Using Dependency Injection to Build Loosely-coupled Applications

Jeremy Clark
DEVELOPER BETTERER

@jeremybytes www.jeremybytes.com

Using DI

Add some abstraction

Use constructor injection

Object composition

SOLID

Requests from the Boss

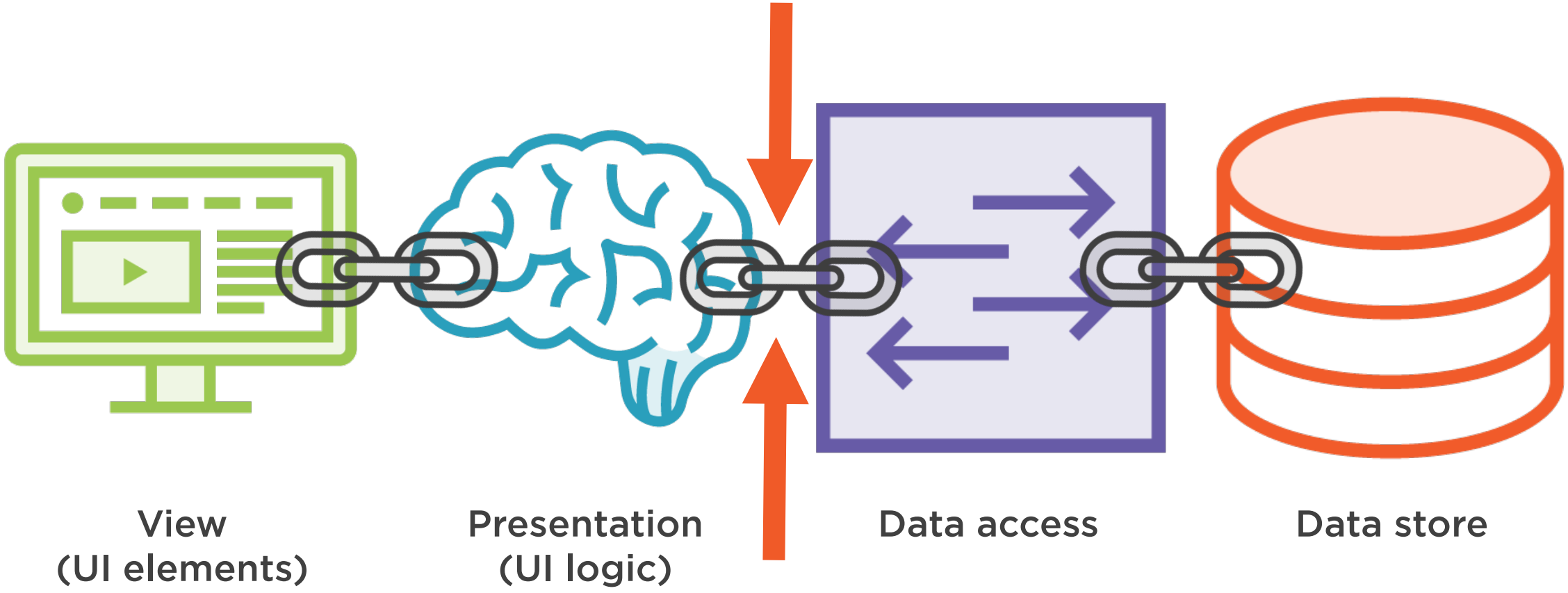
- Different data sources
- Client-side cache
- Unit tests

```
public class PeopleViewModel
{
    protected ServiceReader DataReader;

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

View Model - Data Reader Relationship

Break Tight Coupling



Loosen coupling with an interface

Break coupling with constructor injection

Snap the loosely-coupled pieces together

