**Libraries:**

**NUnit**

**Fluent Assertions**

**ResrSHarp for Rest API**

**TDD**

Use NUnit and Fluent Assertions

The naming convention should show the Arrange, Act, Assert

Eg: PawnCanBePutOnBoard

[TestFixture] - defines a class which has got tests

The the annotation [Test]

Public void methodname{

//write the Arrange code

//write the Act code

//Write the Assert code  
}

The method to be followed is:

Write tests that will fail, write code so that the test will pass, then refactor the code so that the tests continue to pass.

[SetUp]

Public void SetUp(){

\_board=new Board(); //defines the set up

}

TearDown

OnTimeSetUp

OnTimeTearDown

**Parameterised Tests**

[TestCase("mike.walker@softwire.com")]  
[TestCase("sasha.burgoyne@softwire.com")]  
public void FindsValidEmails(string emailToCheck)  
{  
 var isValid = \_checker.isValidEmail(emailToCheck);  
  
 isValid.Should().BeTrue();  
}

**BusBoard**

**Use RestSharp for API calls**

**Var client= new RestClient(**"https://api.tfl.gov.uk/"

**) //give base URL**

**Var request=new RestRequest(**$"StopPoint/{stopCode}/Arrivals", Method.GET

**) //give the end point**

**Var response=client.Get<>.(request).Data;**

**Constructors: You can have multiple constructors (signature has to be different)**

public class Dog  
{  
 public string Name { get; }  
 public int Age { get; }  
  
 public Dog(string name, int age)  
 {  
 Name = name;  
 Age = age;  
 }  
}

**Once the property has been set through the constructor it cannot be again**

Ship It – 25/03

Constansts: Any contact value can be shown like this

private const double MAXTRUCKWEIGHT = 2000;

var myDog = new Dog  
{  
 Name = "Maisie",  
 Age = 2  
};

This is called object initialiser

**Instead of :**

var batches = new List<TruckModel.Batch>();

var batch = new TruckModel.Batch  
{  
 Name = products.Name,  
 Quantity = item.Quantity,  
 Gtin = products.Gtin,  
 WeightPerItem = products.Weight  
};

batches.add(batch)

**Do this:**

Batches.add(new batch{

Name = products.Name,  
 Quantity = item.Quantity,  
 Gtin = products.Gtin,  
 WeightPerItem = products.Weight

});

**Instead Of:**

foreach (var lineItem in lineItems)  
{  
 var batch = new TruckModel.Batch();  
 batch = CreateBatchItem(lineItem);  
 batches.Add(batch);  
   
}

**Do this:**

var batches = lineItems  
 .SelectMany(CreateBatchItem)  
 .OrderByDescending(item => item.TotalWeight);

**Instead Of**

foreach (var truck in trucks)  
{  
 if (truck.TotalWeight > batch.TotalWeight)  
 {  
 return truck;  
 }  
}

**DO This:**

return trucks.FirstOrDefault(truck => TruckHasSpace(truck, batch));

**Use Select:**

**This is converting each lineItem into another form**

var batches = lineItems  
 .SelectMany(CreateBatchItem)  
 .OrderByDescending(item => item.TotalWeight);

**Instead of**

var Items = new List<Item>  
 {new Item {Name = "Elixir of the Mongoose", SellIn = 5, Quality = 7}};  
\_gildedRose=new GildedRose.GildedRose(Items);

**DO this:**

var item = new Item { Name = "Test Item", Quality = 10, SellIn = 5};  
var gildedRose = new GildedRose.GildedRose(new List<Item> { item });

**Instead of**

**Calling functions with switch statements**

**Do this**

return item.Name switch

{

"Aged Brie" => GetAgedBrieQualityChange(item),

"Backstage passes to a TAFKAL80ETC concert" => GetBackStagePassQualityChange(item),

"Conjured Mana Cake" => GetConjuredItemQualityChange(item),

\_ => GetDefaultQualityChange(item)

};

