Dynamic Loading & Unit Testing

Jeremy Clark
Developer Betterer

@jeremybytes www.jeremybytes.com

How & Why

Real-world scenarios
- Changing a data reader
- Multiple clients

Focus on important functionality

Change behavior without recompiling

Easier unit testing

Scenario 1: Changing a Data Reader

Application

Interface

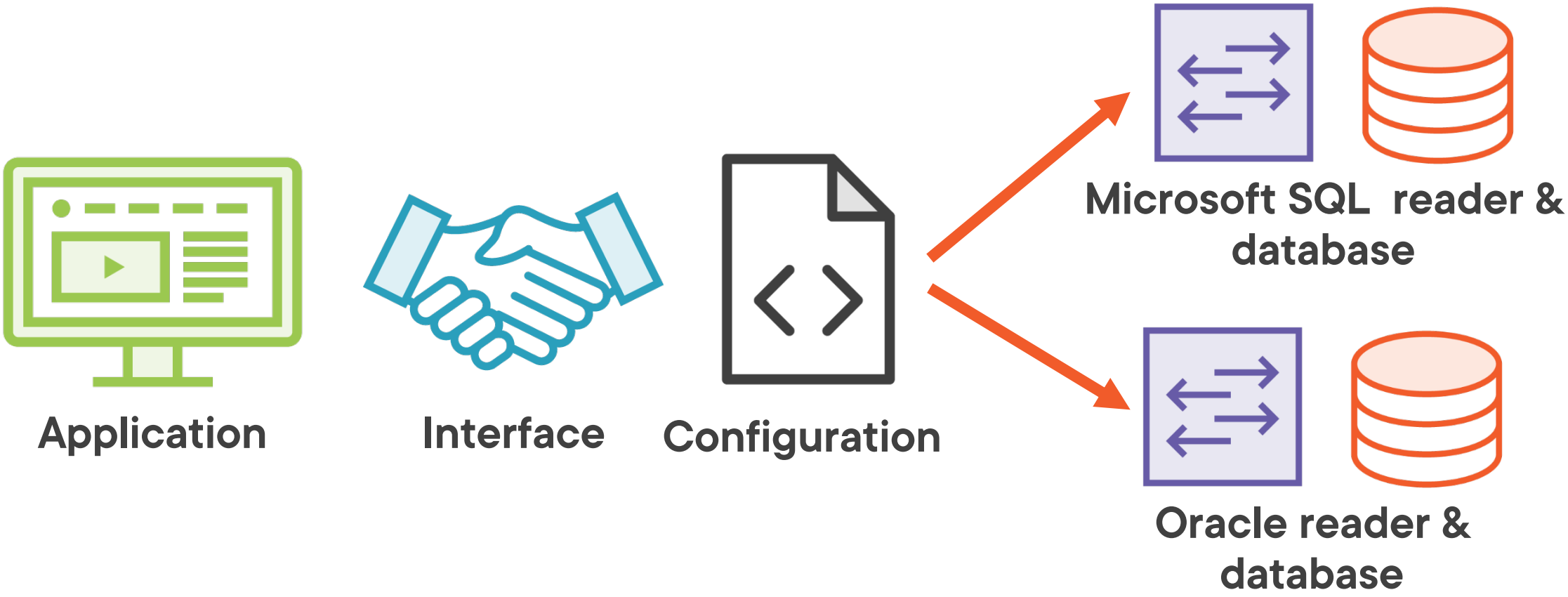
Microsoft SQL reader & database

Application

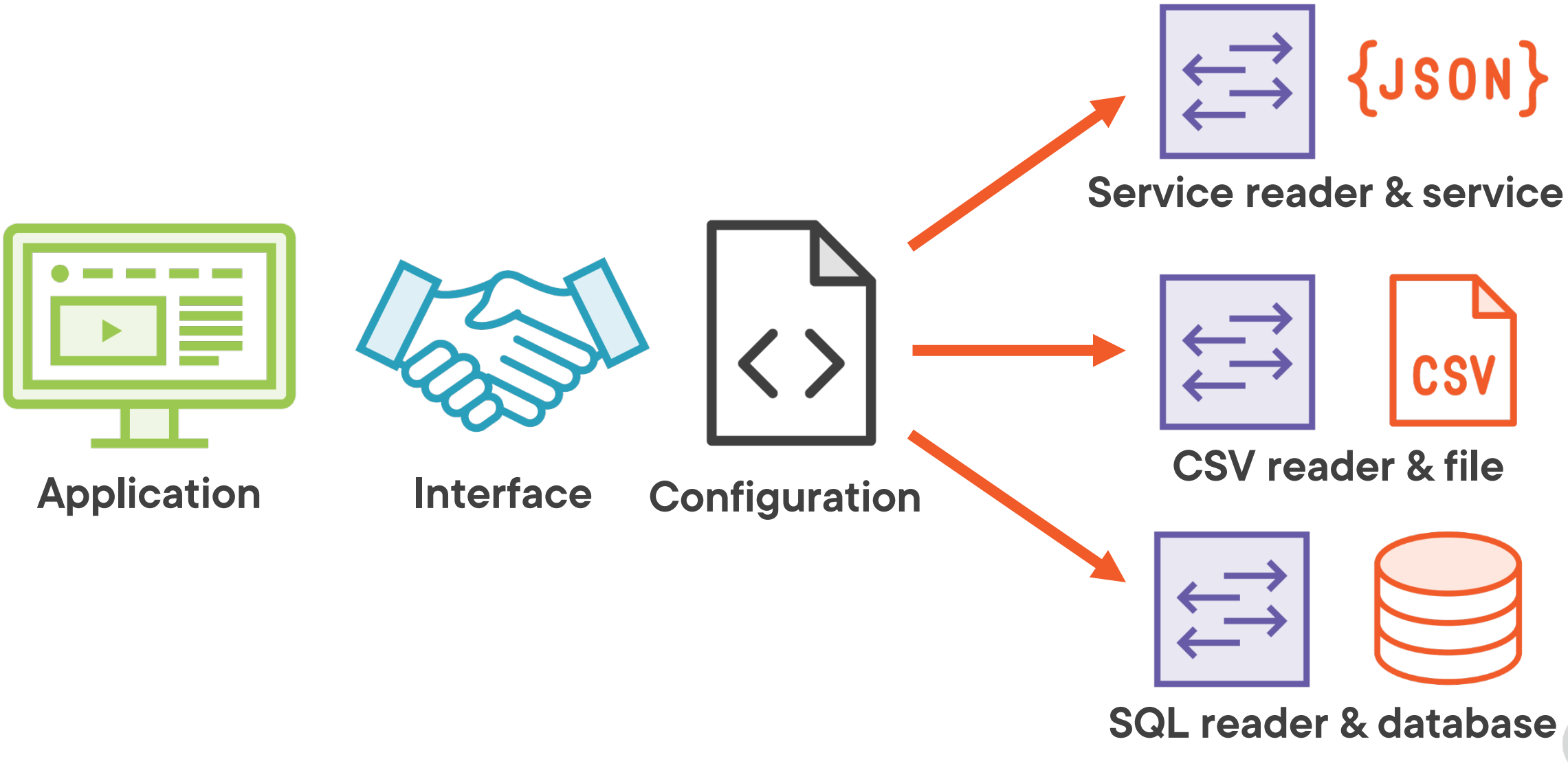
Interface

Oracle reader & database

Select a Data Reader with Configuration



Select a Data Reader with Configuration



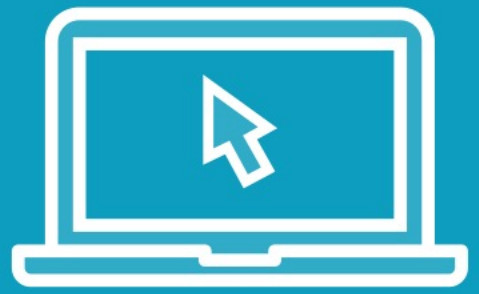
Additional Resources

