Session 7: OLTP Multiple Records

Adding & Edit Invoice
& Invoice Line

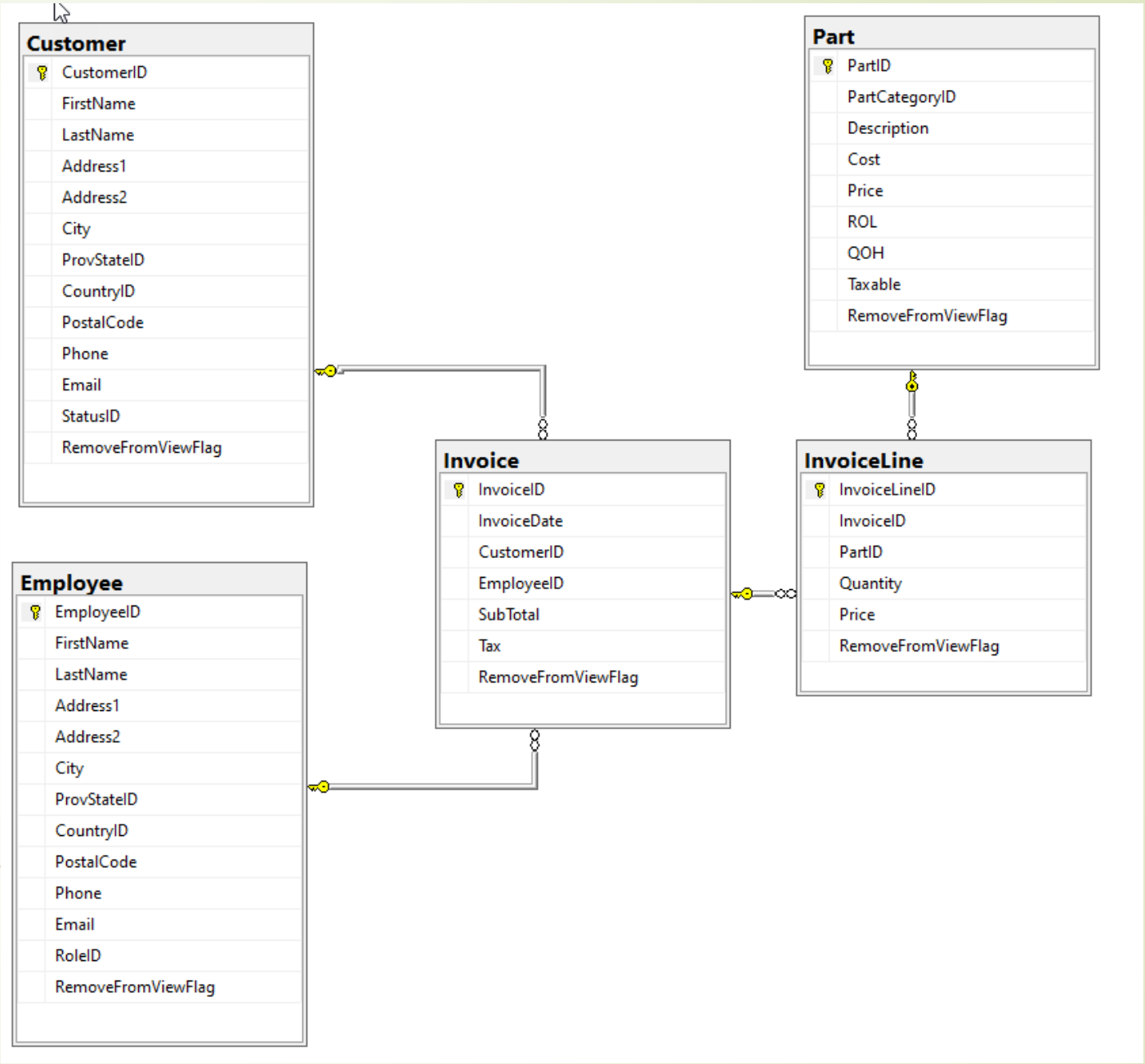
OLTP with Parent & Children Items

Granularity of Data:	• OLTP handles individual invoices and their specific lines, supporting operations like adding, updating, or deleting items.
High Volume of Simple Transactions:	• OLTP systems manage many straightforward transactions daily, such as creating an invoice or adding an invoice line.
Data Integrity and Availability:	• OLTP prioritizes consistent, accurate data and quick response times for high transaction rates.
Normalized Database Structures:	• OLTP databases use normalized structures to prevent redundancy, which may require joining tables for complex queries.

