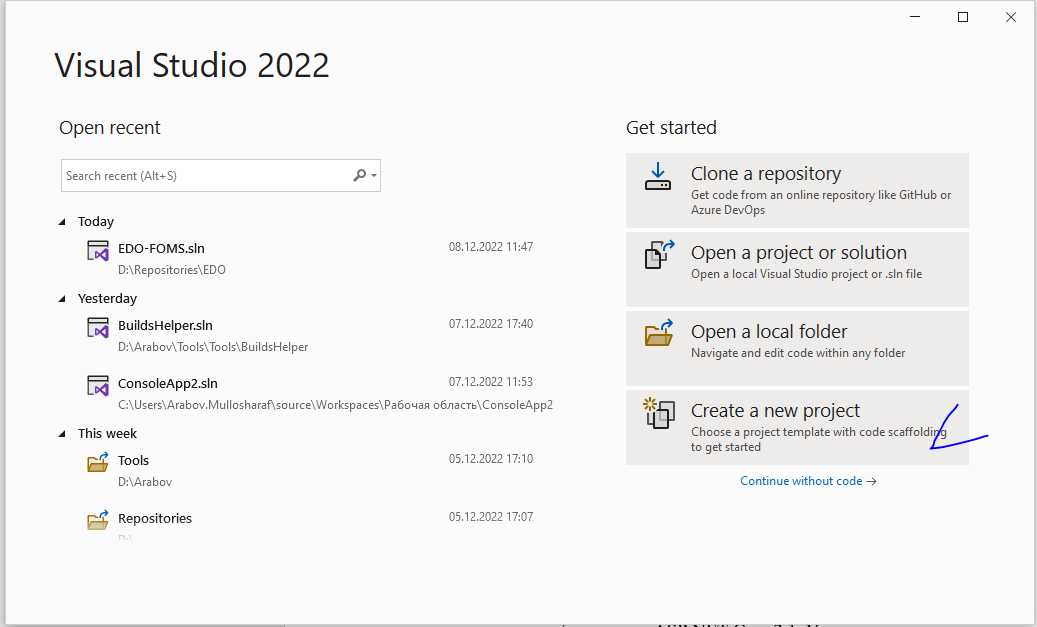
## 4. Настройка проекта

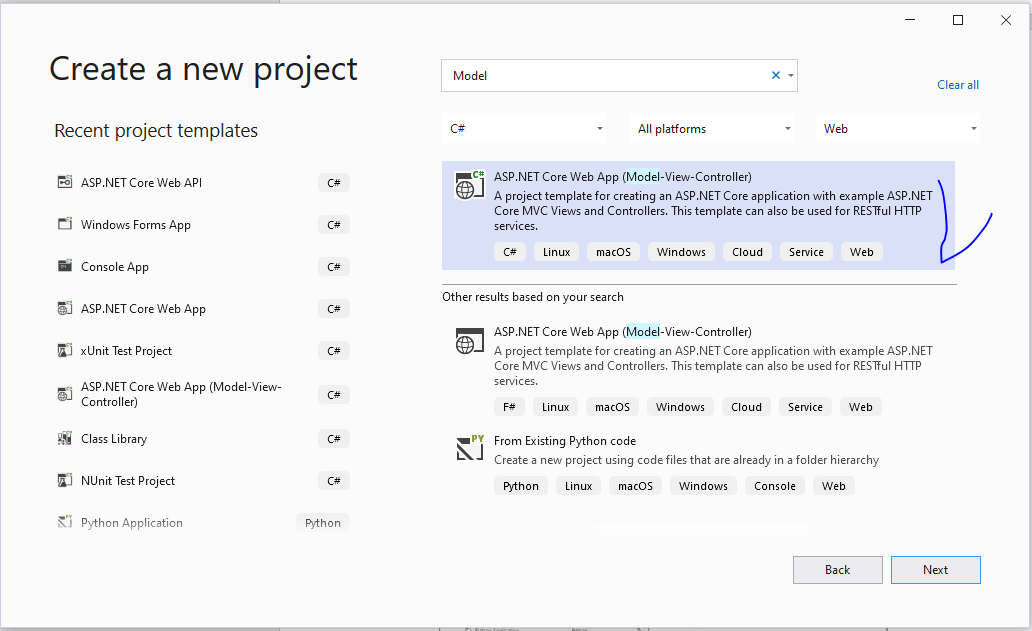
В этом сеансе мы создадим проект в Visual Studio 2022 или более поздней версии, который будет использоваться для этой демонстрации. Помимо основного проекта, он также будет включать проект тестирования, где будут написаны все необходимые Unit Test Cases. Для этой демонстрации мы используем Visual Studio 2022, но любой может использовать более позднюю версию Visual Studio для создания ядра Asp.NET. Итак, давайте начнем создавать проект Asp.Net Core вместе с проектом тестирования xUnit для DevOps Demo.

