Report 2: API Good Practice

Good Practice for provision of and consuming APIs

Document details

Author:	Marieke Guy
Date:	April 2009
Version:	0.2 (draft)
Document Name:	good_practice_api_ v0.2.doc
Notes:	

Acknowledgements

Table of Contents

JISC Good API Report

A review of good practice in the provision of machine interfaces and use of services

1   Executive Summary

This report is the second of two provided to the JISC as deliverables for the Good APIs project. It provides a number of good practice techniques for provision of and consuming APIs. The content of this report is based primarily on feedback provided from the developer community via two consultation mechanisms: an online survey of and interviews with the HE developer community.

It is anticipated that the good practice elements of this work will be shared with the HE developer community by chunking into a blog, either the writetoreply.org blog[i] or the Good APIs blog[ii]. Each section will be one post, and each post is open to comments from the public. This approach will probably take place over a specified timescale.

1.1   Intended Target Audience

This report is intended for use by Higher Education development community. It will also have some relevance for managers by providing them with a better understanding of the good practice use of APIs in the Higher Education Sector.

1.2   Licence

This report is licensed under a Creative Commons Attribution-Non-Commercial 2.0 UK: England & Wales Licence[iii.]

1.3   Quotations used in the Report

The quotations given in *block quote style* in this report are from the Good APIs survey. When conducting the survey it was indicated that all information would be annonymised and so all quotations are not attributed.

2   Good Practice for Provision of APIs

Developing and provision of APIs can be like most other forms of software development and many of the best practice techniques are well established. However the API process, and here we are referring mainly to Web APIs, is also dissimilar from other forms of development because of the unpredictable nature of APIs. This may mean that the users and use of an API are unknown. To achieve the goal of well-written and released API developers will do well to consider the entire API lifecycle from conception onwards. Best practice can cover areas from the basics aimed at getting more people using APIs, to general 'etiquette' for API consumers aimed at easing relationships with API providers.

2.1   Plan

Effective projects require effective planning. Rather than just adding an API to an existing service/software and moving straight into coding developers should consider properly planning, resourcing and managing the creation, release and use of APIs. They need to check that there isn't already a similar API available before gathering data or developing something new. Then spend time defining requirements and making sure they consider the functionality they want the user to access.

Although formal planning may not always be appropriate in some 'quick and dirty' projects some form of prototyping can be very helpful. Some areas that might need consideration are scale, weighing up efficiency and granularity.

Authors who change their specification or don't produce an accurate specification in the first place may find themselves in trouble later on in a project.

> *You need to implement the same functionality at least 3 times before you've got enough of a feel for it to be able to factor out an API for it.*

2.2 Gather Requirements

Talking to your users and asking what they would like is just as important in API creation as user interface creation. At times it may be necessary to second-guess requirements but if you have the time it is always more efficient to engage with potential users. Technical people need to ask the user what they are actually after. You could survey a group of developers or ask members of your team.

> *The development of a good set of APIs is very much a chicken-and-egg situation - without a good body of users, it is very hard to guess at the perfect APIs for them, and without a good set of APIs, you cannot gather a set of users. The only way out is to understand that the API development cannot be milestoned and laid-out in a precise manner; the development must be powered by an agile fast iterative method and test/response basis. You will have to bribe a small set of users to start with, generally bribe them with the potential access to a body of information they could not get hold of before. Don't fall into the trap of considering these early adopters as the core audience; they are just there to bootstrap and if you listen too much to them, the only audience your API will become suitable for is that small bootstrap group.*

2.3 Make it Useful

When creating an API look at it both from your own perspective and from a user's perspective, offer something that can add value or be used in many different ways. One option is to consider developing a more generic application from the start as it will open up the possibilities for future work. Anticipate common requests and optimise your API accordingly. Open up the functions you're building.

Get feedback from others on how useful it is. Consider different requirements of immediate users and circumstances against archival and preservation requirements.

Collaborating on any bridges and components is a good way to help developers tap into other team knowledge and feedback.

2.4 Keep it Simple

The adage "*complex things tend to break and simple things tend to work*" has been fairly readily applied to the creation of Web APIs. Although simplicity is not always the appropriate remedy, for most applications it is the preferred approach. APIs should be about the exposed data rather than application design.

Keep the specifications simple, especially when you are starting out. Documenting what you plan to do will also help you avoid scope creep. Avoid having too many fields and too many method calls. Offer simplicity, or options with simple or complex levels.

Developers should consider only adding API features if there is a provable extension use case. One approach might be to always ask "do we actually need to expose this via our API?".

> *A good API will be easy to use and hard to misuse.*

2.5 Make it Modular

It is often better to create an API that has one function and does it well rather than an API that does many things. Good programming is inherently modular. This allows for easier reuse and sustains a better infrastructure.

