Extracted from:

RubyMotion

iOS Development with Ruby

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

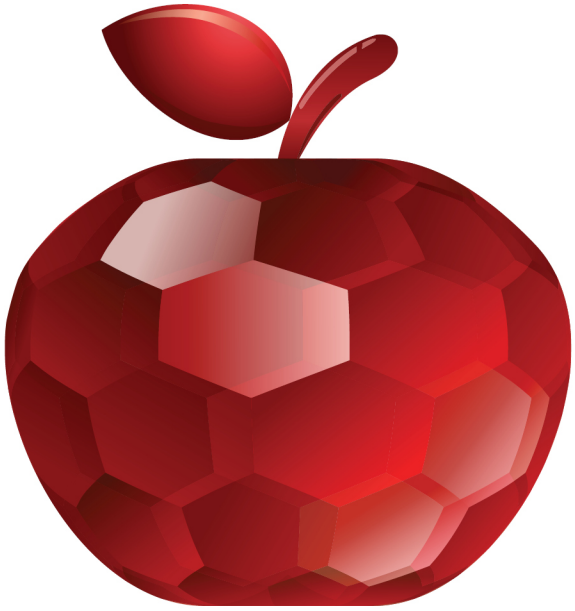
The Pragmatic Programmers

Pragmatic exPress

RubyMotion

iOS Development with Ruby

Updated for RubyMotion 2



Clay Allsopp

Foreword by Laurent Sansonetti,
lead developer of RubyMotion

edited by Fahmida Y. Rashid

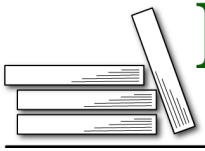
RubyMotion

iOS Development with Ruby

Clay Allsopp

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Fahmida Y. Rashid (editor)
Kim Wimpsett (copyeditor)
David J. Kelly (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-937785-28-4
Encoded using the finest acid-free high-entropy binary digits.
Book version: P3.0—July 2014

CHAPTER 4

Representing Data with Models

We've covered views and controllers, but where's the love for the *M* in MVC? Well, wait no more, because we're going to dive into models. We now understand views and controllers, but in practice models will play just as big of a role as the sexier, user-facing code.

In iOS, there are two big components to models: *CoreData* and, well, everything else. CoreData is an iOS object graph and persistence framework, sort of similar to ActiveRecord in Rails-land. It's an incredibly powerful framework to save and query objects using a database, so it deserves a chapter or even a book onto its own. But even without touching CoreData, we can do a whole lot with just "everything else" about models. It's time to get down to business.

Writing Basic Models

Unlike controllers and views, there's no default superclass that models inherit from; they're just plain-old Ruby objects. We can use the standard `attr_accessor`, `reader`, and `writer` functions to declare getter and setter methods, which sometimes are all you need.

Many apps have users and profiles, so let's work through a portion of that sort of app and use some nice models. Let's create a new project (such as `motion create UserProfile`) and two subdirectories within `./app`: `models` and `controllers`. We're not only going to cover models; we're going to keep building on what we already know.

Our app will let us create, view, and edit users. Since we're doing an awful lot of work with users, this sounds like a good place to begin with our first model. To get started, we first create `user.rb` in `./app/models`. For now, each user will have a name, email, and ID. This is a pretty basic implementation of `User`:

models/UserProfile/app/models/user.rb
```
class User
  attr_accessor :id
  attr_accessor :name
  attr_accessor :email
end
```

That's all we need for now. Let's add our first controller, which we can use to view a user. Create user_controller.rb in app/controllers, which will be a subclass of UIViewController. We'll use a custom initializer that takes a User and fills the UI appropriately. Sound good? Let's start with that initializer.

models/UserProfile/app/controllers/user_controller.rb
```
class UserController < UIViewController
  attr_accessor :user

  def initWithUser(user)
    initWithNibName(nil, bundle:nil)
    self.user = user
    self.edgesForExtendedLayout = UIRectEdgeNone
    self
  end
```

Pretty typical initializer, isn't it? We're going to assume the user passed is a User object, but we'll worry about that later. Next, we need to set up the view. This will be kind of lengthy, but that's nothing new for us at this point.

