ITI Intake 43 (Web API Course)

Welcome to the repository where you'll find comprehensive lecture summaries and guidance for completing assignments.

Explore lecture videos on our Google Drive: Web API ITI

If you're working with the Web API projects in this repository, follow these steps to ensure smooth setup:

1. **Download the Repository**: Start by cloning or downloading the repository to your local machine.
2. **Adjust Connection String**: Open the `appsettings.json` file within each Web API project. Locate the `ConnectionStrings` section and ensure that the connection string named `Default` is properly configured to point to your desired database. Modify the connection string parameters (such as server name, database name, authentication details) as necessary to match your environment.

Feel free to browse through the content, enhance your understanding of Web API concepts, and tackle practical assignments to reinforce your learning. Happy coding!

Day 1

Lecture Repository

Simple CRUD Operations in the API Project

This API project offers basic CRUD functionality for managing a single model. Here's a quick overview:

1. **Create**: Add a new entity by sending a POST request to the appropriate endpoint with the entity data in the request body.
2. **Read**: Retrieve entities by sending GET requests to the corresponding endpoints. You can fetch all entities or get a specific entity by its criteria.
3. **Update**: Modify an existing entity by sending a PUT request to the endpoint with the updated entity data in the request body.
4. **Delete**: Remove an entity by sending a DELETE request to the appropriate endpoint with the entity's identifier.

Code Structure:

- **Controllers**: The CRUD operations are implemented in controller classes within the API project. Each controller handles requests for a specific entity type and maps them to corresponding methods for CRUD operations.
- **Models**: The model class representing the entity is defined within the project, typically in a `Models` or `Entities` folder. This class defines the structure and properties of the entity.

Assignment Repository

Project Overview

This API project embodies a structured architecture with three tiers: the Core layer, EF layer, and Presentation layer represented with the API Project.

Dependency Structure:

- **API Project**: This layer serves as the presentation layer and depends on the EF Layer, which in turn relies on the Core Layer.
- **EF Class Library**: This layer encapsulates the Entity Framework (EF) functionality and serves as the data access layer. It relies on the Core Layer.
- **Core Layer**: This layer contains data transfer objects (DTOs), interfaces, and the model.

Key Features:

1. **Code-First Approach**: The project follows the code-first approach, enabling seamless database creation and updates based on entity models.
2. **Unit of Work**: The implementation of the unit of work pattern ensures transactional integrity and facilitates managing interactions with the database.
3. **Repository Pattern**: Utilizing a generic repository interface and implementation enhances code reusability and maintainability by providing a consistent way to interact with data.
4. **Dynamic Search**: The project employs dynamic search capabilities in GET methods, leveraging `Expression<Func<T, bool>>` for flexible and efficient querying.

Core Layer Components:

- **DTOs**: Data transfer objects facilitate smooth communication between layers for create and update actions, accompanied by simple mapping logic.
- **Interfaces**: The Core Layer houses interfaces defining the unit of work and the base repository for consistent data access patterns.
- **Model**: The singular model, 'Employee', serves as the primary entity within the application, representing employee data.

EF Layer Components:

- **Context**: The EF Layer hosts the Application DbContext, providing the bridge between the application and the underlying database.
- **Migrations**: EF Core's migration feature automates the database schema updates, ensuring seamless integration of changes via commands like `add-Migration` and `Update-Database`.
- **Repositories**: Implementation of the base repository, along with dynamic search capabilities using expressions and function delegates, enriches data access functionality. Additionally, the Employee Repository caters to specific entity operations.
- **Unit of Work (UOW)**: The implementation of the unit of work pattern within the EF Layer orchestrates database operations and maintains transactional consistency via the ApplicationDbContext.

- **EFServicesInjection**: This component resolves all dependencies required by the EF Layer, including the unit of work, and configures the connection string for the ApplicationDbContext.

Theoretical Part

Website vs Webservice ?

- **Website:** Integrates frontend and backend components for a complete application (Frontend + Backend).
- **Webservice:** Focuses on the backend, serving as an interface for sharing business logic (Backend only).

Why we need backend services?

When the same business logic needs to be utilized across multiple platforms (e.g., Mobile apps, Desktop apps, Website apps), you have two options:

1. **Code Duplication:** Rewrite the logic for each platform.
2. **Class Library (DLL):** Centralize business logic in a class library and reference it. This approach aligns with the concept of Backend services.

what is the disadvantages of using class library (DLL)?

1. **Device Dependency:** The DLL must be present on the same device. (Resolve by deploying the DLL on a hosting server with a designated URL).
2. **Framework Compatibility:** Ensure compatibility by matching the project framework with the .NET version of the DLL.

