UNIVERISTY OF PIRAEUS - DEPARTMENT OF INFORMATICS

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ – ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

MSc «INFORMATICS»

ΠΜΣ «ΠΛΗΡΟΦΟΡΙΚΗ»

MSc Thesis

Μεταπτυχιακή Διατριβή

Thesis Title: Τίτλος Διατριβής:	**Development of a WebAPI software** Ανάπτυξη λογισμικού WebAPI
Student's name-surname: Ονοματεπώνυμο φοιτητή:	**THEODOROS MOUZAKITIS** ΘΕΟΔΩΡΟΣ ΜΟΥΖΑΚΙΤΗΣ
Father's name: Πατρώνυμο:	**MICHAIL** ΜΙΧΑΗΛ
Student's ID No: Αριθμός Μητρώου:	ΜΠΠΛ17030
Supervisor: Επιβλέπων:	**EFTHIMIOS ALEPIS, Associate Professor** ΕΥΘΥΜΙΟΣ ΑΛΕΠΗΣ, Αναπληρωτής Καθηγητής

July 2023/ Ιούλιος 2023

3-Member Examination Committee

Τριμελής Εξεταστική Επιτροπή

Efthimios Alepis	**Maria Virvou**	**Constantinos Patsakis**
Associate Professor	**Professor**	**Associate Professor**
Αλέπης Ευθύμιος	Μαρία Βίρβου	Κωνσταντίνος Πατσάκης
Αναπληρωτής Καθηγητής	Καθηγήτρια	Αναπληρωτής Καθηγητής

Table of Contents

Summary

As the developer tasked with implementing the API for managing Customers, Orders, and Products, I am passionate about creating efficient and robust solutions that meet the needs of the modern software landscape. With a strong foundation in C# and a comprehensive understanding of software design principles, I embarked on the challenge of developing an API that adheres to best practices and incorporates essential features.

Throughout the implementation process, I prioritized the utilization of industry-standard tools and patterns to ensure a scalable and maintainable solution. By employing Entity Framework as the persistence framework and implementing the repository pattern, I established a clear separation between the application logic and data persistence concerns. The inclusion of the unit of work pattern further enhanced data integrity and consistency, guaranteeing reliable operations on multiple entities within a single transaction.

To fulfill the requirements of the project, I integrated CRUD operations for Products, enabling administrators to manage the master data while ensuring the integrity of the system. Additionally, I ensured that the API handles scenarios where multiple Customers may share the same name, reflecting the real-world complexity of customer data management.

Recognizing the significance of documentation, I meticulously documented the API using XML comments, providing comprehensive and easily accessible information about its structure, usage, and available methods. By incorporating unit tests using the NUnit framework, I verified the behavior and functionality of the API, ensuring its correctness, reliability, and adherence to specifications.

Embracing modern software development principles, I implemented Domain-Driven Design (DDD) to focus on the core domain logic and encapsulate business rules within the appropriate entities. The adoption of the Command Query Responsibility Segregation (CQRS) pattern allowed for a clear separation between write and read operations, optimizing performance and scalability.

Throughout the project, I made informed decisions based on assumptions and considerations, striving to fulfill the objectives of the API. I ensured that the API supports necessary validations, such as the maximum length of fields, the requirement for a Customer name, and constraints on price and quantity. I also addressed crucial aspects like the non-deletion of Products and the exclusive availability of the Products controller to administrators.

In conclusion, this paper outlines my journey in designing and implementing an API that manages Customers, Orders, and Products, showcasing my technical skills, attention to detail, and commitment to delivering a high-quality solution. By incorporating industry-standard practices, patterns, and documentation, I have developed an API that meets the project requirements and demonstrates my proficiency as a developer in designing and implementing reliable and scalable systems.

Introduction

In the dynamic landscape of software development, designing and implementing efficient application programming interfaces (APIs) is vital for building robust systems. This paper presents the implementation of a C# API that manages Customers, Orders, and Products for a hypothetical system. The objective of this API is to provide comprehensive functionality for creating, updating, and deleting Customers, managing Orders, and performing CRUD operations on Products, without the need for a user interface. The API utilizes a local database for data storage, ensuring a self-contained environment for managing the system's information. No authentication is required to access the API, allowing for simplified testing and development scenarios.

When creating a Customer, the API mandates that a name is provided, while the address and postal code fields are optional. This flexibility accommodates cases where customers may not have an address or postal code on file.

To ensure data integrity and consistency, the API imposes maximum length limits on various fields. These limits prevent the storage of excessively long or invalid data, promoting efficient and well-structured information management.

For the price and quantity fields, two decimal places are deemed sufficient for accuracy and precision. This restriction optimizes storage efficiency without sacrificing the necessary level of detail for financial calculations.

To maintain data validity, the API enforces that the quantity of a Product cannot be negative or zero. This constraint ensures that only meaningful and realistic quantities are recorded for each Product.

When creating an Order, the API requires that both the associated Customer and the Products already exist in the system. This requirement prevents the creation of Orders with non-existent or invalid references, promoting data integrity and avoiding potential inconsistencies.

To safeguard the integrity of the system, Products are not allowed to be deleted. This limitation ensures that Product data remains intact and prevents accidental removal of critical information. However, the Products controller, responsible for manipulating master data, is restricted to administrative access only. This restriction ensures that regular API consumers do not have access to modify the Products data, maintaining control over the system's core product information.

The API accounts for the possibility of multiple Customers having the same name. This consideration acknowledges scenarios where individuals with identical names may exist within the system, allowing for accurate representation and management of customer data.

By incorporating these additional requirements, the API implementation addresses the challenges of managing Customers, Orders, and Products in a controlled and consistent manner. It provides a flexible and reliable foundation for building systems that meet the specific needs of the business domain, while adhering to modern software development principles and patterns.

Field review

Here are some real-world examples of applications that share similarities with the features and functionality described in my paper:

Shopify: Shopify is an e-commerce platform that provides an API for managing customers, orders, and products. It allows developers to create, update, and delete customers, manage orders, and perform CRUD operations on products. The API supports features like order management, inventory management, and customer management.

WooCommerce: WooCommerce is a popular WordPress plugin that enables users to create and manage online stores. It offers an API for handling customers, orders, and products. The API allows for creating, updating, and deleting customers, managing orders, and performing CRUD operations on products.

Square: Square is a comprehensive payment and point-of-sale solution for businesses. It provides an API that allows developers to integrate customer and order management features into their applications. The API supports creating and managing customer profiles, processing payments, and managing orders.

Stripe: Stripe is a payment processing platform that offers an API for managing customers, orders, and payments. The API allows developers to create, update, and delete customer profiles, manage subscriptions, and process payments securely.

