Open Source Frameworks for Rapid Application Development

Marek Krętowski
Krzysztof Bandurski, Tomasz Łukaszuk, Tomasz Rybak

Software Departament
Faculty of Computer Science
Bialystok University of Technology

m.kretowski@pb.edu.pl
k.bandurski@pb.edu.pl, t.lukaszuk@pb.edu.pl, t.rybak@pb.edu.pl

Lecture topic

Object-Relational Mapping, Models

Object-Relational Mapping, Models: Table of content

What is ORM?

- ORM - object-relational mapping
- a programming technique for converting data between incompatible type systems in object-oriented programming languages
- this creates, in effect, a "virtual object database" that can be used from within the programming language
- both free and commercial packages available that perform object-relational mapping
- some programmers opt to create their own ORM tools

Database vs. Programing language

Database:

- relational model of data
- data is stored in tables, with relations and constraints
- quite limited set of operations on data

Programing language:

- object-oriented model of data
- classes and objects, inheritance and aggregation
- a wide range of operations

Database-Application cooperation

First way of cooperation:

- to load data from database into arrays and use it locally
- this is simulation of database inside the program
- used in ADO and ADO.NET
- introduces problems with conflicts between internal state and what is stored in database

Second way of cooperation:

- to allow for programmer to directly manipulate database using SQL
- used in JDBC, ODBC
- forces programmer to think in two modes — object-oriented (including exception) and relational, which uses transactions

Object-Relational Mapping

- There is need to join those two models: object oriented and relational
- There was (and still is) promise (never fully delivered) of object-oriented databases
- Meanwhile Object-Relational Mapping (ORM) was introduced
- Its purpose is to map objects to database and database to objects with as little code as possible
- Class represents table
- Object represents row in table
- Field (or property) represents column

Examples of ORM software

- Hibernate (Java)
- NHibernate, LINQ, Entity Framework (.NET)
- Kohana (PHP)
- SQLAlchemy (Python)
- ...

Data types mapping

- Databases use different data types than programming languages
- Programming languages use types that are close to hardware
- Databases, as more suited for business needs, have types that can be closer to problem domain
- Need to map types from database to the language

OO type	DBMS type
float	numeric, float
string	varchar, text
array	-
array of bytes	blob, bytea
object implementing functionality	timestamp

Some discussion

- Do we put business logic to application or database?
- If in database — it is consistent, but requires more powerful hardware
- It requires writing logic in SQL or other language understood by DBMS
- In case of applications — there can be many programs accessing database
- But programmers know better OO languages than SQL
- On the other hand databases live much longer than single applications
- Database and data with them stay, programs that operate on the data come and go

Types of Object/Relational mappers

- In general, ORM is interface to the database
- The simplest ones are not really mappers, but just ease writing queries for accessing data
- The more sophisticated add abstractions so programmer never sees SQL
- Some ORM tools do not allow for accessing database “behind their back”
- ORM must not assume that it is the only entity accessing data stored in DBMS
- Objects that are created to represent data can have “behaviour”
- They can contain code that is responsible for managing data
- E.g. checking constraints, formatting data
- But this is not always the case
- Domain objects vs. data structures

Source model

- Which model to start with
- No problem if we already have existing schema and need to write application that uses it
- Programmers tend to start with classes
- Administrators (including DBAs) tend to start with database schema
- ORM usually transforms class hierarchy to database schema
- If we create objects based on the schema, we make it hard to change schema, as hierarchy of objects is mirror of the schema
- When schema is generated by ORM tool from objects, it is tuned to needs of those objects
- This may mean that schema is only useful for other needs
- Tables may be denormalised

Knowledge of mechanisms used in database

- ORM is leaky abstraction
- ORM can be used to hide enormous schema
- But programmers should know what seats under the hood
- This is important for them to avoid code that has bad performance implications
- Lazy vs. eager fetching of objects
- Depth of graph of objects/tables that are fetched

Advantages and disadvantages of ORM

Advantages:

- ORM often reduces the amount of code that needs to be written
- the ability to change the database system without changing the application code

Disadvantages:

- negative effect on performance
- ORM software has been pointed to as a major factor in producing poorly designed databases
- not perform well during bulk deletions of data or joins

Database access in Django

- Django is database driven application framework
- It can use MySQL, SQLite, PostgreSQL, Oracle databases
- To avoid problems with the connections, programmer should not use the own connections but rather rely on ones provided by framework
- It presents database as collection of objects
- Each collection has methods to fetch objects (rows), filter them, get them in appropriate order, etc.

