Lab: Implementing and using WebApi

# Scenario

You have been asked to implement the comments functionality, as follows:

* When viewing the details of a photo, a user should also see the existing comments for the given photo.
* Without leaving the page, the user should also be able to add a new comment
* The new comment should appear in the list of existing comments without refreshing the whole page.

To implement this feature you decide to create a RESTful service to manage the comments, then write a JavaScript client that will handle the user interactions, invoke the RESTful API and dynamically build the user interface client side.

# Objectives

After completing this lab, you will be able to:

* Create and implement a RESTful service through an Api Controller
* Create and implement a JavaScript client that makes use of the RESTful service

Estimated Time: 60 minutes

# Exercise 1: Create a RESTful service

## Scenario

You have been asked to implement the comments feature.

* Get the existing comments for the given photo.
* Add a new comment
* Get an existing comment by its id

You decide to create a RESTful service to manage the comments.

In this exercise, you will:

* Create an infrastructure layer to manage comments by implementing a repository
* Create a CommentsController as an ApiController
* Implement the actions to create and retrieve comments and configure their routes.

The main tasks for this exercise are as follows:

1. Create and implement a CommentRepository.
2. Create and implement a CommentsController.

### Task 1: Create and implement a CommentRepository.

1. In the Model folder, create an ICommentRepository interface.
   1. Define a GetCommentsForPhotoAsync method, accepting an int photoId parameter and returning a Task<List<Comment>>
   2. Define a FindAsync method, accepting an int id parameter and returning a Task<Comment>
   3. Define a CreateAsync method, accepting a Comment comment parameter and returning a Task
2. In the Model folder, create a CommentRepository class that implements a ICommentRepository interface
   1. Use Dependency Injection to get hold of the PhotoSharingApplicationContext
   2. Implement the GetCommentsForPhotoAsync method, by returning all the Comments of the PhotoSharingApplicationContext whose PhotoId is equal to the input parameter
   3. Implement the FindAsync method, by returning the first Comment in the PhotoSharingApplicationContext whose Id is equal to the input parameter
   4. Define a CreateAsync method, by adding the input comment to the PhotoSharingApplicationContext and saving the changes

### Task 2: Create and implement a CommentsController.

1. In the Solution Explorer, under the Controllers folder, select Add -> New Controller
   1. Select the WebApi Empty template
   2. Name the controller CommentsController
2. Use Dependency Injection to get hold of the context
   1. Create a private field of type PhotoSharingApplicationContext
   2. Create a constructor with a PhotoSharingApplicationContext parameter
   3. Save the parameter in the private field you created on step a.
3. Implement a GetCommentsByPhotoId action
   1. Let the action return a Task<List<Comment>> and mark it as async
   2. Accept a photoId as a parameter
   3. Configure the route of the action to be available at "/api/photos/{photoId}/comments"
   4. Invoke the GetCommentsForPhotoAsync of your CommentsRepository and return the result
4. Implement a GetById action
   1. Let the action return a Task<ActionResult<Comment>> and mark it as async
   2. Accept an id
   3. Use the HttpGet attribute to configure the route with an {id} parameter
   4. Invoke the FindAsync of your CommentRepository and return the result
   5. If no comment was found with the given id, return a NotFoundResult
5. Implement a Create action
   1. Let the action return a Task<ActionResult<Comment>> and mark it as async
   2. Accept a Comment parameter
   3. Use the HttpPost attribute
   4. Invoke the CreateAsync method of your CommentRepository, passing the Comment input parameter
   5. Return a CreatedAtAction result, passing the name of the GetById action, an anonymous object with an id property set to the Comment.Id, and the whole Comment input parameter

**Results**: At the end of this exercise, you will create a CommentsController RESTful service.

# Exercise 2: Create a JavaScript client

## Scenario

You have been asked to implement the comments front end, as follows:

* When viewing the details of a photo, a user should also see the existing comments for the given photo.
* Without leaving the page, the user should also be able to add a new comment
* The new comment should appear in the list of existing comments without refreshing the whole page.

To implement this feature you decide to write a JavaScript client that will handle the user interactions, invoke the RESTful API and dynamically build the user interface client side.

In this exercise, you will:

* Modify the Photo Details view to include
  + An empty container for the comments
  + An HTML template for a comment
  + HTML input fields and a button to add a comment
  + Javascript code to create an instance of a javascript ViewModel class
* Create a JavaScript commentsRepository object, with methods to get comments given a photo id and to add a comment
* Create a JavaScript ViewModel class that, in its constructor
  + Accepts a photoId parameter
  + Invokes the repository to get the comments for the given photo id the Favorites Slideshow action and View
  + Updates the user interface by dynamically creating copies of the template on the view and adding them to the comments container on the view.
  + Adds an event listener for the click of the add button on the view, so that when the user clicks it
    - the values on the view input boxes are used to set the properties of a new comment object that is sent as a parameter to the repository
    - the UI gets updated with the result

The main tasks for this exercise are as follows:

1. Modify the Details View
2. Create and implement a commentsRepository object
3. Create and implement a ViewModel class

### Task 1: Modify the Details View.

The Details View uses server side rendering to create the HTML of the Photo. You now want to add the UI for the Comments, but this time you will use client side rendering.

