Mapping DDD Patterns with EF Core 2.1

Hands-on Lab

Julie Lerman

October 2018

**Setup**

Prepare to use either Visual Studio Code (Mac, Windows, Linux) or Visual Studio 2017.

If you do NOT have VS2017 installed, please use VS Code.

Best to have at least ONE person on the team familiar with whichever IDE you are using.

Workshop repo for DDD EU 2018 I will walk users through the steps in the workshop.

Find a partner to pair with!

Work in VS2017 (Windows) or Visual Studio Code (Windows, Mac or Linux)

Please have one or both team members be familiar with the IDE of choice and with GIT basics.

You can download VS Code at <https://code.visualstudio.com/>

You must be sure latest .NET Core SDK is installed <https://www.microsoft.com/net/download/>

One lab requires SDK 2.2-Preview. If you don’t mind having that on your machine you can get it from https://www.microsoft.com/net/download/dotnet-core/2.2.

And you'll need my repository.

You can clone it from the command line into a pre-existing directory with   
git clone https://github.com/julielerman/TTK-WS.git

**WORKSHOP STEPS**

For each exercise (after A) there is a pair of BEFORE and AFTER branches. E.g. B BEFORE, B AFTER, C BEFORE, C AFTER, etc. You will work from the BEFORE branches but can skip to the AFTER branches if you want or need to.

I will walk you through exercises A and B. After that, I will introduce each exercise and then do a quick review with the AFTER to be sure you understand the code and lessons of the exercise.

Hints:  
VS2017 might be cranky when you first open the solution. Maybe not. Do rebuild. Don’t run anything from master branch. VS will want you to commit changes to SLN file after compiling. Just do that or ignore so you can move onto a new branch

Also in VS2017, the branches will be originally listed as remotes/origin/branch name.

In VS2017, you will need to build each branch after opening it.

If restore packages is not kicking in for EF Core packages (I don’t know yet why this happens), go to command line (in dddeu folder) and type dotnet restore.

I will demonstrate how to get started in Visual Studio Code.

**Exercise A) Review Domain and Domain Tests\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Branch A is for review of the domain and run domain tests to prove everything works.

**Review EF Core additions**

EF Core and data project have been added to **B\_EncapsulateScalars\_BEFORE**

Note: Migration commands need to be run from within the data directory.

VS2017 users: Because all of the branches in this repository use the CLI version of the EF Core migrations commands, it is easier to just run these commands at the command line from the data directory

1. Prove that EF Core is happy with your starting point by running CLI command FROM data folder

dotnet ef dbcontext info –startup-project ../console

shortcut : *dotnet ef dbcontext info -s ../console*

1. And add a baseline migration with

dotnet ef migrations add init -s../console

Check out the new migration file to see what tables, columns and relationships get added

Optional: Create the SQLite database from the migration:

**dotnet ef database update -s../console**

Inspect the database in the relevant IDE if you have one installed

**TESTING NOTES:**

In VSCode, you have two options to run tests. 1) With CodeLens on (default) you can select run test or even debug test for each test. 2) Alternatively, you can use the CLI to run tests.

With the CLI, to run tests in a particular class, use the filter parameter:dotnet test --filter test.InMemoryTests

In VS2017, you can also use CodeLens or the Test Explorer to run tests. That will require building a solution when you open a new branch.

**Exercise B) Encapsulate scalar properties**

*You will implement a new business rule which is that Team Name can only be set via constructor and never edited. Therefore, the Name scalar property will be encapsulated.*

*Then you will see how EF Core can map this encapsulated property.*

*Note: Don’t run tests until instructed. They will just fail.* ☺

1. Modify the team class to reflect the new rule by introducing a private field, \_teamName, then altering the TeamName property to only return that field:

private string \_teamname;

public string TeamName=> \_teamname;

1. Modify the constructor to set value of \_teamname field rather than TeamName prop
2. Save the file
3. Run tests, filter on *DomainClassTests*, to make sure nothing is broken. (*The EFCore InMemory test will fail)*
4. Add a new migration (in the data folder) to see how this change affects the database

dotnet ef migrations add immutableTeamName -s ../console

Result: The migration *removes* TeamName column.   
Q: Why?   
A: For convention, property needs to be discoverable by model and that requires getter and setter.

You could just add the setter in, but then you are altering the domain model to satisfy the ORM. This is an anti-pattern for domain modeling.

