Issue 34 | Jan-Feb 2018

DNCMagazine

www.dotnetcurry.com

Angular Machine Learning DevOps Cosmos DB C# ASP.NET Core Docker Azure Unit Testing Integration Testing Cognitive Services Dataflow Pattern Xamarin

EDITORIAL

Suprotim Agarwal

Editor in Chief

suprotimagarwal@a2zknowledgevisuals.com

Whether we're ready for it or not, Artificial Intelligence (AI) is infiltrating our daily lives. There's a lot to be guardedly excited about in the AI and Machine Learning (ML) space. In our 34th Edition, Rahul Sahasrabuddhe gives a good primer on ML and gets us started with Azure Machine Learning Studio - a drag and drop machine learning platform, with a visual interface! Microsoft provides you with massive AI and ML power that can be accessed through simple REST APIs (Cognitive Services). Gerald Versluis, shows us how to use these APIs to make our Xamarin applications more intelligent by adding features such as facial recognition to it..

In our coverage of C#, David Pine walks us through the various versions of C# and shares his favorite features from each release. Damir Arh emphasizes on why it is important to understand how a C# feature is implemented, and highlights some quirks, gotchas and pitfalls.

Subodh, in his feature, unravels the mystery of DevOps, and draws our attention to its core tenets - increasing collaboration, sharing responsibility, and automating process flows, to continuously deliver value. Yacoub explores the potential of the DataFlow pattern that he had briefly touched upon in the 33rd Edition, and introduces you to ProceduralDataflow, a new library he has created to help write clean dataflows. For our Azure fans, Francesco showcases how to do fast Azure CosmosDB development using a package he has created, while Ravi demonstrates how to create a Docker image for an Angular application and deploy it on Azure.

Last but not the least, Daniel urges developers to understand the importance of a good testing strategy and demonstrates how to perform Unit and Integration testing for ASP.NET Core applications.

As we start this exciting New Year, I want to thank you from the bottom of our hearts for making us the great magazine we've become. Happy New Year, readers!

Editor In Chief : Suprotim Agarwal

Art Director : Minal Agarwal

Contributing Authors : Yacoub Massad, Subodh Sohoni, Ravi Kiran, Rahul Sahasrabuddhe, Gerald Versluis, Francesco Abbruzzese, David Pine, Daniel Jimenez Garcia, Damir Arh.

Technical Reviewers : Yacoub Massad, Suprotim Agarwal, Subodh Sohoni, Mahesh Sabnis, Daniel Jimenez Garcia, Damir Arh.

Next Edition : March 2018

THANK YOU

FOR THE 34th EDITION

@subodhsohoni

@yacoubmassad

@sravi_kiran

@dani_djg

@damirarh

@rahul1000buddhe

@davidpine

@jfversluis

@F_Abbruzzese

@suprotimagarwal

@maheshdotnet

@saffronstroke

WRITE FOR US

mailto: suprotimagarwal@dotnetcurry.com

RAVENDB 4.0

Your Fully Transactional NoSQL Database

LEARN IT LIVE

Workshops '18

Join RavenDB CEO Oren Eini for a first look at the exciting features of newly released RavenDB 4.0. He will take you from the starting point, assuming this is your first experience with a NoSQL database, and rapidly ramp up your skills to use advanced functionality new to the world of NoSQL. This workshop is for developers and their operations teams who want to advance their expertise in RavenDB. You will discover how to:

SIGN UP AT

workshops.ravendb.net

- Set up and secure your database in minutes
- Establish constant availability by deploying nodes to a distributed cluster
- Achieve super-fast aggregation with Map-Reduce Indexes
- Code with RavenDB 4.0 and our SQL-like query language
- Query efficiently by taking advantage of intelligent Auto Indexes
- Model Data in a NoSQL Document Database

Hibernating Rhinos

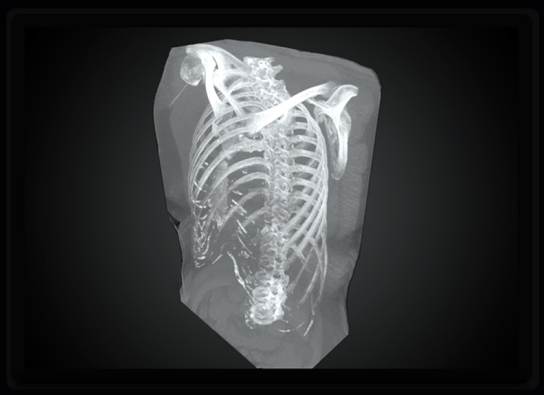
www.ravendb.net

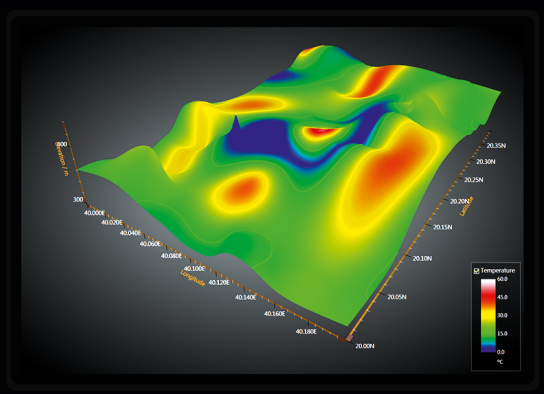
Arction

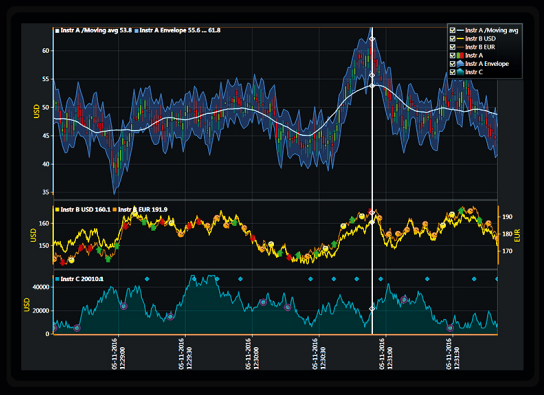
[WPF]
[Windows Forms]
[Free Gauges]
[Data Visualization]
[Volume Rendering]
[3D / 2D Charts] [Maps]

LightningChart®

The fastest and most advanced charting components







Create eye-catching and powerful charting applications for engineering, science and trading

- DirectX GPU-accelerated
- Optimized for real-time monitoring
- Supports gigantic datasets
- Full mouse-interaction
- Outstanding technical support
- Hundreds of code examples

NEW
- Now with Volume Rendering extension
- Flexible licensing options

Get free trial at
LightningChart.com/dnc

CONTENTS

Daniel Jimenez Garcia

If you want to make sure your application behaves as expected, then you need to test it. There is no other way around it.

However, this is easier said than done!

There are quite a few different testing methods, each with its own unique strengths. Each method concentrates on testing different aspects of your application.

In order to truly test your application, you will need to devise a strategy that combines these testing techniques. Getting to know these methods should also make it easier to adopt development processes like Test Driven Development (TDD) or Behavior Driven Development (BDD), should you decide so.

UNIT TESTING

ASP.NET Core Applications

In this series of articles, we will revisit an earlier project about writing clean frontend code in ASP.NET Core and use it as our test subject while we go through an automated testing strategy involving unit tests, integration tests, end–to-end tests and load tests.

Before diving into these automated testing methods, I cannot finish this introduction without stressing the importance of complementing any automated testing strategy with some manual exploratory testing, preferably by QA specialists!

The code discussed through the article is available on its GitHub repo.

The Automated Testing Strategy

As you consider automating the testing of your application, you need to carefully think how you are going to combine the different testing methods in order to cover as many different aspects of your application as possible.

Let's start with a quick recap on these testing methods and their strengths and weaknesses.

This might be old news for some of you, so feel free to skip into the next section titled "The Test Subject" which introduces the test subject project.

Unit Testing

Unit Testing concentrates on exercising an individual *unit* isolated from the rest of the system. It *mocks* or *stubs* its dependencies to make sure it produces the desired outcomes, given a well-known set of inputs or system state.

We typically consider our classes as *units*, testing their public methods while mocking their dependencies. This has the additional benefit of leading your code towards a loosely coupled design, as otherwise you won't be able to isolate your classes from their dependencies.

We also can't ignore the importance of being able to write and run code without having to start the entire application

- You will get the most benefits from them on methods rich in business logic, where you want to exercise different inputs, corner cases or error conditions.

- However, in areas of your code that concentrate on dealing with external dependencies and infrastructure (like databases, files, services, 3rd party libraries/frameworks), these tests add considerably lesser value. Most of what this code does is dealing with external dependencies that you will be mocking in your tests!

- Hence the importance of a testing strategy that combines different methods!

When it comes to running your unit tests, they are fast and require no special configuration on your Continuous Integration (CI) servers.

Integration Testing

These tests will exercise several *units* together, treating them as a black box which is provided with some inputs, and should produce the expected outcomes.

In the context of web applications, it is common to consider your web server as the black box, writing integration tests that exercise its public endpoints without the need for a browser or client-side code.

This means your integration tests will exercise your real web server code with real HTTP requests, while it will be up to you to decide whether to include external dependencies like databases within your test or not.

When providing REST APIs, these tests are extremely valuable.

Since web servers need to be started, databases need to be isolated or reset and seed, and each test involves real HTTP requests; we can safely consider integration tests as more expensive to write and maintain, than unit tests. They are also slower to run and are harder to debug when they fail.

That's why they are good candidates to ensure that the most important and common use cases for each endpoint works with your real server.

At the same time, Integration tests are not so apt for trying all the different scenario variations, corner cases, error and boundary conditions; which is something you want to leave for your unit tests.

End–to-End Testing

End-to-End tests will exercise your system using the same interface than any regular user, through the UI. These makes them the most expensive tests to write and maintain, as they are tightly coupled to your UI, breaking often, as a result of changes to your interface.

Again, in the context of web applications, these tests will interact with a browser the same way that a user would do, using frameworks like Selenium to drive the browser. This makes these tests the slowest to run and more expensive to write and maintain! As you can imagine, investigating a failure in one of these tests is pretty close (if not the same) to debugging the application.

However, they are great to run smoke tests or golden thread tests on real browsers and devices. You want these tests to verify that your most important scenarios are not broken from the user perspective.

Other ways of testing your code

The testing methods we just discussed concentrate on ensuring that the application behaves as expected. But that's not the end of it, since your application might produce the desired output but may still have flaws like high response times or security issues.

That's why you can complement your testing strategy with additional methods like Load Testing, Penetration Testing or Static Code Analysis.

Automated testing is a huge subject that goes beyond unit, integration and end–to-end tests. Stay tuned to read about Load Testing in the last article of the series (coming soon).

Finally, don't ignore the power of manual exploratory testing, especially when done by a seasoned QA engineer. They will not only catch bugs on scenarios you didn't anticipate, but will also ensure your application behaves closer to the way a real user would expect or want to.

The Testing Pyramid

One of the main points to take from this overview is that different types of tests have different implementation/maintenance costs and have different purposes. That's why people like Martin Fowler have been talking for quite some time about the concept of the *testing pyramid*.

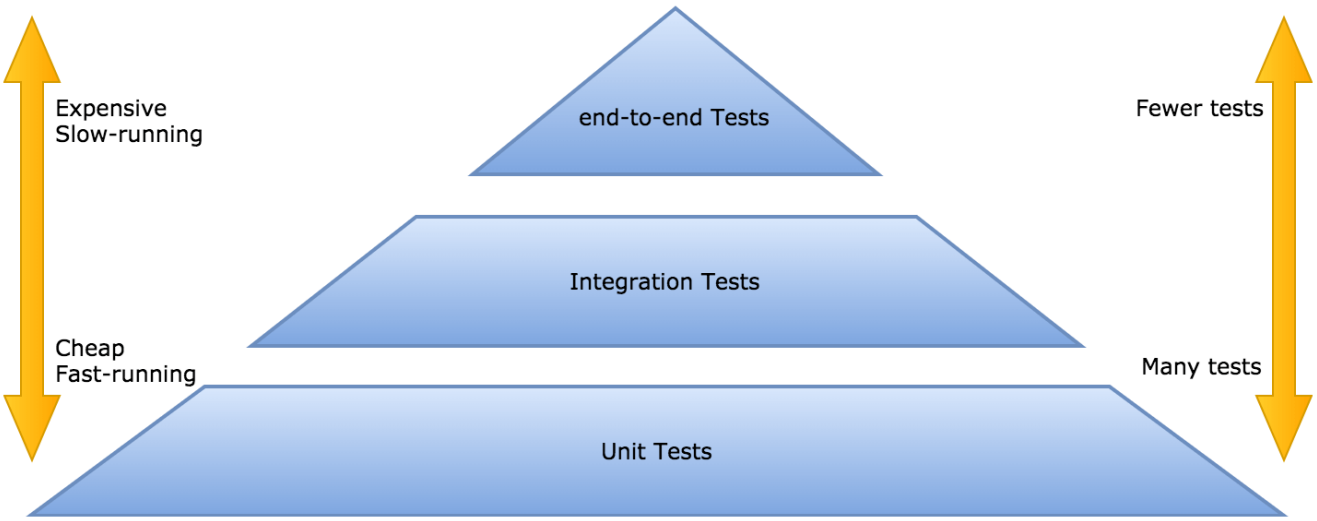


Figure 1, the Testing Pyramid

As you can see, it is a graphical depiction of a testing strategy that plays the strengths and weaknesses of the different testing methods.

- At the bottom you have a large suite of unit tests that exhaustively tests every piece of code with meaningful business logic. These provide constant feedback to developers about their code changes and are constantly run by CI servers.

- On the next level, you have a reduced number of integration tests, making sure several of your units working together still produce the expected results (In web applications, this would be your web server without client-side code). Developers typically run these at specific moments like adding a new feature to the system or before committing a change. CI servers will either run these on every change or after deploying a staging/integration environment, depending on the exact nature and implementation of the tests and how external dependencies like databases were isolated.

- Finally, at the top, you have a very exclusive number of End-to-End tests that make sure your application isn't fundamentally broken from a real user perspective. These will mostly be run by CI servers after environments are deployed, while developers will run them even more infrequently than integration tests.

In the remaining article, we will see how to write unit tests for our test subject.

Note: *The next articles in the series will cover the upper levels of the testing strategy. This magazine edition also covers Integration Tests, so make sure to check it out.*

The Test Subject

As mentioned previously, we will use an earlier project that talks about writing clean frontend code in ASP.NET Core and use it as our test subject. The application is a blogging site where authenticated users can write new blog posts and everyone else can read them.

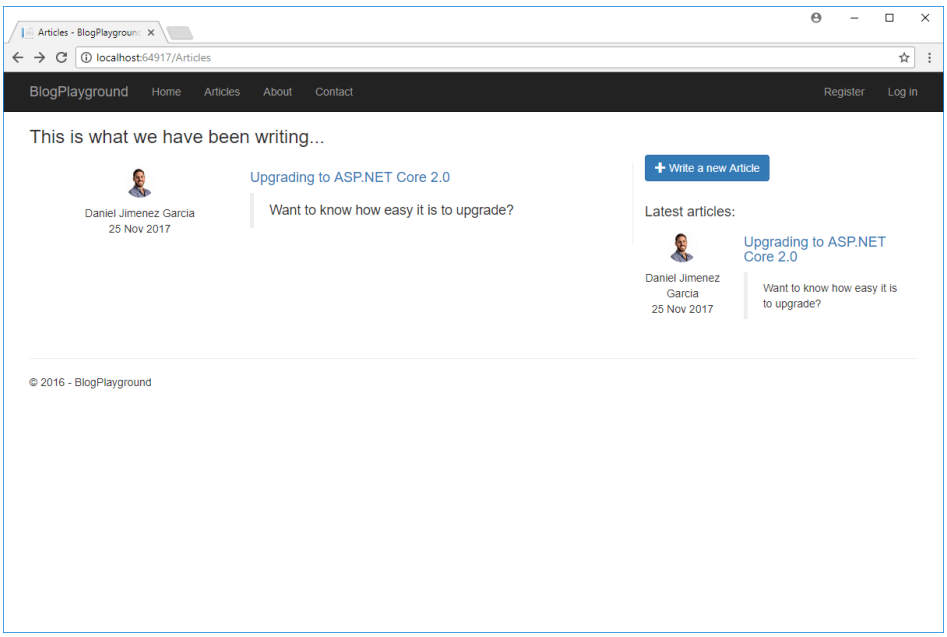


Figure 2, The example blogging application used as Test Subject

Technically, it is a standard ASP.NET Core web application which has been upgraded to ASP.NET Core 2.0 and has no tests at all!

This is a simple application logic-wise, but it will serve as a good example for writing the different types of tests without getting lost in the subtleties of the domain, business rules and use cases. Fear not, it will be enough to highlight many of the common challenges you will face when writing these tests.

Writing Unit Tests for ASP.NET Core applications

We will start from the bottom of the pyramid, adding unit tests to our sample blogging application. In order to write and run the unit tests, we will be using:

- xUnit as the test framework
- Moq as the mocking framework
- A new project that will contain all of our unit tests

The first thing we need is a new unit test project where we can write the unit tests!

Note: *If this was a real application with no tests, it would be more interesting to first add integration and end-to-end tests! As unit tests would most likely require code changes in order to make your code testable, you would have the other tests as a safety net, while you refactor the application.*

The reason why I am not following that approach is because I believe it is easier to introduce the concepts by first looking at unit tests, understand its strengths and shortcomings, highlighting the need for higher level tests, before moving into integration and end-to-end tests.

In Visual Studio, right click your solution and select *Add* > *New Project*, then select *xUnit Test Project*.

Since the project we want to test is named **BlogPlayground**, name the new test project **BlogPlayground.Test**. Alternatively, you can run from the command line…

```
mkdir BlogPlayground.Test && cd BlogPlayground.Test && dotnet new xunit
```

…to achieve the same results.

Note: If you are wondering what's the difference between the *xUnit Test Project* and *Unit Test Project* templates, it all boils down to the test framework used. The first one uses xUnit while the latter uses MSTest. If you prefer MSTest to xUnit, you can still follow along, as the principles are the same. You will just need to adjust your test code for the differences between the frameworks like attributes and setup/teardown code.

Let's finish setting up our test project by adding a reference to BlogPlayground (since our test code will *use* the real code) and installing the NuGet package Moq (which will be needed pretty soon to mock dependencies).

We will focus on adding unit tests for the `ArticlesController` class, so rename the generated test class as `ArticlesControllerTest` and move it inside a new folder *Controller* inside the test project. Your solution explorer should look like Figure 3 at this stage (notice how the structure of the Unit Test project matches that of the real project):

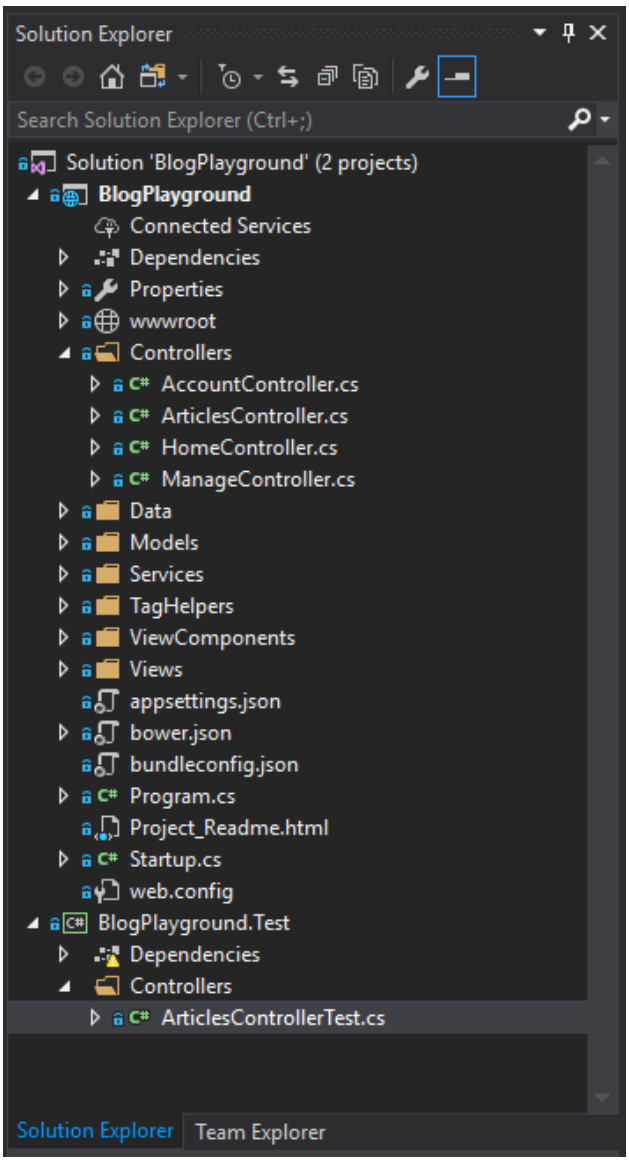


Figure 3, Solution after adding the new Unit Testing project

We are ready to write our first test for the `ArticlesController`! Let's start by adding a test that ensures the Index method renders the expected view with the articles list as its model.

However, just by inspecting its code it is obvious we have hit our first blocker:

How can we create an instance of `ArticlesController` without a db context and an UserManager? And if we were to test the Index method, how can we mock the context so we can test the Index method?

```
public ArticlesController(ApplicationDbContext context,
UserManager<ApplicationUser> userManager)
{
  _context = context;
  _userManager = userManager;
}

public async Task<IActionResult> Index()
{
  var applicationDbContext = _context.
      Article.Include(a => a.Author);

  return View(await applicationDbContext.
      ToListAsync());
}

// Other simpler methods omitted here!

public async Task<IActionResult> Create([Bind("Title, Abstract,Contents")] Article
article)
{
  if (ModelState.IsValid)
  {
    article.AuthorId = _userManager.GetUserId(this.User);
    article.CreatedDate = DateTime.Now;
    _context.Add(article);

    await _context.SaveChanges();
    return RedirectToAction("Index");
  }
  return View(article);
}

public async Task<IActionResult> DeleteConfirmed(int id)
{
  var article = await _context.Article.SingleOrDefaultAsync(m => m.ArticleId ==
  id);
    _context.Article.Remove(article);
    await _context.SaveChanges();
    return RedirectToAction("Index");
}
```

Our class right now is tightly coupled to the `ApplicationDbContext` and `UserManager` classes, and we will need to refactor our code to ensure it is loosely coupled and testable. Otherwise we won't be able to test it unless we provide real instances of those dependencies, in which case we would end up running the tests against a real db context connected to a database.

Refactoring towards a loosely coupled design

Let's start by refactoring our database access code so that our controller doesn't depend directly on the `ApplicationDbContext`. While this was fine within the context of the original application, typically most applications would have some layer(s) between the controller and the data access code, like a service layer, business layer or repository layer.

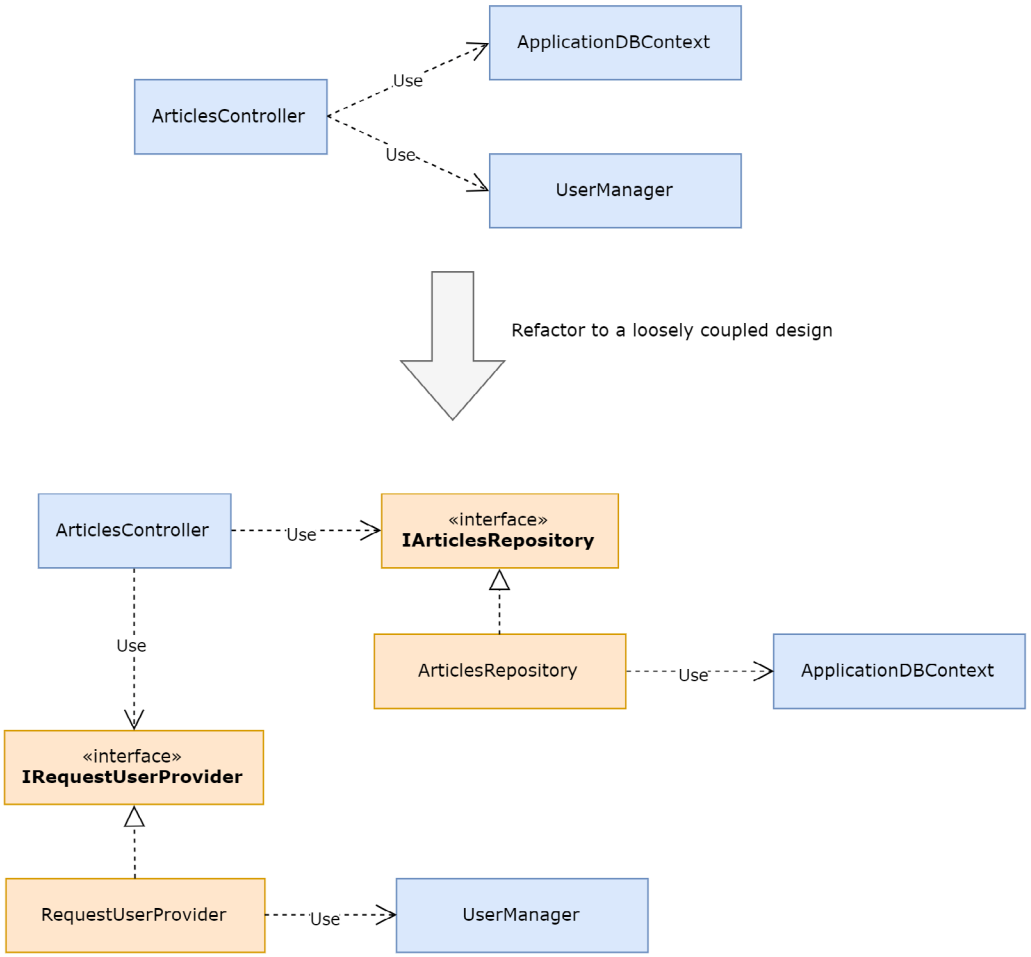


Figure 4, refactoring towards a loosely coupled design

There are many ways of breaking the dependency between the controller and the context classes, but the important message here is that you need to introduce an abstraction between these two classes.

Note: *As you might have noticed, this is pushing our tight coupling down to a new class! However, this is fine as long as you keep the code that really matters to your users - the one that implements your business logic, separated from infrastructure and plumbing code. Remember we will still have other tests like integration and end–to-end that verify everything works when put together!*

In this article, we will create a simple `IArticlesRepository` interface that our controller will depend upon.

You can find many great discussions about the different implementations of the generic repository pattern and unit of work. While there are better implementations, I have decided to keep things simple and to-the-point - the point breaking the dependencies in tightly coupled code!

```
public interface IArticlesRepository
{
  Task<List<Article>> GetAll();
  Task<Article> GetOne(int id);
  void Add(Article article);
  void Remove(Article article);
  Task SaveChanges();
}
```

The implementation is pretty straightforward. We are adding a new `ArticlesRepository` class that depends on the `ApplicationDbContext`:

```
public class ArticlesRepository : IArticlesRepository
{
  private readonly ApplicationDbContext _context;

  public ArticlesRepository(ApplicationDbContext context)
  {
    _context = context;
  }

  public Task<List<Article>> GetAll() =>
    _context.Article.Include(a => a.Author).ToListAsync();

  public Task<Article> GetOne(int id) =>
    _context.Article.Include(a => a.Author)
      .SingleOrDefaultAsync(m => m.ArticleId == id);

  public void Add(Article article) =>
    _context.Article.Add(article);

  public void Remove(Article article) =>
    _context.Article.Remove(article);

  public Task SaveChanges() =>
    _context.SaveChangesAsync();
}
```

Now we can modify the controller so it depends on our new abstraction, the `IArticlesRepository` interface. If you have any trouble modifying the methods not shown here, please check the source code on GitHub:

```
private readonly IArticlesRepository _articlesRepository;
private readonly UserManager<ApplicationUser> _userManager;

public ArticlesController(IArticlesRepository articlesRepository,
UserManager<ApplicationUser> userManager)
{
  _articlesRepository = articlesRepository;
  _userManager = userManager;
}

public async Task<IActionResult> Index()
{
  return View(await _articlesRepository.GetAll());
}
```

```
// Other simpler methods omitted here!

public async Task<IActionResult> Create([Bind("Title, Abstract,Contents")] Article
article)
{
  if (ModelState.IsValid)
  {
    article.AuthorId = _requestUserProvider.GetUserId();
    article.CreatedDate = DateTime.Now;
    _articlesRepository.Add(article);
    await _articlesRepository.SaveChanges();
    return RedirectToAction("Index");
  }
  return View(article);
}

public async Task<IActionResult> DeleteConfirmed(int id)
{
  var article = await _articlesRepository.GetOne(id);
  _articlesRepository.Remove(article);
  await _articlesRepository.SaveChanges();
  return RedirectToAction("Index");
}
```

Don't forget to register it on your startup method as a scoped dependency. Otherwise your app would fail at runtime and your integration tests would fail! (See the importance of the different testing methods?)

```
// Add repos as scoped dependency so they are shared per request.
services.AddScoped<IArticlesRepository, ArticlesRepository>();
```

We are almost done!

The controller is still tightly coupled to the `UserManager` class to get the id of the user making the request, so it can be saved as the `AuthorId` field when creating a new article. We can follow the same approach and introduce a new abstraction:

```
public class RequestUserProvider: IRequestUserProvider
{
  private readonly IHttpContextAccessor _contextAccessor;
  private readonly UserManager<ApplicationUser> _userManager;

  public RequestUserProvider(IHttpContextAccessor contextAccessor,
  UserManager<ApplicationUser> userManager)
  {
    _contextAccessor = contextAccessor;
    _userManager = userManager;
  }

  public string GetUserId()
  {
    return _userManager.GetUserId(_contextAccessor.HttpContext.User);
  }
}
```

Updating the controller code to use this new interface should be trivial, but you can always check the code in GitHub.

Our first tests mocking the dependencies

Finally, we are ready to start writing tests since the `ArticlesController` now depends only on two interfaces that we can mock in our tests.

Let's go back to our `ArticlesControllerTest` class and let's update it so that in every test we get a fresh controller instance with mocks injected, using Moq to create and setup the mocks:

```
public class ArticlesControllerTest
{
  private Mock<IArticlesRepository> articlesRepoMock;
  private Mock<IRequestUserProvider> requestUserProviderMock;
  private ArticlesController controller;

  public ArticlesControllerTest()
  {
    articlesRepoMock = new Mock<IArticlesRepository>();
    requestUserProviderMock = new Mock<IRequestUserProvider>();
    controller = new ArticlesController(articlesRepoMock.Object,
    requestUserProviderMock.Object);
  }

  [Fact]
  public void IndexTest_ReturnsViewWithArticlesList()
  {

  }
}
```

This setup is important, as it ensures every individual test gets its own instance of the controller with fresh mocks. It will also help ensuring our test code stays clean and maintainable.

Writing good unit tests is an art in itself. If you need some directions, I would recommend reading the book The *Art of Unit Testing* by Roy Osherove, but by all means, feel free to do your own research and reading!

Let's now implement the first test, verifying that the `Index` method will render the expected view with the expected articles list.

```
// GET: Articles
public async Task<IActionResult> Index()
{
  return View(await _articlesRepository.GetAll());
}
```

The test follows the classic **arrange/act/assert** pattern so it contains these clear sections that arrange any data or mocks required, execute the code under test (act) and perform assertions based on the result:

```
[Fact]
public async Task IndexTest_ReturnsViewWithArticlesList()
{
  // Arrange
  var mockArticlesList = new List<Article>
  {
    new Article { Title = "mock article 1" },
```

```
    new Article { Title = “mock article 2” }
  };

  ArticlesRepoMock
    .Setup(repo => repo.GetAll())
    .Returns(Task.FromResult(mockArticlesList));

  // Act
  var result = await controller.Index();

  // Assert
  var viewResult = Assert.IsType<ViewResult>(result);
  var model = Assert.IsAssignableFrom<IEnumerable<Article>>(viewResult.ViewData.
  Model);

  Assert.Equal(2, model.Count());
}
```

I hope you find it easy to follow, but we are basically setting the behavior of our `IArticlesRepository` mock so that it returns a predefined list of articles. We then call the controller's `Index` method and verify that its result is a `ViewResult` with a model that is an `IEnumerable<Article>` with two elements.

Now run the tests, for example using `Ctrl+R,A` in Visual Studio or `dotnet test` from the command line.

Congratulations, you have written and executed the first test!

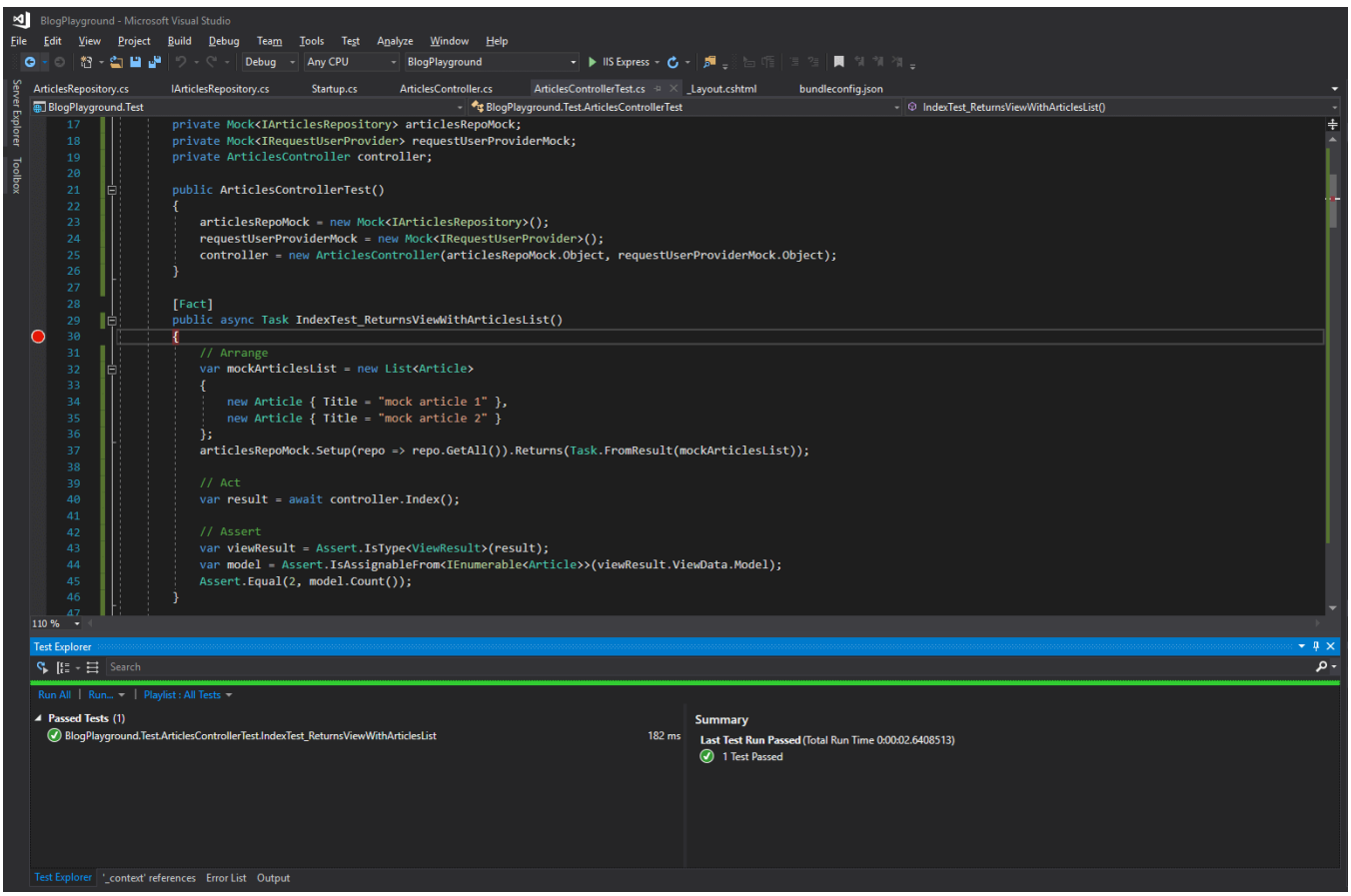


Figure 5, running the first unit test from visual studio

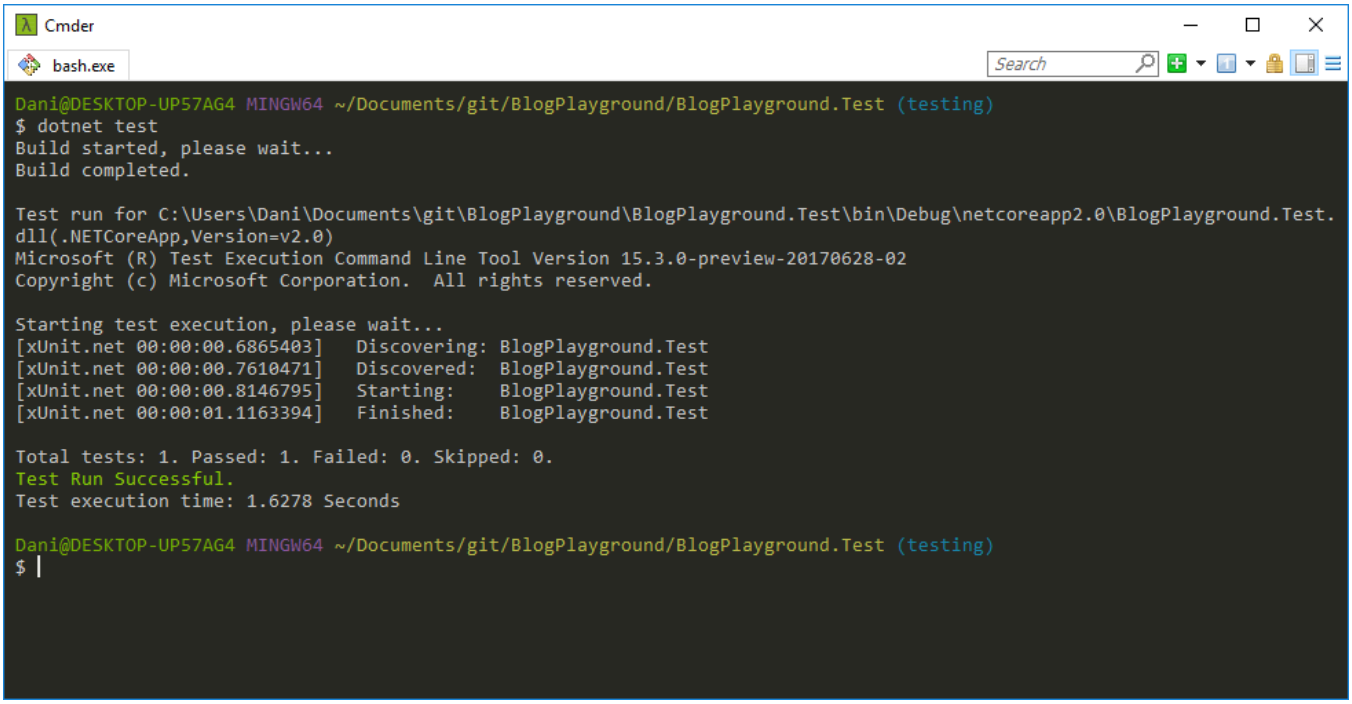


Figure 6, running the unit test from the command line using dotnet test

Additional tests

Let's now move our attention to the `Details` controller method. This one is a bit more interesting since there is some logic involved in making sure we can find an article for the given `id`:

```
// GET: Articles/Details/5
public async Task<IActionResult> Details(int? id) {
  if (id == null) {
    return NotFound();
  }
  var article = await _articlesRepository.GetOne(id.Value);
  if (article == null) {
    return NotFound();
  }
  return View(article);
}
```

We can easily add a test where we supply `id` as `null` and verify that we get a `NotFoundResult` as the response:

```
[Fact]
public async Task DetailsTest_ReturnsNotFound_WhenNoIdProvided(){
  // Act
  var result = await controller.Details(null);
  // Assert
  var viewResult = Assert.IsType<NotFoundResult>(result);
}
```

With a little help from our repository mock, we can add another similar test that verifies we get another `NotFoundResult` when there is no article in the repository with the given id:

```
[Fact]
public async Task DetailsTest_ReturnsNotFound_WhenArticleDoesNotExist() {
```

```
  // Arrange
  var mockId = 42;
  articlesRepoMock.Setup(repo => repo.GetOne(mockId)).Returns(Task.
  FromResult<Article>(null));
  // Act
  var result = await controller.Details(mockId);
  // Assert
  var viewResult = Assert.IsType<NotFoundResult>(result);
}
```

Finally, we can complete the coverage of the controller's `Details` method by verifying that the controller renders a view passing the article from the repository, in case we find an article with the given id:

```
[Fact]
public async Task DetailsTest_ReturnsDetailsView_WhenArticleExists() {
  // Arrange
  var mockId = 42;
  var mockArticle = new Article { Title = "mock article" };
  articlesRepoMock.Setup(repo => repo.GetOne(mockId)).Returns(Task.
  FromResult(mockArticle));

  // Act
  var result = await controller.Details(mockId);

  // Assert
  var viewResult = Assert.IsType<ViewResult>(result);
  Assert.Equal(mockArticle, viewResult.ViewData.Model);
}
```

As you can see, we had to do some groundwork to ensure our class was testable. But once you get to that stage, testing your class is really straightforward.

Before we move on, we are going to write tests for the `Create POST` method (a test for the GET method would be trivial). The code for it is as follows:

```
// POST: Articles/Create
[HttpPost]
[ValidateAntiForgeryToken]
[Authorize]
public async Task<IActionResult> Create([Bind("Title, Abstract,Contents")] Article
article){
  if (ModelState.IsValid) {
    article.AuthorId = _requestUserProvider.GetUserId();
    article.CreatedDate = DateTime.Now;
    _articlesRepository.Add(article);
    await _articlesRepository.SaveChanges();
    return RedirectToAction("Index");
  }
  return View(article);
}
```

We can start by verifying what happens when the model state is not valid. This is as simple as manually setting the controller's `ModelState` property before executing the `Create` method:

```
[Fact]
public async Task CreateTest_Post_ReturnsCreateView_WhenModelStateIsInvalid() {
  // Arrange
  var mockArticle = new Article { Title = "mock article" };
  controller.ModelState.AddModelError("Description", "This field is required");
```

```
  // Act
  var result = await controller.Create(mockArticle);

  // Assert
  var viewResult = Assert.IsType<ViewResult>(result);
  Assert.Equal(mockArticle, viewResult.ViewData.Model);
  articlesRepoMock.Verify(repo => repo.Add(mockArticle), Times.Never());
}
```

Did you observe that the specific assertion verifying the repo wasn't called?

I am adding this because we are writing the tests AFTER the controller code was written. If you follow a TDD process, you would never add code unless a test requires it, which means you won't need to add verifications for things that shouldn't happen!

Next, let's make sure the article gets added to the repository and we are redirected to the index view:

```
[Fact]
public async Task CreateTest_Post_AddsArticleToRepository_AndRedirectsToIndex() {
  // Arrange
  var mockArticle = new Article { Title = "mock article" };
  articlesRepoMock.Setup(repo => repo.SaveChanges()).Returns(Task.CompletedTask);

  // Act
  var result = await controller.Create(mockArticle);

  // Assert
  articlesRepoMock.Verify(repo => repo.Add(mockArticle));
  var viewResult = Assert.IsType<RedirectToActionResult>(result);
  Assert.Equal("Index", viewResult.ActionName);
}
```

The remaining logic to be covered would be ensuring both the `AuthorId` and `CreatedDate` are set in the article that gets saved to the database. Both present a unique interesting challenge since the author is retrieved from the other mock and the creation date depends on the current time:

```
[Fact]
public async Task CreateTest_Post_SetsAuthorId_BeforeAddingArticleToRepository() {
  // Arrange
  var mockArticle = new Article { Title = "mock article" };
  var mockAuthorId = "mockAuthorId";
  articlesRepoMock.Setup(repo => repo.SaveChanges()).Returns(Task.CompletedTask);
  requestUserProviderMock.Setup(provider => provider.GetUserId()).
  Returns(mockAuthorId);

  // Act
  var result = await controller.Create(mockArticle);

  // Assert
  articlesRepoMock.Verify(repo =>
    repo.Add(It.Is<Article>(article =>
      article == mockArticle && article.AuthorId == mockAuthorId)));
}
[Fact]
public async Task CreateTest_Post_SetsCreatedDate_BeforeAddingArticleToRepository()
{
  // Arrange
  var mockArticle = new Article { Title = "mock article" };
  var startTime = DateTime.Now;
```

```
  articlesRepoMock.Setup(repo => repo.SaveChanges()).Returns(Task.CompletedTask);

  // Act
  var result = await controller.Create(mockArticle);
  var endTime = DateTime.Now;

  // Assert
  articlesRepoMock.Verify(repo =>
    repo.Add(It.Is<Article>(article => article == mockArticle
      && article.CreatedDate >= startTime
      && article.CreatedDate <= endTime)));
}
```

You should be able to follow the same steps in order to test the remaining methods, but please check the source code in GitHub if you run into trouble.

Testing API Controllers

The Controller we have tested previously is an MVC controller which returns views and expects form data as input. Let's see an example of how to test an API controller. Since the project didn't have any, let's begin by adding one!

```
[Produces("application/json")]
[Route("api/articles")]
public class ArticlesApiController : Controller {
  private readonly IArticlesRepository _articlesRepository;
  private readonly IRequestUserProvider _requestUserProvider;

  public ArticlesApiController(IArticlesRepository articlesRepository,
  IRequestUserProvider requestUserProvider) {
    _articlesRepository = articlesRepository;
    _requestUserProvider = requestUserProvider;
  }

  [HttpGet()]
  public async Task<IEnumerable<Article>> GetArticles() {
    return await _articlesRepository.GetAll();
  }

  [HttpGet("{id}")]
  public async Task<ActionResult> GetArticle(int id) {
    var article = await _articlesRepository.GetOne(id);
    if (article == null) return NotFound();
    return Ok(article);
  }

  [HttpPost()]
  [ValidateAntiForgeryToken]
  [Authorize]
  public async Task<ActionResult> AddArticle([FromBody]Article article) {
    if (!ModelState.IsValid) return BadRequest(ModelState);

    article.AuthorId = _requestUserProvider.GetUserId();
    article.CreatedDate = DateTime.Now;
    _articlesRepository.Add(article);
    await _articlesRepository.SaveChanges();
    return Ok(article);
  }
```

```
  [HttpDelete("{id}")]
  [ValidateAntiForgeryToken]
  [Authorize]
  public async Task<ActionResult> DeleteArticle(int id) {
    var article = await _articlesRepository.GetOne(id);
    if (article == null) return NotFound();
    _articlesRepository.Remove(article);
    await _articlesRepository.SaveChanges();
    return NoContent();
  }
}
```

It is a fairly standard API controller. It provides a similar functionality to the previous controller intended to be used as a REST API, that sends and receive JSON data.

If you read through the code, you will notice we have again used dependency injection and the `IArticlesRepository` and `IRequestUserProvider` abstractions so we can unit test it as well. That means we can write tests in the same way we did before. For example, this is how we would write the first simple test that verifies that the `GetArticles` method works as expected:

```
private Mock<IArticlesRepository> articlesRepoMock;
private Mock<IRequestUserProvider> requestUserProviderMock;
private ArticlesApiController controller;

public ArticlesApiControllerTest() {
  articlesRepoMock = new Mock<IArticlesRepository>();
  requestUserProviderMock = new Mock<IRequestUserProvider>();
  controller = new ArticlesApiController(articlesRepoMock.Object,
  requestUserProviderMock.Object);
}

[Fact]
public async Task GetArticlesTest_RetursArticlesList() {
  // Arrange
  var mockArticlesList = new List<Article> {
    new Article { Title = "mock article 1" },
    new Article { Title = "mock article 2" }
  };
  articlesRepoMock.Setup(repo => repo.GetAll()).Returns(Task.
  FromResult(mockArticlesList));

  // Act
  var result = await controller.GetArticles();

  // Assert
  Assert.Equal(mockArticlesList, result);
}
```

I hope it is evident that we are structuring our tests in exactly the same way, with the arrange/act/assert stages and we keep mocking our dependencies as before.

Let's see something a bit more interesting, like the tests for the `AddArticle` method which returns an `ActionResult`:

```
[Fact]
public async Task AddArticleTest_ReturnsBadRequest_WhenModelStateIsInvalid() {
  // Arrange
```

```
  var mockArticle = new Article { Title = "mock article" };
  controller.ModelState.AddModelError("Description", "This field is required");

  // Act
  var result = await controller.AddArticle(mockArticle);

  // Assert
  var actionResult = Assert.IsType<BadRequestObjectResult>(result);
  Assert.Equal(new SerializableError(controller.ModelState), actionResult.Value);
}

[Fact]
public async Task AddArticleTest_ReturnsArticleSuccessfullyAdded() {
  // Arrange
  var mockArticle = new Article { Title = "mock article" };
  articlesRepoMock.Setup(repo => repo.SaveChanges()).Returns(Task.CompletedTask);

  // Act
  var result = await controller.AddArticle(mockArticle);

  // Assert
  articlesRepoMock.Verify(repo => repo.Add(mockArticle));
  var actionResult = Assert.IsType<OkObjectResult>(result);
  Assert.Equal(mockArticle, actionResult.Value);
}
```

There is nothing new in the preceding code, other than adapting our tests to the different `ActionResult` return types. As an exercise, you can try completing the coverage of the controller. Check the code on its GitHub repo if you run into trouble.

Client-side Unit Tests

If you are writing a web application, it is very likely that your client-side JavaScript code is as complex (if not more) than your server-side code. This means you will want to give it the same test treatment than your server-side code will receive. While we won't be covering this in the article, the principles stay the same and only the tooling changes!

You will still need:

- A test framework like Mocha or Jasmine (depending on whether your tests run purely with Node.js or require a browser, even a headless one like Phantom.js)
- A mocking library like Sino n.JS and a tool like proxyquire to replace module dependencies with mocks.
- More importantly, you still need to isolate your units from its dependencies and write readable and maintainable tests.

