Manager of Developer Relations:

(Or, API Coordinator): A Job Description

by
Emily Berk
Armadillo Associates, Inc.
PO Box 370588
Montara, CA 94037
650-728-0376

Contents

Executive Summary

This document proposes a model for developing, managing and deploying an API targeted at *external* developers.

The thesis is that creating a successful API is a cross-functional team effort that is best managed by a single person who can interact effectively with the various members of this diverse group. That person would serve as the API Coordinator, in that he or she would oversee the design, development and deployment of the External API. But that person would also have a much more significant function -- to serve as the interface between the company that publishes an API and the external developers who use it. I have therefore chosen to call that person the Manager of Developer Relations. When discussing development of the API in this paper, I use the title API Coordinator. However, I think it would be a good idea for the person who serves as API Coordinator to *become* the Manager of Developer Relations (or, Developer Evangelist, perhaps) once the API design is completed.

The document begins with a discussion of what an API is and how an API targeted at internal users differs from a commercial API.

This paper then discusses the tasks that must be accomplished in creating and distributing a successful API to external developers and the role the API Coordinator should play in the process.

What makes an API Commercial?

What is an API?

An Applications Program Interface (API) is a set of programming tools that allows a computer programmer to write programs that interact in a safe, structured way with an existing application (or other body or bodies of code).

Note: The distinction between an API and an SDK is an extremely fine one. In this document, from now on, we will call any API or SDK-like product, that is, any product composed of a set of one or more libraries of functions that are callable via a programming interface, an *API*.

Pairing existing code with a good API is one of the best ways to foster rapid development of productive extensions to an application. The API provides developers with building blocks that speed development of custom and new features. Developing according to guidelines set forth by the API promotes applications that work well with each other and imposes a consistent look and feel that users in today's markets have come to expect.

Internal vs. External APIs

Growing an Internal API

Typically, an API begins life as a collection of tools created for and used by the internal developers who have built an application. This sort of "Internal API" provides scaffolding upon which adaptations of the existing application as well as new applications are built.

In certain complex software development projects, APIs are planned from the get-go. Often, however, internally-used APIs evolve over time, or are thrown together from pieces that were not developed according to a central plan.

Internal APIs often start out as loose collections of routines, each of which evolved to solve a particular problem that arose during the development of a software product. Thus, the routines, protocols, and tools that form the heart of the API tend to be implemented in a disorganized fashion -- slowly, by a variety of different programmers and programming groups serving a variety of different taskmasters. The end result is (often) a mishmash of quirky, poorly documented code. Each component gets the job done, but there is no guarantee that any two components work in similar ways.

Internally, this usually doesn't matter much, at the beginning at least. The first users of any API set are internal developers who have an intimate understanding of the product they are working on. These tend to be folks who don't care how quirky a routine is as long as it does its job. Experienced, knowledgeable users can perform wonders with an API set that is poorly organized, poorly documented, and rife with inconsistencies and flaws. After all, if they can't figure out how a routine works, they can usually walk down the hall and ask the guy who developed that routine. As long as new programmers can get hold of old hands

willing to help them understand the hows and whys of the API, workers can stay productive.

At some point the engineers realize, internal and external consultants, Value Added Resellers (VARs), and corporate developers require access to many of the power routines that internal developers who built the application required. The obvious solution is to repackage the Internal API as an “External API”. And then, as the API and the company grows, and once the API is released to the general public, the routines that make up the API take on new life.

Going public with an internal API

Usually, changes must be made to an internal API before it can be released publicly. Arcane routines that add complexity and are irrelevant to external programmers may be eliminated. In addition, routines that expose internals of the application must be packaged in such a way as to hide the secret parts, but support whatever functionality is required.

In addition, it is advisable to impose a consistent look and feel and functional approach to the routines that remain part of the External API. External developers don’t have the same access to internal mentors that users of an internal API have. They have paid for this product and they are apt to get cranky when they encounter poorly documented or inconsistently invoked API calls.

