Overthrowing the Tyranny of the Database with the Repository and Adapter Patterns

Benjamin Day
TRAINER | COACH | DEVELOPER

@benday www.benday.com

Overview

Unit testing the database

- Finding creative ways to NOT test the database

Design data access logic for testability

Repository Pattern

Adapter Pattern

How to do this with Entity Framework Core

Using a fake in-memory Repository to do unit testing

First up:
Why is the database 100% pure poison for unit testing?

...and what does this have to do with the 1986 movie Karate Kid Part II?

Databases are at the
center of the coding universe.

Unit testing goal:
Test in isolation and
test without dependencies

Databases are the
ultimate dependency.

Unit Testing Tips Summary

Code against interfaces

- Code against IPerson not Person

Use dependency injection

- Represent dependencies as interfaces
- Pass dependencies in through the constructor of the relying class

Single Responsibility Principle (SRP)

- A class should do only one thing

Databases are the
ultimate dependency.

Database Unit Testing Goals

Hide your database behind an interface

Hide the database implementation details

Hide the ~~database~~ implementation details

Think “Persistence” Not “Database”

Persistence = permanently stored

Persistence mechanism could be...

- Relational database
- NoSQL database
- Document database
- Web service
- In-memory
- File system

The “how” doesn’t matter

Repository Pattern

The Repository Pattern

Encapsulate persistence logic

- Data access logic

Usually in the context of a single class

Need data access for a Person class?

→ IPersonRepository

SqlServerPersonRepository : IPersonRepository

CosmosDocumentDbPersonRepository : IPersonRepository

IPersonRepository

```
public interface IPersonRepository
{
    void Save(IPerson saveThis);
    IPerson GetById(int id);
    List<IPerson> GetAll();
    void Delete(IPerson deleteThis);
}
```

Next up:
The Object-relational Impedance Mismatch Problem and the Adapter Pattern

IPersonRepository

```
public interface IPersonRepository
{
    void Save(IPerson saveThis);
    IPerson GetById(int id);
    List<IPerson> GetAll();
    void Delete(IPerson deleteThis);
}
```