https://bit.ly/3tYeAee

https://github.com/jeremybytes/csharp-interfaces-resources

Includes how to run the samples with Visual Studio Code

Demo

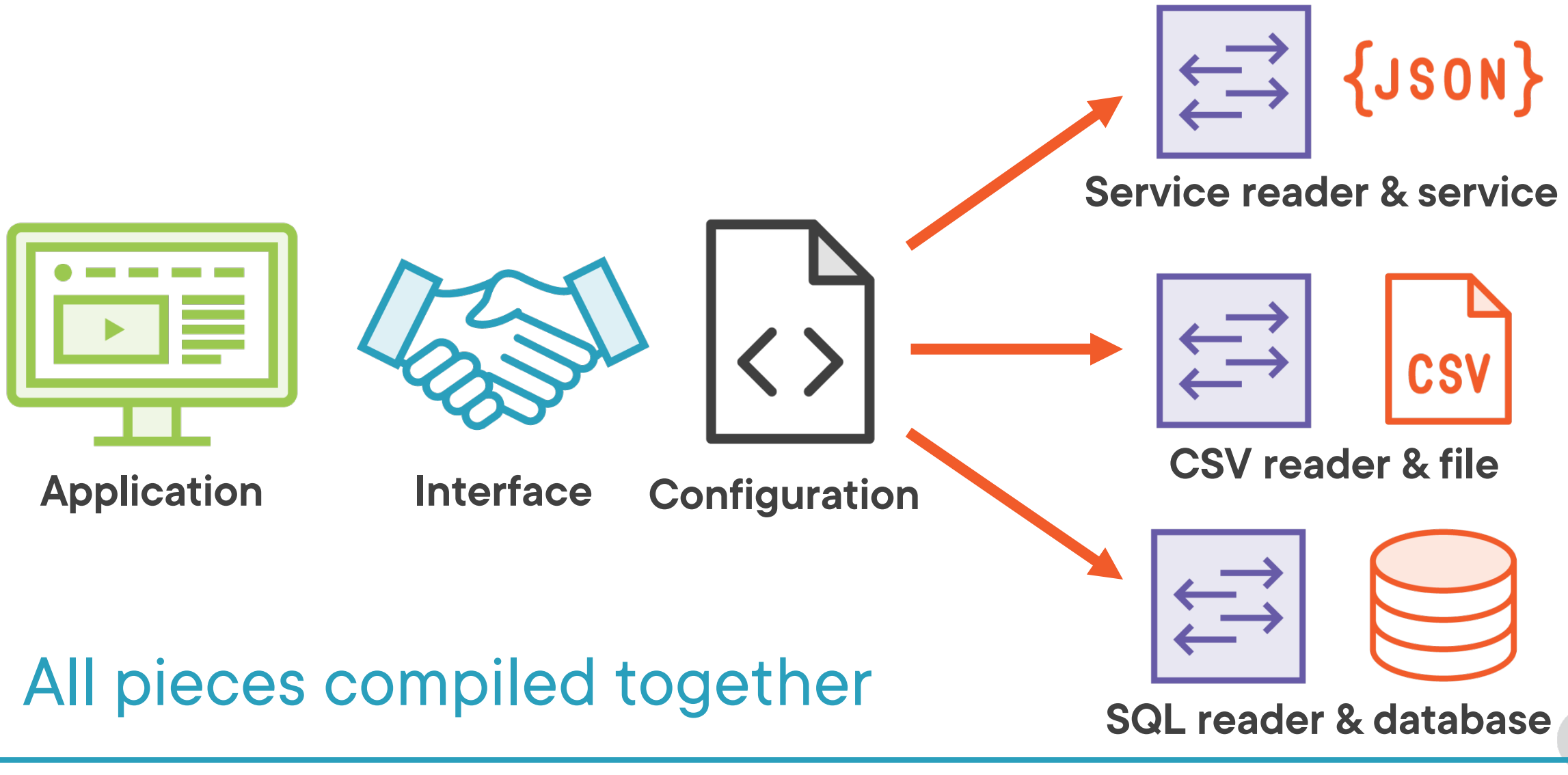


Add configuration for a data reader

Update the controller to use configuration

Update the user interface (view)

