Addison-Wesley Professional Ruby Series

THE MERB WAY

Foreword by **Obie Fernandez,** *Series Editor*

FOY SAVAS

Library of Congress Cataloging-in-Publication Data
Savas, Foy.
The Merb way / Foy Savas.
p. cm.
ISBN 978-0-321-60638-9 (pbk. : alk. paper)
1. Web site development. 2. Merb (Electronic resource) I. Title.

TK5105.888.S2775 2009
006.7'6—dc22 2009013263

ISBN-13: 978-0-321-60638-9
ISBN-10: 0-321-60638-8

Text printed in the United States on recycled paper at Donnelley in Crawfordsville, Indiana.
First printing, June 2009

Editor-in-Chief
Mark Taub

Acquisitions Editor
Debra Williams Cauley

Development Editor
Michael Thurston

Managing Editor
John Fuller

Project Editor
Anna Popick

Copy Editor
Barbara Wood

Indexer
Richard Evans

Proofreader
Barbara Wood

Technical Reviewers
Matthew Knox
Jen Lindner

Editorial Assistant
Kim Boedigheimer

Cover Designer
Chuti Prasertsith

Compositor
ITC

Foreword

I viewed the Merb project with a big dollop of suspicion almost from the very start. First of all, it started out as a really simple concept, but got a lot more complicated as additional contributors got involved. And whether my suspicion was misplaced or not, I couldn't shake the feeling that the Merb people were insisting on reinventing Ruby on Rails simply to be difficult.

Merb isn't *that* different from Rails, nor is it a major improvement as far as I was concerned. It was only *slightly* better (in some regards), and only *slightly* different from Ruby on Rails. Sure there were some benchmarks showing big performance gains for using Merb, but I was not convinced enough to switch. And I couldn't bear to use Merb and Rails concurrently. You see, they were very similar, yet different in subtle ways. The last thing I want to do when I'm programming on hard deadlines is to slow down to remember the differences between Merb's `render` method and the one in Rails, etc.

Life continued, and while I was very happy and making a ton of money working with and writing about Rails, I still occasionally looked at Merb, if only to see what they were up to. Sure enough, by 2008 my perception of Merb had started to change. I met Foy Savas and he gave me the gist of why it would make sense to write *The Merb Way.* I concurred and Foy got busy writing.

As Merb approached a 1.0 release, it appeared that Yehuda Katz and his band of merry discontents were actually achieving a viable platform, one that would attain permanent status as an alternative to Rails. I started hearing about significant production projects running on Merb, such as yellowpages.com. My appreciation for Merb grew, as I got familiar with the project goals and noticed how the competition was acting as a healthy stimulus for Rails to not rest on its laurels. I wasn't ready to switch to Merb myself, since my key criterion is maturity, which Rails has in spades, but no longer was I suspicious.

In November of 2008 I was extremely happy about Foy being almost finished writing the first manuscript of this book, knowing that it would be a good complement to the Professional Ruby Series and a good companion for *The Rails Way*. Foy delivered an awesome conference presentation in Boston where he covered some of the key concepts from the book in an engaging and entertaining manner—the crowd ate it up and I knew *The Merb Way* would be a winner.

It wasn't too many weeks later that the unthinkable happened. After an eruption of feuding between members of the Rails and Merb teams, some secret meetings occurred and shocking news was unveiled right before Christmas: DHH and Yehuda happily proclaimed that the core teams of both frameworks were merging and that within a year, the Merb codebase would be merged with Rails in order to produce Rails Version 3, after which development of Merb would discontinue.

The million-dollar question (okay, not quite that much) for Foy and myself as the Series Editor was whether to continue with the publication of *The Merb Way*. After all, if the Merb framework was going away, then what was the point?

After letting the matter settle for a while, we arrived at our answer, which you already know since you are holding this book. It turns out that learning about Merb is valuable in a number of important ways. First, Merb is still used rather widely and we suspect that its lifetime will exceed what the current core team has in mind. Second, Foy is a gifted writer and his descriptions of the philosophies that impacted the design and implementation of Merb are definitely useful to everyone who will be affected by the changes that will happen in Rails 3.

It's with great pleasure that I welcome the incomparable Foy Savas and publish *The Merb Way* as a full-fledged and proud installment of the Professional Ruby Series. I sincerely hope you get as much out of this book as I have.

Obie Fernandez
March 31, 2009

Introduction

道 可 道 非 常 道
名 可 名 非 常 名
—老子

A way that can be taken rarely stays the way.
A name that can be given rarely stays the name.
—Laozi
(translated by your author)

The first two lines of the *Dao De Jing* capture truths so fundamental that we find them everywhere—including, without exception, in this book. Originally entitled the *The Merb Companion*, it was intended to be the advanced practitioner's bible. But the little framework named Merb grew quickly, so much so that the Rails world took notice, and before long, the core teams came together and decided to merge the two. Yet despite Merb's fundamentals being seen as the basis for the features of the future, Rails conventions needed to be maintained long-term. Incidentally, this book's content was left in an awkward limbo. Should we wait for the completion of the merge or perhaps publish immediately? Should we preserve the chapters we had written so far or produce them anew after the merge? Soon, however, an epiphany came. Though the book had material highly relevant to the Merb developers of the day and was equally suitable for bringing foresight to the Rails developers of tomorrow, its greatest and lasting potential was in shedding light on how Merb, the framework that even Rails envied, had been designed. Thus, we arrive here, at the introduction of *The Merb Way*.

Born a pastie

Zoom back to September 9, 2006. If you were using Rails, it was version 1.1. As for Git, it was still just something the Linux kernel used. So naturally, when Ezra Zygmuntowicz needed to publicly distribute the then only 120-line source code for Merb, a pastie made the most sense. Turn back a few pages and you'll find that pastie, prepended to this book in all of its glory. Look it over once and you'll realize Ezra had an itch to scratch: fast Ruby template rendering with the smallest possible memory footprint. To do this he used only two gems, Mongrel and ERB, which, for those keen on etymology, also serve as the roots of the name Merb itself. But like any project that starts off both small and practical, it grew. A few months later a gem was put up on RubyForge. For a while it came with an example app that touted Merb's ability to handle file uploads while not locking out other requests. This became the most popular use of Merb, often appearing coupled with Rails applications in the wild.

Created by rebels

But Merb has moved far beyond the stage when its application generator could produce the `uploader` application. This may have been because it quickly became a breeding ground in which Rails concepts could be rethought. Eventually, this led to a shared vision among developers that Merb would be Rails done right. Thankfully, the openly opinionated nature of Rails actually kept them away from simply adopting alternative opinions. Instead, they chose as one of their maxims agnosticism, in the form of opt-in modularity. This allowed Merb application developers to work with whatever tools they needed for their projects. Merb's versatility meant that it would become the underlying framework for numerous customized stacks that otherwise did not easily fit within the opinions of Rails. Its developers, consequently, have often been thought of as rebels from the Rails mentality, since many of them maintain strong opposing views on particular Rails opinions.

However, that's not all there is to the Merb development story. The crowd of developers also had an arguably more scientific bent, aiming to make the best selection of the methods they employed through microbenchmarks. Ruby is often cited as one of the world's slowest programming languages from a computational perspective, but to the developers of Merb, this didn't mean that microseconds didn't matter. After all, web development frameworks are often tested in terms of responses per second. So when it came to Merb internals, whether it was in route matching or filter chaining, only the fastest Ruby constructs were used. The end result was Merb blazing past Rails in response time benchmarks. More profoundly, Merb raised the bar for Ruby web framework performance in general, finally allowing Ruby to win when pitted against other frameworks from different languages.

Interestingly enough, this combination of performance and modularity is also indicative of another Merb development character trait: its desire to solve higher-level problems. Whereas Rails features have nearly consistently been born of necessity, Merb's focus has been on abstract goals like versatility and design. This has pushed the limits of what can be done with Merb, resulting in advancements like the abstract controller class, code slices, and arbitrary application layout. In sum, Merb in many ways conceptually broke the "frame" in framework by thinking of itself less as a way to produce Model-View-Controller (MVC) web apps and more as a highly modular platform upon which nearly all Ruby web development, no matter how unusual, can occur.

The future of Rails

Put all this goodness together, and you're bound to attract the attention of most developers. Not surprisingly, teams large and small began using Merb as a full application framework in production environments for their most demanding applications. This finally put Merb to the test, and the results were completely in line with its benchmarks: more responses per second with a solidly smaller memory footprint. All of this occurred at the cusp of its reaching its 1.0 release on November 7, 2008.

Quickly the Merb development team oriented itself toward larger goals through what would have been a 2.0 release. Objectives included, among other things, an abstracted Object Relational Mapper (ORM) that could be used to create an agnostic admin panel and convenient direct routing to templates. But with the stable release already out, developers at large began taking Merb much more seriously, and soon online comment wars fanned flames between the Rails and Merb core teams. Thus prompted to confront each other, the core development teams began to talk it over. However, soon they began earnestly asking themselves whether the large amount of duplication between the two frameworks was warranted. At the end they concluded two things: first, that each side had aspects that would be beneficial to the other, and second, that yes, they could work together. Announced officially on December 23, 2008, the Rails and Merb development effort would be merged, with a promise of a smooth transition from both Merb 1 and Rails 2 to the combined Rails 3.

