# Введение

Перед разработчиками приложения всегда встаёт вопрос, как создать систему объектов, способную смоделировать конкретную сложную систему. По мнению М. Фаулера [1], основное преимущество объектной парадигмы состоит в облегчении понимания запутанной логики. Разработка хорошей ***модели предметной области*** (Domain Model [2]) для сложной проблемы реальной задачи является фундаментом всего проекта. Однако, и это еще не все. Модель предметной области должна быть отображена в базе данных (БД). Причем, именно в этой последовательности: сначала продумывается архитектура приложения, а потом строится БД, сохраняющая все необходимые данные. Объектно-ориентированная модель предметной области часто похожа на соответствующую схему БД, однако, в отличие от схемы данных, помимо данных, она содержит и методы работы с ними.

NHibernate это популярный, готовый и открытый ресурс объектно-ориентированного отображения ORM (Object Relational Mapping), основанный на Java Hibernate. Технологии ORM, такие как, LINQ to SQL, Entity Framework и NHibernate, позволяют отобразить реляционную модель базы данных, состоящую из таблиц, столбцов и ключей в объектную модель приложения, состоящую из классов и свойств [1].

**Важная терминология NHibernate**

POCO, Plain Old CLR Object.Альтернативный термин для класса в C#, который:

* Не наследуются от классов, определенных в любом фреймворке (framework).
* Не реализуют интерфейсы, определенные в любом фреймворке.
* Не используют никаких атрибутов, определенных в любом фреймворке.

Entity – сущность. Класс, имеющий идентификатор, который определяет экземпляр на диаграмме сущностей, обслуживаемой NHibernate. Этот класс является классом POCO.

Persistent Class – сохраняемый класс. Любой класс, который можно сохранить в одной или нескольких таблицах.

Attribute/property – атрибут/ свойство. В случае использования в классе POCO имеется в виду геттеры и сеттеры свойства или поля класса. В случае отсылки к XML будет иметься в виду обычные атрибуты XML.

# Требования, предъявляемые к именованию ключевых атрибутов в базе данных (Code convention)

В реляционных БД для идентификации записей таблиц используются ключи, в частности первичные. Для идентификации объекта в памяти ключи не нужны, идентификация осуществляется по значению его полей или по адресу объекта в памяти. Для поддержки соответствия между объектом приложения и записью в таблице БД применяется типовое решение (паттерн) ***поле идентификации*** (Identity Field) [1].

Отображение внешних ключей (Foreign Key Mapping) отображает ассоциации между объектами на ссылки внешнего ключа между таблицами БД. Типовое решение ***отображение внешних ключей*** отображает объектную ссылку на внешний ключ БД [1].

К именованию ключевых атрибутов в БД предъявляются следующие требования:

1. Все сущности в БД должны иметь одинаковые имена для ключевых атрибутов и одинаковые типы данных (GUID или целое число).
2. Все внешние ключевые атрибуты должны иметь имена, составленные по одному и тому же правилу. Например, ID\_ИмяГлавнойТаблицы.

Такие жёсткие требования связаны с тем, что NHibernate имеет своё соглашение по именованию главных и вторичных ключей (Code convention) для отображения (mapping). Для каждой сущности в БД обязательно имеется главный ключ, который выбирается из семантики предметной области. Если такого ключа нет, то используется суррогатный ключ, который необходим только для уникальной идентификации данных в БД. NHibernate настоятельно рекомендует использование именно суррогатного ключа. Причина такого требования заключается в том, что сущность может иметь составной первичный ключ, работа с которым сложна и неудобна.

## Выбор ID генератора

NHibernate предлагает много опций для генерации уникального значения ключевого поля Id. **Non-insert POID** (persistent object identifier) generators (устанавливается значение свойства поля Id до того, как оно будет сохранено (метод INSERT) в базу данных). Эти генераторы позволяют NHibernate присвоить сущности сохраняемый объект без записи данных в БД, разрешая NHibernate отложить запись до тех пор, пока все транзакции не утверждены, уменьшая, таким образом, обращения к БД. К таким генераторам относятся, например:

**hilo** – генерирует целое число, используя Hi/Lo алгоритм, в котором входной диапазон целых чисел зарезервирован и используется, когда требуется. Когда все числа в диапазоне будут использованы, резервируется следующий диапазон. Т.к. резервирование сущности управляется таблицей БД, данный генератор безопасен для использования для любых серверных приложений, которые используют одну БД;

**guid** – генератор создаёт GUID (Global Unique Identifier глобальный уникальный идентификатор), вызываемый командой System.GIUD.NewGuid(). Все генераторы, основанные на данном генераторе безопасны для распределённых БД [2];

**foreign** – просто копирует ключ для связи один-к-одному.

Применение **Post-insert POID** генераторов не рекомендуется. Применяются в том случае, когда данные уже существуют в БД, а другие приложения влияют на изменение этих данных.

## Отображение связи один – ко – многим.

## Lazy loading collections

Для улучшения работы приложения NHibernate поддерживает lazy loading [1] [3] (ленивую или отложенную загрузку) из БД до тех пор, пока данные не понадобятся приложению. Рассмотрим шаги, которые проделывает NHibernate при получении данных из БД:

1. NHibernate получает данные для всех полей для данного ID, кроме списка, содержащего связь один – ко – многим.
2. NHibernate создаёт объект, соответствующий таблице на стороне связи «один».
3. NHibernate заполняет данными все свойства объекта, кроме свойства, отвечающего за список.
4. NHibernate создаёт специальный lazy loading объект, который создаёт IList<КлассСущностиНаСторонеМногие> специальную отдельную реализацию интерфейса IList и устанавливает свойство типа список.
5. NHibernate возвращает в приложение созданный объект.

Как только приложение использует объект, lazy loading объект инициализируется. Он загружает данные из ассоциированной таблицы по значению внешнего ключа.

Мы можем отключить «ленивую загрузку» коллекции, однако это не рекомендуется!

## Отображение Модели (Mapping)

Наша задача – разработать модель, которая является ядром приложения. Наше приложение должно работать с БД. Мы понимаем, что существует огромная разница в работе с моделью и БД. Модель – объектно-ориентированная, база данных – реляционная. Модель работает с отдельными объектами, в то время как БД – с набором данных. Это является концептуальным несоответствием между двумя подходами часто называемым «несоответствием импеданса» ***impedance mismatch***. Отображение (Mapping) является «мостом» между этим несоответствием. Отображение определяет, как данные, которые «живут» в модели внутри объектов и их свойств находят соответствие с таблицами БД и их полями, описанными в реляционной схеме БД.

Когда мы начинаем новый проект с нуля, сначала необходимо определить модель, а БД спроектировать таким образом, чтобы она «следовала» проекту. Не нужно определять сначала схему БД и пытаться поставить ею сверху над моделью. В случае, когда БД следует требованиям модели, отображение проходит легче. Мы можем использовать соглашения для большинства частей отображения, следовательно, мы однозначно определим множество деталей. В экстренном случае мы можем даже позволить фреймворку автоматически создать отображение за нас.

М. Фаулер даёт следующую формулировку архитектурному шаблону проектирования ***преобразователь данных*** (Data Mapper): слой преобразователей (Mapper), который осуществляет передачу данных между объектами и БД, сохраняя последние независимыми друг от друга и от самого преобразователя [1].

### Типы отображения

* Отображение на основе XML.
* Отображение на основе атрибутов.
* Текучее отображение (fluent mapping).
* Отображение на основе соглашений, иногда называемое автоотображением (auto – mapping).

#### Отображение на основе XML (XML- based mapping)

Каждый тип модели развёртывается с помощью документа XML в таблицу схемы БД. Структура этого документа определена в nhibernate-mapping.xsd файле, который является частью загружаемого NHibernate.

Этот вариант отображения модели в схему БД является наиболее гибким. Исторически, NHibernate предлагал только этот вариант отображения. Гибкость поставляется по цене. Для того чтобы определить документ для каждой и любой сущности и значимого объекта, требуется много работы, однако XML не отличается удобочитаемостью.

Наибольшее преимущество перед другими вариантами отображения заключается в том, что, если мы не можем отобразить нашу модель, используя XML, то мы не сможем отобразить нашу модель никакими другими вариантами. Минусами данного варианта является низкая удобочитаемость документа отображения по сравнению с другими вариантами.

#### Отображение на основе атрибутов

Вместо того чтобы представлять отдельный документ для отображения сущности в указанную таблицу БД, таблица может также использовать атрибуты для декорирования сущности и ею членов. Эти атрибуты вместе с их метаданными определяют детали отображения. Имеется несколько плюсов и минусов относительно этого варианта отображения модели.

Преимущества:

* Как было сказано выше, мы не предоставляем никаких других документов для отображения.
* Определение сущности и отображения находятся в одном месте, т.о. легче понять, как происходить отображение сущности.
* Отображение происходит с определёнными в C# типами данных, т.е. нет необходимости использовать другой язык (такой как XML) для определения отображения.

Недостатки:

* При добавлении атрибутов к нашей сущности, мы соединяем сущности точно в соответствии с тем, как она будет сохранена. Наша сущность больше не является POCO(Plain old CLR object) [4].
* Внедрение атрибутов вносит хаос в нашу сущность. Следовать бизнес-логике, которую содержит сущность, станет тяжелее.
* Атрибуты ограничены в своём использовании. Большинство преимуществ отображения не могут быть достигнуты при использовании данного варианта.

Имеется два главных фреймворка (frameworks), предоставляющих данный вариант отображения: NHibernate contribution project и Castle Active Records.

С появлением текучего отображения (fluent mapping), данный вид отображения потерял свою привлекательность.

#### Текучее отображение (fluent mapping)

Для данного варианта отображение сущностей предметной области в таблицы БД происходит путём сохранения в соответствующие типы, используя текучий API [5] [6], который делает код простым и читаемым. Отображение определено в отдельном классе, который не засоряет сущность. Фреймворк (framework), который обеспечивает нам возможность текуче определить отображение, использует деревья выражений (expression trees) и статическую рефлексию, чтобы обеспечить нам безопасность типов, где нам не нужно использовать «магические» строки. Фреймворк Fluent NHibernate обеспечивает нас возможностью текучего определения нашего отображения.

Преимущества текучего отображения:

* Код, используемый для отображения простой и читабельный.
* Отображение хранится отдельно от сущности, что не засоряет сущность.
* Отображение сохраняется по типам (type-safe). Мы не используем «магические строки».
* Можно выбирать или устанавливать отдельные свойства для каждого элемента, т.к. текучий API полностью поддерживается IntelliSense.