Instead, EF Core lets you configure it in the fluent API

1. First, let’s remove the last (bad) migration with

dotnet ef migrations remove -s ../console

1. Add the following HasField configuration to let EF Core know to link the property & field:

modelBuilder.Entity<Team>().Property(b => b.TeamName).HasField("\_teamname");

1. Save the file and run the immutableTeamName migration again. Up & Down should be empty.

With this mapping in place, you can benefit from EF Core 2.1 ability to materialize entities without parameterless constructors. Feel free to comment out the parameterless ctor in Team.cs.

I will demo this feature further at the end of this exercise. (You might need to remind me!)

**Testing the Encapsulation with EF Core’s InMemory provider**

**CanStoreAndRetrieveImmutableTestname** in EFCoreInMemoryTests uses an InMemory provider. It sets an InMemory provider on TeamContext, instantiates the context, adds & saves new Team with Name, and then queries what was stored. The test is verifying that the completely encapsulated Team Name is recognized by EF when constructing SQL and materializing the query results. We knew that from the migration, but it’s still nice to see.

**Exercise C) ENCAPSULATE NAVIGATION COLLECTION**

**Time to protect the players collection property using known coding patterns. EF Core is now able to map this to the database because it recognizes Enumerable. Yahoo!**

1. In team.cs, create backing field for Players:

private List<Player> \_players;

1. change List<Player> to be Enumerable<Player> and also return not just the variable, but a “defensive copy” via ToList(). It is very, very encapsulated now.

public IEnumerable<Player>=>\_players.ToList();

1. Replace all uses of Players property with \_players field
2. Build (VS Build or dotnet build ) to be sure it compiles
3. Run the *domain* tests for players to make sure they still work
4. Add a migration to reflect this change in the data model  
   *Remember to do this in the data folder*

dotnet ef migrations add encapsulatedPlayers –s ../console

The migration should be empty, meaning that as far as EF Core is concerned, nothing has changed.

1. Run **CanStoreAndRetrieveTeamPlayers** test. It will force you to add a parameterless ctor on Player. Remember, EF Core needs these to materialize objects but they can be private.

**Test still fails because it can’t figure out how to eager load players because of the backing field combined with the defensive copy return from Players. WAT?**

1. You can fix this in EF! Configure the context to comprehend the navigation through the backing field using the following code. First to identify the navigation property, then specify that it can be set using the field.

var navigation = modelBuilder.Entity<Team>()

.Metadata.FindNavigation(nameof(Team.Players));

navigation.SetPropertyAccessMode(PropertyAccessMode.Field);

1. Run the test again and it passes.
2. Run the 2nd Players test --which verifies that my domain logic that depends on pre-loaded team players –and verifies that it, too, is working properly. This won’t work without the private setter.

**Exercise D) MAP VALUE OBJECTS**

*The Player and Manager names are both using the PersonFullName value object. We need to make sure EF Core can map that value object to the database.*

This branch has a new test (CanStoreAndRetrievePlayerName) to run.

1. Run **CanStoreAndRetrievePlayerName** which, which of course, will fail. ☺ Why?

Player contains PersonFullName which is a DDD value object. Notice that in TeamContext, that is ignored…the model is totally unaware of it.

1. Comment out the mapping with Ignore<PersonFullName>.
2. Try running the test again. Sorry, but it will still fail. But worse, it is actually throwing an exception. EF Core can’t infer value objects. We need to help!

EF Core introduced a concept called **Owned Entities**. In EF6, PersonFullName would have been an EF Complex Type. Now, EF Core will handle it underneath the covers as though it were an entity and it will “fake itself out” by using the id value of the entity that owns the value object. So that could be person or manager.

1. But we MUST configure that in the context. It’s just a fluent API mapping -- and easy.

modelBuilder.Entity<Player>().OwnsOne(p=>p.NameFactory);

1. Do the same for Manager otherwise you’ll get this error at runtime:  
   The entity type 'Player.NameFactory#PersonFullName' is configured as owned, but the entity type 'Manager.NameFactory#PersonFullName' is not. All entity types sharing a CLR type must be configured as owned.
2. Before running the test, let’s see how this affects the database by adding a new migration:

dotnet ef migrations add personfullname -s ../console

This adds 2 new columns to the player table and 2 to the manager table.

For those familiar with the 2.0 issue with getting owned entities via Include ([github.com/aspnet/EntityFrameworkCore/issues/9210)](https://github.com/aspnet/EntityFrameworkCore/issues/9210)), it’s been resolved in 2.1.

