# 90322: ”SOLID Programming in C#”

## Workshop Description

**Wincubate ApS**

**07-03-2024**

V3.2.1

# Table of Contents

## Prerequisites

The present labs require the course files accompanying the course to be extracted in some directory path, e.g.

```
C:\Wincubate\90322
```

with Visual Studio 2022 with .NET 6 (or later) installed on the PC.

We will henceforth refer to the chosen installation path containing the lab files as *PathToCourseFiles* .

# Workshop A: "Creating a SOLID Domain Layer Structure"

## A.1: "Initial Setup and Inspection of Project"

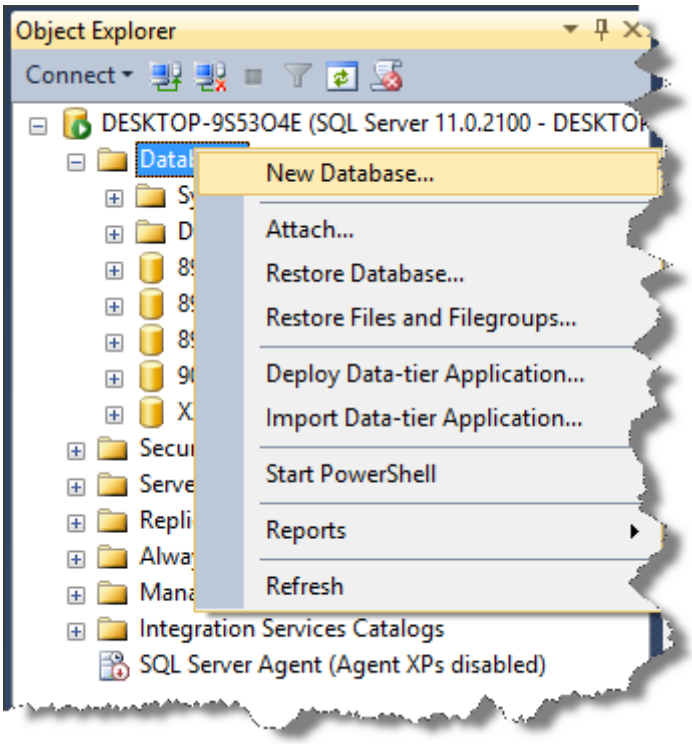
This step will get you set up for workshop development.

- Open the starter project in
  *PathToCourseFiles*`\Workshop\A.1\Starter` ,
  which contains the Visual Studio solution `Cinema`.
- Familiarize yourself with the pre-existing code and structure
  - Notice that a number of code files have already been added.

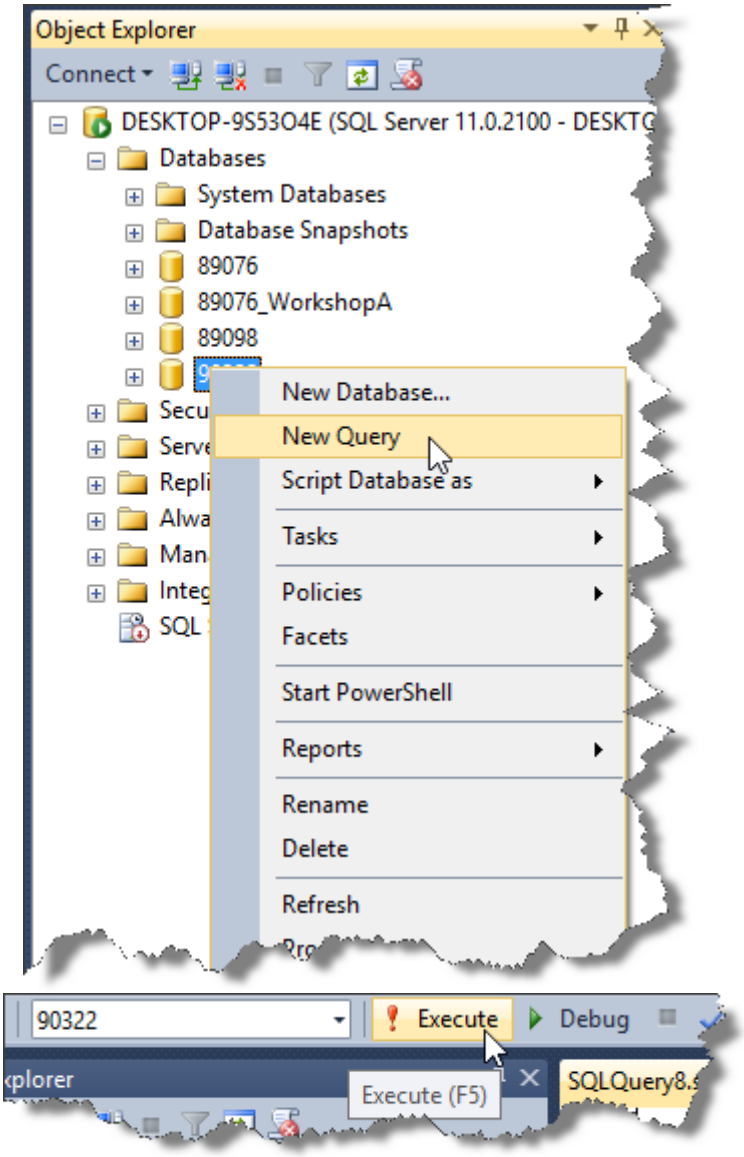
You will now create the database and tables used throughout this workshop in a SQL Server Express.

