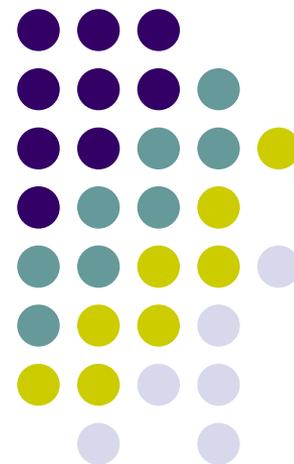


Технология ORM и её реализации



Что такое ORM?



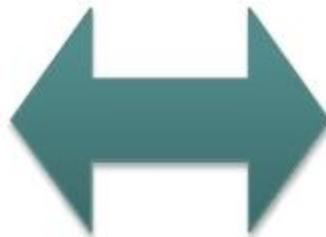
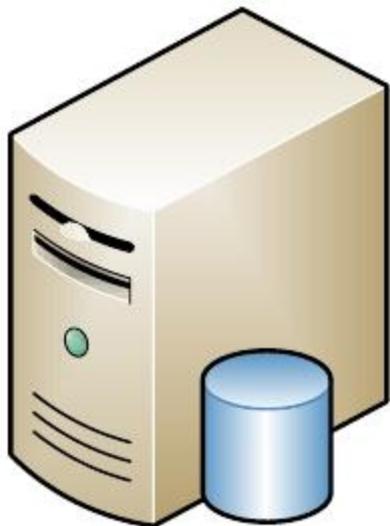
ORM (Object-relational mapping) — технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования, создавая «виртуальную объектную базу данных».

Что такое ORM?



Задача ORM состоит в управлении трансляцией объектных типов в записи баз данных и обратно.

Основная проблема состоит в том, что объекты имеют иерархическую структуру, а базы данных — реляционную.



Классы



- Классы определяют сущность
- Классы могут содержать данные и методы
- Классы могут наследовать данные и интерфейс других классов
- В качестве данных классы могут содержать экземпляры других классов, в том числе списки.

Базы данных



- Основным элементом БД является таблица
- Таблицы могут содержать только простые типы данных
- Таблицы не могут содержать массивы и списки
- Таблицы могут быть связаны внешними ключами

Зачем нужны ORM?



Ручное преобразование реляционных данных в объекты достаточно трудоёмкий процесс, который ведёт к увеличению числа ошибок.

ORM берёт на себя операции по преобразованию данных в объекты, абстрагируя программиста от знания о конкретных СУБД.

Объекты состояние которых может быть сохранено, а затем восстановлено называются хранимыми или персистентными (от англ. «persistent» — постоянный, устойчивый).

Минусы ORM решений



- Дополнительный слой абстракции может сказаться на производительности.
- Решение простых задач может оказаться слишком сложным.
- ORM решение может не быть достаточно гибким.
- Дизайн системы может оказаться зависимым от конкретной ORM-библиотеки.

NHibernate



NHibernate — ORM-решение для платформы Microsoft.NET, портированное с Java.

Это бесплатная библиотека с открытым кодом, распространяется под лицензией GNU LGPL.

Текущая версия: 3.1.0.

NHibernate



NH абстрагирует ваше приложение от лежащей в основе СУБД и диалекта языка SQL.

Поддерживаемые СУБД:

- Microsoft SQL Server
- Oracle
- Microsoft Access
- Firebird
- PostgreSQL
- DB2 UDB
- MySQL
- SQLite

Использование NHibernate



Процесс использования NHibernate состоит из четырёх этапов:

- Определение конфигурации подключения к БД (тип СУБД, SQL-диалект, строку подключения).
- Определение доменных классов, которые будут отображаться на таблицы БД.
- Написание конфигурационных XML файлов, осуществляющих отображение (mapping) реляционной структуры базы данных на доменные классы вашего приложения.
- Подключение к БД и манипуляция данными в терминах доменных классов.