### **4.1 Создание основного проекта Asp.Net**

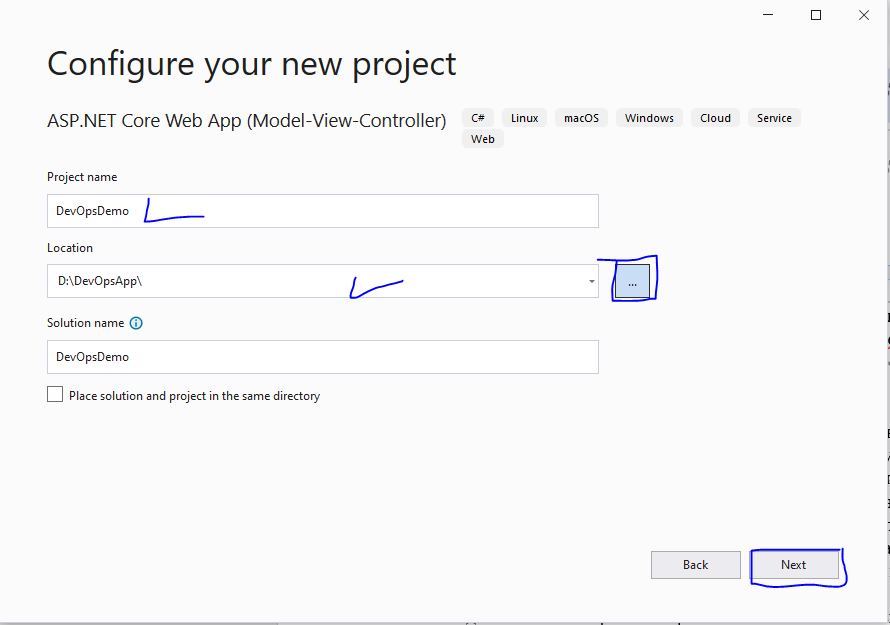
Давайте создадим основной проект Asp.Net в Visual Studio. Откройте Visual Studio 2022 или более позднюю версию, и создайте новый проекте выбирая пункта «Create a new project».



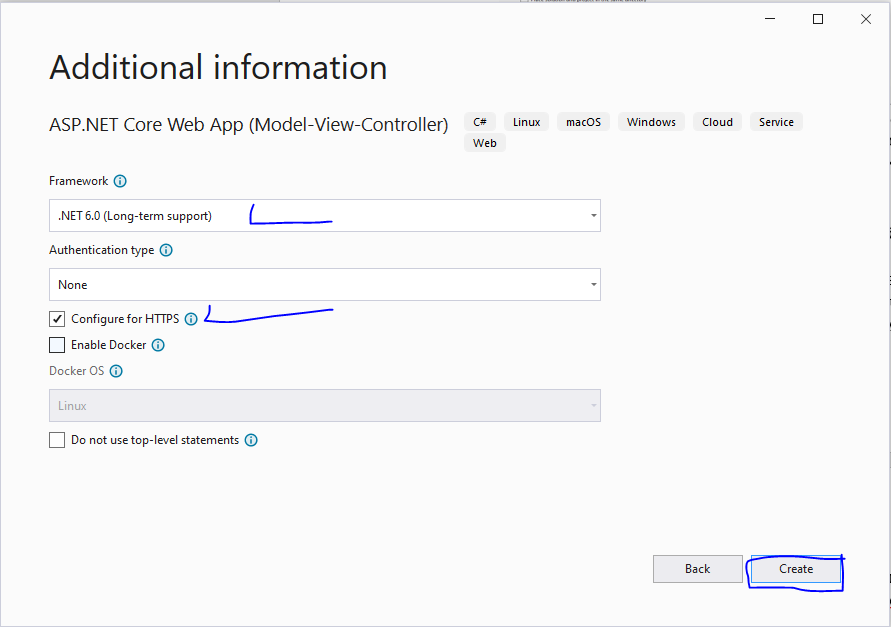
В следующее окно выберите «ASP.NET Core Web App (model-View-Controller)» и нажмите кнопку «Next».



