Lab: Testing and Debugging ASP.NET MVC Core Web Applications

# Scenario

The Photo Sharing application is in the early stages of development. However, frequent errors are hindering the productivity of the development team. The senior developer advises that you intercept exceptions and other flaws as early as possible. You have been asked to perform unit tests of the PhotoController to ensure that all scenarios work as expected and to avoid problems later in the web application development life cycle. You have also been asked to ensure that when critical errors occur, developers can obtain helpful technical information.

# Objectives

After completing this lab, you will be able to:

* Refactor the controller by using a Repository Pattern
* Perform unit tests of the components of an MVC web application.
* Configure an exception handling strategy for an MVC web application.
* Use Visual Studio debugging tools against a web application.

**Estimated Time**: 90 minutes

# Exercise 1: Refactor the Photos Controller and the PhotoGalleryViewComponent by using a Repository Pattern

## Scenario

The PhotosController and the PhotoGalleryViewComponent make direct use of the PhotoSharingApplicationContext. To better separate the Application concerns from the Infrastructure concerns, you decide to use a Repository Pattern. This will help to better understand what to test at a controller level.

In this exercise, you will:

* Create a repository for the Photos model
* Move the infrastructure code from the controller / view component to the repository class
* Use the repository from the controller and the view component

The main tasks for this exercise are as follows:

1. Create an IPhotosRepository Interface
2. Create a PhotosRepository class that implements the IPhotosRepository Interface
3. Modify the controller to use the repository class
4. Modify the view component to use the repository class
5. Register the repository class as a service

### Task 1: Create an IPhotoRepository interface

1. In the Models folder, add a new Interface named IPhotoRepository
2. Add a GetPhotosAsync
   1. Input: int number=0
   2. Output: Task<List<Photo>>
3. Add a FindAsync
   1. Input: int id
   2. Output: Task<Photo>
4. Add a CreateAsync
   1. Input: Photo photo
   2. Output: Task
5. Add a RemoveAsync
   1. Input: int id
   2. Output: Task
6. Save the file

### Task 2: Create a PhotoRepository class the implements the IPhotoRepository interface

1. In the Models folder, add a new class named PhotoRepository
2. Implement the IPhotoRepository interface
3. Add a private readonly field of type PhotoSharingApplicationContext
4. Add a public constructor that accepts a PhotoSharingApplicationContext parameter and saves the parameter in the private field created in step 3
5. Implement the CreateAsync
   1. Open the PhotosController class
   2. Locate the Create action with the HttpPost constraint
   3. Transfer the code that uses the PhotoSharingApplicationContext from the controller to the repository
6. Implement the FindAsync
   1. In the PhotosController, locate the Details action
   2. Transfer the code that uses the PhotoSharingApplicationContext from the controller to the repository
   3. In the repository class, return the photo
7. Implement the GetPhotosAsync
   1. Open the PhotoGalleryViewComponent class
   2. Transfer the code that makes use of the context from the view component to the repository
   3. In the repository class, return the list of photos
8. Implement the RemoveAsync
   1. In the PhotosController, locate the Delete action
   2. Transfer the code that uses the PhotoSharingApplicationContext from the controller to the repository
9. Save the repository class file

### Task 3: Modify the PhotosController to use the repository

1. Open the PhotosController
2. Replace the private readonly field with a \_repository of type IPhotoRepository
3. Modify the constructor to accept a repository parameter of type IPhotoRepository
   * Let the constructor save the parameter in the private field created in step 2
4. Modify the Details action
   * Locate the code that makes use of the PhotoSharingApplicationContext class
   * Replace it with code that invokes the GetAsync method of the repository
5. Modify the Create action with the HttpPost constraint
   * Locate the code that uses the PhotoSharingApplicationContext
   * Replace it with code that invokes the CreateAsync method of the repository
6. Modify the Delete action
   * Locate the code that makes use of the PhotoSharingApplicationContext class
   * Replace it with code that invokes the GetAsync method of the repository
7. Modify the Delete action with the HttpPost constraint
   * Locate the code that makes use of the PhotoSharingApplicationContext class
   * Replace it with code that invokes the RemoveAsync method of the repository
8. Modify the GetImage action
   * Locate the code that makes use of the PhotoSharingApplicationContext class
   * Replace it with code that invokes the GetAsync method of the repository
9. Save the controller file

