MacRuby on iOS – RubyMotion review

Posted by Matt Aimonetti in ruby on May 4th, 2012

Yesterday, RubyMotion was released and let's be honest, it is one the best alternatives to Objective-C out there (if not the best). RubyMotion is a commercial, proprietary fork of MacRuby that targets iOS. This is not a small achievement, MacRuby relies on Objective C's Garbage Collector (libauto) which is not available on iOS. Static compilation and new memory management solution was required to target the iOS platform . The new runtime had to be small and efficient. Furthermore, being able to run code on iOS isn't enough, you need tools to interact with the compiler, to debug, to packages applications etc...

I don't think anyone will contest the fact that RubyMotion is a well done product. The question however is, "is it worth for you to invest some money, time and energy in this product instead of using Apple's language and tools". In this article, I'll try to balance the pros and cons of RubyMotion so you can have a better understanding of what RubyMotion could mean for you. As a disclaimer I should say that I was beta testing RubyMotion, that they are strong ties between RubyMotion and the MacRuby project I'm part of and finally that having MacRuby on iOS has been something I've been looking forward for a very long time.

Over the last few months I've seen RubyMotion take shape and finally hit the big 1.0. As you can see from Twitter and HackerNews, the Ruby community is excited about being able to use their language to write statically compiled, native iOS apps. Spoiler alert, they are right, it's a lot of fun.

What I like about RubyMotion:

Ruby Language

I don't mind Objective-C, I think it's a fine superset of C, with the arrival of blocks, new literals and automatic memory management via ARC, Objective-C is actually getting better over time. But frankly, it's not Ruby. You still have to deal with headers, you always have to compile your code via some weird Xcode voodoo settings, testing is a pain, the language, even with the new literals is quite verbose. On the other hand, using Ruby syntax I can get much more flexibility, reuse my code via mixins, easily reopen existing classes etc... At the end of the day, I end up with some code that seems cleaner, easier to

understand and maintain even though I'm calling the same underlying APIs. Ruby's flexibility also allows developers to make their own higher level APIs, take a look at some of the wrappers/helpers I wrote while playing with RubyMotion.

MacRuby

RubyMotion is based on MacRuby, meaning that all the time and energy invested in the project will benefit RubyMotion's users. All the concepts I explain in my MacRuby book apply to RubyMotion. You don't have to find workarounds to work with native APIs, Ruby objects are Objective-C objects and performance is great. I do regret Apple didn't decide to embrace MacRuby for iOS but at the same time, even though we lost the Open Source aspect of the project and Apple's backing, we gained much more flexibility and freedom on Laurent's part.

REPL/Interactive shell

RubyMotion doesn't currently have a debugger, but it does have something Objective-C developers don't have, a REPL working with the simulator. This feature is quite handy when debugging your application or learning the Cocoa APIs. You can click on a visual element in the simulator and start modifying the objects in real time in a terminal window and see the modifications in the simulator. It reminds me of the first time I used firebug to edit the html/css of a web page and saw the changes in real time.

Not dependent on Xcode

Xcode is fine when you write Objective-C code, but it crashes often, it has a complicated UI and never really worked well for MacRuby due to the fact that Objective-C and Ruby have different requirements and the that Xcode is not open source. It's also fully controlled by Apple and doesn't provide APIs for 3rd party developers. (That said, the Xcode team has often helped out when a new released of Xcode broke MacRuby, so thank you guys).
Being able to use simple rake tasks to compile, simulate and deploy applications is just really really nice. I'm sure we'll end up with better IDE integration, nice GUIs for some who like that, but in the meantime, as a "hacker", I really enjoy the simplicity of the Rake tasks and not being forced in using a specific IDE.

Memory management

Even though ARC made memory management much easier for

Objective-C developers, when using RubyMotion you don't have to worry about memory (well at least not explicitly, don't be dumb and create a bazillion objects and hold references to them either). This includes the CoreFoundation objects that you still have to manually manage in Objective-C. Memory management is transparent and in most cases it's really nice.

What I like less about RubyMotion

Here is a list of things that are cons to using RubyMotion, note that while the list is longer than my list of "pros", I listed a lot of small things. I also think that most of these issues will get solved in the next few months.

