SE 407 (Advanced .NET)

Lab #1:  
Part 1:  
Create a MS Visual Studio 2015 ASP.Net Core Website Application (.NET Framework).  This web application will be the basis for all assignments due for this course.  The web application will need to use the .NET Framework 4.6.1 or higher, 4.6.1 recommended.  Create the project using the following steps:  
• Create a new project by selecting File | New | Project

t• Under Visual C#, select Web | ASP.NET Core Web Application (.NET Framework)

• Enter project name: SE406\_YourLastName

• Enter location to save (keep this readily accessible!) and click OK

• Under Select a template, select Web Application and click OK

• Remove the Project\_Readme.html

• Add a Models folder to src/Application Root

Part 2:  
You will need to apply responsive styling to your web application using CSS and HTML design elements and a responsive framework (Bootstrap) in the \_Layout.cshtml.  This will be the layout file thatall view content will use for your web application.  You can refer to the in class demo/hands on work and/or demo website on Canvas.  Make this “Branding” reflective of the state your DOT BridgeMaintenance application will be managing as well as the vendor of the application.  The vendor can be the same state organization or it can be a fictitious company you create.  It should be clear to the user what state/department/vendor is represented by the style elements/branding applied.  You will need to create 3 distinct sections in your \_Layout.cshtml page:• A header section with images, branding, etc.• A main content section• A widget content section  
The layout will be of your design and choosing.  The only guidelines are as follows

:• The header must contain the bootstrap navigation menu or be directly under it for fixed-top

bootstrap menus

• The content section must contain the RenderBody() script• The widget section must be big enough to display some informative text information or list of links, etc.  
• You must use Bootstrap and you must use a Bootstrap theme other than the default Bootstrapinstall (Bootswatch, etc.)

• You must make a concerted effort to brand and theme your layout.  Minimal effort or copies of known templates will receive a low grade.

Lab #2:

Create MVC controllers that will be used for our project. Complete the following steps:

• Add a Controllers folder in the src/root folder of your solution if it does not exist

• Add the following MVC Controller Classes to your Controllers Folder:

1. Bridge2. Inspector3. Inspection4. Maintenance

• For each of the above Controllers, add the following IActionResult methods:

1. Index()2. Error()

• In each of the IActionResult methods, create a “Title” and “Message” item in the ViewDatacollection.  These should indicate their perspective controllers.  The Title item should indicate the name of the controller and Message item should indicate the name of the pages controlled bythe controller.  (For example, Bridge: Title=Bridge, Message=This is the Bridge controller page)  
• For each of the MVC Controller Class created above, add folder to src/root/Views folder• In each of the View folders created above, add an MVC View Page named Index• On each of the Index.cshtml files created, add the ViewData Title and Message items in the format shown on Vews/Demo/Index.cshtml• Add a <li> to your Bootstrap menu in \_Layout.cshtml for each of the MVC Controller Classeswith the controller set to the proper name and the action set to Index.

Lab #3:

• Run the Step\_1\_Drop\_Objects.sql script on Canvas against your database

• Run the Step\_2\_Create\_Objects.sql script on Canvas against your database

• Run the Data\_Dictionary.sql script from Canvas  
• In the results pane of the Data Dictionary script, right click and choose “Select All”, right click again and the choose “Copy with Headers”.  Paste the results into MS Excel and save locally for

use as your data dictionary.

• In your MVC application project, create new Model classes for each table created in the BridgeMaintenance database script.  Ensure the names and data types match.  A Model is simply a C#class.  Each Model will need to have Auto Implemented properties for each column in your database table(s).

Lab #4:

Refer to the EF\_Demo application posted to Canvas for reference.  The goal of this lab is to createthe initial Entity Framework components for our Bridge Tracking application.  You must havecompleted Lab 3 first as your domain object models are needed for this lab.  Follow these steps tocreate the basic Entity Framework components for each of your Domain Objects:

• Bridges

• ConstructionDesigns

• Counties

• FunctionalClasses

• InspectionCodes

• Inspections

• Inspectors

• MaintenanceActions

• MaintenanceRecords

• MaterialDesigns

• StatusCodes

1. Using the NuGet package manager, add the following packages to your solutions:  
"Microsoft.EntityFrameworkCore": "1.0.1", and "Microsoft.EntityFrameworkCore.SqlServer":

"1.0.1".  
2. Add a ConnectionStrings section to your appsettings.json file.  The connection string will need your specific database access credentials.

