

MovieDbLite

Database Project



April 22, 2020

Steven Anderson

Databases 605.641

[**https://github.com/stevo9510/MovieDbLiteMVC**](https://github.com/stevo9510/MovieDbLiteMVC)

Contents

[1. Introduction 3](#_Toc38167344)

[1.1. Scope and Purpose of Document 3](#_Toc38167345)

[1.2. Project Objective 3](#_Toc38167346)

[2. System Requirements 4](#_Toc38167347)

[2.1. Hardware Requirements 4](#_Toc38167348)

[2.1.1. Database Server Requirements 4](#_Toc38167349)

[2.1.2. Web Application Server Requirements 5](#_Toc38167350)

[2.1.3. Website Front-End Access Requirements 5](#_Toc38167351)

[2.2. Software Requirements 5](#_Toc38167352)

[2.2.1. Database Server Requirements 6](#_Toc38167353)

[2.2.2. Web Application Server Requirements 6](#_Toc38167354)

[2.2.3. Website Front-End Requirements 6](#_Toc38167355)

[2.3. Functional Requirements 6](#_Toc38167356)

[2.4. Database Requirements 7](#_Toc38167357)

[3. Database Design Description 8](#_Toc38167358)

[3.1. Design Rationale 8](#_Toc38167359)

[3.2. E/R Model 12](#_Toc38167360)

[3.2.1. Entities 12](#_Toc38167361)

[3.2.2. Relationships 21](#_Toc38167362)

[3.2.3. E/R Diagram 26](#_Toc38167363)

[3.3. Relational Model 31](#_Toc38167364)

[3.3.1. Data Dictionary 31](#_Toc38167365)

[3.3.2. Integrity Rules 36](#_Toc38167366)

[3.3.3. Operational Rules 37](#_Toc38167367)

[3.3.4. Operations 37](#_Toc38167368)

[3.4. Security 38](#_Toc38167369)

[3.4.1. Password Storage 38](#_Toc38167370)

[3.4.2. SQL Injection 39](#_Toc38167371)

[3.4.3. Database Connection Strings 40](#_Toc38167372)

[3.4.4. Network Connection 41](#_Toc38167373)

[3.5. Database Backup and Recovery 41](#_Toc38167374)

[3.5.1. Automated Database Backups 41](#_Toc38167375)

[3.5.2. Full Database Recovery Model 42](#_Toc38167376)

[3.6. Using Database Design or CASE Tool 42](#_Toc38167377)

[3.6.1. Visual Paradigm v15.2 43](#_Toc38167378)

[3.6.2. SQL Management Studio (SSMS) v17.8.1 43](#_Toc38167379)

[3.6.3. Visual Studio 2019 45](#_Toc38167380)

[3.6.4. GitHub for Git-Based Source Control 47](#_Toc38167381)

[3.7. Other Possible E/R Relationships 49](#_Toc38167382)

[4. Implementation Description 51](#_Toc38167383)

[4.1. Data Dictionary 53](#_Toc38167384)

[4.2. Advanced Features 57](#_Toc38167385)

[4.2.1. Triggers 57](#_Toc38167386)

[4.2.2. Views 58](#_Toc38167387)

[4.2.3. Table-Valued Functions 58](#_Toc38167388)

[4.2.4. User-Defined Table Types 61](#_Toc38167389)

[4.2.5. Stored Procedures 62](#_Toc38167390)

[4.2.6. Constraints, User-Defined Data Types, and Rules 66](#_Toc38167391)

[4.2.7. Naming Standards and Conventions 68](#_Toc38167392)

[4.2.8. Importing Data from TheMovieDb.org API 69](#_Toc38167393)

[4.2.9. MovieDbLite Website 71](#_Toc38167394)

[4.2.10. Indexes 77](#_Toc38167395)

[4.3. Queries 78](#_Toc38167396)

[4.3.1. Movie Details Report (Primary Details with Languages/Genres) 78](#_Toc38167397)

[4.3.2. All Cast and Crew Member Names and Roles 80](#_Toc38167398)

[4.3.3. Movie and Role Information for Film Member 81](#_Toc38167399)

[4.3.4. Retrieve User Movie Reviews/Ratings 81](#_Toc38167400)

[4.3.5. Retrieve Helpful Vote Details of a Particular Review 82](#_Toc38167401)

[4.3.6. Retrieve Film Members in Multiple Movies/Roles 83](#_Toc38167402)

[4.3.7. Award Winners for a Particular Year 84](#_Toc38167403)

[4.3.8. Awards for a Particular Movie 84](#_Toc38167404)

[4.3.9. Awards for a Particular Film Member 85](#_Toc38167405)

[4.3.10. Director Candidates 85](#_Toc38167406)

[4.3.11. “You may know this film member from…” information 86](#_Toc38167407)

[5. CRUD Matrix 87](#_Toc38167408)

[5.1. List of Entity Types 88](#_Toc38167409)

[5.2. List of Functions 89](#_Toc38167410)

[6. Concluding Remarks 90](#_Toc38167411)

[6.1. Appendices 92](#_Toc38167412)

[6.2. Appendix A - DDL, INSERT, SELECT Statements 92](#_Toc38167413)

[6.3. Appendix B - Data Dictionary Index 93](#_Toc38167414)

[6.4. References 95](#_Toc38167415)

# Introduction

Movies are a common form of entertainment in the lives of many people. With the evolution of the internet and various streaming services, the access to films are more available than ever. The MovieDbLite project aims to be simple, easy-to-use website for viewing the details about movies and credits, as well as allowing users to rate and review the movies they have seen. In alignment with the website, the database itself is intended to be simple and intuitive for people with database experience.

For myself personally, I am a huge movie buff, and (prior to COVID-19) still went to the theaters multiple times a month with my wife to watch new releases. I chose this project because of my interest in movies and the people involved in delivering these films. In addition, the ownership of a movie database allows for me to personalize the reports or features that may not be found in other popular movie database applications. Furthermore, it opens up the possibility to create APIs (Application Programming Interfaces) for consumption of the data by other systems, use cases, and applications (e.g. trivia games). This project also allowed me to gain experience retrieving data from existing Movie database APIs, such as [www.TheMovieDb.org](http://www.TheMovieDb.org). In addition, I also was able to deliver a front-end website that communicates with the database. It implements a few of the features you’d expect from a movie website (such as searching/viewing details about movies and film members).

One of the main things I would like to highlight here is that I have video recorded demonstrations of the API data exchange and website, and those can be found in these respective advanced feature sections: **4.2.8 Importing Data from TheMovieDb.org API** and **4.2.9 MovieDbLite Website**

## Scope and Purpose of Document

The purpose of this document is to describe design, development, requirement, and implementation details of the MovieDbLite database and website application, with the primary focus being on the database that is used to store movie and film industry data. Specifically, when it comes to the design of the database, the focus is on the conceptual and logical design and implementation. An existing, industry-standard Relational Database Management System (RDBMS) environment that handles the physical implementation of the database will be used. Therefore, lower-level details about physical storage and implementation of the database can be found via online resources of the respective RDBMS environment.

Overall, this document is bounded to describing the MovieDbLite database and front-end website for accessing basic details and reports of data within database. Accordingly, information about external applications that may consume the database data (e.g. through an API) is not intended to be covered by this document.

## Project Objective

The objective of this project is to design and implement a simple, yet useful movie database that tracks information related to the film industry. Further, the project intends to provide a front-end website to deliver an easy-to-use user interface for allowing users to view data and rate movies. This project shall be built using modern RDBMS and web-based technologies to captures the data elements and functional requirements that are mentioned in the upcoming sections. Ultimately, this database is intended to serve as a stable and maintainable back-end to enable future opportunities for APIs to be built for external applications to consume.

# System Requirements

The MovieDbLite is composed of three high-level system components that have their distinct set of system requirements:

1. The SQL Server hosted MovieDbLite Database
   1. Hosted on a remote server
2. The MovieDbLite Web Application Environment (Back-End)
   1. Hosted on a remote server
3. The MovieDbLite Website (Front-End)
   1. Accessed by an end-user via web browser

This section will cover the System Requirements (software and hardware) for the three major system components of the MovieDbLite system, as well as the functional requirements that make apart building these three components.

Although the database (system component #1) and web application (system component #2) are hosted remotely, this does not necessarily mean they will be hosted on the same remote server/machine, nor does it mean the requirements for them are the same. In other words, the database and web application may be hosted on different server environments. Therefore, the following sections may distinguish and discretely separate the requirements for these respective components as appropriate.

## Hardware Requirements

This section describes the hardware requirements for the three components of the MovieDbLite system. In some of the hardware listings, a minimum hardware requirement is listed, as well as a recommended. The minimum hardware requirement is meant to indicate the minimum possible hardware specification to run or host the component. The recommended listing is the suggested minimum to mitigate deterioration of component performance. Overall, the system will support hardware specifications with greater amounts than the minimum/recommended listings, which would subsequently increase or maintain component performance and/or reliability.

### Database Server Requirements

The database server will require hardware that is able to host SQL Server Standard Edition 2017. The full server hardware requirements can be found in Microsoft’s documentation [here](https://docs.microsoft.com/en-us/sql/sql-server/install/hardware-and-software-requirements-for-installing-sql-server?view=sql-server-ver15#hardware-requirements).

This is the overall breakdown of the hardware requirements for SQL Server Standard Edition:

* **Memory** - Minimum of 1GB, recommended of 4GB
* **Hard Drive**
  + Computer with [NTFS](https://searchwindowsserver.techtarget.com/definition/NTFS) or [ReFS](https://docs.microsoft.com/en-us/windows-server/storage/refs/refs-overview) file format for security reasons.
  + Minimum of 6GB of available hard-disk space
    - Note: [Hard Disk Space Requirements](https://docs.microsoft.com/en-us/sql/sql-server/install/hardware-and-software-requirements-for-installing-sql-server?view=sql-server-ver15#HardDiskSpace) will apply depending on which components are installed
* **Monitor** - Super-VGA (800x600) or higher resolution monitor
* **Internet** - Internet access is required
* **Processor**
  + Speed: Minimum of 1.4GHz, recommended of 2.0+ GHz
  + Type: x64 Processor (e.g. AMD Opteron, AMD Athlon 64, Intel Xeon with Intel EM64T support, Intel Pentium IV with EM64T support)
* **Drive** – A DVD drive is required if installation is from disc

### Web Application Server Requirements

The web application environment (back-end to the website) will require an environment that can host an ASP.NET Website targeting .NET Core v3.1. The baseline hardware requirements can be found [here](https://help.syncfusion.com/aspnet-core/installation-and-upgrade/system-requirements#hardware-environment).

* **Memory** - Minimum of 512MB, recommended of 1GB
* **Hard Drive**
  + Up to 2GB of available space may be required
  + 300MB of free space is required in boot drive
* **Internet** – Internet access is required
* **Processor**
  + x86 and x64 processors are supported
  + Although specifications on speed is not provided, modern web servers recommend processor of 2.0+ GHz

### Website Front-End Access Requirements

The end-user will require hardware compatible to run a modern web browser (such as Chrome, Firefox, Edge) in order to access the website.

The following are the minimum hardware requirements for Chrome. These are generally the minimum requirements amongst other browser types:

* **Hard Drive** – Minimum 100MB of free hard drive space is required
* **Memory** – Minimum 128MB of RAM
* **Internet** – Network access is required to connect
* **Processor** – Pentium 4 and above
* **Keyboard (or equivalent)** – Typing will be required to perform searches
* **Mouse (or equivalent)** – Clicking will be required to navigate the web pages

More information can be found at this [link](https://d.docs.live.net/df1cecd6e646b923/Documents/Grad_School/Databases_605_641/Project/•%09https:/smallbusiness.chron.com/google-chrome-software-requirements-48820.html).

## Software Requirements

This section covers the software required to operate the components of MovieDbLite. In addition to Software application requirements, this section may list the Operating System (OS) requirements – as appropriate.

### Database Server Requirements

The database server will require software installed to host SQL Server Standard Edition 2017. The full server software requirements can be found in Microsoft’s documentation [here](https://docs.microsoft.com/en-us/sql/sql-server/install/hardware-and-software-requirements-for-installing-sql-server?view=sql-server-ver15#hwswr).

In summary, the following two software components are required to be installed on the server hosting the database:

1. .NET Framework (v4.6)
2. Network Software that support the following protocols: Shared Memory, Named Pipes, TCP/IP, and VIA.

The following [link](https://docs.microsoft.com/en-us/sql/sql-server/install/hardware-and-software-requirements-for-installing-sql-server?view=sql-server-ver15#operating-system-support) outlines the supported OS versions to host SQL Server 2017. Overall, the operating system must be a Windows Server 2012+ (2012, 2016, 2019) environment. Although Windows 8 and Windows 10 are listed as well, these are not intended to be used for remote hosting of the database.

### Web Application Server Requirements

The web application environment (back-end to the website) will require an OS environment that can host an ASP.NET Website targeting .NET Core v3.1. The full supported OS versions can be found [here](https://github.com/dotnet/core/blob/master/release-notes/3.1/3.1-supported-os.md). This section will focus on the remote server environments hosted by Windows, to remain consistent with Windows hosted database environment using SQL Server. In addition, the security model used for authentication to SQL Server will utilize Integrated Security which is Windows specific. More information on the connection string security is mentioned in section 3.4.3 Database Connection Strings.

The following are the recommended OS environments:

* Windows Server 2012 R2+
  + x64 and x86
* Windows Nano Server 1803+
  + x64 and ARM32

### Website Front-End Requirements

A modern web browser that supports HTML5 is required for the end-user to access the website.

The following are the preferred browsers to access the website:

* Google Chrome
* Microsoft Edge
* Mozilla Firefox
* Safari

Note: Although some features of the website may work in Internet Explorer, it is recommended to use one of the listed browsers above for full support.

## Functional Requirements

The MovieDbLite website will contain basic functionality to allow users to view details about movies and members of the film industry. Users may also create user accounts and rate/review movies. Administrative access will be available to allow for modification of movie and film industry details to keep the database’s data current.

This section covers the different functional requirements of the MovieDbLite web application in detail below.

* Track movie/film basic information (e.g. title, description, release date, duration, etc.)
  + Note: Full data details to be described in 4.1 Data Dictionary section
* Track the different film members that are involved in the film industry, as well as basic information about them (preferred name, gender, date of birth, etc.)
* Track cast and crew members of a movie by their role (director, producer, actor, costume designer, choreographer, etc.)
* Track the different restriction ratings (e.g. G, PG, PG-13, R, etc.) that a film can be assigned to
* Track the different international languages (English, Spanish, German, etc.) that a film can be assigned to
* Track different annual award show events and the particular awards they give out
* Track the film members and movies that win awards at annual award show instances
* Track the different movie genres that a movie can be classified as
* Allow user to create a user account login (user name and password)
* Allow user access to modify details of their user account (user name, email address, password)
* Allow user to search for movies by movie title
* Allow user to perform an advanced movie search by other fields (description, release date, average user rating, restriction rating)
* Allow user to search for film members by their name
* Allow user to write a review and assign a user-rating (from 1-10) to a movie
* Allow user to mark a review as helpful or unhelpful
* Allow user to sort reviews by date, helpfulness, or rating
* Reports/Queries
  + Report a movie’s basic details (title, restriction rating, language, release date, etc.)
  + Report the film members (actors, producers, crew members, etc.) of a movie
  + Report the movies a film member has been a part of
  + Report the awards given to a certain movie
  + Report the awards given to a certain film member
  + Report the user reviews for a movie
  + Report the helpful vote details of a review
* Administratively allow adding/editing movie information
* Administratively allow adding/editing the film members of a movie
* Administratively allow inputting results from award show instances

## Database Requirements

The MovieDbLite Database is hosted using Microsoft’s SQL Server 2017 (Standard Edition). [Standard Edition](https://www.microsoft.com/en-us/sql-server/sql-server-2017-editions) will support up to 524 PB (Petabytes) of data, which is more than enough for the database. SQL Server was chosen itself for its maturity and longevity; it has been one of the most used RDBMS by enterprises for several years. It is full-featured and built for production-level usage. The tooling of Microsoft’s SQL Server also integrates nicely with Microsoft Visual Studio, which is also used to develop the application. There is potential in the future to upgrade to Microsoft SQL Server Enterprise Edition if it becomes necessary, however for cost-savings at the time Standard Edition was chosen.

The development/management environment used to perform DML and DDL operations against the database is Microsoft SQL Server Management Studio 17 (SSMS). The full version details of SSMS are seen below.



# Database Design Description

This section describes the details related to the database design of MovieDbLite. The overall design process started with analyzing the functional requirements of the application, and designing from a top-down approach (i.e. first identifying entities, and then attributes). During design, there were sometimes multiple potential design approaches/options to meet a specific functional requirement. Choosing an approach required weighing the pros and cons amongst the options. The upcoming section on design rationale will dive into some of the justifications amongst different approaches or standards that were adopted for the design. Later in this section the entities, relationships, and columns will be described in full detail.

## Design Rationale

One of the main objectives with the design, in addition to correctly and successfully modeling the requirements, was to try to be as consistent and standardized as possible. This includes how entities and their relationships were modeled, as well as the naming conventions used for the database elements. These are some general design decisions that were made, with the rationale behind them:

1. Do use a single primary key value for an entity if that entity is referenced by another entity (i.e. as a foreign key). The reasons for this are as follows:
   1. Simplify the maintenance and foreign key reference (child table only requires a single column as the foreign key to the parent entity)
   2. This almost always saved on space as well in the child entity (as the composite key would typically take up more space)
   3. MovieCastMember is an example of an entity that does not have a single primary key value. This is because the entity is not referenced by other entities, and it simplifies the maintenance of the entity to not create the single primary key on it.
2. In conjunction with #1, make this single primary key artificial if at least one of the following scenarios apply:
   1. There is no natural key for the entity
      1. E.g. The Movie and FilmMember entities are examples of this.
   2. The natural key for the entity is greater than 16 bytes
      1. E.g. The “GenreName” column for the Genre entity is a natural key, but is varchar(25) – which takes up more than 16 bytes. An artificial smallint could be used instead to significantly reduce the storage necessary for foreign key references.
   3. The artificial key would take up less space than the natural key
      1. E.g. the “Code” column in RestrictionRating is varchar(10), which is equal to 10 bytes + 2 bytes to hold the length (12 bytes total). However, a smallint (2 bytes) artificial key could be used to save 10 bytes per reference.
   4. The key value has a chance of changing (which would require cascading the update to child entities) and/or reordering of the entity’s clustered index
      1. Typically, Name/Text based columns do have a chance of changing, albeit small chance (e.g. GenreName, UserRoleName, etc.), and thus would require cascading updates to references.
      2. MovieImage is an example of an entity that contains a Natural key of {MovieId, ImageName}. However, the ImageName has a fair chance of a changing, which would make for a more expensive write to maintain the entity’s clustered index order.
   5. Note: The only entity that is referenced by another entity (meeting criteria #1) and does not have an artificial primary key is the [Language] entity. This is because the LanguageIsoCode (ISO Code 639-1) is well-defined/standardized (unlikely to change), and only requires two bytes of storage (Adhering to conditions 2a, 2b, 2c, and 2d listed above).
3. Do use ‘smallint’ data type as a primary key for entities that will not have large amounts of data.
   1. The primary reason for this is that smallint is a 2-byte integer, which can hold a value up to 32,767 (more than enough for several entities in the database, such as AwardShow, FilmRole, Genre, ImageType, RestrictionRating, UserRole). Although storage is cheap, a movie database will contain a significant amount of data. Foreign key references back to entities with a smallint will store half the number of bytes than a normal int.
4. Do use an ‘int’ (max 231 value) data type as a primary key for entities with a medium amount of data (e.g. User)
5. Do use a ‘bigint’ (max 263 value) data type as a primary key for entities with potentially large amounts of data (e.g. Movie, MovieImage, FilmMember, etc.)
6. Do use non-identity (in other words, assigned artificial IDs) for certain smallint/static entities that regular users cannot modify (e.g. Genres, RestrictionRatings, MovieImageType, UserRoles, AwardShows, etc.)
   1. These entities should be managed by the Development Team members, and they will not be updated often. In addition, they may require code references to these certain ID values (as an Enum) in order to conditionally add logic or functionality. An auto-incremented ID has no guarantee that the value will remain the same on all environments, since the value is assigned by the DBMS engine.
   2. For example, see how the UserRole entity can match an Enum class in code. The Enum values can match the Id values in the actual entity itself. An assigned artificial ID (non-auto-increment) allows for this consistent referencing:
7. Do use identity/auto-increment for all other artificial keys (on dynamic entities or even for certain relatively static entities that are not necessary to be treated with Enum values).
   1. This allows for less maintenance on the developer, and the DBMS has a reliable way of assigning IDs.
   2. E.g. Movie, FilmMember, AwardShowInstance, etc. contain identity/auto-increment artificial keys.
8. Do add non-clustered unique-index constraints on natural key values
   1. This enforces a constraint on the entity, as well as makes lookups for the column faster.
9. Do add a clustered index constraint on the primary key of every entity (unless there is a good reason not to)
   1. Every entity in the database uses the primary key as the clustered key. For artificial and auto-increment keys, this is very easy on the DBMS engine to maintain the physical order of the records.

The following are MovieDbLite-specific rationale behind relationships and entities:

1. A single entity is used to store all members of the film industry: the FilmMember entity.
   1. Did not split up the FilmMember entity into separate, role-based entities (e.g. ActorEntity, DirectorEntity, etc.) because FilmMembers can (and many times do) participate in several different roles of the film industry. For example, Ben Affleck is an actor, but also a screen writer and producer for certain films.
2. The Movie and FilmMember entities contain a fair amount of NULL columns
   1. The Movie entity does contain several nullable columns, however, this is intentional to allow for announced/planned Movies to be tracked in the system before details are known.
   2. FilmMember contains a number of NULL columns as well, but many of these are optional fields for a FilmMember (such as a Prefix, Suffix, Middle Name, Biography, etc.) that either may not be known (i.e. lesser known figure may not have a Bio) or not relevant (i.e. not everyone has a name Suffix).
3. A required PreferredFullName column is added to the FilmMember entity
   1. This allows for the film member to specify the stage name they go by (and want to be reported as in the website), while also entering their real First/Middle/LastNames in the respective fields.
   2. E.g. Ben Affleck’s actual name is Benjamin Geza Affleck-Boldt, but he does not go by that.
4. Movie cast and crew are stored in different places. The movie cast members (i.e. actors) are stored in the MovieCastMember entity. The movie director is stored in the Movie entity. The movie crew members (i.e. all remaining non-directors, non-actors that participated in the movie) are stored in the MovieCrewMember entity. This one I went back and forth with, but ultimately there were a few reasons for this design:
   1. MovieCastMember allows us to store additional movie/actor specific information, such as the CharacterName. This CharacterName would not be relevant for any of the Crew members.
   2. MovieCastMembers will typically be searched, looked up, or reported more often than MovieCrewMembers (because users are typically more interested in the figure heads / actors of the movie). Therefore, splicing the data out into its own entity makes for faster lookups – as opposed to it all being crammed into one MovieCastAndCrewMember entity.
   3. A crew member in a movie can have multiple roles (e.g. producer and screenwriter), so therefore we specify the RoleId in the entity and as part of the Primary Key. Also, there are several Movie roles, so it did not make sense to partition n-role amount of entities out (from a scalability standpoint).
   4. Films almost always have one director, and they are credited in that manner for movies (see [link](https://www.quora.com/Approximately-what-percent-of-films-have-more-than-one-director)). Exceptions are situations like established duos such as the Coen brothers, and this is a future requirement that could be added – which I mention in the 6 Concluding Remarks section. In the meantime, situations like that can involve either specifying the directors in the MovieCrewMember entity, or specifying the primary director in Movie.DirectorFilmMemberId, and secondary one in the MovieCrewMember entity. I still wanted the Director in the Movie entity because this is arguably the most-important role in making the movie (outside of the actors), and will frequently be reported. Putting it directly on the Movie entity when required less joins/logic to report, and because it meets 99% of the cases, it made sense.