The Object-relational Impedance Mismatch Problem

Object-relational Impedance Mismatch

The data for an object is one thing

When stored in a database, it might be multiple things

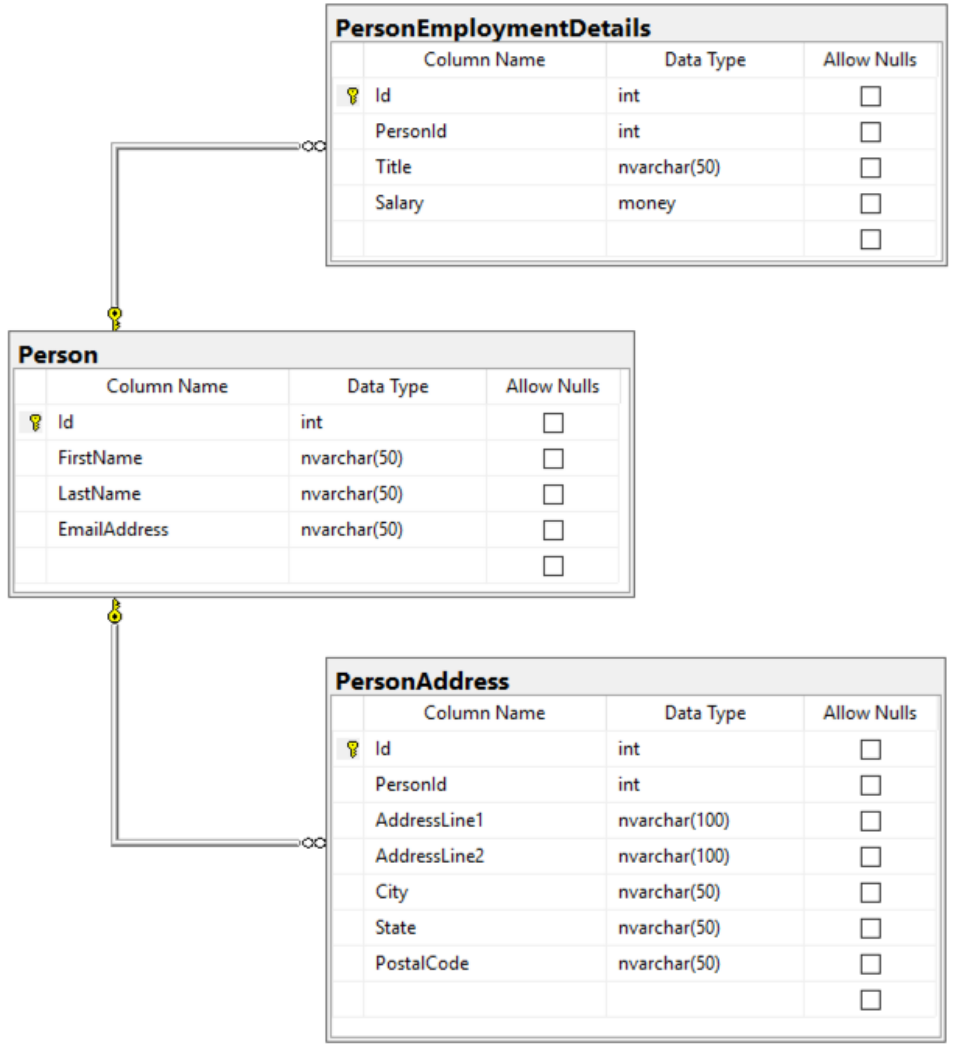
Huge source of complexity

Complexity → Bugs

High probability of bugs →
"I want to test this"

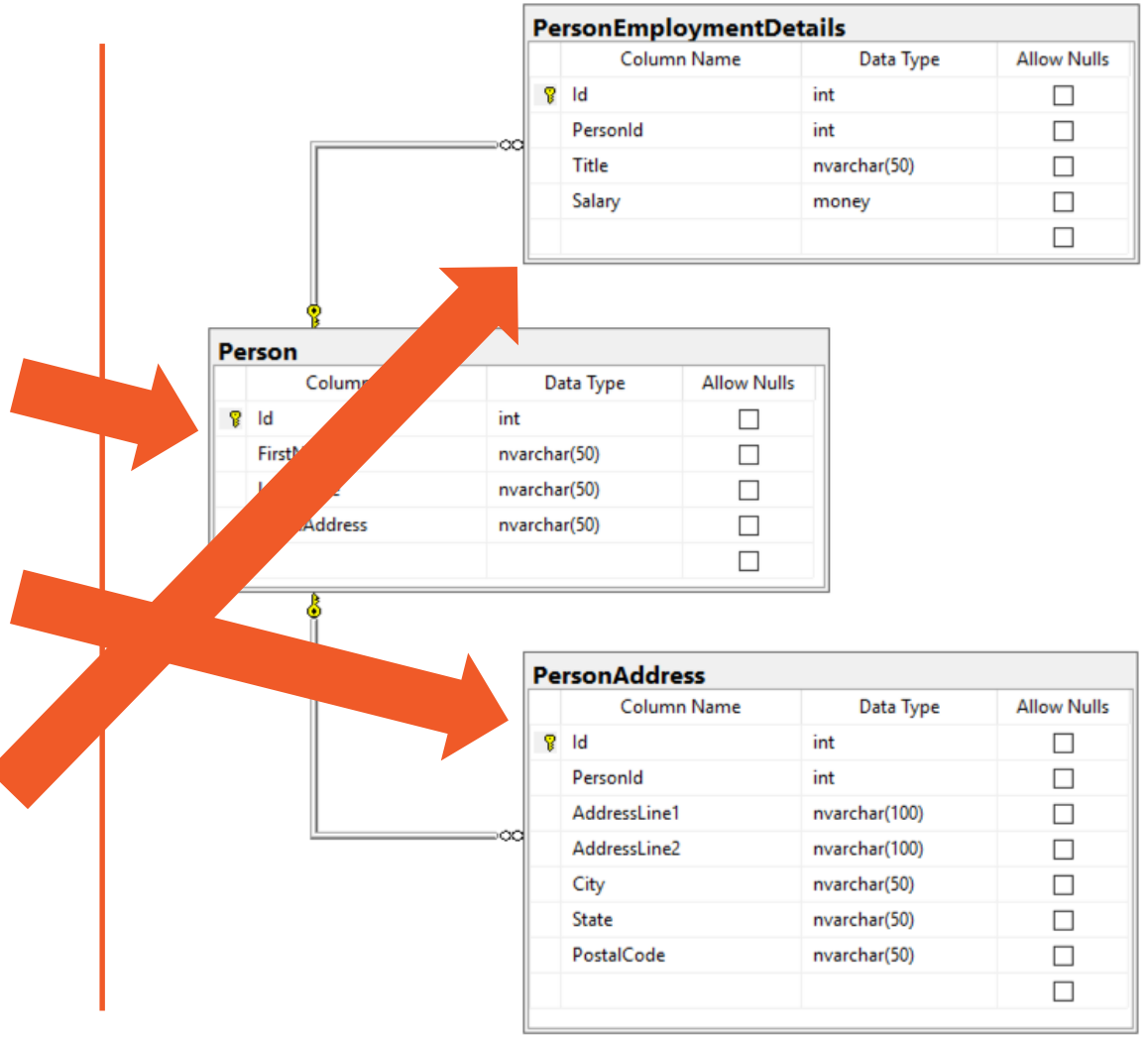
Class vs. Database

```
public class Employee
{
  public int Id { get; set; }
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public string EmailAddress { get; set; }
  public string AddressLine1 { get; set; }
  public string AddressLine2 { get; set; }
  public string City { get; set; }
  public string State { get; set; }
  public string PostalCode { get; set; }
  public string Title { get; set; }
  public float Salary { get; set; }
}
```



