Issue 01

DevMonologue Magazine

Going Open Source

In this issue

Going Open source

Editorial

About this issue

This issue marks the very first publication for the Dev Monologue website, as well as for me as an author. I have no experience with publishing so be patient with me. The reason why I chose to switch to this format, instead of traditional blogging is that I believe it is more appealing and brings nostalgic for the past days of magazines and newspapers. Also, a whole issue is an opportunity to dig deeper into a subject and write several articles that complete each other, which is not something I found to be working very well with blog posts. So... without further ado... going open source!

The open source world

There's a lot of discussion about open source software on the internet. It is gaining more and more positive publicity among programmers and IT guys alike. And of course it is an excellent way to both share knowledge and improve coding efficiency and quality and the same time. As a programmer, I'm lucky enough to appreciate open source both from a developer perspective, and as a consumer.

Before I turned to the dark side of Mac OS, I was (and still am) a strong supporter of Linux. And as a linux user, I could experience the open source phenomenon as a consumer. I never got any benefit from being able to read my kernel's code, but I could still see the power of the community. A software made from the same people that use it. No corporate nonsense. It is amazing... (and/or I'm nerdy enough to appreciate it).

Naturally, talking about open source, kernel and community at the same time, I cannot ignore the notoriously hostile environment of contributing to the Linux kernel. To be honest I think it's a real shame what's happening there. And whereas I can totally understand a maintainer or an experienced developer's frustration with the code of lesser programmers, I cannot justify such aggressive behavior. Hopefully it gets better with time.

Anyway, such large and infamous projects aside, there are many thousands of small chunks of code floating around in GitHub, just begging for your attention. And that's also amazing. You don't have to be a kernel developer, you don't have to work for RedHat or own a company that managed to have a successful open source business plan. If you worked on something cool and thought someone might like it, just put it out there. It might get popular, it might just get 5-6 stars on GitHub, or not get viewed by anyone at all. Who cares? You're still part of the open source community.

> **“How can you make money when your product is just out there for anyone to see?”**

An obvious question outside people ask about open source software is "How can you make money when your product is just out there for anyone to see?". It's surely a fair question. But you should also consider that software is inherently a difficult intellectual property to protect. It's just a piece of information that you have to distribute to people. And no matter how good you obfuscate it, it's still possible to read to some extent. Cracks and pirated software are a hard evidence supporting that.

So, by being in the software business, you already need to be prepared to

have other people "see" what you do. With open source, you just openly acknowledge that.
And quite frankly, the proverb that you should always hide in plain sight, also hold true. Just take a look at virtually any security/crypto algorithm. They are all open source. Because the power or the community outweighs the threat of exposing too much information.

In recent times, the business model for software companies has shifted away from the actual code. Especially on mobile, the software is not build to make money. It is build so that it popularizes other services that make the money. For instance, the Wordpress mobile clients are 100% open source. And without a doubt that cannot hurt revenues, because no one pays for these apps, they are just designed for easier access to blogging for Wordpress users. And nowadays, so many companies have the same situation - they don't make money from apps and they don't contain any trade secrets, so open sourcing cannot hurt them.

This brings us to another point. Who pays for open source? Obviously you cannot open source everything - there should be something else in your business plan that cannot be easily replicated just by seeing your code. For many core linux projects, it's actually companies like Canonical and RedHat that provide consulting services and/or need the linux system to be reliable enough for their other ventures. Also with the rise of "The cloud", it's easy to open source many of your software components but still provide them as a hosted service (like Wordpress).

All this reminds me of a very good article by Jonathan Zdziarski in his personal blog - Free Software Always Costs Something.

Why I chose this topic

As I'm writing this, I'm about to release the code of one of my pet projects into the wild. For a long time, it was in a private repo, but since it is never going to make money or make me famous, why not publish it. It's probably not going to be very useful to people, but maybe someday, someone is going to see it and check out the code to see how I solved some problem, he or she is having. Maybe...who knows... anything can happen in this crazy world.

But before actually releasing the kraken, there are several things to consider. What license should I use... how do I handle dependancies, how can I transfer the repository from one place to another most easily... What do I write in the README...?

Open source licenses

This article is about the different available open source licenses and how to choose the one you're comfortable with.

