# Guidelines for Funding Open Source Software Development

16 Nov 2017

Karl Fogel
James Vasile



© 2017 J. Paul Getty Trust
This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International license.
(CC-BY-SA, https://creativecommons.org/licenses/by-sa/4.0/)

# 1 Introduction

Foundations are increasingly asked to fund projects that have a software development component. Whether the resultant software is the project's primary goal, or is merely a tool built along the way to other goals, it is in everyone's interests that the software's development and maintenance be done well. By "well", we mean specifically:

- The software has realistic scope, and a development[1] plan that fits within the expected budget.
- The software has a plausible route to acquiring a user base: an identifiable set of users who will adopt the software because they recognize the benefits it brings to them. The most reliable way for software to survive its initial funding is to have an intrinsically motivated user base.
- The skill level and cost outlay needed to deploy the software are specified from the start. That is, the requirements for the software's deployment are an aspect of the software's design, and be taken into account from the beginning of development.
- If the software collects data, there is a long-term plan to manage that data: to publish it, make it searchable, maintain it over time, etc.
- If the software handles sensitive data, it undergoes security review appropriate to the level of sensitivity.
- If the grantee has legacy data that will need to be updated and imported into the new system, sufficient resources are allocated to manage this transition.
- The software supports the funders' broader goals as much as possible; it should be open to uses outside the project's funding and current purpose.
- No one has monopoly control of the software; in particular, that vendors or contractors who were involved in the software's development do not end up with a *de facto* monopoly on future maintenance and enhancement.
- The software and its users can be supported by third parties, including commercial support.

These are achievable goals, but they are rarely achieved by accident. Managing software development is harder than it looks, and groups that have not done it before often do it poorly the first time.

The software production framework that most reliably meets the above goals is **open source** development, and we strongly recommend that grant-funded projects be open source by default. This is discussed in more detail in 4 "Open Source".

[1] Throughout this document, we use "development" to refer to the process of designing and building software. In the non-profit world, there is another common meaning of "development" — to raise funds to support the organization's mission — and we're not using that meaning here.

If neither the funder nor the fundee has experience managing software projects, we further recommend that some party with such experience be involved, so that any problems can be detected early and mitigated. It is okay to outsource development to a contractor, but see 8 "Commercial Ecosystems, Without Monopolies" for some notes about managing third-party development.

The guidelines in the rest of this document are based on years of experience observing and participating in grant-funded software development. The authors have at one time or another served in most of the roles discussed: funder, grant recipient, primary developer (in-house and contracted), technical consultant, project management consultant, commercial deployment and support provider, and end user. We have also talked with other foundation program officers, who were generous in sharing their hard-won knowledge. Any errors or misjudgments that remain are, of course, the authors' alone.

# 2 Scope, Credibility, Users, and Sustainability

Grant-funded projects usually happen in the open, which means their software development process can broadcast its intentions and elicit interest from potential users long before the software is ready. Doing so brings several benefits:

- Other funders may be contemplating similar projects; they and their applicants will likely want to know about your project.
- Involvement by early adopters, or even just constructive scrutiny from interested parties, is helpful to the software development process.
- Clear signaling of intentions helps keep the development effort focused. That is, one of the audiences for supposedly outbound communications is your own development team.
- Running in the open from the start is one of the recommended practices for developing open source software anyway; see 4 "Open Source".

However, in order for such broadcasting of intentions to work, a few conditions must be met.

**The project must have a realistic, clearly defined scope.** People who apply for grants are often visionaries, and people who *get* grants are frequently visionaries who have demonstrated some track record of accomplishment. But skill in one area does not necessarily translate to skill in software project management. Because software seems so malleable, people often assume that if they can imagine it, they can build it – and these are, by definition, imaginative people.

As a result, overly ambitious scope is a somewhat common problem in foundation-funded software projects. Someone experienced in software development, and familiar with the

particular field the software is intended for, should evaluate the proposal, and perhaps help the grantee reshape the software component if necessary. Such a domain expert often knows about other projects that have attempted the same ambitious scope and failed; examination of those failures can and should improve the outcomes for later projects. If a grantee can consistently articulate the things their software will definitely *not* do, that is a good sign.

