CS526 Enterprise and Cloud Computing
Stevens Institute of Technology—Fall 2024
Assignment Two—Data Models and Security

This assignment requires the use of JetBrains Rider and ASP.NET 8.0. You are provided with a partially completed Web app, which you should complete. The app includes a banner image and two style sheets (one default, one for ADA clients). You may modify the look and appearance of the Web site (e.g. by changing the banner image), but you should leave the general structure: a header area, a navigation bar with navigation links for the Web site, a sidebar on the left, a content area on the right, and a footer. You will use this template to build a simple social network that is intended to allow people to share photographic images. The layout for all Web pages is specified in the `Layout` view (rather than the default `_Layout` view); the `_ViewStart` configuration view specifies that this be used as the template for all views.

In this assignment, we add a model to the Web application from the first assignment. The `Image` entity model is used to store metadata about images in a backend database:

```
public class Image {
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public virtual int Id { get; set; }
    [MaxLength(40)]
    public virtual string Caption { get; set; }
    [MaxLength(200)]
    public virtual string Description { get; set; }
    [DataType(DataType.Date)]
    [DisplayFormat(DataFormatString = "{0:d}")]
    public virtual DateTime DateTaken { get; set; }

    [ForeignKey("User")]
    public virtual string UserId { get; set; }
    public virtual ApplicationUser User { get; set; }

    [ForeignKey("Tag")]
    public virtual int TagId { get; set; }
    public virtual Tag Tag { get; set; }

    // The image has been validated.
    public virtual bool Valid { get; set; }

    // The image has been approved.
    public virtual bool Approved { get; set; }
}
```

There are two related entities: Tags are used to categorize images, and for simplicity we assume a one-to-many relationship from tags to images (so each image has just one category):

```
public class Tag {
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public virtual int Id {get; set; }
    [MaxLength(20)]
    public virtual string Name {get; set;}

    public virtual ICollection<Image> Images {get;set;}
}
```

The other related entities are users. In this case, our user entity class, `ApplicationUser`, extends the class of users from ASP.NET Identity, which we use for authentication:

```
public sealed class ApplicationUser : IdentityUser {

    public bool ADA { get; set; }
    public bool Active { get; set; }

    public ICollection<Image> Images { get; set; }
}
```

In addition to user information, the database defined by ASP.NET identity includes information about roles, for which your initialization code should define three roles: `User`, `Approver` and `Administrator`. The `User` role is for users who upload, view and manage images. The `Approver` role is for administrators who control the approval of images uploaded to the site (e.g., this may be a check for indecent content or breach of copyright). We will add approval functionality in a later assignment. For now, we will assume that all images that are uploaded are approved (and validated). The `Administrator` role is for administrators who control user accounts (including locking a user account).

The controller classes subclass a `BaseController` class that includes functionality for managing users common to many operations:

```
public class BaseController : Controller {

     protected UserManager<ApplicationUser> userManager;

     protected async Task<ApplicationUser> GetLoggedInUser()
     {
         var user = HttpContext.User;
         return await userManager.FindByNameAsync(user.Identity.Name);
     }

     protected ActionResult ForceLogin()
     {
         return RedirectToAction("Login", "Account");
     }
```

The `AccountController` class includes operations for registering a user, managing user accounts, and logging in. The `Manage` action allows an administrator to deactivate a user account: The user can no longer log in, and any images they have uploaded should be deleted (permanently). Implement this by associating a Boolean flag with every user account (in the `User` table), that is initially true to indicate that the user is active. The `Manage` action allows this flag to be set to false (disabling the account) and later back to true to reactivate the account. All images uploaded by a user are deleted whenever their account is deactivated. The `Manage` form presents a table with all usernames, in order to allow several accounts to be disabled or re-enabled at once. Only an administrator can manage user accounts, and this should be enforced using ASP.NET access control. The `Register` and `Login` actions (which you should complete) obviously should not require authorization, since no-one has any permissions until they have registered as a user and logged in. Administrative accounts are created during the initialization of the database.

