

WEB524

WEB PROGRAMMING ON WINDOWS

WEEK 8 - LECTURE 1

CREATING A PERSISTENT STORE

Review Weeks 2 - 6

- We used the Chinook sample database consisting of several entities (for example: customer, employee, invoice, track, album, etc.).
- This sample database was included in the project template (Web App Project Template v1).
- In the Entity Models folder:
 - There are several classes for real-world entities and their associations with other classes.
 - The design classes were created using a Visual Studio code generator by examining the structure of an existing database (Chinook).
 - There is a code file that defines a data context - the app's gateway to the persistent store (database).
 - The data context includes `DbSet<TEntity>` collection property definitions for each entity in the design model. These properties are used by the Manager class methods to perform typical data service tasks (fetch, add, edit, and delete).

Review Week 7

- You were introduced to security.
- You created a new project that used the built-in Visual Studio MVC Web App template configured to use “Individual User Accounts”.
- You saw that there were no design model classes. Visual Studio creates a new web app with the foundational components and expects that you will add your own data model.
- You saw that the data context class was located in an IdentityModels.cs code file. A web app with security needs the foundational components needed for identity management, authentication, and authorization. This source code file configures some of these components, including the data context.

Review of Persistent Storage in MVC

- In an ASP.NET MVC web app, persistent storage for the data model is provided by the Entity Framework.
- You should, review the week 2 lecture notes. They introduced:
 - Persistent data storage in a database
 - Facade service (Entity Framework)
 - Design model classes
- You have created web apps that included those components as part of the new-project template.
- You can create and configure these components yourself, but you don't need to in this course. But...

Creating your own Persistence Layer

If you're interested in **creating your own persistence layer and adding to an existing project**:

1. Using the NuGet Package Manager Console,
install-package entityframework
install-package automapper
2. Add a connection string to the web app's *Web.config* file.
3. Write one or more design model classes.
4. Write a data context class (that inherits from *DbContext*) and add *DbSet<TEntity>* properties for the design model classes that will be persisted.
5. Add a "manager" class that will handle data service tasks for the app.

After the above steps, you will have a project like template provided by your teacher.

New Project Template with Authentication

- A new project with security has source code files and classes that are slightly different than those you worked with earlier in the course:
 - There are no design model classes
 - The data context class is in the IdentityModels.cs code file
- This kind of project is ready for your own data model. You will need to:
 1. Use the NuGet Package Manager Console, install-package automapper.
 2. Write one or more design model classes
 3. In the data context class, add DbSet<TEntity> collection properties for the design model classes.
 4. Add a “manager” class that will handle data service tasks for the app.
- This is still a lot of work! Your teacher has created another template that captures this work

Web App Project v2 template

- This new template includes starter code for the tasks outlined in the previous slide. You will need to do tasks 2, 3 and 4.
- Prepare your computer and test the new template.
 1. Download the template and add to the templates folder (just like you did for v1).
 2. Create a new throw-away project based on the new template. Build/compile and then run the web app.
 3. When the web app loads, the home page outlines two important tasks:
 - Customize the role claims.
 - Write the design model classes.
 4. A full tour of the web app's features are highlighted in the Task List. Open it to view the list of "TODO" comment tokens.

Web App Project v2 template

Register

- If you run the “Register” action, you will notice that the form includes generic or placeholder names for the role claims.

Web app project V3 Home Register Log in

Register.

Create a new account.

Email address

Password

Confirm password

Given (first) name(s)

Surname (family name)

Role(s) - select one or more

- ☐ RoleOne
- ☐ RoleTwo
- ☐ RoleThree

Register

Customize the names of the roles, to match the needs of your web app

© 2016 - BT1420 and INT422 Faculty

Web App Project v2 template Roles

- The roles, e.g. RoleOne, RoleTwo, RoleThree, will not show up on the page until you load them in the persistent storage.
- In the project template, they are defined in the GET (*form-building*) Register() method in the account controller:

```
// Define your custom role claims here
// However, in a real-world in-production app, you would likely maintain
// a valid list of custom claims in persistent storage somewhere
Manager m = new Manager();
var roles = m.RoleClaimGetAllStrings();

// Define a register form
var form = new RegisterViewModelForm();
form.RoleList = new MultiSelectList(roles);

// Send it to the view
return View(form);
```

Web App Project v2 template Roles

- As it states in the code comments, in a real-world in-production app, you would likely maintain a valid list of custom claims in persistent storage.
- A general approach is to define an entity set **RoleClaim** in your design model and use it as a “lookup table”.

Defining and Creating a New Persistent Store

CustomStoreForCarBusiness Code Example

- Open and study the **CustomStoreForCarBusiness** code example.
- In the DesignModelClasses.cs source code file, write classes for your domain objects.
- Assume that our problem domain is the familiar car business. It has classes for Country, Manufacturer, and Vehicle.
- A country has zero or more manufacturers.
- A manufacturer has a collection of vehicles.
- A vehicle is associated with one manufacturer.