## E/R Model

This section will cover details about the Entity Relationship Model for the database.

In general, the database model can be broken apart and described as four main components/portions of functionality, each loosely interconnected with each other to make apart the entire database structure. The 3.2.3 E/R Diagram section will contain the visual diagram for the four main components, to help illustrate the entities involved and the relationships amongst them.

These four portions of functionality include the following:

1. Movie Details / Information
   1. This relates to the general information/attributes related to Movies and Films. For example, the release date of a movie, or the genres of a movie.
2. User Access and Reviews
   1. This relates to information that MovieDbLite that involves user interaction. For example, creating a user account or writing a user review.
3. Movie Cast and Crew
   1. This relates to information about film members in the movie industry, as well as the movies they have been a part of (cast and crew).
4. Award Show Winners
   1. This relates to information about annual award shows, the awards that are handed out, and whom they are handed out to.

In each of these four components, the Movie entity (soon to be described in more detail) is the glue between connecting them all. Additionally, in some circumstances the Film Member also makes connections amongst these pieces. The Movie and Film Member are the primary strong entities of the database.

### Entities

This database consists of twenty entities that are the building blocks to delivering the functional requirements for the project. This section will group the entities by their respective component/portion that was mentioned in the previous section, as these are the entities that are related to one-another.

The entities can also be categorized based on whether they are static (admin/developer managed) or dynamic entities. Static entities can be defined as entities that are pre-defined and seeded with data upfront, and remain mostly static (meaning that the data in it is not added/modified/deleted often). The primary purpose of these static entities is to avoid redundantly storing the names/descriptors for the values within other entities in the database (which may vary in length), where the value can simply be referenced by a static identifier. It also serves as a consistent/standard way of setting and viewing data in the database. For example, if these types did not exist, one may enter “Sci Fi” as a genre for a given movie, whereas another person may spell it out as “Science Fiction”. This would result in two different values being stored that mean the same thing.

The semantics about whether an entity is static or dynamic is not too important, however, I chose to organize each subsection below to begin with the static entities first. This is because these static entities are strong entities that are referenced by other entities, so introducing them upfront seemed logical. A screenshot is included of sample data for the static entities to help give a perspective of what the entity is used for, and what static data is stored within it. Although this is somewhat of an implementation detail, I felt like it was useful to introduce in this design section as well because it illustrates why the entity was designed in the corresponding manner.

The primary key rules for the following entities adhere to the standards/guidelines listed in the 3.1 Design Rationale section. Since that section outlines how primary keys are defined, I have omitted the full details in this section.

#### General Movie Information

1. Genre (Static Entity, Strong Entity)
   1. The purpose of this entity is to track the different genre types that movies can be classified by. A Genre contains a required (and unique/natural key) GenreName and optional Description.
   2. Contains artificial, assigned (non-identity) single primary key column.
   3. This is considered a static entity because the records of this entity will not be modified often, as it is uncommon that new movie genres will be created. The existing movie genres have remained relatively unchanged for a while now (e.g. Action, Comedy, Horror, Romance, Drama, etc.)



1. Image Type (Static Entity, Strong Entity)
   1. The purpose of this entity is to track the different types of images that are supported to be uploaded to the database (e.g. for movie images / thumbnails). This contains required Image Extension (unique key/natural key) and Name columns. The ImageExtension is used to help determine the type of image file being uploaded, and if any special logic is necessary for saving/downloading the file’s contents.
   2. Contains artificial, assigned (non-identity) single primary key column.
   3. This is considered a static entity because the records of this entity will not be modified often. The different types of images have been well-established and stable for a while now.



1. Language (Static Entity, Strong Entity)
   1. The purpose of this entity is to track the languages that the dialogue of a movie can be classified as. The primary key for the entity is the Language Iso Code which is an [ISO Code 639-1](https://en.wikipedia.org/wiki/List_of_ISO_639-1_codes) specification (two letter code for denoting languages). This entity also contains a required LanguageName column which is a user-friendly name of the column.
   2. Contains non-artificial, assigned (non-identity) single primary key column.
   3. This is considered a static entity because the records of this entity will not be modified often, as it is uncommon that new languages are created or established to be used in the film industry.



1. RestrictionRating (Static Entity, Strong Entity)
   1. The purpose of this entity is to track the different viewer [restriction ratings](https://en.wikipedia.org/wiki/Motion_Picture_Association_of_America_film_rating_system) that a film is classified as by the Motion Picture Association of America (MPAA). This entity contains all required fields (Code (unique/natural key), ShortDescription, LongDescription, and IsActive). IsActive can be used to indicate the rating is no longer used (e.g. rated “M”).
   2. This is considered a static entity because the records of this entity do not change often. The current MPAA ratings have not changed since 1996.
   3. Contains artificial, assigned (non-identity) single primary key column.
   4. You will notice how the columns match up to the properties of the described on the MPAA website [here](https://www.motionpictures.org/film-ratings/).



**Figure 1:** **Green boxed indicates the ShortDescription. Red boxed indicates the Code. Blue boxed indicates part of the LongDescription.**



1. MovieImage (Dynamic Entity, Strong Entity)
   1. The purpose of this entity is to track the different uploaded images (posters/thumbnails/action photos/etc.) associated with a movie. An artificial primary key is added to the table per guidelines/standards discussed in an earlier section. The unique/natural key of the entity is the combination of {MovieId, ImageName} such that each uploaded image for a movie must be provided a unique name to distinguish it from others. In addition, the Image Type (required) (e.g. jpg, png, etc.) is stored with the uploaded movie image, as well as the raw binary File Contents (required) (using varbinary(max) in SQL Server). A DateUploaded (required) is stored as well for when the movie’s image was uploaded/associated. An optional Description is available if additional details are used to describe the image.
   2. Contains artificial, sequential (identity) single primary key column.
2. Movie (Dynamic Entity, Strong Entity)
   1. The purpose of this entity is to track all the movies/films that are tracked in MovieDbLite. This entity consists of several of the one-to-one attributes of a Movie, including a required Title and Description. The remaining fields are optional/nullable because they may not be known when the Movie is first entered into the database (e.g. an announced movie that will be released in the future). However, it is intended that these attributes are eventually populated when the information is known. These optional fields include the ReleaseDate, RestrictionRatingId, DirectorFilmMemberId, and DurationInMinutes. There is also one derived/calculated optional field, which is the AverageUserRating column. This contains an aggregated average based off the MovieUserReview entity. More details will be discussed in the database triggers section pertaining to this AverageUserRating field.
   2. An index (non-unique) is added to the Title field as it will be common for searches to be performed against this field.
   3. Contains artificial, sequential (identity) single primary key column.
3. Movie\_Genre (Dynamic Entity, Weak Entity)
   1. The purpose of this entity is to track the genres associated with a movie. This is an intersection entity used to define a many-to-many relationship between Movies and Genres. A movie may have multiple genres, and a genre may (and is most likely) associated with multiple movies.
   2. Contains non-artificial, composite primary key column
4. Movie\_Language (Dynamic Entity, Weak Entity)
   1. The purpose of this entity is to track the different supported languages associated with a movie. For example, if a movie contains a significant amount of English and Spanish dialogue, then it may be classified as both. Similarly, if the movie is recorded and released in different languages, that relationship will be tracked in this entity. This is an intersection entity used to define a many-to-many relationship between Movies and Languages. A movie may have multiple languages, and a language may (and is most likely) associated with multiple movies.
   2. Contains non-artificial, composite primary key column

#### User Access and Reviews

1. UserRole (Static Entity, Strong Entity)
   1. The purpose of this entity is to track the different roles that a user can be assigned to. The user’s role will drive the permissions they have for accessing or modifying data in MovieDbLite. This entity contains required (and unique/natural key) RoleName and Description columns. The Description is intended to briefly describe the purpose of the role and the access restrictions.
   2. This is considered a static entity because the records of this entity do not change often (changes would normally involve new functionality or significant changes to areas of the website), and are managed by the development team.
   3. Contains artificial, assigned (non-identity) single primary key column.



1. User (Dynamic Entity, Strong Entity)
   1. The purpose of this entity is to track all of the users in MovieDbLite. Each user creates a required (unique key) UserName to use for logins. A required email is also stored to use for email communication with the user. The user also enters a password upon account creation, and the hashed version of it is stored in the database. More details about that are discussed in the security section of this document. Lastly, each user is assigned to one and only one User Role. This user role dictates their permissions to the application.
   2. Contains artificial, sequential (identity) single primary key column.
2. MovieUserReview (Dynamic Entity, Strong Entity)
   1. The purpose of this entity is to track different reviews/ratings that users give to a certain movie. This entity contains a unique/natural key of {MovieId, UserId} indicating that a certain user can write only one review for a given movie. The required Rating column is a value 1-10 (1 being the worst, 10 being the best) that the user assigns to the film. The Review column is an optional large text field if the user chooses to describe their review. Lastly, a required DatePosted date is stored with the review, which is useful for sorting circumstances.
   2. The Movie.AverageUserRating column is meant to be a derived column that is calculated based on the MovieUserReviews for the particular Movie. Implementation details on how this derived column is maintained via a trigger is discussed in the advanced features section.
   3. Contains artificial, sequential (identity) single primary key column.
3. MovieUserReviewHelpful (Dynamic Entity, Weak Entity)
   1. The purpose of this entity is to track the helpfulness of user reviews (based off what other users think are helpful reviews). This can be thought of as an upvote/downvote (thumbs up/thumbs down) type system for reviews. The MovieUserReview and UserId make apart the primary key. This means that a certain user could only give one opinion of whether a certain review is helpful or not. In other words, they cannot upvote or downvote a particular review multiple times (though they can change their upvote to a downvote, and vice-versa, if they wish). The required IsHelpful column is used to store whether the user upvoted or downvoted the review.
   2. Contains non-artificial, composite primary key column

#### Movie Cast and Crew

1. FilmRole (Static Entity, Strong Entity)
   1. The purpose of this entity is to track the different role types of the personnel/film members that contribute to making films/movies. A FilmRole contains a required (and unique/natural key) RoleName column and optional Description.
   2. This is considered a static entity because the records of this entity will not be modified often, as the film roles in the industry have been well established over the years (e.g. Director, Writer, Producer, Actor, Costume Designer, etc.)
   3. Contains artificial, assigned (non-identity) single primary key column.



1. FilmMember (Dynamic Entity, Strong Entity)
   1. The purpose of this entity is to track all the persons participating in the making of movies/films. This entity tracks several fields for each FilmMember, including a Prefix (e.g. Dr., Mr., Mrs., Ms., etc.) (optional), FirstName (required), MiddleName (optional), LastName (required), Suffix (e.g. Jr., Sr., IV, etc.) (optional), PreferredFullName (required), Gender (M or F) (required), DateOfBirth (optional), DateOfDeath (optional), and Biography. The PreferredFullName field is an important field because this is what is intended to be used in most areas of the application to display the name of the FilmMember. This field is synonymous with “StageName”, and may differ from the person’s concatenated prefix/first/middle/last/suffix name. E.g. Katheryn Elizabeth Hudson goes by “Katy Perry”.
   2. An index (non-unique) is added to the PreferredFullName field as it will be common for searches to be performed against this field.
   3. Contains artificial, sequential (identity) single primary key column.
2. MovieCastMember (Dynamic Entity, Weak Entity)
   1. The purpose of this entity is to track the different cast members (i.e. actors) associated with a movie. The Primary key consists of the Movie and FilmMember. A Movie Cast Member record also contains a required CharacterName attribute and optional Sequence. The CharacterName stores the name of the cast member in the film. The Sequence is used to order the actors in a listed manner if desired (e.g. by importance, significance of role, order of appearance, etc.) This entity is not intended to store film members of the movie that do not act in a scene (e.g. crew members).
   2. Contains non-artificial, composite primary key column
3. MovieCrewMember (Dynamic Entity, Weak Entity)
   1. The purpose of this entity is to track the different crew members (e.g. writers, producers, costume designers, etc.) that worked on the film. The Primary key consists of the Movie, FilmMember, and FilmRole that the corresponding film member participated in. It is possible that the same FilmMember shows up in this entity more than once for a movie, if they participated in more than one role when making the movie (e.g. if a film member was both a producer and screenwriter). In addition, it is possible that more than one film member of a certain film role worked on the movie (e.g. if multiple producers produced a certain movie). It is also possible that a member of the MovieCastMember entity is also a member of the MovieCrewMember entity (e.g. if one of the actors is also one of the writers).
   2. This entity is intended to be used to store all roles aside from Actors and Directors, since those are stored within MovieCastMember and Movie.DirectorFilmMemberId respectively.
   3. Contains non-artificial, composite primary key column

#### Award Shows and Winners

1. AwardShow (Static Entity, Strong Entity)
   1. The purpose of this entity is to track the different types of annual movie/film award shows that present awards to movies/film members. The AwardShow contains a required (and unique/natural key) ShowName column and optional Description.
   2. This is considered a static entity because (for the most part) the records in this entity are not modified often, and it is not often the case that Award Shows change. The standard ones have been around for a while (e.g. Oscars, Golden Globes, Critics’ Choice Awards, etc.)
   3. Contains artificial, assigned (non-identity) single primary key column.



1. Award (Static Entity, Strong Entity)
   1. The purpose of this entity is to track the different types of movie/film member awards that can be presented at annual award show events. Each award is tied to an Award Show. The Award contains a required AwardName column and optional Description.
   2. A unique/natural key for the entity is the combination of {AwardShowId, AwardName}. In other words, the AwardName should be unique for the given AwardShow. Consequently, this does also imply the AwardName by itself is not unique. For example, the Oscars and Critics’ Choice Awards both have an award named “Best Actress” (amongst other awards that also share the same names).
   3. This is considered a static entity because (for the most part) the records in this entity are not modified often, and have historically been fairly consistent on an annual basis when the award shows are hosted.
   4. Contains artificial, sequential (identity) single primary key column.



1. AwardShowInstance (Static Entity, Strong Entity)
   1. The purpose of this entity is to keep track of each instance of an AwardShow. In other words, the actual award show event that takes place at a particular year. Each AwardShowInstance contains an Award Show it is an instance of. It also contains a required Year field (the year the awards are being presented for) and DateHosted field (the Date the event instance took place).
   2. The unique/natural key for the entity is the combination of {AwardShowId, Year}. In other words, there should only be one instance of an AwardShow for a given Year. It has been the standard for a long time that these Award Shows are done on an annual basis, which is the reasoning behind the constraint.
   3. This entity could borderline be considered static; however, multiple entries will be added to this on at least an annual basis, and for that reason, it can be classified as dynamic.
   4. Contains artificial, sequential (identity) single primary key column.
2. AwardWinner (Dynamic Entity, Weak Entity)
   1. The purpose of this entity is to keep track of the winners/recipients of Awards of a particular Award Show Instance. Each AwardWinner record tracks the AwardShowInstance it belongs to, the actual Award that was won, the Film Member it was presented to, and the Movie it was for. All of these columns are required.
   2. The combination of {AwardShowInstance, AwardId, FilmMemberId} make up the PK for the entity. Multiple film members may be awarded a particular award (at an award show instance), and so this column must make up the unique primary key. For example, sometimes multiple people are awarded Best Costume Design or Best Cinematography for a particular movie.
   3. Both a FilmMember and a Movie are required to be entered for a record. This is because each Award that is presented is toward a particular Movie (there are currently not awards that exist otherwise, or at least are not tracked by this database). In addition, every Award is accepted/presented to a certain Film Member. This includes “Best Picture” award that may seemingly appear to be a “movie-only” award. However, the Producer(s) accept this award on behalf of the Movie.
   4. Contains non-artificial, composite primary key column

### Relationships

This section describes the relationships between entities in MovieDbLite.

#### Award to Award Show

* **Description:** An award is associated with one and only one award show. An award show can have zero to many awards.
* **Layman’s Terms:** This relationship tracks how an award show can award out many different awards.
* **Cardinality:** This is a one-to-many relationship.
* **Parent Entity:** Award Show
* **Child Entity**: Award
* **Participation:** The Award (being the child entity) has **total participation** in the relationship, and the Award Show (being the parent entity) has **partial participation** in the relationship.
* **Identification:** This relationship is a **non-identifying** relationship.

#### AwardShowInstance to AwardShow

* **Description:** An award show instance is associated with one and only one award show. An award show can have zero to many award show instances.
* **Layman’s Terms:** This relationship tracks how award shows can have many different events (instances) over time.
* **Cardinality:** This is a one-to-many relationship.
* **Parent Entity:** Award Show
* **Child Entity**: Award Show Instance
* **Participation:** The Award Show Instance (being the child entity) has **total participation** in the relationship, and the Award Show (being the parent entity) has **partial participation** in the relationship.
* **Identification:** This relationship is a **non-identifying** relationship.

#### AwardShowInstance to Award to FilmMember (AwardWinner)

* **Description:** An award show instance can give out zero to many awards to film members.
* **Layman’s Terms:** An award show instance/event can give out many awards. Each award could be given to one to many film members (i.e. some awards are given to multiple persons on the same film).
* **Cardinality:** This is a unique case of a tertiary (three column) primary key. The AwardShowInstance, Award, and FilmMember respectively have a one-to-many relationship with the AwardWinner entity. However, the AwardWinner entity represents a many-to-many-to-many relationship between Awards, AwardShowInstances, and FilmMembers respectively.
* **Parent Entities:** Award Show Instance (Strong), Award (Strong), and Film Member (Strong)
* **Child Entity**: AwardWinner (Weak)
* The AwardWinner is an intersection entity used to store the many-to-many-to-many relationship between the three entities.
* The relationships from AwardShowInstance, Award, and FilmMember to child AwardWinner are **identifying** relationships because the child entity could not exist without the three parent entities.
* The relationships from AwardShowInstance, Award, and FilmMember to child AwardWinner are **partial participation** because not necessarily all AwardShowInstances, Awards, and FilmMembers may be a part of the relationship/intersection entity.
* The relationships from AwardWinner to AwardShowInstance, Award, and FilmMember are **total participation**

#### AwardWinner to Movie

* **Description:** An award winner record is associated with one and only one movie. A movie can have zero or many award winners.
* **Layman’s Terms:** A movie can win multiple awards (but may win none).
* **Cardinality:** This is a one-to-many relationship.
* **Parent Entity:** Movie
* **Child Entity**: Award Winner
* **Participation:** The Award Winner (being the child entity) has **total participation** in the relationship, and the Movie (being the parent entity) has **partial participation** in the relationship.
* **Identification:** This relationship is a **non-identifying** relationship. The MovieId is not a part of the AwardWinner’s Primary Key.

#### Movie to FilmMember (Director)

* **Description:** A movie can have zero or one film member that is the director. A director can direct zero or many movies.
* **Layman’s Terms:** A director can direct zero or many movies. A movie can have at most one director. A movie may have no director before production has started.
* **Cardinality:** This is a one-to-many relationship.
* **Parent Entity:** Film Member
* **Child Entity**: Movie
* **Participation:** The Movie has **partial participation** in the relationship, and the Film Member has **partial participation** in the relationship.
* **Identification:** This relationship is a **non-identifying** relationship.

#### Movie to RestrictionRating

* **Description:** A movie can have zero or MPAA viewer restriction rating. A restriction rating can be associated with zero or many movies.
* **Layman’s Terms:** A movie can be assigned at most one restriction rating (e.g. Titanic is PG-13). It may not be assigned (e.g. if not released to theatres or not yet released). There can be many movies of a certain restriction rating (e.g. there are many rated G movies)
* **Cardinality:** This is a one-to-many relationship.
* **Parent Entity:** Restriction Rating
* **Child Entity**: Movie
* **Participation:** The Movie has **partial participation** in the relationship, and the Restriction Rating has **partial participation** in the relationship.
* **Identification:** This relationship is a **non-identifying** relationship.

#### Movie to Genre (Movie\_Genre Entity)

* **Description:** A movie can be classified as zero or many genres. A genre can be associated with zero or many movies.
* **Layman’s Terms:** This tracks the genres that are associated with movies.
* **Cardinality:** This is a many-to-many relationship between movies and genres
* **Parent Entities:** Movie (Strong) and Genre (Strong)
* **Child Entity**: Movie\_Genre (Weak)
* The Movie\_Genre is an intersection entity used to store the many-to-many relationship between the two entities.
* The relationships from Movie and Genre to child Movie\_Genre are **identifying** relationships because the child entity could not exist without the two parent entities
* The relationships from Movie and Genre to child Movie\_Genre are **partial participation** because not necessarily all Movies or Genres may be a part of the relationship/intersection entity.
* The relationships from Movie\_Genre to Movie and Genre are **total participation** (as normal for the required attribute in the child entity of a relationship)

#### Movie to Language

* **Description:** A movie can have dialogue in zero or many languages. A language can be associated with zero or many movies.
* **Layman’s Terms:** This tracks the dialogue languages that are associated with movies.
* **Cardinality:** This is a many-to-many relationship between movies and languages.
* **Parent Entities:** Movie (Strong) and Language (Strong)
* **Child Entity**: Movie\_Language (Weak)
* The Movie\_Language is an intersection entity used to store the many-to-many relationship between the two entities.
* The relationships from Movie and Language to child Movie\_Language are **identifying** relationships because the child entity could not exist without the two parent entities
* The relationships from Movie and Language to child Movie\_Language are **partial participation** because not necessarily all Movies or Languages may be a part of the relationship/intersection entity.
* The relationships from Movie\_Language to Movie and Language are **total participation** (as normal for the required attribute in the child entity of a relationship)

#### Movie to FilmMember (MovieCastMember)

* **Description:** A movie can have zero to many actors in it. An actor can act/cast in zero to many movies.
* **Layman’s Terms:** This relationship tracks the actors that act in movies.
* **Cardinality:** This is a many-to-many relationship between movies and film members (in the form of actors).
* **Parent Entities:** Movie (Strong) and Film Member (Strong)
* **Child Entity**: MovieCastMember (Weak)
* The MovieCastMember is an intersection entity used to store the many-to-many relationship between the two entities.
* The relationships from Movie and FilmMember to child MovieCastMember are **identifying** relationships because the child entity could not exist without the two parent entities
* The relationships from Movie and FilmMember to child MovieCastMember are **partial participation** because not necessarily all Movies or FilmMembers may be a part of the relationship/intersection entity. E.g. some FilmMembers may not be actors.
* The relationships from MovieCastMember to Movie and FilmMember are **total participation**

#### Movie to FilmMember to FilmRole (MovieCrewMember)

* **Description:** A movie can have multiple film/crew members of different roles.
* **Layman’s Terms:** A movie can have multiple crew members, and the crew member could work on one to many film roles for the movie. (e.g. a producer may also be a screenwriter for the film).
* **Cardinality:** This is a unique case of a tertiary (three column) primary key. The Movie, FilmMember, and FilmRole respectively have a one-to-many relationship with the MovieCrewMember entity. However, the MovieCrewMember entity represents a many-to-many-to-many relationship between Movies, FilmMembers, and FilmRoles respectively.
* **Parent Entities:** Movie (Strong), Film Member (Strong), and Film Role (Strong)
* **Child Entity**: Movie Crew Member (Weak)
* The MovieCrewMember is an intersection entity used to store the many-to-many-to-many relationship between the three entities.
* The relationships from Movie, FilmMember, and FilmRole to child MovieCrewMember are **identifying** relationships because the child entity could not exist without the three parent entities.
* The relationships from Movie, FilmMember, and FilmRole to child MovieCrewMember are **partial participation** because not necessarily all Movies, FilmMembers, and FilmRoles may be a part of the relationship/intersection entity.
* The relationships from MovieCrewMember to Movie, FilmMember, and FilmRole are **total participation**

#### MovieImage to ImageType

* **Description:** A movie image is associated with one and only one image type. An image type can be associated with zero or many movie images.
* **Layman’s Terms:** This relationship tracks the image type (e.g. .jpg, .png) associated with a movie image.
* **Cardinality:** This is a one-to-many relationship.
* **Parent Entity:** Image Type
* **Child Entity**: Movie Image
* **Participation:** The Movie Image (being the child entity) has **total participation** in the relationship, and the Image Type (being the parent entity) has **partial participation** in the relationship.
* **Identification:** This relationship is a **non-identifying** relationship.