Microsoft Technologies for Backend Solutions

1. **Web Service (Using HTTP Protocol):**
    - Lower security, outputs a DLL, necessitates the use of HTTP.
2. **WCF (Windows Communication Foundation) (Using HTTP/TCP Protocols):**
    - Introduced in .NET Framework 3.5, use SOAP Protocol for data transfer.
3. **Restful WCF (Using HTTP Protocol):**
    - Facilitates function calls without adding references, supports data transfer in JSON or XML.
4. **Web API (Using HTTP Protocol) (.NET Framework):**
    - Efficiently returns data without mandating specific formats like JSON or XML.
5. **Web API (Using HTTP Protocol) (.NET Core):**
    - Standardizes data transfer with JSON, facilitating communication between different software.

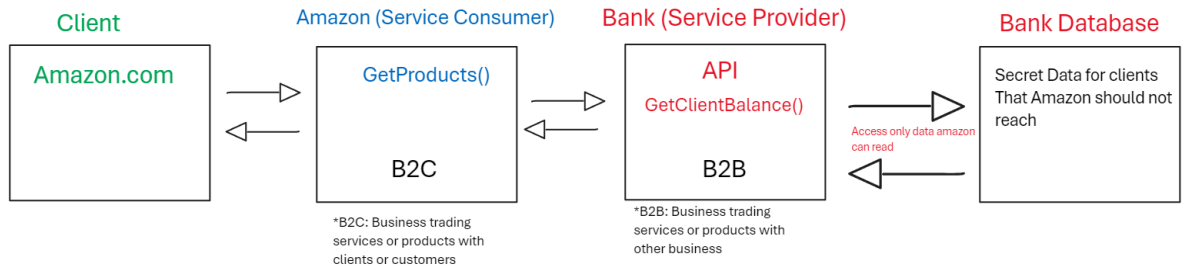
Restful API

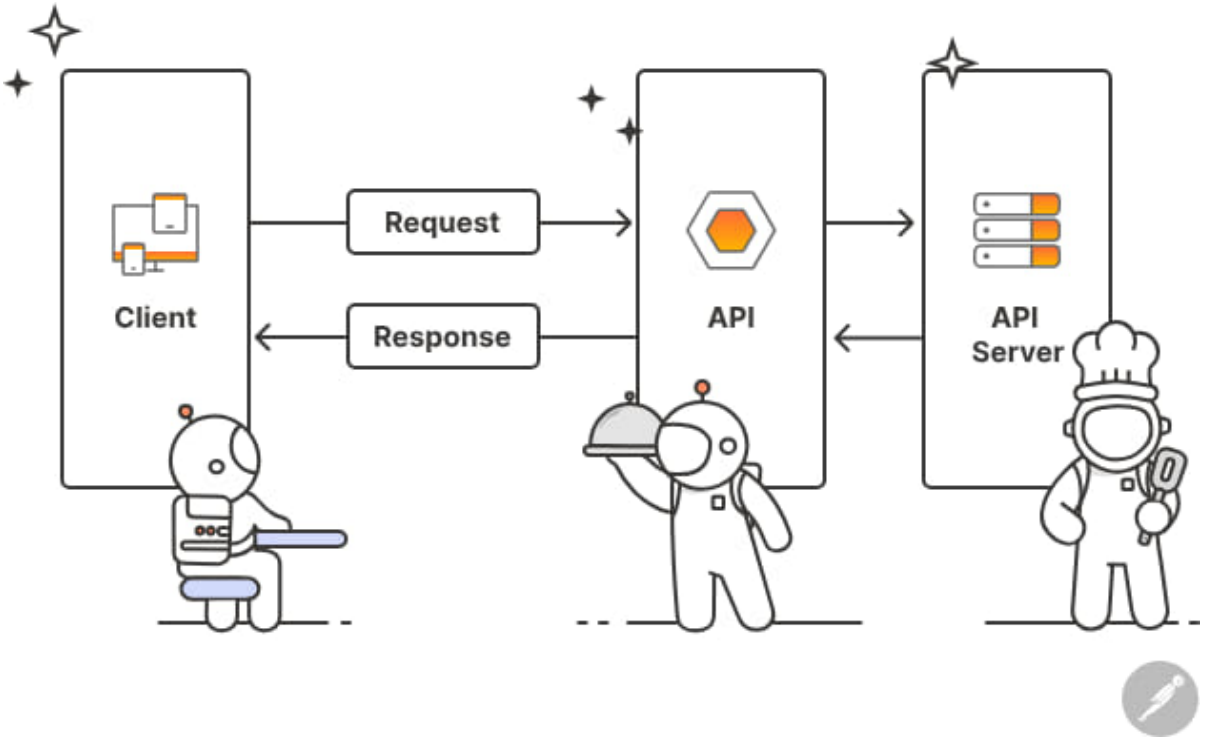
When to use Restful Web API ?