Models

- Generally, each model maps to a single database table
- Each model is a Python class that subclasses `django.db.models.Model`
- Each attribute of the model represents a database field
- Django gives you an automatically-generated database-access API

Quick example:

```
1  from django.db import models
2
3  class Person(models.Model):
4      first_name = models.CharField(max_length=30)
5      last_name = models.CharField(max_length=30)
```

Django model vs. database table

Django model:

```
1  from django.db import models
2
3  class Person(models.Model):
4      first_name = models.CharField(max_length=30)
5      last_name = models.CharField(max_length=30)
```

Create a database table SQL code:

```
1  CREATE TABLE myapp_person (
2      "id" serial NOT NULL PRIMARY KEY,
3      "first_name" varchar(30) NOT NULL,
4      "last_name" varchar(30) NOT NULL
5  );
```

Fields

- The most important part of a model
- ... and the only required part of a model
- List of database fields
- Fields are specified by class attributes

```
1   class Musician(models.Model):
2       first_name = models.CharField(max_length=50)
3       last_name = models.CharField(max_length=50)
4       instrument = models.CharField(max_length=100)
5
6   class Album(models.Model):
7       artist = models.ForeignKey(Musician)
8       name = models.CharField(max_length=100)
9       release_date = models.DateField()
10      num_stars = models.IntegerField()
```

Field types

- Field should be an instance of the appropriate `Field` class
- Field class type determine: the database column type, the widget to use in Django's admin interface, the minimal validation requirements
- Django ships with dozens of built-in field types

Type of fields

IntegerField	
AutoField	IntegerField that generates its values; usually for declaring primary keys; implemented using sequences or SERIAL
FloatField	
BooleanField	
CharField	
TextField	
EmailField	TextField that checks if it contains email
URLField	TextField that stores URL
XMLField	TextField containing XML
PhoneNumberField	
IPAddressField	
DateTimeField	also DateField and TimeField
FilePathField	
FileField	

Field options

They are given as named parameters to field object constructor.

null	boolean, whether field can contain NULL
blank	boolean, whether field can be blank
choices	collection of possible values to be stored in this field
db_column	name of column field is stored in
db_index	boolean, whether there should be index for this column in database
editable	boolean, points if field can be edited
help_text	description that is displayed in admin module
primary_key	boolean, true if field is primary key in the table; usually used for AutoField
unique	
verbose_name	

Choises field

```
1  from django.db import models
2
3  class Person(models.Model):
4      GENDER_CHOICES = (
5          (u'M', u'Male'),
6          (u'F', u'Female'),
7      )
8      name = models.CharField(max_length=60)
9      gender = models.CharField(max_length=2, choices=GENDER_CHOICES)
```

Many-to-one relationships

`django.db.models.ForeignKey`

```
1  class Manufacturer(models.Model):
2      # ...
3
4  class Car(models.Model):
5      manufacturer = models.ForeignKey(Manufacturer)
6      # ...
```

Many-to-many relationships

`django.db.models.ManyToManyField`

```
1  class Topping(models.Model):
2      # ...
3
4  class Pizza(models.Model):
5      # ...
6      toppings = models.ManyToManyField(Topping)
```

```
1  class Person(models.Model):
2      ...
3
4  class Group(models.Model):
5      members = models.ManyToManyField(Person, through='Membership')
6
7  class Membership(models.Model):
8      person = models.ForeignKey(Person)
9      group = models.ForeignKey(Group)
10     ...
```

One-to-one relationships

`django.db.models.OneToOneField`

This is most useful on the primary key of an object when that object "extends" another object in some way.

```
1  class Place(models.Model):
2      ...
3
4  class Restaurant(models.Model):
5      place = models.OneToOneField(Place, primary_key=True)
6      ...
```