So what can we expect from this merger? Well, above all will be the fusion of attitudes and objectives from both sides. The Rails of the future will undoubtedly appeal to developers in both pragmatic and idealistic ways. Merb will bring speed, performance, and modularity along with a tempering of the preestablished Rails opinions. We will see a flowering of custom Rails stacks, some of which may eventually push the limits of the Rails framework itself. Toward what—who knows? But with the possibility of such a greatly expanded domain, two things are certain. The first is that anyone in software who needs to build an agile and sophisticated web application upon a high-performance framework should definitely get on this train. The second, which applies more to developers themselves, is that a firm understanding of the design decisions behind Merb's development will not only prepare them for the future of Rails but can also open up their own talents through the elegance of its code.

This book is thus focused on giving developers the deepest possible understanding of Merb itself. While you may use it as a reference guide in the development of Merb applications (which we strongly recommend without hesitance until the release of Rails 3), you may also appreciate its guided exploration of the Merb source. That said, for those afraid of code, it may perhaps be a good time to put this book back on the shelf. Everyone else, be warned; we're going to cut deep into the framework itself, revealing not only what was done, but what can be learned from it. As you move forward, do not lose focus. After all, they say that one should always aim to learn from the best, and through Merb, the framework that without exaggeration brought the Rails monoculture to its knees, you are doing just that.

CHAPTER 5
Models

Models are related data and algorithms that respectively represent the properties and behaviors of domain objects used within an application. Models are used in nearly every decently complex object-oriented application because they organize application data in object form. Within the scope of a typical Merb application, examples of modeled objects may include users, posts, entries, or comments. These objects are most often persisted in a database through the use of an Object Relational Mapper, or ORM. As agnosticism has been central to Merb's design, application developers are free to integrate whatever Ruby ORM they may need. Nonetheless, the default ORM included as part of the standard Merb stack is DataMapper, an ActiveRecord-like ORM that aims to push the boundaries of object-data interaction even more significantly toward the object side.

Given its acceptance as part of the standard Merb stack, we will cover the use of only DataMapper in this chapter. However, don't let that stop you from using ActiveRecord, Sequel, or any of the other options, since each of these ORMs is capable of creating, retrieving, updating, and deleting persisted data and is fully supported by the Merb core.

5.1 Configuration

Prior to being able to use the DataMapper ORM within your Merb application, you will have to make sure that you have done three things:

- Included the DataMapper dependencies within `config/dependencies.rb`
- Selected DataMapper as the ORM in `init.rb`
- Configured your database connection in `config/database.yml`

However, if you used `merb-gen` to generate a standard application and have SQLite3 installed, then you'll find that all of these are already covered for you. If your application was generated in some other manner, you may want to see Chapter 1 to figure out how your configuration files should be laid out.

Other standard dependencies

The standard Merb generator inserts several other DataMapper dependencies into the file `config/dependencies.rb`. These commonly used DataMapper plugins provide functionality such as migrations, automatic timestamping, validations, and record aggregation. You are free to comment out or remove any of these dependencies, but as we'll see through examples, even the most basic application models tend to want them to be available.

DataMapper works with several different database back ends. Support for MySQL, PostgreSQL, and SQLite3 are all included via the DataMapper core. If you're using another database application, you may be able to find an appropriate adapter in the `dm-more` or `dm-adapters` gems. Alternatively, if your computer has the memory for it, there's a DataMapper adapter that will store your model records completely in memory.

Not just databases

When you use DataMapper, you aren't limited to interacting only with conventional databases. Instead, the DataMapper concept of a repository applies equally well to just about any data source composed of records. This includes numerous web service APIs where CRUD-like operations are available. A great example of such an adapter is the Salesforce adapter Yehuda Katz has written. In just under 300 lines of code, the adapter provides full DataMapper interaction via the Salesforce API. You'll find that these adapters can also be used to augment models through secondary repositories, which is commonly the case when using Bernerd Schaefer's full-text-search Ferret adapter.

Repository configuration is done in a YAML file named `config/database.yml`. The standardly generated `config/database.yml` file is broken into four environments—development, test, production, and rake—corresponding to the environments that are standardly used by a Merb application. If you have another environment—for instance, a staging environment—it, too, can be listed in this file.

Typically there is overlap in the configuration of environments, so YAML node anchors and aliases are used to reduce redundancy.

What's a YAML node?

YAML nodes are named blocks of content in a YAML file. You can set up an anchor to the content of a node by using an ampersand and word after the naming of the node. Below we connect the content of `development` with the anchor `&default`.

```
development: &default
  adapter: mysql
  database: example_development
```

We can reference this anchor by using the alias `*default` later on. If we use this alias as the value on a merge key, `<<`, it will include the previously defined content. Here we merge in the defaults from before and then override the value for the database:

```
test:
  <<: *default
  database: example_test
```

5.2 Model classes

Model classes exist in the directory `app/models`. These classes typically have singular names and exist one class per file. Examples that may appear in a Merb blog application are `Post` in the file `app/models/post.rb` and `FavoriteLink` in `app/models/favorite_link.rb`. Organizing model classes in this way isn't strictly necessary, though, since Merb by default includes everything inside `app/models` by recursive glob. In other words, so long as you make sure that your model class is somewhere in `app/models`, it will be available at boot. Nonetheless, our recommendation is that you do arrange your model classes in a reasonable way, so that both you and others can easily find them.

While various DataMapper modules can incorporate different functionality within your model class, the fundamental module that must be included in order for your class to work with DataMapper is `DataMapper::Resource`. Below we include this module and effectively set up our `User` class as a DataMapper model.

```
class User
  include DataMapper::Resource
end
```

Design decision: resource module versus base class

DataMapper model classes are not created through inheritance from an abstract base class. If you've used ActiveRecord, which does do so, you may wonder why this alternate design decision has been made. Interestingly enough, early versions of DataMapper did actually use a parent class called `DataMapper::Base`. With time, however, the developers of DataMapper grew concerned that this was conceptually coupling application logic with DataMapper's own. In other words, application models are more reasonably thought of as being enhanced through database persistence rather than as stemming off a library's base class that provides such functionality.

In some ways, this design decision can be seen as an application of the Principle of Substitutability, which insists that subtypes are capable of replacing their parents without affecting the correctness of the program. While this principle doesn't exactly translate well when using Ruby (proving correctness through types isn't anywhere near Ruby's slant), we can learn from the principle by recognizing that inheritance is best used when a subclass makes no behavioral constraining modifications on the parent.

Let's take a look at the source behind the `Resource` module to get a feel for how it affects the class in which it is included:

```
module DataMapper
  module Resource

    # ...

    # @api public
    def self.append_inclusions(*inclusions)
      extra_inclusions.concat inclusions
      true
    end

    def self.extra_inclusions
      @extra_inclusions ||= []
    end

    # When Resource is included in a class this
    # method makes sure it gets all the methods
    #
    # -
    # @api private
    def self.included(model)
      model.extend Model
      model.extend ClassMethods
      if defined?(ClassMethods)
    model.const_set('Resource', self)
      unless model.const_defined?('Resource')
    extra_inclusions.each { |inclusion|
      model.send(:include, inclusion) }
    descendants << model
    class << model
      @_valid_model = false
      attr_reader :_valid_model
    end
  end

  # ...
 end
end
```

Focusing on the last method listed above, we can see that the `Resource` module extends the class in which it is included (above called `model`). In the process, it principally extends it with the `Model` module, which contains the logic for property persistence and object retrieval. As we come to these methods later on, we'll open up the `Model` class source as well. For now, however, take note of `Resource`'s ability to include extra modules through the class method `append_inclusions`. This method is used extensively by DataMapper plugins that need to extend the functionality of all DataMapper models. For instance, below is some of the source for `dm-timestamps`, which automatically sets timestamps on properties of particular names.

```
module DataMapper
 module Timestamp
  Resource.append_inclusions self

  # ...

  def self.included(model)
    model.before :create, :set_timestamps
    model.before :update, :set_timestamps
    model.extend ClassMethods
  end

 end
end
```

The first line appends the module to all DataMapper models. Later on, using its own `self.included`, the module includes its own logic into the model. This cascading of modules is thus particularly effective for code as modular as DataMapper.

5.3 Properties

Each DataMapper model is able to persist its data. The kind of data it is able to store is defined through its properties. If you're using a typical database, these properties correlate with the columns of the model's corresponding table. Below is an example of a DataMapper model with three properties.

```
class TastyAnimal
  include DataMapper::Resource

  property :id, Serial
  property :name, String
  property :endangered, TrueClass

end
```

In many ways, you can think of properties as persistent accessors. In fact, taking a look into the source of the property method (found in the `Model` resource we spoke about earlier), we find that a dynamic getter and setter are created using `class_eval`:

```
def property(name, type, options = {})
  property = Property.new(self, name, type, options)

  create_property_getter(property)
  create_property_setter(property)

  # ...
end

# ...

# defines the getter for the property
def create_property_getter(property)
  class_eval <<-EOS, _ _FILE_ _, _ _LINE_ _
    #{property.reader_visibility}
    def #{property.getter}
      attribute_get(#{property.name.inspect})
    end
  EOS

  # ...

end

# defines the setter for the property
def create_property_setter(property)
  unless instance_methods.include?("#{property.name}=")
```

```
    class_eval <<-EOS, __FILE__, __LINE__
      #{property.writer_visibility}
      def #{property.name}=(value)
        attribute_set(#{property.name.inspect}, value)
      end
    EOS
  end
end
```

The most important thing to learn from the source shown above is that properties dynamically create getter and setter methods. Additionally, these methods can end up protected or private through visibility attributes. Finally, the getters and setters produced are not exactly equivalent to `attr_reader` and `attr_writer` because of their internal use of the methods `attribute_get` and `attribute_set`.

