ActiveRecord

Basics

ActiveRecord is the ORM (Object-Relational Mapping) for Rails. It allows you to interact with the database without writing raw SQL queries, which is convenient and also prevents SQL-injection attacks.

Recall how to use a generator to create a model and the corresponding migration:

```bash
rails generate model User name:string age:integer
```

At this point, you have created the model, but the corresponding table in the database does not exist. To create it, run the migration:

```bash
rake db:migrate
```

Now, if you look in `db/schema.rb`, you can see that your model is now part of the database schema:

```ruby
...  
  create_table "users", :force => true do |t|  
    t.string   "name"  
    t.integer  "age"  
    t.datetime "created_at", :null => false  
    t.datetime "updated_at", :null => false  
  end  
...
```

A few notes about this:

1. Rails automatically creates and maintains timestamps (`created_at` and `updated_at`) for your models.
2. A primary key called `id` is also created for you automatically, although it is not listed in the schema.
3. The name of the table in the database is `users`, although the name of the model is `User`. Rails has pluralized your model name! If your model is called `Knife`, rails will actually create a table called `knives`.

You can revert your most recent migration by running:

```bash
rake db:rollback
```

Check `db/schema.rb` to confirm that the `users` table has been removed. Now, run `rake db:migrate` again for purposes of the rest of the demo.
In Project 1.1, you learned that you can add a row to the table that corresponds to a model as follows. (Open up rails console to follow along with the code snippets.)

```ruby
user = User.create :name => "Stephan", :age => 21
```

That was the Create part of the CRUD operations. But you can also Read: Now, you can fetch a particular user by id:

```ruby
user = User.find(1)
```

Or by name:

```ruby
user = User.where("name = ?", "Stephan").first
```

**Note:** user = User.where(:name => "Stephan").first is better because the above is basically writing SQL and also may be susceptible to injection attacks.

You can also chain the where method:

```ruby
user = User.where(:name => "Stephan").where(:age => 21).first
```

Maybe contrary to intuition, this will not cause two database accesses (first filtering for name == "Stephan" and then separately filtering for age == 21). So don’t be afraid of using chaining.

Or to get all users:

```ruby
users = User.all
```

To check if a particular user exists, you can do this by id:

```ruby
if User.exist?(1)
    # the user with id 1 exists
end
```

Or by something else:

```ruby
if User.where("name = ?", "Stephan").exists?
    # a user named Stephan exists
end
```

If you have an instance of a model and modify it, you can save it back to the database with the save method:

```ruby
first_user = User.first
first_user.name = "Dalton"
first_user.save
```

Finally, to delete a user from the database:
User.find(1).destroy

Note: destroy doesn’t actually destroy the user object from memory (you can still read the object). However, it will freeze the object so that it cannot be modified--which is a nice feature for code safety reasons.

Associations

Suppose we wanted to build a notifications system, where each user can have zero or more notifications, and each notification belongs to exactly one user. Given our User model above, you might create a Notification model like this:

rails generate model Notification user_id:integer message:string

Now if you have a notification and you want to get the user that it belongs to, you can look up the user by the user_id stored in the notification. If you want to find all the notifications that belong to a particular user, you could use a "join" to do this, but there’s an easier way.

Figure 1: An example of a has_many/belongs_to association.

We can tell Rails about this relationship between notifications and users -- this is called an association. Open app/models/user.rb:

class User < ActiveRecord::Base
  attr_accessible :age, :name
end
Now modify it so rails knows that users have notifications:

class User < ActiveRecord::Base
  attr_accessible :age, :name
  has_many :notifications, :dependent => :destroy
end

Similarly, open app/models/notification.rb:

class Notification < ActiveRecord::Base
  attr_accessible :message, :user_id
end

And modify it so rails knows that notifications belong to users:

class Notification < ActiveRecord::Base
  attr_accessible :message, :user_id
  belongs_to :user
end

Now it is easy to get all notifications for a particular user:

# get all notifications that belong to the first user
notifications = User.first.notifications

Similarly, it is also easy to get the user that owns a particular notification:

# get the user that owns the first notification
user = Notification.first.user

It is also easy to create a notification for a particular user:

# give the first user a notification
User.first.notifications.create :message => "Hello, Stephan!"

The new notifications field of User uses some fancy Ruby metaprogramming to allow you to access the rows in the notifications table that belong to that particular user as if it were an ordinary array. By calling its create method, you are appending a new item to the array and also adding a new record to the database.

