Spring 2018
.NET Web Application Development
Final Project
Rev 1.0

This document contains instructions for the final project. Deadlines:

- The Entity Framework data model and business logic are due Wednesday, April 18, 2018 at 11:55 PM. Submission is mandatory and all parts must be submitted.
- Because there is no class on April 16, groups are welcome to use the classroom and the class period to meet and work on this assignment. **No interaction between groups is permitted.**
- A group status report is due Wednesday, April 25, 2018. Details will be forthcoming.
- Each group will present their project to the class on May 7, 2018, as it exists at the time. The project is expected to be functional and mostly complete by this date. The completed project is due for grading on Friday, May 11, 2018.
- Should it become necessary to issue subsequent revisions of this document, revision numbers are identified at the top of the document, beginning with revision 1.0. Should later revisions become necessary, the main intent shall be to refine and clarify the initial requirements, but not to change or materially expand the requirements. If additional revisions are released, they will be announced on the NYU Classes course web site (and also via email).

1 Group Project

Each group will be required to implement a fully functional web application using either web forms or MVC. The group must implement the web site according to the requirements stated below.

The requirements state *what* features must be present, but your group chooses *how* each feature is to be implemented unless explicitly stated. These requirements specify the absolute minimum functionality required to be present, functional, and well-designed, for the purposes of grading. However, the requirements are not exhaustive and not intended to be exhaustive. The group is expected to make reasonable choices within the scope of the requirements below, and use common sense and good judgment in determining what other features should be included over and above the minimum requirements.

Although the group will be working together as a single unit, each person is individually responsible for contributing meaningfully to the project. Each student's contribution will be ascertained for grading purposes through feedback from other group members as well as through version control inspection.

With respect to grading of the final project submission, emphasis will be placed largely on web site functionality, good design, and operational correctness, rather than appearance and performance. Groups should be mindful of obvious security considerations, and should be taking measures to avoid at least the XSS and CSFR attacks, discussed in class.

2 Bitbucket

Your group's first task is to create a Git repository in Bitbucket. This will serve as your group's repository for the remainder of the course.

To do this, each member of the group should first create an account at `bitbucket.org`. Although not required, I *strongly* recommend that each member of the group practice creating a repository in their Bitbucket accounts, and also practice creating projects in Visual Studio, linking them to the repository, and using the repository to commit code. This practice exercise should be undertaken before making any commits to the official group repository, described below.

The group leader should create a team consisting of the group members, then create a private repository owned by the team. This will serve as the group's "official" repository for the purposes of the course project and grading. Only commits relating to official group work (including the later parts of this assignment) should be made to this repository. This should not be used as a test or practice repository. All members of the group are individually responsible for understanding how basic Git version control works prior to collaborating with the official group repository.

Once the official group repository has been created, the group leader should add all members of the group to the project with "Write" access. Additionally, "Read" access should be given to user `praneethy91`. The group leader should then submit the URL of the group's official repository as part of the final project submission. This URL can be obtained from the repository's home page in Bitbucket.

2.1 Project Requirements

Craigslist (www.craigslist.org) is a web site that allows users to post items and services for sale. It is particularly notable for its simplistic user interface. The final project will be to create a web site in the same spirit as Craigslist.

The system will be comprised of the following core components:

- There will be 3 main user roles: *admin*, *user*, and *anonymous*. The first two roles require an account on the system. Anonymous users are those who use the web site without logging in. The user role allows one to create, modify, and delete posts, read other people's posts, and respond to posts. The anonymous role can read posts, but cannot create or respond to them.

  The admin role is is for web site administrators to set up and configure the web site, but can also access all of the same functions as the user role.

- A *post* is a single listing for an item or service. From the standpoint of data modeling, a post must have at least a unique identifier,timestamp, expiration, owner, title, body, location, type, and picture (4-person groups only).

- A timestamp is the date and time that the post was created or last modified.
- An expiration is the date and time on which the post will expire. Your site can decide when this is (e.g. 5 days from the creation of the post).
- An owner is the user who created the post.
- A title is a short description of the thing being advertised.
- A body is a longer detailed description of the thing being advertised.
- A location is broken into *area* and *locale*. Area describes a large geographical region (e.g., state), whereas locale describes a specific location within the area (e.g., city or county)[1].

  Since RESTful designs will likely show locations as part of the URL, it is advisable for locations to have a shorter URL-friendly slug. For example, a hypothetical area "New York" might have the slug "new-york." The slug can be entered manually when the location is created, or generated automatically. Be mindful of the legal characters for URLs when validating or generating URLs.
- A type is broken down into *category* and *subcategory*. Categories are named groupings of related subcategories. Categories might include housing, jobs, or services. The "housing" category, for example, might have the following subcategories: houses, apartments, offices, or vacation rentals. See the real Craigslist site for further examples. The same advice above regarding slugs also apply to post types.
- Note: types should exist independently of locations. That is, in the object-oriented sense, types don't "have" locations and locations don't "have" types.
- Groups of 4 only: the user should optionally be able to upload and store a single picture as part of the post.