Note: You are free to use either SQL Server, SQL Server Express, or SQL LocalDB depending upon what is available to you, but you might have to adjust the connection strings in the projects accordingly.

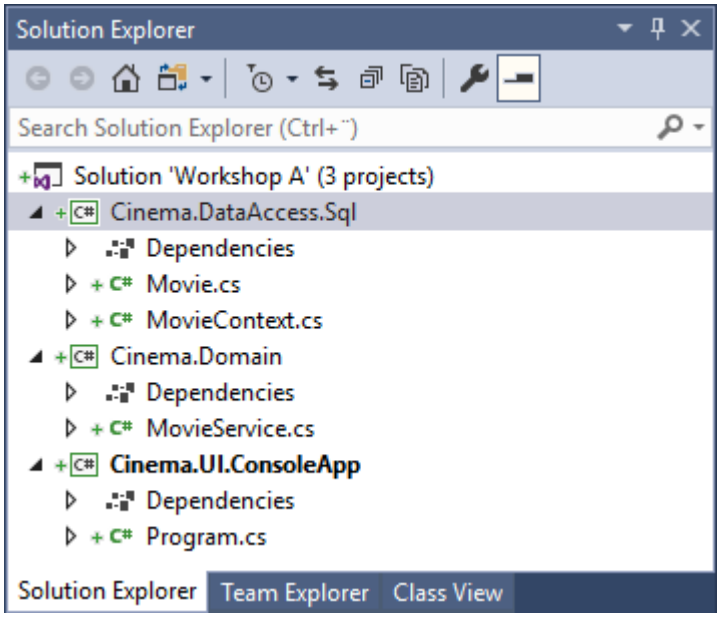
- Open SQL Management Studio and create a new database named `90322`
  - Right-click the `Database` node and select `New Database`…




  - Create the 90322 database using default settings.
- Generate the `Movies` SQL database from the scripts located in
  *PathToCourseFiles*`\Workshop\01 - Create Movies DB.sql`
  by executing this script on the database just created.
  - Choose `New Query` and drag the script file to the query window
  - Click `Execute` (or press F5)




You are provided with an existing Starter solution consisting of a number of distinct projects. An overview is provided here:



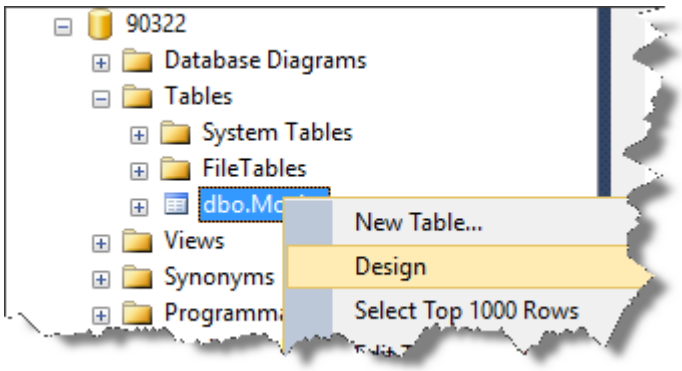

You change existing projects and add more projects to the above solution as you progress through the workshop. Class libraries are .NET Standard whereas the executable projects are .NET Core. References between the projects have already been set up.

- **Cinema.UI.ConsoleApp**
  - Is a console application which you will use to run your application.
- **Cinema.Domain**
  - Contains domain layer (or business logic) of your application.
- **Cinema.DataAccess.Sql**
  - Contains a SQL-based Data Access Layer unit tests for the **Cinema.Domain** project.