1- In case you are developing Service provider application (B2B)

EX: Banks provide services to pay online or check client balance (Service Provider)

While Amazon use this service in it's website (Service Consumer), as amazon does not have access to bank database or store client balances in it's database





2- if you do not which platform or application you will create (Mobile application - Web application - Desktop Application)

1. **Service Provider Application (B2B):**
   - Example: Banks offering online payment or balance checking services (Service Provider).

     Amazon, as a Service Consumer, utilizes these services without direct access to the bank database or storing client balances locally.
2. **Undecided Platform/Application:**
   - Suitable when uncertain about the target platform or application (Mobile, Web, Desktop).
   - Enables flexibility for integration with various applications without platform constraints.

Restful

It is style that provide standard communication between different systems.

Restful Constraints:

1. **Server / Client:**
    - There must be provider and consumer.
2. **Stateless:**
    - Each request server receives is independent and does not relate to requests that came prior to it or next to it.
3. **Enable To Cache :**
    - Facilitates caching of data to optimize resource utilization for subsequent requests with similar data requirements.

    NOTE: HTTP Request Can Do (1,2,3)

4. **Uniform Name:**
    - Each resource has a singular URL.
    - Example: `Domain.com/Employee` serves various purposes (e.g., retrieving, editing, deleting, or creating data) based on the HTTP request verb.

HTTP Request

Request Verbs

1. GET: Retrieve a single item (Resource) or a list of items (Resources).
2. POST: Create an item (Resource).
3. Put: Update an item (Resource).
4. PATCH: Partially update an item (Resource).
5. DELETE: Delete an item (Resource).

NOTE: If there is 2 methods with same verb then we use [Route("URL")] to differentiate between methods

Request Anatomy

Header:

- **Request URL:** …./Employee
- **Request Verb:** GET|POST|PUT|DELETE
- **Content-Type:** Specifies the format of data in the request body, such as JSON or XML.
- **Accept-Type:** Specifies the desired format for retrieving data in the response, whether JSON or XML.

Body:

- Contains the request data in JSON or XML format.

Response Anatomy

Header:

- **Request URL:** Indicates the request to which this response corresponds.
- **Status Code:** Indicates the state of response
  - **Successful Range (200 - 299)**
    - 200: OK
    - 201: Created
    - 204: No Content
  - **Client Error Response Range (400 - 499)**
    - 400: Bad Request
    - 401: Unauthorized
    - 403: Forbidden
    - 404: Not Found
  - **Server Error Response Range (500-599)**
    - 500: Internal Server Error
    - 502: Bad Gateway

JSON Deserialization vs Serialization

- **Deserialization:** Mapping a JSON File To C# class (.NET Type)
- **Serialization:** Convert .NET Type to JSON

Practical Part

Setup Configuration

1. Install **Microsoft.EntityFrameworkCore.SqlServer** and **Microsoft.EntityFrameworkCore.Tools**
2. **Create Application DbContext** inherits from DbContext
   - Inject connection string to Application DBContext Class from program.cs **(Don't forget constructor of DBContext that takes argument options)**
   - It preferable to save the connection string in user secret
3. **Create Department Model**
   - Add Department Model inside Application DbContext

     `DbSet<Department> Departments { get; set; }`
4. **Add migration** then **Update Database**

   `add-migration init`

   `Update-Database`

API Controller vs MVC Controller

- **From Model Binder POV**
  - **MVC Model Binder:**
    - Searches in Route, Form, and Collection (Query String).
    - Binds data to parameters based on route configuration and form input.
  - **API Model Binder:**
    - Depends on the type of parameter.
      - If Primitive Type (int, string, datetime), searches in the URL (Segment or Query String).
      - If Complex Type (Class), searches in the Request Body.
- **From Inheritance POV**
- MVC Controller inherits Controller inherits ControllerBase (MVC Controller Name:Controller:ControllerBase)
- API Controller inherits ControllerBase (API Controller Name:ControllerBase)

NOTE: As we observe, both MVC controller and API controller inherit from ControllerBase. To distinguish API controller from MVC Controller, use the [ApiController] attribute.

NOTE: API controller does not contain anything related to views.