- The home page of the web site should be accessible to all user roles and have the following functionality:
  - All categories and subcategories currently in the system must be shown on the page, unless hidden (more on hiding later). For each category shown, the list of all non-hidden subcategories should be grouped beneath it. All categories and subcategories must hyperlink to the *categories* screen, discussed below. The posts themselves should not appear on the home page.
  - The user should be able to select a location (both area and locale) that corresponds to where they want to search for posts. The site must prominently identify the currently selected location, preferably near the top of the page. The selection mechanism should show the areas and locales in alphabetical order. The user should be able to select either an entire area or one specific locale within a specified area. Selecting a location will not directly effect what is shown on the home page. However, if the user clicks any category or subcategory on the home page (to go to the *categories* screen), the posts shown on these screens should be restricted to only those posts within the location selected on the home page.
  - The web site can set the "default" location to an arbitrary area and/or locale (e.g. first one in each list.) For extra credit: your site should attempt to infer the user's location based on the IP address of the HTTP request and auto-select the area or locale, to the extent it is possible. See this site for ideas about geolocation services. If the location reported by the geolocation service does not correspond to any area or locale, then the site may fall back to the default functionality described above. If you implement this feature, you must ensure that it works on at least some of the locations in your database. Your project submission should document these tests by indicating the IP address(es) and corresponding locations you tested. It is okay to put this information in source code comments near any calls to the geolocation service.
  - There should exist a search bar that allows the user to search posts for the presence of *all* keywords, in any order. The title and description properties of each post should be considered for the purposes of searching. A button to the right of the search bar should execute the search and take

[1] On the real Craiglist, the area is determined from the subdomain. For example, `newyork.craigslist.org` refers to the New York area. Only locales are shown on the screen. In this solution, both area and locale will be shown on the screen, since we have not discussed subdomains.

the user to the *search results* page, discussed below. The selected location on the home page must be transmitted to the search results page.

  - If the user is logged in (admin, user), links or buttons must exist for the following:
    * Creating a new post. The selected location information from the home page must be transmitted to the *create post* page (discussed below) so that the location input fields are pre-populated with the selected location. The user should be able to change the selections. If the user comes directly to the page or the location is not transmitted to the page for any reason, the user needs to select the location manually.
    * Viewing existing posts. This should take the user to the *list posts* screen, discussed below.
    * Viewing the inbox. See the *inbox screen* below for more details.

    If the user is not logged in, there should exist some kind of messaging indicating that the user can log in to create posts.

- The *list posts* screen allows a logged in user to see a list of their own unexpired and expired posts, either separately or in one list. This screen requires authorization. From this screen, a user should be able to create, modify, or delete active posts. Expired posts can only be viewed, but not modified or deleted. Posts should not be shown as expired if they were deleted before they expired. Creating posts should lead to the *create post* screen, discussed below. Modification is discussed below. Deleting a post should lead to a delete confirmation screen (not discussed here). The posts on the screen must be paginated.

- The *create post* screen allows admin or user to create a new post. This screen requires authorization. It must solicit all of the fields described above except the unique identifier and date/time. The unique identifier must be computed internally, the owner must be inferred from the current login, and date/time field must be set to the date and time of the post. The expiration should be computed based on the date and time of the post creation. All other fields are entered by the user and are required.

- The *modify post* screen is similar to *create post*, except that the user can edit an existing post. The screen should not allow deleted or expired posts to be modified. The owner, and unique identifier must remain the same, but the time stamp should be updated to reflect the time of the modification. Owner, expiration, and unique identifier should be shown on the screen, but cannot be editable and **must be ignored** when posted back to the server. The time stamp should not be shown on the screen and it too must be ignored by the server. Not ignoring these fields could result in an over-posting attack. I suggest familiarizing yourself with over-posting attacks, what they are, and how to prevent them. See the `[Bind]` annotation for ideas.

- The *categories* screen shows posts for the selected type and location. If a category is specified without a subcategory, then it should show all posts in the category (spanning all subcategories). If a category and subcategory are specified, only posts within the category and subcategory should be shown. Posts should further be filtered by location in a similar way: if an area is specified without a locale, all posts for the area (spanning all locales) should be shown. If both an area and locale are specified, then only posts that were created within the area and locale should be shown. To be clear, posts must pass through *both* the location and type filters above in order to be shown on this screen.