For each of our User properties, we'll create two labels: one to tell us what value we're looking at ("Email") and one right beside it that presents the value of that property for our user ("clay@mail.com"). Since we are using Ruby's nifty send(), we can do this in one loop.

models/UserProfile/app/controllers/user_controller.rb
```
  def viewDidLoad
    super

    self.view.backgroundColor = UIColor.whiteColor

    last_label = nil
    ["id", "name", "email"].each do |prop|
      label = UILabel.alloc.initWithFrame(CGRectZero)
      label.text = "#{prop.capitalize}:"

      label.sizeToFit
      if last_label
        label.frame = [
          [last_label.frame.origin.x,
            last_label.frame.origin.y + last_label.frame.size.height],
          label.frame.size]
      else
```

```
        label.frame = [[10, 10], label.frame.size]
      end
      last_label = label

      self.view.addSubview(label)

      value = UILabel.alloc.initWithFrame(CGRectZero)
      value.text = self.user.send(prop)
      value.sizeToFit
      value.frame = [
        [label.frame.origin.x + label.frame.size.width + 10, label.frame.origin.y],
        value.frame.size]
      self.view.addSubview(value)
    end
    self.title = self.user.name
  end
end
```

Don't get overwhelmed! We just created two UILabels for each property and laid them out nicely.

Before we run our app, we need to set up our AppDelegate to get the controller on the screen. First, we create a new User in app_delegate.rb and use it to initialize a UserController. We're going to wrap it in a UINavigationController so we get the nice effect with self.title.

models/UserProfile/app/app_delegate.rb

```
class AppDelegate
  def application(application, didFinishLaunchingWithOptions:launchOptions)
    @window = UIWindow.alloc.initWithFrame(UIScreen.mainScreen.bounds)

    @user = User.new
    @user.id = "123"
    @user.name = "Clay"
    @user.email = "clay@mail.com"
    @user_controller = UserController.alloc.initWithUser(@user)
    @nav_controller =
        UINavigationController.alloc.initWithRootViewController(@user_controller)
    @window.rootViewController = @nav_controller
    @window.makeKeyAndVisible
    true
  end
end
```

Let's give it a rake and see what happens! Your app should look something like Figure 7, *A controller for our User model*, on page 8. Our view isn't the prettiest, but a good designer could dress up even this limited information into something shippable.

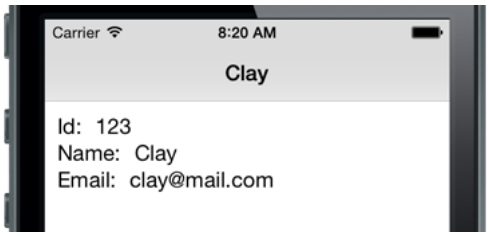


Figure 7—A controller for our User model

This is great, but this User class didn't help out all that much. We're going to give it some abilities that will make our code much more flexible while showing off what smart models can do.

Preparing Scalable Models

The word *scalable* gets thrown around a lot, usually in terms of databases or web back ends, but client-side code can easily become unscalable as well. For instance, if we wanted to add more attributes to our User, we'd have to make changes in three places: our class definition, our controller, and where we instantiate our objects. In a world where engineers move fast, having all this overhead can be a big time sink.