Ruby language

There are some cases where Ruby just isn't that great or is not an option. Examples include dealing with API relying heavily on pointers, when using some of the lower level APIs or when you have to interact with C++ (video game engines for instance). The good news is that within the same project, you can write part of your code in Objective-C and the rest in RubyMotion. The other thing that bothers me a little bit with writing Ruby code for iOS is that you can't easily enforce argument types and therefore you are losing a lot of the features provided by Clang to the Objective-C developers. I dream of an optionally typed Ruby — but that's a different topic.
Another downside of using Ruby is that Ruby developers will assume all standard libraries and gems will be compatible with RubyMotion. This isn't the case. You need to think of RubyMotion as only offering the Ruby syntax (modulo a few differences). To be honest, most of the std libs and gems aren't that useful when writing iOS apps. Even when I write MacRuby apps, I rarely rely on them and pick libraries designed to work in a non-blocking, multi-threaded environment (usually ObjC libs that I wrap).

Cocoa Touch

If you're already an iOS/OS X developer, you know that most of the hurdles aren't the language syntax but the Cocoa APIs. These APIs are what you need to interact with to create your application. Cocoa APIs are usually much lower-level compared to what you usually see in Python, Ruby or even Java. While they are quite consistent, the APIs still have a stiff learning curve and currently, if you want to write iOS

applications, even if you know Ruby, you still have to learn Cocoa. However, I do think that with RubyMotion now building a userbase, we will start seeing more and more wrappers around these sometimes hideous APIs.

No Xcode/IDE

There are cases where an IDE is really practical, especially when learning new APIs. Being able to have code completion, quick access to the documentation, instrumentation, debugging, interface builder, refactoring tools are things that Objective-C developers might have a hard time with when switching to RubyMotion. If you don't know either Ruby or Cocoa, getting started with RubyMotion might be quite hard and you are probably not currently in the target audience.

Writing UI code by hand

In some cases, it makes sense, in other, it should be much easier. I know that Laurent is working on a DSL to make that easier and I'm looking forward to it. But in the mean time, this is quite a painful exercise, especially due to the complexity of the Cocoa UI APIs. Using Xcode's interface builder and Storyboards is something I know a lot of us wish we could do with RubyMotion when developing specific types of applications.

No debugger

Again, this is eventually coming but the current lack of debugger can be problematic at times, especially when the problem isn't obvious.

Lack of clear target audience

It's hard to blame a brand new product for not having clearly defined a target audience. But as a developer I find myself wondering "when should I use RubyMotion and for what kinds of problems?" Is RubyMotion great for quick prototypes I can then turn into production code? Or is good for throw away prototypes? Is it reserved for "fart and flash light" applications? Is it ready for prime time and should I invest and write my new awesome apps using it? Should I convert over my existing code base over from Titanium (or whatever other alternatives you used)? Should I use RubyMotion every time I would use Objective-C?

I guess we will see when the first applications start hitting the app store and people start reporting on their experience.

Documentation

I'm partially to blame here since I could have moved my butt and start

writing a book but the point is nonetheless valid. All the iOS documentation out there is for Objective-C, all the APIs and samples provided by Apple are obviously only for Objective-C. Thankfully, you can use the 2 MacRuby books available out there to understand how to convert this existing documentation into something useful, but RubyMotion will need to provide better and more adapted documentation for beginners. I have no doubt that this is coming sooner than later.

Proprietary solution

RubyMotion isn't open source and currently fully relies on the shoulders of a single man. If unfortunately, Laurent goes out of business or decides to do something else then we will have to rewrite our apps in Objective-C. Using RubyMotion for a professional product represents a significant business risk, which is exactly the same as using proprietary technology from any vendor. Apple could also decide to switch to JavaScript or rewrite iOS in Java and deprecate Objective-C. Let's just say that it is unlikely.

I usually favor open source solutions, from the programming language I use to the OS I deploy on. This isn't always possible and if you want to write iOS applications, you don't currently have a choice. I do wish Laurent had found a way to make money while keeping the source code open. But who knows — after he makes his first million(s), he might change his mind.