Persisting with too broad a scope can also damage a project's credibility from the outset. The more familiar other parties are with the problem domain and the history of prior attempts to address it with software, the less likely they are to invest their attention in a project that hasn't drawn clear boundaries. Because attention from early adopters is so helpful to software development, mis-scoping is not just a technical or design issue, but a socialization issue, and it will interfere with user acquisition if not corrected.

One way to tell whether a project's scope is too broad is to look at whether the project plan depends on everyone choosing to use the software. That is, when a project attempts to provide a single, unified solution to a disparate set of field-wide problems – *"assuming everyone uses our software, we'll finally solve the data interchange and ontology issues that have plagued our field"* – it is unlikely to succeed, or at least to succeed as envisioned. Those kinds of powerfully unifying solutions are extremely rare in the history of software. The invention of the spreadsheet, say, or of the relational database, might count as instances, but it is hard to think of many others.

No matter how passionately your grantee makes the case for their software's general applicability, the chances are quite low that they have come up with something that will immediately be an obvious choice for all potential users. If the project plan is not compatible with a user base that starts small and expands gradually, it is not a realistic plan.

**Potential users must be free to make an objective evaluation.** Excitement about a just-announced software project is a precious kind of information. If people are genuinely excited because they see that the project has the potential to meet their needs, that is a sign that the project is on the right track.

But sometimes foundations use their financial leverage in a way that interferes with the accurate transmission of this information. When a foundation attempts to acquire a guaranteed user base for one grantee's software by requiring, or strongly encouraging, other grantees to commit to using the software, this cannot in the long run sustain a user base, and in the meantime will hamper the project's ability to know how it is doing. Users must ultimately have an intrinsic desire to use the software. Extrinsic motivations, such as having one's own funding depend on using a tool that one probably wouldn't have chosen otherwise, last only as long as the funding condition is in place, and sometimes not even that long.

When such a funding dependency is present, those other grantees can never be fully honest about their reactions, even to themselves. After all, evaluating new software is already a probabilistic enterprise. One can easily fool oneself into seeing the positives and ignoring the negatives, when the former protects a funding condition and the latter endangers it.

**Investment should be proportional to benefit, for everyone.** The project should only ask for small investments of attention in its early stages, when it can only offer small rewards (for example, the reward of being an early influencer in a still-new project that might or might not succeed). As the project matures and has more benefits to offer, it is reasonable for it to solicit more and deeper attention from a wider array of sources.

To some degree, especially at the beginning of a project, responsiveness is itself a reward: people who make early inquiries are likely to continue coming back if they receive a prompt and to-the-point answer. For a new project, the best approach to communications is to think deliberately about where and how to solicit interest, gather key stakeholders early, make the expected effort/investment ratio clear to everyone from the start, and be responsive to inquiries.

## 2.1 Sustainability Comes From Users, Not From Sustainability Models

Funders and projects sometimes place a lot of emphasis on having a software sustainability plan included in the grant application. Typically, such plans map out the possibilities for commercial revenue, or for subscription models, or for continuation funding from other non-profit sources, etc.

While it is worthwhile to ask grantees to give some thought to these possibilities, and we discuss some of them elsewhere in this document, it is much more important for software to simply acquire motivated users. Software that has users survives; software that does not have users does not survive, although it can walk in zombie form for a while, animated artificially by funding.