The solution currently does not compile! To make it compile and run, you will need to add the contents of the `Movie` class with properties reflecting the schema of the `Movies` table in the 90322 database, as it is currently empty:

```
public class Movie
{
    // TODO: Add properties
}
```

- Inspect the schema of the `Movies` tables by right-clicking the the table in SQL Management Studio and choosing `Design` :



- Add properties to the Movie class mapping to the appropriate .NET types, e.g.
    - `Id` should be of type `Guid`
    - `Year` should of type `short`
    - etc…

  Remember to make to nullable SQL types also nullable C# types, e.g.
    - `ImdbRating` should be `double?`
    - `TicketPrice` should of type `decimal?`
    - `LastUpdated` should of type `DateTime?`

When everything compiles;

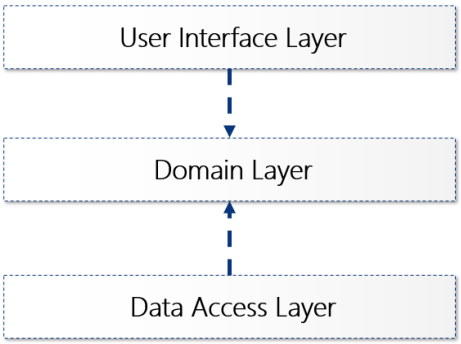
- Ensure that the **Cinema.UI.ConsoleApp** is set as startup project.
- Run the program and checks that the application displays a view of the movies in the database which are currently showing.

## A.2: "Data Access Layer with Repository"

Following the discussion in Module 02, you will now fix most of the problems with the current design. You will also implement the **Repository** pattern for improved separation, maintainability, and testability of the application.

Improve the application by completing the following steps:

- Fix the references between the layers such that it follows the guidelines presented in the module presentation:




  - Move classes and references for e.g. `Movie` and `MovieContext` to the right layers.
  - Note: If you want, you're allowed to split one or more layers into two projects, though this is in no way mandatory.
- Implement the **Repository** pattern for data access.
  - Feel free to implement whichever version of repository you see fit; Either of Simple, Queryable, Generic, … will do!
- Run your new, beautiful program and check that it still outputs the correct set of movies.

### If Time Permits…

- Since there is no particular reason to have all the unused members of `Movie` propagate to the higher layers of the system, modify the service to return a more focused domain type, e.g. `MovieShowing` containing only the relevant properties of `Movie`.
  - What would this type look like?
  - And what is the correct location for this type?
- Consider if it would be possible to introduce a Presentation Logic Layer with view models and supporting classes?
- Would such a step somehow simplify or accelerate creating other UIs in e.g. WPF or ASP.NET (or both) presenting the same information?

## A.3: "Test the Domain Layer and Change the Data Access Layer"

In this part of the workshop we will illustrate just how testable and adaptable the new application design is. In fact, we can supply positive answers to all three questions from the module discussion.

### Unit Testing the Movie Service

Firstly, we address the testability of our design.

- Create a unit test of the `MovieService` class ensuring that it returns exactly those movies where the `IsShowing` property is true
    - **Important**: To create a unit project, you can add an **MSTest Test Project (.NET Core)** project to the solution in Visual Studio
        - Depending upon your Visual Studio configuration, you might have to update the **MSTest.*** nuget packages after project is added
    - Note: During the test, you should not read any files, config, or access the SQL database.

### If Time Permits…

Secondly, we will illustrate the loosely-coupled nature of the design by swapping out the current SQL Data Access Layer for an XML-based implementation.

- Change the Data Access Layer to load the movies from an XML file with the following contents:

```
<?xml version="1.0"?>
<Movies xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Movie>
    <Id>180eb091-c529-4d57-8a9d-ce51d362a495</Id>
    <Name>XML as an exciting data source... Not!</Name>
    <Year>2019</Year>
    <Plot>The movie is set in a galaxy far, far away where XML is
considered exciting</Plot>
    <ImdbRating>4.2</ImdbRating>
    <TicketPrice>79</TicketPrice>
    <IsShowing>true</IsShowing>
    <LastUpdated>05/06/2019 11:19:54</LastUpdated>
  </Movie>
  <Movie>
    <Id>eb7031f8-ff4c-49a8-9920-5490c61c7c1c</Id>
    <Name>Singleton is an evil motherf****r</Name>
    <Year>2019</Year>
    <Plot>This is the story about a good guy from 1994 which turned evil
over time</Plot>
    <ImdbRating>9.9</ImdbRating>
    <TicketPrice>109</TicketPrice>
    <IsShowing>true</IsShowing>
    <LastUpdated>05/06/2019 11:19:54</LastUpdated>
  </Movie>
</Movies>
```

Note: The file is also available in

*PathToCourseFiles*\Workshop\02 – Movies.xml

- If you have not done so already, now might be a good time to introduce Solution Folders in Visual Studio for organizing the projects into the layers to which the belong.