Conclusion

I would strongly suggest you consider giving RubyMotion a try. I can assure you that it will provide at least a few hours of 'hacking fun' (and you will be able to brag about havng written your own iPhone app). It will also help support financially someone who's taking a risk in trying to push mobile development to the next level.

RubyMotion is, by far, my favorite alternative to Objective-C. But it is hard to tell, just 48 hours after its release, what people will do with it. Can it transcend the programming language barriers and attract Python, PHP, Java, ObjC and JavaScript developers? What is the sweet spot for RubyMotion applications? Will it affect the native vs web app battle? Can it make iOS development more accessible to the masses? Only time will tell.

What do you think?

iOS, ruby, RubyMotion

29 Comments

Introduction to mruby

Posted by Matt Aimonetti in ruby on April 27th, 2012

A couple days ago, I wrote an introduction article to help developers getting started with mruby (aka mrb).

Besides explaining the difference between mrb and the other implementations, the article shows concrete examples to embed Ruby inside a C software application. The article doesn't mention a few nice tricks such as mruby allowing you replace double by float (though still imperfect), the possibility to replace the memory allocator and it was even reported to me that mruby can run on the Lego Mindstorms platform which only has 250K of memory!
mruby is still in alpha stage but it's getting more interesting every day and at this rate it will soon become a real alternative to Lua.
Learn how to get started with mruby now.
mruby, ruby
1 Comment

new Ruby: mruby and mobiruby Ruby for iOS/Android

Posted by Matt Aimonetti in News, ruby on April 23rd, 2012

A few days ago, I wrote an article covering Ruby creator Matz' new Ruby implementation: mruby and its first related project: MobiRuby which aims to let Ruby developers write iOS and Android applications using their favorite language.

ruby
No Comments

Building and implementing a Single Sign-On solution

Posted by Matt Aimonetti in Software Design, Tutorial on April 4th, 2012

Most modern web applications start as a monolithic code base and, as complexity increases, the once small app gets split apart into many

“modules”. In other cases, engineers opt for a SOA design approach from the beginning. One way or another, we start running multiple separate applications that need to interact seamlessly. My goal will be to describe some of the high-level challenges and solutions found in implementing a Single-Sign-On service.

Authentication vs Authorization

I wish these two words didn’t share the same root because it surely confuses a lot of people. My most frequently-discussed example is OAuth. Every time I start talking about implementing a centralized/unified authentication system, someone jumps in and suggests that we use OAuth. The challenge is that OAuth is an authorization system, not an authentication system.

It’s tricky, because you might actually be “authenticating” yourself to website X using OAuth. What you are really doing is allowing website X to use your information stored by the OAuth provider. It is true that OAuth offers a pseudo-authentication approach via its provider but that is not the main goal of OAuth: the Auth in OAuth stands for Authorization, not Authentication.

Here is how we could briefly describe each role:

- Authentication: recognizes who you are.
- Authorization: know what you are allowed to do, or what you allow others to do.

If you are feel stuck in your design and something seems wrong, ask yourself if you might be confused by the 2 auth words. This article will only focus on authentication.

A Common Scenario

This is probably the most common structure, though I made it slightly more complex by drawing the three main apps in different programming languages. We have three web applications running on different subdomains and sharing account data via a centralized authentication service.

Goals:

- Keep authentication and basic account data isolated.
- Allow users to stay logged in while browsing different apps.

Implementing such a system should be easy. That said, if you migrate an existing app to an architecture like that, you will spend 80% of your time decoupling your legacy code from authentication and wondering what data should be centralized and what should be distributed. Unfortunately, I can’t tell you what to do there since this is very domain specific. Instead, let’s see how to do the “easy part.”

Centralizing and Isolating Shared Account Data

At this point, you more than likely have each of your apps talk directly to shared database tables that contain user account data. The first step is to migrate away from doing that. We need a single interface that is the only entry point to create or update shared account data. Some of the data we have in the database might be app specific and therefore should stay within each app, anything that is shared across apps should be moved behind the new interface.
Often your centralized authentication system will store the following information:

- ID
- first name
- last name
- login/nickname
- email
- hashed password
- salt
- creation timestamp
- update timestamp
- account state (verified, disabled ...)