The server will send static HTML and some JavaScript. The browser will then execute the JavaScript to invoke the RESTful service and complete the rendering with the results.

Your View needs to contain:

* An empty container for the comments
* An HTML template for a comment
* HTML input fields and a button to add a comment

Open the Details view of the Photo Controller

1. At the end of the existing content, add an UL element that will act as container for the comments. Set its id attribute to “comments”
2. Add an INPUT field of type TEXT. Set its id attribute to “new-comment-subject”
3. Add an INPUT field of type TEXT. Set its id attribute to “new-comment-body”
4. Add a BUTTON. Set its id to “add-comment”
5. Add a TEMPLATE tag. Set its id attribute to “comment-template”
6. In the template tag, add an LI element
7. In the LI element, add a DIV element. Set its id attribute to “comment-subject”
8. Still in the LI element, add a DIV element. Set its id attribute to “comment-body”
9. Still in the LI element, add a DIV element. Set its id attribute to “comment-username”
10. At the end of the page, add a razor section named “scripts”
11. In the scripts section, add a SCRIPT tag. Set its type to “module”
12. Import the ViewModel class from the “./js/view-model.js” module. You’re going to create this file in an upcoming task.
13. Create an instance of the ViewModel class, passing the @Model.Id value as a parameter

Task 2: Create and implement a commentsRepository object

The next task is to create a repository for the comments. This object will invoke the API and return the deserialized responses.

1. Under the wwwroot / js folder, create a comments-repository.js file
2. Create a commentsRepository constant
3. Set its value to an object with a “serviceUrl” property set to “/api/comments”
4. Add an async getCommentsByPhotoId method that accepts an id
   1. Invoke the fetch method passing a value of `/api/photos/${id}/comments`
   2. Asynchronously wait for the fetch method and save the result in a “res” constant
   3. Invoke the json method of the res object and return the result
5. Add an async createComment method accepting a comment parameter
   1. Invoke the fetch method passing
      1. the value of this.serviceUrl
      2. An object with the following properties:
         1. method: 'POST',
         2. body: JSON.stringify(comment),
         3. headers: new Headers({'Content-Type': 'application/json'})
6. Export the commentsRepository constant as default

Task 3: Create and implement a ViewModel class

Having the view and the repository, it’s now time to create the layer that

* Invokes the repository to get the comments and dynamically builds the view
* Handles the click event of the view and invokes the repository to add a comment

1. Under the wwwroot / js folder, create a view-model.js file
2. Import the commentsRepository object from the “./comments-repository.js” module
3. Create a ViewModel class
4. Add a constructor accepting a photoId parameter
   1. Initialize a photoId property setting its value to the photoId parameter
   2. Get the document element whose id is “comment-template” and save it in a new “template” property
   3. Get the document element whose id is “comments” and save it in a new “target” property
   4. Invoke the getCommentsById method of the commentsRepository object passing the photoId property
   5. Pass a method to the promise fulfillment that accepts a comments parameter (use a lambda expression)
      1. For each comment of the comments parameter, invoke a this.addCommentToUi, passing the comment item as a parameter. You’re going to implement this method in a later step
   6. Find the document element with the id of “add-comment” and handle its click event by passing a lambda expression
      1. Create a newComment object with the following properties
         1. photoId: this.photoId
         2. id: 0
         3. subject: document.querySelector(`#new-comment-subject`).value
         4. body: document.querySelector(`#new-comment-body`).value
         5. userName: ``
         6. photo: null
      2. Invoke the insertComment method of the commentsRepository object passing the newComment object as a parameter
      3. Pass a method to the promise fulfillment that accepts a comment parameter (use a lambda expression). In the expression, invoke the this.addCommentToUi passing the comment
5. Add an addCommentToUi method accepting a comment parameter
   1. Find the this.template element whose class is “comment-subject” and set its textContent to the subject property of the comment parameter
   2. Find the this.template element whose class is “comment-body” and set its textContent to the body property of the comment parameter
   3. Find the this.template element whose class is “comment-username” and set its textContent to the userName property of the comment parameter
   4. Invoke the importNode method of the document object, passing this.template.content as first parameter and true as second parameter to indicate a deep import
   5. Append the result to this.target as a child node

Run the application, chose one photo and view its details. Try to add a couple of comments and see the results.

**Result: After completing this exercise, you will be able to build a JavaScript client that fetches data from an API service and uses partial page updates to dynamically build the HTML view client side.**