#### MovieImage to Movie

* **Description:** A movie image is associated with one and only one movie. A movie can have zero or many movie images.
* **Layman’s Terms:** This relationship tracks the image associated with a movie.
* **Cardinality:** This is a one-to-many relationship.
* **Parent Entity:** Movie
* **Child Entity**: Movie Image
* **Participation:** The Movie Image (being the child entity) has **total participation** in the relationship, and the Movie (being the parent entity) has **partial participation** in the relationship.
* **Identification:** This relationship is a **non-identifying** relationship.

#### Movie to User (MovieUserReview)

* **Description:** A user can write zero to many reviews for movies. A movie can reviews written by zero to many users.
* **Layman’s Terms:** This relationship tracks the user reviews written for movies.
* **Cardinality:** This is a many-to-many relationship between movies and users to track user movie reviews.
* **Parent Entities:** Movie (Strong) and User (Strong)
* **Child Entity**: MovieUserReview (Weak)
* The MovieUserReview is an intersection entity used to store the many-to-many relationship between the two entities.
* The relationships from Movie and User to child MovieUserReview are **non-identifying** relationships because the MovieUserReview entity has its own artificial key.
* The relationships from Movie and User to child MovieUserReview are **partial participation** because not necessarily all Movies or Users may be a part of the relationship/intersection entity. E.g. Some Users may not write reviews; or some movies may not have reviews written for it.
* The relationships from MovieUserReview to Movie and User are **total participation**

#### MovieUserReview to User (MovieUserReviewHelpful)

* **Description:** A movie user review can be marked helpful by zero to many users. A user can mark zero to many movie user reviews helpful.
* **Layman’s Terms:** This relationship tracks the movie user reviews that are marked helpful by users.
* **Cardinality:** This is a many-to-many relationship between movie user reviews and users to track the reviews that are helpful/unhelpful.
* **Parent Entities:** MovieUserReview (Strong) and User (Strong)
* **Child Entity**: MovieUserReviewHelpful (Weak)
* The MovieUserReviewHelpful is an intersection entity used to store the many-to-many relationship between the two entities.
* The relationships from MovieUserReview and User to child MovieUserReviewHelpful are **identifying** relationships because the child entity could not exist without the two parent entities
* The relationships from MovieUserReview and User to child MovieUserReviewHelpful are **partial participation** because not necessarily all MovieUserReviews or Users may be a part of the relationship/intersection entity. E.g. Some reviews may not be marked helpful; and some users may not mark any reviews as helpful.
* The relationships from MovieUserReviewHelpful to MovieUserReview and User are **total participation**

#### User to UserRole

* **Description:** A user is associated with one and only one user role. A user role can be assigned to zero or many users.
* **Layman’s Terms:** This relationship tracks the role associated with each user.
* **Cardinality:** This is a one-to-many relationship.
* **Parent Entity:** User Role
* **Child Entity**: User
* **Participation:** The User (being the child entity) has **total participation** in the relationship, and the User Role (being the parent entity) has **partial participation** in the relationship.
* **Identification:** This relationship is a **non-identifying** relationship.

### E/R Diagram

This section describes the E/R diagram for the database. The diagrams displayed were created using Visual Paradigm Community Edition.

The following ERD is a diagram for the entire database and contains a Legend that explains the meaning for the visual icons/indicators. We will focus on four main components/portions of the overall diagram in this section, as these components encompass pieces of the primary functional requirements that can be individually explained.



#### General Movie Information

The diagram in this section illustrates how general movie information and details are stored in the database. The parent-most entity that contains Movie information in this relationship is the Movie entity. The Movie entity contains a number of attributes that are “one-to-one” relational with a Movie (e.g. Title, Description, Release Date, etc.) Many of the columns of the Movie entity are nullable because data for the Movie may be entered prior to the movie being released (e.g. in early stages when the Movie is announced). However, it is expected that these attributes are eventually filled in as the details are known.

Child entities are created off of Movie (such as MovieImage, Movie\_Language, and Movie\_Genre) to store additional details related to a Movie. Movie\_Genre and Movie\_Language are many-to-many intersection entities; MovieImage is a one-to-many entity (a Movie may have many Images). Language, Genre, and ImageType are other Parent entities used by the respective Movie’s child entity entities.



#### User Access and Reviews

The diagram in this section illustrates how Users of MovieDbLite are stored, as well as how the ratings/reviews that user’s provide to Movies are stored. A user account can write a review for a Movie (stored in the MovieUserReview entity), and other users can mark the Review as Helpful or Unhelpful (via MovieUserReviewHelpful). Each User is tied to a UserRole to indicate whether they are an Admin or regular User, which affects the permissions they have over certain areas of the application (although, both Users and Admins can write reviews for movies).

Note: Attributes from the Movie entity in the diagram below were removed for brevity. The important part was to show the relationship between MovieUserReviews and Movie.



#### Movie Cast and Crew

The diagram in this section illustrates how the cast and crew of a movie are stored. The FilmMember parent entity stores all the persons/members of the film industry that participate as cast or crew members in a movie. The Movie entity itself stores the DirectorFilmMemberId (as there can only be one director per movie). The MovieCastMember is used to store the cast members (i.e. actors) of a movie and associated information about them (e.g. their CharacterName).

The MovieCrewMember stores the crew members of a movie. The parent entity FilmRole is used to describe what the crew member’s role is for the given movie (e.g. screenwriter, costumer designer, etc.)



#### Award Show and Winners

The diagram in this section illustrates how the awards that are awarded for movies are stored. There are different types of AwardShows that have different types of Awards (every Award is associated with an AwardShow). The winners of awards are stored in the AwardWinner entity. Each AwardWinner is associated with an AwardShowInstance, which is a parent entity that keeps track of annual award show events. An AwardWinner also is associated with a Movie that is associated with the award.

Note: Attributes from the Movie and FilmMember entity in the diagram below were removed for brevity. The important part was to show the relationship between AwardWinners and Movies / FilmMembers.



## Relational Model

This section covers more specifics about the relational model for the database, such as details about each column in the database, as well as some of the operational and data integrity rules that are put into place to allow correct transactions in the system.

### Data Dictionary

This is the Data Dictionary of the columns during database design. I alternate shades of color in the table to illustrate different entities (groupings with the same color are columns that belong to the same entity). The description of the table will describe the purpose of the column, as well as what entity it is associated with. Constraints marked as a Unique Key are shown to indicate the “Natural Key” for the entity. Although an entity may have a Natural Key, this may not necessarily be the Primary Key for the entity, as described in the 3.1 Design Rationale section. Valid Values is filled out for entities that have specific restrictions for its values. If left blank, there is not any special validation/integrity required for the column (outside of its data type and size).

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Column Name** | **Description** | **Data Type** | **Size** | **Constraint Type** | **Not Null?** | **Valid Values** |
| Id | Award Id - Primary surrogate key | int |  | Primary Key | Y |  |
| AwardShowId | The award show that grants this award | smallint |  | Foreign Key Part of Unique Key | Y |  |
| AwardName | Name of the award (e.g. Best Actor, Best Picture, etc.) | varchar | 50 | Part of Unique Key | Y |  |
| Description | Description of award | varchar | 200 |  | N |  |
| Id | Award show Id - Primary surrogate key | smallint |  | Primary Key | Y |  |
| ShowName | Name of award show (e.g. Oscars, Critics’ Choice Awards) | varchar | 50 | Unique Key | Y |  |
| Description | Description of award show and its purpose | varchar | 200 |  | N |  |
| Id | Award show instance Id - Primary surrogate key | int |  | Primary Key | Y |  |
| AwardShowId | The award show that this is an instance of. | smallint |  | Foreign Key Part of Unique Key | Y |  |
| Year | The film year the awards are presented for (this is typically the year prior to the host date year) | smallint |  | Part of Unique Key | Y | 4-digit positive value |
| DateHosted | Date the award show instance takes or took place | date |  | Constraint Type | Y |  |
| AwardShowInstanceId | The award show instance the winner won an award from | int |  | Part of Primary Key Foreign Key | Y |  |
| AwardId | The award the winner won | int |  | Part of Primary Key Foreign Key | Y |  |
| FilmMemberId | The winner's film member Id | bigint |  | Part of Primary Key Foreign Key | Y |  |
| MovieId | The movie the award belongs to | bigint |  | Foreign Key | Y |  |
| Id | Film Member Id - Primary Surrogate Key | bigint |  | Primary Key | Y |  |
| Prefix | Optional prefix for name (e.g. Mr., Mrs., Dr., Prince, etc.) | varchar | 10 |  | N | Not enforced since there are several valid values |
| FirstName | First name of film member | varchar | 50 |  | Y |  |
| MiddleName | Middle name of film member | varchar | 50 |  | N |  |
| LastName | Last name of film member | varchar | 50 |  | Y |  |
| Suffix | Optional suffix for film member (e.g. Jr, Sr., Jnr., Snr., III, IV). | varchar | 5 |  | N | Not enforced since there are several valid values |
| PreferredFullName | The required preferred name for the film member, which is how they would like to be reported to others (e.g. Katy Perry instead of Katheryn Hudson) | varchar | 150 | Non-Clustered Index | Y |  |
| Gender | The film member's gender | char | 1 |  | Y | 'M' or 'F' |
| DateOfBirth | The film member's date of birth | date |  |  | N |  |
| DateOfDeath | The film member's date of death | date |  |  | N |  |
| Biography | A biography of variable length for the film member | varchar | max |  | N | No maximum length is specified. |
| Id | Film role id - primary surrogate key | smallint |  | Primary Key | Y |  |
| RoleName | The user-friendly role name for a film role (e.g. Actor, Producer, Costume Designer) | varchar | 50 | Unique Key | Y |  |
| Description | Description of the film role's purpose | varchar | 200 |  | N |  |
| Id | Genre Id - primary surrogate key | smallint |  | Primary Key | Y |  |
| GenreName | User-friendly genre name for a film (e.g. Action, Horror, Drama) | varchar | 25 | Unique Key | Y |  |
| Description | Description of the genre | varchar | 500 |  | N |  |
| Id | Image type id - primary surrogate key | int |  | Primary Key | Y |  |
| ImageExtension | File extension for the image (e.g. .jpg, .png) | varchar | 10 | Unique Key | Y | Extension starts with period |
| Name | Image type name (e.g. JPEG, PNG) | varchar | 25 |  | Y |  |
| LanguageIsoCode | ISO-639-1 code that is a standardized nomenclature used to classify languages (e.g. "en", "fr", "es") | char | 2 | Primary Key | Y | Two-letter code |
| LanguageName | User-friendly name for language (e.g. English, French, Spanish) | varchar | 50 |  | Y |  |
| Id | Movie id - primary surrogate key | bigint |  | Primary Key | Y |  |
| Title | The movie title name (e.g. Pulp Fiction) | varchar | 150 | Non-Clustered Index | Y |  |
| Description | A description of the movie that potentially costs of high-level plot details | varchar | 500 |  | Y |  |
| ReleaseDate | The scheduled or a actual release date for the movie | date |  |  | N |  |
| RestrictionRatingId | The restriction rating of the given movie | smallint |  | Foreign Key | N |  |
| DirectorFilmMemberId | The director for the movie | bigint |  | Foreign Key | N |  |
| DurationInMinutes | The duration of the movie, in minutes (e.g. 154) | int |  |  | N | A positive integer value that describes the movie length in minutes. |
| AverageUserRating | An average of all user ratings for the movie (e.g. 8.9) | decimal |  |  | N | A decimal value between 1.0-10.0 |
| MovieId | The movie id that is associated with the given genre | bigint |  | Part of Primary Key Foreign Key | Y |  |
| GenreId | The genre that is associated with the given movie | smallint |  | Part of Primary Key Foreign Key | Y |  |
| MovieId | The movie id that is associated with the given language. | bigint |  | Part of Primary Key Foreign Key | Y |  |
| LanguageIsoCode | The language id that is associated with the given movie. | char | 2 | Part of Primary Key Foreign Key | Y | Two-letter code |
| MovieId | The movie id that the cast member is a part of | bigint |  | Part of Primary Key Foreign Key | Y |  |
| ActorFilmMemberId | The actor/cast film member id | bigint |  | Part of Primary Key Foreign Key | Y |  |
| CharacterName | The actor's character name in the given movie (e.g. Tony Montana) | varchar | 150 |  | N |  |
| Sequence | A sequence number associated the movie cast member to order/rank the cast members of a given movie | int |  |  | N | A positive integer value. |
| MovieId | The movie id that the crew member is a part id. | bigint |  | Part of Primary Key Foreign Key | Y |  |
| FilmMemberId | The crew member's film member id. | bigint |  | Part of Primary Key Foreign Key | Y |  |
| FilmRoleId | The role that the film member participated in when making the movie. | smallint |  | Part of Primary Key Foreign Key | Y |  |
| Id | Move image id - primary surrogate key | bigint |  | Primary Key | Y |  |
| MovieId | The movie id that the image is associated with | bigint |  | Foreign Key Part of Unique Key | Y |  |
| ImageName | The user-friendly name given to the image | varchar | 100 | Part of Unique Key | Y |  |
| ImageTypeId | The image type (e.g. jpg, png) | int |  | Foreign Key | Y |  |
| Description | Description of the image | varchar | 500 |  | N |  |
| FileContents | The actual raw binary contents of the image | varbinary | max |  | Y | Raw binary contents of the file |
| DateUploaded | The date the image was uploaded | datetime2 |  |  | Y |  |
| Id | Movie user review id - primary surrogate key | bigint |  | Primary Key | Y |  |
| MovieId | The movie the review is associated with | bigint |  | Foreign Key Part of Unique Key | Y |  |
| UserId | The user that wrote the review | int |  | Foreign Key Part of Unique Key | Y |  |
| Rating | The rating the user wrote for the review (e.g. 8) | smallint |  |  | Y | An integer value from 1-10 |
| Review | The optional review text that the user wrote | varchar | 8000 |  | N |  |
| DatePosted | The date the review was posted | datetime2 |  |  | Y |  |
| MovieUserReviewId | The movie user review that was deemed helpful. | bigint |  | Part of Primary Key Foreign Key | Y |  |
| UserId | The user that marked the review helpful | int |  | Part of Primary Key | Y |  |
| IsHelpful | Whether the review was actually helpful or unhelpful, respective of the bit field | bit |  |  | Y |  |
| Id | Restriction rating id - primary surrogate key | smallint |  | Primary Key | Y |  |
| Code | The MPAA restriction rating that can be assigned to a movie (e.g. PG-13) | varchar | 10 | Unique Key | Y | A valid MPAA film rating code (G, PG, PG-13, R, etc.) |
| ShortDescription | A short description of the rating (e.g. "General Audiences") | varchar | 50 |  | Y |  |
| LongDescription | A more detailed description of the rating | varchar | 200 |  | Y |  |
| IsActive | An active flag for whether or not the rating is still used. Certain ratings (e.g. "M") are no longer used. | bit |  |  | Y |  |
| Id | User Id - Primary surrogate key | int |  | Primary Key | Y |  |
| UserRoleId | The user's role for the MovieDbLite application | smallint |  | Foreign Key | Y |  |
| UserName | A username/handle chosen by the user (e.g. stevo9510) | varchar | 25 | Unique Key | Y | Letters and numbers only (no special characters) |
| EmailAddress | The user's email address | varchar | 255 |  | Y | Must be in a valid email format. |
| HashedPassword | The user's password secured with a bcrypt hash | varchar | 60 |  | Y | The password must contain a capital letter, lowercase letter, number, and be 10 or more characters. The password is a bcrypt hashed value (up to 60 chars) |
| Id | User role id - primary surrogate key | smallint |  | Primary Key | Y |  |
| RoleName | The role name for the role (e.g. Admin, User, etc.) | varchar | 25 | Unique Key | Y |  |
| Description | A description of the role and its general permissions | varchar | 500 |  | Y |  |

### Integrity Rules

Data integrity of the database involved utilizing some of the features of the RDBMS environment.

The basic and essential features of data integrity started with specifying each field as either required (via NOT NULL) or optional (NULLable). The mindset used during initial design was to start/default with each field as NOT NULL (required), and then figure out which fields may be optional (or may not have a value immediately known). This led to a database where ~80% of the columns (67/85) are specified as NOT NULL, and only ~20% of the columns are NULLable. Generally, the more enforcement of NOT NULL in the database, the better for data integrity – because it means that all the required/essential data must be entered for the record to commit. The majority of the NULLable columns fall into these entities, where it has minimal impact on the functionality of the system when they are null:

* Movie – A number of these columns are NULL because they may not be known when the movie is first announced. However, it is expected these are eventually filled in once the details are known
* FilmMember – A number of these columns are truly optional (e.g. not everyone has a Prefix or Suffix for their name, nor is the date of birth known for every film member that is credited in movies)

In addition to nullability constraints, basic data type constraints are enforced as well on columns based on their appropriate domain. For example, the FilmMember.DateOfBirth column is of the data type of ‘Date’ to ensure correctly formatted dates are entered into the field. In addition, ‘bit’ fields are used for Boolean (yes or no) values, such as MovieUserReviewHelpful.IsHelpful or RestrictionRating.IsActive. If values are numeric, such as Movie.DurationInMinutes, an approach numeric data type (such as ‘int’ in this scenario) is used. For Movie.AverageUserRating, the decimal(5, 2) type is used, because we only want to go to ‘2’ decimal places for the scale. The precision of ‘5’ is more than enough since our range of ratings is 1-10. The data type of char() is used when the amount of characters for the value is fixed (e.g. language iso code is a char(2)). Other data types (such as varchar and varbinary) are used as appropriate for text and binary related fields.

Referential integrity is established via the use of Foreign Key constraints in the RDBMS engine. All relationships described in the 3.2.2 Relationships section are referentially enforced by foreign key constraints in SQL Server. In addition to referential integrity constraints, the database enforces Unique Key constraints on natural key values for each entity. For example, an Award’s unique key is a combination of the {AwardShowId, AwardName}. A MovieUserReview’s unique key is a combination of the {MovieId, UserId} because the same user cannot write more than one review for a given movie. Further, many of the static entities have the corresponding entity’s name column as the unique key. For example, the Genre’s GenreName column is the unique natural key. Two genres with the same name would be indistinguishable if reported to the user.

In terms implementation of constraints, the 4.2.6 Constraints, User-Defined Data Types, and Rules section will cover advanced RDBMS specific features that were used to enforce data integrity, including custom constraints and user-defined data types.

### Operational Rules

There are several operational constraints imposed on the system. The following are examples of them:

1. Awards, Award Show Instances, and Award Shows can only be deleted if there is not an associated Award Winner with those entities.
2. A user’s username cannot be changed after it is created
3. A FilmMember can only be deleted if he/she is not associated with any Movies (i.e. not in the Cast/Crew/Director of any Movie)
4. An Award Winner’s Film Member must be a part of the Cast and Crew of the movie associated with the Movie Winner
5. The Award Winner’s Award must be an actual Award presented by the Award Show (in other words, if the Award Show Instance is the Oscar’s, it cannot hand out an Award from the Critics’ Choice Awards).
6. A Movie can only be deleted if only basic details have been entered (movie details, language, genre). In other words, if cast/crew/images/user ratings/etc. are entered, it cannot be removed.
7. Only valid image attachments defined within the ImageType entity can be uploaded
8. For the privacy of data control of our users, a user can delete their account, and it should remove all their corresponding movie review related information.
9. A movie crew member should not be assigned to the Actor/Director role (within that entity).
10. Several unique key constraints exist (to prevent duplicate entries), and are mentioned in more detail within the Entities section.
11. Film members with the same First and Last Name can exist in the database. The preferred full name is typically what distinguishes film members, however, that is not a guarantee.
12. Major strong entities such as Language, Genre, RestrictionRating, ImageType, FilmRole, UserRole are managed by development/administration team (for inserts/updates/deletes).
13. User roles will dictate many of application functionality. The only CUD functionality regular (non-admin) users have is in the form of user reviews/marking them as helpful.

### Operations

The purpose of this section is to describe the operations of a particular use.

We will describe the operations involved for the use case of adding/modifying a movie’s basic details (i.e. its general movie details, genre, and languages). In order to perform this type of operation within the website, multiple retrievals are required upfront to populate the form with option/list/drop-down data to allow the admin to easily select these elements. For example, the Movie’s Restriction Rating, Genre(s), Language(s), and Director.

Once the form is populated, the fields can be entered and the data can be submitted initially to perform an Insert operation against the Movie, Movie\_Language, and Movie\_Genre entities. The admin has the option to modify the Movie afterwards if they choose to (e.g. to make corrections, or to update some of the fields). Submitting a modification will perform an Update against the Movie entity, and potentially Inserts/Deletes on the Movie\_Language/Movie\_Genre entities (to tie any new mappings or remove any old mappings).

The movie can also be completely deleted (within the operational constraints mentioned in the previous section), in which case deletes are performed against the Movie, Movie\_Genre, and Movie\_Language entities.

## Security

This section covers the security provisions taken by MovieDbLite to ensure security of data and user operations within the system.

### Password Storage

Users have the ability to create user accounts to rate and review movies that they have watched. User accounts require a username, email, and password. The password is transmitted over the network using HTTPS (see upcoming section), and is never stored as a plaintext in the database. Instead, a hashed version of the password is stored in the database. Hash functions are useful because they cannot be algorithmically reversed back to plaintext.