Do not duplicate this data in each app, instead have each app rely on the account ID to query data that is specific to a given account in the app. Technically that means that instead of using SQL joins, you will query your database using the ID as part of the condition.
My suggestion is to do things slowly but surely. Migrate your database schema piece by piece assuring that everything works fine. Once the other pieces will be in place, you can migrate one code API a time until your entire code base is moved over. You might want to change your DB credentials to only have read access, then no access at all.

Login workflow

Each of our apps already has a way for users to login. We don't want to change the user experience, instead we want to make a transparent modification so the authentication check is done in a centralized way instead of a local way. To do that, the easiest way is to keep your current login forms but instead of POSTing them to your local apps, we'll POST them to a centralized authentication API. (SSL is strongly recommended)

As shown above, the login form now submits to an endpoint in the authentication application. The form will more than likely include a

login or email and a clear text password as well as a hidden callback/redirect url so that the authentication API can redirect the user's browser to the original app. For security reasons, you might want to white list the domains you allow your authentication app to redirect to. Internally, the Authentication app will validate the identifier (email or login) using a hashed version of the clear password against the matching record in the account data. If the verification is successful, a token will be generated containing some user data (for instance: id, first name, last name, email, created date, authentication timestamp). If the verification failed, the token isn't generated. Finally the user's browser is redirected to the callback/redirect URL provided in the request with the token being passed.
You might want to safely encrypt the data in a way that allows the clients to verify and trust that the token comes from a trusted source. A great solution for that would be to use RSA encryption with the public key available in all your client apps but the private key only available on the auth server(s). Other strong encryption solutions would also work. For instance, another appropriate approach would be to add a signature to the params sent back. This way the clients could check the authenticity of the params. HMAC or DSA signature are great for that but in some cases, you don't want people to see the content of the data you send back. That's especially true if you are sending back a 'mobile' token for instance. But that's a different story. What's important to consider is that we need a way to ensure that the data sent back to the client can't be tampered with. You might also make sure you prevent replay attacks.
On the other side, the application receives a GET request with a token param. If the token is empty or can't be decrypted, authentication failed. At that point, we need to show the user the login page again and let him/her try again. If on the other hand, the token can be decrypted, the content should be saved in the session so future requests can reuse the data.
We described the authentication workflow, but if a user logins in application X, (s)he won't be logged-in in application Y or Z. The trick here, is to set a top level domain cookie that can be seen by all applications running on subdomains. Certainly, this solution only works for apps being on the same domain, but we'll see later how to handle apps on different domains.

The cookie doesn't need to contain a lot of data, its value can contain the account id, a timestamp (to know when authentication happened and a trusted signature) and a signature. The signature is critical here since this cookie will allow users to be automatically logged in other sites. I'd recommend the HMAC or DSA encryptions to generate the

signature. The DSA encryption, very much like the RSA encryption is an asymmetrical encryption relying on a public/private key. This approach offers more security than having something based a shared secret like HMAC does. But that's really up to you.
Finally, we need to set a filter in your application. This auto-login filter will check the presence of an auth cookie on the top level domain and the absence of local session. If that's the case, a session is automatically created using the user id from the cookie value after the cookie integrity is verified. We could also share the session between all our apps, but in most cases, the data stored by each app is very specific and it's safer/cleaner to keep the sessions isolated. The integration with an app running on a different service will also be easier if the sessions are isolated.

Registration

For registration, as for login, we can take one of two approaches: point the user's browser to the auth API or make S2S (server to server) calls from within our apps to the Authentication app. POSTing a form directly to the API is a great way to reduce duplicated logic and traffic on each client app so I'll demonstrate this approach.

As you can see, the approach is the same we used to login. The difference is that instead of returning a token, we just return some params (id, email and potential errors). The redirect/callback url will also obviously be different than for login. You could decide to encrypt the data you send back, but in this scenario, what I would do is set an auth cookie at the .domain.com level when the account is created so the "client" application can auto-login the user. The information sent back in the redirect is used to re-display the register form with the error information and the email entered by the user.
At this point, our implementation is almost complete. We can create an account and login using the defined credentials. Users can switch from one app to another without having to re login because we are using a shared signed cookie that can only be created by the authentication app and can be verified by all "client" apps. Our code is simple, safe and efficient.