The reason why I am not covering client-side testing here is that the actual approach and tooling is greatly influenced by the client-side framework you are using. I would encourage you to do some research and add unit tests adapted to your framework of choice. The dynamic nature of JavaScript makes them even easier!

Conclusion

Unit Tests are a powerful tool available to any developer. Having good unit tests covering the logic intensive areas of your application, will lead you towards a better design and will allow you to work without being

afraid of modifying your code.

This ability to exercise your code without having to fully start your application cannot be overstated enough and will shine even more when trying to reproduce and fix a bug. Once you get used to working in projects with tests, moving to one without them, feels like a huge step backwards!

But don't be deceived! Writing good unit tests is an art of which I merely brushed a few strokes.
There is a learning curve, but it's really worth it! Bad unit tests will do more harm than good ones and end up slowing you down, without adding much real value.

During the article we have added unit tests to a simple example ASP.NET Core project that provided a very straightforward API to manage blog posts. Although useful to showcase how to make your code testable and how to write unit tests, I would be really surprised if by this stage you didn't have the following concerns given the effort required:

- How can you make sure that the simple repository class you added works as expected?

- What happens with all the attributes you use in your controller class and methods that greatly affect the final behavior of your application in areas like routing, binding or security?
- How can these tests catch simple errors, like the one where you forget to register the new interfaces within the `ConfigureServices` method of your `Startup` class?

- Is the effort and discipline required to keep your code testable and to write and maintain these unit tests, worth it?

That's why besides unit tests, you also need to consider **integration** and **end-to-end** tests in your testing strategy.

You want full coverage with unit tests on the areas of your application that are rich in logic, particularly the core of your application where your specific business rules live. Dealing with external dependencies, infrastructure code and plumbing is pushed to the boundaries from this core via the right abstractions.

Higher level tests like integration and end-to-end tests will make sure that your entire application works together as expected. If unit tests were done right, you will just need to test a reduced number of scenarios to make sure your code and its dependencies play along as expected.

Daniel Jimenez Garcia

Author

Daniel Jimenez Garcia is a passionate software developer with 10+ years of experience. He started as a Microsoft developer and learned to love C# in general and ASP MVC in particular. In the latter half of his career he worked on a broader set of technologies and platforms while these days is particularly interested in .Net Core and Node.js. He is always looking for better practices and can be seen answering questions on Stack Overflow.

Thanks to Damir Arh for reviewing this article.

DocuVieware 3

HTML5 Viewer & Document Management Kit

NEW RELEASE

Easy integration

Full support for custom snap-in

Zero-footprint solution

Fully customizable UI

Mobile devices optimization

Fast & crystal-clear rendering

Check the **New Features** and the **Online Demos**

DOWNLOAD YOUR FREE TRIAL www.docuvieware.com

Powered by GdPicture.NET 14

DocuVieware is an ORPALIS imaging technologies product

Rahul Sahasrabuddhe

A Primer on Machine Learning

Machine Learning as a concept or term is fairly known in the tech world now. This article is a short and sweet primer on this topic. After reading this article, you can confidently have a 10 minute elevator talk to anyone on Machine Learning.

Introduction

The term "Machine Learning" learning was coined somewhere in the 1950s. However, of late, there has been a lot of exploration, analysis etc. around Machine Learning. The Google Trend graph in Figure 1 justifies this well enough.

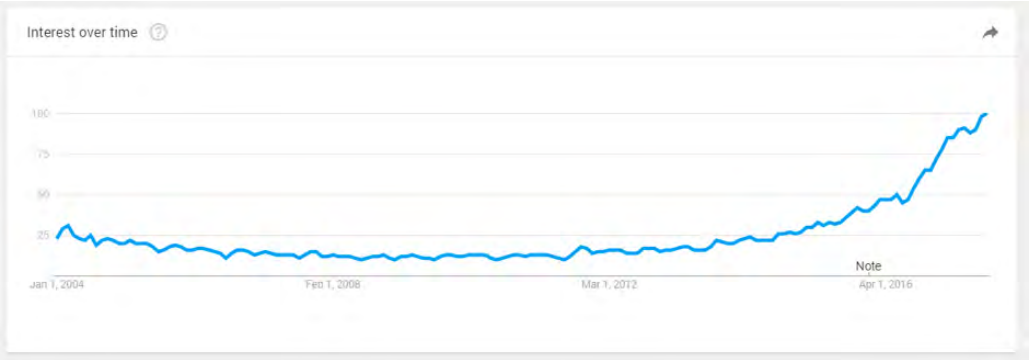


Figure 1 : Google Trend graph showing ML is trending

So, what is Machine Learning (ML)?

Well, learning anything is challenging. Building machines is challenging too. So, making machines learn would sound very challenging as well!

Isn't it so?

Let's first find out why do machines need to learn in the first place.

We have so many machines or gadgets around us that are used for various purposes. They help us perform different tasks. Let's take a simple example.

You may usually have a nicely brewed coffee every morning. So, you typically would be using a coffee maker "machine" that makes coffee after you have selected your options - like what kind of coffee you'd like to have, how much milk, how much sugar and so on.

Now, what if, the "machine" *learns* how to make the coffee for you based on your daily routine. Wouldn't that be cool? Indeed, it would be!

So, if your coffee machine is taught to study your routine behavior so that you get your daily coffee exactly how you like it, it will make your life easier. If the coffee machine can also identify you through face recognition and can make the coffee that you like (and not how your friend/companion likes), that would be really awesome!

That's machine learning for you!

In a nutshell, machine learning can be applied to situations where:

1. A machine is used for a specific task to be completed - obviously. The machine term here is an all-encompassing entity as it would involve some app/program/appliance/device/system that is being used.

2. The behavior is repeatable and predictable so that past data can be used to predict the future actions.

3. Behavior is pattern-based or rule-driven so that it can be "taught" to the machine. That will help machines to learn and match data against a pattern to take actions or decisions.

4. Large volume of data is being processed. This will typically hold true when it is humanly impossible to look at the specific data elements in the sea of data to identify a potential issue or problem.

Why is ML so important now?

If ML was existing as a concept from 1950s, then one would wonder as to why has it suddenly become so important or has come into vogue?

Well, amongst many others, here are some key driving factors:

1. **Connectivity of machines:** Generally speaking the internet connectivity aspect or ability for devices to be online or simply put, the Internet of Things (IoT) has made it easy for devices to be connected with each other (and servers) in real-time. So the data that was lying with machines now, is available for processing over cloud.

2. **Data** – lots of it: Devices/machines collect a lot of data in real time and with the connectivity available, they can now transmit all this data in real time. By leveraging all of this data, we can gain insights into user behavior. The ever-expandable elastic storage power through the cloud makes it easier to handle large volumes of data and process it on the cloud.

3. **Processing power:** Moore's law still holds true. Devices have got enough processing power now to handle loads of data at the edge (i.e. on the device).

All these three aspects put together have resulted into an exciting opportunity to apply ML techniques to solve business problems.

ML is touching your lives already!

If you are thinking that ML is some futuristic Sci-Fi stuff, then it is not!

You are already living in a digital world where ML is touching your lives on a daily basis. You don't believe?

Here are some examples of ML being used already:

1. If you have a Gmail account, Google's Priority Inbox features uses ML to categorize the mails as priority/spam from the mail avalanche that hits your email account every single day. It learns over time as to which emails are important to you, and takes actions accordingly. It also uses ML to validate the email

messages for phishing attack related threats.

2. Google's self-driving car or for that matter the famous Tesla cars use ML extensively while driving itself by mapping what it "sees" to what it has learnt in terms of digitized imagery of the terrain around.

3. Online recommendation based on your purchases are pushed by Online commerce giants like Amazon while you surf for products. It also uses ML to provide the most competitive price to customers during online shopping. The same technique is being used when you book online tickets, to show online ads, or if you are into online dating websites.

There are many more similar examples where ML has already "arrived" in your lives. With more connected devices coming up, it is only going to get more and more integrated (and interesting) and eventually ML would become an integral part of your day-to-day life.

In many cases, this integration will be seamless. You won't even notice it.

The Technical ABCs of ML

Let's delve into some technical basics of ML now. Let's start with the generic approach followed in any machine learning process.

Any ML process would have the following key steps:

1. **Data collection & preparation:** You will need to identify the right data sources and prepare data that can be fed to the ML algorithm to be used.

2. **Learning:** This includes a key aspect of choosing the right ML algorithm to be used to get the right results. You would use the dataset as the outcome of step 1 here, to feed to the model so that it learns accordingly. The aspects considered while choosing the right algorithm could be:

   a. Complexity of the data
   b. Choice of data set

3. **Prediction**: In this step, you would use the model or algorithm to predict outcomes for you based on the data model supplied or provided.

The following diagram depicts the flow:

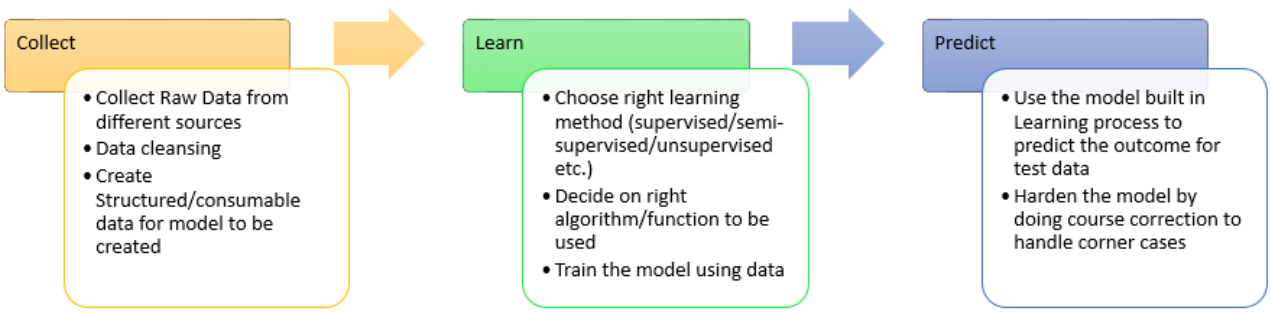


Figure 2 : the ML Learning Process

Let's look at the various ML techniques used currently.

Supervised Learning

In supervised learning, you build a model that can be used for prediction. When you have a function `f(x)` that predicts the result `y` for the given `x` data, then it is termed as supervised learning.

An important aspect to note here is the use of the right approximation algorithm as the function, so that it predicts new `y` values based on past data used to teach.

This process of an algorithm that is used to learn from an existing dataset is synonymous to a teacher supervising the learning process. So, we train the model with a known understanding about the kind of data we are using (i.e. the labelled data).

Hence it is termed as **supervised learning**.

Typically, the steps involved are:

1. Provide known/labeled data and known output to algorithm to learn, so that it creates the model.

2. The model coupled with the algorithm that has "learnt" would result into predicted response for a new data set entered.

Supervised Learning is used when you need an output in the following two forms:

1. **Classification**: You need the ML algorithm to output a category from given categories – something like priority mail or spam mail.

2. **Regression**: Over here, the ML algorithm would output a real value like predicted sales values based on input product prices.

Various techniques are used in supervised learning. Some of them are as follows:

1. **Decision trees:** the algorithm would use some threshold values for classifying the data.
2. **Support Vector Machines:** This involves binary classification of data
3. **ANN (Artificial Neural Network):** ANN concepts are used for data classification in this case.

Unsupervised Learning

When you have a cluster of data and you would like it to be analyzed to deduce conclusions or infer from it, then it is termed as unsupervised learning.

The basic premise here is that there is no specific end goal in mind and hence there is no "teaching the model" aspect involved.

The data is also not really labelled in this case since this is more of an exploration of figuring out the pattern, from an available dataset. So you are essentially taking a bulk of data and working on finding the hidden pattern by doing the cluster analysis of the data you have.

There are various algorithms used for doing this. Some of them are mentioned below:

- **k-Means Clustering:** this essentially partitions data in various distinct clusters
- **Hierarchical clustering:** the algorithm attempts to build a hierarchy of clusters in a tree-like fashion
- **Self-organizing maps:** it uses the concepts of neural networks to build the clusters from data

Semi-supervised Learning

This technique essentially addresses a specific need of machine learning where we have some understanding of output to be generated from input. However, this is not true for all the data. So, this technique is essentially a combination of supervised and unsupervised learning.

You can use various combinations of supervised and unsupervised learning algorithms to get desired result.

Reinforced Learning

This is another variation of ML where the concept of reward or penalty is applied to learning process. This is similar to students taking an examination in order to evaluate how the student (in this case the algorithm) has performed in the learning process.

Learning ML techniques by Example

Let's try and understand the different types of ML learning techniques we just learnt about, with an example.

Let's say that you want to build a **machine learning algorithm for face detection**. Here is how to apply the different ML methods to make this possible:

You have a dataset of images labelled as faces (based on attributes like round face, a specific area of photo representing skin color etc.). You can use this dataset to train the model to identify a face. You can then provide a new image to the algorithm for recognizing if it is an image of a face, or otherwise.

The algorithm would logically compare the attributes of the test image with the images from which it has learnt how to recognize a face. Using this logic, the algorithm would predict if the image is a face or not. **This is supervised learning.**

If you don't provide a labelled data, the ML algorithm would try to cluster (group) the data into face-like shapes (round shape) based on various properties defined in the model (the roundness of a shape, proximity of the shape to some circular figure etc.). Based on this clustered data, the ML algorithm would recognize the faces. **This is unsupervised learning.**

The third possibility is a mixed set of data with some labelled data and some unlabeled data. So, when you would use labelled data (i.e. images labelled as faces), the ML algorithm will first learn from labelled data. Then you can use the trained model to predict labels from unlabeled data (i.e. images that

are not labelled as faces). This is called pseudo-labelled data. And then you use both labelled and pseudo-labelled data, to retrain the model. **This is semi-supervised learning.**

The fourth possibility is that you use an algorithm to recognize the face first and then have the algorithm decide the success of recognizing the face by coming up with some success coefficient. If the coefficient is closer to some value, then it means that the algorithm has succeeded, else it has to re-evaluate. T**his is at a very high level, reinforced learning process.**

Deep Learning

A quick note here – Deep Learning (DL) is a term used quite often alongside Machine Learning.

Deep Learning is a subset of ML and closely tries to mimic the structures (neurons) utilized by our brains. It is essentially taking ML to the next level since DL algorithms go really deep in terms of handling data. It is a specialized way of handling data where in the learning part is more human-like in terms of execution process, as compared to ML.

Editorial Note: *Deep learning is a subject of its own. To know more about what Deep Learning can really do, check out The Dark Secret at the Heart of AI.*

Future of Machine Learning

Machine Learning has eventually matured as a specialization that compliments various key areas like analytics, AI and so on.

As explained earlier, the overall learning process is divided into three parts:

1. Data collection
2. Learning
3. Prediction

As things stand now, the data collection happens on the edge (i.e. device), whereas the data processing & learning happens on the cloud. This is how traditionally it has been envisioned.

However, things are supposed to get more interesting!

Once the learning from data is accomplished, the "learnt" logic can be pushed back to device/edge and then the prediction part can happen on the edge again. This kind of approach is very critical and important when:

1. The decision making based on ML needs to happen fast enough: Let's consider a case where ML is used to predict failure of some component and based on that, some other decisions are required to be taken at that very instant. In this case, it is vital to have the learnt logic available at the edge for further decision making process.
2. Data privacy: In case of using ML in context of medical or healthcare use cases, it is prohibited to have the data move out of healthcare facility due to privacy issues or compliance restrictions. In such cases, the prediction needs to happen at the edge.

So, it is not going to be a distant future when the AI/ML combination is going to make the edge more

intelligent. So, in a near future, after you drink a full cold bottle of your favorite fizzy drink from the refrigerator, the refrigerator would tell Alexa to order medicine for cough and cold for you because in some parts of the world, you always fall sick after drinking a cold drink!

Azure Machine Learning Studio

Enough of theory; let's look at how you can actually build an Azure Machine Learning experiment yourself, using **Azure Machine Learning Studio**.

Editorial Note: *Microsoft has long been at the forefront of Machine Learning and AI. After being successful at Hadoop-based Azure HDInsight and PowerBI for Office 365, Microsoft channeled its efforts to consumerize ML. In addition to solving existing business problems using ML, Microsoft also wanted to make available the breakthroughs of ML to those developers and engineers, who did not have a ML background.*

Azure Machine Learning (also known as "Azure ML") was a step in that direction. Azure ML runs on the Azure public cloud and is built on top of the machine learning capabilities of several Microsoft products and services. ML Studio is a simple browser-based integrated development environment. It allows you to visually drag-and-drop elements where no to little coding is necessary.

Open up your favorite browser and go to http://studio.azureml.net. As goes with all Microsoft offerings, this is a DIY studio for you to get your hands dirty with ML.

You will need an Azure account to login. **Get a free trial** account at https://azure.microsoft.com/en-us/free/ if you don't have one already.

Once you login, you will be presented with the following UI.

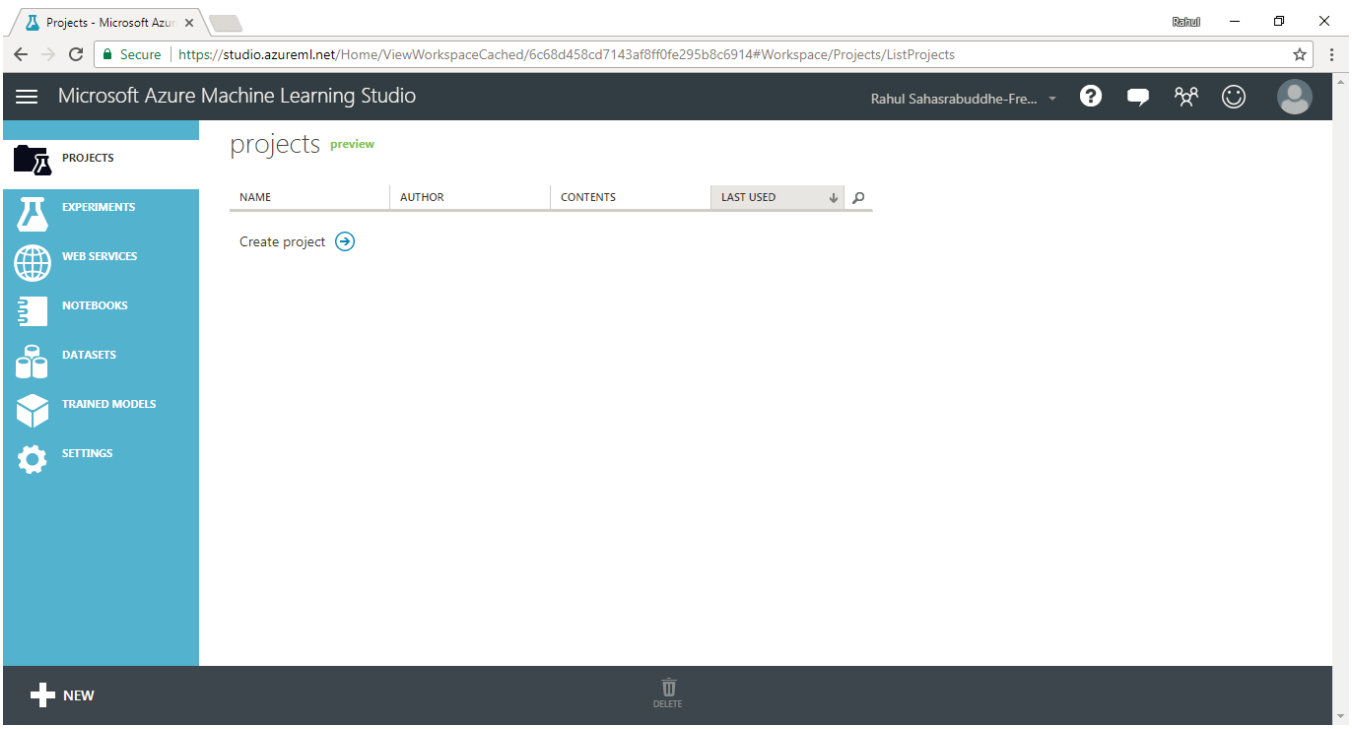


Figure 3 : Azure ML Studio Workbench

To follow a detailed documentation for ML Studio, head over to docs.microsoft.com/en-us/azure/machine-learning/studio/.

Azure ML Studio organizes things in the following manner:

1. **Projects**: This is like any other project in Visual Studio that will have all the required nuts and bolts for you to perform your ML tasks.

2. **Experiments**: A project would contain multiple experiments that you would like to carry out. You can add various datasets and modules. You need to have at least one dataset and one module in the experiment.

3. **Datasets**: Azure ML Studio has some sample datasets available that you can play around with.

4. **Module**: This is essentially an algorithm that you would use for analyzing your datasets

So you get the drift now. Let's create our first experiment. Please follow the step-by-step guide below:

Step 1: Create a new experiment: Click on the "New" button on the bottom left corner of the screen and then select "Blank Experiment".

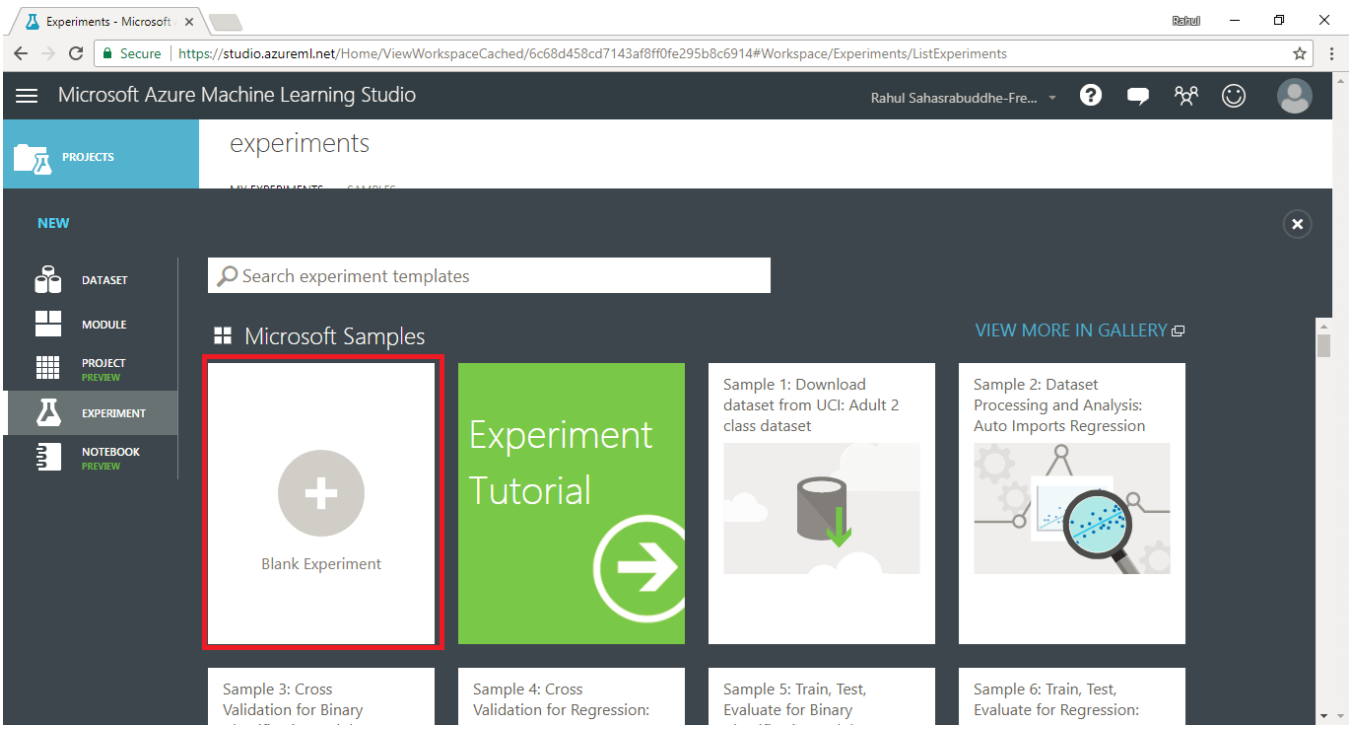


Figure 4 : Create an Experiment

You can also go through the Experiment Tutorial or if you would like to check out some sample experiments, you can select the appropriate options. Azure ML Studio has provided a lot of samples as shown in Figure 5.

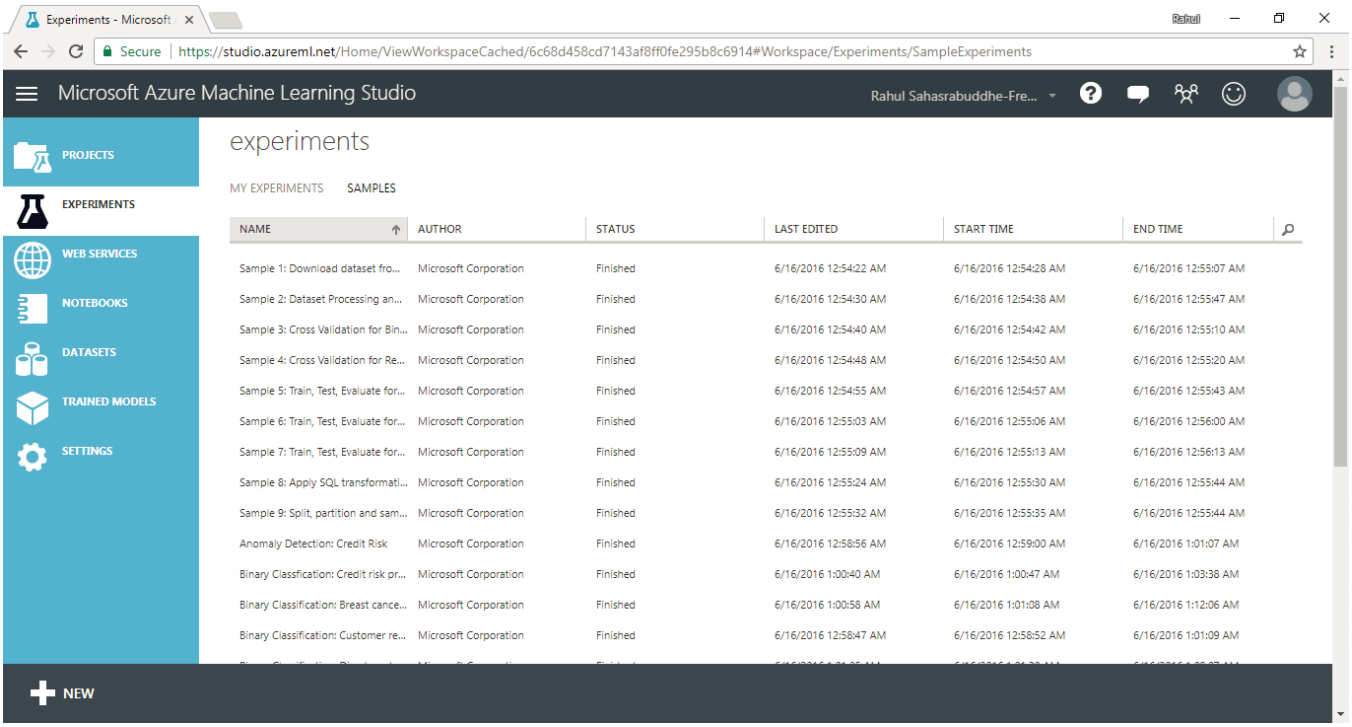


Figure 5 : Azure ML Studio Samples

After clicking on Blank Experiment, you will be presented with the following workbench:

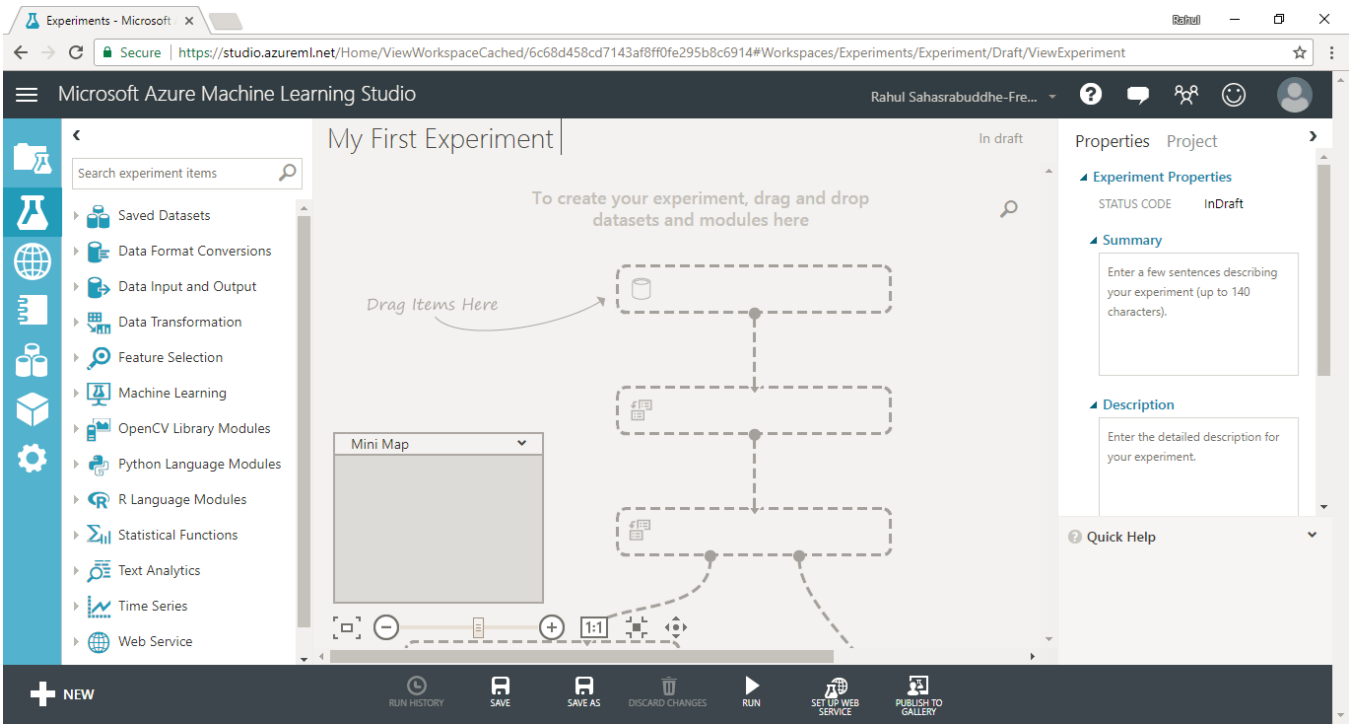


Figure 6 : Creating a Blank Experiment

Step 2: Now, for us to understand Machine Learning well enough, we need to apply the concepts of ML explained earlier, to solve some specific real-life problem.

As mentioned earlier, one of the key pre-requisites for ML is "data". So, we need to first get some real-life data in order to apply ML and analyze a real-life problem.

There are many data sets available online for free that you can choose from and then you will have to

import them into the Azure ML Studio. Alternatively, you can pick up one of the data sets from Azure in order to understand how ML works.

Step 3: Let us pick up the data set "Energy Efficiency Regression Data" from the available sample datasets in Azure ML Studio. If you want to pick up some other data set, then please go through docs.microsoft.com/en-us/azure/machine-learning/studio/use-sample-datasets that provides a brief background about the various sample data sets provided by Azure.

Let us type in "energy" in the search box. That will bring up Energy Efficiency Regression data as shown in Figure 7. Let's drag it and drop it on to the workspace.

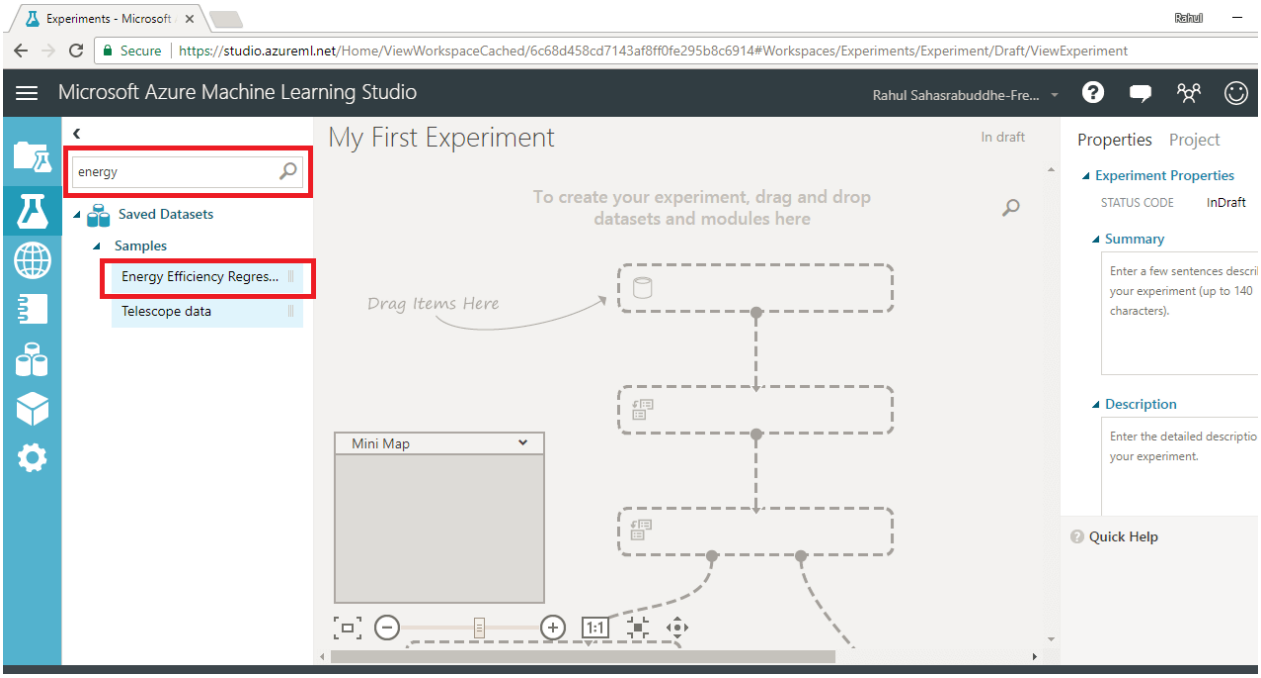


Figure 7 : Choosing a Data Set

The workspace should look something like this:

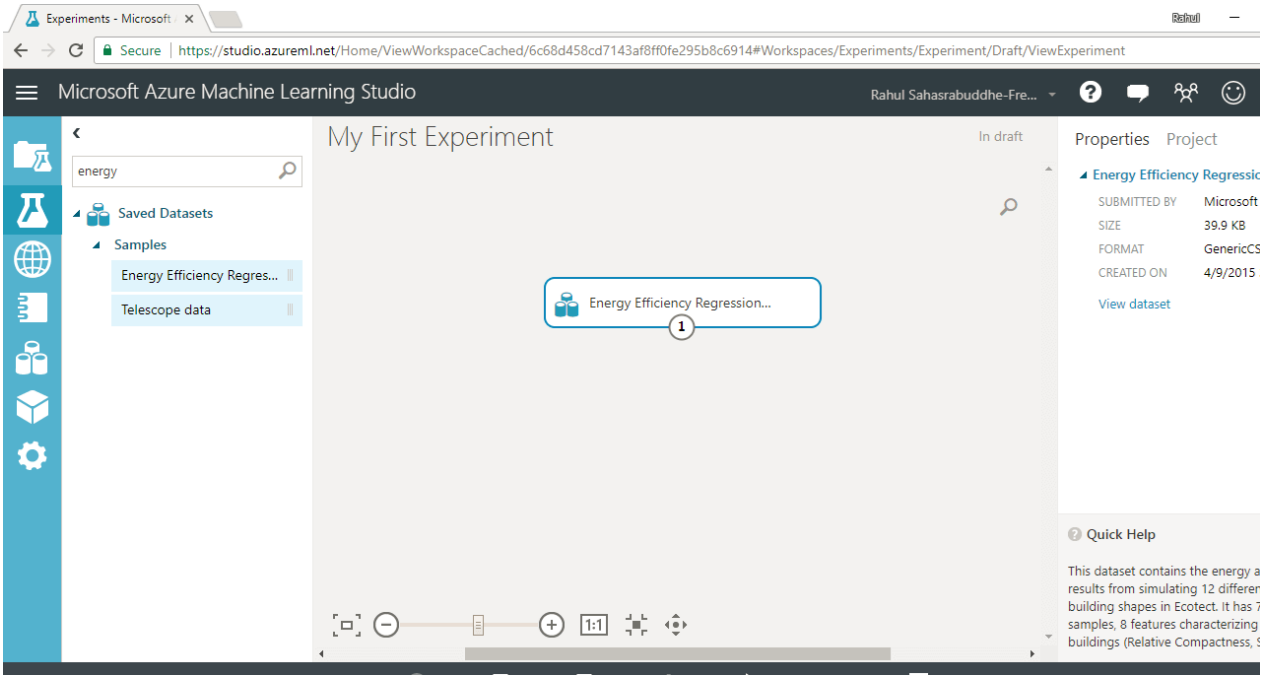


Figure 8 : Data Set is chosen

Please note that you can also import the data from your data source. You can refer to Azure ML Studio documentation that explains this well.

Step 4: The sample data set that we have chosen is a collection of energy profiles for twelve different buildings of various shapes.

The key objective here is to use ML technique to help us assess energy efficiency of these buildings. For this, we will need to understand the factors that affect the heating or cooling of the building. I would request you to skim through a paper (people.maths.ox.ac.uk/tsanas/Preprints/ENB2012.pdf) to get an in-depth idea of the problem we are trying to solve by using ML.

Step 5: Before we proceed, we can take a peek at the sample data set. To do so, you can right click on the data set and click on "Visualize".

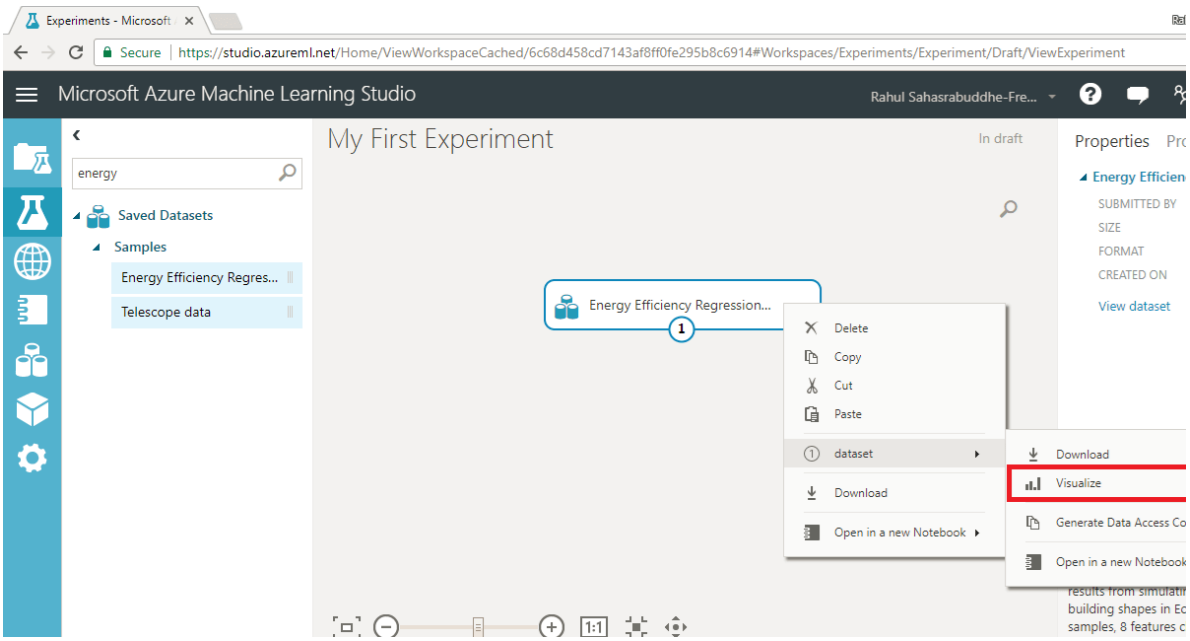


Figure 9 : Visualize Data Set

The following data gets displayed:

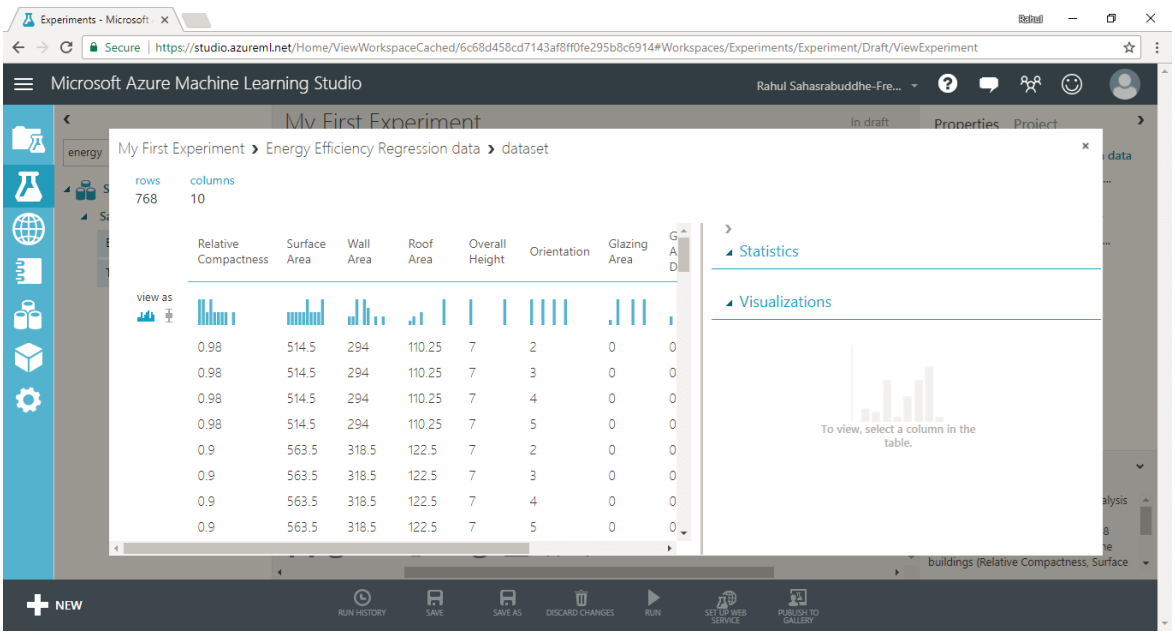


Figure 10 : Visualized Data Set

Step 6: Now you can view statistics and visualization for each column. Just click on any column and you will see the details about that column updated in "Statistics" and "Visualization" sections as shown in Figure 11:

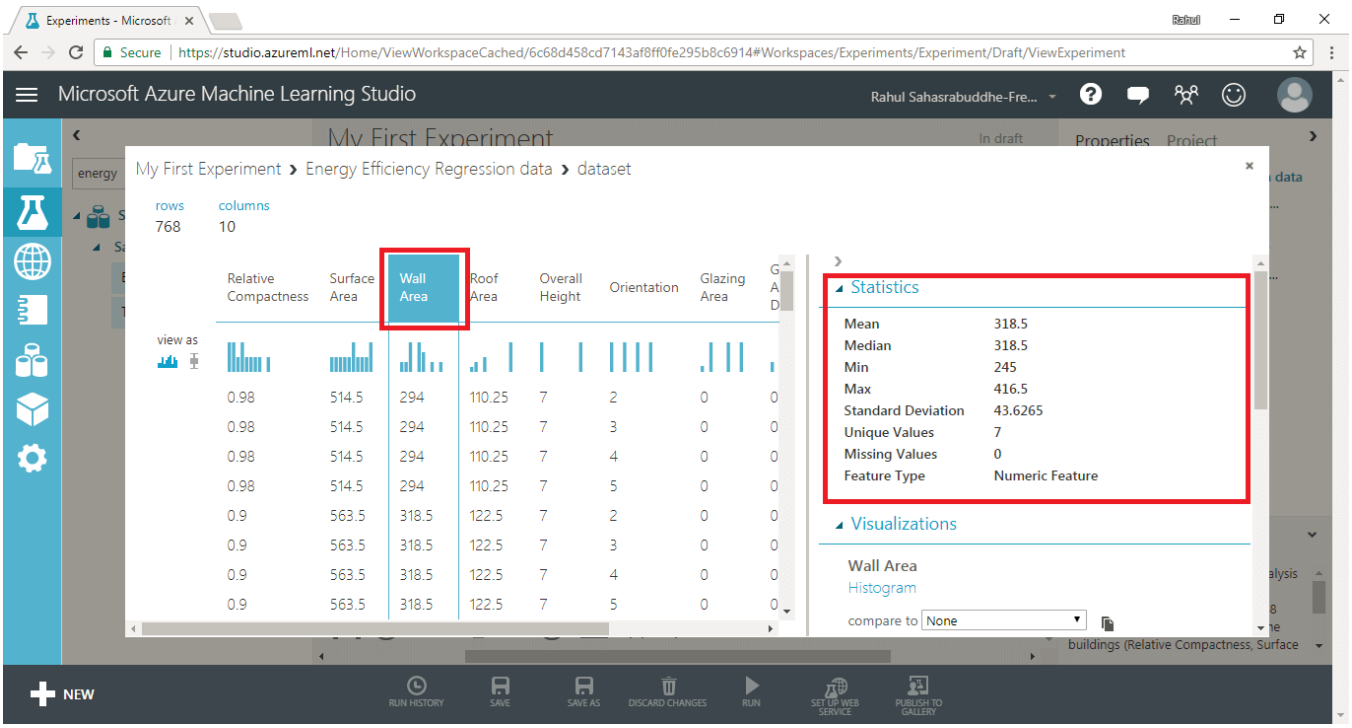


Figure 11 : Data Set Statistics

I have chosen another column (Grazing Area Distribution) and after scrolling down a bit, I see the data visualized in a bar graph as shown in Figure 12.

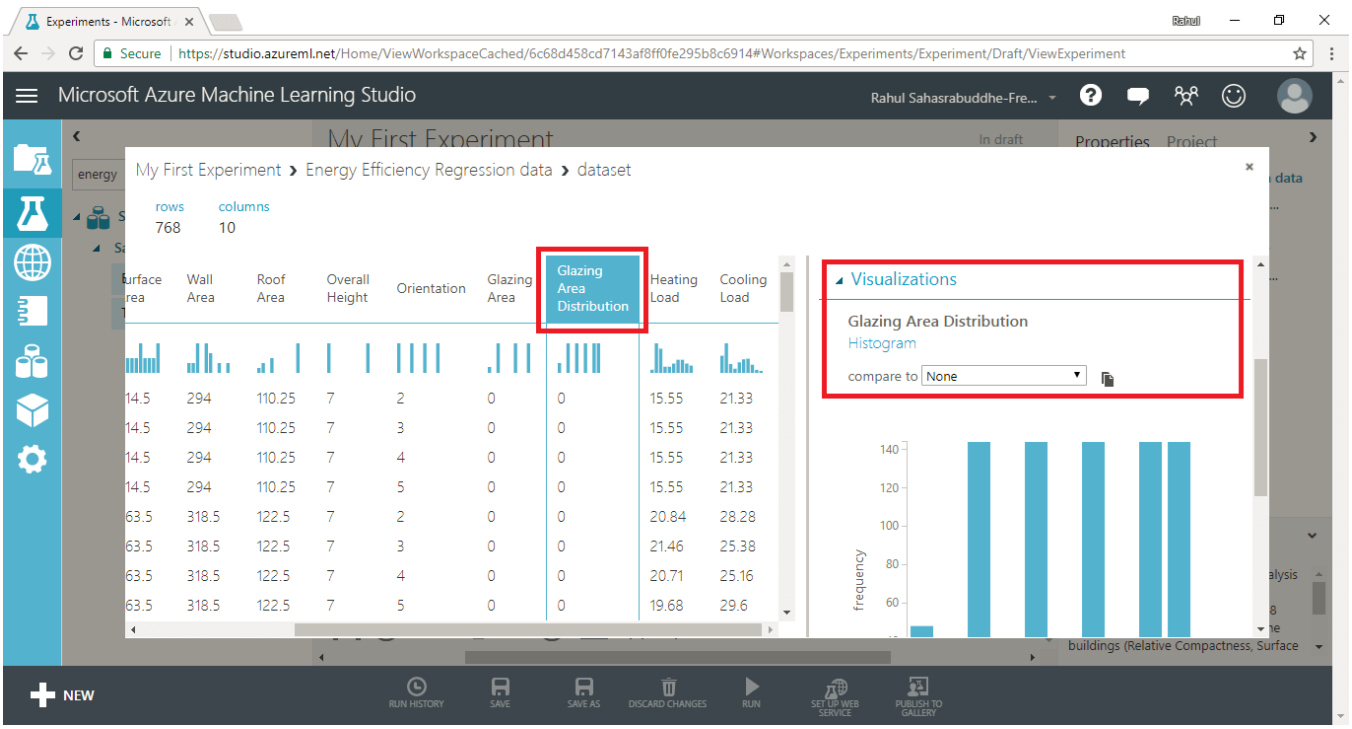


Figure 12 : Viewing Data Set Statistics for other columns

Step 7: We will select some columns from the dataset, for further processing. To do so, you can enter "select column" in the search box and then drag the "Select Column in Dataset" module onto the workspace. Your screen should appear similar to Figure 13:

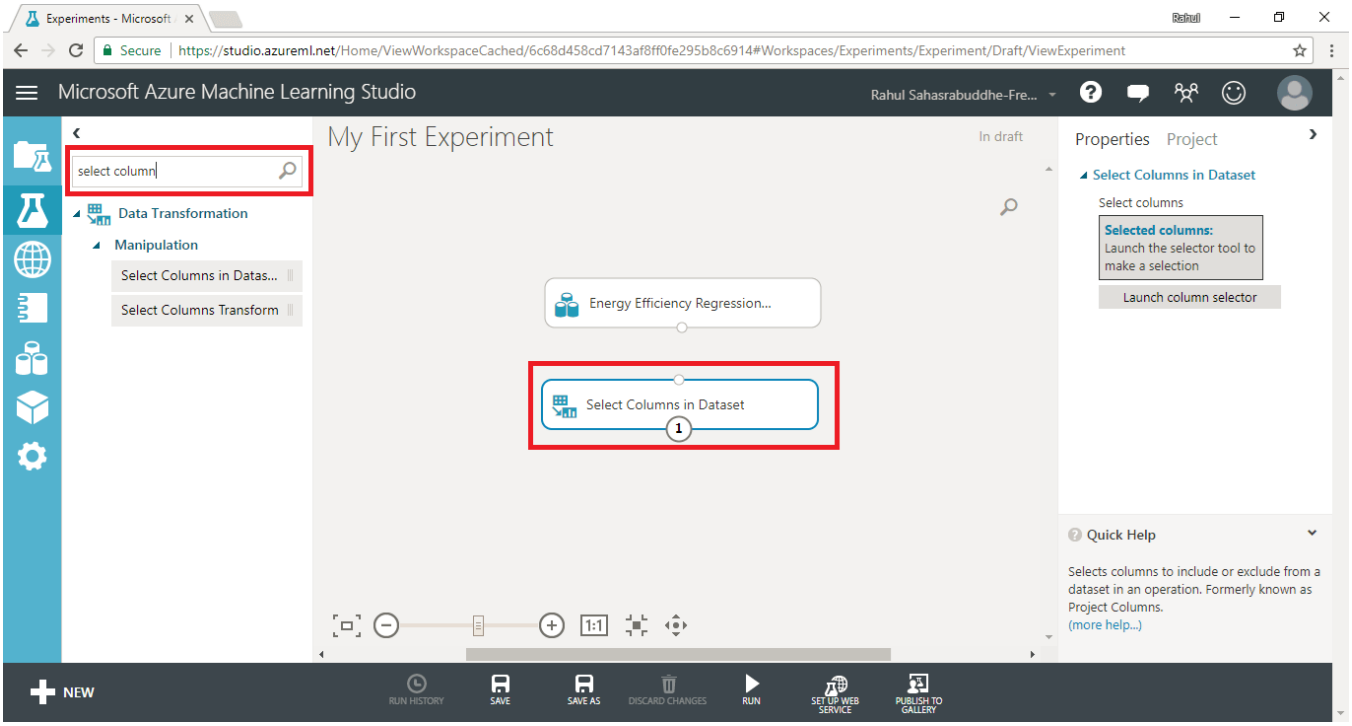


Figure 13 : Using Select Column in Dataset module

After I click on the data module, it shows me the following items:

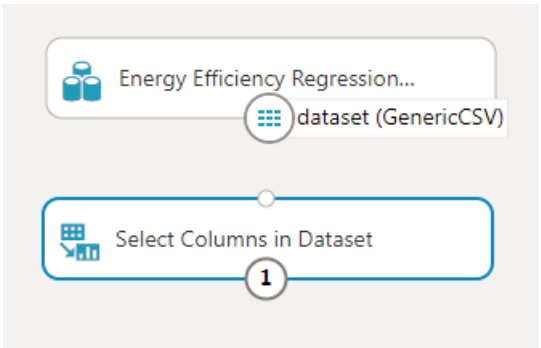


Figure 14 : Select Column in Dataset chosen

We will join the boxes and as you can observe in Figure 15, there is a red exclamation icon that appears.