OpenCart: OpenCart is an open-source e-commerce platform that provides an API for managing customers, orders, and products. The API enables developers to create, update, and delete customers, manage orders, and perform CRUD operations on products.

These examples demonstrate the implementation of APIs that encompass similar functionalities to the ones outlined in my paper. They can serve as references to understand how real-world applications handle customer and order management, product manipulation, and integration with various systems.

User's Manual

API for Customers, Orders, and Products Management

Thank you for choosing the API for Customers, Orders, and Products Management. This user's manual provides an overview of the API's functionality and guidelines for interacting with it effectively.

Table of Contents

Introduction

The API for Customers, Orders, and Products Management is designed to facilitate the creation, retrieval, updating, and deletion of Customers, Orders, and Products. This user's manual aims to guide you through the process of utilizing the API effectively.

API Authentication

The API does not require authentication to access its endpoints. However, appropriate security measures should be implemented in your application to protect sensitive data and prevent unauthorized access. The current project does not support native authorization and should BE implemented behind a user management layer that handles authentication and authorization.

API Base URL

The base URL for accessing the API is [https://localhost:7035/]. Please ensure that you prefix the API endpoint URLs with this base URL when making API requests.

Endpoints

Customers

The Customers endpoint allows you to manage customer-related operations.

- GET /customers
  Retrieves all customers from the database.
  Method: GET
  URL: `/customers`
  Response: Returns a collection of customers.
- GET /customers/{id}
  Retrieves the corresponding customer based on the provided id.
  Method: GET
  URL: /customers/{id}
  Parameters:

Id: Unique identifier for the requested customer.
Response: Returns the customer information.

- POST /customers
  Creates a new customer and returns the generated customer ID.
  Method: POST
  URL: `/customers`
  Request Body: Model for customer creation.
  Response: Returns the generated customer ID if the creation is successful. Otherwise, returns an error status code (400) indicating the failure.
- PUT /customers/{id}
  Updates the customer corresponding to the provided `id`.
  Method: PUT
  URL: `/customers/{id}`
  Parameters:
  id: Customer's ID.
  Request Body: Model for customer update.
  Response: Returns an error status code (400) if the update fails. Otherwise, returns a success status code (200).
- DELETE /customers/{id}
  Deletes a customer based on the provided `id`.
  Method: DELETE
  URL: `/customers/{id}`
  Parameters:
  id: The ID of the customer to be deleted.
  Response: Returns an error status code (400) if the delete fails. Otherwise, returns a success status code (200).

Orders

The Orders endpoint allows you to manage order-related operations.

- GET /orders
  Retrieves all orders from the database.
  Method: GET
  URL: `/orders`
  Response: Returns a collection of orders.
- GET /orders/{id}
  Retrieves the corresponding order based on the provided id.
  Method: GET
  URL: `/orders/{id}`
  Parameters:
  `id`: The ID for the requested order.
  Response: Returns the order information.
- GET /orders/customer/{customerId}
  Retrieves all orders of the customer with the provided `customerId`, sorted by the orders' date.
  Method: GET
  URL: `/orders/customer/{customerId}`
  Parameters:
  `customerId`: Customer's ID.
  Response: Returns the orders associated with the customer. If the customer has no orders or doesn't exist, returns a success status code (204) indicating the absence of content.

- POST /orders
  Creates a new order and returns the generated order ID.
  Method: POST
  URL: `/orders`
  Request Body: Model for order creation.
  Response: Returns the generated order ID if the creation is successful. Otherwise, returns an error status code (400) indicating the failure.
- PUT /orders/{id}
  Updates the order corresponding to the provided `id`.
  Method: PUT
  URL: **`/**orders/{id}`
  Parameters:
  `Id`: Order's ID.
  Request Body: Model for order update.
  Response: Returns an error status code (400) if the update fails. Otherwise, returns a success status code (200).
- DELETE /orders/{id}
  Deletes an order based on the provided `id`.
  Method: DELETE
  URL: `/orders/{id}`
  Parameters:
  `id`: The ID of the order to be deleted.
  Response: Returns an error status code (400) if the delete fails. Otherwise, returns a success status code (200).
- GET /orders/ofcustomer/{customerId}
  Gets all orders of the customer with the provided `customerId`, sorted by the order dates.
  Method: GET
  URL: `/orders/ofcustomer/{customerId}`
  Parameters:
  `customerId`: Customer's ID.
  Response: Returns the orders associated with the customer, sorted by the order dates. If the customer has no orders or doesn't exist, returns a success status code (204) indicating the absence of content.

Products

The Products endpoint allows you to manage product-related operations.

- GET /products
  Retrieves all products from the database.
  Method: GET
  URL: `/products`
  Response: Returns a collection of products.
- GET /products/{id}
  Retrieves the corresponding product based on the provided `id`.
  Method: GET
  URL: `/products/{id}`
  Parameters:
  `id`: Unique identifier for the requested product.
  Response: Returns the product information.
- POST /products
  Creates a new product and returns the generated product ID.
  Method: POSTURL: `/products`

Request Body: Model for product creation.
Response: Returns the generated product ID if the creation is successful. Otherwise, returns an error status code (400) indicating the failure.

- PUT /products/{id}
  Updates the product corresponding to the provided id.
  Method: PUT
  URL: `/products/{id}`
  Parameters:
  `id`: Product's ID.
  Request Body: Model for product update.
  Response: Returns an error status code (400) if the update fails. Otherwise, returns a success status code (200).
- DELETE /products/{id}
  Deletes a product based on the provided `id`.
  Method: DELETE
  URL: `/products/{id}`
  Parameters:
  `id`: The ID of the product to be deleted.
  Response: Returns an error status code (400) if the delete fails. Otherwise, returns a success status code (200).

Architecture

The architecture of the project was designed with a focus on domain logic and draws inspiration from the principles of Domain-Driven Design (DDD). By following DDD principles, the project aims to align closely with the business domain and ensure a clear separation of concerns. Utilizing the following concepts, we achieve a serviceable, maintainable, and robust codebase.

Dependency Injection

The project utilizes Dependency Injection to achieve loose coupling and improve testability and maintainability. By decoupling dependencies, different components can be easily swapped or mocked during testing, promoting flexibility and modularity.

Command Query Responsibility Segregation

The Command Query Responsibility Segregation (CQRS) pattern was employed to segregate commands (write operations) and queries (read operations) into separate paths. This separation allows for optimized handling of read and write operations, improving performance and scalability. For the current implementation, the Mediator pattern is utilized to mediate communication between the command and query handlers, enabling loose coupling and promoting reusability.

