Overview of Dependency Injection Containers

Jeremy Clark
DEVELOPER BETTERER

@jeremybytes   www.jeremybytes.com

Dependency Injection Containers

Autofac

Ninject

Unity

Castle Windsor

Spring.NET

DI Containers

Why Containers?

Ninject

Autofac

Late binding

ASP.NET Core MVC

Dependency Injection Containers

Auto-registration

Auto-wiring

Lifetime management

Demo

Using Ninject

- Configuring the container
- Lifetime management
- Composing the objects

Demo

Using a decorator with Ninject

Demo

Using Autofac

- Configuring the container
- Lifetime management
- Composing the objects

Demo

Using Autofac

- Auto-registration
- Manual registration

Demo

Using a decorator with Autofac

Demo

Late binding with Autofac

Demo

ASP.NET Core MVC dependency injection

- Constructor injection on a controller
- Composing objects in Startup

Dependency Injection Containers

Auto-registration

Auto-wiring

Lifetime management

DI Containers

Why Containers?

Ninject

Autofac

Late binding

ASP.NET Core MVC

Dependency Injection (DI)

A set of software design principles and patterns that enable us to develop loosely coupled code.

van Deursen and Seeman. *Dependency Injection in .NET*. Manning, 2018.

Benefits of Loose Coupling

Easy to extend

Easy to test

Easy to maintain

Facilitates parallel development

Facilitates late binding

S	• Single Responsibility Principle
O	• Open/Closed Principle
L	• Liskov Substitution Principle
I	• Interface Segregation Principle
D	• Dependency Inversion Principle

Dependency Injection Patterns

Constructor Injection

Property Injection

Method Injection

Ambient Context

Service Locator

Constructor Injection

```
public class PeopleViewModel
{
    protected IPersonReader DataReader;

    public IEnumerable<Person> People ...

    public PeopleViewModel(IPersonReader dataReader)
    {
        DataReader = dataReader;
    }

    public void RefreshPeople()
    {
        People = DataReader.GetPeople();
    } ...
}
```

Dependency

Inject the dependency using the constructor

Dependency

Property Injection

```
public class CSVReader : IPersonReader
{
    public ICSVFileLoader FileLoader { get; set; }

    public CSVReader()
    {   ...
        FileLoader = new CSVFileLoader(filePath);
    }

    public IEnumerable<Person> GetPeople()
    {
        string fileData = FileLoader.LoadFile();
        IEnumerable<Person> people =
            ParseDataString(fileData);
        return people;
    } ...
}
```

By default, uses the real file loader

Property Injection

```
[TestMethod]
public void GetPeople_WithGoodRecords_ReturnsAllRecords()
{
    var reader = new CSVReader();
    reader.FileLoader = new FakeFileLoader("Good");
```

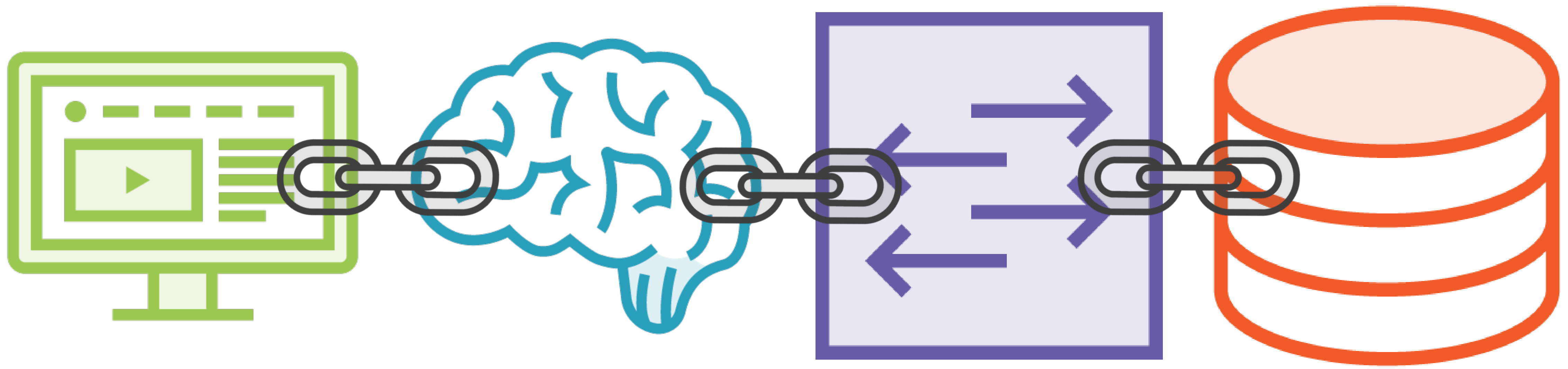
Injection point to override default behavior for tests

```
    var result = reader.GetPeople();

    Assert.AreEqual(2, result.Count());

}
```

