

Hands-on lab

Lab: Entity Framework + MVVM

In the chapter on layered architecture, we built an HR application which can be used to generate pair-programming pairs for the developers of a team.

In the infrastructure layer we used an in memory repository to retrieve stored developers. In this lab we will replace the in memory repository with a database repository that communicates with a MS SQL Server database using Entity Framework (EF).

In the presentation layer the views contain (too much) logic. We will refactor the presentation layer to use the MVVM pattern to make it more testable, maintainable and extendible.

Exercise 1: Creating the data model and database

Download the *HumanResourcesApp* demo code from blackboard. Open the solution in Visual Studio.

Add the necessary NuGet packages to the solution (right-click on the solution -> Manage NuGet packages for solution...):

- Infrastructure layer
 - *Microsoft.EntityFrameworkCore.SqlServer*. The version should be the latest 6.x version (not 7.0 or higher).
- Presentation layer (executing program)
 - *Microsoft.EntityFrameworkCore.Tools*. The version should be the latest 6.x version (not 7.0 or higher).

Now we can start to create the data model by creating a class that inherits from *DbContext*.

- Add an internal class *HumanResourcesContext* to the infrastructure layer
 - Inherit from *DbContext*
 - Add *DbSet* for employees
 - Add *DbSet* for teams
 - Tell EF which data provider and connection string to use
 - Override *OnConfiguring*
 - Use the *UseSqlServer* extension method from the *Microsoft.EntityFrameworkCore.SqlServer* package. The connection string should be `"Data Source = (localdb)\MSSQLLocalDB; Initial Catalog = HumanResourcesDb"`.
 - Make sure EF logs database commands to the debug console. You can use the the following code snippet:

C#

```
string connectionString = "Data Source=(localdb)\\MSSQLLocalDB;Initial
Catalog=HumanResourcesDb;Integrated Security=True;";
optionsBuilder
    .UseSqlServer(connectionString)
    .LogTo(message => Debug.WriteLine(message), new[] {
        DbLoggerCategory.Database.Command.Name }, LogLevel.Information);
```

- Give EF some extra information about the model by overriding *OnModelCreating* and using Fluent API
 - Explain that *EmployeeNumber* has a maximum length of 20 and should be used as the primary key for the *Employees* table.

Now that we have defined a data model for EF to use, we can create an initial migration that creates the database using the data model:

- Open the *Package Manager Console* window in Visual Studio
 - Set the default project to the infrastructure layer
 - Use a powershell command to create a migration named “*Initial*” (Tip: use the *-verbose* flag to get more detailed output in the console)

Now that the initial migration is created, we can create the database itself.

In this exercise we will let the application automatically execute pending migrations when the application starts. Add the following method to the *HumanResourcesContext* class:

C#

```
...
    public void EnsureIsMigrated()
    {
        if (Database.GetPendingMigrations().Any())
        {
            Database.Migrate();
        }
    }
...
}
```

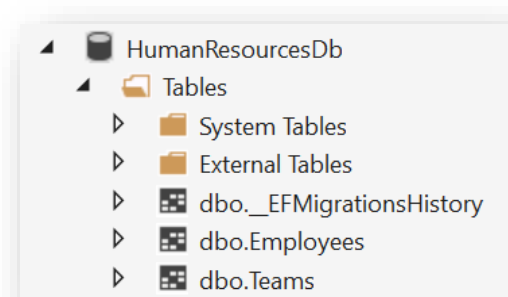
The *DataBase* property of the *DbContext* class enables you to do actions on the database. The *EnsureIsMigrated* method checks if there are pending migrations (this is true when the database is not in sync with the most recent migration in code). When there are pending migrations, the pending migrations will be executed one by one (generates SQL statements and runs them against the database server, just as the *update-database* command would do).

Add the following code to the *OnStartup* method of the application to trigger the migration when the application starts:

```
C#  
  
...  
    protected override void OnStartup(StartupEventArgs e)  
    {  
        var context = new HumanResourcesContext();  
        context.EnsureIsMigrated();  
  
        ...  
    }  
}
```

When the application is started, the database should be created.

Use the *SQL Server Object Explorer* window in Visual Studio to check if the database is created.



Exercise 2: Seeding the database

The application should have some employees and a team in the database when the application starts.