Updating or deleting an account

The next thing we will need is to update or delete an account. In this case, this is something that needs to be done between a "client" app and the authentication/accounts app. We'll make S2S (server to server) calls. To ensure the security of our apps and to offer a nice way to log requests, API tokens/keys will be used by each client to communicate

with the authentication/accounts app. The API key can be passed using a X-header so this concern stays out of the request params and our code can process separately the authentication via X-header and the actual service implementation. S2S services should have a filter verifying and logging the API requests based on the key sent with the request. The rest is straight forward.

Using different domains

Until now, we assumed all our apps were on the same top domain. In reality, you will often find yourself with apps on different domains. This means that you can't use the shared signed cookie approach anymore. However, there is a simple trick that will allow you to avoid requiring your users to re-login as they switch apps.

The trick consists, when a local session isn't present, of using an iframe in the application using the different domain. The iframe loads a page from the authentication/accounts app which verifies that a valid cookie was set on the main top domain. If that is the case, we can tell the application that the user is already globally logged in and we can tell the iframe host to redirect to an application end point passing an auth token the same way we did during the authentication. The app would then create a session and redirect the user back to where (s)he started. The next requests will see the local session and this process will be ignored.
If the authentication application doesn't find a signed cookie, the iframe can display a login form or redirect the iframe host to a login form depending on the required behavior.
Something to keep in mind when using multiple apps and domains is that you need to keep the shared cookies/sessions in sync, meaning that if you log out from an app, you need to also delete the auth cookie to ensure that users are globally logged out. (It also means that you might always want to use an iframe to check the login status and auto-logoff users).

Mobile clients

Another part of implementing a SSO solution is to handle mobile clients. Mobile clients need to be able to register/login and update accounts. However, unlike S2S service clients, mobile clients should only allow calls to modify data on the behalf of a given user. To do that, I recommend providing opaque mobile tokens during the login process. This token can then be sent with each request in a X-header so the service can authenticate the user making the request. Again, SSL

is strongly recommended.
In this approach, we don't use a cookie and we actually don't need a SSO solution, but an unified authentication system.

Writing web services

Our Authentication/Accounts application turns out to be a pure web API app.
We also have 3 sets of APIs:

- Public APIs: can be accessed from anywhere, no authentication required
- S2S APIs: authenticated via API keys and only available to trusted clients
- Mobile APIs: authenticated via a mobile token and limited in scope.

We don't need dynamic HTML views, just simple web service related code. While this is a little bit off topic, I'd like to take a minute to show you how I personally like writing web service applications.
Something that I care a lot about when I implement web APIs is to validate incoming params. This is an opinionated approach that I picked up while at Sony and that I think should be used every time you implement a web API. As a matter of fact, I wrote a Ruby DSL library (Weasel Diesel) allowing you describe a given service, its incoming params, and the expected output. This DSL is hooked into a web backend so you can implement services using a web engine such as Sinatra or maybe Rails3. Based on the DSL usage, incoming parameters are be verified before being processed. The other advantage is that you can generate documentation based on the API description as well as automated tests.
You might be familiar with Grape, another DSL for web services. Besides the obvious style difference Weasel Diesel offers the following advantages:

- input validation/sanitization
- service isolation
- generated documentation
- contract based design

Here is a hello world webservice being implemented using Weasel Diesel and Sinatra:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
23 24 25 26 27 28 29
describe_service "hello_world" do |service|
  service.formats   :json
  service.http_verb :get
  service.disable_auth # on by default
```

```
  # INPUT
  service.param.string  :name, :default => 'World'

  # OUTPUT
  service.response do |response|
    response.object do |obj|
        obj.string :message, :doc => "The greeting
message sent back. Defaults to 'World'"
      obj.datetime :at, :doc => "The timestamp of when
the message was dispatched"
      end
  end

  # DOCUMENTATION
  service.documentation do |doc|
    doc.overall "This service provides a simple hello
world implementation example."
    doc.param :name, "The name of the person to greet."
    doc.example "<code>curl -I 'http://localhost:9292/
hello_world?name=Matt'</code>"
 end