### Task 4: Modify the PhotoGallery View Component to use the repository

1. Open the PhotoGalleryViewComponent class
2. Replace the private readonly field with a \_repository of type IPhotoRepository
3. Modify the constructor to accept a repository parameter of type IPhotoRepository
   * Let the constructor save the parameter in the private field created in step 2
4. Modify the InvokeAsync method
   * Locate the code that makes use of the PhotoSharingApplicationContext class
   * Replace it with code that invokes the GetPhotosAsync method of the repository
5. Save the view controller file

### Task 5: Register the repository as a service

1. Open the Startup class
2. Locate the ConfigureServices method
3. Add a scoped service of type IPhotoRepository with an implementation of the IPhotoRepository interface to the service collection
4. Save the Startup file

# Exercise 2: Performing Unit Tests

## Scenario

In this exercise, you will:

* Create a test project and write the following tests.
  + IndexReturnsView: This test checks that the Index action returns a view named Index.
  + DetailsReturnsNotFound\_WhenIdIsNull: This test checks that the Details action returns a NotFoundResult if the id parameter is null
  + DetailsReturnsNotFound\_WhenPhotoNotFound: This test checks that the Details action returns a NotFoundResult when the Photo is not found
  + DetailsReturnsView\_WithAPhoto: This test checks that the Details action returns a View with a Photo in the model
  + GetImageReturnsFile: This test checks that the GetImage action returns a file and not a view.
  + CreateReturnsCreateView\_WhenModelStateIsInvalid: This test checks that the Create action returns the Create View when the model state is invalid
  + CreateReturnsRedirect\_WhenModelStateIsValid: This test checks that the Create action returns a Redirect to the Index action when the model state is valid

Note: The tests you add to the solution in this exercise will improve the quality of code and prevent bugs as development proceeds. However, this exercise does not conform to the principles of TDD because the PhotoController class already exists. In TDD, you would create these and other tests first, and then create a PhotoController class that passes the tests.

The main tasks for this exercise are as follows:

1. Create a test project.
2. Write the tests.
3. Add further tests.

### Task 1: Create a test project.

1. In Visual Studio, right click on the Solution Explorer and Add a New Project
   * Project name: PhotoSharingApplication.Tests
   * Project type: xUnit Test Project (.NET Core)
2. Add a reference to the PhotoSharingApplication project
3. Add a reference to the Microsoft.Aspnetcore.App NuGet Package
4. Add a reference to the Moq NuGet Package