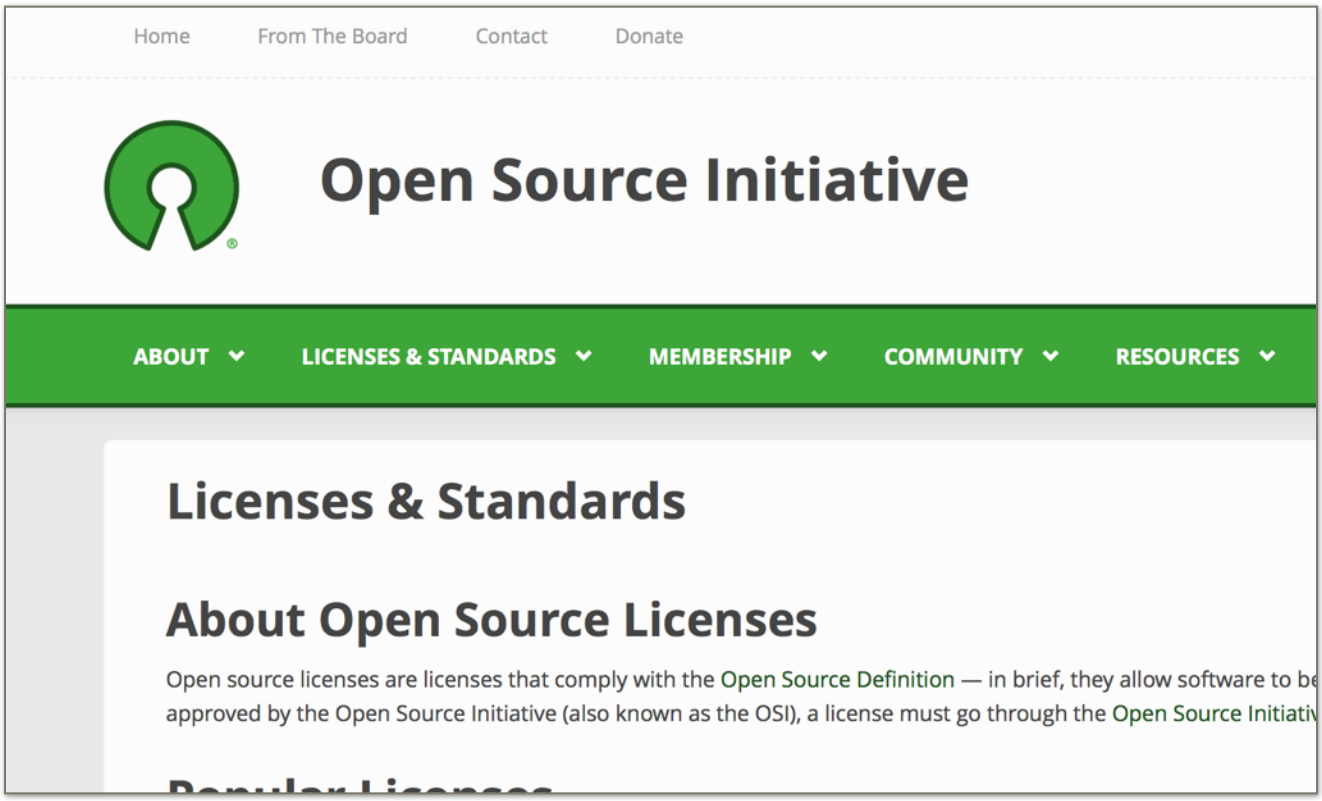
Licensing

As a developer, trying to give something back to the community, the first encounter with open source licenses can be overwhelming. There seems to be an endless selection to choose from and it can sometimes be difficult to distinguish between options. Of course, you can always make your own license, but I'm going to assume most people don't have the expertise and/or the desire to do so. Leave it to corporations.

The number one thing you should be interested in when choosing a license is how open you want your code to be. More specifically, what are the rules others need to follow if they want to use and/or modify your code.

Commonly used licenses

What better way to explore the world of open source licenses, than going back to the origin - The Open Source Initiative. Heading over to the licenses section of this website, we can see a number of "popular" options.



Here's the list at the time of writing:

- Apache License 2.0
- BSD 3-Clause "New" or "Revised" license
- GNU General Public License (GPL)
- GNU Library or "Lesser" General Public License (LGPL)
- MIT license
- Mozilla Public License 2.0
- Common Development and Distribution License
- Eclipse Public License

The GPL logo, silently judging you when you don't open source your code

What you might find strange here is that you can see company names. As mentioned, you are free to create your own terms, if you have the money and expertise to do so. So some big companies, like Mozilla in this case, do that, but for most people choosing an existing one is fine.

Choosing a license

If you read around the OSI website a little bit, and open the page for some of the licenses shown above, you will notice that, even though it is legal text, it's not really hard to read. And quite short. For instance, the Apache License has 9 relatively short sections. So I encourage every single one of you to actually read at least the most popular ones.

When choosing a license, there are mostly several things to think about.

First, and definitely most important, are the rules for using the code (and any documentation, configurations files and whatever else you have there). What are the rules, other developers need to obey, if they want to use your hard work. Surely, since you open sourced your code, you welcome people to enjoy it, but still you need to decide how open you want to be about this. For instance, the MIT license, which is a popular choice for "liberal" developers, permits almost every kind of usage scenario. The only thing you cannot do, is claim the code is actually yours. On the other hand, the GNU General Public License allows you to use and change the source, but only if you, in turn share your code and any changes you might have made. So basically this limits the projects appeal for commercial products. It is a fairly popular choice in the Linux world, where people promote free and open software.

Most of the other text within a open source software license is about liability and contribution. Here, there is less to choose since most options are relatively the same. They tend to disclaim any sort of liability obligations. And for good reason. You were happy to share your work with the world, but do you really want to be held responsible for problems that might arise in other people's projects? Probably not. So OSI comes to the rescue and makes sure you don't run into those kinds of problems.

Another liability topic, licenses handle in a similar way, are contributions. If you share your code with others, the best this that could happen is to start getting code back in the form if bug fixes and features. And surely, you think that's great. But what does your lawyer think? What if tomorrow someone claims that their code is

in your project so they should have a piece of the fame? Thankfully, open source licenses don't allow that. All popular ones claim that contributions that you might get are free of charge and are yours for the taking.

> *Something cool I found in some licenses (The Apache), is section 9 called "Accepting Warranty or Additional Liability". It allows you to bring support and consulting to your business model for open source projects. This is something that many companies like Canonical and RedHat already do. It means that whereas the code is free, not providing any type or warranty, and might actually not allow you to used in commercial project (not the case with Apache), you can still charge money in exchange for support and accepting warranty.*

So the elephant in the room of choosing a license is actually what we discussed in the first paragraph - do you want people to start using your code for profit. It's to some degree a philosophical question, but also your decision might change from project to project. What I'm seeing in other people, and it also happens to be my personal opinion, is that for small projects and pieces of code, you should choose a more "liberal" license. The MIT license for instance, is a great choice in that matter. The truth is, if your project is just a few files of code, who cares if it's used by an evil corporation. And even if you protect it with a strict license, like GPL (General Public License), the code base is so small, a developer can re-write it base on your ideas and solutions. For bigger projects, however, let's say for frameworks and whole applications, you might have to weigh in your options more carefully. If you're building this product just for fun, think about it in a "political" way. Do you accept that people might be profiting using your work, without giving anything back? Are you a strong supporter of open source? And if you're trying to run a business with that code base, can other companies leverage that in order to gain a competitive advantage over you? Can they use your code, extend it a little without in turn sharing their modifications, to take the upper-hand? If so, then a more permissive license might not be a good choice for you.

Anyway, based on how liberal they are towards commercial use, the popular licenses you can see on OSI's website, can be divided in two:

Open to commercial use

- Apache License 2.0
- BSD license
- MIT license
- Mozilla Public License 2.0
- Common Development and Distribution License
- Eclipse Public License

Open source only

- GNU General Public License (GPL)

Note: Saying "Open to commercial use", doesn't mean you cannot use a GPL license for commercial products. It just means that if you incorporate a GPL licensed project in your own, you need to also open source any modification you make to the code base and the way it's used. Surely, there a lot of loopholes, but generally, that's the rule. And that's what makes it often unsuitable for commercial products.

Digging deeper, beyond distribution terms, most licenses seem similar to the common programmer. I encourage you to read those licenses to see how they differ, but generally, after you choose between those two sections above, you're almost ready.

But before we move on to "Applying the license", there are several interesting topics many licenses address. One of them, that addresses the so called "privacy loophole", makes sure cloud services cannot bypass their responsibility towards your code. Since many options are fairly old (GPL v1 for instance was created in 1989), they don't address hosted services. Many companies go around strict licenses by saying that they don't distribute the protected source code - they just deploy it once internally, and then provide it to users as a SaaS (Software as a Service). Since this is not exactly fair and sometimes unacceptable for the authors, several licenses address that issue, namely:

- Affero GNU Public License
- Common Public Attribution License 1.0
- Non-Profit Open Software License 3.0
- Open Software License 3.0
- Reciprocal Public License 1.5

Another interesting topic is the way third parties should acknowledge their use of open source software. Normally, the rule is that you should distribute a copy of the license with your product, but that doesn't mean it needs to be easily accessible. However, authors might choose a license that makes other developers acknowledge their use of the open source code in a user accessible place within their software (most likely somewhere in the UI). That's called **enhanced attribution**. This time the options are:

- Adaptive Public License
- Affero GNU Public License
- Attribution Assurance Licenses
- Common Public Attribution License 1.0
- Reciprocal Public License 1.5

Finally, a consideration that I wish anyone of us had, is the **no promotion** feature. It protects authors from being used as advertisement for a product. It forbids third parties to use the author's name to promote their business.