Scenario 2: Multiple Clients



Program to an abstraction rather than a concrete type

Program to an interface rather than a concrete class

```
public IActionResult PopulatePeopleView(string readerType)
{
    IPersonReader reader = readerFactory.GetReader(readerType);

    IEnumerable<Person> people = reader.GetPeople();

    ViewData["ReaderType"] = reader.GetType().ToString();
    return View("Index", people);
}
```

Program to an Interface

No references to concrete data reader types

Compile-time Factory

```
public IPersonReader GetReader(string readerType)
{
    switch (readerType)
    {
        case "Service": return new ServiceReader();
        case "CSV": return new CSVReader();
        case "SQL": return new SQLReader();
        default: throw new ArgumentException(...);
    };
}
```

Downsides with Multiple Clients

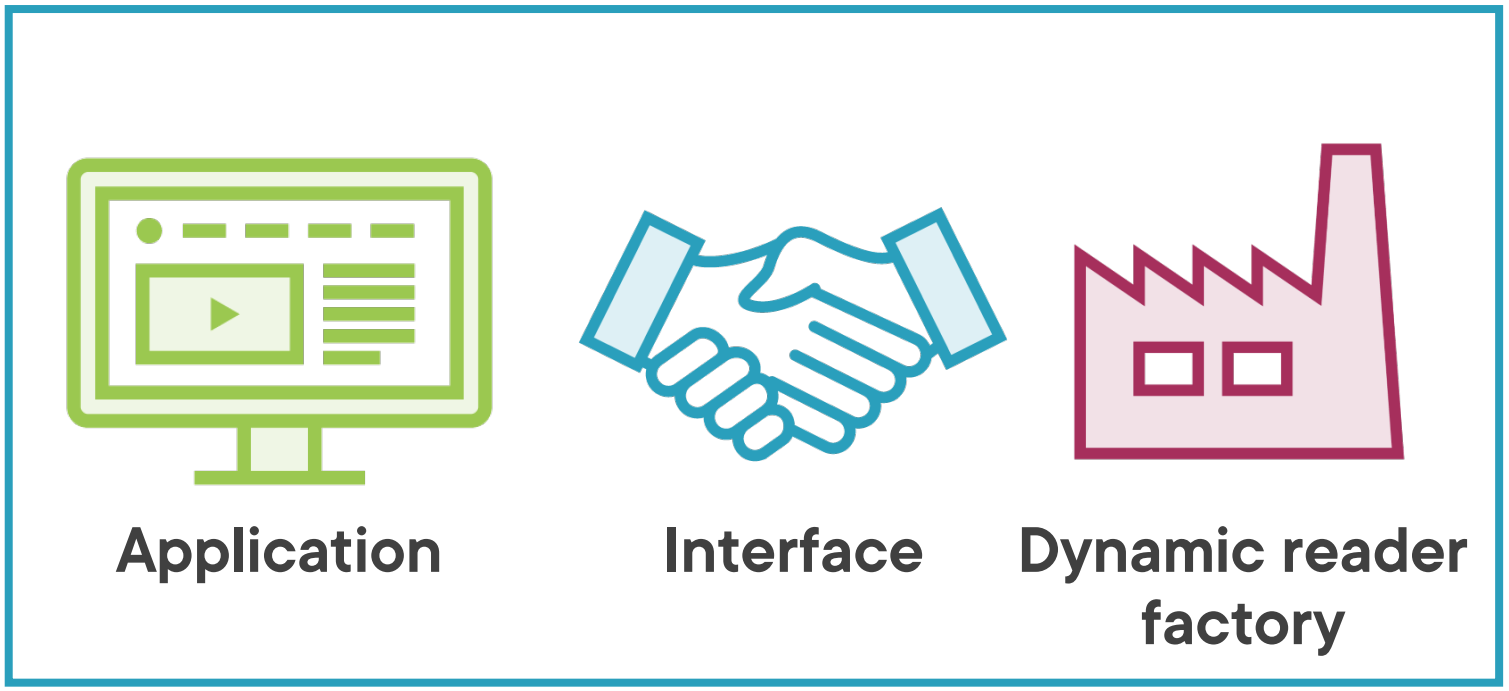
If a new client has different data storage, the entire application must be recompiled.