Going back to the `Resource` source, we can find these two methods manipulating the values of model properties, once again located in `Model`. You'll have to excuse this volleying back and forth, but the point of the `Resource` and `Model` modules is to separate individual resource methods from those related to the model as a whole.

```
# @api semiplugin
def attribute_get(name)
  properties[name].get(self)
end

# @api semipublic
def attribute_set(name, value)
  properties[name].set(self, value)
end

protected

def properties
  model.properties(repository.name)
end
```

You may have noticed the `@api semipublic` comment above the getter and setter methods. This is because application developers should not ordinarily need to use these methods. Plugin developers, on the other hand, may need to use them as the easiest way to get and set properties while making sure they are persisted.

For application developers, however, this does bring up one important point: Do not use instance variables to set property values. The reason is that while this will set the object's value, it will unfortunately short-circuit the model code that is used to track whether a property is dirty. In other words, the property value may not persist later upon save. Instead, you should use the actual property method. Below you'll find an example with comments that should get the point across.

```
class Fruit
 include DataMapper::Resource

 property :id, Serial
 property :name, String
 property :eaten, TrueClass

 def eat
  unless eaten?
   # will not persist upon save
   @eaten = true

   # will persist upon save
   eaten = true
  end
 end

end
```

Before we describe the extended use of properties, let's take a look at the database side to understand how persistence works.

5.3.1 Database storage

In order to persist the data of model objects, we need to set up our database for that data to be stored. The default-generated configuration files use a SQLite3 database file called `sample_development.db`. This setup is perfect for most development scenarios given its quickness to get up and running. With that in mind, we'd say stick with it whenever possible, leaving the alteration of `config/database.yml` for production or staging environments.

Create your database with `rake`

If you are using MySQL or PostgreSQL for development, though, it may be useful to know that there is a Merb DataMapper rake task that can create your database for you. After making sure your `config/database.yml` file contains the appropriate username and password, `rake db:create` issues the database admin command you may have otherwise forgotten.

5.3.1.1 Automigrating the DB schema

Databases typically need to be prepped for the data they will store during application development. The process by which DataMapper does this is called **automigration**, because DataMapper uses the properties listed in your models to automatically create your database schema for you. Using the provided Merb DataMapper rake task, we can automigrate the model that we created earlier and then take a peek inside the database to see what was done:

```
$ rake db:automigrate
$ sqlite3 sample_development.db
sqlite> .tables
tasty_animals
sqlite> .schema
CREATE TABLE "tasty_animals" ("id" INTEGER NOT NULL
  PRIMARY KEY AUTOINCREMENT, "name" VARCHAR(50), "
  is_endangered" BOOLEAN);
```

As you can see, a table with a pluralized and snake-cased name was created for our model, `TastyAnimal`. Remembering the various properties of the model class, we can also spot corresponding columns inside the schema's `CREATE` statement. Note that while Ruby classes were used on the property lines, standard SQL types appear in the database.

Design decision: emphasize the model as an object

DataMapper has notably chosen to include properties within the model files. Those familiar with ActiveRecord may find this odd, having been accustomed to distributing schema information across migration files. There are, however, a number of benefits to the DataMapper way of handling properties, including

- An explicitness that helps developers understand how a model is used
- An ability to work around legacy systems far more easily due to the lack of schema dependence
- The possibility of distributing model property persistence across multiple databases or repositories
- The time-saving nature of automigrations

These benefits may take a while to appreciate given their subtleness, but there is a more general paradigm shift going on that we should mention: The inclusion and use of properties in DataMapper model classes essentially pushes the ORM concept even further object-side, where the resulting combination of less magic and less maintenance is a total win.

The code behind automigration is definitely worth studying, so let's take a look at the module `AutoMigrations`, which includes itself within the `Model` module:

```
module DataMapper
  module AutoMigrations
    def auto_migrate!(repository_name =
      self.repository_name)

      auto_migrate_down!(repository_name)
      auto_migrate_up!(repository_name)
    end

    # @api private
    def auto_migrate_down!(repository_name =
      self.repository_name)

     # repository_name ||= default_repository_name
      repository(repository_name) do |r|
        r.adapter.destroy_model_storage(r,
          self.base_model)
      end
    end

    # @api private
    def auto_migrate_up!(repository_name =
      self.repository_name)

      repository(repository_name) do |r|
        r.adapter.create_model_storage(r,
          self.base_model)
      end
    end

    def auto_upgrade!(repository_name =
      self.repository_name)

      repository(repository_name) do |r|
        r.adapter.upgrade_model_storage(r, self)
      end
    end

    Model.send(:include, self)

  end # module AutoMigrations
end # module DataMapper
```

As you can see, there are two API public class methods you can use with models, `auto_migrate!` and `auto_upgrade!`. These effectively call the three adapter methods `destroy_model_storage`, `create_model_storage`, and `upgrade_model_storage`. Let's go deep into the source and see how these three methods do the heavy lifting:

```
class DataMapper::Adapters::AbstractAdapter
 module Migration

   def upgrade_model_storage(repository, model)
     table_name = model.storage_name(repository.name)

     if success = create_model_storage(repository,
       model)

       return model.properties(repository.name)
     end

     properties = []

     model.properties(repository.name).
       each do |property|

       schema_hash = property_schema_hash(repository,
         property)

       next if field_exists?(table_name,
         schema_hash[:name])

       statement = alter_table_add_column_statement(
         table_name, schema_hash)

       execute(statement)
       properties << property
     end

     properties
   end

   def create_model_storage(repository, model)
     return false if storage_exists?(
       model.storage_name(repository.name))

     execute(create_table_statement(repository, model))
```

```
    # ... create indexes

    true
  end

  def destroy_model_storage(repository, model)
    execute(drop_table_statement(repository, model))
    true
  end

 end
end
```

The simplest of these, `destroy_model_storage`, executes a drop table statement. The `create_model_storage` method, on the other hand, first checks to see if the model storage already exists, returning false if it does or true if it does not, and consequently has the chance to create it. Finally, `upgrade_model_storage` is the most complicated of the three. It first attempts to create the storage (effectively testing whether it exists or not) and then attempts to add new columns for new properties. This leaves existing data in place and is perfect if you have simply added properties to a column. Lest this appear to be no more than hand waving, let's dig even deeper into the methods that the `AbstractAdapter` uses to create the SQL for these statements:

```
class DataMapper::Adapters::AbstractAdapter

  # immediately following the previous code

  module SQL
    private

    def alter_table_add_column_statement(table_name,
      schema_hash)

      "ALTER TABLE "+
      quote_table_name(table_name)+
      "ADD COLUMN "+
      property_schema_statement(schema_hash)
    end

    def create_table_statement(repository, model)
    repository_name = repository.name

      statement = <<-EOS.compress_lines
        CREATE TABLE
        #{quote_table_name(
          model.storage_name(repository_name))}

        (#{model.properties_with_subclasses(
          repository_name).map { |p|

          property_schema_statement(
            property_schema_hash(repository, p))

          } * ', '}
      EOS

      if (key = model.key(repository_name)).any?
        statement << ", PRIMARY KEY(#{ key.map { |p|
          quote_column_name(p.field(repository_name))
        } * ', '})"
      end

      statement << ')'
      statement
    end

    def drop_table_statement(repository, model)
      "DROP TABLE IF EXISTS "+
      quote_table_name(model.storage_name(
        repository.name))
    end

    def property_schema_hash(repository, property)
      schema = self.class.type_map[property.type].
        merge(:name => property.field(repository.name))

      if property.primitive == String &&
      schema[:primitive] != 'TEXT'
        schema[:size] = property.length
      elsif property.primitive == BigDecimal ||
      property.primitive == Float
        schema[:precision] = property.precision
        schema[:scale]     = property.scale
      end

      schema[:nullable?] = property.nullable?
      schema[:serial?]   = property.serial?

      if property.default.nil? ||
      property.default.respond_to?(:call)
        schema.delete(:default)
          unless property.nullable?
      else
        if property.type.respond_to?(:dump)
          schema[:default] = property.type.dump(
            property.default, property)
        else
          schema[:default] = property.default
        end
      end

      schema
    end

    def property_schema_statement(schema)
      statement = quote_column_name(schema[:name])
      statement << " #{schema[:primitive]}"

      if schema[:precision] && schema[:scale]
        statement << "(#{[ :precision, :scale ].map {
          |k| quote_column_value(schema[k])
        } * ','})"
      elsif schema[:size]
        statement << "("+
          quote_column_value(schema[:size])}+")"
      end

      statement << ' NOT NULL'
        unless schema[:nullable?]
      statement << " DEFAULT " +
        quote_column_value(schema[:default]) if
          schema.has_key?(:default)
      statement
    end
  end
  include SQL

end
```

The first thing you may notice is that the methods are included within a module called `SQL` and that the module is included immediately after it is closed. The reason for this is that within DataMapper adapters, code is often organized by use, and thus the encapsulation of private methods into a module easily allows for alternating regions of public and then private methods.

Now, turning to the actual methods, we can see that some of them—for instance, `drop_table_statement`—are just a line of simple SQL. Likewise, `alter_table_column_statement` is just a single line that outputs add column statements.

The `create_table_statement`, however, is far more complex, relying on various other methods to get its work done. One of these, `properties_with_subclasses`, pulls up all model properties, including those that are simply keys used with relationships. We'll go further into `properties_with_subclasses` later on when we examine model relationships, but for now let's take a look at the method `property_schema_statement`, which quotes the property as a column name and then appends its type. It also adds the appropriate SQL for decimals, non-nullables, and default values.

