Entity Modeling for April3_2025 WEBAPI

May4, 2024 – C. Vidotto

Contents

SQL Server Database Logical Model (to inform creation of Entity Framework Core ENTITIES): 3

- REFERENCE_RESOURCE 3
- CD_RESOURCE_TYPE 3
- CD_SUBJECT 3
- AUTHOR 4
- BOOK 4
- BOOK_AUTHOR 4
- Database Diagram Containing the above 6 Tables: 5
- Sample SQL Test queries: 6
- RESOURCE_CHATGPT 9
- RESOURCE_GOOGLE 9
- RESOURCE_YouTube_Video 9
- RESOURCE_Magazine_Article 9
- RESOURCE_Meeting 9
- RESOURCE_Conference 9
- RESOURCE_Tutoring 9
- RESOURCE_Class 9
- RESOURCE_Informal_Discussion 9
- PERCENTAGE_BOOK_COMPLETION 10

APPENDIX A: 11

- CREATE SCRIPTS: Test DB Relationships Created Correctly 11
  - Create CD_RESOURCE_TYPE 11
  - Create CD_RESOURCE_TOPIC 11
  - Create RESOURCE 11
  - Create PERCENTAGE_BOOK_COMPLETION 12
  - Create RESOURCE_BOOK 12

Create RESOURCE_BOOK Entity that will be used to create RESOURCE_BOOK table in SQL Server db 13

- See document: Aug 6 - 2025 Book and Resource_Reference Entity Relationship Definitions.docx 13
- At this path: `E:\Documents\____AAADeveloper_new\__DEVELOPER REFERENCES DOCUMENTS` 13

Creating The Initial Migration 14

- Using Tools | NuGet Package Manager | Package Manager Console: 14
  - PM> Add-Migration InitialMigration 14

PM> Update-Database 15
. 18
APPENDIX B: 19
TEST QUERIES 19
Get Resource Topic and Resource Type 19
Get Resource Topic and Resource Type by Resource Type 19
Get Book Data That Includes a Calculation of Percentage of Each Book Read. 19
Entity Framework Core Datamodelling 20
Entities/Vehicle.cs (Parent Class) 26
Entities/Car.cs (Child Class) 26
3. Create the ApplicationDbContext 26
4. Configure the DbContext in Startup.cs or Program.cs 28
Example Code for ApplicationDbContext.cs in /Data Folder: 32
REVISED ENTITY FRAMEWORK CORE CAR (standing in for ResourceBook) TABLE GENERATION 37
Addendum Info for DateOnly and Double fields in the Child Table (in my application: ResourceBook) 43
APPENDIX C: MISCELLANEOUS SQL CODE FOR FIXES/ADJUSTMENTS 53
ADD COLUMN statement to just add a column (that is nullable) 53
ALTER TABLE statement to change a column’s nullability 53
ALTER TABLE: Change a column’s datatype to decimal with 4 numbers before the decimal place and 2 after 53

SQL Server Database Logical Model (to inform creation of Entity Framework Core ENTITIES):

REFERENCE_RESOURCE

ResourceID	Title	SubjectID	ResourceTypeID
1	Clean Architecture	1	1
2	Test Driven Development By Example	2	1
3	Clean Code	9	1
4	Building Real-World Web Applications with Vue.js 3	11	1
5	Building Microservices	10	1
6	Azure DevOps Explained	6	1
7	Refactoring	13	1
8	Patterns of Enterprise Application Architecture	1	1
9	Continuous Delivery	8	1
11	Effective TypeScript	14	1
12	Concurrency in C# Cookbook	16	1
13	Professional Git	15	1
14	Visual Studio Code	18	1
15	The Road to React	12	1
16	Microservices Architecture Development with C# 9 and .NET 5	10	1
17	DevOps Design Patterns	6	1
18	Head First Object-Oriented Analysis & Design	26	1
19	Head First Design Patterns	27	1
20	Monolith to Microservices	10	1
25	Docker Deep Dive	28	1
27	Python Essentials	25	1

CD_RESOURCE_TYPE

ResourceTypeID	ResourceTypeID	Name
1		Book
2		ChatGPT
3		Google Search
4		YouTube Video
5		Magazine Article
6		Meeting
7		Conference
8		Tutoring
9		Class
10		Informal Discussion

CD_SUBJECT

SubjectID	Name
1	Software Architecture
2	Test Driven Development
3	Functional Programming
4	C#
5	DevOps
6	Azure DevOps
7	Continuous Integration
8	Continuous Delivery
9	Coding Best Practices
10	Microservices
11	Vue.js
12	React
13	Refactoring
14	TypeScript
15	Git
16	Concurrency
17	Visual Studio
18	Visual Studio Code
19	Entity Framework Core
20	SQL Server
21	Oracle
22	Java
23	Continuous Deployment
24	Containerization
25	Python
26	OOP
27	Design Patterns
28	Docker

Entity Modeling for April3_2025 WEBAPI

May4, 2024 – C. Vidotto

AUTHOR

AuthorID	FullName	Bio
1	Robert C. Martin	
2	Ken Beck	
3	Joran Qjuinten	
4	Sam Newman	
5	Sjoukje Zall, ,	
6	Stefano Demiliani	
7	Amit Malik	
8	Martin Fowler	
9	Jez Humble	
10	David Farley	
11	Nigel Poulton	
12	Dan Vanderkam	
13	Stephen Cleary	
14	Brent Laster	
15	Bruce Johnson	
16	Robin Wieruch	
17	Andrew M. Jones	
18	Pradeep Chintale	
19	Brett D. McLaughlin	
20	Gary Pollice	
21	David West	
22	Eric Freeman	
23	Elisabeth Freeman	
24	Kathy Sierra	
25	Bert Bates	
26	John C. Shovic	NULL
27	Alan Simpson	NULL

BOOK

BookID	Pages	ISBN	PAGES_READ
1	NULL		NULL
2	216		19
3	NULL		NULL
4	284		31
5	573		71
6	NULL		NULL
7	NULL		NULL
8	NULL		NULL
9	NULL		NULL
11	NULL		NULL
12	NULL		NULL
13	NULL		NULL
14	NULL		NULL
15	280		49
16	NULL		NULL
17	NULL		NULL
18	NULL		NULL
19	NULL		NULL
20	NULL		NULL
25	NULL	NULL	NULL
27	NULL	NULL	NULL

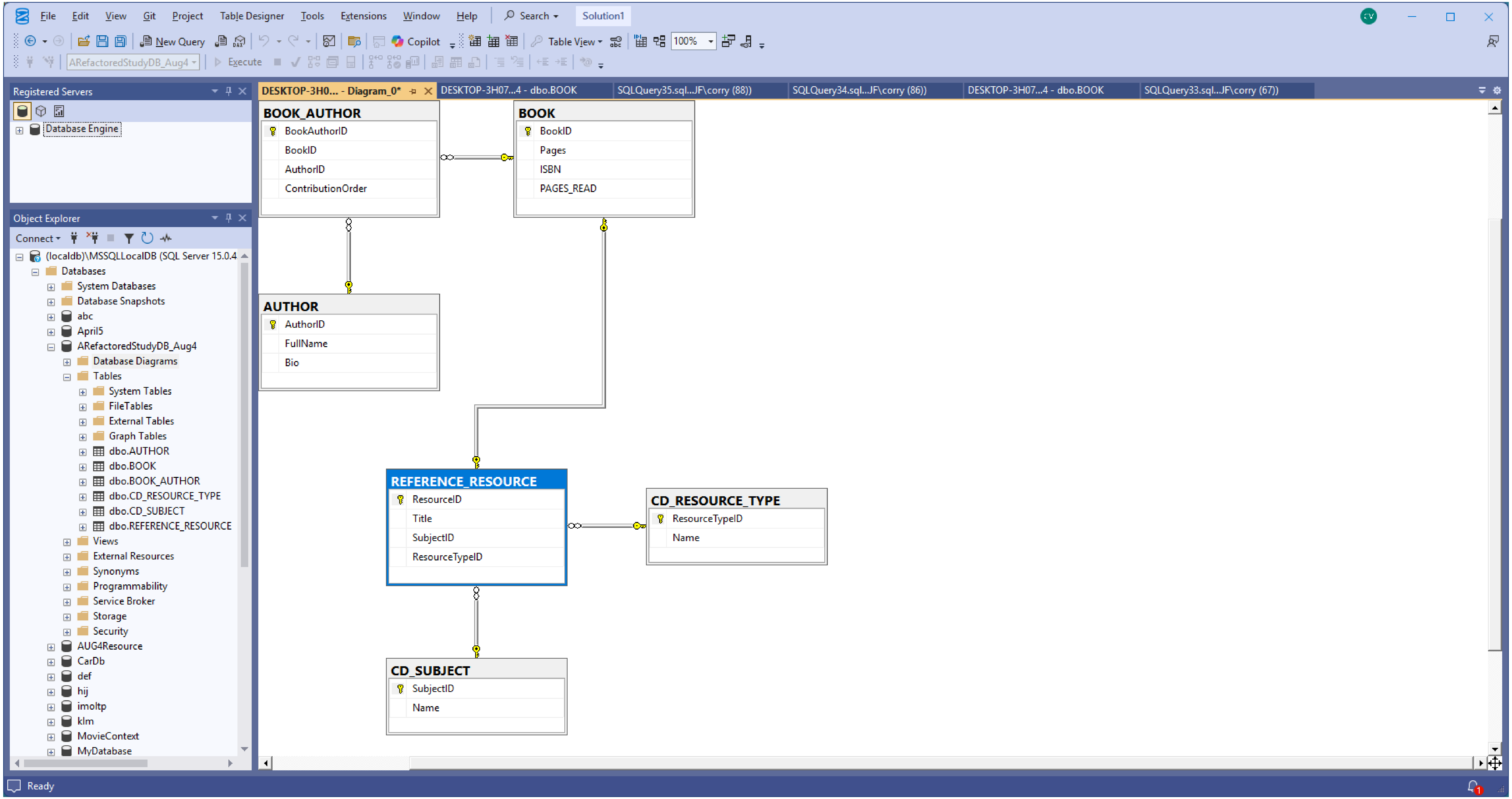
BOOK_AUTHOR

BookAuthorID	BookID	AuthorID	ContributionOrder
1	1	1	NULL
2	2	2	NULL
3	3	1	NULL
4	4	3	NULL
5	5	4	NULL
6	6	5	NULL
7	6	6	NULL
8	6	7	NULL
9	7	8	NULL
10	8	8	NULL
11	9	9	NULL
12	9	10	NULL
14	11	12	NULL
15	12	13	NULL
16	13	14	NULL
17	14	15	NULL
18	15	16	NULL
19	16	17	NULL
20	17	18	NULL
21	18	19	NULL
22	18	20	NULL
23	18	21	NULL
24	19	22	NULL
25	19	23	NULL
26	19	24	NULL
27	19	25	NULL
28	20	4	NULL
29	25	11	NULL
34	27	26	NULL
35	27	27	NULL

Entity Modeling for April3_2025 WEBAPI

May4, 2024 – C. Vidotto

Database Diagram Containing the above 6 Tables:



BOOK_AUTHOR
- BookAuthorID
- BookID
- AuthorID
- ContributionOrder

BOOK
- BookID
- Pages
- ISBN
- PAGES_READ

AUTHOR
- AuthorID
- FullName
- Bio

REFERENCE_RESOURCE
- ResourceID
- Title
- SubjectID
- ResourceTypeID

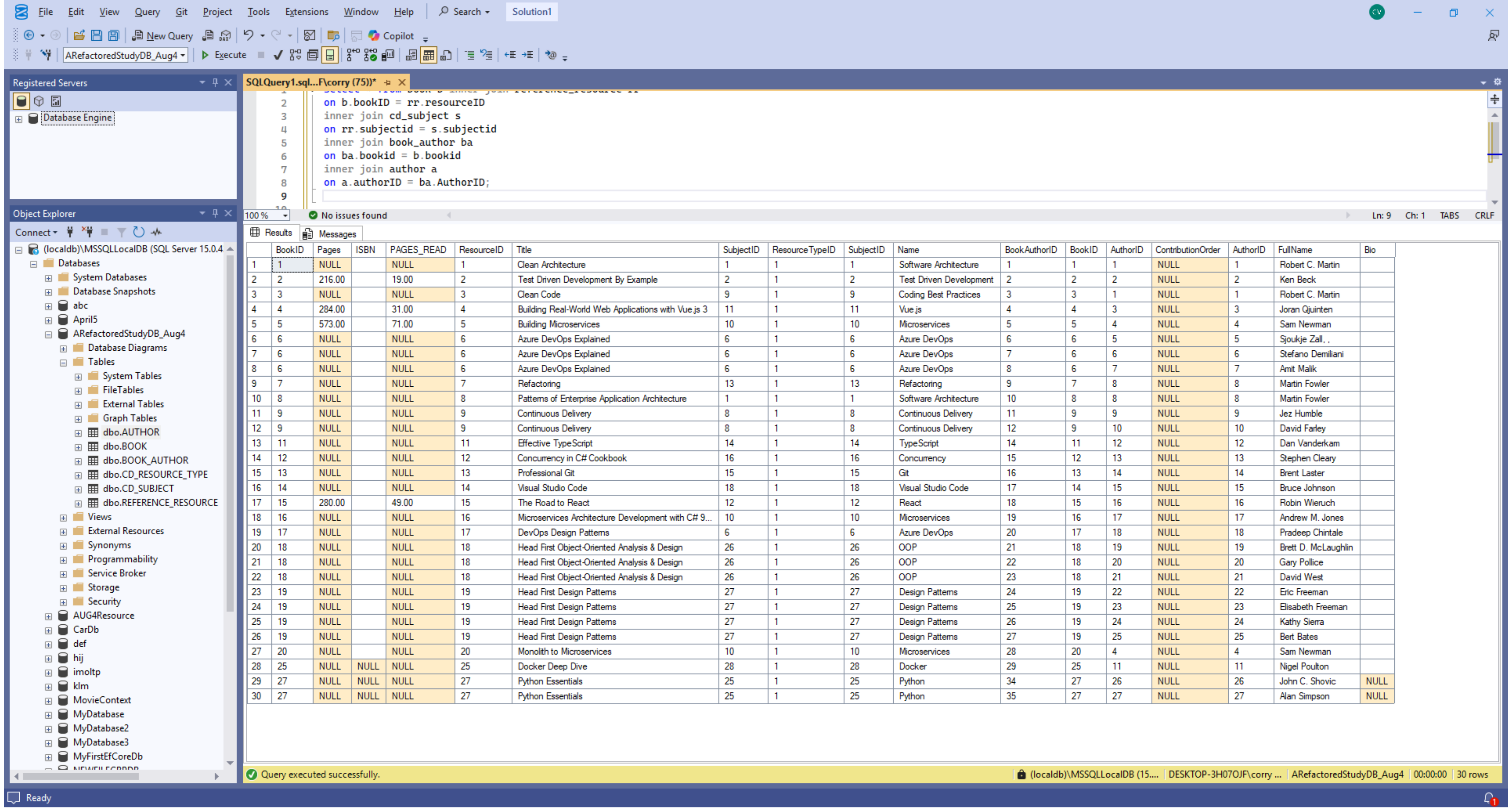
CD_RESOURCE_TYPE
- ResourceTypeID
- Name