What you go for is largely a matter of choice. For instance, if I wanted to just share some code with other people, I'd use MIT or BSD, since they are so simple and short.

And if I ever release a framework or an application I want to protect better, I'd go for GPL. However, as a disclaimer, I'd like to say I'm no expert so I might be missing something. For best results, just read the different options and make your own choice.

Where to go from here?

An excellent resource for choosing between open source licenses, is the OSS Watch License differentiator. There, you can input up to 7 parameters for your particular case and it will give you a list of licenses that are suitable for what you are trying to achieve. I recommend that you go check it out. Just explore a little, start playing with the values and you'll become a open source pro!

Applying a license

When I wanted to release some code "to the wild" for the first time, and I thought about licensing, I started wandering how I can obtain one. Surely, if you want a legally binding document for your project, you need to talk to someone. At that point, several years ago, I remember it wasn't plainly obvious how you can apply a license to your code. A simple search didn't yield any useful answers immediately. But soon enough, I found it. The only thing you need to do is... specify which license you're using somewhere in your project.

Also, if you look into any of the license pages at OSI, you will see at the bottom a appendix explaining how to apply the license to your code.

> APPENDIX: How to apply the Apache License to your work

> To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

```
Copyright [yyyy] [name of copyright owner]
Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

     http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```

I guess at some point GitHub realized as a major problem that developers didn't know to apply a license to their code and added a license selection step to their "new project" wizard.

To summarize, the only thing you need to do in oder to apply a license to your project is to add that text somewhere in your code. Most of the time, it means adding a LICENSE.txt file to your project and maybe add a comment to the beginning of your source code files. For the actual text, refer to your license of choice's webpage.

Importing other people's code

What about the opposite? If you had a library you wanted to use, what do you need to do to acknowledge the authors?
Well, normally you need to specifically state that you're using someone else's code. And also add their “LICENSE” and “NOTICE” files. However, not many people realize, this doesn't need to be that complicated.

As already discussed above, as long as the code is not protected by a license with an **enhanced attribution** policy, all the law says is that these things need to be shipped with the actual product. So for mobile application, it is enough to put those files in the package that you upload to (and people download from) the Store. Yes, it's not easily accessible, but a copy of all that is included in the product. Also, many applications choose not to include the actual acknowledgements for open source code that they are using, but to provide a link to their website where they enumerate all the projects and authors.

iOS/Mac open source world

In this article, we will delve deeper into the world of open source projects, specifically for iOS and Mac. And by that, I mean mostly iOS. On Mac, many of the really good open source applications are actually cross platform and more native to Linux than Mac OS.

Going to GitHub and starting a search for either Swift or Objective-C yields mostly frameworks and chunks of reusable code. Whereas this is really useful to other developers for learning and sharing, here we will focus on complete applications (with some exceptions). Something you can actually run and see.

Lets get started and see some prime examples of the Apple open source community:

Artsy

Artsy is definitely an exemplary company in regards to open source. Most of what they do is out in the open. It's always on GitHub and it's always free and open source. They even use the MIT license, so you are free to do whatever you like with their code.

It probably takes a lot of courage to let go of all your intellectual property like Artsy does, but I guess it worked out in the end. Even regarding this kind of structural aspect of the company, they are really open to sharing. You can find all kinds of information on their blog about virtually any aspect of their operation.

> ***COMPANIES REVOLVE AROUND IDEAS, AND UNDERSTANDING WHAT YOUR CORE VALUE IS IMPORTANT. A COMPANY WHO MAKE MONEY PURELY OFF SELLING THEIR APPS COULD BE EASILY COPIED, AND OSS BY DEFAULT WON'T WORK FOR THEM. ARTSY IS A PLATFORM, BUT OSS BY DEFAULT CAN WORK FOR US BECAUSE A TECHNICAL PLATFORM IS JUST ONE ASPECT OF WHAT WE OFFER.***
>
> **— OSS expectations, Artsy blog**

For example, you can find their thoughts on going open source in this article. They bring up a lot of good points regarding the threats and opportunities of open source software. Having a management team with a technical background definitely helps selling the idea, but the text makes a very good argument about it.

This is a really solid point, especially regarding mobile. Users are expecting to get apps for free and paid applications are getting less and less popular. Most companies don't really expect revenue from the App Stores. Their applications are solely there to complement their business. So why hide it? It's a code base specifically tailored for a proprietary service in mind and doesn't give much (if any) advantage to your competition. Unfortunately, looking at the majority of apps in the AppStore, most companies don't seem to share my vision.