Recall that when we modified app/models/user.rb to tell rails that users have notifications, we had this clause:

:dependent => :destroy

This means that if we destroy a user, all of that user's notifications are automatically destroyed as well:

# destroy the first user and all his notifications
User.first.destroy
Note that there are several other types of associations, but has_many/belongs_to is perhaps the most common one. Learn more here: [http://guides.rubyonrails.org/association_basics.html](http://guides.rubyonrails.org/association_basics.html)

## JavaScript

### Event Model

You may be used to writing *synchronous* programs, which means that your program waits for each command to finish before executing the next. On the other hand, most JavaScript programs respond to events *asynchronously*. This means that you specify what code should execute when a particular event happens (e.g. a user clicks on a button), and the rest of your program happily continues without waiting.

All of this revolves around the concept of a *closure*. In JavaScript (just as in Python), you can declare a function that refers to variables declared outside its scope. For example:

```javascript
// declare a variable
var x = 10;

// this function modifies the variable
// even though the variable is defined in an outer scope
function modify_x() {
    x = 20;
}

// print the value of x
console.log(x);

// test our function
modify_x();

// print the modified value of x
console.log(x);
```

The function is said to “close over” the variable `x`. Typically, closures are *anonymous* functions (i.e. they are declared without a name):

```javascript
// declare an anonymous function and assign it to a variable
var f = function() {
    console.log("Hello");
}
```

Functions are *first-class* entities in JavaScript, which means you can treat them as ordinary values and pass them to other functions. This manner of passing code around is exactly how events work in JavaScript. For example, suppose I want to show the user a message when the
window.onload = function() {
    alert("Hello!");
}

The window object has a property called onload, which is a function that is executed when the user enters the page. Other events include button presses, form submissions, when the cursor hovers over an item, when the text in a text field changes, etc. You can find a good list of common events here: http://www.webmonkey.com/2010/02/javascript_events/

AJAX (Asynchronous JavaScript and XML)

Most HTTP requests are synchronous, and early web pages only had static content that was fetched by the browser synchronously. If you wanted to see new content on a website, you would have to reload the page or go to a different page. However, these days it's common for web pages to make subsequent requests back to the server after the page loads to get more data. For example, in Google Maps, when you move the map around and explore new areas, the page is dynamically retrieving data from Google servers without reloading the page. This leads to a better user experience and also less data needs to be transferred (the browser does not need to re-download the entire page).

To make an asynchronous request, we must create an XMLHttpRequest object:

var xhr = new XMLHttpRequest();

Before making the actual request, you should define a handler function that will be called in response to various events. This handler will be called once for each of the four cases:

xhr.onreadystatechange = function() {
    if (xhr.readyState == 1) {
        console.log('The call to open(...) succeeded.');
    }
    if (xhr.readyState == 2) {
        console.log('The call to send(...) succeeded. Waiting for response...');
    }
    if (xhr.readyState == 3) {
        console.log('The response is coming in!');
    }
    if (xhr.readyState == 4) {
        console.log('We now have the complete response: ' + xhr.response);
    }
}

Now we can actually make the request:

xhr.open('GET', '/url/to/something', true);
xhr.send('');
Here, the true argument means *asynchronous*. We could set this to `false`, and then the call would block until the request is received. Also, ‘GET’ is the HTTP verb. You can also make POST requests (or any other type of request). If you were making a POST call, you might also want to pass data to the call to send, rather than the empty string.

Note that depending on how you set up your routes, you can pass data in the URL either like this:

```javascript
xhr.open('GET', '/a/b/c', true);
```

...or like this:

```javascript
xhr.open('GET', '/url?x=a&y=b&z=c', true);
```

Either way, there are limitations on what characters can be in a URL, and generally you want to *encode* your arguments before using them in a URL. In JavaScript, you can do this with the `encodeURIComponent(x)` function. These will automatically be decoded by Rails.

For example, if you wanted to use a URL as an argument in another URL, you might escape it like so:

```javascript
url2 = “/visit?url=” + encodeURIComponent(url1)
```