CD_SUBJECT
- SubjectID
- Name

Sample SQL Test queries:

Get all info:

```
select * from book b inner join reference_resource rr
on b.bookID = rr.resourceID
inner join cd_subject s
on rr.subjectid = s.subjectid
inner join book_author ba
on ba.bookid = b.bookid
inner join author a
on a.authorID = ba.AuthorID;
```



```
on b.bookID = rr.resourceID
inner join cd_subject s
on rr.subjectid = s.subjectid
inner join book_author ba
on ba.bookid = b.bookid
inner join author a
on a.authorID = ba.AuthorID;
```

No issues found

	BookID	Pages	ISBN	PAGES_READ	ResourceID	Title	SubjectID	ResourceTypeID	SubjectID	Name	BookAuthorID	BookID	AuthorID	ContributionOrder	AuthorID	FullName	Bio
1	1	NULL		NULL	1	Clean Architecture	1	1	1	Software Architecture	1	1	1	NULL	1	Robert C. Martin	
2	2	216.00		19.00	2	Test Driven Development By Example	2	1	2	Test Driven Development	2	2	2	NULL	2	Ken Beck	
3	3	NULL		NULL	3	Clean Code	9	1	9	Coding Best Practices	3	3	1	NULL	1	Robert C. Martin	
4	4	284.00		31.00	4	Building Real-World Web Applications with Vue.js 3	11	1	11	Vue.js	4	4	3	NULL	3	Joran Qjuinten	
5	5	573.00		71.00	5	Building Microservices	10	1	10	Microservices	5	5	4	NULL	4	Sam Newman	
6	6	NULL		NULL	6	Azure DevOps Explained	6	1	6	Azure DevOps	6	6	5	NULL	5	Sjoukje Zall, ,	
7	6	NULL		NULL	6	Azure DevOps Explained	6	1	6	Azure DevOps	7	6	6	NULL	6	Stefano Demiliani	
8	6	NULL		NULL	6	Azure DevOps Explained	6	1	6	Azure DevOps	8	6	7	NULL	7	Amit Malik	
9	7	NULL		NULL	7	Refactoring	13	1	13	Refactoring	9	7	8	NULL	8	Martin Fowler	
10	8	NULL		NULL	8	Patterns of Enterprise Application Architecture	1	1	1	Software Architecture	10	8	8	NULL	8	Martin Fowler	
11	9	NULL		NULL	9	Continuous Delivery	8	1	8	Continuous Delivery	11	9	9	NULL	9	Jez Humble	
12	9	NULL		NULL	9	Continuous Delivery	8	1	8	Continuous Delivery	12	9	10	NULL	10	David Farley	
13	11	NULL		NULL	11	Effective TypeScript	14	1	14	TypeScript	14	11	12	NULL	12	Dan Vanderkam	
14	12	NULL		NULL	12	Concurrency in C# Cookbook	16	1	16	Concurrency	15	12	13	NULL	13	Stephen Cleary	
15	13	NULL		NULL	13	Professional Git	15	1	15	Git	16	13	14	NULL	14	Brent Laster	
16	14	NULL		NULL	14	Visual Studio Code	18	1	18	Visual Studio Code	17	14	15	NULL	15	Bruce Johnson	
17	15	280.00		49.00	15	The Road to React	12	1	12	React	18	15	16	NULL	16	Robin Wieruch	
18	16	NULL		NULL	16	Microservices Architecture Development with C# 9...	10	1	10	Microservices	19	16	17	NULL	17	Andrew M. Jones	
19	17	NULL		NULL	17	DevOps Design Patterns	6	1	6	Azure DevOps	20	17	18	NULL	18	Pradeep Chintale	
20	18	NULL		NULL	18	Head First Object-Oriented Analysis & Design	26	1	26	OOP	21	18	19	NULL	19	Brett D. McLaughlin	
21	18	NULL		NULL	18	Head First Object-Oriented Analysis & Design	26	1	26	OOP	22	18	20	NULL	20	Gary Pollice	
22	18	NULL		NULL	18	Head First Object-Oriented Analysis & Design	26	1	26	OOP	23	18	21	NULL	21	David West	
23	19	NULL		NULL	19	Head First Design Patterns	27	1	27	Design Patterns	24	19	22	NULL	22	Eric Freeman	
24	19	NULL		NULL	19	Head First Design Patterns	27	1	27	Design Patterns	25	19	23	NULL	23	Elisabeth Freeman	
25	19	NULL		NULL	19	Head First Design Patterns	27	1	27	Design Patterns	26	19	24	NULL	24	Kathy Sierra	
26	19	NULL		NULL	19	Head First Design Patterns	27	1	27	Design Patterns	27	19	25	NULL	25	Bert Bates	
27	20	NULL		NULL	20	Monolith to Microservices	10	1	10	Microservices	28	20	4	NULL	4	Sam Newman	
28	25	NULL	NULL	NULL	25	Docker Deep Dive	28	1	28	Docker	29	25	11	NULL	11	Nigel Poulton	
29	27	NULL	NULL	NULL	27	Python Essentials	25	1	25	Python	34	27	26	NULL	26	John C. Shovic	NULL
30	27	NULL	NULL	NULL	27	Python Essentials	25	1	25	Python	35	27	27	NULL	27	Alan Simpson	NULL

Query executed successfully. (localdb)\MSSQLLocalDB (15.... DESKTOP-3H07OJF\corry ... ARefactoredStudyDB_Aug4 00:00:00 30 rows

Get book, all authors (if more than 1), subject:

```
select rr.title BookTitle, s.name Subject, a.fullname
from book b
inner join reference_resource rr
on rr.resourceID = b.bookID
inner join cd_subject s
on s.subjectid = rr.subjectid
inner join book_author ba
on ba.bookid = b.bookid
inner join author a
on a.authorID = ba.AuthorID;
```

Get book, pages, pages_read, subject:

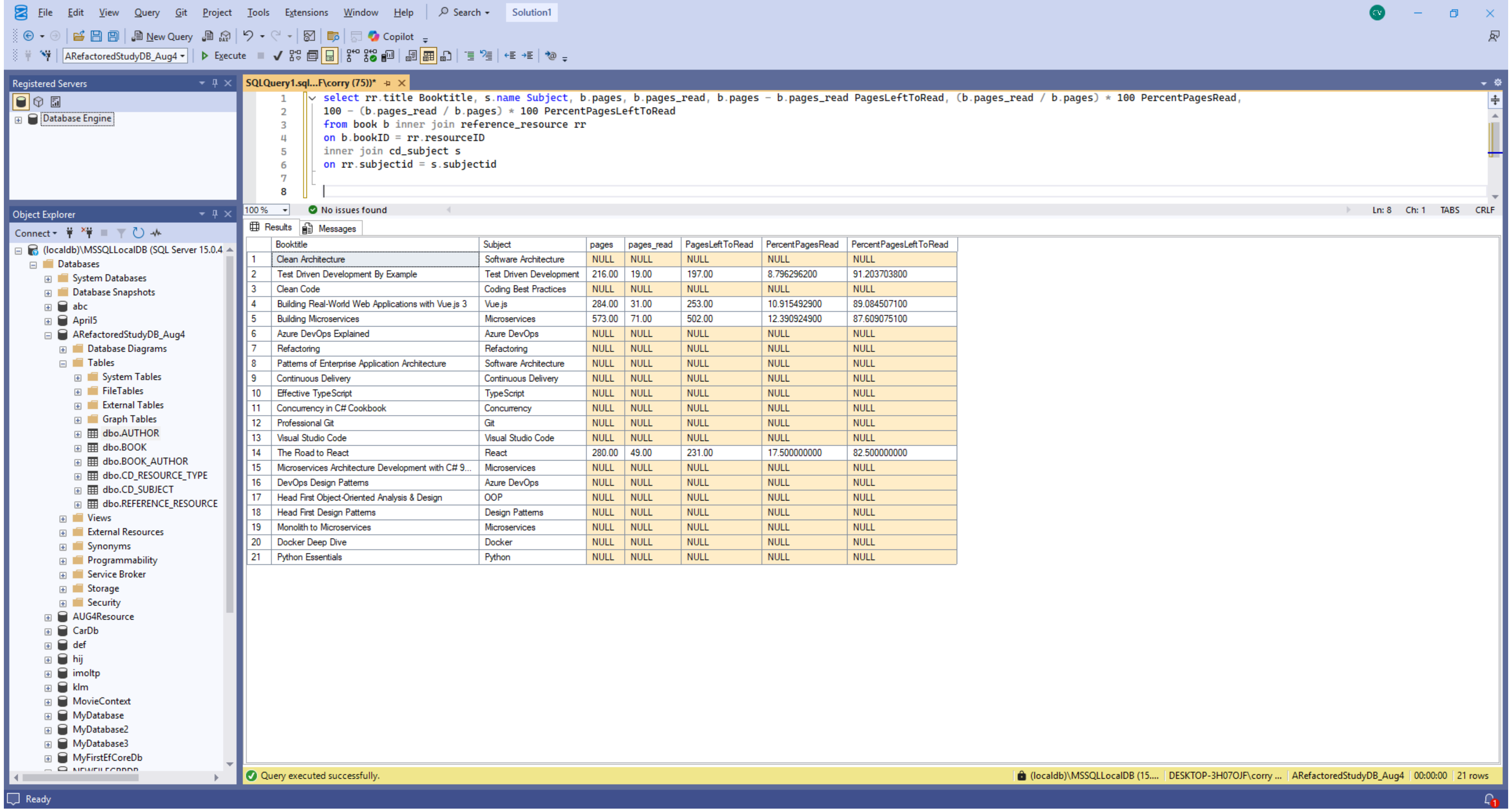
```
select rr.title BookTitle, s.name Subject, pages, pages_read
from book b inner join reference_resource rr
on b.bookID = rr.resourceID
inner join cd_subject s
on rr.subjectid = s.subjectid
```

Get booktitle, subject, pages, pages_read, percent pages read:

```
select rr.title Booktitle, s.name Subject, b.pages, b.pages_read, b.pages - b.pages_read PagesLeftToRead, (b.pages_read / b.pages) * 100 PercentPagesRead
from book b inner join reference_resource rr
on b.bookID = rr.resourceID
inner join cd_subject s
on rr.subjectid = s.subjectid
```

Get booktitle, subject, pages, pages_read, percent pages read, percent pages left to read:

```
select rr.title Booktitle, s.name Subject, b.pages, b.pages_read, b.pages - b.pages_read PagesLeftToRead, (b.pages_read / b.pages) * 100 PercentPagesRead,
100 - (b.pages_read / b.pages) * 100 PercentPagesLeftToRead
from book b inner join reference_resource rr
on b.bookID = rr.resourceID
inner join cd_subject s
on rr.subjectid = s.subjectid
```



```
select rr.title Booktitle, s.name Subject, b.pages, b.pages_read, b.pages - b.pages_read PagesLeftToRead, (b.pages_read / b.pages) * 100 PercentPagesRead,
100 - (b.pages_read / b.pages) * 100 PercentPagesLeftToRead
from book b inner join reference_resource rr
on b.bookID = rr.resourceID
inner join cd_subject s
on rr.subjectid = s.subjectid