It is not necessary for a piece of software to have a large number of users. A small number of highly motivated users can work too. Simplistically, a software project's **sustainability index** could be thought of as:

$$\texttt{Number\_of\_Users} \times \texttt{Motivation\_per\_User} = \texttt{Total\_Sustainability}$$

But there's more to sustainability than the raw number of users. It is up to the project to give the most highly-motivated users a route by which their energy and support can be constructively organized and directed toward the project. This can happen through those users bearing some of the maintenance cost themselves (e.g., by hiring developers and having them work in the project), or commercial support contracts that they purchase, or direct funding of the existing development team, or other means. As long as the project is able to receive that energy in an organized way, then the sustainability index looks more like this:

$$\texttt{Number\_of\_Users} \times \texttt{Motivation\_per\_User} \times \texttt{Organized\_Connections\_Among\_Users} = \texttt{Total\_Sustainability}$$

The previous section argued against the notion that *"If you build it, they will come"*; this section argues that *"If they come, you will build it"*. The key metrics for sustainability are growth in total user base and growth in participation by the most-motivated users. Instead of looking for sustainability models *per se*, look for usage models. If there are enough users getting enough of what they need, they will find the sustainability models needed to keep the software maintained. Software that has users and is open source never goes away, and technical success will open up new possibilities for sustainability.

# 3 Budgeting and Project Management

This document is not a guide to software project management and budgeting in general, and cannot cover those rich and complex topics in much depth. Our observations here are limited to a few areas that seem particularly relevant to grant-funded software projects.

## 3.1 Allow some flexibility

Every active software project exists in a state of tension between planning ahead and offering iterative responsiveness to unexpected requests. Don't expect grantees to be able to plan everything ahead of time; do be ready to change the nature of your investment as needed.

In fact, changes of direction – minor or otherwise – are one of the most useful outcomes of an open and iterative development process, and are one of the reasons we strongly recommend an open source development model. The important thing about changes of direction is not predicting whether or not they will occur, but rather structuring the project so that if they *do* occur, they occur as early as possible, and are based on high-quality information.

Before insisting that deliverable deadlines be met as originally planned, a funder should look at the feedback coming in from early-adopter stakeholders, and be prepared to adjust. For example, if there is a user-training component to the project, but the software is delayed because of adjustments based on early feedback, then the training courses might need to be adjusted to put conceptual or theoretical material first, while the software is completed. Or if early feedback shows that much more can be accomplished through the software's user interface than originally thought, then perhaps less training is needed in the first place, and the project's budget should be rebalanced accordingly.

Grantees should still start with a pretty clear idea of where they're going, of course, and have a budget and development plan based on at least one plausible route to get there. But if early contact with users reveals that the projects' roadmap or priorities need to shift, the grantee should feel able to come to you and explain why. Assuming the project is open source, the primary sources for all of the new information should be available for inspection. As a funder, you may not have time to inspect all of them deeply, but you can (and

should) spot check some of the explanations by looking at those sources, both to provide meaningful oversight and so that you have a better idea of what's happening in the project.

### 3.2 Get experienced help

If no one in the grantee organization has software project management experience, bring in someone who does have that experience, for regular progress checks and occasional assistance. The cost of such help is small, usually on the order of 5%-10% of the total project budget, and it can make a huge difference. More than once we have seen projects where a developer went down the wrong path and no one realized it until halfway through the project, because the project managers were not technical enough, or not experienced enough, to spot what was happening early. A couple of mistakes like that can eat up most of a project's budget.

Even when there are no such blatant mistakes, having a second informed opinion on design decisions, collaboration tools, development process questions, etc, can be a worthwhile investment. However, its value is much lower when an experienced project manager is already in place, because an experienced manager will already tend to seek outside opinions on her own, and will have the connections to obtain them.

Many funders favor projects that propose to use an "Agile" approach to software development. Despite its ubiquity as a buzzword, "Agile" is still a meaningful term that defines a definite and recognizable type of software development process[2]. That process is very appropriate for certain kinds of projects, somewhat appropriate (or appropriate in modified form) for other projects, and not really appropriate at all for still other projects.

For funders, the important thing is that "Agile" is not a synonym for "good". It's a software development process that can be very effective in certain circumstances. It should never be a blanket requirement across all proposals, and when a grantee does propose to use it, their proposal should explain why they think it is appropriate and give some specifics about exactly how the Agile process will work in their particular project. If the word is deployed blithely and without further qualification, beware and start asking questions.