Model metadata

Model metadata is "anything that's not a field", such as ordering options (ordering), database table name (db_table), or human-readable singular and plural names (verbose_name and verbose_name_plural).

```
1  class Ox(models.Model):
2      horn_length = models.IntegerField()
3
4      class Meta:
5          ordering = ["horn_length"]
6          verbose_name_plural = "oxen"
```

Model methods

Custom methods on a model add custom "row-level" functionality to objects.
This is a valuable technique for keeping business logic in one place – the model.

```
1   from django.contrib.localflavor.us.models import USStateField
2
3   class Person(models.Model):
4       first_name = models.CharField(max_length=50)
5       last_name = models.CharField(max_length=50)
6       birth_date = models.DateField()
7       address = models.CharField(max_length=100)
8       city = models.CharField(max_length=50)
9       state = USStateField()
10
11      def is_midwestern(self):
12          "Returns True if this person is from the Midwest."
13          return self.state in ('IL', 'WI', 'MI', 'IN', 'OH', 'IA', 'MO')
14
15      def _get_full_name(self):
16          "Returns the person's full name."
17          return '%s %s' % (self.first_name, self.last_name)
18      full_name = property(_get_full_name)
```

Methods automatically given to each model

- __str__ - a Python "magic method" that defines what should be returned if you call str() on the object
- __unicode__ - method is called whenever you call `unicode()` on an object
- get_absolute_url - method to tell Django how to calculate the URL for an object

```
1   class Person(models.Model):
2       first_name = models.CharField(max_length=50)
3       last_name = models.CharField(max_length=50)
4
5       def __str__(self):
6           return smart_str('%s %s' % (self.first_name, self.last_name))
7
8       def __unicode__(self):
9           return u'%s %s' % (self.first_name, self.last_name)
10
11      def get_absolute_url(self):
12          return "/people/%i/" % self.id
```

Making queries (intro)

```
1  class Blog(models.Model):
2      name = models.CharField(max_length=100)
3      tagline = models.TextField()
4      def __unicode__(self):
5          return self.name
6
7  class Author(models.Model):
8      name = models.CharField(max_length=50)
9      email = models.EmailField()
10     def __unicode__(self):
11         return self.name
12
13 class Entry(models.Model):
14     blog = models.ForeignKey(Blog)
15     headline = models.CharField(max_length=255)
16     body_text = models.TextField()
17     pub_date = models.DateTimeField()
18     mod_date = models.DateTimeField()
19     authors = models.ManyToManyField(Author)
20     n_comments = models.IntegerField()
21     n_pingbacks = models.IntegerField()
22     rating = models.IntegerField()
23     def __unicode__(self):
24         return self.headline
```

Making queries I

- Creating objects

```
1  b = Blog(name='Beatles Blog', tagline='All the latest Beatles news.')
2  b.save()
```

- Saving changes to objects

```
1  b5.name = 'New name'
2  b5.save()
3
4  entry = Entry.objects.get(pk=1)
5  cheese_blog = Blog.objects.get(name="Cheddar Talk")
6  entry.blog = cheese_blog
7  entry.save()
```

- Retrieving objects

Making queries II

```
1  all_entries = Entry.objects.all()
2
3  Entry.objects.filter(pub_date__year=2006)
4
5  Entry.objects.exclude(pub_date__gte=datetime.now())
6
7  one_entry = Entry.objects.get(pk=1)
```

- Comparing objects

```
1  some_entry == other_entry
2  some_entry.id == other_entry.id
```

- Deleting objects

```
1  e.delete()
2
3  Entry.objects.filter(pub_date__year=2005).delete()
```

Making queries III

- Updating multiple objects at once

```
1   Entry.objects.filter(pub_date__year=2007).update(headline='the same')
```

Retrieving objects - Methods returning collection of objects

all	returns all objects
filter	returns objects matching given condition
exclude	returns objects that do not match given condition
distinct	returns set (not collection) of objects; adds “DISTINCT” to SELECT clause
order_by	sorts returned object with regard to given fields
values	instead of objects returns collection of dictionaries; one dictionary represents one row from database
extra	allows to add part of literal SQL to generated query