When the user attempts to authenticate/login to the system, the typed/entered password is transmitted to the server (via HTTPS) and hashed using the same algorithm that was used to store the password in the database. The hashed entered/typed password is then compared against the hashed value in the database to see if they match, and if so, the user has successfully authenticated. An authorization token (with scopes based on the user’s role) is issued back to the client to be used in subsequent requests.

The hash algorithm used is [BCrypt](https://en.wikipedia.org/wiki/Bcrypt), implemented by the BCrypt.Net-Core package/library available in Microsoft’s NuGet Package Repository. BCrypt is a modern hashing algorithm that uses a salt/work factor to mitigate against [rainbow table](https://en.wikipedia.org/wiki/Rainbow_table) attacks (i.e. precomputed hash tables for reverse engineering hashes). It is a recommend password hashing algorithm by many sources, including this [one](https://auth0.com/blog/hashing-in-action-understanding-bcrypt/) on the Auth0 website.

The following is a snippet of the BCrypt library in use for MovieDbLite. This example shows a method that takes the unhashed (plaintext) password, and then returns the hashed version of it using BCrypt and a work factor/salt of 15. Note that we would normally not want to share the work factor/salt or make it known to others (for security reasons), however, for the intentions of this project I wanted to show how it worked with a real value.

### SQL Injection

MovieDbLite prevents SQL Injection attacks through the use of reliable .NET framework libraries and tools, such as ADO.NET and Entity Framework ORM (Object Relational Mapping) functionality.

ADO.NET contains [SqlCommand](https://docs.microsoft.com/en-us/dotnet/api/system.data.sqlclient.sqlcommand?view=netframework-4.8) and [SqlParameter](https://docs.microsoft.com/en-us/dotnet/api/system.data.sqlclient.sqlparameter?view=netframework-4.8) classes that prevent SQL Injection attacks by parameterizing any arguments to pass to a stored procedure. When these classes are used for a SQL Server database connection, they [abstract](https://stackoverflow.com/a/4892205) away functionality to call into the sp\_executesql system-level stored procedure in SQL Server, which isolates the parameters and treats them as data.

Here is an example of usage of these libraries in the MovieDbLite web application to insert an award winner (via the usp\_InsertAwardWinner stored procedure). Additionally, other operations in MovieDbLite utilize the Entity Framework ORM technology to perform CRUD. Entity framework is built with SQL injection protection by way of using its functions, allowing the developer to not have to put too much effort in mitigating the risk. The example below shows how an AwardShowInstance is added in MovieDbLite utilizing Entity Framework’s DbContext object (see MovieDbLiteContext \_context). The new instance is added to the database by simply adding it to the context’s collection (via Add method), and then calling SaveChangesAsync() against the context. No raw SQL is used in this operation, and consequently there is no SQL injection risk.



### Database Connection Strings

The database connection string for SQL Server utilizes Windows Authentication, also known as Integrated Security (SSPI). This security model is tightly integrated with Windows user and group accounts that are assigned and allowed to access the database.

The primary advantage of this security model is that there is no password management involved, and subsequently, no need to store SQL authentication user and password for the application. Instead, the application authenticates via secure, underlying Windows OS functionality based on the user account that is attempting to access the database. The user account groups that are granted access to the database are managed by administrators and permissions to the database will be limited. Further, groups can be assigned to particular SQL Server [Database-Level Roles](https://docs.microsoft.com/en-us/sql/relational-databases/security/authentication-access/database-level-roles?view=sql-server-ver15) which can limit the access the group member has to database object. For example, a group can be setup such that they have read-only access to the database, whereas another group can be setup to have full read/write access to the database.

For the deployed web application that requires access to the database, the Web Server’s Application Pool User (the proxy user that the web application runs on behalf of) will be granted read/write access to the database tables. Additionally, the App Pool user will be restricted from making any DDL changes (e.g. dropping tables, altering columns, etc.) because all of that should be performed by a DBA with allowed permissions.

Generally, Windows Authentication is a recommended authentication technique for SQL Server if it can be supported. More information about it can be found [here](https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/sql/authentication-in-sql-server).

This is an example of the connection string for a Windows Authentication / SSPI based connection. Notice that there is no user name or password specified, as would be if it used a traditional SQL Authentication connection.

### Network Connection

The Website is hosted using HTTPS secure communication protocol over a computer network. This method of communication encrypts the network data while in-transit between the client and the server, preventing “man-in-the-middle” attacks.

## Database Backup and Recovery

### Automated Database Backups

A backup of the MovieDbLite database will be performed on a nightly basis, and output to the database server and a secondary archive server environment (to achieve some redundancy of the data in the event of a database server failure). These backups will be kept for at least 6 months in the event that recovery of the database is necessary. SQL Server has the ability to schedule automatic backups of the database via the SQL Server Agent Jobs function, as seen [here](https://www.sqlshack.com/multiple-methods-for-scheduling-a-sql-server-backup-automatically/).

Here is an example of a SQL script used to perform database backups for MovieDbLite. It timestamps the backup with the date to ensure no collisions with other backups, as well as identifying when the backup took place.Output from backup:



### Full Database Recovery Model

The [recovery model](https://docs.microsoft.com/en-us/sql/relational-databases/backup-restore/recovery-models-sql-server?view=sql-server-ver15#RMov) for the database will be set to Full. This allows the database to be restored to an arbitrary point of time (e.g. before an error occurred) if that is necessary. The Full recovery model relies on transaction log files of the database to be created and stored. [Backups](https://docs.microsoft.com/en-us/sql/relational-databases/backup-restore/transaction-log-backups-sql-server?view=sql-server-ver15#LogBackupSequence) of the transaction log files will also be required by the DBA, otherwise the log files will grow forever. The backup process for a transaction log file is a similar process to the aforementioned process of backing up a database.

This is an example of how to set the Recovery Model to Full via the Database’s properties using SSMS.



## Using Database Design or CASE Tool

Software engineering tools were used to help create the MovieDbLite database and web application. These tools were used to help with both the Design and Implementation of the database.

### Visual Paradigm v15.2

[Visual Paradigm](https://www.visual-paradigm.com/) was used to draft up the initial conceptual database design for the project. It provides a nice and easy-to-use way of creating ER diagrams by providing a number of automated features (such as easily migrating foreign key column to child entities). The models designed are very customizable in the manner in which they are displayed, how they are organized, etc.

In addition, there were options in the tool to script out the design into a SQL Server compatible script. This made it trivial to baseline the initial MovieDbLite SQL Server database schema and design.



### SQL Management Studio (SSMS) v17.8.1

SSMS was used as the primary database management tool for the project’s logical implementation and creation of advanced feature objects (such as stored procedures, triggers, constraints, indexes, etc.) Further, it contains features to help perform or schedule backups, and generate DDL/DML scripts of the database. It was common during development to generate DDL/DML scripts as minor tweaks or data entries were added to the database. In many cases, data was added into the database via the website front-end. It was then possible for me to script out the DML statements in SSMS so that I could port the local database to another computer (e.g. from my desktop to laptop).

This is an example of the Generate Scripts wizard in SSMS. The “Types of data to script” can be changed from “Data Only” to “Schema Only” to “Schema and Data” if desired:



Here is an example of the output from a generated data script:SSMS also contains a built-in way to diagram databases. This was helpful in visualizing the logical structure of the database after it was exported from Visual Paradigm into SQL. Small tweaks and adjustments had to be made to the schema after conceptualized, in the visual diagramming in SSMS was able to assist in that. More to be discussed with this in the



### Visual Studio 2019

The MovieDbLite web site was developed using the Visual Studio 2019 IDE. The programming languages are C#, JavaScript, HTML, and CSS using the MVC (Model-View-Controller) design pattern in an ASP.NET Core Project.



Entity Framework (EF) Core was an ORM (Object Relational Mapping) used to communicate with the back-end database for some of the basic CRUD operations.



There were several CLI commands that could be executed to automate some of the SQL Server database and Entity Framework operations. For example, the following command could be issued to Visual Studio to generate the c# database model classes based off the database:



Models generated:

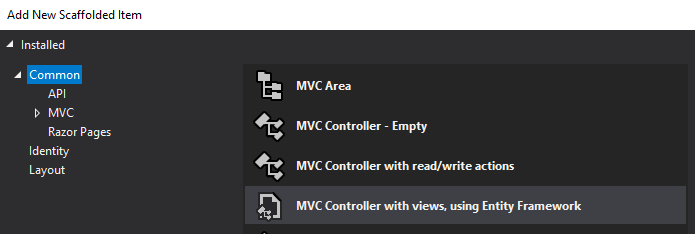


Example of the MovieCastMember model:

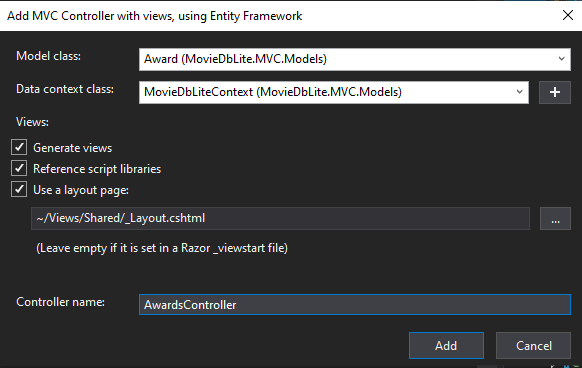


Once all your Models are generated within Visual Studio, there are features to automatically generate controllers (server side endpoints that communicate with database) and views (i.e. webpages, html/css/javascript) to perform CRUD operations for the corresponding model entity/table.

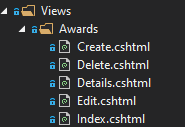
This example shows the creation of ‘Award’ entity Controller/View.



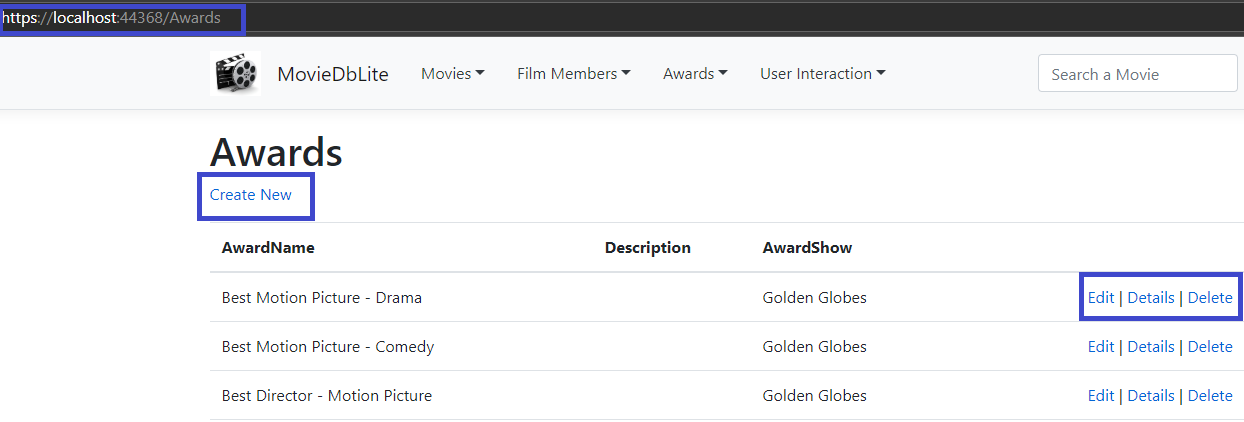
Select the ‘Award’ Model class and specify to generate views based off the Entity Framework data context.



This generates the following Awards Views and controller for populating the Views with data.



In the website, there is now an Awards page available to perform CRUD operations against the Award entity/table.



### GitHub for Git-Based Source Control

The project was version controlled using the GitHub platform and Git. The repository is publicly available here: <https://github.com/stevo9510/MovieDbLiteMVC>



Most of the repository contains the .NET projects required to build the backend, as well as an importer tool to help bring in data from TheMovieDb.org. The database DML and DDL scripts were also version controlled, and available in the project artifacts folder here: [Project Artifacts](Project%20Artifcats)

Using version control was essential to backup, revert, manage changes, etc. to the project throughout the semester. Additionally, it allowed me to work on the project from multiple computers (e.g. some days my desktop, and some days my laptop). One of the things that I would improve upon in the future (in terms of version control) is to independently source control many of the database objects (such as tables, stored procedures, functions, views, etc.) This would allow for isolated tracking of these objects, clearly establishing a history of changes to the individual elements.

Visual Studio contains menus to help commit/push source control changes back to the remote git repository.



## Other Possible E/R Relationships

With database design, there are sometimes multiple ways to solve the same problem – which each way having its respective pros and cons. The design of MovieDbLite was no exception, and there were other considerations when designing the database. These were a few possible alternative E/R relationships:

1. Merge MovieCastMember, MovieCrewMember, and Movie.DirectorFilmMemberId into one single entity (with a structure similar to MovieCrewMember)
   1. This would make the querying for all cast and crew “simpler” (all in one place)
      1. This was resolved by creating a table-valued functions to perform the unioning of the cast and crew members for movies/film members, exposing it out as a single query.
   2. The main drawbacks of one entity structure was as follows:
      1. Performance would be impacted (table would be quite large). Not all places need to query for all cast and crew at once, so fragmenting into multiple tables made sense – especially since the Cast table will be a popular table to be queried.
      2. More difficult to enforce one-to-one relationship between Director and Movie
      3. Role specific information (such as character name and sequence for Actor/Cast Members) required to be stored. This would be unused for Crew members
2. Split FilmMember into multiple entities per role
   1. This would potentially help performance by reducing the amount of film members in the table.
   2. The main drawbacks of splitting the table were:
      1. A lot of redundancy of data between split tables. E.g. a lot of actors may perform other crew member roles, and likewise, crew members may not be in the same role their entire career
      2. More difficult to trace a single film member’s career credits
   3. A hierarchical parent/child structure (e.g. FilmMember base entity, with child Director or Actor entities) was considered as well, however, there were not any “Actor” or “Director” (or whatever role) specific attributes that needed to be tracked yet. This may be a future possibility if role specific information needed to be tracked for a film member.
3. Calculate AverageUserRating on the fly (instead of storing in Movie entity)
   1. This would help mitigate potential for AverageUserRating on Movie entity becoming out of sync (due it being redundant).
      1. A trigger on the MovieUserReview is used to easily resolve this problem and ensure the field stays in sync
   2. It would also slightly help performance on user review CUD operations
      1. The performance impact is negligible because reviews are typically done on a one-by-one basis, and against a particular movie. So, the trigger’s aggregation and lookup to update the corresponding Movie record is cheap.
   3. The main drawbacks for calculating on the fly:
      1. This would be very expensive when performing a search for movies that are above a certain rating because there will be a significant amount of MovieUserReviews in the database
      2. Not easily available to be reported or queried against
4. Store calculated IsHelpful/NotHelpful counts against MovieUserReview
   1. This would be a similar approach to the AverageUserRating, and allow for quickly looking up these details.
   2. Overall, this was a tradeoff between query performance and review performance. It was considered not worth the extra fields at this time because searches for reviews are on a per movie basis, which utilizes an index and will be fast in general. The additional calculations to aggregate the data on the fly are joined up via a clustered index on the MovieUserReviewHelpful entity, which will be very fast as well.
5. Allow user to be tied to multiple user roles
   1. This was considered not worth it at the time because there are only two roles (users and admins). It is easy to set it up in such a way that admins have all privileges and users have a subset of the privileges. Perhaps if more roles are added in the future, it may make sense to allow users to be more than one role.
6. Adding some sort of natural key to Movie
   1. This was very difficult to do. There are not guidelines in place to prevent two Movie’s from having the same name, let alone having the same name and releasing on the same day (in theory).
7. Adding a natural key to FilmMember
   1. This would be an SSN to the entity so that FilmMembers can be uniquely identified by that, however, there are several problems with that:
      1. Security impact. Now sensitive data is at risk of being exposed by the database.
      2. What is the main value of tracking it? There is no value in reporting it (nor could it be because of security), and it could not be searched either.
      3. Not all film members will have an SSN (since it is U.S. specific)
   2. First/Last/and even Preferred Name could not guarantee to be unique
8. Require only an email address for users
   1. This would allow users to sign up by using their email address as their username.
   2. The main drawback for this is that their email can now publicly be posted against a review they write (which would not be good for their inbox).
9. Tracking First Name/Last Name information for Users
   1. The main drawback for this is that it is now Personal Identifiable Information (PII) being tracked within the database, and there is not much benefit of it being there (because reviews will simply show the User Name for privacy reasons)
10. Creating a customized artificial key for FilmMember or Movie that is shorter in length
    1. This would make more simpler URLs references to Movies/FilmMembers, and also a consistent ID across database environments.
    2. This is not trivial to do in SQL Server, so it would be a future requirement if there was more time with the project.

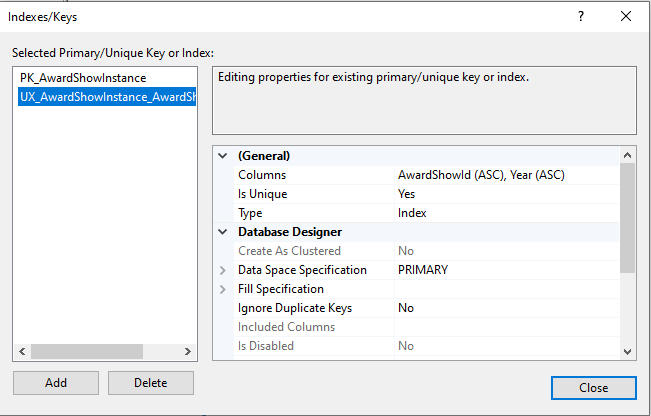
# Implementation Description

Implementation-wise, the logical and actual schema (table/column) design of the database was pretty much in-line with the relational model. I was able to port over my relational model from Visual Paradigm over to SQL Server very easily. SQL Management Studio (SSMS) contains a visual designer (seen in image below) itself that was able to assist in implementing the database.



Example of diagram in SSMS

Many of the design choices described in the previous section, such as indexes, were able to be added/implemented using the SSMS visual designer (as indexes were not setup during the Design phase. In addition, SSMS was able to assist in implementing many of the advanced features of the database, such as stored procedures, functions, triggers, user-defined types, etc. These will be covered in more detail in this section. In addition, queries and reports that the application supports will be covered as well. Furthermore, I cover the naming conventions and standards I used during the implementation as it pertains to SSMS Primary Key Names, Unique Key Names, etc.



Example of adding an index using ssms

## Data Dictionary

This section shows the data dictionary generated by the DBMS. SQL Server contains a stored procedure similar to DESCRIBE which is called sp\_columns. The output of the result from that in each table is shown below. Note that some of the sp\_columns output columns are omitted for brevity (e.g. database name, schema name, etc. that are all the same or uninteresting).

exec sp\_columns 'Award'

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| COLUMN\_NAME | DATA\_TYPE | TYPE\_NAME | PRECISION | LENGTH | SCALE | RADIX | NULLABLE |
| Id | 4 | int identity | 10 | 4 | 0 | 10 | 0 |
| AwardShowId | 5 | smallint | 5 | 2 | 0 | 10 | 0 |
| AwardName | 12 | varchar | 50 | 50 | NULL | NULL | 0 |
| Description | 12 | varchar | 200 | 200 | NULL | NULL | 1 |

exec sp\_columns 'AwardShow'

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| COLUMN\_NAME | DATA\_TYPE | TYPE\_NAME | PRECISION | LENGTH | SCALE | RADIX | NULLABLE |
| Id | 5 | smallint identity | 5 | 2 | 0 | 10 | 0 |
| ShowName | 12 | varchar | 50 | 50 | NULL | NULL | 0 |
| Description | 12 | varchar | 200 | 200 | NULL | NULL | 1 |

exec sp\_columns 'AwardShowInstance'

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| COLUMN\_NAME | DATA\_TYPE | TYPE\_NAME | PRECISION | LENGTH | SCALE | RADIX | NULLABLE |
| Id | 4 | int identity | 10 | 4 | 0 | 10 | 0 |
| AwardShowId | 5 | smallint | 5 | 2 | 0 | 10 | 0 |
| Year | 5 | smallint | 5 | 2 | 0 | 10 | 0 |
| DateHosted | -9 | date | 10 | 20 | NULL | NULL | 0 |

exec sp\_columns 'AwardWinner'

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| COLUMN\_NAME | DATA\_TYPE | TYPE\_NAME | PRECISION | LENGTH | SCALE | RADIX | NULLABLE |
| AwardShowInstanceId | 4 | int | 10 | 4 | 0 | 10 | 0 |
| AwardId | 4 | int | 10 | 4 | 0 | 10 | 0 |
| FilmMemberId | -5 | bigint | 19 | 8 | 0 | 10 | 0 |
| MovieId | -5 | bigint | 19 | 8 | 0 | 10 | 0 |

exec sp\_columns 'FilmMember'

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| COLUMN\_NAME | DATA\_TYPE | TYPE\_NAME | PRECISION | LENGTH | SCALE | RADIX | NULLABLE |
| Id | -5 | bigint identity | 19 | 8 | 0 | 10 | 0 |
| Prefix | 12 | varchar | 10 | 10 | NULL | NULL | 1 |
| FirstName | 12 | varchar | 50 | 50 | NULL | NULL | 0 |
| MiddleName | 12 | varchar | 50 | 50 | NULL | NULL | 1 |
| LastName | 12 | varchar | 50 | 50 | NULL | NULL | 0 |
| Suffix | 12 | varchar | 5 | 5 | NULL | NULL | 1 |
| PreferredFullName | 12 | varchar | 150 | 150 | NULL | NULL | 0 |
| Gender | 1 | char | 1 | 1 | NULL | NULL | 0 |
| DateOfBirth | -9 | date | 10 | 20 | NULL | NULL | 1 |
| DateOfDeath | -9 | date | 10 | 20 | NULL | NULL | 1 |
| Biography | -1 | text | 2147483647 | 2147483647 | NULL | NULL | 1 |

exec sp\_columns 'FilmRole'

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| COLUMN\_NAME | DATA\_TYPE | TYPE\_NAME | PRECISION | LENGTH | SCALE | RADIX | NULLABLE |
| Id | 5 | smallint | 5 | 2 | 0 | 10 | 0 |
| RoleName | 12 | varchar | 50 | 50 | NULL | NULL | 0 |
| Description | 12 | varchar | 200 | 200 | NULL | NULL | 1 |

exec sp\_columns 'Genre'

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| COLUMN\_NAME | DATA\_TYPE | TYPE\_NAME | PRECISION | LENGTH | SCALE | RADIX | NULLABLE |
| Id | 5 | smallint | 5 | 2 | 0 | 10 | 0 |
| GenreName | 12 | varchar | 25 | 25 | NULL | NULL | 0 |
| Description | 12 | varchar | 500 | 500 | NULL | NULL | 1 |

exec sp\_columns 'ImageType'

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| COLUMN\_NAME | DATA\_TYPE | TYPE\_NAME | PRECISION | LENGTH | SCALE | RADIX | NULLABLE |
| Id | 5 | smallint | 5 | 2 | 0 | 10 | 0 |
| ImageExtension | 12 | varchar | 10 | 10 | NULL | NULL | 0 |
| Name | 12 | varchar | 25 | 25 | NULL | NULL | 0 |

exec sp\_columns 'Language'