The `ImagesController` class includes operations for managing images: Uploading, viewing, editing, and deleting. Viewing actions include the ability to view all images, or filter images by user or by tag. When a table of image information is displayed, the rows for images uploaded by the current user should include buttons for editing and deleting those images (see the `ListAll` view). Operations for uploading and editing images should be restricted to site users, given by the general role of `User`. Images can only be viewed by users logged into the system (Any user no matter their role may list images and view image details). Attempting to perform an image action without logging in should result in redirection to the login page.

It is important that you take steps to ensure that your application is robust against malicious attacks.

1. Access to the application should be restricted to authenticated users, except for the `Register` and `Login` actions for the accounts controller. Use the `Authorize` attribute to restrict access, requiring specific role-based permissions for certain actions (`Administrator` permission to deactivate/reactivate users, and `User` permission for uploading and editing images).
2. Protect users against CSRF attacks using secret token validation: Any form that uses `asp-action` or `asp-controller` tag helpers automatically includes a secret token when it is rendered, and your code must then use `ValidateAntiForgerToken` to validate the presence of such a token when posted back. You should also protect against open redirect attacks and avoid over-posting attacks by never binding an entity model using the model binder (use view models instead).
3. ASP.NET applications require communication over SSL *as a default* (to protect the session cookie). In general, you should protect against XSS attacks using HTML encoding and use measures to protect against cookie stealing (and consider using a library like AntiXSS), but we will not require it for the assignment.

The storage of entity objects in the database is represented by the `ApplicationDbContext` class, that extends the database used by ASP.NET Identity to store user and role information for authentication and authorization:

```
public class ApplicationDbContext : IdentityDbContext<ApplicationUser> {

    public ApplicationDbContext
           (DbContextOptions<ApplicationDbContext> options) : base(options) {
    }

    /*
     * There is a table for users defined in the Identity DB Context.
     */

    public DbSet<Image> Images { get; init; }
    public DbSet<Tag> Tags { get; init; }
}
```

The configuration of the application in `Program.cs` will need to add the database context as a service, so it can be injected into controllers using dependency injection, as well as the identity service. You will need to configure the database context to use SQL Server as the database server, and allow logging of "sensitive" information like passwords in case you need to debug failures to log in. You will also have to add the identity service, specifying the Entity Framework store to be used for storing identity information. See the lecture materials on Data Models for how to enable dependency injection of the database context with SQL Server. See the lecture materials on Web Security for how to configure the app to use ASP.NET Identity with Entity Framework.

You use an instance of `ApplicationDbContext` (obtained through dependency injection) to insert an image into the database:

```
protected ApplicationDbContext db;
...
Image image = new Image();
...
await db.Images.AddAsync(image);
await db.SaveChangesAsync();
```

This last step is necessary before you save the image file itself in the file system (You should try commenting it out and see what happens). This has important implications for the performance of the system, forcing the app to wait until the new image record has been synchronized with the database. In later assignments, we will see how to improve this with asynchronous event-driven execution.

The logic for initializing the database is in `ApplicationDbInitializer` (Add your own code to add additional users and tags). Since this initialization is done outside the scope of a controller, which has the scope of a Web request and uses dependency injection to inject the resources it needs, you will need to create a scope into which you can manually inject the resources that the database initializer will need, when the application has not yet started. See the lecture materials on Data Models for how to manually inject the dependencies for the database initializer, including a scoped database context.

Working with Entity Framework and SQL Server

You will work with Entity Framework Core as your ORM; the project settings already include dependencies on the EF Core libraries, installed via Nuget. You can check the installed packages in Visual Studio using `Tools | NuGet | Show NuGet Packages.` You will need to install the `dotnet-ef` command line for managing Entity Framework[1]:

```
dotnet tool install --global dotnet-ef
```

If you ever need to update this tool, do so as follows:

```
dotnet tool update --global dotnet-ef
```

[1] From the terminal window in JetBrains Rider.

You use the `dotnet-ef` tool to create an initial "migration" for the database, and then use this to set up the database tables. See the lecture materials for Entity Framework and for Web Security for more information (Remember to specify `ApplicationDbContext` for the context).