## 4 Open Source

We recommend that all foundation-funded software development be open source. The reasons for this recommendation are by now familiar to most foundations, but we will reiterate the most important ones:

- Open source licensing maximizes return on funding. Anyone in the world who finds the code useful can grab a copy and start using it, and start contributing to development if they choose.

[2] *https://en.wikipedia.org/wiki/Agile_software_development*

- No party has monopoly control over the code. At most, a party could monopolize *their copy* of the code, but even this rarely happens, because it makes more economic sense for everyone to work in the same shared copy whenever possible.
- Because there is no monopoly, open source offers a greater "surface area" for commercial activity and for unexpected uses around the code, which in turn increases the value of the software to many of its users.
- Open source offers a known participation route and a development methodology that many developers are already familiar with, thus reducing onboarding time for new contributors.
- Developers are often very motivated to work on open source projects, partly because unexpected collaboration is stimulating, and partly because when each developer's work happens in public it means that their résumé is portable (i.e., their employability increases in general, and increases specifically in proportion to the success of the project).

Open source development is a large topic and we cannot cover it in great depth here[3]. Below are a few especially important recommendations for setting an open source project out on the right foot, based on our experience with grant-funded projects.

**Be open from day one**. Sometimes a development team decides to do all their development behind closed doors until the software is done, or nearly done, and only then make it available under an open source license; we have found this to be especially common when an outside vendor who does not have much experience leading open source projects is responsible for the development process.

This is a mistake. Opening up at the end instead of at the beginning almost always leads to a dysfunctional open source project, and sometimes to a project that cannot be open sourced at all (because of unexpected proprietary dependencies, outbound license incompatibilities, or other reasons).

The single most important piece of open source advice we can give is: run in the open from the very first day of the project. All code should live in the public repository; all bug reports should be filed in the public bug tracker; all discussion should take place in the kinds of public forums that are typical for an open source project.

**If no one on the team or at the funder has experience running an open source project, get help.** Even if the project team has *used* open source software before, that is often not enough to ensure that they will run an open source project well. There are many ways to do it badly, to the measurable detriment of the project. If the team is not experienced at running open source projects, bring in someone who is to work alongside them for the first month or so. Setting standard open source habits in the project early makes a big difference; a project's culture will self-perpetuate as new contributors join, so it's important to pay attention to it early on.

[3]See https://opentechstrategies.com/#resources for more on this topic.

**If possible, spread funding among multiple distinct entities.** When the project is large enough, spreading funding around – e.g., by hiring multiple different contractors, or by supplementing in-house development with outside contractors – forces inter-entity collaboration from the start, and primes the ecosystem for later cooperation among commercial support vendors. For a more detailed discussion of this strategy, the report "OpenDRI & GeoNode: A Case Study on Institutional Investments in Open Source"[4] is a good place to start. Below a certain level of funding, however, this strategy is unfortunately unavailable, because there is naturally a certain minimum size below which it is not useful to go in allocating funding chunks.

## 4.1 Addressing Open Source Misconceptions

Despite the fact that open source has become the norm in modern software development, many people outside the tech industry are still unfamiliar with some of its tenets. Funders and grantees may be particularly concerned about loss of control over the direction of their project, or long-term viability of open source solutions. The following points address some of the most commonly-encountered misconceptions:

- **Open your code, keep control:** Open sourcing your code does not obligate you to accept contributions from anyone who suggests them. Each copy, or "instance", of an open source project is run by one governing body (of greater or lesser formality), and that body decides what changes and contributions to accept. In practice, most projects center around one instance where everyone voluntarily pools their contributions, and thus voluntarily accepts its governance.
- **Open source does not imply open data:** Similarly, open sourcing your code does not require you to share your data. In some cases, opening your data may be beneficial (see 7 "Data Management"), but that is a separate consideration from opening the code.
- **Open source software is commercial software:** See 8 "Commercial Ecosystems, Without Monopolies" for a full discussion of this point.
- **Open source software is as secure as proprietary software:** People sometimes mistakenly assume that if the code is visible, as is the case with open source software, it must be less secure than "closed source" (proprietary) software. This is not how software security works. In actual practice, open source code does not suffer from a higher rate of security vulnerabilities than closed code, and open source software is regularly used in security-sensitive applications by technologically sophisticated entities such as banks and national defense agencies — in part precisely because its code can be fully inspected.