### Task 2: Write the tests.

* Rename the Test class as PhotoControllerTests.
* Create a method using the following information:
  + Accessibility: private
  + Return type: List<Photo>
  + Name: GetTestPhotos
  + Input Parameters: none
* In the GetTestPhotos method:
  + Return a new List of Photo objects initialized with two Photo instances
  + Initialize the first Photo with an Id of 1, a Title of “Test1”, a ContentType of “img/jpg”
  + Initialize the second Photo with an Id of 2 and a Title of “Test2”, a ContentType of “img/gif”
* Create a Test method: IndexReturnsView
  + Create a Mock of type IPhotoRepository
  + Create an instance of a PhotosController and pass the Object property of the mock instance
  + Invoke the Index action and capture the result
  + Assert that the result is of type ViewResult
* Add a new DetailsReturnsNotFound\_WhenIdIsNull test method
  + Create a Mock of type IPhotoRepository
  + Setup the mock so that the FindAsync invokes the FindAsync of any int returns null
  + Create an instance of a PhotosController and pass the Object property of the mock instance
  + Invoke the Details action passing null and capture the result
  + Assert that the result is of type NotFoundResult
* Add a new DetailsReturnsNotFound\_WhenPhotoNotFound test method
  + Create a testPhotoId variable and set it to -1
  + Create a Mock of type IPhotoRepository
  + Setup the mock so that the FindAsync invokes the GetTestPhotos and returns the first item with the same id as the testPhotoId variable
  + Create an instance of a PhotosController and pass the Object property of the mock instance
  + Invoke the Details action passing testPhotoId and capture the result
  + Assert that the result is of type NotFoundResult
* Add a new DetailsReturnsView\_WithAPhoto test method
  + Create a testPhotoId variable and set it to 1
  + Create a Mock of type IPhotoRepository
  + Setup the mock so that the FindAsync invokes the GetTestPhotos and returns the first item with the same id as the testPhotoId variable
  + Create an instance of a PhotosController and pass the Object property of the mock instance
  + Invoke the Details action passing testPhotoId and capture the result
  + Assert that the result is of type ViewResult
  + Assert that the Model property of the ViewData is of type Photo
  + Assert that the Title property of the model is “Test1”
  + Assert that the Id property of the model is equal to the testPhotoId
* Add a new GetImageReturnsFile test method by using the following information:
* Annotation: [Theory], [InlineData(1)], [InlineData(2)]
* Scope: public
* Return type: async Task
* Name: GetImageReturnsFile
* Parameters: an integer named testPhotoId and a string named contentType
* In the GetImageReturnsFile test
  + Create a Mock of type IPhotoRepository
  + Setup the mock so that the FindAsync invokes the GetTestPhotos and returns the first item with the same id as the testPhotoId parameter
  + Create an instance of a PhotosController and pass the Object property of the mock instance
  + Invoke the GetImage action passing the testPhotoId and capture the result
  + Assert that the result is assignable from type IFileResult
  + Assert that the ContentTipe of the result is is equal to the contentType parameter
* Add a new CreateReturnsCreateView\_WhenModelStateIsInvalid test method
  + Create a Mock of type IPhotoRepository
  + Setup the mock so that the CreateAsync of any photo returns a completed task
  + Create an instance of a PhotosController and pass the Object property of the mock instance
  + Add a model error to the modelstate of the controller
  + Invoke the Create action passing a new Photo with a PhotoId of 1 and a Title of Test1 and capture the result
  + Assert that the result is of type ViewResult
  + Assert that the Model property of the ViewData is of type Photo
  + Assert that the Title property of the model is “Test1”
  + Assert that the Id property of the model is equal to the testPhotoId
* Add a new CreateReturnsRedirect\_WhenModelStateIsValid test method
  + Create a Mock of type IPhotoRepository
  + Setup the mock so that the CreateAsync of any photo returns a completed task
  + Create an instance of a PhotosController and pass the Object property of the mock instance
  + Invoke the Create action passing a new Photo with a PhotoId of 1 and a Title of Test1 and capture the result
  + Assert that the result is of type RedirectToAction
  + Assert that the name of the controller is null
  + Assert that the name of the action is “Index”
* Run all the tests in the PhotoSharingTests project and examine the results.

**Results**: After completing this exercise, you will be able to add a set of PhotoController tests defined in the PhotoSharingTests project of the Photo Sharing application.

# Exercise 2: Optional—Configuring Exception Handling

## Scenario

Now that you have developed unit tests for the Photo Sharing application, you need to configure an exception handling strategy for the MVC web application. This would ensure that when exceptions occur, the exception message is displayed in a custom MVC error view. You also want to display a custom message for when a page is not found. You also need to implement a placeholder action for the SlideShow action in the PhotoController view. This action will be completed during a later iteration of the project.

Complete this exercise if time permits.

The main tasks for this exercise are as follows:

1. Edit Startup.cs for exception handling.
2. Create a custom error view.
3. Raise errors.

### Task 1: Edit Startup.cs to redirect the status code pages.

1. Open the Startup.cs file in the root level folder of the PhotoSharingApplication project.
2. Add the UseStatusCodePagesWithRedirects by using the following path:

* "~/errors/{0}.html"

1. Add a new HTML page to the PhotoSharingApplication project by using the following information:

* Folder: ~/wwwroot/errors
* Template: HTML Page
* Name: 404.html

1. In the 404.html file, set the contents of the TITLE element to Error.
2. Add content to the 404.html file to explain to users that the entered address does not exists.

### Task 3: Configure errors in the PhotoController class.

1. Add a new action to the PhotoController class by using the following information:

* Scope: public
* Return type: ActionResult
* Name: SlideShow
* Parameters: None

1. In the new action, throw an exception by using the following information:

* Type: NotImplmentedException
* Message: The SlideShow action is not yet ready

### Task 4: Raise errors.

1. In Visual Studio, in the Solution Explorer, right click the PhotoSharingApplication project and select Properties
2. In the debug tab, select the Environment Variable ASPNETCORE\_ENVIRONMENT and change the value from Development to Production
3. Start debugging and display Sample Photo 5.
4. In the Internet Explorer window, request the relative URL and view the error details.

* URL: /Photo/Display/malformedID

1. In the Internet Explorer window, request the relative URL.

* URL: /Photo/SlideShow

1. Stop debugging and close Visual Studio.

**Results**: After completing this exercise, you will be able to: Configure a custom error handling strategy for an MVC application.