Retrieving objects - Methods that do not return collection

get	returns exactly one object; if conditions cause return of zero or more than one objects exception is raised
get_or_create	creates object if it does not exist; then return one object, that either existed before or was just created
create	creates object; can be used instead of constructor
count	returns number of objects matching given condition
in_bulk	gets collection of primary keys and returns dictionary with given values as keys and objects as values
latest	accepts column name (of time-related type) and returns last objects according to given column

Retrieving objects - Field lookup

exact	value must be equal; this is implied condition if none is provided
iexact	case-insensitive comparison
contains	given value is part of field's value
icontains	case insensitive "contains"
gt, gte, lt, lte	arithmetical comparisons
in	accepts collection and ensures that values come from this collection
startswith, istartswith	
endswith, iendswith	
range	accepts pair and ensures that values are between given values
isnull	boolean, checks whether value is or is not NULL
pk	comparison to primary key

Source of models

- In Django classes are primary source of models
- There is possibility to read schema from database but it is not recommended
  - It is rather slow process
  - It ties Django application to particular database
  - Relational model cannot provide all details classes need
- By convention tables in database are given name “application_class”
- If our class does not contain primary key, Django will create column named “id”
- By design all primary keys must consist of single column
- This is restriction imposed on relational models

Managing database

By using command manage.py with appropriate options:

dbshell	opens command-line shell to database
validate	checks correctness of declarations of models
sqlall	prints SQL to create all objects (tables, indices, custom objects)
sql	prints SQL to create tables
sqlcustom	prints SQL needed to customize tables
sqlclear	prints SQL to drop all objects
sqlflush	prints SQL that is required so database returns to initial state, just after application creation
sqlreset	prints SQL that drops and then creates objects in database
syncdb	checks which database objects exist and creates those that are defined but do not exist yet
cleanup	removes old data from database (old sessions, etc.)
dumpdata	outputs content of the database

Database access in RoR

- ActiveRecord manages persistence of objects
- http://rubyforge.org/projects/activerecord/
- Class inheriting from ActiveRecord are mapped to relational database
- Based on Active Record Pattern
- http://en.wikipedia.org/wiki/Active_record_pattern
- ActiveRecord constructor accepts hash of names of fields and their values

Rails Database Migrations

- Migrations are a convenient way for you to alter your database in a structured and organized manner.
- Active Record tracks which migrations have already been run.
- Migrations using Ruby.
- Migrations are database independent.
- A migration is a subclass of `ActiveRecord::Migration` that implements two class methods: `up` (perform the required transformations) and `down` (revert them).

Anatomy of a Migration

```
1  class CreateProducts < ActiveRecord::Migration
2    def self.up
3      create_table :products do |t|
4        t.string :name
5        t.text :description
6
7        t.timestamps
8      end
9    end
10
11   def self.down
12     drop_table :products
13   end
14 end
```

Active Record provides methods that perform common data definition tasksin a database independent way.

`create_table`, `change_table`, `drop_table`, `add_column`, `change_column`, `rename_column`, `remove_column`, `add_index`, `remove_index`

Model class

```
1  class Person < ActiveRecord::Base
2      #validations
3
4      #assosiations
5
6      #user defined methods
7  end
```

Active Record Validations Helpers I

- validates_acceptance_of - a checkbox on the user interface was checked
- validates_associated - model has associations with other models and they also need to be validated
- validates_confirmation_of - two text fields that should receive exactly the same content
- validates_exclusion_of - the attributes' values are not included in a given set
- validates_format_of - validates the attributes' values by testing whether they match a given regular expression
- validates_inclusion_of - the attributes' values are included in a given set
- validates_length_of - validates the length of the attributes' values

Active Record Validations Helpers II

- validates_numericality_of - attributes have only numeric values
- validates_presence_of - the specified attributes are not empty
- validates_uniqueness_of - the attribute's value is unique right before the object gets saved
- validates_with - passes the record to a separate class for validation
- validates_each - validates attributes against a block