[4] *https://opendri.org/wp-content/uploads/2017/03/OpenDRI-and-GeoNode-a-Case-Study-on-Institutional-Investments-in-Open-Source.pdf*

# 5 Early Focus on Developers and Deployers

Early in a project, *who* matters more than *what*. Finding the right partners can make a difference not just in getting the word out about the software, but in its technical progress as well.

Because a project in its early stages is not ready for production use, those partners are by definition going to be groups or individuals who are comfortable working with pre-release, untested code. It is worth considerable investment to make their lives as easy as possible – the investment will be paid back by their participation in the project.

Therefore, encourage grant applicants to structure their software projects with these priorities in mind:

- Make the software easy to get and install from the start. Document the system requirements and intended deployment scenarios very early on[5]. Even though the project's initial development team doesn't need that documentation themselves (they already know those things), it's worth taking some people off of coding and having them write that documentation, so that potential partners can more quickly recognize if the software is for them.
- Plan for responsiveness even during early development. For example, the project's budget should explicitly include some time just for responding to questions and bug reports from people new to the project, even at the beginning. Some of those people will stick around and become contributors.
- Among early adopters, notice individuals and cultivate them. There's no such thing as "organizational expertise" in reality: there are only people who have knowledge and share it. Individuals may move from organization to organization, but if the project has relations with the *people*, then those people are likely to not only retain their interest in the project, but to advocate for the project at the next organization they join, and even arrange for project-directed work to be part of their job. (Instances of this exact dynamic are documented in the "OpenDRI & GeoNode: A Case Study on Institutional Investments in Open Source" report mentioned earlier.)
- If budget permits, hold in-person hackathons and training events even before the software reaches its 1.0 release. When associated with topically related conferences, such events can be a tremendous driver of interest and word-of-mouth news about the project.

Most successful open source projects prioritize getting their most-motivated users involved as early as possible. This means making sure that there are always paths available to those who have an intrinsic motivation to contribute. Many grantees are not aware of the potential interest that is available to be harvested, if their project sends the right signals.

[5] See *https://www.archesproject.org/implementation-considerations/* and *https://www.archesproject.org/documentation/* for examples.

Funders should encourage grantees to plan and budget for the steps necessary to find that interest, even at the cost of (apparently) slowing down development slightly at the beginning of the project. The investment is almost always worth it.

# 6 Security

If a software project is going to build a product that might handle sensitive or confidential data, arrange to have third-party security auditing start sooner rather than later.

Performing audits during early development means that the developers can learn from each mistake and institute new code review practices at the day-to-day project development level. This improves code quality overall, and in particular lowers the rate at which further security vulnerabilities are introduced.

There are firms that specialize in security audits of standalone code bases. This can be worthwhile at any stage of development, even after 1.0. But the greatest benefit to the project will probably be had from doing real-time, change-by-change review early in the project, with the security reviewers interacting directly with the programmers. Even experienced programmers often accidentally write security vulnerabilities in their code, so no grant applicant should turn down security review. For grantees inexperienced in software development, the funder may need to make sure the grantee understands the importance of treating security seriously, and of budgeting for security review.

# 7 Data Management

When a software project is funded to collect data, the ongoing publication, management, and maintenance of that data must be part of the plan. "Put it on the web" is not a plan. Some of the questions that need to be answered are:

- Will the data be searchable? If so, is custom software needed to implement the search engine, and does the proposed software project include that functionality?
- Will the data be accessible via an API? (See 7.1 "Insist on APIs for Data Access" for more about what this means.) Usually the answer should be "yes", and the same questions asked in the previous item apply here too.
- Does any of the data involve privacy concerns? (See 6 "Security" for more on this.)
- Are there authorization distinctions to be made among users – that is, are there people who can view some parts of the data but not other parts? There may be legal work to be done, such as formulating electronically signable terms-of-use or confidentiality agreements, before the data can be made available to users.