  # ACTION/IMPLEMENTATION
  service.implementation do
    {:message => "Hello #{params[:name]}", :at =>
Time.now}.to_json
  end

end
```

view raw

hello_world.rb

This Gist brought to you by GitHub.

Basis test validating the contract defined in the DSL and the actual output when the service is called:

```
1 2 3 4 5 6 7 8
class HelloWorldTest < MiniTest::Unit::TestCase

  def test_response
    TestApi.get "/hello_world", :name => 'Matt'
    assert_api_response
  end

end
```

view raw

gistfile1.rb
This Gist brought to you by GitHub.
Generated documentation:

If the DSL and its features seem appealing to you and you are interested in digging more into it, the easiest way is to fork this demo repo and start writing your own services.
The DSL has been used in production for more than a year, but there certainly are tweaks and small changes that can make the user experience even better. Feel free to fork the DSL repo and send me Pull Requests.
architecture, crypto, SSO
25 Comments

Learning from Rails' failures

Posted by Matt Aimonetti in Misc, rails, Software Design on February 29th, 2012

Ruby on Rails undisputedly changed the way web frameworks are designed. Rails became a reference when it comes to leveraging conventions, easy baked in feature set and a rich ecosystem. However, I think that Rails did and still does a lot of things pretty poorly. By writing this post, I'm not trying to denigrate Rails, there are many other people out there already doing that. My hope is that by listing what I think didn't and still doesn't go well, we can learn from our mistakes and improve existing solutions or create better new ones.

Migration/upgrades

Migrating a Rails App from a version to the other is very much like playing the lottery, you are almost sure you will lose. To be more correct, you know things will break, you just don't know what, when and how. The Rails team seems to think that everybody is always running on the cutting edge version and don't consider people who prefer to stay a few version behind for stability reasons. What's worse is that plugins/gems might or might not compatible with the version you are updating to, but you will only know that by trying yourself and letting others try and report potential issues.
This is for me, by far, the biggest issue with Rails and something that should have been fixed a long time ago. If you're using the WordPress blog engine, you know how easy and safe it is to upgrade the engine or the plugins. Granted WordPress isn't a web dev framework, but it gives you an idea of what kind of experience we should be striving for.

Stability vs playground zone

New features are cool and they help make the platform more appealing to new comers. They also help shape the future of a framework. But from my perspective, that shouldn't come to the cost of stability. Rails 3's new asset pipeline is a good example of a half-baked solution shoved in a release at the last minute and creating a nightmare for a lot of us trying to upgrade. I know, I know, you can turn off the asset pipeline and it got better since it was first released. But shouldn't that be the other way around? Shouldn't fun new ideas risking the stability of an app or making migration harder, be off by default and turned on only by people wanting to experiment? When your framework is young, it's normal that you move fast and sometimes break, but once it matures, these things shouldn't happen.

Public/private/plugin APIs

This is more of a recommendation than anything else. When you write a framework in a very dynamic language like Ruby, people will "monkey patch" your code to inject features. Sometimes it is due to software design challenges, sometimes it's because people don't know better. However, by not explicitly specifying what APIs are private (they can change at anytime, don't touch), what APIs are public (stable, will be slowly deprecated when they need to be changed) and which ones are for plugin devs only (APIs meant for instrumentation, extension etc..), you are making migration to newer versions much harder. You see, if you have a small, clean public API, then it's easy to see what could break, warn developers and avoid migration nightmares. However, you need to start doing that early on in your project, otherwise you will end up like Rails where all code can potentially change anytime.

Rails/Merb merge was a mistake

This is my personal opinion and well, feel free to disagree, nobody will ever be able to know to for sure. Without explaining what happened behind closed doors and the various personal motivations, looking at the end result, I agree with the group of people thinking that the merge didn't turn up to be a good thing. For me, Rails 3 isn't significantly better than Rails 2 and it took forever to be released. You still can't really run a mini Rails stack like promised. I did hear that Strobe (company who was hiring Carl Lerche, Yehuda Katz and contracted Jose Valim) used to have an ActionPack based, mini stack

but it was never released and apparently only Rails core members really knew what was going on there. Performance in vanilla Rails 3 are only now getting close to what you had with Rails 2 (and therefore far from the perf you were getting with Merb). Thread-safety is still OFF by default meaning that by default your app uses a giant lock only allowing a process to handle 1 request at a time. For me, the flexibility and performance focus of Merb were mainly lost in the merge with Rails. (Granted, some important things such as ActiveModel, cleaner internals and others have made their way into Rails 3)