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| COLUMN\_NAME | DATA\_TYPE | TYPE\_NAME | PRECISION | LENGTH | SCALE | RADIX | NULLABLE |
| LanguageIsoCode | 1 | iso\_code639\_1 | 2 | 2 | NULL | NULL | 0 |
| LanguageName | 12 | varchar | 50 | 50 | NULL | NULL | 0 |

exec sp\_columns 'Movie'

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| COLUMN\_NAME | DATA\_TYPE | TYPE\_NAME | PRECISION | LENGTH | SCALE | RADIX | NULLABLE |
| Id | -5 | bigint identity | 19 | 8 | 0 | 10 | 0 |
| Title | 12 | varchar | 150 | 150 | NULL | NULL | 0 |
| Description | 12 | varchar | 500 | 500 | NULL | NULL | 0 |
| ReleaseDate | -9 | date | 10 | 20 | NULL | NULL | 1 |
| RestrictionRatingId | 5 | smallint | 5 | 2 | 0 | 10 | 1 |
| DirectorFilmMemberId | -5 | bigint | 19 | 8 | 0 | 10 | 1 |
| DurationInMinutes | 4 | int | 10 | 4 | 0 | 10 | 1 |
| AverageUserRating | 3 | decimal | 5 | 7 | 2 | 10 | 1 |

exec sp\_columns 'Movie\_Genre'

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| COLUMN\_NAME | DATA\_TYPE | TYPE\_NAME | PRECISION | LENGTH | SCALE | RADIX | NULLABLE |
| MovieId | -5 | bigint | 19 | 8 | 0 | 10 | 0 |
| GenreId | 5 | smallint | 5 | 2 | 0 | 10 | 0 |

exec sp\_columns 'Movie\_Language'

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| COLUMN\_NAME | DATA\_TYPE | TYPE\_NAME | PRECISION | LENGTH | SCALE | RADIX | NULLABLE |
| MovieId | -5 | bigint | 19 | 8 | 0 | 10 | 0 |
| LanguageIsoCode | 1 | iso\_code639\_1 | 2 | 2 | NULL | NULL | 0 |

exec sp\_columns 'MovieCastMember'

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| COLUMN\_NAME | DATA\_TYPE | TYPE\_NAME | PRECISION | LENGTH | SCALE | RADIX | NULLABLE |
| MovieId | -5 | bigint | 19 | 8 | 0 | 10 | 0 |
| ActorFilmMemberId | -5 | bigint | 19 | 8 | 0 | 10 | 0 |
| CharacterName | 12 | varchar | 150 | 150 | NULL | NULL | 0 |
| Sequence | 4 | int | 10 | 4 | 0 | 10 | 1 |

exec sp\_columns 'MovieCrewMember'

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| COLUMN\_NAME | DATA\_TYPE | TYPE\_NAME | PRECISION | LENGTH | SCALE | RADIX | NULLABLE |
| MovieId | -5 | bigint | 19 | 8 | 0 | 10 | 0 |
| FilmMemberId | -5 | bigint | 19 | 8 | 0 | 10 | 0 |
| FilmRoleId | 5 | smallint | 5 | 2 | 0 | 10 | 0 |

exec sp\_columns 'MovieImage'

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| COLUMN\_NAME | DATA\_TYPE | TYPE\_NAME | PRECISION | LENGTH | SCALE | RADIX | NULLABLE |
| Id | -5 | bigint | 19 | 8 | 0 | 10 | 0 |
| MovieId | -5 | bigint | 19 | 8 | 0 | 10 | 0 |
| ImageName | 12 | varchar | 100 | 100 | NULL | NULL | 0 |
| ImageTypeId | 5 | smallint | 5 | 2 | 0 | 10 | 0 |
| Description | 12 | varchar | 500 | 500 | NULL | NULL | 1 |
| FileContents | -4 | image | 2147483647 | 2147483647 | NULL | NULL | 0 |
| DateUploaded | -9 | datetime2 | 27 | 54 | NULL | NULL | 0 |

exec sp\_columns 'MovieUserReview'

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| COLUMN\_NAME | DATA\_TYPE | TYPE\_NAME | PRECISION | LENGTH | SCALE | RADIX | NULLABLE |
| Id | -5 | bigint identity | 19 | 8 | 0 | 10 | 0 |
| MovieId | -5 | bigint | 19 | 8 | 0 | 10 | 0 |
| UserId | 4 | int | 10 | 4 | 0 | 10 | 0 |
| Rating | 5 | smallint | 5 | 2 | 0 | 10 | 0 |
| Review | 12 | varchar | 8000 | 8000 | NULL | NULL | 1 |
| DatePosted | -9 | datetime2 | 27 | 54 | NULL | NULL | 0 |

exec sp\_columns 'MovieUserReviewHelpful'

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| COLUMN\_NAME | DATA\_TYPE | TYPE\_NAME | PRECISION | LENGTH | SCALE | RADIX | NULLABLE |
| MovieUserReviewId | -5 | bigint | 19 | 8 | 0 | 10 | 0 |
| UserId | 4 | int | 10 | 4 | 0 | 10 | 0 |
| IsHelpful | -7 | bit | 1 | 1 | NULL | NULL | 0 |

exec sp\_columns 'RestrictionRating'

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| COLUMN\_NAME | DATA\_TYPE | TYPE\_NAME | PRECISION | LENGTH | SCALE | RADIX | NULLABLE |
| Id | 5 | smallint | 5 | 2 | 0 | 10 | 0 |
| Code | 12 | varchar | 10 | 10 | NULL | NULL | 0 |
| ShortDescription | 12 | varchar | 50 | 50 | NULL | NULL | 0 |
| LongDescription | 12 | varchar | 200 | 200 | NULL | NULL | 0 |
| IsActive | -7 | bit | 1 | 1 | NULL | NULL | 0 |

exec sp\_columns 'User'

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| COLUMN\_NAME | DATA\_TYPE | TYPE\_NAME | PRECISION | LENGTH | SCALE | RADIX | NULLABLE |
| Id | 4 | int identity | 10 | 4 | 0 | 10 | 0 |
| UserRoleId | 5 | smallint | 5 | 2 | 0 | 10 | 0 |
| UserName | 12 | varchar | 25 | 25 | NULL | NULL | 0 |
| EmailAddress | 12 | varchar | 255 | 255 | NULL | NULL | 0 |
| HashedPassword | 12 | varchar | 60 | 60 | NULL | NULL | 0 |

exec sp\_columns 'UserRole'

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| COLUMN\_NAME | DATA\_TYPE | TYPE\_NAME | PRECISION | LENGTH | SCALE | RADIX | NULLABLE |
| Id | 5 | smallint | 5 | 2 | 0 | 10 | 0 |
| RoleName | 12 | varchar | 25 | 25 | NULL | NULL | 0 |
| Description | 12 | varchar | 500 | 500 | NULL | NULL | 0 |

## Advanced Features

There were numerous advanced features implemented in this project to help enforce business rules or simplify logic.

### Triggers

A trigger was added to the MovieUserReview table to update the AverageUserRating column on the Movie table based on the aggregated average all the reviews for the Movie. Therefore, AverageUserRating can be considered a derived field based on the MovieUserReview records. This trigger ensures that the average stays up to date with any change (Insert/Update/Delete) to the MovieUserReview table. The purpose of the AverageUserRating column is to make lookups/searches against it cheap (without having to do an aggregation at the time of the query, which could be expensive since the MovieUserReview table will be very large), because that would be a common search criterion for users looking for movies to watch. This is what the trigger looks like:

CREATE TRIGGER [dbo].[tg\_MovieUserReview\_UpdateMovieReviewAverage]

ON [dbo].[MovieUserReview]

AFTER INSERT,DELETE,UPDATE

AS

BEGIN

-- SET NOCOUNT ON added to prevent extra result sets from

-- interfering with SELECT statements.

SET NOCOUNT ON;

-- Gather up all the Movie Ids of affected Movies into a temp table.

-- We'll recalculate the averages for all of them.

-- This temp table is primarily for readability.

SELECT MovieId

INTO #temp\_AffectedMovies

FROM inserted -- inserted and deleted are built in tables that SQL Server gives access to on triggers for altered records.

UNION

SELECT MovieId

FROM deleted

;WITH MovieAverageRatings

AS

(

SELECT #temp\_AffectedMovies.MovieId,

CAST(AVG(MovieUserReview.Rating \* 1.0) as decimal(5,2)) as AverageRating

FROM #temp\_AffectedMovies

LEFT JOIN MovieUserReview ON MovieUserReview.MovieId = #temp\_AffectedMovies.MovieId

GROUP BY #temp\_AffectedMovies.MovieId

)

UPDATE Movie

SET AverageUserRating = MovieAverageRatings.AverageRating

FROM Movie

INNER JOIN MovieAverageRatings ON MovieAverageRatings.MovieId = Movie.Id

DROP TABLE #temp\_AffectedMovies

END

GO

### Views

A View was created to greatly simplify the retrieval of Movie Award Winner information, and allow easily filtering on it. This view is used in part of a report displayed in the application website. It returns all winners for awards over the years with the award name, award show name, winner’s preferred full name, and movie name. This view contains five joins to retrieve the data it requires, and using the view greatly increases code readability.

CREATE VIEW [dbo].[vw\_AwardWinnerInfo] AS

SELECT

Award.Id as AwardId,

Award.AwardName,

AwardShow.Id as AwardShowId,

AwardShow.ShowName,

AwardShowInstance.Year,

AwardWinner.MovieId,

Movie.Title,

FilmMember.Id as FilmMemberId,

FilmMember.PreferredFullName,

AwardShowInstance.DateHosted

FROM AwardWinner

INNER JOIN AwardShowInstance ON AwardShowInstance.Id = AwardWinner.AwardShowInstanceId

INNER JOIN Award ON Award.Id = AwardWinner.AwardId

INNER JOIN FilmMember ON FilmMember.Id = AwardWinner.FilmMemberId

INNER JOIN Movie ON Movie.Id = AwardWinner.MovieId

INNER JOIN AwardShow ON AwardShow.Id = AwardShowInstance.AwardShowId

GO

### Table-Valued Functions

Table-Valued Functions were created to simplify the retrieval of certain information that required a parameter being passed to it. Specifically, the table-valued functions used help in retrieving cast/crew/movie related information.

#### tvf\_GetAllMovieFilmMembers

This shows a Table-Valued Function used to combine all the movie film members (cast, crew, and director) of a particular movie. It takes in a MovieId and unions the film members from the corresponding MovieCastMember, MovieCrewMember, and Movie tables. The FilmRoleId is returned with each FilmMember to identify what role they played in the movie. The Director and Actor are implied based on their corresponding column and table (i.e. the Movie.DirectorFilmMemberId is a director role, and any value in the MovieCastMember table is an actor role).

This function is important to help retrieve the cast and credits for a particular movie. For example, a webpage that shows the movie’s cast/crew would utilize this function.

CREATE FUNCTION [dbo].[tvf\_GetAllMovieFilmMembers]

(

-- Add the parameters for the function here

@MovieId bigint

)

RETURNS TABLE

AS

RETURN

(

SELECT

MovieCastMember.ActorFilmMemberId as FilmMemberId,

cast(2 as int) as FilmRoleId, -- @FilmRoleId\_Actor

MovieCastMember.CharacterName,

MovieCastMember.Sequence

FROM MovieCastMember

WHERE MovieCastMember.MovieId = @MovieId

UNION

SELECT

MovieCrewMember.FilmMemberId,

MovieCrewMember.FilmRoleId,

NULL, -- CharacterName

NULL -- sequence

FROM MovieCrewMember

WHERE MovieCrewMember.MovieId = @MovieId

UNION

SELECT

Movie.DirectorFilmMemberId,

3, -- @FilmRoleId\_Director

NULL, -- CharacterName

NULL -- sequence

FROM Movie

WHERE Movie.Id = @MovieId

)

Here is an example of this Table-Valued Function being used to return details about the Titanic film’s cast and crew.

DECLARE @MovieId\_Titanic int = (SELECT id FROM Movie WHERE Title = 'Titanic' AND YEAR(ReleaseDate) = 1997)

SELECT \*

FROM tvf\_GetAllMovieFilmMembers(@MovieId\_Titanic) mFilmMember

Results:



#### tvf\_GetAllFilmMemberMovies

This shows a Table-Valued Function used to combine all the movies a film member has been a part of, as well as the role that film member played in contributing to the movie. It takes in a FilmId and unions the Movies from the corresponding MovieCastMember, MovieCrewMember, and Movie tables. The FilmRoleId is returned with each Movie to identify what role they played in the movie. The Director and Actor are implied based on their corresponding column and table (i.e. the Movie.DirectorFilmMemberId is a director role, and any value in the MovieCastMember table is an actor role).

This function is important to help retrieve the movies a particular film member has participated in. For example, a webpage that shows the film member’s previous work would utilize this function.

-- =============================================

-- Author: Steven Anderson

-- Create date: 04/05/2020

-- Description: Get all Movies that a Film Member plays a role in (Cast, Crew, Director)

-- =============================================

CREATE FUNCTION [dbo].[tvf\_GetAllFilmMemberMovies]

(

-- Add the parameters for the function here

@FilmMemberId bigint

)

RETURNS TABLE

AS

RETURN

(

SELECT

MovieCastMember.MovieId,

cast(2 as int) as FilmRoleId, -- @FilmRoleId\_Actor

MovieCastMember.CharacterName

FROM MovieCastMember

WHERE MovieCastMember.ActorFilmMemberId = @FilmMemberId

UNION

SELECT

MovieCrewMember.MovieId,

MovieCrewMember.FilmRoleId,

NULL -- CharacterName

FROM MovieCrewMember

WHERE MovieCrewMember.FilmMemberId = @FilmMemberId

UNION

SELECT

Movie.Id,

3, -- @FilmRoleId\_Director

NULL -- CharacterName

FROM Movie

WHERE Movie.DirectorFilmMemberId = @FilmMemberId

)

GO

Usage of this function:

DECLARE @FilmMemberId\_Dicaprio bigint =

(SELECT id FROM FilmMember WHERE PreferredFullName = 'Leonardo DiCaprio')

SELECT \*

FROM dbo.tvf\_GetAllFilmMemberMovies(@FilmMemberId\_Dicaprio) f

Results:



### User-Defined Table Types

Simple user-defined table types were created to help with stored procedure inserts for multiple records at once.

#### ut\_GenreIdList

Used to pass a list of Genre Ids into a stored procedure all at once so that they could be associated with a Movie (via the Movie\_Genre table). The MovieId is not included within this type because it is assumed that this would be used on a per Movie basis, and therefore the MovieId would be specified as a different parameter.

CREATE TYPE dbo.ut\_GenreIdList AS TABLE(

[GenreId] [int] NOT NULL,

PRIMARY KEY CLUSTERED

(

[GenreId] ASC

)WITH (IGNORE\_DUP\_KEY = OFF)

)

#### ut\_LanguageIsoCodeList

Used to pass a list of LanguageIsoCodes into a stored procedure all at once so that they could be associated with a Movie (via the Movie\_Language table). The MovieId is not included within this type because it is assumed that this would be used on a per Movie basis, and therefore the MovieId would be specified as a different parameter.

CREATE TYPE dbo.ut\_LanguageIsoCodeList AS TABLE(

[LanguageIsoCode] [dbo].[iso\_code639\_1] NOT NULL,

PRIMARY KEY CLUSTERED

(

[LanguageIsoCode] ASC

)WITH (IGNORE\_DUP\_KEY = OFF))

#### ut\_MovieCastMember

Used to pass a collection of MovieCastMembers into a stored procedure all at once so that they can be inserted into the database. In this scenario, the MovieId is actually included as a parameter because there might be situations where the CastMember is having their profile of Movies they cast in populated for them in bulk.

CREATE TYPE [dbo].[ut\_MovieCastMember] AS TABLE(

[MovieId] [bigint] NOT NULL,

[ActorFilmMemberId] [bigint] NOT NULL,

[CharacterName] [varchar](150) NOT NULL,

[Sequence] [int] NULL,

PRIMARY KEY CLUSTERED

(

[MovieId] ASC,

[ActorFilmMemberId] ASC

)WITH (IGNORE\_DUP\_KEY = OFF)

)

#### ut\_MovieCrewMember

Similar to MovieCastMember, but for MovieCrewMember.

CREATE TYPE [dbo].[ut\_MovieCrewMember] AS TABLE(

[MovieId] [bigint] NOT NULL,

[FilmMemberId] [bigint] NOT NULL,

[FilmRoleId] [varchar](150) NULL,

PRIMARY KEY CLUSTERED

(

[MovieId] ASC,

[FilmMemberId] ASC

)WITH (IGNORE\_DUP\_KEY = OFF)

)

### Stored Procedures

Stored procedures were used to simplify DML statements and/or add additional data integrity error checks that are more complicated to implement via constraints. Some of them utilize the user-defined table-types mentioned previously.

Note: See

#### usp\_InsertMovieDetails

This stored procedure makes for an easy way to insert basic movie information all in one function call. It is used by the website to easily add new movies to the database in one call. Basic movie details include the information in the Movie table as well as the Genres/Languages assigned to the movie (note: other information like cast members, crew members, and images can be added separately). The Genres/Languages are passed to the procedure using the aforementioned user-defined table-types, and then call into child stored procedures that will be covered below. Here is the stored procedure as a whole:

ALTER PROCEDURE [dbo].[usp\_InsertMovieDetails]

@Title varchar(150),

@Description varchar(500),

@ReleaseDate date,

@RestrictionRatingId smallint,

@DirectorFilmMemberId bigint,

@DurationInMinutes int,

@LanguageIsoCodes ut\_LanguageIsoCodeList readonly,

@GenreIds ut\_GenreIdList readonly,

@MovieId bigint output

AS

BEGIN

-- SET NOCOUNT ON added to prevent extra result sets from

-- interfering with SELECT statements.

SET NOCOUNT ON;

INSERT INTO Movie

(

Title,

Description,

ReleaseDate,

RestrictionRatingId,

DirectorFilmMemberId,

DurationInMinutes

)

VALUES

(

@Title,

@Description,

@ReleaseDate,

@RestrictionRatingId,

@DirectorFilmMemberId,

@DurationInMinutes

)

SET @MovieId = SCOPE\_IDENTITY()

EXEC usp\_UpdateMovieGenres @MovieId, @GenreIds

EXEC usp\_UpdateMovieLanguages @MovieId, @LanguageIsoCodes

END

#### usp\_UpdateMovieGenres

This stored procedure takes in a Movie Id and a user-defined table-type of all the Genres that should be the newly assigned Genres. It performs a MERGE operation against the Movie in order to INSERT any new Genres, and DELETE any genres that previously were assigned but no longer are. This type of setup allows for a very reusable stored procedure, as it can be used by parts of the application to quickly update/correct the Genre’s associated with a Movie without involving itself with other details of the Movie. As mentioned before, it’s also used by the usp\_InsertMovieDetails procedure to insert the initially known Genres of a movie.

ALTER PROCEDURE [dbo].[usp\_UpdateMovieGenres]

-- Add the parameters for the stored procedure here

@MovieId bigint,

@GenreIds ut\_GenreIdList readonly

AS

BEGIN

-- SET NOCOUNT ON added to prevent extra result sets from

-- interfering with SELECT statements.

SET NOCOUNT ON;

MERGE Movie\_Genre as trg

USING @GenreIds AS src

ON src.GenreId = trg.GenreId

AND trg.MovieId = @MovieId

WHEN NOT MATCHED BY TARGET THEN

INSERT (MovieId, GenreId)

VALUES (@MovieId, src.GenreId)

WHEN NOT MATCHED BY SOURCE AND trg.MovieId = @MovieId THEN

DELETE;

END

#### usp\_UpdateMovieLanguages

Similar to the Movie Genres stored procedure, but for Languages. However, MERGE could not be used in this scenario because SQL Server has a limitation with data types that contain Rules (which are mentioned in the upcoming constraints section), and the ut\_LanguageIsoCodeList contains a user-defined data type that has a rule associated with it. So this stored procedure simulates a MERGE with the full INSERT/DELETE statements.

ALTER PROCEDURE [dbo].[usp\_UpdateMovieLanguages]

-- Add the parameters for the stored procedure here

@MovieId bigint,

@LanguageIsoCodes ut\_LanguageIsoCodeList readonly

AS

BEGIN

-- SET NOCOUNT ON added to prevent extra result sets from

-- interfering with SELECT statements.

SET NOCOUNT ON;

-- Note: MERGE statement cannot be used in this case because

-- ut\_LanguageIsoCodeList contains a user-defined data type that has

-- a rule against it (and SQL Server does not allow MERGE to be used if that

-- is the case)

INSERT INTO Movie\_Language

(

MovieId,

LanguageIsoCode

)

SELECT @MovieId,

LanguageIsoCode

FROM @LanguageIsoCodes isosToInsert

WHERE NOT EXISTS

(

SELECT 1

FROM Movie\_Language

WHERE Movie\_Language.MovieId = @MovieId

AND Movie\_Language.LanguageIsoCode = isosToInsert.LanguageIsoCode

)

DELETE FROM Movie\_Language

WHERE Movie\_Language.MovieId = @MovieId

AND Movie\_Language.LanguageIsoCode NOT IN

(SELECT LanguageIsoCode FROM @LanguageIsoCodes)

END

#### usp\_InsertMovieWinner

This stored procedure is used to insert a new AwardWinner record. The primary purpose of it is that it contains some basic data integrity error checks to ensure valid data is inserted. It ensures that the Award given to the winner is actually an Award that given out by the AwardShow (that is associated with the AwardShowInstance). In other words, if the Award is not valid for that AwardShowInstance, the procedure raises an error and returns. In addition, the procedure ensures that the FilmMember actually worked on that Movie, via a call to the tvf\_GetAllMovieFilmMembers table-valued function. If the FilmMember did not work on the Movie, then they should not be awarded for the Movie.

The website application will also contain input validation logic to help mitigate bad parameters being passed to this stored procedure. However, there is the chance the user could “spoof” (or alter parameters passed to) the REST API endpoint that calls this procedure, and so having this database level validation is good rather than relying on the client.

ALTER PROCEDURE [dbo].[usp\_InsertAwardWinner]

-- Add the parameters for the stored procedure here

@AwardShowInstanceId int,

@AwardId int,

@FilmMemberId bigint,

@MovieId bigint

AS

BEGIN

-- SET NOCOUNT ON added to prevent extra result sets from

-- interfering with SELECT statements.

SET NOCOUNT ON;

-- Error check

IF NOT EXISTS

(

SELECT 1

FROM AwardShowInstance

INNER JOIN AwardShow ON AwardShow.Id = AwardShowInstance.AwardShowId

INNER JOIN Award ON Award.AwardShowId = AwardShow.Id

WHERE AwardShowInstance.Id = @AwardShowInstanceId

AND Award.Id = @AwardId

)

BEGIN

raiserror('The specified Award is not awarded for this Award Show. The update has been cancelled.', 16, 1)

RETURN;

END

IF NOT EXISTS

(

SELECT 1

FROM tvf\_GetAllMovieFilmMembers(@MovieId)

WHERE FilmMemberId = @FilmMemberId

)

BEGIN

raiserror('The specified film member was not involved of the making of the awarded film. The update has been cancelled.', 16, 1)

RETURN;

END

INSERT INTO AwardWinner

(

AwardShowInstanceId,

AwardId,

FilmMemberId,

MovieId

)

VALUES

(

@AwardShowInstanceId,

@AwardId,

@FilmMemberId,

@MovieId

)

END

### Constraints, User-Defined Data Types, and Rules

In addition to some of the basic constraints listed in design, more advanced DBMS constraints were used. These were a few examples of constraints used:

1. The FilmMember Gender field allows for either an ‘M’ for Male or ‘F’ or Female. A special constraint was added to the column to enforce it.

