Date: 07 Feb 2024

Basic Programming Constructors in TypeScript

Problem Statements in TypeScript

1) Reverse Name:

Create a TypeScript function that accepts your name as a parameter and prints it in reverse.

2) Find the Greatest Number:

Design a TypeScript function that takes three numbers as parameters and displays the greatest among them.

3) Check Vowel using Switch Case:

Implement a TypeScript function that takes an alphabet as a parameter and uses a switch case to display whether it is a vowel or not.

4) Check Vowel using If-Else:

Develop a TypeScript function that takes an alphabet as a parameter and uses if-else statements to display whether it is a vowel or not.

5) Validate User Credentials:

Write a TypeScript function that accepts a username and password as parameters. Check if the username is 'Administrator' and the password is 'Admin@123'. Return true if the credentials match; otherwise, return false. Call this function within another function and display "Login Successful" or "Login Failed" based on the returned result.

NOTE: Implement all the above functions, a) a normal typescript functions

b) as arrow functions

Solutions using functions

1)ReverseName solution

```
function reverseName(name: string): string {
   return name.split('').reverse().join('');
}
const reversedName = reverseName("John Doe");
console.log("Reversed Name:", reversedName);
```

2) Find the Greatest Number

```
function findGreatestNumber(num1: number, num2: number, num3: number): number {
```

```
    return Math.max(num1, num2, num3);
}
const greatestNumber = findGreatestNumber(5, 12, 8);
console.log("Greatest Number:", greatestNumber);
```

3) Check Vowel using Switch Case

```
function checkVowelWithSwitch(alphabet: string): string {
    switch (alphabet.toLowerCase()) {
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u':
            return "Vowel";
        default:
            return "Not a Vowel";
    }
}
const vowelResult = checkVowelWithSwitch('A');
console.log("Check Vowel with Switch:", vowelResult);
```

4) Check Vowel using If else

```
function checkVowelWithIfElse(alphabet: string): string {
    const lowerAlphabet = alphabet.toLowerCase();
    if (lowerAlphabet === 'a' || lowerAlphabet === 'e' || lowerAlphabet === 'i' || lowerAlphabet
=== 'o' || lowerAlphabet === 'u') {
        return "Vowel";
    } else {
        return "Not a Vowel";
    }
```

```
}
const vowelResultIfElse = checkVowelWithIfElse('E');
console.log("Check Vowel with If-Else:", vowelResultIfElse);

5. Validate User Credentials
function validateCredentials(username: string, password: string): boolean {
   return username === 'Administrator' && password === 'Admin@123';
}
function loginStatus(username: string, password: string): void {
   const isLoggedIn = validateCredentials(username, password);
   if (isLoggedIn) {
      console.log("Login Successful");
   } else {
      console.log("Login Failed");
   }
}
// Example usage
loginStatus('Administrator', 'Admin@123');
Solutions with Arrow functions
    1) ReverseName
const reverseName = (name: string): string => name.split('').reverse().join('');
const reversedName = reverseName("John Doe");
console.log("Reversed Name:", reversedName);
    2) Find the Greatest Number
const findGreatestNumber = (num1: number, num2: number, num3: number): number =>
   Math.max(num1, num2, num3);
const greatestNumber = findGreatestNumber(5, 12, 8);
console.log("Greatest Number:", greatestNumber);
    3) Check Vowel using Switch case
const checkVowelWithSwitch = (alphabet: string): string => {
   switch (alphabet.toLowerCase()) {
```

```
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u':
            return "Vowel";
        default:
            return "Not a Vowel";
    }
};
const vowelResult = checkVowelWithSwitch('A');
console.log("Check Vowel with Switch:", vowelResult);
```

4) Check Vowels using If-Else

```
const checkVowelWithIfElse = (alphabet: string): string => {
    const lowerAlphabet = alphabet.toLowerCase();
    if (lowerAlphabet === 'a' || lowerAlphabet === 'e' || lowerAlphabet === 'i' || lowerAlphabet
=== 'o' || lowerAlphabet === 'u') {
        return "Vowel";
    } else {
        return "Not a Vowel";
    }
};
const vowelResultIfElse = checkVowelWithIfElse('E');
console.log("Check Vowel with If-Else:", vowelResultIfElse);
```

5) Validate User Credentials

```
const validateCredentials = (username: string, password: string): boolean =>
    username === 'Administrator' && password === 'Admin@123';
const loginStatus = (username: string, password: string): void => {
    const isLoggedIn = validateCredentials(username, password);
    isLoggedIn ? console.log("Login Successful") : console.log("Login Failed");
```

};

// Example usage

loginStatus('Administrator', 'Admin@123');

Create Stored procedure in WebAPI CORE -EFCORE using Code First

PreRequisite:

Design a simplified Human Resources Management System (HRMS) using Entity Framework Core's Code First approach. Implement two entities, Department and Employee, with a one-to-many relationship between them. Create a Web API with controllers for managing Employee and Department entities, allowing the insertion of new records through HTTP POST requests.