High Level Overview.

Creating an Invoice with Invoice Lines:	Initialize a new invoice with relevant details (e.g., date, recipient).	Add initial invoice lines detailing items/services, quantities, and prices.	Calculate and record the subtotal, taxes, and total amount.	Save or issue the invoice.
Adding More Invoice Lines to an Existing Invoice:	Retrieve the desired invoice by its invoice #.	Append new invoice lines with additional items/services.	Update and recalculate the subtotal, taxes, and total amount.	Save or reissue the updated invoice.

Database Schema



Invoice Class

The following fields are used as placeholders for storing data about our invoice. The values are not stored in the database but are updated when we retrieve the data.

- Customer Name
- Employee Name

The sub-total and tax properties are updated from the total calculated from the "Invoice Lines."

```
public class InvoiceView
{
    public int InvoiceID { get; set; }
    public DateTime InvoiceDate { get; set; }
    public int CustomerID { get; set; }
    public string CustomerName { get; set; }
    public int EmployeeID { get; set; }
    public string EmployeeName { get; set; }
    public decimal SubTotal { get; set; }
    public decimal Tax { get; set; }
    public List<InvoiceLineView> InvoiceLines { get; set; } = new List<InvoiceLineView>();
    public bool RemoveFromViewFlag { get; set; }
}
```

Invoice Class

The inclusion of 'InvoiceLines' within the 'InvoiceView' class allows for a natural representation of an invoice. Having the invoice lines directly within the invoice object streamlines data retrieval, ensuring that when an invoice is fetched or displayed, its associated line items are immediately accessible without additional database queries. This is especially useful in views or user interfaces where a complete breakdown of an invoice, including its items or services, is required.

```
public class InvoiceView
{
    public int InvoiceID { get; set; }
    public DateTime InvoiceDate { get; set; }
    public int CustomerID { get; set; }
    public string CustomerName { get; set; }
    public int EmployeeID { get; set; }
    public string EmployeeName { get; set; }
    public decimal SubTotal { get; set; }
    public decimal Tax { get; set; }
    public List<InvoiceLineView> InvoiceLines { get; set; } = new List<InvoiceLineView>();
    public bool RemoveFromViewFlag { get; set; }
}
```

Invoice Lines Class

The **InvoiceLineView** class represents an individual line item on an invoice, providing a clear and organized structure for managing the essential details associated with each product or service on an invoice.

```
public class InvoiceLineView
{
    public int InvoiceLineID { get; set; }
    public int InvoiceID { get; set; }
    public int PartID { get; set; }
    public string Description { get; set; }
    public int Quantity { get; set; }
    public decimal Price { get; set; }
    public bool RemoveFromViewFlag { get; set; }
}
```

Add & Edit Invoice Method Overview

- A method to add a new or update an existing invoice.
- Validates invoice data according to business rules.
- Handles potential errors and exceptions.

Method Parameter and Initial Setup

The method accepts an **InvoiceView** parameter.

An error list is initialized to track any business rule violations.

```
public InvoiceView AddEditInvoice(InvoiceView invoiceView)
{

    #region Business Logic and Parameter Exceptions
    //  create a list<Exception> to contain all discovered errors
    List<Exception> errorList = new List<Exception>();
    #endregion
}
```

Basic Business Rule Validations

Checking	Checking if the provided invoice is null.
Ensuring	Ensuring valid Customer and Employee IDs.
Ensuring	Ensuring there are line items for the invoice.

csharp Copy code

```csharp
if (invoiceView == null) { ... }
if (invoiceView.CustomerID == 0) { ... }
if (invoiceView.EmployeeID == 0) { ... }
if (invoiceView.InvoiceLines.Count == 0) { ... }
```

Invoice Line-Item Validations

Each line item should have an associated part.