  - This screen is viewable by all three roles.
  - The screen must be paginated.
  - The page must receive information concerning at least the category, subcategory (optional), area, and locale (optional) as input. The location and post types must be shown on the screen and must be selectable so that the user can change the location or post types they wish to see when on the page. Any received information must be used to pre-select the location or post type.
  - For 3-person groups: must have a *list view* of posts. This is a text-based list (with hyperlinks) of posts that, when clicked, take the user to the *view post* screen (discussed below). Pictures are not shown in the list view. For 4-person groups, the user should be able to select between the just-described list view or a second *gallery* view. The gallery view should show both the

picture and title of each post, organized 3 posts per row. Pictures should be shown at a uniform size—consider implementing a filter for this purpose.

  - For the admin and user roles only, this screen must hyperlink to a *response screen* where the user can respond to a post.
  - For 4-person groups, the admin and user oles must allow for the ability for any person in the admin or user role to flag inappropriate posts. This can be as simple as clicking a link somewhere in the posting. Each time a post is flagged in this way, the web site should algorithmically decide if the post should be removed from the site. (For example, after 5 users flag it). If the algorithm decides to remove it, it should no longer be shown from that point on, and it should also be removed from the originator's list of posts.
  - Posts must be shown in reverse chronological order (most recent first) in both views above.

- The *response screen* allows a person in the admin or user role to respond to a post. There should be a separate This screen should include an ability to write a freeform message to the originator of the post and send. There should be an ability to return back to the screen from which the user originated.

- The *search results* page can be reached from the home page. It should be the same as the *categories* screen, except that it should also have a search bar at the top to execute additional searches, which should direct the user back to the same screen. All of the same requirements of the *categories* screen apply here too.

- The *inbox screen* allows a user to see responses to their posts. (Responses to posts are private communications between the reader of the post and the originator of the post.) Responses to a post can continue to exist, even if the post itself is removed. It is at each group's discretion to decide if further two-way communication should be allowed in an email-like manner. Extra credit will be rewarded if the group decides to implement these expanded capabilities.

- Users in the admin role must be able to perform the following activities in addition to everything described above. We leave it to each group to decide what specific screens should be implemented, and how:

  - Change the role of any person in the user role to the admin role.
  - List, add, modify, and hide categories and subcategories
  - List, add, modify, and hide areas and locales
  - List, add, modify, or delete posts for any user on the system

  Hiding is similar to deleting, except that it isn't removed from the database, but is rather hidden from view so as to give users the impression that it were deleted from the database. It is okay to implement "deleting" as hiding too if the group wishes.

  The first user to be assigned an admin role can be done so using one of these two approaches:

  - Use the convention that the very first user on the system is admin by default. Any other users in the admin role must be set through the administration feature above.
  - Manually set the role of the first admin directly in the database, rather than through the application. Changes of additional users to the admin role must be possible through the web site, however, and only by users in the admin role.

  Your submission must document the username and password of at least one admin so that the admin functionality can be tested. Please include a file CREDENTIALS.txt as part of your submission.

2.2 Other Requirements

1. The application must be an ASP.NET Web Forms application (not an ASP.NET website) or an ASP.NET MVC application. MVC is strongly encouraged. All projects must use Entity Framework for database operations. Use of the "code first" approach is required.

2. Your group may use pre-compiled third-party libraries and NuGet packages at will. No citation of sources is necessary in this case, since use of such libraries/packages can be easily determined. It is common to read technical tutorials and use code from the tutorial as a basis for your own implementation. This is fine as long as any code originating from others outside your group is properly cited and the use of these tutorials is reasonable and limited to very small portions of the overall project. All citations must be included in a file called CITATIONS.txt.

3. The final application submission must be published to Azure.

4. There must be one solution containing separate distinct projects for the user interface, business logic, and data model. The user interface layer is usually the ASP.NET Web Application. The requirement of having the business logic and data model separate is relaxed for the first milestone.

5. Business logic should not be performed in the user interface. Business logic methods should be called from the controller or web form code-behind. Business logic is expected to be thoroughly tested by the group before work on the MVC or Web Forms project begins. The use of unit test projects and unit testing is strongly encouraged, but not required for the first milestone. This topic will be discussed in the remaining weeks of the course.

6. The solution should use database transactions when it makes sense to do so. That is, several logically related operations should be grouped into a transaction. Totally unrelated operations should not. Remember that Entity Framework can create implicit transactions, as discussed in class.

7. The system must use a common layout/theme across all of the pages and look like a finished product—not a work in progress. It should not look like several different web sites rolled into one solution. The real Craigslist web site is not terribly beautiful to look at. Your site does not need to be either. However limited in appearance, it must nevertheless still be properly styled and contain valid HTML. The web site itself must have a name. Every page must have a title. Sloppily organized screens or sites (e.g. overlapping elements, non-uniform spacing, etc.), sites that are completely unimaginative, or sites in which nobody bothered to change the default navbar or the copyright message (for example) are all unacceptable. The web site is the product of your group—make sure it appears that way. Be sure to confirm the validity of your HTML using the HTML validator.

