EF MIGRATIONS

AGENDA

- Migration The why, what, where
  - First migration
  - Update database
  - Rolling back migrations
- Query
  - Loading strategies
  - Tracing
  - Create / Update

SQLITE VS SQL SERVER

- SQL Server localDb
  - Development database,
  - Almost same set of functions as SQL Server
- SQL Server
  - Full weight database
  - Connections string are non-trivial
  - -> So from now and onwards we will use this
- SQLite is very lightweight
  - Can not remove columns only add
  - Mostly used for development / tests

WHAT IS MIGRATIONS

1. Technic EfCore uses to track of changes in database schema
2. Files that EfCore uses to create schema

WHY MIGRATIONS

- It's not a feasible to delete database every time we make a change
  - Keeping Development / Production environment in sync
- Avoid making changes by hand

BENEFITS OF MIGRATIONS

- Files generated by EfCore based on your OOP models
- Keeps OOP model and database tables in sync
- Edit database schema without losing data (development and production)
- Provide a way to make rollbacks on database (same as in VCS)
- Version control for database

DRAWBACK OF MIGRATIONS

- Harder to make merges in larger teams. Migrations files should be handled especially carefully

WHERE DO MIGRATIONS LIVE (1/2)

Lives in: /Migrations/"

Migrations file:

```
public partial class AddContactPhoneNumber : Migration {
    protected override void Up(MigrationBuilder mb) {
        mb.AddColumn<string>(
            name: "PhoneNumber",
            table: "Contacts",
            nullable: true);
    }

    protected override void Down(MigrationBuilder mb) {
        mb.DropColumn(
            name: "PhoneNumber",
            table: "Contacts");
    }}
```

WHERE DO MIGRATIONS LIVE (2/2)

ModelSnapshot.cs:

```
protected override void BuildModel(ModelBuilder mb) {
  #pragma warning disable 612, 618
  mb.HasAnnotation("ProductVersion", "2.2.0-rtm-35687");
  mb.Entity("MyFirstEfCoreApp.Models.Contact", b => {
          b.Property<int>("Id")
              .ValueGeneratedOnAdd();
          b.Property<string>("Email");
          b.Property<string>("FirstName");
          b.Property<string>("LastName");
          b.Property<string>("PhoneNumber");
          b.HasKey("Id");
          b.ToTable("Contacts");
      });
      ...}
```

CREATE MIGRATION

- In Visual Studio (open Package Manager Console)

```
PM> Add-Migration <MigrationName>
```

- .Net Core cli

```
$ dotnet ef migrations add <MigrationName>
```

- Creates a .cs file with timestamp and name of migration + plus creates/updates Snapshot.cs file in Migrations folder.

UPDATE DATABASE

- In Visual Studio (open Package Manager Console)

```
PM> Update-Database
```

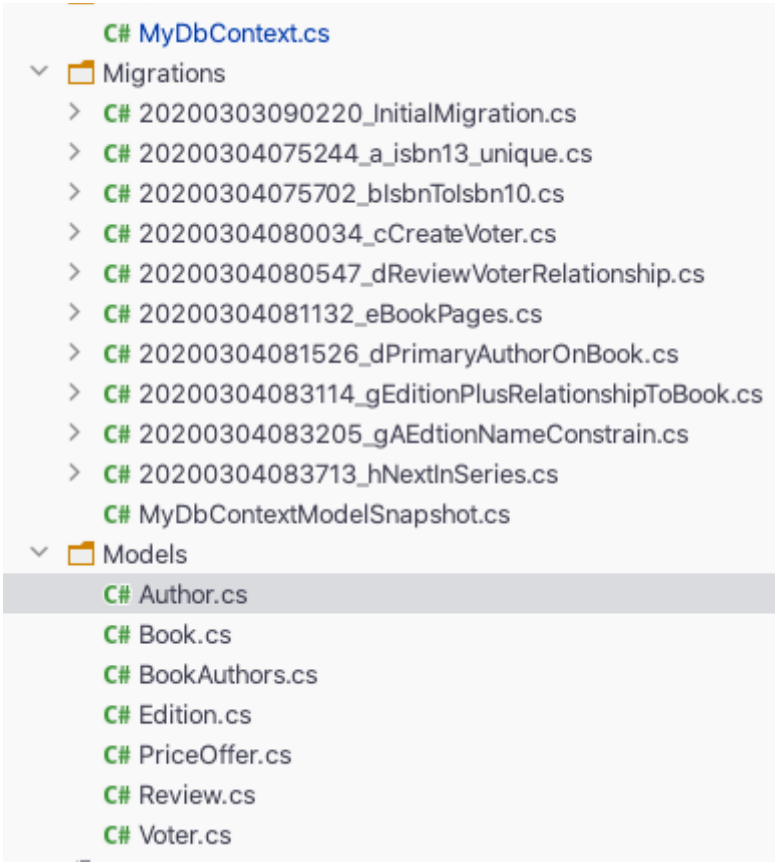
- .Net Core cli

```
$ dotnet ef database update
```