If an existing client changes their data storage, the entire application must be recompiled.

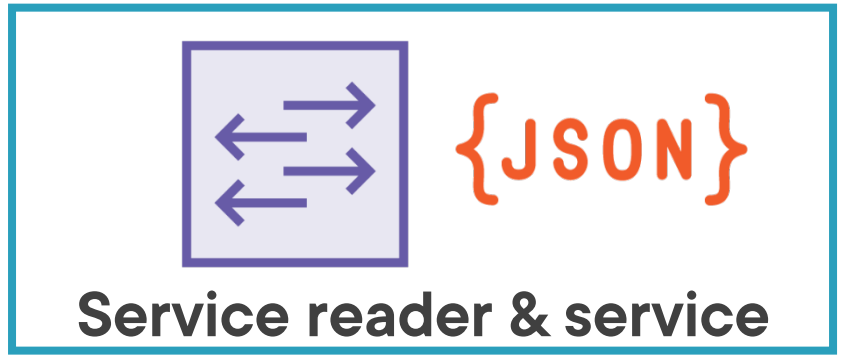
Each client has code for *all* of the data readers, even if they never need them.

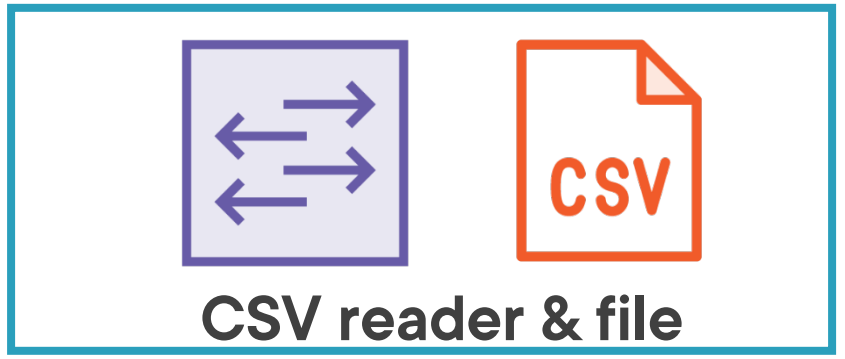
Different clients will potentially have different versions of the application, making support difficult.

