriteMessage(string message);
}
```

```
public class MyDependency : IMyDependency
{
    public void WriteMessage(string message)
    {
        Console.WriteLine($"MyDependency.WriteMessage Message: {message}");
    }
}
```

```
public class Index2Model : PageModel
{
    private readonly IMyDependency _myDependency;

    public Index2Model(IMyDependency myDependency)
    {
        _myDependency = myDependency;
    }

    public void OnGet()
    {
        _myDependency.WriteMessage("Index2Model.OnGet");
    }
}
```

의존성 주입

객체 간의 의존성을 외부에서 주입하여 객체의 생성과 의존성 관리 책임을 분리하는 설계 패턴

```
// Program.cs
using DependencyInjectionSample.Interfaces;
using DependencyInjectionSample.Services;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddScoped<IMyDependency, MyDependency>();

var app = builder.Build();
```

의존성 주입

서비스의 생명 주기

1. **Singleton:** 애플리케이션 전체에서 단일 인스턴스를 공유
2. **Scoped :** HTTP 요청 단위로 인스턴스를 공유
3. **Transient:** 요청 시마다 새로운 인스턴스를 생성

```
public class FileService
{
    private readonly ConcurrentDictionary<string, byte[]> _cache;

    public FileService()
    {
        _cache = new ConcurrentDictionary<string, byte[]>();
    }

    public byte[] GetFileContent(string filePath)
    {
        return _cache.GetOrAdd(filePath, _ =>
        {
            return File.ReadAllBytes(filePath);
        });
    }
}
```

```
public class RequestItemsService
{
    private readonly Dictionary<string, object> _items;

    public RequestItemsService()
    {
        _items = new Dictionary<string, object>();
    }

    public void Set(string name, object value)
    {
        _items[name] = value;
    }

    public object Get(string name)
    {
        return _items[name];
    }
}
```

의존성 주입

Best Practices

1. 주입 받은 서비스는 read only field/property 로 선언하기
2. 스레드 안전하게 상태 관리하기
3. 메모리 누수에 주의하기
4. Captive Dependency 주의하기 : 수명이 더 긴 서비스가 수명이 더 짧은 서비스를 가지고 있는 것
5. HTTP 요청 밖에서 Scoped 서비스를 사용할 때는 자식 스코프를 사용하기

의존성 주입

Captive Dependency 주의하기 : 수명이 더 긴 서비스가 수명이 더 짧은 서비스를 가지고 있는 것

```
namespace DependencyInjection.AntiPatterns;

public class Foo(Bar bar)
{
}
```

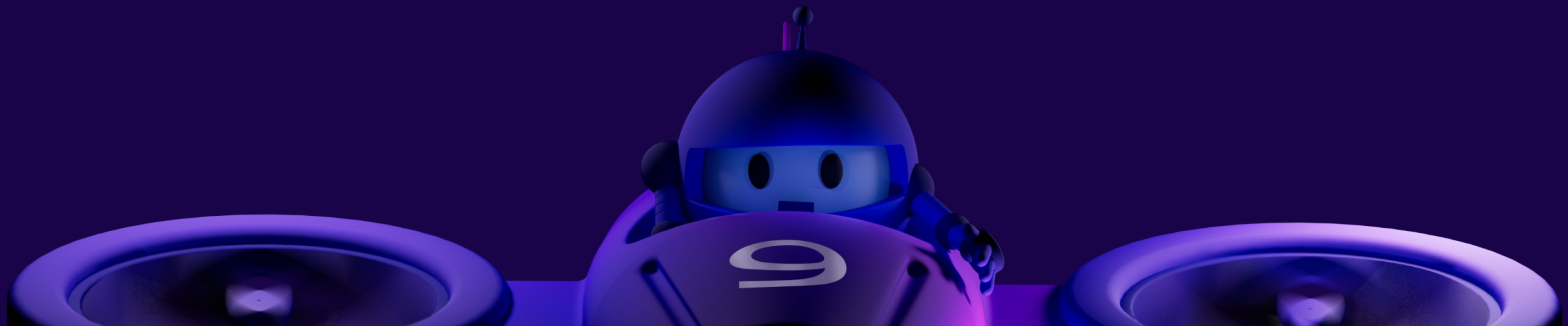
```
static void CaptiveDependency()
{
  var services = new ServiceCollection();
  services.AddSingleton<Foo>();
  services.AddScoped<Bar>();

  using ServiceProvider serviceProvider = services.BuildServiceProvider();