I wouldn't bring this up if there weren't a better way. We're going to use a nice Ruby trick that lets our models become more flexible and readies them for a typical API. In user.rb, change our three attr_acessor() lines into this:

models/UserProfile_flex/app/models/user.rb

```
class User
  PROPERTIES = [:id, :name, :email]
  attr_accessor *PROPERTIES
```

Nifty, right? Now we have one data structure containing our desired properties, instead of multiple independent lines. This lets us refactor code that should apply to all properties into more loops like PROPERTIES.each. For example, we can now make our User initializable with a hash.

models/UserProfile_flex/app/models/user.rb

```
def initialize(properties = {})
  properties.each do |key, value|
    if PROPERTIES.member? key.to_sym
      self.send("#{key}=", value)
    end
  end
end
```

This lets us initialize users in one line instead of needing one line for every property. Plus, when we add new properties, the initializer code still works. Let's start updating our old code to use this concept. In UserController, change our properties from being hard-coded to using User::PROPERTIES.

models/UserProfile_flex/app/controllers/user_controller.rb

```
last_label = nil
User::PROPERTIES.each do |prop|
  label = UILabel.alloc.initWithFrame(CGRectZero)
```

While we're at it, let's change how we created the @user instance variable in AppDelegate.

models/UserProfile_flex/app/app_delegate.rb

```
@window = UIWindow.alloc.initWithFrame(UIScreen.mainScreen.bounds)

@user = User.new(id: "123", name: "Clay", email: "clay@mail.com")

@user_controller = UserController.alloc.initWithUser(@user)
```

If you run our app now, nothing *looks* different, but under the hood we've made some important changes. Just to show how useful this is, let's add a new phone property to User so that we can see how we just change the input when creating a new object.

models/UserProfile_flex2/app/models/user.rb

```
class User
  PROPERTIES = [:id, :name, :email, :phone]
  attr_accessor *PROPERTIES
```

models/UserProfile_flex2/app/app_delegate.rb

```
@window = UIWindow.alloc.initWithFrame(UIScreen.mainScreen.bounds)

@user = User.new(id: "123", name: "Clay",
    email: "clay@mail.com", phone: "555-555-5555")
```

Once you run rake, you'll see that we now have a matching UI element for our new property without having to add new code in the controller. This is a great example of how smart(er) models can save us time, but what if we wanted to edit this data?

Changing Models with Key-Value Observing

Making our model code flexible is the low-hanging fruit, and keeping a view up-to-date with a model is usually a hard problem. A model object has to be aware of all the views accessing its data, which just leads to all sorts of problems and code spaghetti. Imagine a world where it all just works: you

change the name on a User, and the appropriate label instantly reflects those changes...no code spaghetti or zero-ing references.

I'm talking this up so much because it is possible to implement. iOS has a concept of *key-value observing* (KVO). Built into the frameworks is a system by which one object can passively observe changes in properties of another object. Those properties are referred to by *keys*, which typically correspond to the variable name of the attribute.

In our User case, our controller would observe the "name" key of the User. In the callback for that observation, we would reset the label's text to reflect the new name value.

I don't know about you, but I'm anxious to write some code. But not so fast! The original implementation of KVO isn't very Ruby-like. Thankfully, there is a popular RubyGem called BubbleWrap (http://bubblewrap.io) that "wraps" many Objective-C APIs into idiomatic Ruby structures. To install it, run gem install bubble-wrap in your shell and add require "bubble-wrap" to your Rakefile.

models/UserProfile_kvo/Rakefile

```
$:.unshift("/Library/RubyMotion/lib")
require 'motion/project/template/ios'
require 'bubble-wrap'

begin
  require 'bundler'
  Bundler.require
rescue LoadError
end
```

Note here that the order of these statements does matter: BubbleWrap's require should come before the block involving Bundler.

BubbleWrap is an extensive library with many different features, so definitely browse its documentation sometime. For now, we're just going to use the wrappers it creates for key-value observing on our model. We'll be working with UserController, so let's open that and include BubbleWrap's KVO module.

models/UserProfile_kvo/app/controllers/user_controller.rb

```
class UserController < UIViewController
  include BubbleWrap::KVO
  attr_accessor :user
```

This is a nice and easy first step. This gives us access to the observe(object, key) method, which we'll use right now. In the loop over User::PROPERTIES, we're going to observe each property and update the value label accordingly. After we initialize the label, add the observe() method.

Third-Party Libraries and RubyMotion

RubyMotion does not currently support require in our source code files, so we need to use the RubyGems package manager (http://rubygems.org/) to bundle external libraries. RubyGems comes pre-installed on most versions of OS X, but you should visit the RubyGems website to download the latest version if necessary.

