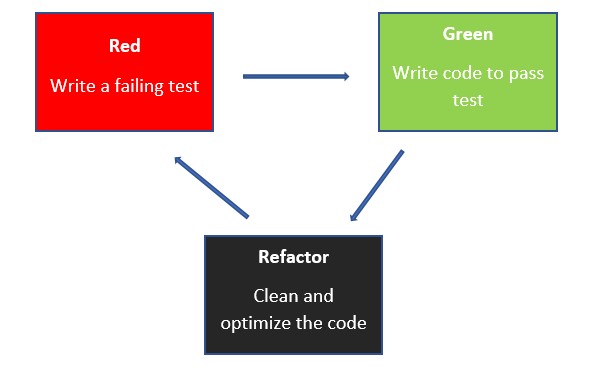
Test-Driven Development

Test-Driven Development, or TDD for short, is a method used to write tests before we start our implementation. Before you start, you might list the requirements that need to be fulfilled in your application.

We use RED – GREEN – REFRACTOR Cycle in TDD

Take the first requirement and write a failing test. The test fails and it is RED as you haven’t developed it yet.

But the test describes already what your code should do to fulfill the requirement. Now you need to make the test GREEN by writing the necessary code to make the test pass. After you write the code to make the test green you need to Refactor the code. Maybe you can simplify the code or extract few code lines into a method to make code more readable and maintainable. After you are done with the first requirement, you can continue with the next requirement. This means you iterate the entire cycle (Red -> Green -> Refactor) for another requirement and so on.



**Advantages**



1. TDD makes you think with the needed API from the beginning. You need to think about what classes, properties, API’s are needed. This will usually lead to great API design.
2. After you know the class and properties, another big advantage is that you need to think about what the code should do than how it should do. As you start with test you don’t need to have any idea about implementation. You just need to write a test for what the code should do. After writing the test you can think of requirements and their development.
3. While thinking of your requirements, you get quick feedback about your requirements by running the test. The fact that you get quick feedback means you even don’t need a fully working application at all. You just need a class library to build your business logic and don’t need the entire project.
4. This helps you create modular code. You can decouple the dependencies from the beginning and TDD makes you do that from the beginning. This decoupling of dependency makes you write a modular code by isolating the dependencies like a database that is not ready yet and the web API isn't ready when beginning the development.
5. This leads to a maintainable codebase, as you will have one test per requirement. You can write code to add new functionality and run all the unit tests to ensure that the existing code doesn’t break. You can be confident about your new code as well as the existing code.
6. These tests will serve as good documentation. For example, the test for the code written by others will help you understand why the code has been written.

**Disadvantage**

The only disadvantage is that TDD is not so easy to start by writing tests for beginners. In fact, TDD is an art that every developer should master.

Let’s take a simple scenario where a user needs to book a ticket. The user needs to fill a form with basic details to book the ticket.

*Ticket Booking Solution*

*TicketBookingCore (TicketBookingRequestProcessor) (.Net Core Class Library)*

*TicketBookingCore.Tests (TicketBookingRequestProcessorTests) (XUnit test project)*

* Getting started with TDD
* Testing and implementing business logic.
* Adding features in asp.net core app.

Let’s first start with creating a test project named TicketBookingCore.Test for business logic. We will start writing the first failing test and continue from that.

Then let’s work on adding additional business logic by implementing tests and iterating through the TDD cycle. We have more complex requirements that will force us to decouple dependencies and we need to mock those classes while writing the test.

 Using TDD to write business logic

**Requirements**

1. Response should contain the same values as request after booking.

2. Booking should be saved to database.

Understand the First Requirement

The user will submit a form to book a ticket which will make a call to TicketBookingRequestProcessor to book a ticket. The processor must return the same data after the booking is successful. The TicketBookingRequestProcessor will use Book to book a ticket and it will receive TicketBookingRequest as input and return TicketBookingResponse as result.



This simple requirement is good to start with TDD.