После выбора шаблона приложения .Net Core укажите подходящее имя для проекта, а также имя решения, например DevOpsDemo. Этот проект создается для демонстрации DevOps, поэтому оставьте простое название DevOpsDemo. Не забудьте указать местоположение проекта и нажмите «ОК».

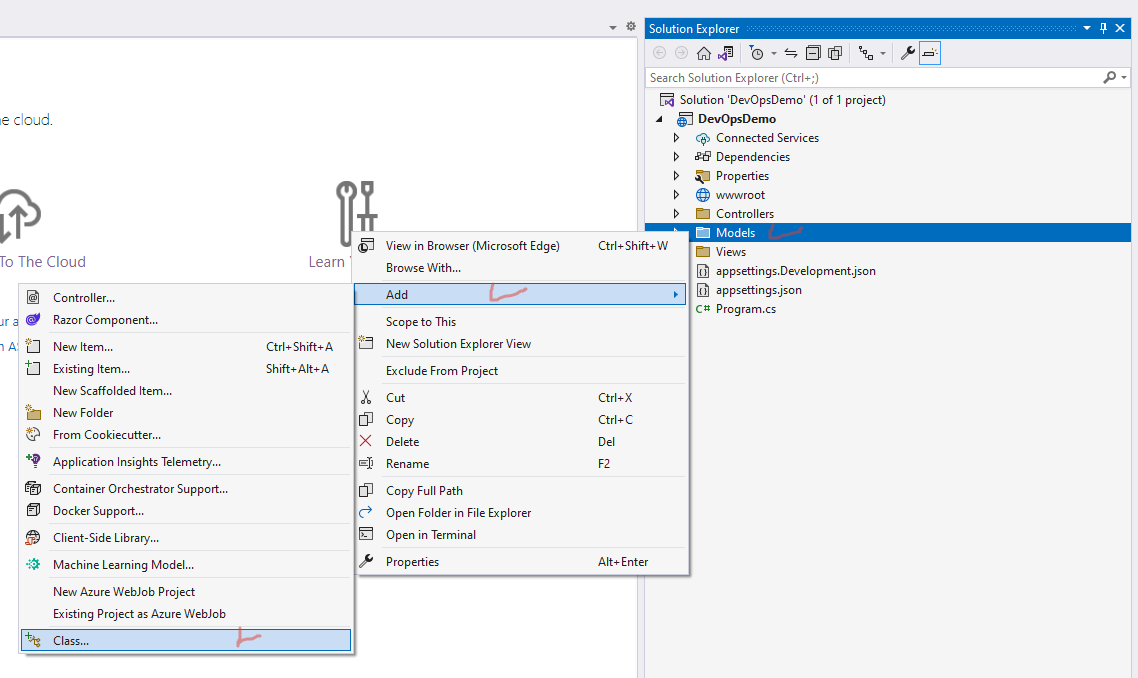


В следующем диалоговом окне будет предложено выбрать шаблон проекта, здесь мы работаем над ASP.NET Core 6.0. У нас доступны различные типы шаблонов проектов, такие как API, веб-приложение, контроллер представления модели и т. д. Здесь мы выберем веб-приложение (контроллер модели-представления), которое активирует функциональность MVC. Помимо этого, нам потребуются еще две вещи: сначала установите флажок для настройки HTTPS и изменения аутентификации и выберите «Без аутентификации». Теперь нажмите ОК.