3. Add an item to the buildOptions section of your project.json file.  The item to add is:  
“copyToOutput”: “appsettings.json”;  this will make the appsettings.json file available to all of

your DBContext classes at runtime.

4. Create DbContext classes for each domain object above.  The class file will reside in the Modelsfolder and be named ObjectDBContext.cs.  For example, Bridges will be BridgesDBContext.cs.Ensure the class inherits the DbContext class from the EntityFramework.  These classes will define the data source context for which EntityFramework will use.  It is defined for each domain object since they can be from different data sources.  Add public properties for IConfigurationRoot and DbSet<DomainObject> to each DBContext class.  Add a constructor that defines the configuration for each DBContext class.  Next, add an override function forOnConfiguring that will define the connection string to be used to each DBContext class.  SeeEF\_Demo on Canvas or your in-class demo for specific examples.

5. Modify the Startup.cs class in your solution to use the EntityFramework.  Add anEntityFramework service to your IServiceCollection for one of the DBContext classes(BridgeDBContext) created in step 4. See EF\_Demo on Canvas or your in-class demo for specific examples.

6. Create a ViewModel for each of the domain objects listed above.  These ViewModel classes will be created in your Models folder.  Name each class as follows:  DomainObjectViewModel. For example, Bridges will be BridgesViewModel.cs.  In each of the ViewModel classes, define two public properties:  a List<DomainObject> and single Domain Object.  For example, Bridgeswill have: public List<Bridge> BridgeList {get; set;} and public Bridge NewBridge {get; set;}.See EF\_Demo on Canvas or your in-class demo for specific examples.

7. Create MVC Controllers for each of your Domain Objects above.  These controller classes will be created in your Controllers folder.  Name each controller as follows:DomainObjectController.  For example, Bridge controller will be BridgeController.  In each of  
your controllers, modify the GET Index() IActionResult to use the model’s view model and db  
context class and return the view model to view.  Next, add a HTTP POST Index()IActionResult that receives the view model as a parameter and uses the db context class to add a new item to the database.  See EF\_Demo on Canvas or your in-class demo for specificexamples

.8. Create a folder in your Views folder for each of the domain objects above.  Within each of thefolders, add a new MVC View Page named Index.cshtml.  In each of the Index.cshtml view pages, create a list of fields to be used to add new domain objects by clicking a submit button and add a table of all domain object records that currently exist in your database tables.  Ensure these elements are encased in Bootstrap responsive framework CSS classes.  See EF\_Demo onCanvas or your in-class demo for specific examples.

Lab #5:

Refer to the EF\_Demo application posted to Canvas for reference.  The goal of this lab is to createvalidation for our Bridge Tracking application Models.  You must have completed Lab 3 first asyour domain object models are needed for this lab.  Follow these steps to create the validationneeded using Data Annotations for each of your Domain Objects:

• Bridges• ConstructionDesigns• Counties• FunctionalClasses• InspectionCodes• Inspections• Inspectors• MaintenanceActions• MaintenanceRecords• MaterialDesigns• StatusCodes

 Run the following script against your database if you haven’t already:

--status codessp\_rename 'StatusCodes.StatusId', 'StatusCodeId', 'COLUMN'GO--maintenance recordssp\_rename 'MaintenanceRecords.MaintenaceId', 'MaintenanceRecordId', 'COLUMN'GOsp\_rename 'MaintenanceRecords.MantenanceProjectedCost', 'MaintenanceProjectedCost', 'COLUMN'GOALTER TABLE [dbo].[MaintenanceRecords]  WITH CHECK ADD  CONSTRAINT[FK\_MaintenanceRecords\_MaintenanceActions] FOREIGN KEY([MaintenanceActionId])REFERENCES [dbo].[MaintenanceActions] ([MaintenanceActionId])GOALTER TABLE [dbo].[MaintenanceRecords] CHECK CONSTRAINT[FK\_MaintenanceRecords\_MaintenanceActions]GO

This will rename some columns that are misnamed/misspelled and add a missing foreign key constraint.