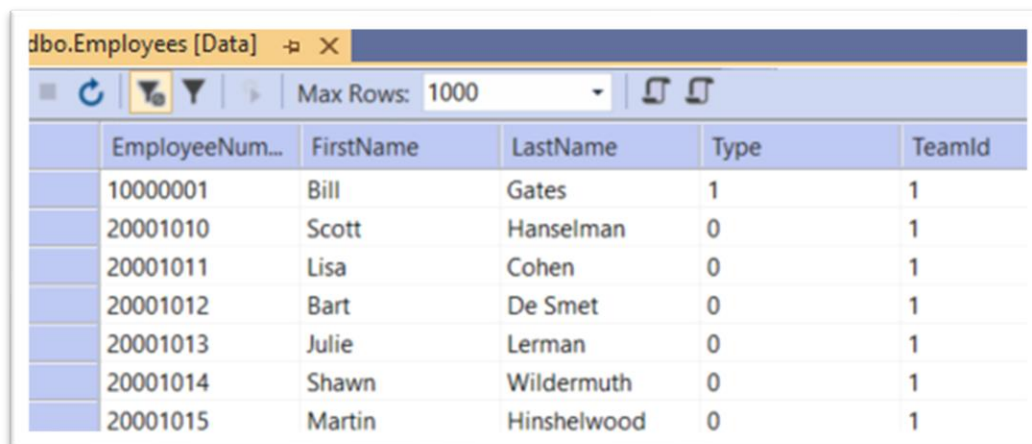
So we are going to change the data model to contain some employees and a team. Next we are going to create a migration that can synchronize the database with the changed data model.

- Use Fluent Api in the *OnModelCreating* method of *HumanResourcesContext* to add a team to the data model (Note that you must explicitly set the *Id* property when seeding data. Normally, when adding a team, the *Id* would be set by the database and is automatically filled in by EF.)
 - Name: Dream team, Id: 1
- Use Fluent Api in the *OnModelCreating* method of *HumanResourcesContext* to add some employees to the data model
 - EmployeeNumber: 20001010, FirstName: Scott, LastName: Hanselman, TeamId: 1, Type: Developer

- EmployeeNumber: 20001011, FirstName: Lisa, LastName: Cohen, TeamId: 1, Type: Developer
- EmployeeNumber: 20001012, FirstName: Bart, LastName: De Smet, TeamId: 1, Type: Developer
- EmployeeNumber: 20001013, FirstName: Julie, LastName: Lerman, TeamId: 1, Type: Developer
- EmployeeNumber: 20001014, FirstName: Shawn, LastName: Wildermuth, TeamId: 1, Type: Developer
- EmployeeNumber: 20001015, FirstName: Martin, LastName: Hinshelwood, TeamId: 1, Type: Developer
- EmployeeNumber: 10000001, FirstName: Bill, LastName: Gates, TeamId: 1, Type: Account manager

Now create a new migration named “SeedData” to the code.

Start the application (which will execute the migration) and use the *SQL Server Object Explorer* window to check if the tables are filled with data.



EmployeeNum...	FirstName	LastName	Type	TeamId
10000001	Bill	Gates	1	1
20001010	Scott	Hanselman	0	1
20001011	Lisa	Cohen	0	1
20001012	Bart	De Smet	0	1
20001013	Julie	Lerman	0	1
20001014	Shawn	Wildermuth	0	1
20001015	Martin	Hinshelwood	0	1

Exercise 3: Query the database

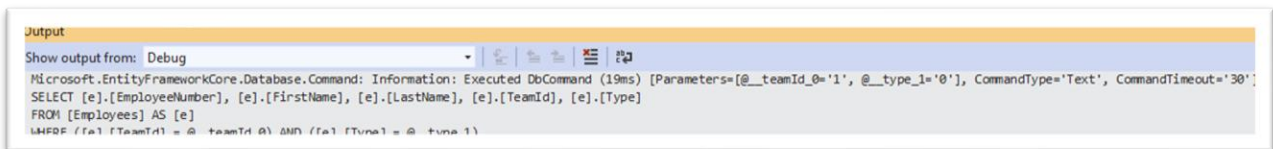
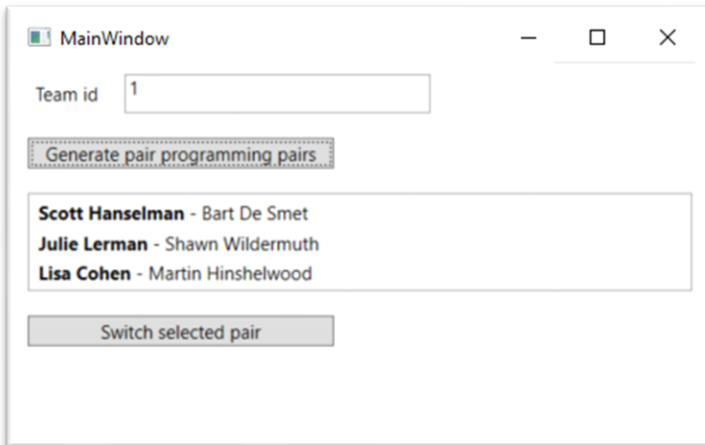
We will provide a new implementation of the *IEmployeeRepository* interface. This time the repository will use EF to communicate with a database.

- Add an internal class *EmployeeDbRepository* to the infrastructure layer. The class should implement the *IEmployeeRepository* interface from the business layer.
- Inject an instance of *HumanResourcesContext* in the constructor of *EmployeeDbRepository* and store the injected parameter value in a private readonly field.