The price for each line item should be valid.

csharp Copy code

```csharp
foreach (var invoiceLine in invoiceView.InvoiceLines)
{
    if (invoiceLine.PartID == 0) { ... }
    if (invoiceLine.Price < 0) { ... }
}
```

Checking for Duplicated Parts

Parts should not be duplicated on the same invoice.

Code identifies and adds duplicated part errors to the error list.

csharp

Copy code

```csharp
List<string> duplicatedParts = invoiceView.InvoiceLines...
if (duplicatedParts.Count > 0) { ... }
```

Fetching or Initializing the Invoice

```
Invoice invoice = Invoices
                    .Where(x => x.InvoiceID == invoiceView.InvoiceID)
                    .Select(x => x)
                    .FirstOrDefault();

//  check if invoice exists
if (invoice == null)
{
    invoice = new Invoice();
    invoice.InvoiceDate = DateTime.Now;
}
else
{
    invoice.InvoiceDate = invoiceView.InvoiceDate;
}
invoice.CustomerID = invoiceView.CustomerID;
invoice.EmployeeID = invoiceView.EmployeeID;
invoice.SubTotal = 0;
invoice.Tax = 0;
```

Fetching or Initializing the Invoice

Objective:

The primary goal in this segment is to determine if we're dealing with an existing invoice or creating a new one.

Invoice Retrieval:

The Invoices collection is queried using LINQ to search for an invoice that matches the InvoiceID provided in the invoiceView.

If the invoice exists, it's fetched for further processing; otherwise, it's assumed we need to create a new invoice.

Handling New Invoices:

If the invoice is null (i.e., not found in our data source), we initialize a new Invoice object.

The invoice date is set to the current date and time using DateTime.Now. This ensures the new invoice has a timestamp indicating when it was created.

Handling Existing Invoices:

If the invoice already exists (i.e., it's not null), we're in "edit" mode.

We update the InvoiceDate with the date provided in the invoiceView. This allows changes to the invoice date to be persisted.

Flexibility and Efficiency:

This approach provides flexibility for handling both new and existing invoices within a single method.

It efficiently uses the provided InvoiceID to decide the course of action and minimizes unnecessary data operations.

Processing Each Line Item

```
// Process each line item in the provided view model.
foreach (var invoiceLineView in invoiceView.InvoiceLines)
{
    InvoiceLine invoiceLine = InvoiceLines
                                .Where(x => x.InvoiceLineID == invoiceLineView.InvoiceLineID
                                        && x.PartID == invoiceLineView.PartID)
                                .FirstOrDefault();

    // If the line item doesn't exist, initialize it.
    if (invoiceLine == null)
    {
        invoiceLine = new InvoiceLine();
        invoiceLine.PartID = invoiceLineView.PartID;
    }

    // Map fields from the line item view model to the data model.
    invoiceLine.Quantity = invoiceLineView.Quantity;
    invoiceLine.Price = invoiceLineView.Price;
    invoiceLine.RemoveFromViewFlag = invoiceLineView.RemoveFromViewFlag;

    // Handle new or existing line items.
    if (invoiceLine.InvoiceLineID == 0)
    {
        invoice.InvoiceLines.Add(invoiceLine); // Add new line items.
    }
    else
    {
        InvoiceLines.Update(invoiceLine); // Update existing line items.
    }
```

Processing Each Line Item

```
    //  need to update total and tax if the
    //      invoice line item is not set to be removed from view.
    if (!invoiceLine.RemoveFromViewFlag)
    {
        invoice.SubTotal += invoiceLine.Quantity * invoiceLine.Price;
        bool isTaxable = Parts
                            .Where(x => x.PartID == invoiceLine.PartID)
                            .Select(x => x.Taxable)
                            .FirstOrDefault();
        invoice.Tax += isTaxable ? invoiceLine.Quantity * invoiceLine.Price * .05m : 0;
    }
}
```

Processing Each Line Item

1. **Objective**:
   - This segment aims to process each line item provided in the **invoiceView**. It's about mapping data from the view model to the data model and making necessary calculations.
2. **Line Item Retrieval**:
   - The **InvoiceLines** collection is queried using LINQ. We're searching for an invoice line that matches both the **InvoiceLineID** and **PartID** from the **invoiceLineView**.
   - The rationale behind checking both IDs is to ensure we retrieve the correct line item for updating or identify when a new line needs to be created.