Использование NHibernate

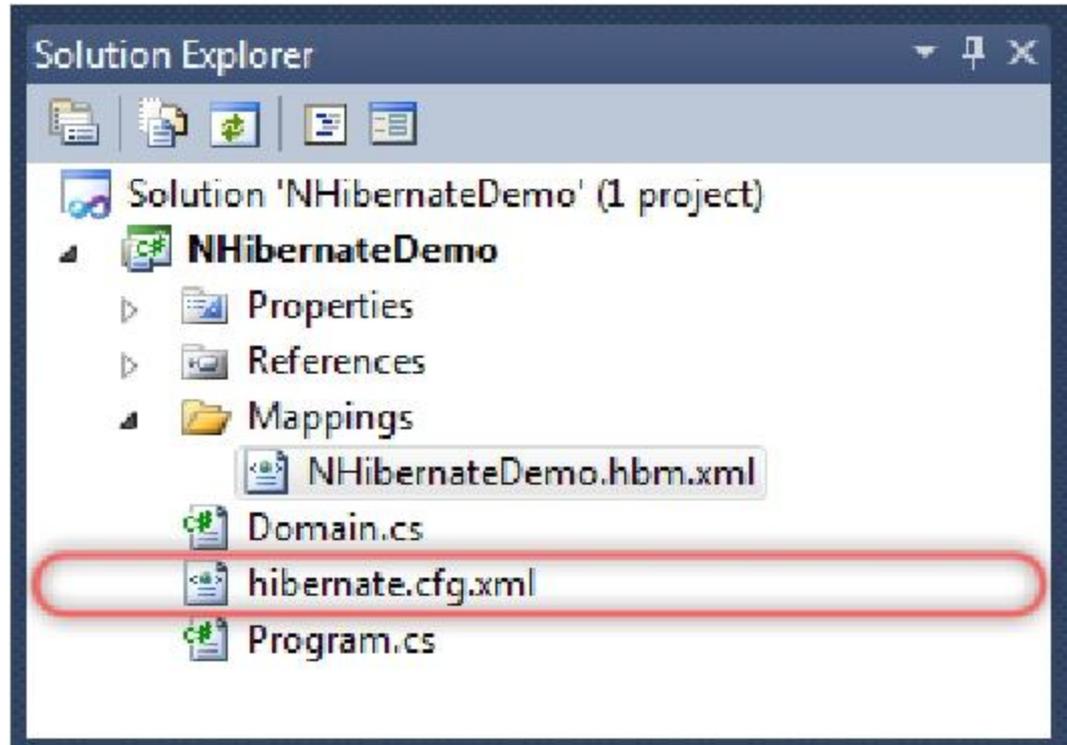


Управление параметрами подключения может осуществляться с помощью объекта **Configuration** следующими способами:

- Вызовами метода `SetProperty`.
- Определением конфигурационного XML файла с именем вида `*.cfg.xml` и загрузкой его методом `Configure`.

Метод `Configure` может быть вызван без параметра, что указывает на необходимость загрузки конфигурационного файла с именем `hibernate.cfg.xml`

Использование NHibernate



```
Configuration configuration = new Configuration();  
configuration.Configure();
```