Installing RubyMotion-specific gems isn't any different from normal desktop gems: in your terminal, run gem install [gem name]. It will be downloaded like any other gem and can be required in your project's Rakefile; however, there will usually be an exception if you try to use it in non-RubyMotion apps. Since these are normal RubyGems, you can use Bundler (http://bundler.io/) to manage your gems like any other Ruby project.

models/UserProfile_kvo/app/controllers/user_controller.rb

```
value = UILabel.alloc.initWithFrame(CGRectZero)
value.text = self.user.send(prop)
observe(self.user, prop) do |old_value, new_value|
  value.text = new_value
  value.sizeToFit
end

value.sizeToFit
```

We pass it the object we want to observe (self.user) and the key we want updates about (prop). The callback block returns both the old and new values for the property, which could be useful if we were making more discriminatory UI updates.

Since the controller's title is originally set to the user's name, it's a nice idea to update that as the name changes.

models/UserProfile_kvo/app/controllers/user_controller.rb

```
self.title = self.user.name
  observe(self.user, "name") do |old_value, new_value|
    self.title = new_value
  end
```

Finally, we need to do a quick cleanup by overriding viewDidUnload(), one of the controller life-cycle methods. Since we're creating these observations in viewDidLoad(), we need to mirror their un-observing in the counterpart method.

models/UserProfile_kvo/app/controllers/user_controller.rb

```
def viewDidUnload
  unobserve_all
  super
end
```

Whew, all done. Let's play with our app and actually see the fruits of our labor. Go ahead and rake and get ready to use the debugger!

In the interactive debugger, grab the @user instance variable of our AppDelegate. BubbleWrap includes a nifty shortcut for grabbing the app's delegate, which we can now use.

```
(main)> user = App.delegate.instance_variable_get("@user")
=> #<NSKVONotifying_User @id="123", @email="clay@mail.com", @phone="555-555-5555">
```

Kind of a weird class name, isn't it? I won't get too technical, but under the hood KVO does a lot of tricks involving dynamically subclassing your observed objects. But it is proof that our user is being observed! So, let's make some changes:

```
(main)> user.email = "my_new_email@host.com"
=> "my_new_email@host.com"
(main)> user.name = "Charlie"
=> "Charlie"
```

In *Creating a New App*, we learned to make changes to the UI using the debugger, but now we're not even playing with the view objects! Everything just works. This is an incredibly powerful asset in your toolbox when it comes to making more complex, reactive apps.

So, now that we can make all these changes and synchronize the UI, how can we save them? If we quit the app right now and restart, our old "Clay" user will still be hanging around. And I personally have no problem with that, but most apps will want to save the user's changes.

Saving Data with NSUserDefaults and NSCoding

Applications generally have long-lasting consequences: we take a picture, create a presentation, or just unlock a new level. iOS will try to keep your app in memory for a reasonable amount of time, but eventually you need to permanently save something to the disk. There are several ways of doing this, ranging from writing files to using a SQLite database. We're going to use something in the middle: NSUserDefaults.

NSUserDefaults lets us persist basic objects (strings, numbers, arrays, and hashes) through a simple key-value interface. It handles the serialization mechanics for us, saving you the trouble of constructing a custom file serialization scheme. But it has one more trick up its sleeve: coupled with NSCoding, we can actually save arbitrary objects, not just the primitives classes.

An NSCoding-compliant object implements two specific methods that describe how to save and restore its properties into primitive objects. We can then take this collection of primitive objects and turn it into raw data, which NSUserDefaults understands. That all sounds a bit heady, so let's try implementing this fancy stuff in our app to get a better idea of what it can do.

