Getting Started with Dependency Injection in .NET

WHAT IS TIGHT COUPLING AND HOW DOES IT AFFECT YOUR APPLICATIONS?

Jeremy Clark
DEVELOPER BETTERER

@jeremybytes www.jeremybytes.com

Typical Introduction

```
private void BuildMainWindow()
{
    var builder = new ContainerBuilder();

    builder.RegisterType<SQLReader>().As<IPersonReader>()
        .SingleInstance();

    builder.RegisterSource(
        new AnyConcreteTypeNotAlreadyRegisteredSource());

    IContainer Container = builder.Build();

    Application.Current.MainWindow =
        Container.Resolve<PeopleViewerWindow>();
}
```

I Need to Add Some Code

Copy / paste

Change a few things

Hope that it works

A Better Approach

Step back

Look at key concepts

Why and how

Lots of code

Prerequisites

- Classes
- Interfaces
- Constructors
- Properties

Development Environment

Visual Studio 2017 (any edition)

- .NET desktop development
- ASP.NET and web development

.NET Projects

- .NET Framework 4.7.2
- .NET Standard 2.0
- .NET Core 2.2

Overview

- What is dependency injection?
- Benefits
- Patterns
- Dependency injection containers
- Sample code that illustrates our problem

Defining Dependency Injection

Ask 10 developers

Get 12 answers

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

Dependency Injection Containers

Autofac

Ninject

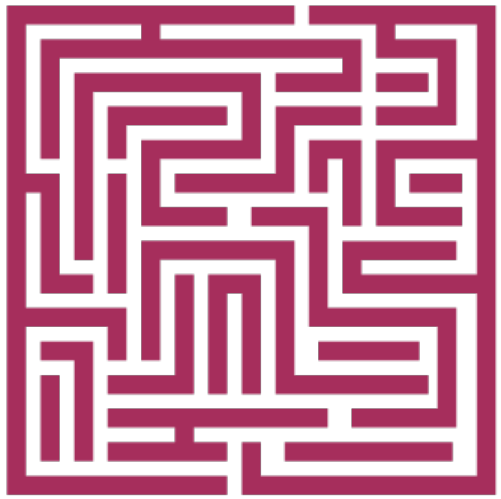
Unity

Castle Windsor

Spring.NET

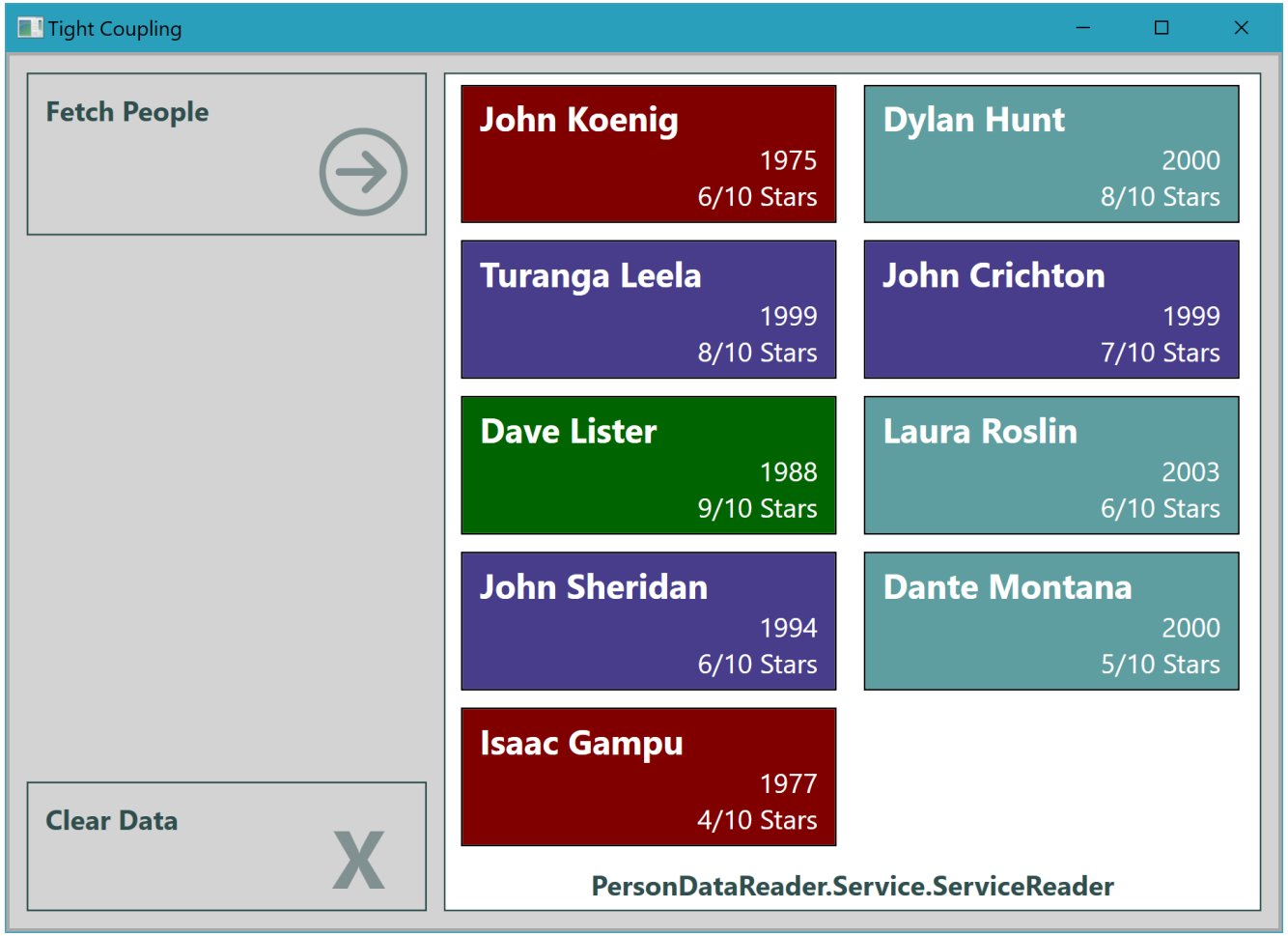
Sample Application

Simpler than it needs to be

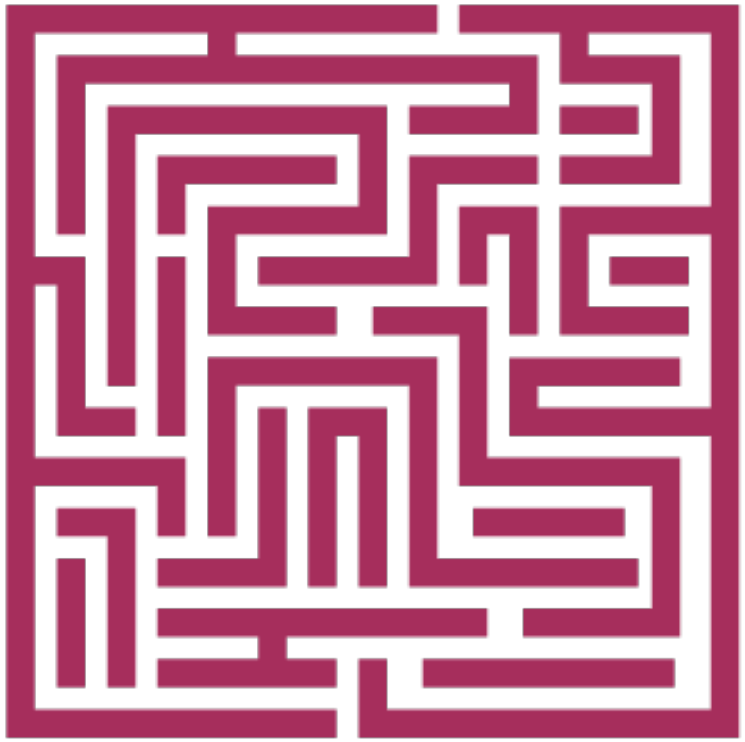


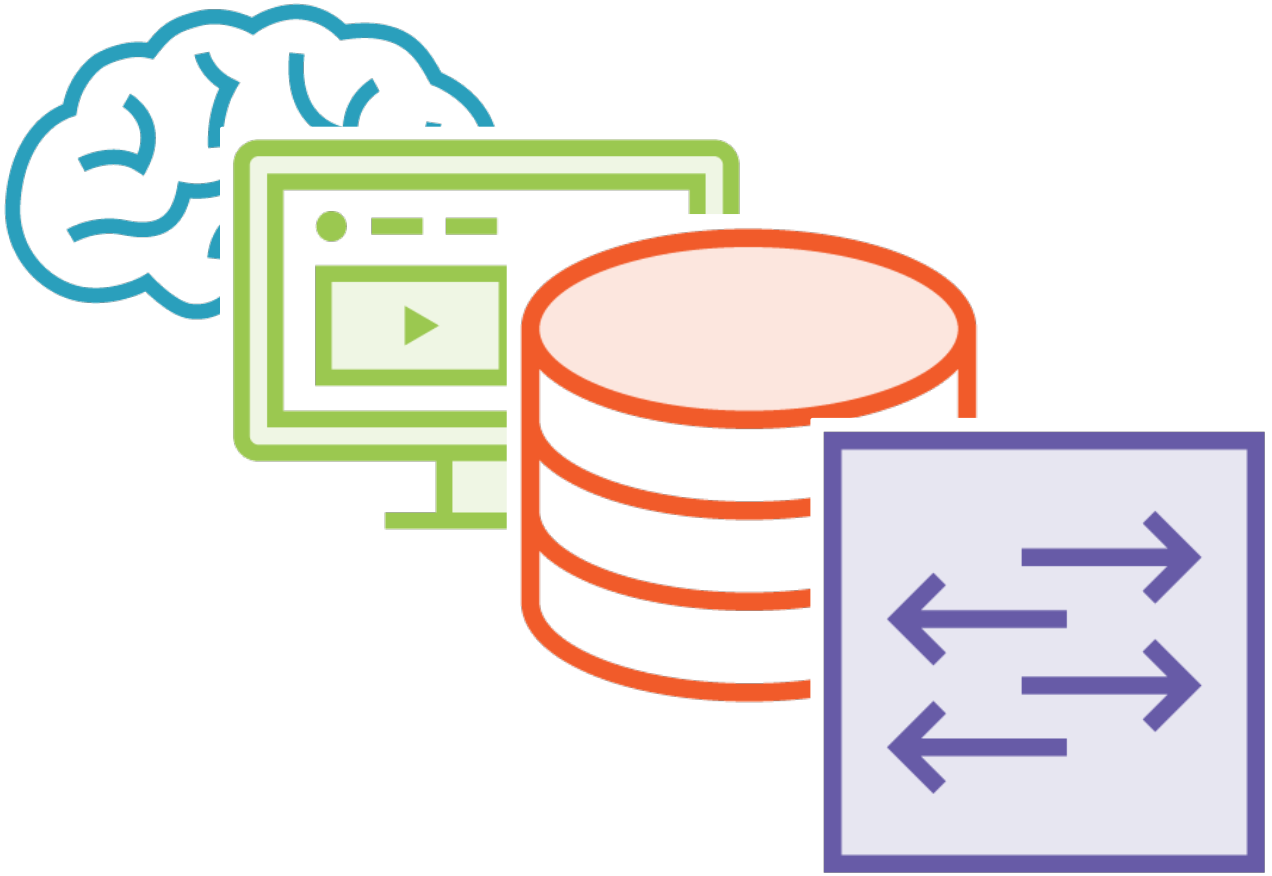
More complex than it needs to be

Simple: One Screen Application