Another difference between Internal and External APIs is that one works best when it keeps on changing and the other works best when it remains consistent and stable. There are several reasons for this fact. As stated earlier, Internal APIs are developed by and for internal developers whose job it is to extend the functionality of core applications. Such programmers have in-depth knowledge about the application and access to the source code that lets them tap into that application. If they need a new API routine, they are in a position to write it or have access to someone who can write it for them. Generating new tools as needed fosters fast, efficient application development.

External programmers are in a very different position. Such programmers do not generally have access to API or application source code and they wouldn’t have the knowledge to use it efficiently if they did. What they need are consistently implemented hooks and routines that structure their interactions with the application. Making frequent changes to an API that is used by external users is a risky proposition. In general, External APIs have a larger, farther-flung, less sophisticated user-base. External users will rely on the stability of both implementation and features of the API much more heavily than internal users will. While all users will clamor for increased functionality, external users are much less able to adapt to structural change within the API than internal users are.

In other words, a commercial API, if not “forever”, needs to be designed to last a long time.

Creating a successful External API

In order to be successful, an API that is going to be used by external developers must:

- Provide a coherent, useful set of functions to clearly identified categories of developers;
- Be relatively straightforward to learn to use;

- Provide performance, feature or economic benefits not available in the shrink-wrapped installation;
- Provide functions that are invoked consistently.

  Similar functions in the API should be called in a similar way. For example, if more than one function in the API uses the same set of parameters, the parameters should consistently be listed in the same order.
- Provide useful hooks for creating extensions to the existing application.
- Shield external users from the complexities of what is going on inside the core application.

  End-user applications work best when they present users with a simple, consistent interface. Although APIs by their nature are more complicated than most end-user applications, the cleaner their design, the easier they are to document, use, maintain and support.
- Be **very** well documented and supported.

  An API is a toolkit, but the tools within the toolkit are not easily visible to external users. It is mostly through collateral materials that API users glimpse what the API does, how it works and how to use it.

  The "interface" between a developer and an API consists of the API documentation, and collateral programs such as training, support, examples, Web site, listserv, conferences, conference materials, etc.
- Keep secret functions secret and discourage external developers from going around the API

  The set of functions that make up an External API serves as a secure boundary between application internals and the outside world. While it may be fine if internal developers, "go around" the API to get something done more efficiently, external developers should not be provided with this capability. When external developers adhere to the rules imposed by the External API they produce code and application extensions that are consistent and interoperate with other extensions. When they go around the documented hooks they create code that is hard to maintain and that threaten the security of the core program.
- Be designed and supported by developers.

  Whereas it may be possible for an entirely non-technical person to design an accounting application or a computer game, no API can be defined without input from engineers who understand what functions the core software affords, how those functions can be "bundled" into publicly callable chunks and how to interface those functions with real-world computer languages.

  Users of an API are unlikely to call support because they don't know how to plug in a modem or get their printer to work. On the other hand, they are likely to be side-tracked by highly platform-specific implementation details that never trouble most end-users of software. For this reason, the people supporting an API must usually be trained as developers, not as end-user support workers.

An External API need *NOT:*

- Provide all the functionality accessible in an Internal API.

Stages in API development

Creating a successful API is a cross-functional effort

Creating a well thought-out, well documented, architecturally consistent API requires a concerted effort by a cross-functional team. It requires an integrated effort from many groups: marketing, engineering, support, documentation, training, quality assurance, consulting, etc. Coordinating the efforts of all these diverse talents can be difficult, but it is essential in order to come up with an External API that can squeeze as much efficiency as is possible from external developers.

Key to pulling it off is coming up with the right person to manage the project. That person needs to have a wide range of skills allowing him or her to talk with the various constituencies, and, more importantly, to encourage these various constituencies to talk with each other effectively. The person who coordinates the development of an API must be able to talk to internal and external developers, marketing folks and management on their own terms. This person must have the political savvy to come up with good compromises but have the authority to resolve conflicts by royal edict when necessary to move the process forward efficiently.