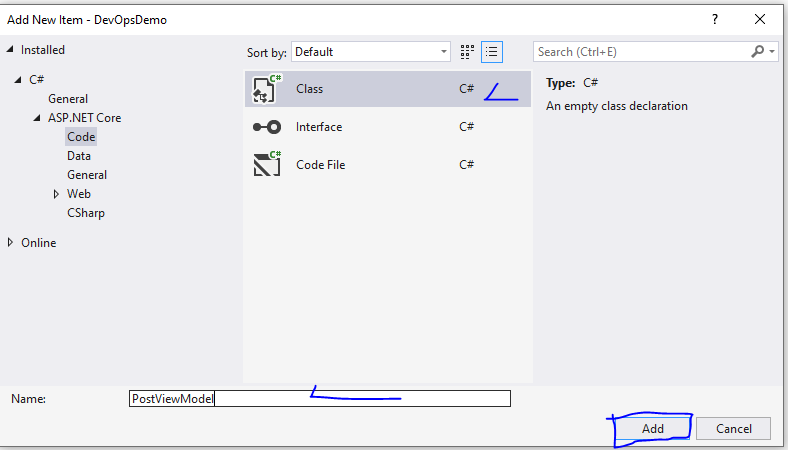


Создание и настройка проекта займет несколько минут, и окончательный проект будет готов. Как только проект будет готов, мы увидим базовое приложение структуры MVC в веб-приложении ASP.NET Core.

Давайте добавим некоторые функции Model-View-Controller. Итак, давайте создадим класс Model для «Post». Щелкните правой кнопкой мыши папку «Модель» в проекте и выберите «Добавить», а затем выберите «Класс».



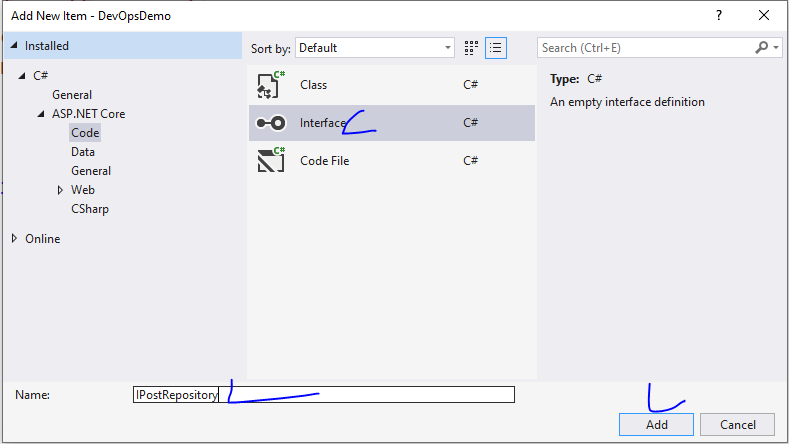
Откроется диалоговое окно «Добавить новый элемент», здесь мы можем выбрать различные типы файлов данных модели, такие как класс, интерфейс и т. д. Итак, просто выберите «Класс» и укажите имя для класса «PostViewModel.cs» и нажмите «Добавить».



Откройте PostViewModel и обновите код класса следующим образом. Здесь мы добавляем четыре свойства, такие как PostId, Title, Description и Author для класса PostViewModel.

|  |
| --- |
| namespace DevOpsDemo.Models  {  public partial class PostViewModel  {  public int PostId { get; set; }  public string Title { get; set; }  public string Description { get; set; }  public string Author { get; set; }  }  } |

Следующий шаг для создания информации репозитория. Итак, давайте создадим еще одну папку для классов и интерфейсов репозитория с названием Repository. После создания папки репозитория щелкните правой кнопкой мыши папку и выберите «Добавить»> «Новый элемент». Откроется диалоговое окно «Добавить новый элемент», в котором мы можем выбрать тип файла. Итак, сначала выберите интерфейс и укажите имя интерфейса как IPostRepository.cs и нажмите «Добавить».



