Scope of injected dependencies:

**Transient Scoped Services:**

A new instance is created whenever such dependency is resolved in a client class. If a service called IAccountService is injected in class A, during the class object initiation (constructor call), this is resolved an creates an instance of AccountService. Similarly, one more instance is created when resolved in class B.

No need to worry about thread safety of such dependency object, as new object is resolved and injected for every client’s consumption

**Singleton Scoped Services:**

It creates a singleton instance for the application. It must behave in a thread safe as it’s shared across the clients. Large memory consuming dependencies are not the right candidates for Singleton scoping.

**Scoped Services (Per Request):**

It creates only one object of dependency per request. Unlike Transient, where every time the dependency is resolved in each class it creates a new object, here it creates only one object of dependency and injects in all the classes of this request where it is resolved.

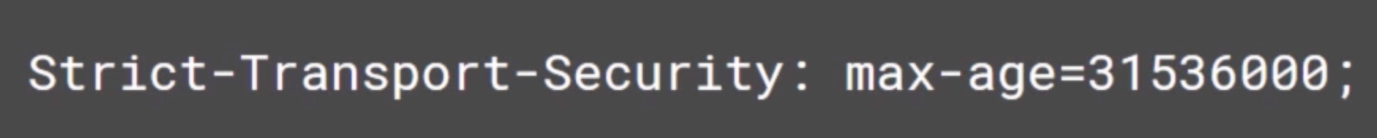
docker run -d --name Homer -e 'ACCEPT\_EULA=Y' -e 'SA\_PASSWORD=myPassw0rd' -p 1433:1433 mcr.microsoft.com/mssql/server

**ASP.Net Security**

**HSTS Header (Http Strict Transport Security) - Enforcing users to use SSL**

To avoid users communicating for insecure (http) channels, send the following response header, which will not let users to communicate with the site for certain amount of time (say for 1 year). So that users are enforced to communicate only by secure channels (https).

Max-age in the following header refers to 31 million seconds i.e. 1 year.



You can send this header in the first response from the site to the user. Mention it in the Configure method of Startup.cs.

app.UseHsts(h => h.MaxAge(days: 365).Preload());

**SQL Injection:A screenshot of a cell phone

Description automatically generated**

**Imagine the name param comes from the query string and user types in OR ‘1’=’1’ along with value of name parameter. (-- refers to the command execution ins sql). This will list down all the customer data to the requestor. Even in the worst case, the user may type the following for name parameter and send to the server.**

**Name=’a’; DROP TABLE Customer --.**

**This will create a havoc.**

**A screenshot of a cell phone

Description automatically generated**

**To control this**

* + **Check the input**
  + **Use the least privileged account for DB (in connection string), so that user don’t have admin privileges to DROP TABLE.**
  + **Use ORM like EF, which will parameterize inputs for SP.**

**CSRF (Cross Site Resource Forgery):**

Here the hacker steals the auth cookie of the user (steals the session) and encourages the user to click a luring add, but internally he uses the stolen information to cause a loss.A close up of a logo

Description automatically generated

When the hacker submits his version of the form containing amount and account number, he will get through the servers as he has the cookie with him and successfully able to transfer the money.

In order to stop this, a long-valued token is added to the cookie as well as form. When hacker submits his form, on the server side these values won’t match and server rejects the form submission.

To enable this decorate the POST action form with [ValidateAntiForgeryToken] attribute. ASP.Net Core will take care of rest of the matter.

**Cross Site Scripting (XSS):**

In this attack, the hacker would enter a malicious JavaScript snippet in one of the form fields and gets complete control of the page including cookie information (cookie stealing\cookie poisoning).

To control this, we should to Html encode the contents of the page data and validate the source of the javascripts your page can access (such as local server javascripts and js libararies from the CDNs). This js source validation is called

Content Security Policy (CSP):

This will validate the js code source, meaning it allows the js from the domain of the web application (self), CDNs. This also rejects any inline js snippets (entered by hacker in form fields).

Along with js, it also validates the sources of images, fonts etc…

A screenshot of a cell phone

Description automatically generated

Configure the CSP as a middleware and ensure only ‘Self’ and defined CDNs to supply js code.

A screenshot of a cell phone

Description automatically generated

.Self() – Will ensure the js from the domain of the web app.

.CustomSources(“maxcdn…”) - Will allow the js from CDN of domain “maxcdn.boot…”

The .ReportUris section at the end will generate the report at the path “/report” about policy violations and sends back.

**Open Redirection Attack:**

**A close up of a logo

Description automatically generated**

Hacker sends an email\message link with valid bank URL to log on (**bank.com**), when user enters, he returns the page from his own malicious bank site (which looks exactly like genuine bank website (**bank.net**)) stating incorrect password. This forces the user to reenter credentials but on the fake bank website. This way hacker steals the credentials through his fake website.

In order to prevent this, make sure every Url encountered is a local Url (i.e. of same domain bank.com). In case if not (i.e. from another domain bank.net), then refuse to process.

A close up of a logo

Description automatically generated

**Click Jacking:**

A screenshot of a cell phone

Description automatically generated

Here hacker lures the user to a fake website. The ‘You have won $100’ and ‘click here’ button is placed in a form and this form is placed in an Iframe. This Iframe is positioned on the ‘Transfer’ button of the bank.com (original) page. When user enters account details and clicks transfer button, this click become effective on the fake frame placed by the hacker. Thus, hacker will cause the money transfer to his account.

In order to control, we should deny the use of frames or allow only valid\known frames.

A black sign with white text

Description automatically generated

You can configure this with following middleware.