Как обычно, не обойтись и без недостатков:

* Нам необходимо определять отображаемый класс для каждой сущности и значимых объектов.
* Некоторые «экзотические» сценария отображения не поддерживаются текучим отображением, однако, ситуация улучшается с каждым новым релизом.
* На настоящий момент, определение текучего отображения вносит небольшую задержку в производительности при запуске приложения и инициализации библиотеки, так как текучее отображение должно быть транслировано в XML отображение, которое будет распознано и использовано. В этом дополнительном шаге нет необходимости, когда вы используете XML отображение.
* Когда имя таблицы БД отличается от имени сущности или поля таблицы отличаются от свойств сущности, приходится использовать «магические строки». К сожалению, другой альтернативы нет.

##### **Деревья выражений (expression trees)**

Когда Microsoft внедрил LINQ, он сначала создал необходимую инфраструктуру. LINQ в значительной мере зависит от лямбда-выражений и деревьев выражений. Что же такое деревья выражений? Когда компилятор разбирает код, он конвертирует его в структуру похожую на дерево, которая может быть лучше проанализирована и оптимизирована. Обычно, после создания и оптимизации дерева выражений, компилятор транслирует его в исполняемый код. Однако иногда было бы неплохо, если разработчик имел бы возможность поработать с этим деревом также как с исполняемым кодом. Почему? Да потому, что дерево выражений содержит всю информацию об оригинальном коде и его структуре, которую мы можем использовать. Анализируя данные, хранящиеся в дереве и в структуре дерева, мы можем получить больше информации, чем при применении рефлексии. Этот анализ деревьев выражений также называется статической рефлексией, в отличие от стандартной рефлексии, использующей тип Type и называемой динамической рефлексией.

Давайте посмотри, как мы можем это использовать. Нам хотелось бы использовать отображение без использования «магических» строк. Следующий код использует строку, определяя какое свойство необходимо отобразить:

Map (“Some Property”);

Вместо него, мы можем использовать следующий сохраняющий тип код:

Map(x => x.SomeProperty);

Этот код использует лямбда-выражение для определения, какое свойство мы хотим отобразить. Заметьте, теперь в нем нет «магических» строк!

Функция Map выполняется с помощью Fluent NHibernate API и ею сигнатура выглядит так:

public Propery Part<T> Map

(Expression<Func<T, object>> expression)

{…}

Тип параметра Expression<Func<T, object>> представляет собой дерево выражений. Метод Map получает это дерево выражений, которое он может анализировать и понимать какое свойство мы имеем в виду.

# Пример приложения

## Создание схемы базы данных

Работу с библиотекой NHibernate рассмотрим на примере разработки приложения «Библиотека». Для выбранной предметной области создадим схему БД. Все сущности, поля и ключи будут иметь одинаковые названия и будут построены на одинаковых доменах, но в двух разных СУБД: Access и MS SQL Server для демонстрации преимущества технологии ОRМ. При именовании уникальных идентификаторов принято соглашение: имя первичного ключа - ID, тип - INT . Имя вторичного ключа – ID\_ИмяТаблицы.

Схема БД «Библиотека» в СУБД MS SQL Server представлена на следующем рисунке (см. Рисунок 1). БД имеет следующие таблицы:

* Книга (Book);
* Автор (Author);
* КнигаАвтор (AuthorBook);
* Издательство (Publishing);
* Город (City);
* Серия (Series);
* Комната (Room);
* Полка (Shelf).



Рисунок 1

## Разработка приложения

Для демонстрации работы с библиотекой NHibernate, создадим консольное приложение. Наше приложение будет содержать несколько папок, в которых разместим классы, разделённые по функциям.

Для работы нам понадобятся следующие пакеты: NHibernate, FluentNHibernate, Lesi.Collections. Для установки данных пакетов, выберите опцию «Управление пакетами NuGet» и скачайте последние версии.

### Создание основной сущности

Мы хотим реализовать основной класс для всех сущностей. Этот класс реализует логику работы поля ID, которое является общим для всех сущностей. Для сущностных классов создайте папку Entities.

1. Добавьте новый класс в папку Entities. Это класс со спецификатором доступа public и обобщением T. Код будет следующим:

using System;

namespace Library.Entities

{public abstract class Entity<T> where T : Entity<T>

{}}

1. Добавьте авто-свойство ID типа Int32. Измените спецификатор доступа для установки значения (set сеттер) на protected. Это будет уникальный идентификатор для нашей сущности. Для новой сущности ID будет автоматически равен нулю.

public virtual Int32 ID {get; protected set ;}

1. Переопределите метод Equals данного класса. Код должен работать в трёх случаях:
2. Другая сущность (с которой мы сравниваем текущую), если не является сущностью того же типа, не является эквивалентной, поэтому мы просто возвращаем значение false.
3. Если обе сущности являются новыми объектами (текущая и та, с которой мы сравниваем) и не были сохранены в БД, то мы считаем эти сущности эквивалентными, если они ссылаются на одну и ту же сущность в области памяти, или в терминологии .NET, если их ссылки эквивалентны.
4. Если две сущности, которые подлежат сравнению, одинакового типа, не являются новыми, тогда мы просто сравниваем их ID. Отрывок кода приведён ниже:

public override Boolean Equals(Object obj)

{

var other = obj as T;

if (other == null) return false;

var thisIsNew = ID == 0;

var otherIsNew = other.ID == 0;

if (thisIsNew && otherIsNew)

return ReferenceEquals(this, other);

return ID.Equals(other.ID);

}

Сущности, которые никогда не были сохранены в БД, называются **transient**. Сущности, которые были сохранены, называются **persistent**.

1. Поскольку мы переопределили метод Equals, нам также необходимо реализовать метод GetHashCode. В этом методе мы просто вернём хэш-код ID. Но существуют специфический случай, который необходимо рассмотреть отдельно. Этот случай работает на том принципе, что сущность не должна изменять свой хэш-код, пока остаётся в памяти. Если сущность является новой сущностью с неопределённым ID (в нашем случае, равным нулю), а какой-то метод (например, HashSet<T> или Dictionary<K,T>, к которому сущность будет добавлена), запросил ею хэш-код, позднее этой сущности будет дан ею хэш-код (с высокой вероятностью, потому что значение будет сохранено в БД), но в нашем случае вернётся хэш-код неопределённого ID для новой сущности. Принимая во внимание только что рассмотренный случай, наш код выглядит так:

private Int32? oldHashCode;

public override Int32 GetHashCode()

{

// если у нас есть хэш-код, мы его никогда не изменяем

if (oldHashCode.HasValue)

return oldHashCode.Value;

//когда этот экземпляр сущности новый, мы используем

//основной хэш -код и запоминаем его, т.о. экземпляр //сущности НИКОГДА не сможет изменить этот хэш-код

var thisIsNew = ID == 0;

if (thisIsNew)

{

oldHashCode = base.GetHashCode();

return oldHashCode.Value;

}

return ID.GetHashCode();

}

1. И, в конечном счёте, мы переопределяем два оператора == и !=, таким образом мы можем сравнивать две сущности, не прибегая к методу Equals. Внутри оба метода используют Equals:

public static Boolean operator ==(Entity<T> lhs, Entity<T> rhs)

{

return Equals(lhs, rhs);

}

public static Boolean operator !=(Entity<T> lhs, Entity<T> rhs)

{

return !Equals(lhs, rhs);

}

### Создание сущностных классов

Для каждой сущности необходимо написать соответствующий ей класс (для сущностей, созданных только для того, чтобы разорвать связь «многие-ко-многим» класс создавать не нужно).

В папке Entities создайте классы по следующим правилам:

1. Каждое свойство имеет открытый виртуальный (public virtual) спецификатор доступа. Это необходимо, чтобы библиотека NHibernate использовала «ленивую загрузку».
2. Все типы данных отображаются соответственно типам данных, выбранных для нашей БД.
3. Поля, соответствующие вторичным ключам, имеют тип класса внешней таблицы (сторона, соответствующая «многие» для связи «один–ко –многим»).
4. Поля на стороне «один» имеют тип IList<КлассСущностиНаСторонеМногие>.

Рассмотрим код для класса *Серия* (Series):

using System;

namespace Library.Entities

{

public class Series : Entity<Series>

{

public virtual String Title {get; set ;}

public virtual String Note {get; set ;}

}

}

Наш класс наследуется от базового класса Entity. В нем реализуются следующие авто-свойства: *Название серии* (Title) строкового типа и *Описание* ([Note]) также строкового типа.

По тому же принципу реализуем класс *Издательство* (Publishing) с полями *Название офиса* (PublishingOffice) строкового типа, *Описание* ([Note]строкового типа, *Город* (City) целое число – вторичный ключ).

### Определение связей между сущностями

#### Владеет или содержит

Значимые объекты (value object) никогда не существуют в одиночестве. Они становятся значимыми в соединении с сущностью. Сущность может владеть или содержать нуль или много значимых объектов. В нашем примере (*Библиотека*) имеется отношение *Автор* (Author), с полями *Имя* (FirstName), *Фамилия* (LastName) и *Отчество* (MiddleName). Создадим значимый объект Name с указанными полями:

public class Name

{

public String FirstName {get; protected set ;}

public String LastName {get; protected set ;}

Public String MiddleName {get; protected set ;}

public Name()

{

FirstName = "defaultFirstName";

LastName = "defaultLastName";

MiddleName = "defaultMiddleName";

}

public Name(String lastName, String firstName,

String middleName)

{

if (String.IsNullOrWhiteSpace(lastName))

throw new ArgumentException

("Фамилия должна быть определена");

if (String.IsNullOrWhiteSpace(firstName))

throw new ArgumentException

("Имя должно быть определено");

LastName = lastName;

FirstName = firstName;

MiddleName = middleName;

}

}

Переопределим метод GetHashCode. Если значение поля *Отчество* (MiddleName) не определено (null), то значение хэш-кода мы берём равным нулю. Код должен выглядеть следующим образом:

public override Int32 GetHashCode()

{

unchecked

{

var result = LastName.GetHashCode();

result = result\*397)^(MiddleName?.GetHashCode()??0);

result = (result\*397)^FirstName.GetHashCode();

return result;

}

}

Для того чтобы закончить описание класса Name, переопределим метод Equals. В нем нам необходимо сделать три шага:

1. Мы проверяем, является ли передаваемый параметр null. Если да, тогда эта сущность и та, с которой мы сравниваем, не равны, и мы возвращаем false.
2. Далее мы определяем, являются ли данная и сравниваемая сущности одной и той же. Если да, то возвращаем true.
3. И наконец, мы сравниваем каждое свойство индивидуально. Если значения всех свойств совпадает, мы возвращаем true, нет - false. Код представлен ниже:

public override Boolean Equals(Object other)

{

return Equals(other as Name);

}

public Boolean Equals(Name other)

{

if (other == null)return false;

if (ReferenceEquals(this, other))return true;

return Equals(other.LastName, LastName) &&

Equals(other.FirstName, FirstName) &&

Equals(other.MiddleName, MiddleName);

}

Для удобного вывода переопределим метод ToString:

public override String ToString()

{

return $"{LastName} {FirstName} {MiddleName}";

}

Далее создадим класс Author, наследуемый от Entity. Создадим открытое свойство AuthorName типа Name:

public class Author : Entity<Author>

{

public Name AuthorName {get; private set ;}

public void ChangeAuthorName(String lastName, String firstName, String middleName)

{

AuthorName = new Name(lastName, firstName, middleName);

}

}

В нашем случае значимый объект Name содержится в сущности Author. Однако значимый объект не знает, кто является его владельцем.