**27/03/2020**

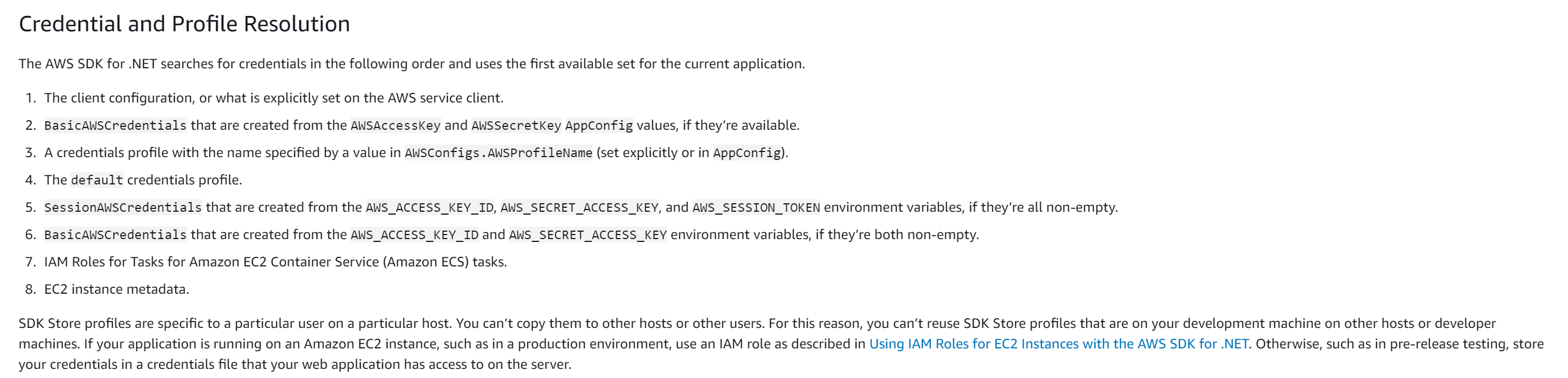
**Size of pages can slow the page , if lots of javascript is to be run**

30/3/2020

You can create environment variables and do this

**var ACCESS\_KEY = Environment.GetEnvironmentVariable("ACCESS\_KEY");**var SECRET\_KEY=Environment.GetEnvironmentVariable("SECRET\_KEY");  
var s3Client = new AmazonS3Client(ACCESS\_KEY, SECRET\_KEY, Amazon.RegionEndpoint.EUWest2);

or follow this



Create environment variables with exactly the names shown above

var s3Client = new AmazonS3Client();

**Project Structure (see GasMon)**

Main program

Initialise all classes here

Pass what you need into the constructors of a class

Keep classes small so that they are testable

# Dependency Injection

When class A uses some functionality of class B, then its said that class A has a dependency of class B.

before we can use methods of other classes, we first need to create the object of that class

**transferring the task of creating the object to someone else and directly using the dependency is called dependency injection**

## **Why should I use dependency injection?**

Let’s say we have a car class which contains various objects such as wheels, engine, etc.

Here the car class is responsible for creating all the dependency objects. Now, what if we decide to ditch **MRFWheels**in the future and want to use **Yokohama** Wheels?

We will need to recreate the car object with a new Yokohama dependency. But when using dependency injection (DI), we can change the Wheels at runtime (because dependencies can be injected at runtime rather than at compile time).

#### There are basically three types of dependency injection:

1. **constructor injection:** the dependencies are provided through a class constructor.
2. **setter injection:** the client exposes a setter method that the injector uses to inject the dependency.
3. **interface injection:** the dependency provides an injector method that will inject the dependency into any client passed to it. Clients must implement an interface that exposes a [setter method](https://en.wikipedia.org/wiki/Setter_method) that accepts the dependency.

**So now its the dependency injection’s responsibility to:**

1. Create the objects
2. Know which classes require those objects
3. And provide them all those objects

#### Benefits of using DI

1. Helps in Unit testing.
2. Boiler plate code is reduced, as initializing of dependencies is done by the injector component.
3. Extending the application becomes easier.
4. Helps to enable loose coupling, which is important in application programming.

#### Disadvantages of DI

1. It’s a bit complex to learn, and if overused can lead to management issues and other problems.
2. Many compile time errors are pushed to run-time.
3. Dependency injection frameworks are implemented with reflection or dynamic programming. This can hinder use of IDE automation, such as “find references”, “show call hierarchy” and safe refactoring.

DI in ASP.NET MVC

Steps:

1. Define the interface

Public interface IDateTime{

Public DateTime Now{get;}

}

1. Create class that implements that interface

public class SystemDateTime : IDateTime

{

public DateTime Now

{

get { return DateTime.Now; }

}

}

1. Add the class to COnfigurationServices in StartUp.cs file

public void ConfigureServices(IServiceCollection services)

{

services.AddSingleton<IDateTime, SystemDateTime>();

services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_2);

}

This means whenever IDateTime is called create object SystemDateTime

AddSingleton – is for one time

AddTransient is all all times

public class HomeController : Controller

{

private readonly IDateTime \_dateTime;

public HomeController(IDateTime dateTime)

{

\_dateTime = dateTime;

}

public IActionResult Index()

{

var serverTime = \_dateTime.Now;

## Action injection with FromServices

You can use Action Injection instead of Constructor Injection using  [FromServicesAttribute](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.fromservicesattribute) enables injecting a service directly into an action method without using constructor injection:

C#Copy

public IActionResult About([FromServices] IDateTime dateTime)

{

ViewData["Message"] = $"Current server time: {dateTime.Now}";

return View();

}

Refer to Bookish for DI excerise