Case Study 4 - Sales Order Management System

Scenario:

A company wants to develop a Sales Order Management System to handle its sales operations. The system should allow sales representatives to create and manage sales orders, track order status, and generate reports on sales performance. The system should also maintain customer information and product details.

Please execute the following create table script before attempting the queries:

CREATE TABLE customers (

customer\_id NUMBER(10) PRIMARY KEY,

first\_name VARCHAR2(50) NOT NULL,

last\_name VARCHAR2(50) NOT NULL,

email VARCHAR2(50) NOT NULL,

phone\_number VARCHAR2(20) NOT NULL,

address VARCHAR2(200) NOT NULL

);

CREATE TABLE products (

product\_id NUMBER(10) PRIMARY KEY,

product\_name VARCHAR2(50) NOT NULL,

price NUMBER(10,2) NOT NULL,

quantity NUMBER(10) NOT NULL

);

CREATE TABLE sales\_orders (

order\_id NUMBER(10) PRIMARY KEY,

order\_date DATE NOT NULL,

customer\_id NUMBER(10) NOT NULL,

CONSTRAINT fk\_customers FOREIGN KEY (customer\_id) REFERENCES customers(customer\_id)

);

CREATE TABLE order\_items (

order\_id NUMBER(10) NOT NULL,

product\_id NUMBER(10) NOT NULL,

quantity NUMBER(10) NOT NULL,

CONSTRAINT fk\_sales\_orders FOREIGN KEY (order\_id) REFERENCES sales\_orders(order\_id),

CONSTRAINT fk\_products FOREIGN KEY (product\_id) REFERENCES products(product\_id)

);

Queries:

Create a trigger that automatically updates the quantity of a product in the products table when a new order is placed. Name the trigger as "update\_product\_quantity\_trigger".

Create a cursor that retrieves all sales orders along with the customer details and the total order amount. Name the cursor as "sales\_orders\_cur".

Create a procedure to calculate the total revenue generated from sales orders in a given date range and store it in a variable. Name the procedure as "calculate\_total\_revenue".

Create a function to retrieve the product name for a given product ID. Name the function as "get\_product\_name".

Create an exception handler for the procedure that creates a new sales order, to handle the case when the customer ID provided does not exist.

To develop an Online Course Management System, we can start by designing the necessary database tables. Here's a suggested table structure:

1. Courses:

- course\_id (Primary Key)

- course\_name

- course\_description

- instructor\_id (Foreign Key referencing instructors table)

2. Students:

- student\_id (Primary Key)

- student\_name

- email

- phone\_number

3. Enrollments:

- enrollment\_id (Primary Key)

- student\_id (Foreign Key referencing students table)

- course\_id (Foreign Key referencing courses table)

- enrollment\_date

4. CourseModules:

- module\_id (Primary Key)

- course\_id (Foreign Key referencing courses table)

- module\_name

- module\_description

5. Progress:

- progress\_id (Primary Key)

- enrollment\_id (Foreign Key referencing enrollments table)

- module\_id (Foreign Key referencing course\_modules table)

- is\_completed

6. Instructors:

- instructor\_id (Primary Key)

- instructor\_name

- email

- phone\_number

Now, let's proceed to implement the desired features using PL/SQL.

1. Creating Course Modules:

```sql

CREATE PROCEDURE create\_course\_module(

p\_course\_id IN courses.course\_id%TYPE,

p\_module\_name IN course\_modules.module\_name%TYPE,

p\_module\_description IN course\_modules.module\_description%TYPE

) AS

BEGIN

INSERT INTO course\_modules (module\_id, course\_id, module\_name, module\_description)

VALUES (module\_id\_seq.NEXTVAL, p\_course\_id, p\_module\_name, p\_module\_description);

COMMIT;

END;

```

2. Tracking Student Progress:

```sql

CREATE PROCEDURE update\_student\_progress(

p\_enrollment\_id IN enrollments.enrollment\_id%TYPE,

p\_module\_id IN course\_modules.module\_id%TYPE,

p\_is\_completed IN NUMBER

) AS

BEGIN

UPDATE progress

SET is\_completed = p\_is\_completed

WHERE enrollment\_id = p\_enrollment\_id AND module\_id = p\_module\_id;

COMMIT;

END;