Breaking Tight Coupling



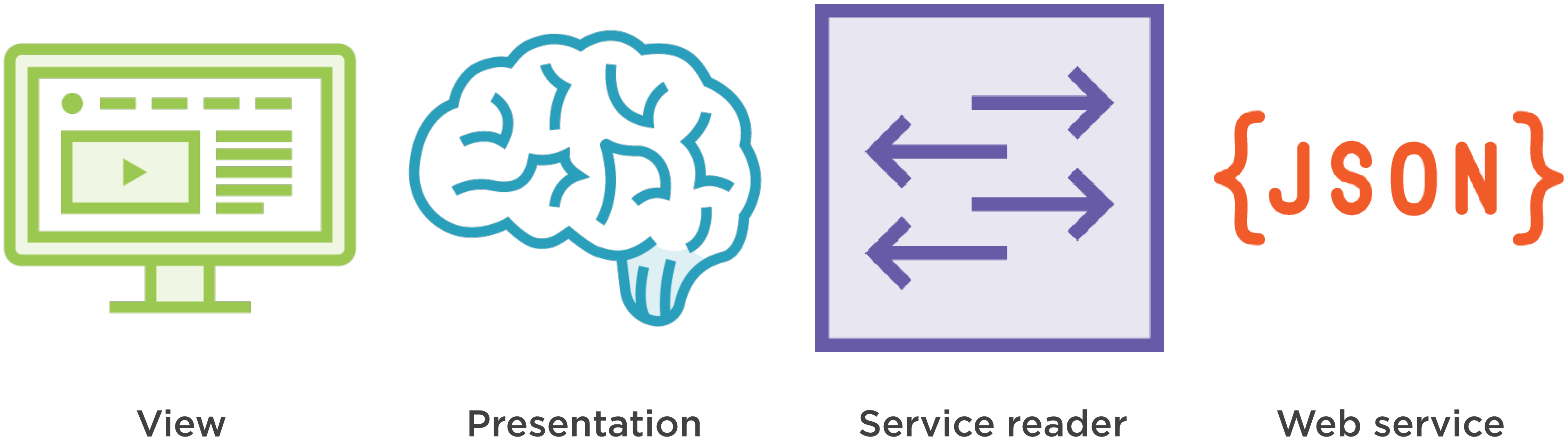
View
(UI elements)

Presentation
(UI logic)