Processing Each Line Item - Continue

3. **Handling New Invoice Lines**:
   - If the retrieved **invoiceLine** is **null**, it means we have a new line item.
   - We then initialize a new **InvoiceLine** object and assign the relevant **PartID** from the **invoiceLineView**.
4. **Mapping from ViewModel to DataModel**:
   - After identifying if we're working with a new or existing line item, fields from the **invoiceLineView** are mapped to the **invoiceLine** (like quantity, price, and other attributes).
   - This ensures that any changes or new data provided in the view model are reflected in the data model.

Processing Each Line Item - Continue

5. **Efficiency and Maintainability**:
   - Using a loop for this process ensures that all line items, regardless of their number, are processed efficiently.
   - This modular approach to handling line items also makes the code more maintainable and easier to debug, as each line item undergoes the same set of operations.

Update New Invoice and Saving

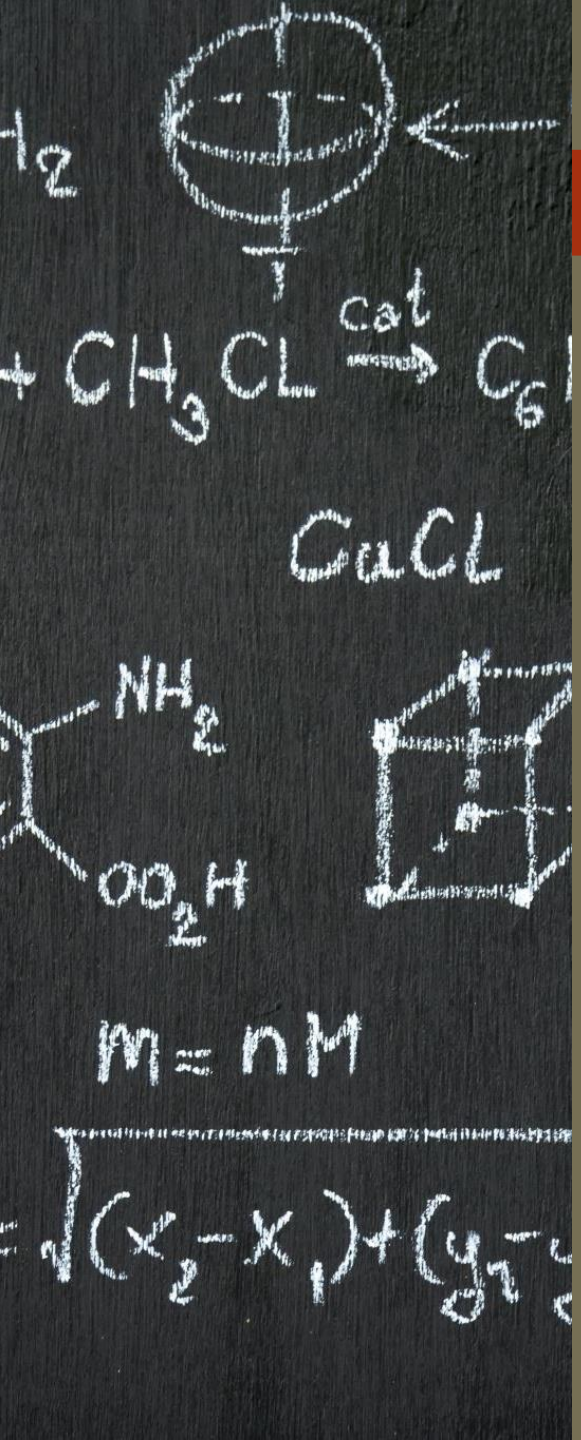
```
if (invoice.InvoiceID == 0)
{
    Invoices.Add(invoice);
}

// Handle any captured errors.
if (errorList.Count > 0)
{
    // Clear changes to maintain data integrity.
    ChangeTracker.Clear();
    string errorMsg = "Unable to add or edit Invoice or Invoice Lines.";
    errorMsg += " Please check error message(s)";
    throw new AggregateException(errorMsg, errorList);
}
else
{
```