Обновите код интерфейса IPostRepository следующим образом. Здесь мы добавляем один метод, который будет возвращать список сообщений.

Тот же процесс необходимо выполнить, чтобы добавить один новый класс в качестве PostRepository.cs в папку репозитория следующим образом.

|  |
| --- |
| using DevOpsDemo.Models;  namespace DevOpsDemo.Repository  {  public interface IPostRepository  {  List<PostViewModel> GetPosts();  }  } |

Вот класс PostRepository, который реализует IPostRepository. Мы создаем некоторые фиктивные данные для сообщений. Мы сохраняем простоту и не реализуем управляемую базой данных функциональность для получения данных в реальном времени. Это потому, что эта демонстрация предназначена только для понимания того, как работает DevOps и как мы можем реализовать непрерывную интеграцию и непрерывную доставку.

|  |
| --- |
| using DevOpsDemo.Models;  namespace DevOpsDemo.Repository  {  public class PostRepository : IPostRepository  {  public List<PostViewModel> GetPosts()  {  var posts  = new List<PostViewModel> {  new PostViewModel(){ PostId =101, Title = "DevOps Demo Title1",  Description ="DevOps Demo Description 1", Author="Adam Samad"},  new PostViewModel(){ PostId =102, Title = "DevOps Demo Title2",  Description ="DevOps Demo Description 2", Author="Ozar Fariz"},  new PostViewModel(){ PostId =103, Title = "DevOps Demo Title3",  Description ="DevOps Demo Description 3", Author="Rahul Omar"},  };  return posts;  }  }  } |

Теперь пришло время показать данные в представлении, которые возвращаются из репозитория. Итак, давайте откроем HomeController и реализуем внедрение зависимостей конструктора для получения экземпляра класса PostRepository и создадим ActionResult as Index. Здесь, в методе Index, мы будем извлекать данные, используя экземпляр PostRepository, и возвращать данные в представление.

|  |
| --- |
| using DevOpsDemo.Models;  using DevOpsDemo.Repository;  using Microsoft.AspNetCore.Mvc;  using System.Diagnostics;  namespace DevOpsDemo.Controllers  {  public class HomeController : Controller  {  private readonly ILogger<HomeController> \_logger;  IPostRepository postRepository;  public HomeController(IPostRepository \_postRepository, ILogger<HomeController> logger)  {  postRepository = \_postRepository;  \_logger = logger;  }  public IActionResult Index()  {  var model = postRepository.GetPosts();  return View(model);  }  public IActionResult About()  {  ViewData["Message"] = "Your application description page.";  return View();  }  public IActionResult Contact()  {  ViewData["Message"] = "Your contact page.";  return View();  }  public IActionResult Privacy()  {  return View();  }  [ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = true)]  public IActionResult Error()  {  return View(new ErrorViewModel { RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier });  }  }  } |

Мы будем заполнять данные в представлении в табличном формате. Итак, как только данные будут там, мы будем повторять их и отображать данные следующим образом.

|  |
| --- |
| @model IList<DevOpsDemo.Models.PostViewModel>  @{  ViewData["Title"] = "Home Page";  }  <div class="row">  <h2>Post List</h2>  <table class="table">  <thead>  <tr>  <th>Post Id</th>  <th>Title</th>  <th>Description</th>  <th>Author</th>  </tr>  </thead>  <tbody>  @foreach (var item in Model)  {  <tr>  <td>@Html.DisplayFor(modelItem => item.PostId)</td>  <td>@Html.DisplayFor(modelItem => item.Title)</td>  <td>@Html.DisplayFor(modelItem => item.Description)</td>  <td>@Html.DisplayFor(modelItem => item.Author)</td>  </tr>  }  </tbody>  </table>  </div> |