> *Another problem is that methods sometimes aren't quite configurable enough. Sometimes there is a simple method which will do the job nine times out of ten and then a similar but more advanced, more difficult to use method which offers extra functionality -- this is a good solution to the problem.*

The service should define itself and all methods available. This means as you add new features to the API, client libraries can automatically provide interfaces to those methods without needing new code.

> *It is not enough to put a thin layer on top of a database and provide a way to get data from each table separately. Many common pieces of information can only be retrieved in a useful way by relating data between tables. A decent API would seek to make retrieving commonly-related sets of data easy.*

2.6 Follow Standards

It is advisable to follow standards where applicable. If possible it makes sense to piggy-back on to accepted Web-oriented standards and use well know standards from international authorities: IEEE, W3C, OAI or from successful, established companies. You could refer to the W3C Web Applications Working Groupiv. Where an existing standard isn't available or appropriate then be consistent, clear, and well-documented.

Although standards are useful and important it is crucial that you don't get bogged down by them. Some standards may be difficult to interpret or not openly available. Innovation is sometimes more important than standards.

> *Having a hand-designed XML (<records><record><mymagictitle>...) response is much less attractive than reusing standards such as Dublin Core, SIOC, SKOS, Bibliontology and so on.*

2.7 Use Consistent Naming Structures

Use consistent, self explanatory method names and parameter structures, explicit name for functions and follow naming conventions. For example, similar methods should have arguments in the same order. Developers who fail to use naming conventions may find that their code is difficult to understand, other developers find it difficult to integrate and so go elsewhere. Naming decisions are important, and there can be multilingual and cultural issues with understanding names and functionality so check your ideas with other developers.

2.8 Make it Easy to Access

External developers are important, they can potentially add value to your service so you need to make it easy for them to do so and make sure that there is a low barrier to access. The maximum entry requirements should be a login (username and password) which then emails out a link.

If it is for a specific institution and contains what could be confidential information then it will need to contain some form of authentication that can be transmitted in the request.

If you need to use a Web API key make it straightforward to use. You should avoid the bottle neck of user authorisation, an overly complex or non-standard authentication process. One option is publish a key that anyone can use to make test API calls so that people can get started straight away. Another is to provide a copy of the service for developers to use that is separate from your production service. You could provide a developer account, developers will need to test your API so try to be amenable. If you release an open API then it needs to be open.

Make sure you support Linked Data. Also publish resources that reflect a well-conceived domain model and use URIs that reflect the domain model.

2.9 Let Developers Know it Exists

Making sure that potential users know about your API is vital. You could consider the following:

> Contact your development community using email, RSS, Twitter and any other communication mechanisms you have available.

> Write about your API on developer forums. Make sure that you follow this up by having some of your developers monitoring the forum and answering questions.

> Publish a listing for your API on Programmable Web.

> Blog about your API.

> Make yourself known. Twitter and chat about APIs with other developers you'll get a name as a developer and people will be interested when you release APIs.

> Add a "developers" link in the footer of your Web site. If you have released a number of APIs then the developer section of your site a comprehensive microsite with useful documentation.

> Link to working third-party applications that use your API, or third-party libraries that access it.

2.10 Version Control

Deal with versioning from the start of your project. Ensure that you add a version number to all releases and keep developers informed. Either commit to keeping APIs the same or embed in version numbers so that applications can continue to use earlier versions of APIs if they change. You could use SourceForge or a version repository to assist.

2.11 Provide Documentation

Although a good API should be, by its very nature, intuitive and theoretically not need documentation it is good practice to provide clear useful documentation and examples for prospective developers. This documentation should be well written, clear and full. Inaccurate, inappropriate or documentation of your API is the easiest way to lose users.

A key to using APIs effectively is the understanding of the key concepts they are based around and the types of interaction they are designed to facilitate. API documentation should not forget this high-level conceptual view as part of the orientation.

Developers should give consideration to including most, if not all, of the following:

Information on and links to related functions

Worked examples and suggestions for use. The examples should be easy to clone, from different programming languages.

Case studies. Real world examples in real world languages: PHP Java Ruby, python etc.

Demos – if you want to entice someone to use your API you need good examples that can be re-used quickly. Provide a 'Getting started' for guide.

Tutorials and walkthroughs

Documentation for less technical developers

A trouble shooting guide

A reference client/server system that people can code against for testing and possibly access to libraries and example code

Opportunities for user feedback, on both the documentation and the API itself

Migration tips

A clear outline of the terms of service of the API. e.g. This is an experimental service, we may change or withdraw this at any time" or "We guarantee to keep this API running until at least January 2012"

Any ground rules.

An appendix with design decisions. Knowing why an API developed the way it did can often help a new developer understand the interface more rapidly.

Good documentation is effectively a roadmap of the API that helps to orientate a new developer quickly. It will allow others to pick up and run with your API. Providing it on release of your API will result in less time spent taking support calls.

Human effort is generally more expensive than compute cycles. Beyond a certain point, spending development effort optimising your documentation will be cheaper for the community as a whole than optimising your system.