First let's make our User NSCoding compliant. We need to add two new methods: initWithCoder(decoder) and encodeWithCoder(encoder). When we need to serialize and deserialize our object, these methods will be called. The objects they pass as arguments have simple APIs for retrieving and setting values. For our User, they look something like this:

models/UserProfile_persist/app/models/user.rb

```
def initWithCoder(decoder)
  self.init
  PROPERTIES.each do |prop|
    saved_value = decoder.decodeObjectForKey(prop.to_s)
    self.send("#{prop}=", saved_value)
  end
  self
end
def encodeWithCoder(encoder)
  PROPERTIES.each do |prop|
    encoder.encodeObject(self.send(prop), forKey: prop.to_s)
  end
end
```

initWithCoder(decoder) is called when we want to deserialize our object out of the decoder instance. Thus, we use decodeObjectForKey(key) on all of our PROPERTIES (see how that keeps making our life easier!).

Conversely, encodeWithCoder(encoder) gets called when we want to save our object and encode its properties with encodeObject:forKey:. The values and keys we use here are exactly those we use in initWithCoder:.

Our User can be encoded and decoded, but now what? We want to save and load the AppDelegate's @user, which requires us to use the NSUserDefaults. If we're using primitive types like strings or arrays, saving them directly just works:

```
defaults = NSUserDefaults.standardUserDefaults
defaults["some_array"] = [1,2,3]
defaults["some_number"] = 4

some_name = defaults["some_name"]
```

However, putting NSCoding objects such as @user into NSUserDefaults requires NSKeyedArchiver and NSKeyedUnarchiver. These two classes take NSCoding objects and transform them into instances of NSData, which can be safely stored or

retrieved from the defaults like normal. NSKeyedArchiver uses the encodeWithCoder: we implemented earlier, while NSKeyedUnarchiver uses initWithCoder:. That's a lot of NS prefixes, I know. Here's what an NSCoding serialization looks like:

```
my_object = # some NSCoding-compliant object
defaults = NSUserDefaults.standardUserDefaults
defaults["some_object"] = NSKeyedArchiver.archivedDataWithRootObject(my_object)

my_saved_data = defaults["some_object"]
my_saved_object = NSKeyedUnarchiver.unarchiveObjectWithData(my_saved_data)
```

That's going to get really tedious really fast to repeat everywhere for all our Users, so let's wrap it in some helper methods.

models/UserProfile_persist/app/models/user.rb

```
USER_KEY = "user"
def save
  defaults = NSUserDefaults.standardUserDefaults
  defaults[USER_KEY] = NSKeyedArchiver.archivedDataWithRootObject(self)
end
def self.load
  defaults = NSUserDefaults.standardUserDefaults
  data = defaults[USER_KEY]
  # protect against nil case
  NSKeyedUnarchiver.unarchiveObjectWithData(data) if data
end
```

Ah, much better. In a production app, our USER_KEY would probably be a function of the user's id, but since we have only one user in our app, it's not a big deal. All we have to do now is save and load our object when the app opens and closes.

models/UserProfile_persist/app/app_delegate.rb

```
@window = UIWindow.alloc.initWithFrame(UIScreen.mainScreen.bounds)
@user = User.load
@user ||= User.new(id: "123", name: "Clay",
    email: "clay@mail.com", phone: "555-555-5555")
@user_controller = UserController.alloc.initWithUser(@user)
```

models/UserProfile_persist/app/app_delegate.rb

```
def applicationDidEnterBackground(application)
  @user.save
end
```

In addition to application:didFinishLaunchingWithOptions:, the application delegate can respond to many more application life-cycle methods, similar to UIViewController. Apple recommends we save user data after the application has entered the background (as not to accidentally freeze the interface), so we use that callback to call @user.save.

Why don't you take our app for a spin, alter some @user properties, and then close our app? In fact, hold down the home button on the simulator and force-quit it with the red icon just to make sure we really got it. Open it back up, and your changes should reappear!

This small app had only one controller, but you can see how KVO and NSUserDefaults scales to more complex objects and relationships. But what about our user interfaces? How can we display hundreds or thousands of models on the screen and keep our app running smoothly? Well, that's where the incredibly versatile UITableView comes into play in *Showing Data with Table Views*.