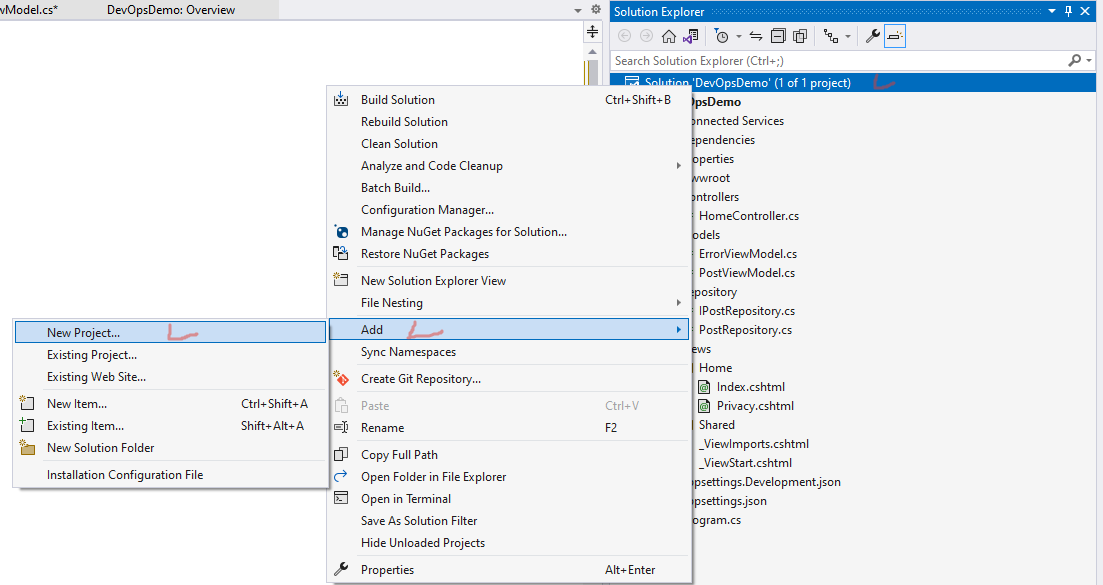
### **4.2 Тестовый проект xUnit**

Тестирование является важным аспектом любого продукта. Без тестирования мы не можем думать, что продукт готов к поставке.

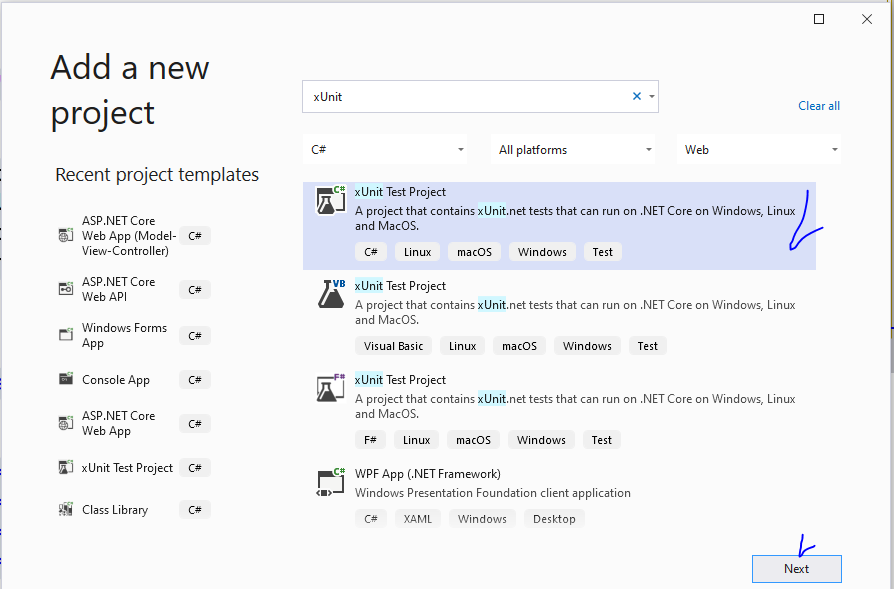
Мы проводим тестирование разными способами, но на этапе разработки, когда мы пишем код для конкретной функциональности, мы пишем тестовые примеры для этой функциональности и проверяем, все ли работает нормально, как ожидалось, или нет.