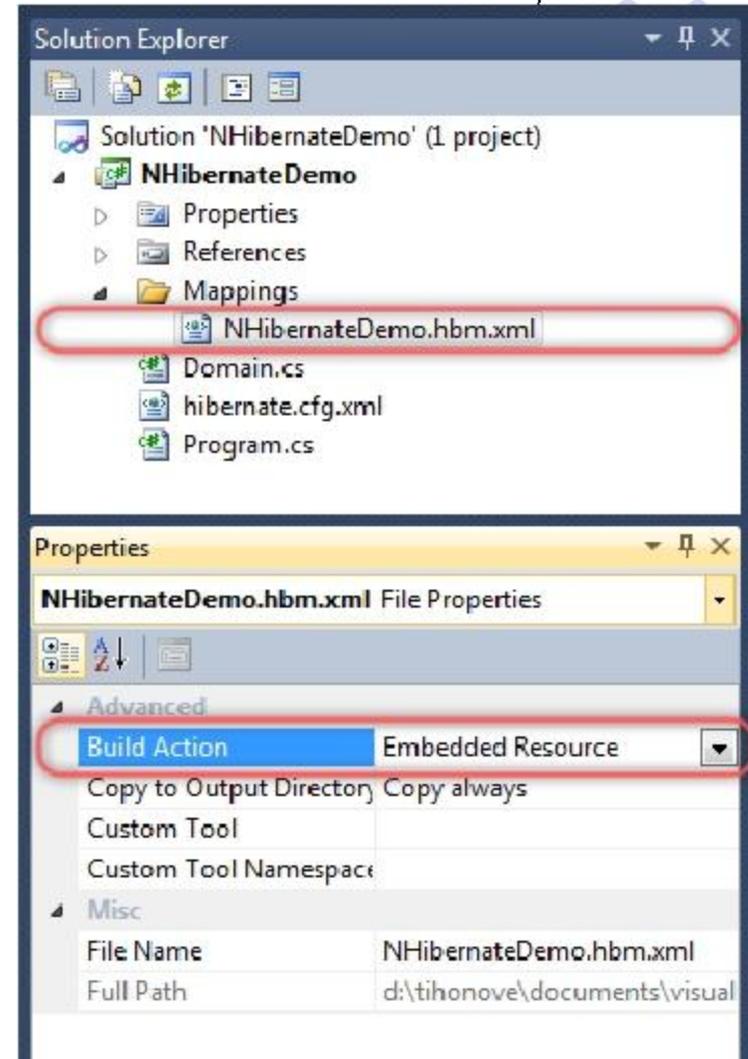
Использование NHibernate



Конфигурационные XML файлы, осуществляющие отображение, должны иметь имя вида *.hbm.xml и подключаться к проекту в качестве Embedded Resource.

Загрузка mapping-файлов осуществляется вызовом метода `AddAssembly` класса `Configuration`:

```
Configuration configuration =  
    new Configuration();  
configuration.Configure()  
configuration.AddAssembly("NHibernateDemo")  
;
```

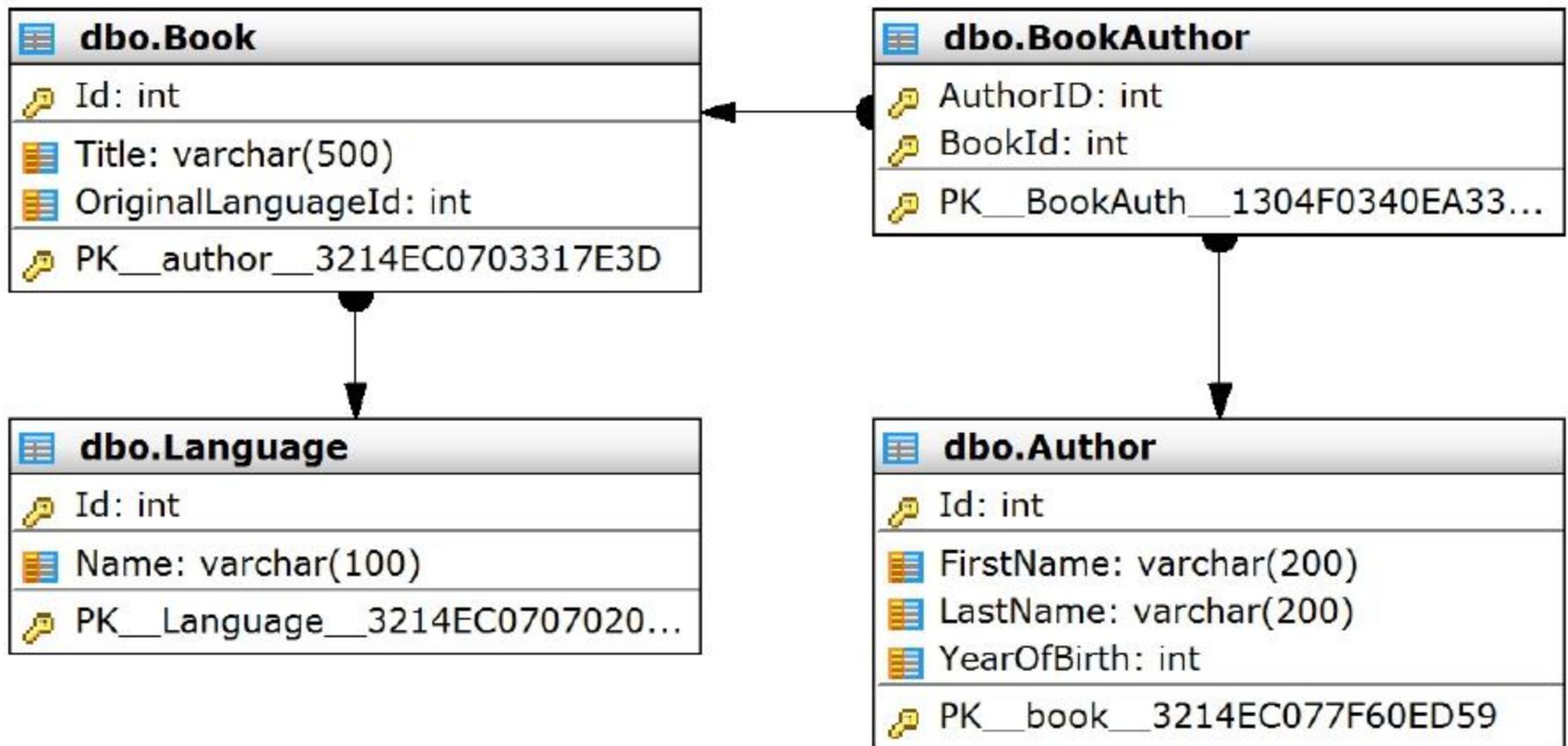


NHibernate. Пример.