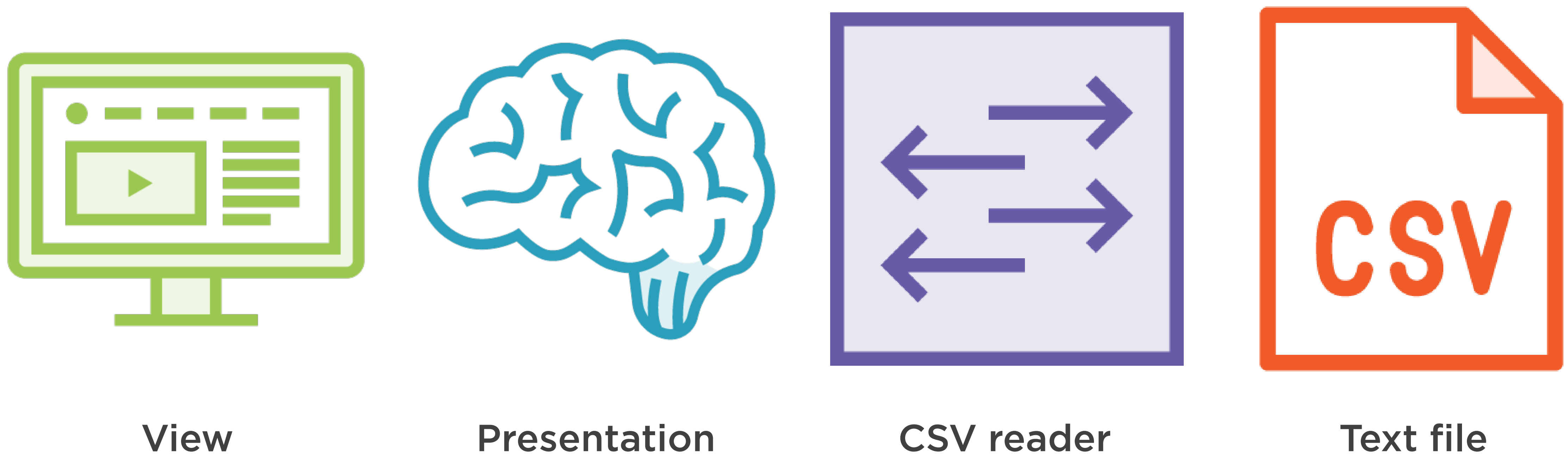
Data access

Data store

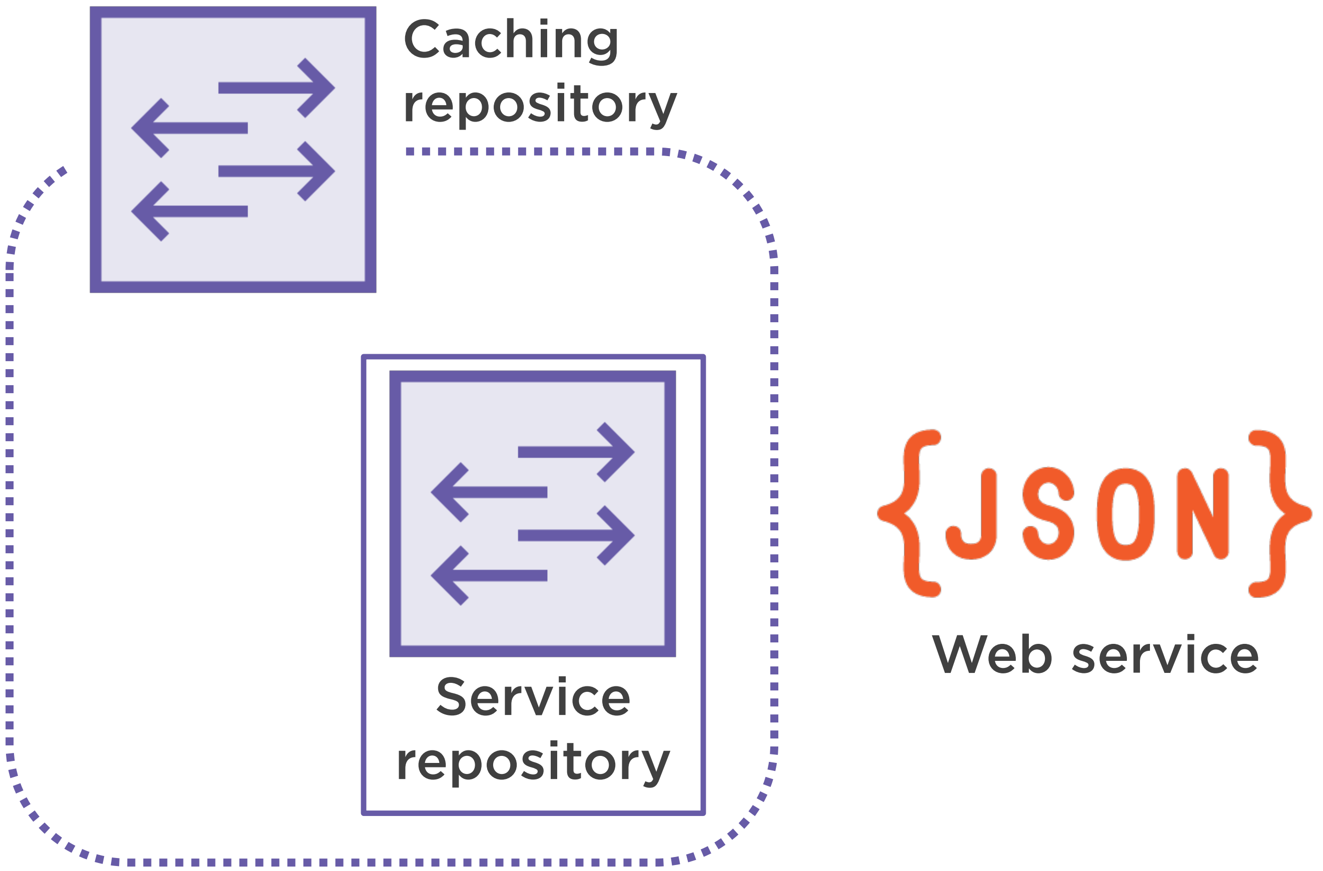
Changing the Data Source



Changing the Data Source



Repository Decorator



Unit Test with DI

```
[TestMethod]
public void People_OnRefreshPeople_IsPopulated()
{
    // Arrange
    IPersonReader reader = GetFakeReader();
    var viewModel = new PeopleViewModel(reader);

    // Act
    viewModel.RefreshPeople();

    // Assert
    ...

}
```

Dependency Injection Containers

Auto-registration

Auto-wiring

Lifetime management

Where to go next

SOLID design principles

Dependency injection containers

Advanced DI concepts

- Lifetime management
- Static vs. volatile dependencies
- Managing over-injection
- Injecting strings and other primitives
- Interception
- Additional patterns