  // Enable scope validation
  // using ServiceProvider serviceProvider = services.BuildServiceProvider(validateScopes: true);

  _ = serviceProvider.GetRequiredService<Foo>();
}
```

의존성 주입

자식 스코프로 명확한 수명 관리

```
public class PriceCalculator
{
    private readonly IServiceProvider _serviceProvider;

    public PriceCalculator(IServiceProvider serviceProvider)
    {
        _serviceProvider = serviceProvider;
    }

    public float Calculate(Product product, int count,
      Type taxStrategyServiceType)
    {
        using (var scope = _serviceProvider.CreateScope())
        {
            var taxStrategy = (ITaxStrategy)scope.ServiceProvider
              .GetRequiredService(taxStrategyServiceType);
            var price = product.Price * count;

            return price + taxStrategy.CalculateTax(price);
        }
    }
}
```

03
비동기 프로그래밍



IEnumerable vs. IAsyncEnumerable

- IEnumerable : 동기적 데이터 순회
- IAsyncEnumerable : 비동기적 데이터 순회

Do

1. 대량 데이터를 내보내야 할 때
2. Pagination 을 해야할 때

비동기 API 사용을 선호하기

"Sync over Async" : 비동기를 동기로 처리하는 현상

Do Not

```
public class BadStreamReaderController : Controller
{
    [HttpGet("/contoso")]
    public ActionResult<ContosoData> Get()
    {
        var json = new StreamReader(Request.Body).ReadToEnd();

        return JsonSerializer.Deserialize<ContosoData>(json);
    }
}
```

Do

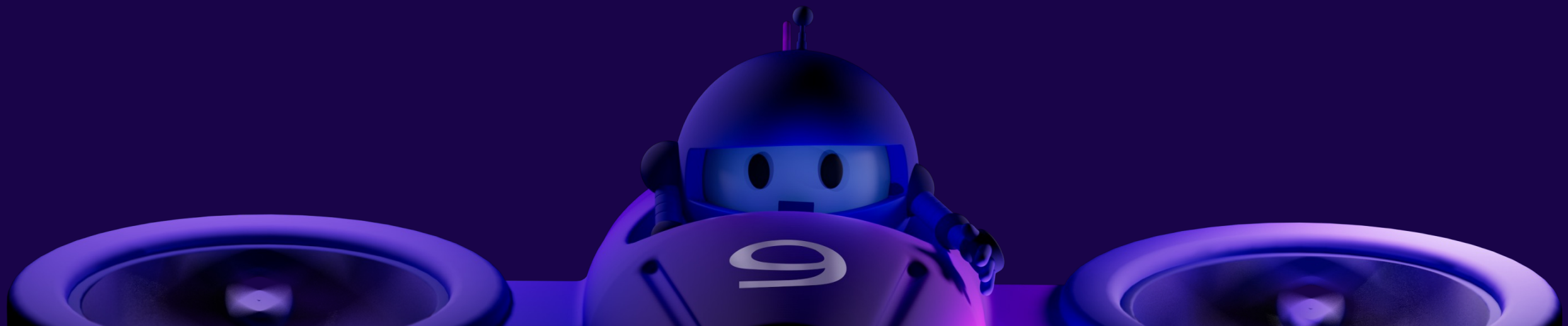
```
public class GoodStreamReaderController : Controller
{
    [HttpGet("/contoso")]
    public async Task<ActionResult<ContosoData>> Get()
    {
        var json = await new StreamReader(Request.Body).ReadToEndAsync();

        return JsonSerializer.Deserialize<ContosoData>(json);
    }
}
```

기타 Best practice

1. Task.Wait 나 Task<TResult>.Result 를 호출해서 비동기 실행을 블로킹하지 말 것 -> await 을 사용
2. Task.Run 을 호출한 뒤 바로 await 하지 말 것 -> cpu 집약적인 작업에서만 사용

04

HttpContext



IHttpContextAccessor.HttpContext 를 field 로 저장하지 말 것

- HttpContext : HTTP 요청에 대한 모든 정보를 담고 있는 객체. 요청마다 달라짐.
- IHttpContextAccessor : 현재 활성화된 HttpContext 를 가져오는 인터페이스

Do Not

```
public class MyBadType
{
    private readonly HttpContext _context;
    public MyBadType(IHttpContextAccessor accessor)
    {
        _context = accessor.HttpContext;
    }

    public void CheckAdmin()
    {
        if (!_context.User.IsInRole("admin"))
        {
            throw new UnauthorizedAccessException("The current user
an admin");
        }
    }
}
```

Do