ADD CONSTRAINT [CK\_FilmMember\_Gender] CHECK (([Gender]='M' OR [Gender]='F'))

If you try to perform an insert/update that violates this constraint, the DBMS enforces it. For example:

update FilmMember

set Gender = 'B'

where PreferredFullName = 'Leonardo DiCaprio'

Results in this error message, and does not commit to the database:

The UPDATE statement conflicted with the CHECK constraint "CK\_FilmMember\_Gender".

1. The MovieUserReview Rating column allows for a value between 1 and 10 inclusively. A special constraint was added to the column to enforce it.

ADD CONSTRAINT [CK\_MovieUserReview\_Rating] CHECK (([Rating]>=(1) AND [Rating]<=(10)))

Similarly, an insert/update to this with an invalid value will violate the constraint and not commit the change:

INSERT INTO MovieUserReview

(

MovieId,

UserId,

Rating,

Review,

DatePosted

)

VALUES

(

@MovieId,

@UserId,

12, -- Invalid rating (not between 1 and 10)

'great movie 12 out of 10',

SYSUTCDATETIME()

)

1. The AwardShowInstance [Year] column is enforced as a positive number with a length of four (we can reasonably assume that the database will not be used after the year 9999, and there were no Award show instances prior to the year 1000). This constraint could arguably be stricter (e.g. between 1900 and 9999) but, this is good enough to catch some basic typos or mistakes.

ADD CONSTRAINT [CK\_AwardShowInstance\_Year] CHECK (([Year]>(0) AND len([Year])=(4)))

1. The Language LanguageIsoCode column / primary key was created as a special User-Defined Data Type called ‘iso\_code639\_1’. The Language [ISO Code 639-1](https://en.wikipedia.org/wiki/List_of_ISO_639-1_codes) are strictly defined as two-letter codes to identify different languages.

The base-type is defined as this (a two-letter char instead of varchar since the length must be two).

CREATE TYPE [dbo].[iso\_code639\_1] FROM [char](2) NOT NULL

Then, a custom rule was created and applied to the Data-Type to enforce that the specified values are two-letters (and no numeric digits are used).

CREATE RULE [dbo].[iso\_code\_rule]

AS

@iso\_code639\_1 like '[A-Z][A-Z]'

The rule is bound to the data type using the sp\_bindrule system-level stored procedure:

EXEC sp\_bindrule 'iso\_code\_rule', 'iso\_code639\_1';

The Language.LanguageIsoCode column is specified as the iso\_code639\_1 type, as seen below. Similarly, the Movie\_Language.LanguageIsoCode column on that entity is defined as the same data type.

CREATE TABLE [dbo].[Language](

[LanguageIsoCode] [dbo].[iso\_code639\_1] NOT NULL,

[LanguageName] [varchar](50) NOT NULL,

…

Now, any inserts/updates to the LanguageIsoCode column that violates the two-letter rule will be enforced by the DBMS. For example:

INSERT INTO [Language]

(

LanguageIsoCode,

LanguageName

)

VALUES

(

'p5', -- This contains a number, which violates the rule

'Pig Latin'

)

Results in the following error output:

A column insert or update conflicts with a rule imposed by a previous CREATE RULE statement. The statement was terminated.

1. Other constraints in the system are handled either at the stored procedure/trigger level, or within higher-level language application logic depending on complexity. For example, email address format can be tricky to validate within the database. A regex within the application can be used to validate the user’s input before it reaches the database to ensure it is a valid email address.

### Naming Standards and Conventions

The implementation of the project design tries to maintain consistency from a naming standards and conventions perspective.

The following naming standards and guidelines were adopted by the project:

1. Table Names are PascalCased
   1. E.g. MovieCastMember (as opposed to MOVIE\_CAST\_MEMBER or movieCastMember)
2. Column Naming is PascalCased.
3. Artificial id primary key columns are always named “Id”
4. Display name columns are generally named {Entity}Name, and are required
   1. E.g. GenreName, AwardName
   2. There are a few exceptions for brevity, but this is the general rule
5. Unique Index/Key Constraints
   1. Unique keys are named UX\_{TableName}\_{columnNamesUnderscoreSeparated}
   2. E.g. UX\_AwardShowInstance\_AwardShowId\_Year
   3. Unique indexes are put on the natural key of the table (if an artificial key already exists)
6. Regular Index Guidelines
   1. Name IX\_{TableName}\_{columnNamesUnderscoreSeparated}
   2. E.g. IX\_Movie\_Title
7. Primary Key Naming
   1. Named PK\_{TableName}
   2. E.g. PK\_MovieCastMember
8. Foreign Key Naming
   1. Named FK\_{ChildTableName}\_{ParentTableName}
   2. E.g. FK\_Movie\_RestrictionRating for Movie.RestrictionRatingId
9. Constraint naming
   1. Named CK\_{TableName}\_{columns}
   2. E.g. CK\_MovieUserRating\_Rating
10. View Naming
    1. Named vw\_{PrimaryTableName}{Descriptor}
    2. E.g. vw\_AwardWinnerInfo
11. Trigger Naming
    1. Named tg\_{TableName}\_{Description}
    2. E.g. tg\_MovieUserReview\_UpdateMovieReviewAverage
12. Stored Procedure
    1. Named usp\_{InsertUpdateDelete}{TableNameOrDescriptor}
    2. E.g. usp\_InsertAwardWinner
    3. “Usp” stands for user-stored procedure, and is prefixed differently to prevent confusion against system level stored procedures that start with an “sp” (e.g. sp\_columns)
13. Table-Valued Function
    1. Named tvf\_{Descriptor}
    2. E.g. tvf\_GetAllFilmMemberMovies
14. User-Defined Table Types
    1. Named ut\_{TableOrColumnName}{Descriptor}
    2. E.g. ut\_GenreIdList, ut\_MovieCastMember
15. Film vs. Movie Naming
    1. No hard rule here. They are pretty much synonymous. Used what felt right for the table name. It seemed that film industry members want to be held more prestigiously than just “Movies” (and many of them have made non-box-office films) and so stuck with formal terminology of FilmMember/FilmRole for that.

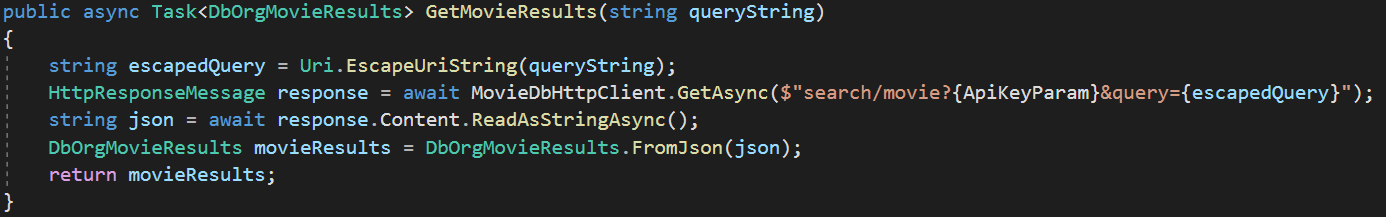
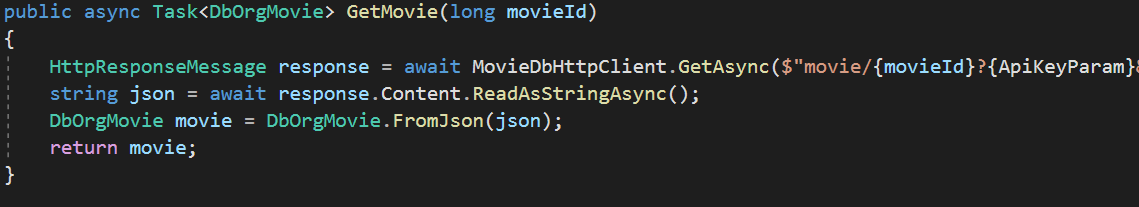
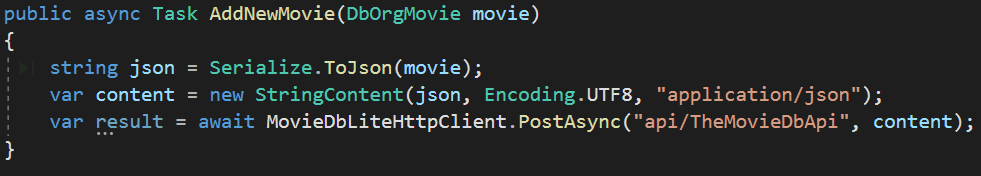
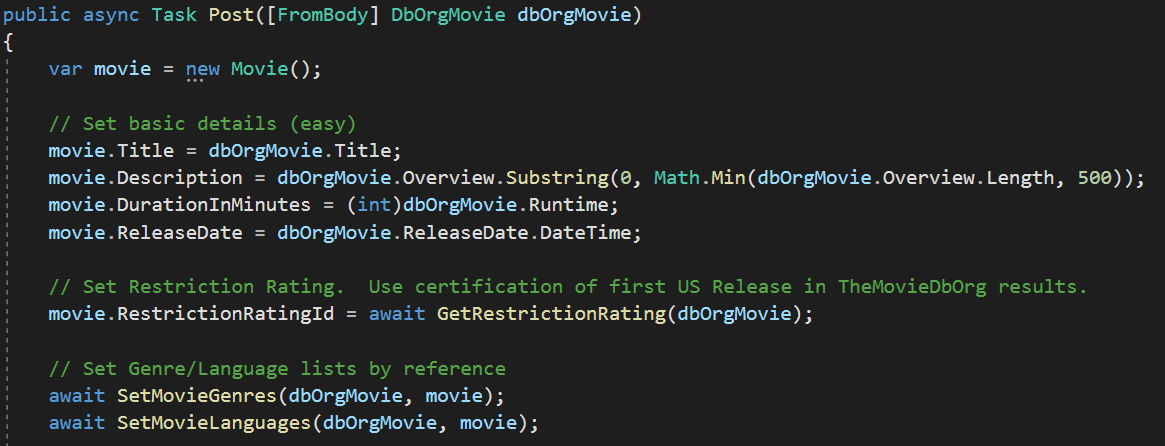
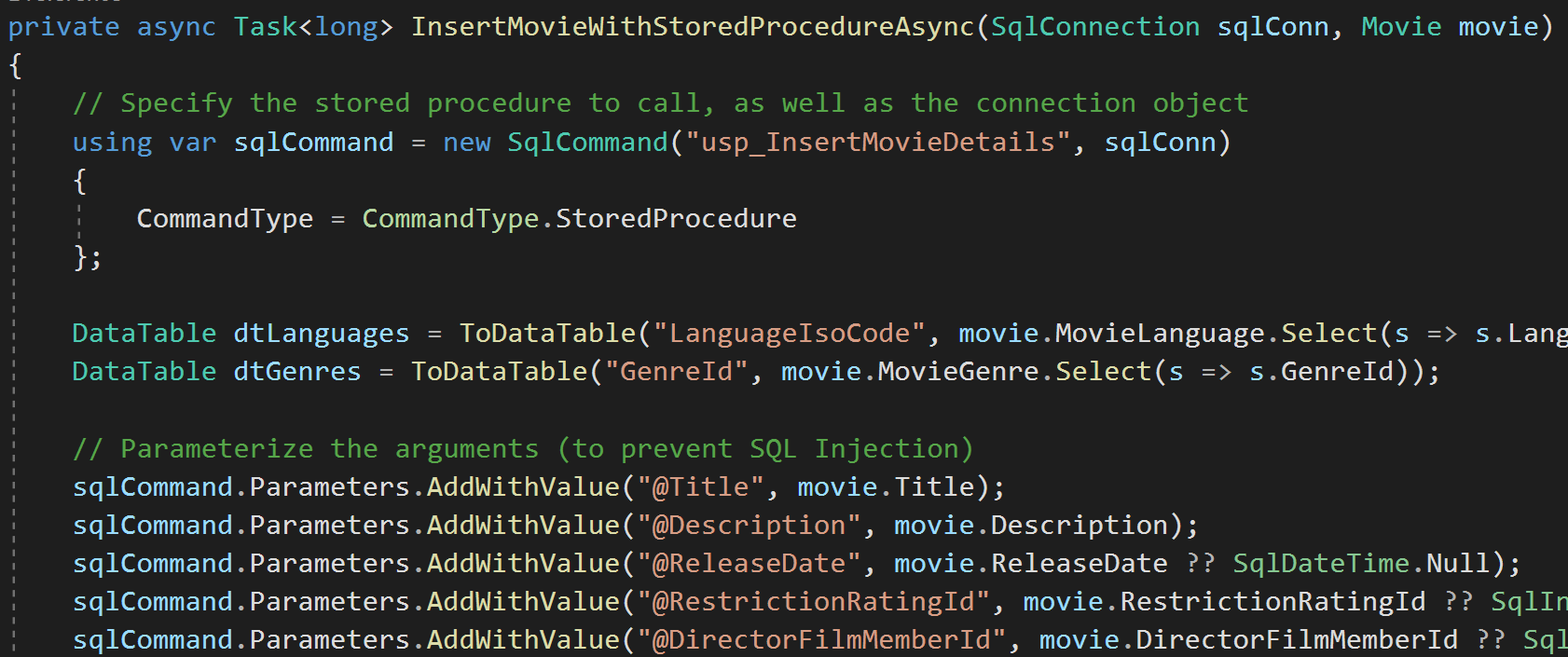
### Importing Data from TheMovieDb.org API

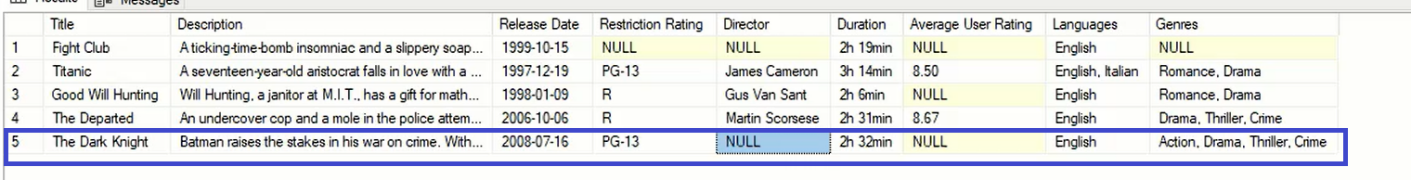
I recorded a video to demonstrate this section. You can access the video here that explains this section in full detail (showing the API website, stepping through code, etc.):

* [Project Demonstrations Folder (OneDrive)](https://1drv.ms/u/s!AiO5RubW7BzfgYVB_W1A-iQE4Qavug?e=ImbTMG)
  + You can download either one of these two files to view the demonstration. Note that the .7z version is available (if you have that compression tool available) because it’s much smaller than the regular .zip version
    - 7zApiDemo.7z
    - ZipApiDemo.zip

One of the capabilities I was able to demonstrate with this project is the ability to consume data from external/existing Movie Database APIs. This helped me populate certain data in my MovieDbLite database without having to hand-jam it all in.

I discovered the <https://www.themoviedb.org/> website and noticed that they had an API available free of use. I was able to sign up for it and get an API key within minutes. The [API documentation page](https://developers.themoviedb.org/3/) documents all the endpoints exposed by the APIs, with request parameter details, and response information. For my demonstration purposes, I utilized two API calls: /search/movie (Search Movies) and /movie/{movie\_id} (Movie Details). These were the following high-level steps to import movie data into my database. Some minor nuances in URLs are left out for brevity:

1. Make HTTP get request to [https://api.themoviedb.org/3/search/movie?query={movieName}](https://api.themoviedb.org/3/search/movie?query=%7bmovieName%7d)
   1. This returns the JSON movie results (for my purposes, I passed “The Dark Knight” as the movie name), and deserializes it into an object
   2. Purpose of this request is to get the Movie\_id of a Movie to use in the next step.
   3. 
2. Make HTTP get request to [https://api.themoviedb.org/3/movie/{movie\_id}](https://api.themoviedb.org/3/movie/%7bmovie_id)
   1. Use movie\_id retrieved from step #1 for this request
   2. This retrieves the details of a given movie (in JSON, and also deserializes it to an object)
   3. 
3. Make HTTP post request to my MovieDbLite REST API at api/TheMovieDbApi
   1. 
4. Post request maps the DbOrgMovie object to the MovieDbLite Movie object
   1. 
5. A stored procedure is called to write the data into MovieDbLite
   1. 
6. This is the new record in the database after the stored procedure executes:



Overall, the consumption of TheMovieDb.org API and import into the MovieDbLite database was a good proof of concept to show the potential of data exchange between the services. Future iterations of the project could import more information from the API, such as film members and award information.

### MovieDbLite Website

A front-end website was built for the MovieDbLite project using Microsoft’s ASP.NET Core MVC framework for building web applications, as well as leveraging the Entity Framework (EF) Object Relational Mapper (ORM) Library to perform many of the database operations in the database.

A demo video of the website can be downloaded here:

* [Project Demonstrations Folder (OneDrive)](https://1drv.ms/u/s!AiO5RubW7BzfgYVB_W1A-iQE4Qavug?e=ImbTMG)
  + You can download either one of these two files to view the demonstration. Note that the .7z version is available (if you have that compression tool available) because it’s much smaller than the regular .zip version
    - 7zWebsiteDemo.7z
    - ZipWebsiteDemo.zip

All the source code from the website is available on GitHub here:

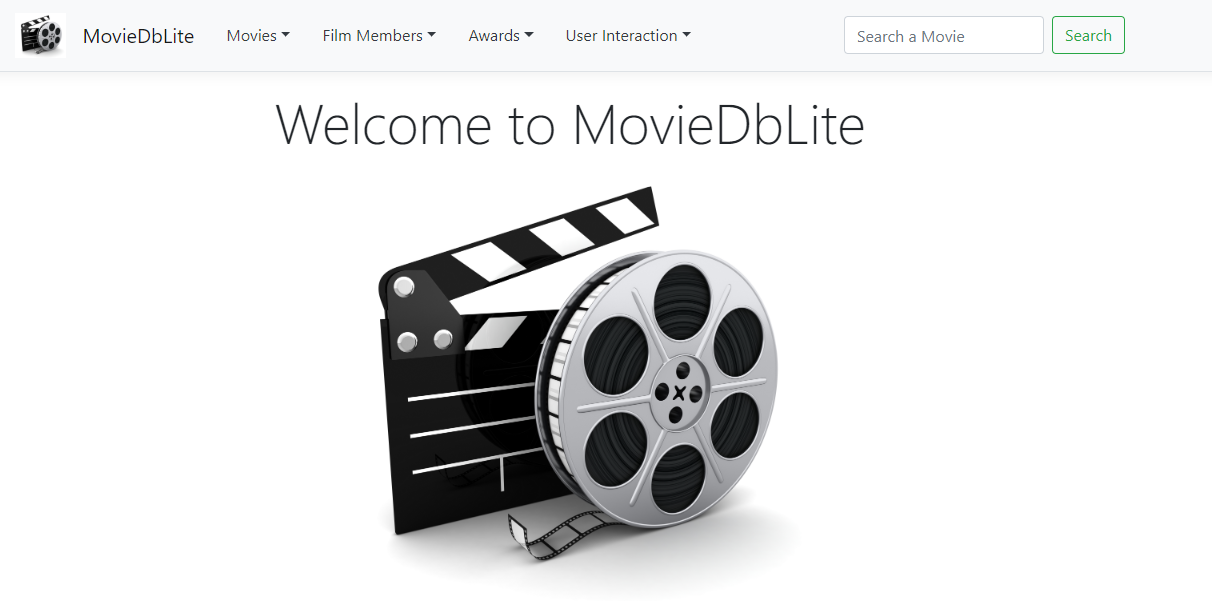
* <https://github.com/stevo9510/MovieDbLiteMVC>

The primary features of this website include:

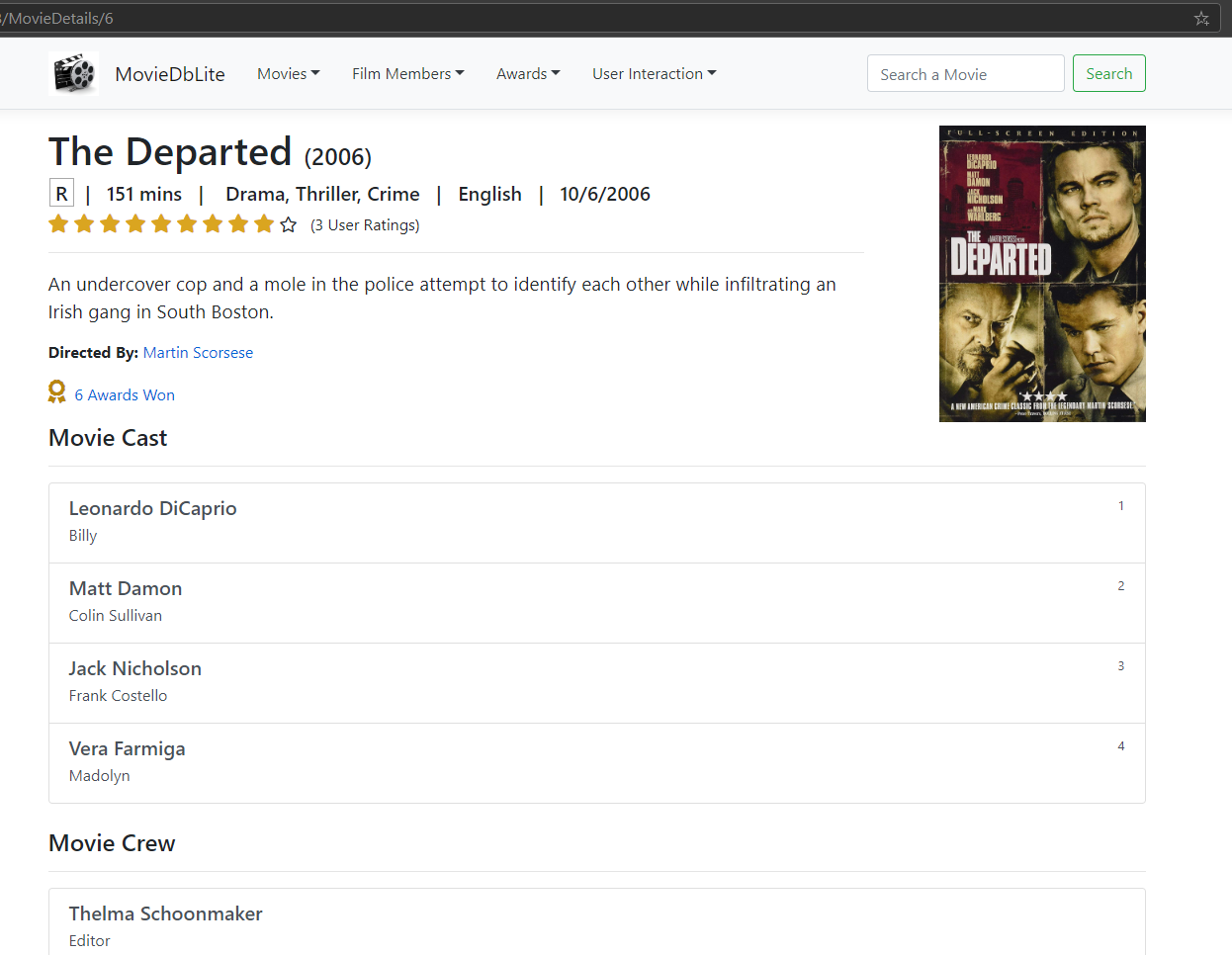
1. Searching a Movie by Title
2. Viewing Full Movie Details Page
   1. E.g. Genres, Languages, Average User Rating, Awards, Cast and Crew, Image, etc.
3. View Film Member Details Page
   1. Biography, DOB, Movie Credits, Awards, etc.
4. Admin CRUD Forms
   1. Almost all entities in database have a CRUD form generated for it to allow easily adding data by admins
5. Reports
   1. E.g. Award Winners Report is available for user to see all the award winners tracked the database

This is the home page of the website. It is very simple right now. There are links at the top to bring the user to some of the advanced menus (aforementioned admin CRUD menus) and reports.