First Step

1. Create a new C# XUnit test project named TicketBookingCore.Tests.
2. As you need to test TicketBookingProcessor Class, create a new class named TicketBookingRequestProcessorTests.
3. Create a first test method as ShouldReturnTicketBookingResultWithRequestValues.
4. Mark the method with [Fact] attribute to indicate it as a test.
5. Now create a processor instance as TicketBookingRequestProcessor and press Ctrl + . to create a class in a new file.

namespace TicketBookingCore.Tests

{

    public class TicketBookingRequestProcessorTests

    {

        [Fact]

        public void ShouldReturnTicketBookingResultWithRequestValues()

        {

            var processor = new TicketBookingRequestProcessor();

        }

    }

}

Create a TicketBookingRequest with FirstName, LastName and Email properties and set the values and again press Ctrl + . to create the class in new file.

[Fact]

public void ShouldReturnTicketBookingResultWithRequestValues()

{

    var processor = new TicketBookingRequestProcessor();

    var request = new TicketBookingRequest

    {

        FirstName = "Rama",

        LastName = "Sree",

        Email = "sreeram.vsnet@hotmail.com"

    };

}

Add the following line

TicketBookingResponse response = processor.Book(request);

and press Ctrl + . to generate the Book method in TicketBookingRequestProcessor class.

Next, we need to assert if the input and output are equal.

To Assert the input and output, first, create TicketBookingResponse class with the same properties as TicketBookingRequest.

Modify the Book Method in TicketBookingRequestProcessor to return TicketBookingResponse.

internal class TicketBookingRequestProcessor

{

    public TicketBookingRequestProcessor()

    {

    }

    internal TicketBookingResponse Book(TicketBookingRequest request)

    {

      throw new NotImplementedException();

    }

}

We are all set with AAA (Arrange, Act, and Assert) of unit tests.

We can implement According to AAA

[Fact]

public void ShouldReturnTicketBookingResultWithRequestValues()

{

    // Arrange

    var processor = new TicketBookingRequestProcessor();

    var request = new TicketBookingRequest

    {

        FirstName = "Rama",

        LastName = "Sree",

        Email = "sreeram.vsnet@hotmail.com"

    };

    // Act

    TicketBookingResponse response = processor.Book(request);

    // Assert

    Assert.NotNull(response);

    Assert.Equal(request.FirstName, response.FirstName);

    Assert.Equal(request.LastName, response.LastName);

    Assert.Equal(request.Email, response.Email);

}

Open Test Explorer. You can now see the test is listed in the test explorer. Now Click on the Run (Green Triangle) button. You can see that the test fails

## Write Code to make test Green

 Implemented the below code in Book Method of TicketBookingRequestProcessor class.

internal TicketBookingResponse Book(TicketBookingRequest request)

{

    return new TicketBookingResponse

    {

        FirstName = request.FirstName,

        LastName = request.LastName,

        Email = request.Email

    };

}

Now run the tests we see the tests pass

## Refactor to improve the code

1. Create New .Net Core Class Library Named TicketBookingCore and move TicketBookingRequest, TicketBookingRequestProcessor and TicketBookingResponse into this project.
2. Fix Namespaces of the files in TicketBookingCore project.
3. Change the access modifier of all classes and methods to the public.
4. Create a base class TicketBookingBase and move the properties from TicketBookingRequest and TicketBookingResponse and inherit from TicketBookingBase class.
5. Add a reference to TicketBookingCore project in the TicketBookingCore.Tests project.
6. Now Build the solution and run the tests again.
7. The test should pass.

So we separated our DLL and Test Projects

Let’s write another test using TDD to quickly verify that request is not null while calling a Book method.