Another great contribution from Artsy to the open source world, is the way they segment their code into libraries and more specifically how they manage those library's lifecycle. Getting a chunk of code and making it reusable by turning it into a library is a great way to encourage other developers to integrate it for their own projects. But you can also say it's more about code quality than being open and sharing knowledge. And that's surely true. By no means is open sourcing only benefiting others. By thinking about the community, you also better structure your own code. And that by itself is quite powerful. This is actually something the Artsy team talked about in their How we Open Source'd Eigen article.

> ***3 MONTHS ON THE WAY WE OPERATE HAS CHANGED. WE'RE A LOT MORE ORGANIZED, AND THE EIGEN REPO IS EASILY THE MOST WELL RUN PROJECT ON THE MOBILE TEAM. IT HAS ACTIVE MILESTONES, THAT REPRESENT LONG TERM GOALS AND THE CURRENT SPRINT. WE DISCUSS A LOT OF THE INTERESTING CULTURAL CHOICES PUBLICLY ON ISSUES AND IN OUR MOBILE TEAM REPO. HAVING THIS APP IN THE OPEN, AND THE EXPERIENCE OF DOING SO HAS ALSO IMPROVED OUR WORKFLOW ON OTHER APPS.***
>
> **— How we Open Source'd Eigen post, Artsy blog**

I guess people start working differently if they know someone could be watching. It's probably the same difference between working alone in a dark room with your monitor facing the wall and working in the middle of a huge open office with everyone looking into your screen. It's just harder to hide something embarrassing so you tend to avoid it.

But going back to extracting code into frameworks, Artsy does something that would make many managers cringe. When a developer commits to creating a library out of some pile of code, they do so in their own name in GitHub. Effectively, this developer owns the code, not the company. It sounds strange at first, but first of all the code is open source either way, so everyone can use it. And secondly, their idea is that even if that developers decides to leave the team, he is still the owner of that library's code. So maybe after resign, they will continue working on the code base. And since it's 100% open source, Artsy can still use it. Awesome! You can read all about that in Issue 22 of objc.io.

So, in conclusion, there's a lot we can learn about open source from Artsy. It's an excellent example of a company that is not scared of open source, but actually leverages it to create a dynamic and vibrant development environment. I didn't write much about what their business is about, but I encourage you to go to their website, download their apps and start using their great technology.

Companies like Artsy can show us a lot about monetizing open source software and doing business in the open. I hope it inspires people to be less scared of uploading their code for everyone to see. But that's not what open source is entirely about. It's about the little people. The developers that are driven by desire to create something for themselves but also shared it with others so that everyone can use it. The projects that are not made because of money, but because writing code can be fun. Which brings us to another amazing project:

MiniKeePass

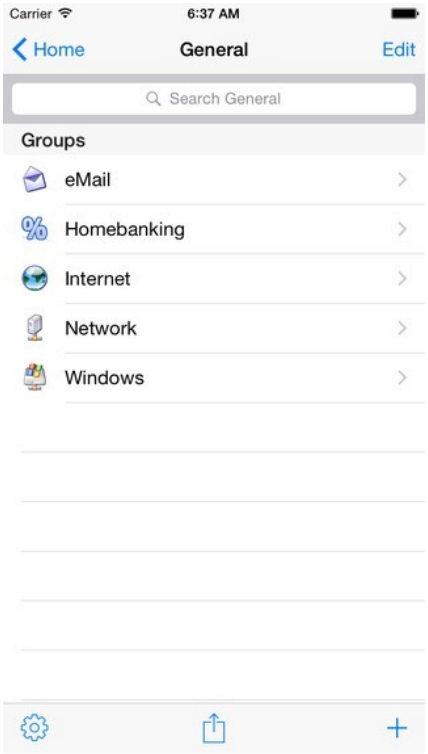
Are you using a password manager? If not, you should. A lot of people nowadays use 1Password for that, but an older, and more open source alternative, is KeePass. So, why would you use it?

- It's cross platform
- It's secure
- It's awesome

And why would you not use it?

- It's not as flashy as other solutions
- Doesn't sync your database (at least out-of-the-box)

Without going too deep into a review of the product, KeePass is a really popular, cross platform password manager solution that I personally use and love. But one question remains... does *cross platform* also include mobile? That's where MiniKeePass comes in!



MiniKeePass is a mostly independent (from KeePass itself) project that brings the popular password manager to iOS and Android. It's almost exclusively written by Jason Rush and John Flanagan. It is released under GPL (General Public License). If you wonder what that means, refer to the Open Source Licenses article of this issue. Basically it's a relatively conservative license that protects the code from being used commercially in closed source projects.

The projects is not in very active development, but it is by no means abandoned. it receives updates with fixes whenever an iOS release breaks something and also gets support for new features as the iOS operating system evolves and gives applications more freedom.

Apart from the usual benefits of an open source project, I think MiniKeePass is an excellent way to learn a little bit about security and encryption. Just clone the repository and see how a world class password manager protects your accounts.

Artsy and MiniKeePass are both great showcases, but they miss one really amazing phenomenon associated with open source software. The community! One of the most powerful aspect of open sourcing your code is to leverage the power of the community and start receiving contributions from all over the world (and become famous among nerds). However, MiniKeePass is a too small project, several developers is all it needs and Artsy is too tied to its own services to benefit the people so much that they start making many pull requests. Of course, both projects have contributors, but it's not enough for a whole community.

And now, you guessed it! We can going to look into projects with a community around them (which are also nerdy enough to get my attention).