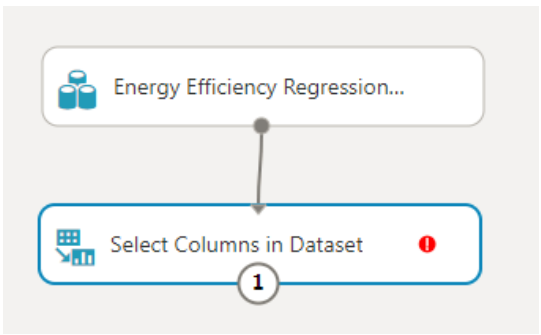


Figure 15 : Select Column in Dataset with error icon

The red icon means that I need to provide additional information to proceed further. We will click on "Launch column selector" as shown in Figure 16 to select columns.

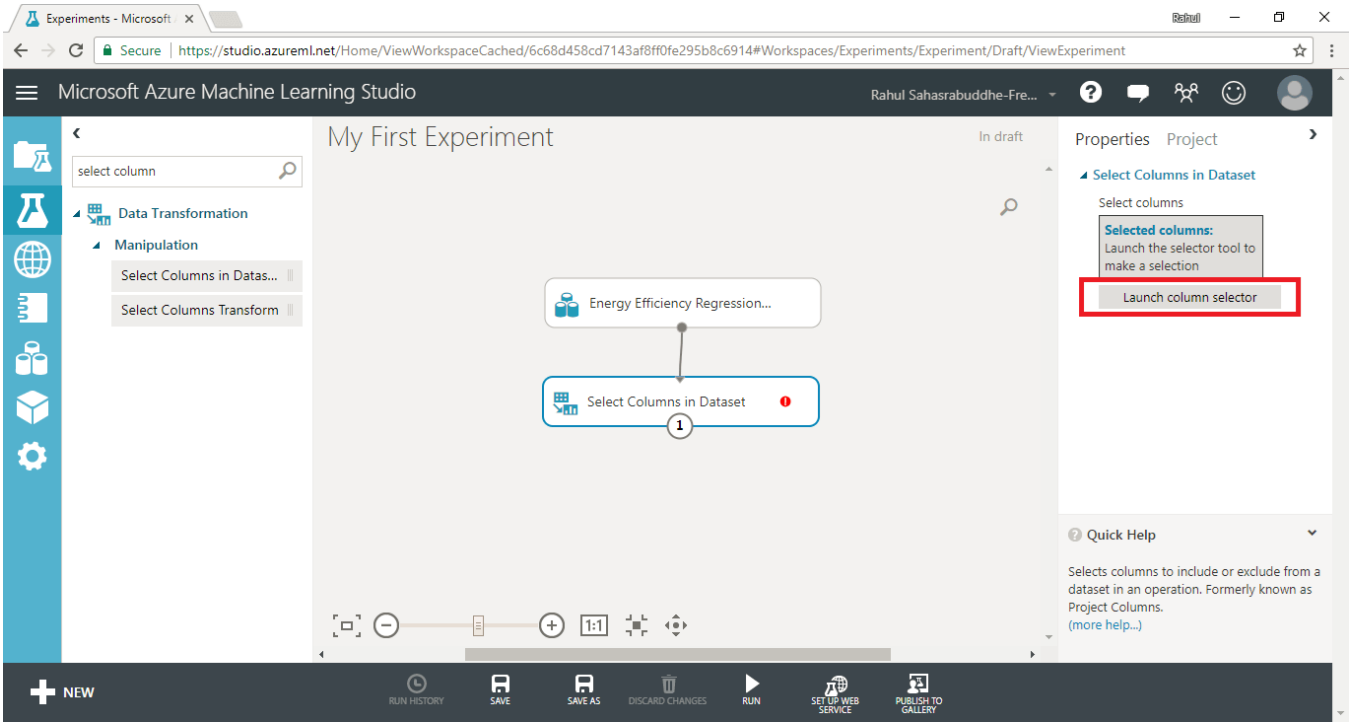


Figure 16 : Launching Column Selector

Doing so brings up a dialog box (Figure 17) with all the data columns shown towards the left. As we choose them, they will move to right. This is self-explanatory and a standard operation. Basically, we are choosing the columns that we want to use to build our model.

These are also called as “features” in ML parlance.

We are leaving out “Cooling Load” for now because we are going to predict only “Heating Load” (refer to the research paper for details).

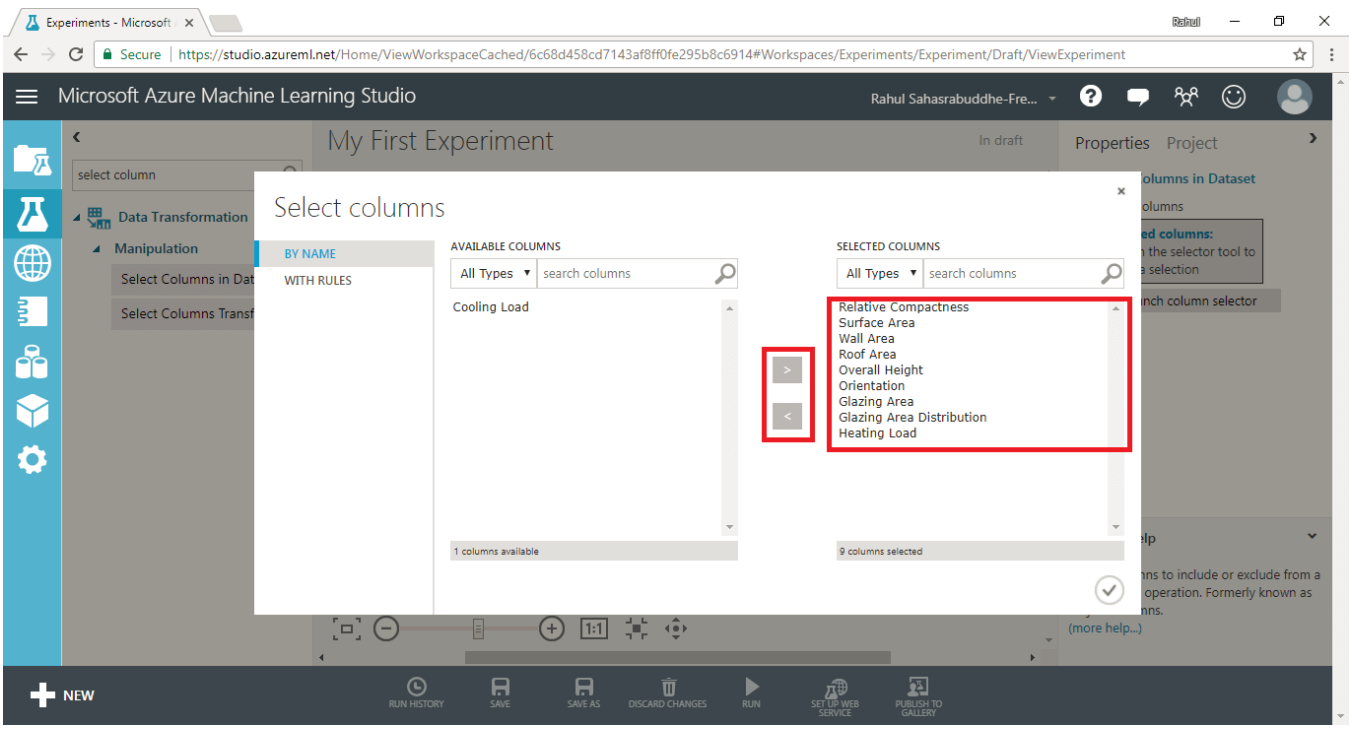


Figure 17 : Selecting features

Step 8: Now close the dialog box after clicking on the tick mark on the bottom right. Now click on the "Run" button at the bottom. This will make Azure ML Studio save and run the model. If everything goes well (and it will), you will see the model run successfully with a green tick as seen in Figure 18.

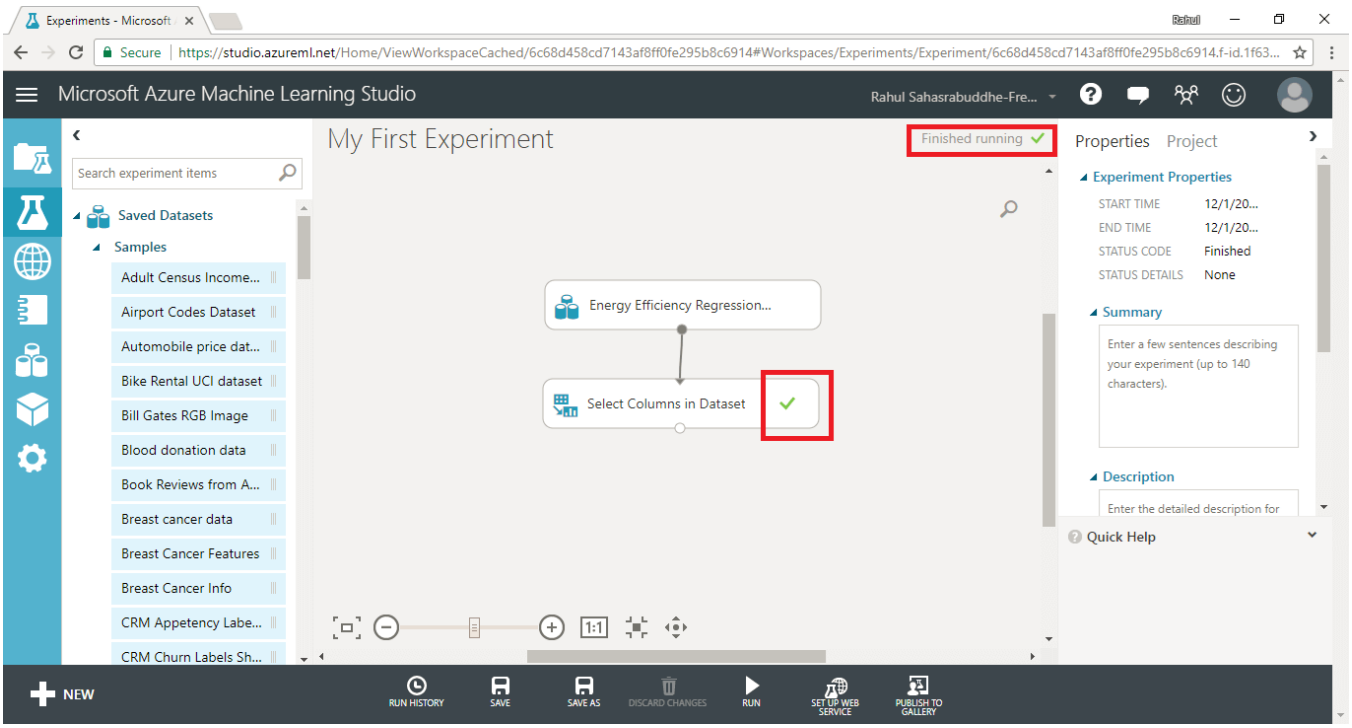


Figure 18 : Running the Experiment

You can double-click on the Select Columns in the Dataset box and add a note as shown in Figure 19.

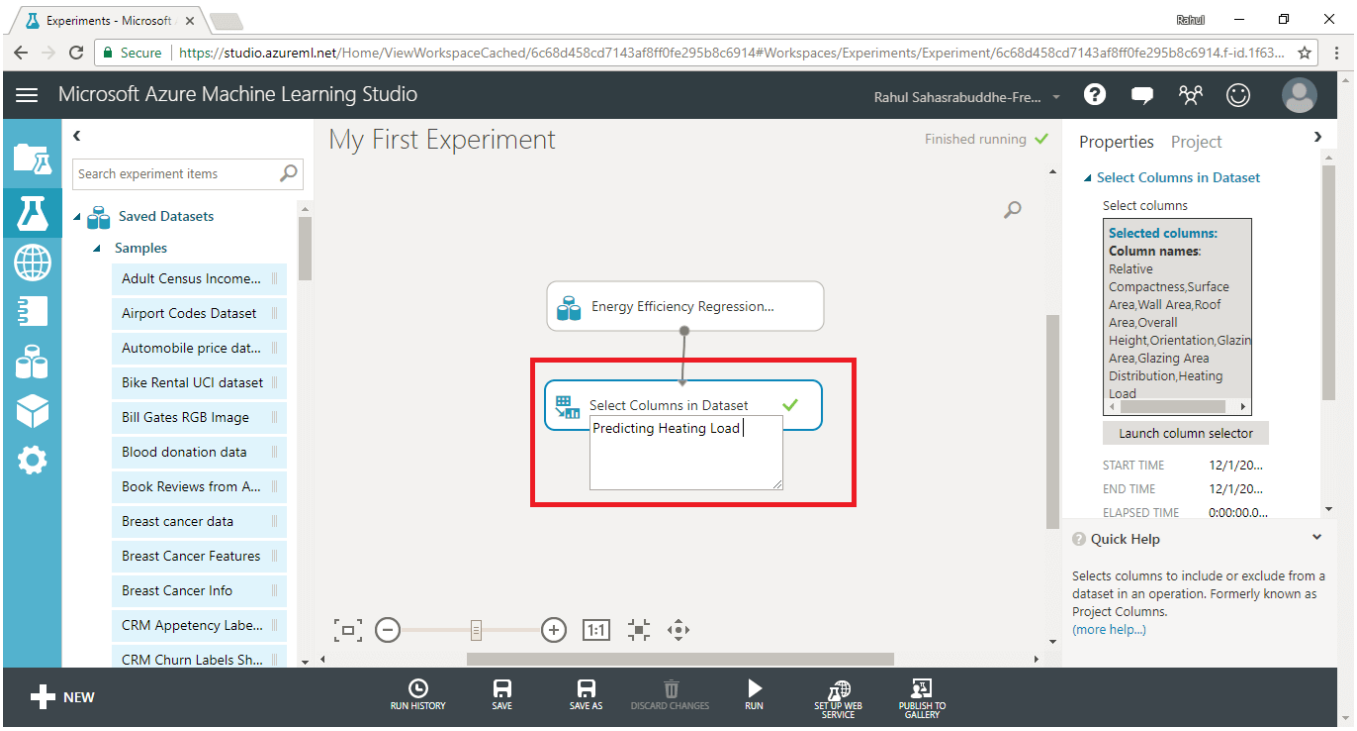


Figure 19 : Adding Notes

After this, the workspace will appear as shown in Figure 20:

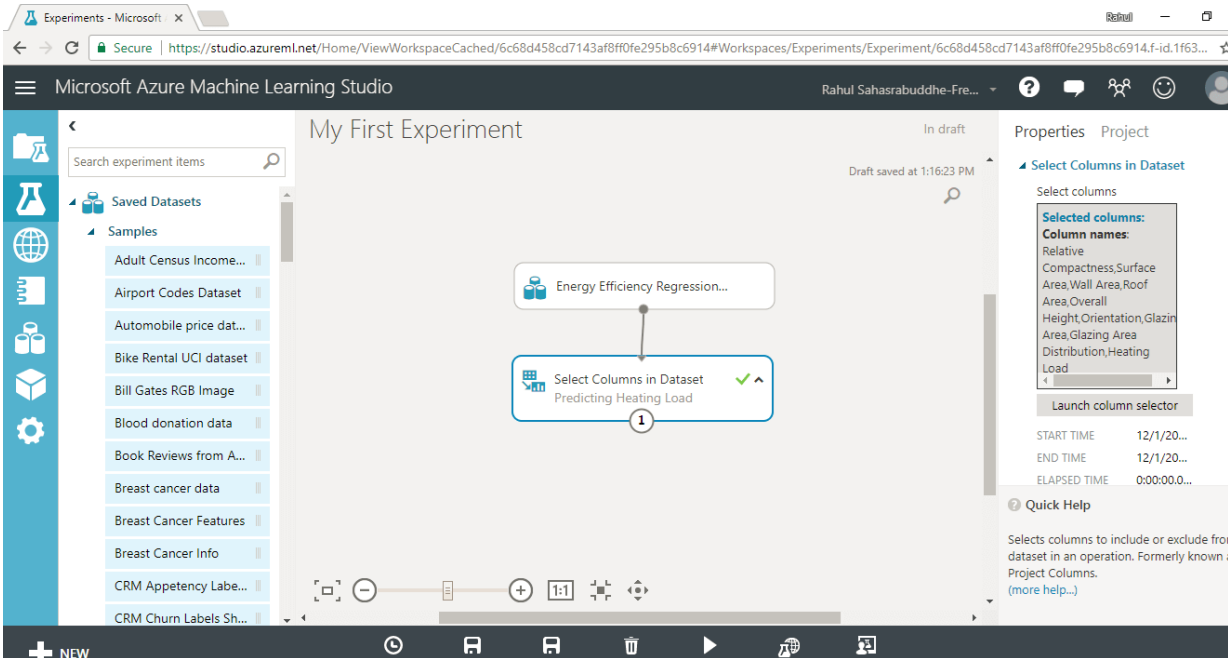


Figure 20: Note is added

Step 9: Now we need to choose a learning algorithm.

In this case, since we want to predict the outcome based on the available parameters or features, we will need to choose a supervised learning algorithm (since we are going to use data to train the model and predict outcome).

Typically, there are two types of supervised learning algorithms viz. **classification** – used for predicting an answer from a defined set of categories or **regression** – used for predicting a specific outcome in the form of a number based on available data.

We will use the regression algorithm here. Enter "linear regression" and then drag the linear regression module onto the workspace as shown in Figure 21.

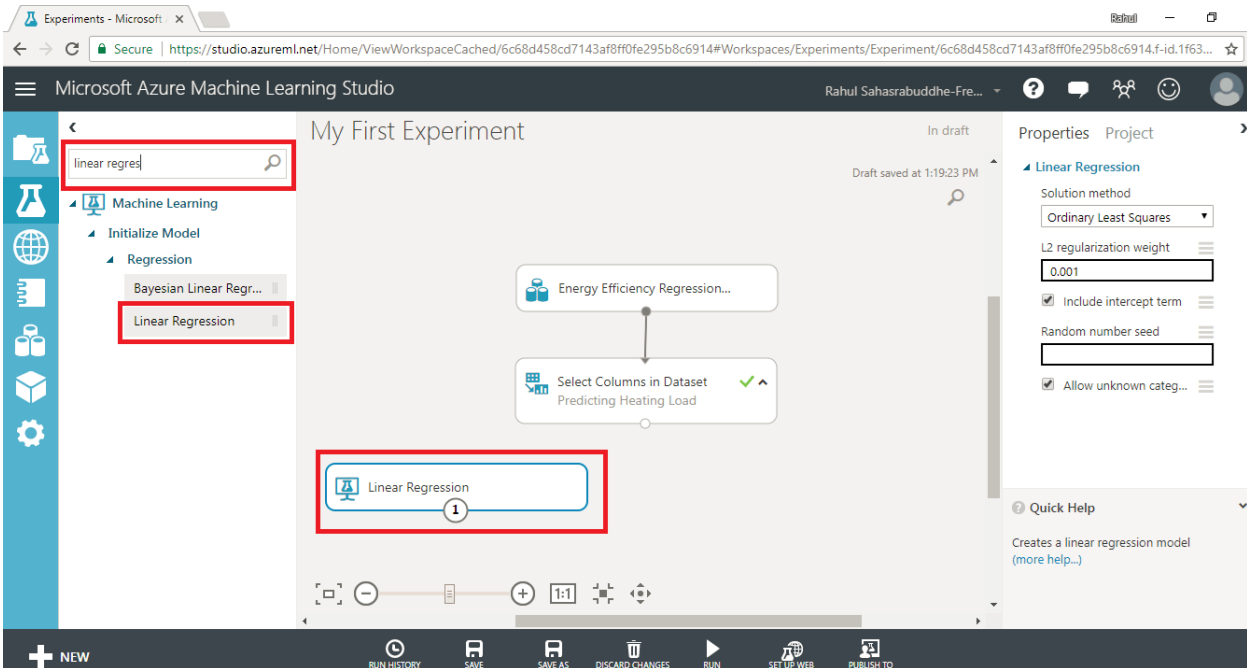


Figure 21 : Choosing the Learning Algorithm

Step 10: Usually for any ML experiment, you would need data to train the model, and then data to test the model upon.

In this case, we can use the same data set. To do so, we need to split data. Let's use 80% of our data to train the model and 20% of it to test the model. We can do so by dragging the "Split Data" module on the workspace as follows:

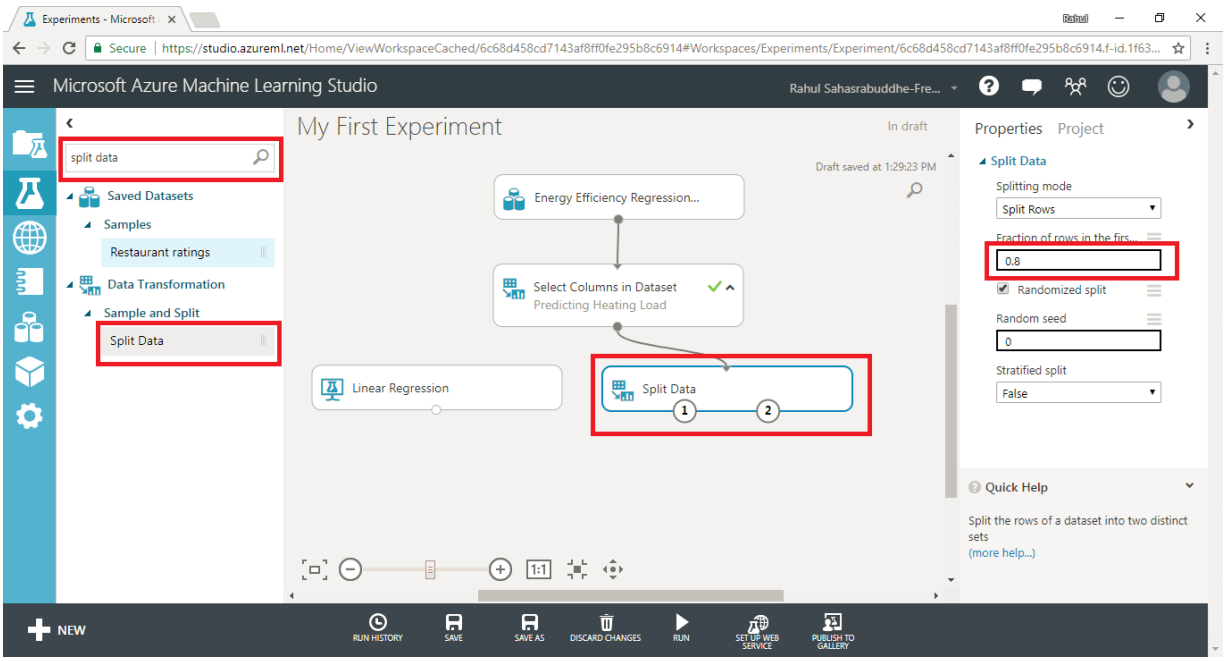


Figure 22 : Splitting Data

Step 11: Now add the "Train model" on to the workspace and connect the 'Linear Regression' module and 'Split Data' module to it. This means that we are going to use the linear regression algorithm and the data, to train the model. The outcome would be a trained model with this algorithm and data set.

This is shown in Figure 23.

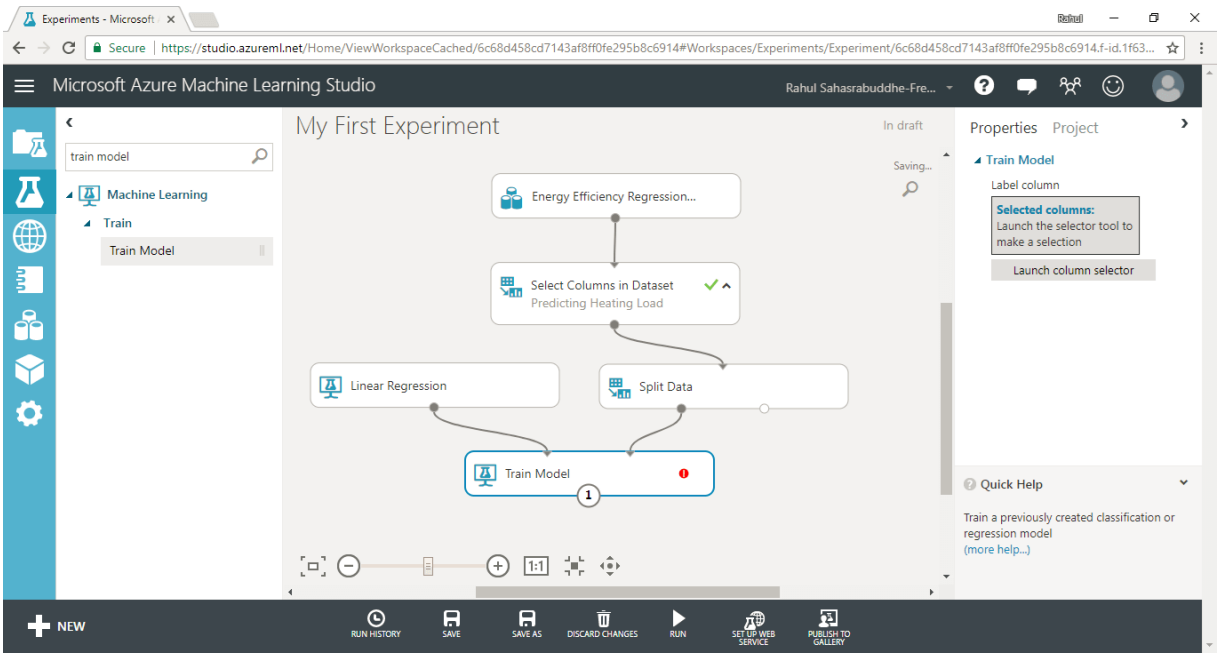


Figure 23 : Training the Model

Notice the red dot again.

This means that we need to let the model know which column it needs to predict the parameter for, using the algorithm we chose. Open up “Launch column selector” and then select “Heating Load” since this is what we are planning to predict.

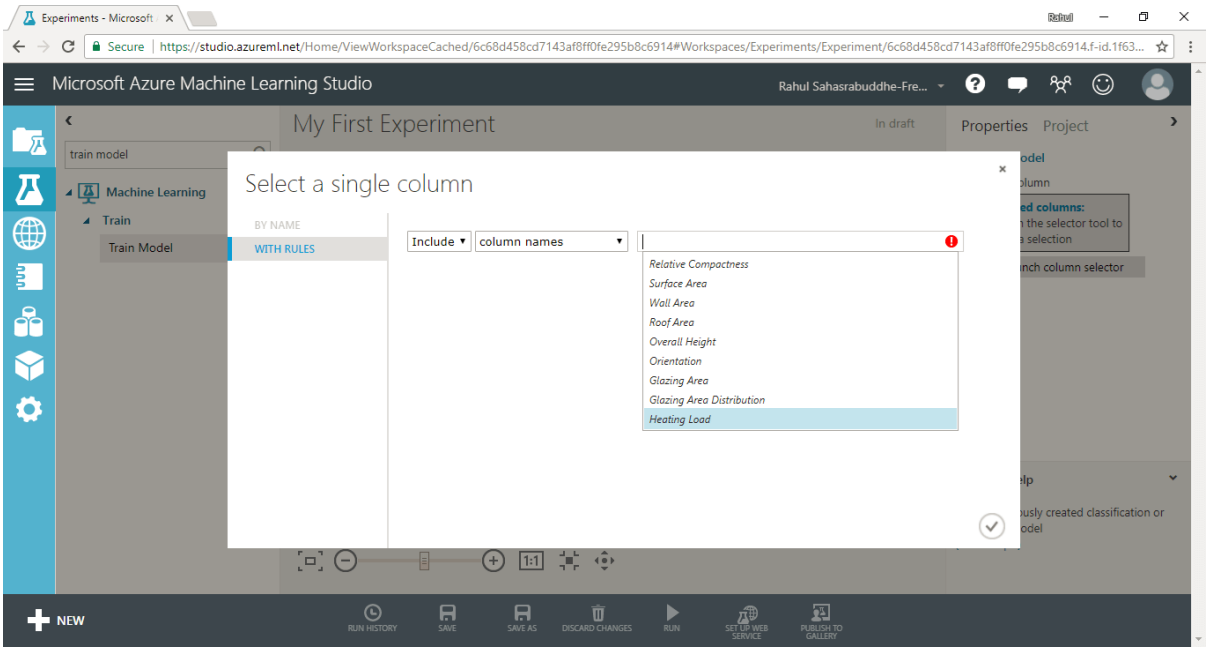


Figure 24 : Selecting Parameter to predict

Step 12: After this, let’s run the model again to ensure that everything is working well. Once this is done, you should see the following result:

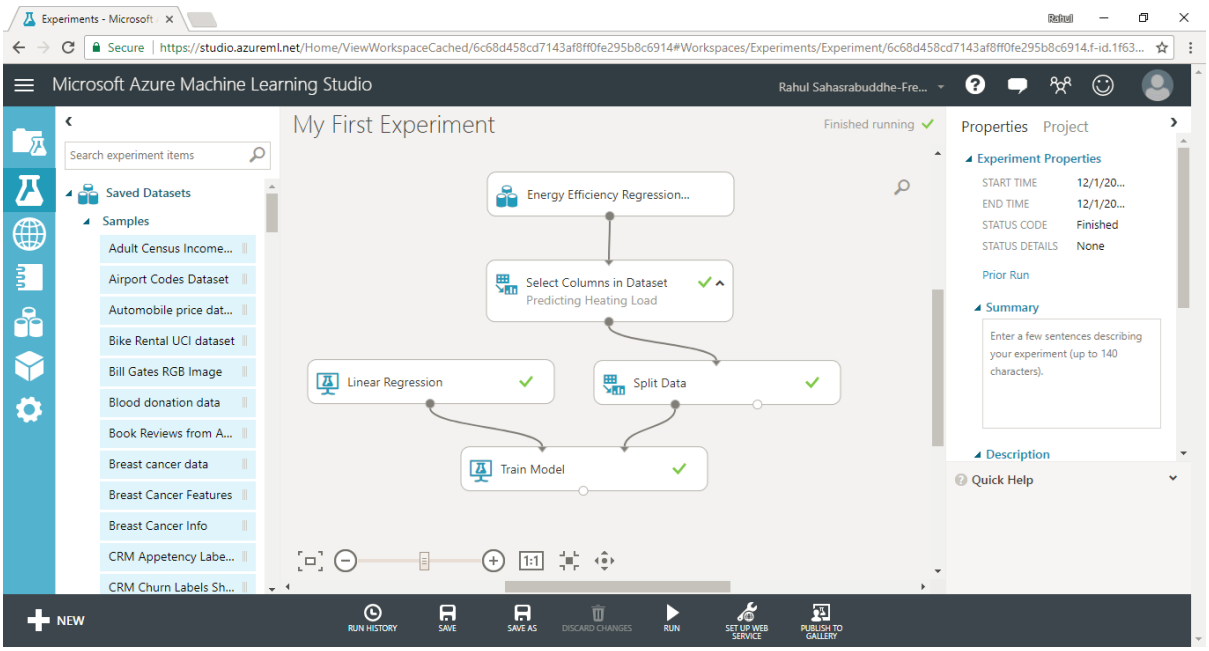


Figure 25: Running the Model

Figure 26 shows all the steps so far:

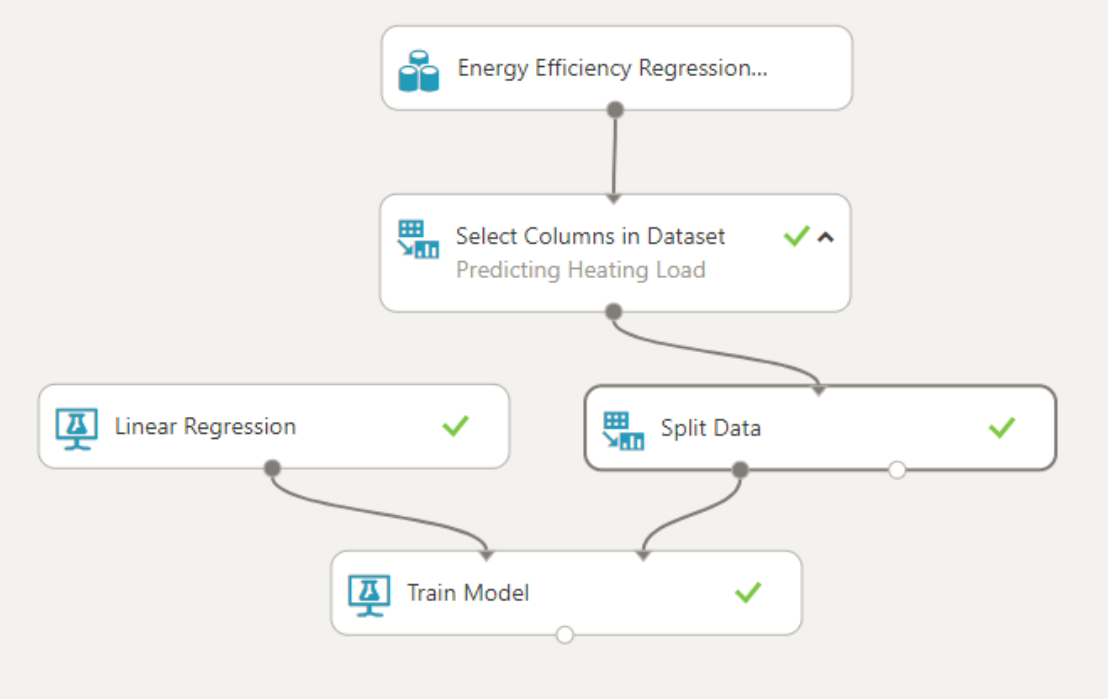


Figure 26: Experiment Steps executed so far

Step 13: Now comes the real interesting bit to see how this has all shaped up.

We will predict the Heating Load based on our data and algorithm. For this, we need to use "Score Model" module. We will use following inputs for this:

a) trained model from 'Train Model' module
b) sample data (remember, the rest 20% data we kept aside).

Once we do that, the workspace would look like the following:

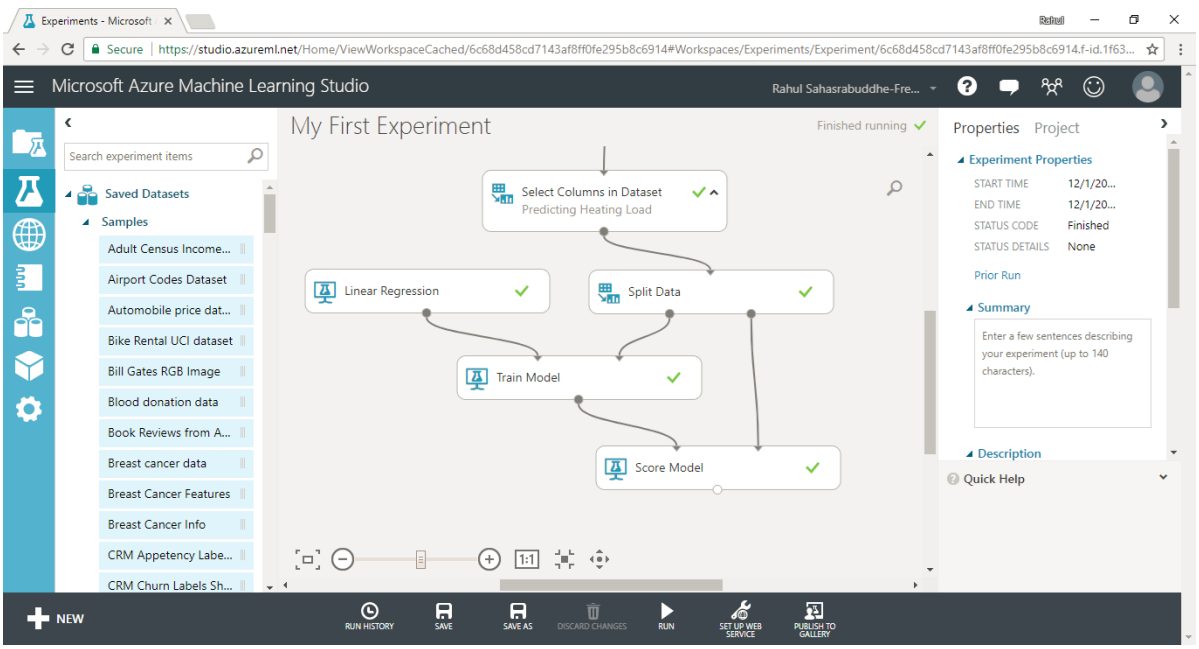


Figure 27 : Running the Experiment again

Step 14: Click on "Visualize" and it will bring up the results. See Figure 28.

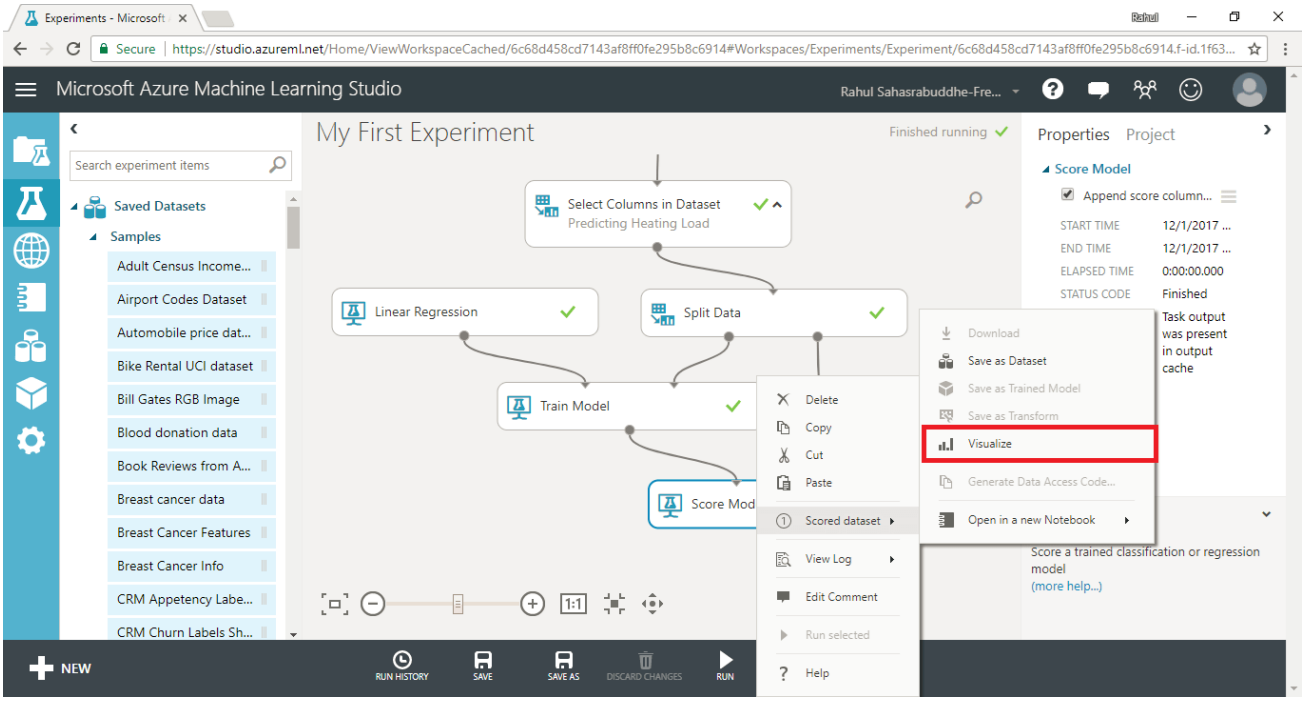


Figure 28 : Visualizing the output

Here is the output:

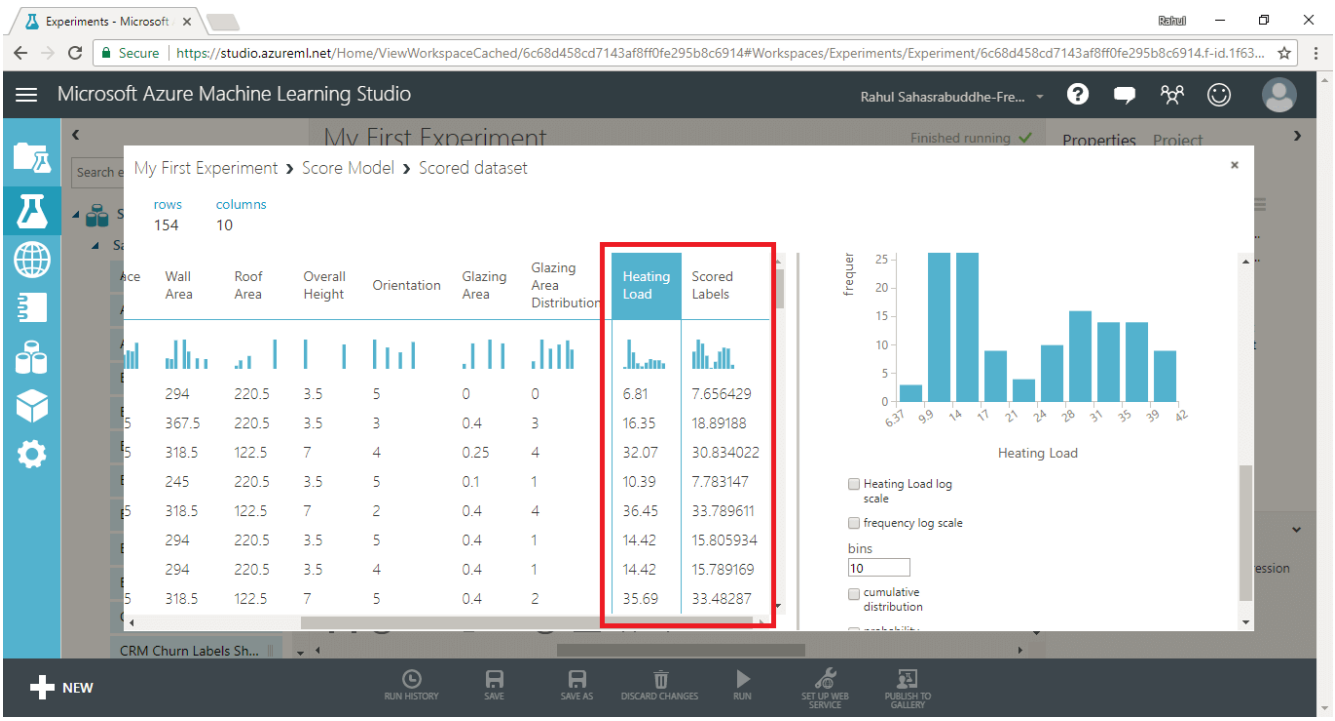


Figure 29 : Final Output

In the screenshot shown in Figure 29, the "Scored Labels" column has the predicted Heating Load.

The following screenshot shows a comparison of Heating Load value in the dataset vs. the same being predicted using the ML algorithm we used.

You can also evaluate the correctness of the results by using "Evaluate model" module. I will leave this step as a short exercise for you to perform.

Heating Load	Scored Labels
6.81	7.656429
16.35	18.89188
32.07	30.834022
10.39	7.783147
36.45	33.789611
14.42	15.805934
14.42	15.789169
35.69	33.48287

Figure 30 : Comparing the output

This short step-by-step tut should have given you a glimpse of what all you can do with Azure ML Studio. Now you can play around with various modules/steps in the ML process and check out the results yourself in terms of how various changes results in corresponding changes in the final outcome. You can also try different data sets, as well as different ML techniques.

Conclusion

We are teaching machines to learn so that they can start taking certain decisions for us.

In the near future, we'll have a self-driving car, a more advanced email client that will respond to our mails and what not. Does it sound scary or exciting?

Well, it depends on how you perceive it. One thing is certain for sure, as developers, we need to continue to learn new things to outsmart machines.

Be it humans or machines, learning never stops!

Rahul Sahasrabuddhe

Author

Rahul Sahasrabuddhe has been working on Microsoft Technologies since last 17 years and leads Microsoft Technology Practice at a leading software company in Pune. He has been instrumental in setting up competencies around Azure, SharePoint and various other Microsoft Technologies. Being an avid reader, he likes to keep himself abreast with cutting edge technology changes & advances.

Thanks to Mahesh Sabnis and Suprotim Agarwal for reviewing this article.

Damir Arh

HOW WELL DO YOU KNOW C#

Although C# is considered to be a language that's easy to learn and understand, the code written in it might behave unexpectedly sometimes, even for developers with years of experience and good knowledge of the language. This article features several code snippets which fall into that category, and explains the reasons behind the surprising behavior.

Null Value

We are all aware that `null` values can be dangerous, if not handled properly.

Dereferencing a `null`-valued variable (i.e. calling a method on it or accessing one of its properties) will result in a `NullReferenceException`, as demonstrated with the following sample code:

```
object nullValue = null;
bool areNullValuesEqual = nullValue.Equals(null);
```

To be on the safer side, we should always make sure that reference type values are not null before dereferencing them. Failing to do so could result in an unhandled exception in a specific edge case. Although such a mistake occasionally happens to everyone, we could hardly call it unexpected behavior.

What about the following code?

```
string nullString = (string)null;
bool isStringType = nullString is string;
```

What will be the value of `isStringType`? Will the value of a variable that is explicitly typed as `string` always be typed as `string` at runtime as well?

The correct answer is **No**.

A null value has no type at runtime.

In a way, this also affects reflection. Of course, you can't call `GetType()` on a `null` value because a `NullReferenceException` would get thrown:

```
object nullValue = null;
Type nullType = nullValue.GetType();
```

Let's look at nullable value types then:

```
int intValue = 5;
Nullable<int> nullableIntValue = 5;
bool areTypesEqual = intValue.GetType() == nullableIntValue.GetType();
```

Is it possible to distinguish between a nullable and a non-nullable value type using reflection?

The answer is **No**.

The same type will be returned for both variables in the above code: `System.Int32`. This does not mean that reflection has no representation for `Nullable<T>`, though.

```
Type intType = typeof(int);
Type nullableIntType = typeof(Nullable<int>);
bool areTypesEqual = intType == nullableIntType;
```

The types in this code snippet are different. As expected, the nullable type will be represented with `System.Nullable`1[[System.Int32]]`. Only when inspecting values, nullable ones are treated the

same as non-nullable ones in reflection.

Handling Null values in Overloaded methods

Before moving on to other topics, let's take a closer look at how null values are handled when calling overloaded methods with the same number of parameters, but of different type.

```
private string OverloadedMethod(object arg)
{
  return "object parameter";
}

private string OverloadedMethod(string arg)
{
  return "string parameter ";
}
```

What will happen if we invoke the method with `null` value?

```
var result = OverloadedMethod(null);
```

Which overload will be called? Or will the code fail to compile because of an ambiguous method call?

In this case, the code ***will compile*** and the method with the `string` parameter *will be called*.

In general, the code will compile when one parameter type can be cast to the other one (i.e. one is derived from the other). The method with the more specific parameter type will be called.