Class vs. Database

```
public class Employee
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string EmailAddress { get; set; }
    public string AddressLine1 { get; set; }
    public string AddressLine2 { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string PostalCode { get; set; }
    public string Title { get; set; }
    public float Salary { get; set; }
}
```



Routing all these pieces of data is a huge source of bugs.

Data access for Employee →
EmployeeDataAccess or
EmployeeRepository

```
SaveEmployee(
  Employee saveThis)
```

Take the Employee object values

Generate INSERT for Person table

Run the INSERT

Get the generated Person.Id value

Generate INSERT for PersonAddress table

Generate INSERT for PersonEmploymentDetails table

Run the remaining INSERT statements

Set the generated Person.Id value back to the Employee object

```
GetEmployeeById(
   int employeeId)
```

Generate SELECT for Person table

Generate SELECT for PersonAddress table

Generate SELECT for PersonEmploymentDetails table

Run the SELECT statements

Create an instance of Employee class

Set the values from the query results onto the Employee class

Return the Employee class

What’s wrong with this?

It violates
Single Responsibility Principle
(SRP).

“Repository Pattern = Data Access Logic
Where’s the SRP violation?”

The Single Responsibility Principle Violation in a Data Access Class

Two types of logic:

Data Access Logic

- Make the calls to the database

Data Conversion Logic

- Shape the data for the database query
- Shape the results for the rest of the app

Data Access Logic vs. Data Conversion Logic

Which has more bugs?

Data access logic works or it doesn't

Data conversion logic has most of the bugs

- Wrong value in the wrong place
- Forgetting to set a value

Adapter Pattern

Adapter Pattern

Logic to turn one type of data into another type of data

Unit Testing and Dependency Management

Repository Pattern

Encapsulates data access logic

Has a HUGE dependency on database

Low bug count

Hard to unit test
Low ROI for the testing effort

Adapter Pattern

Converts data between objects / queries

No dependencies on the database

HUGE bug count

Easy to unit test
High ROI for the testing effort

Focus unit testing on Adapter logic

* ROI = Return on Investment

Goal: Break our unit testing dependency on the database

Best unit testing advice I ever got...

“Best way to avoid a fight:
don’t be there.”