- After this the migration(s) is **applied**

Hint: Apply each migrations instead of mass applying a bunch of migrations

MIGRATIONS


C# MyDbContext.cs
Migrations
C# 20200303090220_InitialMigration.cs
C# 20200304075244_a_isbn13_unique.cs
C# 20200304075702_bIsbnToIsbn10.cs
C# 20200304080034_cCreateVoter.cs
C# 20200304080547_dReviewVoterRelationship.cs
C# 20200304081132_eBookPages.cs
C# 20200304081526_dPrimaryAuthorOnBook.cs
C# 20200304083114_gEditionPlusRelationshipToBook.cs
C# 20200304083205_gAEdtionNameConstrain.cs
C# 20200304083713_hNextInSeries.cs
C# MyDbContextModelSnapshot.cs
Models
C# Author.cs
C# Book.cs
C# BookAuthors.cs
C# Edition.cs
C# PriceOffer.cs
C# Review.cs
C# Voter.cs

DEMO

ROLLBACK MIGRATIONS (UNDO) - IN UNAPPLIED STATE

- In Visual Studio (open Package Manager Console)

```
PM> Remove-Migration <MigrationsName>
```

- .Net Core cli

```
$ dotnet ef migrations remove <MigrationName>
```

ROLLBACK MIGRATIONS (UNDO) - IN APPLIED STATE

- In Visual Studio (open Package Manager Console)

```
PM> Update-Database <MigrationName-1>
PM> Remove-Migration <MigrationsName>
```

- .Net Core cli

```
$ dotnet ef database update <MigrationName-1>
$ dotnet ef migrations remove <MigrationName>
```

WHAT HAPPENS

1. If migrations is applied
    - Database will execute Down
2. Remove migrations
    - Deletes the migrations file
3. Then you can change your models

QUERYING

- Access via DbContext

```
1 _context.Books.Where(b =>
2     b.Title.StartsWith("Database")
3     .ToList();
```

vs

```
from b in _context.Books
where b.Title == "Database"
select b;
```

Note: Requires Linq and EntityFrameworkCore imports

QUERYING

- Access via DbContext

```
1 _context.Books.Where(b =>
2     b.Title.StartsWith("Database")
3     .ToList();
```

vs

```
from b in _context.Books
where b.Title == "Database"
select b;
```

Note: Requires Linq and EntityFrameworkCore imports

QUERYING

- Access via DbContext

```
1 _context.Books.Where(b =>
2     b.Title.StartsWith("Database")
3     .ToList();
```

vs

```
from b in _context.Books
where b.Title == "Database"
select b;
```

Note: Requires Linq and EntityFrameworkCore imports

Speaker notes

1. DbContext property accces

2. A series of LINQ and/or EF core commands

3. An execute command

EXECUTE COMMANDS

- `.ToList()`
- `.ToArray()`
- `.Count()`
- …

ASYNC EXECUTION

- Ends with `Async()`
  - E.g. `.ToListAsync()`
- Exists in EntityFrameworkCore namespace - remember to use `using`
  - Exists as extensions methods
- Can not execute queries in parallel on same DbContext
  - Will not block callers thread