#### Связь один-ко-многим

Давайте посмотри на две сущности: *Комната* (Room) и *Полка* (Shelf). Как они связаны друг с другом?

Каждая полка принадлежит точно одной комнате. В дальнейшем мы сможем определить свойство Room типа Room в классе Shelf. Это свойство является ссылкой от полки к комнате. Оно может применяться для навигации от комнаты к ассоциированной полке. Эта связь между полкой и комнатой может быть представлена следующим кодом:

public class Shelf : Entity < Shelf >

{

public virtual Room Room {get; set ;}

}

Каждая комната может содержать много полок. Следовательно, мы можем определить свойство Shelves в классе Room, которое является коллекцией или набором полок. Код может выглядеть следующим образом:

public class Room: Entity < Room >

{

public virtual IList <Shelf> Shelves {get; set ;}

public Room()

{

Shelves = new List <Shelf>();

}

}

#### Связь многие-ко-многим

Рассмотрим связь многие-ко-многим на примере: связь между автором и книгой. Каждый автор может написать несколько книг, также, книга может быть написана в соавторстве. В нашем случае код будет следующим:

public class Book: Entity < Book >

{

public virtual IList <Author> Authors {get; set ;}

public Book ()

{

Authors = new List< Author >();

}

}

public class Author : Entity< Author >

{

public virtual IList < Book > Books {get; set ;}

public Author()

{

Books = new List<Book>();

}

}

Т.к. в БД может существовать автор, который не написал ни одной книги, но книга, у которой нет автора, существовать не может, то главной таблицей будет таблица *Автор* (Author). Процедура добавления новой книги в класс *Автор* описана ниже:

public void AddBook(Book book)

{

book.Authors.Add(this);

Books.Add(book);

}

### Отображение (Mapping)

Для отображения мы будем использовать в нашем приложении Fluent NHibernate. Давайте рассмотрим, как мы отобразим сущность *Автор* (Author). Для этого добавим в наше приложение папку Mapping, в которую будет добавлять классы, соответствующие нашим сущностям. Все отображаемые классы будем называть СущностьMap и наследовать от класса ClassMap<T>, где T будет класс, который мы хотим отобразить. Базовый класс определён в ClassMap<T>. Он содержит весь необходимый код, который делает отображение лёгкой задачей.

Наш код будет выглядеть так:

public class AuthorMap: ClassMap <Author> {…}

#### Отображение простых свойств

В конструкторе класса мы определим детали отображения. Простые свойстваотображаются, используя метод Map основного класса, и посылаются в качестве параметра лямбда-выражения, определяющего свойство класса, которое мы хотим отобразить. Простые свойства имеют один из основных типов .NET, такие как int, float, double, DateTime и т.д. Для свойства *Дата Рождения* (BirthDate) мы можем добавить следующий код:

Map(x => x.BirthDate);

В представленном коде мы отобразили свойство BirthDate в поле BirthDate таблицы *Автор* (Author) БД. Это основывается на том, что Fluent NHibernate использует соглашения, если мы не указали специфические черты точно.

#### Отображение ID сущности

Для отображения ID сущности, мы используем метод Id основного класса и снова посылаем в качестве параметра лямбда-выражение, которое определяет свойство Id сущности. Следующий код отобразит свойство Id как Первичный Ключ к полю ID таблицы *Автор* (Author) БД. Таким образом, встроенный Id генератор библиотеки NHibernate будет генерировать новые значения первичного ключа.

Id(x => x.ID);

Если мы хотим использовать GUID в качестве первичного ключа, то лучше использовать генератор GuidComb, так как он оптимизирован для использования БД.

Мы указываем только те детали, которые отличаются от определённых по умолчанию значений основных соглашений. Любые детали, в которых нет необходимости, только загрязняют наш код и отвлекают от необходимых элементов.

#### Отображение специфических типов данных

Если необходимо отобразить специфический тип данных, например, перечисляемый тип enum, определённый так:

public enum Cities

{

Moscow,

SaintPetersburg,

…

}

Тогда, для отображения, мы должны добавить функцию CustomType, которая будет выглядеть следующим образом:

Map(x => x.City).CustomType<Cities>();

В этом случае, так как тип enum является по умолчанию целым типом данных, тип поля в БД будет INT для СУБД SQL Server.

Другой пример, который часто используется для отображения свойства типаbool (Boolean) в тип CHAR(1) поля таблицы БД, которое содержит ‘Y’ для значения true и ‘N’ для значения false. NHibernate имеет специальное определение для отображения, называемое YesNo. Мы можем отобразить, например, свойство *Электронная Версия* (IsDigitalCopy) для сущности *Книга* (Book):

Map(x => x.IsDigitalCopy).CustomType(“YesNo”);

#### Ссылки

Для отображения свойств, которые ссылаются на другую сущность, например, Комната (Room) для сущности *Полка* (Shelf), мы используем следующий код:

References(x => x.Room);

В этом отрывке представлена связь многие-к-одному, используемая в предметной области. **Если мы не указали точно, то имя Внешнего ключа к таблице *Комната* (Room) будет Room\_Id. Соглашение по умолчанию называет внешний ключ как комбинацию имени таблицы и присоединённого через нижнее подчёркивание ID.** Мы можем это изменить, указав точно имя внешнего ключа:

References(x => x.Room).ForeignKey("ID\_Room");

#### Коллекции

Здесь мы расскажем о другой стороне связи – один-ко-многие. Одна родительская сущность может иметь коллекцию дочерних сущностей. Продолжим разбираться с отношением *Полка* (Shelf) – *Комната* (Room). Отображение свойства *Полки* (Shelves) сущности *Комната* будет выглядеть следующим образом:

HasMany(x => x.Shelves).Inverse();

Рассмотрим наш пример. Книги стоят на полке – на одной полке много книг (одна книга не может стоять на нескольких полках). Так как книга не может ничего знать о том, в каком порядке и как осуществляется хранение книг на полке, то ответственность за хранение (сохранение и удаление) ложится на полку. Таким образом, можно заключить, что **сторона «много»** (сущность Книга) **является поражённой в правах и делегирует обязанности стороне «один»** (сущность Полка). Inverse() необходимо поставить к той стороне, которая содержит коллекцию (то есть стороне Полка).

Мы используем атрибут Inverse() для спецификации «собственника» ассоциации. Ассоциация может иметь только одного собственника, т.е. один конец связи установлен в Inverse, а другой в «non - Inverse». Если указанной выше ассоциации *Комната-Полка* мы не пометим один конец с помощью атрибута Inverse, то NHibernate выполнит дополнительную операцию UPDATE. Сначала вставит сущность, содержащуюся в коллекции (экземпляр полка), если необходимо, вставит сущность, которая владеет коллекцией (комната), и после проведёт обновление сущности коллекции, таким образом, установится значение внешнего ключа и его ассоциации. Важно, чтобы в этом случае значение внешнего ключа в БД могло иметь значение NULL.

Если мы помечаем атрибутом Inverse один конец, то NHibernate сначала сохранит сущность, которая является собственником коллекции (*Комната*), затем сохранит сущности из коллекции (*Полки*), избегая дополнительного обновления.

Ещё одной важной деталью является поведение относительно того как NHibernate должен вести себя при вставке, удалении и обновлении главных сущностей. Если мы укажем, что NHibernate каскадирует операции, то при добавлении новой полки в коллекцию *Полки* объекта *Комната*, сохраним *Комнату*, то информация о новой полке сохранится автоматически. Другой важный сценарий происходит при удалении существующей полки из коллекции *Полки* объекта *Комната*. Если указано каскадное удаление, то NHibernate автоматически удалит указанную запись о полке из таблицы БД, в противном случае, нам необходимо это будет делать явно.

#### Отображение связи многие-ко-многим

Рассмотрим отображение связи многие-ко-многим сущностей *Автор* (Author) и *Книга* (Book). Сущность Автор будет иметь коллекцию дочерних объектов Книга, также сущность Книга будет иметь коллекцию дочерних объектов Автор. Для спецификации промежуточной таблицы, связывающей в БД таблицы Автор и Книга, указывается ею имя. Код, определяющий данное отображение, представлен ниже:

HasManyToMany(x => x.Books).Table("AuthorBook");

Для связи многие-ко-многим мы должны определить источник ответственности, т.е. мы должны выделить из пары сущностей, находящихся в данном отношении, ту, которая будет отвечать за исполнение базовых операций (UPDATE, DELETE). На стороне той сущности, которая **НЕ ЯВЛЯЕТСЯ** ответственной, мы ставим Inverse(). Рассмотрим наш пример. Для двух сущностей Автор и Книга, сущность Автор является ответственной, соответственно на стороне Книга необходимо поставить Inverse().

public class BookMap: ClassMap<Book>

{

public BookMap()

{

Id(x => x.ID);

…

HasManyToMany(x => x.Authors)

.Cascade.All()

.**Inverse()**

.Table("AuthorBook ");

}

public class AuthorMap : ClassMap<Author>

{

public AuthorMap()

{

Id(x => x.ID);

…

HasManyToMany(x => x.Books)

.Table("AuthorBook");

}

}

#### Отображение значимых объектов (value object)

В нашем приложении имеется значимый объект ФИО (AuthorName) сущности Автор (Author). Определим класс NameMap, унаследованный от класса ComponentMap<Name>:

public class NameMap : ComponentMap<Name>{}

Затем добавим детали отображения в конструктор по умолчанию нашего класса:

public NameMap()

{

Map(x => x.FirstName).Not.Nullable().Length(30);

Map(x => x.MiddleName).Length(30);

Map(x => x.LastName).Not.Nullable().Length(30);

}

В представленном коде мы определили Фамилию (LastName), Имя (FirstName) и Отчество (MiddleName), с максимальным количеством символов, равным 30. Фамилия и Имя не могут быть пустыми (NULL).