Kodi

Kodi, or XBMC, is one of those projects that is so big that you don't know where to start. At its core, it is a media center / home theatre software. Or at least that's how it started. Nowadays, it's full of so many features, that it is difficult to explain. It's compiled for almost all the platforms you can imagine, iOS included (but for jailbroken devices mostly). As far as I remember, it also had an AppleTV version several years ago (Yes, before tvOS). Kodi supports all the codecs and protocols you can imagine. It plays radio, tv, connects to (or creates) a DLNA server and is completely extendable and skinnable.

And it's all 100% open source. A truly amazing project. Millions of people around the world are using it and looking at their [GitHub repository][Kodi GitHub], they have (at the time of writing) 480 contributors. Of course, that's across a lot of platforms, but it makes it even more impressive. It is a whole ecosystem of enthusiasts from different backgrounds, joining forces to build something beautiful. I strongly recommend looking into Kodi's organization. The project is really committed to establishing a strong culture between its members. It's an exemplary work of maintaining projects descriptions, product landing pages, blogs and forums. Right it the README file, people can find all the important links to get them started, they have specific resources for the ones that want to start contributing and you can find all the information you need in the forums.

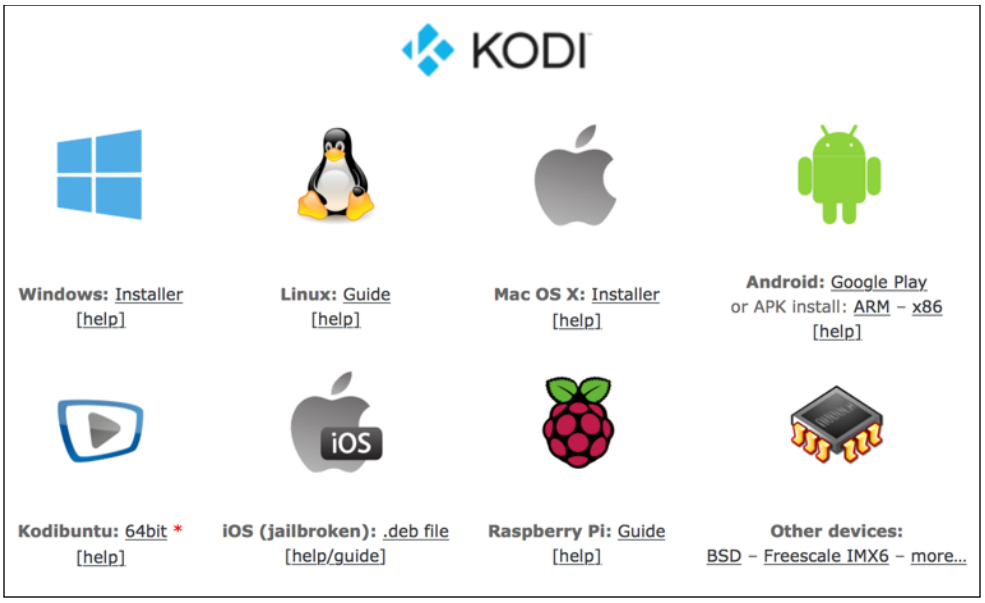
It is no doubt that a community doesn't just happen, it's built and maintained. And Kodi is an ecosystem that shows how that's done.

To some, open source software has a reputation of low quality. And unfortunately, often you cannot expect the same level of polish you'd find in commercial products. But this is not the case with Kodi. It is a mature project that not only manages to bring a large number of features to its users, but also delivers them with

unprecedented quality and taste. You will hardly find something that doesn't work well or is not really well thought out.

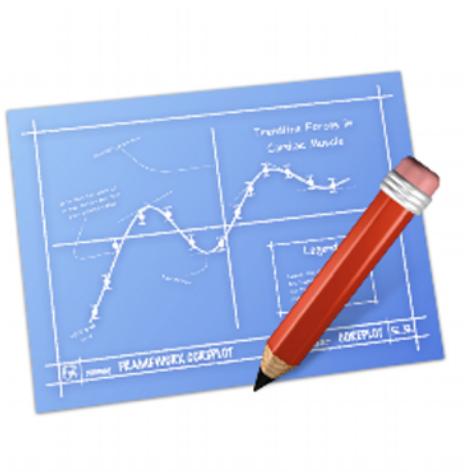
Kodi's code is released under GPLv2. It's not the most permissive license but working with some many dependencies - codecs, network protocols, plugins, it is probably hard to use a more "relaxed" license.

Overall, Kodi is a much better option for your home entertainment than most (actually all) Smart TV operating systems. If you are trying to upgrade your home theatre, get an old PC or a Raspberry Pi and setup Kodi on it. Then downloaded one of the many Kodi Remote mobile applications (I definitely recommend [the official one for iOS][Kodi remote iTunes]) and get started.



Do you remember how I wrote that we are only going to discuss complete applications earlier in this article? Yeah... I lied a little. There is one framework in particular that is really close to my heart. A framework that saved me so much work recently and was constantly amazing me how well design it was. So naturally, I have to write something about it.

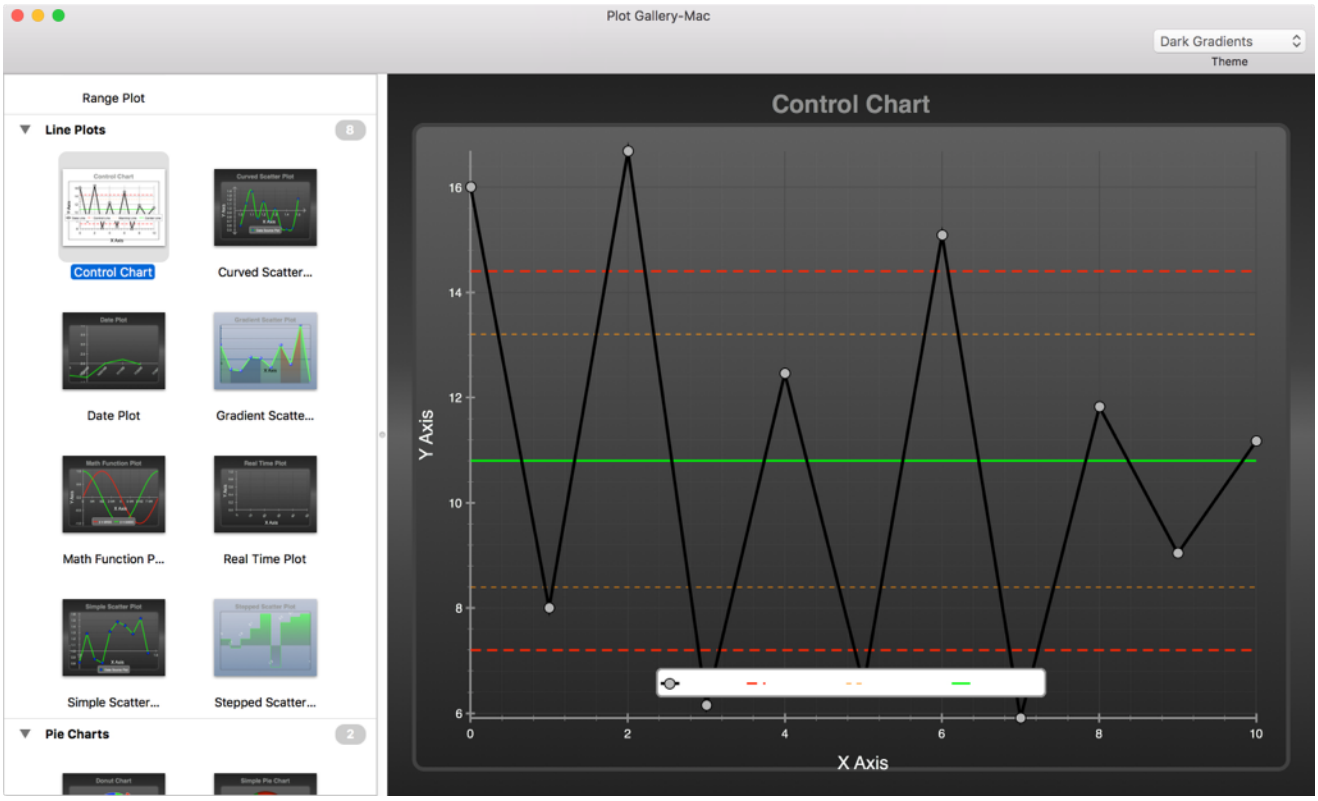
CorePlot



CorePlot is a 2D graph and chart drawing framework built for iOS, Mac OS X and tvOS. And one of not so many, I'd say. Most alternatives I've seen are targeted towards a specific plotting task and are not very generic and reusable.

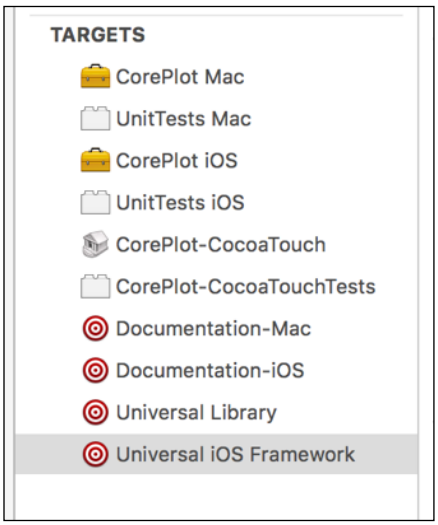
CorePlot on the other hand, supports a wide range of graph types (even pie charts) and each graph is customizable in almost every aspect. And with a current version of 2.1, it is a quite mature project.

Discussing CorePlot right after Kodi, the question of community arises naturally. The project is, I think, popular, but it definitely doesn't have a whole culture around itself. It only has 20 contributors in GitHub so far. But that doesn't mean it is not welcoming to newcomers. Looking into the project description and documentation, there is very little to complain about. All the essential information like the license, contribution guide, important links and installation tutorial are very well explained and easy to follow. The documentation is auto generated with Doxygen and it leaves a lot to be desired, but the project contains a sample application which is **really** excellent. It guides you through most of the framework's features. If you want to implement something, just find a graph within the sample application, lookup its code and copy it. That simple!



The code is released under an MIT license, unlike many of the projects we saw so far, which is very cool of the authors.

So what sets CorePlot apart from others? We already saw several extremely good projects. Artsy was able to build a successful software business "in the open". MiniKeePass was an application that extended the legacy of the core KeePass community and brought an excellent password management utility to the mainstream mobile platforms. And Kodi was an entire software suite bringing all the features you might imagine into your living room's entertainment system, and also building a massive community of loyal users along the way. What could CorePlot possible bring to the table to compete against the others?



By the way - this is how a framework's target section should look like

One of the most important aspects of code reusability is good architecture, intuitive interface, loose coupling and high cohesion. Obviously, if we want to use the same code in different settings, it has to be abstract and very well segmented. But that's easier said than done and all developers know that. it's more of an art than a process. Many have tried and even more have failed.