For the database, we will run SQL Server in a docker container. First, install Docker Desktop to run docker[2]. Configure docker preferences for resources (e.g., one CPU and 1GB for a docker container should be enough for development with SQL Server). Now create a virtual network that we will run our Docker containers on:

```
docker network create --driver bridge cs526-network
```

You will need to pull a docker image for SQL Server[3]:

```
docker pull mcr.microsoft.com/mssql/server:2022-latest
```

Then run the container:

```
docker run -e "ACCEPT_EULA=1"
           -e "MSSQL_SA_PASSWORD=database-password" -p 1433:1433
           --name mssql-server -d --network cs526-network
           mcr.microsoft.com/mssql/server:2022-latest
```

If you are using an ARM Mac, you will have to specify that the image run on an emulated x86_64 architecture:

```
docker run --platform=linux/amd64
           -e "ACCEPT_EULA=1"
           -e "MSSQL_SA_PASSWORD=database-password" -p 1433:1433
           --name mssql-server -d --network cs526-network
           mcr.microsoft.com/mssql/server:2022-latest
```

You can check that the docker container is running as follows:

```
docker ps -a
```

Since the container exposes SQL Server port 1433 on the host machine, you can use the `dotnet ef` tool to initialize the database (See the lectures). Stop and remove the container as follows:

```
docker stop mssql-server
docker rm mssql-server
```

Note that removing the container will also remove the database, which is stored on a part of the file system local to the container. To retain the database, you would need to mount an

[2] https://www.docker.com/products/docker-desktop/
[3] https://learn.microsoft.com/en-us/sql/linux/quickstart-install-connect-docker?view=sql-server-ver16&tabs=cli&pivots=cs1-bash.

external volume on the container file system, but we are just using this for development purposes for now.

To connect to the server from your application, the application reads the connection string in `appsettings.json`. In development settings, the connection string is set as[4]:

```
{
  "Data": {
    "ApplicationDb": {
      "ConnectionString": "Server=localhost;TrustServerCertificate=True",
      "Database": "ImageSharingDb"
    },
  ...
}
```

This connection string is read during setup in `Program.cs` and used to initialize the connection to the database. The connection string must be augmented with information about the name of the database, as well as the credentials the app will use to authenticate to the database server:

```
string dbConnectionString =
            builder.Configuration["Data:ApplicationDb:ConnectionString"];
var connStringBuilder = new SqlConnectionStringBuilder(dbConnectionString);

string database = builder.Configuration["Data:ApplicationDb:Database"];
string dbUser = builder.Configuration["Credentials:ApplicationDb:User"];
string dbPassword =
            builder.Configuration["Credentials:ApplicationDb:Password"];

connStringBuilder.InitialCatalog = database;
connStringBuilder.UserID = dbUser;
connStringBuilder.Password = dbPassword;

dbConnectionString = connStringBuilder.ConnectionString;
```

We do not store credentials in the configuration descriptor in the project; this is an anti-pattern. Some alternatives are:

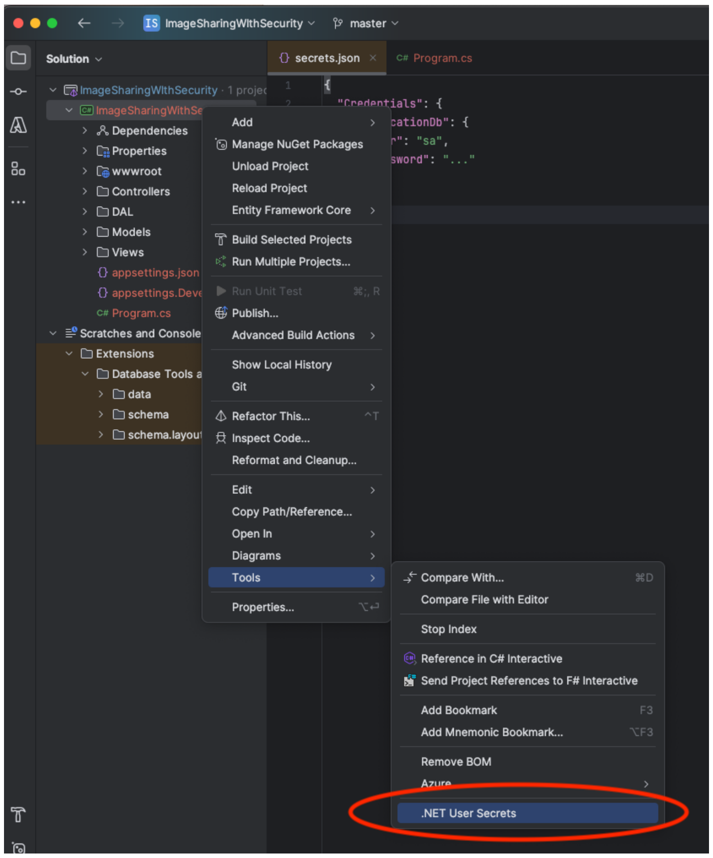
1. Use the ASP.NET Secret Manager[5] to store secrets in a special file outside the project during development, made available to the application configuration when you launch it from the IDE.
2. Pass the configuration information at runtime as environment variables e.g., of the form `"Credentials__ApplicationDb__User"`, where `"__"` (double underscore) is replaced by ASP.NET by `":"` when binding these environment variables in the context of the