Design Model Class - Conventions

- Remember these rules – conventions – when writing design model classes. They will ensure that the database is initialized and configured correctly, and will improve the quality of other coding tasks as you build the app.
 - To expedite other coding tasks, the name of the integer identifier property should be **id**.
 - Collection properties (including navigation properties) must be of type **ICollection<T>**. Initialize as **HashSet<T>**.
 - Valid data annotations are pretty much limited to **[Required]** and **[StringLength(n)]**.
 - When defining an association between two classes, navigation properties MUST be configured in *both* classes
 - Required “to-one” navigation properties must include the **[Required]** attribute
 - Do NOT configure scalar properties (e.g. int, double) with the **[Required]** attribute
 - Initialize **DateTime** and collection properties in a default constructor

Reminder about “virtual”

- A few weeks ago, you were told that you must NOT use the “virtual” keyword when defining navigation properties in your design model classes. Do you remember why?

Reminder about “virtual”

- The “virtual” keyword partly enables the lazy-loading feature. As you will remember, we do not and must not use it in web apps.
- By default, our web apps deactivate this feature.

Design Model Class Diagram

- After you write your design model classes, you should create a class diagram.
- It will give you a visual way to verify that you have coded the design model classes correctly.
- After completing the design model classes, add `DbSet<TEntity>` properties to the `IdentityModels.cs` source code file.

Database Creation

- When does the database get created by the database engine?
- Recall that the data context class is the “gateway” to the persistent data store.
- By convention, the first time a database is accessed, read or written to, the data context calls a “database initializer”.
- The database initializer that is called will create the database if it does not exist.
- If the database does exist, then the database initializer will see it and will return without making any changes.

Adding New Objects to the Persistent Store

Adding New Objects to the Persistent Store

- After defining a data store, we can programmatically add new objects.
- In the ASP.NET and MSDN documentation, the “initializer” strategy is discussed, demonstrated, and recommended. It is a convention-based strategy, that uses a `Seed()` method to “seed” the data store with new objects.
- We have used this in past versions of this course.
- Going forward, we will use a new strategy that is a bit simpler to understand and is compatible with a future “bulk data load” topic.

Using Manager to Add New Objects

- In this strategy, we can code one or more Manager class methods that “load” programmatically-generated objects into the data store.
- The “...V2” project template includes one method example, that you can edit and use:

```
public bool LoadData ()
{
    // Return if there's existing data
    //if (ds.Your_Entity_Set.Count() > 0) { return false; }

    // Otherwise...
    // Create and add objects
    // Save changes

    return true;
}
```

Using Manager to Add New Objects

- Next, we will create a special-purpose (or throw-away) “LoadData” controller that calls into the Manager methods.
- As the programmer, you can choose whether to “protect” the controller (and its methods) with an Authorize attribute.
- You can choose to keep or delete the “LoadData” controller after it has done its job.

Summary of the Strategy

In summary, the strategy includes:

1. Adding one or more Manager methods that add programmatically-generated objects into the data store.
2. Create a special-purpose controller that calls into the Manager method(s).
3. Run the controller action(s)/method(s).