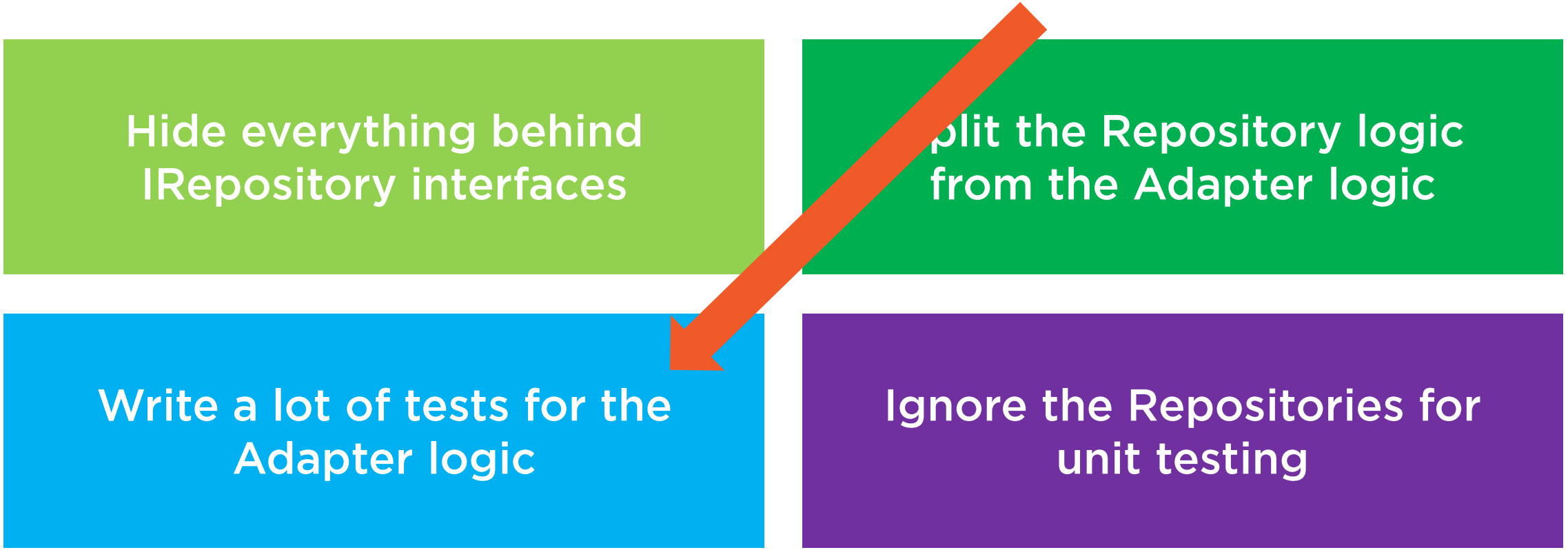
Pat Morita as Mr. Miyagi
Karate Kid II
1986

It’s really hard to write tests against a running database.

So don’t try to write tests against your database.

Mr. Miyagi's School of Database Unit Testing

Most of your defects



Next up:
Unit Testing
Entity Framework Core

Implementing Your Repositories

What are you going to use for data access?

Probably
Entity Framework Core

EF Core

Object-Relational Mapper (ORM)

ORMs try to smooth out the Object-relational Mismatch Problem

Goal: Make it easy to...

- ...code against classes
- ...do database operations with those classes

Super-fast Overview of EF Core

Define the classes that EF Core knows about

- "Entities"

Tell EF Core how to map them into database tables/columns

- Or accept the defaults

Code against those classes

EF Core handles the database operations

- Create, Read, Update, Delete (CRUD)

Coding with EF Core

Create a DbContext for your app

- MyDbContext : DbContext

Create DbSet<T> properties

- One for each Entity class
- DbSet<Order> Orders { get; set; }

Define relationships between Entities

- Foreign keys, Collections, Inheritance

Mappings to the database are defined by...

- ...convention
- ..."fluent" mappings
- ...attributes on the Entities

You're probably using EF Core for data access...

...but there's another option.

ADO.NET

Library for doing data access

- EF Core uses ADO.NET

You write everything yourself

- Lots of control

Connection / SqlConnection

Command / SqlCommand

DataAdapter / SqlDataAdapter

DataSet

DataTable

DataRow

ADO.NET is there for you
if EF Core starts to get in your way
or has performance problems.

How do you unit test EF Core?

You don’t.

How do you unit test ADO.NET?

You don’t.

Realistically,
you'll probably need to write some
integration tests.

Remember:
Split the Repository logic from the Adapter logic in order to maximize what can be unit tested.

Your Repository classes will...

...provide an interface for your persistence operations

- Save(), GetById(), GetAll(), Delete(), etc.

...encapsulate EF Core / ADO.NET logic

...make the actual calls to the database

...depend on Adapter classes to

- Get ready to make DB queries
- Convert results of DB queries to classes used by the rest of the application

I'm going to assume you're using EF Core.

Next up:
Why do we even need Adapters with EF Core?

I can read your mind.

“This Ben Day guy is a complete fraud.”

“EF Core lets me save objects into the database…”

“...therefore, I can skip writing and testing Adapters. I’ll just map my business objects directly to the database. Done and done.”