Complex: Multiple Layers

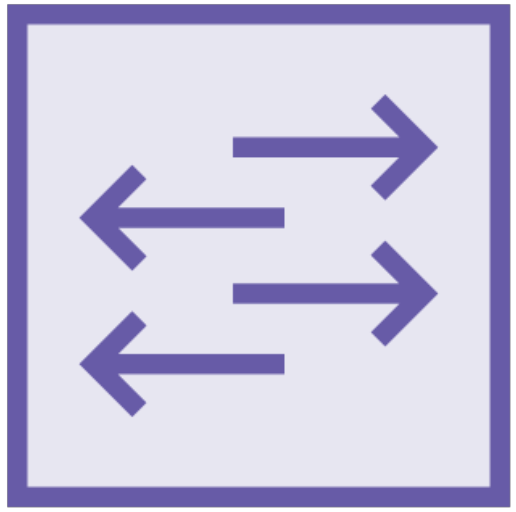




Application Layers

View
(UI elements)

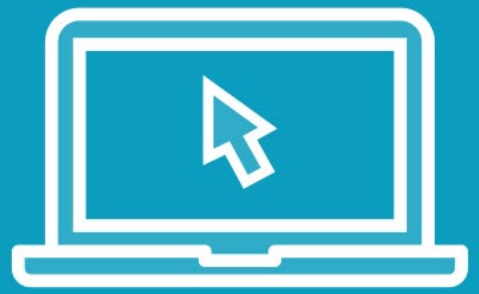
Presentation
(UI logic)



Data access

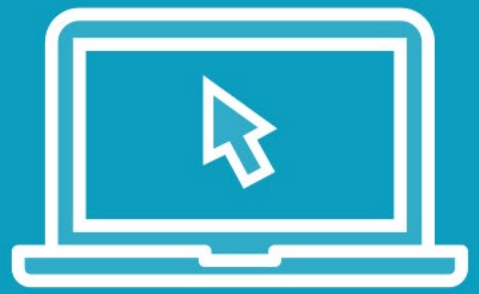
Data store

Demo



Project overview

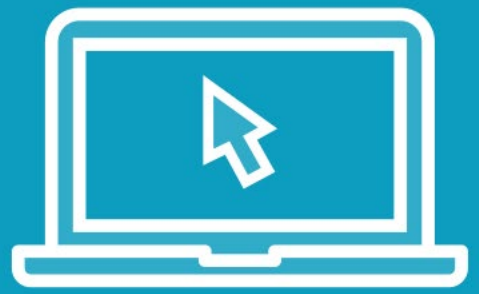
Demo



Data storage

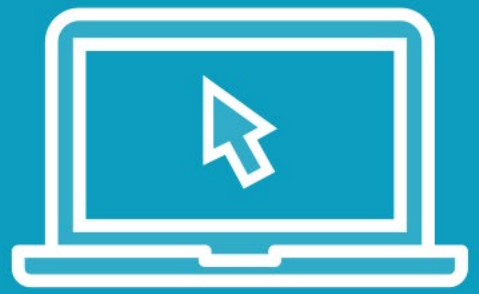
Data access

Demo



The presentation logic

Demo

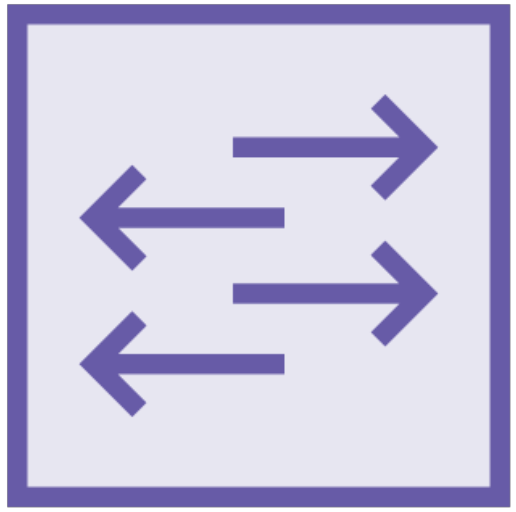


The user interface

Separation of Concerns

View
(UI elements)