### If Time Still Permits…

If you did not finish the *"If Time Permits"* in Workshop A.2, then you might have time to revisit it here:

- Since there is no particular reason to have all the unused members of `Movie` propagate to the higher layers of the system, modify the service to return a more focused domain type, e.g. `MovieShowing` containing only the relevant properties of `Movie`.
    - What would this type look like?
    - And what is the correct location for this type?
- Consider if it would be possible to introduce a Presentation Logic Layer with view models and supporting classes?
- Would such a step somehow simplify or accelerate creating other UIs in e.g. WPF or ASP.NET (or both) presenting the same information?

## A.4: "A Test of Time in the Domain Layer"

With the movie service working a nice and simple way, we will now add a common feature to the Domain Layer: Time…!

Friday is the top night of the week, so let's offer a great deal for customers.

- Implement a "Happy Hour on All Movies" campaign applying a discount of 50% off any movie for the entire duration of Fridays (and Fridays only!)
- Create unit tests asserting that your movie service provides the correct Happy Hour discount on Fridays – and no discounts any other day of the week.

Sounds simple, right? 😊

## A.5: "Create a Null User Context"

Your boss wants to introduce a Members' Club, where users can register and log in. When properly authenticated Club Members will receive an (additional) 10% discount on all pricing – even on top of Happy Hour!

Your boss wants you to complete the movie service today and put it in production tomorrow. You think he's crazy because you have not at all discussed how users should log in or authenticate. And you don't know if the service is going to run in an WPF app or as a web site. The authentication schemes for each are totally different!

He claims that you can easily implement and test(!) the movie service today and worry about the actual authentication scheme next month. He instructs you to use the following enum for user levels:

```
public enum UserRole
{
    Customer,
    ClubMember,
}
```

and the following interface for the users:

```
public interface IUserContext
{
    bool IsInRole(UserRole role);
}
```

Your objectives are now clear:

- Complete implementation of the service providing Club Members with a 10% discount.
- Test that your completed service satisfies both the new (and the existing) ticket price specifications.

Note: Feel free to use any test framework you find appropriate; MS Test, NUnit, or similar.

## Workshop B: "Creating a Service with User Messaging"

In this workshop you will use what you've learnt in Workshop A and apply it in a related, but more realistic scenario. Workshop B will only outline the steps you need to go through, so you are free to implement the steps in any order and in any way, you feel is the right one. Moreover, you will receive less guidance and need to improvise more.

You will work on the Admin interface for the cinema creating users and notifying users with emails and SMS messages that they have been created. Your instructor will show you the completed application before you begin.

The workshop will go through several steps, which you need to complete before your service is capable of sending physical messages. Essentially, your completed application – once you have completed step B.1 to B.6 below – will use the Create User Service as follows:

1. Allow the administrator to enter the Name, Email, Phone, and Preferred Language for the user to create in the UI above.
2. Invoke the `CreateUserService` which will use a `Messenger` component to perform the major part of the work by
    a. Retrieving the `UserIsCreatedOk` template for the preferred language from the database.
    b. Using the `TextSubstitutor` to substitute the proper parameters of the `Body` (not the `Subject`).
    c. Creating an `InstantiatedMessage` object.

Physically sending this `InstantiatedMessage` object to the user using the supplied contact information.

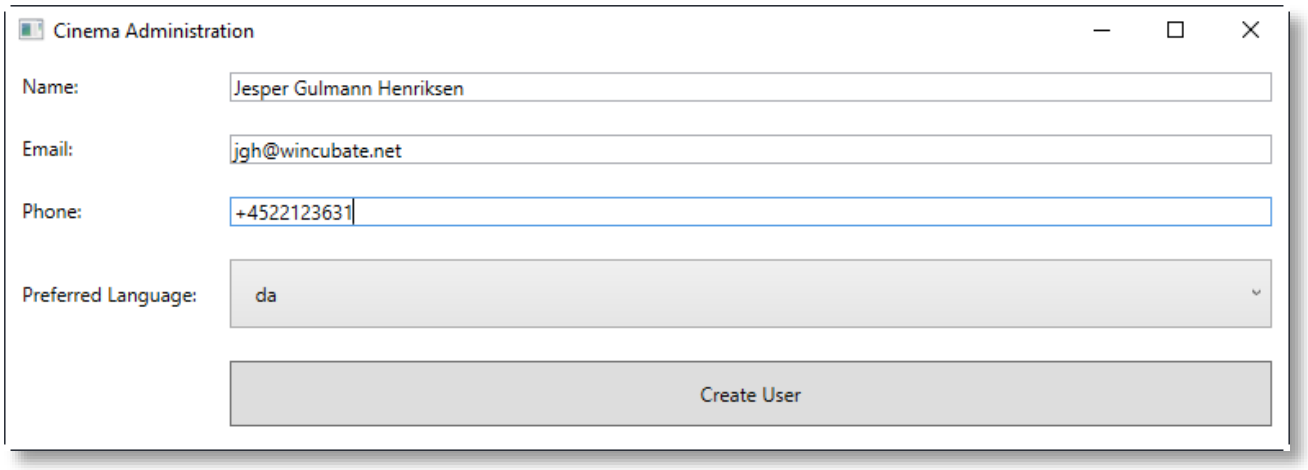
You are now ready to begin development, which starts by creating the DB table first.