Please don’t do that.

Change will happen.

Do Yourself a Favor

Keep your Domain Model objects separate from your EF Core Entities

- ”Business objects”
- The classes your app cares about

Write Adapter objects to go between those two types of classes

It might feel like extra code...

...but it’s an insurance policy

No Adapter vs. Adapter

No Adapter Classes

Domain Model looks exactly like the database

Your enemy is change

Change gets messy

Change goes everywhere

Tight coupling between Domain Model and database schema

Change in one → change in the other

Adapter Classes

Domain Model doesn't have to look like the database

Change is manageable

Loose coupling between Domain Model and database schema

Maintenance needs of the Domain Model and database are separate

Change in one != change in the other

Give you options for performance optimizations

Next up:
Why did President Grover Cleveland make my demo app more complicated?

The Demo App:
US Presidents Database

Requirements

Searchable database of US Presidents

- First name, last name
- Date of birth, death
- Place of birth, death
- Terms in office

Eventually, a searchable database of

- Presidents
- Vice Presidents
- Spouses
- Senators, Representatives
- Children

The design started out simple.

US Presidents can only serve
2 terms in office

You'd Think You Could Do This

```
public class President
{
  public int Id { get; set; }
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public int PresidentNumber { get; set; }
  public DateTime StartOfTerm { get; set; }
  public DateTime EndOfTerm { get; set; }
}
```

President Grover Cleveland ruined it.

Grover Cleveland is the only president to serve two non-consecutive terms.

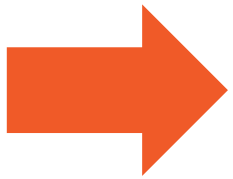
Grover Cleveland, Term 1: 1885 to 1889

Benjamin Harrison: 1889 to 1893

Grover Cleveland, Term 2: 1893 to 1897

Grover Cleveland Ruins My Design

```
public class President
{
  public int Id { get; set; }
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public int PresidentNumber { get; set; }
  public DateTime StartOfTerm { get; set; }
  public DateTime EndOfTerm { get; set; }
}
```



```
public class President
{
  public int Id { get; set; }
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public List<Term> Terms { get; set; }
}

public class Term
{
  public int Id { get; set; }
  public int PresidentNumber { get; set; }
  public DateTime StartOfTerm { get; set; }
  public DateTime EndOfTerm { get; set; }
}
```

The Design

Domain Model / Business Objects

- President
- Term

Database Tables

- Person
- PersonFact

Database uses Entity Attribute Value Model

- Very flexible
- Almost no schema changes

EF Core Implementation

Database Tables

- Person
- PersonFact

EF Core Entities

- Person
- PersonFact

Tip: Keep it simple

- Heretical advice
- Entities almost always look exactly like their database table
- Good for performance and maintenance

Repositories and Adapters

Person is our “Aggregate Root”

Aggregate Root

- Domain Model term
- The object through which you access a collection of related objects

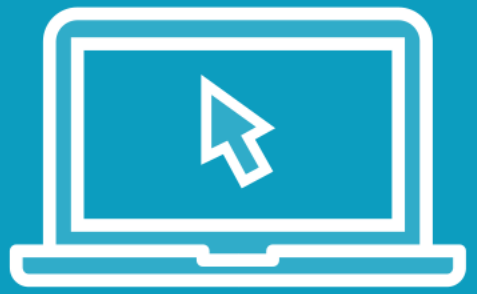
IRepository<Person>

SqlEntityFrameworkPersonRepository

PersonToPresidentAdapter

Next up:
Let’s implement the adapters

Demo



Implement and unit test an Adapter

PersonToPresidentAdapter

PersonToPresidentAdapterFixture

George Washington

```
SELECT *
FROM Person
Where Id = 1

SELECT *
FROM PersonFact
WHERE PersonId = 1
```