[Fact]

public void ShouldThrowExceptionIfRequestIsNull()

{

    // Arrange

    var processor = new TicketBookingRequestProcessor();

    // Act

    var exception = Assert.Throws<ArgumentNullException>(() => processor.Book(null));

    // Assert

    Assert.Equal("request", exception.ParamName);

}

Now if you run the above from test explorer, the test will fail with NullReferenceException instead of ArgumentNullException

Now let’s write the minimum required code to make the test pass.

public TicketBookingResponse Book(TicketBookingRequest request)

{

    if (request is null)

    {

        throw new ArgumentNullException(nameof(request));

    }

    return new TicketBookingResponse

    {

        FirstName = request.FirstName,

        LastName = request.LastName,

        Email = request.Email

    };

}

The next step is to refactor.

Since we need TicketBookingRequestProcessor in both the test, we can remove that from both the test and move it to TicketBookingRequestProcessorTests constructor and use that in our test methods.

public class TicketBookingRequestProcessorTests

{

    private readonly TicketBookingRequestProcessor \_processor;

    public TicketBookingRequestProcessorTests()

    {

        \_processor = new TicketBookingRequestProcessor();

    }

    [Fact]

    public void ShouldReturnTicketBookingResultWithRequestValues()

    {

        // Arrange

        var request = new TicketBookingRequest

        {

            FirstName = "Rama",

            LastName = "Sree",

            Email = "sreeram.vsnet@hotmail.com"

        };

        // Act

        TicketBookingResponse response = \_processor.Book(request);

        // Assert

        Assert.NotNull(response);

        Assert.Equal(request.FirstName, response.FirstName);

        Assert.Equal(request.LastName, response.LastName);

        Assert.Equal(request.Email, response.Email);

    }

    [Fact]

    public void ShouldThrowExceptionIfRequestIsNull()

    {

        // Act

        var exception = Assert.Throws<ArgumentNullException>(() => \_processor.Book(null));

        // Assert

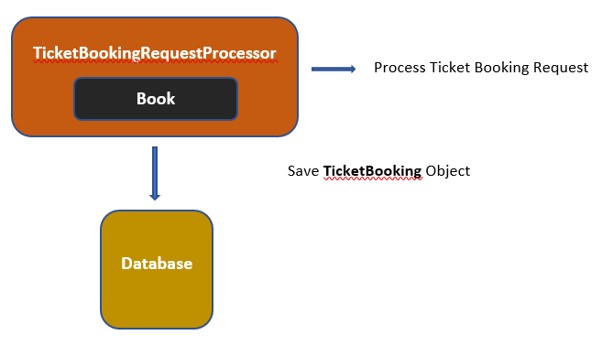
        Assert.Equal("request", exception.ParamName);

    }

}

## Understand the second Requirement

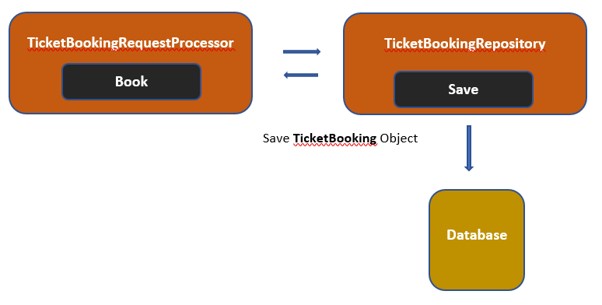
Now we need to save the booking to the database. To save to database we need to modify the book method to save the booking to the database and return the booking request values.



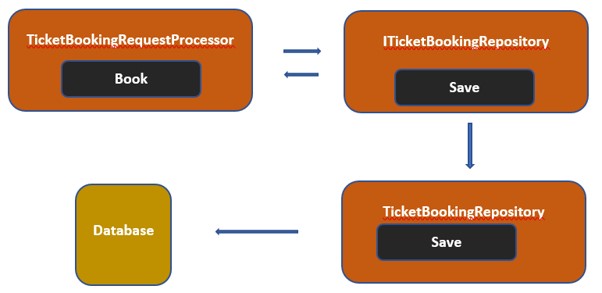
## Decouple the Dependencies

If you look at the above image, TicketBookingRequestProcessor has too many responsibilities. One is to process the booking request and another is to save it to the database.