- Use the injected context to implement the *FindEmployeesOfTeam* method

Change the wiring code in *App.xaml.cs* so that the *EmployeeDbRepository* is used instead of the *InMemoryEmployeeRepository*.

Start the application. It should now retrieve employees from the database and output EF database commands to the debug window.



Exercise 4: Refactor the presentation layer to use MVVM

Our last goal is to make the presentation layer more extendible, testable and maintainable.

We will do this by rewriting the code to use the MVVM pattern.

While doing this, we will introduce a home view and a menu to navigate from one view to another view.

- Add a *Command* folder. Add a *DelegateCommand* class in this folder (the same class that is used in the Pluralsight course)
- Add a *View* folder. Add a *HomeView* UserControl and a *PairProgrammingView* UserControl in this folder.
- Now move the XAML code and C# logic of the *MainWindow* to the *PairProgrammingView* UserControl.
 - The C# logic needs an implementation of *IPairProgrammingService*. For now, just create an instance of *PairProgrammingService* in the constructor of the UserControl (instead of injecting it in a constructor parameter)
 - Add a menu and a *ContentControl* containing the *PairProgrammingView* to the *MainWindow* by placing the following Grid into the content of the Window

XAML

```
...
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Menu FontSize="20">
        <MenuItem Header="_Menu">
            <MenuItem Header="_Home" />
            <MenuItem Header="_Generate pairs" />
        </MenuItem>
    </Menu>
    <ContentControl Grid.Row="1">
        <view:PairProgrammingView></view:PairProgrammingView>
    </ContentControl>
</Grid>
```

- At this point you should be able to start the application. Everything should work as before.

- Add a *ViewModel* folder.
 - Add a *ViewModelBase* class

C#

```
public abstract class ViewModelBase : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler? PropertyChanged;

    protected virtual void OnPropertyChanged([CallerMemberName] string? propertyName
= null)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }

    public virtual void Load() { }
}
```

- Add a *HomeViewModel* that inherits from *ViewModelBase*
- Add a *PairProgrammingViewModel* that inherits from *ViewModelBase*
- Add a *MainViewModel* that inherits from *ViewModelBase*
- The *MainViewModel* will be responsible for determining which View(Model) should be displayed:

C#

```
public class MainViewModel : ViewModelBase
{
    public PairProgrammingViewModel PairProgrammingViewModel { get; }
    public HomeViewModel HomeViewModel { get; }

    private ViewModelBase _selectedViewModel;

    public ViewModelBase SelectedViewModel
    {
        get => _selectedViewModel;
        private set
        {
            _selectedViewModel = value;
            _selectedViewModel.Load();
            OnPropertyChanged();
        }
    }
}
```



```

    public DelegateCommand SelectViewModelCommand { get; }

    public MainViewModel(HomeViewModel homeViewModel, PairProgrammingViewModel
pairProgrammingViewModel)
    {
        HomeViewModel = homeViewModel;
        PairProgrammingViewModel = pairProgrammingViewModel;

        SelectViewModelCommand = new DelegateCommand(SelectViewModel);

        SelectedViewModel = HomeViewModel;
    }

    private void SelectViewModel(object? obj)
    {
        ViewModelBase viewModel = (ViewModelBase)obj!;
        SelectedViewModel = viewModel;
    }

    public override void Load()
    {
        SelectedViewModel.Load();
    }
}

```

- Link the MainWindow to the MainViewModel:

C#

```

public partial class MainWindow : Window
{
    private readonly MainViewModel _viewModel;

    public MainWindow(MainViewModel mainViewModel)
    {
        InitializeComponent();

        _viewModel = mainViewModel;
        Loaded += MainWindow_Loaded;
        DataContext = _viewModel;
    }

    private void MainWindow_Loaded(object sender, RoutedEventArgs e)
    {
        _viewModel.Load();
    }
}

```

- You will have to fix the wiring of the application (App.xaml.cs)
- Add DataTemplates for the *HomeViewModel* and *PairProgrammingViewModel* in the Resources of the *MainWindow* (XAML)
- Use data binding to set the *Content* property of the *ContentControl* to be the *SelectedViewModel* of the *MainViewModel*
- At this point you should be able to start the application. Everything should work as before.
- Finally we will move the logic in the *PairProgrammingView* to the *PairProgrammingViewModel*. **After this step, there should not be any logic in the codebehind of the *PairProgrammingView*.** Tips:
 - Inject an *IPairProgrammingService* in the *ViewModel*
 - There should be a *ObservableCollection<ProgrammingPair> Pairs* property in the *ViewModel*
 - There should be a *int SelectedTeamId* property in the *ViewModel*
 - There should be a *ProgrammingPair? SelectedPair* property in the *ViewModel*
 - There should be a *GenerateCommand* and a *SwitchCommand* in the *ViewModel*
 - The *SwitchCommand* can only be executed when there is a *SelectedPair*
 - The team with id 1 is selected by default
 - *ObservableCollection* has a *Clear* method that removes all the items from the collection