Presentation
(UI logic)



Data access

Data store

Tightly-coupled code

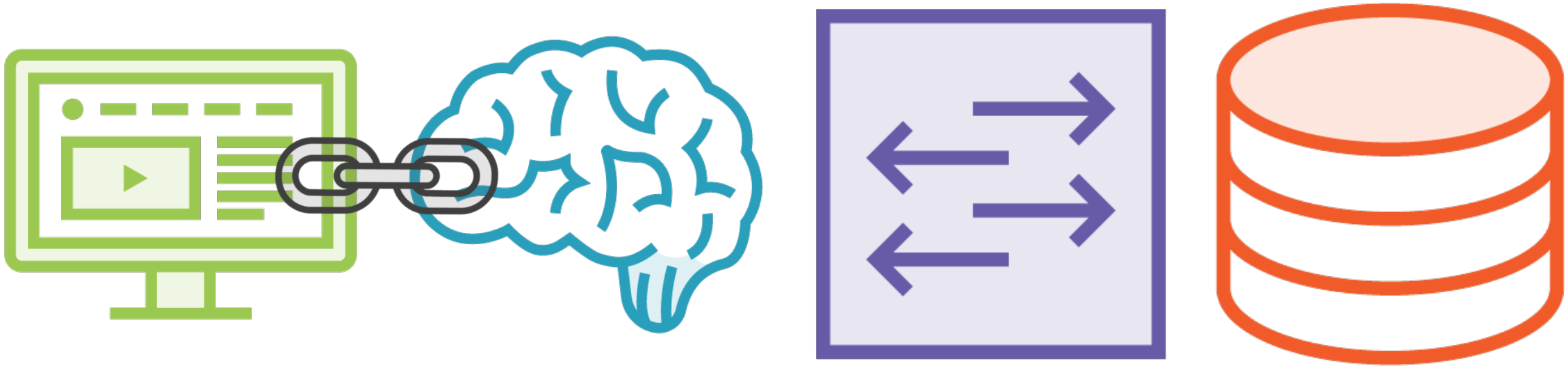
```
public PeopleViewerWindow()
{
    InitializeComponent();
    viewModel = new PeopleViewModel();
    this.DataContext = viewModel;
}
```

View - View Model Relationship

Requires a compile-time reference

Lifetime responsibility

Tight Coupling



View
(UI elements)

Presentation
(UI logic)

Data access

Data store

```
public class PeopleViewModel
{
    protected ServiceReader DataReader;

    public PeopleViewModel()
    {
        DataReader = new ServiceReader();
    } ...
}
```