Модульные тестовые случаи помогают нам находить ошибки или проблемы в коде на более ранней стадии при создании кода. Во время сборки кода и создания артефакта он также выполняет тестовые случаи, и если тестовые случаи не выполняются, это означает, что что-то не так, и функциональность не соответствует ожидаемой.

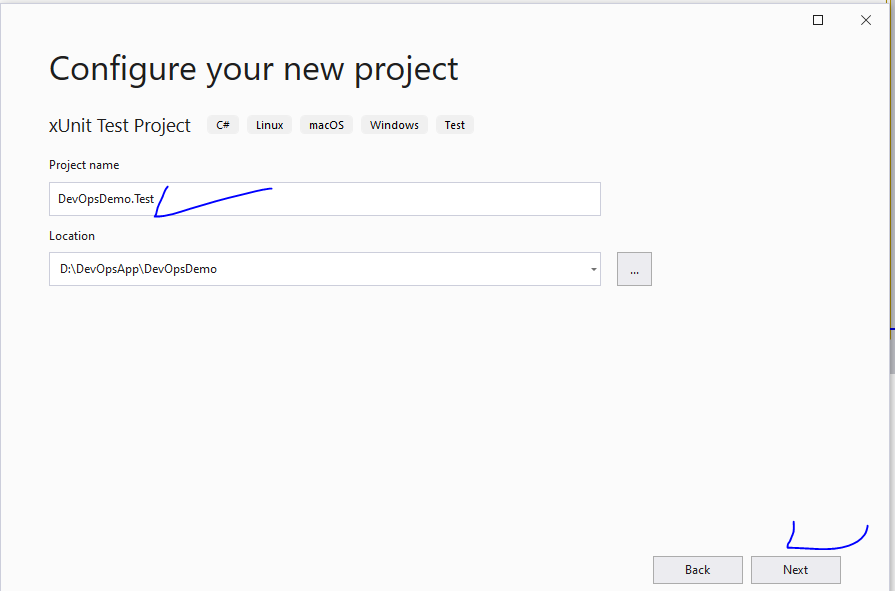
Здесь мы создадим отдельный проект тестирования для основного веб-приложения ASP.NET. Итак, щелкните правой кнопкой мыши решение DevOpsDemo и выберите «Добавить» > «Новый проект», как показано на следующем рисунке.



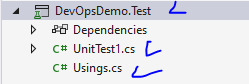
Откроется диалоговое окно «Новый проект», здесь мы должны выполнить тот же процесс, что и выше, для создания нового проекта веб-приложения ASP.NET Core. Только нам нужно изменить шаблон приложения, для проекта тестирования мы выберем xUnit Test Project (.Net Core).



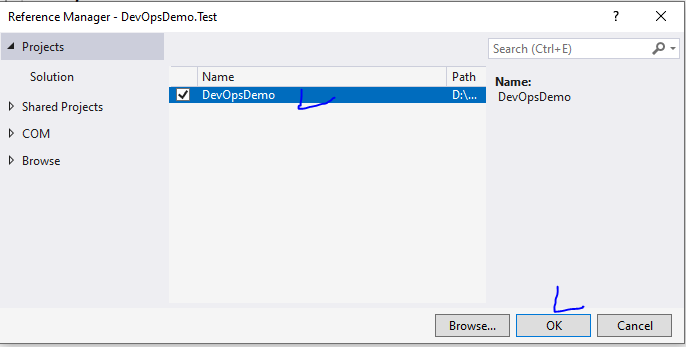
После выбора шаблона проекта укажите имя проекта тестирования как DevOpsDemo.Test и нажмите Next.



Через несколько секунд проект тестирования xUnit будет готов. Он будет содержать один класс модульного теста как UnitTest1.cs. Прежде чем двигаться дальше, просто переименуйте UnitTest1 в PostTestController.



Так же будем тестировать основной функционал проекта. Итак, пришло время добавить ссылку на основной проект в тестовый проект, чтобы мы могли получить доступ к репозиторию и контроллеру для написания тестовых случаев. Итак, щелкните правой кнопкой мыши проект «Зависимости в тестировании» и выберите «Добавить ссылку». Откроется диспетчер ссылок для проекта DevOpsDemo.Test. В разделе «Проекты» на левой панели выберите DevOpsDemo (отметьте флажком) и нажмите «ОК».



