Purpose of MSBuildTaskBase Class

In an effort to make MSBuild Tasks more unit-testable, we are introducing this abstract base class to provide dependency injection support for the MSBuild Tasks classes. Enabling dependency injection will allow us to inject mock dependencies into the MSBuild Tasks classes for a better unit testing experience.

The `MSBuildTaskBase` abstract class provides a consistent way to implement dependency injection for the `Microsoft.Build.Utilities.Task` type. In a web service or command line application, the entry point into the program serves as a place to configure the dependency injection service collections and provider. Since there isn't an entry point like that for MSBuild Tasks, we have created an abstract class to handle the common elements of configuration.

MSBuildTaskBase class can be found here.

For more reading about dependency injection: - Architectural Principles: Dependency Injection - Dependency Injection in ASP.NET Core - Dependency Injection: Constructor Injection Behavior - PluralSight course: Dependency Injection in ASP.NET Core

How to Implement in an MSBuild Task

1. In your MSBuild Task class, inherit the `MSBuildTaskBase` class found the in `Microsoft.Arcade.Common` project.

   ```
   public class TaskClassName : MSBuildTaskBase
   {
       ...
   }
   ```

2. Implement the `ExecuteTask` method. Since the abstract base class uses reflection to find this method so that it can look up the dependencies being injected into the class, these are the requirements for its implementation:

   - Must be public
   - Must be named exactly `ExecuteTask`
   - Must return a boolean
   - Can accept any (or no) parameters, which will be resolved from the service collection configured by ConfigureServices

   `ExecuteTask` replaces the `Execute` method from the `Microsoft.Build.Utilities.Task` class. Any functionality you would include in the `Execute` method should go into `ExecuteTask` instead. This method also mimics the behavior of a constructor that would take in the dependencies for a class, and should include the interfaces of the dependencies being injected into the Task as parameters.

   ```
   public bool ExecuteTask(IThing thing, IPizza pizza, IHamburger hamburger)
   {
   ```

```
    ...
}
```

3. Implement `ConfigureServices` to contain the configuration for the Service Collection that will resolve the dependencies. Ensure that you have configured all dependencies, including the downstream dependencies that are used by the top-level dependencies being injected into the Task.

```
public override void ConfigureServices(IServiceCollection collection)
{
    collection.TryAddSingleton<IThing, ThingClass>();
    collection.TryAddSingleton<IPizza, PizzaClass>();
    ...
}
```

How to Use with Tests

Validating Dependency Injection Configuration

To ensure that the service collection has been properly configured, you should include a test to validate the registration of the dependencies. This test uses a helper class that is provided in the `Microsoft.DotNet.Internal.DependencyInjection.Testing` project.

The first parameter in `IsDependencyResolutionCoherent` attempts to configure the services in the Task to verify that the required services are configured correctly. The last parameter verifies that the top-level dependencies that are defined in `ExecuteTask`'s parameters are configured correctly.

```
[Fact]
public void AreDependenciesRegistered()
{
    MSBuildTaskClass task = new MSBuildTaskClass();

    var collection = new ServiceCollection();
    task.ConfigureServices(collection);
    var provider = collection.BuildServiceProvider();

    DependencyInjectionValidation.IsDependencyResolutionCoherent(
            s =>
            {
                task.ConfigureServices(s);
            },
            out string message,
            additionalSingletonTypes: task.GetExecuteParameterTypes()
        )
        .Should()
        .BeTrue(message);
}
```

Mocking Services for Task Unit Tests

1. Create the mocks for the services to be injected into the task, and add them to a new service collection to be used in the test.

   ```
   var collection = new ServiceCollection()
        .AddSingleton(thingMock.Object)
        .AddSingleton(pizzaMock.Object);
   ```

2. Pass that collection into the Task's `ConfigureServices` method. If the `ConfigureServices` method registers dependencies using `TryAdds`, it shouldn't add conflicting dependency registrations into the collection.

   ```
   MSBuildTaskClass task = new MSBuildTaskClass();
   task.ConfigureServices(collection);
   ```

3. Build the service provider and pass it into the method that will resolve the dependencies and invoke the `ExecuteTask` method.

   ```
   using var provider = collection.BuildServiceProvider();
   bool result = task.InvokeExecute(provider);
   ```

What's Happening Behind the Scenes

The `MSBuildTaskBase` class inherits `Microsoft.Build.Utilities.Task`, so it implements the `Execute` method that MSBuild Task classes implement. It uses reflection to accomplish resolving the services defined in the concrete class' `ExecuteTask` method.

The Task's `Execute` is traditionally the method that is called when running the task. It is required to be implemented when inheriting the `Microsoft.Build.Utilities.Task` class.

The `Execute` method serves as the Task's entry point, that handles the registration of dependencies that are defined in the concrete implementation of the `MSBuildTaskBase` class. In this method, a new `ServiceCollection` is created. It is passed into the `ConfigureServices` method to be populated with the configuration defined in the concrete implementation. Then, it builds the service provider that is passed into the `InvokeExecute` method.

Link to code

```
public override sealed bool Execute()
{
    ServiceCollection collection = new ServiceCollection();
    ConfigureServices(collection);
    using (var provider = collection.BuildServiceProvider())
    {
```

```
        return InvokeExecute(provider);
    }
}
```

The implementation of `ConfigureServices` on the concrete implementation of `MSBuildTaskBase` will configure the resolution of dependencies.

`MSBuildTaskBase` contains a virtual implementation of `ConfigureServices` that configures common dependencies, such as `IFileSystem` (if you want to abstract the File and Directory classes so they can be mocked out for tests) and a `TaskLoggingHelper` object called `Log`. It is expected that this method will be overridden.

`InvokeExecute` is where the reflection magic happens. It looks up the `ExecuteTask` method on the concrete implementation, looks up the parameters (the dependencies) that are defined in the method, and resolves the dependencies with the provider that was configured and built. Then it passes along the required dependency implementations into the method when it ultimately invokes it.

Link to code

```
public bool InvokeExecute(ServiceProvider provider)
{
    return (bool)GetExecuteMethod().Invoke(this, GetExecuteArguments(provider));
}
```

Was this helpful?