```

	Booktitle	Subject	pages	pages_read	PagesLeftToRead	PercentPagesRead	PercentPagesLeftToRead
1	Clean Architecture	Software Architecture	NULL	NULL	NULL	NULL	NULL
2	Test Driven Development By Example	Test Driven Development	216.00	19.00	197.00	8.796296200	91.203703800
3	Clean Code	Coding Best Practices	NULL	NULL	NULL	NULL	NULL
4	Building Real-World Web Applications with Vue.js 3	Vue.js	284.00	31.00	253.00	10.915492900	89.084507100
5	Building Microservices	Microservices	573.00	71.00	502.00	12.390924900	87.609075100
6	Azure DevOps Explained	Azure DevOps	NULL	NULL	NULL	NULL	NULL
7	Refactoring	Refactoring	NULL	NULL	NULL	NULL	NULL
8	Patterns of Enterprise Application Architecture	Software Architecture	NULL	NULL	NULL	NULL	NULL
9	Continuous Delivery	Continuous Delivery	NULL	NULL	NULL	NULL	NULL
10	Effective TypeScript	TypeScript	NULL	NULL	NULL	NULL	NULL
11	Concurrency in C# Cookbook	Concurrency	NULL	NULL	NULL	NULL	NULL
12	Professional Git	Git	NULL	NULL	NULL	NULL	NULL
13	Visual Studio Code	Visual Studio Code	NULL	NULL	NULL	NULL	NULL
14	The Road to React	React	280.00	49.00	231.00	17.500000000	82.500000000
15	Microservices Architecture Development with C# 9...	Microservices	NULL	NULL	NULL	NULL	NULL
16	DevOps Design Patterns	Azure DevOps	NULL	NULL	NULL	NULL	NULL
17	Head First Object-Oriented Analysis & Design	OOP	NULL	NULL	NULL	NULL	NULL
18	Head First Design Patterns	Design Patterns	NULL	NULL	NULL	NULL	NULL
19	Monolith to Microservices	Microservices	NULL	NULL	NULL	NULL	NULL
20	Docker Deep Dive	Docker	NULL	NULL	NULL	NULL	NULL
21	Python Essentials	Python	NULL	NULL	NULL	NULL	NULL

Query executed successfully. | (localdb)\MSSQLLocalDB (15.... | DESKTOP-3H07OJF\corry ... | ARefactoredStudyDB_Aug4 | 00:00:00 | 21 rows

RESOURCE_CHATGPT

RES_ID	Question Subject	SAVE_LOCATION	
17	Visual Studio Solution Structure	E:\Documents\____AAADeveloper_new__DEVELOPER REFERENCE DOCUMENT	

RESOURCE_GOOGLE

RES_ID	AUTHOR	URL	SAVE_LOCATION	LOOKUP_DATE
18	Martin Fowler	https://martinfowler.com/bliki/CQRS.html	E:\Documents\____AAADeveloper_new__DEVELOPER REFERENCE DOCUMENT	2025-03-01

RESOURCE_YouTube_Video

RES_ID	AUTHOR	URL	SAVE_LOCATION	LOOKUP_DATE
18				

RESOURCE_Magazine_Article

RES_ID	AUTHOR	MAGAZINE	SAVE_LOCATION	DATE_READ
18				

RESOURCE_Meeting

RES_ID	TOPIC	MEETING_LOCATIONL	SAVE_LOCATION	DATE_ATTENDED

RESOURCE_Conference

RES_ID	TOPIC	CONFERENCE_ADDRESS	CONFERENCE_LOCATION	DATE_ATTENDED

RESOURCE_Tutoring

RES_ID	TUTORS_NAME	TOPIC	SAVE_LOCATION	TUTORING_DATE

RESOURCE_Class

RES_ID	SCHOOL	CLASS_NAME	SAVE_LOCATION	CLASS_DATE

RESOURCE_Informal_Discussion

RES_ID	DISCUSSION_GROUP	DISCUSSION_TOPIC	SAVE_LOCATION	DATE_OF_DISCUSSION

PERCENTAGE_BOOK_COMPLETION

RES_ID	PAGE_REACHED	DATE_REACHED	PERCENT_COMPLETED	COMPLETED		
1	377	2025-04-15	100	Y		*CLEAN ARCHITECTURE*
2	44	2025-05-11	20.37	N		*TDD BY EXAMPLE*
3	83	2025-05-11	20.39	N		*CLEAN CODE*
9	37	2025-05-11	8.37	N		*CONTINUOUS DELIVERY*
8	9	2025-05-11	1.76	N		*Patterns of Enterprise Application Architecture*
7	84	2025-05-11	20.79	N		*Refactoring*
5	46	2025-05-12	8.18	N		*Building Microservices*
16	31	2025-05-11	11.07	N		*The Road to React*
4	30	2025-05-11	10.56	N		*Building Real-World Web Applications with Vue.js 3*
21	589	2025-05-11	100	Y		*Head First Object-Oriented Analysis & Design*
10	49	2025-05-11	16.95	N		*Docker Deep Dive*
		2025-05-11		N		
		2025-05-11		N		
		2025-05-11		N		

APPENDIX A:

CREATE SCRIPTS: Test DB Relationships Created Correctly

Code to create a test db before using Fluent API to create the db and tables in the VS 2022 Solution

Create CD_RESOURCE_TYPE

```sql
CREATE TABLE CD_RESOURCE_TYPE
(
RESOURCE_TYPE_ID INT not null identity(1,1) primary key,
RESOURCE_NAME VARCHAR(50)
);
```

Create CD_RESOURCE_TOPIC

```sql
CREATE TABLE CD_RESOURCE_TOPIC
(
TOPIC_ID INT not null identity(1,1) primary key,
TOPIC_NAME VARCHAR(50)
);
```

Create RESOURCE

```sql
CREATE TABLE RESOURCE
(
RESOURCE_ID INT not null identity(1,1) primary key,
TITLE VARCHAR(100),
RESOURCE_TYPE_ID INT,
RESOURCE_TOPIC_ID INT,
CONSTRAINT FK_RESOURCE_TYPE_RESOURCE_TYPE_ID FOREIGN KEY (RESOURCE_TYPE_ID) REFERENCES
CD_RESOURCE_TYPE(RESOURCE_TYPE_ID),
CONSTRAINT FK_RESOURCE_TOPIC_TOPIC_ID FOREIGN KEY (RESOURCE_TOPIC_ID) REFERENCES CD_RESOURCE_TOPIC
(TOPIC_ID)
);
```

Create PERCENTAGE_BOOK_COMPLETION

NUMERIC datatype: eg. PERCENTAGE_COMPLETED NUMERIC(5,2)

- 5 is the Precision: the total number of digits in the value, i.e. on both sides of the decimal point
- 2 is for Scale: number of digits after the decimal point

Use BIT datatype for Boolean column datatypes

- use the BIT datatype to represent boolean data. A BIT field's value is either 1 (TRUE), 0, or null.

Enter Date in SSMS Express

- To enter April 1 2025 (e.g. `2025-04-01`) enter 05/04/01

```
CREATE TABLE PERCENTAGE_BOOK_COMPLETION
(
RESOURCE_ID INT,
PAGE_REACHED INT,
DATE_REACHED DATETIME,
PERCENTAGE_COMPLETED NUMERIC(5,2),
COMPLETED BIT
CONSTRAINT FK_RESOURCE_RESOURCE_ID FOREIGN KEY (RESOURCE_ID) REFERENCES RESOURCE(RESOURCE_ID),
);
```

Create RESOURCE_BOOK

NUMERIC datatype: eg. PERCENTAGE_COMPLETED NUMERIC(5,2)

- 5 is the Precision: the total number of digits in the value, i.e. on both sides of the decimal point
- 2 is for Scale: number of digits after the decimal point

Enter Date in SSMS Express

- To enter April 1 2025 (e.g. `2025-04-01`) enter 05/04/01

```
CREATE TABLE RESOURCE_BOOK
(
RESOURCE_ID INT,
AUTHOR VARCHAR(50),
PAGES INT,
COST NUMERIC(5,2),
DATE_REACHED DATETIME
CONSTRAINT FK_RESOURCE_BOOK_RESOURCE_ID FOREIGN KEY (RESOURCE_ID) REFERENCES RESOURCE(RESOURCE_ID),
);
```

```
Each RESOURCE_BOOK record may have one and only one RESOURCE_ID
```

Create RESOURCE_BOOK Entity that will be used to create RESOURCE_BOOK table in SQL Server db:

See document: Aug 6 - 2025 Book and Resource_Reference Entity Relationship Definitions.docx

At this path: `E:\Documents\____AAADeveloper_new\__DEVELOPER REFERENCES DOCUMENTS`

Creating The Initial Migration

Using Tools | NuGet Package Manager | Package Manager Console:

PM> Add-Migration InitialMigration

```
Build started...
Build succeeded.
Unable to create a 'DbContext' of type 'ApplicationDbContext'. The exception 'No suitable constructor was found for entity type 'ResourceBook'. The following constructors had parameters that could not be bound to properties of the entity type:
  Cannot bind 'TotalPages', 'ReadPages' in 'ResourceBook(int TotalPages, int ReadPages)'
  Cannot bind 'totPages', 'readPages' in 'ResourceBook(decimal totPages, decimal readPages)'
Note that only mapped properties can be bound to constructor parameters. Navigations to related entities, including references to owned types, cannot be bound.' was thrown while attempting to create an instance. For the different patterns supported at design time, see https://go.microsoft.com/fwlink/?linkid=851728
```

(A)
I had to install
- **Microsoft.EntityFrameworkCore (906)**
- **Microsoft.EntityFrameworkCore.SQLServer (906)**
- **Microsoft.EntityFrameworkCore.Tools (906)**
- **Microsoft.EntityFrameworkCore.Design (906)**

Then the **Add-Migration InitialMigration** command worked in Package Manager Console

(B)
Then I ran the **Update-Database** command in Package Manager Console but this too failed.
To Resolve, I had to implement the correct connectionstring in **appsettings.json:**

- To do this, I adapted the successfully implemented connectionstring I'd used in another Web API Project I had previously created (**AspNetCoreWebApi6**)

```
"ConnectionStrings": {
"MovieContext": "Server=(localdb)\\mssqllocaldb;Database=MovieContext;Trusted_Connection=True;MultipleActiveResultSets=true"
},
```

Adapted for StudyResource

```
"ConnectionStrings": {
"DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=StudyResource;Trusted_Connection=True;MultipleActiveResultSets=true"
},
```

- I also had to fix the **name** of the ConnectionString in **Program**.cs to reference its correct name as the name appears in appsettings.json, which was "DefaultConnection":

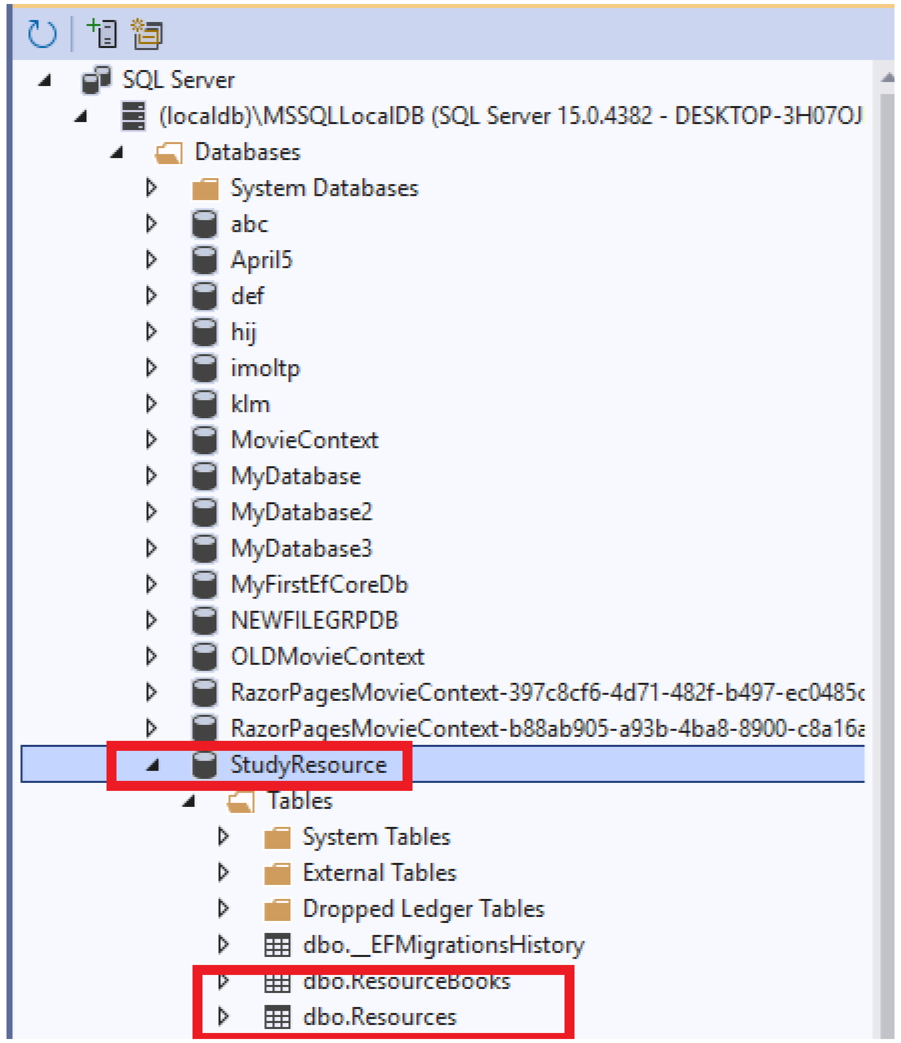
```
"ConnectionStrings": {
"DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=StudyResource;Trusted_Connection=True;MultipleActiveResultSets=true"
},

builder.Services.AddDbContext<ApplicationDbContext>(opt =>
opt.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));
```