Теперь, когда мы определили значимый объект, мы можем определить отображение свойства ФИО (AuthorName) сущности Автор (Author). Добавим следующую строку кода в конструкторе по умолчанию класса отображения сущности Автор (Author):

Component(x => x.AuthorName);

### Использование соглашений отображения (Mapping conventions)

Fluent NHibernate использует множество соглашений для отображения нашей предметной области в схему БД. Все эти соглашения имеют значение в повседневном использовании. Иногда нам может понадобиться добавить собственное соглашение или изменить существующее. Соглашения добавляются в систему во время инициализации, т.е. перед отображением.

#### Соглашения для ID (ID Conventions)

***Предположим***, что для первичного ключа был выбран тип GUID. Тогда в процессе определения отображения нашей «Библиотеки» нам было бы необходимо отображать для каждой сущности свойство ID. Мы определили, что хотим использовать алгоритм GuidComb() для генерации значения Id. И каждый раз мы копировали эту строку для всех сущностей. Если мы не сделаем этого, будет использован генератор, используемый по умолчанию.

Давайте переопределим конструктор по умолчанию для применения нашего соглашения. Этого можно добиться применением соответствующего интерфейса, определённого Fluent NHibernate. Каждый тип соглашения имеет свой собственный интерфейс. Интерфейс для Id соглашения имеет интерфейс. Имплементация для нашего собственного соглашения может выглядеть так:

public class MyIdConvention: IIdConvention

{

public void Apply(IIdentityInstance instance)

{

instance.GeneratedBy.GuidComb();

}

}

Теперь, когда соглашение добавлено, мы можем просто отображать Id всех наших сущностей, используя такой код:

Id(x => x.ID);

#### Соглашения для вторичного ключа (Foreign Key Conventions)

По умолчанию NHibernate требует, чтобы именование вторичного ключа было Имя\_таблицы\_Id. Если это не так, то следует переписать класс. В данном примере выбрана следующая нотация:

ID\_ИмяТаблицы. Тогда наш класс будет выглядеть следующим образом:

public class MyForeignKeyConvention : ForeignKeyConvention

{

protected override String GetKeyName(Member property, Type type)

{

var refName = property == null ? type.Name : property.Name;

return String.Format("ID\_{0}", refName);

}

}

#### Соглашение для свойств

В нашей БД почти в каждой таблице есть поле с названием *Заметки* (Note). При отображении из СУБД Access с полем такого названия возникает ошибка. Нам необходимо определить это свойство, указав, что его длина может быть максимум 100 символов, а название отображать в квадратных скобках, что представлено в следующем коде:

public class MyNoteConvention : IPropertyConvention

{

public void Apply(IPropertyInstance instance)

{

if (instance.Name != "Note") return;

instance.Column($"[{instance.Name}]");

instance.Length(100);

instance.Nullable();

}

}

## Инициализация конфигурации

В NHibernate есть две основных абстракции: ISessionFactory и ISession. Фабрика сессии создаёт сессию, и подразумевается, что сессия будет использоваться в приложении для единственной задачи – это может быть единичная транзакция или составные успешные транзакции в быстрой последовательности. Вам следует использовать, а потом быстро ликвидировать NHibernate сессии. С другой стороны, подразумевается, что фабрика сессий будет храниться на протяжении всего жизненного цикла приложения, поэтому она может использоваться для создания всех сессий.

Интерфейс ISession – это абстракция, но реализация, которую предоставляет NHibernate, требует некоторых разъяснений. Следующий листинг демонстрирует, как создать фабрику сессий, которая будет использоваться на протяжении всего жизненного цикла приложения.

### Создание фабрики сессий для СУБД MS SQL Server

public static ISessionFactory GetSessionFactory

(String connectionString)

{

var configuration = Fluently.Configure() .Database(**MsSqlConfiguration.MsSql2008** .ConnectionString(connectionString))

.Mappings(m => .FluentMappings

.AddFromAssembly(typeof(Program).Assembly)

//при использовании соглашения по первичному ключу //.Conventions.Add<MyIdConvention>()

//при использовании специального соглашения .Conventions.Add<MyNoteConvention>()

//при использовании соглашения по внешнему ключу .Conventions.Add<MyForeignKeyConvention>())

.BuildConfiguration();

return configuration.BuildSessionFactory();

}

}

}

Создание фабрики сессий – трудозатратно. Она выполняет довольно много инициализации и валидации для того, чтобы убедиться, что она может быстро выполнять доступ к данным с помощью объекта сессии. Объект конфигурации читает файл hibernate.cfg.xml (который представляет собой внепроцессный вызов) и затем создаёт фабрику сессий, используя эту конфигурацию. При создании фабрики сессий объект конфигурации будет применять все свойства, найденные в конфигурационном файле. Если в комплект были включены вложенные XML преобразования, то объект конфигурации извлечёт все эти файлы преобразований из DLLs (который является ещё одним внепроцессным вызовом). Каждый файл преобразований будет проанализирован с помощью XML DOM. Независимо от того, используете ли вы преобразования кода или XML преобразования, NHibernate будет использовать рефлексию для всех типов для того, чтобы убедиться в том, что каждое свойство, объявленное в преобразовании, существует в указанных типах. Если отложенная загрузка разрешена (по умолчанию), то NHibernate также проверит, что все открытые свойства и методы отмечены с помощью virtual [4].

NHibernate сессия, с другой стороны, является малозатратной. Мы будем создавать и разрушать множество таких объектов. В приложении, имеющем внутреннее состояние, мы будем использовать сессию для единичной транзакции или пользовательской операции. Код создания сессии похож на следующий:

ISession session = SessionFactory.OpenSession();

### Создание фабрики сессий для СУБД Access

public ISessionFactory GetSessionFactory

(String fullPath)

{

return Fluently.Configure() .Database(**JetDriverConfiguration**

.Standard.ConnectionString

(c => c.DatabaseFile(fullPath))

.ShowSql())

.Mappings(m => m

.FluentMappings

.AddFromAssemblyOf<MainWindow>())

.BuildSessionFactory();

}

### Создание строки подключения к БД для СУБД MS SQL Server

Для работы с БД через наше приложение необходимо создать строку подключения. Параметры подключения зависят от СУБД. В строке подключения необходимо указать имя и расположение БД, логин и пароль, если необходимо, указать проверку подлинности (проверка подлинности Windows (Windows Authentication) или SQL Server(SQL Authentication)).

Для проверки подлинности Windows используйте Trusted\_Connection=True; или Integrated Security=SSPI; или Integrated Security=true. В другом случае такие параметры указывать не нужно.

В первом приближении укажем параметры подключения, используя «магические» строки:

const String serverName = @"...\sqlexpress";

const String databaseName = "Library";

var connString = "Server={0};Database={1};Trusted\_Connection=Yes;"

.FormatWith(serverName, databaseName);

var sessionFactory = GetSessionFactory (connString);

Имя БД и путь к ней можно добавить в файл конфигурации. Для этого откройте файл App.config своего проекта и добавьте в него параметры (после закрывающего тэга </startup>):

<appSettings>

<add key="databaseName" value = "Library"/>

<add key="databaseLocation" value =

"...\sqlexpress "/>

Затем в проект добавьте ссылки (Рисунок 2) на System.Configuration и System.Configuration.Install.

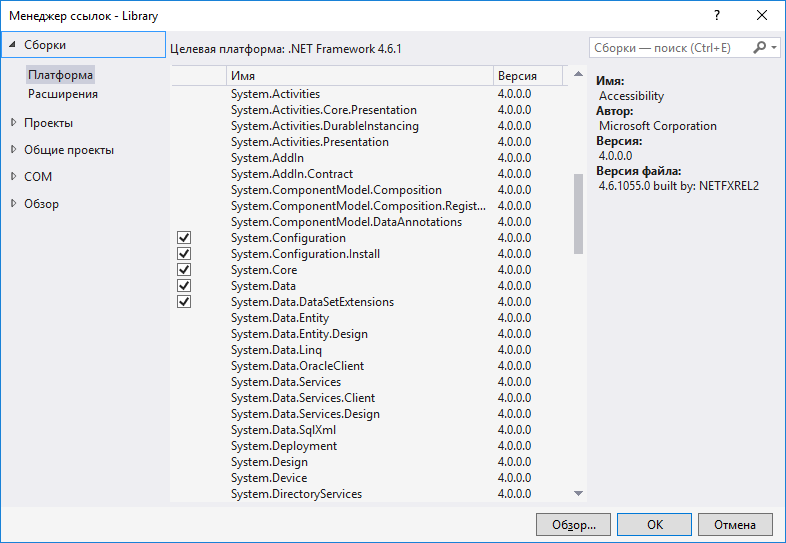


Рисунок 2

Создайте фабрику сессий с параметрами, указанными в файле конфигураций:

var builder = new SqlConnectionStringBuilder

{

DataSource = ConfigurationManager

.AppSettings["databaseLocation"],

InitialCatalog = ConfigurationManager

.AppSettings["databaseName"],

IntegratedSecurity = true

};

var sessionFactory = SessionFactory

.GetSessionFactory(builder.ConnectionString);

Существует несколько ключевых компонентов, указанных на диаграмме (Рисунок 3).

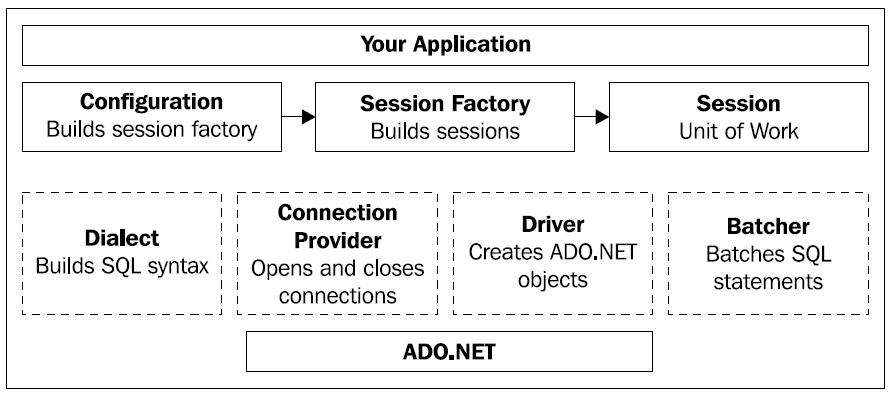


Рисунок 3

**Configuration.** При запуске приложения NHibernate создаёт объекты конфигурации (Configuration). В данном случае мы создаём эти объекты из файла конфигурации App.config.Объекты конфигурации ответственны за загрузку отображения, приспособляемость приложения к дополнительной информации, создание метаданных отображения, и, в конечном счёте, создание фабрики сессий. Создание фабрики сессий является дорогой операцией, которая должна быть выполнена только один раз за время жизни приложения.