Mediator Pattern

In object-oriented programming, objects often need to communicate with each other to perform certain tasks or exchange information. However, directly coupling objects together can lead to tight dependencies, making the system less flexible and harder to maintain. This is where the Mediator pattern comes in.

The Mediator pattern aims to reduce the direct dependencies between communicating objects by introducing a central component called the mediator. Instead of objects directly communicating with each other, they interact with the mediator, which encapsulates the communication logic and facilitates the exchange of information between the objects.

Here are some key points to elaborate on regarding the reduction of direct dependencies:

1. Loose Coupling:
   The Mediator pattern promotes loose coupling between objects by removing their direct knowledge and dependency on each other. Objects don't need to have explicit references to other objects they need to communicate with. Instead, they rely on the mediator to handle the communication.
2. Decoupled Communication:
   By relying on the mediator, objects can communicate without needing to know the details of how the other objects are implemented. They only need to understand the mediator's interface and the messages they can send and receive. This decoupling allows for greater flexibility and extensibility as objects can interact with new objects or be modified without affecting the entire system.
3. Simplified Object Interfaces:
   With the Mediator pattern, objects only need to expose a simplified interface for communication with the mediator. This interface typically includes methods or events to send and receive messages. By having a standardized interface, objects can focus on their core responsibilities, resulting in cleaner and more maintainable code.
4. Centralized Control:
   The mediator acts as a centralized control unit that manages the communication and coordination between objects. It becomes responsible for routing messages between objects, handling message routing logic, and managing the state of the communication process. This centralized control allows for better organization and control over complex communication flows.
5. Independence of Object Changes:
   Objects participating in the mediation process can evolve independently. Modifying or adding new objects to the system typically requires changes only within the mediator and the objects themselves. Other objects in the system are not directly affected, as they communicate through the mediator's interface.

By reducing direct dependencies between communicating objects, the Mediator pattern enhances the flexibility, maintainability, and extensibility of the system. It allows objects to interact in a decoupled manner, simplifies object interfaces, provides centralized control over communication, and enables independent evolution of participating objects. This results in a more modular and loosely coupled architecture, making the system easier to understand, modify, and maintain.

Entity Framework

The project leverages Entity Framework, a popular object-relational mapping (ORM) framework, to handle database interactions. Entity Framework simplifies data access and provides an abstraction layer that maps the domain entities to the underlying database tables.

Data Annotations

Data Annotations are used in conjunction with Entity Framework to define metadata and validation rules for the domain entities. These annotations allow for declarative configuration of the entities, ensuring data integrity and enforcing business rules.

Unit Testing

The project incorporates unit tests using the NUnit framework to verify the behavior and correctness of the implemented functionalities. Unit tests enable early detection of bugs, aid in code refactoring, and ensure that the project functions as expected.

Unit testing is a fundamental practice in software development that involves testing individual units of code to ensure they function correctly in isolation. In the context of our web API creation objective, unit testing plays a crucial role in verifying the functionality, reliability, and robustness of the API endpoints and business logic. In this subchapter, we will explore the key concepts and benefits of unit testing in the context of our web API.

Overview of Unit Testing

Unit testing involves writing automated tests that target specific units of code, such as individual methods or classes, and verify their behavior against expected outcomes. The goal is to isolate each unit of code and test it in isolation, independently of other components or external dependencies.

Key concepts in unit testing include:

- Test Cases:
  A test case is a set of inputs, preconditions, and expected outputs designed to exercise a specific unit of code. Test cases should cover a range of scenarios to ensure comprehensive coverage and identify potential issues or bugs.
- Test Framework:
  A test framework provides the tools and utilities to write and execute unit tests. It typically includes assertion libraries to compare expected and actual results, test runners to execute the tests, and reporting mechanisms to provide feedback on the test outcomes.
- Test Doubles:
  Test doubles, such as mocks, stubs, and fakes, are used to simulate external dependencies or collaborator objects to isolate the unit under test. They help control the behavior of external dependencies and enable focused testing of the unit.

Benefits of Unit Testing

Unit testing offers several benefits in the context of our web API creation objective:

- Early Bug Detection:
  By identifying issues at the unit level, unit testing helps catch bugs early in the development process. It allows for quick feedback on the correctness of individual units of code, enabling prompt debugging and fixing of issues.
- Improved Code Quality:
  Writing unit tests often leads to better code design and improved quality. It promotes modular, reusable, and loosely coupled code by enforcing encapsulation and separation of concerns. Unit testing encourages the use of best practices such as dependency injection, single responsibility principle, and testable design patterns.
- Regression Testing:
  Unit tests serve as a safety net against unintended regressions. As the codebase evolves, unit tests can be executed automatically to ensure that existing functionality remains intact after modifications or refactoring.
- Documentation and Collaboration:
  Well-written unit tests act as living documentation for the codebase. They provide insights into the expected behavior and usage of the units, making it easier for other developers to understand and collaborate on the project.

Unit Testing Web API Components

In the context of our web API, unit testing can be applied to various components, including:

- API Endpoints:
  Unit tests can verify the correct handling of incoming requests and the generation of

appropriate responses. They can validate input validation, error handling, authentication/authorization mechanisms, and adherence to API contracts.

- Business Logic:
  Unit tests can validate the behavior and correctness of the business logic components of the web API. They can ensure that business rules, calculations, and data transformations are functioning as expected.
- Data Access Layer:
  Unit tests can validate the interaction between the web API and the underlying data access layer. They can verify the correctness of CRUD operations, data integrity, and proper handling of database transactions.

Testing Frameworks and Tools

To facilitate unit testing in our web API project, we will utilize a testing framework such as NUnit. NUnit provides a rich set of features, including assertion libraries, test runners, and reporting capabilities. Additionally, we can utilize other tools and libraries, such as mocking frameworks like Moq, to create test doubles and simulate external dependencies.

Test-Driven Development (TDD)

Test-Driven Development (TDD) is a development practice that involves writing tests before implementing the corresponding code. It follows a red-green-refactor cycle, where tests are written to fail initially, then the necessary code is implemented to make the tests pass, and finally, the code and tests are refactored for clarity and maintainability. TDD can provide additional benefits, such as improved code coverage, design clarity, and faster feedback loops.

Test Coverage and Continuous Integration

To ensure comprehensive testing, we will strive for high test coverage, aiming to cover critical code paths and edge cases. Test coverage tools, such as OpenCover or DotCover, can be utilized to measure the percentage of code covered by unit tests. Additionally, integrating unit tests into a continuous integration pipeline, using tools like Jenkins or Azure DevOps, can automate the execution of tests and provide timely feedback on the health of the codebase.