Update New Invoice and Saving

- The code first checks if the provided invoice is new by examining the **InvoiceID**.
  - If the **InvoiceID** is 0, indicating it's a new invoice, it adds the invoice to the **Invoices** collection.
- The code then assesses the presence of any errors captured in the **errorList**.
  - If there are errors (as indicated by the count of the **errorList** being greater than 0):
    - It clears any pending changes to prevent data inconsistencies using the **ChangeTracker.Clear()** method.
    - It throws an **AggregateException** with a descriptive message and the list of captured errors.
- If no errors are detected:
  - The changes are persisted to the database with the **SaveChanges()** method.

Driver (Main)

Non-Unit Tests



Data Sources

```
Customers
    .OrderBy(x => x.CustomerID)
    .Take(2).Dump("Customer");

Employees
    .OrderBy(x => x.EmployeeID)
    .Take(2).Dump("Employee");

Parts
    .OrderBy(x => x.PartID)
    .Take(5).Dump("Part");
```

Data Sources

Customer

EntityQueryable<Customer> (2 items)

CustomerID	FirstName	LastName	Address1	Address2	City	ProvStateID	CountryID	PostalCode	Phone	Email	StatusID	RemoveFromViewFlag	Country	ProvState	Status	Invoices
1	Sam	Smith	12345 - 67 St		Edmonton	9	11	T5J1X1	7804444444	ssmith@hotmail.com	14	False	Country	ProvState	Status	Invoices
2	John	Jones	23456 - 78 St		Edmonton	9	11	T5J1X2	7804322222	jjones@hotmail.com	14	False	Country	ProvState	Status	Invoices

Employee

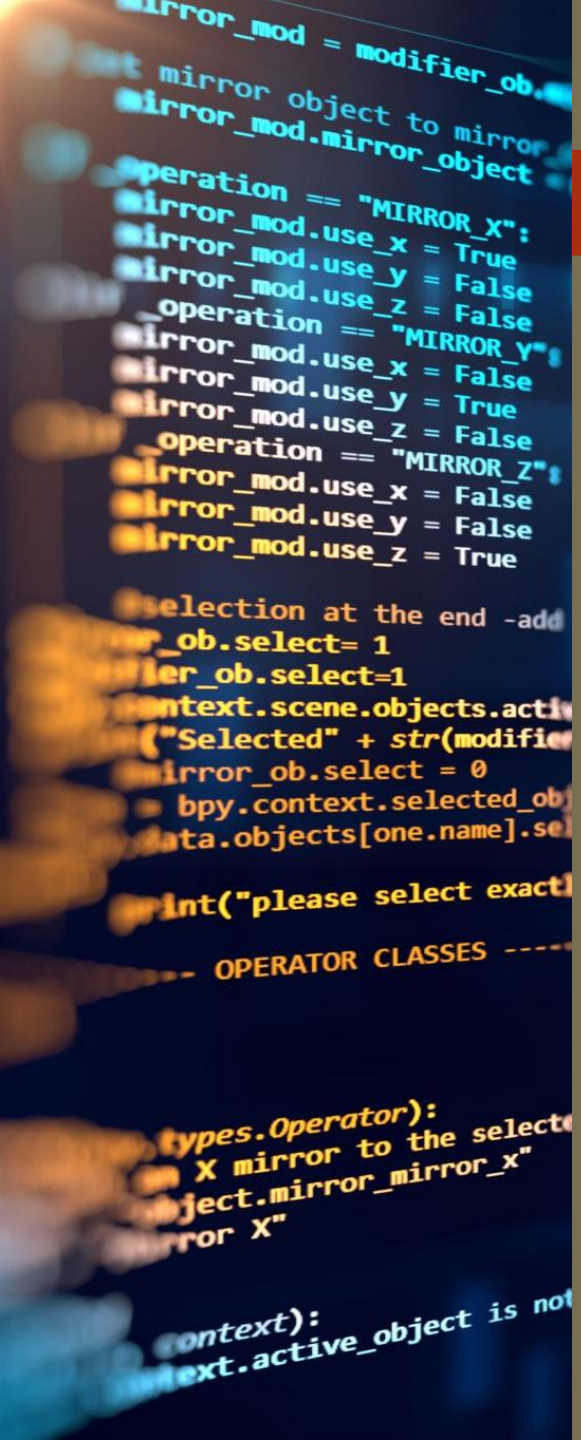
EntityQueryable<Employee> (2 items)