Then in Package Manager Console in Visual Studio 2022 I again ran the Update-Database command and received the result shown below:

PM> Update-Database

```
Build started...
Build succeeded.
Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (123ms) [Parameters=[], CommandType='Text', CommandTimeout='60']
      CREATE DATABASE [StudyResource];
Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (34ms) [Parameters=[], CommandType='Text', CommandTimeout='60']
      IF SERVERPROPERTY('EngineEdition') <> 5
      BEGIN
          ALTER DATABASE [StudyResource] SET READ_COMMITTED_SNAPSHOT ON;
      END;
Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (4ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT 1
Microsoft.EntityFrameworkCore.Migrations[20411]
      Acquiring an exclusive lock for migration application. See https://aka.ms/efcore-docs-migrations-lock for more information if this takes too long.
Acquiring an exclusive lock for migration application. See https://aka.ms/efcore-docs-migrations-lock for more information if this takes too long.
Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (11ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      DECLARE @result int;
      EXEC @result = sp_getapplock @Resource = '__EFMigrationsLock', @LockOwner = 'Session', @LockMode = 'Exclusive';
      SELECT @result
Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (4ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      IF OBJECT_ID(N'[__EFMigrationsHistory]') IS NULL
      BEGIN
          CREATE TABLE [__EFMigrationsHistory] (
              [MigrationId] nvarchar(150) NOT NULL,
              [ProductVersion] nvarchar(32) NOT NULL,
              CONSTRAINT [PK___EFMigrationsHistory] PRIMARY KEY ([MigrationId])
          );
      END;
Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (0ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT 1
Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (0ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT OBJECT_ID(N'[__EFMigrationsHistory]');
Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (1ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT [MigrationId], [ProductVersion]
      FROM [__EFMigrationsHistory]
      ORDER BY [MigrationId];
Microsoft.EntityFrameworkCore.Migrations[20402]
      Applying migration '20250624032708_InitialMigration'.
Applying migration '20250624032708_InitialMigration'.
Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (1ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      CREATE TABLE [Resources] (
          [ResourceId] int NOT NULL IDENTITY,
          [Title] nvarchar(50) NOT NULL,
          [ResourceTypeId] int NOT NULL,
          [ResourceTopicId] int NOT NULL,
          CONSTRAINT [PK_Resources] PRIMARY KEY ([ResourceId])
      );
Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (1ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      CREATE TABLE [ResourceBooks] (
          [Id] int NOT NULL IDENTITY,
          [ResourceId] int NOT NULL,
          [Author] nvarchar(50) NULL,
          [Pages] int NULL,
          [Cost] decimal(18,2) NULL,
          [PublishDate] date NULL,
          CONSTRAINT [PK_ResourceBooks] PRIMARY KEY ([Id]),
          CONSTRAINT [FK_ResourceBooks_Resources_ResourceId] FOREIGN KEY ([ResourceId]) REFERENCES [Resources] ([ResourceId]) ON DELETE CASCADE
      );
Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (1ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      CREATE UNIQUE INDEX [IX_ResourceBooks_ResourceId] ON [ResourceBooks] ([ResourceId]);
Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (1ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      INSERT INTO [__EFMigrationsHistory] ([MigrationId], [ProductVersion])
      VALUES (N'20250624032708_InitialMigration', N'9.0.6');
Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (2ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      DECLARE @result int;
      EXEC @result = sp_releaseapplock @Resource = '__EFMigrationsLock', @LockOwner = 'Session';
      SELECT @result
Done.
```

I then opened SQL Server Object Explorer in Visual Studio and found my database, plus the 2 tables, indicating the Migration was successful:



- SQL Server
  - (localdb)\MSSQLLocalDB (SQL Server 15.0.4382 - DESKTOP-3H07OJ
    - Databases
      - System Databases
      - abc
      - April5
      - def
      - hij
      - imoltp
      - klm
      - MovieContext
      - MyDatabase
      - MyDatabase2
      - MyDatabase3
      - MyFirstEfCoreDb
      - NEWFILEGRPDB
      - OLDMovieContext
      - RazorPagesMovieContext-397c8cf6-4d71-482f-b497-ec0485c
      - RazorPagesMovieContext-b88ab905-a93b-4ba8-8900-c8a16a
      - StudyResource
        - Tables
          - System Tables
          - External Tables
          - Dropped Ledger Tables
          - dbo.__EFMigrationsHistory
          - dbo.ResourceBooks
          - dbo.Resources

-

APPENDIX B:

TEST QUERIES

Get Resource Topic and Resource Type

```
select r.title, tp.topic_name, ty.resource_name from
resource r inner join cd_resource_topic tp
on r.resource_topic_id = tp.topic_id
inner join cd_resource_type ty
on r.resource_type_id = ty.resource_type_id
order by tp.topic_name;
```

Get Resource Topic and Resource Type by Resource Type

```
select r.title, tp.topic_name, ty.resource_name from
resource r inner join cd_resource_topic tp
on r.resource_topic_id = tp.topic_id
inner join cd_resource_type ty
on r.resource_type_id = ty.resource_type_id
where ty.resource_type_id = 3
order by tp.topic_name;
```

Get Book Data That Includes a Calculation of Percentage of Each Book Read.

```
select r.title, rb.author, tp.topic_name, ty.resource_name,  rb.pages, pbc.page_reached,
pbc.percentage_completed,
CAST(pbc.page_reached AS DECIMAL(7,2))/CAST(rb.pages AS DECIMAL(7,2)) * 100 AS PERCENTAGE_OF_BOOK_READ
from resource r inner join cd_resource_topic tp
on r.resource_topic_id = tp.topic_id
inner join cd_resource_type ty
on r.resource_type_id = ty.resource_type_id
inner join resource_book rb
on rb.resource_id = r.resource_id
inner join percentage_book_completion pbc
on pbc.resource_id = r.resource_id
--where ty.resource_type_id = 3
order by tp.topic_name;
```

Entity Framework Core Datamodelling

database model with child tables having diverse columns to each other but same parent table

ChatGPT said:

When designing a database with a parent table and multiple child tables, each child table having diverse columns but sharing the same parent table, you're essentially looking at a **single parent-child relationship** where each child table may have distinct attributes.

Common Approach: Single Table Inheritance or Class Table Inheritance

There are two primary ways to model this scenario:

1. **Single Table Inheritance** (STI) – All data is stored in a single parent table. Each child table type (or subclass) is represented by a column to indicate its specific type. However, this can lead to many nullable fields since not all columns will be used by each child.
2. **Class Table Inheritance** (CTI) – The parent table holds common attributes, and each child table stores its specific attributes. There will be a foreign key reference to the parent table.

I'll explain the **Class Table Inheritance** approach (CTI), which is usually the best way to handle diverse columns across child tables while maintaining the relationship to a single parent table.

Class Table Inheritance (CTI) Model:

Example Scenario:

Let's say we have a **Vehicle** as the parent table. The child tables could be **Car**, **Truck**, and **Motorcycle**, each of which will have different attributes.

---

1. Parent Table: vehicle

The vehicle table holds all the common attributes for all types of vehicles.

vehicle_id	make	model	year	type
1	Toyota	Camry	2020	Car
2	Ford	F-150	2018	Truck
3	Harley	Sportster	2021	Motorcycle

- **vehicle_id**: Primary key, unique identifier for each vehicle.
- **make**: The brand/manufacturer of the vehicle.
- **model**: The model name or number of the vehicle.
- **year**: The manufacturing year of the vehicle.
- **type**: A type field that could contain values such as "Car", "Truck", "Motorcycle", etc., to indicate the type of vehicle.

---

2. Child Tables:

Each of the child tables (car, truck, motorcycle) will contain unique fields for the attributes specific to that type of vehicle.

Table: car

car_id	vehicle_id	doors	transmission	fuel_type
1	1	4	Automatic	Petrol

- **vehicle_id**: Foreign key linking back to the parent table vehicle.
- **doors**: The number of doors in the car.
- **transmission**: Type of transmission (e.g., automatic, manual).
- **fuel_type**: Type of fuel used (e.g., petrol, diesel, electric).

Table: truck

truck_id	vehicle_id	bed_size	towing_capacity	wheels
1	2	Large	15000 lbs	6

- **vehicle_id**: Foreign key linking back to the parent table vehicle.
- **bed_size**: Size of the truck bed (e.g., small, medium, large).
- **towing_capacity**: Maximum weight the truck can tow.
- **wheels**: Number of wheels (e.g., 4 for regular trucks, 6 for larger ones).

Table: motorcycle

motorcycle_id	vehicle_id	engine_capacity	is_sport
1	3	1200cc	True

- **vehicle_id**: Foreign key linking back to the parent table vehicle.
- **engine_capacity**: The engine size in cc (cubic centimeters).
- **is_sport**: Whether the motorcycle is a sports bike (True/False).

ER Diagram for this Design:

lua

CopyEdit

```
                  +-----------+
                  |  Vehicle  |
                  +-----------+
                  | vehicle_id| <--------+
                  | make      |          |
                  | model     |          |
                  | year      |          |
                  | type      |          |
                  +-----------+          |
                        |                |
          +------------+------------+    |
          |            |            |    |
     +--------+   +--------+   +------------+
     |  Car   |   |  Truck |   | Motorcycle |
     +--------+   +--------+   +------------+
     | car_id |   | truck_id|   | motorcycle_id |
     | vehicle_id| | vehicle_id | | vehicle_id |
     | doors   |   | bed_size |   | engine_capacity |
     | transmission| | towing_capacity| | is_sport |
     | fuel_type |   | wheels  |    +------------+
     +--------+   +--------+
```

Advantages of this Model (CTI):

- **Data Integrity**: Each vehicle type has its own table, with the shared attributes stored only once in the vehicle table.
- **Scalability**: If you add another vehicle type (like a Bicycle), you only need to create a new child table and link it to the vehicle table.
- **Performance**: Common queries can be optimized on the parent table. Queries for specific vehicle types can be optimized separately in child tables.

Challenges:

- **Complex Queries**: When querying all vehicles, you'll need to join the parent table with the child tables.
- **Foreign Key Constraints**: Ensuring referential integrity across tables.

Alternate Model: Single Table Inheritance (STI)

Another approach is using a **single table** for all vehicle types, where a type column differentiates the vehicle type. However, this might lead to many nullable fields for the attributes that are only relevant to specific types.

Here's a simple example of the **Single Table Inheritance** model:

vehicle_id	make	model	year	type	doors	transmission	fuel_type	bed_size	towing_capacity	engine_capacity	is_sport
1	Toyota	Camry	2020	Car	4	Automatic	Petrol	NULL	NULL	NULL	NULL
2	Ford	F-150	2018	Truck	NULL	NULL	NULL	Large	15000 lbs	NULL	NULL
3	Harley	Sportster	2021	Motorcycle	NULL	NULL	NULL	NULL	NULL	1200cc	True

This model is simpler to implement but less normalized and can cause problems with performance as the table grows.

Conclusion:

- If you want to store different attributes for different child entities, **Class Table Inheritance (CTI)** is usually the more flexible and normalized approach.
- If the differences are minimal, or if you don't mind sparse columns, **Single Table Inheritance (STI)** may be simpler.

Does this make sense, or would you like help refining it based on your specific use case?

You said:

You don't mention that each child table has its own Primary key - e.g. car_id, truck_id. Are these needed. Is it good practice to include these?

ChatGPT said:

Great question! Yes, each child table (e.g., car, truck, motorcycle) can have its own **primary key** (e.g., car_id, truck_id, motorcycle_id). However, the inclusion of these primary keys depends on the design choice and the specific relationship between the parent and child tables.

Primary Key Design in Class Table Inheritance (CTI)

In Class Table Inheritance (CTI), it's common practice to include a **primary key for each child table**, but there's a specific reason behind it, and this needs to be done carefully:

Key Points to Consider:

1. **Foreign Key as Primary Key:**
   - The vehicle_id in each child table (e.g., car, truck, motorcycle) is typically a **foreign key** that references the parent table (vehicle).
   - Often, the vehicle_id in the child table **also acts as the primary key** for the child table because each child entry is uniquely identified by its corresponding parent entry.
2. **Separate Primary Key for Each Child Table:**
   - In some cases, a separate car_id, truck_id, or motorcycle_id **primary key** is included to maintain clarity and structure. This approach can be useful if you need to query the child tables independently or maintain strict normalization.

When to Use Separate Primary Keys (car_id, truck_id):

1. **To Ensure Entity Uniqueness per Child Table:**
   - If you want to **explicitly distinguish** between child entities (e.g., Car, Truck, Motorcycle) and ensure that each child has a unique identifier, having a separate primary key like car_id, truck_id, etc., might make sense.
   - This can be useful when the child tables are expected to grow independently or have specific use cases that require unique keys for each child entity.
2. **Clarifying Relationships:**
   - If you're going to join the parent and child tables in complex queries or if you're working with a large set of child-specific data, having distinct primary keys (car_id, truck_id) can make the structure clearer.

Why vehicle_id Could Serve as the Primary Key:

If you choose to use the vehicle_id as the **primary key** in each child table, you're essentially saying that **each child entry is directly related to exactly one vehicle**, and there's no need for an additional unique identifier in the child table. In this case, the child table will inherit the primary key from the parent table.