Repository Pattern with Unit of Work

The Repository pattern is employed to encapsulate the data access logic and provide a consistent and simplified interface for working with data entities. The Unit of Work pattern is utilized to manage transactions and ensure atomicity when working with multiple entities within a single operation.

By adopting these architectural modules, the project achieves a modular and maintainable codebase that adheres to best practices. The use of Dependency Injection, CQRS with Mediator Pattern, Entity Framework, Data Annotations, Unit Testing with NUnit, and the Repository Pattern with Unit of Work collectively contribute to a scalable, testable, and domain-driven architecture.

By structuring the project in this manner, it becomes easier to understand, maintain, and extend the application as it evolves over time. The architecture promotes modularity, separation of concerns, and adherence to domain-driven principles, resulting in a robust and flexible solution.

Repository Pattern

The Repository Pattern is a design pattern commonly used in the development of web APIs to abstract and encapsulate data access logic. It provides a layer of separation between the data access code and the rest of the application, promoting modularity, maintainability, and testability. In this subchapter, we

will explore the key concepts and benefits of the Repository Pattern in the context of our web API creation objective.

The Repository Pattern introduces the concept of repositories, which act as mediators between the application's business logic and the underlying data storage. Repositories encapsulate the operations required for data access, retrieval, modification, and deletion, providing a clean and consistent interface for interacting with the data.

The primary goals of the Repository Pattern are as follows:

- Abstraction of Data Access:
  By utilizing repositories, the details of how the data is stored and accessed are abstracted away from the rest of the application. This abstraction simplifies the codebase, as other components do not need to be aware of the specific data access implementation details.
- Separation of Concerns:
  The Repository Pattern facilitates the separation of concerns between the business logic and data access layers. Business logic interacts with repositories using a well-defined interface, without being tightly coupled to the underlying data storage technology or implementation.
- Centralized Data Access Logic:
  Repositories consolidate the data access logic within a single component, allowing for consistent implementation of common data operations across the application. This centralization avoids code duplication and promotes maintainability by having a single point of change for data access behavior.
- Testability:
  The Repository Pattern enhances testability by enabling the use of mock repositories during unit testing. By abstracting the data access operations behind a repository interface, the business logic can be easily tested in isolation without requiring a live database connection.

Repository Pattern Implementation

The implementation of the Repository Pattern typically involves the following components:

- Repository Interface:
  The repository interface defines a set of methods that represent the common data operations such as Create, Read, Update, and Delete (CRUD) operations. It acts as a contract between the business logic and the repository implementation.
- Concrete Repository:
  The concrete repository implements the repository interface and provides the actual implementation for the data access operations. It interacts with the underlying data storage, which could be a relational database, document database, or any other data storage mechanism.
- Data Context:
  The data context represents the connection or session with the underlying data storage. It provides the necessary infrastructure to perform data access operations and manages the lifecycle of the data access operations within a unit of work.

Benefits of the Repository Pattern

The Repository Pattern offers several benefits in the context of our web API creation objective:

- Improved Maintainability:
  By encapsulating data access logic within repositories, changes to the underlying data storage or schema can be easily accommodated without affecting the rest of the application. This modularity and separation of concerns contribute to improved maintainability.

- Flexibility in Data Source:
  The Repository Pattern allows for flexibility in choosing different data sources or storage technologies without impacting the business logic. It enables seamless switching between different databases or even transitioning to a different data storage approach, such as using a web service instead of a local database.
- Code Reusability:
  Repositories encapsulate common data access operations, promoting code reusability across different components of the application. By utilizing the same repository interfaces, the same data access logic can be shared among multiple parts of the application, reducing code duplication, and promoting consistency.
- Testability and Mockability:
  The abstraction provided by repositories facilitates easier testing by allowing the use of mock repositories during unit testing. Business logic can be tested independently without relying on the actual data storage, improving testability and reducing dependencies on external resources.

Application of the Repository Pattern in our Web API

In our web API implementation, we will utilize the Repository Pattern to separate the data access logic from the business logic. We will define repository interfaces for each entity or aggregate in our domain model, and concrete repository implementations will handle the interaction with the chosen data storage, such as a relational database.

The repositories will provide the necessary methods for creating, reading, updating, and deleting entities, ensuring consistency and uniformity in data access operations. The business logic components of our web API will interact with these repositories through the defined interfaces, promoting loose coupling and modularity.

By adopting the Repository Pattern, our web API will benefit from improved maintainability, flexibility in data sources, code reusability, and enhanced testability. These advantages will contribute to a scalable and maintainable architecture that can easily evolve and adapt to changing requirements.

Unit Of Work

The Unit of Work pattern is a design pattern commonly used in the development of web APIs to manage transactions and ensure consistency when working with multiple entities within a single operation. It provides a way to treat multiple database operations as a single logical unit, enabling atomicity, integrity, and concurrency control. In this subchapter, we will explore the key concepts and benefits of the Unit of Work pattern in the context of our web API creation objective.

Overview of the Unit of Work Pattern

The Unit of Work pattern introduces the concept of a unit of work, which represents a cohesive set of operations that should be treated as a single transaction. It allows you to group multiple database operations together and ensures that they are either all committed, or all rolled back, providing transactional consistency.

The primary goals of the Unit of Work pattern are as follows:

- Transaction Management: The Unit of Work pattern helps manage database transactions by encapsulating a group of related database operations within a single transaction. It ensures that all operations within the unit of work are either committed or rolled back together, maintaining data integrity and consistency.
- Atomicity: By treating multiple operations as a single unit of work, the pattern ensures that all operations are executed or none. If any operation within the unit of work fails, the entire

unit of work is rolled back, preventing partial updates, and maintaining the integrity of the data.

- Concurrency Control: The Unit of Work pattern helps manage concurrent access to data by providing a centralized control point for managing transactional operations. It ensures that only one transaction can modify the data being worked on, preventing conflicts and maintaining data consistency.

Unit of Work Implementation

The implementation of the Unit of Work pattern typically involves the following components:

- Unit of Work:
  The unit of work is a central component that represents the context for a group of related database operations. It provides methods for managing the transaction, such as starting, committing, and rolling back the transaction. It also coordinates the persistence operations across multiple repositories.
- Repositories:
  Repositories are responsible for performing data access operations for specific entities or aggregates. Within the unit of work, repositories work together to perform database operations as part of a single transaction. They communicate with the unit of work to coordinate their operations and ensure consistency.