When there's no cast between the two types, the code won't compile.

To force a specific overload to be called, the `null` value can be cast to that parameter type:

```
var result = parameteredMethod((object)null);
```

Arithmetic Operations

Most of us don't use bit shifting operations very often!

Let's refresh our memory first. Left shift operator (<<) shifts the binary representation to the left for the given number of places:

```
var shifted = 0b1 << 1; // = 0b10
```

Similarly, the right shift operator (>>) shifts the binary representation to the right:

```
var shifted = 0b1 >> 1; // = 0b0
```

The bits don't wrap around when they reach the end. That's why the result of the second expression is 0. The same would happen if we shifted the bit far enough to the left (32 bits because integer is a 32-bit number):

```
var shifted = 0b1;
```

```
for (int i = 0; i < 32; i++)
{
  shifted = shifted << 1;
}
```

The result would again be 0.

However, the bit shifting operators have a second operand. Instead of shifting to the left by 1 bit 32 times, we can shift left by 32 bits and get the same result.

```
var shifted = 0b1 << 32;
```

Right? **Wrong**.

The result of this expression will be 1. Why?

Because that's how the operator is defined. Before applying the operation, the second operand will be normalized to the bit length of the first operand with the modulo operation, i.e. by calculating the remainder of dividing the second operand by the bit length of the first operand.

The first operand in the example we just saw was a 32-bit number, hence: 32 % 32 = 0. Our number will be shifted left by 0 bits. That's not the same as shifting it left by 1 bit 32 times.

Let's move on to operators `&` (and) and `|` (or). Based on the type of operands, they represent two different operations:

- For `Boolean` operands, they act as logical operators, similar to `&&` and `||`, with one difference: they are eager, i.e. both operands are always evaluated, even if the result could already be determined after evaluating the first operand.

- For integral types, they act as logical bitwise operators and are commonly used with `enum` types representing flags.

```
[Flags]
private enum Colors
{
  None = 0b0,
  Red = 0b1,
  Green = 0b10,
  Blue = 0b100
}
```

The `|` operator is used for combining flags and the `&` operator is used for checking whether flags are set:

```
Colors color = Colors.Red | Colors.Green;
bool isRed = (color & Colors.Red) == Colors.Red;
```

In the above code, I put parenthesis around the bitwise logical operation to make the code more clear. Are parenthesis required in this expression?

As it turns out, **Yes**.

Unlike arithmetic operators, the bitwise logical operators have lower priority than the equality operator.

Fortunately, code without the parenthesis wouldn't compile because of type checking.

Since .NET framework 4.0, there's a better alternative available for checking flags, which you should always use instead of the & operator:

```
bool isRed = color.HasFlag(Colors.Red);
```

Math.Round()

We will conclude the topic of arithmetic operations with the Round operation. How does it round the values at the midpoint between the two integer values, e.g. 1.5? Up or down?

```
var rounded = Math.Round(1.5);
```

If you predicted **up**, you were right. The result will be 2. Is this a general rule?

```
var rounded = Math.Round(2.5);
```

No. The result will be 2 again. By default, the midpoint value will be rounded to the nearest even value. You could provide the second argument to the method to request such behavior explicitly:

```
var rounded = Math.Round(2.5, MidpointRounding.ToEven);
```

The behavior can be changed with a different value for the second argument:

```
var rounded = Math.Round(2.5, MidpointRounding.AwayFromZero);
```

With this explicit rule, positive values will now always be rounded upwards.

Rounding numbers can also be affected by the precision of floating point numbers.

```
var value = 1.4f;
var rounded = Math.Round(value + 0.1f);
```

Although, the midpoint value should be rounded to the nearest even number, i.e. 2, the result will be 1 in this case, because with single precision floating point numbers there is no exact representation for 0.1 and the calculated number will actually be less than 1.5 and hence rounded to one.

Although this particular issue does not manifest itself when using double precision floating point numbers, rounding errors can still happen, albeit less often. When requiring maximum precision, you should therefore always use `decimal` instead of `float` or `double`.

Class Initialization

Best practices suggest that we should avoid class initialization in class constructors as far as possible to prevent exceptions.

All of this is even more important for static constructors.

As you might know, the static constructor is called before the instance constructor when we try to instantiate it at runtime.

This is the order of initialization when instantiating any class:

- Static fields (first time class access only: static members or first instance)
- Static constructor (first time class access only: static members or first instance)
- Instance fields (each instance)
- Instance constructor (each instance)

Let's create a class with a static constructor, which can be configured to throw an exception:

```
public static class Config
{
  public static bool ThrowException { get; set; } = true;
}

public class FailingClass
{
  static FailingClass()
  {
    if (Config.ThrowException)
    {
      throw new InvalidOperationException();
    }
  }
}
```

It shouldn't come as a surprise that any attempt to create an instance of this class will result in an exception:

```
var instance = new FailingClass();
```

However, it won't be `InvalidOperationException`. The runtime will automatically wrap it into a `TypeInitializationException`. This is an important detail to note if you want to catch the exception and recover from it.

```
try
{
  var failedInstance = new FailingClass();
}
catch (TypeInitializationException) { }
Config.ThrowException = false;
var instance = new FailingClass();
```

Applying what we have learned, the above code should catch the exception thrown by the static constructor, change the configuration to avoid exceptions being thrown in future calls, and finally successfully create an instance of the class, right?

Unfortunately, not.

The static constructor for a class is only called once. If it throws an exception, then this exception will be rethrown whenever you want to create an instance or access the class in any other way.

The class becomes effectively unusable until the process (or the application domain) is restarted. Yes, having even a minuscule chance that the static constructor will throw an exception, is a very bad idea.

Initialization Order in Derived classes

Initialization order is even more complex for derived classes. In edge cases, this can bring you into trouble. It's time for a contrived example:

```
public class BaseClass
{
  public BaseClass()
  {
    VirtualMethod(1);
  }

  public virtual int VirtualMethod(int dividend)
  {
    return dividend / 1;
  }
}

public class DerivedClass : BaseClass
{
  int divisor;
  public DerivedClass()
  {
    divisor = 1;
  }

  public override int VirtualMethod(int dividend)
  {
    return base.VirtualMethod(dividend / divisor);
  }
}
```

Can you spot a problem in `DerivedClass`? What will happen when I try to instantiate it?

```
var instance = new DerivedClass();
```

A `DivideByZeroException` will be thrown. Why?

Well, the reason is in the order of initialization for derived classes:

- First, instance fields are initialized in the order from the most derived to the base class.
- Then, constructors are called in the order from the base class to the most derived class.

Since the class is treated as `DerivedClass` throughout the initialization process, our call to `VirtualMethod` in `BaseClass` constructor invokes the `DerivedClass` implementation of the method before the `DerivedClass` constructor had a chance to initialize the `divisor` field. This means that the value was still 0, which caused the `DivideByZeroException`.

In our case, the problem could be fixed by initializing the divisor field directly instead of in the constructor.

However, the example showcases why it can be dangerous to invoke virtual methods from a constructor. When they are invoked, the constructor of the class they are defined in, might not have been called yet, therefore they could behave unexpectedly.

Polymorphism

Polymorphism is the ability for different classes to implement the same interface, in a different way.

Still, we usually expect a single instance to always use the same implementation of a method, no matter to which type it is cast. This makes it possible to have a collection typed as a base class and invoke a particular method on all instances in the collection, resulting in the specific implementation for each type to be called.

Having said that, can you think of a way to have a different method be called when we downcast the instance before calling the method i.e. to break polymorphic behavior?

```
var instance = new DerivedClass();
var result = instance.Method(); // -> Method in DerivedClass
result = ((BaseClass)instance).Method(); // -> Method in BaseClass
```

The correct answer is: by using the `new` modifier.

```
public class BaseClass {
  public virtual string Method()
  {
    return “Method in BaseClass “;
   }
}

public class DerivedClass : BaseClass {
  public new string Method()
  {
    return “Method in DerivedClass”;
  }
}
```

This hides the `DerivedClass.Method` from its base class, therefore `BaseClass.Method` is called when the instance is cast to the base class.

This works for base classes, which can have their own method implementations. Can you think of a way to achieve the same for an interface, which cannot contain its own method implementation?

```
var instance = new DerivedClass();
var result = instance.Method(); // -> Method in DerivedClass
result = ((IInterface)instance).Method(); // -> Method belonging to IInterface
```

It’s **explicit interface implementation**.

```
public interface IInterface
{
  string Method();
}
```

```
public class DerivedClass : IInterface
{
  public string Method()
  {
    return “Method in DerivedClass”;
  }

  string IInterface.Method()
  {
    return “Method belonging to IInterface”;
  }
}
```

It’s typically used to hide the interface methods from the consumers of the class implementing it, unless they cast the instance to that interface. But it works just as well if we want to have two different implementations of a method inside a single class. It’s difficult to think of a good reason for doing it, though.

Iterators

Iterators are the construct used for stepping through a collection of items, typically using a `foreach` statement. They are represented by the `IEnumerable<T>` generic type.

Although they are very easy to use, thanks to some compiler magic, we can quickly fall into a trap of incorrect usage if we don’t understand the inner workings well enough.

Let’s look at such an example. We will call a method, which returns an `IEnumerable` from inside a `using` block:

```
private IEnumerable<int> GetEnumerable(StringBuilder log)
{
  using (var context = new Context(log))
  {
    return Enumerable.Range(1, 5);
  }
}
```

The `Context` class of course implements `IDisposable`. It writes a message to the log to indicate when its scope is entered and exited. In real world code, this context could be replaced by a database connection. Inside it, rows would be read from the returned result set in a streaming manner.

```
public class Context : IDisposable {
  private readonly StringBuilder log;

  public Context(StringBuilder log) {
    this.log = log;
    this.log.AppendLine(“Context created”);
  }

  public void Dispose() {
    this.log.AppendLine(“Context disposed”);
  }
}
```

To consume the `GetEnumerable` return value, we iterate through it with a `foreach` loop:

```
var log = new StringBuilder();
foreach (var number in GetEnumerable(log))
{
  log.AppendLine($"{number}");
}
```

What will be the contents of `log` after the code executes? Will the returned values be listed between the context creation and disposal?

No, they won't:

```
Context created
Context disposed
1
2
3
4
5
```

This means that in our real world database example, the code would fail – the connection would be closed before the values could be read from the database.

How can we fix the code so that the context would only be disposed after all values have already been iterated through?

The only way to do it is to iterate through the collection already inside the `GetEnumerable` method:

```
private IEnumerable<int> GetEnumerable(StringBuilder log)
{
  using (var context = new Context(log))
  {
    foreach (var i in Enumerable.Range(1, 5))
    {
      yield return i;
    }
  }
}
```

When we now iterate through the returned `IEnumerable`, the context will only be disposed at the end as expected:

```
Context created
1
2
3
4
5
Context disposed
```

In case you're not familiar with the `yield return` statement, it is syntactic sugar for creating a state

machine, allowing the code in the method using it to be executed incrementally, as the resulting `IEnumerable` is being iterated through.

This can be explained better with the following method:

```
private IEnumerable<int> GetCustomEnumerable(StringBuilder log)
{
  log.AppendLine(“before 1”);
  yield return 1;
  log.AppendLine(“before 2”);
  yield return 2;
  log.AppendLine(“before 3”);
  yield return 3;
  log.AppendLine(“before 4”);
  yield return 4;
  log.AppendLine(“before 5”);
  yield return 5;
  log.AppendLine(“before end”);
}
```

To see how this piece of code behaves, we can use the following code to iterate through it:

```
var log = new StringBuilder();
log.AppendLine(“before enumeration”);
foreach (var number in GetCustomEnumerable(log))
{
  log.AppendLine($”{number}”);
}
log.AppendLine(“after enumeration”);
```

Let’s look at the log contents after the code executes:

```
before enumeration
before 1
1
before 2
2
before 3
3
before 4
4
before 5
5
before end
after enumeration
```

We can see that for each value we iterate through, the code between the two `yield return` statements gets executed.

For the first value, this is the code from the beginning of the method to the first `yield return` statement. For the second value, it’s the code between the first and the second `yield return` statements. And so on, until the end of the method.

The code after the last `yield return` statement is called when the `foreach` loop checks for the next

value in the IEnumerable after the last iteration of the loop.

It's also worth noting that this code will get executed every time we iterate through IEnumerable:

```
var log = new StringBuilder();
var enumerable = GetCustomEnumerable(log);
for (int i = 1; i <= 2; i++)
{
  log.AppendLine($"enumeration #{i}");
  foreach (var number in enumerable)
  {
    log.AppendLine($"{number}");
  }
}
```

After executing this code, the log will have the following contents:

```
enumeration #1
before 1
1
before 2
2
before 3
3
before 4
4
before 5
5
before end
enumeration #2
before 1
1
before 2
2
before 3
3
before 4
4
before 5
5
before end
```

To prevent the code from getting executed every time we iterate through IEnumerable, it's a good practice to store the results of an IEnumerable into a local collection (e.g. a List) and read it from there if we are planning to use it multiple times:

```
var log = new StringBuilder();
var enumerable = GetCustomEnumerable(log).ToList();
for (int i = 1; i <= 2; i++)
{
  log.AppendLine($"enumeration #{i}");
  foreach (var number in enumerable)
  {
    log.AppendLine($"{number}");
  }
}
```

Now, **the code will be executed only once** – at the point when we create the list, before iterating through it:

```
before 1
before 2
before 3
before 4
before 5
before end
enumeration #1
1
2
3
4
5
enumeration #2
1
2
3
4
5
```

This is particularly important when there are slow I/O operations behind the `IEnumerable` we are iterating through. Database access is again a typical example of that.

Conclusion:

Did you correctly predict the behavior of all the samples in the article?

If not, you might have learned that it can be dangerous to assume behavior when you are not completely certain how a particular feature is implemented. It's impossible to know and remember every single edge case in a language, therefore it's a good idea to check the documentation or try it out yourself when you are unsure about an important piece of code which you have encountered.

More important that any of this is to avoid writing code which might surprise other developers (or may be even you, after a certain amount of time passes). Try to write it differently or pass the default value for that optional parameter (as in our `Math.Round` example) to make the intention clearer.

If that's not possible, write the tests in such way that they will clearly document the expected behavior!

Microsoft® MVP Most Valuable Professional

Damir Arh

Author

Damir Arh has many years of experience with Microsoft development tools; both in complex enterprise software projects and modern cross-platform mobile applications. In his drive towards better development processes, he is a proponent of test driven development, continuous integration and continuous deployment. He shares his knowledge by speaking at local user groups and conferences, blogging, and answering questions on Stack Overflow. He is an awarded Microsoft MVP for .NET since 2012.

Thanks to Yacoub Massad for reviewing this article.

A MAGAZINE FOR .NET AND JAVASCRIPT DEVS

- ☑ AGILE
- ☑ ASP.NET
- ☑ MVC, WEB API
- ☑ ANGULAR.JS
- ☑ NODE.JS
- ☑ AZURE
- ☑ VISUAL STUDIO
- ☑ .NET
- ☑ C#, WPF

We've got it all!

100K PLUS READERS

300 PLUS AWESOME ARTICLES

34 EDITIONS

FREE SUBSCRIPTION USING YOUR EMAIL

EVERY ISSUE DELIVERED

RIGHT TO YOUR INBOX

NO SPAM POLICY

SUBSCRIBE TODAY!

Ravi Kiran

Deploying Angular apps to Azure using docker

A tutorial demonstrating how to create a Docker image for an Angular application that runs on both browser and from the server, and then deploying it to Azure.

A web application has to be deployed to a web server to make it accessible to users.

The process of deployment is different for every application. Every application uses different technologies of different versions, and with a distinct set of libraries to achieve their functionality. They also have different types of configuration at runtime.

The deployment server should have the right set of software installed to support the application. Even though cloud providers like Azure support almost every platform in the market, one might still encounter a situation that makes him or her think "That works fine on my computer, what's the problem with this server?".

The ***container*** approach solves this problem by providing an isolated and more predictable platform. This approach got a lot of popularity due to Docker. Today many developers and firms are using Docker.

This article will introduce you to Docker, and demonstrate how to deploy an Angular application in Docker and then deploy it to Azure.

Environment

Here are some pre-requisites for this article:

- Node.js – Download and install it from the official site
- Angular CLI – It has to be installed as a global npm package:

  ```
  > npm install -g @angular/cli
  ```

- Docker – Download and install docker from the official site or install Docker toolbox if your platform doesn't support Docker

This article will not show how to build an Angular application from scratch. It uses the Pokémon Explorer sample built in the article Server Side Rendering using Angular Universal and Node.js to create a Pokémon Explorer App (www.dotnetcurry.com/angularjs/1388/server-side-rendering-angular-nodejs) to deploy on Azure.

To follow along this article, you may download the sample and run it once on your system.

Getting Familiar with Docker

Docker is an open source platform that provides an environment to develop, test and deploy applications. Using docker, one can develop an application much faster and deploy it in an environment that is similar to the environment where it is developed. Because of the reduced gap between the development and the runtime environments, the application is more predictable.

Images and containers are the most important objects to deal with while working on Docker.

An image contains a set of instructions to setup the container and run it. In general, an image extends an existing image and applies its own customizations. For example, an image to run a C program will be extended from an image that comes with an OS, and will install the environment required to run the C

program.

A container is the running instance of an image.

The container is isolated from the other containers and the host system (The system on which Docker would be running). The container is created based on the Kernel of the OS on the host system. So, the container is extremely light in weight and it doesn't contain any software installed on the host.

On top of the kernel, it installs the software configured in the docker image and either uses the storage from the host system itself, or stores the files inside its own context based on the instructions in the image. The container stays in running mode as long as the program installed in the container runs or till someone stops the container.

Because of being light weight, it is easy to spin up or bring down a container. The process of creating or deleting a container is much faster when it is compared to VMs.

Since VMs consume a lot of memory on the host system, they require stand-alone operating systems which involve license cost and it takes time to bring them to an up and running state. On the other hand, a container doesn't install a new OS, it gets ready within a few minutes using the host's kernel and the only thing to be done is to install the software required for the application.

Docker consists of three components. Host, Client and Registry. The following diagram shows the way these components work together:

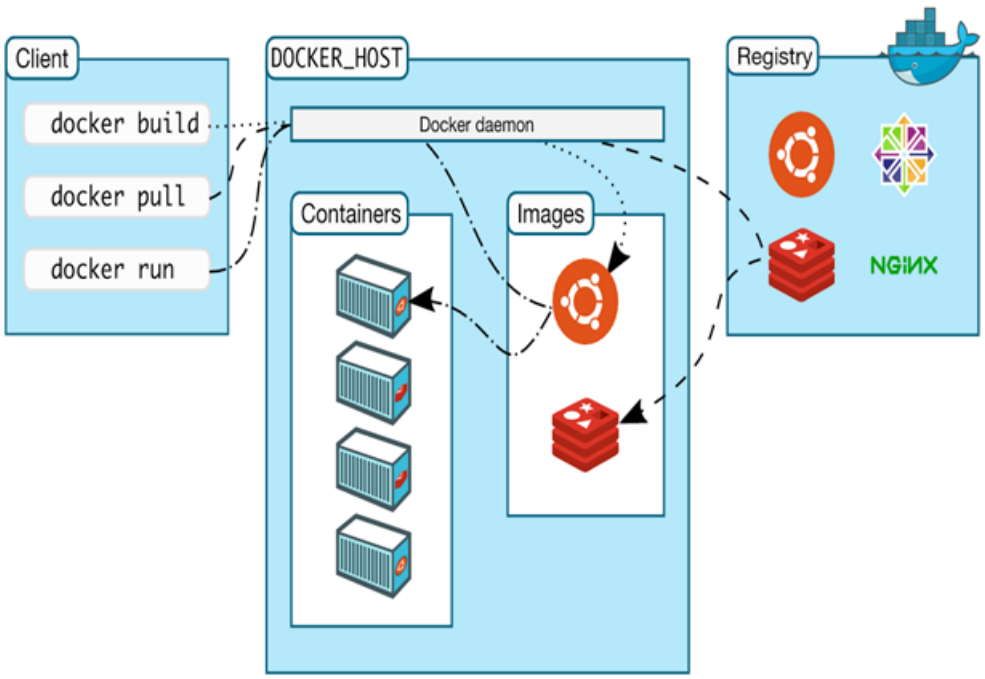


Figure 1 – Docker architecture (source: https://docs.docker.com/engine/docker-overview)

- **Client** is the command line interface used to interact with Docker. It is used to manage images and containers on the host. Docker provides commands to interact with the registry and these commands use the Docker API.

- **Host** is the system on which Docker runs. The images and containers are created on the host machine. The host contains Docker daemon, that listens to the Docker API requests and manages the containers

and images. As the diagram shows, the daemon sits at the center of client, containers, images and the registry. All the requests pass through the daemon and the results of these requests are stored on the host system.

- **Registry** is the hub of docker images. Docker Hub is the default public registry. Anyone can use the public images available in this registry. Docker can be configured to use any other registry like Docker Cloud, Azure Container Registry or your company's own registry.

Getting your hands dirty with Docker

If you have installed Docker, it should automatically start running in the background on your system. Check it on your system tray and if it isn't running yet, then start it from the start menu.

Open a command prompt and run the following command to check if docker is running:

```
> docker -v
```

If Docker is in a running state, it will display the version of the Docker installed. The best way to check if the installation works correctly is by running a container. Docker provides a Hello World image that can be used to test if Docker is running correctly. To get this image and run it, run the following command

```
> docker run hello-world
```

The output produced by this command is similar to Figure 2:

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
ca4f61b1923c: Pull complete
Digest: sha256:be0cd392e45be79ffeffa6b05338b98ebb16c87b255f48e297ec7f98e123905c
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://cloud.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/engine/userguide/
```

Figure 2 – Docker hello world

The list of operations performed by Docker are quite visible in the log printed on the console. In Figure 2, Docker tried finding the image *hello-world* on a local machine. Since the image was not present on the local system, it pulled the image from the registry. Then it executed the commands configured in the image. The commands printed the Hello World message and the list of things that Docker checked on the system. To run it again, run the same command and now it will not pull the image from registry as it is already available on the system.

To see the list of images on the system, run the following command:

```
> docker images
```

Once you run an image, it creates a container. The list of containers currently running on the system can be viewed using the following command:

```
> docker ps
```

This command may not produce any output if there are no active containers. The list of all the containers on the system can be seen using the following command:

```
> docker ps -a
```

The *hello-world* container stops as soon as it prints the messages on the console. Let's look at an image that keeps running unless we stop. Pull the *nginx* image.

```
> docker pull nginx
```

The *docker run* command needs a few parameters to run this container. The following command shows these parameters:

```
> docker run -p 8080:80 -d nginx
```

The following listing describes the options used in the above command:

- The parameter `-p` is used to pass port number. The *nginx* container runs a web application on the port 80 inside the container, it has to be mapped to a local port number. 8080 is the port number on the host through which the application running in the container is made available

- The `-d` option runs the container in detached mode. This mode makes sure the container exits when the process used to run the container exits.

Once this command runs, you can open a browser and change the URL to http://localhost:8080. You will see an output similar to the following screenshot:

Figure 3 – nginx output

Now the *docker ps* command will list the nginx container, similar to the following figure:

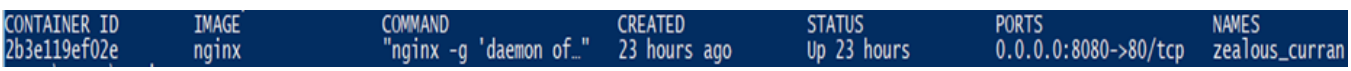


Figure 4 – Docker ps

The container can be stopped by using the name or the first few characters of the container ID. The

following commands show this:

```
> docker stop 2b3e
```

Or

```
> docker stop zealous_curran
```

Running Angular Application in Docker Container

Now let's deploy the pokémon explorer application in a docker container. To do so, a custom image has to be created. As the Angular application needs Node.js on a system to perform its tasks, the image has to be based on the Node.js image.

As mentioned earlier, this article uses the pokémon explorer sample for the demonstration. Download it and set it up before proceeding. This sample is written for an article on server-side rendering of Angular applications so that it can be executed as a browser based application and also as a server rendered application.

To run this sample, open a command prompt and navigate to the folder containing the code and run the following commands:

```
> npm install
> ng serve
```

Open a browser and change the URL to http://localhost:4200 after running the above commands. You will now see the page rendered. To run it from server, run the following commands:

```
> npm run build:ssr
> npm run serve:ssr
```

Open the URL http://localhost:4201 on a browser and you will see the same page getting rendered from the server. We will build images to deploy the application both ways in a Docker container.

Deploying Browser Rendered Angular App

First, let's build an image to run the Angular application on the browser. This image will be based on Node.js image, so it will have Node.js and npm installed from the base image. On top of it, it needs to have the commands to copy the code, build it and run it.

The custom image to host the source code can be created using a Dockerfile. A Dockerfile is a text document that contains a list of commands to be executed inside the image. These instructions derive the image from an existing image and then perform tasks like installing any additional software needed, copying the code, setting up the environment for the application and running the application.

This file will be passed as an input to the *docker build* command to create the image.

Following are the tasks to be performed in the *Dockerfile* for the custom image:

- Extend from the node:8.9-alpine image
- Copy code of the sample into the image
- Install npm packages
- Build the code using *ng build* command
- Set the port number to 80 and expose this port for the host system to communicate with the container
- Install *http-server* package
- Start a server using *http-server* in the *dist/browser* folder

Add a new file to the root of the application and name it *Dockerfile*. Add the following code to this file:

```
FROM node:8.9-alpine as node-angular-cli
LABEL authors=”Ravi Kiran”

# Building Angular app
WORKDIR /app
COPY package.json /app
RUN npm install
COPY . /app

# Creating bundle
RUN npm run build -- --prod

WORKDIR /app/dist/browser
EXPOSE 80
ENV PORT 80
RUN npm install http-server -g
CMD [ “http-server” ]
```

I used *http-server* in the above example to keep the demo simple. If you prefer using a web framework like Express, you can add a Node.js server file and serve the application using Express.

Note: *Notice that the Dockerfile doesn't have a command to install Angular CLI globally. This is because, the Angular CLI package installed locally in the project can be used using npm scripts. The command for creating application bundle invokes the build script added in* ***package.json****.*

Now that the Dockerfile is ready, it can be used to create an image. Before doing that, Docker has to be made aware of the files that don't need to be copied into the image. This can be done using the *.dockerignore* file.

The *.dockerignore* file is similar to *.gitignore* file. It has to contain a list of files and folders to be ignored while copying the application's code into the image. Add a file to the root of the project and name it *.dockerignore*. Add the following statements to this file:

```
node_modules
npm-debug.log
Dockerfile*
docker-compose*
.dockerignore
.git
.gitignore
README.md
LICENSE
.vscode

dist/node_modules
dist/npm-debug.log
```

To create an image for this application, run the following command:

```
> docker build -f .\Dockerfile -t sravikiran/node-angular-cli .
```

The above command provides two inputs to the *docker build* command. One is name of the file provided with the `-f` option and the other is name of the image provided with `-t` option. You can change these names. The period at the end of the command is the context, which is the current folder. This command logs the status of the statements on the console. It produces the following output:

```
Sending build context to Docker daemon  6.025MB
Step 1/12 : FROM node:8.9-alpine as node-angular-cli
 ---> 8bcb66e2abe9
Step 2/12 : LABEL authors="Ravi Kiran"
 ---> Using cache
 ---> 2ec7326c0b49
Step 3/12 : WORKDIR /app
 ---> Using cache
 ---> 3c3d90c29d3e
Step 4/12 : COPY package.json /app
 ---> Using cache
 ---> ef1df76c0da7
Step 5/12 : RUN npm install
 ---> Using cache
 ---> 443176d364a3
Step 6/12 : COPY . /app
 ---> Using cache
 ---> 3db4b4bde0f4
Step 7/12 : RUN npm run build -- --prod
 ---> Using cache
 ---> c285fce365f7
Step 8/12 : WORKDIR /app/dist/browser
 ---> Using cache
 ---> 0ecb9bdf8d14
Step 9/12 : EXPOSE 80
 ---> Using cache
 ---> d0a337af7af2
Step 10/12 : ENV PORT 80
 ---> Using cache
 ---> b1dc25e18ca9
Step 11/12 : RUN npm install http-server -g
 ---> Using cache
 ---> 10db1859761c
Step 12/12 : CMD [ "http-server" ]
 ---> Using cache
 ---> 0ab0763ea313
Successfully built 0ab0763ea313
Successfully tagged sravikiran/node-angular-cli:latest
SECURITY WARNING: You are building a Docker image from Windows against a non-Windows Docker host. All files and directories added to build context will have '-rwxr-xr-
x' permissions. It is recommended to double check and reset permissions for sensitive files and directories.
```

Figure 5 – Docker build

This command will take a couple of minutes, as it has to install the npm packages of the project and then the global *http-server* module. The last statement containing the *CMD* command is used to provide a command to execute in the container. This command doesn't run unless the container is created.

Once the command finishes execution, you can find the image in the list of images on the system. You will find an entry as shown in the following screenshot:

```
> docker images
REPOSITORY                      TAG                 IMAGE ID            CREATED             SIZE
sravikiran/node-angular-cli     latest              0ab0763ea313        20 hours ago        349MB
```

Figure 6 – Docker image

To create the container using this image, run the following command:

```
> docker run -d -p 8080:80 sravikiran/node-angular-cli
```

This command is similar to the command used to run the nginx container earlier. This command runs the application inside the container and makes the application accessible through port 80 of the host system. Open the URL http://localhost:8080 on a browser to see the application.

Though it works, the image of this application takes 349 MB on the host system. This is because, all of the code and npm packages of the application, still live in the container. As the code is running from the *dist*

folder, the source files and the npm packages are not needed in the image.

This can be achieved using multi-step build.

The Dockerfile supports staging builds. The Dockerfile can be used to define multiple images in it and Docker will create the container using the last image in the file. Contents of rest of the images can be used in the last image.

The Image in the Dockerfile created earlier can be divided into two images. One will copy the code and generate the *dist* folder. The other will copy the *dist* folder from the first image and start the *http-server*. The following snippet shows the modified Dockerfile with two images:

```
# First stage image labelled as node-angular-cli
FROM node:8.9-alpine as node-angular-cli
LABEL authors="Ravi Kiran"

# Building Angular app
WORKDIR /app
COPY package.json /app
RUN npm install
COPY . /app
RUN npm run build -- --prod

# This image will be used for creating container
FROM node:8.9-alpine
WORKDIR /app
# Copying dist folder from node-angular-cli image
COPY --from=node-angular-cli /app/dist/browser .
EXPOSE 80
ENV PORT 80
RUN npm install http-server -g
CMD [ "http-server" ]
```

The comments in the above snippet explain the modified parts of the file. Run the following command to create the image:

```
> docker build --rm -f .\Dockerfile -t sravikiran/pokemon-app .
```

This command has the `--rm` option in addition to other options and inputs we had in the previous *docker build* command. This option removes the intermediate images. Otherwise, the image used for staging will stick on the host system. Check the size of this image, it should be around 60 MB, which is significantly smaller than the previous image.

Deploying Server Rendered Angular App

The image file for the server rendered Angular application will have the following differences:

- It will use the *build:ssr* script to create bundles for the browser and the server
- The compiled *server.js* file will be used to start the Node.js server

The following snippet shows this image:

```
FROM node:8.9-alpine as node-angular-cli
```

```
LABEL authors=”Ravi Kiran”

# Building Angular app
WORKDIR /app
COPY package.json /app
RUN npm install
COPY . /app
RUN npm run build:ssr

FROM node:8.9-alpine
WORKDIR /app
COPY --from=node-angular-cli /app/dist ./dist
EXPOSE 80
ENV PORT 80
CMD [ “node”, “dist/server.js” ]
```

Save this image in a file named Dockerfile-ssr. Run the following command to build the image:

```
> docker build --rm -f .\Dockerfile-ssr -t sravikiran/pokemon-app-ssr .
```

Once the image is ready, run the following command to run the container:

```
> docker run -d -p 8080:80 sravikiran/pokemon-app-ssr
```

If you want to see both browser and the server rendered applications running at the same time, you can change the port number to a number of your choice.

Deploying to Azure

Microsoft Azure supports Docker very well. Azure supports container registry, which is similar to the Docker hub registry, where one can push the images and use them for deployment. Other than this, Azure also supports the Docker container clustering services like Docker Swarm, Kubernetes and DCOS. This tutorial will use the Azure Container Service to push the images and deploy them.

To create a Docker registry, login to Azure portal and choose New > Azure Container Registry and click the Create button in the dialog that appears. Provide inputs in the Create container registry form and click on Create.

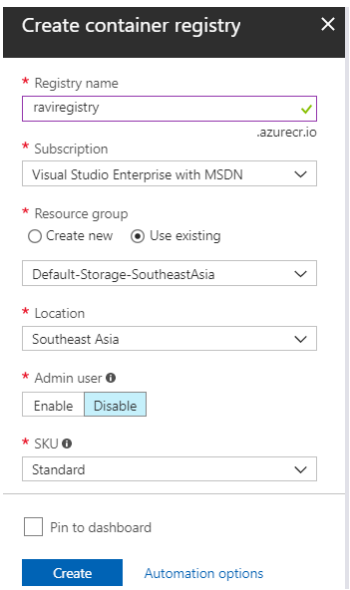


Figure 7 – Create container registry

The first field is the name of the registry you would be creating and it has to be unique. Azure will take a couple of minutes to create the container. Once the registry is created, the Azure portal will show an alert and you can click on the link in the alert to see the resource. On the registry resource page, go to the Access Keys tab and enable the Admin user option.

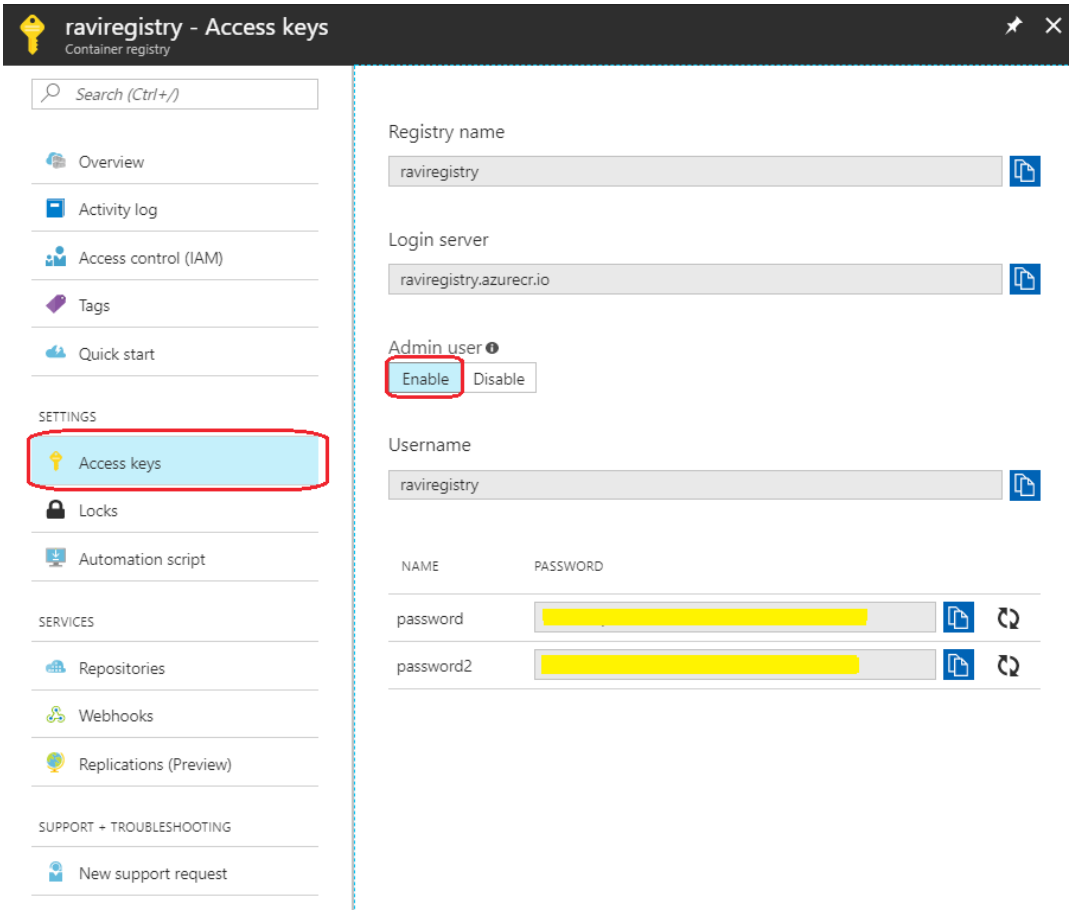


Figure 8 – Registry Access Keys

You will see the passwords of your access keys after enabling the Admin user option. These passwords will be used to login to the registry. To push images to the registry created, Docker client has to be provided access to it. Run the following command to login to the registry:

```
> docker login <your-registry>.azurecr.io
```

Running the command will prompt for credentials. Username would be the name of the registry and one of the passwords from the Access Keys tab can be used for password. Figure 9 shows this:

```
> docker login raviregistry.azurecr.io
Username: raviregistry
Password:
Login Succeeded
```

Figure 9 – Logging into registry

After logging in, create a tag for the image to be pushed. The following commands create a tag for the pokemon-app-ssr image and then pushes it to Azure:

```
> docker tag sravikiran/pokemon-app-ssr:latest raviregistry.azurecr.io/sravikiran/
pokemon-app-ssr