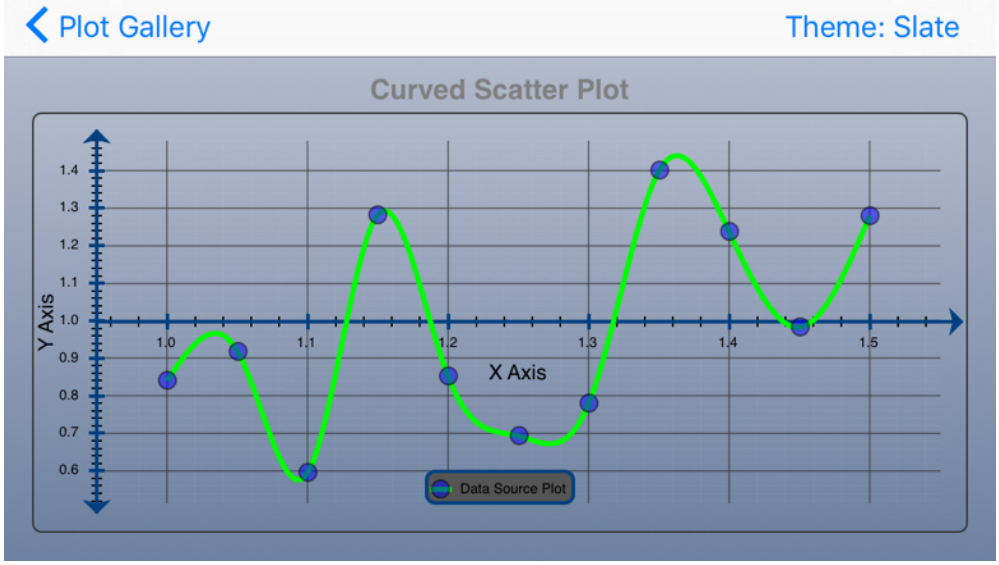
Looking back at the specific case of CorePlot, the problem that framework is trying to resolve is inherently difficult. There are lots of components in a graph and there are more types and styles of graphs that any of us can imagine. So how do you create a library that's, at the same time easy to comprehend, fully customizable and well structured? Well, I have no idea... but it seems the CorePlot authors sort of do.

The framework itself is so big and the architecture and core principles behind it so deep that it would be impossible to explain in this article. In fact, it really deserves a whole dedicated issue (coming soon?).

What I can tell you right here is that the framework does its best to give you freedom to access and customize every element in your graph and still keep a consistent interface in order to keep the API manageable and relatively easy to understand.

Not going too deeply into that, the way they achieve that is by incorporating a "style" object into most elements that you see on screen. These styles have mostly the same interface, so you always know how to change a color or increase a font size. And if that is not enough, often you have on option to substitute a CorePlot element with your own `CALayer`, effectively injecting your own visuals into the graph. This can be a lifesaver when your design requires something quite unusual and custom made.

And while we are still on the topic of layers, CorePlot is entirely CALayer-backed, which makes the graphs highly interactive and animatable.



Overall quite an enjoyable and useful framework. Even if you don't necessarily need graphs in your app, it is still worth the time checking CorePlot out and thinking about all the design choices made there. It can serve as an inspiration and example next time you need to layout the architecture for a new app.

Others?

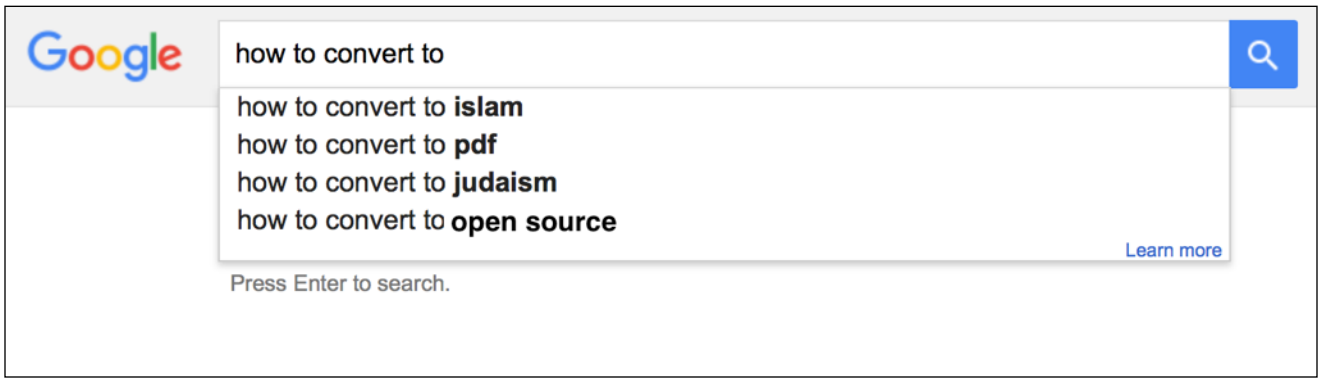
There are, naturally, many other really nice open source applications for iOS and Mac OS. However, it doesn't make sense to talk about all of them here. Additional projects that I was considering writing about include Wordpress for iOS, Wikipedia for iOS, and Hammerspoon - a desktop automation tool for Mac OS. All of those have an interesting story to tell and are an important addition to the open source world, showing that you don't need to keep your code all to yourself and still get the job done.

In in order to cut this article short before it gets too long, we will not be covering those projects. I hope that the ones we discussed were enough to get you inspired and encourage you to develop in the open. Unless you have something really special to hide, it is probably worth thinking if you can benefit from sharing your code with the world, both for the sake of helping others, and also helping yourself. Even if you cannot build a community around your product, keeping everything for the world to see, might have a highly positive effect on your business. As they say:

You can write much better code with a monitor facing the entire office than if you are stuck alone in a dark basement.

How to convert to open source

This article is going to be about turning a pile of **code** you wrote for whatever reason, into **open source code**.



There are surprisingly a lot of things to think about when open sourcing your project. Of course, you can just take your code as-is and upload it to GitHub, but what's the fun in that. We will look into the reason to do some additional work around open sourcing later, so lets leave it for now.

For this particular article, we will focus on the technical logistics, namely:

- Transferring the repo
- Looking into dependencies
- Project metadata

Transferring the repo (Git)

The obvious thing in open sourcing your project is uploading your code to a place people can see. You can always expose the repository you're already using to the world, if it is self hosted (you are using a repository, right...RIGHT?). But does it really make sense. For better or for worse, the de facto standard in open source hosting nowadays, is GitHub. So it makes sense to leverage its massive exposure and social features and give your project the best chance to get popular. And as much as I like to be a hipster about things like this, I think the GitHub is an excellent option and would highly recommend it.

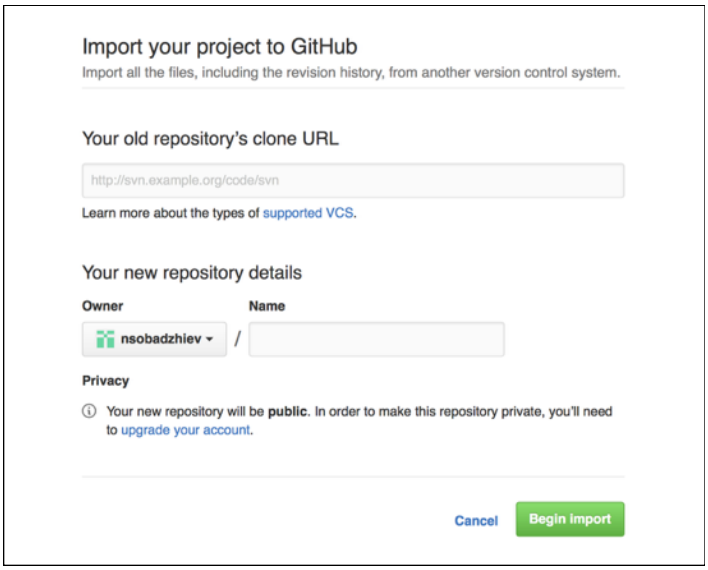
GitHub has extensions for SVN and other source control management systems at this point, but it mainly promotes git. And since I'm using git on daily basis, I'm going to focus on that.

As far as I'm concerned, there are three ways to achieve that - the easy way, the hard way and the lame way.

The easy way

The easy way is to use the power of GitHub. They have a wizard for importing an existing repository, be it SVN, GIT or TFS.

Basically, you input the path to your previous repository, set the project name and start importing... Super easy!



The hard way

Q: Why are you reading this when there's a Github wizard that's much easier to use?
A: Because you're a nerd...Nerd!

For the hard way, we shall delve deeper into git commands. Yes, wizards are cool, but it's sometimes nice to know how to do stuff "with your own hands". Also, going this path, we will not have to clone a new repository or change our working directory (much). We will add another **remote** to our existing local repository, which is going to point to GitHub. Then we will say that this is the default push location so that we don't have to always remember to specify a new location when running `git push`.

It is becoming a bit of a git tutorial, but lets try it. Check out this snippet:

```
$ git remote add other path/to/new/repository.git
$ git push other master
Counting objects: 3, done.
Writing objects: 100% (3/3), 235 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
```

```
$ cd path/to/new/repository.git
$ git checkout -b develop
$ cd back/to/first/repo
$ git push other master
Counting objects: 3, done.
Writing objects: 100% (3/3), 235 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To path/to/new/repository.git
```

Who is TFS??? It's a source control management software build by Microsoft.

We can see several things here. First of all, we add a "remote" to our local git repository (this means the one on your computer that you actually use for development, not the server you probably use to store your code). Essentially, this is telling git that there is another repository you will be interacting with.

After that, we can attempt to push our stuff to the new location. And what happens in this snippet, is that an error occurs. It doesn't happen all the time. In fact, if you will be pushing your code to GitHub it will certainly not. This is a specific case when you try to push to a repository that is not **bare**. For instance, you create a non-bare repository by typing `git init name path/to/repo`. On the other hand this - "*git init --bare name path/to/repo*", creates a bare one.

The difference is that a bare repository doesn't have a working directory. It's ideal for a central location to store the work of all your developers (for instance GitHub). Which brings is to the actual error that we saw. Git was actually complaining that in the remote location, the master branch exists and it is checked out. So it is a bit concerned that if it pushes there, some changes might get lost. A solution to that error is either to make sure the remote repository is bare, or switch to a branch that's not the one you're pushing into. For instance, you might create a new branch and switch to it. This is exactly what the following part of the snippet does:

```
$ cd path/to/new/repository.git
$ git checkout -b develop
$ cd back/to/first/repo
$ git push other master
```

After this is executed, you should be able to see your exiting commits (only the ones in the branch you pushed) in the new repository.

The lame way

I guess the easiest way to do it for some people is to create the new repository and copy the existing project files in it. And surely, it works 100%. The only problem is that you lose all the commit history you previously had.

And yes, even though this section is called "The lame way", it might be a viable option for many projects. For instance if you didn't have any history in the first place. Or more importantly, you might not want to share your commit history with others.

Handling dependencies

Unless you're building something small and/or are a purist that doesn't like to use other people's code if he can write it himself, chances are your project has at least one dependency. And by "dependency" I mean a third party library, an internal project or someone else's code altogether. It is also quite likely that those

dependencies are open sourced. And now that you're, in turn, releasing your code into the wild, it is time to think about dependencies in a more flexible way.

The easiest way to start using someone else's code, is surely to copy it in your project tree. But that poses some issues, especially with modularity and code reuse. How do you differentiate between your code, and the third party code. What happens if the author of that code updates it and you have to merge the changes? It's pretty clear that separating this module from the core project will be beneficial at some point. And there are numerous ways to achieve that. Depending on your platform, there's always a way for your IDE or programming language to combine several projects in a workspace.

It is often possible to build a framework or a library and just import and link that. It's a good option, but you can run into problems with your source control management system. And on the topic of source control, we can now ease into git submodules. Git submodules are awesome! With them, you let git know that your project depends on another project. You specify where its git repository is and which revision you wish to use and git can start tracking that relationship. If you don't know about git submodules, you should definitely read about it.

Back to the point of open sourcing, you should really start using submodules before releasing your code to the public - it just makes it so much easier. Either way, most of the third party modules that you use are written by you (or your company) or are open source. For both cases, it is nice to declare them as dependancies of your project. That way, if one of those modules is updated, you can easily test that the change works for you and update. Then, all the people that potentially use your code, can also update with a push of a button.

Also, imagine that you are using an XML parser, but you only copied the source files to your project because it's easier. If someone decides to incorporate your code inside his project, a collision might occur. It is possible that he is also using the same XML library. So now he has to remove the duplicating code and figure out how to compile successfully again. Not a very friendly open source experience.

In general, try to make your code as reusable as possible. And never assume any use case, because you never know how people would be using it. After all, that really is the essence of open source - being reusable. The more concise and modular your code is, the less people have to modify it to make it suitable for their needs. So:

BE A GOOD OPEN SOURCE CITIZEN AND HELP OTHERS UNDERSTAND YOUR CODE EASIER

Project description

Even though they are developers, people who will be viewing and potentially using your code are still quite lazy and picky. They love flashy things and projects that look polished and pretty. A good README with a lot of images and examples goes a long way. It shows professionalism and care. And also a few words about the code really save a lot of time trying to understand it.

Achieving that is also not really hard, so you have no excuse not to do it. For consumer products you need websites, videos and colorful graphics, but in the developer world, a good README goes a long way. Just opening the GitHub page for a project, makes that file prominent and serves as a landing page for your work.

The de-facto standard for formatting a README is Markdown. Markdown is a really simple language for styling text. It lets you focus on content with minimal regard to formatting. It's universal, cross platform and much easier than HTML. In fact, this is exactly what I'm using right now writing this article. I'm purely focusing on the content and disregarding the visualization of the text. Later, when the text is complete, I will switch to styling and choose fonts, color and page templates. And it's not just me, many other authors and bloggers are turning to Markdown for their publishing needs.

Moving towards the contents of your README file, there are several items, questions, that you should answer in the text:

- What is this project about?
- What features does it have?
- What is the technology it is running behind?
- What dependencies does it have?
- How can it be installed and run?
- What license is it released with?
- Examples?
- Contributing rules?

Most of these item are fairly straight-forward. Of course you need to explain what this project is, and what language/platform it is intended for. One side note is that for full applications, and for user facing frameworks, it is really helpful to include pictures or gifs showing your code in action. And if it's just a utility library or a chunk of code, you wish to share, an example or a code snippet also goes a long way. For instance, if I'm searching a library of cool animations, I'd much rather see them in action immediately than have to clone and run a whole project to see of it fits my requirements.

Secondly, I cannot stress enough that you need to be explicit with any dependencies you have. Many developers might not care that you're importing 10 other libraries, but the good and experienced ones will carefully consider if they want to inherit your dependencies list. And as a general rule, don't force other people to use yet another framework, unless you have to. I had a problem recently with a third party framework I was using. They made a major refactoring release. There, they decided to import a fairly big and complex logging framework (The Cocoa Lumberjack framework) in order to improve their debugging abilities. I didn't have anything against Lumberjack, but in my opinion it was too complex and big for my needs. Additionally, I had my own logging solution so I was reluctant to import it into my workspace. So in the end, I had to change the frameworks's code to exclude Lumberjack as a dependency. This was especially frustrating since something like this could have easily been extracted into an interface, giving other developers the opportunity to swap logging implementations. But again, this is the beauty of open source - if you need it, fork it and implement it.

Maybe the second most important task to accomplish in a README, are good steps for installation and usage. Remember that others didn't write the code and they sometimes have no idea what they need to do in order to successfully use your code. So take a few minutes to explain your setup and assumptions. Or even better, setup the project in a way that has minimum extra work to run on a different machine.

Something that is obviously missing in the list above, is documentation. And lets face it... if few people write documentation, even fewer read it. In fact, I think most of the time a good examples page or a sample

application work way better. A prime example of that is the CorePlot framework for iOS and Mac. But we talked about that in the iOS/Mac open source world article.

Epilogue

So uhmmm... yeah...

This concludes Issue 01 of the DevMonologue magazine. And as every other good school presentation, it ends with "So... yeah...". If you are reading this, I will assume you find these articles entertaining and/or useful. I really had fun making it and I hope I can continue with many more issues to come. Of course, to justify that, your support will really help. So, any feedback is welcome. If you liked the magazine, found an error, have ideas for topics and resources, feel free to contact me (you can find contact information below). Send me a message, tweet email, picture, reaction gif... whatever.

Is "Going Open source" open source

In the spirit of this issue's topic - Going Open Source, the entire magazine is, and probably always will be available in GitHub. And in the spirit of everything we discussed in "Open Source licenses", the magazine uses... none of the mentioned options. It is subject to the Creative Commons license. Creative Commons is a license that is popular for art, pictures and articles. For instance, Wikipedia uses Creative Commons to license all contributions that are uploaded to the platform. However, it is not really suitable for code, which is why it was never featured in "Open Source licenses".

Want to get involved?

Again, in the spirit of "Going open source", I need to leverage the power of the community in order to ~~monetarize~~ improve the magazine.

No, really, as it turns out, creating an even small magazine issue is a **lot** of work. So it would be really great and a lot of fun to actually collaborate with you guys. If you have a dream of trying out your writing skills or have an awesome idea for an issue, drop me a line. I'd love to hear your ideas. Especially if you have experience with graphic design and/or technical writing since these are probably two of my many weaknesses.

In "iOS/Mac open source world" we already discussed how good open source projects have a good README file containing a "Contributions" section. That way, people know how they can collaborate and get involved with the product. So, in order to put my money where my mouth is, I'm going to also add such section to the README of the DevMonologue repository.

What you will find in that section is how you can get involved as a co-author (spoiler alert: write me a message or email). And also since we are programmers, pull requests are more than welcome. I'm sure there are loads of typos and error that are making you pull your hair, so... fix them yourself and file a pull request, you lazy man.

*Drops mic

About the author

This issue of DevMonologue magazine is brought to you by... me, Nikola. An unknown iOS developer that somehow thinks he can be a technical writer. If you want to get in touch with me and tell me how little talent I have, you can reach me via:

Twitter: @sobadjiev
Wordpress: http://devmonologue.com/ios/
email: magazine@devmonologue.com