```

3. Generating Course Completion Certificates:

```sql

CREATE FUNCTION generate\_certificate(

p\_enrollment\_id IN enrollments.enrollment\_id%TYPE

) RETURN VARCHAR2 AS

v\_student\_name students.student\_name%TYPE;

v\_course\_name courses.course\_name%TYPE;

BEGIN

SELECT s.student\_name, c.course\_name

INTO v\_student\_name, v\_course\_name

FROM students s

JOIN enrollments e ON s.student\_id = e.student\_id

JOIN courses c ON e.course\_id = c.course\_id

WHERE e.enrollment\_id = p\_enrollment\_id;

RETURN 'Certificate of Completion\n\nThis is to certify that ' || v\_student\_name ||

' has successfully completed the course ' || v\_course\_name ||

' on ' || TO\_CHAR(SYSDATE, 'DD-MON-YYYY');

END;

```

These are just simplified examples of the PL/SQL procedures and functions to implement the core functionalities of an Online Course Management System. You can further enhance and customize the system based on your specific requirements and business logic.

Certainly! Here's an example of the Charity Donation Management System using PL/SQL with the implementation of cursors, triggers, functions, procedures, and collections:

Tables:

1. Donors:

- donor\_id: NUMBER

- donor\_name: VARCHAR2(50)

- email: VARCHAR2(50)

- phone\_number: VARCHAR2(20)

- address: VARCHAR2(200)

2. Campaigns:

- campaign\_id: NUMBER

- campaign\_name: VARCHAR2(50)

- start\_date: DATE

- end\_date: DATE

- target\_amount: NUMBER

3. Donations:

- donation\_id: NUMBER

- donor\_id: NUMBER

- campaign\_id: NUMBER

- donation\_date: DATE

- amount: NUMBER

PL/SQL Components:

1. Cursor: Retrieve Donors who donated more than a specified amount.

2. Trigger: Update the campaign's current amount whenever a new donation is made.

3. Function: Calculate the total amount donated by a specific donor.

4. Procedure: Generate a donation report for a specific campaign.

5. Collection: Use a collection to store donor information and their total donations.

These are the basic components of the Charity Donation Management System. You can further expand and customize the system based on your specific requirements and business logic.

Certainly! Here are the queries for the Charity Donation Management System using PL/SQL:

1. Cursor Query to Retrieve Donors who Donated More than a Specified Amount:

```sql

DECLARE

CURSOR donors\_cursor(amount\_limit NUMBER) IS

SELECT donor\_id, donor\_name, email, phone\_number, address

FROM donors

WHERE donor\_id IN (

SELECT donor\_id

FROM donations

GROUP BY donor\_id

HAVING SUM(amount) > amount\_limit

);

BEGIN

FOR donor\_rec IN donors\_cursor(500) LOOP

DBMS\_OUTPUT.PUT\_LINE('Donor ID: ' || donor\_rec.donor\_id);

DBMS\_OUTPUT.PUT\_LINE('Donor Name: ' || donor\_rec.donor\_name);

DBMS\_OUTPUT.PUT\_LINE('Email: ' || donor\_rec.email);

DBMS\_OUTPUT.PUT\_LINE('Phone Number: ' || donor\_rec.phone\_number);

DBMS\_OUTPUT.PUT\_LINE('Address: ' || donor\_rec.address);

DBMS\_OUTPUT.PUT\_LINE('--------------------------');

END LOOP;

END;

/

```

2. Trigger to Update Campaign's Current Amount on Donation Insertion:

```sql

CREATE OR REPLACE TRIGGER update\_campaign\_amount\_trigger

AFTER INSERT ON donations

FOR EACH ROW

BEGIN

UPDATE campaigns

SET current\_amount = current\_amount + :NEW.amount

WHERE campaign\_id = :NEW.campaign\_id;

COMMIT;

END;

/

```

3. Function to Calculate Total Amount Donated by a Specific Donor:

```sql

CREATE OR REPLACE FUNCTION calculate\_total\_donation(donor\_id NUMBER)

RETURN NUMBER IS

total\_donation NUMBER := 0;

BEGIN

SELECT SUM(amount)

INTO total\_donation

FROM donations

WHERE donor\_id = calculate\_total\_donation.donor\_id;

RETURN total\_donation;

END;

/