> docker push raviregistry.azurecr.io/sravikiran/pokemon-app-ssr
```

The push command will take a couple of minutes, as it has to copy the whole image to the container registry. Once the image is pushed, it can be seen under the repositories tab of the registry, as shown in Figure 10:

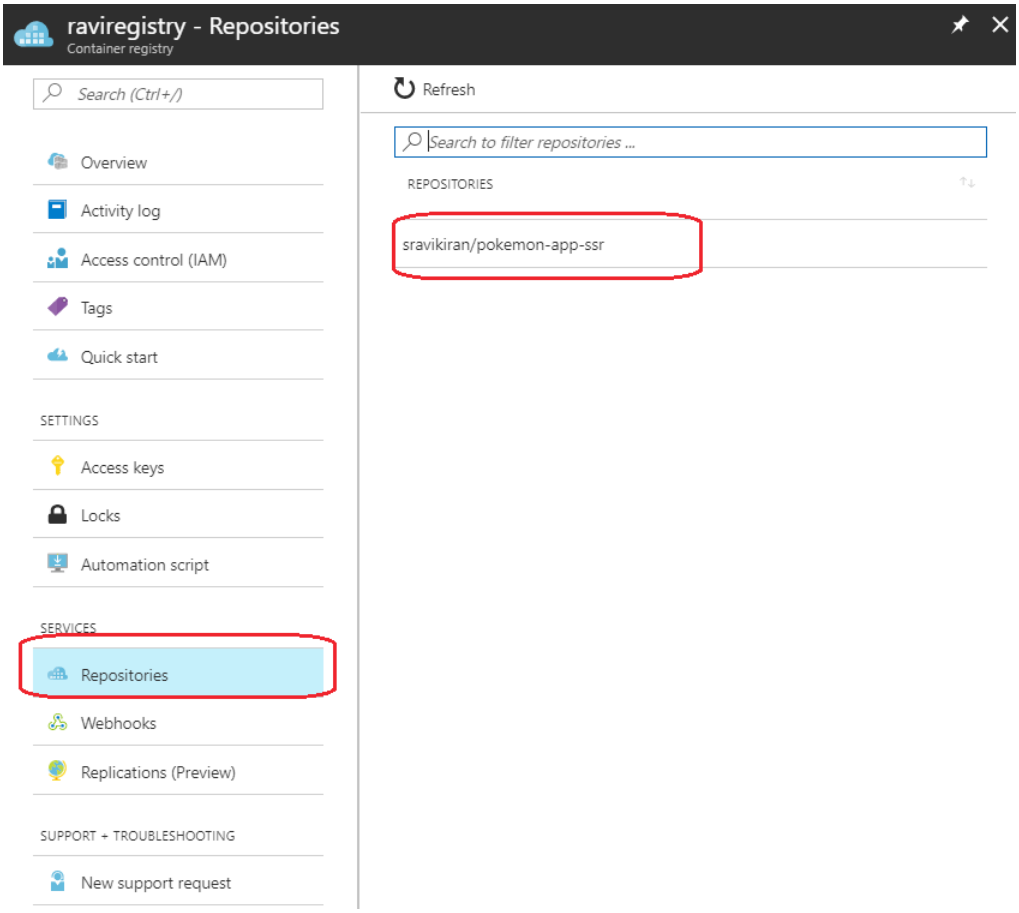


Figure 10 – Repositories tab of registry

A web application can be deployed using this image. To do that, choose New > Web Apps for Containers and click the Create button. Provide inputs for the web app. Then choose Configure container and provide details of the registry, image and tag as shown in Figure 11. After filling the details, click on OK and then on the Create button to get the web application created.

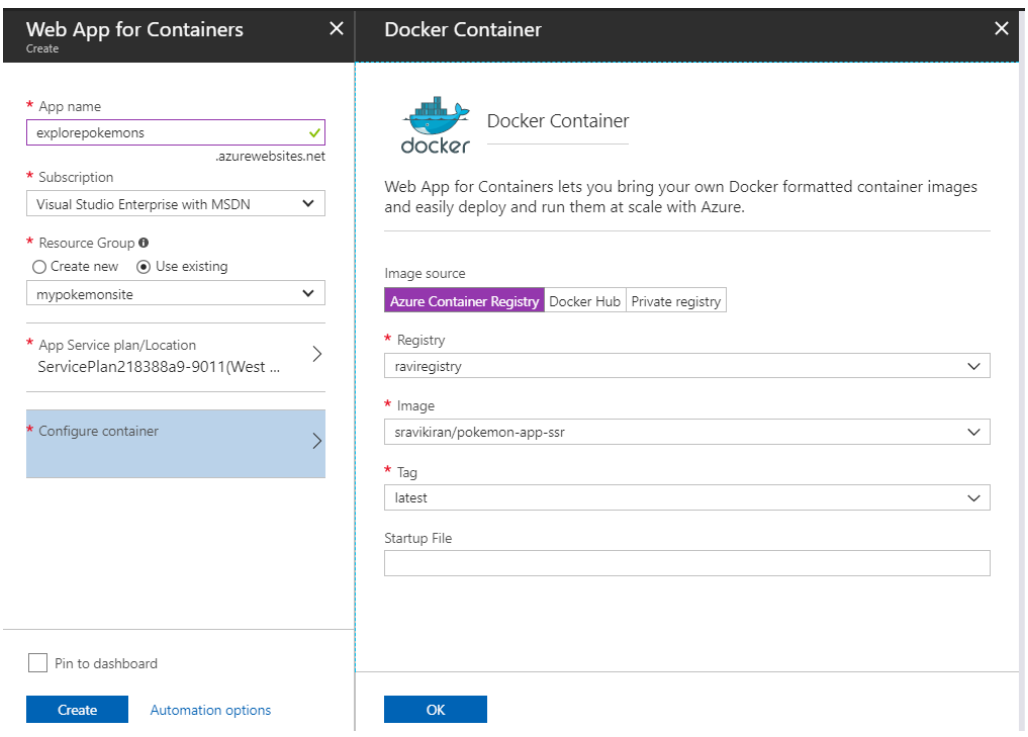


Figure 11 – Inputs for web app

Wait for the deployment to complete and then visit the URL of your site. In my case, the URL is http://explorepokemons.azurewebsites.net. Once this page opens on the browser, you will see the server side rendered site.

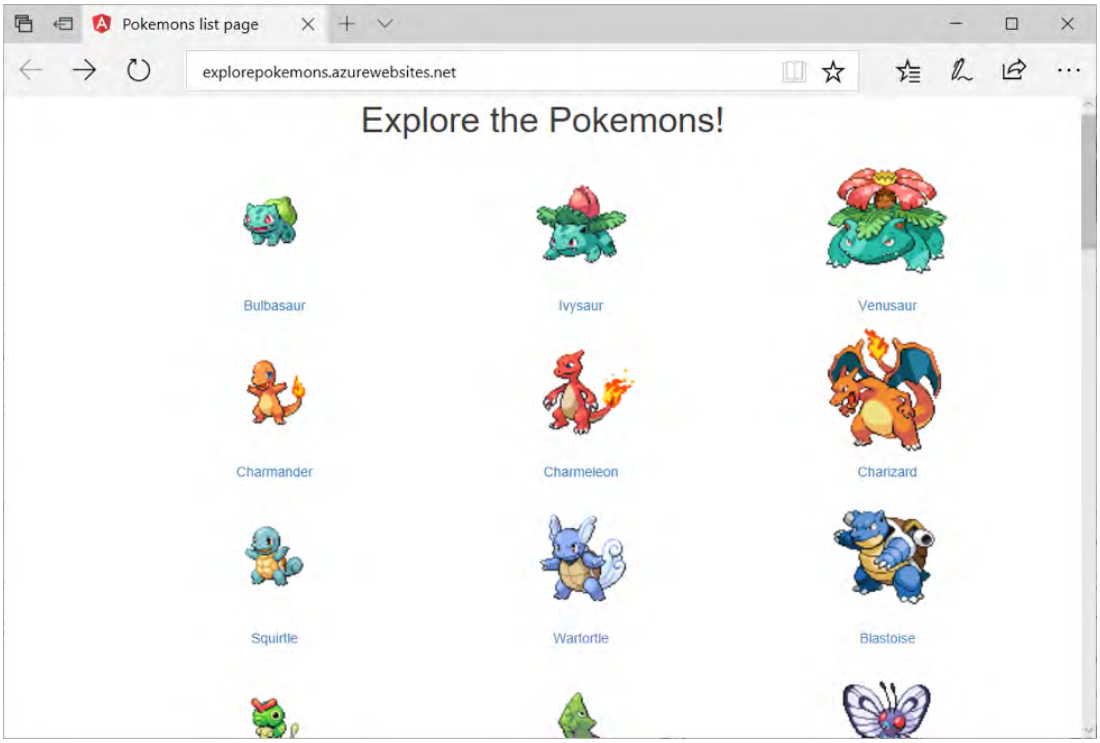


Figure 12 – Pokémon explorer site hosted on Azure

The same process can be followed to deploy the browser rendered version of the application.

Conclusion

Docker is an excellent platform to deploy applications and Azure has an excellent support for Docker. As this tutorial demonstrated, it is very easy to take a Docker image built from an application, push it to an Azure container and get it running. This combination brings developers more control over the environment and it is also quite easy to do.

Let's use this toolset to build and deploy great applications!

Microsoft® Most Valuable Professional

Ravi Kiran
Author

Ravi Kiran (a.k.a. Ravi Kiran) is a developer working on Microsoft Technologies at Hyderabad. These days, he is spending his time on JavaScript frameworks like AngularJS, latest updates to JavaScript in ES6 and ES7, Web Components, Node.js and also on several Microsoft technologies including ASP.NET 5, SignalR and C#. He is an active blogger, an author at SitePoint and at DotNetCurry. He is rewarded with Microsoft MVP (ASP.NET/IIS) and DZone MVB awards for his contribution to the community.

Thanks to Subodh Sohoni for reviewing this article.

Subodh Sohoni

Unraveling the mysteries of DevOps

A majority of IT professionals in the software industry have misconceived ideas about DevOps.

DevOps is considered as something that can be 'implemented' by following 'some practices' by any team.

Some think of a DevOps engineer as a designation in the organization. Others feel that it is an invasion of the development team on the operations team. Another school of thought is that DevOps is an automation of activities carried out by the operations team.

And there are many more notions similar to these. Alas, none of them grasp the essence of DevOps!

In this article, I will try to provide some clarity about the concept of DevOps.

Image Courtesy: Wikimedia Commons

What is DevOps?

To understand the concept of DevOps, we first need to have a clarity about Agile development.

Editorial Note: *If you are new to Agile Development, make sure you read these tuts to understand its benefits.*

www.dotnetcurry.com/agile/1297/agile-for-fixed-bid-projects
www.dotnetcurry.com/software-gardening/1087/what-is-agile-software-development

Agile development seeks to provide early and rapid (frequent) delivery of software that is under making. This has two benefits.

The Customer can start using at least a part of the software, in a short time. This provides a sense of satisfaction to the customer that there is some return on the investment (ROI).

It also provides an opportunity for the customer and the team that is developing the software, to review the features of the software that are ready for use. These frequent reviews and related feedbacks make it possible to take corrective actions, incorporate changes in requirements and fix observed bugs, at the earliest and at the minimum cost.

By using agile development, most of the products and projects that develop custom software, stand a bigger chance to succeed.

Agile development is not a concept that can be implemented by any team. The team needs to become agile to do Agile development. Just like a fighter aircraft which is designed to be agile vs a passenger aircraft not designed to be agile; **the team that is doing Agile development, needs to be designed to be agile**.

The benefits of Agile development are accrued to the customer only when the software is developed, tested thoroughly and then deployed to production.

One of the issues faced by some of the agile teams is that the development process is made quite agile, but the process of build and deployment to environments for testing and production, are not designed to be agile.

The team that is developing software needs to embrace Agile practices for these operations as well.

Some of the impediments to Agile Development are:

- There is a team that is doing development. Let us call it the 'Development Team'. The team that is doing build and deployment is a different team. Let us call it the 'Operations Team'. The two teams may not be located at the same place. They may be insulated from each other.

- Development Team and Operations Team communicate only in a formal and indirect channel. Formal emails and approvals may be necessary to run a build or do deployment in certain environments.

- These teams are not in sync with each other regarding their goals, schedules and planned activities. The Operations team may have their schedule of build and deployment, which may not be in line with the Agile development practices. This can increase the time for delivery.

- The build and deployment activities may not be sufficiently automated to give the fastest results possible. Manual processes are slow to implement and are also error prone.

- Testing team may have their own schedule for cycles of testing. Those may not be in sync with the development and release cycles.

- Tools used by Development team, testing team and operations team may not integrate very well.

- Development team desires to use the latest technology and do the development in the shortest possible time. Testing team considers that maintaining quality is only their job and may not consider the rapid delivery to be of prime importance. Operations team always resists any changes because, sometimes, those may lead to crashes and service stoppages. These mindsets pull the teams in different directions.

Do these inhibitions sound familiar to you?

This list may not be comprehensive, but is thought provoking to the extent that you start wondering why the benefits of Agile development should not be passed on to the customer and the end user, as early and rapidly, as the development phase.

That is why we extend the agile team to become a DevOps team.

DevOps team

The DevOps team is a team that has representation from developers, testers and operations.

This team has a common goal - to provide early, rapid and high-quality product to the customer. Their mindset is aligned, they do not consider team members with external roles and use well integrated tools to achieve the common goal.

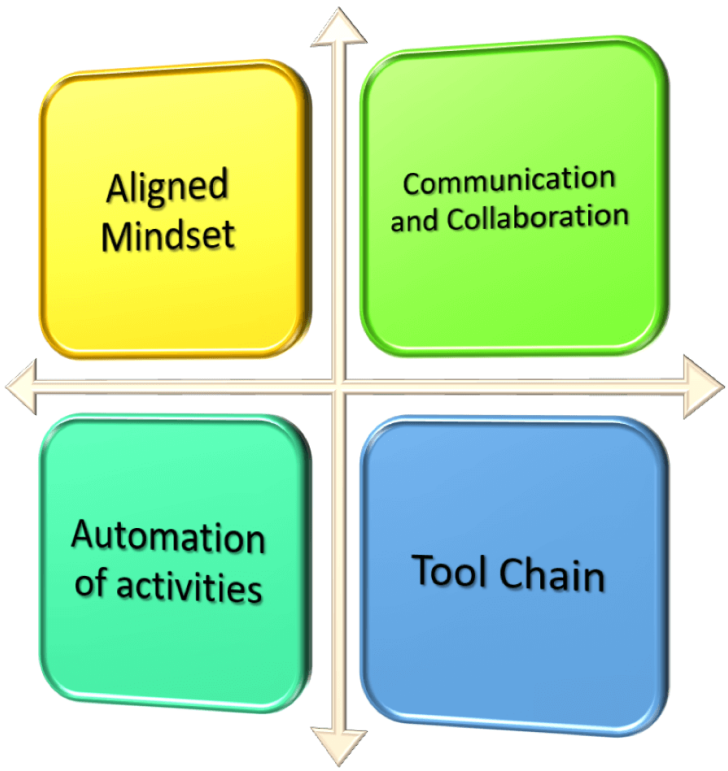


Figure 1 – Four pillars of DevOps

DevOps - Mindset Changes

DevOps team is an integrated team comprising of members of Development, Testing and Operations. All the team members have same schedule and work in-sync to provide earliest and rapid delivery to the customers.

Most importantly, all the team members have the same mindset.

The success criteria for the team consists of early and most frequent delivery of usable software by the customer.

This criterion makes productivity, quality and rapid delivery as the goals for every team member. To adopt the mindset of DevOps, the team members have to break their earlier mindsets.

For eg: Development team members have to now also consider quality and operations as responsibilities owned by them. Testers should now work in line with the schedules of development and release promises.

Similarly, team members who were part of the Operations team have to now find different innovative ways to provide delivery of the software as rapidly as the software gets developed, without losing the reliability of the deployed software.

The biggest impediment to forming a DevOps team is the resistance to change the original mindsets of each role and create a new aligned mindset for the entire team. It is a misconception that every team member in the team has to be skilled in development, testing and operations. What is desired is that the team should have sufficient team members from all these streams and that they have common mindset, so that as a team, they can provide the most rapid increments of the product to the customer.

Communication and Collaboration

Like any other modern development team, the DevOps teams also depends heavily upon seamless and clear communication.

It is desirable to have co-located teams so that such communication is not dependent upon the vagaries of Internet and other devices. The team should be able to have a formal and informal channel of communication by which the synchronization of tasks between team members doing development, testing and operations, will be achieved.

Direct communication and collaboration is recommended to have unambiguous and continuous communication. Collaboration tools can be used as desired but should not be the impediments in communication.

Teams should be able to describe, detail out and decide on the schedules for the creation of environments, needs of provisioning, testing cycles, branching and merging as per the decided strategy etc. This list is not comprehensive but will give you some idea about how collaboration can improve the time - from development to deployment.

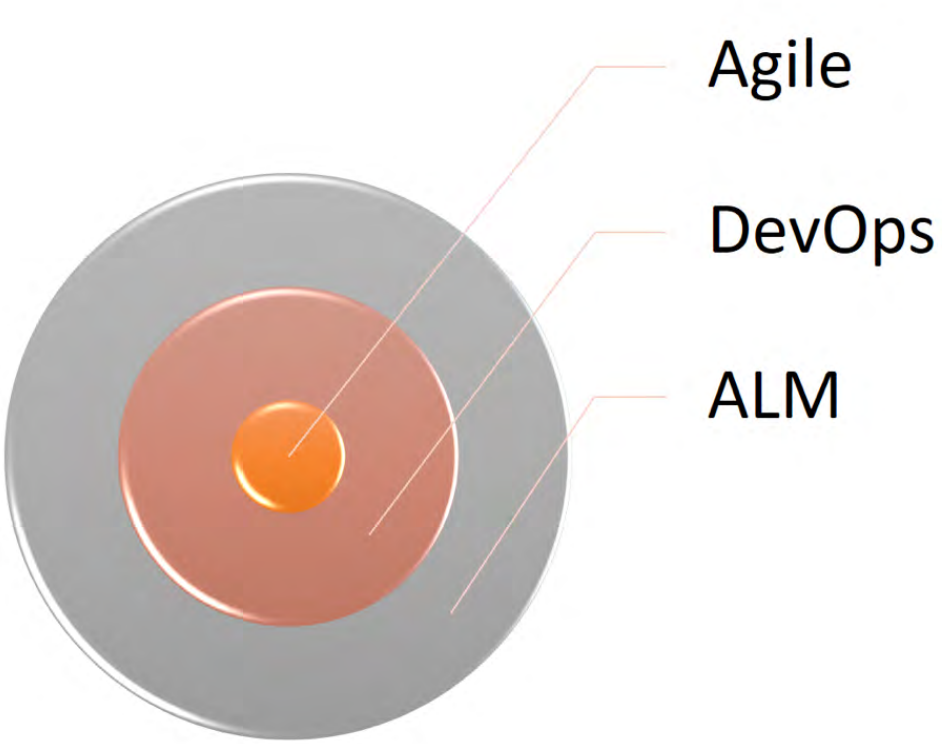


Figure 2 – Relationship between Agile – DevOps – ALM

DevOps shares most of the values with Agile. It does extend some of these values to suite the unique needs of the DevOps teams, and defers to a certain extent from those values. Let us go through some of them.

Individuals and Interactions Over Process and Tools – DevOps gives equal importance to Process and Tools. The team has to achieve not only development but also rapid deployment. Continuous Integration has to be extended to Continuous Deployment and if possible, Continuous Testing. This extension cannot be easily achieved without appropriate processes and tools.

Working Software Over Comprehensive Documentation – DevOps is in line with this value.

Customer Collaboration Over Contract Negotiation – DevOps extends this value by providing a set of concrete opportunities and tools for the customer to collaborate while creating the product backlog, as well as post deployment.

DevOps covers the post deployment feedback as well, as issues raised by the customer. In this regard, it goes towards ITSM.

Responding to Change Over Following a Plan – DevOps allows team to not only develop the code, but also build and deploy it in line with the business needs and feedback received.

We can consider DevOps as extending Agile to cover Operations also.

Naturally, DevOps team will also have to follow the practices of Agile development and agility in requirements management, as well as efforts management.

Agile development practices like continuous integration, test driven development, code reviews, pair programming are extended by operations practices like continuous deployment, automated provisioning, collecting application health data, feedback from client etc. Sprint planning now should consider plans of testing, provisioning and deployment too. The tasks for those activities should be included in the sprint plan and monitored through the task board and burndown chart.

Editorial Note: *If you are new to concepts like CI, TDD, Pair programming and such, read www.dotnetcurry.com/visualstudio/1338/agile-development-best-practices-using-visual-studio.*

Automation

DevOps does not need automation just for the sake of it. What it essentially needs is the rapid delivery to the customer.

Besides actual coding, it is a necessity to rapidly and frequently create environments, provision those with necessary software, deploy software and test the deployed software.

This may be needed to be done many times in a day in case of initial environments like Dev and Test environments. This frequency can only be achieved if the DevOps team adopts automation to execute these activities.

Manual execution of the same may not be as quick, as required. Another advantage of automation that has been defined properly, is repeatability without any errors.

Let us define the activities which need to be automated.

1. Build should be triggered as and when the code changes in the repository due to Commit – Push or Check-in
2. Activities of the build should be executed in a specified order
3. Build should run unit tests as configured
4. A release should be created that should link the created build output with the deployment scripts
5. Environment should be created and provisioned, if necessary
6. Build output should be deployed to Dev environment where smoke tests can be run.
7. Automated tests should be run as the final bit of the deployment
8. Necessary emails for notification and authorization requests should be sent
9. Deployment to subsequent environment should happen automatically, albeit after manual approval for those.

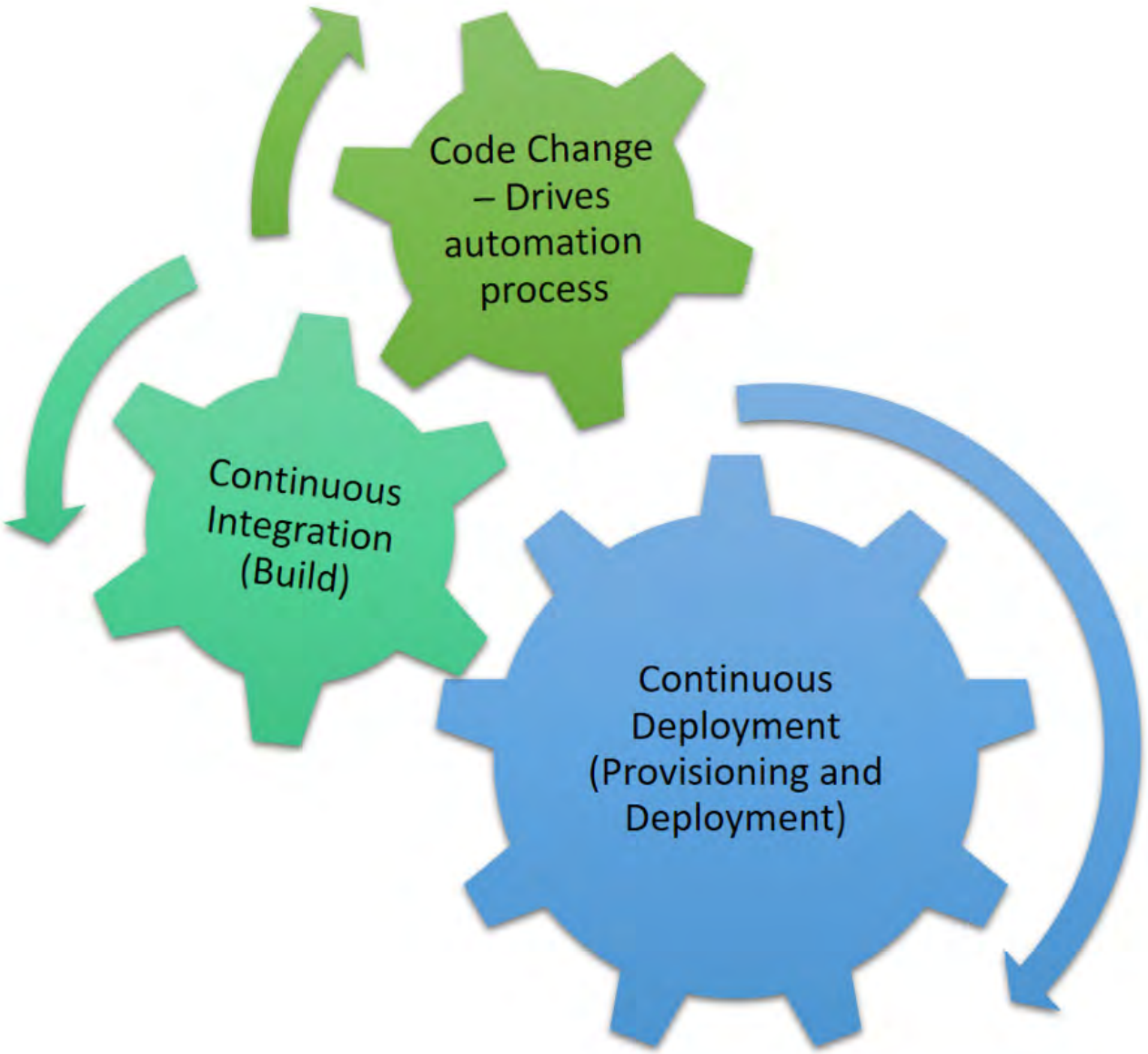


Figure 3 – Automation in DevOps

These automation tasks are possible only with some appropriate tools.

Let us have a look at those tools.

Tools for Automation

Tools that automate various activities and facilitate continuous delivery is also an essential part of DevOps.

There are a number of tools for different activities. Many a times, these tools form a toolchain for activities in a serial order.

The following table summarizes the most popular tools.

Operation	Microsoft Tools	Non-Microsoft Tools
Agile Management	TFS, VSTS	Jira, RTC, Rally
Version Control	TFS, VSTS	Git, BitBucket, Mercurial, RTC, SVN
Build	MSBuild, Visual Studio, TFS, VSTS	Ant, Maven, Jenkins, Gradle, Make, Cruise Control, Buildbot, TeamCity, Gulp, Grunt
Test	MSTest, VSTest, Unit Test, MTM, CUIT, Web Performance Test, Load Test	nUnit, Xunit, jUnit, QTP, Selenium, HP ALM, RFT, WinRunner, LoadRunner, SilkTest
Packaging	Windows Installer, TFS, VSTS	InstallShield, NuGet, NPM
Provisioning	TFS, VSTS, SCVMM, ARG Template	Chef, Puppet, Ansible, Vagrant
Deploy	RM (TFS, VSTS), PowerShell Scripts	Octopus Deploy, Juju, Buildbot
Monitor	Application Insights	New Relic, Stackify, Nagios, AppDynamics, SolarWinds
Feedback	Feedback Client	UserVoice

Many of these tools are industry leaders in their own field of work.

For example, Jenkins for build automation, Selenium for automation of testing and Octopus for deployment, are industry leading softwares.

Although these are excellent tools in their field, they do not integrate with other tools seamlessly. It takes a lot of efforts and pains to integrate those with each other.

For example, to integrate Jira with Jenkins for build, and Octopus for deployment, may require efforts from the development team. To reduce the pain of such integration, many organizations desire to have integrated set of tools that are integrated by design.

Microsoft TFS, Microsoft VSTS, Rational Team Concert (RTC), HP ALM are such suites of tools that are integrated by design. These tools provide almost all the toolchain under one roof. Needless to say, these are much more convenient to enterprises.

Many of these tools are offered as installable on the hardware that is owned by the organization using

it. Microsoft Visual Studio Team Services (VSTS) is an Azure cloud offering which works as Software as a Service (SaaS). It is installed, maintained and updated frequently by Microsoft in a multi-tenant model. Since the organizations do not have the expenses of hardware, maintenance of software and basic administration, this works out to be quite attractive.

Dos and Don'ts for a DevOps Team

Here are some pitfalls for the DevOps team to avoid and some best practices:

- **Focus only on one of the aspects of DevOps** – Tools, Automation or Culture. DevOps has these equally important facets. Looking at only one of them and ignoring others will not give the team the desired benefits.

- **Make operations team common to all development teams** – I have seen many companies keeping the same operations team, as was earlier. They then try to improve communication between development team and operations team. This may not give the desired level of collaboration because of the adherence to old mindsets and not having shared goals.

- **Choosing speed over quality** – Early and Rapid delivery is a goal that DevOps team should desire to achieve, but not at the cost of quality. The customer cannot get any value from the deployed product if that product has bugs in it. Although no one can say with 100% surety that a software has no bugs, we should ensure that at the least there are no known bugs at the time of deployment. DevOps team should take care of that by testing in multiple stages.

- **Ignoring Database – DataOps** – Coding is not the only way to create deliverables. Design, implementation and versioning of database is equally important for the success of an application. DevOps team should provide equal importance for the activities related to database development.

- **Forgetting Security** – In the hurry to deliver software early, DevOps team should not take shortcuts in ensuring the security of the application. Any vulnerabilities and loopholes in the security can result in disasters in the long run.

- **Not involving all concerned persons and stakeholders early on** – I have observed many teams involving the testing and operations team members at a very later stage in the development cycle. These team members may not be able to plan, complete their activities in time or may not be able to execute those activities, with sufficient quality. It is my strong suggestion to ***involve all the team members and stakeholders, early-on in the development process***.

- **Creating a separate special "DevOps Team"** – The entire team that is delivering software is a DevOps team and every team member of that team is equally accountable for development, build, deployment and testing of that software. It is not that a separate team is to be formed for these activities. If an organization has a post of DevOps Engineer or has a separate DevOps Team, then that organization has not really unraveled the mystery of DevOps.

- **Not understanding that Scrum ≠ Agile ≠ DevOps** – Many organizations feel that if they do a daily standup meeting, they have adopted Scrum and that means they have become agile. A team can become agile only if it has agile values, it follows agile principles and practices agile development and agile management. Scrum is only one part of becoming agile, that is of agile management. Extending this reasoning, being agile does not mean that the team is a DevOps team. ***When the agile team is extended for operations and testing, only then it becomes a DevOps team.***

- **Failing to look at the larger picture of ALM** – Some organizations and many software professionals believe that developing software is an end in itself. I need to remind them that software is only one of the ways that businesses can achieve higher productivity, quality and in general profitability.

  Software supports business and not vis-à-vis. DevOps team should not forget that besides DevOps practices, there are other practices of ALM like requirement management, efforts management, risk management etc. which are equally important for the business. Organizations should look at this larger picture of ALM, should know why they are developing software and remain a part of this frame.

Summary

In this article, I have tried to demystify the term ***DevOps***. As observed in this tutorial, the Four pillars of DevOps are:

Adoption of mindset of a team that is well integrated with members from development, testing and operations, equally accountable and passionate about all activities towards the goal of early and rapid delivery.

Seamless Communication and Collaboration between all team members.

Automation to reduce the time to deliver software with high frequency and

Tools that facilitate such automation.

When all these pillars are of equal strength, only then the promise of early and rapid delivery can be fulfilled by the DevOps team!

Microsoft® MVP Most Valuable Professional

Subodh Sohoni

Author

Subodh is a consultant and corporate trainer. He has overall 28+ years of experience. His specialization is Application Lifecycle Management and Team Foundation Server. He is Microsoft MVP – VS ALM, MCSD –ALM and MCT. He has conducted more than 300 corporate trainings and consulting assignments. **He is also a Professional SCRUM Master. He guides teams to become Agile and implement SCRUM**. *Subodh is authorized by Microsoft to do ALM Assessments on behalf of Microsoft. Follow him on twitter @subodhsohoni*

Thanks to Suprotim Agarwal for reviewing this article.

Gerald Versluis

Adding AI to your Xamarin Apps Using Cognitive Services

These are awesome times to be a software developer!

There is so much power at the tip of your fingers for you to use. Take for instance, Cognitive Services.

Microsoft provides you with massive Artificial Intelligence (AI) and machine learning power and you can access it through a simple REST API.

In this article, we will see what Cognitive Services is and how easily we can incorporate it into Xamarin apps, to make them smarter.

Background on Artificial Intelligence (AI)

If you have been working with Microsoft technologies (or some others as well for that matter), you couldn't have missed all the excitement and announcements around AI and machine learning. While this is a very extensive area, I will just be focusing on a subset; the **Cognitive Services by Microsoft**.

But first, to understand the *what* and *how*, let's take a step back and look at the overall concept of AI.

Artificial Intelligence isn't new. It has been the subject of many movies like The Terminator and The Matrix and there also has been a lot of talk about it in real life. All the way back in the 1950's, two key figures played a key role in modern day AI: Alan Turing and Marvin Minsky.

Turing Test

Alan Turing was a brilliant mathematician and a computer scientist among other things and played a great role in WWII.

But more relevant to AI, he developed the Turing test in 1950. Basically, what this test describes is a way to determine if a system is intelligent. The standard interpretation (there are variants nowadays) is to setup one human interrogator with another person and a machine, and let them communicate with just written natural language. If the human interrogator cannot tell apart the human from the system, the system is intelligent.

This setup is shown in Figure 1.

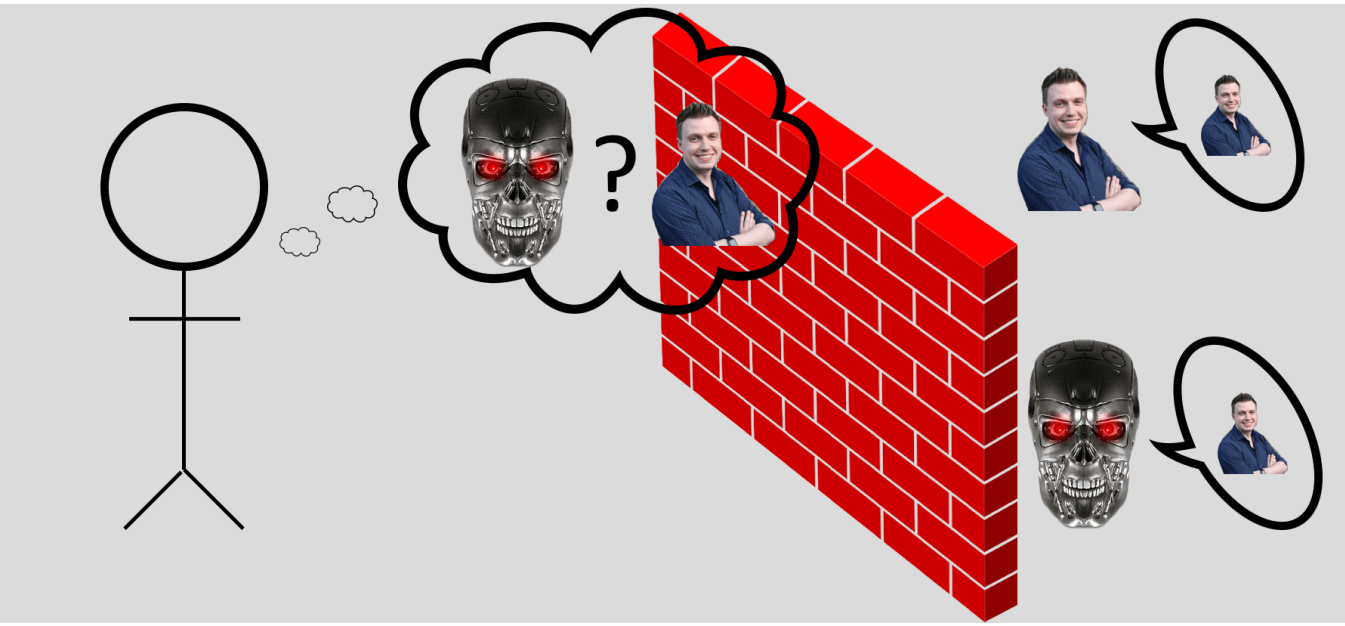


Figure 1: Turing Test Setup

While the test is widely known, it is also highly criticized. Read more on the Turing test on Wikipedia: https://en.wikipedia.org/wiki/Turing_test.

Dartmouth

In the summer of 1956, the Dartmouth Summer Research Project on Artificial Intelligence took place and is now considered (by some, not others) to be the foundation of the modern ideas on artificial intelligence.

Although **John McCarthy** is named as 'the father of AI', there were others too working on this project and are duly mentioned for defining different areas of AI.

One of the outcomes the brilliant minds of the Dartmouth project came up with, is the definition of full AI. They identified five areas a system has to comply to, in order to be self-aware. The five areas are:

- **Knowledge**: the ability to learn, know where to find new knowledge and link entities together;
- **Perception**: see objects and recognize them, identify them and classify them;
- **Language**: understand spoken and written language, as well as producing written and spoken language;
- **Problem solving/planning**: analyze a problem and solve it by taking consecutive actions which built up to a solution. This includes navigation;
- **Reasoning**: being able to explain its behavior and produce arguments why it is the best solution or action.

All these areas combined, produce a self-aware system that is a fully functional artificial intelligence. From this point in time, we probably need to start worrying...!

What are the Cognitive Services?

In line with the Dartmouth Project, Cognitive Services by Microsoft follows the same pattern.

If we look at the categories defined in Figure 2, you will notice how they are very similar to the areas identified during the Dartmouth Project.

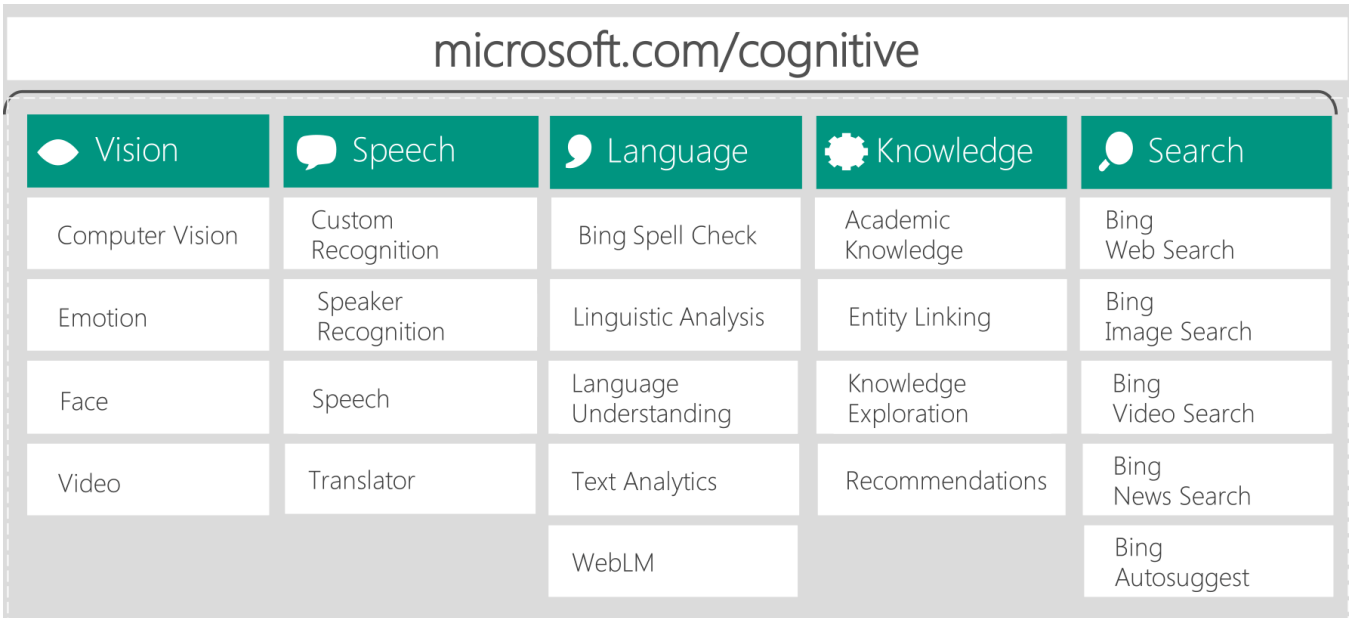


Figure 2: Cognitive Services Overview

In Figure 2, you can see the five categories of Cognitive Services: vision, speech, language, knowledge and search. Under each category there are several sub-categories in which Microsoft has placed its products.

In this article, I will be focusing mostly on the **vision**, but be sure to check out the products in the other areas as well. There is a lot of awesome and powerful stuff in there. For instance, under *speech and language*, there are solutions to interpret spoken language or filter out the sentiment of a sentence.

You can also leverage a powerful spell checker, translate written or spoken text as well as leverage the power of Bing. To get you started, check out www.microsoft.com/cognitive, with some samples over there to get you started.

How all of this is related to mobile

Everything that I have mentioned so far is not specific to mobile or Xamarin. You can incorporate it into any project, it doesn't even have to be .NET!

Since Cognitive Services enables developers to harvest this powerful technology through simple REST APIs, you can use it from any programming language that allows you to work with HTTP and JSON.

But using it with mobile has one big advantage: a mobile device tends to have all the sensors available to capture speech or images. So, it makes sense that you use these APIs from within a mobile app because it allows the end-user to snap a picture, speak into the microphone or let them enter text.

For this article I will focus on the Vision APIs of Cognitive Service.

Cognitive Services & Xamarin

In this example, I will build a simple app that can take a picture and describe the object that is on the image. To show you how easy it is to implement another API, I will also make use of the **Emotion API** to describe whether a person on the image is happy, sad, angry, etc.

For this example, I will be using Visual Studio for Mac, but everything you see here is also possible with Visual Studio on Windows. I will not go through all the Xamarin.Forms specifics in detail, as this article is mostly about Cognitive Services.

Editorial Note: *To learn more about what's new in the latest Xamarin.Forms v3, check www.dotnetcurry.com/xamarin/1393/xamarin-forms-v3-new-features*

All code for the upcoming example app can be found at
https://github.com/jfversluis/DNC-CognitiveServices/

Gathering the pre-requisites

Cognitive Services are part of the Azure suite.

To be able to communicate with the APIs, you need to generate a key. This key should be included in all the requests.

Depending on the specific API that you want to use, there is usually a free tier and a priced tier available. Limitations are imposed in the form of rate-limits, i.e. a specific number of requests within a certain amount of time. Some APIs are still in preview which means they are free to use while they remain in the preview phase.

Create an Azure account if you do not have one already and log into the portal at https://portal.azure.com. In the top-left corner, find the 'New' button to add a new service to you Azure subscription.

If you look at the resulting panel, you will notice the 'AI + Cognitive Services' option in the list. This can be seen in Figure 3.

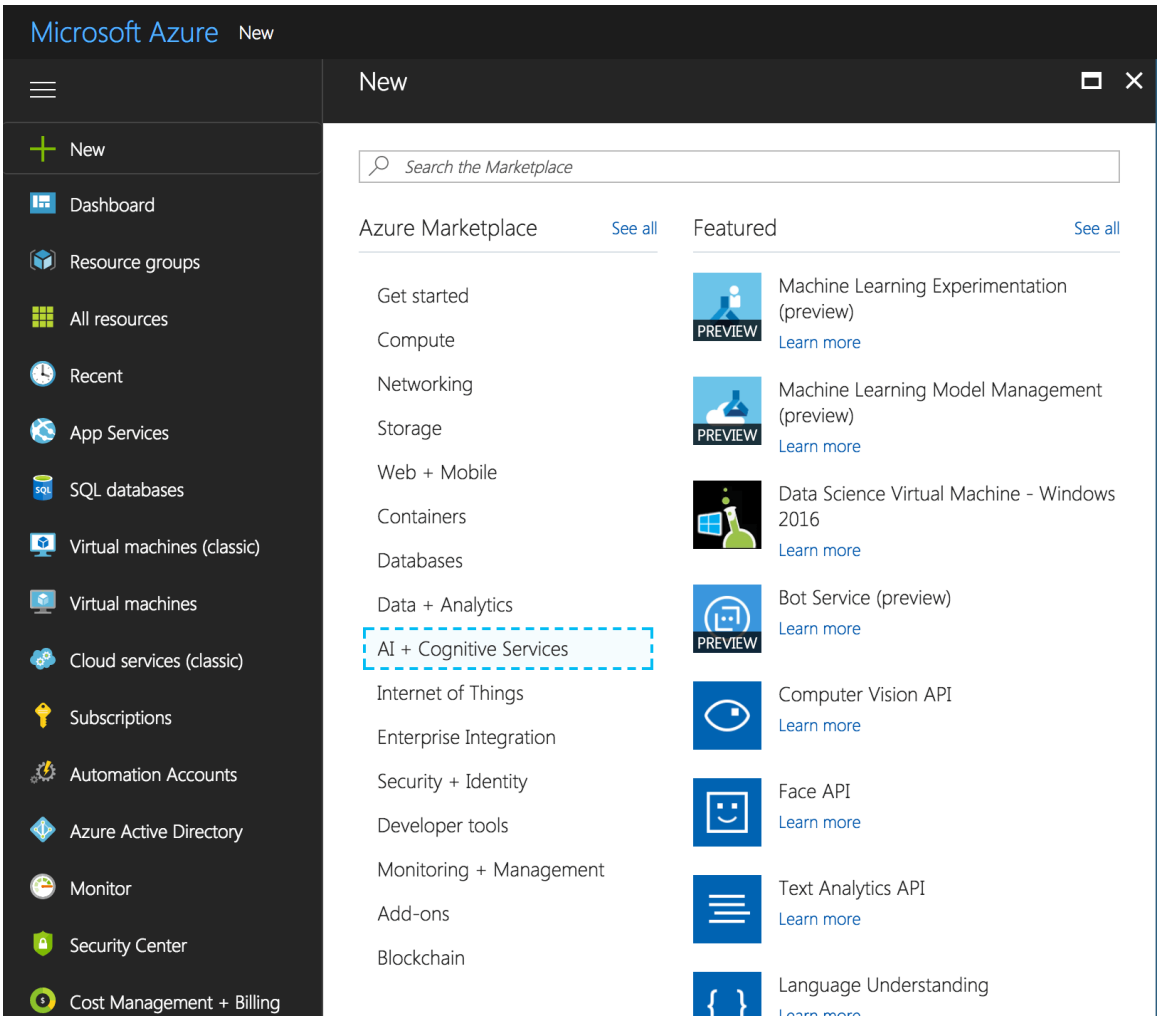


Figure 3: AI and Cognitive Services in Azure Portal

Here you can choose the service that you are after. I will start with the Computer Vision API. This API can be used to recognized objects in images. Later on, I will use the Emotion API.

Here's how to retrieve a key for this procedure. After clicking on the Computer Vision API option, you will have to configure a few basics. Name the service appropriately, choose the right subscription (if applicable), geographic location, pricing tier and resource group. When done, click the 'Create' button and let Azure work its magic. Once done, go into the newly created service and find the 'Keys' pane, shown in Figure 4.

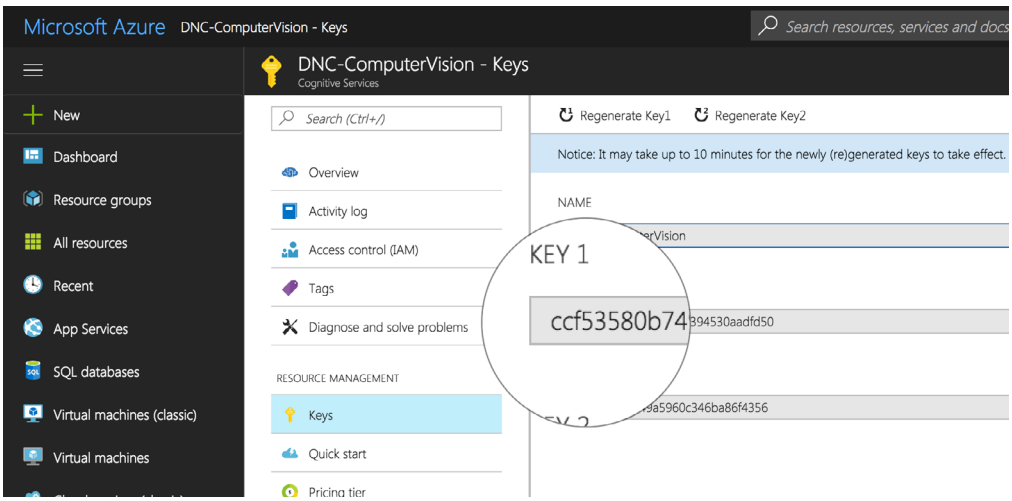


Figure 4: Computer Vision API Key

Here although two keys are shown, you only need one out of them. You can repeat this process for any other Cognitive Services APIs. Also, go into the 'Overview' pane and make note of the URL under 'endpoint'.

With this key in hand, you can now fire requests at the REST API, although to make life easier, there are also a range of NuGet packages available for us to use. We will see this in a little bit.

Setting up the Xamarin project

For starters, I will just 'new-up' a Xamarin.Forms application that will be the base of this example. I will use a Portable Class Library (PCL) instead of a shared library. On Windows, you should be able to use .NET Standard 2.0 out of the box by now.

The UI will be straight-forward - with one `Image` control to preview the image taken by the user, and a `Button` to take a new image.

The first thing we will do is add a couple of NuGet packages. This will drastically simplify taking pictures and accessing the Cognitive Services APIs.

The first Nuget package, for taking pictures is the **Xam.Plugin.Media** package my James Montemagno. You need to install this package on all of your projects. Don't forget to add the right permissions on Android and info.plist entries on iOS. This is described in the packages' readme.txt or on the GitHub page at https://github.com/jamesmontemagno/MediaPlugin.

Second, we install the packages for Cognitive Services. Unlike what you would expect, the packages are named **Project Oxford**. This was the codename that Microsoft gave to Cognitive Services. The packages have not been renamed accordingly, probably due to the wide spread breaking changes.

For the Computer Vision service, we need the **Microsoft.ProjectOxford.Vision** package. This only need to be installed on the PCL project.

Now we have everything we need to start implementing the real code.

Say Cheese Please! Taking the pictures

The code needed to take a picture is quite easy and is shown here.

```
var photo = await CrossMedia.Current.TakePhotoAsync(new Plugin.Media.Abstractions.
StoreCameraMediaOptions());
if (photo != null)
{
PhotoImage.Source = ImageSource.FromStream(() => { return photo.GetStream(); });
  // TODO: Send to Cognitive Services
}
```

This code executes the camera command and opens a camera. When a picture has been taken successfully, it is used as a source for our `Image` control. I will put this code in the `Clicked` handler of our `Button`.

Note: *Since the camera isn't always supported by simulators/emulators, be sure to run this code on a physical device.*

The code to send a request to the Computer Vision API can be found below. I will walk you through it. This

code will be placed instead of the TODO in the above code.

```
var client = new VisionServiceClient("ccf53580b7494775954394530aadfd50",
  "https://westeurope.api.cognitive.microsoft.com/vision/v1.0");

var result = await client.DescribeAsync(photo.GetStream());
await DisplayAlert("Result", result.Description.Captions.FirstOrDefault()?.Text ??
"", "Wow, thanks!");
```

First, a new `VisionServiceClient` is instantiated. This is done by specifying the key and endpoint URL we got earlier. And actually that is all we need to do to invoke requests!

In the above code, I have chosen the `DescribeAsync` method, but this is just one of the possibilities here. You can also analyse handwriting, generate a thumbnail, recognize text and much more. For this example, I will stick with just retrieving a description.

The result is then shown in a simple alert.

When we now run the app, press the button and take a picture, you will receive a description from the Computer Vision API. Take a look at Figure 5 where you see the picture on the left, and the result on the right.

Figure 5: Computer Vision API Output

In the result, I have also added a percentage of confidence, which in our case was 76.67%. This is something that is returned for each caption. Please refer to the extended code below to make this possible.

```
var client = new VisionServiceClient("ccf53580b7494775954394530aadfd50",
            "https://westeurope.api.cognitive.microsoft.com/vision/v1.0");

var result = await client.DescribeAsync(photo.GetStream());

var topCaption = result.Description.Captions.OrderBy(
c => c.Confidence).FirstOrDefault();

var caption = string.Empty;

  if (topCaption == null)
    caption = "Oh no, I can't find any smart caption for this picture... Sorry!";
  else
    caption = $" I think I see: {topCaption.Text}, I am {Math.Round(topCaption.
Confidence*100, 2)}% sure";
```

```
await DisplayAlert("Result", caption, "Wow, thanks!");
```

In `blue`, you can find the code added to implement the percentage of confidence. You will notice how I order the captions by its confidence to get the top-most one. Then I check to see if there was a caption at all, and pour it into a human readable form, together with the confidence level.

But you can retrieve much more than just the description. It can also contain a set of tags, faces that are detected - whether it's adult content, the color usage and more. It is also possible to return just the amount of detail that you are looking for, by adding parameters to your requests, and save yourself from the overhead of all the unwanted data .

Everybody happy?!

To show you that other APIs are just as powerful and easy to integrate, I will also demonstrate the Emotion API. With this service, you can identify emotions of a human face through Artificial Intelligence (AI).

Retrieve a key and the right endpoint from the Azure Portal and install the `Microsoft.Oxford.Emotion` package on the PCL project of our app.

To access the API, we can now simply create an instance of the `EmotionServiceClient`. I have added a second button to the (already spectacular) UI to snap a picture for our emotion recognition. I added the following code to the button click:

```
var client = new EmotionServiceClient("API-KEY",
             "https://westus.api.cognitive.microsoft.com/emotion/v1.0");

var result = await client.RecognizeAsync(photo.GetStream());

var topEmotion = result.FirstOrDefault()?.Scores?.ToRankedList().FirstOrDefault();

  var caption = string.Empty;

  if (topEmotion == null || !topEmotion.HasValue)
        caption = "Oh no, no face or emotion could be detected. Are you Vulcan?";
  else
        caption = $"{topEmotion.Value.Key} is the emotion that comes to mind..";

  await DisplayAlert("Result", caption, "Wow, thanks!");
```

As you can see, this code is almost identical to the code for the Computer Vision API. For a few results, refer to Figure 6, below.

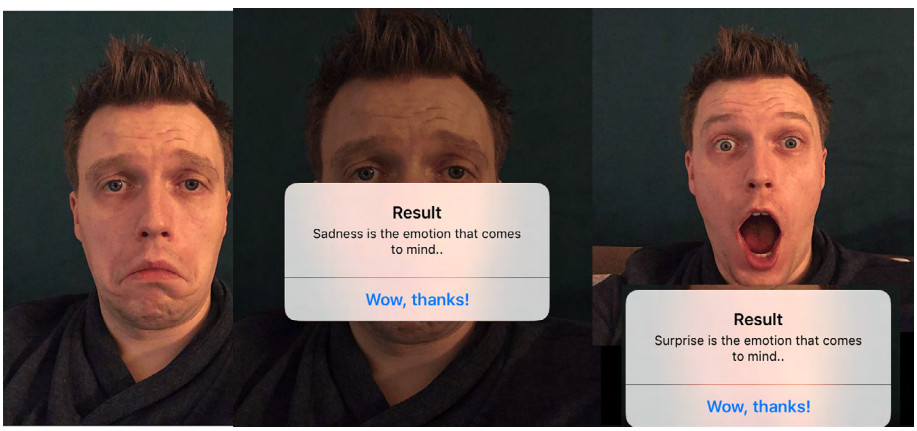


Figure 6: Emotion API in action

Besides the Emotion API, there is also the Face API. I was very impressed by the Face services especially. The amount of detail that is returned is astonishing. It can tell you the age, sex, if someone is wearing make-up, glasses or even swim goggles!

Why not download the sample app at https://github.com/jfversluis/DNC-CognitiveServices/ and try to add the Face API yourself and Tweet me the results? I am on twitter at @jfverlius.

Final thoughts

Cognitive Services are not just fun to play around with, they are really powerful tools that can help you with a broad variety of tasks. For example; programmatically detect adult images, scan product reviews for negative sentiment, retrieve text or numbers through OCR to prefill fields in your app and much, much more.

All of these tasks were very hard to accomplish, or even impossible, not too long ago.

There are a few demo projects for you to play around with - how-old.net/, www.captionbot.ai/, www.celebslike.me/ and www.projectmurphy.net/. While all these options are fun, if you think about it, this stuff is very powerful.

A more serious project is the Seeing AI app (www.microsoft.com/en-us/seeing-ai/). This project is designed for the low vision community, basically enabling them to 'see'. The app narrates the world around you and combines a lot of Cognitive Services. It can read short texts to you, identify products by the barcode, recognize people and friends, etc. Imagine this being integrated into a powerful device like the HoloLens, the possibilities will be endless.

I hope you found this article useful and you will start integrating all this goodness into your own applications. I am very curious to see what you can come up with, so don't hesitate to reach out to me. All the technology described in this article is available to you, TODAY, so... what are you waiting for?

Smile please!!

Download the entire source code from GitHub at
bit.ly/dncm34-cognitive

Microsoft® MVP Most Valuable Professional

Gerald Versluis
Author

Gerald Versluis (@jfversluis) is a full-stack software developer and Microsoft MVP (Xamarin) from Holland. After years of experience working with Xamarin and .NET technologies, he has been involved ina number of different projects and has been building several apps. Not only does he like to code, but he is also passionate about spreading his knowledge - as well as gaining some in the bargain. Gerald involves himself in speaking, providing training sessions and writing blogs (https://blog.verslu.is) or articles in his free time. Twitter: @jfversluis Email: gerald@verslu.is Website: https://gerald.verslu.is

Thanks to Suprotim Agarwal for reviewing this article.

David Pine

FAVORITE FEATURES THROUGH THE YEARS

Anytime I get the chance to write about C#, I'm eager to do so. This time was no System. Exception!

As of this writing, C# has been around for over 17 years now, and it is safe to say it's not going anywhere. The C# language team is constantly working on new features and improving the developer experience.

In this article, join me as I walk through the various versions of C# and share my favorite features from each release. I'll demonstrate the benefits while emphasizing on practicality.

C# Version 1.0

Version 1.0 (ISO-1) of C# was really barebones, there was nothing terribly exciting and it lacked many of the things that developers love about the language today. There is one particular feature that does come to mind however, that I would consider my favorite - **Implicit and Explicit Interface Implementations.**

Interfaces are used all the time and are still prevalent in modern C# today. Consider the following `IDateProvider` interface for example.

```
public interface IDateProvider
{
  DateTime GetDate();
}
```

Nothing special, now imagine two implementations – the first of which is implicitly implemented as follows:

```
public class DefaultDateProvider : IDateProvider
{
  public DateTime GetDate()
  {
    return DateTime.Now;
  }
}
```

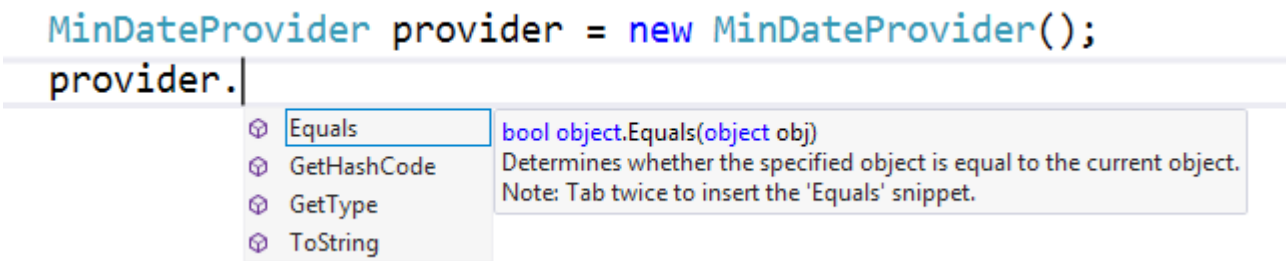
The second implementation is explicitly implemented as such:

```
public class MinDateProvider : IDateProvider
{
    DateTime IDateProvider.GetDate()
    {
        return DateTime.MinValue;
    }
}
```

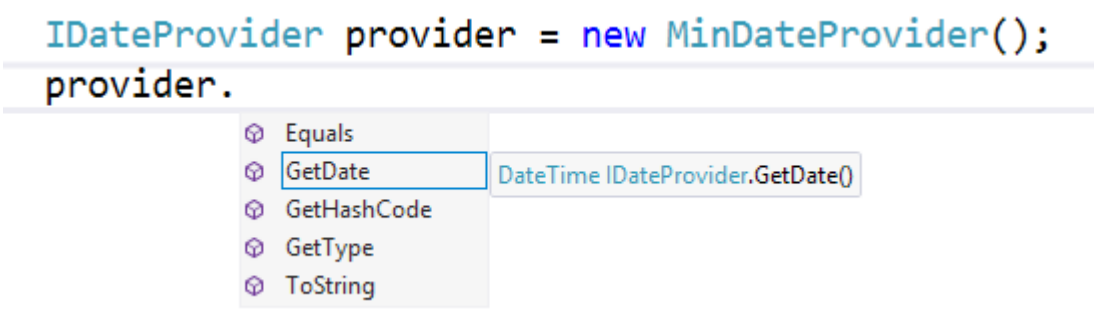
Notice how the explicit implementation omits an access modifier. Also, the method name is written as `IDateProvider.GetDate()`, which prefixes the interface name as a qualifier.

These two things make the implementation explicit.

One of the neat things about explicit interface implementation is that it enforces consumers to rely on the interface. An instance object of a class that explicitly implements an interface does not have the interface members available to it – instead the interface itself must be used.



However, when you declare it as the interface or pass this implementation as an argument that is expecting the interface – the members are available as expected.



This is particularly useful as it enforces the use of the interface. By working directly with the interface, you are not coupling your code to the underlying implementation. Likewise, explicit interface implementations handle naming or method signature ambiguity – and make it possible for a single class to implement multiple interfaces that have the same members.

Jeffery Richter warns us about explicit interface implementations in his book CLR via C#. The primary two concerns are that value type instances are boxed when cast to an interface and methods that are explicitly implemented, cannot be called by a derived type.

Keep in mind that boxing and unboxing carry computational expenses. As with all things programming, you should evaluate the use case to determine the right tool for the job.

C# Version 2.0

For reference I'll list all the features of C# 2.0 (ISO-2).

- Anonymous methods
- Covariance and contravariance
- Generics
- Iterators
- Nullable types
- Partial types

My favorite feature was a tossup between **Generics and Iterators**, I landed on Generics and here is why.

This was an extremely difficult choice for me and I ended up deciding on Generics because I believe that I use them more often than writing iterators. A lot of the SOLID programming principles are empowered by the adoption of generics in C#, likewise it helps keep code DRY. Don't get me wrong, I do write my fair share of iterators and that is a feature worth adopting in your C# today!

Let's look at Generics in more detail.

> *Generics introduce to the .NET Framework the concept of type parameters, which make it possible to design classes and methods that defer the specification of one or more types until the class or method is declared and instantiated by client code.*

Let's imagine that we have a class named `DataBag` that serves as, well... a bag of data. It might look like this:

```
public class DataBag
{
  public void Add(object data)
  {
    // omitted for brevity...
  }
}
```

At first look this seems like an awesome idea, because you can add anything in an instance of this bag of data object. But when you really think about what this means, it might be rather alarming.

Everything that is added is implicitly upcast to `System.Object`. Furthermore, if a value-type is added – boxing occurs. These are performance considerations that you should be mindful of.

Generics solve all this while also adding type-safety. Let's modify the previous example to include a type parameter `T` on the class and notice how the method signature changes too.