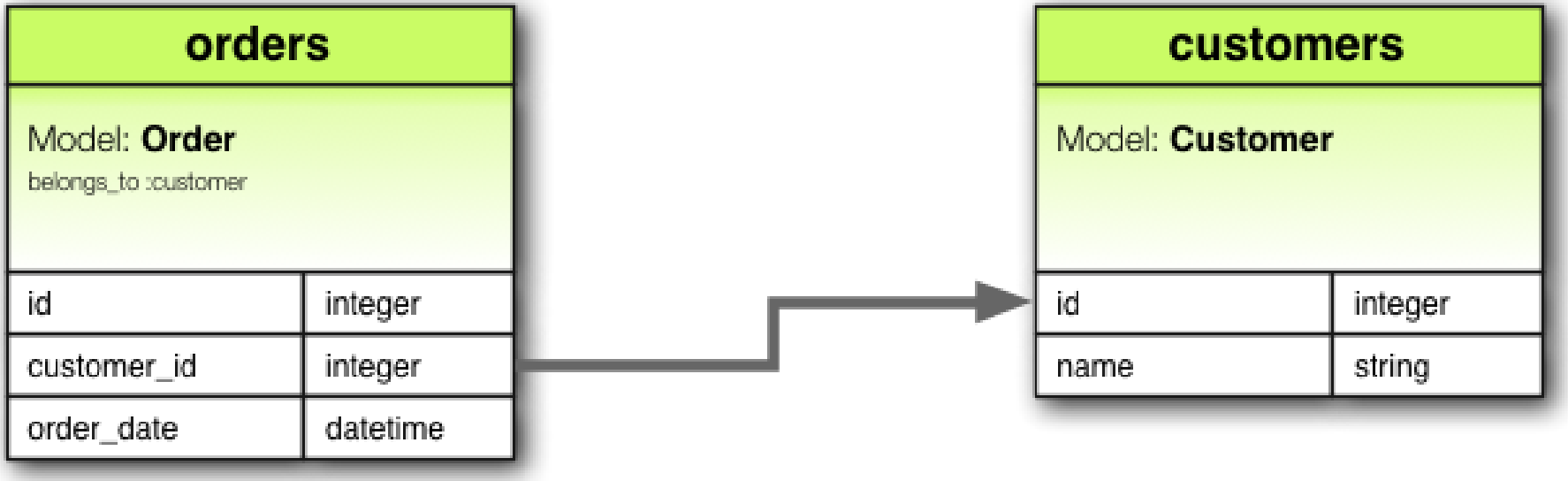
```
1  class Person < ActiveRecord::Base
2    validates_acceptance_of :terms_of_service
3    validates_confirmation_of :email
4    validates_presence_of :email_confirmation
5    validates_length_of :name, :minimum => 2
6    validates_format_of :legacy_code, :with => /\A[a-zA-Z]+\z/,
7      :message => "Only letters allowed"
8  end
```

Active Record Associations

Type of associations:

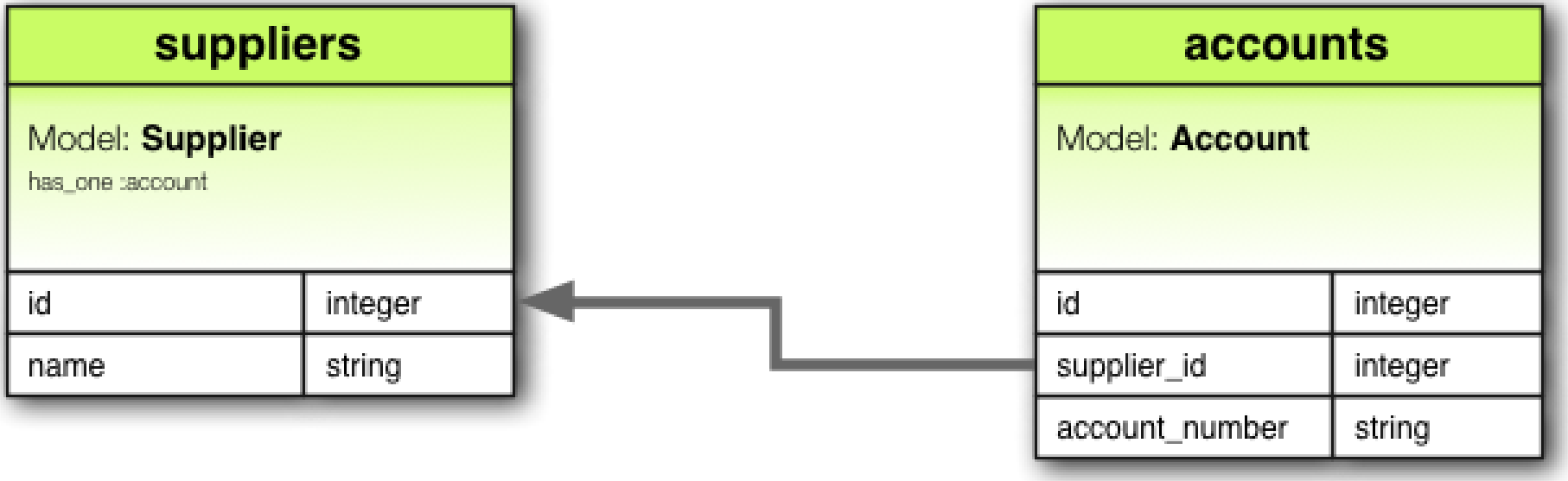
- belongs_to
- has_one
- has_many
- has_many :through
- has_one :through
- has_and_belongs_to_many

