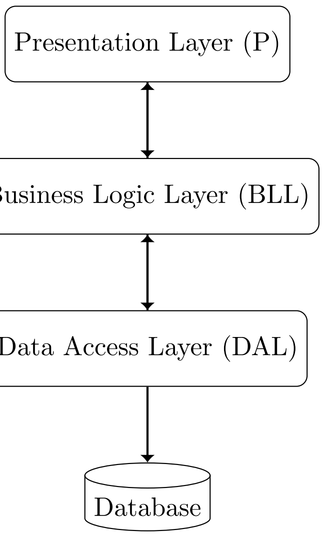
Dependency Injection

Joseph Roehm

March 26, 2025

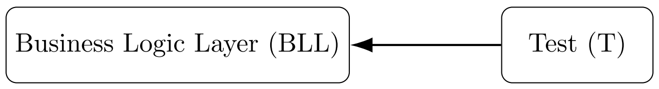
Note: Arrows represent dependencies.



The diagram above models the dependency relationships between the Data Access Layer, the Business Logic Layer, and the Presentation Layer.

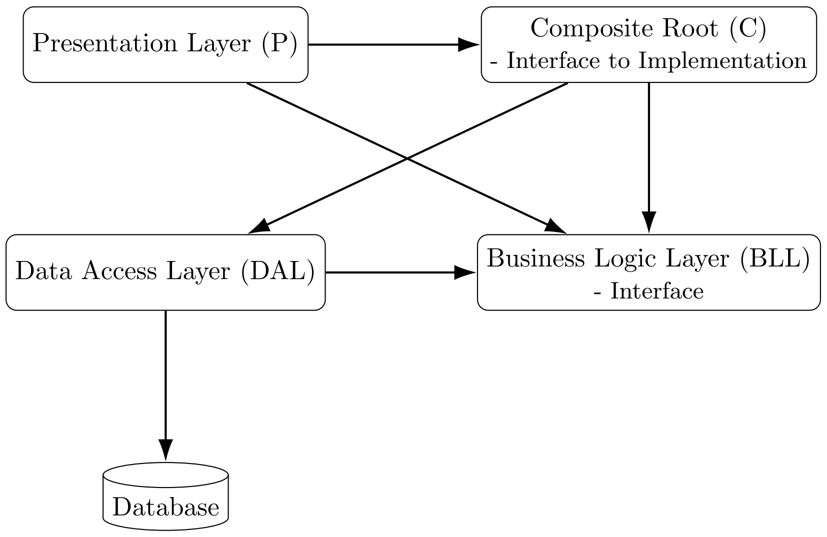
This presents an issue: How do developers independently test the BLL? With the current model, input cannot be easily validated by a test case due to the nondeterministic nature of external systems. For instance, a BLL that relies on the database to be in a certain state necessitates a complete build of that database to satisfy the expected output. *The database has to have the exact expected data to ensure the test is successful.* This can be costly in both money (if ran in the cloud) and time (if the database is large and requires time to build).

A better solution is to remove BLL dependencies on other systems. The ideal situation for testing is the following:



The solution to this problem is to use *dependency injection* to remove the dependencies the BLL has on other systems. This achieves our objective of isolating the BLL from external dependencies, allowing for much simpler unit tests.

The design implementation ultimately looks like the diagram below.



The primary addition is the *Composite Root (C)*. The Composite Root is a class that maps BLL *interfaces* with *implementations*. The interface defines what methods, poperies, or behaviors a class must implement without dictating how they are implemented. No logic is present in the interface. The implementation is the class that fulfills the interface requirements. It contains the actual *logic* that defines how the work is done. An interface can have many implementations (one-to-many relationship). The BLL contains the interface classes (for example, a logger class), and the implementation would reside in the infrastructure layer (i.e. external libraries or libraries we create alongside of the BLL).

In the BLL, an interface class looks like the following:

```
// Example C# interface
public interface ILogger
{
    void Log(string message);
}
```

The implementation could look like the following in the infrastructure layer or alongside the BLL in a separate file:

```
// Example C# implementation
public class SmartLogger : ILogger
{
    public void Log(string message)
    {
        // Business-logic-level logging implementation
    }
}
```

The composite root is usually stored in a `Startup.cs` or `Project.cs` file. First though, you must add the mapping your `IServiceCollection` registration list. First, add the mappings to your service collection:

```
// Example C# registration
public static class InfrastructureRegistration
{
    public static IServiceCollection AddInfrastructure(this
        IServiceCollection services)
    {
        services.AddScoped<ILogger, SmartLogger>();
        // other mappings
        return services;
    }
}
```

Then add the `AddInfrastructure` class to the `Startup.cs` file:

```
// Example C# startup file
public void ConfigureServices(IServiceCollection services)
{
    services.AddInfrastructure(); // calls your extension
        method
    services.AddControllers();
}
```