[4] This assumes that the app runs on the same machine where the database server is running (albeit in a Docker container exposing port 1433). When you run the webapp in production mode in a Docker container, the connection string (in `appsettings.json`) will specify that the server runs on "virtual host" `mssql-server` on the same virtual network as the app.
[5] https://blog.jetbrains.com/dotnet/2023/01/17/securing-sensitive-information-with-net-user-secrets/.

application. This will be the preferred option for us when launching the app in a Docker container.

3. Use Azure Vault to store the secret information in the cloud. This is the option we will make use of in the next assignment when we deploy in the Azure cloud.

For the first of these options (which you will need when using `dotnet` to initialize the database), you can access user secrets in Rider from the Solution Explorer:



```
{
  "Credentials": {
    cationDb": {
    r": "sa",
    sword": "..."
```

The app expects a user secrets file of the form:

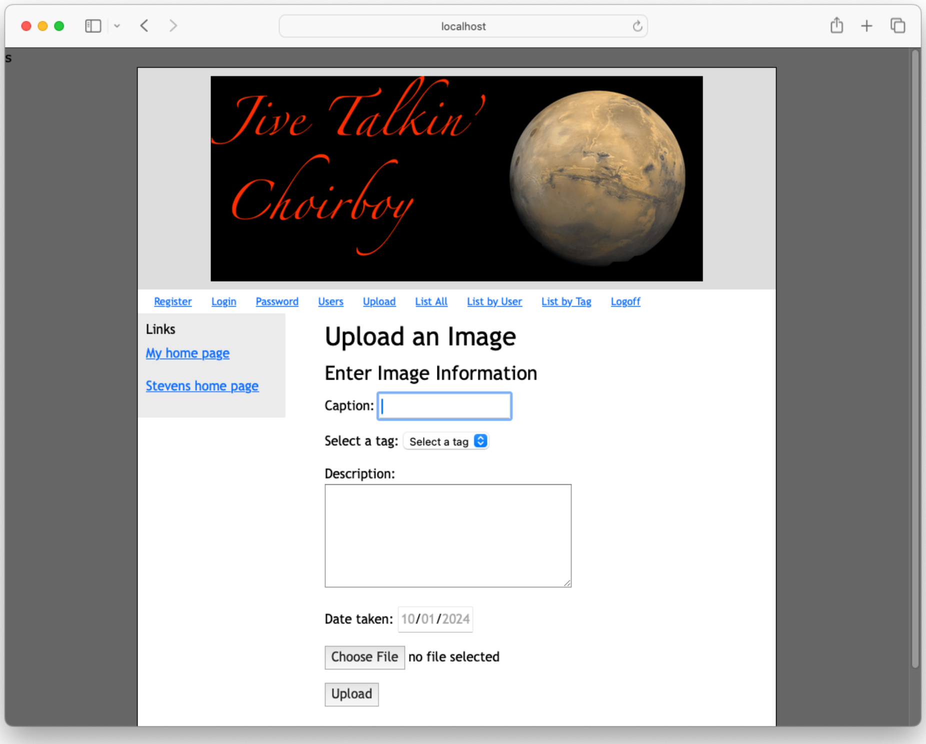
```
{
  "Credentials": {
    "ApplicationDb": {
      "User": "sa",
      "Password": "..."
    }
  }
}
```