```

4. Procedure to Generate Donation Report for a Specific Campaign:

```sql

CREATE OR REPLACE PROCEDURE generate\_donation\_report(campaign\_id NUMBER) IS

BEGIN

FOR donation\_rec IN (

SELECT d.donor\_id, d.donor\_name, d.email, d.phone\_number, d.address, dn.amount

FROM donors d

JOIN donations dn ON d.donor\_id = dn.donor\_id

WHERE dn.campaign\_id = generate\_donation\_report.campaign\_id

) LOOP

DBMS\_OUTPUT.PUT\_LINE('Donor ID: ' || donation\_rec.donor\_id);

DBMS\_OUTPUT.PUT\_LINE('Donor Name: ' || donation\_rec.donor\_name);

DBMS\_OUTPUT.PUT\_LINE('Email: ' || donation\_rec.email);

DBMS\_OUTPUT.PUT\_LINE('Phone Number: ' || donation\_rec.phone\_number);

DBMS\_OUTPUT.PUT\_LINE('Address: ' || donation\_rec.address);

DBMS\_OUTPUT.PUT\_LINE('Donation Amount: ' || donation\_rec.amount);

DBMS\_OUTPUT.PUT\_LINE('--------------------------');

END LOOP;

END;

/

```

These are just examples to demonstrate the usage of cursors, triggers, functions, and procedures in the context of the Charity Donation Management System. You can modify and customize these queries as per your specific requirements and business logic.

Title: Employee Management Web API

Description: Create a RESTful web API to manage employee data for a company. The API will allow users to perform various operations on employee records, including retrieving employees, retrieving a single employee by ID, adding a new employee, updating an existing employee, and deleting an employee. The employee data will include attributes such as Employee ID, First Name, Last Name, Email, Department, Position, Salary, and Hire Date.

Technical Requirements:

- Use ASP.NET Web API to implement the RESTful API.

- Use ADO.NET or an ORM (Object-Relational Mapping) tool like Entity Framework to interact with the database.

- Implement HTTP GET, POST, PUT, and DELETE verbs to correspond to the required operations.

- Utilize JSON as the data format for input and output.

- Return appropriate HTTP status codes to indicate the success or failure of each operation.

- Implement error handling and exception management to handle unexpected scenarios gracefully.

- Incorporate input validation to ensure that the data is well-formed and within reasonable limits.

- Optionally, implement security measures like authentication and authorization to protect the API from unauthorized access.

Deliverables:

- Provide the source code for the API, including any necessary configuration files.

- Include a script to create the database schema.

- Document the API, including instructions on how to use it and how to deploy it.

- Optionally, provide a sample client application or API documentation tool that demonstrates how to consume and interact with the API.

Please note that the provided information is a general outline, and you'll need to adapt and customize the implementation based on your specific requirements and technologies used.

Creating the complete source code for an Employee Management Web API is beyond the scope of a simple text-based interaction. However, I can provide you with a high-level overview of the code structure and logic for the API. You can use this as a starting point and customize it to fit your specific needs.

Here is an example of how the source code for the Employee Management Web API might look:

1. Create a new ASP.NET Web API project in Visual Studio.

2. Define the Employee model:

- Create a new class named "Employee" with properties for Employee ID, First Name, Last Name, Email, Department, Position, Salary, and Hire Date.

- Decorate the properties with appropriate attributes for validation, such as Required, StringLength, and DataType.

3. Set up the database:

- Choose a database management system (e.g., SQL Server, MySQL, SQLite) and create a new database.

- Create a table named "Employees" with columns corresponding to the attributes of the Employee model.

4. Create a Data Access Layer (DAL):

- Implement a class that handles the database operations for the Employee model, such as retrieving all employees, retrieving a single employee by ID, adding a new employee, updating an existing employee, and deleting an employee.

- Use ADO.NET or an ORM like Entity Framework to interact with the database.

5. Create a Web API Controller:

- Implement a controller class that defines the API endpoints for employee operations.

- Use the appropriate HTTP verbs (GET, POST, PUT, DELETE) to handle the corresponding operations.

- Inject an instance of the DAL into the controller to access the employee data.

6. Implement the API endpoints:

- Define methods in the controller for each of the required operations.

- Map the methods to the appropriate HTTP verbs and route templates.

- Use the DAL methods to perform the corresponding database operations.

7. Handle input validation and error handling:

- Validate the input data received from API requests and return appropriate error responses if the data is invalid or the requested operation fails.

- Use exception handling to catch and handle any unexpected errors that may occur during the API operations.

8. Configure routing and other necessary settings:

- Set up routing in the API configuration to map the API endpoints to their corresponding controller actions.