CRUD operations

In Department Controller

GET Methods

```
        [HttpGet]
        //retrieving all items
        public IActionResult Get()
        {
            List<Department> departments = _context.Departments.ToList();
            return Ok(departments);
        }
```

Customizing Route Patterns on Endpoints

- **Using route attribute:**

```
[Route("api/Department/{id}")]
[Route("{id}")]
```

  - `{id}` is a placeholder for your parameter.
  - If the route begins with a `/` like this `[Route("/{id}")]`, it means you want to replace the controller URL, which is "api/Department," with "{id}".
  - If the route begins with a placeholder directly like this `[Route("{id}")]`, it means you want to concatenate with the controller URL, resulting in "api/Department/{id}".

- **Using request verb attribute overload:**

```
[HttpGet("{id}")]
```

  - Follows the same rules as the Route attribute.

```
        [HttpGet("{id}")]
        //retrieving one item by id
        public IActionResult GetById(int id)
        {
            Department department = _context.Departments.FirstOrDefault(x=>x.Id
== id);
            return department is null ? Ok(department) : NotFound();
        }
```

POST Method

Using Model State in validation according to Department Model

```
    [HttpPost]
    //add new item to database
    public IActionResult Create(Department department)
    {
        if(ModelState.IsValid)
        {
            _context.Departments.Add(department);
            _context.SaveChanges();
            return Ok("Created");//or Create();
        }
        // in case data in request body not valid then return Status Code
BadRequest
        return BadRequest(ModelState);
    }
```

PUT Method

Forcing Model Binder to Retrieve Values from Specific Sources

You can specify the source from which the model binder should retrieve values by adding the corresponding attribute:

- `[FromQuery]`: Retrieves values from the query string (e.g., `api/Department?id=5`).
- `[FromRoute]`: Retrieves values from the URL route (e.g., `api/Department/5`).
- `[FromBody]`: Retrieves values from the body of the request.
- `[FromHeader]`: Retrieves values from the header of the request.

```
public IActionResult Update([FromRoute]int id,[FromBody]Department department)
```

```
        [HttpPut("{id}")]
```

```
        //updating one item by id
        public IActionResult Update(int id,Department department)
        {
            if (ModelState.IsValid)
            {
                Department orgDep = _context.Departments.FirstOrDefault(x => x.Id
== id);
                orgDep.Name = department.Name;
                orgDep.ManagerName = department.ManagerName;
                _context.SaveChanges();
                return Ok("Updated");//or NoContent();
            }
            // in case data in request body not valid then return Status Code
BadRequest
            return BadRequest(ModelState);
        }
```

DELETE Method

```
     [HttpDelete("{id}")]
     //deleting item by id
     public IActionResult Delete(int id)
     {
         try
         {
             Department orgDep = _context.Departments.FirstOrDefault(x => x.Id ==
id);
             _context.Remove(orgDep);
             _context.SaveChanges();
             return Ok("Deleted");//or NoContent();
         }
         catch (Exception ex)
         {
             return BadRequest(ex.Message);
         }
     }
```

Handling Routing Ambiguity

In scenarios where there's another `GET` method named `GetByName`:

```
[HttpGet("{name}")]
//retrieving one item by name
public IActionResult GetByName(string name)
{
    Department department = _context.Departments.FirstOrDefault(x => x.Name ==
name);
    return department is null ? NotFound() : Ok(department);
}
```

Executing this in Swagger may lead to the error:

```
Microsoft.AspNetCore.Routing.Matching.AmbiguousMatchException: The request
matched multiple endpoints
```

This issue arises because the routing can't distinguish between `[HttpGet("{name}")]` and `[HttpGet("{id}")]`, as it cannot differentiate between string and int inside placeholders.

To resolve this, use constraints inside the placeholders like this `[HttpGet("{name:alpha}")]` and `[HttpGet("{id:int}")]`.

You can use more than one constraint to provide clear distinctions.

However, if we have another `GET` method named `GetByManager` that also uses the `alpha` constraint:

```
        [HttpGet("{manager:alpha}")]
        //retrieving one item by manager
        public IActionResult GetByManagerName(string manager)
        {
            Department department = _context.Departments.FirstOrDefault(x =>
x.ManagerName == manager);
            return department is null ? NotFound() : Ok(department);
        }
```

Then we must use literal differentiation in the segment:

```
    [HttpGet("Manager/{manager:alpha}")]
    //retrieving one item by manager
    public IActionResult GetByManagerName(string manager)
    {
        Department department = _context.Departments.FirstOrDefault(x =>
x.ManagerName == manager);
        return department is null ? NotFound() : Ok(department);
    }
```

Assignment

Web API implements CRUD Operations with Repository Pattern.