Рассмотрим простой пример.

Схема базы данных имеет следующий вид:



NHibernate. Пример.



Самый простой класс нашего домена имеет следующий вид:

```
namespace Books.Domain
{
    public class Language
    {
        public virtual int Id
        { get; set; }

        public virtual string Name
        { get; set; }
    }
}
```

Отметим, что свойства класса, которые отображаются на колонки таблиц, должны быть виртуальными (особенность реализации NH).

NHibernate. Пример.



Часть XML-файла, которая отвечает за отображение данных из таблицы Language на класс Books.Domain.Language, имеет следующий вид:

```
<class name="Books.Domain.Language, NHibernateDemo"
      table="Language"
  >
  <id name="Id" type="System.Int32" >
    <column name="Id" not-null="true"
  />
  </id>
  <generator class="identity"/>
  >
  <property name="Name" column="Name"/>
</class>
>
```

НHibernate. Пример.



Ключевыми элементами конфигурации являются:

- `<class>` — связывает хранимый класс с таблицей БД.
- `<id>` — связывает свойство класса с ключевой колонкой таблицы БД.
- `<property>` — связывает свойство класса с колонкой таблицы БД.

NHibernate. Пример.



Перейдем к классу Author. Обратим внимание, что сущности Book и Author находятся в отношении «МНОГИЕ КО МНОГИМ»:

```
namespace
Books.Domain
{
    public class Author
    {
        public virtual int Id { get; set; }

        public virtual string FirstName { get; set; }

        public virtual string LastName { get; set; }

        public virtual int YearOfBirth { get; set; }

        public virtual Iesi.Collections.Generic.ISet<Book> Books
        { get; set; }
    }
}
```

NHibernate. Пример.



Настройка отображения таблицы Author на соответствующий класс будет выглядеть так:

```
<class name="Books.Domain.Author, NHibernateDemo" table="Author">
  <id name="Id" type="System.Int32" >
    <column name="Id" not-null="true"
  />
</id>
  <generator class="identity"/>
  <property name="FirstName" column="FirstName"/>
  <property name="LastName" column="LastName"/>
  <property name="YearOfBirth" type="System.Int32"
    column="YearOfBirth"/
  >
  <set name="Books"
    <key name="BookAuthor">
      <many-to-many class="Books.Domain.Book, NHibernateDemo"
        column="BookId"></many-to-many
    </key>
  </set>
  >
</class>
>
```

NHibernate. Пример.



NHibernate предоставляет набор атрибутов для отображения связей между таблицами в списки.

- **<set>** — управляет отображением данных в свойства типа `ISet` или `ISet<T>` (коллекции без отношения порядка, не допускающие дубликаты).
- **<list>** — управляет отображением данных в свойства типа `IList` или `IList<T>` (коллекции с отношением порядка, допускающие дубликаты).

Кроме того NHibernate поддерживает связи типа:

- многие ко многим;
- один ко многим;
- многие к одному.

NHibernate. Пример.



Рассмотрим класс Book, который имеет внешний ключ на таблицу Language:

```
namespace Books.Domain
{
    public class Book
    {
        public virtual int Id { get; set; }

        public virtual string Title { get; set; }

        public virtual Language OriginalLanguage
        { get; set; }
    }
}
```

NHibernate. Пример.



Настройка отображения таблицы Book на соответствующий класс будет выглядеть так:

```
<class name="Books.Domain.Book, NHibernateDemo" table="Book">
    <id name="Id" type="System.Int32" >
        <column name="Id" not-null="true"
        />
    </id> <generator class="identity"/>
    <property name="Title" column="Title"/>
    <many-to-one name="OriginalLanguage"
        column="OriginalLanguageId"
        class="Books.Domain.Language, NHibernateDemo" />
</class>
>
```

NHibernate. Пример.



Теперь рассмотрим, как после определения отображения таблиц на доменные классы оперировать с данными в этих таблицах.

Ключевыми сущностями NHibernate являются:

- **ISessionFactory** — объект, создаваемый в одном экземпляре на базу данных.
- **ISession** — ключевой объект для операций над данными. Позволяет получать и сохранять информацию.
- **ITransaction** — инкапсулирует транзакции базы данных.