1. Run the test **CanStoreAndRetrievePlayerName.** Finally, it should pass.
2. You can also run **TeamAllowsAddingPlayersToExistingTeamWhenPlayersAreLoaded**

The AddPlayer method checks for duplicate names, so if the data is coming from the database, it relies on the Player.Name being populated thanks to the OwnedEntity. We couldn’t run this test until the value object was mapped.

**Exercise E) MANY to MANY**

*Team has two different relationships to manager. The first is a one to one with the current manager. The second is a many to many relationship with past managers. Coming from domain perspective this is logical. Is it too complex? That’s up to you to decide. Coming from a database perspective it may not make sense. But we can define it in the domain and tell EF Core how to deal with it!*

Manager can store the history of teams that manager has managed. But since a team can also have different managers over time, this requires a many-to-many relationship. My domain doesn’t require that I navigate from a team to all of its past managers, so I haven’t created the navigation property in the team class but it does exist in the manager class.

EF Core currently only supports M2M via explicit join entities. However, you do have to explicitly map the key for a join entity. So the way I’ve defined the many-to-many relationship is by creating a join entity called ManagerTeamHistory. You definitely need the keys of the entities it’s joining. You can also of course have their navigation properties. I didn’t include navigation properties.

Note that ManagerTeamHistory has a parameterized constructor and no parameterless ctor. This works out because the parameters in the constructor match the properties.

1. Run test **CanStoreAndRetrieveManagerTeamHistory**. Oh my, it failed!
2. Modify mapping for ManagerTeamHistory (which we’ve told EF Core to ignore thus far.)  
   Comment out the Ignore method and add a mapping to specify the composite key of the join entity.

// modelBuilder.Ignore<ManagerTeamHistory>();

modelBuilder.Entity<ManagerTeamHistory>()  
 .HasKey(m=>new{m.ManagerId,m.TeamId});

1. Create a new migration to verify that it will map to the database correctly

dotnet ef migrations add manytomanymanagerteam -s ../console

*With 2.1, this migration fails (most likely because of the changing constraints and my weird model) though it was OK in 2.0 and its OK again in 2.2 (preview). I’ve raised this on GitHub. But I wanted you to see this, so if you ran into it you won’t panic. 2.2 requires updated .NET Core SDK (1.5 also preview) so a quicker solution is:*

1. Delete the migrations folder completely and run the migration again. It succeeds because it’s from scratch, not trying to work out the constraint changes

Note that a join table is created for ManagerTeamHistory.

So that’s good, but the **CanStoreAndRetrieveManagerTeamHistory** test will still fail.

That’s because the List is null. We need to instantiate it in the Manager class (good practice anyway.)

Note that there is already a private ctor in Manager. That’s because the parameterized ctor does not align with the properties, so EF Core can’t use that for materialization.

1. To fix, modify the parameterless ctor in Manager to instantiate the PastTeams list:

private Manager () {

PastTeams = new List<ManagerTeamHistory> ();

}

1. Change signature of other ctor to call that via a cascading constructor:

public Manager (string firstname, string lastname) **: this ()** {

1. Run the test again and this time it should pass.
2. *If you open the AFTER branch for this exercise, you will find that there is a test class for SQLite tests. The tests are the same as in InMemory and they will all pass*

SIDE NOTE ABOUT MANAGER.CS  
I discovered a logic blunder in how team history worked in the manager class. OMG *SO* embarrassed.  
Therefore in this branch, Manager has new methods and team.ChangeManagement has been refactored!

Whoops! (I did not want to go back and change all of the other branches. ☺)

Note there are new tests validating the revised logic.

**Exercise F) ENCAPSULATE THE ONE-TO-ONE (Team: Manager)**

Note that EF Core convention has less rules to discover 1:1 mappings whereas earlier (EF6 etc) required navigation properties on both ends as well as a mapping to identify the principal and dependent ends of the relationship. Team and Manager have a 1:1 relationship where manager is the dependent.

The goal is to totally encapsulate the Team.Manager navigation property which has a 1:1 relationship with Team. This exercise has a new InMemory test: **CanStoreAndRetrieveTeamManager** to prove you can do that and continue to store and retrieve the manager. However as you start, it will be hidden from the compiler. I have my reasons. ☺ This exercise might test your patience, but I want you to be familiar with the problems you may run into, so I’m not taking a direct happy path. You’ll thank me someday.