One of the main capabilities is the ability to Search a Movie.

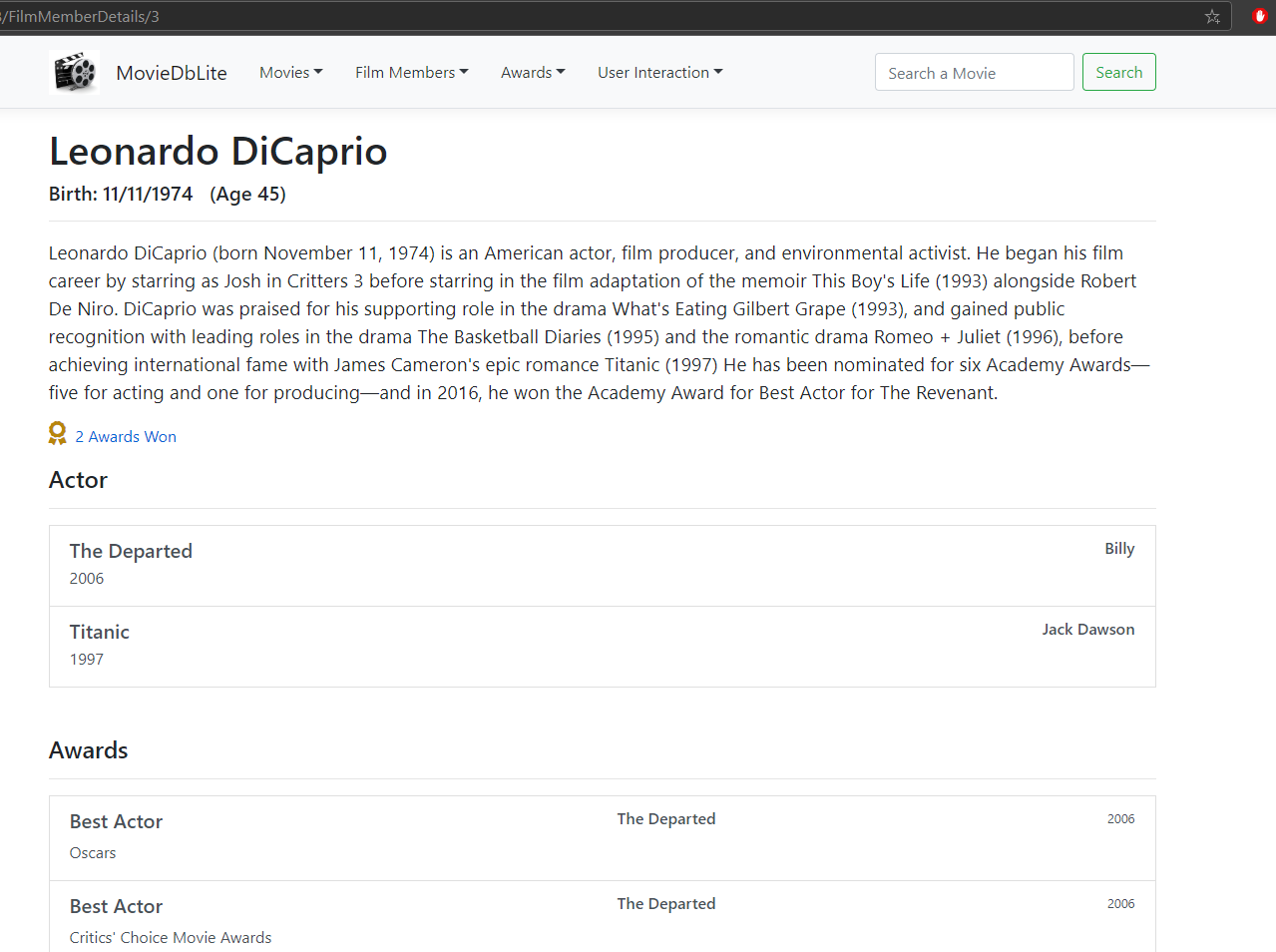


When a movie is searched, the user is brought to a details page (shown below) to display the full details of the movie. The user can interact with some of the links in the webpage. For example, they can click on a cast member to be brought to the respective film member’s detail page.

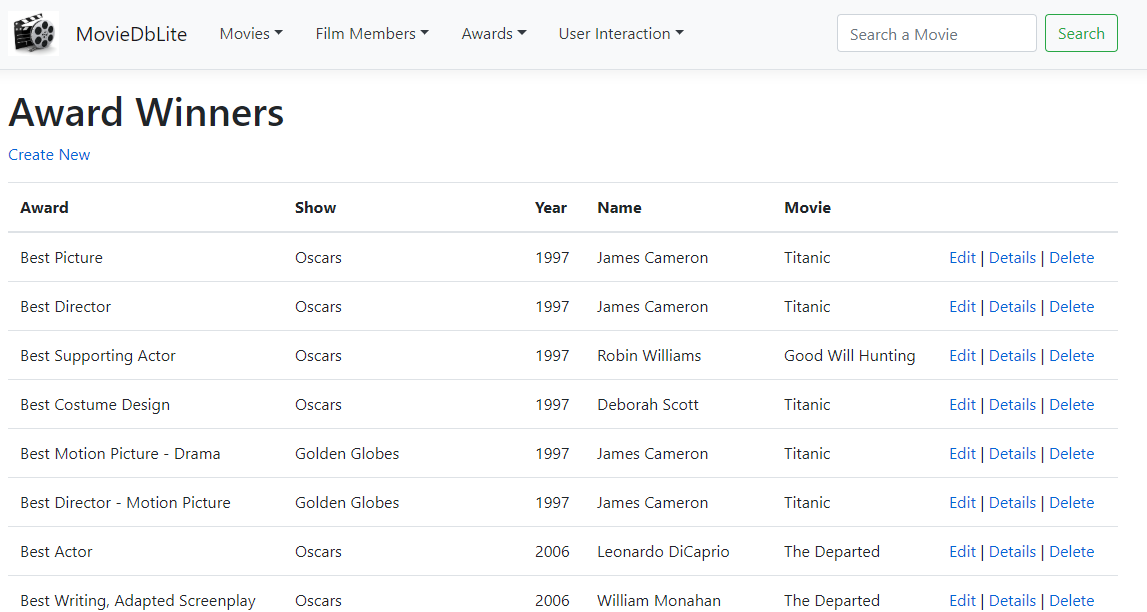


The below screenshot shows the details page of a film member. Note that URL is in the format of {webpage}/FilmMember/{FilmMemberId} so that URLs can be shared to others easily. Leonardo DiCaprio’s film member id happens to be **3**. This page shows information about the film member, as described in the summary of this section.

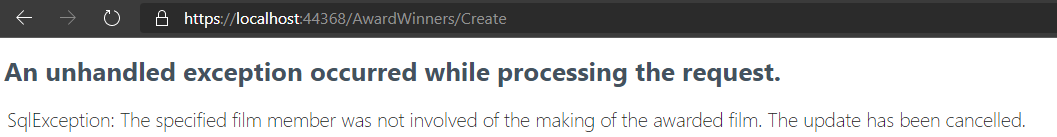
One of the future features that would be nice for the website is the ability to show a portrait/image of the film member on their page. This is something noted in the Concluding Remarks of this paper.



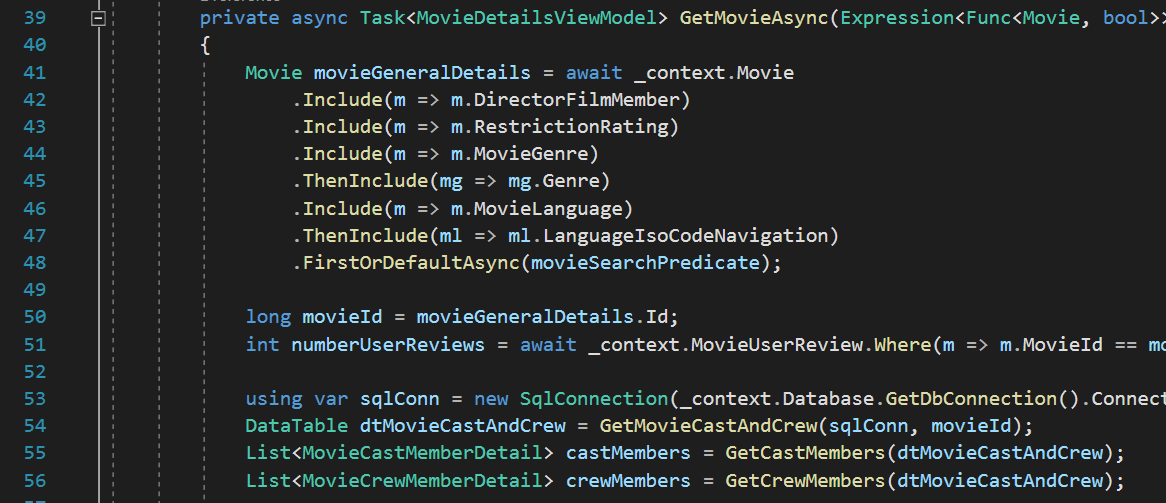
There are administration pages that allow you to view specific entities in the database, and perform Create/Edit/Delete operations against them. Most of these administrator pages were autogenerated based off tooling in Visual Studio 2019. More information about this tooling can be found in 3.6.3 Visual Studio 2019 which is within the CASE tooling section.



Error handling is in place to prevent mistakes from being make, and to enforce data integrity. For example, this screenshot shows an exception thrown by the usp\_InsertAwardWinner stored procedure mentioned in a previous section. Note that in a real Production website, we’d have a more user-friendly looking error page.



This website was primarily built using .NET Core MVC Framework, Entity Framework, SQL Server, and HTML/CSS (e.g. BootStrap)/JavaScript/C#. The chunk of code shown below is used to retrieve information to display on the movie details page. Lines 41-48 uses Entity Framework to query the database for certain movie details. Line 54 uses .NET SQL libraries to execute raw SQL statements against the database. This particular call queries a Table-Valued Function to get the cast and crew of a movie. More details are shown in the recorded video demonstration for this section.



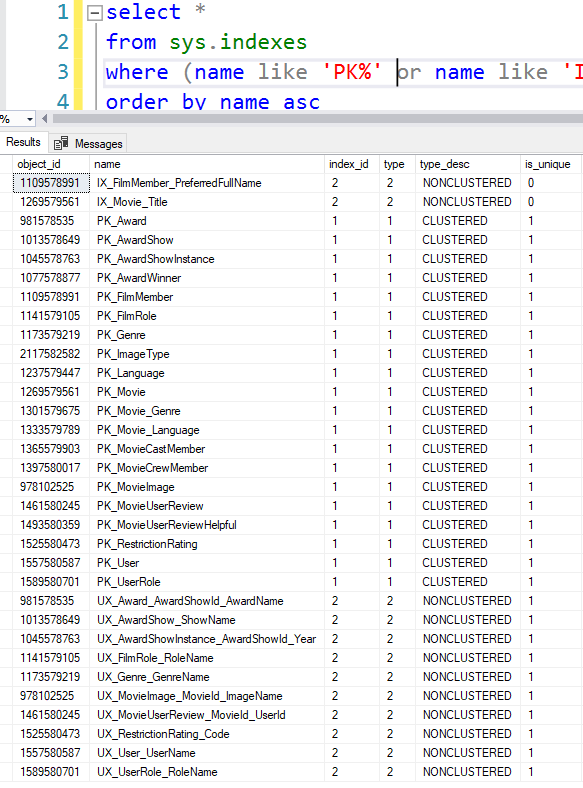
Overall, this website implements some of the basic features one may expect from a Movie database website. It was a great learning experience to explore some of the advanced SQL Server functionality, as well as leverage modern ORM tooling and website development frameworks. Please view the recorded video mentioned atop this section to see a demonstration of the website which explains more specifics about how it works and what it displays.

### Indexes

It has been mentioned in the previous sections that indexes were added to the database to help with query performance, and this section will mention a few other specific details related to indexes.

The query further below shows all the indexes that were defined across all the MovieDbLite tables (filtered using my standard naming conventions for the tables). You’ll notice that there are clustered and non-clustered indexes created. SQL Server automatically adds clustered indexes against the primary key for the table (although you can override this if you’d like). Clustered indexes against the primary key are essentially a combination of the Primary and Clustering indexes described in Module 11. In SQL Server, clustered indexes affect how the data is physically stored on disk (in general, it tries to maintain a physical ordering although fragmentation can occur over time).

Non-clustered indexes are similar to secondary indexes. These are typically applied to non-primary key columns, but can be tagged as “Unique” or “Not-Unique”, which will enforce a constraint in the former. In addition, unique-non-clustered indexes means a more effectively stored B+-tree because there are fewer collisions between the data involved.



Overall, clustering indexes were created against the primary key of every table in the database (meaning faster lookups against the PK of the table). In addition, I tried to add non-clustered indexes where they made most-sense, especially to enforce unique constraints and help with common lookups. Some non-unique indexes exist, such as searches by Movie title or Film Member name, because those will be common operations in the application.

## Queries

This section covers some of the primary queries and reports of MovieDbLite. Some of these queries leverage the Views and Table-Valued Functions described in the previous section.

Note: There may be some minor deviations between some of the results/data shown amongst

### Movie Details Report (Primary Details with Languages/Genres)

This query is used to report the primary details of the all movies in the database. Tables that may have multiple values per movie are comma delimited (e.g. genre and languages). These are pre-delimited ahead of time and inserted into temp tables, primarily for readability of the final report query. Alternatively, we could have delimited them within final report itself if we grouped by all columns and used STRING\_AGG. We also do some formatting to convert the DurationInMinutes for the movie into Hours/Minutes.

The data that is returned from this query is user-friendly (no artificial IDs) and could be used for displaying a webpage with the movie’s information on it. This could easily be achieved by filtering the query by a provided MovieId.

Note that you may notice this corresponds with details that are populated by the usp\_InsertMovieDetails stored procedure.

-- Step #1: Create Temp tables to comma delimit multiple result tables

-- such as Languages and Genres

SELECT Movie\_Language.MovieId,

-- STRING\_AGG is a built-in function within SQL Server 2017 to delimit

-- the result based on the grouped column(s)

STRING\_AGG([Language].LanguageName, ', ') as Languages

INTO #MovieLanguages\_Delimited

FROM Movie\_Language

INNER JOIN [Language] ON [Language].LanguageIsoCode = Movie\_Language.LanguageIsoCode

GROUP BY Movie\_Language.MovieId

SELECT Movie\_Genre.MovieId,

STRING\_AGG(Genre.GenreName, ', ') as Genres

INTO #MovieGenres\_Delimited

FROM Movie\_Genre

INNER JOIN Genre ON Genre.Id = Movie\_Genre.GenreId

GROUP BY Movie\_Genre.MovieId

-- Step #2: Do actual report query

SELECT Movie.Title,

Movie.[Description],

Movie.ReleaseDate as 'Release Date',

RestrictionRating.Code as 'Restriction Rating',

Director.PreferredFullName as 'Director',

CASE WHEN Movie.DurationInMinutes IS NOT NULL THEN

-- FORMATMESSAGE works like a PRINTF() statement

FORMATMESSAGE('%dh %dmin', HrDuration.Hrs, MinDuration.Mins)

END as 'Duration',

Movie.AverageUserRating as 'Average User Rating',

#MovieLanguages\_Delimited.Languages,

#MovieGenres\_Delimited.Genres

FROM Movie

LEFT JOIN RestrictionRating ON RestrictionRating.Id = Movie.RestrictionRatingId

LEFT JOIN FilmMember Director ON Director.Id = Movie.DirectorFilmMemberId

-- Join on temp tables

LEFT JOIN #MovieLanguages\_Delimited ON #MovieLanguages\_Delimited.MovieId = Movie.Id

LEFT JOIN #MovieGenres\_Delimited ON #MovieGenres\_Delimited.MovieId = Movie.Id

-- Outer applys used here to calculate hours/mins.

-- This is used to keep the SELECT portion of the query cleaner

OUTER APPLY (SELECT Movie.DurationInMinutes / 60 as Hrs) HrDuration

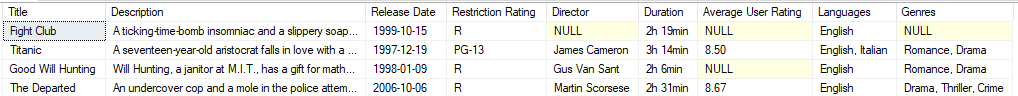
OUTER APPLY (SELECT Movie.DurationInMinutes % 60 as Mins) MinDuration

-- Cleanup temp tables

DROP TABLE #MovieLanguages\_Delimited

DROP TABLE #MovieGenres\_Delimited

Results:



### All Cast and Crew Member Names and Roles

This query returns all the cast and crew member names and role information (including character name for Actors). It utilizes the aforementioned tvf\_GetAllMovieFilmMembers table-valued function to get the foreign key ids that are used to retrieve FilmMember and FilmRole details. The results are sorted in order by Director, Actor, Sequence (for Actors), and then FilmRoleId for the remainder. CASE statement is used to prioritize the Directors and Actors first. ORDER BY operations in SQL Server default to sorting in ascending order (if not specified).

This report is a user-friendly view (e.g. no artificial IDs) of the cast and credits of a particular movie, and can be reported using this data on a webpage.

DECLARE @MovieId\_Titanic int = (SELECT id FROM Movie WHERE Title = 'Titanic' AND YEAR(ReleaseDate) = 1997)

SELECT FilmMember.PreferredFullName,

FilmRole.RoleName,

mFilmMember.CharacterName

FROM tvf\_GetAllMovieFilmMembers(@MovieId\_Titanic) mFilmMember

INNER JOIN FilmMember ON FilmMember.Id = mFilmMember.FilmMemberId

INNER JOIN FilmRole ON FilmRole.Id = mFilmMember.FilmRoleId

ORDER BY

-- List Director First

CASE WHEN mFilmMember.FilmRoleId = 3 -- @FilmRoleId\_Director

THEN 0

ELSE 1

END,

-- Prioritize Actors secondly

CASE WHEN mFilmMember.FilmRoleId = 2 -- @FilmMemberId\_Actor

THEN 0

ELSE 1

END,

-- Order Actors by Sequence

mFilmMember.Sequence,

-- Simply order by Role Id after this

mFilmMember.FilmRoleId

Results:



### Movie and Role Information for Film Member

This query returns all the movie and role information (including character name for Actors) for a given film member (Leonardo DiCaprio in this case). It utilizes the aforementioned tvf\_GetAllFilmMemberMovies table-valued function to get the foreign key ids that are used to retrieve Movie and FilmRole details.

This report is a user-friendly view (e.g. no artificial IDs) of the movies of a particular film member, and can be reported using this data on a webpage.

SELECT Movie.Title,

FilmRole.RoleName,

FilmMemberMovies.CharacterName

FROM FilmMember

CROSS APPLY dbo.tvf\_GetAllFilmMemberMovies(FilmMember.Id) FilmMemberMovies

INNER JOIN Movie ON Movie.Id = FilmMemberMovies.MovieId

INNER JOIN FilmRole ON FilmRole.Id = FilmMemberMovies.FilmRoleId

WHERE FilmMember.PreferredFullName = 'Leonardo DiCaprio'

Results:



### Retrieve User Movie Reviews/Ratings

This query returns all the user movie reviews/ratings of a particular movie (The Departed in this case). An aggregation is performed to do a sum of all the upvotes/downvotes by users of each review. Normally, you cannot SUM bits in SQL server (you can only call SUM on a number type, such as an integer). However, we do a little trick here to convert the bit to an integer by multiplying the bit \* 1. Also, we use the bitwise negate operator (tilde ~) to sum the downvotes.

This report is a user-friendly display of all the reviews for a given movie.

DECLARE @MovieTitle varchar(150) = 'The Departed'

SELECT

[User].UserName,

MovieUserReview.DatePosted,

MovieUserReview.Rating,

MovieUserReview.Review,

SUM(MovieUserReviewHelpful.IsHelpful \* 1) as Upvotes,

SUM(~MovieUserReviewHelpful.IsHelpful \* 1) as Downvotes

FROM MovieUserReview

INNER JOIN Movie ON Movie.Id = MovieUserReview.MovieId

INNER JOIN [User] ON [User].Id = MovieUserReview.UserId

LEFT JOIN MovieUserReviewHelpful ON MovieUserReviewHelpful.MovieUserReviewId = MovieUserReview.Id

WHERE Movie.Title = @MovieTitle

GROUP BY

MovieUserReview.MovieId,

[User].UserName,

MovieUserReview.DatePosted,

MovieUserReview.Rating,

MovieUserReview.Review

ORDER BY

MovieUserReview.MovieId,

MovieUserReview.DatePosted

Results:



### Retrieve Helpful Vote Details of a Particular Review

This report provides more specifics on which users upvoted/downvoted a certain review. Using our previous report example, we examine the users that upvoted/downvoted Superman33’s review of the Departed. We first start by gathering the UserId/MovieId of the particular user review we want to look up, and then use the ReviewId in the final query to gather the details. We sort by helpful (i.e. upvoted) reviews floating to the top.

This query would be useful to drill into more details about a particular movie review.

DECLARE @UserName varchar(25) = 'SuperMan33'

DECLARE @UserId int = (SELECT id FROM [User] WHERE UserName = @UserName)

DECLARE @MovieId bigint = (SELECT id FROM Movie WHERE Title = 'The Departed')

DECLARE @MovieUserReviewId bigint =

(SELECT id FROM MovieUserReview WHERE MovieId = @MovieId AND UserId = @UserId)

SELECT [User].UserName,

MovieUserReviewHelpful.IsHelpful

FROM MovieUserReviewHelpful

INNER JOIN [User] ON [User].Id = MovieUserReviewHelpful.UserId

WHERE MovieUserReviewHelpful.MovieUserReviewId = @MovieUserReviewId

ORDER BY IsHelpful DESC

Results:



### Retrieve Film Members in Multiple Movies/Roles

This report retrieves the film members that worked on multiple movies OR had multiple roles in a movie. In other words, a film member that had multiple movie roles during their career. It uses a GROUP BY and HAVING filter to achieve these results.