App.UseXfo(o => o.Deny());

**CORS (Cross Origin Resource Sharing):**

You can define certain policies in the Startup.cs file. Selectively you can define these policies over the controller or it’s action method(s).

Let’s define a policy for CORS, and we let only GET calls for the requests matching this policy and we let only one controller’s action method to entertain the calls matching this same policy.

**Startup.cs:**

**Define the policy at Configure Services method.**

Public void ConfigureServices(IServiceCollection services) {

Services.AddCors(options =>

{

Options.AddPolicy(“**AllowBankCom**”, c=> c.WithOrigins(<https://bank.com>));

}

}

Now in the Configure method, you can use this policy in the CORS middleware.

**App.UseCors(“AllowBankCom”);**

Now decorate the desired controller with this Cors policy.

**[EnableCors(“AllowBankCom”)]**

Public class MyController

{

}

# ASP.Net Core Security

**Token Base Security:**

**Token Types:**

Identity Tokens: They contain claims of the user. They are used by the clients (Web App, services, mobile apps etc…)

Access Tokens: Then contain the information and API calls needed by the client. They are used by the APIs.

The identity tokens follow the standards called OpenID Connect (OIDC)

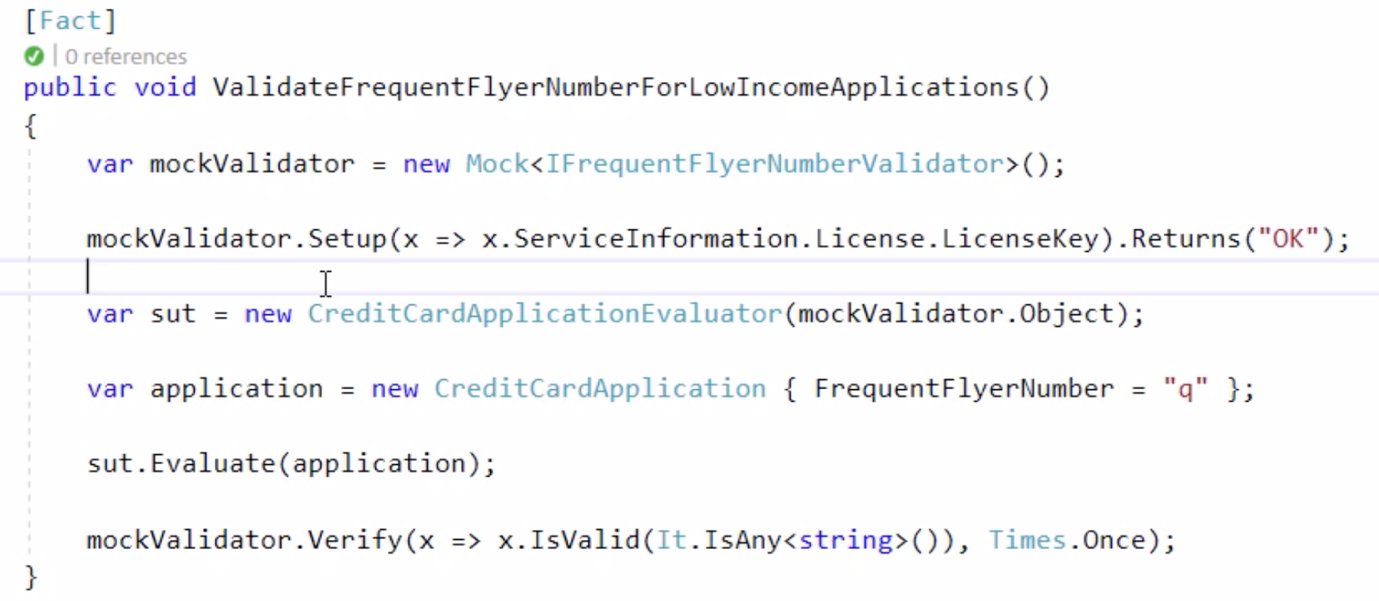
The access tokens follow the standards called OAuth2

**Unit Testing: Behavior Testing**

The Moq object can be used to check whether a particular method is called, how many times it’s called, whether a property value is accessed (Get) and so on. This kind of behavior check of the application is called “**Behavior Testing**”.

The Verify() and VerifyGet() methods of Moq object can be used for behavior testing.

Ex: To check whether IsValid() method is called only one.



Similarly to make sure whether the IsValid() method is never being called, we can write

mockValidator.Verify(x => x.IsValid(It.IsAny<string>()), Times.Never);

Ex: To check whether LicenceKey property is accessed to GET it’s value.

mockValidator.VerifyGet(x => x.ServiceInfo.LicenceKey);

Ex: To check whether ValidationMode property is SET to some value.

mockValidator.VerifySet(x => x.ValidationMode = ValidationMode.Detailed);

Ex: To check a method (say under the loop) is called exactly 2 times.

mockValidator.Verify(x => x.IsValid(10)), Times.Exactly(2));

**Testing for the Exceptions Thrown:**

In order to verify whether the routine throws the exception, set up the exception in Moq.

mockValidator.Setup(x => x.IsValid(It.IsAny<string>())).Throws<Exception>();

**Returning Different Results for Sequential Calls:**

If a same method returns a different value on the second call, we get different results on 1st call and 2nd call. This can be tested as below. (Assuming 1st call returns false and 2nd call returns true).

Note that mockValidator.SetupSequence(…) will helps to chain different .Returns() method with false and true values 2 times.

Down the line Evaluate method (which internally calls IsValid()) is called twice to verify different results.