Here's the rationale behind this approach:

1. **Simplified Design:**
   - Using vehicle_id as the primary key in each child table reduces redundancy and complexity. You don't need extra primary keys like car_id or truck_id.
   - The primary key (vehicle_id) ensures that each record in the child table is uniquely tied to a record in the parent table.
2. **No Redundant Data:**
   - If vehicle_id is already unique and serves as the primary key, you don't need another car_id, truck_id, etc., to identify rows in the child tables. This reduces unnecessary columns and makes the schema more efficient.
3. **Foreign Key Integrity:**
   - The foreign key (vehicle_id) in each child table enforces referential integrity, ensuring that each child entry corresponds to a valid entry in the parent table.

Example with vehicle_id as Primary Key:

Let's look at the car table when vehicle_id is the primary key:

Parent Table: vehicle

vehicle_id	make	model	year	type
1	Toyota	Camry	2020	Car
2	Ford	F-150	2018	Truck
3	Harley	Sportster	2021	Motorcycle

Child Table: car

vehicle_id	doors	transmission	fuel_type
1	4	Automatic	Petrol

- In this case, vehicle_id is both the **primary key** and the **foreign key** linking the child (car) to the parent (vehicle).

When to Use Separate Primary Keys (car_id, truck_id):
If you're using a separate primary key, like car_id, you might have the following:
Parent Table: vehicle

vehicle_id	make	model	year	type
1	Toyota	Camry	2020	Car
2	Ford	F-150	2018	Truck
3	Harley	Sportster	2021	Motorcycle

Child Table: car

car_id	vehicle_id	doors	transmission	fuel_type
1	1	4	Automatic	Petrol

- **car_id** is the **primary key** in the car table, while **vehicle_id** is the **foreign key** that references the vehicle table.
- This setup is useful if you want to clearly differentiate between car_id, truck_id, and other entity-specific identifiers, while still maintaining the connection to the parent.

Conclusion:

- **If vehicle_id is unique across all vehicle types (Car, Truck, Motorcycle)** and you don't foresee a need for separate keys in child tables, you can use **vehicle_id as the primary key** in each child table.
- **If each child table requires independent identifiers** (e.g., for clarity or specific use cases like tracking car, truck, or motorcycle IDs independently), then **separate primary keys** (car_id, truck_id, etc.) can be used. These would still link back to the vehicle_id as a foreign key.

Both approaches are valid; it's just a matter of what fits your business logic and use case best. If you don't need the extra primary key for child tables, using just vehicle_id is simpler and avoids redundancy. But if you need unique identifiers for each child entity (or need to perform child-specific operations), adding them makes sense.

Let me know if you'd like further clarification on any part!

You said:

On this question of the need for a separate PK, does SQL Server conventions prefere a separate PK (e.g. as an identity column)?

ChatGPT said:

Yes, SQL Server conventions often **prefer** a separate primary key for child tables (like car_id, truck_id, etc.), especially when the child tables are considered **independent entities** with their own **identity**. This is particularly the case when the child tables will be used in **complex operations** or when you need to uniquely identify each child record in ways that don't necessarily depend on the parent.

Let's break down the reasoning behind SQL Server conventions and why a separate **identity** column (like car_id, truck_id) is often favored:

1. SQL Server's Identity Column Convention:

In SQL Server, it's a common practice to use an **identity column** for automatically generating primary keys for tables. This is often the case for child tables in a **Class Table Inheritance** model, where the child entity (e.g., Car, Truck, Motorcycle) might have its own specific set of operations or independent queries.

For example, in SQL Server, you might define a child table (Car) with an **identity column** as the primary key:

sql
CopyEdit

```
CREATE TABLE car (
   car_id INT IDENTITY PRIMARY KEY,
   vehicle_id INT NOT NULL,
   doors INT,
   transmission VARCHAR(50),
   fuel_type VARCHAR(50),
   FOREIGN KEY (vehicle_id) REFERENCES vehicle(vehicle_id)
);
```

Here, the car_id is an **identity column**, which means SQL Server will automatically generate unique values for it. This is often used for **internal purposes** and makes the table easier to work with for queries, joins, and operations.

2. Why Use a Separate Primary Key (car_id, truck_id, etc.) in SQL Server?

SQL Server conventions often favor a **separate primary key** for the following reasons:

1. Performance:

- **Independent Primary Keys** can improve the performance of queries, especially when child tables grow large.
- Having a unique car_id, truck_id, etc., allows you to perform operations on child tables without needing to always join with the parent (vehicle) table, which could improve query performance in certain cases (e.g., indexing, searching).

2. Flexibility in Querying:

- **Separate identity columns** give you more **flexibility** in performing operations that are specific to the child entity (e.g., querying only cars or performing a DELETE operation on just the car table without affecting the vehicle table).
- For example, you could execute an independent DELETE on just the Car table without needing to reference the parent Vehicle table each time:

sql
CopyEdit

```sql
DELETE FROM car WHERE car_id = 1;
```

This makes it easier to manage child records directly when they need to be handled independently.

3. Better Handling of Relationships:

- SQL Server is optimized to handle **foreign key relationships** between tables with independent **primary keys**. When you use a separate identity primary key (car_id, truck_id), it allows for more straightforward **referential integrity** enforcement and indexing.

4. Compatibility with Other Systems:

- Many developers or systems prefer **separate identity columns** for better compatibility with ORM (Object-Relational Mapping) frameworks, API integrations, or migration tools. ORM frameworks like **Entity Framework** or **Django ORM** typically expect a primary key to be a simple **auto-incrementing column**, even if it's used as a foreign key in other tables.

When You Might Skip the Separate Primary Key (car_id, truck_id):

While SQL Server prefers a separate identity column in child tables for the reasons mentioned above, there are some cases where using **just the vehicle_id as the primary key** might be appropriate:

1. Simplicity and Smaller Tables:

- If your child tables are small or if you don’t anticipate complex queries or operations on them independently (e.g., only querying vehicles as a whole and rarely needing child-specific operations), you might choose to just use the vehicle_id as the **primary key** in each child table.

2. Less Redundancy:

- By using vehicle_id as the primary key in child tables, you **avoid redundancy** and save space. In this case, the child tables are tightly coupled with the parent, and you don't need an additional identifier like car_id, truck_id, etc.

sql
CopyEdit

```sql
CREATE TABLE car (
   vehicle_id INT PRIMARY KEY,   -- vehicle_id is both PK and FK
   doors INT,
   transmission VARCHAR(50),
   fuel_type VARCHAR(50),
   FOREIGN KEY (vehicle_id) REFERENCES vehicle(vehicle_id)
);
```

3. When vehicle_id is Sufficient for All Operations:

- If every child entity (like a Car, Truck, Motorcycle) will always be accessed **via its vehicle_id** (e.g., queries will always reference vehicle_id), then you might not need a separate identity column, as vehicle_id uniquely identifies the row.

However, this approach is generally more appropriate in smaller applications or situations where **simplicity** is a priority, and you don’t foresee needing complex operations on the child tables themselves.

Example: Using Separate Identity Columns (car_id, truck_id)

sql
CopyEdit

```sql
-- Parent Table: vehicle
CREATE TABLE vehicle (
   vehicle_id INT IDENTITY PRIMARY KEY,
   make VARCHAR(50),
   model VARCHAR(50),
  year INT,
  type VARCHAR(50)
);

-- Child Table: car
CREATE TABLE car (
  car_id INT IDENTITY PRIMARY KEY,   -- Separate primary key
  vehicle_id INT NOT NULL,
  doors INT,
  transmission VARCHAR(50),
  fuel_type VARCHAR(50),
  FOREIGN KEY (vehicle_id) REFERENCES vehicle(vehicle_id)
);

-- Child Table: truck
CREATE TABLE truck (
  truck_id INT IDENTITY PRIMARY KEY,   -- Separate primary key
  vehicle_id INT NOT NULL,
  bed_size VARCHAR(50),
  towing_capacity INT,
  wheels INT,
  FOREIGN KEY (vehicle_id) REFERENCES vehicle(vehicle_id)
);
```

In this setup:

- **car_id** and **truck_id** are **identity columns** that SQL Server automatically increments.
- **vehicle_id** is the **foreign key** that relates the child tables to the parent (vehicle).

Summary:

- **SQL Server convention** often prefers **separate identity columns** (car_id, truck_id, etc.) for child tables in a Class Table Inheritance model, as it provides better performance, flexibility, and compatibility with tools like ORMs.
- Using **vehicle_id as the primary key** in each child table (without a separate identity column) can be simpler, but is typically only suitable for smaller or less complex systems.

If you're building a system where child entities will have a lot of **independent behavior**, need to be accessed **independently** of the parent, or need to be referenced easily in **other tables**, it's a good idea to use separate identity columns for each child table.

Does this help clarify the conventions? Let me know if you need more details!

You said:

Can you show me ALL the necessary code, when using EF Core, needed in various folders, to implement a Car class Entity that is a child of a Vehicle class, but with some unique columns, and implemented using Fluent API? Those Entities will be used in a code-first migration to create tables in SQL Server. This might be an Entity used in a .NET Core Web API application.

ChatGPT said:

Certainly! To implement a **Car** class entity that is a child of a **Vehicle** class with unique columns, using **EF Core** with **Fluent API** in a **code-first migration**, we'll break down the required steps and organize the code in the typical folder structure you'd use for an **ASP.NET Core Web API** project.

Folder Structure:

- **Entities**: Where the model classes (Vehicle and Car) reside.
- **Data**: Contains the DbContext (e.g., ApplicationDbContext).
- **Migrations**: Contains the migrations that will be generated from the code-first approach.

Let's go step-by-step:

---

1. Install EF Core and SQL Server Packages

First, ensure you have the necessary NuGet packages installed in your **ASP.NET Core Web API** project.

bash

CopyEdit

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.Tools
```

---

2. Create the Vehicle and Car Entity Classes
You'll create your Vehicle and Car entities. The Car class will have additional unique columns like Doors, Transmission, etc., while inheriting common properties from Vehicle.

Entities/Vehicle.cs (Parent Class)

```
namespace YourApp.Entities
{
    public class Vehicle
    {
        public int VehicleId { get; set; }  // Primary Key
        public string Make { get; set; }
        public string Model { get; set; }
        public int Year { get; set; }
        public string Type { get; set; }  // Used to distinguish between vehicle types like "Car", "Truck", etc.
    }
}
```

Entities/Car.cs (Child Class)

```
namespace YourApp.Entities
{
    public class Car : Vehicle
    {
        public int Doors { get; set; }
        public string Transmission { get; set; }
        public string FuelType { get; set; }
    }
}
```

---

3. Create the ApplicationDbContext

The ApplicationDbContext class will manage the relationship between the entities and the database.

Data/ApplicationDbContext.cs
csharp
CopyEdit

```
using Microsoft.EntityFrameworkCore;
using YourApp.Entities;

namespace YourApp.Data
{
    public class ApplicationDbContext : DbContext
    {
        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) : base(options)
        {
        }
```

```
public DbSet<Vehicle> Vehicles { get; set; }
public DbSet<Car> Cars { get; set; }

// Fluent API configuration goes here
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    // Vehicle configuration
    modelBuilder.Entity<Vehicle>()
        .HasKey(v => v.VehicleId);  // Primary key for the Vehicle table

    modelBuilder.Entity<Vehicle>()
        .Property(v => v.Make)
        .IsRequired()
        .HasMaxLength(50);

    modelBuilder.Entity<Vehicle>()
        .Property(v => v.Model)
        .IsRequired()
        .HasMaxLength(50);

    modelBuilder.Entity<Vehicle>()
        .Property(v => v.Year)
        .IsRequired();

    modelBuilder.Entity<Vehicle>()
        .Property(v => v.Type)
        .IsRequired()
        .HasMaxLength(20);
```

```
            // Car configuration
            modelBuilder.Entity<Car>()
                .HasKey(c => c.VehicleId);  // Car uses VehicleId as its primary key

            modelBuilder.Entity<Car>()
                .HasOne(c => c.Vehicle)
                .WithMany()
                .HasForeignKey(c => c.VehicleId)
                .OnDelete(DeleteBehavior.Cascade);

            modelBuilder.Entity<Car>()
                .Property(c => c.Doors)
                .IsRequired();

            modelBuilder.Entity<Car>()
                .Property(c => c.Transmission)
                .HasMaxLength(20);

            modelBuilder.Entity<Car>()
                .Property(c => c.FuelType)
                .HasMaxLength(20);
        }
    }
}
```

Explanation of Fluent API in OnModelCreating:

- The Vehicle entity's properties are configured to ensure they are required and have appropriate lengths.
- The Car entity uses the VehicleId from the parent Vehicle as its primary key.
- The Car entity also has its own specific properties (Doors, Transmission, FuelType).
- The foreign key relationship between Car and Vehicle is set up, with cascading deletes.

---

4. Configure the DbContext in Startup.cs or Program.cs

In the **Program.cs** or **Startup.cs** file (depending on your .NET Core version), you'll configure the connection string and set up EF Core to use SQL Server.

Program.cs (for .NET 6+)

csharp
CopyEdit

```
using Microsoft.EntityFrameworkCore;
using YourApp.Data;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

builder.Services.AddControllers();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
```

```
var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();

app.MapControllers();

app.Run();
```

Make sure that in your **appsettings.json**, you have the correct connection string:

appsettings.json

json

CopyEdit

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=YourAppDb;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

---

5. Create the Migration and Update the Database

After setting up your DbContext, you can create your migration and apply it to the database.

Run the following commands in the **Package Manager Console** or **CLI**:

bash

CopyEdit