But what's worse than everything listed so far is that the lack of competition and the internal rewrites made Rails lose its headstart. Rails is very much HTML/view focused, its primarily strength is to make server side views trivial and it does an amazing job at that. But let's be honest, that's not the future for web dev. The future is more and more logic pushed to run on the client side (in JS) and the server side being used as an API serving data for the view layer. I'm sorry but adding support for CoffeeScript doesn't really do much to making Rails evolve ahead of what it currently is. Don't get me wrong, I'm a big fan of CoffeeScript, that said I still find that Rails is far from being optimized to developer web APIs in Rails. You can certainly do it, but you are basically using a tool that wasn't designed to write APIs and you pay the overhead for that. If there is one thing I wish Rails will get better at is to make writing pure web APIs better (thankfully there is Sinatra). But at the end of the day, I think that two projects with different philosophies and different approaches are really hard to merge, especially in the open source world. I wouldn't go as far as saying like others that Rails lost its sexiness to node.js because of the wasted time, but I do think that things would have been better for all if that didn't happen. However, I also have to admit that I'm not sure how much of a big deal that is. I prefer to leave the past behind, learn from my own mistake and move on.

Technical debts

Here I'd like to stop to give a huge props to Aaron "@tenderlove" Patterson, the man who's actively working to reduce the technical debts in the Rails code base. This is a really hard job and definitely not a very glamorous one. He's been working on various parts of Rails including its router and its ORM (ActiveRecord). Technical debts are unfortunately normal in most project, but sometimes they are overwhelming to the point that nobody dares touching the code base to clean it up. This is a hard problem, especially when projects move fast like Rails did. But looking back, I think that you want to start tackling technical debts on the side as you move on so you avoid

getting to the point that you need a hero to come up and clean the piled errors made in the past. But don't pause your entire project to clean things up otherwise you will lose market, momentum and excitement. I feel that this is also very much true for any legacy project you might pick up as a developer.

Keep the cost of entry level low

Getting started with Rails used to be easier. This can obviously argued since it's very subjective, but from my perspective I think we forgot where we come from and we involuntary expect new comers to come with unrealistic knowledge. Sure, Rails does much more than it used to do, but it's also much harder to get started. I'm not going to argue how harder it is now or why we got there. Let's just keep in mind that it is a critical thing that should always be re-evaluated. Sure, it's harder when you have an open source project, but it's also up to the leadership to show that they care and to encourage and mentor volunteers to focus on this important part of a project.

Documentation

Rails documentation isn't bad, but it's far from being great. Documentation certainly isn't one of the Ruby's community strength, especially compared with the Python community, but what saddens me is to see the state of the official documentation which, should, in theory be the reference. Note that the Rails guides are usually well written and provide value, but they too often seem too light and not useful when you try to do something not totally basic (for instance use an ActiveModel compliant object). That's probably why most people don't refer to them or don't spend too much time there. I'm not trying to blame anyone there. I think that the people who contributed theses guides did an amazing job, but if you want to build a strong and easy to access community, great documentation is key. Look at the Django documentation as a good example. That said, I also need to acknowledge the amazing job done by many community members such as Ryan Bates and Michael Hartl consistently providing high value external documentation via the railscasts and the intro to Rails tutorial available for free.

In conclusion, I think that there is a lot to learn from Rails, lots of great things as well as lots of things you would want to avoid. We can certainly argue on Hacker News or via comments about whether or not I'm right about Rails failures, my point will still be that the mentioned issues should be avoided in any projects, Rails here is just an example.

Many of these issues are currently being addressed by the Rails team but wouldn't it be great if new projects learn from older ones and avoid making the same mistakes? So what other mistakes do you think I forgot to mention and that one should be very careful of avoiding?

Updates:

1. Rails 4 had an API centric app generator but it was quickly reverted and will live as gem until it's mature enough.
2. Rails 4 improved the ActiveModel API to be simpler to get started with. See this blog post for more info.

django, documentation, merb, migration, open source, rails, ruby
36 Comments