We hope this has brought you deep enough into the inner workings of automigration to both appreciate its design and get a feel for how adapter code handles the production of SQL more generally. But it would also be nice to be able to use some of it practically, and thankfully you can do so. For instance, if you're in mid-development, you may fire up interactive Merb and use `auto_upgrade!` on a model to which you've added properties:

```
> Fruit.auto_upgrade!
```

Likewise, you may want to refresh the data of a model using `auto_migrate!` in the middle of a test file. Here's an example we've spotted in the wild:

```
before :each do
  Invite.auto_migrate!
end
```

5.3.2 Defining properties

Let's now take a more rigorous look at properties as well as the options we have while defining them. As we've seen, each property is defined on its own line by using the method `property`. This class method is mixed in via the inclusion of `DataMapper::Resource`. It takes a minimum of two arguments, the first being a symbol that effectively names the property and the second being a class that defines what type of data is to be stored. As we will see soon, an optional hash of arguments may also be passed in.

5.3.2.1 Property types

While abstracting away the differences across database column types, DataMapper has chosen to stay true as much as possible to using Ruby to describe properties types. Below is a list of the various classes supported by the DataMapper core. Note that the inclusion of `DataMapper::Resource` will include `DM` in your model class, and that when defining properties, you will not have to use the module prefix `DM::` before those that use it.

- `Class`—stores a Ruby `Class` name as a string. Intended for use with inheritance, primarily through the property type `DM::Discriminator`.
- `String`—stores a Ruby `String`. Default maximum length is 50 characters. Length can be defined by the optional hash key `:length`.
- `Integer`—stores a Ruby `Integer`. Length can be defined by the optional hash key `:length`.
- `BigDecimal`—stores a Ruby `BigDecimal`, intended for numbers where decimal exactitude is necessary. Can use the option hash keys `:precision` and `:scale`.
- `Float`—stores a Ruby `Float`. Primarily intended for numbers where decimal exactitude is not critical. Can use the two options hash keys `:precision` and `:scale`.
- `Date`—stores a Ruby `Date`.
- `DateTime`—stores a Ruby `DateTime`.
- `Time`—stores a Ruby `Time`.
- `Object`—allows for the marshaling of a full object into a record. It is serialized into text upon storage and when retrieved is available as the original object.
- `TrueClass`—a Boolean that works with any of the values in the array `[0, 1, 't', 'f', true, false]`. In MySQL it translates down to a tinyint, in PostgreSQL a bool, and in SQLite a boolean.
- `DM::Boolean`—an alias of `TrueClass`. This is around for legacy DataMapper support, simply to provide a more commonly recognized name for the type.
- `Discriminator`—stores the model class name as a string. Used for single-table inheritance.
- `DM::Serial`—used on the serial ID of a model. Serial IDs are auto-incremented integers that uniquely apply to single records. Alternatively, a property can use the `Integer` class and set `:serial` to true. You will nearly always see this type applied to the `id` property.
- `DM::Text`—stores larger textual data and is notably lazy-loaded by default.

You may be interested in knowing how the casting in and out of property values works. For the primitive types, values coming out of the database are cast using the method `Property#typecast`. Below we see how this methods prunes results, modifying them into what we want in Ruby.

```
def typecast(value)
  return type.typecast(value, self) if type.respond_to?(:typecast)
  return value if value.kind_of?(primitive) || value.nil?
  begin
    if    primitive == TrueClass
      %w[ true 1 t ].include?(value.to_s.downcase)
    elsif primitive == String
      value.to_s
    elsif primitive == Float
      value.to_f
    elsif primitive == Integer
      value_to_i = value.to_i
      if value_to_i == 0
        value.to_s =~ /^(0x|0b)?0+/ ? 0 : nil
      else
        value_to_i
      end
    elsif primitive == BigDecimal
      BigDecimal(value.to_s)
    elsif primitive == DateTime
      typecast_to_datetime(value)
    elsif primitive == Date
      typecast_to_date(value)
    elsif primitive == Time
      typecast_to_time(value)
    elsif primitive == Class
      self.class.find_const(value)
    else
      value
    end
  rescue
    value
  end
end
```

Custom types, however, are handled by subclasses of an abstract type class called `DataMapper::Type`. These load and dump data in whatever way they are programmed to do. We'll see custom types later on when we examine some DataMapper-type plugins, but for now let's take a look at one of the custom types from the DataMapper core, `Serial`:

```
module DataMapper
  module Types
    class Serial < DataMapper::Type
      primitive Integer
    serial true
  end # class Text
 end # module Types
end # module DataMapper
```

Note its use of the methods `primitive` and `serial`, which are defined in the class `DataMapper::Type`:

```
class DataMapper:Type
 PROPERTY_OPTIONS = [
  :accessor, :reader, :writer,
  :lazy, :default, :nullable, :key, :serial, :field,
  :size, :length, :format, :index, :unique_index,
  :check, :ordinal, :auto_validation, :validates,
  :unique, :track, :precision, :scale
 ]

 # ...

 class << self

  PROPERTY_OPTIONS.each do |property_option|
   self.class_eval <<-EOS, _ _FILE_ _, _ _LINE_ _
    def #{property_option}(arg = nil)
     return @#{property_option} if arg.nil?

     @#{property_option} = arg
    end
   EOS
  end

  def primitive(primitive = nil)
   return @primitive if primitive.nil?
   @primitive = primitive
  end

  # ...

 end
end
```

From this we can first see that the `primitive` method sets the type to which the property value should be dumped. The `serial` method, on the other hand, is an example of the property option, which we're about to address.

5.3.2.2 Option hash

The third argument that the `property` method can take is an option hash, which affects various behavioral aspects of the property. For instance, below we've specified that a property should default to some value.

```
class Website
  include DataMapper::Resource

  property :id, Serial
  property :domain, String
  property :color_scheme, String, :default => 'blue'
end
```

Here's a list of the various property options and their uses:

- `:accessor`—takes the value `:private`, `:protected`, or `:public`. Sets the access privileges of the property as both a reader and a writer. Defaults to `:public`.
- `:reader`—takes the value `:private`, `:protected`, or `:public`. Sets the access privileges of the property as a reader. Defaults to `:public`.
- `:writer`—takes the value `:private`, `:protected`, or `:public`. Sets the access privileges of the property as a writer. Defaults to `:public`.
- `:lazy`—determines whether the property should be lazy-loaded or not. Lazy-loaded properties are not read from the repository unless they are used. Defaults to false on most properties, but is notably true on `DM::Text`.
- `:default`—sets the default value of the property. Can take any value appropriate for the type.
- `:nullable`—if set to true it will disallow a null value for the property. When `dm-validations` is used this invalidates a model.
- `:key`—defines a property as the table key. This allows for natural keys in place of a serial ID. This key can be used as the index on the model class in order to access the record.
- `:serial`—sets the property to be auto-incremented as well as to serve as the table key.
- `:field`—manually overrides the field name. Best used for legacy repositories.
- `:size`—sets the size of the property type.
- `:length`—alias of `:size`.

- `:format`—used with the `String` property type. When used with a `dm-validations` format can set a regular expression against which strings must validate.
- `:index`—sets the property to be indexed for faster retrieval. If set to a symbol instead of to true, it can be used to create multicolumn indexes.
- `:unique_index`—defines a unique index for the property. When used with `dm-validations`, new records with nonunique property values are marked invalid. If set to a symbol instead of true, it can be used to create multicolumn indexes.
- `:auto_validation`—when used with `dm-validations`, can be used to turn off autovalidations by using the value true.
- `:track`—determines when a property should be tracked for dirtiness. Takes the values `:get`, `:set`, `:load`, and `:hash`.
- `:precision`—sets the number of decimal places allowed for `BigDecimal` and `Float` type properties.
- `:scale`—sets the number of decimal places after the decimal point for `BigDecimal` and `Float` type properties.

5.4 Associations

DataMapper also supports the defining of associations between models. If you've used ActiveRecord before, these are nearly the same. Otherwise, know that associations allow you to define the relationships between models (one-to-one, one-to-many, etc.), automatically creating database keys where necessary while also making it possible to conveniently pull up related model objects from associates. Let's survey the various relationships possible with DataMapper.

Design decision: generalized association methods

The largest distinction between DataMapper and ActiveRecord associations is DataMapper's use of only two methods, `has` and `belongs_to`. This differs from ActiveRecord's one method per association type and has benefited DataMapper through simplicity of both internal design and interface. We'll see the elegance of the internal design later on, but know that these two generalizations allow application developers to simplify the cognitive steps needed to arrive at the appropriate associations. That is, to determine if an association should use `has` or `belongs_to`, you only need to ask whether a related model key should maintain a property for the ID of the other model. Moving from here, you can then define the cardinality and pathway of the relationship if necessary. This may not seem like anything special to an experienced developer, but we've found that this easing of an interface's learning curve tends to be indicative of cleaner design, making it a win-win situation for everyone.

5.4.1 Belongs to

In general, you should know that a belongs-to association is meant to help you quickly retrieve an associated resource by defining a one-to-something association between two models (specifically, a child and its parent), where the child class should store its parent's key as a property. So, for instance, the following associates a comment with a user:

```
class Comment
  include DataMapper::Resource

  property :id, Serial
  property :body, Text

  belongs_to :user
end
```

Note that upon automigration the belongs-to association automatically creates the column `user_id` within the comments table. This means that any model object now has two new methods accessible, `user_id` and `user`. The first is simply the ID of the associated user, but the second actually retrieves the user resource for you. These defaults may not always fit your domain logic, however, so they can be altered.

```
class Paper
  include DataMapper::Resource

  property :id, Serial
  property :body, Text

  belongs_to :author, :class_name => "User",
    :child_key => [:author_id]

  belongs_to :reviewer, :class_name => "User",
    :child_key => [:reviewer_id]
end
```

Here we have two user objects parenting our paper object. To handle ambiguity, we use the hash keys `class_name` and `child_key`. The first is a string representation of the parent class name, and the second is an array indicating how the key should be stored within our child. At the end this produces the methods `author_id`, `reviewer_id`, author, and reviewer on papers, where the first two are essentially association properties and the second two are means of retrieving the associated objects.