Начало работы



Перед тем как начать операции с базой данных, необходимо к ней подключиться. Для этого потребуется определить контекст для NHibernate. Сделаем это определив конфигурационный файл NHibernate:

```
<hibernate-configuration xmlns="urn:hibernate-configuration-2.2"
n
  <session-factory>
    <property name="connection.driver_class">
      NHibernate.Driver.SqlClientDriver
    </property>

    <property
name="connection.connection_string">
      Server=(local);initial
catalog=books_nhibernate
    </property>

    <property name="dialect">
```

NHibernate. Пример.



Рассмотрим создание простейшего приложения, которое выводит список всех книг, имеющихся в базе данных:

```
public void ShowBooks()
{
    Configuration configuration = new Configuration();
    configuration.Configure();
    configuration.AddAssembly("NHiberanteDemo");
    ;
    ISessionFactory factory =
    configuration.BuildSessionFactory();
    ISession session = factory.OpenSession();

    // создаём запрос
    IQuery query = session.CreateQuery("from Book");

    // выполняем запрос и получаем
    данные
    IList books = query.List();
    foreach (Book book in books)
        Console.WriteLine(book.Title)
} ;
```

Построение запросов к БД



Для построения запросов к БД NHibernate предоставляет несколько механизмов:

- Criteria API
- HQL
- QBE

Criteria API



Рассмотрим пример запроса к БД с помощью Criteria API

```
public void ShowBooks()
{
    Configuration configuration = new Configuration();
    configuration.Configure();
    configuration.AddAssembly("NHibernateDemo");

    ISessionFactory factory = configuration.BuildSessionFactory();
    ISession session = factory.OpenSession();

    ICriteria criteria = session.CreateCriteria(typeof (Book));
    criteria.SetMaxResults(40);
    IList books = criteria.List();
    foreach (Book book in books)
        Console.WriteLine(book.Title)
;
    IList booksStartWithA =
        session.CreateCriteria(typeof(Book))
            .Add(Restrictions.Like("Title", "A%"))
            .List();
    foreach (Book book in booksStartWithA)
        Console.WriteLine(book.Title);
}
```

HQ

L

HQL (Hibernate Query Language) — SQL-подобный язык запросов, используемый в библиотеке Hibernate.

Рассмотрим примеры использования HQL для

получения данных:

```
public void ShowNumberOfBooks()
{
    Configuration configuration = new Configuration();
    configuration.Configure();
    configuration.AddAssembly("NHibernateDemo")
    ;
    ISessionFactory factory = configuration.BuildSessionFactory();
    ISession session = factory.OpenSession();

    IQuery query = session.CreateQuery("select count(*) from Book");
    int bookCount = (int)query.UniqueResult();
    Console.WriteLine(bookCount);
}
```



QBE



QBE (Query By Example) — механизм получения группы объектов похожих на предоставленный объект.

Пример использования:

```
public void QBEExample(ISession session)
{
    Author exampleAuthor = new Author();
    exampleAuthor.FirstName = "Лев";
    exampleAuthor.LastName = "Тостой";

    IList results =
        session.CreateCriteria(typeof(Author))
            .Add(Example.Create(exampleAuthor))
            .List();
    foreach (Author author in results)
        Console.WriteLine(author.FirstName + " " + author.LastName);
}
```

Модификация данных



Модификация данных осуществляется при помощи объекта сессии и механизма транзакций:

```
public void AddAuthorAndBook(Configuration configuration)
{
    ISessionFactory factory =
    configuration.BuildSessionFactory();
    ISession session = factory.OpenSessionBeginTransaction();
    session.beginTransaction();

    Book newBook = new Book();
    newBook.Title = "Над пропастью во
    ржи";
    session.Save(newBook);
    Author newAuthor = new Author();
    newAuthor.FirstName = "Джером";
    newAuthor.LastName =
    "Сэлинджер";
    newAuthor.Books.Add(newBook);
    session.Save(newAuthor);
    session.Commit();
    ;
    session.Close();
}
```

Doctrine



Doctrine — ORM-решение для языка PHP. Одной из ключевых возможностей Doctrine является запись запросов к БД на собственном объектно-ориентированном диалекте DQL (Doctrine Query Language), который базируется на идеях HQL.

Начиная с версии 2.0, требует PHP 5.3+.

Текущая версия: 2.0.1

Doctrine. Начало работы



Процесс использования Doctrine состоит из следующих этапов:

- Определение конфигурации подключения к БД.
- Определение доменных классов, которые будут отображаться на таблицы БД.
- Написание конфигурационных mapping файлов (XML или YAML).
- Подключение к БД и манипуляция данными в терминах доменных классов.