- Who is most likely to be interested in the data, and for what purposes? Are the ways in which the data will be made available appropriate for those purposes? For example, if it is geographically-based data, perhaps it should be made available via interactive map as well as via API.
- How quickly will the data set grow, and how much of the project budget will be required to store and serve the data?
- How quickly will parts of the data set expire or become obsolete? In other words, what are the turnover rate and turnover characteristics for the data? Does the project include plans for dealing with data turnover?
- How is the data curated? Who can contribute to the data set, and who can attach metadata and labeling to the project's central copy of the data?
- What format is used to store and share the data? Using proprietary formats for data could mean that certain users will not have access to it. As much as possible, data should be stored and transmitted using public, standardized formats — complying with those standards means that more existing programs can make use of the data right away.

Just as it is often easier to find someone who will donate to have a building built than it is to find someone to endow that building's maintenance budget, grantees often find it easier to get funding to *gather* data than to get funding to *maintain* the data and keep it properly available.

Relatedly, if the grantee has legacy data in an older format that will need to be updated and integrated into their new platform, ensure that they have adequately budgeted for that conversion work in their proposal. It is typically a good investment to do this conversion in an automated fashion, through (for example) an "Extract Transfer Load" (ETL) process. At times, multiple sources of legacy data may exist, and it is worthwhile to take the time to automate the process of converting each of these into the new data format and storage system.

Funders should help grantees think through the long-term life of the data they gather, and encourage them to budget realistically for the costs of data publication and maintenance — including the time when their new data becomes legacy data, and needs to move to another format or storage system.

## 7.1 Insist on APIs for Data Access

The concept of an API ("Application Programming Interface") is so important to data services that it deserves a section of its own.

It is unfortunate that such a key concept is hidden behind the technical acronym "API". If you aren't already familiar with what APIs do, please take a moment to read Appendix I "Introduction to APIs" and then come back here.

The importance of providing a documented API as part of a data-oriented service cannot be overstated. If data is made available at all, eventually someone will want to work with that data in a programmatic way. Any researcher or other user of the data who wants to work with it at scale can only do so effectively through an API, and will *expect* an API to be provided.

In some cases, the API could be as simple as an "export" button, causing a spreadsheet (usually in CSV[6] format) to be downloaded. But often a more sophisticated API will be needed, or at least will be very helpful to many users. For example, there may be queries or interrelations that are specific to this particular kind of data and that are unlikely to be provided by most spreadsheet programs. A tailored API would be aware of such specificities and would provide query functionality to support them. Or sometimes there's just a lot of data, and it would be too slow to download it all in a CSV file; an API allows people to target the exact subset of the data they need.

The question of precisely what kind of API is appropriate for a given project must be answered on a case-by-base basis. Here we can only emphasize that an API should be considered for virtually every data-oriented project, except perhaps for the smallest and simplest datasets. Even if the grantee cannot anticipate exactly how an API might be used, or who might use it, if their project will make data available, then it is almost certain that the more successful the project is the more demand there will be for data access via API.

# 8 Commercial Ecosystems, Without Monopolies

Open source software is also commercial software, not just in theory but in practice: there is commercial support available for most widely-used open source software packages, and often that support is available from more than one source.[7]

Ideally, this should be the situation for grant-funded software too, at least once it reaches maturity. Of course, some projects inherently occupy too small a niche to attract commercial support, and there is little that can be done to generate commercial interest in those cases. But for projects that do have that potential – and many do – then it is worth taking steps to encourage it.

Whether or not an outside vendor is responsible for primary development, every vendor who is active in the project should be made aware that participation by representatives of other companies is to be encouraged, and that it is bad form to try to redirect development energy or user inquiries away from the main project and toward one's own (often proprietary) offerings.

[6] A "comma-separated value" file, a kind of baseline import/export format supported by all spreadsheet programs. See *https://en.wikipedia.org/wiki/Comma-separated_values* for more.