SELECT

FilmMember.FirstName,

FilmMember.MiddleName,

FilmMember.LastName,

MultiFilmMemberRoles.TotalNumberOfFilmRoles,

FilmMember.PreferredFullName,

FilmMember.Gender,

FilmMember.DateOfBirth,

FilmMember.Biography

FROM FilmMember

INNER JOIN (

SELECT FilmMember.Id as FilmMemberId,

COUNT(\*) as TotalNumberOfFilmRoles

FROM FilmMember

CROSS APPLY dbo.tvf\_GetAllFilmMemberMovies(FilmMember.Id) FilmMemberMovies

GROUP BY FilmMember.Id

HAVING COUNT(\*) > 1

) MultiFilmMemberRoles ON MultiFilmMemberRoles.FilmMemberId = FilmMember.Id

Results:



### Award Winners for a Particular Year

This report retrieves the award show winners of a particular year (1997 in this case). The query is relatively simple in this example because the details are abstracted away within the [vw\_AwardWinnerInfo](#_View) view that was mentioned in Advanced Features section.

SELECT

AwardName,

ShowName,

[Year],

Title,

PreferredFullName,

DateHosted

FROM vw\_AwardWinnerInfo

WHERE [Year] = '1997'

Results:



### Awards for a Particular Movie

This report retrieves the award show winners of a particular movie (The Departed in this case). The query is relatively simple in this example because the details are abstracted away within the [vw\_AwardWinnerInfo](#_View) view that was mentioned in Advanced Features section. This query can be leveraged when on the webpage of a particular movie, and the user wants to view the awards that the movie achieved.

DECLARE @MovieId bigint = (SELECT id FROM Movie WHERE Title = 'The Departed')

SELECT

AwardName,

ShowName,

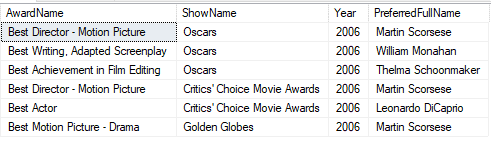
[Year],

PreferredFullName

FROM vw\_AwardWinnerInfo

WHERE MovieId = @MovieId

Results:



### Awards for a Particular Film Member

This report retrieves the awards that a particular film member has won over their career. The query is relatively simple in this example because the details are abstracted away within the [vw\_AwardWinnerInfo](#_View) view that was mentioned in Advanced Features section. This query can be leveraged when on the webpage of a particular film member, and the user wants to view the awards that film member has won in the past.

DECLARE @FilmMemberId bigint = (SELECT id FROM FilmMember WHERE PreferredFullName = 'James Cameron')

SELECT

AwardName,

ShowName,

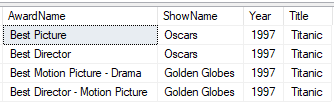
[Year],

Title

FROM vw\_AwardWinnerInfo

WHERE FilmMemberId = @FilmMemberId

Results:



### Director Candidates

This query is used to populate the option/drop-down selector for Director candidates. It orders by determining the most qualified/probable candidates based on whether or not they are still alive (deceased candidates are toward the bottom of the list), and the number of films they’ve directed (the more films they’ve directed, the higher they are on the list). The Id is returned as the value member of the drop-down, and the PreferredFullName acts as the Display property in the drop down.

Note that in a future state we would not show all film members in an option/drop-down (since that would be quite large). Some sort of search / incremental search feature would be implemented instead. In addition, there may be more criteria/variables involved in determining good director candidates than the two used in this query.

SELECT FilmMember.Id,

FilmMember.PreferredFullName

FROM FilmMember

LEFT JOIN (

SELECT DirectorFilmMemberId,

COUNT(\*) as NumberTimesDirected

FROM Movie

GROUP BY DirectorFilmMemberId

) DirectorStats ON DirectorStats.DirectorFilmMemberId = FilmMember.Id

ORDER BY

-- Sort by people that are alive first

CASE WHEN DateOfDeath IS NULL

THEN 0

ELSE 1

END ASC,

-- Get persons that have directed most first

DirectorStats.NumberTimesDirected DESC

Results:



### “You may know this film member from…” information

This query is one that I personally wanted to include, and a feature that I would like to see from a movie website. This report would be used when a user is viewing details about a certain movie. It provides the user with information about which cast/crew members “they may know” from other films they have watched/rated. This can be helpful in figuring out where you recognize a particular actor/actress from, or whether or not you may like a movie based on the director’s other films you may have seen.

One thing neat about this query is that it utilizes both table-valued functions to retrieve the results. The first one to get all the film members in the movie, and the second one to get all the movies for each film member returned by the first table-valued function. Additionally, this query is unique in that it ties together user related information (i.e. user reviews) with the cast and crew of movies.

DECLARE @MovieId bigint = (SELECT Id FROM Movie WHERE Title = 'Titanic')

DECLARE @UserId int = (SELECT Id FROM [User] WHERE UserName = 'StevenAnderson')

-- Gets information about "You may know this actor from another movie"

-- based off user's reviews of other movies.

SELECT

FilmMember.PreferredFullName as 'Film Member Name',

Movie.Title as 'Other Movie',

MovieUserReview.Rating,

MovieUserReview.Review

FROM tvf\_GetAllMovieFilmMembers(@MovieId) movieFilmMembers

-- Get all the movies for each film member

CROSS APPLY tvf\_GetAllFilmMemberMovies(movieFilmMembers.FilmMemberId) filmMemberOtherMovies

INNER JOIN MovieUserReview ON MovieUserReview.MovieId = filmMemberOtherMovies.MovieId

AND MovieUserReview.UserId = @UserId

INNER JOIN FilmMember ON FilmMember.Id = movieFilmMembers.FilmMemberId

INNER JOIN Movie ON Movie.Id = filmMemberOtherMovies.MovieId

WHERE filmMemberOtherMovies.MovieId <> @MovieId -- Ensure not the same movie

Results:

Layman’s terms: If the user is viewing the “Titanic” movie page, it could show them that they may know “Leonardo DiCaprio” from “The Departed” based on a movie the user has reviewed.



# CRUD Matrix

This section covers the CRUD Matrix for the MovieDbLite website application. The Entity Types are listed across the top as column headers, and the major/primary Functions of the front-end are listed as Row Headers across the left.

The column and row titles can be hovered over to reveal a tooltip of what the actual entity type / function is.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | [E1](#_List_of_Entity) | [E2](#_List_of_Entity) | [E3](#_List_of_Entity) | [E4](#_List_of_Entity) | [E5](#_List_of_Entity) | [E6](#_List_of_Entity) | [E7](#_List_of_Entity) | [E8](#_List_of_Entity) | [E](#_List_of_Entity)9 | [E](#_List_of_Entity)10 | [E1](#_List_of_Entity)1 | [E1](#_List_of_Entity)2 | [E1](#_List_of_Entity)3 | [E1](#_List_of_Entity)4 | [E1](#_List_of_Entity)5 | [E1](#_List_of_Entity)6 | [E1](#_List_of_Entity)7 | [E1](#_List_of_Entity)8 | [E1](#_List_of_Entity)9 | [E20](#_List_of_Entity) |
| [F1](#_List_of_Functions) |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | D | D |  | C R U D |  |
| [F2](#_List_of_Functions) |  |  |  |  | C R U D |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| [F3](#_List_of_Functions) |  |  |  |  | R |  | R |  | R | C R U D | C R D | C R D |  |  |  |  |  | R |  |  |
| [F4](#_List_of_Functions) |  |  |  |  |  |  |  | R |  |  |  |  |  |  | C R U D |  |  |  |  |  |
| [F5](#_List_of_Functions) | C R U D | C R U D | C R U D |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| [F6](#_List_of_Functions) | R | R | R | C R U D | R |  |  |  |  | R |  |  |  |  |  |  |  |  |  |  |
| [F7](#_List_of_Functions) |  |  |  |  | R |  |  |  |  | R |  |  | C R U D |  |  |  |  |  |  |  |
| [F8](#_List_of_Functions) |  |  |  |  | R | R |  |  |  | R |  |  |  | C R U D |  |  |  |  |  |  |
| [F9](#_List_of_Functions) |  |  |  |  |  |  |  |  |  | U R |  |  |  |  |  | C R U D |  |  |  |  |
| [F10](#_List_of_Functions) |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | R | C R U D | R | R |  |
| [F11](#_List_of_Functions) |  |  |  |  |  |  |  |  |  | R |  |  |  |  |  |  |  |  |  |  |
| [F12](#_List_of_Functions) |  |  |  |  | R |  | R |  | R | R | R | R |  |  |  |  |  | R |  |  |
| [F13](#_List_of_Functions) |  |  |  |  | R | R |  |  |  | R |  |  | R | R |  |  |  |  |  |  |
| [F14](#_List_of_Functions) | R | R | R | R | R |  |  |  |  | R |  |  |  |  |  |  |  |  |  |  |
| [F15](#_List_of_Functions) |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | R | R |

## List of Entity Types

The following are the corresponding Entity Types that are associated with the CRUD Matrix.

|  |  |
| --- | --- |
| E1 | Award |
| E2 | AwardShow |
| E3 | AwardShowInstance |
| E4 | AwardWinner |
| E5 | FilmMember |
| E6 | FilmRole |
| E7 | Genre |
| E8 | ImageType |
| E9 | Language |
| E10 | Movie |
| E11 | Movie\_Genre |
| E12 | Movie\_Language |
| E13 | MovieCastMember |
| E14 | MovieCrewMember |
| E15 | MovieImage |
| E16 | MovieUserReview |
| E17 | MovieUserReviewHelpful |
| E18 | RestrictionRating |
| E19 | User |
| E20 | UserRole |

## List of Functions

The following are the Functions that are associated with the CRUD Matrix. These are the major/primary functions of the system.

* **F1: Manage User Account** - Create/Update/Delete/Read User Account Information (User)
  + Note: Deleting of user account will delete all personal data and operations tied to user (e.g. User Reviews)
* **F2: Manage Film Member** - Create/Update/Delete/Read FilmMember Information (Admin)
* **F3: Manage Movie Basic Details1** – Create/Update/Delete/Read Movie Related Basic Information (Admin)
  + Note: Since Movie\_Language/Movie\_Genre is many-to-many (with composite primary key), there’s not a concept of Update. It is always an Insert/Delete
* **F4: Manage Movie Images** – Create/Update/Delete/Read MovieImages (Admin)
  + Note: The ImageType is looked up during creation to verify valid ImageType is uploaded (e.g. mp4 file would not be a valid image type)
* **F5**: **Manage Award Show Info** –Create/Update/Delete/Read Award/Show/Instance related information (Admin)
  + Note: For data integrity reasons, deletes of an entity cannot occur if they’re tied to an AwardWinner instance. This must be performed by an application developer.
* **F6: Manage Award Show Winners1** – Create/Update/Delete/Read AwardWinners (Admin)
* **F7: Manage Movie Cast1** – Create/Update/Delete/Read MovieCastMembers (Admin)
* **F8: Manage Movie Crew1** – Create/Update/Delete/Read MovieCrewMembers (Admin)
* **F9: Write User Reviews1** – Create/Update/Delete/Read MovieUserReviews (User/Admin)
  + Note: Movie table is updated as a result of trigger to maintain Movie.AverageUserRating
* **F10: Mark Review Helpful1** – Create/Update/Delete/Read MovieUserReviewHelpful (User/Admin)
* **F11: Search Movie Title** – Read Movie entity (User)
* **F12: Report Basic Movie Details** – Read Basic Movie Information (User)
* **F13: Search Movie Cast and Crew** – Read Movie Cast and Crew Information (User)
* **F14**: **Report Award Show Winners** – Read Award Winner Information (User)
* **F15: User Login** – Read User/UserRole Information to get authorization token (User)

**1 Denotes that entity types marked solely as ‘R’ (Reads) in the matrix for that row are necessary to populate drop-down/lists for easily selecting entries in the website form. For example, when adding Movie basic information, a drop-downs will be available to select the corresponding Restriction Rating, Genre, Language, etc. that are populated via respective database entity.**

# Concluding Remarks

The MovieDbLite database was a very informative, yet enjoyable project to work on over the duration of the semester. It taught me a lot about the complexities and nuances of database design. With the advancements of RDBMS tooling, it may seem that creating a new database is an easy or trivial process. It may be true that someone could put together a basic design in a short amount of time, however, there are many considerations and decisions that have to be made when utilizing the database for a production-level application.

For example, these are some of the challenges and questions that commonly arise during design (and in fact, arose during the duration of this project):

1. Dealing with performance of the application/database
   1. i.e. Properly designing for performance, creating the appropriate indexes, etc.
2. Proper normalization of the schema
   1. And conversely, when is it acceptable or okay to denormalize the schema for performance?
3. Data integrity
   1. Usage of constraints vs. stored procedures vs. triggers. What is best for what scenarios?
   2. Should a field be nullable or not? How strict do we want to enforce this data type in the database?
4. Superclass/subclass relationships
   1. What is the best approach to handling them? E.g. out of the approaches discussed in Module 8.
5. When to create artificial primary keys vs. using table’s natural key?
6. What are good naming conventions for the tables and columns?
7. What is the best way to map the database to code?
   1. E.g. ORM vs. usage of Stored Procedures vs. other methodologies?

With all those challenges in mind, I found the best approach was to take them one-by-one and look at each as practical as possible. Consider how the end-user will be impacted by every decision, as well as the DBA/development team. I think overall I was able to adequately find solutions to each of the challenges above, however, it required a significant amount of thought and weighing the benefits. These are what I consider some of the primary strengths of the design:

1. Data integrity is robust and supported by many advanced features (triggers, constraints, user-defined data types, rules, unique-constraints, etc.)
2. Clustered indexes exist on every table, and non-clustered indexes exist on almost every table. Non-clustered indexes are added to columns that will be often searched (e.g. Movie Title, Film Member preferred name, etc.)
3. Common reporting scenarios are simplified by usage of table-valued functions and views.
4. Naming conventions are standardized and consistent to allow others to understand more easily
5. Contains a front-end website and utilizes ORM technologies to help with some of the basic CRUD operations.

The biggest lesson learned and takeaway from the project for me was that upfront design is very important, and arguably the most important part of the project. There were some things in my upfront design that did not quite pan out as I started the implementation phase, and that required a fair amount of rework. The database rework has a trickle effect in that it also impacts the code (of the website) that references it. ORM tooling did not seem to make the refactoring any easier. Therefore, I discovered that database rework can be expensive and sometimes tedious if you are trying to preserve data. Fortunately, this was not a production-level database, and so some shortcuts could be made to make appropriate changes to the database schema.

If time permitted, these are the following changes or new fields I would consider:

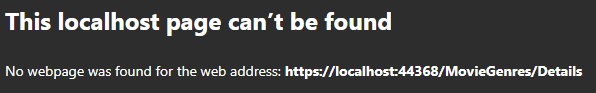
1. AwardShowInstance could potentially have a “Location” column where the award show event takes place
2. Track production companies (e.g. Warner Brothers, Fox Studios, etc.) and associate with movies
3. Support for thumbnail/portrait images to associate with FilmMembers
4. Allow assigning a “primary” image for a movie (to display a thumbnail next to the Movie itself when it shows up in a listing)
5. Support for Movie Trailers/Videos
6. Support for Critic Reviews
   1. Potentially import this from outside data sources or API
7. Track a history of award show names
   1. E.g. The Oscars used to be more formerly called the Academy Awards
8. In addition to AwardWinners, store the film members that are nominated for awards.
9. Better integration of other Movie Database data (e.g. TheMovieDb.org via APIs)
10. More advanced reporting features on the website
    1. E.g. being able to filter and display all award show winners for a particular award show instance.
11. UI improvements for the website
12. Better search for movies (e.g. search as you type)
13. Customized simplified/artificial IDs for Movies and FilmMembers, for easier referencing (as opposed to the very long bigint variable).
    1. E.g. IMDb has this structure for movie ids, which is simpler but still can support many combinations of IDs: ‘tt8228288’
14. Actual user authentication and authorization for certain views in the application
    1. The application support users creating accounts right now, but does not enable them the ability login or authorize them to retrieve only allowed resources.

Overall, I am happy with how the project turned out and it was a great learning experience along the way. I plan to continue to make improvements to the front-end side of the project even after the course completes.

## Appendices

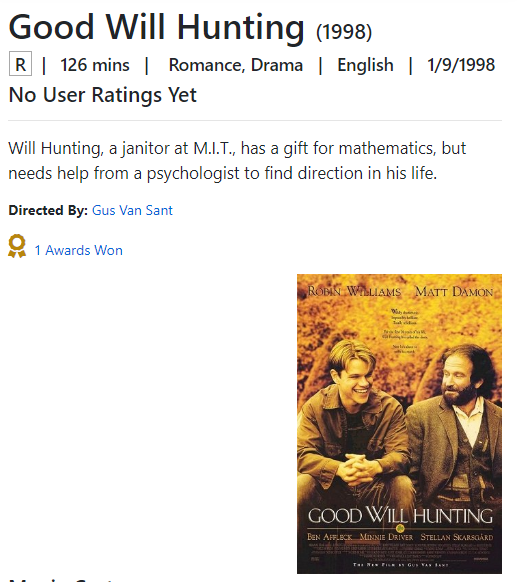
From a database perspective, there are not any known bugs. All discovered bugs have been resolved throughout the semester. There are functional enhancements and improvements to make for the database (e.g. storing additional data) that are mentioned in the previous section.

In regards to the website, there are some bugs that need to be fixed. For example, there are pages within the scaffolded code that have dead-links (that result in 404 errors). This only affects some of the administrator forms.



In addition, there are not graceful error messages when searching for a movie that is not found currently. An unhandled exception page is generated by ASP.NET and displayed to the user.

Lastly, there are some minor UI quirks with some of the pages in terms of responsiveness based on screensize. For example, if the page is shrunk (to a mobile-like view), the UI leaves empty space that could be filled better (e.g. below image shows that the poster on a mobile device should shift left to be left-aligned and remove some of the spacing). There are also some minor UI quirks when certain fields are null for a movie.



## Appendix A - DDL, INSERT, SELECT Statements

* See “MovieDbLite\_DDL.sql” file submitted with this project for all the DDL (CREATE/ALTER/TRIGGER/STORED PROC/etc.) statements.
* See “MovieDbLite\_DML.sql” file submitted with this project for all the INSERT and SELECT all statements (from each table) for the database.
  + Note: I’ve omitted data from the MovieImage table because that table stores actual binary data for the images (which is quite large for a script).

## Appendix B - Data Dictionary Index

This is an Index for the Implementation Data Dictionary.

Each column is listed in Alphabetical order. The Table name is shown in parenthesis beside the ColumnName, to help distinguish it in the case of duplicates.

In Microsoft Word, you can use **Ctrl-G** to quickly jump to the corresponding Page number of the index.

A

ActorFilmMemberId (MovieCastMember) 53

AverageUserRating (Movie) 53

AwardId (AwardWinner) 52

AwardName (Award) 51

AwardShowId (Award) 51

AwardShowId (AwardShowInstance) 52

AwardShowInstanceId (AwardWinner) 52

B

Biography (FilmMember) 52

C

CharacterName (MovieCastMember) 53

Code (RestrictionRating) 54

D

DateHosted (AwardShowInstance) 52

DateOfBirth (FilmMember) 52

DateOfDeath (FilmMember) 52

DatePosted (MovieUserReview) 54

DateUploaded (MovieImage) 54

Description (Award) 51

Description (AwardShow) 51

Description (FilmRole) 52

Description (Genre) 52

Description (Movie) 53

Description (MovieImage) 54

Description (UserRole) 55

DirectorFilmMemberId (Movie) 53

DurationInMinutes (Movie) 53

E

EmailAddress (User) 55

F

FileContents (MovieImage) 54

FilmMemberId (AwardWinner) 52

FilmMemberId (MovieCrewMember) 54

FilmRoleId (MovieCrewMember) 54

FirstName (FilmMember) 52

G

Gender (FilmMember) 52

GenreId (Movie\_Genre) 53

GenreName (Genre) 52

H

HashedPassword (User) 55

I

Id (Award) 51

Id (AwardShow) 51

Id (AwardShowInstance) 52

Id (FilmMember) 52

Id (FilmRole) 52

Id (Genre) 52

Id (ImageType) 53

Id (Movie) 53

Id (MovieImage) 54

Id (MovieUserReview) 54

Id (RestrictionRating) 54

Id (User) 55

Id (UserRole) 55

ImageExtension (ImageType) 53

ImageName (MovieImage) 54

ImageTypeId (MovieImage) 54

IsActive (RestrictionRating) 54

IsHelpful (MovieUserReviewHelpful) 54

L

LanguageIsoCode (Language) 53

LanguageIsoCode (Movie\_Language) 53

LanguageName (Language) 53

LastName (FilmMember) 52

LongDescription (RestrictionRating) 54

M

MiddleName (FilmMember) 52

MovieId (AwardWinner) 52

MovieId (Movie\_Genre) 53

MovieId (Movie\_Language) 53

MovieId (MovieCastMember) 53

MovieId (MovieCrewMember) 54

MovieId (MovieImage) 54

MovieId (MovieUserReview) 54

MovieUserReviewId (MovieUserReviewHelpful) 54

N

Name (ImageType) 53

P

PreferredFullName (FilmMember) 52

Prefix (FilmMember) 52

R

Rating (MovieUserReview) 54

ReleaseDate (Movie) 53

RestrictionRatingId (Movie) 53

Review (MovieUserReview) 54

RoleName (FilmRole) 52

RoleName (UserRole) 55

S

Sequence (MovieCastMember) 53

ShortDescription (RestrictionRating) 54

ShowName (AwardShow) 51

Suffix (FilmMember) 52

T

Title (Movie) 53

U

UserId (MovieUserReview) 54

UserId (MovieUserReviewHelpful) 54

UserName (User) 55

UserRoleId (User) 55

Y

Year (AwardShowInstance) 52

## References

* <https://www.microsoft.com/en-us/sql-server/sql-server-2017-editions>
  + Info about SQL Server additions
* <https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-web-api?view=aspnetcore-3.1&tabs=visual-studio>
  + Help with .NET Core Web API
* <http://venkateswarlu.net/SQLServer/UserDefined_Data_Types_in_sql_server.aspx>
  + How to create user-defined data types in SQL Server
* <https://stackoverflow.com/a/590469>
  + Surrogate vs. Natural Keys
* <https://docs.microsoft.com/en-us/sql/relational-databases/backup-restore/recovery-models-sql-server?view=sql-server-ver15#RMov>
  + Recovery Models in SQL Server
* <https://docs.microsoft.com/en-us/sql/relational-databases/security/choose-an-authentication-mode?view=sql-server-ver15>
  + Authentication Modes in SQL Server
* <https://cmatskas.com/a-simple-net-password-hashing-implementation-using-bcrypt/>
  + BCrypt hashing in .NET
* <https://www.themoviedb.org/>
  + API used to help populate data into MovieDbLite
* <https://www.imdb.com/>
  + Used as a reference to help manually input data for certain Film Members or Movies