Benefits of the Unit of Work Pattern

The Unit of Work pattern offers several benefits in the context of our web API creation objective:

- Transactional Consistency:
  The Unit of Work pattern ensures that multiple database operations within a single unit of work are treated as a single transaction. This guarantees that all operations are committed or rolled back together, maintaining data integrity and consistency.
- Atomicity:
  By treating multiple operations as a single unit, the Unit of Work pattern ensures atomicity. If any operation within the unit of work fails, the entire unit of work is rolled back, preventing inconsistent or partially updated data.
- Concurrency Control:
  The Unit of Work pattern provides a centralized control point for managing transactional operations. It helps prevent conflicts and data corruption by ensuring that only one transaction can modify the data being worked on at a time.
- Improved Performance:
  By grouping related operations into a single unit of work, the Unit of Work pattern can optimize performance by reducing the number of databases round-trips. It allows for efficient batching and optimizing database operations.

Application of the Unit of Work Pattern in our Web API

In our web API implementation, we will utilize the Unit of Work pattern to manage transactions and ensure consistency when working with multiple entities within a single operation. The unit of work will be responsible for coordinating the persistence operations across multiple repositories.

By encapsulating related database operations within a unit of work, we will ensure that changes to the data are treated atomically and consistently. The unit of work will manage the transaction and provide methods for starting, committing, or rolling back the transaction. It will coordinate with the repositories to ensure that all operations are performed within the scope of the transaction.

The Unit of Work pattern will enable us to achieve transactional consistency, atomicity, and concurrency control in our web API. It will contribute to the reliability, integrity, and performance of our data operations, providing a robust foundation for our application.

Entity Framework

Introduction to Entity Framework

Entity Framework (EF) is an open-source, object-relational mapping (ORM) framework developed by Microsoft. It allows .NET developers to interact with databases using .NET objects.

Entity Framework is a powerful tool that can help you to reduce the amount of code you need to write to access your database and improve the performance of your database access code. It is also extremely useful to maintain your database access code and abstract away the details of the database schema from your application code.

Entity Framework is a mature framework that has been used by many developers to build successful applications. It is a good choice for .NET developers who need to access databases.

Benefits

Using Entity Framework provides a multitude of benefits. It significantly reduces the amount of code you need to write and takes care of mapping your database tables to objects, so you don't have to write as much code to access your data. Additionally, it improves the performance of your database access code using a variety of techniques such as lazy loading and caching.

Makes it easier to maintain your database access code: Entity Framework uses a consistent object-oriented API, so your database access code is easier to read, understand, and maintain.

Abstracts away the details of the database schema from your application code: Entity Framework allows you to work with your data in terms of objects, so you don't have to worry about the details of the database schema.

If you are a .NET developer who needs to access databases, Entity Framework is a good choice. It is a powerful tool that can help you to reduce the amount of code you need to write, improve the performance of your database access code, and make it easier to maintain your database access code.

The concept of entities

In Entity Framework, an entity is a .NET object that represents a row in a database table. Entities are used to represent data in a way that is both object-oriented and relational.

There are two main types of entities in Entity Framework: POCO entities and complex entities.

POCO entities (Plain Old CLR Objects) are simple entities that do not have any relationships with other entities. POCO entities are the simplest type of entity and are the easiest to create.

Complex entities are entities that have relationships with other entities. Complex entities are more complex than POCO entities, but they provide more flexibility and functionality.

Benefits of using the entities of Entity Framework:

- Object-oriented: Entities are objects, so they can be used in a natural way with other object-oriented code.
- Relational: Entities are mapped to database tables, so they can be used to access data in a relational database.
- Abstraction: Entities abstract away the details of the database schema, so you don't have to worry about how the data is stored in the database.

- Different types of entities that can be created in Entity Framework:
- Value entities: Value entities are entities that represent simple data types, such as integers, strings, and dates.
- Reference entities: Reference entities are entities that represent complex data types, such as objects and collections.
- Derived entities: Derived entities are entities that are derived from other entities. Derived entities can be used to represent a specialization of a base entity.

Relationships

There are four types of relationships that can be created between entities:

- One-to-one: A one-to-one relationship is a relationship between two entities where each entity in the first entity can have only one corresponding entity in the second entity, and vice versa.
- One-to-many: A one-to-many relationship is a relationship between two entities where each entity in the first entity can have zero or more corresponding entities in the second entity, but each entity in the second entity can only have one corresponding entity in the first entity.
- Many-to-many: A many-to-many relationship is a relationship between two entities where each entity in the first entity can have zero or more corresponding entities in the second entity, and each entity in the second entity can have zero or more corresponding entities in the first entity.
- Self-referential: A self-referential relationship is a relationship between two entities of the same type.

The benefits of using relationships in Entity Framework:

- Represents real-world relationships: Relationships can be used to represent real-world relationships between entities. For example, a one-to-one relationship can be used to represent the relationship between a person and their address.
- Enforces referential integrity: Relationships can be used to enforce referential integrity in your database. For example, a one-to-many relationship can be used to ensure that a product cannot be deleted if there are still orders for that product.
- Improves performance: Relationships can be used to improve the performance of your database access code. For example, a one-to-many relationship can be used to fetch all the orders for a product in a single query.

Methods

Entity Framework provides several methods for performing CRUD operations on entities.

Create: To create an entity, you can use the Add method of the DbContext class.

Read: To read an entity, you can use the Find method of the DbContext class.

Update: To update an entity, you can use the Update method of the DbContext class.

Delete: To delete an entity, you can use the Delete method of the DbContext class.

Here are some additional considerations for performing CRUD operations in Entity Framework:

- Transactions: Entity Framework transactions are used to ensure that all of the changes made to an entity are committed to the database or rolled back if an error occurs.
- Validation: Entity Framework validation is used to ensure that the data in an entity is valid before it is saved to the database.
- Lazy loading: Lazy loading is a technique that is used to load entities only when they are needed. Lazy loading can improve the performance of your application by reducing the amount of data that is loaded from the database.

Migrations

This section would discuss how to use Entity Framework migrations to manage changes to your database schema. Entity Framework migrations are a way to manage changes to your database schema. Migrations allow you to track the changes that you make to your database schema and to apply those changes to your database in a consistent way.

To use Entity Framework migrations, you need to create a migration file. A migration file is a text file that contains the SQL statements that are used to make changes to your database schema.