**SessionFactory.** Фабрика сессий ответственна за создание сессий. В отличие от создания фабрики, создание сессий является дешёвой операций.

Session. Сессия представляет **единицу работы** [1](unit of work) в приложении. Мартин Фаулер определяет единицу работы как объект, который обслуживает список объектов, участвующих в процессе транзакции, и координирующий запись изменений и решение проблем, связанных с проблемами параллельности. Сессия отслеживает изменения, которым подвергается сущность, записывая все изменения единожды БД. В библиотеке NHibernate этот процесс - записать все в БД и за один раз - называется транзакция отложенной записи (transaction write-behind). В дополнении, сессия является точкой входа для множества API.Единица работы – это объектно-реляционное типовое решение, предназначенное для моделирования поведения, которое содержит список объектов, охватываемых бизнес-транзакцией, координирует запись изменений в БД и разрешает проблемы параллелизма [1]. Единица работы – это объект, который должен быть создан каждый раз, как только вы начинаете работу с БД, и отслеживает и координирует все выполняемые изменения. Более подробно о шаблоне проектирования «единица работы» читайте в книге [1] (Patterns of Enterprise Application Architecture).

Сессия действует как посредник между нашим приложением и множеством ключевых компонентов библиотеки NHibernate. Типичное приложение не будет взаимодействовать с этими компонентами напрямую, но понимание, как они работают, влияет на понимание работы всей библиотеки NHibernate. Unit of Work - наверно, самый популярный паттерн модификации данных. Он описывает следующую стратегию работы с данными [3]:

1. Получили объекты из ORM.
2. Изменили какое угодно их количество.
3. Сказали ORM сохранить данные в базу (или откатиться).
4. ORM сохранил все изменения одним махом (или откатился).

**Dialect.** Диалект используется для построения синтаксиса SQL конкретного СУБД. Например, в Microsoft SQL Server для вывода результирующего набора максимум в 20 строк, применяется следующий оператор SELECT TOP 20. Для того, чтобы сделать тоже самое в SQLite, необходимо выбрать другой оператор, который мы добавляем в конец запроса LIMIT 20. Каждый диалект обеспечивает необходимый синтаксис SQL и другую информацию для того, чтобы построить корректный запрос в конкретной СУБД.

**Driver.** Драйвер ответственный за построение пакетировщика (batcher), создающего объекты IDbConnection и IDbCommand и готовящего эти команды. Connection provider ответственный только за открытие и закрытие связи с БД. Пакетный обработчик управляет объектами чтения данных и упаковкой команд, которые будут посланы к БД. Между тем, только SqlClientDriver и OracleDataDriver поддерживают пакетную обработку (упаковку в пакеты). Те драйверы, что не поддерживают возможность пакетной обработки, предоставляют NonBatchingBatcher для управления IDbCommands и IDataReaders и симулируют существование единственного логического пакета команд.

### Создание строки подключения для СУБД Access

В первом приближении укажем параметры подключения, используя «магические» строки:

const String fullPath = @"...\Library.mdb";

Путь к БД также можно добавить в файл конфигурации. Для этого откройте файл App.config своего проекта и добавьте в него параметры:

<appSettings>

<add key="databaseLocation" value = "...\Library.mdb"/>

Создайте фабрику сессий с параметрами, указанными в файле конфигураций:

var builder = new SqlConnectionStringBuilder

{

DataSource = ConfigurationManager

.AppSettings["databaseLocation"],

};

var sessionFactory = SessionFactory

.GetSessionFactory(builder.ConnectionString);

### Создание фабрики сессий с возможностью выбора СУБД

Для реализации необходимости в переключении СУБД (Access, MS SQL Server 2008, SQLite), определим папку Staff, а в ней - следующие классы:

public enum DatabaseManagementSystem

{

MS\_Access,

MS\_SQL\_Server\_2008,

SQLite

}

public static class SessionFactory

{

public static ISessionFactory GetSessionFactory(DatabaseManagementSystem dbmsType)

{

var configuration = Fluently

.Configure() .Database(GetConfiguration(dbmsType)) .Mappings(m => m.FluentMappings

.AddFromAssembly(Assembly

.GetExecutingAssembly()) .Conventions

.Add<MyNoteConvention>() .Conventions

.Add<MyForeignKeyConvention>())

.BuildConfiguration();

return configuration.BuildSessionFactory();

}

private static IPersistenceConfigurer GetConfiguration(DatabaseManagementSystem dbmsType)

{

BaseConnectionConfigurer configurer;

switch (dbmsType)

{

case DatabaseManagementSystem.MS\_Access:

configurer = new AccessConnectionConfigurer();

break;

case DatabaseManagementSystem.MS\_SQL\_Server\_2008:

configurer = new Mssql2008ConnectionConfigurer();

break;

case DatabaseManagementSystem.SQLite:

configurer = new SqliteConnectionConfigurer();

break;

default:

throw new ArgumentOutOfRangeException(nameof(dbmsType));

}

return configurer.GetConfiguration();

}

}

}

Реализуем общий класс для любой СУБД

public abstract class BaseConnectionConfigurer

{

protected DbConnectionStringBuilder Builder;

protected String Prefix;

public abstract IPersistenceConfigurer GetConfiguration();

}

Для СУБД Access реализуем следующий класс

public class AccessConnectionConfigurer : BaseConnectionConfigurer

{

public AccessConnectionConfigurer()

{

Prefix = "access";

Builder = new OleDbConnectionStringBuilder

{

DataSource =

$"{ConfigurationManager

.AppSettings[$"{Prefix}DatabaseLocation"]}

{ConfigurationManager

.AppSettings[$"{Prefix}DatabaseName"]}

{ConfigurationManager

.AppSettings[$"{Prefix}DatabaseExtention"]}"

};

}

public override IPersistenceConfigurer GetConfiguration()

{

return JetDriverConfiguration

.Standard

.ConnectionString(c => c

.DatabaseFile(((OleDbConnectionStringBuilder)Builder)

.DataSource));

}

}

Для СУБД MS SQL Server

public class Mssql2008ConnectionConfigurer : BaseConnectionConfigurer

{

public Mssql2008ConnectionConfigurer()

{

Prefix = "mssql";

Builder = new SqlConnectionStringBuilder

{

DataSource = ConfigurationManager

.AppSettings[$"{Prefix}DatabaseLocation"],

InitialCatalog = ConfigurationManager

.AppSettings[$"{Prefix}DatabaseName"],

IntegratedSecurity = true

};

}

public override IPersistenceConfigurer GetConfiguration()

{

return MsSqlConfiguration

.MsSql2008 .ConnectionString(Builder

.ConnectionString);

}

}

Для СУБД SQLite

public SqliteConnectionConfigurer()

{

Prefix = "sqlite";

Builder = new OleDbConnectionStringBuilder

{

DataSource =

$"{ConfigurationManager

.AppSettings[$"{Prefix}DatabaseLocation"]}

{ConfigurationManager

.AppSettings[$"{Prefix}DatabaseName"]}

{ConfigurationManager

.AppSettings[$"{Prefix}DatabaseExtention"]}"

public override IPersistenceConfigurer GetConfiguration()

{

return SQLiteConfiguration

.Standard

.UsingFile(((OleDbConnectionStringBuilder) Builder)

.DataSource);

}

}

Определим элементы настройки по средствам узлов XML элементов

<?xml version="1.0" encoding="utf-8"?>

<configuration>

<startup>

<supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.6.1"/>

</startup>

<appSettings>

<!-- MS SQL Server 2008 -->

<add key="mssqlDatabaseName" value="Library" />

<add key="mssqlDatabaseLocation" value="… \sqlexpress" />

<!-- MS SQL Server 2008 -->

<!-- MS Access -->

<add key="accessDatabaseLocation" value="..\..\databases\" />

<add key="accessDatabaseName" value="Library" />

<add key="accessDatabaseExtention" value=".mdb" />

<!-- <add key="accessDatabaseProvider" value="Microsoft.Jet.OLEDB.4.0" /> -->

<!-- MS Access -->

<!-- SQLite -->

<add key="sqliteDatabaseLocation" value="..\..\databases\" />

<add key="sqliteDatabaseName" value="Library" />

<add key="sqliteDatabaseExtention" value=".sqlite" />

<!-- <add key="accessDatabaseProvider" value="Microsoft.Jet.OLEDB.4.0" /> -->

<!-- SQLite -->

</appSettings>

</configuration>

## Сессии и Транзакции

Сессии и транзакции являются двумя важными объектами, которые проводятся NHibernate фреймворком. Через объект сессии мы можем взаимодействовать с БД и выполнять различные операции на ней. Объект транзакции позволяет управлять множеством операций как единым блоком.

### Сессии

NHibernate сессию можно представить как абстрактный или виртуальный канал для БД. Раньше вам надо было создавать ADOConnection, открывать Connection, передавать Connection к объекту Command, создавать DataReader из объекта Connection и так далее.

С NHibernate мы просим sessionFactory объект Session, и все. NHibernate поддерживает все «реальные» сессии с БД (соединение, пулы и т.д.). Мы используем преимущества соединения с БД без знания всех тонкостей, лежащих в его основе.

Через объект сессии мы можем добавить новые данные в БД, изменить или удалить существующие данные, а также считывать данные из БД. Все эти операции могут быть выполнены в стиле ООП, без знания SQL и других специфических тонкостей БД. Объект сессии позволяет нам взаимодействовать с данными, хранящимися в БД, и не зависеть от выбранной СУБД (SQL Server, MySQL, Oracle и т.д.). NHibernate полностью абстрагирует эти детали от нас.

### Транзакции

Транзакция – это концепция, которая позволяет нам работать с множеством заданий как с одним блоком. Результат выполнения транзакции такой же, если бы все составные шаги были бы выполнены успешно, а, если хотя бы один шаг не выполнился, то вся система возвратилась в исходное состояние, как будто ничего и не происходило.

Транзакция обладает следующими характеристиками (ACID – atomic, consistent, isolated, durable):

* *Атомарность* – операция должна быть выполнена как одно целое и не может быть разделена на меньшие выполняемые части.
* *Согласованность* – результат выполнения транзакции должен оставлять систему в согласованном состоянии.
* *Изолированность* – операция должна проходить изолированно, т.е. незаконченные транзакции не должны быть видны извне.
* *Долговечность* – результат операции является долговечным.