### B.1: "Initial Setup and Inspection of Project"

This step will get you set up for workshop development.

- Open the starter project in
  *PathToCourseFiles*`\Workshop\B.1-B.6\Starter` ,
  which contains the Visual Studio solution `Admin`.
- Familiarize yourself with the pre-existing code and structure
    - Notice that a number of projects containing a number of classes and interfaces have already been added.

There is an existing UI in the **Admin.UI.WpfApp** project:




The UI is virtually complete, except that you – as you progress through the steps of the workshop – have to complete the **Composition Root** to make the app run, once you have completed the Create User Service in the other steps of the workshop.

Your completed application will use the Create User Service as follows:

3. Allow the administrator to enter the Name, Email, Phone, and Preferred Language for the user to create in the UI above.
4. Invoke the `CreateUserService` which will use a `Messenger` component to perform the major part of the work by
    a. Retrieving the `UserIsCreatedOk` template for the preferred language from the database.
    b. Using the `TextSubstitutor` to substitute the proper parameters of the `Body` (not the `Subject`).
    c. Creating an `InstantiatedMessage` object.
    d. Physically sending this `InstantiatedMessage` object to the user using the supplied contact information.

You will now create the database and tables used throughout this workshop in an SQL Server Express in a manner very similar to Workshop A.

- Generate the `Movies` SQL database from the scripts located in
  *PathToCourseFiles*`\Workshop\03 - Create MessageTemplate DB.sql`
  by executing this script on the database created in Workshop A.

Note:
You don't need to use the Message Template data from the SQL database until later..!

Notice that the project comes equipped with a preexisting `IMessageTemplateRepository` in **Admin.Domain.Interfaces** and an associated `InMemoryMessageTemplateRepository` implementation in **Admin.Domain**, which supplies the same data as in the database table. You can use this for all practical purposes throughout this workshop.

If time permits at the end of the workshop, you can implement real SQL database access for `MessageTemplate` instances in B.6.

### Background

The following data class is already defined in the **Admin.Domain.Interfaces** project:

```
/// <summary>
/// POCO class capturing a single message template definition
/// from the underlying database.
/// </summary>
public class MessageTemplate
{
    /// <summary>
    /// Gets or sets the Id of the <see cref="MessageTemplate"/>.
    /// </summary>
    public int Id { get; set; }

    /// <summary>
    /// Gets or sets the kind of the <see cref="MessageTemplate"/>.
    /// </summary>
    public string Kind { get; set; }

    /// <summary>
    /// Gets or sets the culture identifier of the
    /// <see cref="MessageTemplate"/>.
    /// </summary>
    public string Culture { get; set; }

    /// <summary>
    /// Gets or sets the Subject text of the
    /// <see cref="MessageTemplate"/>.
    /// </summary>
    public string Subject { get; set; }

    /// <summary>
    /// Gets or sets the Body Template of the
    /// <see cref="MessageTemplate"/>.
    /// </summary>
    public string BodyTemplate { get; set; }

    /// <summary>
    /// Provides a string representation of the
    /// <see cref="MessageTemplate"/>.
    /// </summary>
    /// <returns>String representation of the current
    /// <see cref="MessageTemplate"/>.</returns>
    public override string ToString() =>
$"{Id}\t{Culture}\t{BodyTemplate}";
}
```

## B.2: "Create a Text Substitution Class"

You will now create a text substitution mechanism in the **Admin.Domain** project for providing actual contents for the substitution placeholders of the retrieved message templates.

- In the **Admin.Domain** project, complete the `TextSubstitutor` class with the following signature:

```
public string Substitute( MessageTemplate messageTemplate,
                          IEnumerable<object> parameters )
```

- Test your implementation validating it against the two pre-existing unit tests in the `TextSubstitutorTest` class of the **Admin.Domain.Test** project to make sure it works correctly.

Note: Don't advance to the next step until these unit tests have been run and successfully passed.