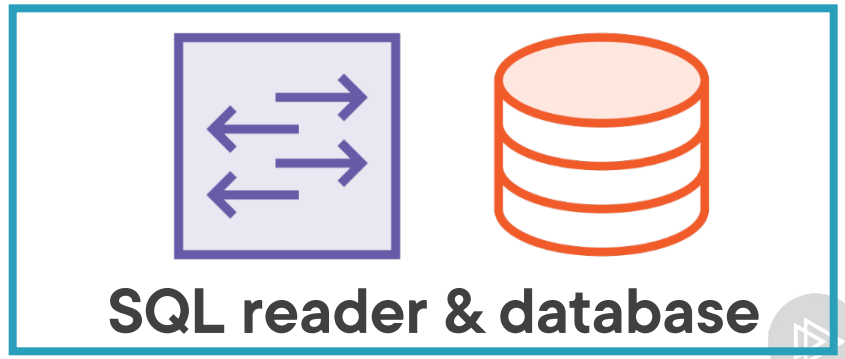
Separate Compilation and Deployment



Compiled separately







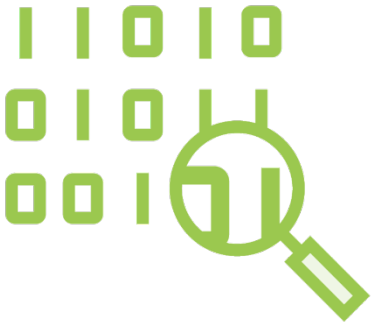
Dynamic Factory

Check configuration

Load the assembly

Look for the type

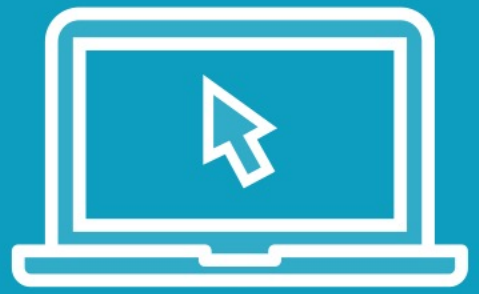
Create the data reader



Return the data reader

Don’t worry if you don’t understand the details. This is mainly to show what is possible with interfaces.

Demo



Review dynamic loading code

No compile-time references

Change data reader without recompiling

Results of Dynamic Loading

To add or change data readers, we provide the client with the new data reader and configuration.

Support is easier since all clients can be on the same version of the core application.

Incorrect configuration or data reader files lead to runtime errors.

Unit Testing

Testing pieces of functionality in isolation.

Interfaces help us isolate code
for easier unit testing.

What We Want to Test

```
public class PeopleController : Controller
{
    public IActionResult UseRuntimeReader()
    {
        IPersonReader reader = readerFactory.GetReader();
        IEnumerable<Person> people = reader.GetPeople();

        ViewData["Title"] = "Using a Runtime Reader";
        ViewData["ReaderType"] = reader.GetType().ToString();
        return View("Index", people);
    }
}
```

Things I Do Not Want to Deal With

```
public class PeopleController : Controller
{
    public IActionResult UseRuntimeReader()
    {
        IPersonReader reader = readerFactory.GetReader();
        IEnumerable<Person> people = reader.GetPeople();

        ViewData["Title"] = "Using a Runtime Reader";
        ViewData["ReaderType"] = reader.GetType().ToString();
        return View("Index", people);
    }
}
```

Dependency Injection

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.

Dependency Injection

The fine art of making things someone else's problem.

Injecting the Data Reader

```
public class PeopleController : Controller
{
    private IPersonReader reader;

    public PeopleController(IPersonReader dataReader)
    {
        reader = dataReader;
    }

    public IActionResult UseRuntimeReader()
    {
        IEnumerable<Person> people = reader.GetPeople();
        ... (some code left out here)
        return View("Index", people);
    }
}
```

No Reader Factory

Course Suggestion

Getting Started with Dependency Injection in .NET

Jeremy Clark

Fake Reader in Unit Tests

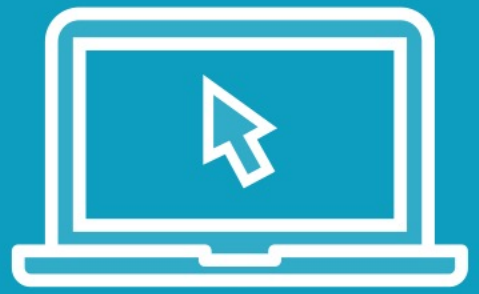
```
[TestMethod]
public void PeopleController_OnRuntimeReaderAction_ModelIsPopulated()
{
    IPersonReader reader = new FakeReader();
    var controller = new PeopleController(reader);

    ViewResult? result = controller.UseRuntimeReader() as ViewResult;
    Assert.IsNotNull(result, "Controller action failed.");

    var model = result?.Model as IEnumerable<Person>;

    Assert.IsNotNull(model, "Model is not populated");
    Assert.AreEqual(2, model?.Count(), "Unexpected number of items");
}
```

Demo



- Inject the data reader into the controller
- Use fake data reader for tests
- Unit test the controller functionality

How & Why

Real-world scenarios
- Changing a data reader
- Multiple clients

Focus on important functionality

Change behavior without recompiling

Easier unit testing