Откройте контроллер PostTestController и напишите несколько юнит-тестов для HomeController следующим образом.

|  |
| --- |
| using DevOpsDemo.Controllers;  using DevOpsDemo.Models;  using DevOpsDemo.Repository;  using Microsoft.AspNetCore.Mvc;  namespace DevOpsDemo.Test  {  // Определение класса PostTestController для тестирования HomeController  public class PostTestController  {  private PostRepository repository;  // Конструктор класса, создающий экземпляр репозитория PostRepository  public PostTestController()  {  repository = new PostRepository();  }  // Юнит-тест для проверки, что результат выполнения метода Index является объектом типа ViewResult  [Fact]  public void Test\_Index\_View\_Result()  {  // Arrange  // Создаем экземпляр контроллера HomeController и передаем ему экземпляр репозитория PostRepository  var controller = new HomeController(this.repository);  // Act  // Вызываем метод Index контроллера HomeController  var result = controller.Index();  // Assert  // Проверяем, что результат выполнения метода Index является объектом типа ViewResult  Assert.IsType<ViewResult>(result);  }  // Юнит-тест для проверки, что результат выполнения метода Index не равен null  [Fact]  public void Test\_Index\_Return\_Result()  {  // Arrange  // Создаем экземпляр контроллера HomeController и передаем ему экземпляр репозитория PostRepository  var controller = new HomeController(this.repository);  // Act  // Вызываем метод Index контроллера HomeController  var result = controller.Index();  // Assert  // Проверяем, что результат выполнения метода Index не равен null  Assert.NotNull(result);  }  // Юнит-тест для проверки, что результат выполнения метода Index содержит коллекцию объектов типа PostViewModel  // и что каждый элемент коллекции имеет определенные значения свойств, в том числе PostId и Title.  [Fact]  public void Test\_Index\_GetPosts\_MatchData()  {  // Arrange  // Создаем экземпляр контроллера HomeController и передаем ему экземпляр репозитория PostRepository  var controller = new HomeController(this.repository);  // Act  // Вызываем метод Index контроллера HomeController  var result = controller.Index();  // Assert  // Проверяем, что результат выполнения метода Index является объектом типа ViewResult  var viewResult = Assert.IsType<ViewResult>(result);  // Получаем модель представления из ViewData объекта  var model = Assert.IsAssignableFrom<IEnumerable<PostViewModel>>(viewResult.ViewData.Model);  // Проверяем, что модель представления содержит три элемента  Assert.Equal(3, model.Count());  // Проверяем, что первый элемент коллекции имеет значение свойства PostId равное 101  Assert.Equal(101, model.ElementAt(0).PostId);  // Проверяем, что первый элемент коллекции имеет значение свойства Title равное "DevOps Demo Title1"  Assert.Equal("DevOps Demo Title1", model.ElementAt(0).Title);  }  }  } |

Здесь:

* Arrange - этап подготовки к тестированию, на котором создаются необходимые объекты и устанавливаются начальные значения. В данном случае на этом этапе создается экземпляр контроллера HomeController и передается ему экземпляр репозитория PostRepository.
* Act - этап выполнения действия, которое необходимо протестировать. В данном случае на этом этапе вызывается метод Index контроллера HomeController.
* Assert - этап проверки результата выполнения действия. В данном случае на этом этапе проверяется, что результат выполнения метода Index является объектом типа ViewResult, не равен null, а также содержит коллекцию объектов типа PostViewModel, в которой каждый элемент имеет определенные значения свойств.

Откройте Test Explorer и нажмите на кнопку "Run All", как показано на следующем изображении, чтобы начать выполнение модульных тестов, которые начнут сборку решения и выполнение модульных тестов. Как только все модульные тесты будут выполнены, вы увидите статус зеленого цвета. Как показано на следующем изображении, мы можем видеть, что все тесты пройдены.