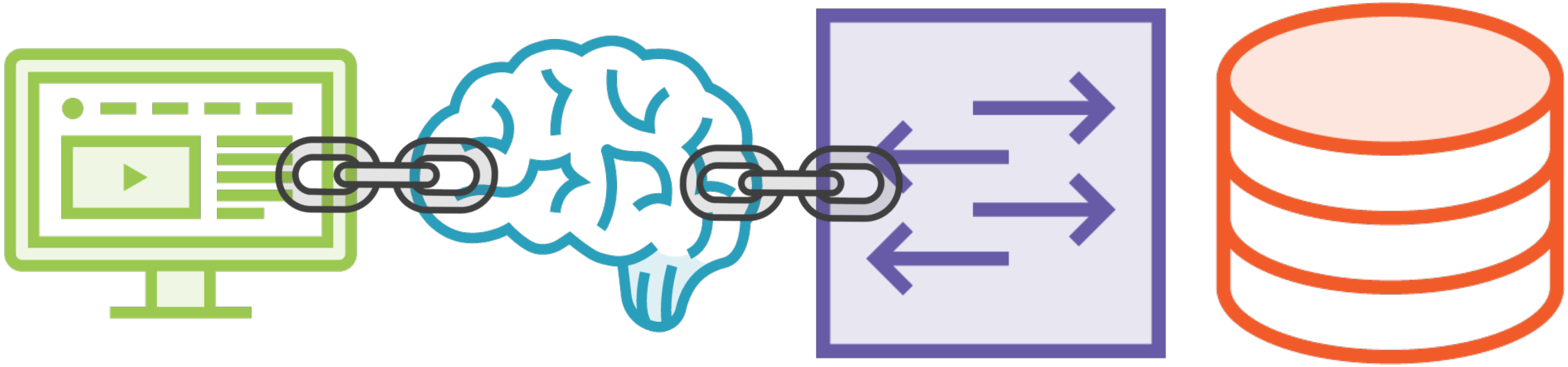
View Model - Data Reader Relationship

Requires compile-time reference

Lifetime responsibility

Selects the data access technology

Tight Coupling



View
(UI elements)

Presentation
(UI logic)

Data access

Data store

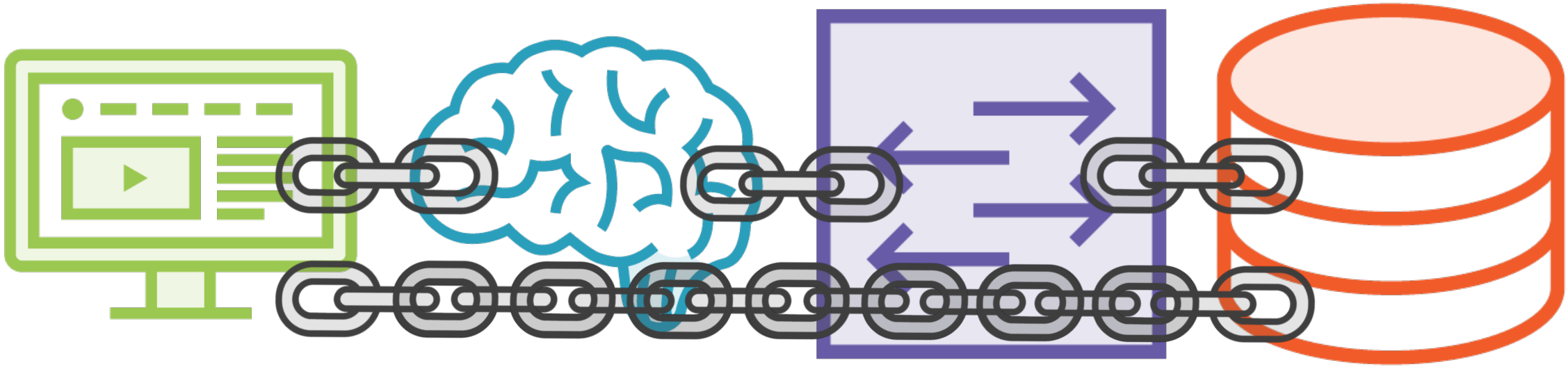
```
public class ServiceReader
{
    WebClient client = new WebClient();
    string baseUri = "http://localhost:9874/api/people";
    ...
}
```

Data Reader - Data Store Relationship

Requires compile-time reference

Lifetime responsibilities

Tight Coupling



View
(UI elements)

Presentation
(UI logic)

Data access

Data store

Does it really matter?

Great application!

But...