You can connect to the database container to verify your login credentials, running the `sqlcmd` command line tool in a Bash shell in the container:

```
docker exec -it mssql-server bash
# /opt/mssql-tools18/bin/sqlcmd -S localhost -U sa -No
Password:
```

```
1> quit
# exit
```

Once you have used `dotnet ef` to create the initial migration for the app and created the database based on the resulting schema, you should be able to run the app from Rider[6]:



Upload an Image

Enter Image Information

Caption:

Select a tag:

Description:

Date taken: 10/01/2024

no file selected

Running in Docker

In preparation for cloud deployment, you must demonstrate running your app in Docker. When you run the app in docker, it will run in production mode using the default application settings (`appsettings.json`) rather than the development application settings. This should have the same information about the database as the development settings, with one difference: Since the app runs inside its own container, isolated from the host machine (and the database server), so that `localhost` just refers to the container itself, it needs to reference the database server using the container identifier for the latter:

```
{
  "Data": {
    "ApplicationDb": {
      "ConnectionString": "Server=mssql-server;TrustServerCertificate=True",
      "Database": "ImageSharingDb"
    },
  ...
}
```

[6] Remember that `launchsettings.json` configures the IDE to expose the app at ports 5000 (HTTP) and 5001 (HTTPS).

Use the `dotnet` command line tool, installed when you installed .NET, to publish the contents of the folder:

```
dotnet publish -c Release --no-self-contained
               -o ~/tmp/cs526/ImageSharingWithSecurity/publish
cd ~/tmp/cs526/ImageSharingWithSecurity
```

Use an editor of your choice to create a file in this directory called Dockerfile with these four lines:

```
FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS runtime
WORKDIR /app
COPY publish .
ENTRYPOINT ["dotnet", "ImageSharingWithSecurity.dll"]
```

Now build an image of the application you are going to deploy:

```
docker build -t cs526/imagesharingsecurity .
```

And run the application (all one line):

```
docker run -it --rm --network cs526-network -p 8080:8080 -p 8181:8181
     -e "ASPNETCORE_HTTPS_PORT=8181"
     -e "Credentials__ApplicationDb__User=sa"
     -e "Credentials__ApplicationDb__Password=database-password"
     --name imagesharing cs526/imagesharingsecurity
```

The webapp will by default be exposed at HTTP port 8080 in the container, and the environment variable `ASPNETCORE_HTTPS_PORT` is set to be port 8181 for HTTPS. Both of these ports are exposed on your laptop with the `-p` options. The credentials for accessing the database are provided as environment variables that are provided as configuration information to the app. The running container is given the name `imagesharing`. The logs for the container are output when you run it interactively, or you can view it in another terminal window using docker logs:

```
docker logs imagesharing
```

Submission

Submit your assignment as a zip archive file. This archive file should contain a single folder with your name, with an underscore between first and last name. For example, if your name is Humphrey Bogart, the folder should be named Humphrey_Bogart. This folder should contain a single folder for your solution named `ImageSharingWithSecurity`, with the complete source code (including assets) for the working ASP.NET project.

In addition, record mpeg videos demonstrating your deployment working in Docker. Demonstrate:

1. Registration of a user.
2. Automatic redirection to login upon an attempt to perform an action that requires authentication.

3. Prevention of a user from doing a restricted action (e.g., administrative user trying to view images or ordinary user trying to manage users).
4. Deletion of a user and their images. Show the user being prevented from logging in, and their images no longer available, after they are activated, and the user being able to log in again after being reactivated.

Make sure that your name appears at the beginning of the video, for example as the name of the administration user who manages the Web app. *Do not provide private information such as your email or cwid in the video.* Be careful of any "free" apps that you download to do the recording, there are known cases of such apps containing Trojan horses, including key loggers.