Making Changes to the Design of the Persistent Store

Making Changes to the Design of the Persistent Store

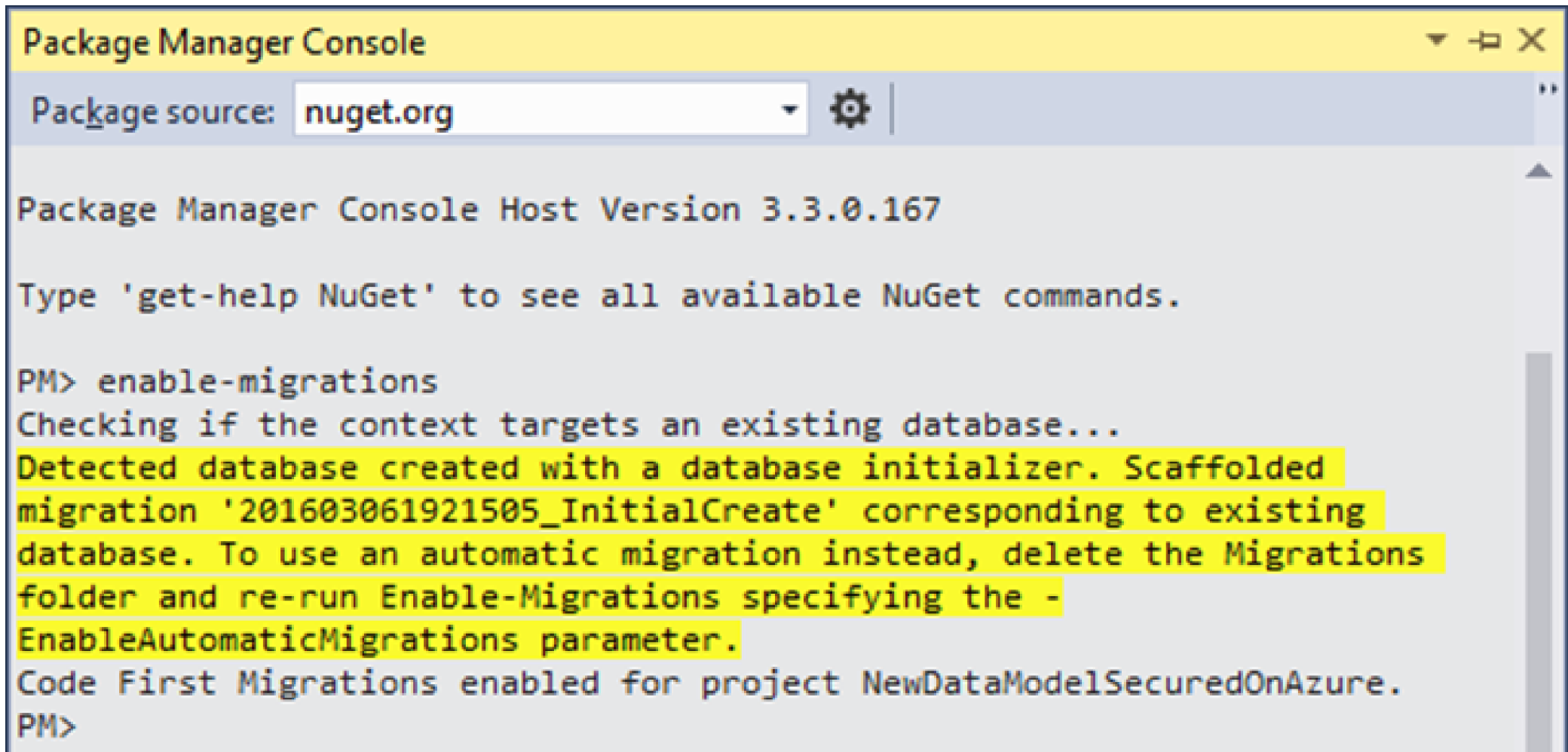
- After an app is up-and-running, how do you handle a situation where you need to add another entity class to a design model? Or, add a property to a design model class? Or change a property's configuration?
- This can be done if you activate a feature named (Code First) Migrations. Its biggest benefit is that it will attempt to keep your existing data after minor changes to the design model and database.
- It also helps us to publish our web app to a public host.
- Changes to an existing property's configuration will work correctly for data transformations and conversions that are implicit. These “widening” transformations will work without any additional code or data preparation.
- For our changes, let's attempt to ensure that our changes are implicit in nature.

Configuring Migrations

- Configuring Migrations requires two simple tasks:
 1. Enable migrations.
 2. After a change (or set of changes), add a “migration” definition and update the data store.

How to Enable Migrations:

- In Visual Studio, open the Package Manager Console.
- Type the following command. After you do this, you will not have to do it again for the current project: *enable-migrations*
- This command configures the base, or start state, of your design model classes and persistent store implementation. The following is a typical result from running the “enable-migrations” command:

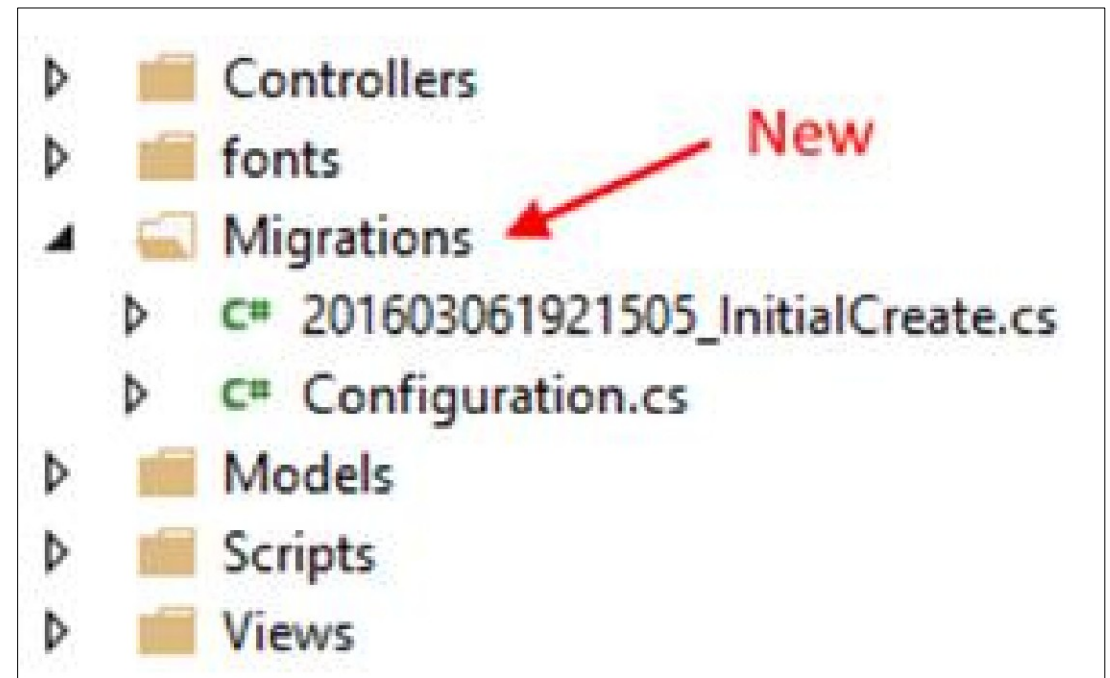


The screenshot shows the Package Manager Console window with a yellow title bar. The package source is set to 'nuget.org'. The console output shows the command 'enable-migrations' being executed, followed by a message indicating that a migration was scaffolded because an existing database was detected. The message suggests deleting the Migrations folder and re-running the command with the 'EnableAutomaticMigrations' parameter. The console ends with the prompt 'PM>'.

```
Package Manager Console
Package source: nuget.org
Package Manager Console Host Version 3.3.0.167
Type 'get-help NuGet' to see all available NuGet commands.
PM> enable-migrations
Checking if the context targets an existing database...
Detected database created with a database initializer. Scaffolded
migration '201603061921505_InitialCreate' corresponding to existing
database. To use an automatic migration instead, delete the Migrations
folder and re-run Enable-Migrations specifying the -
EnableAutomaticMigrations parameter.
Code First Migrations enabled for project NewDataModelSecuredOnAzure.
PM>
```

Enabling Migrations

- After running the command, look in the Solution Explorer.
- You will notice a new “Migrations” folder and two new code files.
- The classes in these code files are used by the Entity Framework data migrator process and by the Microsoft Azure services publishing process.
- Typically, we do not edit these classes.



Updating Migration Definitions

- If you have existing data and have not made a change to your design model classes then you can skip the second task.
- Read the package manager console messages carefully – it will tell you the status.
- Later, whenever you have made a change to your design model classes, do the second task by executing the following two commands in the Package Manager Console:

PM> add-migration descriptive_name_for_the_change

PM> update-database

- If you anticipate performing many changes then you can do the add-migration and update-database several times, or you could just make all your changes and run these commands once.

Migration Notes

- Working with migrations is simple but it is easy to make it more complicated than it is.
- Follow the guidance in this slide.
- There's some reference information that you could read/skim, to learn more about the technology:
 - [Introduction and overview – Code First Migrations](#)
- What can you do if you run into trouble while using/configuring migrations?
Try this, before contacting your teacher:
 1. In Visual Studio Solution Explorer, delete the Migrations folder.
 2. Do the first task described above – enable-migrations – which will re-write/re-create the migration code.

Major Design Model Changes

- If the data transformation/conversion is “narrowing” in nature, which could cause loss of data or precision, then you must inspect and/or prepare the data before and after the property configuration change.
- For example, if you plan to add a [StringLength(200)] attribute to a string property that did not have such an attribute already, it is possible that some objects in the data store will have property values that exceed 200 characters in length.
- Before editing the property configuration, you must locate, and potentially edit/truncate, the existing data, so that the property configuration change will succeed.
- For example, if you plan to convert a double property into an integer property, it is possible that some existing objects will have real numbers (with a decimal portion) in the property value. Before editing the property configuration, you must locate and edit the existing data.

Alternative Strategy

- An alternative strategy, which works for existing in-production apps, would involve several tasks:
 1. Add a new property, which will hold the destination value.
 2. As a special task, go through all objects in the entity set, and use data from the existing property to populate the desired value in the new property.
 3. If you want to start using the new property in your manager and controller code, go ahead.
 4. Alternatively, you could delete the old property, add it back again with the desired data type, go through all the objects in the entity set to copy from the property added in step 1 above, to the just-added-again property, and then finally, delete the property added in step 1 above

Real-world in-production handling of database changes

- Real-world in-production apps tend to separate the persistence layer from the application.
- In this technique, we create a “class library” project that defines and hosts the design model classes and data context. This action creates a “persistence layer” (aka “data layer” in some documentation sets).
- In a web app project, we “add a reference” to the persistence layer, so that our Manager class can work with the data.
- When we need changes to the database, then we can use traditional database administration tools to perform the modifications or Migrations.
- If not using Migrations, we modify the code in the persistence layer project to match the new reality in the persistence layer.