Requirements:

Department Entity:

Create a class named Department representing a department in the organization.

Include the following properties:

DepartmentId (Primary Key)

DeptName

Location

Employee Entity:

Create a class named Employee representing an employee in the organization.

Include the following properties:

EmployeeId (Primary Key)

EmpName

Designation

DepartmentId (Foreign Key referencing Department)

Entity Framework Code First Approach:

Use Entity Framework Core to define the database schema using the Code First approach.

Set up a one-to-many relationship between Department and Employee where one department can have multiple employees.

Web API Controllers:

Create a Web API controller for managing Department entities.

Implement HTTP GET, POST, PUT, and DELETE methods for CRUD operations on departments.

Create a Web API controller for managing Employee entities.

Implement HTTP GET, POST, PUT, and DELETE methods for CRUD operations on employees.

HttpPost to Insert Data:

Implement the HTTP POST method for both Department and Employee controllers to allow the insertion of new records.

For the Department controller, the HTTP POST method should accept a JSON payload containing DeptName and Location.

For the Employee controller, the HTTP POST method should accept a JSON payload containing EmpName, Designation, and DepartmentId.

Example HTTP POST Payloads:

For creating a new department:

json

```
{
   "DeptName": "IT Department",
   "Location": "Headquarters"
}
```

For creating a new employee:

json

```
{
   "EmpName": "John Doe",
   "Designation": "Software Engineer",
   "DepartmentId": 1
}
```

Ensure appropriate error handling, validation, and response messages are implemented in the Web API controllers. The goal is to provide a functional and reliable API for managing departments and employees within the organization.

Problem Statement

To above program add a stored procedure called spGetEmployeeDepartment using Code first approach. The stored procedure should have a inner join to fetch data from Employees and Departments tables

PreRequisite Work

Step 1: Create Department and Employee Classes

Create two classes, Department and Employee, with the necessary properties.

```
public class Department
```

```
{
    [Key]
    public int DepartmentID { get; set; }
    public string? DeptName { get; set; }
    public string? Location { get; set; }

}

   public class Employee
   {
      [Key]
      public int EmployeeId { get; set; }
      public string? EmpName { get; set; }
      public string? Designation { get; set; }

      [ForeignKey("DepartmentID")]
      public int DepartmentID { get; set; }

      [JsonIgnore]
      public virtual Department? Department { get; set; }
   }
```

Step 2: Set Up DbContext

Create a class that inherits from DbContext and define DbSet properties for Department and Employee. Set up the relationship between them.

```
public class HrmsDbContext : DbContext

{

  public HrmsDbContext(DbContextOptions< HrmsDbContext > options):base(options)
  {

  }

    public DbSet<Department> Departments { get; set; }

    public DbSet<Employee> Employees { get; set; }

}
```

Step 3: Initialize DbContext in Program.cs

Configure the DbContext in the Program.cs file.

```
builder.Services.AddDbContext< HrmsDbContext>
    (options => options.UseSqlServer
    (builder.Configuration.GetConnectionString("Constr")));
```