Let's take a look at how the belongs_to magic is performed:

```
module DataMapper::Associations
  def belongs_to(name, options={})
    @_valid_relations = false

    if options.key?(:class_name) &&
      !options.key?(:child_key)

      warn "..." # must set both
    end

    relationship = ManyToOne.setup(name, self, options)
  end
end

module DataMapper::Associations::ManyToOne

  # Set up many-to-one relationship between two models
  # -
  # @api private
  def self.setup(name, model, options = {})
    assert_kind_of 'name',    name,    Symbol
    assert_kind_of 'model',   model,   Model
    assert_kind_of 'options', options, Hash

    repository_name = model.repository.name

    model.class_eval <<-EOS, _ _FILE_ _, _ _LINE_ _
      def #{name}
        #{name}_association.nil? ?
          nil : #{name}_association
      end

      def #{name}=(parent)
        #{name}_association.replace(parent)
      end

      private

      def #{name}_association
        @#{name}_association ||= begin
          unless relationship = model.relationships(
            #{repository_name.inspect})[:#{name}]
            raise ArgumentError,
              "Relationship #{name.inspect} "+
              "does not exist in \#{model}"
          end

          association = Proxy.new(relationship, self)
          child_associations << association
          association
        end
      end
    EOS

    model.relationships(repository_name)[name] =
      Relationship.new(
        name,
        repository_name,
        model,
        options.fetch(:class_name,
          Extlib::Inflection.classify(name)),
        options
      )
  end
end
```

Starting with the `Associations` module, we can see that `belongs_to` fires off the creation of a many-to-one association. Moving on to `ManyToOne.setup`, we find extensive class evaluation where new methods for the association are defined. These allow us to get or set the association. Note that the `reader` method essentially proxies to the parent model (using a `Proxy` class later defined within `ManyToOne`). It also employs the `Relationship` class, DataMapper's most generalized way of storing information on associations within model classes. Finally, note that the use of `ManyToOne` does not strictly indicate that the relationship between the two models needs to be many-to-one. It may indeed be one-to-one (as determined within the other model), but from the perspective of the child model the more generalized `ManyToOne` class is appropriate for handling both possibilities.

5.4.2 Has

At this point you may be wondering about the flip side of the relationship, that is, the parent. Has associations are meant to handle this. The characteristics of has associations, however, differ in that they are meant to associate varying numbers of related model resources without storing information within the model object itself.

Let's create the counterpoint of the comment model we created in the previous section:

```
class User
  include DataMapper::Resource

  property :id, Serial
  property :login, String, :nullable => false

  has n, :comments
end
```

Note that the has method takes a minimum of two parameters. The first of these is the cardinality, which may be specified by a number, series, or n, and the second is the symbolized name of the associated class. If you're scratching your head over n, just know that it is equivalent to 1.0/0 and that it allows an indefinite number of associates. If you're coming from the ActiveRecord world, you can think of this as the "many" in has_many.

As we did before, we can tweak our relationship for the sake of the domain logic:

```
class User
  include DataMapper::Resource

  property :id, Serial
  property :login, String, :nullable => false

  has 1, :authored_papers, :class_name => "Paper",
    :child_key => [:author_id],
    :remote_name => :author

  has n, :reviewed_papers, :class_name => "Paper,
    :child_key => [:reviewer_id],
    :remote_name => :reviewer

end
```

There are plenty of things to notice this time. Once again, we've specified the associated class name along with the child_key, but we've also set remote_name, which is the symbolized name of the relationship in our other model. Last, note that we set the cardinality of the first to 1, which limits users to authoring only one paper, effectively making the relationship one-to-one.

Having now seen the use of has, let's go into the source to understand how it works:

```
module DataMapper::Associations
  def has(cardinality, name, options = {})
```

```
    if name.kind_of?(Hash)
      name_through, through =
      name.keys.first, name.values.first
    end

    options = options.merge(
      extract_min_max(cardinality))
    options = options.merge(
      extract_throughness(name))

    # ... some warnings

    klass = options[:max] == 1 ? OneToOne : OneToMany

    # ... we'll show you later

    relationship = klass.setup(
      options.delete(:name), self, options)
  end

  private

  def extract_min_max(constraints)
    assert_kind_of 'constraints', constraints,
      Integer, Range unless constraints == n

    case constraints
      when Integer
        { :min => constraints, :max => constraints }
      when Range
        if constraints.first > constraints.last
          raise ArgumentError, "..."
        end
        { :min => constraints.first,
          :max => constraints.last }
      when n
        { :min => 0, :max => n }
    end
  end

end
```

From this we can see that `has_many`, like `belongs_to`, creates an association, but that it may be `OneToOne` or `OneToMany` based upon the max cardinality. Because we've already looked inside one of these associations and because the others are set up in similar ways, we'll leave it up to you to explore further if you like.

5.4.3 Has through

You may need to work with one-to-many-through or many-to-many relationships. To handle these, DataMapper uses `through`. Let's tackle one-to-many-through first and then take a look at many-to-many relationships:

```
class Post
 include DataMapper::Resource

 has n, :taggings
 has n, :tags, :through => :taggings
end

class Tagging
 include DataMapper::Resource

 belongs_to :post
 belongs_to :tag
end

class Tag
 include DataMapper::Resource

 property :id, Serial
 property :value, String

 has n, :taggings
 has n, :posts, :through => :taggings
end
```

These examples show us three associated models where the `Tagging` class acts like a join table bridging the one-to-many relationships from both sides. Sometimes, though, you don't want to explicitly define this middle table. DataMapper lets you do this by setting `through` to `Resource`:

```
class Post
 include DataMapper::Resource

 has n, :post
 has n, :tags, :through => Resource
end

class Tag
 include DataMapper::Resource

 property :id, Serial
 property :value, String
```

```
  has n, :posts, :through => Resource
end
```

This automatically creates the bridging model for us dynamically. But this isn't magic; remember the line from the `def has` snippet of source code that we didn't show you? Here it is:

```
  klass = ManyToMany if options[:through] ==
    DataMapper::Resource
```

With that inside the previous snippet, it's easy to see that the use of the `through` option with `Resource` changes the association setup to a `ManyToMany`. This special association is used to create a join table model for you. Here's part of the setup method showing just that:

```
module DataMapper::Associations::ManyToMany

  def self.setup
    # ... the usual

    unless Object.const_defined?(model_name)
      model = DataMapper::Model.new(storage_name)

      model.class_eval <<-EOS, _ _FILE_ _, _ _LINE_ _
        def self.name; #{model_name.inspect} end
        def self.default_repository_name
          #{repository_name.inspect}
        end
        def self.many_to_many; true end
      EOS

      names.each do |n|
        model.belongs_to(
          Extlib::Inflection.underscore(n).gsub(
            '/', '_').to_sym)
      end

      Object.const_set(model_name, model)
    end

    relationship
  end
```

Note the particularly unique creation of a model through `Model.new` as opposed to a standard class definition. This is meant only for dynamically defined models like the one above.

5.5 CRUD basics

DataMapper as an ORM is intended to create, retrieve, update, and delete records from a repository through interactions with Ruby objects. This means that we don't have to write SQL statements through the normal course of usage. In fact, DataMapper's versatility, intelligence, and performance will probably leave you never needing to write a single SQL statement in your entire application.

Throughout this section, we will assume the existence of the following model:

```
class BlogEntry
  include DataMapper::Resource

  property :id, Serial
  property :live, TrueClass
  property :title, String
  property :text, Text
end
```

5.5.1 Creating records

The creation of DataMapper records is a two-step process. The first step is the creation of a new model object. This is as simple as initializing with the `new` method. This method can also take an attributes hash that will set the model object's properties. The second step of this process is the saving of the object's data into the database as a record. This is done via the `save` method. Below we create a new blog entry and then save it immediately after.

```
blog_entry = BlogEntry.new(:title => "Model Magic!",
  :text => "Persistently cool.")

blog_entry.save
```

At the end, this issues a SQL insert command, saving the data in our database. However, let's take a look first at the most superficial methods inside the DataMapper that make this work:

```
module DataMapper::Resource
  def save

    # ... association related

    saved = new_record? ? create : update

    if saved
      original_values.clear
    end

    # ... association related

    (saved | associations_saved) == true
  end

  def new_record?
    !defined?(@new_record) || @new_record
  end

  protected

  def create
    return false if new_record? &&
      !dirty? && !model.key.any? { |p| p.serial? }

    # set defaults for new resource
    properties.each do |property|
      next if attribute_loaded?(property.name)
      property.set(self, property.default_for(self))
    end

    return false unless repository.create([ self ]) == 1

    @repository = repository
    @new_record = false

    # ... IdentityMap related

    true
  end

  private

  def initialize(attributes = {})
    assert_valid_model
    self.attributes = attributes
  end

end
```

As you can see, the default `initialize` has been overridden so that it can set attributes. You'll also spot a method `assert_valid_model`, but it isn't of much interest since all it does is confirm that the model class does in fact have properties defined. Moving on to the `save` method, you'll find that it first checks to see if the model object should be a new record. To do this it uses the public method `new_record?`, which is also available to you should you need it as an application or plugin developer. Then, given that our record is new, it invokes the protected method `create`. This method effectively cascades through the resource's repository object down to an adapter, where a SQL create statement is executed.