To create a migration file, you can use the Add-Migration command-line tool or the Add-Migration method in Visual Studio. For example, the following command creates a migration file called InitialCreate:

Add-Migration

Once you have created a migration file, you can apply the changes to your database by running the Update-Database command-line tool or the Update-Database method in Visual Studio. For example, the following command applies the changes in the InitialCreate migration file to your database:

Update-Database

Entity Framework migrations provide several benefits, including:

- Trackability: Migrations allow you to track the changes that you make to your database schema. This makes it easy to see what changes have been made to your database and to revert those changes if necessary.
- Consistency: Migrations allow you to apply changes to your database in a consistent way. This ensures that your database schema is always consistent with your code.
- Reliability: Migrations are reliable. They have been used by many developers to manage changes to their database schemas.

Here are some additional considerations for using Entity Framework migrations:

- Versioning: Migrations use a versioning system to track the changes that you make to your database schema. This allows you to roll back changes to a previous version of your database schema if necessary.
- Dependencies: Migrations can depend on each other. This means that you need to apply the migrations in the correct order.
- Rollback: If you make a mistake when creating a migration file, you can roll back the changes by running the Rollback-Migration command-line tool or the Rollback-Migration method in Visual Studio.

Entity Framework Performance

To utilize the framework's full performance gain there are implementations techniques to consider:

- Use the right mapping strategy.
  The mapping strategy that you use can have a significant impact on the performance of your application. If you are using a complex mapping strategy, you may want to consider using a simpler mapping strategy.
- Use lazy loading.
  Lazy loading is a technique that is used to load entities only when they are needed. Lazy loading can improve the performance of your application by reducing the amount of data that is loaded from the database.

- Use indexes.
  Indexes can improve the performance of your application by making it easier for Entity Framework to find the data that you need.
- Use batching.
  Batching is a technique that is used to send multiple queries to the database at once. Batching can improve the performance of your application by reducing the number of round trips to the database.
- Use caching.
  Caching is a technique that is used to store frequently accessed data in memory. Caching can improve the performance of your application by reducing the number of times that data needs to be retrieved from the database.
- Use profiling.
  Profiling is a technique that is used to measure the performance of your application. Profiling can help you to identify the areas of your application that are causing performance problems.

There are also some considerations that affect each use case.

- The size of the database.
  The size of your database can have a significant impact on the performance of your application. If your database is large, you may want to consider using a different database engine.
- The number of concurrent users.
  The number of concurrent users that are accessing your application can also have a significant impact on the performance of your application. If you have many concurrent users, you may need to scale your application.
- The hardware that you are using.
  The hardware that you are using can also have a significant impact on the performance of your application. If you are using old or outdated hardware, you may need to upgrade your hardware.

Appendices

```
builder.Services.AddScoped<IUnitOfWork, UnitOfWork>();
builder.Services.AddScoped<ICustomerService, CustomerService>();
builder.Services.AddScoped<IOrderService, OrderService>();
builder.Services.AddScoped<IProductService, ProductService>();
```

Figure 1:Example Of Dependency Injection

```
[Table("Orders")]
[Index(nameof(DocDate))]
46 references
public class Order
{
    [Key]
    [Required]
    7 references
    public Guid Id { get; set; }

    [ForeignKey(nameof(Customer))]
    13 references
    public Guid CustomerId { get; set; }

    [Required]
    31 references
    public Customer Customer { get; set; } = null!;

    [Required]
    8 references
    public DateTime DocDate { get; set; }

    7 references
    public decimal DocTotal { get; set; }

    [Required]
    14 references
    public ICollection<Item> Items { get; set; } = null!;
}
```

Figure 2 Customer DB Model

```
public class ApplicationDbContext : DbContext
{
    0 references
    public ApplicationDbContext(DbContextOptions dbContextOptions) : base(dbContextOptions)
    {

    }

    5 references
    public DbSet<Customer> Customers { get; set; } = null!;
    6 references
    public DbSet<Order> Orders { get; set; } = null!;
    4 references
    public DbSet<Product> Products { get; set; } = null!;
    6 references
    public DbSet<Item> Items { get; set; } = null!;
}
```

Figure 3 DB Context

```
2 references
public class UnitOfWork : IUnitOfWork
{
    private readonly ApplicationDbContext _dbContext;
    14 references
    public ICustomerRepository Customers { get; private set; }
    9 references
    public IOrderRepository Orders { get; private set; }
    9 references
    public IProductRepository Products { get; private set; }
    7 references
    public IItemRepository Items { get; private set; }

    0 references
    public UnitOfWork(ApplicationDbContext dbContext)
    {
        _dbContext = dbContext;
        Customers = new CustomerRepository(dbContext);
        Orders = new OrderRepository(dbContext);
        Products = new ProductRepository(dbContext);
        Items = new ItemRepository(dbContext);
    }

    9 references
    public async Task CompleteAsync()
    {
        await _dbContext.SaveChangesAsync();
    }

    3 references
    public async Task BeginTransaction()
    {
        await _dbContext.Database.BeginTransactionAsync();
    }

    3 references
    public async Task CommitTransaction()
    {
        if (_dbContext.Database.CurrentTransaction != null)
            await _dbContext.Database.CommitTransactionAsync();
    }

    3 references
    public async Task RollbackTransaction()
    {
        if (_dbContext.Database.CurrentTransaction != null)
            await _dbContext.Database.RollbackTransactionAsync();
    }
}
```

Figure 4 Unit Of Work Implementation

```
public class CreateOrderCommand : IRequest<OrderResponseModel?>
{
    3 references
    public OrderRequestModel Order { get; private set; }

    1 reference
    public CreateOrderCommand(OrderRequestModel order)
    {
        Order = order;
    }
}
```

Figure 5 Create Order Command

```
1 reference
public class CreateOrderHandler : IRequestHandler<CreateOrderCommand, OrderResponseModel?>
{
    private IOrderService _orderService;

    0 references
    public CreateOrderHandler(IOrderService orderService)
    {
        _orderService = orderService;
    }

    0 references
    public async Task<OrderResponseModel?> Handle(CreateOrderCommand request, CancellationToken cancellationToken)
    {
        Order? newOrder = new()
        {
            CustomerId = request.Order.CustomerId,
            Items = request.Order.Items.Select(i => new Item()
            {
                ProductId = i.ProductId,
                Quantity = i.Quantity,
            }).ToList()
        };

        newOrder = await _orderService.CreateOrder(newOrder);

        if (newOrder is null)
            return null;
```

Figure 6 Create Order Handler Part A

```
        OrderResponseModel orderResponse = new()
        {
            Id = newOrder.Id,
            Customer = new CustomerResponseModel()
            {
                Id = newOrder.Customer.Id,
                FirstName = newOrder.Customer.FirstName,
                LastName = newOrder.Customer.LastName,
                Address = newOrder.Customer.Address,
                PostalCode = newOrder.Customer.PostalCode
            },
            DocDate = newOrder.DocDate,
            DocTotal = newOrder.DocTotal,
            Items = newOrder.Items.Select(i => new ItemResponseModel()
            {
                Id = i.Id,
                Quantity = i.Quantity,
                Product = new ProductResponseModel()
                {
                    Id = i.Product.Id,
                    Name = i.Product.Name,
                    Price = i.Product.Price
                }
            }).ToList()
        };

        return orderResponse;
    }
}
```