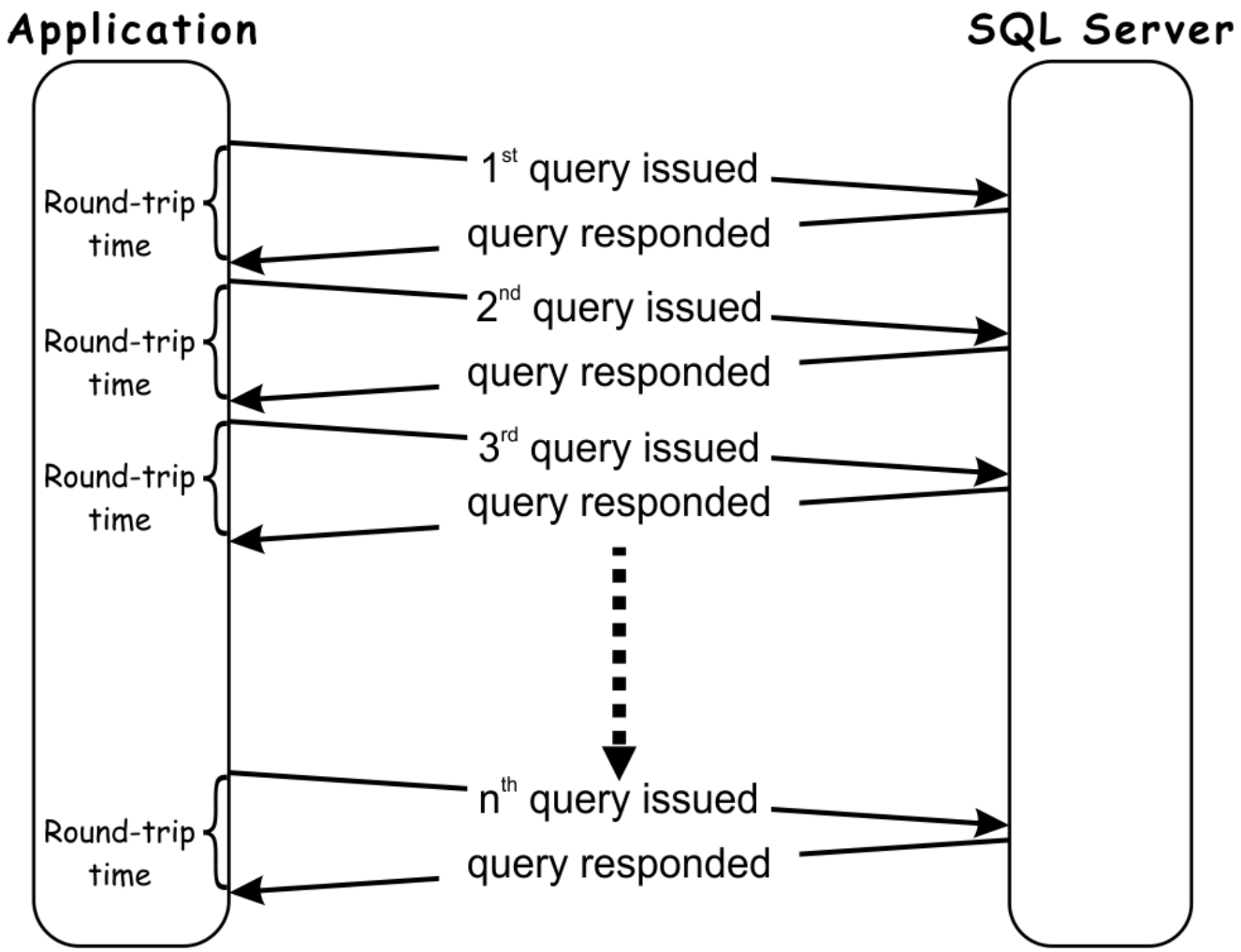
ASYNC IN C#

- Returns a Task<A>
- Use async and await - we will come to that in SWD

```
var blog = new Blog { Url = "http://sample.com" };
context.Blogs.Add(blog);
await context.SaveChangesAsync();

var blogs = await context.Blogs.Where(b => b.Rating > 3).ToLis
```

ROUND-TRIPS


Application
SQL Server
1st query issued
query responded
2nd query issued
query responded
3rd query issued
query responded
nth query issued
query responded
Round-trip time
Round-trip time
Round-trip time
Round-trip time

LOADING STRATEGIES - EXPLICIT

```
public class AClass {
  public async Task<IEnumerable<Book>> LoadExplicit() {
    var books = await _context.Books.ToListAsync();
    foreach(var book in books) {
      await _context.Entry(book).Reference(b => b.Author)
                .LoadAsync();
    }
    return books;
  } }
```

- + Load relationship when needed
- % More database round-trips
- Usage:
  - e.g. when library only returns primary entity
  - Data only used in some circumstances, so we only load needed data

Speaker notes

```
public class Book {
    ...
    public Author Author {get; set;} //  Navigational property
}
```

LOADING STRATEGIES - EAGER

```
public class AClass {
  public IEnumerable<Book> LoadEager() {
    var books = _context.Books
      .Include(b => b.Author)
      .Include(b => b.Review)
      .ToList();
    return books;
  } }
```

- + Loaded by EF Core efficiently with a minimum of round-trips
- % Load all data, even when not needed
- If relationship does not exists, EF does not fail
- Since 3.0 this uses JOIN extensively - Be AWARE

Speaker notes

```
public class Book {
    ...
    public Author Author {get; set;} //  Navigational property
    public Review Review {get; set;} //  Navigational property
}
```

LOADING STRATEGIES - MULTIPLE LEVELS

```
public class AClass {
  public IEnumerable<Book> LoadMultipleLevels() {
    var books = _context.Books
      .Include(b => b.Author)
      .Include(b => b.Review)
        .ThenInclude(r => r.Voter)
      .ToList();
    return books;
  } }
```

- ThenInclude can be chained

Speaker notes

```
public class Book {
    ...
    public Author Author {get; set;} //  Navigational property
    public Review Review {get; set;} //  Navigational property
}



public class Review {
    public Voter Voter {get; set;} //  Navigational property
}
```