But this violates the Single Responsibility Principle (SRP) – which says a class should have a single responsibility. So, to adhere to the SRP principle, we need to move the save to database logic to a separate class like TicketBookingRepository.



We can now save the TicketBooking object to the database using the TicketBookingRepository class. But now TicketBookingRequestProcessor depends on the TicketBookingRepository class to save to the database. This is not for the unit test, as the test needs to run in isolation. So here comes the Dependency Inversion Principle (DI), which says a class should always depend on abstraction, not on implementation. We can implement this by introducing a new interface ITicketBookingRepository. This interface implements TicketBookingRepository and saves to the database.



So now TicketBookingRequestProcessor depends on ITicketBookingRepository and we don’t need to worry about the database. This means to write a test we can create a mock (fake) object and use that to save to database and we can verify by this from the mock object that Save method is called at least once. This is how we can use interface to decouple the dependencies.

Now let’s create a failing Red unit test,

1. Create a new test method ShouldSaveToDatabase.
2. Create a request object and pass it to the processor save method.

[Fact]

public void ShouldSaveToDatabase()

{

    // Arrange

    var request = new TicketBookingRequest

    {

        FirstName = "Rama",

        LastName = "Sree",

        Email = "sreeram.vsnet@hotmail.com"

    };

    // Act

    TicketBookingResponse response = \_processor.Book(request);

}

Now to save to database, we need ITicketBookingRepository and that needs to be injected to TicketBookingRequestProcessor.

1. Add ITicketBookingRepository to TicketBookingCore project
2. Add Save() method with TicketBooking object as a parameter.
3. Press Ctrl + . to create a TicketBooking class and inherit from the TicketBookingBase class.

public interface ITicketBookingRepository

{

    void Save(TicketBooking ticket);

}

Now we need a mock object for ITicketBookingRepository. We can use the mock library to fake the repository.

1. Add \_ticketBookingRepositoryMock = new Mock<ITicketBookingRepository>(); to TicketBookingRequestProcessorTests constructor.
2. press Ctrl + . to download install Moq nuget package and repeat to add a private readonly field Mock<ITicketBookingRepository> \_ticketBookingRepositoryMock for that repository
3. Pass the field to TicketBookingRequestProcessor constructor and press Ctrl + . to add that as a parameter to TicketBookingRequestProcessor class.

public class TicketBookingRequestProcessorTests

{

    private readonly Mock<ITicketBookingRepository> \_ticketBookingRepositoryMock;

    private readonly TicketBookingRequestProcessor \_processor;

    public TicketBookingRequestProcessorTests()

    {

        \_ticketBookingRepositoryMock = new Mock<ITicketBookingRepository>();

        \_processor = new TicketBookingRequestProcessor(\_ticketBookingRepositoryMock.Object);

    }

}

public class TicketBookingRequestProcessor

{

    public TicketBookingRequestProcessor(ITicketBookingRepository ticketBookingRepository)

    {

    }

}

Now we need to write a setup in mock repository to make a callback when Save is called in that repository. And we need to assert that the Save method in the repository is called at least once and verifies the properties in the callback object.

public void ShouldSaveToDatabase()

{

    // Arrange

    TicketBooking savedTicketBooking = null;

    \_ticketBookingRepositoryMock.Setup(x => x.Save(It.IsAny<TicketBooking>()))

        .Callback<TicketBooking>((ticketBooking) =>

        {

            savedTicketBooking = ticketBooking;

        });

    var request = new TicketBookingRequest

    {

        FirstName = "Rama",

        LastName = "Sree",

        Email = "sreeram.vsnet@hotmail.com"

    };

    // Act

    TicketBookingResponse response = \_processor.Book(request);

    // Assert

    \_ticketBookingRepositoryMock.Verify(x => x.Save(It.IsAny<TicketBooking>()), Times.Once);

    Assert.NotNull(savedTicketBooking);

    Assert.Equal(request.FirstName, savedTicketBooking.FirstName);

    Assert.Equal(request.LastName, savedTicketBooking.LastName);

    Assert.Equal(request.Email, savedTicketBooking.Email);

}

Now we are done with test setup. Run the test and the test should fail.

