Contents

[What is a Model? 2](#_Toc168208400)

[Working with a Database 3](#_Toc168208401)

[Database Context Class 3](#_Toc168208402)

[The Connection String 3](#_Toc168208403)

[EFCore Migrations 4](#_Toc168208404)

[Add-Migration 4](#_Toc168208405)

[Update-Database 5](#_Toc168208406)

[CRRUD 6](#_Toc168208407)

[DbContext Injection 6](#_Toc168208408)

[ReadAll 6](#_Toc168208409)

[Create 7](#_Toc168208410)

[Read 9](#_Toc168208411)

[Update 10](#_Toc168208412)

[Delete 12](#_Toc168208413)

# What is a Model?

* A model is an abstraction of the objects from the problem domain.
* It stores the data elements and how to process them.
* For example, let us assume that our problem domain involves tracking video games.
  + Using the UML, the model for this domain is a VideoGame class:

|  |
| --- |
|  |
|  |

* Using C#, the model is then:

|  |
| --- |
| public enum RatingCriteria  {  LoveIt, LikeIt, NotForMe  }  public class VideoGame  {  public int Id { get; set; }  public string Title { get; set; } = string.Empty;  public string Developer { get; set; } = string.Empty;  public string Genres { get; set; } = string.Empty;  public RatingCriteria Rating { get; set; }  } |

# Working with a Database

## Database Context Class

* The database context class is the bridge into the actual database.
* You interact with the database through this class.

|  |
| --- |
| public class ApplicationDbContext : IdentityDbContext  {  public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)  : base(options)  {  }  **public DbSet<VideoGame> VideoGames => Set<VideoGame>();**  } |

# The Connection String

* You will need to set up the database connection string.
* Best practice dictates that it is stored externally of the source code
  + e.g., in the configuration file ‘appsettings.json.’

|  |
| --- |
| ***appsettings.json*** |
| {  "ConnectionStrings": {  "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=**VideoGameDb**;Trusted\_Connection=True;MultipleActiveResultSets=true"  },  "Logging": {  "LogLevel": {  "Default": "Information",  "Microsoft.AspNetCore": "Warning"  }  },  "AllowedHosts": "\*"  } |

* The connection string needs to be linked to the DbContext.
  + This is configured in *Program.cs*.

|  |
| --- |
| ***Program.cs*** |
| var builder = WebApplication.CreateBuilder(args);  **var connectionString = builder.Configuration.GetConnectionString("DefaultConnection")**  **?? throw new InvalidOperationException("Connection string 'DefaultConnection' not found.");**  **builder.Services.AddDbContext<ApplicationDbContext>(options =>**  **options.UseSqlServer(connectionString));**  … |

# EFCore Migrations

* We use EFCore migration to manage the process of transforming the model classes into the database schema.
* We must add migrations to be able to generate and update the database from code.

## Add-Migration

* We use the Add-Migration command to set up the migration.
* To add a migration:
  + Give the migration a name.
  + In this example, the name is ‘*Mig01*’.

|  |
| --- |
| **PM> Add-Migration Mig01** |

* If the command is successful, a *Migrations* folder is created along with the classes needed to perform the migration.

|  |
| --- |
|  |

## Update-Database

* We use the Update-Database command to update the database based on the added migration.

|  |
| --- |
| **PM> Update-Database** |

* **Tip**: Use the **SQL Server Object Explorer** to view the database

|  |
| --- |
|  |

# CRRUD

* CRRUD is an acronym for the operations: **Create**, **Read**, **Read All**, **Update**, and **Delete**.
* These operate on data in the data store.
  + Create (or insert) a new object into a data store
  + Read an object or many objects
  + Read all objects
  + Update (or edit) an existing object
  + Delete (or remove) an existing object.

## DbContext Injection

* We inject the DbContext class into the controller.

|  |
| --- |
| public class HomeController : Controller  {  private readonly ILogger<HomeController> \_logger;  **private readonly ApplicationDbContext \_db;**  public HomeController(ILogger<HomeController> logger, **ApplicationDbContext db**)  {  \_logger = logger;  **\_db = db;**  }  …  } |

# ReadAll

* The ReadAll implementation reads all records from the *VideoGame* table and returns them as a collection.
* The Index page is typically a collection of objects from the domain. For our example, a collection of VideoGames.
* The HomeController Index:
  1. Reads the video games model
  2. Sends the model to the view

|  |
| --- |
| public async Task<IActionResult> Index()  {  var allVideo games = await \_VideoGameRepo.ReadAllAsync();  return View(allVideo games);  } |

* Scaffold a Razor View
  + View name: **Index**
  + Template: **List**
  + Model class: **VideoGame**
  + DbContext class: ApplicationDBContext

|  |
| --- |
|  |
|  |

# Create

* The Create implementation involves:

1. Add the object to the database using the database context
2. Commit the changes
3. Return the object

* In the controller, creating a VideoGame involves the following actions:

1. Presenting a form to accept data - GET: /Home/create
2. Processing the submitted form - POST: /Home/create (with post data)
   1. If the model is valid
      1. Add it to the database
      2. Redirect to the index view
   2. If the model is invalid
      1. Return the create view

|  |
| --- |
| public IActionResult Create()  {  return View();  }  [HttpPost]  public IActionResult Create(VideoGame newVideoGame)  {  if (ModelState.IsValid)  {  \_db.VideoGames.Add(newVideoGame);  \_db.SaveChanges();  return RedirectToAction("Index");  }  return View(newVideoGame);  } |