Alternatively, you can shorten this process to a single step by using the class method `create` defined within the `Model` module. Here we use it just as we did before:

```
blog_entry = BlogEntry.create(:title => 'Models Rule!',
  :text => 'Persistently cool.')
```

Taking a peek at the source code, we find that the class method `create` does exactly what we did ourselves before but returns the model object for our convenience:

```
module DataMapper::Model

  def create(attributes = {})
    resource = new(attributes)
    resource.save
    resource
  end

end
```

5.5.2 Retrieving records

The retrieval of model records is principally done through the two methods `all` and `first`. These two methods pull up a collection of records or access a single record, respectively. They can easily be chained, allowing for the refining of the data to be retrieved. Let's take a look at some basic examples:

```
user = User.first(:login => 'foysavas')
groups = Group.all(:name => '%Ruby%')
admin_groups = groups.all(:user => user)
```

The first line looks up a user by login, the second retrieves all groups with the word *Ruby* in them, and the third refines the collection of the second to only those where the user is the admin. Let's look at the source behind the methods `first` and `all` to get an understanding of how they work:

```
module DataMapper::Model

  def all(query = {})
    query = scoped_query(query)
    query.repository.read_many(query)
  end

  def first(*args)
    query = args.last.respond_to?(:merge) ?
      args.pop : {}

      query = scoped_query(
        query.merge(:limit => args.first || 1))

    if args.any?
      query.repository.read_many(query)
    else
      query.repository.read_one(query)
    end
  end

end
```

Both `all` and `first` use the method `scoped_query` to integrate new query parameters with any preexisting ones that may exist higher up on a collection on which the method may be acting:

```
module DataMapper::Model

  private

  def scoped_query(query = self.query)
    assert_kind_of 'query', query, Query, Hash

    return self.query if query == self.query

    query = if query.kind_of?(Hash)
      Query.new(query.has_key?(:repository) ?
        query.delete(:repository) :
        self.repository, self, query)
    else
      query
    end

    if self.query
      self.query.merge(query)
    else
      merge_with_default_scope(query)
    end
  end

end
```

DataMapper uses the method `assert_kind_of` as a way of enforcing types and throws errors when types do not match. Thus, above, we see that `scoped_query` accepts only queries and hashes. The hashes are really just cases of yet-to-be-initiated queries coming from the parameters of some method like `all` or `first`. If both new query parameters and an existing query exist, the two are merged. The model's default scope (typically having no conditions and all non-lazy model fields) is used to merge in further conditions.

Design decision: query object algebra

If there was ever something to be excited about with regard to DataMapper, this is it. Under the hood, the reason why DataMapper can do things like chain retrieval methods and use strategic eager loading is that it essentially handles queries as elements of an algebraic structure. This is completely unlike most ORMs, which simply treat queries as undesirable SQL remnants to be executed. We recommend letting this subtlety take its time to soak in. However, the essence of it all is that with each call of a retrieve method, DataMapper creates a new query by operating on two other queries (one of them possibly empty). With this fresh perspective on DataMapper, we recommend visiting the code behind the method `Query#update`, which we're just about to do.

The method `Query#merge` duplicates the query and then seeks to update it. Below we see this method as it leads into `Query#update`.

```
class DataMapper::Query

  def update(other)
    assert_kind_of 'other', other, self.class, Hash

    assert_valid_other(other)

    if other.kind_of?(Hash)
      return self if other.empty?
      other = self.class.new(@repository, model, other)
    end

    return self if self == other

    @reload = other.reload?
      unless other.reload? == false
    @unique = other.unique?
      unless other.unique? == false
    @offset = other.offset
```

```
      if other.reload? || other.offset != 0
    @limit = other.limit
      unless other.limit == nil
    @order = other.order
      unless other.order == model.default_order
    @add_reversed = other.add_reversed?
      unless other.add_reversed? == false
    @fields = other.fields
      unless other.fields == @properties.defaults
    @links = other.links
      unless other.links == []
    @includes = other.includes
      unless other.includes == []

    update_conditions(other)

    self
  end

  def merge(other)
    dup.update(other)
  end

end
```

Note that the method `update` picks out special query parameters before updating the conditions and finally returning itself.

5.5.2.1 Special query parameters

The parameters passed into `all` and `first` are mostly understood simply as conditions upon parameters. However, certain keys are understood as special query parameters that shape the query in other ways. The following list should make the use of each of these clear:

- `add_reversed`—reverses the order in which objects are added to the collection. Defaults to false.
- `conditions`—allows SQL conditions to be set directly using an array of strings. Conditions are appended to conditions specified elsewhere.
- `fields`—sets the fields to fetch as an array of symbols or properties. Defaults to all of a model's non-lazy properties.
- `includes`—includes other model data specified as a list of DataMapper property paths.

- `limit`—limits the number of records returned. Defaults to 1 in the case of `first` and is otherwise not set.
- `links`—links in related model data specified by an array of symbols, strings, or associations.
- `offset`—the offset of the query, essential for paging. Defaults to 0.
- `order`—the query order specified as an array or properties (or symbols) modified by the two direction methods `desc` and `asc`.
- `reload`—causes the reloading of the entire data set. Defaults to false.
- `unique`—groups by the fields specified, resulting in a unique collection. Defaults to false.

5.5.2.2 Lazy loading of collections

DataMapper does not load collections or issue database queries until the data is absolutely needed. The major benefit here is that application developers can worry less about the database side of things once again, knowing that unless they actually use the data of a resource, no database query will be executed. With Merb, we've also found that this means simpler controller code, since we can use chained relationships or pagination inside the view. With any other ORM, this may be extremely bad form, given that it implies littering the view with lines of supporting code as well as incurring performance penalties based on the retrieval of possibly unused data. Below we present the practical application of collection lazy loading.

```ruby
# Posts Controller
# app/controllers/posts.rb

class Posts
 before :set_page

 def index
  @posts = Post.all
  render
 end

 private

 def set_page
  @p = params[:page] > 0 ?
   params[:page] : 1
 end
end
```

```
# Posts Index View
# app/views/posts/index.html.haml
- @posts.all(:limit => 10, :offset => 10*@p).each do |i|
  .post
    = @posts.name
```

Note that this executes only one database query, specifically at the `each`. To see how and why this works, we need to take a look at some of the code in the parent class of `Collection`, `LazyArray`:

```
class LazyArray # borrowed partially from StrokeDB
  instance_methods.each { |m|
    undef_method m unless %w[
      _ _id_ _ _ _send_ _ send class dup object_id
      kind_of? respond_to? equal? assert_kind_of
      should should_not instance_variable_set
      instance_variable_get extend ].include?(m.to_s)
  }

  # add proxies for all Array and Enumerable methods
  ((Array.instance_methods(false)
    | Enumerable.instance_methods(false)).map { |m|
      m.to_s
    } - %w[ taguri= ]).each do |method|

    class_eval <<-EOS, _ _FILE_ _, _ _LINE_ _
      def #{method}(*args, &block)
        lazy_load
        results = @array.#{method}(*args, &block)
        results.equal?(@array) ? self : results
      end
    EOS

  end


  def load_with(&block)
    @load_with_proc = block
    self
  end

  # ...

  private

  def lazy_load
    return if loaded?
    mark_loaded
    @load_with_proc[self]
    @array.unshift(*@head)
    @array.concat(@tail)
    @head = @tail = nil
    @reapers.each { |r|
      @array.delete_if(&r)
    } if @reapers
    @array.freeze if frozen?
  end

  # ...

end
```

Starting at the top, we can see that all but the quintessence methods are undefined. This is because `LazyArray` is meant to emulate the primitive `Array` class, and starting off with a slate that is as blank as possible helps us get there. The next few lines define various instance methods from both `Array` and `Enumerable`, essentially making `LazyArray` a proxy to a real array but prefacing the call of any array method with `lazy_load`. The `lazy_load` method itself either simply returns true if already loaded, or uses a Proc defined through the `load_with` method to populate the array. All in all, the lazy loading of `LazyArray` has a profound impact on the DataMapper API, arguably serving as the foundation for the elegance and straightforwardness of the query algebra.

5.5.2.3 Lazy loading of properties

Some property data is not automatically retrieved when a model object is loaded. For instance, by default, text properties are not loaded unless you specifically request them. This form of lazy loading is facilitated by code with the `Resource` module and `PropertySet` class. Let's see it in action before taking an in-depth look at how it has been put together:

```
# app/models/post.rb
class Post
  include DataMapper::Resource

  property :id, Serial
  property :title, String
  property :body, Text
end

# Example Merb Interaction
> post = Post.first
```

```
 ~ SELECT "id", "title", "is_basic" FROM "posts" ORDER BY
   "id" LIMIT 1
 => #<Post id=1 title="First Post!" body=<not loaded>>
> post.body
 ~ SELECT "body", "id" FROM "posts" WHERE ("id" = 1) ORDER BY "id"
 => "Nothing to see here"
```

Note that if we had multiple text properties, they would all have been loaded by the second line of interaction. To prevent this from happening, you can define lazy contexts on properties, thus segmenting the retrieval of lazy property data:

```
# app/models/post.rb
class Article
  include DataMapper::Resource

  property :id, Serial
  property :title, String, :lazy => true
  property :abstract, Text, :lazy => [:summary, :full]
  property :body, Text, :lazy => [:full]
end
```

It's time to see how this is done. We'll have to open up `Resource` and `PropertySet`, with the insight that a property when either get or set calls the method `Resource#lazy_load`:

```
module DataMapper::Resource

  def lazy_load(name)
    reload_attributes(
      *properties.lazy_load_context(name) -
      loaded_attributes)
  end

end

class DataMapper::PropertySet

  # ...

  def property_contexts(name)
    contexts = []
    lazy_contexts.each do |context,property_names|
      contexts << context
        if property_names.include?(name)
    end
    contexts
  end
  def lazy_load_context(names)
    if names.kind_of?(Array) && names.empty?
      raise ArgumentError, '+names+ cannot be empty',
        caller
    end

    result = []

    Array(names).each do |name|
      contexts = property_contexts(name)
      if contexts.empty?
        result << name # not lazy
      else
        result |= lazy_contexts.values_at(*contexts).
          flatten.uniq
      end
    end
    result
  end
end
```

The methods of `PropertySet` aren't anything special, but seeing how they work certainly clears up any ambiguity that may have existed within the concept of lazy load contexts.