Step 4: Create Web API Controllers

Create two controllers, one for Department and another for Employee, using the [ApiController] attribute.

```
[Route("api/[controller]")]
[ApiController]
public class DepartmentsController : ControllerBase
{
    private readonly HrmsDbContext _context;

    public DepartmentsController(HrmsDbContext context)
    {
        _context = context;
    }

    // GET: api/Departments
    [HttpGet]
    public async Task<ActionResult<IEnumerable<Department>>> GetDepartments()
    {
        return await _context.Departments.ToListAsync();
    }

    // GET: api/Departments/5
    [HttpGet("{id}")]
    public async Task<ActionResult<Department>> GetDepartment(int id)
    {
        var department = await _context.Departments.FindAsync(id);

        if (department == null)
        {
            return NotFound();
        }

        return department;
    }

    // PUT: api/Departments/5
    // To protect from overposting attacks, see https://go.microsoft.com/fwlink/?linkid=2123754
    [HttpPut("{id}")]
    public async Task<IActionResult> PutDepartment(int id, Department department)
    {
        if (id != department.DepartmentID)
        {
            return BadRequest();
        }

        _context.Entry(department).State = EntityState.Modified;

        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
```

```
            if (!DepartmentExists(id))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }

        return NoContent();
    }

    // POST: api/Departments
    // To protect from overposting attacks, see https://go.microsoft.com/fwlink/?linkid=2123754
    [HttpPost]
    public async Task<ActionResult<Department>> PostDepartment(Department department)
    {
        _context.Departments.Add(department);
        await _context.SaveChangesAsync();

        return CreatedAtAction("GetDepartment", new { id = department.DepartmentID }, department);
    }

    // DELETE: api/Departments/5
    [HttpDelete("{id}")]
    public async Task<IActionResult> DeleteDepartment(int id)
    {
        var department = await _context.Departments.FindAsync(id);
        if (department == null)
        {
            return NotFound();
        }

        _context.Departments.Remove(department);
        await _context.SaveChangesAsync();

        return NoContent();
    }

    private bool DepartmentExists(int id)
    {
        return _context.Departments.Any(e => e.DepartmentID == id);
    }
}
```

```
[Route("api/[controller]")]
[ApiController]
public class EmployeesController : ControllerBase
```

```
{
    private readonly HrmsDbContext _context;

    public EmployeesController(HrmsDbContext  context)
    {
        _context = context;
    }

    // GET: api/Employees
    [HttpGet]
    public async Task<ActionResult<IEnumerable<Employee>>> GetEmployees()
    {
        return await _context.Employees.ToListAsync();
    }

    // GET: api/Employees/5
    [HttpGet("{id}")]
    public async Task<ActionResult<Employee>> GetEmployee(int id)
    {
        var employee = await _context.Employees.FindAsync(id);

        if (employee == null)
        {
            return NotFound();
        }

        return employee;
    }

    // PUT: api/Employees/5
    // To protect from overposting attacks, see https://go.microsoft.com/fwlink/?linkid=2123754
    [HttpPut("{id}")]
    public async Task<IActionResult> PutEmployee(int id, Employee employee)
    {
        if (id != employee.EmployeeId)
        {
            return BadRequest();
        }

        _context.Entry(employee).State = EntityState.Modified;

        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!EmployeeExists(id))
            {
                return NotFound();
            }
```

```
            else
            {
                throw;
            }
        }

        return NoContent();
    }

    // POST: api/Employees
    // To protect from overposting attacks, see https://go.microsoft.com/fwlink/?linkid=2123754
    [HttpPost]
    public async Task<ActionResult<Employee>> PostEmployee(Employee employee)
    {
        _context.Employees.Add(employee);
        await _context.SaveChangesAsync();

        return CreatedAtAction("GetEmployee", new { id = employee.EmployeeId }, employee);
    }

    // DELETE: api/Employees/5
    [HttpDelete("{id}")]
    public async Task<IActionResult> DeleteEmployee(int id)
    {
        var employee = await _context.Employees.FindAsync(id);
        if (employee == null)
        {
            return NotFound();
        }

        _context.Employees.Remove(employee);
        await _context.SaveChangesAsync();

        return NoContent();
    }

    private bool EmployeeExists(int id)
    {
        return _context.Employees.Any(e => e.EmployeeId == id);
    }


}
```

Step 5: Implement HTTP POST methods

In each controller, implement the HTTP POST methods for creating new records.

Step 6: Test the API

Run the application and use tools like Postman or curl to test the API endpoints for creating departments and employees.

This step-by-step solution provides a foundation for creating a Web API using Entity Framework Core's Code First approach for managing departments and employees in a simplified HRMS. Remember to handle exceptions, implement validation, and add error messages for a production-ready solution.

Solution

Step1: Create a migration file using the following command

Add-migration CreateProcedure

Step2: Open the CreateProcedure.cs file present in migrations folder and write the following syntax in Up method

```
var sql = @"create procedure NewProcedure
        as
        begin
        select e.EmployeeId,e.EmpName,e.Designation,d.DeptName,d.Location
        from Employees e join Departments d
        on d.DepartmentID =  e.DepartmentID
        end";

migrationBuilder.Sql(sql);
```

Step3: Build your application

Step4: update-database

Step5: Verify your database . open programmability folder under your database, expand stored procedures and check for the procedure name **spGetEmployeeDepartment.**

Step6: open a new query editor and execute the following command

exec **spGetEmployeeDepartment**

check whether stored procedure has given the result of the join