Способ организации бизнес-логики по процедурам, каждая из которых обслуживает один запрос, инициируемый слоем представления, назван М. Фаулером (**сценарий транзакции** Transaction Script[1]). Сценарий транзакций организует логику вычислительного процесса в виде единой процедуры, которая обращается к БД напрямую или при помощи кода тонкой оболочки. Каждой транзакции ставится в соответствие собственный сценарий транзакции [1].

### Фабрика сессий (session factory)

NHibernate использует объект фабрика, чтобы создать сущности сессии. Один объект фабрики сессии может создавать множество объектов сессии. Создание новой сессии очень дешёвая операция, в отличие от трудозатратной операции – создание фабрики сессий. В зависимости от сложности системы, на создание объекта фабрики сессий тратится значительное количество времени. По этой причине мы должны создавать фабрику сессий единожды в течении жизненного цикла приложения.

Фабрика сессий специфична к БД. Если наше приложение нуждается только в одной БД, нам необходима только одна фабрика. Если наше приложение взаимодействует с несколькими разными БД, тогда нам необходим объект фабрики сессий для каждой БД. В более развитых сценариях может случится так, что понадобится более, чем один объект фабрики сессий, даже, если мы имеем доступ к единственной БД. Причиной для этого может служить то, что наша БД имеет несколько разных схем, и мы хотим иметь для каждой схемы свой объект фабрики сессий. Другой причиной, по которой может существовать более одного объекта фабрики сессий для единственной БД, является существование нескольких предметных областей (часто называемый **ограниченный контекст** [6]), и мы хотим иметь доступ к БД из каждой подобласти через сессии, созданные фабрикой сессий ограниченного контекста.

Фабрика сессий потокобезопасна. Код, запущенный в разных потоках, может использовать тот же самый объект фабрики сессий для создания новых объектов сессии. Это контрастирует с тем фактом, что объект сессии может быть использован только в единственном потоке. Другими словами, объект сессии не потокобезопасен.

### Создание вашей первой сессии

Имея объект sessionFactory мы можем вызвать метод OpenSession на нем:

var session = sessionFactory.OpenSession();

Обычно, мы можем открывать сессию в выражении using для того, чтобы гарантировать, что сессия закрыта и прекратила своё существование. Не имеет значения, с каким результатом. Мы можем так же использовать транзакцию для лучшей предсказуемости исхода операций. Транзакция может либо полностью удачно завершиться, либо полностью быть отменена, других исходов быть не может. Это облегчает написание нашего приложения:

using (var session = sessionFactory. OpenSession())

{

using (var transaction = session.BeginTransaction())

{

//create, update, delete or read data

transaction.Commit();

}

}

### Зачем нам вызывать Commit?

Вы заметили, что пример содержит выражение transaction.Commit(). Обычно, Commit закрывает нашу транзакцию и синхронизирует кэшированные объекты с БД. Только после успешного выполнения метода выполняются все наши изменения в БД. Помните, что только после явного вызова все сделанные изменения проталкиваются из буфера памяти и будут сохранены.

В этой простой конфигурации NHibernate использует кэш первого уровня (или кэш сессий) для сокращения объектов. Когда вы сохраняете или запрашиваете объект из БД, он помещается в кэш. Если вы не говорите NHibernate удалить или изменить объект, то NHibernate выталкивает объект из кэш, а не из БД, улучшая тем самым производительность.

### CRUD

CRUD (create, read, update, delete) - акроним для четырех основных функций БД (создание, чтение, обновление и удаление).

### Добавление новых данных в БД

Для создания новых сущностей мы вызываем метод Save объекта NHibernate сессии. Только после этого NHibernate будет знать о существовании соответствующей сущности и возьмёт ответственность за сохранение её в БД:

var newShelfId = (Int32)session.Save(newShelf);

Заметьте, что метод возвращает Id новой сгенерированной записи. Т.к. существуют различные алгоритмы создания Id (int против GUID), тип возвращаемого объекта мы должны явно преобразовать к ожидаемому типу. Вместо того, чтобы использовать этот тип возвращаемого значения метода Save и преобразовывать его к нужному типу, можно получить доступ к свойству Id только что записанной сущности, используя полученное значение Id:

var newShelfId = newShelf.Id;

Также мы можем использовать метод Persist объекта сессии, чтобы сохранить временные сущности. Возвращаемый тип метода Persist - void, поэтому мы можем напрямую использовать свойство Id сущности для получения значения только сгенерированного Id сущности. В этом случае нет необходимости в преобразовании типов.

Важно заметить, что для сохранения данных мы использовали методы Save и Persist. Однако NHibernate не собирается ничего вставлять в БД. Только единожды сессия добавляет соответствующие данные в БД. Единственным исключением является случай использования **Post-insert POID** для создания ID нашей сущности (создание ID на стороне БД). С другой стороны, этот круговорот к БД ломает концепцию **единицы работы**.Это является основной причиной, почему не рекомендуется использовать генераторы ID на стороне БД, в отличие от генераторов, например, hilo или GUID.

### Чтение данных из БД

Для чтения единственной записи из БД, имеющей идентификатор первичного ключа, мы можем использовать методы Get или Load. Следующий фрагмент кода загружает сущность из БД Shelf с ID, равным 1:

var shelf = session.Get<Shelf>(1);

Т.е., для того, чтобы использовать Get или Load, мы должны знать ID сущности, к которой мы хотим получить доступ.

Если мы хотим загрузить список всех хранящихся сущностей заданного типа, мы должны использовать метод Query (расширение) объекта сессии. Этот метод является частью движка LINQ to NHibernate, более детально который будет описан ниже.

Для того чтобы загрузить список всех издательств, мы можем использовать следующий фрагмент кода:

var allPublishing = session.Query<Publishing>().ToList();

Отметим, что выражение ToList() вызывается в конце. LINQ to NHibernate возвращает список типа IQueryable<Publishing>, который загружается «лениво». Если мы хотим, чтобы NHibernate загружал список «жадно», мы вызываем ToList().

### Get против Load

Вы можете быть удивлены, зачем сессия NHibernate поддерживает методы Get и Load. Одинаковы ли оба? Если нет, какой выбрать?

Дело в том, что методы Get и Load работают совершенно по-разному при разрешённой «ленивой» загрузке. Оба эти метода являются обобщёнными к загружаемой сущности. Если мы хотим загрузить сущность Publishing, будем использовать следующий фрагмент кода:

var publishing = session.Get<Publishing>(1);

Метод физически извлекает запрашиваемую сущность из БД, если такая существует, иначе, если запрашиваемой записи с указанной ID нет, то метод возвратит NULL.

Если мы используем метод Load, то NHibernate не извлекает сущность из БД, а создаёт прокси объект для нас. Единственным свойством прокси-сущности, ассоциированным с данными, является ID, посланным нами в качестве параметра. Таким образом, когда выполняется следующий фрагмент кода:

var publishing = session. Load <Publishing>(1);

мы имеем прокси сущность Publishing, ID которой равен 1. В тот момент, когда наш код пытается получить доступ к свойству, отличному от ID, NHibernate загружает сущность из БД. На самом деле, это то, что мы называем отложенной загрузкой, так как вызов БД откладывается до последнего момента.

Если в БД не существует записи с ID равной 1, NHibernate все равно создаст прокси для сущности Publishing и проассоциирует её с ID равной 1. Однако, в тот момент, когда мы попытаемся получить доступ к другим свойствам этой сущности, NHibernate будет пытаться получить сущность из БД и, т.к. такой записи нет, выдаст ошибку.

Зная все эти особенности, будем ли использовать метод Load? Мы хотим использовать метод Get объекта сессии, когда нам необходимо получить доступ и влиять на сущность. Однако, мы захотим использовать метод Load, когда нам нет необходимости действительно что-то менять. Например, если, нам необходимо изменить *Город* (City) уже существующей записи *Издательства* (Publishing). Publishing принадлежит City, поэтому сущность Publishing имеет свойство типа City. Нам не нужно изменять одновременно сущность Publishing и City, только использовать City. Следующий фрагмент кода показывает, как достигнуть желаемых результатов:

var publishing = session.Get<Publishing>(publishingId);

publishing.City = session.Load<City>(newCityId);

NHibernate будет генерировать выражение SELECT для того, чтобы загрузить *Издательство*, но не будет загружать *Город*. Просто будет просто создавать прокси, который мы можем установить как новый город для выбранного издательства.

Т.к. метод Load не получает доступ к БД для того, чтобы верифицировать, что запрашиваемые данные действительно существуют в БД, мы сами должны быть уверены, что они действительно существуют. Иначе, NHibernate выбросит исключение типа foreign key violation в тот момент, когда происходит запись в БД.

### Обновление существующих данных

Т.к. объект сессии NHibernate сохраняет все изменения, которые мы сделали к загруженному объекту, мы не должны явно вызывать метод update или любые другие методы, позволяющие сохранить изменения к БД. В тот момент, когда происходит запись объекта сессии, все изменения автоматически будут сохранены. Следующий фрагмент показывает, как добиться желаемого результата:

using (var session = sessionFactory. OpenSession())

{

using (var tx = session.BeginTransaction())

{

var publishing = session.Get<Publishing>(1);

publishing.PublishingOffice = “Packt Publishing Ltd.”;//новое название издательства

tx.Commit();

}

}

### Удаление данных

Для того, чтобы удалить существующую запись из БД, мы сначала должны загрузить её, и затем послать объект в качестве параметра метода Delete, как показано в примере:

var publishingToDelete = session.Load <Publishing>(publishingId);

session.Delete(publishingToDelete);

Сущность только удалена из БД с помощью кода, показанного выше, но она не удалена из памяти (ещё), она снова стала временной.

Отмечая это, и в свою очередь, чтобы не вызвать ненужных обращений к БД, мы можем использовать метод вместо метода. Результатом выполнения описанного выше кода, будет единственное SQL delete выражение, которое посылается в момент записи объекта сессии.

Приведённая выше оптимизация не применима в том случае, когда сущность, которая будет удалена, содержит зависимые сущности (и/ или) коллекции дочерних сущностей, отображаемые с ограничением каскадного удаления. В этом случае должен будет загрузить сущность в память.

### Кэш первого уровня или коллекция объектов(identity map [1])

Для достижения лучшей производительности NHibernate интеллектуально кэширует данные. Существуют разные механизмы кэширования данных. Наиболее важным из них является кэширование первого уровня. Каждый объект сессии обслуживает свой собственный кэш. Кэш создается тогда же, когда создается объект сессии и разрушается во время уничтожения объекта сессии.

Кэш – это хэш-таблица. Хэш-таблица хранит значения уникального ключа. Зная этот ключ, значения могут быть извлечены очень эффективно.