- Configure other settings, such as JSON serialization options and error handling, in the API configuration.

9. Test the API:

- Use a tool like Postman or any other HTTP client to test the API endpoints and verify their functionality.

- Test different scenarios, such as retrieving all employees, adding a new employee, updating an employee, and deleting an employee.

Remember to handle authentication and authorization if required, and ensure that you follow best practices for security and performance in your API implementation.

Note: This is a high-level overview, and the actual code implementation will require writing the specific code logic in each step. For a complete implementation, it is recommended to refer to detailed tutorials, documentation, or books that provide step-by-step instructions for creating an Employee Management Web API using ASP.NET Web API and the database technology of your choice.

Certainly! Here's an example of how the code for an Employee Management Web API might look using ASP.NET Web API and Entity Framework as the ORM:

1. Create the Employee model:

```csharp

public class Employee

{

public int EmployeeId { get; set; }

public string FirstName { get; set; }

public string LastName { get; set; }

public string Email { get; set; }

public string Department { get; set; }

public string Position { get; set; }

public decimal Salary { get; set; }

public DateTime HireDate { get; set; }

}

```

2. Implement the Data Access Layer (DAL) using Entity Framework:

```csharp

public class EmployeeContext : DbContext

{

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

public EmployeeContext(DbContextOptions<EmployeeContext> options) : base(options)

{

}

}

public class EmployeeRepository

{

private readonly EmployeeContext \_context;

public EmployeeRepository(EmployeeContext context)

{

\_context = context;

}

public IEnumerable<Employee> GetAllEmployees()

{

return \_context.Employees.ToList();

}

public Employee GetEmployeeById(int id)

{

return \_context.Employees.Find(id);

}

public void AddEmployee(Employee employee)

{

\_context.Employees.Add(employee);

\_context.SaveChanges();

}

public void UpdateEmployee(Employee employee)

{

\_context.Employees.Update(employee);

\_context.SaveChanges();

}

public void DeleteEmployee(Employee employee)

{

\_context.Employees.Remove(employee);

\_context.SaveChanges();

}

}

```

3. Create the Web API Controller:

```csharp

[Route("api/employees")]

[ApiController]

public class EmployeesController : ControllerBase

{

private readonly EmployeeRepository \_employeeRepository;

public EmployeesController(EmployeeRepository employeeRepository)

{

\_employeeRepository = employeeRepository;

}

[HttpGet]

public IActionResult GetAllEmployees()

{

var employees = \_employeeRepository.GetAllEmployees();

return Ok(employees);

}

[HttpGet("{id}")]

public IActionResult GetEmployeeById(int id)

{

var employee = \_employeeRepository.GetEmployeeById(id);

if (employee == null)

{

return NotFound();

}

return Ok(employee);

}

[HttpPost]

public IActionResult AddEmployee(Employee employee)

{

\_employeeRepository.AddEmployee(employee);

return CreatedAtAction(nameof(GetEmployeeById), new { id = employee.EmployeeId }, employee);

}

[HttpPut("{id}")]

public IActionResult UpdateEmployee(int id, Employee employee)

{

if (id != employee.EmployeeId)

{

return BadRequest();

}

\_employeeRepository.UpdateEmployee(employee);

return NoContent();

}

[HttpDelete("{id}")]

public IActionResult DeleteEmployee(int id)

{

var employee = \_employeeRepository.GetEmployeeById(id);

if (employee == null)

{

return NotFound();

}

\_employeeRepository.DeleteEmployee(employee);

return NoContent();

}

}

```

4. Configure routing and dependency injection in Startup.cs:

```csharp

public class Startup

{

public void ConfigureServices(IServiceCollection services)

{

services.AddDbContext<EmployeeContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

services.AddScoped<EmployeeRepository>();

services.AddControllers();

}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

else

{

app.UseExceptionHandler("/error");

app.UseHsts();

}

app.UseRouting();

app.UseEndpoints(endpoints =>

{

endpoints.MapControllers();

});

}

}

```

Please note that this is a simplified example and you'll need to configure the database connection, connection string, and other necessary settings based on your specific setup. Also, don't forget to install the required NuGet packages such as `Microsoft.EntityFrameworkCore` and `Microsoft.EntityFrameworkCore.SqlServer`.

This code should give you a basic foundation for implementing an Employee Management Web API using ASP.NET Web API and Entity Framework. Remember to adapt it to your specific needs and requirements.