### Background

The exception-related types `AdminException` and `AdminExceptionReason` are already defined in the **Admin.Domain.Interface** project.

## B.3: "Create a Messenger Class Physically Transmitting Messages"

You will now create a `Messenger` class which accepts a message from the client and sends the substituted (or "resolved") version of the message as a message to the specified recipient.

The message from the client to be sent will be supplied in the shape of a `Message` object. That type is already defined in the **Admin.Domain** project as follows:

```
/// <summary>
/// Implementation class for messages to be resolved
/// and sent using the <see cref="Messenger"/>.
/// </summary>
public class Message
{
    /// <summary>
    /// Gets or sets the recipient of the <see cref="Message"/>.
    /// </summary>
    public User Recipient { get; set; }

    /// <summary>
    /// Gets or sets the kind of the <see cref="MessageTemplate"/>
    /// to use.
    /// </summary>
    public string MessageTemplateKind { get; set; }

    /// <summary>
    /// Gets or sets the Culture of the
    /// <see cref="MessageTemplate"/> to use.
    /// </summary>
    public string Culture { get; set; }

    /// <summary>
    /// Gets or sets the parameters to be substituted into the
    /// <see cref="MessageTemplate"/>.
    /// </summary>
    public IEnumerable<object> Parameters { get; set; }
}
```

The `Messenger` implementation is an instance of the **Façade** pattern essentially wrapping these three steps to getting the message sent:

a. Retrieving the `UserIsCreatedOk` template for the preferred language from the database.
b. Using the `TextSubstitutor` to substitute the proper parameters of the Body (not the `Subject`).
c. Creating an `InstantiatedMessage` object.
d. Physically sending this `InstantiatedMessage` object to the user using the supplied contact information.

The latter type `InstantiatedMessage` is also predefined:

```
/// <summary>
/// Class for instantiated instances, i.e. it has
```

```
/// already been resolved and substituted from a
/// <see cref="Message"/>.
/// </summary>
public class InstantiatedMessage
{
    /// <summary>
    /// Unique application-identifier for message instance.
    /// </summary>
    public Guid InstanceId { get; }

    /// <summary>
    /// Message string subject.
    /// </summary>
    public string Subject { get; }

    /// <summary>
    /// Message string body.
    /// </summary>
    public string Body { get; }

    /// <summary>
    /// Creates a new <see cref="InstantiatedMessage"/> with
    /// the message contents specified in <paramref name="body"/>.
    /// </summary>
    /// <param name="subject">Message string subject.</param>
    /// <param name="body">Message string body.</param>
    public InstantiatedMessage(string subject, string body)
    {
        Subject = subject;
        Body = body;
        InstanceId = Guid.NewGuid();
    }
}
```

The user-related type `User` is already defined in the **Admin.Domain.Interface** project.

Your job is now to perform as to construct the code of steps a) – d).

- Complete the `Messenger` class with the `SendAsync()` method with the following signature:
  ```
  public Task SendAsync( Message message )
  ```
- Implement the physical transmitter functionality of `Messenger` as a strategy of the **Strategy** pattern where each concrete strategy in this workshop transmits the `InstatiatedMessage` by sending physically by some distinct medium

Note: In the next parts of this exercise, you will implement concrete strategies for transmitting the messages via Email and SMS. But since this will be the topic of the subsequent two parts of the workshop, you do not yet have the required information for sending the messages via Email or SMS, so you should for the time being only implement a **Null Object** strategy for transmitting messages.

However, before moving on to the final parts of Workshop B sending the messages as real emails or SMS'es to users, you would like to ensure that the functionality of the message is 100% correct without sending a physical email every time we run our unit tests. Furthermore, the tests should be carried out without connecting to the database.

Complete the two unit tests in the **Admin.Domain.Test** project proving that the `Messenger` class works correct by using a **Test Spy** concrete strategy for transmitting messages.

1. Sending of a single message will result in the correctly substituted template being transmitted if the correct template exists in the database.
2. Sending of a single message will throw an exception if the correct template does not exist in the database.

- Ensure that all tests in **Admin.Domain.Test** succeed before moving on to the last two parts of Workshop B. 😊

## B.4: "Create an Email Transmission Strategy"

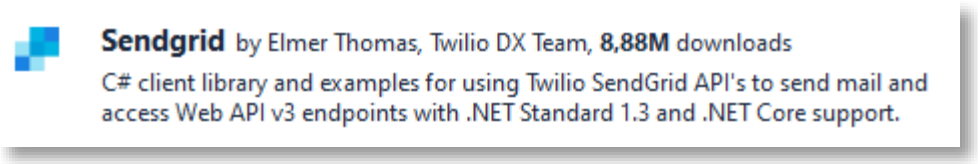
In the previous part of the workshop, we created the transmission infrastructure but did not specify a way to actually send the messages. We will now adapt that infrastructure to send the messages physically as emails to the user.