8. All views/pages must meet these requirements:

   - Page is straightforward to understand and use by an ordinary lay person.
   - Page should contain a reasonable amount of content. This means not too little and not too much per page.
   - Visual elements are reasonably spaced and laid out on the page in an intuitive manner.
   - Page input is validated, HTML encoding is performed where necessary.
   - Page is tested and works in all reasonably known cases. No broken links, exceptions, or any other anomaly should be visibly present.

2.3 Data Model and Business Logic Milestone

The Entity Framework data model and business logic are due Wednesday, April 18, 2018 at 11:55 PM. Submission is mandatory and the entire data model and business logic for all parts of the requirements must be submitted. Only the group leader must submit the deliverable on behalf of the group.

Business logic refers to any calculations or computations, apart from data model validation, that your project relies upon to perform non user interface related functions. You've been writing business logic the

entire course. With Web Forms, your business logic was coded directly in each of the page code-behind files. With Homework 4, your business logic was (most likely) coded directly into the controller.

This time, we are going one step further to divorce it from any particular framework or user interface mechanism. When this portion of the project is completed, one should hypothetically be able to take your data model and business logic, and move it into a mobile app, Windows application, console application, or any other type of application, where the only remaining step is to work on the user interface. There should be no assumptions regarding the use of MVC or any other particular design pattern.

As you are aware, the Entity Framework data model already includes functionality for performing basic operations such as adding objects to containers and managing the relationship between objects in the database. There is no need to artificially wrap any of that functionality inside your business logic. Your business logic should mainly be focused on any other functionality *not* provided by Entity Framework that is specific to the application. This might include:

- Routines for setting and applying location and type filters
- Routines for querying all posts matching specified filters
- Routines for searching for posts by keyword
- Routines for responding to posts

And so on. The data model and business logic should jointly be able to perform every single computational function that the web site will need to be perform—all without a user interface.

We have not formally discussed testing to date, so unit test projects will not be a requirement at this time. However, your console application should test the business logic layer (i.e., the routines itemized above) in a meaningful way by populating all of the entities with data and calling all of the major routines. Although the business logic does not necessarily have to be 100% bug-free at this juncture, it should be devoid of major issues and be usable (e.g. should not throw exceptions or exhibit any other unstable behavior).

At this stage, you may place your **using** clauses and database operations *inside* the business logic. Since the business logic depends on the model, this is generally not an issue. However, a well-designed site would further separate database operations from business logic. Later in the course, I will discuss strategies for accomplishing this. However, such separation is not a requirement for at least this first milestone.

Shortly after the deadline, feedback will be provided to the group in an informal manner. Although a formal grade will not be issued, failure to submit the deliverables or failure to submit deliverables that are not generally in the expected state as described above, may be subject to penalties. Groups should not wait for feedback before proceeding with the remaining portions of the project. Groups can make modifications to the data model or business layer at a later itme, in response to any feedback that is received.

The deliverables for this milestone are as follows:

- A ZIP file containing *one* solution with two or three projects:
    1. A Console Application project that demonstrates the data model and business logic. This must call into all of the major routines and demonstrate—through the use of messages on the standard output—the business logic and model layer working. If you already know what Unit Test Projects are, you may use a Unit Test project in lieu of a Console Application. Whichever you choose, this piece needs to initialize the first user in the admin role so that functionality for people in the user and admin roles can be demonstrated.
    2. A Class Library (.NET Framework) project consisting of the data model. This will yield a .DLL file.
    3. A Class Library (.NET Framework) project consisting of the business logic. You may combine the data model and business logic into one project for this first pass, but your group will ultimately be expected to separate them. This too will yield .DLL files.
- An inline submission indicating the Bitbucket repository for the group. All persons in the group are expected to have Bitbucket accounts and be actively contributing to the project. Source control must be used during development.

- You should let Entity Framework create the database for you and populate it as necessary using code. Do not attach a pre-existing database with your submission. Your code is responsible for seeding the database with initial data. You can do this inside the *Seed* routine that is created when you execute `Enable-Migrations` from the package manager console, although you could do it elsewhere too.
- To the extent that your group needs to cite outside code, include CITATIONS.txt in your solution ZIP file. I recommend checking it into your version control repository and maintaining it there.

In the case that you decide to split business logic and data model projects, the business logic can depend on the model, but not vice versa. To create a dependency from one project to another, right-click on *References*, select *Add Reference...* and select the dependency. If the data model and business logic are in different namespaces, don't forget to reference the data model namespace from within the business logic code.

Any questions concerning the project requirements should be directed to the NYU Classes forum for further clarification.