```
public class MyGoodType
{
    private readonly IHttpContextAccessor _accessor;
    public MyGoodType(IHttpContextAccessor accessor)
    {
        _accessor = accessor;
    }

    public void CheckAdmin()
    {
        var context = _accessor.HttpContext;
        if (context != null && !context.User.IsInRole("admin"))
        {
            throw new UnauthorizedAccessException("The current user isn't
an admin");
        }
    }
}
```

HttpContext 를 여러 스레드에서 동시에 접근하지 말 것

HttpContext 는 단일 스레드에서 사용하도록 설계되어있다

Do Not

```
public class AsyncBadSearchController : Controller
{
    [HttpGet("/search")]
    public async Task<SearchResults> Get(string query)
    {
        var query1 = SearchAsync(SearchEngine.Google, query);
        var query2 = SearchAsync(SearchEngine.Bing, query);
        var query3 = SearchAsync(SearchEngine.DuckDuckGo, query);

        await Task.WhenAll(query1, query2, query3);

        var results1 = await query1;
        var results2 = await query2;
        var results3 = await query3;

        return SearchResults.Combine(results1, results2, results3);
    }
    ...
```

```
public class AsyncBadSearchController : Controller
{
    private async Task<SearchResults> SearchAsync(
        SearchEngine engine, string query)
    {
        var searchResults = _searchService.Empty();
        try
        {
            _logger.LogInformation("Starting search query from {path}.",
                                   HttpContext.Request.Path);
            searchResults = _searchService.Search(engine, query);
            _logger.LogInformation("Finishing search query from {path}.",
                                   HttpContext.Request.Path);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Failed query from {path}",
                             HttpContext.Request.Path);
        }

        return await searchResults;
    }
```

HttpContext 를 여러 스레드에서 동시에 접근하지 말 것

Do

```
public class AsyncGoodSearchController : Controller
{
    [HttpGet("/search")]
    public async Task<SearchResults> Get(string query)
    {
        string path = HttpContext.Request.Path;
        var query1 = SearchAsync(SearchEngine.Google, query,path);
        var query2 = SearchAsync(SearchEngine.Bing, query, path);
        var query3 = SearchAsync(SearchEngine.DuckDuckGo, query, path);

        await Task.WhenAll(query1, query2, query3);

        var results1 = await query1;
        var results2 = await query2;
        var results3 = await query3;

        return SearchResults.Combine(results1, results2, results3);
    }

    ...
```

request 가 완료된 이후에는 HttpContext 를 사용하지 말 것

HttpContext 는 요청 처리 중에만 유효하며 이후에는 재활용 된다

Do Not

```
public class AsyncBadVoidController : Controller
{
    [HttpGet("/async")]
    public async void Get()
    {
        await Task.Delay(1000);

        await Response.WriteAsync("Hello World");
    }
}
```

Do

```
public class AsyncGoodTaskController : Controller
{
    [HttpGet("/async")]
    public async Task Get()
    {
        await Task.Delay(1000);

        await Response.WriteAsync("Hello World");
    }
}
```

백그라운드 스레드에서 HttpContext 를 직접 참조하지 말 것

Do Not

```
[HttpGet("/fire-and-forget-1")]
public IActionResult BadFireAndForget()
{
    _ = Task.Run(async () =>
    {
        await Task.Delay(1000);

        var path = HttpContext.Request.Path;
        Log(path);
    });

    return Accepted();
}
```

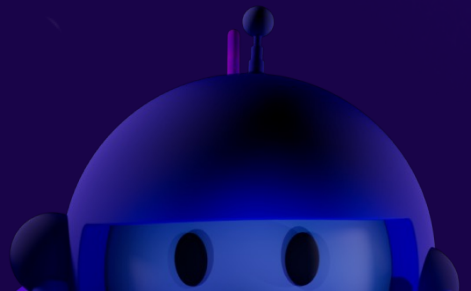
Do

```
[HttpGet("/fire-and-forget-3")]
public IActionResult GoodFireAndForget()
{
    string path = HttpContext.Request.Path;
    _ = Task.Run(async () =>
    {
        await Task.Delay(1000);

        Log(path);
    });

    return Accepted();
}
```

1. 프로젝트 구조

2. 의존성

3. 비동기 프로그래밍

4. HttpContext

감사합니다