Part 1: Data Annotations

To validate our Models, each model must have the following Data Annotations:• All Data Annotations must have a descriptive error message defined

• All columns in each table/model that are NOT NULLABLE in the database schema must have a Required annotation

• All NON NULLABLE string columns must have a MinLength annotation set to

• All string columns must have a MaxLength annotation set to the max length of the column

• All foreign key Guid/Uniqueidentifier columns must have a Required annotation Model Specific Data Annotation Requirements  
• The Bridge model Built property will need a Range annotation to set the year from 1600 and2025

• The Bridge model Reconstructed property will need a Range annotation to set the year from1600 and 2025  
Part 2: Validation/Error Messages in MVC View Pages

We need to modify our MVC View Pages to display the error messages that our Data Annotations will now generate when a form field is invalid.  Follow these steps for each Model that you added DataAnnotations to:  
• For each Model/form field in your MVC View Page, add a span tag that will be used to display the error messages.  The asp-validation-for Tag Helper will be set to the same Model.Property as the Html.TextBoxFor element.  The span tag will need to have a CSS class applied that represents a Bootstrap text designator, such as text-danger.

• In your controller for each model, add a condition check (if ModelState.IsValid) to each HTTPPOST Index() IActionResult to check the model state for validity before any save changes actions are called.  Ensure your return statement is outside of this conditional check.Ensure that the script tags for jquery.validate.js and jquery.validate.unobtrusive.js are referenced  
at the bottom of your \_Layout.cshtml for validation to work properly on the client side.

Lab #6:           Due: Friday Week 8

Part 1: Drop Down Lists for Foreign Keys/Lookup tables in Models/Views  
We need to create Drop Down Lists for our lookup tables in the following Models:• Bridge• Inspection• MaintenanceRecord

To add this functionality, follow these steps (example in demo posted on Canvas):• In your domain object ViewModel class ( example: BridgeViewModel) for each model above add a public property of type List<SelectListItem> (T=Model) representing each foreign key in

the model.  For the Bridge model, there are 5, the Inspection model has 5, and the MaintenanceRecord model has 1.  For example, the Bridge model has MaterialDesginId first.The property to add to the BridgeViewModel would be public List<SelectListItem>MaterialDesigns {get; set;}.• Each controller for the models above, modify the GET Index() IActionResult to populate eachproperty created in the step above by means of a private static function that returns the specifiedList<SelectListItem>.  The code for populating this exists already in the foreign key view model,you just have to call it locally in the model that has the foreign key property.  Using the step above as an example, MaterialDesignViewModel should already have a property of typeList<MaterialDesign> names MaterialDesignList.• Next, in the MVC View Page for each Model (Index.cshtml), add a select list to the edit form.  Itshould have the same bootstrap styling applied and also contain the Html.LabelFor designation.The asp-for Tag Helper for the html select control should be set to the Model.Property where theModel.Property represents the foreign key, for example, Model.Bridge.MaterialDesignId.  The asp-items Tag Helper for the html select control should be set to the List<SelectListItem>property in the Model, for example Model.Bridge.MaterialDesigns.  Each html select control should have a static option added with the value set to an empty string and text set to “Choose….” representing the item you wish them to select.• Finally, using Linq, modify the columns in the html table used to display all current records to show the text value of the foreign key instead of the Guid.  Note, you do not have to display all properties for each model in the MVC View Page tables.  Select a subset including 4-6 columns that best represent each row in the table.  For models with less than 4 properties, show all properties in the table

Lab #7:

We need to edit records in all of our Models:To add this functionality, follow these steps (example in demo posted on Canvas):

• In each of your Views folders (Bridge, etc.) modify the Index.cshtml form to include an edit link in the table displaying all current records for that particular Model.  Achieve this by using TagHelpers in a standard HTML Anchor tag.  Add a Tag Helper for asp-action with the value set to “Edit” and a Tag Helper for asp-route-id with the value set to the @model primary key property for example, Bridge would be @bridge.BridgeId).  Set the text of the HTML Anchor tag to

“Edit”

.• In each of your MVC Controllers, create a public IActionResult for an HTTP GET action named  
Edit.  The GET Edit IActionResult will need to accept a GUID parameter named “id”.  Within  
this method, instantiate the appropriate View Model and set the new object property within this View Model equal to the Model the id parameter represents.  This is achieved using the Linq Where filtering clause.  Return the View Model to the View.