[7] The myth that open source is somehow non-commercial remains in circulation, despite decades of commercial activity around open source projects. For a thorough explanation of how open source and commerce go together, see *http://blog.ieeesoftware.org/2016/04/dissecting-myth-that-open-source.html*.

Vendors who have prior experience in open source projects will be comfortable with this, and will understand that these standards are in the project's – and therefore their own – long-term interests. Vendors who do not have such experience sometimes misbehave, however. They may, for example, set up alternative user forums that compete with the project's existing forums. Or they may claim a greater control over project direction than they actually have, in order to persuade customers of a purported ability to steer the project's priorities in the direction that customer wants.

Projects can help prevent these kinds of problems by having an explicit "Commercial Code of Conduct"[8] that spells out appropriate conduct for commercial participants. Note that these provisions would apply to funders just as they would to any other participant in the project.

Any vendor is still free to do whatever they want. The Commercial Code of Conduct merely governs their relationship with the grantee, the funder, and with any other entities that have agreed to abide by the Code of Conduct. A company that violates that agreement can still use the software freely as permitted by its open source license, but that company will have lost the implicit endorsement of most of the community surrounding the software – the company will have given itself a bad name in the very forums that first come up when someone is looking for information about the software. The company may also face ostracism when trying to get its changes accepted into the communally-maintained code base, and be forced to go it alone by maintaining their own separate **and** divergent copy of the code.

These penalties are usually too great for most companies. They choose to abide by the agreement and reap the benefits of remaining in good standing in the project community.

In order for the penalties to have teeth, however, someone has to be able to enforce the dissociation between the bad actor and the core project. This is easier when there is an enforceable trademark, which is the subject of the next section.

## 8.1 Open Source and Trademarks

In order to prevent identity confusion, whether deliberate or accidental, it is normal for the funder or the grantee to retain firm control of trademarks related to the software project.

There is no contradiction with open source licensing or with the open source spirit here. Open source is about the freedom to copy, use, and modify the code itself, whereas trademark is essentially a way of ensuring truth in labeling – a kind of fraud-prevention measure. To control a trademark is to enforce a certain definition for a certain label: the name X, as embodied in the software product Y. Those who do not abide by the trademark license can still use Y as software, but they are restricted in certain ways from using the term X commercially.

The general topic of formulating written trademark policies for open source projects is too complex to go into in detail here. Funders should simply be aware that there is much prior

[8] See *https://www.archesproject.org/code-of-conduct/* for example.

art in this area, and make it available to grantees as needed. A good distillation of that prior art can be found at *http://modeltrademarkguidelines.org/*, and some examples are given in *http://producingoss.com/en/trademarks.html*.

A trademark policy designed for public-good open source software should not discourage use of the software's name in general. It is desirable for the software to be recognized and discussed. The policy should just prevent situations that might cause confusion, e.g., don't let someone name a business or a web site the exact same name as the software.

Funders should also take care to keep their own marks separate from the project's marks. If the software project's identity gets entangled with the funding organization's identity, it can be very difficult to create a trademark policy for the software that will pass muster with the funder's institutional legal department.

Trademark law is not the same around the world, and in some circumstances projects should consider registering trademarks in multiple jurisdictions (the European Union and the U.S. are a common pair). This document does not offer legal advice. One thing funders can do to help their grantees is make the funder's own legal resources available for one-time decisions such as whether and how to seek trademarks.

# Appendix I Introduction to APIs

An "Application Programming Interface" or "API" is essentially a *contract* – an agreement between two computer programs, perhaps running on different machines across a network, about how they will exchange data.

That may seem a bit abstract, so think of electrical sockets. There is an expectation that when you plug a lamp into a wall socket, the plug and the socket will match in terms of size and shape, and that electricity will flow at a certain rate into the lamp so the light turns on. You could say that when you plug in a lamp you invoke the "charge" functionality of the "socket" API. The manufacturers of the plug and socket are different organizations, but you can be confident that when you buy a new lamp with a new plug you will still be able to plug it into your existing sockets. This is because manufacturers have agreed on a standard way to work together.