Different Data Sources

Web service

Text file

SQL database

Document database

Cloud service

Azure functions

Repository Pattern

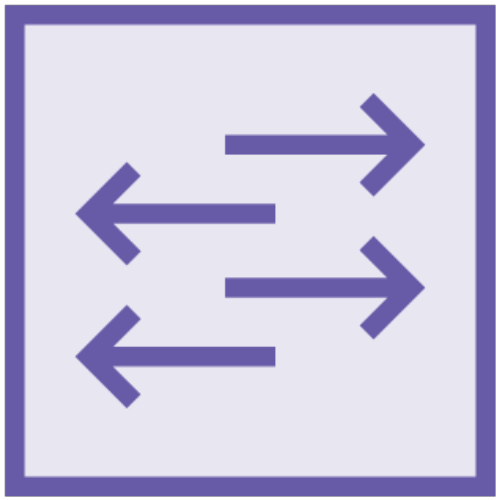
Mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects.

Fowler, et al. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.

Repository Pattern

Separates our application from the data storage technology

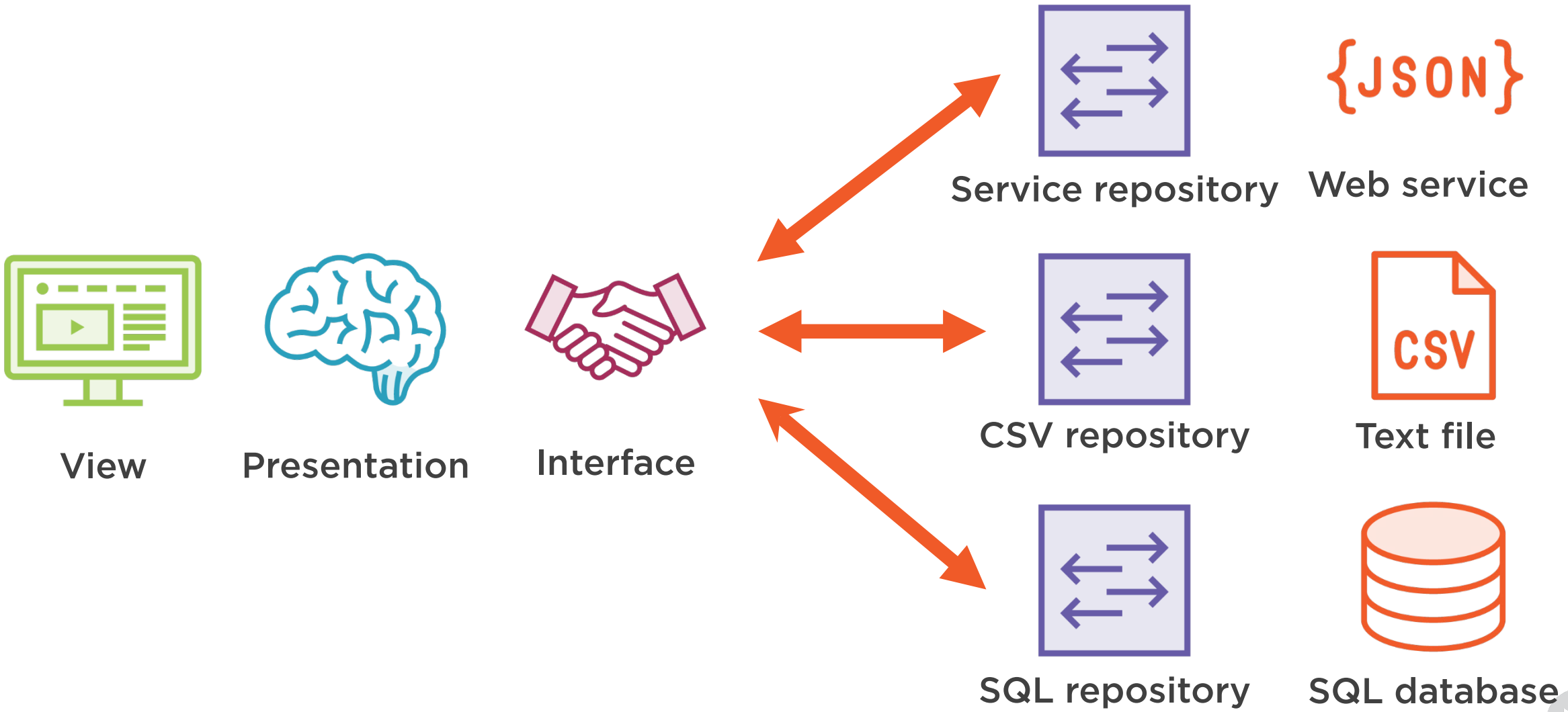
Application



Repository

Data store

Adding an Interface



CRUD Repository

Create

Read

Update

Delete

I • Interface Segregation Principle

Full repository

- Read
- Write

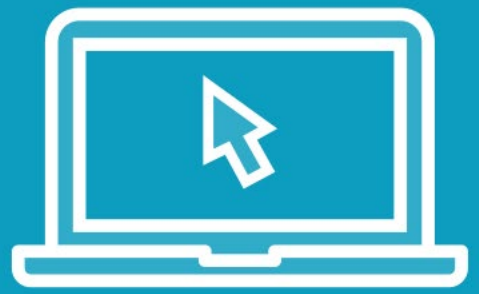
Our needs