EmployeeID	FirstName	LastName	Address1	Address2	City	ProvStateID	CountryID	PostalCode	Phone	Email	RoleID	RemoveFromViewFlag	Country	ProvState	Role	Invoices
1	Nole	Body	3215 - 66 St		Edmonton	9	11	T5J1X1	780.432.9876	nbody@eBikes.com	18	False	Country	ProvState	Role	Invoices
2	Willie	Work	12345 - 67 St		Edmonton	9	11	T5J1X1	780.444.3535	wwork@eBikes.com	18	False	Country	ProvState	Role	Invoices

Part

EntityQueryable<Part> (5 items)

PartID	PartCategoryID	Description	Cost	Price	ROL	QOH	Taxable	RemoveFromViewFlag	PartCategory	InvoiceLines
1	23	Forged pistons	25.00	50.00	7	6	**True**	False	PartCategory	InvoiceLines
2	23	O-ring gaskets	30.00	50.00	7	5	**True**	False	PartCategory	InvoiceLines
3	23	Exhaust system	250.00	400.00	6	24	**True**	False	PartCategory	InvoiceLines
4	23	Rear brakes	40.00	60.00	10	5	**True**	False	PartCategory	InvoiceLines
5	23	Front brakes	40.00	60.00	10	5	**True**	False	PartCategory	InvoiceLines
			385.00	620.00	40	45				



Using Main() Method

- Coding in the Main() methods is very much like coding in your Web Page. This might be things such as saving data, retrieving invoice and their invoice lines, etc.
- Like doing Unit Tests, we might ask you just to code in the Main() method to show that your method works.

Coding Add Invoice in Main()

1. Create two areas (Before and After Action)
2. Create your "Before Action"
   a) Create your Invoice
   b) Add Invoice Lines
3. Output the values for review.
4. Execute your methods.
5. Output the "After Action" values for review.

```
void Main()
{
    //  coding for the AddEditInvoice method
    //  setup Add Invoice
    //  before action (Add)                                   2
    InvoiceView beforeAdd = new InvoiceView();
    //  do not set the InvoiceID

    //  Sam Smith (Customer)
    beforeAdd.CustomerID = 1;                                 2a
    // Willie Work (Employee)
    beforeAdd.EmployeeID = 2;

    //  add invoice items
    InvoiceLineView invoiceLine = new InvoiceLineView();
    invoiceLine.PartID = 1;
    invoiceLine.Description = "Forged pistons";
    invoiceLine.Quantity = 10;
    invoiceLine.Price = 50.00m;
    beforeAdd.InvoiceLines.Add(invoiceLine);

    invoiceLine = new InvoiceLineView();
    invoiceLine.PartID = 4;                                   2b
    invoiceLine.Description = "Rear brakes";
    invoiceLine.Quantity = 20;
    invoiceLine.Price = 60.00m;
    beforeAdd.InvoiceLines.Add(invoiceLine);

    //  showing results
    beforeAdd.Dump("Before Add");                             3

    //  execute                                               4
    InvoiceView afterAdd = AddEditInvoice(beforeAdd);

    //  after action (Add)
    //  showing results                                       5
    afterAdd.Dump("After Add");

}
```