The belongs_to Association



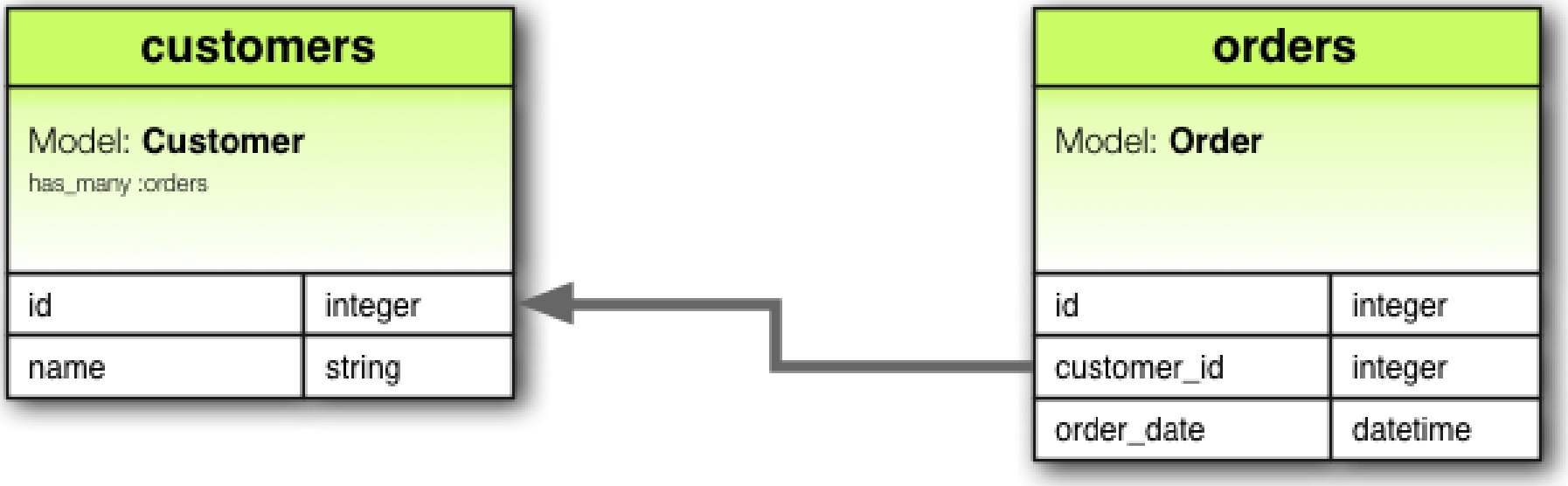

```
class Order < ActiveRecord::Base
  belongs_to :customer
end
```