We need a few more things

Request 1: Different Data Sources

Web service

Text file

SQL database

Document database

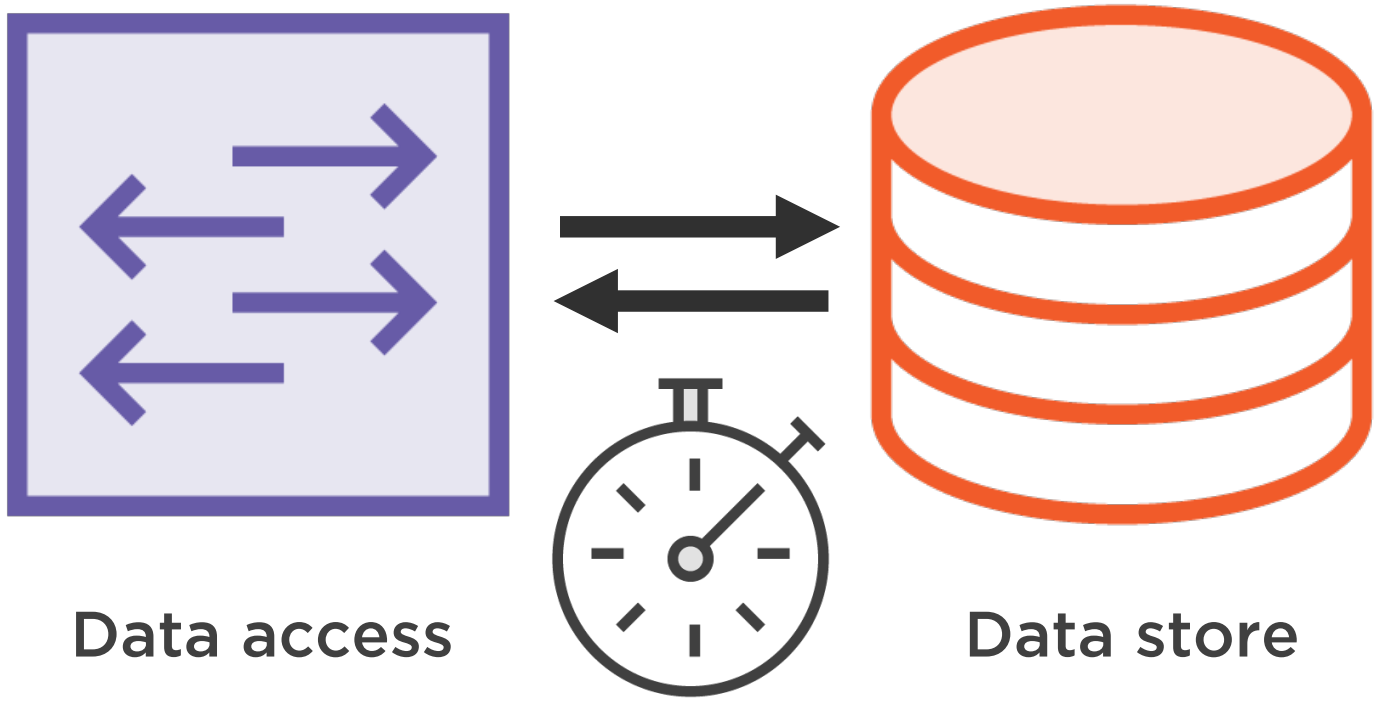
Cloud service

Azure functions

Request 1: Different Data Sources

```
public PeopleViewModel()
{
    switch (dataReaderType)
    {
        case "service": DataReader = new ServiceReader();
            break;
        case "text": DataReader = new CSVReader();
            break;
        case "sql": DataReader = new SQLReader();
            break;
    }
}
```

Request 2: Client-side Cache



Request 2: Client-side Cache

```
    public PeopleViewModel() {
        switch (dataReaderType) {
            case "service": DataReader = new ServiceReader();
                break;
            case "service_cached": DataReader = new CachedServiceReader();
                break;
            case "text": DataReader = new CSVReader();
                break;
            case "text_cached": DataReader = new CachedCSVReader();
                break;
            case "sql": DataReader = new SQLReader();
                break;
            case "sql_cached": DataReader = new CachedSQLReader();
                break;
    }
}
```

Violation

S • Single Responsibility Principle

Current view model responsibilities

- Presentation logic
- Picking the data source
- Managing object lifetime
- Deciding to use a cache

Request 3: Unit Tests

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

    ...
}
```

```
public PeopleViewModel()
{
    DataReader = new ServiceReader();
}
```

```
public class ServiceReader
{
  WebClient client = new WebClient();
  ...
}
```

Requests from the Boss

- Different data sources
- Client-side cache
- Unit tests

There has to be
a better way

Loose coupling with dependency injection

Overview

- What is dependency injection?
- Benefits
- Patterns
- Dependency injection containers
- Sample code that illustrates our problem