Results

Before Add

InvoiceView

UserQuery+InvoiceView

InvoiceID	0
InvoiceDate	0001-01-01 12:00:00 AM
CustomerID	1
CustomerName	null
EmployeeID	2
EmployeeName	null
SubTotal	0
Tax	0
InvoiceLines	List<InvoiceLineView> (2 items)
RemoveFromViewFlag	False

InvoiceLineID	InvoiceID	PartID	Description	Quantity	Price	RemoveFromViewFlag
0	0	1	Forged pistons	10	50.00	False
0	0	4	Rear brakes	20	60.00	False
				30	110.00	

After Add

InvoiceView

UserQuery+InvoiceView

InvoiceID	161
InvoiceDate	2023-10-10 12:00:00 AM
CustomerID	1
CustomerName	Sam Smith
EmployeeID	2
EmployeeName	Willie Work
SubTotal	1700.00
Tax	85.00
InvoiceLines	List<InvoiceLineView> (2 items)
RemoveFromViewFlag	False

InvoiceLineID	InvoiceID	PartID	Description	Quantity	Price	RemoveFromViewFlag
942	161	1	Forged pistons	10	50.00	False
943	161	4	Rear brakes	20	60.00	False
				30	110.00	

Coding Edit Invoice in Main()

1. Create two areas (Before and After Action)
2. Create your "Before Action"
   a) Retrieve your Invoice
      a) Update the employee from "Willie Work" to "Nole Body."
      b) Update the quantity on the first invoice line (10 to 1)
      c) Remove line item "Rear Brake (Soft Delete).
      d) Add "Exhaust system"
3. Output the values for review.
4. Execute your methods.
5. Output the "After Action" values for review.

```
//  setup Edit Category
//  before action (Edit)
int invoiceID = Invoices
                .OrderByDescending(x => x.InvoiceID)
                .Select(x => x.InvoiceID).FirstOrDefault();
InvoiceView beforeEdit = GetInvoice(invoiceID);

//  showing results
beforeEdit.Dump("Before Edit");

//  change Employee
beforeEdit.EmployeeID = 1;

//  update the first invoice line quantity to 1
beforeEdit.InvoiceLines[0].Quantity = 1;

//  soft delete second line
beforeEdit.InvoiceLines[1].RemoveFromViewFlag = true; ;

//  add one more item
invoiceLine = new InvoiceLineView();
invoiceLine.PartID = 3;
invoiceLine.Description = "Exhaust system";
invoiceLine.Quantity = 5;
invoiceLine.Price = 400.00m;
beforeEdit.InvoiceLines.Add(invoiceLine);

//  execute
InvoiceView afterEdit = AddEditInvoice(beforeEdit);

//  after action (Edit)
//  showing results
afterEdit.Dump("After Edit");
```

Results

Before Edit

InvoiceView

UserQuery+InvoiceView	
InvoiceID	161
InvoiceDate	2023-10-10 12:00:00 AM
CustomerID	1
CustomerName	Sam Smith
EmployeeID	2
EmployeeName	Willie Work
SubTotal	1700.00
Tax	85.00
InvoiceLines	List<InvoiceLineView> (2 items)
RemoveFromViewFlag	False

InvoiceLineID	InvoiceID	PartID	Description	Quantity	Price	RemoveFromViewFlag
942	161	1	Forged pistons	10	50.00	False
943	161	4	Rear brakes	20	60.00	False
				30	110.00	

After Edit

InvoiceView

UserQuery+InvoiceView	
InvoiceID	161
InvoiceDate	2023-10-10 12:00:00 AM
CustomerID	1
CustomerName	Sam Smith
EmployeeID	1
EmployeeName	Nole Body
SubTotal	2050.00
Tax	102.50
InvoiceLines	List<InvoiceLineView> (2 items)
RemoveFromViewFlag	False

InvoiceLineID	InvoiceID	PartID	Description	Quantity	Price	RemoveFromViewFlag
942	161	1	Forged pistons	1	50.00	False
944	161	3	Exhaust system	5	400.00	False
				6	450.00	