```
# Create migration files
dotnet ef migrations add InitialCreate

# Apply migration to the database
dotnet ef database update
```

This will generate the SQL commands to create the Vehicle and Car tables in your SQL Server database, including the necessary foreign key and column constraints.

---

6. Create an API Controller (Optional)

You might want to create an API controller to interact with the Vehicle and Car entities.

Controllers/CarController.cs

csharp

CopyEdit

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using YourApp.Data;
using YourApp.Entities;

namespace YourApp.Controllers
```

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

    public CarController(ApplicationDbContext context)
    {
      _context = context;
    }

    // GET: api/Car
    [HttpGet]
    public async Task<ActionResult<IEnumerable<Car>>> GetCars()
    {
      return await _context.Cars.Include(c => c.Vehicle).ToListAsync();
    }

    // POST: api/Car
    [HttpPost]
    public async Task<ActionResult<Car>> PostCar(Car car)
    {
      _context.Cars.Add(car);
      await _context.SaveChangesAsync();

      return CreatedAtAction(nameof(GetCars), new { id = car.VehicleId }, car);
    }
  }
}
```

This CarController exposes endpoints to get a list of cars and add a new car.

Conclusion

The folder structure and code implementation for the Vehicle and Car entities (with Fluent API) in an EF Core **code-first** migration are as follows:

1. **Entities**: Defines the Vehicle and Car models.
2. **Data**: Contains ApplicationDbContext that configures the database and relationship between the Vehicle and Car tables.
3. **Program.cs**: Configures the database connection.
4. **Migration**: Generates SQL schema and applies it to SQL Server.
5. **API Controller**: Exposes CRUD operations for Car via a Web API.

Now, you can proceed with using these entities in your Web API to perform operations on Vehicle and Car. Would you like to explore any further details or implementations?

You said:

does ApplicationDbContext go in the same folder as the Entities in a .NET Core Web API project folder structure in a vs2022 Solution?

ChatGPT said:

Great question! In a typical **ASP.NET Core Web API** project structure, **ApplicationDbContext** is usually placed in a **separate folder** from the **Entities** folder. This separation helps maintain a clean and well-organized project, especially as the project grows and more components are added.

Folder Structure in a VS2022 Solution

Here's a suggested folder structure for a typical **ASP.NET Core Web API** project:

graphql

CopyEdit

```
/YourApp
│
├── /Controllers         # API controllers for handling HTTP requests
│   ├── CarController.cs
│   └── VehicleController.cs
│
├── /Entities           # Entity classes representing database models
│   ├── Vehicle.cs
│   └── Car.cs
│
├── /Data               # Contains ApplicationDbContext and database-related services
│   └── ApplicationDbContext.cs
│
├── /Migrations         # EF Core migration files (automatically generated)
│   └── ...
│
├── /Models             # (Optional) DTOs and ViewModels for API responses
│   └── CarDto.cs
│
├── /Services           # (Optional) Business logic services
│   └── CarService.cs
│
├── /wwwroot            # Static files (if any)
│   └── ...
│
├── appsettings.json     # Configuration file for connection strings and app settings
├── Program.cs           # Entry point and application setup
└── Startup.cs           # If using .NET 5 or earlier, this file is still used for setup
```

Explanation of Folder Placement:

1. **/Entities**: This folder contains your **entity classes —** the models representing the tables in the database. In this case, Vehicle.cs and Car.cs go here. These classes define your domain model and usually mirror the structure of the database tables (using EF Core).
2. **/Data**: This folder contains your **database context class** (ApplicationDbContext.cs) and any data access code. It's separate from your entities to help decouple the data layer from your domain model. The ApplicationDbContext is responsible for configuring the EF Core context, including any relationships and Fluent API configurations (like the Car-Vehicle inheritance).
3. **/Controllers**: Contains **API controllers** responsible for handling HTTP requests, interacting with services, and returning data to the client.
4. **/Migrations**: This folder is where EF Core stores the **migration files** that are automatically generated when you run dotnet ef migrations add.
5. **/Models** (Optional): This folder contains **DTOs** (Data Transfer Objects) or **ViewModels**. You might use these to shape the data being sent or received by your API, separate from your domain entities.
6. **/Services** (Optional): If you have **business logic** or **application services** to abstract away from your controller, this is where you'd place them. For example, a CarService might contain logic for creating cars, validating data, etc.

Putting ApplicationDbContext in the /Data Folder

ApplicationDbContext logically belongs in the /Data folder because it is the **data access layer** that interacts with the database. This separation follows a clean architecture principle, keeping your **data access logic** separate from your **domain model** and **controller logic**.

Example:

Here's how your folder structure might look in Visual Studio 2022:

pgsql

CopyEdit

/YourApp

```
│
├── /Controllers
│   └── CarController.cs
│   └── VehicleController.cs
│
├── /Entities
```

```
│   └── Vehicle.cs
│   └── Car.cs
│
├── /Data
│   └── ApplicationDbContext.cs  <-- Contains DB context and Fluent API configuration
│
├── /Migrations
│   └── 20230531000000_InitialCreate.cs
│   └── 20230531000000_InitialCreate.Designer.cs
│   └── 20230531000000_InitialCreate.sql
│
├── /Models
│   └── CarDto.cs
│
├── appsettings.json
├── Program.cs
└──Startup.cs
```

Why Separate ApplicationDbContext from Entities?

- **Separation of Concerns**: The ApplicationDbContext is responsible for interacting with the database, while Entities represent your domain model. Keeping them in different folders helps maintain clarity and makes the code easier to manage.
- **Scalability**: As your project grows, you may introduce new models (entities), database-related logic, or services. A Data folder keeps the database-related code separate from the business logic (in Services) and presentation logic (in Controllers).
- **Testing**: With this structure, it becomes easier to mock or isolate parts of the data layer for unit testing. For example, you can easily write tests for your ApplicationDbContext by isolating it into the /Data layer.

Example Code for ApplicationDbContext.cs in /Data Folder:

csharp
CopyEdit

```
using Microsoft.EntityFrameworkCore;
using YourApp.Entities;

namespace YourApp.Data
{
  public class ApplicationDbContext : DbContext
  {
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
      : base(options)
    { }

    public DbSet<Vehicle> Vehicles { get; set; }
    public DbSet<Car> Cars { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
      // Vehicle configuration
      modelBuilder.Entity<Vehicle>()
        .HasKey(v => v.VehicleId);

      modelBuilder.Entity<Vehicle>()
```

```
            .Property(v => v.Make)
            .IsRequired()
            .HasMaxLength(50);

        modelBuilder.Entity<Vehicle>()
            .Property(v => v.Model)
            .IsRequired()
            .HasMaxLength(50);

        modelBuilder.Entity<Vehicle>()
            .Property(v => v.Year)
            .IsRequired();

        modelBuilder.Entity<Vehicle>()
            .Property(v => v.Type)
            .IsRequired()
            .HasMaxLength(20);

        // Car configuration
        modelBuilder.Entity<Car>()
            .HasKey(c => c.VehicleId); // Car uses VehicleId as its primary key

        modelBuilder.Entity<Car>()
            .HasOne(c => c.Vehicle)
            .WithMany()
            .HasForeignKey(c => c.VehicleId)
            .OnDelete(DeleteBehavior.Cascade);

        modelBuilder.Entity<Car>()
            .Property(c => c.Doors)
            .IsRequired();

        modelBuilder.Entity<Car>()
            .Property(c => c.Transmission)
            .HasMaxLength(20);

        modelBuilder.Entity<Car>()
            .Property(c => c.FuelType)
            .HasMaxLength(20);
    }
  }
}
```

This code assumes you are using the **Fluent API** to configure the Vehicle and Car entities and their relationships.

Conclusion

To sum it up:

- The **ApplicationDbContext** should go in the **/Data** folder, separate from the **/Entities** folder.
- This organization helps maintain a clean separation of concerns, improves maintainability, and supports scaling your application as it grows.

- This structure will also help you implement better **unit tests** and **services** in the future.

Would you like to explore anything else related to this structure, or do you have additional questions about Entity Framework Core in this context?

You said:

if the solution contains a .net core web api project and a mstest project does entity framework core go in the solution folder or just in the .net core web api project folder?

ChatGPT said:

In a typical **solution** structure containing both a **.NET Core Web API project** and a **MSTest project**, **Entity Framework Core (EF Core)** and its related dependencies should generally reside in the **.NET Core Web API project** folder. This is because EF Core is used directly in your application code for database access, and it's usually **not needed in the MSTest project**.

Here's how the folder structure should look:

Suggested Folder Structure:

graphql

CopyEdit

```
/YourSolution
│
├── /YourApp.WebAPI             # .NET Core Web API Project
│   ├── /Controllers
│   ├── /Entities
│   ├── /Data                   # Contains ApplicationDbContext and DB-related logic
│   ├── /Migrations             # EF Core migration files
│   ├── /Models                 # DTOs, ViewModels
│   ├── /Services               # Business logic
│   ├── appsettings.json        # DB connection string, app settings
│   ├── Program.cs              # Application setup and services configuration
│   └── Startup.cs              # For .NET 5 or earlier (configures services and middleware)
│
├── /YourApp.Tests              # MSTest Project for unit tests
│   ├── /Controllers            # Unit tests for API controllers
│   ├── /Services               # Unit tests for application services
│   └── /Data                   # Unit tests for data layer or repository (optional)
│
├── /YourApp.sln                # Solution file
└── /README.md                  # Project documentation
```

Key Points:

1. **EF Core in the Web API Project**:
   - ○ **Entity Framework Core** should go into the **Web API project** (YourApp.WebAPI) because it's the project that directly interacts with the database.
   - ○ The ApplicationDbContext, migrations, and EF Core-related logic (such as configuring the database, setting up DbSets, Fluent API configurations, etc.) should all be part of this project.
2. **MSTest Project**:
   - ○ The **MSTest project** is meant for testing and should not contain EF Core by itself.
   - ○ However, it **will reference** the **Web API project** (YourApp.WebAPI) so it can test the application code, including the database-related operations (via ApplicationDbContext).
   - ○ **Unit tests for data access** can be performed in the MSTest project, but you don't need EF Core in the test project directly — instead, you'll interact with the Web API's ApplicationDbContext or mock the DB context in tests.
3. **Dependencies**:
   - ○ In your **Web API project**, you'll install EF Core, SQL Server, and migration tools via NuGet. For example:

bash

CopyEdit

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.Tools
```

   - ○ In your **MSTest project**, you typically wouldn't add EF Core dependencies directly unless you're specifically testing DB-related operations and need to interact with the real database (which might require mocking the DB context or using an in-memory database).

Example of Reference Setup Between Projects:

- The **MSTest project** will **reference** the **Web API project** to be able to test the Web API and data-related logic. You do this by adding a project reference from the **MSTest project** to the **Web API project**:

In the MSTest project folder, run the following command:

bash
CopyEdit
```bash
dotnet add reference ../YourApp.WebAPI/YourApp.WebAPI.csproj
```
This way, your MSTest project can access the Web API's models, DbContext, and services while writing tests.

Example of Folder Details:
Web API Project (YourApp.WebAPI)

- **Entities Folder**:
  Contains the classes for your domain models like Vehicle, Car, etc.

csharp
CopyEdit
```csharp
namespace YourApp.Entities
{
    public class Vehicle
    {
        public int VehicleId { get; set; }
        public string Make { get; set; }
        // other properties...
    }
}
```

- **Data Folder**:
  Contains ApplicationDbContext which is responsible for database interaction.

csharp
CopyEdit
```csharp
namespace YourApp.Data
{
    public class ApplicationDbContext : DbContext
    {
        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
            : base(options)
        { }

        public DbSet<Vehicle> Vehicles { get; set; }
        // other DbSets...
    }
}
```
MSTest Project (YourApp.Tests)

- **Controllers Tests**:
  You can write unit tests for your Web API controllers in this folder.

csharp
CopyEdit
```csharp
using Microsoft.VisualStudio.TestTools.UnitTesting;
using YourApp.WebAPI.Controllers;
using YourApp.WebAPI.Data;
using Microsoft.Extensions.DependencyInjection;

[TestClass]
public class CarControllerTests
```

```
{
    private ServiceProvider _serviceProvider;
    private ApplicationDbContext _context;

    [TestInitialize]
    public void Setup()
    {
        var services = new ServiceCollection();
        services.AddDbContext<ApplicationDbContext>(options =>
            options.UseInMemoryDatabase("TestDb"));  // Using an in-memory DB for testing
        _serviceProvider = services.BuildServiceProvider();
        _context = _serviceProvider.GetRequiredService<ApplicationDbContext>();
    }

    [TestMethod]
    public void TestAddCar()
    {
        var controller = new CarController(_context);
        var result = controller.PostCar(new Car { Make = "Toyota", Model = "Corolla" });
        Assert.IsNotNull(result);
    }
}
```

In Summary:

- **EF Core** should be installed and used in the **Web API project** where the database context (ApplicationDbContext) is defined and migrations are applied.
- The **MSTest project** should reference the Web API project to test its functionality, but you don't need to install EF Core in the MSTest project.
- **MSTest** should focus on testing business logic, API routes, and database operations via integration tests, potentially using an in-memory database or mocking the DbContext as needed.