100 %

Results | Messages

	Id	FirstName	LastName
1	1	George	Washington

	Id	EndDate	FactType	FactValue	PersonId	StartDate
1	1	0001-01-01 00:00:00.0000000	Image Filename	george-washington.jpg	1	0001-01-01 00:00:00.0000000
2	2	0001-01-01 00:00:00.0000000	Birth City	Westmoreland County	1	0001-01-01 00:00:00.0000000
3	3	1732-02-22 00:00:00.0000000	Birth Date	Birth Date	1	1732-02-22 00:00:00.0000000
4	4	0001-01-01 00:00:00.0000000	Birth State	Virginia	1	0001-01-01 00:00:00.0000000
5	5	0001-01-01 00:00:00.0000000	Death City	Mount Vernon	1	0001-01-01 00:00:00.0000000
6	6	1799-12-14 00:00:00.0000000	Death Date	Death Date	1	1799-12-14 00:00:00.0000000
7	7	0001-01-01 00:00:00.0000000	Death State	Virginia	1	0001-01-01 00:00:00.0000000
8	8	1797-03-04 00:00:00.0000000	President	1	1	1789-04-30 00:00:00.0000000

Grover Cleveland

SQLQuery2.sql - G...GRAVY\benday (55))* | SQLQuery1.sql - G...GRAVY\benday (54))*

```
SELECT *
FROM Person
Where FirstName = 'Grover'

SELECT *
FROM PersonFact
WHERE PersonId = 22
```

100 %

Results | Messages

	Id	FirstName	LastName
1	22	Grover	Cleveland

	Id	EndDate	FactType	FactValue	PersonId	StartDate
1	169	0001-01-01 00:00:00.0000000	Image Filename	grover-cleveland.jpg	22	0001-01-01 00:00:00.0000000
2	170	0001-01-01 00:00:00.0000000	Birth City	Caldwell	22	0001-01-01 00:00:00.0000000
3	171	1837-03-18 00:00:00.0000000	Birth Date	Birth Date	22	1837-03-18 00:00:00.0000000
4	172	0001-01-01 00:00:00.0000000	Birth State	New Jersey	22	0001-01-01 00:00:00.0000000
5	173	0001-01-01 00:00:00.0000000	Death City	Princeton	22	0001-01-01 00:00:00.0000000
6	174	1908-06-24 00:00:00.0000000	Death Date	Death Date	22	1908-06-24 00:00:00.0000000
7	175	0001-01-01 00:00:00.0000000	Death State	New Jersey	22	0001-01-01 00:00:00.0000000
8	176	1889-03-04 00:00:00.0000000	President	22	22	1885-03-04 00:00:00.0000000
9	177	1897-03-04 00:00:00.0000000	President	24	22	1893-03-04 00:00:00.0000000

Next up:
Let’s implement the repositories

Demo

Implement a Repository

Integration Tests

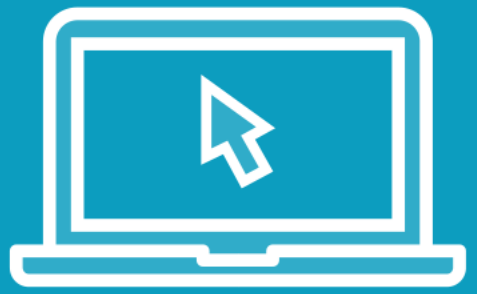
SqlEntityFrameworkPersonRepository

Reusable repository classes

- SqlEntityFrameworkRepositoryBase
- SqlEntityFrameworkCrudRepositoryBase

Next up:
Tie it together with
Service Layer Pattern

Demo



Service Layer Pattern

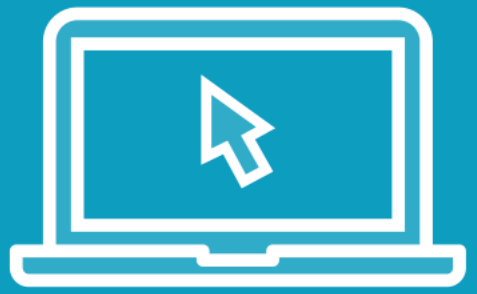
Implement PresidentService

Manage operations related to President

Connects logic Adapt and Repository logic

Next up:
Let’s mock our repositories

Demo



Fun with Dependency Injection and Repository interfaces

PresidentService

Rule: “No Duplicate Presidents”

- Save a President to the database
- Throw an error when you try to save a duplicate President

How do you test this without a real database?!?!?!

Unit test this using an In-Memory Repository

Summary

Unit testing the database

- Finding creative ways to NOT test the database

Design data access logic for testability

Repository Pattern

Adapter Pattern

How to do this with Entity Framework Core

Using a fake in-memory Repository to do unit testing

Next up:
Testing Validation and Calculation Logic