```
public class DataBag<T>
{
  public void Add(T data)
  {
    // omitted for brevity...
  }
}
```

Now for example, a `DataBag<DateTime>` instance will only allow the consumer to add `DateTime` instances. Type-safety, no casting or boxing...and the world is a better place.

Generic type parameters can also be constrained. Generic constraints are powerful and allow for a limited range of available type parameters, as they must adhere to the corresponding constraint. There are several ways to write generic type parameter constraints, consider the following syntax:

```
public class DataBag<T> where T : struct { /* T is value-type */ }
public class DataBag<T> where T : class { /* T is class, interface, delegate or
array */ }
public class DataBag<T> where T : new() { /* T has parameter-less .ctor() */ }
public class DataBag<T> where T : IPerson { /* T inherits IPerson */ }
public class DataBag<T> where T : BaseClass { /* T derives from BaseClass */ }
public class DataBag<T> where T : U { /* T inherits U, U is also generic type
parameter */ }
```

Multiple constraints are permitted and are comma delimited. Type parameter constraints are enforced immediately, i.e. compilation errors prevent programmer error. Consider the following constraint on our `DataBag<T>`.

```
public class DataBag<T> where T : class {
  public void Add(T value) {
    // omitted for brevity...
  }
}
```

Now, if I were to attempt to instantiate a `DataBag<DateTime>`, the C# compiler will let me know that I have done something wrong. More specifically it states:

> *The type 'DateTime' must be a reference type in order to use it as parameter 'T' in the generic type or method 'Program.DataBag<T>'*

C# Version 3.0

Here is a listing of the major features of C# 3.0.

- Anonymous types
- Auto implemented properties
- Expression trees
- Extension methods
- Lambda expression
- Query expressions

I was teetering on the edge of choosing Extension Methods over **Lambda Expressions**. However, when I think about the C# that I write today – I literally use the lambda operator more than any other C# operator in existence.

I love writing expressive C#, I cannot get enough of it.

In C# there are many opportunities to leverage lambda expressions and the lambda operator. The => lambda operator is used to separate the input on the left from the lambda body to the right.

Some developers like to think of lambda expressions as really being a less verbose way of expressing delegation invocation. The types `Action`, `Action<in T, ...>`, `Func<out TResult>`, `Func<in T, ..., out TResult>` are just pre-defined generic delegates in the `System` namespace.

Let's start with a problem that we are trying to solve and apply lambda expressions to help us write some expressive and terse C# code, yeah?!

Imagine that we have a large number of records that represent trending weather information. We may want to perform some various operations on this data, and instead of iterating through it in a typical loop, `for`, `foreach` or `while` – we can approach this differently.

```
public class WeatherData
{
  public DateTime TimeStampUtc { get; set; }
```

```
  public decimal Temperature { get; set; }
}

private IEnumerable<WeatherData> GetWeatherByZipCode(string zipCode) { /* ... */ }
```

Being that the invocation of `GetWeatherByZipCode` returns an `IEnumerable<WeatherData>` it might seem like you'd want to iterate this collection in a loop. Imagine we have a method to calculate the average temperature, and it does this work.

```
private static decimal CalculateAverageTemperature(
  IEnumerable<WeatherData> weather,
  DateTime startUtc,
  DateTime endUtc)
{
  var sumTemp = 0m;
  var total = 0;
  foreach (var weatherData in weather)
  {
    if (weatherData.TimeStampUtc > startUtc &&
      weatherData.TimeStampUtc < endUtc)
      {
         ++ total;
         sumTemp += weatherData.Temperature;
      }
    }
  return sumTemp / total;
}
```

We declare some local variable to store the sum of all the temperatures that fall within our filtered date range and a total of them, to later calculate an average. Within the iteration is a logical `if` block which checks if the weather data is within a specific date range. This could be re-written as follows:

```
private static decimal CalculateAverageTempatureLambda(
  IEnumerable<WeatherData> weather,
  DateTime startUtc,
  DateTime endUtc)
{
  return weather.Where(w => w.TimeStampUtc > startUtc &&
                            w.TimeStampUtc < endUtc)
                .Select(w => w.Temperature)
                .Average();
}
```

As you can see, this is dramatically simplified. The logical `if` block was really just a predicate, if the weather date was within range we'd continue into some additional processing – like a filter. Then we were summing the temperature, so we really just needed to project (or select) this out. We ended up with a filtered list of temperatures `IEnumerable<decimal>` that we can now simply invoke `Average` on.

The lambda expressions are used as arguments to the `Where` and `Select` extension methods on the generic `IEnumerable<T>` interface. `Where` takes a `Func<T, bool>` and `Select` takes a `Func<T, TResult>`.

C# Version 4.0

C# 4.0 was a smaller in terms of the number of major features from its previous versions releases.

- Dynamic binding
- Embedded interop types
- Generic covariant and contravariant
- Named/optional arguments

All of these features were and still are very useful. But for me it came down to named and optional arguments over covariance and contravariance in generics. Between these two, I debated which feature I use most often and which one has truly benefited me the most as a C# developer through the years.

I believe that feature to be **named and optional arguments**. It is such a simple feature, but it scores many points for being practical. I mean, who has not written a method with an overload or an optional parameter?

When you write an optional parameter, you must provide a default value for it. If your parameter is a value-type this must be a literal or constant value, or you can use the `default` keyword. Likewise, you could declare the value-type as `Nullable` and assign it `null`. Let us imagine that we have a `Repository`, and there is a `GetData` method.

```
public class Repository
{
  public DataTable GetData(
    string storedProcedure,
    DateTime start = default(DateTime),
    DateTime? end = null,
    int? rows = 50,
    int? offSet = null)
  {
    // omitted for brevity...
  }
}
```

As we can see, this method's parameter list is rather long – but there are several assignments. This indicates that these values are optional. As such, the caller can omit them and the default value will be used. As you might assume, we can invoke this by only providing the `storedProcedure` name.

```
var repo = new Repository();
var sales = repo.GetData("sp_GetHistoricalSales");
```

Now that we have familiarized ourselves with the optional arguments feature and how those work, let's use some named arguments here. Take our example from above, and imagine that we only want our data table to return 100 rows instead of the default 50. We could change our invocation to include a named argument and pass the desired overridden value.

```
var repo = new Repository();
var sales = repo.GetData("sp_GetHistoricalSales", rows: 100);
```

C# Version 5.0

Like C# version 4.0, there were not a lot of features packed into C# version 5.0 – but of the two features one of them was massive.

- Async / Await
- Caller info attributes

When C# 5.0 shipped, it literally altered the way that C# developers wrote asynchronous code. Still today there is much confusion about it and I'm here to reassure you that it's much simpler than most might think. This was a major leap forward for C# – and it introduced a language-level asynchronous model which greatly empowers developers to write "async" code that looks and feels synchronous (or at the very least serial).

Asynchronous programming is very powerful when dealing with I/O bound workloads such as interacting with a database, network, file system, etc. Asynchronous programming helps with throughput by utilizing a non-blocking approach. This approach instead uses suspension points and corresponding continuations in a transparent async state machine. Likewise, if you have heavy workloads for CPU bound computations, you might want to consider doing this work asynchronously. This will help with the user experience as the UI thread will not be blocked and is instead free to respond to other UI interactions.

Editorial Note: *Here's a good tut on some Best Practices on Asynchronous Programming in C# using Async Await* www.dotnetcurry.com/csharp/1307/async-await-asynchronous-programming-examples.

With C# 5.0, asynchronous programming was simplified when the language added two new keywords, `async` and `await`. These keywords worked on `Task` and `Task<T>` types. The table below will serve as a point of reference:

Keyword	Description
async	Modifies a method, lambda expression or anonymous method indicating that it is asynchronous.
await	Creates a suspension point in the execution of the method retuning control to the caller until the awaited task finishes successfully or errors out.

`Task` and `Task<T>` classes represent asynchronous operations. The operations can either return a value via `Task<T>` or return void via `Task`. When you modify a Task returning method with the `async` keyword, it enables the method body to use the `await` keyword. When the `await` keyword is evaluated, the control flow is returned back to the caller – and execution is suspended at that point in the method. When the awaited operation completes, execution is then resumed at that same point. Time for some code!

```
class IOBoundAsyncExample {
  // Yes, this is the internet Chuck Norris Database of jokes!
  private const string Url = "http://api.icndb.com/jokes/random?limitTo=[nerdy]";

  internal async Task<string> GetJokeAsync()
  {
    using (var client = new HttpClient())
    {
      var response = await client.GetStringAsync(Url);
      var result = JsonConvert.DeserializeObject<Result>(response);

      return result.Value.Joke;
```

```
    }
  }
}

public class Result
{
  [JsonProperty(“type”)] public string Type { get; set; }
  [JsonProperty(“value”)] public Value Value { get; set; }
}

public class Value
{
  [JsonProperty(“id”)] public int Id { get; set; }
  [JsonProperty(“joke”)] public string Joke { get; set; }
}
```

We define a simple class with a single method in it named `GetJokeAsync`, it is at this point we discover that laughter is imminent. The method is `Task<string>` returning and this signifies to the reader that our `GetJokeAsync` method will eventually give you a string – or possibly error out.

The method is modified with the `async` keyword, which enables the use of the `await` keyword. We instantiate and use an `HttpClient` object. We then invoke the `GetStringAsync` function, which takes a string url and returns a `Task<string>`. We await the `Task<string>` returned from the `GetStringAsync` invocation.

When the response is ready a continuation occurs and control resumes from where we were once suspended. We then deserialize the JSON into our `Result` class instance and return the `Joke` property.

A Few of My Favorite Outputs

- Chuck Norris can unit test entire applications with a single assert.
- Chuck Norris can compile syntax errors.
- Project managers never ask Chuck Norris for estimations… ever.

Hilarity ensues! And we learned about C# 5’s amazing asynchronous programming model.

C# Version 6.0

There were lots of great advancements with the introduction of C# 6.0 and it was hard to choose my favorite feature.

- Dictionary initializer
- Exception filters
- Expression bodied members
- `nameof` operator
- Null propagator
- Property initializers
- Static imports

- String interpolation

I narrowed it down to three standout features: String interpolation, Null propagator and `nameof` operator. While the `nameof` operator is awesome and I literally use it nearly every single time I'm writing code, the other two features are more impactful. That left me to decide between string interpolation and null propagation, which was rather difficult. I decided I liked string interpolation the best and here is why.

Null propagation is great and it allows me to write less verbose code, but it doesn't necessarily prevent bugs in my code. However, with string interpolation, runtime bugs can be prevented – and that is a win in my book. String interpolation syntax in C# is enabled when you start a string literal with the `$` symbol. This instructs the C# compiler that you intend to interpolate this string with various C# variables, logic or expressions. This is a major upgrade from manual string concatenation and even the `string.Format` method. Consider the following:

```
class Person {
  public string FirstName { get; set; }
  public string LastName { get; set; }

  public override string ToString()
    => string.Format("{0} {1}", FirstName);
}
```

We have a simple `Person` class with two name properties, for their first and last name. We override the `ToString` method and use `string.Format`. The issue is that while this compiles, it is error prone as the developer clearly intended to have the last name also part of the resulting string – as evident with the `"{0} {1}"` argument. But they failed to pass in the `LastName`. Likewise, the developer could have just as easily swapped the names or supplied both name arguments correctly but messed up the format literal to only include the first index, etc… now consider this with string interpolation.

```
class Person {
  public string FirstName { get; set; } = "David";
  public string LastName { get; set; } = "Pine";
  public DateTime DateOfBirth { get; set; } = new DateTime(1984, 7, 7);

  public override string ToString()
    => $"{FirstName} {LastName} (Born {DateOfBirth:MMMM dd, yyyy})";
}
```

I took the liberty of adding a `DateOfBirth` property and some default property values. Additionally, we are now using string interpolation in our override of the `ToString` method. As a developer it would be much more difficult to make any of the aforementioned mistakes. Finally, I can also do formatting within the interpolating expressions themselves. Take notice of the third interpolation, the `DateOfBirth` is a `DateTime` – as such we can use all the standard formatting that you're accustomed to already. Simply use the : operator to separate the variable and the format.

Example Output

- David Pine (Born July 7, 1984)

Editorial Note: *For a detailed tut on the new features of C# 6.0, read* www.dotnetcurry.com/csharp/1042/csharp-6-new-features

C# Version 7.0

From all of the features packed into C# 7.0.

- Expanded expression bodied members
- Local functions
- Out variables
- Pattern matching
- Ref locals and returns
- Tuples and deconstruction

I ended up debating between Pattern Matching, Tuples and `out` Variables. I ended up choosing `out` Variables and here's why.

Pattern Matching is great but I really don't find myself using it that often, at least not yet. Maybe I'll use it more in the future, but for all of the C# code that I've written thus far, there aren't too many places where I'd leverage this. Again, it's an amazing feature and I do see a place for it – just not my favorite of C# 7.0.

Tuples are a great addition as well. Tuples serve such an important part of the language and becoming a first class citizen is awesome. I would say that "gone are the days of `.Item1`, `.Item2`, `.Item3`, etc... but that isn't necessarily true. Deserialization loses the tuple literal names making this less public API worthy.

I'm also not a fan of the fact that the `ValueTuple` type is mutable. I just do not understand that design decision. I hope that someone can explain it to me, but it feels kind of like an oversight. Thus, I landed on the `out` Variables feature.

The try-parse pattern has been around since C# version 1.0 on various value-types. The pattern is as follows:

```
public boolean TryParse(string value, out DateTime date)
{
  // omitted for brevity...
}
```

The function returned a `boolean`, indicating whether the given string `value` was able to be parsed or not. When true the parsed value was assigned to the resulting out parameter `date`. It was consumed as follows: DateTime date;

```
if (DateTime.TryParse(someDateString, out date))
{
  // date is now the parsed value
}
else
{
  // date is DateTime.MinValue, the default value
}
```

This pattern is useful, however somewhat a nuisance. Sometimes developers take the same course of action regardless of whether the parse was successful or not. Sometimes it's fine to use a default value. The `out` variable in C# 7.0 makes this a lot more compound and in my opinion less complex.

Consider the following:

```
if (DateTime.TryParse(someDateString, out var date))
{
  // date is now the parsed value
}
else
{
  // date is DateTime.MinValue, the default value
}
```

Now we removed the outer declaration atop the `if` block and we in lined the declaration as part of the argument itself. It is legal to use `var` as the type is already known. Finally, the scope of the date variable hasn't changed. It leaks from its inline declaration back out to the top of the `if` block.

You might be asking yourself, "why would this be one of his favorite features?"...It kind of feels like not a whole lot has really changed.

But this changes everything! It allows our C# to be more expressive. Everyone loves extensions methods, right - consider the following:

```
public static class StringExtensions {
  private delegate bool TryParseDelegate<T>(string s, out T result);

  private static T To<T>(string value, TryParseDelegate<T> parse)
    => parse(value, out T result) ? result : default;

  public static int ToInt32(this string value)
    => To<int>(value, int.TryParse);

  public static DateTime ToDateTime(this string value)
    => To<DateTime>(value, DateTime.TryParse);

  public static IPAddress ToIPAddress(this string value)
    => To<IPAddress>(value, IPAddress.TryParse);

  public static TimeSpan ToTimeSpan(this string value)
    => To<TimeSpan>(value, TimeSpan.TryParse);
}
```

This extension method class is terse, expressive and powerful. After defining a private delegate that follows the try-parse pattern we can write a generic compound `To<T>` function, that takes a generic type argument, the string value to parse and the `TryParseDelegate<T>`. Now we can safely rely on these extension methods, consider the following:

```
public class Program
{
  public static void Main(string[] args)
  {
    var str =
    string.Join(
       "",
```

```
        new[] { “James”, “Bond”, “ +7 “ }.Select(s => s.ToInt32()));

    Console.WriteLine(str); // prints “007”
  }
}
```

Editorial Note: To get an overview of all the new features of C# 7, check this tutorial www.dotnetcurry.com/csharp/1286/csharp-7-new-expected-features

Conclusion

This article was rather challenging for me personally. I love so many features of C# that it was really difficult to narrow down just one favorite for each release.

Each newer version of C# is packed full of powerful and impactful features. The C# language team has been innovating in countless ways – one of which is the introduction of point releases. At the time of writing, C# 7.1 and 7.2 have officially shipped. As C# developers we are living in an exciting time for the language! Sorting through all these features was rather insightful for me however; as it helped to shed some light on what is practical and most impactful with my day-to-day development. As always, strive to be a pragmatic developer! Not every feature that is available in a language is necessary for the task at hand, but it is important to know what is available to you.

As we look forward to the proposals and prototypes of C# 8, I'm excited for the future of C#. It seems promising indeed and the language is actively attempting to alleviate “the billion dollar mistake”.

Resources

- https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-version-history
- https://en.wikipedia.org/wiki/C_Sharp_(programming_language)
- https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)
- https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/compiler-options/langversion-compiler-option
- https://www.wintellect.com/clr-via-c-by-jeffrey-richter/

Microsoft® MVP Most Valuable Professional

David Pine

Author

David Pine is a Technical Evangelist and Microsoft MVP working at Centare in Wisconsin. David loves knowledge sharing with the technical community and speaks regionally at meetups, user groups, and technical conferences. David is passionate about sharing his thoughts through writing as well and actively maintains a blog at davidpine.net. David's posts have been featured on ASP.NET, MSDN Web-Dev, MSDN .NET and DotNetCurry. David loves contributing to open-source projects and stackoverflow.com as another means of giving back to the community. David sat on the technical board and served as one of the primary organizers of MKE DOT NET for three years. When David isn't interacting with a keyboard, you can find him spending time with his wife and their three sons, Lyric, Londyn and Lennyx. Follow David on Twitter at @davidpine7.

Thanks to Yacoub Massad for reviewing this article.

A MAGAZINE FOR .NET AND JAVASCRIPT DEVS

- ☑ AGILE
- ☑ ASP.NET
- ☑ MVC, WEB API
- ☑ ANGULAR.JS
- ☑ NODE.JS
- ☑ AZURE
- ☑ VISUAL STUDIO
- ☑ .NET
- ☑ C#, WPF

We've got it all!

100K PLUS READERS

300 PLUS AWESOME ARTICLES

34 EDITIONS

FREE SUBSCRIPTION USING YOUR EMAIL

EVERY ISSUE DELIVERED

RIGHT TO YOUR INBOX

NO SPAM POLICY

SUBSCRIBE TODAY!

Francesco Abbruzzese

FAST AZURE COSMOS DB DEVELOPMENT WITH THE DOCUMENTDB PACKAGE

*The **Mvc Controls Toolkit Core** is a free collection of tools for building professional level ASP.NET Core MVC web applications that encompasses all layers a professional MVC application is composed of: data and business layer tools, controller tools and View tools.*

In this 'how to' article, I will demonstrate how to use the MvcControlsToolkit.Business.DocumentDB Nuget package to quickly build a simple but complete ASP.NET MVC core application based on CosmosDB.

AZURE COSMOS DB

Image Courtesy: www.azure.microsoft.com

What is Cosmos DB?

Cosmos DB is a distributed database available on Azure. Data can be replicated in several geographical areas, and can benefit from the redundancy and scalability features, typically offered by cloud services.

Cosmos DB is not a relational database. Its data elements are called "Documents" that may be represented as JSON objects. These documents have no predefined structure: each document may have different properties, and may contain nested JSON objects and arrays.

Documents are grouped into "Collections" that in general, are sets of heterogeneous documents. Finally, collections are grouped into databases. All the document properties are automatically indexed, but it is possible to perform some tuning on the way indexes are built, and on how the properties are to be indexed.

Cosmos DB not only offers a different data model, but is also cheaper than the SQL Azure Database, that is a distributed relational database,

In fact, Cosmos DB offers somewhat different data models, namely:

1. The strict JSON Document/Collection data model that we just discussed

2. A Graph data model, where documents may be either vertices or labels of an overall graph. This model is useful in representing the semantic relations that connects real-life entities, similar to relationships in a social application. This model may be queried with the Gremlin language.

3. Two different Table/Rows/Columns based data models, namely the old Azure tables and Cassandra that is based on the Apache Cassandra Api.

In this article, I will focus on the strict JSON Document/Collections data model that may be queried either with the same API of the old DocumentDB Azure service, or with the same API of the MongoDB database.

We will focus on the DocumentDB interface that is based on a subset of SQL, and may be queried with LINQ too. In order to use LINQ, JSON objects are mapped to .NET classes with the help of JSON.NET, as we will see later in this article.

Cosmos DB DocumentDB Interface Limitations

Cosmos DB DocumentDB interface has some limitations that are quite relevant in most practical applications.

First of all, documents can't be partially updated, but can only be replaced completely.

Thus, for instance, suppose we would like to change the "Name" property of a `Person` complex object that might also have other properties like "Surname", "Spouse". These properties might further contain a nested Person object, and "Children" that might contain a nested enumeration of Person objects.

As you can see, we are forced to read the entire object modifying it and finally replacing the previous version, with the fully modified object. This limitation comes from the design choice of avoiding document

"locks" to improve performance.

Another limitation is the ability to perform "select" operations on nested enumerations.

Thus, if we consider the previous `Person` document example again in a LINQ query, we may project the root `Person` object in a DTO class that contains just some of the Person properties, but then we are forced to take the whole `Person` object, contained in its Children property.

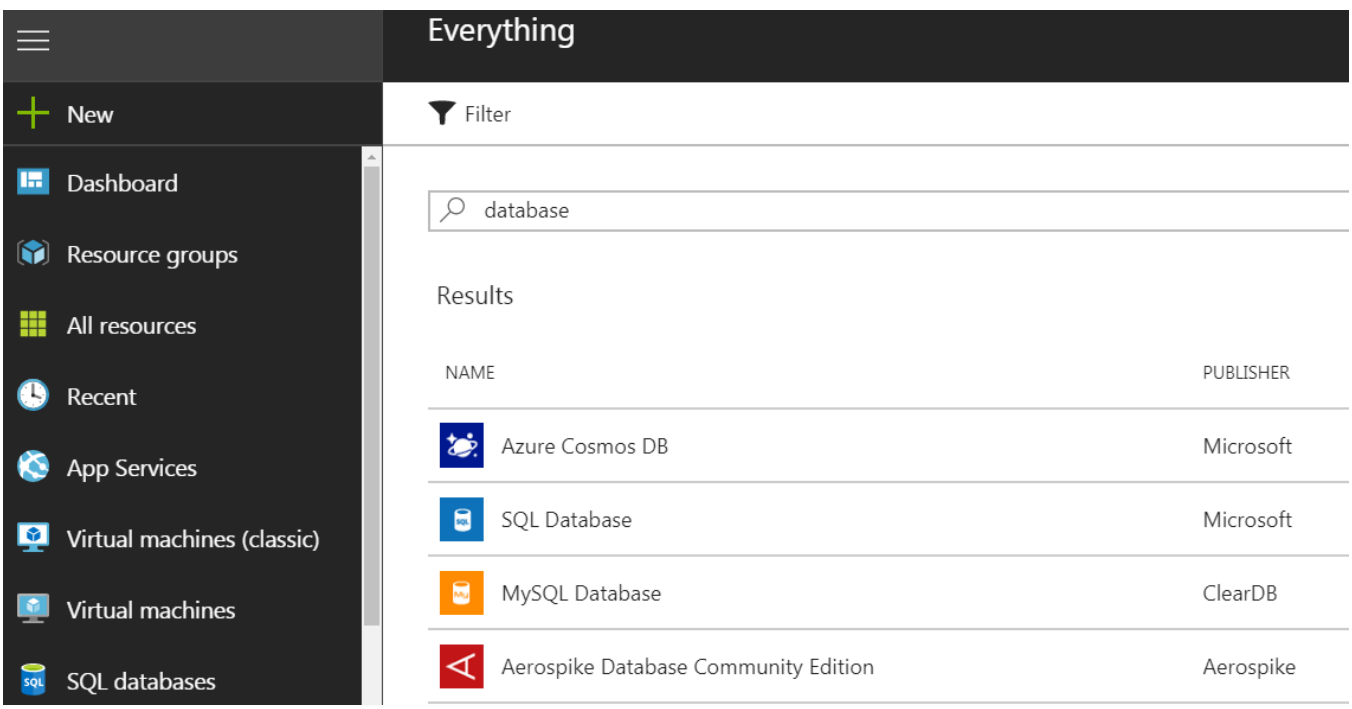
This limitation is quite "troubling" because either we export the whole `Person` object from the data layer to the presentation layer, thus breaking the separation between the two layers; or we are forced to define a chain of intermediate objects and perform a chain of copy operation to get an object tree that doesn't contain data layer objects.

Another shortcoming of performance constraints is the absence of support for the LINQ "skip" operation that is needed for paging efficiently query results.

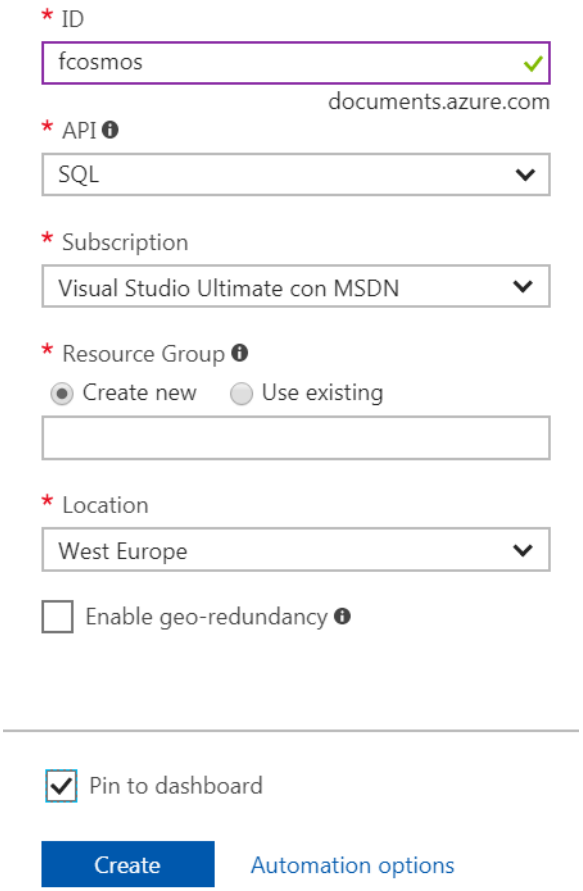
A minor, but relevant limitation is the support for just one property in sorting. Thus, for instance, it is not possible to sort Person documents first by Surname and then by Name, but one has to choose between either Surname or Name.

Creating an Azure Cosmos DB account

If you already have an Azure account, login to your account. Alternatively create a free trial account following the instructions on this page. Once logged in, go to the portal (by clicking the portal link in the top of the page). Then click the "Create a resource button" ("+ New" in the top of the page) and search "database".



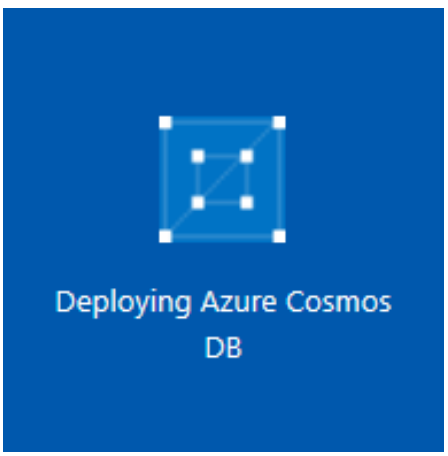
Now click on "Azure Cosmos DB". In the info's page that appears, click the "create" button at the end of the page. You should see a page similar to this one:



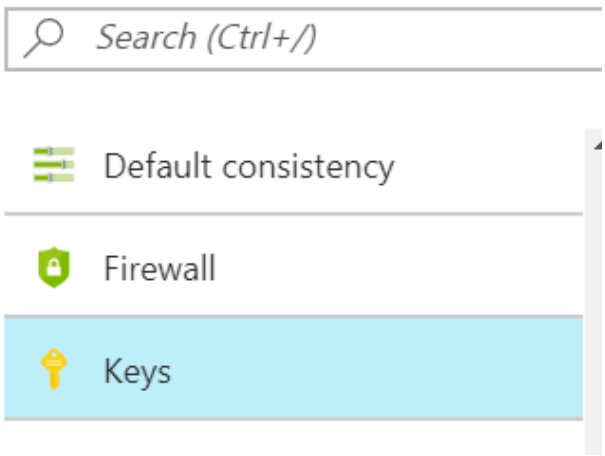
Choose an ID for your Cosmos DB account (different from the one shown in the image), and select the SQL API. You may create a new resource group or you may use an existing one.

Then select a location next to your country, and check the "Pin to dashboard" so your Cosmos DB will appear in your Azure dashboard. Avoid checking "Enable geo-redundancy" since this might quickly diminish your free trial credit. You may add geo-redundancy at a later time. Finally click "Create".

The account creation takes a few minutes. During account creation, the portal displays the "Deploying Azure Cosmos DB tile" on the right side, you may need to scroll right on your dashboard to see the tile. There is also a progress bar displayed near the top of the screen. You can watch either to track progress.



Once deployed, click on the newly created resource, and then find and click the "Keys" menu item.



This will show a page containing connection information. You need the Cosmos DB URI and the PRIMARY KEY. Store them somewhere, or simply keep that page opened.

The Mvc Controls Toolkit Core DocumentDB package

MvcControlsToolkit.Business.DocumentDB Nuget package is part of the Mvc Controls Toolkit Core tools. Though this article does not require a priori knowledge of the Mvc Controls Toolkit, a complete introduction to the Mvc Controls Toolkit may be found at www.dotnetcurry.com/aspnet-mvc/1376/full-stack-development-using-aspnet-mvc-core-toolkit .

All DocumentDB specific details are taken care of by the `Update`, `Add`, `Delete`, `UpdateList`, `GetById`, and `GetPage` methods of a DocumentDB specific implementation of an ICRUDRepository interface.

All repository methods operate on DTO/ViewModels and take care of mapping DTO/ViewModels from/to the actual database objects.

Mappings are recursive and involve objects and collections recursively contained in the DTO/ViewModels. Mappings are based on naming conventions and specifications furnished through LINQ expressions.

The naming conventions automatically handle object flattening and de-flattening, so LINQ expression are usually very simple since they don't need to specify how each property is mapped, but just: 1) exceptions to the name conventions, and 2) which part of the object tree must be mapped. In a few words `GetById` and `GetPage` specify only the subtree of each DocumentDB entry needed by the application into DTO/ViewModels.

The "DocumentDBCRUDRepository" class of the MvcControlsToolkit.Business.DocumentDB Nuget package implements the "ICRUDRepository" interface that in the Mvc Controls Toolkit, defines an extensible standard for connecting controllers with the DB layer. It contains a few common update and query operations expressed directly in terms of DTO/ViewModels, so that data objects are hidden to the presentation layer.

All the update/add/delete and query methods of "ICRUDRepository" will be demonstrated later in this article.

All "DocumentDBCRUDRepository" updates are stacked till the "async SaveChanges" "ICRUDRepository" method is called, then all operations are executed in parallel to maximize performance. All failed operations are reported by the "DocumentsUpdateException<M>" aggregated exceptions and may be retried by passing this exception to the "RetryChanges" method.

Overcoming the Limitations of DocumentDB

"DocumentDBCRUDRepository" also contains some utility methods for building complex LINQ queries overcoming some DocumentDB limitations, namely:

1. It allows partial updates of documents, that are not allowed natively by the DocumentDB interface (only a full document replace is allowed). Partial updates are achieved by automatically retrieving the full document and by copying all changes contained in the DTOs/ViewModels tree on it. Our simulated partial update supports optimistic concurrency. Concurrency is discussed in a dedicated section.

2. It allows projection of nested collections into DTOs/ViewModels collections. Since SELECT operation on nested collections are not allowed by the DocumentDB SQL interface, DocumentDBCRUDRepository first computes all the properties to retrieve from the database, in order to build an efficient query, and then project the results on the DTOs/ViewModels with efficient in-memory projections. Performance is achieved by compiling all projection operations at the start of the program.

3. It allows classical paging of results by simulating the skip operation that is not allowed by the DocumentDB SQL interface. More specifically, it retrieves all document keys up to the current page, takes just the needed keys and then performs a query to retrieve all documents associated to those keys.

In the remainder of the article, I will demonstrate the following:

- how to create a CosmosDB database,
- how to configure your ASP.NET Core project for the usage of MvcControlsToolkit.Business.DocumentDB,
- how to define projections from/to DTO classes,
- how to use the ICRUDRepository methods of the DocumentDBCRUDRepository, and finally
- how to build some Select Many based queries with the DocumentDBCRUDRepository LINQ helper methods.

Setup the project

Open Visual Studio 2017 and select File > New > Project. Then select ASP.NET Core Web Application. Name the project "CosmosDBUtilitiesExample". In the popup window that appears, select ".NET Core" as the framework, and "ASP.NET Core 2.0" as the ASP.NET version. Finally, select MVC application, and "Individual accounts" as authentication (as we need users to filter documents according to the logged user).

Now run the project and register a user with the e-mail: frank@fake.com (there is no email verification), we need this user name to filter our data. When hitting the Register button, you will be asked to apply all pending database migrations. Do it, and refresh the page.

We are ready to install our utilities package. Right click on the dependencies node in the solution explorer

and open the Nuget package installer. Find and install the last version of MvcControlsToolkit.Business.DocumentDB.

Setup the business layer

As a first step, we must prepare the code that builds our CosmosDB database, and our CosmosDB collections (we use a single collection).

Under the "Models" folder, add a new folder called "CosmosDB" for our CosmosDB data models. Then, under this new folder, add a new class called "Person" with the following code:

```
public class Person
{
  [JsonProperty(PropertyName = "name")]
  public string Name{ get; set; }
  [JsonProperty(PropertyName = "surname")]
  public string Surname { get; set; }
  [JsonProperty(PropertyName = "id")]
  public string Id { get; set; }
}
```

Add all the `using` clauses as suggested by Visual Studio. The JsonProperty attributes specifies how to serialize/deserialize each property to communicate with CosmosDB.

Add also a ToDoItem class with the following code:

```
public class ToDoItem
{
  [JsonProperty(PropertyName = "id")]
  public string Id { get; set; }

  public string Name { get; set; }

  public string Description { get; set; }

  public string Owner { get; set; }

  [JsonProperty(PropertyName = "subItems"),
    CollectionKey("Id")]
  public IEnumerable<ToDoItem> SubItems { get; set; }

  [JsonProperty(PropertyName = "assignedTo")]
  public Person AssignedTo { get; set; }

  [JsonProperty(PropertyName = "team"),
    CollectionKey("Id")]
  public IEnumerable<Person> Team { get; set; }

  [JsonProperty(PropertyName = "isComplete")]
  public bool Completed { get; set; }
}
```

A "CollectionKey" attribute is applied to all collections, it declares which property of the collection elements to use for verifying, when two elements represent the same entity. It is also needed to match collection elements when updates are applied to an existing CosmosDB document.

The Owner encodes the connection with the permissions system. In our simple application, it is just the name of the user that created the record, who is also the only user with read/write permissions on it. A more realistic application might contain the name of the group whose users have read permission on the document, and the name of the group whose users have update permission on the document, or more complex claims about the logged user.

Now we are ready to write some code that will initialize and populate the CosmosDB database.

Under the project "Data" folder, add a new "CosmosDBDefinitions" class with the following code:

```
public static class CosmosDBDefinitions
{
  private static string accountURI = "Your URI";
  private static string accountKey = "Your Key";
  public static string DatabaseId { get; private set; } = "ToDoList";
  public static string ToDoItemsId { get; private set; } = "ToDoItems";

  public static IDocumentDBConnection GetConnection()
  {
    return new DefaultDocumentDBConnection(accountURI, accountKey, DatabaseId);
  }

  public static async Task Initialize()
  {
    var connection = GetConnection();

    await connection.Client
      .CreateDatabaseIfNotExistsAsync(
         new Database { Id = DatabaseId });

    DocumentCollection myCollection = new DocumentCollection();
      myCollection.Id = ToDoItemsId;
      myCollection.IndexingPolicy =
        new IndexingPolicy(new RangeIndex(DataType.String)
          { Precision = -1 });
      myCollection.IndexingPolicy.IndexingMode = IndexingMode.Consistent;
      var res=await connection.Client.CreateDocumentCollectionIfNotExistsAsync(
        UriFactory.CreateDatabaseUri(DatabaseId),
        myCollection);
      if (res.StatusCode== System.Net.HttpStatusCode.Created)
        await InitData(connection);
  }
  private static async Task InitData(IDocumentDBConnection connection)
  {
    List<ToDoItem> allItems = new List<ToDoItem>();
    for (int i = 0; i < 6; i++)
    {
      var curr = new ToDoItem();
      allItems.Add(curr);
      curr.Name = "Name" + i;
      curr.Description = "Description" + i;
      curr.Completed = i % 2 == 0;
      curr.Id = Guid.NewGuid().ToString();
      curr.Owner = i > 3 ? "frank@fake.com" : "John@fake.com";
      if (i > 1)
        curr.AssignedTo = new Person
        {
```

```
            Name = “Francesco”,
            Surname = “Abbruzzese”,
            Id = Guid.NewGuid().ToString()
          };
        else
          curr.AssignedTo = new Person
          {
            Name = “John”,
            Surname = “Black”,
            Id = Guid.NewGuid().ToString()
          };
          var innerlList = new List<ToDoItem>();
          for (var j = 0; j < 4; j++)
          {
            innerlList.Add(new ToDoItem
            {
              Name = “ChildrenName” + i + “_” + j,
              Description = “ChildrenDescription” + i + “_” + j,
              Id = Guid.NewGuid().ToString()
            });
          }
          curr.SubItems = innerlList;
          var team = new List<Person>();
          for (var j = 0; j < 4; j++)
          {
            team.Add(new Person
            {
              Name = “TeamMemberName” + i + “_” + j,
              Surname = “TeamMemberSurname” + i + “_” + j,
              Id = Guid.NewGuid().ToString()
            });
          }
          curr.Team = team;
        }
        foreach (var item in allItems)
        {
          await connection.Client
          .CreateDocumentAsync(
          UriFactory
         .CreateDocumentCollectionUri(
          DatabaseId, ToDoItemsId),
          item);

        }
    }
}
```

“Your URI” and “Your Key” must be substituted with the URI and primary key of your CosmosDB account. The static class defines the database name and all collection names (in our case just one collection).

The GetConnection factory method creates an IDocumentDBConnection object based on the data contained in the class.

The “Initialize” method first ensures that a database with the required name already exists, then it ensures all required collections are created (in our case just one).

The “CreateDocumentCollectionIfNotExistsAsync” returns a Created status code if the database doesn’t exist

and so it must be created. In this case the "InitData" method is called to populate the collection with test data.

During both database and collection creations, several options may be specified to configure them. Please refer to the official documentation for more details (it is enough to google the name of the methods). I changed just the indexing policy of the collection for strings to allow sorting on string fields, since sorting is possible only with the non-default "Range" policy that uses tree-based indexes instead of hash-based indexes.

The "Initialize" method must be called at program start. This may be achieved by placing its call in the "Configure" method of Startup.cs immediately after the "UseMvc" call as shown here:

```
app.UseMvc(routes =>
{
  routes.MapRoute(
    name: "default",
    template: "{controller=Home}/{action=Index}/{id?}");
});
var res=CosmosDBDefinitions.Initialize();
res.Wait();
```

Run the project. In your azure portal, you should see the newly created database, collection, and the data contained in the collection.

Coding the business/data layer

Since our application is quite simple, we will not define separate business and data layers, but a unique business/data layer.

Create a "Repository" folder in the project root, and add it a "ToDoItemsRepository" class:

```
public class ToDoItemsRepository: DocumentDBCRUDRepository<ToDoItem>
{
  private string loggedUser;
  public ToDoItemsRepository(
    IDocumentDBConnection connection,
    string userName
  ): base(connection,
    CosmosDBDefinitions.ToDoItemsId,
    m => m.Owner == userName,
    m => m.Owner == userName
    )
  {
    loggedUser = userName;
  }
  static ToDoItemsRepository()
  {

  }
}
```

Our repository inherits from the `DocumentDBCRUDRepository<ToDoItem>` generic repository and does

not do much, but the following:

1. Passing the connection object received as first constructor argument to the base constructor

2. Using the username of the logged user received as a second argument to build two filter clauses that it passes to the base constructor. The first filter clause will be applied automatically to all search operations, while the second filter will be applied to all update and delete operations; thus preventing the current user from modifying documents that he/she is not the owner of. In a real-life application, we would have passed a claim extracted from the logged user like, for instance, a "company-name", instead of a simple user name.

I have also added an empty static constructor. Later on, in the article we will add declarations specifying how to project the ToDoItem data class to and from our DTO classes.

In order to make our repository available to all controllers needing it, we must add it to the "ConfigureServices" method of the startup class:

```
services.AddSingleton<IDocumentDBConnection>(x =>
  CosmosDBDefinitions.GetConnection()
  );

services.AddTransient<ToDoItemsRepository>(x =>
  new ToDoItemsRepository(
    x.GetService<IDocumentDBConnection>(),
    x.GetService<IHttpContextAccessor>()
      .HttpContext?.User?.Identity?.Name
  ));
```

The connection object is added as a singleton since it doesn't contain status information.

The repository object, instead, is created each time an instance is required (since it is stateful).

In our example application, the repository receives the logged user name extracted from the HttpContext to handle permissions. However, in a real-life application, we should pass it one or more claims extracted from the `HttpContext?.User?.Claims` enumeration. Claims may be added when a new user is registered, or at a later time, with the `UserManager<ApplicationUser>.AddClaimAsync` method.

Coding the presentation layer

The first step in the implementation of the presentation layer is the definition of the DTO classes that are needed.

Let add a "DTOs" folder to the project root. We basically need two DTOs - a short one to be used for listing search results, and the second one for edit/detail/add purposes. The short DTO might contain just the item name and the surname of the person the "to do item" has been assigned to:

```
public class ListItemDTO
{
  public string Id { get; set; }
```

```
  public string Name { get; set; }

  public string AssignedToSurname { get; set; }
}
```

The item Id is needed to connect the list page with detail pages. We put the "Surname" of the person the "to do item" has been assigned to, in a property. The property name is the concatenation of the "AssignedTo" property name of the "ToDoItem" data model, and of the Surname property name of the Person data model. This convention for assigning names to DTO properties is called "Flattening".

In order to specify how to project data models to "ListItemDTO" objects, it is enough to add the instruction shown here to the static constructor of the "ToDoItemsRepository" class:

```
DeclareProjection
  (m =>
  new ListItemDTO
  {

  }, m => m.Id
);
```

The first argument is a lambda expression specifying that each "ToDoItem" must be transformed into a "ListItemDTO", while the second argument specifies the property that plays the role of the key in the "ListItemDTO" class.

The lambda expression doesn't specify how to fill the "ListItemDTO" properties with the "ToDoItem" class properties. This is because the "DeclareProjection" method is able to automatically create all property assignments of simple properties (the ones that do not contain collections or nested object), according to a same-name convention that also supports the flattening convention defined above.

For the detail DTO, we assume that our application manipulates just some of the properties contained in the "ToDoItem". Specifically, we discard the "Team" collection but keep the "SubItems" one, discard the "Completed" property and finally keep just "Surname" and "Id" of the person the "to do item" has been assigned to:

```
public class DetailItemDTO
{
  public string Id { get; set; }
  public string Name { get; set; }
  public string Description { get; set; }
  public string Owner { get; set; }
  public IEnumerable<SubItemDTO> SubItems { get; set; }
  public string AssignedToSurname { get; set; }
  public string AssignedToId { get; set; }
}
public class SubItemDTO
{
  public string Id { get; set; }
  public string Name { get; set; }
  public string Description { get; set; }
}
```

We used "Flattening" for the "AssignedTo" nested object properties, and used a shorter class to represent sub-items.

In this case, the projection is a little bit more complex, since we need to specify which collection to include and how to project collection items:

```
DeclareProjection
  (m =>
  new DetailItemDTO
  {
    SubItems = m.SubItems
    .Select(l => new SubItemDTO { })
  }, m => m.Id
);
```

Also in this case, most of properties are inferred automatically with the same-name convention enriched with flattening.

Since we will use the "DetailItemDTO" in the update and addition operations, in this case, we also need to specify how to project back "DetailItemDTO" properties into "ToDoItem" properties:

```
DeclareUpdateProjection<DetailItemDTO>
  (m =>
    new ToDoItem
    {
      SubItems = m.SubItems
      .Select(l => new ToDoItem { }),
      AssignedTo = m.AssignedToId == null ?
      null : new Person { }
     }
  );
```

Over here, assignments of the "Person" class properties nested in the "AssignedTo" property are inferred with the "Unflattening" conventions that is just the converse of the "Flattening" convention.

Nested objects must always be preceded by a check that verifies if the nested object must be created or not. In this case, we use the "AssignedToId" property to verify the existence of a nested person object.

The omission of checks on nested objects might result either in the creation of empty objects or in "Null object" exceptions. Null checks on nested collections are created automatically since they are standard and don't depend on the application logic, and must not be included in the projection definitions.

We don't need to write any repository method since we will use the methods inherited from "DocumentDBCRUDRepository<ToDoItem>", so we can move to the controller definition. We use the already existing "HomeController" for both list and detail pages.

We must add a constructor to get our "ToDoItemsRepository" from dependency injection:

```
ToDoItemsRepository repo;
public HomeController(ToDoItemsRepository repo)
{
  this.repo = repo;
}
```

For the Index page, we will add a new repository method:

```
public async Task<DataPage<ListItemDTO>> GetAllItems()
{
  var vm = await GetPage<ListItemDTO>
    (null,
      x => x.OrderBy(m => m.Name),
      -1, 100);
  return vm;
}
```

Then we redefine the already existing Index method:

```
public async Task<IActionResult> Index()
{

    var vm = await repo.GetAllItems();
    return View(vm);
}
```

The "GetPage" method returns a page of data projected on the DTO specified in the method's generic argument.

The first method argument is a filter expression (in our case null), while the second argument is an optional sorting ("ThenBy" is not supported by CosmosDB).

The third argument is the desired page: in our case the first data page: the minus sign prevents the calculation of the total number of results that in case of distributed databases, might be a very expensive operation.

The fourth argument is the page length: we specified 100 since we don't want to actually page results but just want to list all of them by putting a limit on the maximum number of results.

At this stage, we have also added empty placeholders for the delete, edit and create action methods:

```
public async Task<IActionResult> Delete(string id)
{
  return RedirectToAction("Index");
}
public async Task<IActionResult> Edit(string id)
{
  return View();
}
[HttpPost]
public async Task<IActionResult> Edit(DetailItemDTO  vm)
{
  return View(vm);
}
public async Task<IActionResult> Create()
{
  return View();
}
[HttpPost]
public async Task<IActionResult> Create(DetailItemDTO  vm) {
  return View(vm);
}
```

We will use them for both edit and additions of new items.

Now go to the Views/Home/Index.cshtml view and replace its content with the following code:

```
@using CosmosDBUtilitiesExample.DTOs
@using MvcControlsToolkit.Core.Business.Utilities
@model DataPage<ListItemDTO>
@{
 ViewData[“Title”] = “To do items”;
 string error = ViewData[“error”] as string;
}
<h3>@ViewData[“Title”]</h3>
@if (error != null)
{
 <div class=”alert alert-danger” role=”alert”>
  <strong>@error</strong>
 </div>
}
<table class=”table table-striped table-bordered
 table-hover table-condensed”>
  <thead>
   <tr class=”info”>
    <th></th>
    <th>Name</th>
    <th>Assigned to</th>
   </tr>
  </thead>
  <tbody class=”items-container”>
  @foreach (var item in Model.Data)
  {
   <tr>
    <td>
     <a asp-action=”Delete” asp-controller=”Home”
      asp-route-id=”@item.Id”
      class=”btn btn-xs btn-primary” title=”delete”>
       <span class=”glyphicon glyphicon-minus”></span>
        
     </a>
     <a asp-action=”Edit” asp-controller=”Home”
      asp-route-id=”@item.Id”
      class=”btn btn-xs btn-primary” title=”edit”>
       <span class=”glyphicon glyphicon-edit”></span>
        
     </a>
    </td>
    <td>@item.Name</td>
    <td>@item.AssignedToSurname</td>
   </tr>
  }
 </tbody>
</table>
<a asp-action=”Create” asp-controller=”Home”
 class=”btn btn-primary”
 title=”add new item”>
 Add new item
</a>
```

Run the project.

If you are not logged in, you should see no items at all. As soon as you log in with the frank@fake.com user, you should see the two items owned by frank@fake.com; since in the definition of the "ToDoItemsRepository", we filtered the items owned by the currently logged user.

In a real-life application, the ownership of a record would have been added to a user group and not to a single user.

You may also test the filtering capability of the "GetPage" method contained in the "GetAllItems()" repository method by adding a filter clause as shown below:

```
var vm = await repo.GetPage<ListItemDTO>
  (m => m.Name==”Name5”,
    x => x.OrderBy(m => m.Name),
    -1, 100);
```

Now we may complete the "Delete" action method:

```
public async Task<IActionResult> Delete(string id)
{
  try
  {
    if (id != null)
    {
      repo.Delete<string>(id);
      await repo.SaveChanges();
    }

  }
  catch {
    ViewData[“error”] = “an error occurred”;
  }
  return RedirectToAction(“Index”);
}
```

The code is self-explanatory. We have used the "Delete" repository method to remove the item, and the "SaveChanges" method to commit the delete operation.

In an actual application, a better error handling approach would be required. For instance, in case of network error, the operation might be retried another time before informing the user of connection problems. While in case the item to delete is not found, the user might be informed that the item is not existing or is already deleted.

Moreover, it is worth to also point out that delete operations should never be invoked with an Http "Get. This is both to prevent certain attacks, and for a clean REST semantic as discussed here.

Accordingly, in an actual application, the best option should be an AJAX "Post". Please don't test the delete operation now, but wait after we have implemented the add operation in order to avoid being left with an empty database.

The “Edit” and “Create” action methods may use the same View:

```
public async Task<IActionResult> Edit(string id) {
  try
  {
    var vm = await
    repo.GetById<DetailItemDTO, string>(id);
    return View(vm);
  }
  catch
  {
    return RedirectToAction(“Index”);
  }
}

public IActionResult Create()
{
  return View(“Edit”);
}
```

The “Create” action method returns a page with a null ViewModel, that is with empty input fields. Whereas the “Edit” action method return a page filled with data, from the “to do item” whose id has been passed.

The two generic arguments of the “GetById” repository method are respectively the DTO type, and the type of the principal key “id” passed to the method. When the user submits the form contained in this page, the “HttpPost” overload of the “Edit” method is invoked to perform the needed database modifications:

```
[HttpPost]
public async Task<IActionResult> Edit(DetailItemDTO  vm) {
  if (ModelState.IsValid)
  {
    ModelState.Clear();
    if (vm.SubItems != null)
      vm.SubItems.Where(x => x.Id == null)
      .All(x =>
      {
         x.Id = Guid.NewGuid().ToString();
         return true;
      });

      try
      {
        vm.Owner = User.Identity.Name;
        repo.Update(false, vm);

        await repo.SaveChanges();
        return RedirectToAction(“Index”);
      }
      catch
      {
        ViewData[“error”] = “an error occurred”;
      }
    }
    return View(vm);
}
public async Task<IActionResult> Create(DetailItemDTO vm) {
```

```
  if (ModelState.IsValid)
  {
    ModelState.Clear();
    if (vm.SubItems != null)
      vm.SubItems.Where(x => x.Id == null)
      .All(x =>
      {
        x.Id = Guid.NewGuid().ToString();
          return true;
      });
      try
      {
        vm.Owner = User.Identity.Name;
        vm.Id = Guid.NewGuid().ToString();

        repo.Add(false, vm);

        await repo.SaveChanges();
        return RedirectToAction("Index");
      }
      catch
      {
        ViewData["error"] = "an error occurred";
      }
    }
    return View("Edit", vm);
}
```

In the "Create" method, we will create a fresh principal key, and then perform an addition operation; while in the "Edit" method, the operation is an update. In both cases we call "SaveChanges" to commit the operation.

A fresh new principal key is also added to all newly added subitems, by first filtering them and then modifying them.

The "false" argument passed to all repository methods declares that the DTO is not a "full data item", i.e. its properties do not define all properties of the database data item. Therefore, in case of an update, the modified item must be merged with the original data item instead of replacing it.

In case the argument is set to "true", all properties that are not specified in the ViewModel, are reset to their default values. At the moment, the argument is ignored for creations of new items, but it is reserved for future extensions that will enable the user to specify how to compute all missing properties when this argument is set to false.

If several operations are pending when "SaveChanges" is called, they are executed in parallel to maximize performance. Operation errors are returned in a unique "DocumentsUpdateException<M>" aggregated exception that contains the list of all specific exceptions thrown, and information on both successfully executed, as well as failed operations. It is enough to pass the "DocumentsUpdateException<M>" exception to the repository "RetryChanges" method to retry all failed operations.

The edit View

Right click on the "Home" "Edit" method to create the "Edit" View, then select the "empty without model" scaffolding option. Finally replace the scaffolded code with:

```
@using CosmosDBUtilitiesExample.DTOs
@model DetailItemDTO
@{
  ViewData[“Title”] = “To do item”;
  string error = ViewData[“error”] as string;
  List<SubItemDTO> SubItems = null;
  if(Model != null && Model.SubItems != null)
  {
   SubItems = Model.SubItems.ToList();
  }
}

<h3>@ViewData[“Title”]</h3>
@if (error != null)
{
  <div class=”alert alert-danger” role=”alert”>
  <strong>@error</strong>
  </div>
}
<form method=”post”>
  <div asp-validation-summary=”All” class=”text-danger”></div>
  <div class=”row”>
   <div class=”col-xs-5”>
    <div class=”form-group”>
     <label asp-for=”Name”
      class=”col-xs-12 control-label”></label>
     <div class=”col-xs-12”>
      <input asp-for=”Name”
       class=”form-control” />
      <input type=”hidden” asp-for=”Id”
       class=”form-control” />
      <span asp-validation-for=”Name”
       class=”text-danger”></span>
     </div>
    </div>
   </div>

   <div class=”col-xs-4”>
    <div class=”form-group”>
     <label asp-for=”AssignedToSurname”
      class=”col-xs-12 control-label”></label>
     <div class=”col-xs-12”>
      <input asp-for=”AssignedToSurname”
       class=”form-control” />
      <span asp-validation-for=”AssignedToSurname”
       class=”text-danger”></span>
     </div>
    </div>
   </div>

   <div class=”col-xs-3”>
    <div class=”form-group”>
     <label asp-for=”AssignedToId”
      class=”col-xs-12 control-label”></label>
      <div class=”col-xs-12”>
       <input asp-for=”AssignedToId”
        class=”form-control” />
        <span asp-validation-for=”AssignedToId”
         class=”text-danger”></span>
      </div>
     </div>
```

```
    </div>
   </div>

   <div class="row">
    <div class="col-xs-12">
     <div class="form-group">
      <label asp-for="Description"
       class="col-xs-12 control-label"></label>
      <div class="col-xs-12">
       <input asp-for="Description"
        class="form-control" />
       <input type="hidden" asp-for="Id"
        class="form-control" />
       <span asp-validation-for="Description"
        class="text-danger"></span>
      </div>
     </div>
    </div>

   </div>
   <h4>Subitems</h4>

   <div class="row">
    <div class="col-xs-12">
     <div class="form-group">
      <button type="submit"
        class="btn btn-primary">
       Submit
    </button>
   </div>
  </div>
 </div>
</form>
```

This is quite a standard Bootstrap based "Edit" View for the "DetailItemDTO" class. It doesn't contain any code to render the "Subitems" IEnumerable in edit mode (additions, updates, and deletes).

You may place any editable grid between the "<h4>Subitems</h4>" header and the submit button form group. For instance, you may use an Mvc Controls Toolkit grid if you configure the project for the usage of the Mvc Controls Toolkit as explained in this tutorial. You may also use Angular or Knockout.js based grids.

In the code we will just see, I propose a simple edit in-line grid implementation that relies on the ASP.NET MVC standard model binder for "IEnumerables", whose basic principles are explained in a famous Phil Haack post.

Basically, if grid items input fields have names containing consecutive indexes like "Subitems[0].Name", "Subitems[1].Name", "Subitems[2].Name", etc. it is easy to model bind the whole IEnumerable. However if there are "holes/gaps" because of some rows that have been deleted, the model binder has no easy way to determine if and when to resume counting items.

Moreover, it is very difficult to add new items with the right index from JavaScript. Therefore, a hidden field named "Subitems.Index" is added to each row containing the row index, so that the model binder receives a "Subitems.Index" field whose value is the comma-separated list of all indexes used by the grid in the right order, such as "1,2,3,4, 7,added1,added2". This way the holes problem is overcome, since, for instance, the model binder knows that after the "4"th index it must look for the "7"th index.

Moreover indexes need not to be a sequence of growing integers anymore, but they may be generic strings, so we may assign indices to newly added items with a very simple generator of unique names.

Here's our grid:

```
<table class="table table-striped table-bordered
 table-hover table-condensed">
 <thead>
  <tr class="info">
   <th>
    <button type="button" class="add-button
     btn btn-xs btn-primary"
     title="new sub-item">
      <span class="glyphicon
       glyphicon-plus">
      </span>
    </button>
   </th>
   <th>Name</th>
   <th>Description</th>
  </tr>
 </thead>
 <tbody class="items-container">
  <tr style="display:none">
   <td>
    <button type="button" class="delete-button
     btn btn-xs btn-primary"
     title="delete">
    <span class="delete-button glyphicon
     glyphicon-minus">
    </span>
   </button>
   <input class="binding-index"
    type="hidden" name="SubItems.Index" />
   </td>

   <td>
    <input type="text" name="Name"
     class="form-control" />
    <input  type="hidden" name="Id"
     class="form-control" />
   </td>

   <td>
    <input type="text" name="Description"
     class="form-control" />
   </td>
  </tr>
  @if (Model != null && Model.SubItems != null)
  {
   for (int i=0;i< Model.SubItems.Count(); i++)
   {
    <tr>
     <td>
      <button class="delete-button btn btn-xs btn-primary" title="delete">
       <span class="glyphicon glyphicon-minus">
       </span>
      </button>
      <input type="hidden" name="SubItems.Index" value="@i" />
     </td>
```

```
      <td>
       <input asp-for="@SubItems[i].Name" class="form-control" />
        <input type="hidden" asp-for="@SubItems[i].Id" class="form-control" />
      </td>
      <td>
       <input asp-for="@SubItems[i].Description" class="form-control" />
      </td>
     </tr>
    }

 }
 </tbody>
</table>
```

It is a Bootstrap based table whose first row is a row template for adding new rows (that's why it is styled as hidden). The table header contains a button to add new items, while the first column of each row contains a button to delete the row. The same column contains the "Subitems.Index" hidden fields.

Here's the JavaScript needed to operate the grid:

```
<script type="text/javascript">
(function ($) {
 var addCount = 0;
 var template = $(".items-container")
  .children().first().detach();
 $(document).on("click", ".delete-button", function (evt) {
  $(evt.target).closest("tr").remove();
 });
 $(document).on("click", ".add-button", function (evt) {
  addCount++;
  var index = "_" + addCount;
  var prefix = "SubItems[" + index + "].";
  var item = template.clone();
  item.find(".binding-index").val(index);
  $(".items-container").append(item);
  item.find("input").not(".binding-index")
   .each(function () {
     input = $(this);
     input.attr("name", prefix + input.attr("name"))
  });

  item.show();
 });
})(jQuery);
</script>
}
```

The JavaScript is inserted in-line for simplicity, but you may move it as it is, in a separate JavaScript file.

The "addCount" variable is used to generate unique indexes for newly added items. The template row is removed at program start. When the user clicks a "remove" button, the row containing that button is removed. When the add button is clicked, a new index is generated, the template row is cloned and the clone is appended to the grid. The newly created index becomes the value of the "Subitems.Index" hidden input in the cloned row. All input fields names are updated properly with the newly generated index, and finally "item.show();" makes the newly added row visible.

Run the program and test any kind of update!

Adding custom retrieval pages

What if we want to execute custom queries that are not covered by the standard retrieval methods defined in the "ICRUDRepository" interface?

"DocumentDBCRUDRepository" contains the "Table" method to start a LINQ DocumentDB query. Then you may add some LINQ, and finally you may use the "ToList", "ToSequence", and "FirstOrDefault" repository methods to project the results to your DTOs. The "ToSequence" method extracts a page of results, and a continuation token gets the next page with a similar query.

Suppose, for instance, we would like to list all sub-Items contained in all "to do items" owned by the currently logged user. It is enough to add the following method to our "ToDoItemsRepository" class:

```
public async Task<IList<SubItemDTO>> AllSubItems()
{
  var query=Table(100)
  .Where(SelectFilter)
  .SelectMany(m => m.SubItems);
  return await ToList<SubItemDTO>(query);
}
```

The first argument of the "Table" method is the required page size. It also accepts an optional partition key argument if the collection has a partition key defined on it, and if the query must be confined to a specific partition. Finally, it admits a third "continuation token" string parameter if the query must return the next page of a previous "ToSequence" based query.

"SelectFilter" and "AccessFilter" are read-only properties inherited from DocumentDBCRUDRepository that contains the two filter conditions we passed in the DocumentDBCRUDRepository base constructor.

"ToList<SubItemDTO>" also accepts an optional second argument specifying a number of items to skip, whose default value is 0. In order for "ToList<SubItemDTO>" to work properly, we must define a projection for "SubItemDTO" in the "ToDoItemsRepository" static constructor:

```
DeclareProjection
  (m =>
    new SubItemDTO
    {
    }, m => m.Id);
```

What if we want to list all team members of all "to do items" owned by the currently logged user?
First of all, we need a "PersonListDTO" DTO:

What if we want to list all team members of all "to do items" owned by the currently logged user?

First of all, we need a "PersonListDTO" DTO:

```
public class PersonListDTO
{
  public string Name { get; set; }
  public string Surname { get; set; }
}
```

In this case, the code is a little bit different:

```
public async Task<IList<PersonListDTO>> AllMembers()
{
  var query =
    Table(100)
    .Where(SelectFilter)
    .SelectMany(m => m.Team);
    return await ToList<PersonListDTO, Person>(query);
}
```

In fact, since the "IQueryable" passed to "ToList" is a IQueryable<Person> instead of a IQueryable<ToDoItem>, we must use a different overload of "ToList" that contains a second generic argument.

Also, the projection declaration in the "ToDoItemsRepository" static constructor is a little bit different:

```
DocumentDBCRUDRepository<Person>
  .DeclareProjection
    (m =>
    new PersonListDTO
    {
    }, m => m.Surname
);
```

In fact, this time we invoke the "DeclareProjection" static method of "DocumentDBCRUDRepository<Person>" instead of the one of "DocumentDBCRUDRepository<ToDoItem>".

We also need two more "HomeController" action methods:

```
public async Task<IActionResult> AllSubitems ()
{
  var members = await repo.AllMembers();
  var vm=await repo.AllSubItems();
  return View(vm);
}
public async Task<IActionResult> AllTeamMembers()
{
  var vm = await repo.AllMembers();
  return View(vm);
}
```

..and their associated Views:

```
@using CosmosDBUtilitiesExample.DTOs
@model IList<SubItemDTO>
@{
  ViewData["Title"] = "To do sub-items";
}
<h3>@ViewData["Title"]</h3>
<table class="table table-striped table-bordered
  table-hover table-condensed">
  <thead>
   <tr class="info">
    <th>Name</th>
    <th>Description</th>
   </tr>
  </thead>
  <tbody class="items-container">
```

```
    @foreach (var item in Model)
    {
     <tr>
      <td>@item.Name</td>
      <td>@item.Description</td>
     </tr>
    }
    </tbody>
</table>
```

..and

```
@using CosmosDBUtilitiesExample.DTOs
@model IList<PersonListDTO>
@{
  ViewData[“Title”] = “To do team members”;
}
<h3>@ViewData[“Title”]</h3>
<table class=”table table-striped table-bordered
 table-hover table-condensed”>
 <thead>
  <tr class=”info”>
   <th>Name</th>
   <th>Surname</th>
  </tr>
 </thead>
 <tbody class=”items-container”>
  @foreach (var item in Model)
  {
   <tr>
    <td>@item.Name</td>
    <td>@item.Surname</td>
   </tr>
  }
  </tbody>
</table>
```

Links to the newly added pages may be added in the main menu in the _Layout page:

```
<li>
  <a asp-area=”” asp-controller=”Home” asp-action=”AllSubitems”>
    Sub-items
  </a>
</li>
<li>
  <a asp-area=”” asp-controller=”Home” asp-action=”AllTeamMembers”>
    Team members
  </a>
</li>
```

Optimistic Concurrency

CosmosDB supports optimistic concurrency through the _ETag property that is added to each document, and that is changed each time the document is modified.

When the document is replaced, we may require to abort the operation and to throw an exception if the

current_ ETag value differs from the _ETag value we received before applying our modifications. The old _ETag value must be passed in the document replacement instruction as detailed here.

MvcControlsToolkit.Business.DocumentDB automatically handles _ETags. It is enough to add a new property marked as the following to the data class (in our example the Item class):

```
[JsonProperty(PropertyName = "_ETag",
   NullValueHandling = NullValueHandling.Ignore)]
     public string MyTag { get; set; }
```

If the tag value is copied and handled in the DTO, an update operation will fail if the document changes during the user receiving the document data in the user interface, and submitting its modifications.

In this case, the first user that applies its modifications succeeds, while all other users that were concurrently modifying the same document, fail, and are forced to re-edit the new changed copy of the document.

Instead, if the _ETag is just added to the data class but not to the DTO, the operation fails only if changes occur in a small interval needed by the library to retrieve the old copy of the document, and to apply all modifications returned by the user in the DTO.

This policy forces the merge of all modifications applied concurrently by all users. If a user fails, it may retry its modifications till he/she eventually succeed merging its modifications and the document.

Conclusion:

This article shows how the MvcControlsToolkit.Business.DocumentDB Nuget package may help writing layered applications based on CosmosDB, providing out-of-the-box methods for more common operations, and overcoming some limitations of the DocumentDB SQL interface, moving it closer to a full SQL implementation.

Download the entire source code from GitHub at
bit.ly/dncm34-cosmosdb

Francesco Abbruzzese

Author

Francesco Abbruzzese(@F_Abbruzzese), implements Asp.net Mvc applications, and offers consultancy services on MVC since the beginning of this technology. He is the author of the famous Mvc Controls Toolkit, and his company offers tools, and services for Asp.net Mvc. He moved from decision support systems for banks and financial institutions, to the Video Games arena, and finally started his .Net adventure with the first .Net release. He writes about .Net technologies in his blog http://www.dotnet-programming.com/

Thanks to Daniel Jimenez Garcia for reviewing this article.

Daniel Jimenez Garcia

Integration Testing

for ASP.NET Core Applications

Part II

Any testing strategy should ensure that the entire application works together as expected!

In a previous article (on Page 8), we discussed a testing strategy involving unit, integration and end-to-end methods. We discussed how Unit Tests lead you towards a better design and allow you to work, without being afraid of modifying your code.

We added unit tests to a simple example ASP.NET Core project that provided a very straightforward API to manage blog posts.

This is the second entry in a series of tutorials on testing ASP.NET Core applications.

In this article, we will use the same ASP.NET Core project and take a deeper look at Integration Tests in the context of ASP.NET Core applications, dealing with factors like the Database, Authentication or Anti-Forgery.

The code discussed through the article is available on its GitHub repo.

The Need for Integration Tests

In a previous article of the series (on Page 8), we introduced a testing strategy based on the following Testing Pyramid:

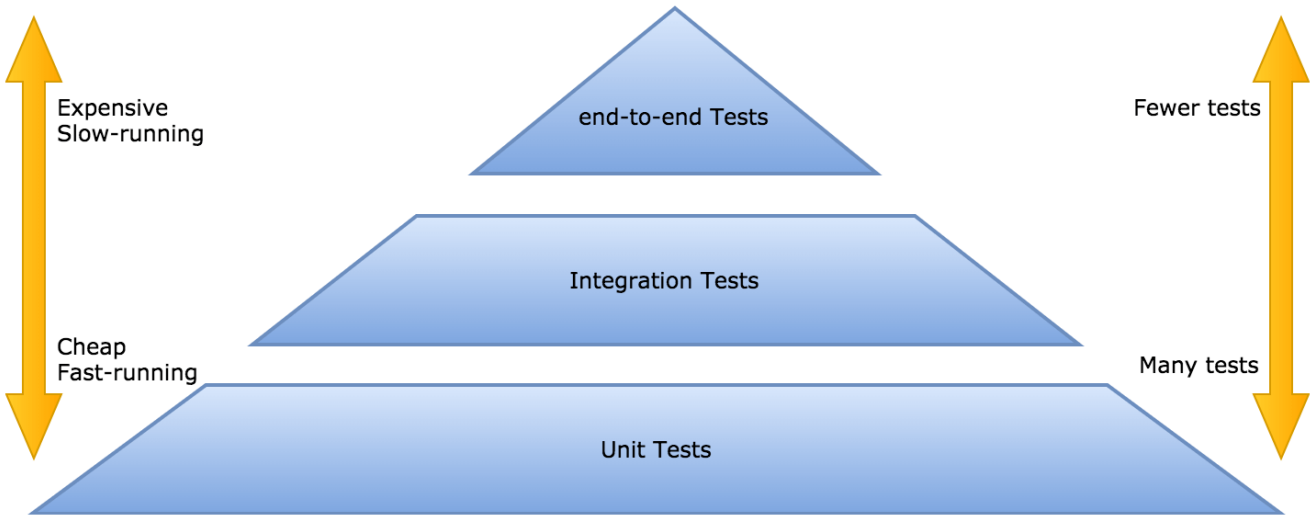


Figure 1, the Testing Pyramid

We then took a deeper look at the base of the pyramid, the Unit tests!

While valuable, I hope it became clear that having only unit tests in your strategy is a risky approach, that will leave many aspects of your application uncovered.

- Unit Tests abstract dependencies, but many parts of your application are dedicated to deal with dependencies like databases, 3rd party libraries, networking, ...
- You won't cover those with unit tests or will replace them with mocks, so you won't really know you are using them properly until higher level tests like integration and end-to-end tests can prove it so!

If you are anything like me, after going through the article dedicated to Unit Tests, you will be eager to learn how to complement those tests with higher level ones.

Now we will start writing integration tests, which will:

- Start the application using a simple Test Host (instead of a real web host like Kestrel or IIS Express)
- Initialize the database used by Entity Framework using an InMemory database instance that can be safely reset and seeded
- Send real HTTP request using the standard `HttpClient` class
- Keep using xUnit as the test framework, since every integration test is written and run as an xUnit test

Let's start again by adding a new *xUnit* Test Project named **BlogPlayground.IntegrationTest** to our solution. Once added, make sure to add a reference to **BlogPlayground** as the integration tests will need to make use of the Startup class and the Entity Framework context.

Editorial Note: Make sure to download the sample ASP.NET Core project before moving ahead.
There is one more thing to be done before we are ready for our first test. Add the NuGet package with the Test Host: **Microsoft.AspNetCore.TestHost**.

Writing Integration Tests with the Test Host

Writing a test that hits the default home index url "/" would be as straightforward as using the Test Host to write a test like the following one:

```
public class HomeTest {
  private readonly TestServer _server;
  private readonly HttpClient _client;

  public IndexTest()
  {
    _server = new TestServer(WebHost.CreateDefaultBuilder()
    .UseStartup<Startup>()
    .UseEnvironment("Development"));
    _client = _server.CreateClient();
  }

  public void Dispose()
  {
    _client.Dispose();
    _server.Dispose();
  }

  [Fact]
  public async Task Index_Get_ReturnsIndexHtmlPage()
```

```
  {
    // Act
    var response = await _client.GetAsync("/");
   // Assert
   response.EnsureSuccessStatusCode();
   var responseString = await response.Content.ReadAsStringAsync();
   Assert.Contains("<title>Home Page - BlogPlayground</title>", responseString);
  }
}
```

You can run it as any of the previous unit test files, since we are using the same xUnit test framework. Either run it from Visual Studio or from the command line with `dotnet test`.

However, your test will fail as it runs into a few problems!

The first problem is that your views will not be found because the content root taken by the Test Host is based on the **BlogPlayground.IntegrationTest project**, but it should be set as the **BlogPlayground project**. To solve this, you can add the following workaround to your setup code, or read *Accessing Views* in the docs to learn how to avoid hardcoding the path:

```
public IndexTest()
{
  var integrationTestsPath = PlatformServices.Default.Application.
  ApplicationBasePath;

  var applicationPath = Path.GetFullPath(Path.Combine(integrationTestsPath,
  "../../../../BlogPlayground"));
  _server = new TestServer(WebHost.CreateDefaultBuilder()
    .UseStartup<Startup>()
    .UseContentRoot(applicationPath)
    .UseEnvironment("Development"));
  _client = _server.CreateClient();
}
```

Once you solve the view location problem, you will hit another one compiling the views. To resolve it, you will need to update **BlogPlayground.IntegrationTest.csproj** adding the `PreserveCompilationContext` property and a new `Target` item that will copy the necessary **.deps.json** file as in the following snippet:

```
<PropertyGroup>
  <TargetFramework>netcoreapp2.0</TargetFramework>

  <IsPackable>false</IsPackable>
  <PreserveCompilationContext>true</PreserveCompilationContext>
</PropertyGroup>
…
<!--
  Work around https://github.com/NuGet/Home/issues/4412.
  MVC uses DependencyContext.Load() which looks next to a .dll
  for a .deps.json. Information isn't available elsewhere.
  Need the .deps.json file for all web site applications.
-->

<Target Name="CopyDepsFiles" AfterTargets="Build"
Condition="'$(TargetFramework)'!=''">
  <ItemGroup>
```

```
    <DepsFilePaths Include="$([System.IO.Path]::ChangeExtension('%(_
    ResolvedProjectReferencePaths.FullPath)', '.deps.json'))" />
  </ItemGroup>
  <Copy SourceFiles="%(DepsFilePaths.FullPath)" DestinationFolder="$(OutputPath)"
Condition="Exists('%(DepsFilePaths.FullPath)')" />
</Target>
```

Once you are done with these changes, you can go ahead and run the test again. You should see your first integration test passing.

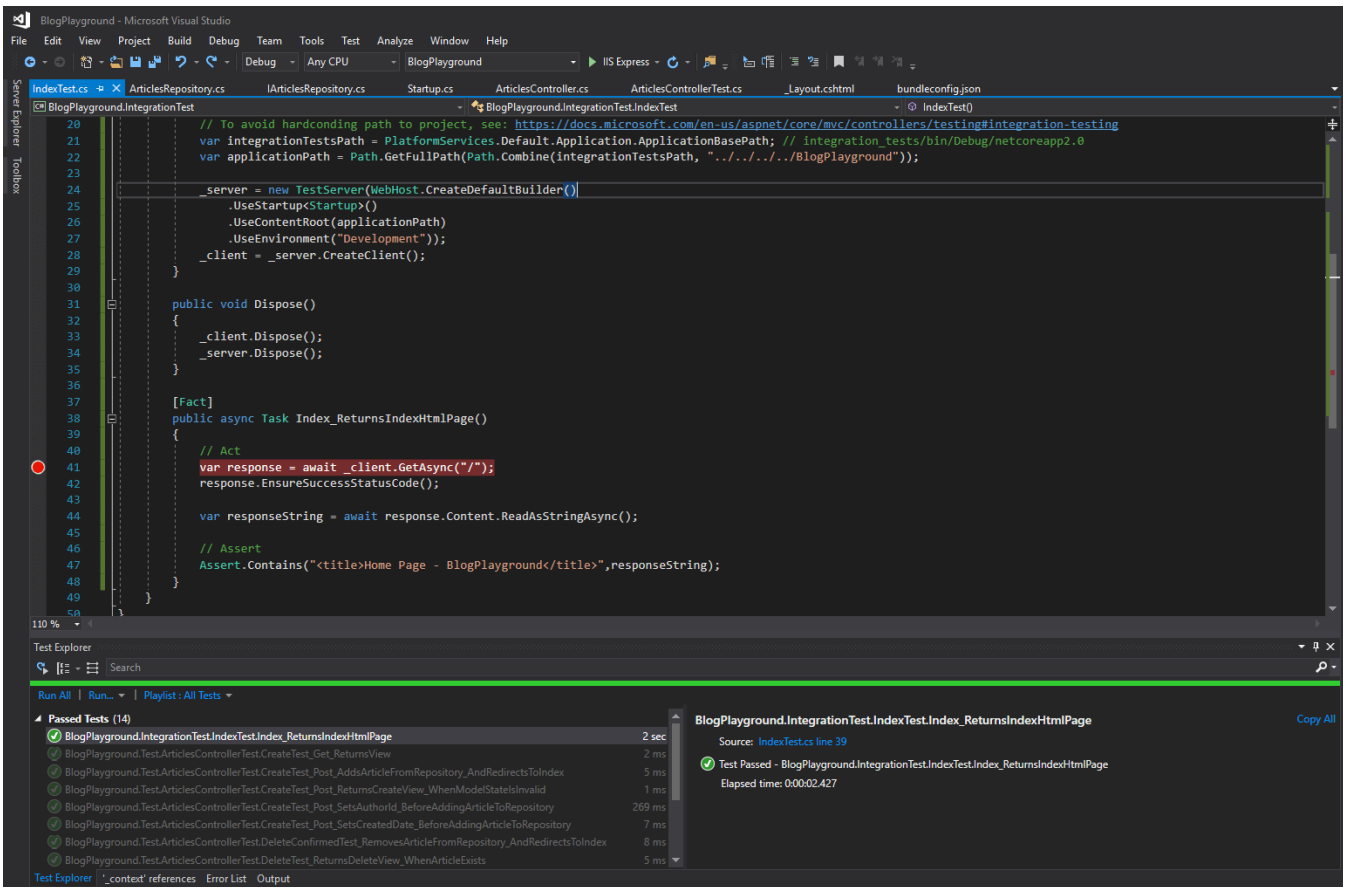


Figure 2, running the first integration test

Dealing with the Database used by Entity Framework

We have our first integration test that verifies that our app can be started and responds to the home endpoint by rendering the index html page. However, if we were to write tests for the Article endpoints, our Entity Framework `dbContext` would be using a real SQL Server DB in the same way as when we start the application.

This will make it harder for us to make certain that no state is shared between tests and to ensure the database is in the right state before starting each test. If only we had a quick and easy way of starting a

database for each test and drop it afterwards!

Luckily for us, when using Entity Framework, there is now an option to use an in-memory database that can be easily started and removed in each test (and seeded with data as we will later see).

Right now, the `ConfigureServices` of your `Startup` class looks like this:

```
services.AddDbContext<ApplicationDbContext>(options =>
  options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
```

Replacing a real SQL server connection for your Entity Framework context with an in-memory database is as simple as replacing the lines above with

```
services.AddDbContext<ApplicationDbContext>(options =>
  options.UseInMemoryDatabase("blogplayground_test_db"));
```

However, we don't want to permanently change our application, we just want to replace the database initialization for our tests without affecting the normal startup of the application. We can then move the initialization code to a virtual method in the Startup class:

```
public void ConfigureServices(IServiceCollection services)
{
  ConfigureDatabase(services);
  // Same configuration as before except for services.AddDbContext
  …
}

public virtual void ConfigureDatabase(IServiceCollection services)
{
  services.AddDbContext<ApplicationDbContext>(options =>
  options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
}
```

Then create a new subclass `TestStartup` in the IntegrationTest project where we override that method by setting an in-memory database:

```
public class TestStartup : Startup
{
  public TestStartup(IConfiguration configuration) : base(configuration)
  {
  }

  public override void ConfigureDatabase(IServiceCollection services)
  {
    services.AddDbContext<ApplicationDbContext>(options =>
    options.UseInMemoryDatabase("blogplayground_test_db"));
  }
}
```

Finally change your `IndexTest` class to use the new startup class:

```
_server = new TestServer(WebHost.CreateDefaultBuilder()
  .UseStartup<TestStartup>()
  .UseContentRoot(applicationPath)
```

```
  .UseEnvironment("Development"));
_client = _server.CreateClient();
```

Now our integration test will always use the in-memory database provided by Entity Framework Core which we can safely recreate on every test.

Note: *As per the official docs, the in-memory database is not a full relational database and some features won't be available like checking for referential integrity constraints when saving new records.*

If you hit those limitations and still want to use an in-memory database with your integration tests, you can try the in-memory mode of SQLite. If you do so, be aware you will need to manually open the SQL connection and make sure you don't close it until the end of the test. Check the official docs for further info.

Seeding the database with predefined users and blog posts

So far, we have an integration test using an in-memory database that is recreated for each test. It would be interesting to seed that database with some predefined data that will be then available in any test.

We can add another hook to the `Startup` class by making its `Configure` method *virtual*, so we can override it in the `TestStartup` class. We will then just need to add the database seed code to the overridden method.

Let's start by creating a new folder named **Data** inside the integration test and add a new file `PredefinedData.cs` where we will define the predefined articles and profiles for our tests:

```
public static class PredefinedData
{
  public static string Password = @"!Covfefe123";

  public static ApplicationUser[] Profiles = new[] {
    new ApplicationUser { Email = "tester@test.com", UserName = " tester@test.com",
    FullName = "Tester" },

    new ApplicationUser { Email = "author@test.com", UserName = " author@test.com",
    FullName = "Tester" }
  };

  public static Article[] Articles = new[] {
    new Article { ArticleId = 111, Title = "Test Article 1", Abstract = "Abstract
    1", Contents = "Contents 1", CreatedDate = DateTime.Now.Subtract(TimeSpan.
    FromMinutes(60)) },

    new Article { ArticleId = 222, Title = "Test Article 2", Abstract = "Abstract
    2", Contents = "Contents 2", CreatedDate = DateTime.Now }
  };
}
```

Next let's add a new class named `DatabaseSeeder` inside the same folder. This is where we will add the EF code that actually inserts the predefined data into the database:

```
public class DatabaseSeeder
{
  private readonly UserManager<ApplicationUser> _userManager;
  private readonly ApplicationDbContext _context;

  public DatabaseSeeder(ApplicationDbContext context, UserManager<ApplicationUser>
  userManager)
  {
    _context = context;
    _userManager = userManager;
  }

  public async Task Seed()
  {
    // Add all the predefined profiles using the predefined password
    foreach (var profile in PredefinedData.Profiles)
    {
      await _userManager.CreateAsync(profile, PredefinedData.Password);
      // Set the AuthorId navigation property
      if (profile.Email == "author@test.com")
      {
        PredefinedData.Articles.ToList().ForEach(a => a.AuthorId = profile.Id);
      }
    }

    // Add all the predefined articles
    _context.Article.AddRange(PredefinedData.Articles);
    _context.SaveChanges();
  }
}
```

Finally, let's make `Startup.Configure` a virtual method that we can override in `TestStartup` with the following definition:

```
public override void ConfigureDatabase(IServiceCollection services)
{
  // Replace default database connection with In-Memory database
  services.AddDbContext<ApplicationDbContext>(options =>
    options.UseInMemoryDatabase("blogplayground_test_db"));

  // Register the database seeder
  services.AddTransient<DatabaseSeeder>();

}

public override void Configure(IApplicationBuilder app, IHostingEnvironment env,
ILoggerFactory loggerFactory)
{
  // Perform all the configuration in the base class
  base.Configure(app, env, loggerFactory);

  // Now seed the database
  using (var serviceScope = app.ApplicationServices.
  GetRequiredService<IServiceScopeFactory>().CreateScope())
  {
    var seeder = serviceScope.ServiceProvider.GetService<DatabaseSeeder>();
    seeder.Seed();
  }
}
```

With these changes, we are now ready to start writing tests that exercise the Article endpoints.

Writing tests using the preloaded in-memory database

Let's start by adding a new integration test that will call the Article Index route. Since the controller returns a view, and the way things are right now, it would be hard to ensure the returned html contains every article from the database.

We could add a data attribute to every article list item that gets rendered on the index view, so that in our test, we can verify if the response gets a success code and an html string that contains a `data-articleid` attribute for every predefined article.

Update **Views/Articles/Index.cshtml** so that the list of articles is rendered as:

```
<ul class="col-md-8 list-unstyled article-list">
@foreach (var article in Model)
{
    <li data-articleid="@article.ArticleId">
        @Html.Partial("_ArticleSummary", article)
    </li>
}
</ul>
```

Now we can simply add another integration test class to our project with the following test:

```
public class ArticlesTest
{
  private readonly TestServer _server;
  private readonly HttpClient _client;

  public ArticlesTest()
  {
    var integrationTestsPath = PlatformServices.Default.Application.
    ApplicationBasePath;

    var applicationPath = Path.GetFullPath(Path.Combine(integrationTestsPath,
    "../../../../BlogPlayground"));

    _server = new TestServer(WebHost.CreateDefaultBuilder()
      .UseStartup<TestStartup>()
      .UseContentRoot(applicationPath)
      .UseEnvironment("Development"));
      _client = _server.CreateClient();
  }

  public void Dispose()
  {
    _client.Dispose();
    _server.Dispose();
```

```
  }

  [Fact]
  public async Task Index_Get_ReturnsIndexHtmlPage_ListingEveryArticle()
  {
    // Act
    var response = await _client.GetAsync("/Articles");

    // Assert
    response.EnsureSuccessStatusCode();

    var responseString = await response.Content.ReadAsStringAsync();
    foreach (var article in PredefinedData.Articles)
    {
      Assert.Contains($"<li data-articleid=\"{ article.ArticleId }\">",
      responseString);

    }
  }
}
```

Run the test and you should see a reassuring green result!

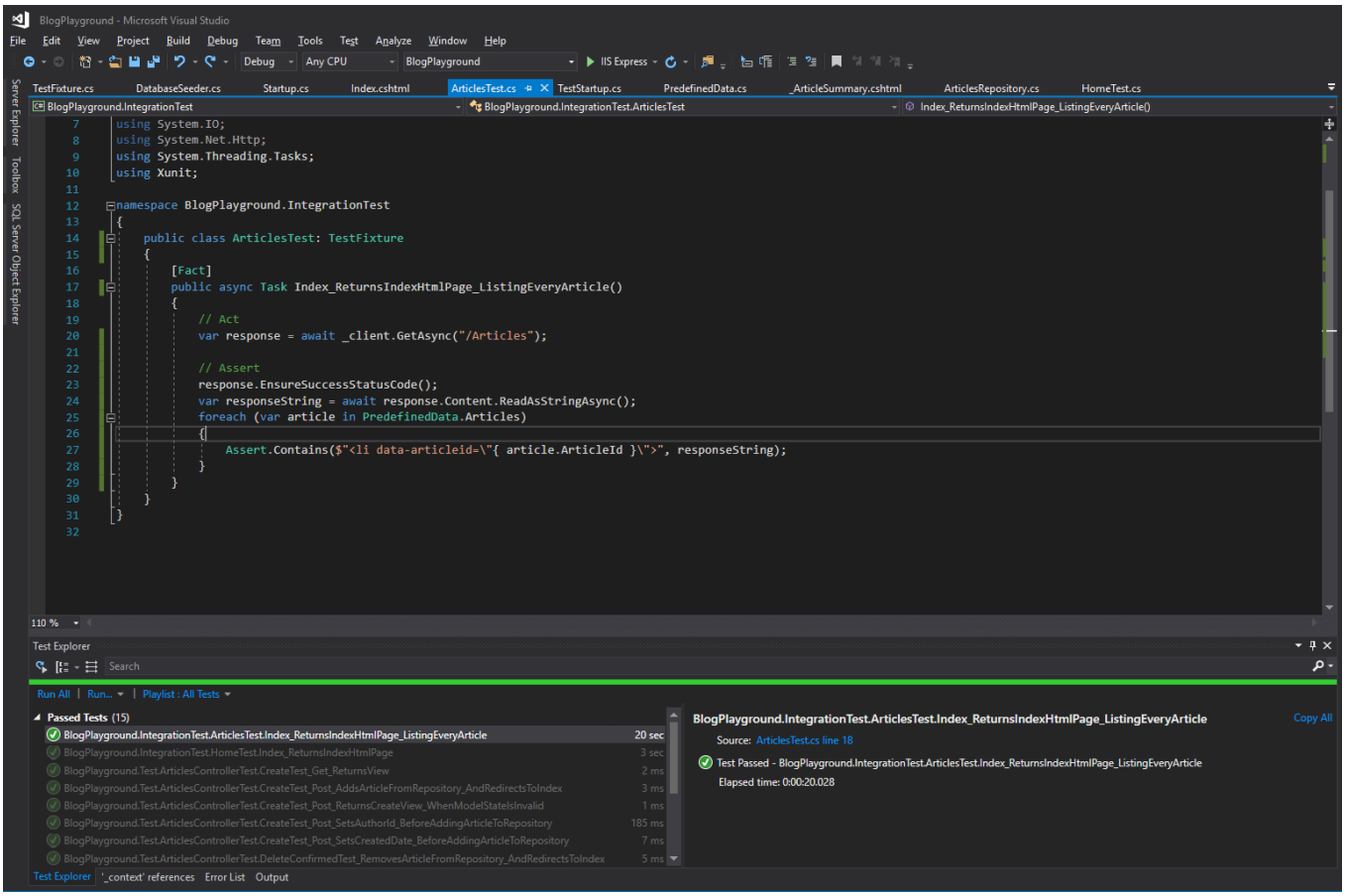


Figure 3, Running the articles index test

Before we move on and try to write tests for the POST/DELETE routes, let's refactor our integration tests slightly since we are currently repeating the initialization and cleanup code on both `HomeTest` and `ArticlesTest`. We could create the classic base `TestFixture` class:

```
public class TestFixture: IDisposable
{
  protected readonly TestServer _server;
  protected readonly HttpClient _client;

  public TestFixture()
  {
    var integrationTestsPath = PlatformServices.Default.Application.
    ApplicationBasePath;
    var applicationPath = Path.GetFullPath(Path.Combine(integrationTestsPath,
      "../../../../BlogPlayground"));

    _server = new TestServer(WebHost.CreateDefaultBuilder()
      .UseStartup<TestStartup>()
      .UseContentRoot(applicationPath)
      .UseEnvironment("Development"));
    _client = _server.CreateClient();
  }

  public void Dispose()
  {
    _client.Dispose();
    _server.Dispose();
  }
}
```

Then update the tests to inherit from it:

```
public class ArticlesTest: TestFixture
{
  [Fact]
  public async Task Index_Get_ReturnsIndexHtmlPage_ListingEveryArticle()
  {
    // Act
    var response = await _client.GetAsync("/Articles");

    // Assert
    response.EnsureSuccessStatusCode();
    var responseString = await response.Content.ReadAsStringAsync();
    foreach (var article in PredefinedData.Articles)
    {
      Assert.Contains($"<li data-articleid=\"{ article.ArticleId }\">",
      responseString);
    }
  }
}
```

Now we are ready to tackle the problems we need to solve in order to write integration tests for the methods that require an anti-forgery token and an authenticated user. Otherwise you won't be able to send a POST request to create or delete articles since those routes are decorated with the `[Authorize]` and `[ValidateAntiForgeryToken]` attributes.

Dealing with Authentication and Anti-Forgery

Our next target is the POST Create method. We should be able to write a similar test like the previous one where we POST an url-encoded form with the new article properties as in:

```
[Fact]
public async Task Create_Post_RedirectsToList_AfterCreatingArticle()
{
  // Arrange
  var formData = new Dictionary<string, string>
  {
    { “Title”, “mock title” },
    { “Abstract”, “mock abstract” },
    { “Contents”, “mock contents” }
  };

  // Act
  var response = await _client.PostAsync(“/Articles/Create”, new
  FormUrlEncodedContent(formData));

  // Assert
  Assert.Equal(HttpStatusCode.Found, response.StatusCode);
  Assert.Equal(“/Articles”, response.Headers.Location.ToString());
}
```

This piece of code might be enough in some situations.

However, since we are using the `[ValidateAntiForgeryToken]` and `[Authorize]` attributes, the request above won't succeed until we are able to include the following within the request:

- A valid authentication cookie
- A valid Anti-Forgery cookie and token in the form

There are several strategies you can follow to allow that test to succeed. The one I have implemented involves sending these requests:

- a GET request to `/Account/Login` in order to extract the anti-forgery cookie and token from the response
- followed by POST request to the `/Account/Login` using one of the predefined profiles, extracting the authentication cookie from the response

Once the required authentication and anti-forgery data is extracted, we can include them within any subsequent request we send in our tests.

Let's start by extracting the anti-forgery cookie and token using new utility methods added to the base `TestFixture` class. The following code uses the default values of the anti-forgery cookie and token form field. If you have changed them in your Startup class with `services.AddAntiforgery(opts => … )` you will need to make sure you use the right names here:

```
protected SetCookieHeaderValue _antiforgeryCookie;
protected string _antiforgeryToken;

protected static Regex AntiforgeryFormFieldRegex = new Regex(@”\<input name=””__
RequestVerificationToken”” type=””hidden”” value=””([^””]+)”” \/\>”);

protected async Task<string> EnsureAntiforgeryToken()
{
  if (_antiforgeryToken != null) return _antiforgeryToken;
```

```
  var response = await _client.GetAsync("/Account/Login");
  response.EnsureSuccessStatusCode();
  if (response.Headers.TryGetValues("Set-Cookie", out IEnumerable<string> values))
  {
    _antiforgeryCookie = SetCookieHeaderValue.ParseList(values.ToList()).
    SingleOrDefault(c => c.Name.StartsWith(".AspNetCore.AntiForgery.",
    StringComparison.InvariantCultureIgnoreCase));
  }
  Assert.NotNull(_antiforgeryCookie);
  _client.DefaultRequestHeaders.Add("Cookie", new CookieHeaderValue(_
  antiforgeryCookie.Name, _antiforgeryCookie.Value).ToString());

  var responseHtml = await response.Content.ReadAsStringAsync();
  var match = AntiforgeryFormFieldRegex.Match(responseHtml);
  _antiforgeryToken = match.Success ? match.Groups[1].Captures[0].Value : null;
  Assert.NotNull(_antiforgeryToken);

  return _antiforgeryToken;
}
```

This code simply sends a GET request to `/Account/Login` and then extracts both the anti-forgery cookie from the response headers and the token embedded as a form field in the response html.

Notice how we call `_client.DefaultRequestHeaders.Add(…)` so that the anti-forgery cookie is automatically added on any subsequent requests. (The token would need to be manually added as part of any url-encoded form data to be posted.)

Now we can write a similar utility method to ensure we login with a predefined user and capture the authentication cookie from the response. We will need to use the previous method in order to include the anti-forgery cookie and token with the request, since the `AccountController` also uses the `[ValidateAntiForgery]` attribute.