This structure allows for clean separation of concerns, making your solution easy to maintain and test.
Let me know if you need any more clarification or examples!
Get smarter responses, upload files and images, and more.
Log in
Sign up for free

Search
Reason
Voice
ChatGPT can make mistakes. Check important info

REVISED ENTITY FRAMEWORK CORE CAR (standing in for ResourceBook) TABLE GENERATION

You said:
Can you show me ALL the necessary code, when using EF Core, needed in various folders, to implement a Car class Entity having it's own separate PK, that is a child of a Vehicle class, but with some unique columns, and implemented using Fluent API? Those Entities will be used in a code-first migration to create tables in SQL Server. This might be an Entity used in a .NET Core Web API application.

ChatGPT said:
Sure! Let's go through all the necessary code and files required to implement a **Car class entity** that has its **own separate primary key** (CarId), while still being a **child of a Vehicle class** (sharing some common properties), but with some **unique columns**. We'll implement this using **Fluent API** and follow the **Code-First migration** approach with EF Core.

This will be a simple setup for a **.NET Core Web API project** with **Entity Framework Core**.

Assumptions:

- Vehicle class is the **parent** class with some common properties.
- Car class is the **child** of Vehicle, with its own primary key (CarId), but shares some properties with Vehicle.
- We are using **Fluent API** for configuration in ApplicationDbContext.
- The project follows a **Code-First** approach (i.e., the database schema is generated from the C# models).

Solution Structure:

graphql

CopyEdit

/YourSolution

```
│
├── /YourApp.WebAPI             # .NET Core Web API Project
│   ├── /Controllers            # API Controllers
│   ├── /Entities              # Entity classes (Vehicle and Car)
│   ├── /Data                  # ApplicationDbContext
│   ├── /Migrations            # EF Core migration files
│   ├── /Models                # DTOs (Optional)
│   ├── /Services              # Business logic (Optional)
│   ├── appsettings.json       # Configuration (Connection strings, etc.)
│   ├── Program.cs             # Entry point & app configuration
│   └── Startup.cs             # Configures services and middleware
│
├── /YourApp.Tests             # MSTest Project for unit tests
│   └── ...
│
├── YourApp.sln                # Solution file
└── README.md                  # Project documentation
```

Code Implementation

Let's go step by step:

1. Vehicle Entity (Parent Entity)

The Vehicle class will be the base class for all vehicle types, such as Car. This class contains common properties like Make, Model, Year, and Type.

csharp

CopyEdit

```
namespace YourApp.Entities
{
    public class Vehicle
    {
        public int VehicleId { get; set; }  // Primary Key for Vehicle
        public string Make { get; set; }
        public string Model { get; set; }
        public int Year { get; set; }
        public string Type { get; set; }  // Used to distinguish between different vehicle types (e.g., "Car", "Truck")
    }
}
```

2. Car Entity (Child Entity with Separate Primary Key)

The Car class will have its own primary key (CarId) and will also have a foreign key (VehicleId) linking it to the Vehicle table. It will also have properties specific to cars, such as Doors, Transmission, and FuelType.

csharp

CopyEdit

```
namespace YourApp.Entities
{
    public class Car
    {
        public int CarId { get; set; }  // Primary Key for Car (separate from Vehicle)
        public int VehicleId { get; set; }  // Foreign Key pointing to Vehicle
        public Vehicle Vehicle { get; set; }  // Navigation Property to Vehicle

        public int Doors { get; set; }
        public string Transmission { get; set; }
        public string FuelType { get; set; }
    }
}
```

3. ApplicationDbContext (with Fluent API Configuration)

The ApplicationDbContext is where we will define the relationships between the Vehicle and Car entities, and configure the primary key, foreign key, and other properties using Fluent API.

csharp

CopyEdit

```
using Microsoft.EntityFrameworkCore;
using YourApp.Entities;

namespace YourApp.Data
{
    public class ApplicationDbContext : DbContext
    {
        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
            : base(options)
        { }

        public DbSet<Vehicle> Vehicles { get; set; }
        public DbSet<Car> Cars { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            // Configure Vehicle entity
            modelBuilder.Entity<Vehicle>()
                .HasKey(v => v.VehicleId);  // VehicleId as primary key

            modelBuilder.Entity<Vehicle>()
                .Property(v => v.Make)
                .IsRequired()
                .HasMaxLength(50);
            modelBuilder.Entity<Vehicle>()
                .Property(v => v.Model)
                .IsRequired()
                .HasMaxLength(50);

            modelBuilder.Entity<Vehicle>()
                .Property(v => v.Year)
                .IsRequired();

            modelBuilder.Entity<Vehicle>()
                .Property(v => v.Type)
                .IsRequired()
                .HasMaxLength(20);

            // Configure Car entity
            modelBuilder.Entity<Car>()
                .HasKey(c => c.CarId);  // CarId as primary key (separate from VehicleId)

            modelBuilder.Entity<Car>()
                .HasOne(c => c.Vehicle)  // Each Car is related to one Vehicle
                .WithOne()  // One-to-one relationship
                .HasForeignKey<Car>(c => c.VehicleId)  // VehicleId as foreign key in Car
                .OnDelete(DeleteBehavior.Cascade);  // Cascade delete behavior

            modelBuilder.Entity<Car>()
                .Property(c => c.Doors)
                .IsRequired();

            modelBuilder.Entity<Car>()
                .Property(c => c.Transmission)
                .HasMaxLength(20);

            modelBuilder.Entity<Car>()
                .Property(c => c.FuelType)
                .HasMaxLength(20);
        }
    }
}
```

4. Program.cs and Startup.cs (Configure Services)

In **Program.cs** (or **Startup.cs** if you are using an older version of .NET), you'll need to configure the dependency injection for ApplicationDbContext to use SQL Server.

Program.cs (for .NET 6 and above)

csharp

CopyEdit

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Hosting;

namespace YourApp.WebAPI
{
  public class Program
  {
        public static void Main(string[] args)
        {
            CreateHostBuilder(args).Build().Run();
        }

        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                });
    }
}
```

Startup.cs

In Startup.cs, you need to configure services for EF Core and specify the connection string for the database.

csharp

CopyEdit

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;

namespace YourApp.WebAPI
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            // Add DBContext with SQL Server connection string
            services.AddDbContext<ApplicationDbContext>(options =>
                options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

            // Add controllers and other services
            services.AddControllers();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            app.UseHttpsRedirection();
            app.UseRouting();
            app.UseAuthorization();
            app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllers();
            });
        }
    }
}
```

In **appsettings.json**, make sure to define your connection string:

json
CopyEdit

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=YourAppDb;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

5. Migration and Database Update

To create the migration and update the database schema, run the following commands in the **Package Manager Console** or terminal:

1. **Add a new migration**:

bash
CopyEdit

```
dotnet ef migrations add InitialCreate
```

2. **Apply the migration to the database**:

bash
CopyEdit

```
dotnet ef database update
```

This will create the tables for both Vehicle and Car in the SQL Server database, with CarId as the primary key for Car, and VehicleId as a foreign key in the Car table.

---

6. Example API Controller to Use Entities

Let’s create a simple **CarController** to handle HTTP requests for adding cars.

csharp
CopyEdit

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using YourApp.Data;
using YourApp.Entities;
```

```
namespace YourApp.WebAPI.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class CarController : ControllerBase
    {
        private readonly ApplicationDbContext _context;

        public CarController(ApplicationDbContext context)
        {
            _context = context;
        }

        // POST: api/car
        [HttpPost]
        public async Task<ActionResult<Car>> PostCar(Car car)
        {
            _context.Cars.Add(car);
            await _context.SaveChangesAsync();

            return CreatedAtAction("GetCar", new { id = car.CarId }, car);
        }
    }
}
```

Summary of Steps:

1. **Vehicle entity**: Contains common properties for vehicles.
2. **Car entity**: Has its own primary key (CarId), and a foreign key (VehicleId) linking to Vehicle.
3. **ApplicationDbContext**: Configures the relationship using Fluent API with one-to-one mapping and cascade delete behavior.
4. **Program.cs/Startup.cs**: Configures dependency injection for ApplicationDbContext to use SQL Server.
5. **Migrations**: Generate and apply migrations to update the database schema.
6. **API Controller**: Example of a simple API controller for adding a Car entity.

This setup provides a clean **code-first** migration approach with EF Core in a **.NET Core Web API** application. Let me know if you need any further explanation!

Addendum Info for DateOnly and Double fields in the Child Table (in my application: ResourceBook)

if Car also had a DateOnly field (no time portion) called YearOfCar, how would that be configured for Car in the ApplicationDbContext file?

ChatGPT said:

Great question! If the Car entity includes a **DateOnly** field (with no time portion), you can configure it in EF Core just like any other property, but with some specific attention to the **DateOnly** type.

Starting from **EF Core 7**, Microsoft added support for DateOnly and TimeOnly types. If you are using EF Core 7 or later, EF Core can handle DateOnly natively without any additional configuration for basic use cases.

If you're using an earlier version of EF Core (such as EF Core 6), **EF Core doesn't natively support DateOnly** yet, and you would need to store it as a DateTime and configure it accordingly.

I'll show you both scenarios.

Scenario 1: EF Core 7 or Later (Native Support for DateOnly)

If you're using **EF Core 7 or later**, the DateOnly type can be mapped directly to SQL Server’s DATE type.

Updated Car Entity with YearOfCar

csharp

CopyEdit

```
namespace YourApp.Entities
{
    public class Car
    {
        public int CarId { get; set; }  // Primary Key
        public int VehicleId { get; set; }  // Foreign Key
        public Vehicle Vehicle { get; set; }  // Navigation Property

        public int Doors { get; set; }
        public string Transmission { get; set; }
        public string FuelType { get; set; }

        public DateOnly YearOfCar { get; set; }  // New DateOnly field
    }
}
```

Configuration in ApplicationDbContext

You do not need additional configuration for DateOnly in EF Core 7 if you just want to store it as a DATE in SQL Server. EF Core automatically maps DateOnly to DATE in the database.

csharp

CopyEdit

```
using Microsoft.EntityFrameworkCore;
using YourApp.Entities;

namespace YourApp.Data
{
    public class ApplicationDbContext : DbContext
    {
        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
            : base(options)
        { }

        public DbSet<Vehicle> Vehicles { get; set; }
        public DbSet<Car> Cars { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
```

```
{
  // Vehicle Configuration
  modelBuilder.Entity<Vehicle>()
    .HasKey(v => v.VehicleId);

  modelBuilder.Entity<Vehicle>()
    .Property(v => v.Make)
    .IsRequired()
    .HasMaxLength(50);

  modelBuilder.Entity<Vehicle>()
    .Property(v => v.Model)
    .IsRequired()
    .HasMaxLength(50);

  modelBuilder.Entity<Vehicle>()
    .Property(v => v.Year)
    .IsRequired();

  modelBuilder.Entity<Vehicle>()
    .Property(v => v.Type)
    .IsRequired()
    .HasMaxLength(20);

  // Car Configuration
  modelBuilder.Entity<Car>()
    .HasKey(c => c.CarId);

  modelBuilder.Entity<Car>()
    .HasOne(c => c.Vehicle)
    .WithOne()
    .HasForeignKey<Car>(c => c.VehicleId)
    .OnDelete(DeleteBehavior.Cascade);

  modelBuilder.Entity<Car>()
    .Property(c => c.Doors)
    .IsRequired();

  modelBuilder.Entity<Car>()
    .Property(c => c.Transmission)
    .HasMaxLength(20);

  modelBuilder.Entity<Car>()
    .Property(c => c.FuelType)
    .HasMaxLength(20);

  // Configure the DateOnly property as a DATE in the DB (this is handled automatically in EF Core 7)
  modelBuilder.Entity<Car>()
```

```
                .Property(c => c.YearOfCar)
                .IsRequired();  // If you want YearOfCar to be required, set this
        }
    }
}
```

Scenario 2: EF Core 6 or Earlier (No Native Support for DateOnly)

If you're using **EF Core 6 or earlier**, **EF Core does not natively support DateOnly**, so you'll need to map the DateOnly type to DateTime (or DateTimeOffset if time zone considerations are needed) for storage, and then use a value conversion to strip out the time portion.

Updated Car Entity with YearOfCar

csharp

CopyEdit

```
namespace YourApp.Entities
{
    public class Car
    {
        public int CarId { get; set; }
        public int VehicleId { get; set; }
        public Vehicle Vehicle { get; set; }

        public int Doors { get; set; }
        public string Transmission { get; set; }
        public string FuelType { get; set; }

        public DateOnly YearOfCar { get; set; }  // DateOnly field
    }
}
```

Configuration in ApplicationDbContext (for EF Core 6 or earlier)

In this case, you can use **Value Conversions** to convert the DateOnly type to DateTime when storing in the database, and convert it back when reading from the database.

csharp

CopyEdit

```
using Microsoft.EntityFrameworkCore;
using YourApp.Entities;
using Microsoft.EntityFrameworkCore.Storage.ValueConversion;

namespace YourApp.Data
{
    public class ApplicationDbContext : DbContext
    {
        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
            : base(options)
        { }

        public DbSet<Vehicle> Vehicles { get; set; }
        public DbSet<Car> Cars { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            // Vehicle Configuration
```

```
modelBuilder.Entity<Vehicle>()
   .HasKey(v => v.VehicleId);

modelBuilder.Entity<Vehicle>()
   .Property(v => v.Make)
   .IsRequired()
   .HasMaxLength(50);

modelBuilder.Entity<Vehicle>()
   .Property(v => v.Model)
   .IsRequired()
   .HasMaxLength(50);

modelBuilder.Entity<Vehicle>()
   .Property(v => v.Year)
   .IsRequired();

modelBuilder.Entity<Vehicle>()
   .Property(v => v.Type)
   .IsRequired()
   .HasMaxLength(20);