5.5.2.4 Strategic eager loading

If you've used ActiveRecord before, you've probably trained yourself to avoid N+1 queries. These come up frequently in ActiveRecord since iteration over the associates of a model object usually forces you to make a query for each associate. Add in the original query for the model itself, and you have N+1 queries in total. However, DataMapper prevents this from happening and instead issues only two queries. Let's take a look at an example in a view to make this more concrete:

```
<% Post.all.each do |post| %>
  <div class="post">
    <h1><%= post.title ></h1>
    <h2>by <%= post.author.name %></h2>
  </div>
<% end %>
```

With the code above, all posts and the names of their authors are outputted using only two queries: the first to get the posts and the second to get their authors. This kind of elimination of N+1 queries is called **strategic eager loading** and is possible thanks to a combination of many different DataMapper implementation decisions. To get an idea of how strategic eager loading works, let's take a look at some code inside the `Relationship` class that would have been used in the previous example:

```
class DataMapper::Associations::Relationship

  # ...

  # @api private
  def get_parent(child, parent = nil)
    child_value = child_key.get(child)
    return nil if child_value.any? { |v| v.nil? }

    with_repository(parent || parent_model) do
      parent_identity_map = (parent || parent_model).
        repository.identity_map(parent_model.base_model)
      child_identity_map  = child.
        repository.identity_map(child_model.base_model)

      if parent = parent_identity_map[child_value]
        return parent
      end

      children = child_identity_map.values
      children << child
        unless child_identity_map[child.key]

      bind_values = children.map {
        |c| child_key.get(c) }.uniq
      query_values = bind_values.reject {
        |k| parent_identity_map[k] }

      bind_values = query_values
        unless query_values.empty?
      query = parent_key.zip(bind_values.transpose).
        to_hash
      association_accessor =
        "#{self.name}_association"

      collection = parent_model.send(:all, query)
      unless collection.empty?
        collection.send(:lazy_load)
        children.each do |c|
          c.send(association_accessor).
            instance_variable_set(
              :@parent,
              collection.get(*child_key.get(c)))
        end
        child.send(association_accessor).
          instance_variable_get(:@parent)
      end
    end
  end

end
```

From this we learn that in the process of getting a parent resource, DataMapper pulls up the identity map of the parent model and child model to see if the resource has already been loaded. If it has, DataMapper short-circuits any retrieval and simply returns the appropriate parent. Most important, if the resource is not already loaded, DataMapper uses the parent keys from all the relevant children within a collection query. The results are then loaded immediately, and after all children are connected with their parents, the parent requested is returned.

5.5.3 Updating records

Resources can be updated by using the `save` method similarly to how it was used with record creation. However, for saving to have any effect, it is necessary that at least one property value be recently set. This causes DataMapper to mark certain properties as dirty and use them during the creation of an update statement. Below we display the two ways of setting a property value and causing it to be marked as dirty.

```
post = Post.first
post.title = "New Title"
post.attribute_set(:body, "New Body")
post.save
```

However, note that the second method `attribute_set` is typically reserved for use inside override writer methods. Note that `save`, in the case of non-new records, cascades to the calling of `update`. Thus we have the option of using that method directly if we want:

```
post.update
```

5.5.3.1 Using `update_attributes`

There is one other way to invoke the updating of attributes. This is to use the method `update_attributes`, which accepts an arbitrary hash and then an optional constraining property array. Consequently, it works well with form parameters a user may have passed in:

```
class Users
  def update
```

```
    if @user.update_attributes(params[:user],
      [:name, :email, :description])
      redirect resource(@user)
    else
      render :edit
    end
  end
end
```

Here we have constrained the user to being able to update only name, email, and description.

5.5.3.2 Original and dirty attribute values

You may at some point want to enhance model logic through the comparison of original and dirty attribute values. Here we do so within the method `update_speed`:

```
class Position
  property :id, Serial
  property :vertical_position, Integer
  property :horizontal_position, Integer
  property :speed, Float

  belongs_to :player

  before :save, :update_speed

  private

  def update_speed
    dy = 0
    dx = 0
    if original_values[:vertical_position]
      dy = vertical_position -
        original_values[:vertical_position]
    end
    if original_values[:horizontal_position]
      dx = horizontal_position -
        original_values[:horizontal_position]
    end
    v = Math.sqrt(dx*dx + dy*dy)
    attribute_set(:speed, v)
  end

end
```

You may notice that we use a `before` hook here. We'll cover hooks in the next section.

5.5.4 Destroying records

Records can be deleted by using the `destroy` method. Alternatively, if you're looking to delete a full collection of resources, you can use the method `destroy!`:

```
User.first(:id => 2).destroy
Post.all(:user_id => 2).destroy!
```

5.6 Hooks

DataMapper hooks differ from those used in Merb controllers. This is because controller filter chains require a decently specific form of logic. DataMapper models, on the other hand, have the benefit of using the more generic `Hook` class of Extlib.

Design decision: explicit hooks over aliasing of methods

DataMapper could have left it up to application developers to fend for themselves when it comes to hooks. After all, Ruby is an extremely versatile language, and the `alias` method could be used to chain model methods like `create` and `save`. However, DataMapper developers, like Merb developers, shared a distaste for the aliased decoration of methods for, among other reasons, its confusing implications on stacktraces. Therefore, `Extlib::Hook` was created to enable application developers to easily decorate both instance and class methods without doing any aliasing.

From the application developer's perspective, the two methods used to create hooks are `before` and `after`. `DataMapper` registers specific hooks for the methods `save`, `create`, `update`, and `destroy`. Each of these can be used with `before` and `after`.

```
module DataMapper
  module Hook
    def self.included(model)
      model.class_eval <<-EOS, _ _FILE_ _, _ _LINE_ _
        include Extlib::Hook
        register_instance_hooks :save, :create,
          :update, :destroy
      EOS
    end
  end
  DataMapper::Resource.append_inclusions Hook
end # module DataMapper
```

5.7 Plugins

Modularity has been a central objective of DataMapper. Thus, many of the features that you might otherwise expect within a standard ORM are with DataMapper found as plugins. This includes timestamping, aggregation, validations, and various data structures. In this section we'll go over the most fundamental of these plugins, understanding not only how they're used but also how they work.

5.7.1 Extra property types

The package `dm-types` provides numerous additional property types. Here's a list of those included:

- `BCryptHash`—encrypts a string using the bcrypt library
- `Csv`—parses strings as `CSV`s using `FasterCSV`
- `Enum`—stores an enumerated value as an integer
- `EpochTime`—converts `Time` and `DateTime` to `EpochTime`, that is, the number of seconds since the beginning of UNIX time
- `FilePath`—stores paths as strings using `Pathname`
- `Flag`—binary flags stored as integers
- `IpAddress`—IP address stored as a string
- `Json`—JSON stored as a string
- `Regexp`—regular expressions stored as strings
- `Serial`—an auto-incrementing integer type
- `Slug`—escapes a stored string, making it suitable to be used as part of a URL
- `URI`—stores an `Addressable::URI` as a string
- `UUID`—creates a UUID stored as a string
- `Yaml`—stores YAML as a string

Let's take a look at the source to one of these for a better understanding of how to create our own types:

```
require 'yaml'

module DataMapper
  module Types
    class Yaml < DataMapper::Type
      primitive String
      size 65535
      lazy true

      def self.load(value, property)
        if value.nil?
          nil
        elsif value.is_a?(String)
          ::YAML.load(value)
        else
          raise ArgumentError.new(
            "+value+ must be nil or a String")
        end
      end

      def self.dump(value, property)
        if value.nil?
          nil
        elsif value.is_a?(String) && value =~ /^---/
          value
        else
          ::YAML.dump(value)
        end
      end

      def self.typecast(value, property)
        # Leave values exactly as they're provided.
        value
      end
    end # class Yaml
  end # module Types
end # module DataMapper
```

As you can see, new DataMapper types can be created by subclass off of `DataMapper::Type`. You will then have to set the primitive type, and this can be done using the class method `primitive`. You may additionally have to set attributes like size and laziness as was done in the case above. Finally, the two methods that do the hard work are the class methods `load` and `dump`. These need to be defined only if the custom type needs to override them from simply returning the value. With the `Yaml` type, strings are converted into YAML when loaded from the database and are converted to strings when they need to be dumped into the database.