Figure 7 Create Order Handler Part B

```
public class GetOrderByIdQuery : IRequest<OrderResponseModel?>
{
    2 references
    public Guid Id { get; private set; }

    1 reference
    public GetOrderByIdQuery(Guid id)
    {
        Id = id;
    }
}
```

Figure 8 Get Order By Id Query

```
2 references
public class OrderRepository : IOrderRepository
{
    private readonly ApplicationDbContext _dbContext;

    1 reference
    public OrderRepository(ApplicationDbContext dbContext)
    {
        _dbContext = dbContext;
    }

    4 references
    public virtual async Task<ICollection<Order>> GetAllAsync()
    {
        return await _dbContext.Orders.ToListAsync();
    }

    13 references
    public virtual async Task<Order?> GetByIdAsync(Guid id)
    {
        Order? order = await _dbContext.Orders.FindAsync(id);

        return order;
    }

    4 references
    public virtual async Task<Order> CreateAsync(Order order)
    {
        Order newOrder = (await _dbContext.Orders.AddAsync(order)).Entity;

        return newOrder;
    }
```

Figure 9 Order Repository Part A

```
public virtual async Task<Order?> UpdateAsync(Order order)
{
    return _dbContext.Orders.Update(order).Entity;
}

5 references
public virtual async Task DeleteAsync(Guid id)
{
    Order? order = await GetByIdAsync(id);

    if (order is null)
        throw new NullReferenceException();

    _dbContext.Orders.Remove(order);
}

3 references
public virtual async Task<ICollection<Order>> GetOrdersOfCustomer(Guid customerId)
{
    return await _dbContext.Orders.Where(o => o.CustomerId == customerId).OrderBy(o => o.DocDate).ToListAsync();
}
```

Figure 10 Order Repository Part B

```
public interface IUnitOfWork
{
    14 references
    ICustomerRepository Customers { get; }
    9 references
    IOrderRepository Orders { get; }
    9 references
    IProductRepository Products { get; }
    7 references
    IItemRepository Items { get; }

    9 references
    Task CompleteAsync();
    3 references
    Task BeginTransaction();
    3 references
    Task CommitTransaction();
    3 references
    Task RollbackTransaction();
}
```

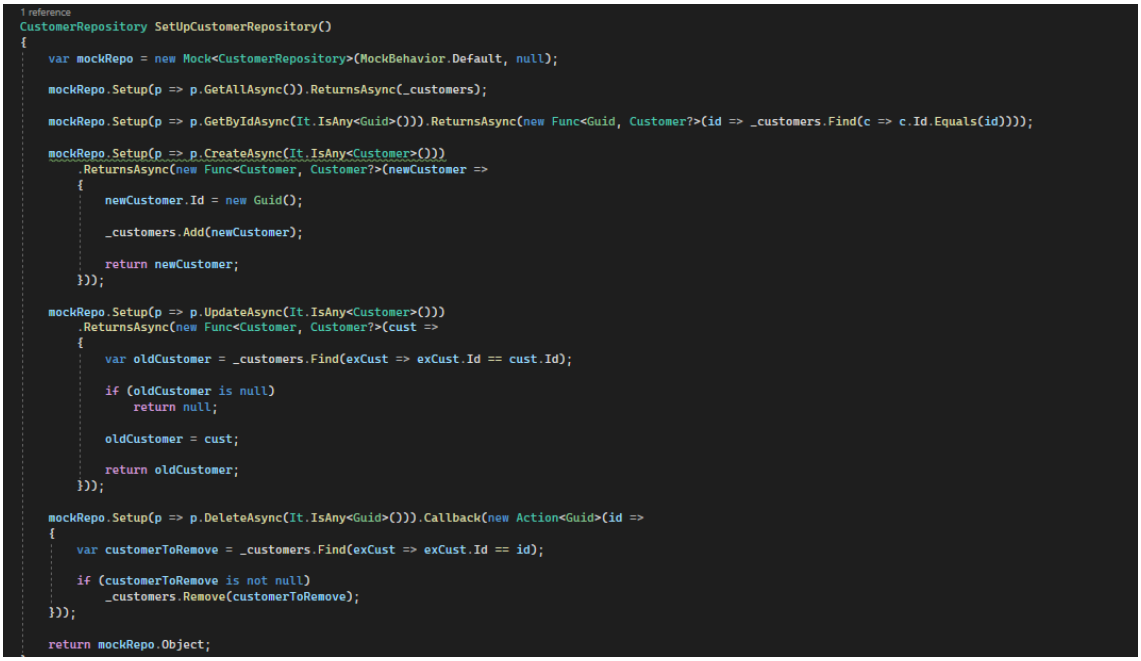
Figure 11 IUnitOfWork Interface

```
0 references
internal class CustomerServicesTest
{
    private ICustomerService _customerService;
    private IUnitOfWork _unitOfWork;
    private List<Customer> _customers;
    private CustomerRepository _customerRepository;

    [SetUp]
    0 references
    public void Setup()
    {
        _customers = SetUpCustomers();
        _customerRepository = SetUpCustomerRepository();
        var unitOfWork = new Mock<IUnitOfWork>();
        unitOfWork.SetupGet(s => s.Customers).Returns(_customerRepository);
        _unitOfWork = unitOfWork.Object;
        _customerService = new CustomerService(_unitOfWork);
    }

    [TearDown]
    0 references
    public void DisposeTest()
    {
        _customers = null;
        _customerService = null;
        _unitOfWork = null;
        _customerRepository = null;
    }
```

Figure 12 CustomerServiceTest Part A

```
[Test]
0 references
public void GetAllCustomersTest()
{
    var customers = _customerService.GetAllCustomers().Result;

    var comparer = new CustomerComparer();

    CollectionAssert.AreEqual(
            customers.OrderBy(customer => customer, comparer),
            _customers.OrderBy(customer => customer, comparer), comparer);
}

1 reference
List<Customer> SetUpCustomers()
{
    List<Customer> customers = DataInitializer.GetAllCustomers();

    foreach (var customer in customers)
    {
        customer.Id = new Guid();
    }

    return customers;
}
```