- Read-only

Data Reader Interface

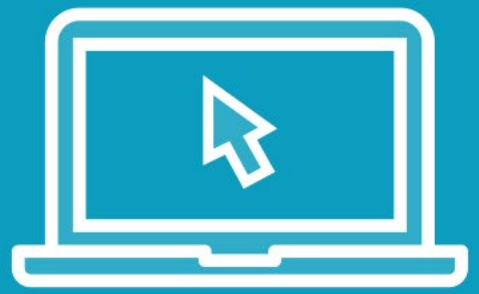
```
public interface IPersonReader
{
    IEnumerable<Person> GetPeople();
    Person GetPerson(int id);
}
```

Demo



Add a data reader interface

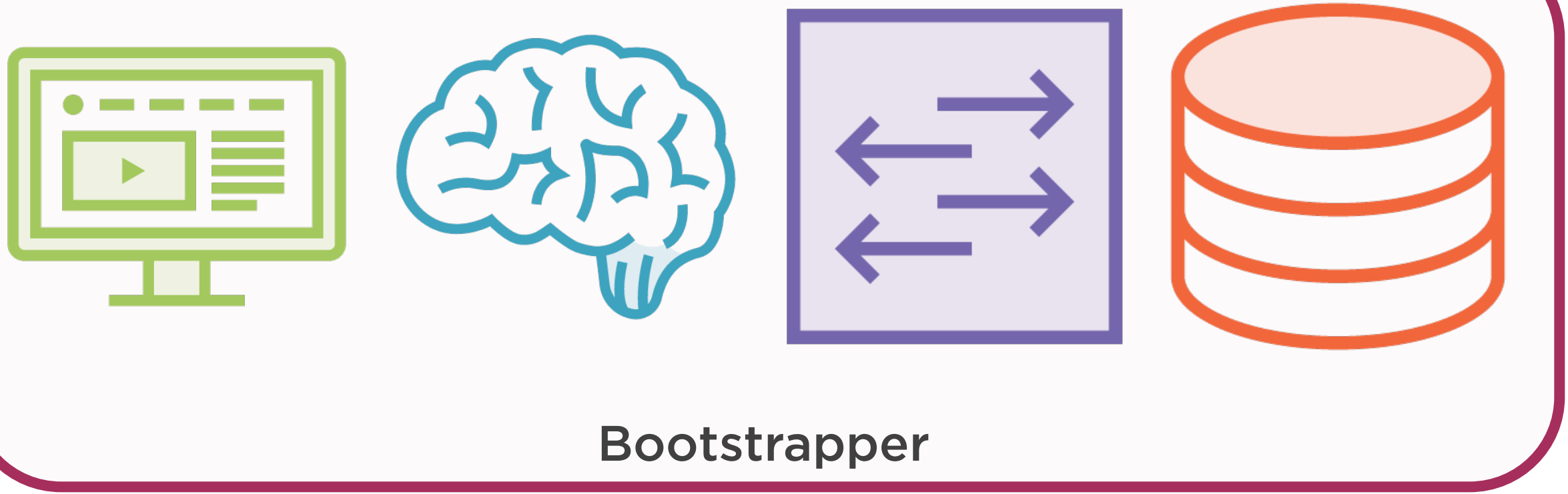
Demo



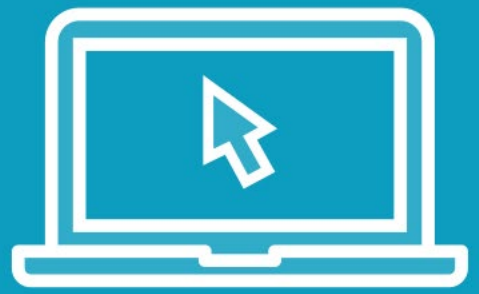
Inject the data reader

Inject the view model

Snap things together


Bootstrapper

Demo



Add a bootstrapper

Separate object composition and UI

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

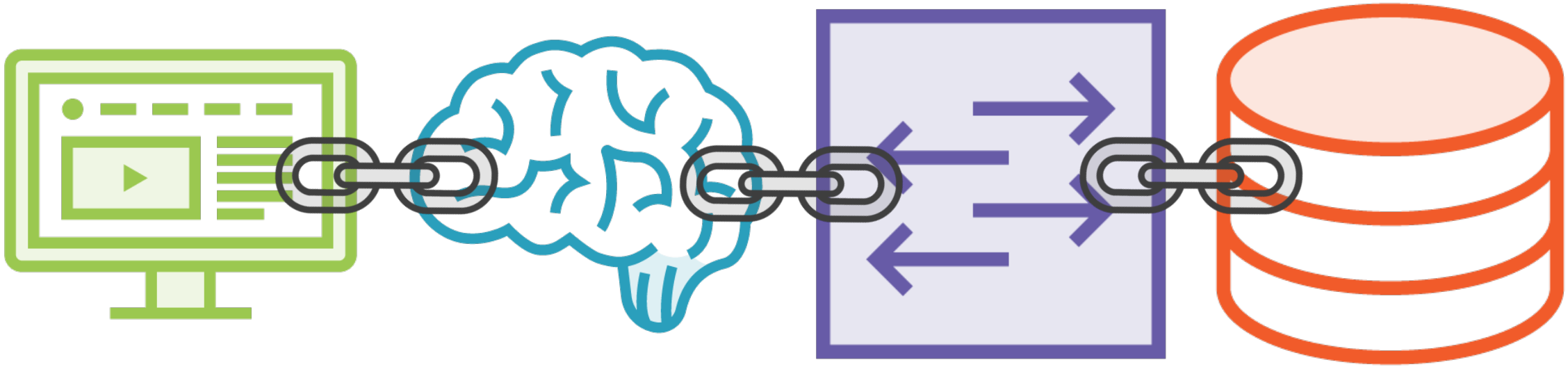
D

- Dependency Inversion Principle

View model

- No longer responsible for dependencies

Break Tight Coupling