Stages

This part of the proposal discusses some of the tasks that must be accomplished to create a successful External API and how to best go about achieving those goals. The stages involved in development of an API are:

1. Setting of organizational goals for the API--Creating the API Mission Statement

2. Toolkit Design: Market plan/architecture/requirements
3. Creating the Market Plan
4. Design of feedback loops
5. Software definition/design
6. Documentation design/“End-user=developer” training plan
7. Identification/design of examples
8. “End-user=developer” support plan
9. QA design
10. Software implementation (extraction of existing functionality, addition of new functions)
11. Creation of documentation
12. Test plan creation
13. Example creation
14. Training materials creation
15. Support ramp-up
16. Marketing
17. Feedback
18. Maintenance

In each of these stages there is a need for integrating the concerns of the various technical and non-technical professionals into the result.

Creating the API mission statement

The API Mission Statement is used to define the goals that must be met in order to create a successful External API. For example, one stated goal might read “Our API will provide users with the means to implement functionality that is not cost-effective for us to implement for mass release. For example, it will allow them to manipulate data that is exported by our application in ways not supported by our application.” The marketing and support groups should be deeply involved in the formulation of the Mission Statement since these are the people most familiar with what the end-users actually want. On the hand, the engineering group must also be involved in the formulation since these are the folks who can determine what realistically can be implemented.

Specifying Marketing/Financial Goals

Sometimes the revenues generated by the sale of API toolkits pays for the development of those toolkits. Sometimes they do not. That profitability is only one factor that needs to be considered when coming up with a budget for an API development effort.

Often, the ultimate goal of an API is to encourage lots of developers to incorporate the services of your core application as part of their information technology solution. If the API makes your application more easily extensible and/or allows it to be easily linked with legacy applications, then consultants, Value-Added-Resellers and corporate developers will be more likely to choose your application.

Developing an intimate understanding of the API’s audience is key to coming up with a set of API features that will make the maximum number of external developers salivate. Thus mar-

keting needs to develop a description of the key types of developers the API will target as well as lists of the features those kinds of developers are looking for. There are couple of key topics that any good API marketing plan needs to tackle:

1. Define the target audience of the API including an analysis of language(s) commonly used by that audience, and the context(s) in which the API will be used.
2. Identify constituencies

   Plumbers, carpenters and electricians all often carry toolkits. And, fact, they all may contain similar tools. For example, all three might carry screwdrivers. However, an electrician is unlikely to carry a plunger in his toolkit, whereas a plumber might not need a wire crimper in his.

   In designing your API, you may want to identify which market segments are most likely to use your toolkit in ways that will be most profitable to your company. Or, you may choose to target those who use tools similar to the ones you are thinking of providing. Then, identify how these constituencies overlap and how they differ to determine how many subsets of the basic toolkit you might want to create.
3. Identify the target platforms

   Creating and supporting a toolkit that runs on multiple platforms can be a highly-complex task. And it's not just a question of determining the most popular platforms and operating system variants in your arena, you also have to take into account factors such as the compilers and programming frameworks that are in common use in the space you are selling into. A good marketing plan factors in these sorts of issues and makes concrete suggestions about how to best roll out support to the various constituencies you are trying to reach.

   Initial API product planning should include lists of potential markets prioritized by importance. Suggestions need to be made about what subset of functions and platforms should be supported in the initial release. Rollout plans for subsequent releases need to be drafted. Plans need to be periodically monitored to certain what changes need be made in light of changing priorities of the customer base.
4. Identify who will make the purchase decision.

   It is often the case that the features that are most in demand by the high muckimucks who have the authority to make the purchase are not features that will be used by the programmers who will use the API. All this should be analyzed as part of the Marketing Plan. That way you will know which features have to be included in the checklist provided as part of the glossy marketing literature and which features may not make it into the glossy brochure but must be rock solid if the product is actually going to be of much use to the foot soldiers who have to live with your API decisions.

Designing the toolkit

An API is a set of tools. Coming up with a list of well defined tools to include in the API is a multi-step process. The first steps involve defining the audience for those tools and specifying what sorts of things that audience will want to do with the toolkit. Once this is done you are ready to produce a detailed description of all tools with each feature ranked according to relative priorities.

In this stage of the process, input from the technical staff is critical. Technicians familiar will all the components bundled into the Internal API set will need evaluate those components in light of the analysis provided in the API marketing plan. Components that are not considered important for external developers can be put aside. Components that are considered important

will need to be analyzed to determine what changes need to be made in before they are ready for release to a community of external developers. Further analysis should lead to an understanding of the ways in which all the tools will work together. For example, if you are going to include a hammer with a claw on its head in your toolkit, you will also want to provide nails thin enough so that they can be grabbed by the claw.