1. In the Team class: 1) create a private field \_manager, 2) make the Manager property private and return the \_manager field and 3) create a public property for manager name that returns the Name property of the \_manager field.

private Manager \_manager;

private Manager Manager =>\_manager ;

public string ManagerName=>\_manager.Name;

1. Replace all 4 instances of Manager with \_manager in **ChangeManagement** method

Next, we need to fix some of the tests to align with this change so that the solution will build.

1. In the *Domain* Tests, **CanAddManager** and **CanReplaceManager** are trying to use Manager.Name. Manager is private now, but we have a public ManagerName property. So switch the code to use that instead.

Assert.Equal(newmanager.Name,team.ManagerName);

Run just those two domain tests. They should pass.

Note: Due to the same problem noted earlier, this broke for my model (not simpler models) with EF Core 2.1 but is working again with 2.2 (preview). Using 2.2 preview also requires having the preview of .NET SDK 2.1.5. I can do the rest of these demos for the class. Or you can just delete the migrations folder and start from scracth

Now let’s see how EF Core handles this.

1. Add a migration

dotnet ef migrations add onetooneprivate -s ../console

Notice there are some disruptive changes. It’s removing ManagerId from team and leaving no connection between manager and team. (If you started from scratch, then you just won’t see a ManagerId in the migration.)

1. Undo it

dotnet ef migrations remove -s ../console

What’s going on? EF Core can’t find Manager now that it’s hidden.

1. We need to configure it.

modelBuilder.Entity<Team> ()

.HasOne (typeof (Manager), "Manager").WithOne ()

.HasForeignKey(typeof (Manager), "CurrentTeamId");

In this configuration you are also ensuring that manager is the “dependent” end of the relationship and forcing it to recognize CurrentTeamId as the FK property.

1. Run the add migration again.

This time it identified the relationship, removed the managerid FK from team and replaced it with a TeamId FK from manager. That is what we wanted.

*Now, you can move on to EF Core InMemory tests*

1. Change the conditional compiler for **CanStoreAndRetrieveTeamManager** to **#if true**

Notice it won’t even compile! On Line 138, it’s trying to Include(t=>t.Manager) but Manager is not a recognized property because it’s private. Noooooooo! (Don’t worry.)

Remember the old way of using Include…with strings? That will rely on the data model at runtime, not the compiler. So you can…

1. … replace the lambda for t=>t.Manager with “Manager”. That’s good.
2. Also fix the t.Manager.Name to t.ManagerName as you did with the domain tests
3. The same method exists in the SQLite tests. Change the conditional compiler to true and fix the include and name issues as above.

**Compiler is happy! Rejoice.**

1. Run all the tests again. Everything passes.

Note that with scalar properties, you can completely remove the property and rely 100% on its backing field. Check the “Fields without a property” section of this EF Core doc: <https://docs.microsoft.com/en-us/ef/core/modeling/backing-field>.

**SHADOW PROPERTIES**

**Defining database only data that’s not needed in the domain model.**

1. Open team.cs and note that there are no properties that are of type DateTime.
2. Open TeamContext.cs and note the 2 new lines (27 & 28) defining two DateTime properties for Team: Created & LastModified
3. Open the only migration file (initwithShadow) and notice that the team table has those two date time columnms defined.

EF now knows how to persist and read the data from the database.

But how do you populate that data?

1. Back in TeamContext, look at the overridden SaveChanges method. Prior to saving team data, the code updates the ModifiedDate via ChangeTracker entries for any new or modified team entries. Additionally, if that entry is new (Added), then the CreateDate value is populated.

The SQLIte test: CanStoreAndRetrieveHomeColors, will add a new team to the SQLIte database. If you have a SQLite viewer (e.g. the VS Code SQLite extension) run the test and then you can see the populated field in the database.

**Some resources:**

MSDN Magazine Sept 2017: Data Points - DDD-Friendlier EF Core 2.0  
<https://msdn.microsoft.com/magazine/mt842503>

MSDN Magazine Oct: Data Points - DDD-Friendlier EF Core 2.0 Part 2  
<http://msdn.microsoft.com/magazine/mt826347>

docs.microsoft.com/ef

**My blog:**

Thedatafarm.com/blog

**My contact info:**

Twitter.com/julielerman

[julielerman@gmail.com](mailto:julielerman@gmail.com)

github.com/julielerman

github.com/julielerman/dddeu

I put this document at a private location on Dropbox:

http://juliel.me/2EuAm0e