NHibernate имеет дело с сущностями. Уникальность сущности определяется с помощью идентификатора ID. Две сущности являются эквивалентными, если они принадлежат к одному типу и их ID совпадают. Существует ограничение на возможность существования двух объектов одинакового типа с одинаковым ID - их не может быть. Причиной тому является невозможность нахождения системы в неконсистентном (несогласованном) состоянии.

NHibernate делает следующее: Сущность, загружается из БД с помощью своего ID. NHibernate кладёт сущность в кэш первого уровня. Ключ для доступа к этой сущности – это ID. Когда система пытается загрузить ту же самую сущность снова из БД, тогда объект сессии сначала опрашивает свой кэш. Если сущность уже существует в кэше, то NHibernate возвращает эту сущность из кэша. Только в том случае, если кэш не содержит сущность с запрашиваемым ID, объект сессии загружает сущность из БД.

Объект сессии использует ID сущности как ключ для хранения в кэше. Кэш первого уровня так же называется коллекция объектов (identity map [3]).

### Очистка кэша

Мы можем попросить сессию убрать сущность из кэша первого уровня, используя следующее выражение:

session.Evict(shelf);

Если мы хотим полностью очистить кэш, то нам следует использовать фрагмент кода:

session.Clear();

Приведённые выше выражения следует использовать только в очень специфических ситуациях, потому что итогом некорректного использования будет значительное снижение производительности. Рекомендуется использовать данные операторы только для написания тестов для системы.

### Обновление сущностей в кэше

Для обновления единственной сущности (уже помещённой в кэш первого уровня из БД) используется следующий код:

session.Refresh(shelf);

Представленный код будет перезагружать состояние сущности shelf из БД. Применение данного сценария имеет смысл только в случае изменения сущности в БД другим приложением или другим процессом при сохранении открытой нашей сессии.

### Ни одного взаимодействия с БД без транзакций!

Какие операции вы бы не проводили с данными, хранимыми в БД, вам следует завернуть их в транзакцию:

using (var session = sessionFactory. OpenSession())

{

using (var tx = session.BeginTransaction())

{

...

tx.Commit();

}

}

При выборке данных для оптимизации работы и предсказуемости результата мы должны явно обернуть операторы чтения в транзакцию. При работе с NHibernate это рекомендуется делать.

### Сессия NHibernate против сессии БД

Когда мы говорим о сессии БД в контексте ADO.NET, мы в действительности подразумеваем об открытом соединении с БД, через которое мы можем проводить различные команды, такие как извлечение и манипулирование данными. Т.е. сессия длится столько, сколько открыто соответствующее ADO.NET соединение с БД. В программе это будет выглядеть так:

using (var connection = new SqlConnection(“...”))

{

connection.Open();

//...запись или чтение из БД

connection.Close();

}

В библиотеке NHibernate сессия имеет немного другое значение. Мы все также можем воспринимать NHibernate сессию как виртуальный или абстрактный канал к БД. Однако, в течение всего времени нет необходимости в устойчивом физическом соединении с БД. NHibernate сессия может охватывать несколько ADO.NET соединений. NHibernate автоматически открывает ADO.NET соединение с БД только, если нужно записать или считать данные.

NHibernate сессия также обеспечивает и управляет первым уровнем кэша, которое улучшает производительность приложения. NHibernate сессия является контейнером единицы работы (unit of work [1]), поэтому сохраняет все изменения, которые мы сделали к системе.

### Пример сохранения записей в БД

private static void CreateRecord()

{

using (var session = sessionFactory.OpenSession())

using (var transaction = session.BeginTransaction())

try

{

var name = new Name

{

FirstName = "Лев",

MiddleName = "Николаевич",

LastName = "Толстой"

};

var author = new Author

{

AuthorName = name,

Note = "test",

Suffix = "smth",

BirthDate = Convert.ToDateTime("09/09/1828"),

DeathDate = Convert.ToDateTime("01/01/2016")

};

var book = new Book

{

Title = "Анна Каренина",

YearOfPublishing = Convert.ToDateTime("01/01/2016")

};

author.AddBook(book);

var room = new Room

{

RoomName = "4523",

Note = "Room"

};

var shelf = new Shelf

{

NumberShelf = 1,

Note = "1"

};

shelf.AddBook(book);

room.AddShelf(shelf);

var series = new Series

{

Note = "Note",

Title = "Title"

};

series.AddBook(book);

var city = new CityName { City = "Moscow" };

var publish = new Publishing

{

PublishingOffice = "Moscow",

Note = "Moscow"

};

city.AddPublishing(publish);

publish.AddBook(book);

session.Save(room);

session.Save(series);

session.Save(publish);

session.Save(city);

session.Save(shelf);

session.Save(book);

session.Save(author);

transaction.Commit();

}

catch (Exception e)

{

Console.WriteLine(e);

transaction.Rollback();

}

finally

{

session.Close();

}

}

## Запросы

Объект запроса (Query Object) – типовое решение объектно-реляционного отображения - объект, представляющий запрос к БД [1]. Применение SQL - выражений для выполнения запросов к БД в приложении ведёт не только к неудобству работы с таким приложением, так как не все разработчики хорошо владеют языком запросом. Также приложение с применением «сырого SQL» подвержено SQL инъекциям. Особенностью объекта запроса является возможность представления запросов в терминах объектов приложения, а не базы данных [1].

### Провайдер LINQ to NHibernate

Внедрение LINQ в .NET 3.5 изменило способ взаимодействия разработчиков с набором данных. До LINQ цикл foreach устанавливал правила игры, когда речь шла о взаимодействии с наборами или списками данных. Сейчас эту роль играет LINQ. Центральным для LINQ to Object является интерфейс IEnumerable<T>. LINQ to Object манипулирует набором объектов, находящихся в памяти. LINQ to NHibernate схожим образом взаимодействует с БД. Центральным интерфейсом для LINQ to NHibernate является интерфейс IQuerable<T>.

### Определение корня нашего запроса

Одним из базовых механизмов библиотеки NHibernate является идея методов расширений, применяемых к интерфейсу ISession (более подробно читайте в [7, p. 434]). Метод расширения является шаблонизированным к T, как показано в следующем отрывке коде:

var list = session.Query<Shelf>();

Метод Query<Shelf> возвращает «коллекцию объектов», удовлетворяющих интерфейсу IQuerable<Shelf>.

Заметьте, что представленный код не будет выполняться немедленно, а только тогда, когда мы начнем использовать данные, например, пробегая по коллекции, как представлено в следующем примере:

Как отмечалось выше, данное поведение называется «ленивым выполнением» (lazy evaluation) [1].

var list = session.Query<Shelf>();

foreach (var shelf in list)

{

//делаем что-нибудь с объектом полка

}

Представленный LINQ запрос будет генерировать выражение, эквивалентное следующему:

SELECT [] FROM SHELF;

Представленный запрос загружает все отображенные поля для всех записей таблицы Shelf.

### Ограничение числа возвращаемых записей

Возвращать все записи из БД является антипаттерном. При первом обращении к БД приложение вернёт все имеющиеся записи, однако, такой подход станет бутылочным горлышком после того, как в БД будут добавляться всё новые и новые записи. Таким образом, мы должны всегда ограничивать число записей до разумного максимума. Это может быть легко выполнено, используя функцию Take:

var shelves = session.Query<Shelf>().Take(20);

Представленный код вернет из БД максимум 20 полок из БД.

### Фильтрация набора данных

Обычно требуется возвращать не просто 20 записей из данной таблицы, а получать определенное подмножество записей, отвечающее некоторому условию. Ключевое слово, используемое для фильтрации – Where.

LINQ определяет метод, который берет в качестве параметра предикат от T. Предикат – это функция с одним параметром, которая возвращает Boolean, как показано на примере

Func<T, bool> predicate

или

Predicate<T> predicate

Если нам необходимо получить список всех книг, имеющих электронную версию книги

var digitalBook = session

.Query<Book>()

.Where(b => b.IsDigitalCopy);

В приведенном запросе мы использовали лямбда выражение для определения предиката. Выражение означает: «Дана книга b, берём наполнение её свойства DigitalCopy и возвращаем его». Если возвращаемое значение – true, то соответствующие книги будут включены в результирующий набор.

Иногда называют фильтрацию данных **функциями** или операциями **ограничения**.

### Отображение набора записей

Нам не всегда необходимо возвращать множество записей, включающих все поля таблицы источника. Для выбора подмножества полей мы можем использовать метод расширения Select для интерфейса IQueryable<T>. Данный метод использует в качестве параметра отображаемую функцию. Определение отображаемой функции представлено в следующем коде:

Func<TSource, TResult> mapper

Если мы хотим получить только поля *Название* и список *Авторов* для **книги**, имеющей электронную копию, то мы можем определить для этой цели класс DigitalBook.

using System;

using System.Collections.Generic;

namespace Library.Entities

{

public class DigitalBook

{

public Int32 ID { get; set; }

public String Title { get; set; }

public IList<Author> Authors { get; set; }

public DigitalBook()

{

Authors = new List<Author>();

}

public override String ToString()

{

return $"{ID} {Title}";

}

}

}

Наш запрос будет выглядеть следующим образом:

var digitalCopy = session.Query<Book>()

.Where(b => b.IsDigitalCopy)

.Select(b => new DigitalBook

{

ID = b.ID,

Title = b.Title,

Authors = b.Authors

});

Мы определили функцию отображения как лямбда выражение.

### Сортировка результирующего набора данных

Не всегда достаточно иметь только отфильтрованный набор данных. Иногда нам необходимо отсортировать результат выбора. За сортировку результата отвечает метод OrderBy(). Если нам сначала нужно видеть книги в порядке их публикации, то код может выглядеть так:

var books = session.Query<Book>()

.OrderBy(b => b.YearOfPublishing)

.Select(b => b);

Если затем нам необходимо отсортировать результат по названию, то в наш код добавим метод ThenBy():

var books = session.Query<Book>()

.OrderBy(b => b.YearOfPublishing)

.ThenBy(b => b.Title);

Мы можем проводить сортировку по всем колонкам, добавляя в конце метод ThenBy(). Для сортировки по убыванию (по году публикации и названию книги) используются соответственно методы OrderByDescending(), ThenByDescending():

var books = session.Query<Book>()

.OrderByDescending(b => b.YearOfPublishing)

.ThenByDescending(b => b.Title);

### Группировка записей

Если нам необходимо посчитать количество книг, находящихся на каждой полке (countBooksOn), то мы будем использовать метод GroupBy():

var countBooksOn = session.Query<Shelf>

.GroupBy(s => s.NumberShelf)

.Select(b => new {

Shelf = b.Key,

Count = b.Count()

});

Представленный код вернёт список анонимный типа, который содержит номер полки и количество книг на ней.