5.7.2 Timestamps

The gem `dm-timestamps` is one of the most commonly used DataMapper plugins. It saves you from having to code timestamping into your models. Note that once the gem is included, it applies to all DataMapper models. Thus we can set the following four properties in any Merb stack model, knowing they will automatically be set when needed:

```
class User
  property :created_at, DateTime
  property :created_on, Date
  property :updated_at, DateTime
  property :updated_on, Date
end
```

Because dm-timestamps is a decently simple plugin but also reveals the foundation of resource extension plugins, let's take a quick look:

```
module DataMapper
  module Timestamp
    Resource.append_inclusions self

    TIMESTAMP_PROPERTIES = {
      :updated_at => [ DateTime,
        lambda { |r, p| DateTime.now } ],

      :updated_on => [ Date,
        lambda { |r, p| Date.today } ],

      :created_at => [ DateTime,
        lambda { |r, p|
          r.created_at ||
            (DateTime.now if r.new_record?) } ],

      :created_on => [ Date,
        lambda { |r, p|
          r.created_on ||
            (Date.today if r.new_record?) } ],
    }.freeze

    def self.included(model)
      model.before :create, :set_timestamps
      model.before :update, :set_timestamps
      model.extend ClassMethods
    end

    private

    def set_timestamps
      return unless dirty?
      TIMESTAMP_PROPERTIES.each do |name,(_type,proc)|
        if model.properties.has_property?(name)
          model.properties[name].set(self,
            proc.call(self,
            model.properties[name])) unless
            attribute_dirty?(name)
        end
      end
    end

    module ClassMethods
      def timestamps(*names)
        raise ArgumentError, '...' if names.empty?

        names.each do |name|
          case name
            when *TIMESTAMP_PROPERTIES.keys
              type, proc = TIMESTAMP_PROPERTIES[name]
              property name, type
            when :at
              timestamps(:created_at, :updated_at)
            when :on
              timestamps(:created_on, :updated_on)
            else
              raise InvalidTimestampName,
              "Invalid timestamp property name '#{name}'"
          end
        end
      end
    end # module ClassMethods

    class InvalidTimestampName < RuntimeError; end
  end # module Timestamp
end # module DataMapper
```

The first line to notice is the third one. Here the module appends itself to `Resource`. As we saw earlier in this chapter, this gets the `Timestamp` module automatically included in all classes that include `DataMapper::Resource`. Moving on, we see the definition of a number of lambdas to be used in setting the four basic timestamps. This is followed by the class method `included`, which sets up before hooks to apply the timestamps. It all extends our model classes with a `timestamps` class method. This is a convenience method for defining the various timestamp properties tersely.

5.7.3 Aggregates

The DataMapper core has been designed to limit its use as a reporting tool and simply act as an ORM. The plugin `dm-aggregates` consequently adds in some of the most common aggregating methods used by SQL databases:

- count—finds the number of records in a collection by directly using a count SQL statement and not the Ruby size method
- min—finds the minimum value of a numerical property using SQL
- max—finds the maximum value of a numerical property using SQL
- avg—finds the average value of a numerical property using SQL
- sum—totals the values of a numerical property using SQL

Chances are you will use dm-aggregates at some point. However, before we look into the source, it's best to recognize that the plugin essentially extends DataMapper's capability to what it was not really meant to do.

```
module DataMapper
  class Collection
    include AggregateFunctions

    private

    def property_by_name(property_name)
      properties[property_name]
    end
  end

  module Model
    include AggregateFunctions

    private

    def property_by_name(property_name)
      properties(repository.name)[property_name]
    end
  end

  module AggregateFunctions
    def count(*args)
      query = args.last.kind_of?(Hash) ? args.pop : {}
      property_name = args.first

      if property_name
        assert_kind_of 'property',
          property_by_name(property_name), Property
      end
      aggregate(query.merge(:fields =>
        [ property_name ?
          property_name.count : :all.count ]))
    end
  end

  module Adapters
    class DataObjectsAdapter
      def aggregate(query)
        with_reader(read_statement(query),
          query.bind_values) do |reader|

          results = []

          while(reader.next!) do
            row = query.fields.zip(
              reader.values).map do |field,value|

              if field.respond_to?(:operator)
                send(field.operator, field.target, value)
              else
                field.typecast(value)
              end
            end

            results << (query.fields.size > 1 ?
              row : row[0])
          end

          results
        end
      end

      private

      def count(property, value)
        value.to_i
      end

      module SQL
        private

        alias original_property_to_column_name
          property_to_column_name

        def property_to_column_name(repository,
          property, qualify)
```

```
          case property
            when Query::Operator
              aggregate_field_statement(repository,
                property.operator, property.target, qualify)

            when Property, Query::Path
              original_property_to_column_name(repository,
                property, qualify)

            else
              raise ArgumentError, "..."
          end
        end

        def aggregate_field_statement(repository,
          aggregate_function, property, qualify)

          column_name = if aggregate_function == :count
            && property == :all
            '*'
          else
            property_to_column_name(repository, property, qualify)
          end

          function_name = case aggregate_function
            when :count then 'COUNT'
            # ...
            else raise "Invalid ... "
          end

          "#{function_name}(#{column_name})"
        end
      end # module SQL

      include SQL

    end
  end

end
```

Above we have included only the code covering the `count` method. However, it's easy to recognize the substantial monkey patching going on, particularly in the case of the `SQL` methods. Otherwise, though, this is a great example of the trickling down of method calls from collections and models into the adapter where SQL statements are formed.

5.7.4 Validations

The plugin dm-validations validates the property values of model objects before saving them. This means that if a model object returns false upon save, you can most likely interpret it as having been caused by undesirable values on properties. Another way to check if a particular model is valid is to directly use the valid? method that is squeezed in before create or update. Before we go any further, here's a list of the validation methods available within your models through dm-validations:

- validates_present—validates the presence of an attribute value.
- validates_absent—validates the absence of an attribute value.
- validates_is_accepted—validates that an attribute is true or optionally not false through :allow_nil => true. It can also work with a custom set of acceptance values using :accept => [values].
- validates_is_confirmed—validates the confirmation of an attribute with another attribute, for instance, matching password and password_confirmation. The default confirmation attribute is the original attribute ending in _confirmation, but :confirm can be used to set it to anything else.
- validates_format—validates the format of an attribute value against a regular expression or Proc associated by :with. Alternatively, it can be used with predefined formats such as Email and Url through :as. The :allow_nil key is also available.
- validates_length—validates the value of a numeric against a :min or :max value. Alternatively, a range can be used along with :within.
- validates_with_method—validates either the model as a whole or a specific property through a method. If only one parameter is given, it is the symbolic form of the method to check the entire model. If two are given, they are the attribute and the method used to check that attribute. Error messages can be passed as true by returning an array where false is the first element and a string for an error message is the second from the validating method.
- validates_with_block—like validates_with_method but uses blocks. You can validate either against the whole model or a specific attribute as well as pass in error messages.
- validates_is_number—validates that the value of an attribute is a number, appropriate for use in checking the precision and scale of floats.
- validates_is_unique—validates that the attribute value is unique, either within the scope of the attribute value of all other model objects or some other scope specified

by an array of property symbols through `:scope`. Also accepts `:allow_nil` to be set.

- `validates_within`—validates that an attribute value is within a set of values specified by `:set`.

Alternatively, instead of directly using the validations methods, you can include particular hash key-and-values on property definitions. Here's a list of what keys automatically create appropriate validations:

- `:nullable`—when set to false, automatically creates a presence validator
- `:length` or `:size`—automatically creates a length validator
- `:format`—creates a format validator
- `:set`—creates a within validator

Additionally, numerical properties are automatically validated using `validates_is_number`. To turn off autovalidation on this or any other property type, use `:auto_validation => false`.

5.7.4.1 Conditions

Validation methods are also capable of generally accepting conditions as Procs assigned to `:if` or `:unless`. The single block parameter for these Procs is the resource itself. Thus we can do the following:

```
class Experiment
  include DataMapper::Resource

  property :id, Serial
  property :name, String
  property :impetus, Text
  property :question, Text
  property :hypothesis, Text
  property :description, Text
  property :conclusion, Text
  property :completed, TrueClass
  property :result, TrueClass

  validates_present :conclusion, :if => proc { |r|
    r.completed? && r.result?
  }
end
```

For terseness, DataMapper validations also allow us to specify a method as a symbol instead of a full Proc. Here we require only that an experiment be complete for it to have a conclusion:

```
 validates_present :conclusion, :if => :completed?
```

5.7.4.2 Contexts

Contexts allow us to do validations with similar conditions, but specified at the point of validation. For instance, assuming we have the same `Experiment` model from before, we may set different contexts on particular property validations specifying that they must be validated together:

```
  validates_present :impetus, :when => [:proposal]
  validates_present :question, :when => [:proposal]
  validates_absent  :completed, :when => [:proposal]
```

The array assigned to **when** is an array of contexts. The default context is known as `:default`. We can now use these contexts by including them as a parameter with `valid?`:

```
exp = Experiment.new(
  :name => 'Great Subjective Experiment',
  :impetus => 'Thoughts on physicalism and the mind',
  :question => 'Is it possible to subjectively test '+
    'the consciousness of other modes of thought '+
    'through their integration with your own?'
  :completed => true)
exp.valid?(:proposal) # => false
```

5.7.4.3 Errors

Every time a model object is validated, it populates (or empties) a hash of errors accessible through the resource instance method `errors`. These errors can be used to indicate to a user that something went wrong or otherwise recognize what particular attributes are invalid:

```
resource.errors.each do |e|
 puts e # => [[:attr_name, ["Error!"]]]
end

resource.errors.on(:attr_name) # => ["Error!"]
resource.errors.on(:another_attr) # => nil
```

Notice how `errors.on` returns nil when there are no errors. Consequently it can be used to test if an error is present on a property. Also note that error messages are given as arrays. This is because multiple validation errors may have occurred on a single attribute.

5.8 Conclusion

DataMapper is an undeniably excellent ORM. It offers application developers the chance to stay as far away from database work as possible by streamlining development migrations and placing model properties within the model. It also lets us treat ORM objects even more casually by virtue of its eager and lazy loading of data sets. Inside the Merb stack or not, the highly modular DataMapper will serve you well.

Index