* Scaffolding the GET Create Razor View
  + View name: **Create**
  + Template: **Create**
  + Model: **VideoGame**
  + Update the input for Rating to use a drop down list:

|  |
| --- |
| <div class="form-group">  <label asp-for="Rating" class="control-label"></label>  **<select asp-for="Rating" class="form-select">**  **<option value="0">Love It</option>**  **<option value="1">Like It</option>**  **<option value="2">Not For Me</option>**  **</select>**  <span asp-validation-for="Rating" class="text-danger"></span>  </div> |

|  |
| --- |
|  |
|  |

# Read

* The Read implementation retrieves an object from the database.
* We will read VideoGames using their Ids:
  + Find(id)
    - Returns the object found given the primary key.
    - If the object is not found, then null is returned.
* HomeController Details(int id) 🡪 GET: /home/details/***{id number}***

1. Read the VideoGame from the database.
2. If the VideoGame is not found
   1. Make a GET request to Index.
3. If the VideoGame is found
   1. Make a GET request to Details.

|  |
| --- |
| public IActionResult Details(int id)  {  var videoGame = \_db.VideoGames.Find(id);  if (videoGame == null)  {  return RedirectToAction("Index");  }  return View(videoGame);  } |

* Scaffolding the Details Razor View
  + View name: **Details**
  + Template: **Details**
  + Model: **VideoGame**

|  |
| --- |
|  |

# Update

* The Update implementation has three major steps:

1. Use the old id to read the object to update
2. Update the properties
3. Commit the changes

* The steps are:

1. Present a form with the values filled in.
2. Process the submitted form.

* HomeController Edit(int id) maps to GET: /home/edit/{***id number***}

1. Read the video game from the database
2. If no video game was found, redirect to the video game index page.

* HomeController Edit(VideoGame videoGame) maps to POST: /home/edit (with post data)

1. If the model is valid
   1. Update the VideoGame in the database
   2. Redirect to the VideoGame index page
2. If the model is invalid
   1. Return the view with the model

|  |
| --- |
| public IActionResult Edit(int id)  {  var videoGame = \_db.VideoGames.Find(id);  if (videoGame == null)  {  return RedirectToAction("Index");  }  return View(videoGame);  }  [HttpPost]  public IActionResult Edit(VideoGame videoGame)  {  if (ModelState.IsValid)  {  var gameToUpdate = \_db.VideoGames.Find(videoGame.Id);  gameToUpdate!.Title = videoGame.Title;  gameToUpdate!.Developer = videoGame.Developer;  gameToUpdate!.Genres = videoGame.Genres;  gameToUpdate!.Rating = videoGame.Rating;  \_db.SaveChanges();  return RedirectToAction("Index");  }  return View(videoGame);  } |

* Scaffolding the edit razor view
  + View name: **Edit**
  + Template: **Edit**
  + Model class: **VideoGame**
  + **Important:** Look for the hidden input for Id:<input type="hidden" asp-for="Id" />
  + Change the Rating input:

|  |
| --- |
| <div class="form-group">  <label asp-for="Rating" class="control-label"></label>  **<select asp-for="Rating" class="form-select">**  **<option value="0">Love It</option>**  **<option value="1">Like It</option>**  **<option value="2">Not For Me</option>**  **</select>**  <span asp-validation-for="Rating" class="text-danger"></span>  </div> |

|  |  |
| --- | --- |
|  | |
|  |  |
|  | |

# Delete

* The Delete implementation has three steps:

1. Retrieve the object to delete
2. Delete the object from the database
3. Commit the changes

* Deleting a VideoGame involves the following:

1. Presenting the UI to confirm the delete
2. Processing the confirmation

* HomeController Delete(int id) maps to GET: /home/delete/***{id number}***

1. Read the VideoGame from the database
2. If the VideoGame was not found, redirect to the VideoGame index view
3. If the VideoGame was found, return the delete view

* HomeController DeleteConfirmed(int id) 🡪 POST: /home/delete (with post data)

1. Delete the VideoGame from the database using the id
2. Make a GET request to the index page (redirect)
3. **Note**: the name of the action method is DeleteConfirmed, and we use ActionName to set the name to Delete

|  |
| --- |
| public IActionResult Delete(int id)  {  var videoGame = \_db.VideoGames.Find(id);  if (videoGame == null)  {  return RedirectToAction("Index");  }  return View(videoGame);  }  [HttpPost, ActionName("Delete")]  public IActionResult DeleteConfirmed(int id)  {  var videoGame = \_db.VideoGames.Find(id);  \_db.VideoGames.Remove(videoGame!);  \_db.SaveChanges();  return RedirectToAction("Index");  } |

* Scaffolding the Razor View
  + View name: **Delete**
  + Template: **Delete**
  + Model class: **VideoGame**
  + **Verify the hidden input for Id in the form (see below)**

|  |
| --- |
| *…*  <form asp-action="Delete">  **<input asp-for="Id" type="hidden" />**  <input type="submit" value="Delete" class="btn btn-danger" /> |  <a asp-action="Index">Back to List</a>  </form>  … |

|  |
| --- |
|  |
|  |
|  |