Метод GroupBy()возвращает

IEnumerable<IGrouping<TKey, TValue>>,

где TKey представляет тип значений, по которому мы группируем, и TValue представляет группируемый тип. В представленном примере TKey будет типом Int32, а TValue – типом Book. Интерфейс IGrouping предлагает нам список часто используемых агрегатных функций, таких как сумма, количество, среднее, максимальное и минимальное значение, в которых одно из значений является типом.

### Ускорение LINQ запроса для немедленного выполнения

Как было отмечено выше, LINQ запрос всегда выполняется лениво, т.е. выполнение начинается только тогда, когда мы перебираем результирующий набор. Иногда нам не нужно такое поведение. Вместо этого, мы хотим ускорить движок LINQ для немедленного выполнения запросов и возвратить результат. Для этого у нас есть разные способы. Первый, мы можем закончить запрос, вызывая методы расширения ToArray или ToList. Эти два метода немедленно начинают перебирать результирующий набор и помещать объекты результата в массив или список T:

var books = session.Query<Book>

.Where(b => b.YearOfPublishing.Year >= year)

.ToArray();

Представленный запрос загружает список всех книг, дата публикации которых не позже указанного года (year), в массив.

Имеется другой способ, позволяющий ускорить запрос для немедленного выполнения. Таким способом является метод ToDictionary. Эта функция удобна, когда нам нужно извлечь список сущностей из БД и сохранить их в коллекцию, использующую уникальный ключ для объекта. Рассмотрим пример из **Группировка записей** и дополним его методом ToDictionary:

var countBooksOn = session.Query<Shelf>

.GroupBy(book => book.Shelf.ID)

.ToDictionary(x => x.Key, x => x.Count());

В представленном примере создаётся словарь с числом книг, находящимся на каждой полке. Ключом является номер полки, значением – число книг на ней.

### Выполнение запросов не на стороне БД, а используя объекты в памяти

Иногда мы хотим выполнять запросы к БД, затем манипулировать этими данными, используя функциональность, которая не поддерживается СУБД. Для этих сценариев нам необходим способ инструктировать провайдер LINQ to NHibernate, что с этого момента все манипуляции будут проводиться в памяти, а не на стороне БД. Метод, который мы будем использовать, называется AsEnumerable().

Допустим, мы решили найти все книги, название которых удовлетворяют шаблону (pattern). Для этого мы загрузили книги из БД, а затем хотели бы использовать регулярное выражение. Однако регулярные выражения не поддерживаются большинством БД. Таким образом, фильтрация будет сделана в памяти, как показано в следующем коде:

var regex = new Regex(pattern);

var booksTitleByPattern = session.Query<Book>()

.Select(book => book.Title)

.AsEnumerable()

.Where(title => regex.Match(title).Success)

.ToList();

В данном примере проекция происходит в БД, а фильтрация - в памяти.

### Запросы с критерием

У библиотеки NHibernate имеется альтернативный API для выполнения запросов к нашей БД. Данный тип запросов называется запрос по критерию. Запрос формируется с помощью множества критериев. Существуют различные типы критериев для фильтрации, сортировки, проецирования (или отображения), а также для работы с группой записей. В прошлом такой вид запросов был возможен только с применением "магических строк". С версии NHibernate 3.0 определен данный вид запросов.

### Нетипизированные запросы с критерием

В данном разделе мы рассмотрим, как создавать запросы с критерием, которые совпадают с результатом, полученным в примерах прошлой главы.

Все запросы с критерием начинаются с определения корня нашего запросы с критерием:

var shelves = session.CreateCriteria<Shelf>();

Метод CreateCriteria возвращает объект, который реализует интерфейс ICriteria. Если мы хотим получить список всех полок, то мы должны использовать метод List<T> реализующий интерфейс ICriteria, как показано в следующем фрагменте кода:

var shelves = session

.CreateCriteria<Shelf>()

.List<Shelf>();

Метод List<Shelf>()возвращает IList<Shelf>(). В отличие от LINQ to NHibernate, запрос будет выполнен немедленно, в случае вызова метода List.

Также имеется необобщенный метод List, определяющий интерфейс ICriteria. Этот метод возвращает объект типа IList. Отдельные элементы этого списка являются массивами объектов, т.е. object[]. Количество объектов и их порядок следования зависит от числа полей и их порядка следования в запросе.

Если мы хотим ограничить число записей в результирующем наборе, то мы можем использовать функцию SetMaxResults. Для получения первых n книг из БД, используйте запрос, пример которого приведен ниже:

var firstNBooks = session  
.CreateCriteria<Book>()  
.SetMaxResults(n)  
.List<Book>();

Давайте получим отфильтрованный список книг, имеющих цифровою копию:

var books = session

.CreateCriteria<Book>()

.Add(Restrictions.Eq("IsDigitalCopy", true))

.List<Book>();

В данном примере мы использовали "магические строки" для определения имени свойства, по которому мы будем фильтровать записи. Если бы мы неправильно написали имя поля, то узнали бы об этом только во время выполнения приложения. Такого недостатка лишены запросы, рассматриваемые позже.

Индивидуальный запрос с критерием является объектом интерфейса NHibernate.Expression.ICriterion [8]. Класс Expression  определяет фабрику методов для определения определенных встроенных типов интерфейса  ICriterion .

### Запросы по образцу

Класс NHibernate.Expression.Example позволяет вам сформировать критерий запроса для заданной сущности.

var book = new Book();

book.IsDigitalCopy = true;

book.YearOfPublishing = DateTime.Now;

var books = session

.CreateCriteria<Book>()

.Add(Example.Create(book))

.List<Book>();

### Строго типизированные критерии запросов

Библиотека NHibernate позволяет определять критерии запросов без использования «магических строк». Для этой цели был добавлен метод QueryOver<T> к интерфейсу ISession. Здесь, обобщенный параметр T представляет тип сущности, к которой мы хотим построить запрос.

Используя QueryOver API, мы определяем корень нашего запроса. Для того чтобы получить список всех книг из БД, можно использовать следующий запрос:

var books = session.QueryOver<Book>();

Здесь, в противовес к Criteria API, мы не должны специализировать возвращаемый тип в списке, т.к. он уже объявлен в уровне QueryOver. Если мы хотим ограничить число записей, возвращаемых запросом, мы можем использовать метод Take. Наш запрос будет очень похож на запрос, который обсуждался ранее при изучении LINQ to NHibernate.

var shelves = session

.QueryOver<Shelf>()

.Take(20)

.List();

API позволяет нам также отфильтровать результирующий набор, используя знакомый метод Where. Чтобы получить список всех книг, имеющих электронную версию, можно написать следующий код:

var digitalBook = session

.QueryOver<Book>

.Where(book => book.IsDigitalCopy == true)

.List();

Мы также можем комбинировать множественную фильтрацию, например, получить список всех книг, имеющих электронную копию с датой выпуска после заданного года, используя следующий код:

var digitalBookAfterDate = session

.QueryOver<Book>

.Where(book => book.IsDigitalCopy == true)

.Where(book => book.YearOfPublishing >= year)

.ToList();

Вместо того чтобы использовать выражение Where, мы также можем использовать комбинацию предикатов в одном условии, используя Boolean операторы:

var digitalBookAfter2012 = session

.QueryOver<Book>

.Where(b => b.IsDigitalCopy == true &&

book.YearOfPublishing >= year)

.List();

Сортировка результатов запросов аналогично LINQ to NHibernate выполняется с помощью методов OrderBy и ThenBy. В отличие от LINQ to NHibernate, QueryOver API определяет порядок сортировки, вызывая методы Asc и Desc соответственно для сортировки по возрастанию и убыванию. Чтобы получить сначала список книг, имеющих электронные версии, а затем остальные, а затем отсортировать результат по названию, то наш код может выглядеть так:

var digitalBook = session. QueryOver <Book>

.OrderBy(b => b.IsDigitalCopy).Asc

.ThenBy(b => b.Title).Desc

.List();

## Создание Хранилища (Repository)

Data Access Object (DAO) — широко распространенный паттерн для сохранения объектов бизнес-области в базе данных. В самом широком смысле, DAO — это класс, содержащий CRUD методы для конкретной сущности. Когда к функциям приложения относится, помимо основных операций, выполнение множества запросов к БД, то лучше создать ещё один слой абстракции. В этот слой помещается весь код, написанный на SQL, который необходимо осуществить в терминах объектов. Типовое решение объектно-реляционного отображения Хранилище (Repository) выступает в роли посредника между слоем домена и слоем отображения данных, предоставляя интерфейс в виде коллекции для доступа к объектам домена [1].

Для создания Хранилища определим интерфейс:

public interface IRepository<T>

{

IList<T> GetAll();

T GetById(Int32 id);

void Save(T entity);

void Delete(T entity);

}

Создадим класс DigitalBookRepository, реализующий созданный выше интерфейс:

public class DigitalBookRepository

{

private static volatile DigitalBookRepository repository;

private static readonly Object SynchronizationRoot = new Object();

private DigitalBookRepository(){}

public static DigitalBookRepository GetInstance(ISessionFactory factory)

{

if (factory == null)

throw new ArgumentNullException(nameof(factory));

if (repository == null)

lock (SynchronizationRoot)

if (repository == null)

repository = new DigitalBookRepository { sessionFactory = factory };

return repository;

}

private ISessionFactory sessionFactory;

protected virtual ISession session => sessionFactory.OpenSession();

}

# Список литературы

|  |  |
| --- | --- |
| [1] | J. Dentler, NHibernate 3.0 Cookbook, Birmingham: Packt Publishing Ltd., 2010. |
| [2] | J. Nilsson, «The Cost of GUIDs as Primary Keys,» PEARSON, 8 March 2002. [В Интернете]. Available: http://mng.bz/4q49. [Дата обращения: 22 Июнь 2016]. |
| [3] | А. Кондуфоров, «Немного о проектировании: паттерны из мира ORM,» [В Интернете]. Available: http://merle-amber.blogspot.ru/2009/02/orm.html. [Дата обращения: .2016 12 7]. |
| [4] | Д. Палермо, «Конфигурационная инфраструктура приложения в NHibernate,» в *ASP.NET MVC 4 в Действии*, Москва, Manning, 2012. |
| [5] | М. Фаулер, Шаблоны корпоративных приложений, Москва: Вильямс, 2010. |
| [6] | В. Вернон, Реализация методов предметно-ориентированного проектирования, Москва: Вильямс, 2016. |
| [7] | Э. Троелсен и Ф. Джепикс, Язык программирования C# 6.0 и платформа .NET 4.6, Москва: Издательсткй дом "Вильямс", 2016. |