The has_one Association



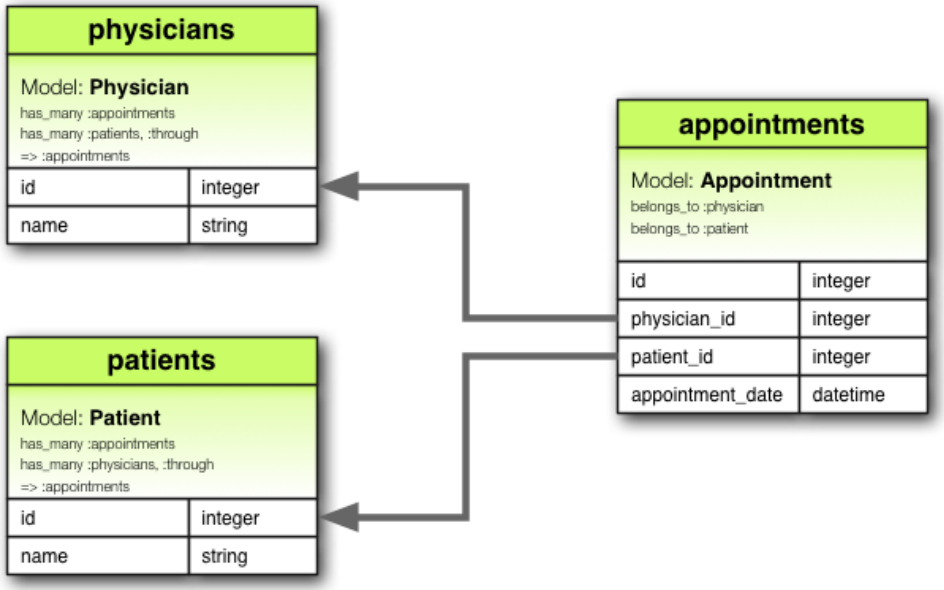

```
class Supplier < ActiveRecord::Base
  has_one :account
end
```

The has_many Association




```
class Customer < ActiveRecord::Base
  has_many :orders
end
```

The has_many :through Association

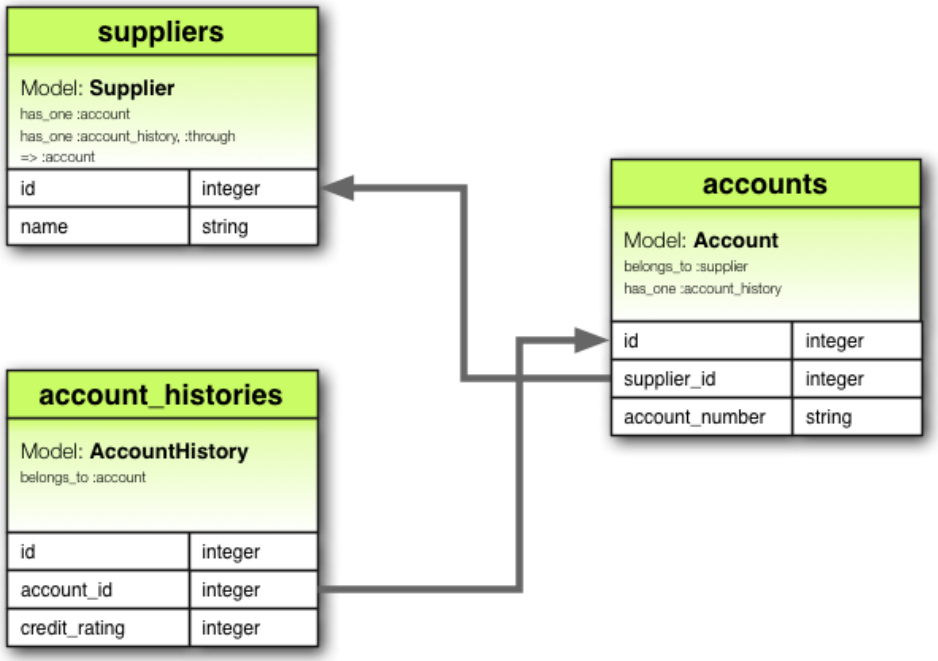



```
class Physician < ActiveRecord::Base
  has_many :appointments
  has_many :patients, :through => :appointments
end

class Appointment < ActiveRecord::Base
  belongs_to :physician
  belongs_to :patient
end

class Patient < ActiveRecord::Base
  has_many :appointments
  has_many :physicians, :through => :appointments
end
```