Конфигурация Doctrine



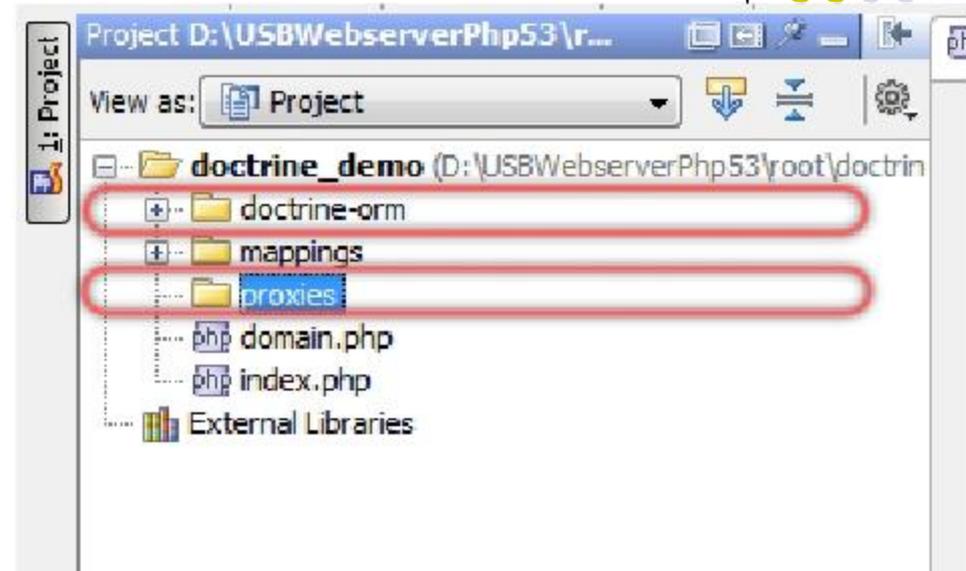
Процесс конфигурирования состоит из нескольких частей:

- Запуск автозагрузки классов.
- Определение каталога для генерации Proxy-классов.
- Определение типа Mapping-файлов.
- Определение параметров подключения.

Конфигурация Doctrine



Запуск автозагрузки
классов и определение
каталога для генерации
Proxy-классов:



```
$classLoader = new \Doctrine\Common\ClassLoader('Doctrine', 'doctrine-orm');  
$classLoader->register();
```

```
$config = new Doctrine\ORM\Configuration();
```

```
$config->setProxyDir('proxies')  
$config->setProxyNamespace('DoctrineDemo\Proxies')  
;  
$config->setAutoGenerateProxyClasses(true);
```

Конфигурация Doctrine



Автозагрузка классов — это новая возможность PHP 5, которая позволяет подключать файлы с классами по мере их необходимости. Эта возможность реализуется путём регистрации callback-функции, которая вызывается, если был обнаружен не определённый ранее класс.

Следующая строка создаёт класс, который отвечает за загрузку классов Doctrine:

```
$classLoader = new \Doctrine\Common\ClassLoader('Doctrine', 'doctrine-orm');
```

Первым параметром конструктор класса принимает пространство имён, в котором находятся классы Doctrine. Вторым параметром является путь, по которому находится библиотека.

Следующий вызов регистрирует callback-функцию загрузки классов, предоставляемую указанным выше классом:

```
$classLoader->register()
```

Конфигурация Doctrine



Для реализации «ленивой» загрузки данных Doctrine генерирует специальные проху-классы, которые загружают данные по мере обращения к свойствам соответствующих доменных классов.

Конфигурация требует определить каталог, в который будут генерироваться проху-классы, и пространство имён, в котором они будут находиться:

```
$config->setProxyDir('proxies')  
$config->setProxyNamespace('DoctrineDemo\Proxies')  
;
```

Следующее свойство определяет будут ли проху-классы генерироваться каждый раз, когда в них возникает необходимость или нет:

```
$config->setAutoGenerateProxyClasses(true)
```

Это свойство необходимо установить в false, при развертывании приложения, чтобы избежать накладных расходов связанных с генерацией проху-классов.

Конфигурация Doctrine



Doctrine поддерживает Mapping-файлы в форматах XML и YAML.

YAML (YAML Ain't Markup Language) — человекочитаемый формат сериализации данных.