Figure 13 CustomerServiceTest Part B



```
1 reference
CustomerRepository SetUpCustomerRepository()
{
    var mockRepo = new Mock<CustomerRepository>(MockBehavior.Default, null);

    mockRepo.Setup(p => p.GetAllAsync()).ReturnsAsync(_customers);

    mockRepo.Setup(p => p.GetByIdAsync(It.IsAny<Guid>())).ReturnsAsync(new Func<Guid, Customer?>(id => _customers.Find(c => c.Id.Equals(id))));

    mockRepo.Setup(p => p.CreateAsync(It.IsAny<Customer>()))
        .ReturnsAsync(new Func<Customer, Customer?>(newCustomer =>
        {
            newCustomer.Id = new Guid();

            _customers.Add(newCustomer);

            return newCustomer;
        }));

    mockRepo.Setup(p => p.UpdateAsync(It.IsAny<Customer>()))
        .ReturnsAsync(new Func<Customer, Customer?>(cust =>
        {
            var oldCustomer = _customers.Find(exCust => exCust.Id == cust.Id);

            if (oldCustomer is null)
                return null;

            oldCustomer = cust;

            return oldCustomer;
        }));

    mockRepo.Setup(p => p.DeleteAsync(It.IsAny<Guid>())).Callback(new Action<Guid>(id =>
    {
        var customerToRemove = _customers.Find(exCust => exCust.Id == id);

        if (customerToRemove is not null)
            _customers.Remove(customerToRemove);
    }));

    return mockRepo.Object;
```

Figure 14 CustomerServiceTest Part C

Conclusion

In conclusion, this thesis explored the creation of a web API for performing CRUD operations on a shop's data. The API has been designed and implemented using various architectural patterns and best practices to ensure a robust and maintainable solution.

The utilization of Entity Framework has simplified the data access layer by providing an object-relational mapping framework. This has reduced the amount of code needed to interact with the underlying database, allowing for efficient and seamless data operations.

The adoption of the Repository Pattern with the Unit of Work pattern has facilitated the separation of concerns and improved the maintainability of the application. By encapsulating the data access logic within repositories and managing transactions through the unit of work, the application ensures transactional consistency and concurrency control.

Domain Driven Design principles have been applied to focus on the core domain logic of the shop. The emphasis on modeling the business domain accurately has led to a better alignment between the codebase and the actual problem domain, resulting in a more understandable and maintainable system.

The Mediator Pattern, implemented for Command Query Responsibility Segregation (CQRS), has enhanced the separation of commands and queries within the API. The mediator acts as a central component that routes messages between the command and query handlers, enabling loose coupling and scalability.

Unit tests, implemented using NUnit, have played a vital role in ensuring the correctness and reliability of the web API. By covering critical code paths and edge cases, unit tests have provided a safety net against regressions and facilitated code quality and maintainability.

Overall, the combination of Entity Framework, Repository Pattern with Unit of Work, Domain-Driven Design, Mediator Pattern for CQRS, and comprehensive unit testing has resulted in the successful creation of a robust and scalable web API for CRUD operations on a shop's data. The architectural choices and patterns applied have promoted modularity, maintainability, and extensibility, aligning with industry best practices.

This thesis project not only provides a functional web API but also serves as a reference implementation for future development projects. It showcases the importance of architectural patterns methodologies in creating reliable and high-quality software solutions.

As technology continues to evolve and business requirements change, the architectural principles and patterns employed in this project will serve as a solid foundation for future enhancements and iterations. By following these best practices, developers can build upon this web API, adapt it to specific business needs, and continue delivering value to end-users.

In conclusion, the successful creation of this web API demonstrates the effectiveness of employing Entity Framework, Repository Pattern with Unit of Work, Domain-Driven Design, Mediator Pattern for CQRS, and comprehensive unit testing to create a reliable, maintainable, and scalable solution for managing shop data through CRUD operations.

Thanks

I want to express my deepest gratitude for the people and things that have helped me in my academic journey. From the bottom of my heart, I want to thank all those who have been a part of this incredible journey with me.

First and foremost, I would like to thank my family – starting with my father, my mother and my brother. They have been a constant source of encouragement and support throughout this process. From helping me with assignments to providing emotional support, they have been there for me every step of the way. Their unwavering commitment has enabled me to reach new heights academically, something which I am extremely thankful for!

I would also like to thank Charalampos Sideris and Nicoleta Samara, two amazing classmates who provided both emotional and practical support when needed. Whether it was helping with an assignment or just being there as a listening ear during tough times – their friendship has made this journey much more enjoyable!

Last but not least, I would like to thank Professor Mr. Alepis for believing in my potentials from day one and pushing me out of my comfort zone so that I could reach new heights academically. His guidance has enabled me to be successful in this field, something which I will be forever grateful for!
So once again, thank you all for being part of this journey – it has been an amazing experience!

References

1. Programming Entity Framework, 2nd Edition (2010): By Julia Lerman
2. Pro ADO NET Entity Framework 4 0 (2011): By Roger Jennings
3. Beginning ASP NET 4 5 in C# and VB (2012): By Imar Spaanjaars
4. Programming Entity Framework, 3rd Edition (2013): By Julie Lerman
5. Design Patterns in C#: A Hands-On Guide, by Steve Metsker.
6. Head First Design Patterns: A Brain-Friendly Guide, by Eric Freeman and Elisabeth Robson.
7. Pro .NET Design Pattern Framework 4.5, by Steven John Metsker and Matthew B Jones.
8. C# 6 for Programmers (6th Edition), by Paul Deitel and Harvey Deitel.
9. Professional C# 7 and .NET Core 2, by Christian Nagel et al..
10. C# 7 and .NET Core 2.0 – Modern Cross-Platform Development, Third Edition, by Mark J. Price.
11. Pro C# 7: With .NET and .NET Core, Eighth Edition, by Andrew Troelsen
12. CLR via C# (Developer Reference), Fifth Edition, by Jeffrey Richter
13. Programming in C# Exam Ref 70-483: Second Edition, by Wouter de Kort
14. Effective C#: 50 Specific Ways to Improve Your C# (Effective Software. Development Series), Sixth Edition, by Bill Wagner
15. Murach's Beginning Visual C# 2019, Fifth Edition ,by Anne Boehm & Joel Murach
16. Pro ASP.NET Core MVC 2, Adam Freeman (Apress)
17. Professional ASP.NET MVC 5, Jon Galloway et al (Wrox)
18. Programming Microsoft ASP.NET MVC, Dino Esposito (Microsoft Press)
19. Beginning ASP NET 4 in C# and VB, Imar Spaanjaars (Wrox)
20. Pro C# 7: With .NET and .NET Core, Andrew Troelsen & Philip Japikse (Apress).