LOADING STRATEGIES - SELECT

```
public class AClass {
  public object LoadSelect() {
    return _context.Books
      .Select(b => new {
        b.Title,
        b.Isbn,
        NumReview = b.Reviews.Count
      });
  } }
```

- +Load specifically the data needed, including database calculations
- % Have to write each query by hand

Note: Includes are ignored when returning instances which are not an entity type

Speaker notes

Use LINQ to create anonymous objects with specific data

```
public class Book {
    ...
    public string Title {get;set;}
    public int ISBN {get;set;}
    public List<Review> Reviews {get; set;} //  Navigational property
}
```

LOADING STRATEGIES - LAZY (1/2)

1. Install NugetPackage 'Microsoft.EntityFrameworkCore.Proxies'
2. a. Enable proxies in DbContext

```
public class Context : DbContext {
  protected override void OnConfiguring(
                      DbContextOptionsBuilder optionsBuilder)
    => optionsBuilder
        .UseLazyLoadingProxies()
        .UseSqlServer(myConnectionString);
}
```

2. b. Or by injecting LazyLoader into service

This enables lazy loading of navigational properties that can be overridden.

LOADING STRATEGIES - LAZY (2/2)

3. Requires that all navigational properties are declared virtual

```
public class Author {
    ...
    public virtual List<Book> Books {get; set;}
}
```

```
public BookServices(ILazyLoader layzyLoader)
```

TRACKING

To track

```
_context.Books.ToList()
```

or to NoTrack

```
_context.Books.AsNoTracking().ToList()
```

- `AsNoTracking` gives better performance in readonly scenarios

CHANGING

Without AsNoTracking - data can be changed:

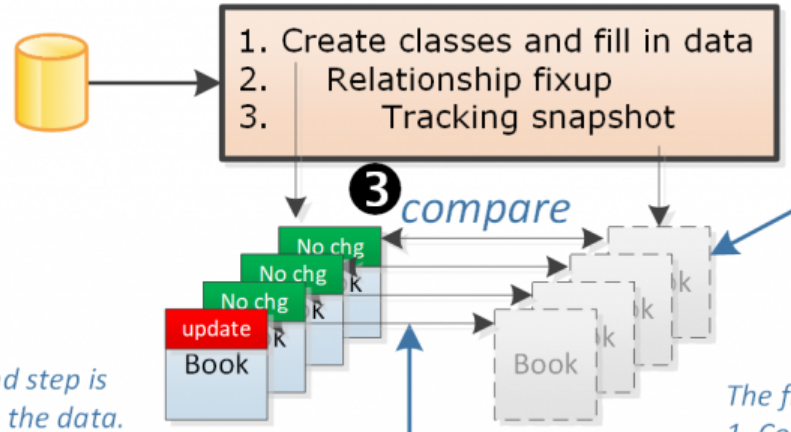
```
var book = _context.Books.Single(p =>
        p.Title == "Database Systems");
book.Isbn = "12341234";
_context.SaveChanges();
```

When SaveChanges is run, EF Core method DetectChanges which compares snapshot with application copy

The three stages of an update

❶
```
var books = context.Books.ToList();
```

First step is to read in the data using a normal query, which creates a "tracking snapshot" which holds a copy of the data



The second step is to update the data. In this case we change the PublishedOn date of the first book

❷
```
books.First().PublishedOn
    = new DateTime(2020, 1, 1);
```

The final step calls SaveChanges. This:
1. *Compares the books list against the tracking snapshot copy.*
2. *For every change it creates a database update command.*
3. *Then it excutes these update inside a transaction.*

```
context.SaveChanges();
```
❸

CREATE

```
public class AClass {
  public void Create() {
    var book = new Book {
      Isbn = "1234",
      Title = "Functional Programming in Scala",
      Author = paulChiusano
   };
   _context.Add(book); // or _context.Books.Add(book);
   ...
   _context.SaveChanges();
  } }
```

- EF Core expects primary key with SQL IDENTITY.
- Primary keys which are eg. GUID should be created with ValueGenerator

Speaker notes

paulChiusano - is an object of type Author

```
public class Book {
    ...
    public string Title {get;set;}
    public string ISBN {get;set;}
    public Author Author {get; set;} //  Navigational property
}
```

UPDATE

```
public class AClass {
  public void Update() {
    var book = _context.Books.Single(p =>
                    p.Title == "Database Systems");
    book.Isbn = "12341234";
    ...
    _context.SaveChanges();
  }
}
```

DELETE

```
public class AClass {
  public void Delete() {
    var book = _context.Books.First();
    _context.Remove(book); // or _context.Books.Remove(book);
    ...
    _context.SaveChanges();
  }
}
```

MANIPULATING CONTENT

- Multiple save/delete/update statements can be made in a single `SaveChanges()`
- `SaveChanges` vs `SaveChangesAsync`

EXERCISES :)

REFERENCES