- Create an Email transmission strategy using the SendGrid Email API.
- Test your implementation by sending yourself the messages as email messages.

### Background

The SendGrid API documentation is available at https://app.sendgrid.com/guide/integrate/langs/csharp . but unfortunately, you need to register for an account in order to see this. Instead, we have snagged the documentation needed and will include it below.

In order to use the SendGrid API for sending email you must first include their nuget package into your project:




Once it is included, the SendGrid documentation states that you should use the following code:

```
var client = new SendGridClient(apiKey);
var from = new EmailAddress("test@example.com", "Example User");
var subject = "Sending with SendGrid is Fun";
var to = new EmailAddress("test@example.com", "Example User");
var plainTextContent = "and easy to do anywhere, even with C#";
var htmlContent = "<strong>and easy to do anywhere, even with C#</strong>";
var msg = MailHelper.CreateSingleEmail(from, to, subject, plainTextContent, htmlContent);
var response = await client.SendEmailAsync(msg);
```

The API string for you to use is:

```
?G.qzFlZizMT7OAfM2QwQ8CcA.Ehn_m0bUARW?hgvIijD1y1ldcnV?WhK0l21Rgc3PlYg
```

Your instructor will supply you with the remaining letters substituting the missing ?'s in the above code.

## B.5: "Create an SMS Transmission Strategy"

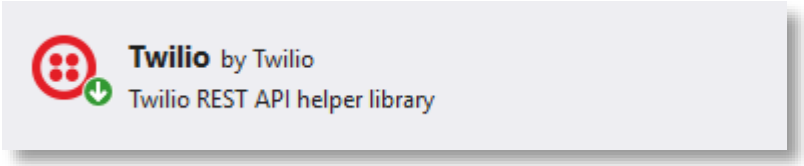
In the previous part of the workshop, we created the transmission infrastructure and instantiated a strategy for sending the messages as emails. We will now see how easy it is to extend the functionality to send SMS test messages instead.

- Create a SMS transmission strategy using the Twilio SMS API.
- Test your implementation by sending yourself the messages as SMS messages.

### Background

See the Twilio SMS API documentation at https://www.twilio.com/docs/sms/api/message-resource .

In order to use the Twilio API for sending SMS messages you must include their nuget package into your project:




Once it is included, you can send an SMS message using the following code:

```
string _accountSid = "ACa5?64844f11c4152c5e4db4bc202c7??";
string _authToken = "4f14?6d?4826993?6c15a02a8605882b";

TwilioClient.Init(_accountSid, _authToken);
MessageResource mr = await MessageResource.CreateAsync(
    to: new PhoneNumber("<phone number>"),
    from: new PhoneNumber("+4676???9439"),
    body: "<contents of SMS>"
);
```

Your instructor will supply you with the remaining digits substituting the missing ?'s in the above codes.

## B.6: If Time Permits… "Create Repository Classes for Accessing MessageTemplates SQL Table"

You can now access the predefined set of message templates which you restored to the database in B.1.

- Open SQL Management Studio and inspect the contents of the `MessageTemplates` table.
- Create a suitable `MessageTemplatesContext` class.
- Use the **Repository** pattern to create
    - Repository interface
    - Repository implementation class

  for accessing the `MessageTemplate` instances in the SQL database from .NET code.

Note: You decide which version of Repository to implement – simple or generic! But aim to place all the classes and files you author in the appropriate pre-existing projects.

## B.7: If Time Permits…

- What happens if the Create User Service should send *both* email and sms messages every time?
- How would we cope with that in a SOLID setting?

## B.8: If Time Permits…

- If you feel up to it, go ahead and implement that the Create User Service will actually store the created users appropriately in the database – not just send them messages! 😊

# Workshop C: "Introducing Dependency Injection Containers"

In this workshop we will add dependency injection containers to the mix of the previous two workshops. You can use any DI container that you see find interesting, but we have assumed that you will be using Unity developed by Microsoft.

## C.1: "Adding Unity DI to Services"

Firstly;

- Add the Unity DI Container to your solution to Workshop A
  - Add it to the Console App UI version first.
  - Use the **Register-Resolve-Release** pattern.
  - Note: If you want to test your injection with your XML provider, you might need to use the
    - `IUnityContainer.RegisterFactory()` or
    - `IUnityContainer.RegisterInstance()` methods,

    if you supplied the file name as a construction parameter.