The end result of this phase is a collection of lists. One list should include a ranking of all the modules currently contained in the Internal API. This list should be ranked according the relative importance of each feature. Must have features will be high on the list. Features that can be lost in the External API release will be low. A second list will define the work that needs to be done to convert each feature to a level that will work for external developers. This list will also include mention of features that will need to be written or rewritten from scratch.

Defining the architecture

Internal APIs don't really need an architectural definition. If different components work in different ways that's OK-- you can rely on the expertise of the programmers to feel their way work their way around the rough edges. External developers, on the other hand, expect that all the components of their API will follow a consistent set of rules spelled out in a unified architecture.

In this phase, you identify the various functional groups in the toolkit. You also define the similarities between the various tools and the distribution of functions. For example, if you determine there's a need for a tool that stands one widget of type Y on its head and that there's the need for a tool that stands two widgets of type Y on their head as well as a tool that stands lots of widgets on their heads, they had better all work the same way. This can be achieved in one of several ways. You might decide to release a single tool that accepts as one of its parameters the number of widgets doing headstands. On the other hand, you might determine that it makes sense to just release a single tool that turns a single widget Y on its head. If your users wish, they can always call the tool as many times as necessary.

Regardless of which of these decisions you make, you will want to impose a similar calling structure on similar tools in your toolkit. For example, if you have a function that is called as follows:

```
upToDown(numOfWidgets, listOfWidgets);
```

If you also need a function to turn those same widgets right side up again, you might want to call it in a similar way:

```
downToUp(numOfWidgets, listOfWidgets);
```

rather than:

```
downToUp(listOfWidgets, numOfWidgets);
```

Identifying technical requirements

Internal API sets can get away with using inconsistent naming conventions. External Developers expect the consistent rules for variable naming, ordering and the types of parameters the API can accept. Imposing such consistency makes the API easier to document and easier for users to learn.

Enforcing consistency upon an existing Internal API set can be a costly endeavor. Thus a careful cost/ benefit analysis needs to be made to determine time required to re-engineer functions and the benefit that will accrue from those changes.

This is the stage of the project at which these kinds of standards can be negotiated and agreed upon.

Creating a marketing plan

Once you know which tools are going to be included in the API, you can come up with a plan to market the API.

Designing feedback loops

An API is a highly technical product. Feedback is likely to be spoken or written in jargon. Means for escalating feedback to appropriate channels when a problem either cannot be understood or resolved should be defined early on. Procedures for tracking versions, bugs and changes, submitting, prioritizing and routing requests, and responding directly to customers should be defined early on whenever possible.

Developers are not always tactful. Thought must be given, early on in the process, to how to organize feedback loops so that questions can be answered in public, but most complaints are aired promptly and, preferably, quietly.

Software definition/design

The code in an API can be (loosely) divided into two parts:

- The interfaces, which are published to the end-users
- The private code, which implements the interfaces and is hidden from the end-users.

  In all likelihood, there will be significantly more private than public code in your API.

In this phase of the project, the detailed design of both the private and the public code is defined.

Creating the documentation/training plan

One difference between an API and a carpenter's toolbox is that a carpenter can open his toolbox, see each tool in it and understand what each tool does. In order to know what each tool in an API does and how to use it, nearly all users will require documentation.

It is unlikely that an API will succeed among external developers without good documentation. Giving thought to the design of the documentation early on in the process will help ensure that documentation is targeted at the appropriate sophistication level, contains sufficient information to make the API useful, includes examples that aid users in learning the API and in implementing unique functions using the API.

Development of the training plan should be considered part of the documentation design. Printed and/or on-line training materials should be developed with the documentation design in mind and vice versa.

As soon as the API architecture is defined, work should start on designing the API documentation and training plan.

Identifying/designing examples

EXAMPLES WILL MAKE OR BREAK YOUR API.