The has_one :through Association

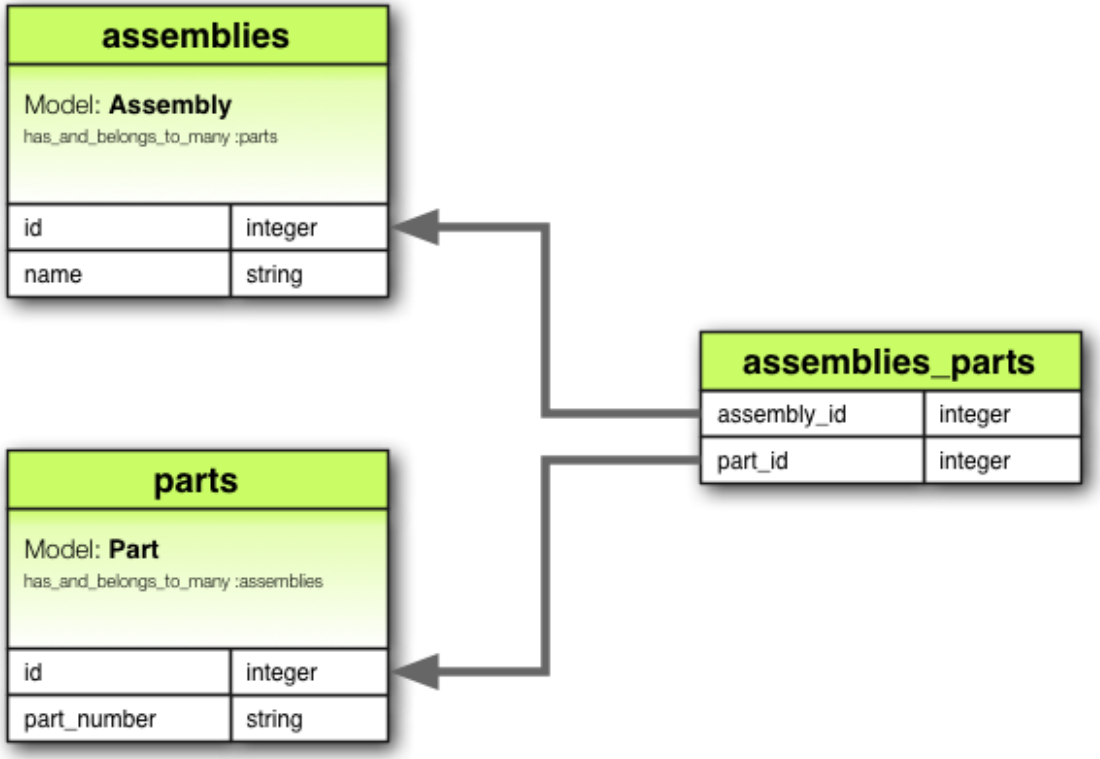



```
class Supplier < ActiveRecord::Base
  has_one :account
  has_one :account_history, :through => :account
end

class Account < ActiveRecord::Base
  belongs_to :supplier
  has_one :account_history
end

class AccountHistory < ActiveRecord::Base
  belongs_to :account
end
```

The has_and_belongs_to_many Association




```
class Assembly < ActiveRecord::Base
  has_and_belongs_to_many :parts
end

class Part < ActiveRecord::Base
  has_and_belongs_to_many :assemblies
end
```

Active Record Query Interface

```
1  class Client < ActiveRecord::Base
2    has_one :address
3    has_many :orders
4    has_and_belongs_to_many :roles
5  end
6
7  class Address < ActiveRecord::Base
8    belongs_to :client
9  end
10
11 class Order < ActiveRecord::Base
12   belongs_to :client, :counter_cache => true
13 end
14
15 class Role < ActiveRecord::Base
16   has_and_belongs_to_many :clients
17 end
```

Active Record finder methods

- where
- select
- group
- order
- limit
- offset
- joins
- includes
- lock
- readonly
- from
- having

Retrieving objects

Retrieving a Single Object

```
1  client = Client.find(10)
2  client = Client.first
3  client = Client.last
```

Retrieving Multiple Objects

```
1  client = Client.find(1, 10)
2
3  User.find_each do |user|
4    NewsLetter.weekly_deliver(user)
5  end
```

Saving the object to the database

- create, create!
- save, save!
- update
- update_attributes, update_attributes!