Secondly;

- Add the Unity DI Container to all automatic tests in your solution to Workshop A
  - How is this slightly different from Unity injection in your Composition Root of your app?
  - Does it make sense to do injections everywhere in the tests?

You're free to slightly change the design if it makes everything more beautiful, of course.

### If Time Permits…

- Add the Unity DI Container also to your solution to Workshop B.
- Add the Unity DI Container to all automatic tests in your solution to Workshop B.

What are the similarities and/or differences in Composition Root, registrations, etc. between Workshop A and B?

## C.2: "Interception"

You have or or less completed your Admin application of Workshop B, so now it moves on to the Test Phase. The assigned tester would like to test the sending of email and SMS messages as close as possible to the production settings. This means that she would like to test with real customers having real names, but have all messages sent to her email address and her phone, respectively.

She would like to have to able to set up her email and phone in the Composition Root without modifying any part of the remaining system.

- Create a `RedirectionMessageTransmissionStrategyProxy` doing the work for her.
  - You choose if you want to use the DI Container version or the Pure DI version.
  - If she specifies a redirection email address that should be used. If not, use the real one.
  - If she specifies a redirection phone number that should be used. If not, use the real one.
- Set up the Composition Root for your own credentials and test it
  - You're also free to see if you can unit test it, of course! 😉

Note: If you decide to go for the DI Container version, you will probably need to consult the following Stack Overflow post by Mark Seemann:

https://stackoverflow.com/questions/36048646/dependency-injection-composition-root-and-decorator-pattern/36101994#36101994

## (For Later) Workshop D: "If Time Permits…"

This workshop is intended for you to try out if you want some additional exercises and challenges after the course completes.

You can complete the parts of Workshop D in any order – or indeed not at all! 😊

### D.1: "Implement a Logging Mechanism for Messenger"

We will now implement diagnostic features to convey what happens inside the `Messenger` class.

- Use the **Abstract Factory** pattern to define a logger and corresponding logger factory which logs entries to a file in an interface similar to:

```
public interface ILogger
{
  void Info( string message, params object[] additional );
  void Warn( string message, params object[] additional );
  void Error( string message, params object[] additional );
}
```

- Add a `FileLogger` class, which logs to a file in the current directory with the name passed to `ILoggerFactcory.Create()`.
- The log lines should log the message and each of the additional parameters with one of the strings below prepended:
  - `"INFO  -- "`
  - `"WARN  -- "`
  - `"ERROR -- "`

  More specifically, a successful run of the Messenger should produce the following two lines of log output in a file called Messenger.log:

```
INFO  -- [[Messenger]] Sending message //
{"Recipient":{"Name":"Jesper","Email":"jgh@wincubate.net","Phone":"+
4522123631","PreferredCulture":"da"},"MessageTemplateKind":"UserIsCr
eatedOk","Culture":"da","Parameters":["Jesper"]}

INFO  -- [[Messenger]] Message instance successfully sent //
{"InstanceId":"37789d8b-e8ba-4803-83f2-
6334b791907f","Subject":"Konto oprettet","Body":"Kære Jesper. Din
konto hos Cinemas 'R Us er nu oprettet. :-)"}
```

- Modify `Messenger` to accept an `ILoggerFactory` at construction time and introducing logging into the `SendAsync()` method to make an invocation perform a log of a successful send (as above).
- Note: In order for the unit tests to compile and run, you will need to use the **Null Object** pattern to device a logging mechanism suitable for unit tests.

Run the application and check that everything is correctly working.

- What was the impact of adding the logger and factory throught the application?

A logger which logs to the console is similarly important for console applications. Since you already have a logging infrastructure set up, you should

- Add a `ConsoleLogger` class, which logs to a file in the current directory with the name passed to `Create()`.
    - The log lines should in the same format as `FileLogger`, but without the `"INFO  -- "`, `"WARN  -- "`, or `"ERROR -- "` strings prepended.

  Instead, info logs should appear as green text in the Console. Similarly, Warn and Error appear in yellow and red, respectively.
- As the formatting for both the `FileLogger` and `ConsoleLogger` are almost the same, you might refactor the existing implementation to a variation of the **Template Method**.

## D.2: "Interception for introducing reusable aspects"

With dependency injection set up in a SOLID application, many behaviors can be orthogonally added seamlessly using Interception.

Investigate the powers of Interception by

- Implementing resilience and retry strategies for sending messages such as
    - Circuit Breaker
    - Retry With Exponential Backoff

  Many such strategies can be implemented in a very simple fashion using the Polly nuget package:
  https://www.nuget.org/packages/Polly/
- Implementing reusable caching mechanisms for various repositories.