Very often, software companies have code lying around that they "throw in" to be used as examples. There are numerous problems with planning to use existing code as the example code. For one thing, over many generations, production code gets sloppy. Parameters and variables that are no longer used may still be present. Lines and whole subroutines may be commented out or deprecated. In addition, production code was not written to make a point. It may teach the reader many things, but it may not highlight the one or two difficult concepts the reader needs to understand to use the API effectively. That is why **defining** a well-thought-out, useful **set** of concise, well-documented examples is vital to the success of an API.

There will be four kinds of examples in the API documentation:

1. Short single-line examples embedded in the description of each function give users helpful background information.
2. Multi-line examples embedded within the description of individual functions can be used to effectively demonstrate how to set up variables and calls to individual functions.
3. Integrated examples can be used to demonstrate how to accomplish particular well-defined tasks by calling multiple functions from the API.
4. Application templates, which outline the general steps which must be taken to generate an API-based program, facilitate day-to-day use of the API.

Creating the "end-user=developer" support plan

Pulling together a support staff that can adequately address standard questions about shrink-wrapped applications can be a challenge. Signing up people who can provide good support for an API is even more daunting. Design of the support plan should occur early in the process. Identification, hiring and training of support personnel should begin as early in the API development process as possible. That way you will have people on hand to interact with frustrated developers early in the beta process. There is no better way to learn about how well your API interacts with standard developers' tools such as compilers and debuggers.

QA design

Because an API under development is just a jumble of functions, it provides unique challenges for quality assurance. QA needs to verify that:

- Each individual function provided by the API works as **designed** and as **documented.**
- That examples provided are useful in demonstrating the philosophy of the API's design and that they work as explained.
- That using only the knowledge provided in the collateral materials and the code released in the API, an external developer can use the API to accomplish goals as defined in the API design process.
- That together, the functions in the API make up a coherent, useful set of tools.

Once the planning ends...

Well, planning never ends. But once the initial plans are adopted, the hard work of implementaion begins. Meanwhile, the API Coordinator should ease into the role of Manager of Developer Relations.

That is, now that we are going to have an API, we want to work on building our bridges to the developer community. We will identify the power users and the groups that are most likely to create interesting add-ons to our application. We will try to manage expectations and build interest. And, in the role of API Coordinator, the Manager of Developer Relations will continue to plan -- prioritizing API enhancements and bug fixes as we learn more about what developers want and need from the API.

The role of the Manager of Developer Relations

Nearly anyone who has ever worked on a technical project has witnessed a discussion in which a marketing or support person asserts that a particular feature is necessary for the success of a product and an engineer counters that such a feature is impossible to implement. Every documentation writer can recall trying to document a feature that no one seems to understand. These are both examples of the problems that can occur when communication breaks down within a development team. In most cases, these sorts of problems don't come about because of willfulness on the part of team members. Instead, they usually result from bad planning, lack of coordination, and failed communications.

Many end-user applications can survive, perhaps even flourish, despite lack of adequate help, documentation, training and support materials, because the interface to an end-user application is built into the application. An API cannot.

APIs that are designed for success are APIs that are developed from the get-go following a well-defined and well-documented vision. It is the job of the API coordinator to be the keeper of that vision. To that end, the API coordinator maintains contact lists, mediates design/implementation discussions, and imposes final decisions when consensus does not arise. The API Coordinator serves as the representative of the users and eventual users of the API, the external developers, while keeping in mind the goals that were established for the API.

Because the API Coordinator must mediate between the various groups, the API Coordinator must be placed in a position where they can remain "neutral" on the issues. Place the API Coordinator under the Marketing Director and technical staff may come to believe that decisions are not made for technically sound reasons. Place the API Coordinator under the Technical Director and marketing may feel their issues are given short shrift. It probably makes sense to place the API Coordinator in an autonomous group -- possibly project management. Or, put the API Coordinator in a group of one, outside the other groups, until the role grows into the role of Manager of Developer Relations. At that point, a group of people may be needed to organize developers' conferences, mediate chat sessions and email lists, etc.

Finding your API Coordinator -- a single individual who can talk the talk of programmer, programming manager, support, documentation, and marketing and who will grow into your Manager of Developer Relations -- will be a tall order,but may be vital to the success of your External API.