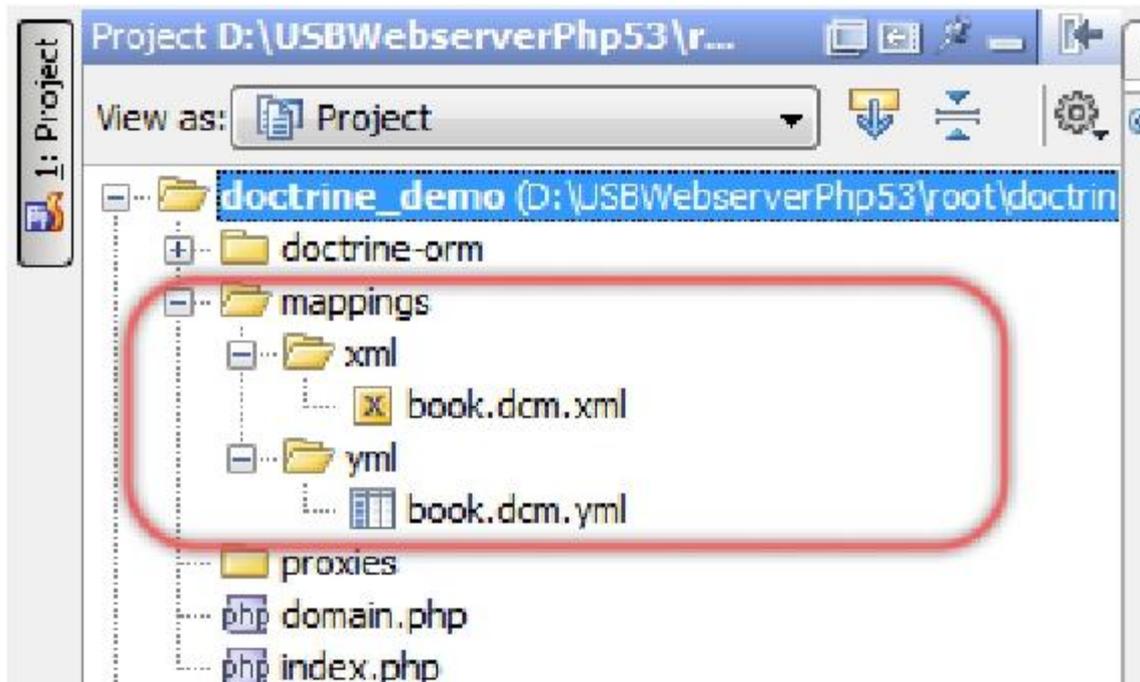
Соответствующие типы определяются следующим образом:

```
// Определение XML Mapping
файлов
$driverImpl =
    Doctrine\ORM\Mapping\Driver\XmlDriver("mappings/xml");
// Определение YAML Mapping
файлов
$driverImpl =
    Doctrine\ORM\Mapping\Driver\YamlDriver("mappings/yml");
$config->setMetadataDriverImpl($driverImpl)
;
```

Конфигурация Doctrine



Mapping-файлы должны находится в указанных каталогах. Имена Mapping-файлов для XML и YAML форматов должны быть *.dcm.xml и *.dcm.yml соответственно.



YAML



Язык изначально был задуман как язык разметки и даже рассматривался как конкурент XML, однако позже стал использоваться в основном как формат конфигурационных файлов. Синтаксис YAML минимален, особенно по сравнению с синтаксисом XML.

Рассмотрим XML-файл, который хранит свойства подключений к БД:

```
<?xml version="1.0" encoding="utf-8"?>
<connections>
  <connection name="connection01">
    <host>localhost</host>
    <database>MyDB</database>
    <password>root</password>
    <login>root</login>
  </connection>
  <connection name="connection02">
    <host>192.168.0.1</host>
    <database>production</database>
    <password>sb593f4s</password>
    <login>ds8(dg#1a</login>
  </connection>
</connections>
```

YAML



L

Древовидная структура YAML определяется при помощи отступов.

Аналогичная информация с помощью YAML может быть записана следующим образом:

```
- connection: connection01
  host: localhost
  database: MyDB
  password: root
  login: root
- connection: connection02
  host:
  database: production
  password: sb593f4s
  login: ds8(dg#1a
```

Более подробно о языке: <http://yaml.org/>

Конфигурация Doctrine



Определение параметров подключения:

```
$connection = array(  
    'driver' => 'pdo_mysql',  
    'host' => 'localhost',  
    'dbname' => 'doctrine_demo',  
    'user' => 'root',  
    'password' => 'root'  
);
```

```
$entityManager = \Doctrine\ORM\EntityManager::create($connection, $config);
```

- pdo_mysql

Doctrine поддерживает следующие типы драйверов:

- pdo_pgsql
- pdo_oci
- oci8

Doctrine. Пример



Определим доменные классы. Класс книги будет выглядеть следующим образом:

```
class Book
{
    private $id;
    private $title;
    private
    $originalLanguage;
    public function getId() { return $this->id; }
    public function setId($id) { $this->id = $id; }

    public function getTitle() { return $this->title; }
    public function setTitle($title) { $this->title = $title; }

    public function getOriginalLanguage()
    { return $this->originalLanguage; }
    public function setOriginalLanguage($originalLanguage)
    { $this->originalLanguage = $originalLanguage; }
}
```

Doctrine. Пример



Класс, определяющий сущность языка:

```
class
Language
{
    private $id;
    private $name;

    public function getId() { return $this->id; }

    public function setId($id) { $this->id = $id; }

    public function getName() { return $this->name; }

    public function setName($name) { $this->name = $name; }
}
```

Doctrine. Пример



Класс, определяющий автора:

```
class
Author
{
    private $id;
    private $firstName;
    private $lastName;
    private
    $yearOfBirth;
    private $books;
    public function __construct()
    {
        $this->books = array();
    }

    public function getId() { return $this->id; }
    public function setId($id) { $this->id = $id; }

    // ...

    public function getBooks() { return $this->books; }
}
```

Doctrine. Пример



Определим XML-файл, отображающий сущность языка на соответствующую таблицу:

```
<doctrine-mapping
 xmlns="http://doctrine-project.org/schemas/orm/doctrine-mapping
 "
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://doctrine-project.org/schemas/orm/doctrine-mapping
 http://doctrine-project.org/schemas/orm/doctrine-mapping.x
 sd">
  <entity name="Language"
 table="language">
    <id name="id" type="integer">
      <generator strategy="AUTO"
 //id
 >
    <field name="name" type="string" />
  </entity
 >
</doctrine-mapping
```

Doctrine. Пример



Аналогичный YAML файл будет следующим:

Language

```
: type: entity
  table:
    language
  id:
    type: integer
    generator:
      strategy: AUTO
  fields
    : name
      : type: string
      length: 100
```

Doctrine. Пример



Определим XML-файл, отображающий сущность книги на соответствующую таблицу:

```
<doctrine-mapping ... >

    <entity name="Book" table="book">
        <id name="id" type="integer">
            <generator strategy="AUTO" />
        </id>

        <join-column name="original_language_id" referenced-column-name="id" />
        <field name="title" type="string" />
    </many-to-one>

    <many-to-one field="originalLanguage"
target-entity="Language">
</entity>

</doctrine-mapping
>
```

Doctrine. Пример



Определим XML-файл, отображающий сущность автора на

соответствующую таблицу:

```
<doctrine-mapping>
  <entity name="Author" table="author">
    <id name="id" type="integer">
      <generator strategy="AUTO" />
    </id>
    <field name="firstName" column="first_name" type="string" />
    <field name="lastName" column="last_name" type="string" />
    <field name="yearOfBirth" column="year_of_birth"
type="integer" />

    <many-to-many field="books" target-entity="Book">
      <join-table name="book_author">
        <join-columns>
          <join-column name="author_id"
referenced-column-name="id"/>
        </join-columns>

        <inverse-join-columns>
          <join-column name="book_id"
referenced-column-name="id"/>
        </inverse-join-columns>
      </join-table>
    </many-to-many>
  </entity>
</doctrine-mapping>
```

Doctrine. Пример



Рассмотрим простейший случай получения данных из базы:

```
$entityManager = \Doctrine\ORM\EntityManager::create($connection,  
$config);
```

```
$author = $entityManager->find("Author", 1);
```

```
echo $author->getFirstName() . ' ' . $author->getLastName();
```

```
foreach($author->getBooks() as $book)  
    echo $book->getTitle();
```

Doctrine. Пример



Основные манипуляции с данными выполняются при помощи языка DQL:

```
$booksQuery = $entityManager->createQuery(
    "SELECT b FROM Book b WHERE b.title LIKE 'B%'");
$booksQuery->setMaxResults(30)
;
$books = $booksQuery->getResult();

foreach($books as $book)
    echo
$book->getTitle();
```

Doctrine. Пример



Рассмотрим пример модификации данных:

```
$entityManager = \Doctrine\ORM\EntityManager::create($connection, $config);  
  
$book = new Book();  
$book->setTitle("Над пропастью во  
ржи");  
$entityManager->persist($book);  
$entityManager->flush()  
;
```