1. Update the User Entity to Include Roles

Modify the `User` model to include a `Role` property:

```
namespace ECommerceAPI.Models
{
	public class User
	{
	public int Id { get; set; }
	public string Username { get; set; }
	public string PasswordHash { get; set; }
	public string Role { get; set; } // New property for role
	}
}
```

Run a new migration to update the database schema:

```
dotnet ef migrations add AddUserRole
dotnet ef database update
```

2. Update the Registration Endpoint

Allow users to specify their role during registration. For simplicity, we'll default the role to `Customer` if not provided.

Update `Register` in `UserController`:

```
[HttpPost("register")]
public async Task<IActionResult> Register(User user)
{
	if (await _context.Users.AnyAsync(u => u.Username == user.Username))
	{
	return BadRequest("Username already exists.");
	}

	user.PasswordHash = BCrypt.Net.BCrypt.HashPassword(user.PasswordHash);
	user.Role = string.IsNullOrEmpty(user.Role) ? "Customer" : user.Role; // Default to
Customer

	_context.Users.Add(user);
	await _context.SaveChangesAsync();
```

```
        return Ok("User registered successfully.");
}
```

3. Include Roles in the JWT Token

Modify the `GenerateJwtToken` method in `UserController` to include the user's role:

```
private string GenerateJwtToken(User user)
{
        var tokenHandler = new JwtSecurityTokenHandler();
        var key = Encoding.UTF8.GetBytes(_authSettings.Secret);

        var tokenDescriptor = new SecurityTokenDescriptor
        {
        Subject = new ClaimsIdentity(new Claim[]
        {
        new Claim(ClaimTypes.Name, user.Id.ToString()),
        new Claim(ClaimTypes.Role, user.Role) // Include role in token
        }),
        Expires = DateTime.UtcNow.AddHours(1),
        SigningCredentials = new SigningCredentials(new SymmetricSecurityKey(key),
SecurityAlgorithms.HmacSha256Signature)
        };

        var token = tokenHandler.CreateToken(tokenDescriptor);
        return tokenHandler.WriteToken(token);
}
```

4. Use Role-Based Authorization

- Protect endpoints using the `[Authorize]` attribute with roles.

Example: Update `ProductsController` to restrict `POST`, `PUT`, and `DELETE` endpoints to the `Admin` role:

```
using Microsoft.AspNetCore.Authorization;

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

```
public ProductsController(ECommerceDbContext context)
{
_context = context;
}

[HttpGet]
public async Task<ActionResult<IEnumerable<Product>>> GetProducts()
{
return await _context.Products.ToListAsync();
}

[HttpGet("{id}")]
public async Task<ActionResult<Product>> GetProduct(int id)
{
var product = await _context.Products.FindAsync(id);

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

return product;
}

[Authorize(Roles = "Admin")]
[HttpPost]
public async Task<ActionResult<Product>> PostProduct(Product product)
{
_context.Products.Add(product);
await _context.SaveChangesAsync();

return CreatedAtAction(nameof(GetProduct), new { id = product.Id }, product);
}

[Authorize(Roles = "Admin")]
[HttpPut("{id}")]
public async Task<IActionResult> PutProduct(int id, Product product)
{
if (id != product.Id)
{
return BadRequest();
}
```

```
        _context.Entry(product).State = EntityState.Modified;

        try
        {
        await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
        if (!ProductExists(id))
        {
                return NotFound();
        }
        else
        {
                throw;
        }
        }

        return NoContent();
        }

        [Authorize(Roles = "Admin")]
        [HttpDelete("{id}")]
        public async Task<IActionResult> DeleteProduct(int id)
        {
        var product = await _context.Products.FindAsync(id);
        if (product == null)
        {
        return NotFound();
        }

        _context.Products.Remove(product);
        await _context.SaveChangesAsync();

        return NoContent();
        }

        private bool ProductExists(int id)
        {
        return _context.Products.Any(e => e.Id == id);
        }
}
```

5. Test the Role-Based Access Control

1. **Register an Admin User**:
    - Endpoint: `POST /api/User/register`
    - Body:

```
{
  "username": "adminuser",
  "passwordHash": "adminpassword",
  "role": "Admin"
}
```

Login as Admin:

- Endpoint: `POST /api/User/login`
- Body:

```
{
  "username": "adminuser",
  "passwordHash": "adminpassword"
}
```

- Use the token received in the response to access admin-protected endpoints.

Register a Customer User:

- Endpoint: `POST /api/User/register`
- Body:

```
{
  "username": "customeruser",
  "passwordHash": "customerpassword"
}
```

Test Access:

- Log in as the `Customer` and try accessing admin-only endpoints (e.g., `POST /api/Products`). You should receive a `403 Forbidden` response.
- Log in as the `Admin` and test access to all endpoints.

Part 4: Adding Categories

1. Create the Category Entity

- Add a new file `Models/Category.cs`:

```
namespace ECommerceAPI.Models
{
	public class Category
	{
	public int Id { get; set; }
	public string Name { get; set; }
	public string Description { get; set; }

	// Navigation property for related products
	public ICollection<Product> Products { get; set; }
	}
}
```

2. Update the Product Entity

Modify `Product.cs` to include a foreign key for `Category`:

```
namespace ECommerceAPI.Models
{
	public class Product
	{
	public int Id { get; set; }
	public string Name { get; set; }
	public string Description { get; set; }
	public decimal Price { get; set; }
	public int Stock { get; set; }

	// Foreign key for Category
	public int CategoryId { get; set; }
	public Category Category { get; set; } // Navigation property
	}
}
```

3. Update the Database Context

- Update `ECommerceDbContext` to include `Categories` and configure the relationship:

```
public DbSet<Category> Categories { get; set; }

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
        modelBuilder.Entity<Category>()
        .HasMany(c => c.Products)
        .WithOne(p => p.Category)
        .HasForeignKey(p => p.CategoryId);

        base.OnModelCreating(modelBuilder);
}
```

Add a migration and update the database:

```
dotnet ef migrations add AddCategoryEntity
dotnet ef database update
```

4. Create the Categories Controller

- Add a new controller `Controllers/CategoriesController.cs`:

```
using ECommerceAPI.Data;
using ECommerceAPI.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;

namespace ECommerceAPI.Controllers
{
        [Route("api/[controller]")]
        [ApiController]
        [Authorize]
        public class CategoriesController : ControllerBase
        {
        private readonly ECommerceDbContext _context;

        public CategoriesController(ECommerceDbContext context)
        {
        _context = context;
        }

        // GET: api/Categories
        [HttpGet]
        public async Task<ActionResult<IEnumerable<Category>>> GetCategories()
```

```
        {
        return await _context.Categories.Include(c => c.Products).ToListAsync();
        }

        // GET: api/Categories/5
        [HttpGet("{id}")]
        public async Task<ActionResult<Category>> GetCategory(int id)
        {
        var category = await _context.Categories.Include(c =>
c.Products).FirstOrDefaultAsync(c => c.Id == id);

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

        return category;
        }

        // POST: api/Categories
        [Authorize(Roles = "Admin")]
        [HttpPost]
        public async Task<ActionResult<Category>> PostCategory(Category category)
        {
        _context.Categories.Add(category);
        await _context.SaveChangesAsync();

        return CreatedAtAction(nameof(GetCategory), new { id = category.Id }, category);
        }

        // PUT: api/Categories/5
        [Authorize(Roles = "Admin")]
        [HttpPut("{id}")]
        public async Task<IActionResult> PutCategory(int id, Category category)
        {
        if (id != category.Id)
        {
                return BadRequest();
        }

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

        try
        {
```

```
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!CategoryExists(id))
            {
            return NotFound();
            }
            else
            {
            throw;
            }
        }

        return NoContent();
        }

        // DELETE: api/Categories/5
        [Authorize(Roles = "Admin")]
        [HttpDelete("{id}")]
        public async Task<IActionResult> DeleteCategory(int id)
        {
        var category = await _context.Categories.FindAsync(id);
        if (category == null)
        {
            return NotFound();
        }

        _context.Categories.Remove(category);
        await _context.SaveChangesAsync();

        return NoContent();
        }

        private bool CategoryExists(int id)
        {
        return _context.Categories.Any(c => c.Id == id);
        }
        }
}
```

5. Update the Products Controller

- Modify the `ProductsController` to include category information when fetching products:

```
[HttpGet]
public async Task<ActionResult<IEnumerable<Product>>> GetProducts()
{
        return await _context.Products.Include(p => p.Category).ToListAsync();
}

[HttpGet("{id}")]
public async Task<ActionResult<Product>> GetProduct(int id)
{
        var product = await _context.Products.Include(p => p.Category).FirstOrDefaultAsync(p
=> p.Id == id);

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

        return product;
}
```

6. Test the Endpoints

Use a tool like Postman or Swagger to test the following endpoints:

1. **Get all categories**:
    - `GET /api/Categories`
2. **Get a category by ID**:
    - `GET /api/Categories/{id}`
3. **Create a new category (Admin only)**:
    - `POST /api/Categories`
    - Body:

```
{
  "name": "Electronics",
  "description": "Devices and gadgets"
}
```

Update a category (Admin only):

- `PUT /api/Categories/{id}`

- Body:

```
{
  "id": 1,
  "name": "Updated Electronics",
  "description": "Updated devices and gadgets"
}
```

Delete a category (Admin only):

- `DELETE /api/Categories/{id}`

Create a product with a category:

- `POST /api/Products`
- Body:

```
{
  "name": "Smartphone",
  "description": "Latest model",
  "price": 999.99,
  "stock": 10,
  "categoryId": 1
}
```

Part 5: Customer Management

In this part, we'll add functionality to manage customers in the e-commerce API. We'll create a `Customer` entity, set up CRUD operations, and validate customer data to ensure proper handling of user input.

1. Create the Customer Entity

- Add a new file `Models/Customer.cs`:

```
namespace ECommerceAPI.Models
{
        public class Customer
        {
        public int Id { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
```

```
        public string Email { get; set; }
        public string PhoneNumber { get; set; }
        public string Address { get; set; }
        }
}
```

2. Update the Database Context

- Add the `Customer` DbSet to `ECommerceDbContext`:

```
public DbSet<Customer> Customers { get; set; }
```

Run a new migration and update the database:
dotnet ef migrations add AddCustomerEntity
dotnet ef database update

3. Create the Customers Controller

- Add a new controller `Controllers/CustomersController.cs`:

```
using ECommerceAPI.Data;
using ECommerceAPI.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;

namespace ECommerceAPI.Controllers
{
        [Route("api/[controller]")]
        [ApiController]
        [Authorize]
        public class CustomersController : ControllerBase
        {
        private readonly ECommerceDbContext _context;

        public CustomersController(ECommerceDbContext context)
        {
        _context = context;
        }

        // GET: api/Customers
        [HttpGet]
        public async Task<ActionResult<IEnumerable<Customer>>> GetCustomers()
```

```
{
return await _context.Customers.ToListAsync();
}

// GET: api/Customers/5
[HttpGet("{id}")]
public async Task<ActionResult<Customer>> GetCustomer(int id)
{
var customer = await _context.Customers.FindAsync(id);

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

return customer;
}

// POST: api/Customers
[HttpPost]
public async Task<ActionResult<Customer>> PostCustomer(Customer customer)
{
if (!IsValidEmail(customer.Email))
{
        return BadRequest("Invalid email format.");
}

_context.Customers.Add(customer);
await _context.SaveChangesAsync();

return CreatedAtAction(nameof(GetCustomer), new { id = customer.Id }, customer);
}

// PUT: api/Customers/5
[HttpPut("{id}")]
public async Task<IActionResult> PutCustomer(int id, Customer customer)
{
if (id != customer.Id)
{
        return BadRequest();
}

if (!IsValidEmail(customer.Email))
{
```

```
        return BadRequest("Invalid email format.");
}

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

try
{
        await _context.SaveChangesAsync();
}
catch (DbUpdateConcurrencyException)
{
        if (!CustomerExists(id))
        {
        return NotFound();
        }
        else
        {
        throw;
        }
}

return NoContent();
}

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

_context.Customers.Remove(customer);
await _context.SaveChangesAsync();

return NoContent();
}

private bool CustomerExists(int id)
{
return _context.Customers.Any(e => e.Id == id);
}
```

```
        private bool IsValidEmail(string email)
        {
        return email.Contains("@") && email.Contains(".");
        }
        }
}
```

4. Validate Customer Data

In the `PostCustomer` and `PutCustomer` methods, we validate:

- **Email Format**: Ensure the email is valid.
- **Phone Number**: (Optional) You can add additional validation for the phone number.

---

5. Test the Endpoints

Use a tool like Postman or Swagger to test the following endpoints:

1. **Get all customers**:
    - `GET /api/Customers`
2. **Get a customer by ID**:
    - `GET /api/Customers/{id}`
3. **Create a new customer**:
    - `POST /api/Customers`
    - Body:

```
{
  "firstName": "John",
  "lastName": "Doe",
  "email": "john.doe@example.com",
  "phoneNumber": "123-456-7890",
  "address": "123 Main St, Springfield, USA"
}
```

Update a customer:

- `PUT /api/Customers/{id}`
- Body:

```
{
  "id": 1,
```

```
  "firstName": "John",
  "lastName": "Doe",
  "email": "updated.email@example.com",
  "phoneNumber": "123-456-7890",
  "address": "456 Elm St, Springfield, USA"
}
```

Delete a customer:

- `DELETE /api/Customers/{id}`

6. Extend Product and Order Integration (Optional)

Later, customers will be associated with orders. For now, this entity is stand-alone, but you can plan relationships like:

- **Customer-Order**: Each order is placed by a customer.
- **Customer Address Validation**: Add regex to validate addresses.