CS526 Enterprise and Cloud Computing
Stevens Institute of Technology—Fall 2020
Assignment Two—Data Models and Databases

Description:
This assignment requires the use of Visual Studio 2019 Community Edition and ASP.NET Core MVC 3.1. 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); you will need to update the `_ViewStart` configuration view to specify that this be used as the template for all views.

For this assignment, you will work with three models that are stored in a database:
- The `Image` model saves information about uploaded images: caption, description, date picture taken, location of image file (on the server), identity of uploader, and also a tag for the image. Every image has exactly one tag.
- The `User` model saves user information, which in this case is just username, an indication if they are visually impaired, and also a list of all the images this user has uploaded.
- The `Tag` model saves information about image tags, which in this case is just the tag name, and also a list of all the images that have this tag.

Since Entity Framework Core is no longer distributed with ASP.NET Core, you will need to install it from the

Use the `LocalDB` development database manager. The connection string is already set up in `appsettings.json`. *You will need to configure the `ApplicationDbContext` database context as a service in the startup class, and invoke the initialization logic in `ApplicationDbInitializer` (adding your own code to add users and tags).* The database context is then injected into the controllers using dependency injection.

Create three controllers, `Home`, `Account`, and `Images`.
- There is a default action, `Index`, for the `Home` controller, that displays a welcome message, personalized for a logged-in user.
- There is an action, `Register`, for the `Account` controller, that allows a user to register themselves. This action will create a record in the database table to store user information for this user, and automatically logs them in. This action should fail (gracefully) if the specified username is already in the database. *You should complete the logic for adding a user record if they do not already exist in the database.*

- There is another action, `Login`, for the `Account` controller, that allows a user to save a cookie identifying themselves in their browser. The username they specify must already exist in the database. If the user indicated when they registered that they are visually impaired, then whenever they log in to the Web site, it should display text in a large font. Do not bother with password-based authentication for now, we will consider alternatives for authentication in a future assignment. *You should complete the operation for saving information about the currently logged-in user in cookies.*
- The `Images` controller provides an action called `Upload`, that allows a logged-in user to upload an image. In addition to the caption, description, date taken and image file, the user should provide a tag for the image, chosen from a dropdown list. This latter list in turn is populated from the tags defined in the database. The user should not explicitly specify an image identifier, instead this is generated automatically when the image is added to the database. If there are validation errors, then the original data should be retained in the form, to allow the user to make amendments. *You should complete this action to upload the file and save its metadata in the database. You will need to complete the view for this action.*
- The `Images` controller provides an action called `Details`, that shows the details of an image identified by its image identifier (as part of the URI). The detailed information includes image identifier, caption, description, date taken, image tag and uploader (as well as the image itself
- The `Images` controller provides an action called `Edit`, that allows the details of an image, identified by its image identifier, to be modified by the user. The details that may be modified include caption, description and date taken. The logged-in user must be the same as the user who originally uploaded the image. *You will need to complete the view for this action.*
- The `Images` controller provides an action called `Delete`, that allows the details of an image, identified by its image identifier, to be deleted by the user. The logged-in user must be the same as the user who originally uploaded the image. *You should update this action (on a Get) to display the image and metadata in its view for confirmation, and (on a Post) to delete the image from the database.*
- The `Images` controller provides an action called `ListAll`, that displays a table with the caption, tag (name) and uploader (username) of every image in the database. Each row in the table should have links for the `Details`, `Edit` and `Delete` actions. The `Edit` and `Delete` links should only exist for those rows where the image was uploaded by the logged-in user.). *You will need to complete the view for this action.*
- The `Images` controller provides an action called `ListByUser`. This displays a form where a username is chosen from a dropdown list, and displays a table of all images uploaded by that user, similarly to the `ListAll` action. *You should update this action (on Get) to provide a dropdown list of all users, and (on Post) to display of all images uploaded by that user. You will need to complete the view for this action.*

- The `Images` controller provides an action called `ListByTag`. This displays a form where a tag is chosen from a dropdown list, and displays a table of all images with that tag, similarly to the `ListAll` action.
- Attempting to perform an image action without logging in should result in redirection to the login page.
- Handle any processing errors such as input-output failures on the `server`, by redirecting the user to a friendly error page (action `Error` of the `Home` controller).

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 `ImageSharingWithModel`, as well as the solution for your project.

In addition, record mpeg or Quicktime videos demonstrating your deployment working. 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.