 IF we run test cases it will fail some test cases

## Writing code to make the test pass

Now we need to write the minimum code to make the test pass which is to create a private readonly field for ITicketBookingRepository make a call to Save method in repository inside the Book method in processor class.

public class TicketBookingRequestProcessor

{

    private readonly ITicketBookingRepository \_ticketBookingRepository;

    public TicketBookingRequestProcessor(ITicketBookingRepository ticketBookingRepository)

    {

        \_ticketBookingRepository = ticketBookingRepository;

    }

    public TicketBookingResponse Book(TicketBookingRequest request)

    {

        if (request is null)

        {

            throw new ArgumentNullException(nameof(request));

        }

        \_ticketBookingRepository.Save(new TicketBooking

        {

            FirstName = request.FirstName,

            LastName = request.LastName,

            Email = request.Email

        });

        return new TicketBookingResponse

        {

            FirstName = request.FirstName,

            LastName = request.LastName,

            Email = request.Email

        };

    }

}

Now run to check he test case pass

## Refactor the code

Now we can improve the code by doing some refactoring.

1. According to Do Not Repeat Principle (DRY), we should avoid repeating the same code
2. In the TicketBookingRequestProcessorTests class, the same TicketBookingRequest object is constructed and used in two methods. This can be moved to the TicketBookingRequestProcessorTests constructor and can be made as a private readonly field.
3. In the TicketBookingProcessor class, we can see that the mapping of properties is done twice. This mapping can be extracted into a generic method.

public class TicketBookingRequestProcessorTests

{

    private readonly TicketBookingRequest \_request;

    private readonly Mock<ITicketBookingRepository> \_ticketBookingRepositoryMock;

    private readonly TicketBookingRequestProcessor \_processor;

    public TicketBookingRequestProcessorTests()

    {

        \_request = new TicketBookingRequest

        {

            FirstName = "Rama",

            LastName = "Sree",

            Email = "sreeram.vsnet@hotmail.com"

        };

        \_ticketBookingRepositoryMock = new Mock<ITicketBookingRepository>();

        \_processor = new TicketBookingRequestProcessor(\_ticketBookingRepositoryMock.Object);

    }

    [Fact]

    public void ShouldSaveToDatabase()

    {

        // Arrange

        TicketBooking savedTicketBooking = null;

        \_ticketBookingRepositoryMock.Setup(x => x.Save(It.IsAny<TicketBooking>()))

            .Callback<TicketBooking>((ticketBooking) =>

            {

                savedTicketBooking = ticketBooking;

            });

        // Act

        \_processor.Book(\_request);

        // Assert

        \_ticketBookingRepositoryMock.Verify(x => x.Save(It.IsAny<TicketBooking>()), Times.Once);

        Assert.NotNull(savedTicketBooking);

        Assert.Equal(\_request.FirstName, savedTicketBooking.FirstName);

        Assert.Equal(\_request.LastName, savedTicketBooking.LastName);

        Assert.Equal(\_request.Email, savedTicketBooking.Email);

    }

}

public class TicketBookingRequestProcessor

{

    private readonly ITicketBookingRepository \_ticketBookingRepository;

    public TicketBookingRequestProcessor(ITicketBookingRepository ticketBookingRepository)

    {

        \_ticketBookingRepository = ticketBookingRepository;

    }

    public TicketBookingResponse Book(TicketBookingRequest request)

    {

        if (request is null)

        {

            throw new ArgumentNullException(nameof(request));

        }

        \_ticketBookingRepository.Save(Create<TicketBooking>(request));

        return Create<TicketBookingResponse>(request);

    }

    private static T Create<T>(TicketBookingRequest request) where T : TicketBookingBase, new ()

    {

        return new T

        {

            FirstName = request.FirstName,

            LastName = request.LastName,

            Email = request.Email

        };

    }

}