```
protected SetCookieHeaderValue _authenticationCookie;

protected async Task<Dictionary<string, string>>
EnsureAntiforgeryTokenForm(Dictionary<string, string> formData = null)
{
  if (formData == null) formData = new Dictionary<string, string>();

  formData.Add("__RequestVerificationToken", await EnsureAntiforgeryToken());
  return formData;
}

public async Task EnsureAuthenticationCookie()
{
  if (_authenticationCookie != null) return;

  var formData = await EnsureAntiforgeryTokenForm(new Dictionary<string, string>
  {
    { "Email", PredefinedData.Profiles[0].Email },
    { "Password", PredefinedData.Password }
  });

  var response = await _client.PostAsync("/Account/Login", new
  FormUrlEncodedContent(formData));
```

```
  Assert.Equal(HttpStatusCode.Redirect, response.StatusCode);

  if (response.Headers.TryGetValues("Set-Cookie", out IEnumerable<string> values))
  {
    _authenticationCookie = SetCookieHeaderValue.ParseList(values.ToList()).
    SingleOrDefault(c => c.Name.StartsWith(AUTHENTICATION_COOKIE, StringComparison.
    InvariantCultureIgnoreCase));
  }
  Assert.NotNull(_authenticationCookie);
  _client.DefaultRequestHeaders.Add("Cookie", new CookieHeaderValue(_
  authenticationCookie.Name, _authenticationCookie.Value).ToString());

  // The current pair of antiforgery cookie-token is not valid anymore
  // Since the tokens are generated based on the authenticated user!
  // We need a new token after authentication (The cookie can stay the same)

  _antiforgeryToken = null;
}
```

The code is very similar, except for the fact that we send a POST request with an url-encoded form as the contents. The form contains the username and password of one of the predefined profiles seeded by our `DatabaseSeeder` and the anti-forgery token retrieved with the previous utility.

An important point to note is that anti-forgery tokens are valid for the currently authenticated user. So, after authenticating that the previous token is no longer valid, we would need to get a fresh token as part of test methods.

Now that we have the right tooling in place, we can ensure the intended test for sending a POST request to `/Articles/Create` passes:

```
[Fact]
public async Task Create_Post_RedirectsToList_AfterCreatingArticle()
{
  // Arrange
  await EnsureAuthenticationCookie();
  var formData = await EnsureAntiforgeryTokenForm(new Dictionary<string, string>
  {
    { "Title", "mock title" },
    { "Abstract", "mock abstract" },
    { "Contents", "mock contents" }
  });

  // Act
  var response = await _client.PostAsync("/Articles/Create", new
  FormUrlEncodedContent(formData));

  // Assert
  Assert.Equal(HttpStatusCode.Found, response.StatusCode);
  Assert.Equal("/Articles", response.Headers.Location.ToString());
}
```

Notice how the test includes both `await EnsureAuthenticationToken()` and `await EnsureAntiforgeryTokenForm()` in order to include the required data within the request.

Armed with these tools, it would be easy to keep testing the other endpoints provided by the controller, like

the DeleteConfirmation:

```
[Fact]
public async Task DeleteConfirmation_RedirectsToList_AfterDeletingArticle()
{
  // Arrange
  await EnsureAuthenticationCookie();
  var formData = await EnsureAntiforgeryTokenForm();

  // Act
  var response = await _client.PostAsync($"/Articles/Delete/{PredefinedData.
  Articles[0].ArticleId}", new FormUrlEncodedContent(formData));

  // Assert
  Assert.Equal(HttpStatusCode.Found, response.StatusCode);
  Assert.Equal("/Articles", response.Headers.Location.ToString());
}
```

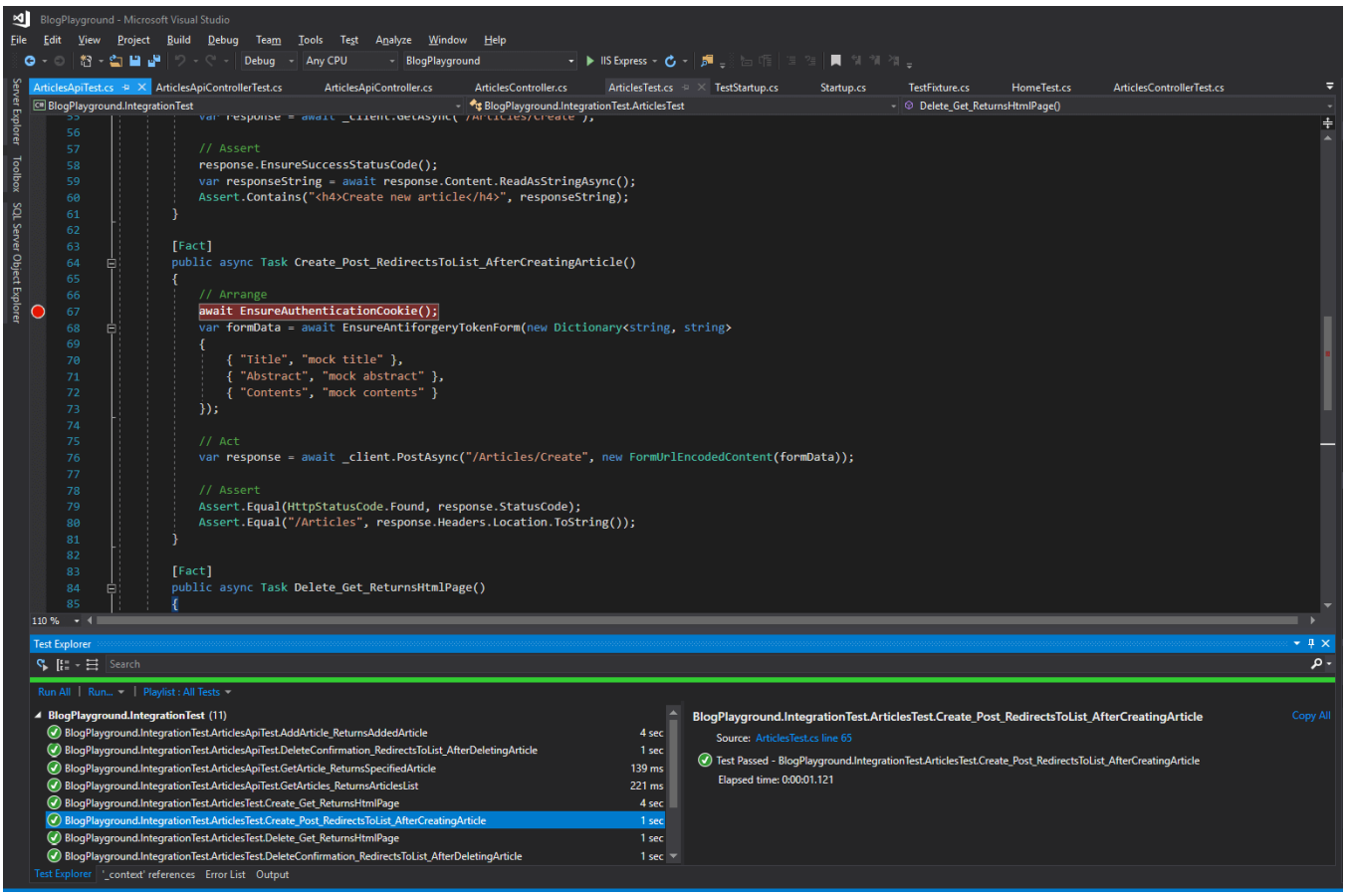


Figure 4, running the integration test with authentication and anti-forgery

You can try and include additional coverage before we move on to the API controller. Check the GitHub repo if you need to.

Testing API controllers

Before we finish with the integration test, let's for a second also consider the API controller we introduced

in the previous article (on Page 8). It provides a similar functionality in the shape of a REST API that sends and receives JSONs.

There are two main differences with the earlier tests:

- We need to setup anti-forgery to also consider reading the token from a header, since we won't be posting url-encoded forms. We will then post a JSON with an article and include a separate header with the anti-forgery token.

- We will need to use **Newtonsoft.Json** in order to serialize/deserialize the request and response contents to/from JSON objects.

Let's deal with anti-forgery first. In your startup class, include the following line before the call to `AddMvc`:

```
services.AddAntiforgery(opts => opts.HeaderName = "XSRF-TOKEN");
```

We just need another utility in our fixture class that sets the token in the expected header:

```
public async Task EnsureAntiforgeryTokenHeader()
{
  _client.DefaultRequestHeaders.Add(
    "XSRF-TOKEN",
     await EnsureAntiforgeryToken()
  );
}
```

We can then write a test that uses **Newtonsoft.Json's JsonConverter** to serialize/deserialize an `Article` into/to JSON. The test for the create method would look like this:

```
[Fact]
public async Task AddArticle_ReturnsAddedArticle() {
  // Arrange
  await EnsureAuthenticationCookie();
  await EnsureAntiforgeryTokenHeader();
  var article = new Article { Title = "mock title", Abstract = "mock abstract",
  Contents = "mock contents" };

  // Act
  var contents = new StringContent(JsonConvert.SerializeObject(article), Encoding.
  UTF8, "application/json");
  var response = await _client.PostAsync("/api/articles", contents);

  // Assert
  response.EnsureSuccessStatusCode();
  var responseString = await response.Content.ReadAsStringAsync();
  var addedArticle = JsonConvert.DeserializeObject<Article>(responseString);
  Assert.True(addedArticle.ArticleId > 0, "Expected added article to have a valid
  id");
}
```

Notice how we can serialise the `Article` we want to add into a `StringContent` instance with:

```
new StringContent(JsonConvert.SerializeObject(article), Encoding.UTF8,
"application/json");
```

Similarly, we deserialize the response into an `Article` with:

```
var addedArticle = JsonConvert.DeserializeObject<Article>(responseString);
```

Testing the additional methods provided by the API controller should be straightforward now that we know how to deal with JSON, authentication cookies and anti-forgery tokens!

Conclusion

We have seen how Integration Tests complement Unit Tests by including additional parts of the system like database access or security in the test, while still running in a controlled environment.

These tests are really valuable, even more so than unit tests in applications with very little or very simple logic. If all your app does is provide simple CRUD APIs with little extra logic, you will be investing your time better in adding integration tests, while reserving unit tests for specific areas with logic that might benefit from them.

However, similar to Unit Tests; Integration tests aren't free either!

They are harder to setup when compared to unit tests, require some significant upfront time to deal with specific features of your application like security, and are considerably slower than unit tests.

Make sure you have the right proportion of unit vs integration tests in your project, maximizing useful coverage against the writing and maintenance cost of each test!

It is also worth highlighting that we are still leaving some pieces of the application out. For instance, the database wasn't a real database, it was an in-memory replacement so we could easily setup and tear it down in every test.

Similarly, the tests send requests in a headless mode without a browser, even though we are building a web application whose client side would end up running in a browser. This means we are not done yet and we need to consider even higher-level tests.

The next upcoming article will look at **end-to-end tests**, where we can ensure a real deployed system is working as expected.

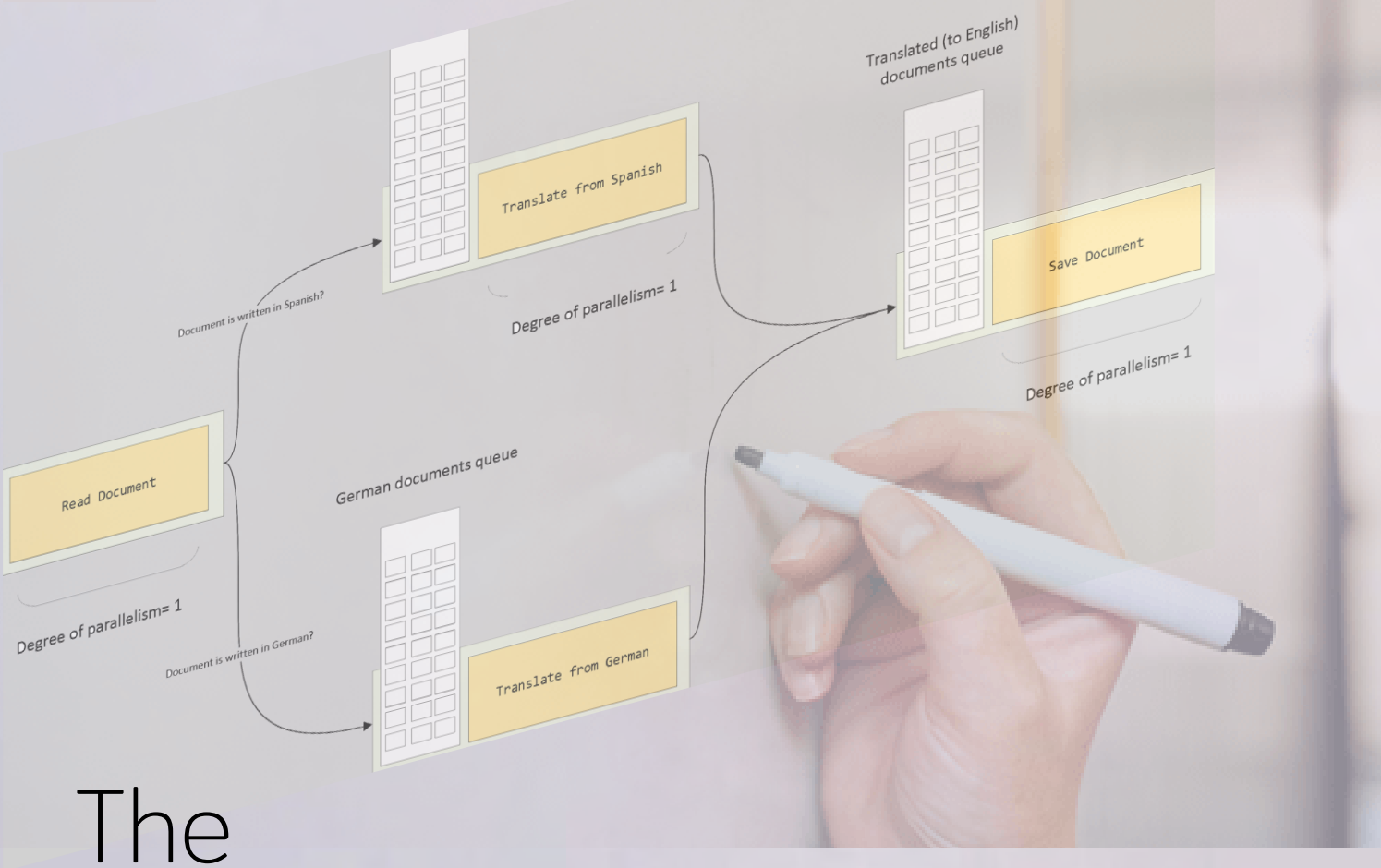
Daniel Jimenez Garcia

Author

Daniel Jimenez Garcia is a passionate software developer with 10+ years of experience. He started as a Microsoft developer and learned to love C# in general and ASP MVC in particular. In the latter half of his career he worked on a broader set of technologies and platforms while these days is particularly interested in .Net Core and Node.js. He is always looking for better practices and can be seen answering questions on Stack Overflow.

Thanks to Damir Arh for reviewing this article.

Yacoub Massad



The Producer-Consumer Dataflow pattern in .NET

This article discusses the dataflow pattern, a variant of the producer-consumer pattern, along with examples of applying this pattern in .NET. Finally, you will also be introduced to ProceduralDataflow, a new library I created to help write clean dataflows.

Introduction

In a previous article, The Consumer-Producer pattern in .NET, I talked about the producer-consumer pattern, the pipeline pattern, and briefly touched upon the dataflow pattern.

In this article I am going to talk about the dataflow pattern in more details, explain the Dataflow API in .NET, and introduce *ProceduralDataflow*, a new library I created that allows us to write dataflows in a procedural way.

Note: *This article is a continuation of the previous one. I recommend that the reader reads that article before reading this one.*

A quick recap

In the simplest implementation of the **Producer-Consumer pattern**, we have a single thread producing some data and putting it in a queue, and another thread consuming the data from the queue.

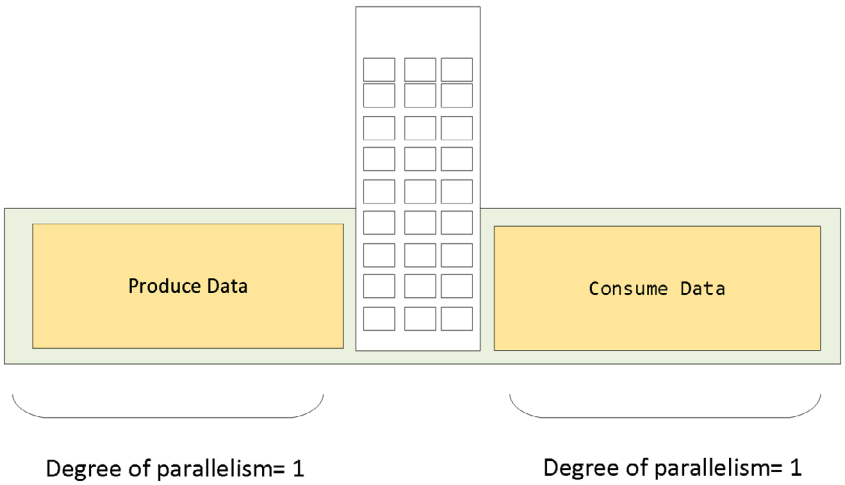


Figure 1: The producer-consumer pattern

The **Pipeline pattern** is a variant of the producer-consumer pattern.

This pattern allows the consumer to also be a producer of data. This data will be put in a second queue, and another consumer will consume it.

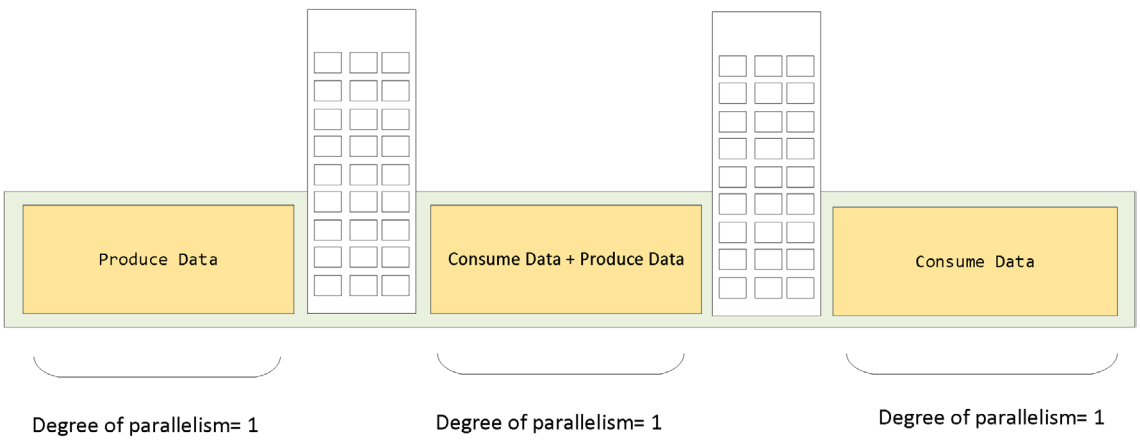


Figure 2: the pipeline pattern

In the example above, we have a pipeline that does three stages of processing.

In the first stage, some data is produced, in the second stage this data is consumed and some other data is produced. This other data will be consumed in the third stage.

In the previous article, I gave an example where, in the first stage, we read documents from some store (e.g. a database), in the second stage we translate them, and in the third stage, we store the translated documents into some destination store. In this example, the first queue stores documents read from the store, while the second queue stores translated documents.

The Dataflow pattern

In the pipeline pattern, there is no restriction on the number of stages we can have. However, the flow has to be *linear*.

The dataflow pattern removes any such restriction!

The flow can branch based on some condition. Or it can branch unconditionally so that an item is sent to two processing nodes, instead of one.

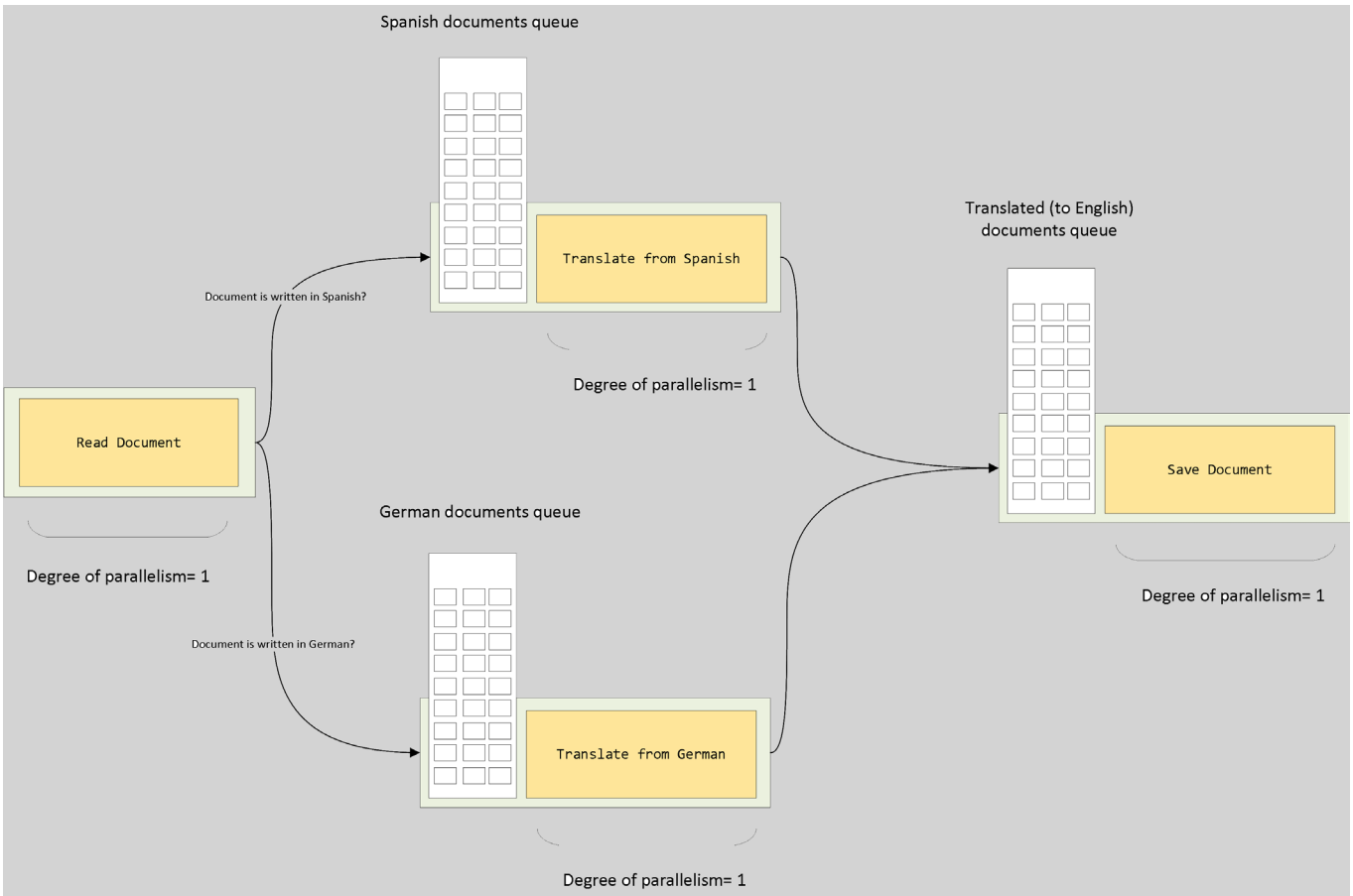


Figure 3: A branch in a dataflow

In this figure, in the first stage, a document is read. Then, based on the language of the document, it is either enqueued in the queue for Spanish documents, or the queue for German documents.

A special consumer thread reads Spanish documents from the first queue and translates them. Meanwhile,

another special consumer thread reads German documents from the other queue and translates them.

These two consumers are also producers since they will add the translated documents to the translated documents queue. A special consumer thread reads the translated documents and writes them to the store (e.g. a database).

Before continuing past this example, let's first discuss why we need this branching, e.g. **why don't we simply use the pipeline pattern as in the following figure?**

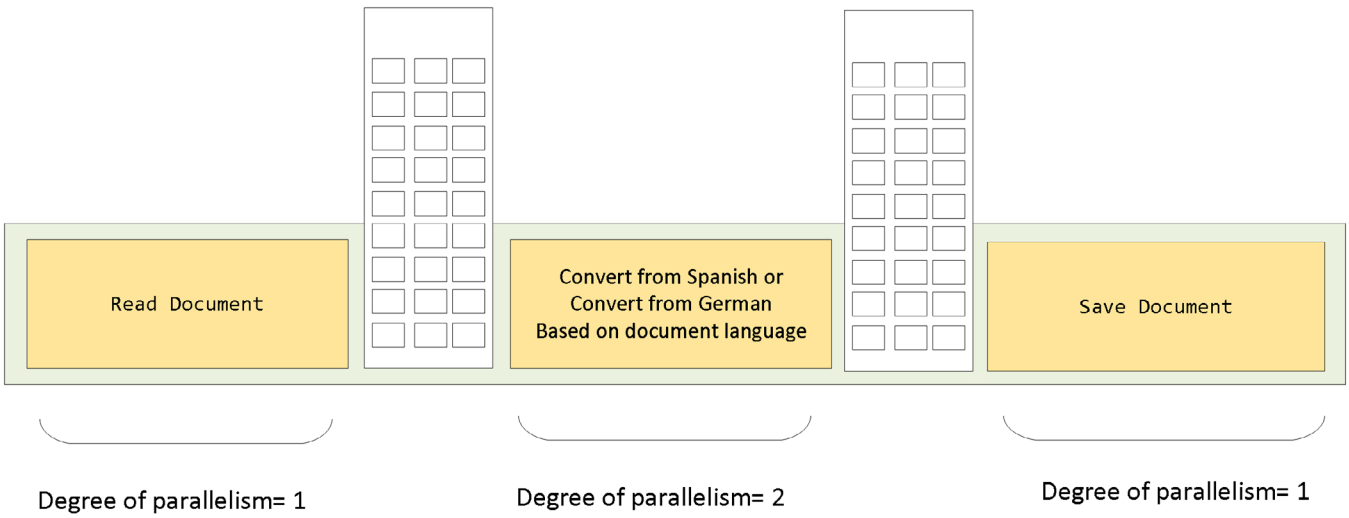


Figure 4: Using a pipeline instead of a dataflow

In this figure, we have a simple pipeline. In the first stage, we read a document and put it in a single queue regardless of the language in which it is written. In the second stage, we determine the document language and decide whether to invoke some code to translate the document from Spanish or some other code to translate from German.

Notice how the degree of parallelism for this stage, is two.

In both the examples in figure 3 and 4, we translate two documents at the same time.

So what benefits does the dataflow approach give us?

In the dataflow approach, at most one Spanish document will be processed at any given time. Also, at most one German document will be processed at a time.

On the other hand, in the pipeline approach, it is possible that two threads will be processing two Spanish documents in parallel (or two German documents).

So, the dataflow approach gives us **better control for the degree of parallelism** for each operation.

Note: *refer to the previous article for reasons you might want to control the degree of parallelism for each operation.*

Also, this approach allows us to have more control on the queue sizes. For example, consider the scenario where the Spanish documents in the source store are very large and German documents are small. We might decide to have at most ten untranslated Spanish documents sitting in their queue and at most 100

untranslated German documents, in its queue.

In the previous example, the branching was conditional. That is, a document either went to the Spanish documents queue or the German documents queue.

Another type of branching is unconditional branching. In this type of branching, the item will be sent both ways unconditionally.

For example, assume that all documents we read from the source store are written in English and we want to translate each document to both Spanish and German. In this case, each document will be sent to two queues.

Next, let's look at how we can implement the first example in .NET.

Implementation

In this section, we will look at different implementation options.

The BlockingCollection class

As in the previous article I am going to first use the `BlockingCollection` class to implement the Dataflow example.

```
public void ProcessDocumentsUsingDataflowPattern()
{
  string[] documentIds = GetDocumentIdsToProcess();

  BlockingCollection<string> inputQueue = CreateInputQueue(documentIds);

  BlockingCollection<Document> spanishDocumentsQueue = new
  BlockingCollection<Document>(10);

  BlockingCollection<Document> germanDocumentsQueue = new
  BlockingCollection<Document>(100);

  BlockingCollection<Document> translatedDocumentsQueue = new
  BlockingCollection<Document>(100);

  var readingTask =
    Task.Run(() =>
    {
      foreach (var documentId in inputQueue.GetConsumingEnumerable())
      {
        var document = ReadDocumentFromSourceStore(documentId);

        var language = GetDocumentLanguage(document);

        if (language == Language.Spanish)
          spanishDocumentsQueue.Add(document);
            else if (language == Language.German)
              germanDocumentsQueue.Add(document);
            }
        });
```

```
    var spanishTranslationTask =
     Task.Run(() =>
     {
       foreach (var readDocument in spanishDocumentsQueue.GetConsumingEnumerable())
       {
         var translatedDocument =
           TranslateSpanishDocument(readDocument, Language.English);

         translatedDocumentsQueue.Add(translatedDocument);
       }
     });

     var germanTranslationTask =
       Task.Run(() =>
       {
         foreach (var readDocument in germanDocumentsQueue.GetConsumingEnumerable())
         {
            var translatedDocument =
            TranslateGermanDocument(readDocument, Language.English);

            translatedDocumentsQueue.Add(translatedDocument);
         }
       });

     var savingTask = Task.Run(() =>
     {
       foreach (var translatedDocument in translatedDocumentsQueue.
       GetConsumingEnumerable())
       {
         SaveDocumentToDestinationStore(translatedDocument);
       }
     });

     readingTask.Wait();

     spanishDocumentsQueue.CompleteAdding();

     germanDocumentsQueue.CompleteAdding();

     spanishTranslationTask.Wait();

     germanTranslationTask.Wait();

     translatedDocumentsQueue.CompleteAdding();

     savingTask.Wait();
}
```

This example is very similar to the examples I provided in the previous article. We simply create three `BlockingCollection` objects for the three queues (plus one for the input queue), and then create four tasks.

The first one reads documents from the store and puts them in the appropriate queue based on their language. This is where the branching occurs.

The next two tasks take documents from the Spanish and German document queues, translate them, and then put them in the translated documents queue.

The fourth task takes documents from the translated documents queue and saves them to the destination store.

What if the German documents translation method, i.e, `TranslateGermanDocument`, is an asynchronous IO-bound method? i.e., what if it had the following signature?

```
public async Task<Document> TranslateGermanDocumentAsync(
    Document document,
    Language language)
```

In this case, it would be better if we don't tie up a thread while we wait for this asynchronous method to complete.

A note about asynchronous operations:

Consider for example that the Spanish document translation happens in the same machine as our application. This means that this operation is a CPU-intensive operation and therefore it requires a thread.

On the other hand, consider that German document translation is done via a commercial web service call. This means that this operation is an I/O-bound operation and therefore does not require a thread.

Threads are expensive resources in server applications that are required to handle a larger number of client requests at the same time. Using an asynchronous method means that we can save such threads.

When we create the German documents translation task, we can use an overload of `Task.Run` that accepts an asynchronous action, i.e., `Func<Task>` like this:

```
var germanTranslationTask =
  Task.Run(async () =>
  {
    foreach (var readDocument in germanDocumentsQueue.GetConsumingEnumerable())
    {
      var translatedDocument =
        await TranslateGermanDocumentAsync(readDocument, Language.English);
        translatedDocumentsQueue.Add(translatedDocument);
    }
  });
```

This way, we don't tie up a thread while we wait for the translation method to complete. However, we still block the current thread in these two places:

- The implicit call on `IEnumerable<T>.MoveNext()` on the enumerable returned by `germanDocumentsQueue.GetConsumingEnumerable()` when the German documents queue is currently empty. That is, when we wait for the empty queue to become non-empty.

- The call to `translatedDocumentsQueue.Add` when the queue is full. That is, when we wait for the full queue to become non-full.

Although we are using an asynchronous action as a parameter for the `Task.Run` method, these two calls

will still be done synchronously. In scenarios where we have an operation with a high degree of parallelism, such synchronous waiting will cause us to tie up many threads unnecessarily which will affect the scalability of the system.

We can solve these problems by using a class similar to the `BlockingCollection` class but that supports asynchronous taking and adding from the queue. That is, a class that allow us to asynchronously block for the queue to be non-empty or non-full.

One such class can be found in the AsyncEx library and is called `AsyncProducerConsumerQueue`. We can replace the `BlockingCollection` class with this class in the previous example to spare threads, while waiting for the queues to become non-empty or non-full. I am not including the details of how to do this here as I leave this as an exercise for the reader.

Please note that it also makes sense to block asynchronously waiting for the queues, even with synchronous operations (e.g. CPU-bound document translation).

The TPL Dataflow API

Next, I am going to show you how to implement the same example using the TPL Dataflow API.

Although not shipped with the .NET framework, the TPL Dataflow library is a library from Microsoft created specially to help us build dataflows. The library provides a set of blocks, each of which have specific features. Here are some example blocks:

- The *TransformBlock* block processes data it receives to produce other data. It includes one input and one output buffers. Such buffers are similar to the queues I have been discussing since the previous article.

- The *ActionBlock* block processes data it receives but does not produce any data out of processing. It includes an input buffer.

Take a look at the TPL Dataflow documentation for more information about the different block types provided by the TPL Dataflow library.

In this article, I am going to show how to implement the same example I used before using the TPL Dataflow library. I will also talk about different features of the API, as I explain the example.

Consider this code:

```
public void ProcessDocumentsUsingTplDataflow()
{
  var readBlock =
    new TransformBlock<string, Document>(
      x => ReadDocumentFromSourceStore(x),
      new ExecutionDataflowBlockOptions { MaxDegreeOfParallelism = 1 }); //1

  var spanishDocumentTranslationBlock =
    new TransformBlock<Document, Document>(
      x => TranslateSpanishDocument(x, Language.English),
      new ExecutionDataflowBlockOptions { BoundedCapacity = 10}); //2

  var germanDocumentTranslationBlock =
    new TransformBlock<Document, Document>(
      x => TranslateGermanDocumentAsync(x, Language.English),
```

```
    new ExecutionDataflowBlockOptions { BoundedCapacity = 100 }); //3

  var saveDocumentsBlock =
    new ActionBlock<Document>(
      x => SaveDocumentToDestinationStore(x),
      new ExecutionDataflowBlockOptions { BoundedCapacity = 100 }); //4

  readBlock.LinkTo(
    spanishDocumentTranslationBlock,
      new DataflowLinkOptions {  PropagateCompletion = true},
      x => GetDocumentLanguage(x) == Language.Spanish); //5

  readBlock.LinkTo(
    germanDocumentTranslationBlock,
      new DataflowLinkOptions { PropagateCompletion = true },
      x => GetDocumentLanguage(x) == Language.German); //6

    spanishDocumentTranslationBlock.LinkTo(
      saveDocumentsBlock); //7

    germanDocumentTranslationBlock.LinkTo(
      saveDocumentsBlock); //8

    string[] documentIds = GetDocumentIdsToProcess(); //9
    foreach (var id in documentIds)
      readBlock.Post(id); //10

    Task.WhenAll(
      spanishDocumentTranslationBlock.Completion,
      germanDocumentTranslationBlock.Completion)
        .ContinueWith(_ => saveDocumentsBlock.Complete()); //11

    readBlock.Complete(); //12

    saveDocumentsBlock.Completion.Wait(); //13
}
```

This method contains 13 statements, duly marked in the comments. I will explain them now.

In the 1st statement, I create the block that will read documents from the store. This block is the `TransformBlock<string, Document>` which means that this block consumes strings and produces Document objects.

In this case the string is the document id. The first parameter in the constructor is a delegate to the function that will transform the string to a document. We give it a lambda expression that invokes the `ReadDocumentFromSourceStore` method. The second parameter allows us to configure more options for this block. I have set the `MaxDegreeOfParallelism` property to 1. This code isn't really needed because the default value is one, but I have nevertheless set it just to show you how you can control the degree of parallelism for this block.

In the 2nd and 3rd statements, I create two `TransformBlock<Document,Document>` objects. The first one translates Spanish documents and the second one, German documents. Notice how the constructor of `TransformBlock` allows me to specify a synchronous transform function in the case of Spanish documents, and an asynchronous one, for German documents.

Note that when I create these two `TransformBlock` objects, I set the bounded capacity to 10 and 100

for the Spanish and German blocks respectively. As I mentioned before, the `TransformBlock` block has an input buffer and an output buffer. The bounded capacity setting allows us to control the sizes of such buffers.

In the 4th statement, I create an `ActionBlock<Document>` block that will save the documents to the destination store. Here we need an `ActionBlock` because the `SaveDocumentToDestinationStore` method does not return anything. This block is the last block in the dataflow.

I still haven't connected any of these blocks. Now, the connecting (or linking as called in the TPL DataFlow library) begins.

In the 5th statement, I use the `LinkTo` method to link the document reading block with the Spanish documents translation block. This basically means that data coming out of the read block will be sent to the Spanish documents translation block.

There are two things to note here.

The first one is that I pass a `DataflowLinkOptions` object with `PropagateCompletion` set to `true`. I will explain this soon.

The second one is that I gave the `LinkTo` method a *predicate* that will be used to filter the data items sent from the source block to the destination block. In this particular case, I only want Spanish documents to be sent to the Spanish documents translation block.

In the 6th statement, I do the same thing for German documents.

In the 7th and 8th statements, I link the Spanish and German translation blocks, respectively, to the save-document block.

Two things to note here.

First, we are linking these two translation blocks to the same target block. Second, here we don't set `PropagateCompletion` to `true`. I will explain the why part soon.

In the 9th statement, I simply get the list of document ids to process.

In the 10th statement, I loop through this list and call the `Post` method on the document reading block. This will cause the processing to actually start. Each item posted to the read block will flow through the blocks. This of course depends on how we have linked the blocks together and the settings we have specified for the different blocks.

Now let's talk about the concept of *Completion* a little bit.

In all of my code examples (in this and the previous article), at the end of processing, I invoked some code to wait for processing to complete. This is needed in many scenarios because we need to make sure that all the data has been processed before doing something else. Or to simply inform the user that all the data has been processed.

In many cases, this meant that I invoked the `Wait` method on some `Task` object before leaving the method. The TPL Dataflow library provides the *Completion concept* to solve this problem.

In the 13th statement, I invoke the getter of the `saveDocumentsBlock.Completion` property. This will give us a `Task` object that completes when the block completes. I then invoke the `Wait` method on this `Task` object to block until this task completes. Basically, I want to wait for all data to be completely processed.

How can the saveDocumentsBlock block "complete"? Even if it has processed every item it received, how does it know that there are no more items to come?

The Dataflow blocks support the concept of propagating a completion signal.

In the 12th statement, I call the `Complete` method on the `readBlock` object. This tells this block that we are done posting new items to it. Not only that, if it was configured to do so, it will send (or propagate) a completion signal to other blocks it is linked to, when it has processed all the items it received.

What I would like to have is the completion signal propagate through all the blocks until it reaches the final block, i.e., the save-document block. This way, once the final block receives the completion signal and completes processing all items, the Task returned from `saveDocumentsBlock.Completion` will complete.

One small problem is that we have a branch. The completion signal can reach the save-document block in two ways (take a look at figure 3). The first way is via the Spanish document translation block. And the second one via the German document translation block.

If we allow the completion signal to be propagated automatically, this might cause the save-document block to complete (and stop receiving new items) before all documents are processed.

For example, this can happen if all Spanish documents are processed while there are still some German documents that are not processed yet. The Spanish document translation block would propagate the completion signal to the save-document block and cause the processing to stop.

This is why I did not setup completion signal propagation from the translation blocks to the save-document block in the 7th and 8th statements.

To fix the problem, in the **11th statement** I use standard TPL code to tell the system to invoke the `Complete` method on the save-document block when both the Spanish and German document translation blocks have completed (i.e., when they have received the completion signal from the read-document block AND have processed all the documents they received).

For another example that uses the TPL Dataflow, consider the Building an Image Resizer using .NET Parallel Dataflow Library in .NET 4.5 article here on DotNetCurry.

The problem with flow clarity

In both the implementation that uses the `BlockingCollection` class and the implementation that uses the TPL Dataflow API, the flow logic is tangled with API code. To explain this further, let me show you how our code would look like if we used simple data parallelism to process our documents:

```
public void ProcessDocumentsUsingParallelForEach()
{
  string[] documentIds = GetDocumentIdsToProcess();
  Parallel.ForEach(documentIds, documentId =>
  {
    var document = ReadDocumentFromSourceStore(documentId);
```

```
    var language = GetDocumentLanguage(document);

      Document translatedDocument;

      if (language == Language.Spanish)
        translatedDocument = TranslateSpanishDocument(document, Language.English);
        else // if (language == Language.German)
        translatedDocument = TranslateGermanDocument(document, Language.English);

      SaveDocumentToDestinationStore(translatedDocument);
  });
}
```

This code looks better, right?

The flow logic is very clear. The body of the `ForEach` loop shows us the steps needed to process a single document. We first read the document. Then we determine its language. Then we branch based on the language; We call `TranslateSpanishDocument` for Spanish documents and `TranslateGermanDocument` for German documents. Finally, we invoke `SaveDocumentToDestinationStore` to save the translated document.

Now try to go back to the dataflow implementations in this article and try to find these pieces of logic. It is all over the place.

In the `BlockingCollection` based example, reading the document and determining its language is in one place. Translating Spanish and German documents is in two other places. And saving documents is yet in another place. Between all of these pieces of code, we have infrastructure code unrelated to logic of the flow.

In the TPL DataFlow API based example, things are even worse. The branching logic for example is split between two different `LinkTo` calls.

Can we fix this?

Can we get the clarity of simple data-parallelism and yet get all the benefits the producer-consumer based patterns provide?

I think we can, and I created a library called ProceduralDataflow to demonstrate this.

The ProceduralDataflow library

Let's look first at the code and then I will explain it to you.

```
public void ProcessDocumentsUsingProceduralDataflow()
{
  var readDocumentsBlock =
    ProcDataflowBlock.StartDefault(
      maximumNumberOfActionsInQueue: 100,
      maximumDegreeOfParallelism: 1);

  var spanishDocumentsTranslationBlock =
    ProcDataflowBlock.StartDefault(
```

```
  maximumNumberOfActionsInQueue: 10,
  maximumDegreeOfParallelism: 1);

var germanDocumentsTranslationBlock =
  AsyncProcDataflowBlock.StartDefault(
    maximumNumberOfActionsInQueue: 100,
    maximumDegreeOfParallelism: 1);

var saveDocumentsBlock =
  ProcDataflowBlock.StartDefault(
    maximumNumberOfActionsInQueue: 100,
    maximumDegreeOfParallelism:1);

DfTask<Document> DfReadDocumentFromSourceStore(
  string documentId) =>
  readDocumentsBlock.Run(
    () => ReadDocumentFromSourceStore(documentId));

DfTask<Document> DfTranslateSpanishDocument(
  Document document, Language destinationLanguage) =>
    spanishDocumentsTranslationBlock.Run(
    () => TranslateSpanishDocument(document, destinationLanguage));

DfTask<Document> DfTranslateGermanDocument(
  Document document, Language destinationLanguage) =>
    germanDocumentsTranslationBlock.Run(
    () => TranslateGermanDocumentAsync(document, destinationLanguage));

DfTask DfSaveDocumentToDestinationStore(
  Document document) =>
    saveDocumentsBlock.Run(
    () => SaveDocumentToDestinationStore(document));

async Task ProcessDocument(string documentId)
{
  var document = await DfReadDocumentFromSourceStore(documentId);

  var language = GetDocumentLanguage(document);

  Document translatedDocument;

  if (language == Language.Spanish)
    translatedDocument =
      await DfTranslateSpanishDocument(document, Language.English);
  else // if (language == Language.German)
    translatedDocument =
      await DfTranslateGermanDocument(document, Language.English);

  await DfSaveDocumentToDestinationStore(translatedDocument);
}

string[] documentIds = GetDocumentIdsToProcess();

var processingTask =
  EnumerableProcessor.ProcessEnumerable(
    documentIds,
```

```
            documentId => ProcessDocument(documentId),
              maximumNumberOfNotCompletedTasks: 100);

    processingTask.Wait();
}
```

I first start by creating four Dataflow blocks.

The `ProcDataflowBlock` and `AsyncProcDataflowBlock` classes come from the `ProceduralDataflow` library. The difference between these two classes is that the `ProcDataflowBlock` class is used for CPU-bound operations and the `AsyncProcDataflowBlock` class is used for asynchronous operations, e.g. I/O bound operations.

This is very similar to the way we created the TPL Dataflow blocks.

Each block has an input queue. I specify the queue size and the degree of parallelism for each block. Note however that here I don't specify the code that each block will execute.

Next, I create four local functions: `DfReadDocumentFromSourceStore`, `DfTranslateSpanishDocument`, `DfTranslateGermanDocument`, and `DfSaveDocumentToDestinationStore`. These functions use the blocks created above to execute the corresponding methods.

Note: *Local functions is a new feature in C#7. For a primer on C# 7, read the "C# 7 – What's New" tutorial at* www.dotnetcurry.com/csharp/1286/csharp-7-new-expected-features.

For example, the `DfReadDocumentFromSourceStore` method uses the `readDocumentsBlock` to run the `ReadDocumentFromSourceStore` method. It does so by invoking the `Run` method on the block. This returns a `DfTask<Document>`. The `DfTask` class is a class from the `ProceduralDataflow` library and is similar to the `Task` class in .NET. There are differences though, but I am not going to delve into the details of how the ProceduralDataflow library works in this article.

The next part is the most interesting part: The `ProcessDocument` local function. The body of this function looks very similar to the `ForEach` loop in the `ProcessDocumentsUsingParallelForEach` method from before.

It shows very clearly the steps each document will go through. The flow logic is expressed in a very clear way.

In this function, we call the Df* version of the methods. These methods return `DfTask<Document>` or `DfTask`. The code uses the `await` keyword to asynchronously wait for these methods to complete.

Please note that since these methods return `DfTask` and not `Task`, the `await` keyword behaves differently. Again, I am not going to delve into details because in this article I want to focus on how to use ProceduralDataflow, and not how it works internally.

Here is how you can think about this method (the `ProcessDocument` local function): It describes the stages of processing a document in a procedural way, i.e., it uses standard procedural code like calling methods, using the `if` statement, etc. And each one of the four blocks will execute only one part of this method.

This is the power of async/await. It allows parts of a single method to be executed as if they are totally separate.

Don't let the procedural logic trick you!

Under the hood, it works like a dataflow. Processing of multiple documents by different blocks is done in parallel, the degree of parallelism for each block is respected, and there are queues to allow slow consumers to slow down fast producers.

Note that the `ProcessDocument` local function takes a document id and returns a Task. This Task represents the whole process for a single document.

Note: *I used local functions in this example for convenience. These methods can be normal methods.*

The code that follows calls the `GetDocumentIdsToProcess` method to get the document ids, and then calls a method from `ProceduralDataflow` called `EnumerableProcessor.ProcessEnumerable` to process all the documents.

There is nothing special about this method. It enumerates the passed enumerable, and calls the supplied delegate for each item (a call to `ProcessDocument` in this example). The only purpose of this method is to manage the processing of items so that we don't enqueue all data at once in the queue if the number of items is huge. In this particular example, I am limiting the number of uncompleted tasks to 100.

This method also returns a `Task` that completes when all the tasks are completed.

Now, consider the following code:

```
async Task ProcessDocument(string documentId)
{
  var document = await DfReadDocumentFromSourceStore(documentId);
  var language = GetDocumentLanguage(document);

  Document translatedDocument;


  try
  {
    if (language == Language.Spanish)
      translatedDocument =
        await DfTranslateSpanishDocument(document, Language.English);
    else // if (language == Language.German)
      translatedDocument =
        await DfTranslateGermanDocument(document, Language.English);
  }
  catch (Exception ex)
  {
    await DfStoreDocumentInFaultedDocumentsStore(documentId, document, ex);
    return;
  }

  await DfSaveDocumentToDestinationStore(translatedDocument);
}
```

This is a modified version of the `ProcessDocument` local function. It adds a try/catch block around the calls to the translation methods. If an exception occurs in translation, a method named `DfStoreDocumentInFaultedDocumentsStore` will be called to store such a document in a special database.

This method also has a corresponding block with its own degree of parallelism and queue size. Note that the `try` block surrounds the two calls to translate documents (Spanish and German translation methods). If any of them fail, the document will be sent to this special database.

Also note that here in the `catch` block, we have access to the `documentId` and the document variables. To appreciate the value of such an approach while implementing a Dataflow, try implementing this example using the `BlockingCollection` class or the TPL Dataflow API.

Other features in the ProceduralDataflow library

The ProceduralDataflow library has more features.

1. It supports deadlock-free loops.

What if the flow of processing has a loop? For example, the flow goes from block one to block two and then back to block one. This might cause a deadlock.

Consider the following figure:

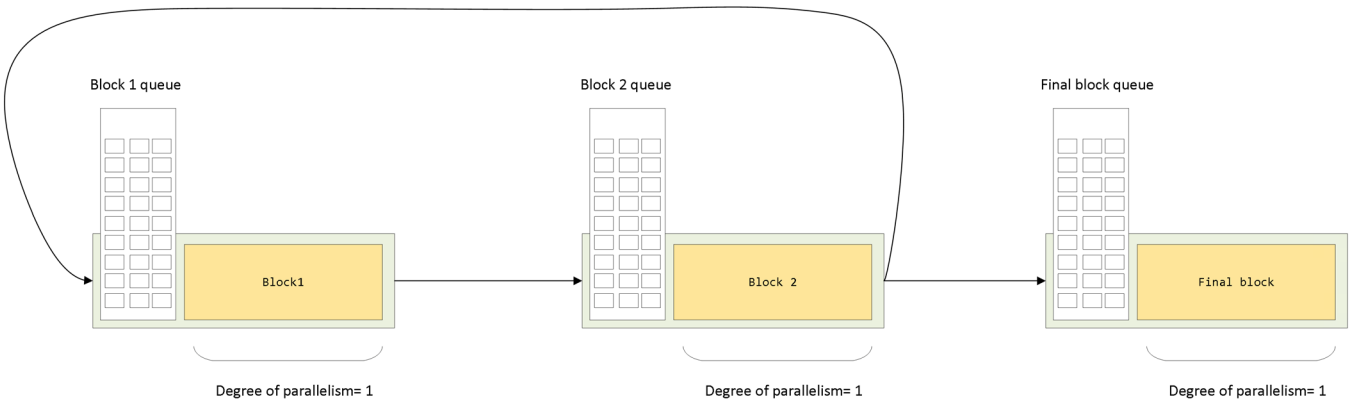


Figure 5: A loop in a Dataflow

In this figure, after Block 2 processes an item, the result of processing might go back to the queue of Block 1 or it might go to the final block queue based on some condition.

Now, assume that all queues are full. Without the loop, this cannot be a problem because eventually the final block queue will become non-full as a result of the final block processing some item.

With the loop, however, Block 1 might be waiting for Block 2's queue to become non-full and Block 2 might be waiting for Block 1's queue to become non-full.

This causes a deadlock.

ProceduralDataflow fixes this problem by making each block have two queues.

The first queue is for data that has not come to the block through a loop, and the second queue is for data that has come to the queue as a result of a loop.

What is special about the second queue is that adding items to such a queue does not cause the producer (the block that is adding the data to the queue) to block, which prevents a deadlock. In other words, this

second queue is not bounded.

Also, when getting the next item to process, a block favors items form the second queue over items from the first queue. The details of how ProceduralDataflow detects whether a data item has come through a loop or not, is outside the scope of this article.

2. It supports unconditional branching and joining:

ProceduralDataflow allows you process a data item with two or more blocks in parallel. Once they are complete, you can join the data from these blocks and process them as a single unit.

For example, you can translate a single English document to both Spanish and German (each via its own block), then asynchronously wait for both operations to complete, and then join the two translated documents as a single data item to create a zip file containing these two documents in a separate block. All this can be done procedurally.

Here is a code example:

```
async Task ProcessDocument(string documentId)
{
  var englishDocument = await DfReadDocumentFromSourceStore(documentId);

  DfTask<Document> toSpanishTask =
    DfTranslateEnglishDocumentToSpanish(englishDocument);

  DfTask<Document> toGermanTask =
    DfTranslateEnglishDocumentToGerman(englishDocument);

  SpanishAndGermanDocuments documents =
    await DfTask.WhenAll(
      toSpanishTask,
      toGermanTask,
      (spanish, german) =>
        new SpanishAndGermanDocuments(spanish, german));

  await DfCompressAndSaveDocuments(documents);
}
```

In this example, after we read an English document, we invoke `DfTranslateEnglishDocumentToSpanish` and `DfTranslateEnglishDocumentToGerman` without awaiting the returned tasks. This means that we don't wait for translation to Spanish to complete before we start translating to German.

Note that each of these translations will be done using a different block.

Then, we use the `DfTask.WhenAll` method to asynchronously wait for both translations to complete. Then, both the translated documents will be given to another block to compress and save the two documents together.

3. It supports sub procedures: If your flow logic is large, you split it into multiple `DfTask` or `DfTask<T>` returning methods.

4. No threads used unnecessarily: ProceduralDataflow does not cause any threads to block while waiting for queues to become non-empty or non-full. Also, no threads are used in asynchronous blocks.

How to get the ProceduralDataflow library

The ProceduralDataflow library is open source and you can find it in the following GitHub repository: https://github.com/ymassad/ProceduralDataflow.

I have also published the library in Nuget. You can find it by the name of **YMassad.ProceduralDataflow**. Currently it is in prerelease, so if you want to use it inside of Visual Studio, make sure you check the "include prerelease" checkbox in the Nuget package manager inside of Visual Studio.

I encourage the reader to try this library out. I would appreciate any feedback or suggestions.

Conclusion:

In this article, I have introduced the Consumer-Producer Dataflow pattern. This pattern is a variant of the Producer-Consumer pattern. Unlike the Pipeline pattern which allows only a linear flow of data between blocks, the Dataflow pattern allows the flow to be non-linear. For example, it allows conditional and non-conditional branching of data.

I have shown examples of how to implement this pattern using the `BlockingCollection` class and the TPL DataFlow API and talked about how such implementations can make the flow logic less readable.

I have also introduced a new library called ProceduralDataflow and showed with an example how it can help us implement the Dataflow pattern while keeping the flow logic clear and readable.

Download the library from GitHub at
bit.ly/dncm34-dataflow

Yacoub Massad
Author

Yacoub Massad is a software developer who works mainly with Microsoft technologies. Currently, he works at Zeva International where he uses C#, .NET, and other technologies to create eDiscovery solutions. He is interested in learning and writing about software design principles that aim at creating maintainable software. You can view his blog posts at criticalsoftwareblog.com

Thanks to Damir Arh for reviewing this article.