• In each of your MVC Controllers, create a public IActionResult for an HTTP POST action named Edit.  The POST Edit IActionResult will need to accept a View Model parameter of the appropriate type.  Within this method, use conditional logic to ensure the ModelState is valid before continuing with the POST action.  Once a valid Model state is confirmed, instantiate the  
appropriate Model from the View Model parameter.  Next, set the instantiated Model’s Id property from the RouteData.Values “id” item using the Guid.Parse method.  Continuing, flag  
the instantiated Model Entry State in the DB Context as Modified.  Save the DB Context changes and redirect to the Index action.

• Create a new MVC View Page in each your Views/Model folders.  Name each of these MVCView Pages Edit.cshtml.  This view page is a copy of the top portion of your Index.cshtml, orjust the form inputs.  This will be used to populate the current record selected for editing.  You will have to modify the Html.BeginForm statement to address the correct action (Edit) and controller, the FormMethod will remain POST.  Finally, add an Html.ActionLink near the form’s submit input that redirects to the Controller’s GET Index IActionResult

Lab #8:  
First, we need to enable the Session middleware in out .NET Core applications.  Add the following NuGet package:

Microsoft.AspNetCore.Session v 1.1.1  
Next, enable the Session in both the ConfigureServices and Configure methods in your Startup.cs classfile.  This will enable us to add items to the session so they are accessible across multiple views.We need to conditionally delete records in all of our Models if a dependency doesn’t exist:To add this functionality, follow these steps (example in demo posted on Canvas):  
• In each of your Views folders (Bridge, etc.) modify the Index.cshtml form to include a deletelink in the table displaying all current records for that particular Model.  Achieve this by usingTag Helpers in a standard HTML Anchor tag.  Add a Tag Helper for asp-action with the value set to “Delete” and a Tag Helper for asp-route-id with the value set to the @model primary keyproperty for example, Bridge would be @bridge.BridgeId).  Set the text of the HTML Anchor

tag to “Delete”

• In each of your MVC Controllers, create a public IActionResult for an HTTP GET action named Delete.  The GET Delete IActionResult will need to accept a GUID parameter named “id”.  
Within this method, instantiate the appropriate View Model(s) and set the new object property within this View Model equal to the Model the id parameter represents.  This is achieved using the Linq Where/FirstOrDefault filtering clause.  If the model has no dependencies to other models, mark the DB context record as Deleted and save changes.  If there are dependencies,instantiate a list of target model and determine if there is a dependency constraint.  If so, opt out of the delete processing.

• Upon successful deletion of a record, notify the user on the View that invoked the action.  Upon  
discovery of a dependency constraint, notify the user that the item cannot be deleted due to a dependency.  The notification will be implemented by utilization of the TempData session storage container.

• On the Models Index view page, check TempData for a value and if it exists, display the message to the user.

Final Project

Using your cumulative project for this quarter, complete the following: Create a new Model class as SQL Server table for User.  Ensure the names and data types match the list below.  A Model is simply a C# class, you will have to create the table using MS SQL Server Management Studio.  The Model will need to have Auto Implemented properties for each column belowand the table will reflect the following schema:

•UserID Guid/Uniqueidentifier require (default value of new sequentialid() in SQL Server)

•FirstName string/varchar 25 required

•LastName string/varchar 25 required

•Email string/varchar 100 required

•Password string/varchar 255 required Create a DbContext class for the User model.  The class file will reside in the Models folder and be named UserDBContext.cs. Ensure the class inherits the DbContext class from the EntityFramework. Add public properties for IConfigurationRoot and DbSet<DomainObject> to each DBContext class. Add a constructor that defines the configuration for each DBContext class.  Next, add an override function for On Configuring that will define the connection string to be used to each DBContext class. Create a ViewModel for the User domain object.  This ViewModel class will be created in your Modelsfolder.  Name the class as follows:  UserViewModel. In the UserViewModel class, define two public properties:  a List<User> and single User.Create MVC Controller for the User model.  This controller class will be created in your Controllers folder.  Name the controller as follows:  User Controller. In each of your controllers, modify the GET  
Index() IActionResult to use the model’s view model and db context class and return the view model to  
view.  Next, add a HTTP POST Index() IActionResult that receives the view model as a parameter and uses the db context class to add a new item to the database. Create a folder in your Views folder for the User model.  Within the folder, add a new MVC View Page named Index.cshtml.  In the Index.cshtml view page, create a list of fields to be used to add new User by clicking a submit button and add a table of all User records that currently exist in your database table. Ensure these elements are encased in Bootstrap responsive framework CSS classes. To validate our User Model, each model must have the following Data Annotations:

• All Data Annotations must have a descriptive error message defined

• All columns in each table/model that are NOT NULLABLE (required) in the database schema

must have a Required annotation

• All NON NULLABLE string columns must have a MinLength annotation set to 5

• All string columns must have a MaxLength annotation set to the max length of the column We need to modify our MVC View Page to display the error messages that our Data Annotations will now generate when a form field is invalid.  Follow these steps for the User model that you added DataAnnotations to:

• For each Model/form field in your MVC View Page, add a span tag that will be used to display the error messages.  The asp-validation-for Tag Helper will be set to the same Model.Property as the Html.TextBoxFor element.  The span tag will need to have a CSS class applied that

represents a Bootstrap text designator, such as text-danger.  
• In your controller for each model, add a condition check (if ModelState.IsValid) to each HTTPPOST Index() IActionResult to check the model state for validity before any save changes actions are called.  Ensure your return statement is outside of this conditional check. In your User Views folder, modify the Index.cshtml form to include an edit link in the table displaying all current records for the User Model.  Achieve this by using Tag Helpers in a standard HTML Anchor tag.  Add a Tag Helper for asp-action with the value set to “Edit” and a Tag Helper for asp-route-id with the value set to the @model primary key property.  Set the text of the HTML Anchor tag to “Edit”. In each your User MVC Controller, create a public IActionResult for an HTTP GET action named Edit. The GET Edit IActionResult will need to accept a GUID parameter named “id”.  Within this method,  
instantiate the appropriate View Model and set the new object property within this View Model equal to the Model the id parameter represents.  This is achieved using the Linq Where filtering clause.  Return the View Model to the View.In your User MVC Controller, create a public IActionResult for an HTTP POST action named Edit. The POST Edit IActionResult will need to accept a View Model parameter of the appropriate type.Within this method, use conditional logic to ensure the ModelState is valid before continuing with the POST action.  Once a valid Model state is confirmed, instantiate the appropriate Model from the ViewModel parameter.  Next, set the instantiated Model’s Id property from the RouteData.Values “id” item  
using the Guid.Parse method.  Continuing, flag the instantiated Model Entry State in the DB Context as Modified.  Save the DB Context changes and redirect to the Index action. Create a new MVC View Page in each your Views/Model folders.  Name each of these MVC ViewPages Edit.cshtml.  This view page is a copy of the top portion of your Index.cshtml, or just the form inputs.  This will be used to populate the current record selected for editing.  You will have to modify the Html.BeginForm statement to address the correct action (Edit) and controller, the FormMethod will remain POST.  Finally, add an Html.ActionLink near the form’s submit input that redirects to the

Controller’s GET Index IActionResult.In your User Views folders modify the Index.cshtml form to include a delete link in the table displaying all current records for that particular Model.  Achieve this by using Tag Helpers in a standard HTMLAnchor tag.  Add a Tag Helper for asp-action with the value set to “Delete” and a Tag Helper for asp-route-id with the value set to the @model primary key.  Set the text of the HTML Anchor tag to “Delete”. In your User MVC Controller, create a public IActionResult for an HTTP GET action named Delete.  
The GET Delete IActionResult will need to accept a GUID parameter named “id”.  Within this method, 3  
instantiate the appropriate View Model(s) and set the new object property within this View Model equalto the Model the id parameter represents.  This is achieved using the Linq Where/FirstOrDefault filtering clause.  If the model has no dependencies to other models, mark the DB context record as Deleted and save changes.  If there are dependencies, instantiate a list of target model and determine if there is a dependency constraint.  If so, opt out of the delete processing.  Upon successful deletion of a record, notify the user on the View that invoked the action.  Upon discovery of a dependency constraint, notify the user that the item cannot be deleted due to a dependency.  The notification will be implemented by utilization of the TempData session storage container.  On the Models Index view page, check TempData for a value and if it exists, display the message to the user.