Creators of computer programs try to do the same thing, by documenting their APIs. The documentation is crucial: it spells out what the agreement is. When you travel to another part of the world, you bring an adapter to allow your plugs to fit into different sockets, because those sockets use a different API than the ones you're accustomed to. API documentation is what allows manufacturers to create all those different adapters that can translate from one API to another, so you can plug in your lamp into a different kind of socket. Documentation likewise tells lampmakers in each country how to shape their plugs so that they can receive electricity from the sockets.

For a computer program, the plug and socket are exchanging information, not electricity, but the concept is similar. The contract specifies that questions sent in an agreed-on

format (plugs shaped a certain way) will receive answers also in an agreed-on format (electricity flowing safely).

From here, it gets a bit more complex than lamp plugs and wall sockets, because the socket API is very simple: all the lamp can ask is "Please send electricity," to which the socket replies by doing so. With a more complicated API, like those found in most computer programs, the conversation can go further than that. For example, if one program sends this:

```
{MY_QUESTION, MY_RETURN_ADDRESS}
```

then the other might respond with:

```
{FIRST_ANSWER, NUMBER_OF_REMAINING_ANSWERS, CONVERSATION_ID}
```

(The `CONVERSATION_ID` is a unique number, generated by the responder, that allows it to track where it is in this conversation — because it might be having similar conversations with many other programs simultaneously.)

A response like the one above allows the asker to plan its next moves. If the number of remaining answers is 0, then the asker knows it is done: there is nothing more the responder can say in this conversation.

If the number is greater than zero, the asker can examine the content of the first answer (perhaps even considering how long it took for the first answer to come back) and decide whether it wants to request the next answer in the series.

If the asker does request the next answer, it might send something like this:

```
{PLEASE_SEND_NEXT_ANSWER, CONVERSATION_ID, MY_RETURN_ADDRESS}
```

The responder, seeing the same `CONVERSATION_ID`, now knows to send the second answer in the series, because it remembers that the asker has already received the first answer.

This is, of course, a simplified example of an API exchange. Real APIs are more complex and are specified in much greater detail, but the basic idea is the same: programs ask each other questions using a restricted and very carefully defined language. Human programmers learn these languages too, in order to write programs that can talk to each other. Programmers are the people shaping the lamp plugs so that they can receive electricity from the wall socket, or building the adapter so that it can take electricity from one kind of socket to a non-matching plug.

An API is the proper way, really the only way, to request a large amount of data from a service. The programs that people work with in daily life have user interfaces – i.e., they react to keyboard presses and mouse clicks, and display things on the screen. But when one needs to request from a data server, say, 15,000 records that meet complicated criteria,

it makes no sense to try to point, click, and type one's way repeatedly through the corresponding interface screens. A human's shoulder and wrists would give out long before the task were done, and besides, the physical speed at which a human can make those moves is far too slow to scale to that number of records.

Instead, the way to fetch those 15,000 records is to have a program do it. Therefore, most programs are designed to have a way to respond to other programs. In fact, many programs work *only* this way, and don't have a user interface at all: their only interface is a programmatic interface – an API.

For example, while your web browser is a program that responds to key presses and mouse clicks coming from you, it translates them into API messages that travel over the network to a web site, and that web site is a computer running a program known as a "web server". The web server *only* responds to API messages. It has no user interface of its own. When it receives a well-formed API message, it crafts a reply and sends that back over the network. Your browser receives that reply and translates it into the appropriate human-visible browser action: draw a web page, or update a page element to indicate that that part of the page has been submitted, etc.

(When a web server receives a non-well-formed API message, or is unable to comply with a request, it simply sends back an error message of some kind. Your browser understands these error replies as well, and you have probably seen them displayed occasionally in place of the information you wanted.)

A common phrase among programmers is that a program's API "wraps" that program's functionality and the data that the program has access to. You could get electricity from your wall socket without using a plug (without calling the API), but it would be difficult. The API is the surface membrane that messages must negotiate their way through, in both directions, in order for someone to access the program's data and in order for responses to come back out in a predictable way.