View
(UI elements)

Presentation
(UI logic)

Data access

Data store

Violation

S • Single Responsibility Principle

View model responsibilities

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

S • Single Responsibility Principle

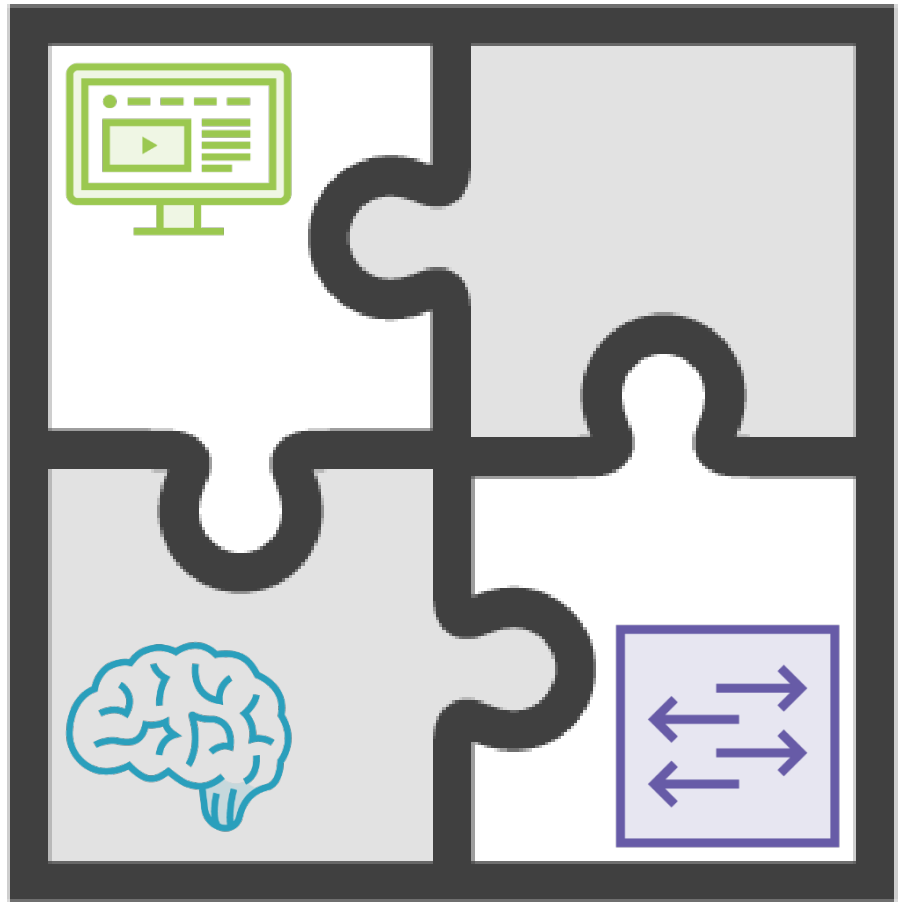
View model responsibilities

- Presentation logic

```
private static void ComposeObjects()
{
    var reader =
        new ServiceReader();

    var viewModel =
        new PeopleViewModel(reader);

    Application.Current.MainWindow =
        new PeopleViewerWindow(viewModel);
}
```



Using DI

Add some abstraction

Use constructor injection

Object composition

SOLID