// Car Configuration
modelBuilder.Entity<Car>()
   .HasKey(c => c.CarId);

modelBuilder.Entity<Car>()
   .HasOne(c => c.Vehicle)
   .WithOne()
   .HasForeignKey<Car>(c => c.VehicleId)
   .OnDelete(DeleteBehavior.Cascade);

modelBuilder.Entity<Car>()
   .Property(c => c.Doors)
   .IsRequired();

modelBuilder.Entity<Car>()
   .Property(c => c.Transmission)
   .HasMaxLength(20);

modelBuilder.Entity<Car>()
   .Property(c => c.FuelType)
   .HasMaxLength(20);

// Configure DateOnly using value conversion to store as DateTime
var dateOnlyConverter = new ValueConverter<DateOnly, DateTime>(
   v => v.ToDateTime(new TimeOnly(0, 0)),  // Convert DateOnly to DateTime
   v => DateOnly.FromDateTime(v)         // Convert DateTime back to DateOnly
```

```
        );

        modelBuilder.Entity<Car>()
            .Property(c => c.YearOfCar)
            .HasConversion(dateOnlyConverter)
            .IsRequired();
    }
  }
}
```

Key Details:

- **EF Core 7+**: The DateOnly type can be used directly, and EF Core will automatically map it to the SQL Server DATE type.
- **EF Core 6-**: Since EF Core doesn't natively support DateOnly, you have to manually configure a **value converter** to convert DateOnly to DateTime for storage, and convert it back when reading from the database.

Example Migration Command (for EF Core 7+)

After making these changes, generate a migration and apply it to the database:

bash

CopyEdit

```
dotnet ef migrations add AddYearOfCar
dotnet ef database update
```

Summary:

- **EF Core 7 and later**: You can use DateOnly directly, and EF Core will map it to the SQL DATE type.
- **EF Core 6 or earlier**: You need to use a **value converter** to convert between DateOnly and DateTime.

Let me know if you need any further clarifications!

You said:

If it's not required is the .IsRequired(); left out, or is another attribute needed to make it optional

ChatGPT said:

If the field is **not required**, you **don't need to use** the .IsRequired() method in the **Fluent API**. By default, in EF Core, properties that are **nullable** types (like DateOnly? or DateTime?) are treated as optional. The .IsRequired() method specifically marks a property as **mandatory**, meaning EF Core will treat it as **non-nullable**.

Here's a breakdown of how to handle optional fields in EF Core:

1. Optional DateOnly Field

If YearOfCar is **optional** and you want it to be nullable in the database, you should change the type to DateOnly? (nullable DateOnly), and **no .IsRequired() method is needed**.

Updated Car Entity with Optional YearOfCar

csharp

CopyEdit

```
namespace YourApp.Entities
{
  public class Car
  {
    public int CarId { get; set; }  // Primary Key
    public int VehicleId { get; set; }  // Foreign Key
    public Vehicle Vehicle { get; set; }  // Navigation Property

    public int Doors { get; set; }
    public string Transmission { get; set; }
    public string FuelType { get; set; }

    public DateOnly? YearOfCar { get; set; }  // Nullable DateOnly field (optional)
  }
}
```

Updated ApplicationDbContext with No .IsRequired()

For **EF Core 7 or later** (natively supports DateOnly), you don't need .IsRequired() for the optional YearOfCar. The property will be optional by default, as DateOnly? (nullable type) is treated as optional.

csharp

CopyEdit

```
using Microsoft.EntityFrameworkCore;
using YourApp.Entities;

namespace YourApp.Data
{
    public class ApplicationDbContext : DbContext
    {
        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
            : base(options)
        { }

        public DbSet<Vehicle> Vehicles { get; set; }
        public DbSet<Car> Cars { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            // Vehicle Configuration
            modelBuilder.Entity<Vehicle>()
                .HasKey(v => v.VehicleId);

            // Car Configuration
            modelBuilder.Entity<Car>()
                .HasKey(c => c.CarId);

            modelBuilder.Entity<Car>()
                .HasOne(c => c.Vehicle)
                .WithOne()
                .HasForeignKey<Car>(c => c.VehicleId)
                .OnDelete(DeleteBehavior.Cascade);

            modelBuilder.Entity<Car>()
                .Property(c => c.Doors)
                .IsRequired();

            modelBuilder.Entity<Car>()
                .Property(c => c.Transmission)
                .HasMaxLength(20);

            modelBuilder.Entity<Car>()
                .Property(c => c.FuelType)
                .HasMaxLength(20);

            // YearOfCar is optional, so no .IsRequired() is needed for nullable DateOnly
            modelBuilder.Entity<Car>()
            .Property(c => c.YearOfCar)
            .HasConversion(v => v.ToDateTime(new TimeOnly(0, 0)), v => DateOnly.FromDateTime(v));  // For EF Core 6 or earlier, use conversion
        }
    }
}
```

2. Non-nullable DateOnly Field

If YearOfCar should **always have a value**, but you want to ensure it's **not nullable**, you would leave it as a **non-nullable DateOnly** (DateOnly without a ?), and use .IsRequired() to enforce the field to always be provided.

Updated Car Entity with Non-nullable YearOfCar

csharp

CopyEdit

```
namespace YourApp.Entities
{
    public class Car
    {
        public int CarId { get; set; }
        public int VehicleId { get; set; }
        public Vehicle Vehicle { get; set; }

        public int Doors { get; set; }
        public string Transmission { get; set; }
        public string FuelType { get; set; }

        public DateOnly YearOfCar { get; set; }  // Non-nullable DateOnly field (required)
    }
}
```

Updated ApplicationDbContext with .IsRequired()

For **non-nullable DateOnly** fields, if you want it **required**, you can explicitly use .IsRequired(), but EF Core will infer that it's required if it's not nullable (i.e., DateOnly instead of DateOnly?).

csharp

CopyEdit

```
modelBuilder.Entity<Car>()
    .Property(c => c.YearOfCar)
    .IsRequired()  // Explicitly marks as required if it's a non-nullable type
    .HasConversion(v => v.ToDateTime(new TimeOnly(0, 0)), v => DateOnly.FromDateTime(v));  // For EF Core 6 or earlier, use conversion
```

Key Points:

1. **Optional Field** (DateOnly?):
    - You **do not need** .IsRequired() for optional fields.
    - Simply use DateOnly? (nullable) in your model, and EF Core will handle it as optional by default.
2. **Required Field** (DateOnly):
    - Use .IsRequired() to mark it as required (for non-nullable types).
    - For EF Core 6 or earlier, **value conversion** may be required for DateOnly.

When to use .IsRequired():

- **Use .IsRequired()** if you want to explicitly enforce that a field **cannot be null**.
- **Omit .IsRequired()** if the field should be **optional** (nullable).

Let me know if you need further clarification!

You said:

if a type is double rather than int, does it make a difference in ApplicationDbContext

ChatGPT said:

Yes, using a double type instead of int in your model class does make a difference in how it is handled in **Entity Framework Core** and in your **ApplicationDbContext** configuration. The difference comes in how the data type is mapped to the database and the kinds of validation that apply to the property (e.g., whether it can have decimals, nullability, precision, etc.).

Key Differences Between int and double:

1. **Data Type in the Database**:
    - An **int** property maps to the **INT** type in SQL Server, which is used for whole numbers (no decimals).
    - A **double** property maps to the **FLOAT** or **REAL** type in SQL Server (depending on the precision you choose), which allows for decimal values (floating-point numbers).
2. **Precision**:
    - **int** is a 32-bit signed integer with a range of values from **-2,147,483,648** to **2,147,483,647**.
    - **double** is a floating-point number with much larger precision and can represent both whole numbers and decimals.
3. **Nullability**:
    - **int** is a non-nullable type by default unless you use int? to make it nullable.
    - **double** is non-nullable unless you explicitly declare it as double? to make it nullable.
4. **EF Core Configuration**:
    - If you're using a double, EF Core will map it to a **numeric** type (e.g., FLOAT in SQL Server).
    - If you're using an int, EF Core will map it to an **integer** type (INT in SQL Server).

How Does This Affect ApplicationDbContext?

For most cases, EF Core handles double and int types in the model automatically without requiring any special configuration. However, if you want to control the precision, scale, or nullability of these types, you can specify it in the **Fluent API** in the OnModelCreating method.

Scenario 1: Using double Instead of int

Let's say you have a Price property of type double instead of int in your Car class.

Updated Car Entity (using double)

csharp

CopyEdit

```
namespace YourApp.Entities
{
  public class Car
  {
    public int CarId { get; set; }
    public int VehicleId { get; set; }
    public Vehicle Vehicle { get; set; }

    public int Doors { get; set; }
    public string Transmission { get; set; }
    public string FuelType { get; set; }

    public double Price { get; set; }  // Now using double instead of int
  }
}
```

ApplicationDbContext Configuration (For double Property)

If you want to configure the Price property to have a specific **precision and scale** (for example, two decimal places), you can do that using the Fluent API in OnModelCreating.

Here's an example:

csharp

CopyEdit

```
using Microsoft.EntityFrameworkCore;
using YourApp.Entities;

namespace YourApp.Data
{
  public class ApplicationDbContext : DbContext
  {
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
      : base(options)
    { }

    public DbSet<Vehicle> Vehicles { get; set; }
    public DbSet<Car> Cars { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
      // Configure Vehicle Entity
      modelBuilder.Entity<Vehicle>()
        .HasKey(v => v.VehicleId);

      // Configure Car Entity
      modelBuilder.Entity<Car>()
        .HasKey(c => c.CarId);

      modelBuilder.Entity<Car>()
        .HasOne(c => c.Vehicle)
        .WithOne()
        .HasForeignKey<Car>(c => c.VehicleId)
        .OnDelete(DeleteBehavior.Cascade);

      modelBuilder.Entity<Car>()
        .Property(c => c.Doors)
        .IsRequired();

      modelBuilder.Entity<Car>()
        .Property(c => c.Transmission)
        .HasMaxLength(20);

      modelBuilder.Entity<Car>()
        .Property(c => c.FuelType)
        .HasMaxLength(20);

      // Configure the double property for Price (Precision and Scale)
      modelBuilder.Entity<Car>()
        .Property(c => c.Price)
        .HasColumnType("decimal(18,2)");  // This will store the Price as a decimal with 18 digits and 2 decimal places
    }
  }
}
```

Important Points:

- **Precision and Scale**:
  - If you want to control how many **total digits** and **decimal places** the double (or decimal) can have, use HasColumnType("decimal(18,2)"). In the example above, we specify 18 total digits, and 2 decimal places.
  - If you don’t set this, EF Core will default to mapping double to FLOAT or REAL in SQL Server, which may not have the level of precision you need.

- **Default Mapping**:
    - EF Core will automatically map double to a FLOAT in SQL Server by default, but you may want to use decimal for monetary values because decimal has better precision for financial calculations (this can be important when dealing with prices or quantities).
- **Nullability**:
    - If you declare Price as double?, it will be treated as optional in the database, meaning the column can accept NULL values. If it's just double, EF Core will assume it's required by default (unless you explicitly specify .IsRequired() or make it nullable).

Scenario 2: If You Want to Make Price Optional

If the Price property can be **optional** (nullable), you would declare it as double? in the model class and it will automatically be treated as nullable in the database.

Updated Car Entity (with Nullable double?)

csharp
CopyEdit

```
namespace YourApp.Entities
{
   public class Car
   {
      public int CarId { get; set; }
      public int VehicleId { get; set; }
      public Vehicle Vehicle { get; set; }

      public int Doors { get; set; }
      public string Transmission { get; set; }
      public string FuelType { get; set; }

      public double? Price { get; set; }  // Nullable double (optional)
   }
}
```

In this case, you don't need to explicitly call .IsRequired() in OnModelCreating because the nullable double? will be mapped to a SQL Server column that allows NULL values by default.

Summary:

1. **Using double instead of int**:
    - The double type will be mapped to FLOAT or REAL in SQL Server, whereas int will be mapped to INT.
    - You can use .HasColumnType("decimal(18,2)") to specify precision for double if you need exact decimal places.
2. **Nullable vs. Non-Nullable**:
    - For nullable types (double?), EF Core will allow NULL in the database.
    - For non-nullable types (double), EF Core will make sure that the field is always populated.
3. **Precision and Scale**:
    - Use .HasColumnType("decimal(18,2)") to define the **precision** (total number of digits) and **scale** (number of decimal places) for double or decimal.

Let me know if you have further questions on this!

Search
Reason
Voice
ChatGPT can make mistakes. Check important info. See Cookie Pr

APPENDIX C: MISCELLANEOUS SQL CODE FOR FIXES/ADJUSTMENTS

ADD COLUMN statement to just add a column (that is nullable)

```
ALTER TABLE book
ADD PAGES_READ INT NULL;
```

ALTER TABLE statement to change a column's nullability

```
ALTER TABLE your_table
ALTER COLUMN your_column NVARCHAR(42) NULL
```

ALTER TABLE: Change a column's datatype to decimal with 4 numbers before the decimal place and 2 after

```
Numbers must be decimal datatypes instead of int in order to do division on them.
ALTER TABLE book
ALTER COLUMN pages_read DECIMAL(6,2)
```