Other suggestions include using a mechanism that allows automatic extraction of the comments, such as Javadoc[v] and providing inline documentation that produces Intellisense-type[vi] context-sensitive help.

If an API has changed but the documentation hasn't been updated then you can easily waste any amount of time before you realise that the instructions you're following don't match up to the pieces you got, IKEA-style.

2.12 Error Handling

Providing good error handling is essential if you want to give the developers using your API an opportunity to correct their mistake. Error messages should be clear and concise and pitched at the appropriate. Messages such as "Input has failed" are highly unhelpful and unfortunately fairly common. Avoid:

- Inconsistency (e.g. different variable order in similar methods)

- Over-general error reporting (a single exception object covering a number of very different possible errors)

Log API traffic with as much context as possible to deal with resolution of errors.

Provide permanently addressable status and changelog pages for your API; if the service or API goes down for any reason, these two pages must still be visible, preferably with why things are down.

2.13 Provide it in Different Languages

A simple Web API is usually REST/HTTP based, with XML delivery of a simple schema e.g. RSS. You may want to offer toolkits for different languages and support a variety of formats (e.g. SOAP, REST, JSON etc.).

Try to provide APIs in XML format then it can also be read by other devices such as kiosks and LED displays. Making returned data available in a number of format (e.g. XML, JSON, PHP encoded array) it saves developers a lot of wasted time parsing XML to make an array.

Provide sample code that uses API in different languages. Try to be general where possible so that one client could be written against multiple systems (even if full functionality is not available without specialization).

For database APIs, provide a variety of output options - different metadata formats and/or levels of detail.

2.14 Make Sure it Works

Make your API scalable (i.e. able to cope with a high number of hits), extendable and design for updates. Test your APIs as thoroughly as you would test your user interfaces and where relevant, ensure that it returns valid XML (i.e. no missing or invalid namespaces, or invalid characters).

Embed your API in a community and use them to test it. Use your own API in order to experience how user friendly it is.

> *Once you have a simple API, use it. Try it on for size and see what works, and what doesn't. Add the bits you need, remove the bits you don't, change the bits that almost work. Keep iterating till you hit the sweet spot.*

2.15 Feedback

Include good error logging, so that when errors happen, the calls are all logged and you will be able to diagnose what went wrong.

> *Fix your bugs in public*

If possible, get other development teams/projects using your API early to get wider feedback than just the local development team. Engage with your API users and

encourage community feedback. Provide a clear and robust contact mechanism for queries regarding the API. Ideally this should be the name of an individual who could potentially leave the organisation. Provide a way for users of the API to sign up to a mailing list to receive prior notice of any changes.

> *An API will need to evolve over time in response to the needs of the people attempting to use it, especially if the primary users of the API were not well defined to begin with.*

2.16 Maintain your API

Once an API has been released it should be kept static and not be changed. If you do have to change an API maintain backwards compatibility. Contact the API users and warn then well in advance and ask them to get back to you if changes affect the services they are offering. Provide a transitional frame-time with deprecated APIs support.

> *By making the API's network aware and destination-agnostic you enhance an API's usability.*

Logging the detail of API usage can help identify the most common types of request, which can help direct optimisation strategies. When using external APIs it is best to design defensively: e.g. to cater for situations when the remote services are unavailable or the API fails.

Consider having a business model in place so that your API remains sustainable

> *Understand the responsibility to users which comes with creating and promoting APIs: they should be stable, reliable, sustainable, responsive, capable of scaling, well-suited to the needs of the customer, well-documented, standards-based, and backwards compatible.*

3 Good Practice for Consuming APIs

Although consumption of third-party APIs means a developer is working with someone else's code there are a number of good practice techniques that they can use to make for a smoother implementation. These include:

3.1 Choose Carefully

Choose the APIs you use carefully. You can find potential APIs by signing up to RSS feeds, registering for email notifications for when new APIs are released, checking forums and searching API directories.

A decision on using an API can be made for a number of reasons (it's the only one available, we've been told to use it etc.) but developers should consider checking the following:

That it is the best fit for your needs. There may well be other APIs out there that are more appropriate. Good research is very important as it is more than likely that for popular APIs someone has probably done the hard work and produced a library for your language of choice. That said it is possible that you might have to compromise.

What the API does. Spend some time finding out.

How good the documentation is. Check that the documentation correctly matches the API being used. Request sample application code that communicates with the API. Initially commercial software vendors were reluctant to provide good well documented services often only provided simple data transaction services. Good documentation is now accepted as critical.

That the APIs is connected to a functional description, i.e. an overall description of the function of the entire application.

That there is a dialogue with the developers such as a forum or email list. This will help establish if there is continued support for bug fixes etc.

That this API does not clash with each others you are using and will be able to 'keep in step'.

That it is a stable API. APIs that are still evolving are liable to change.

How reliable it is. Some API providers have a better reputation than others.

How popular it is. Popular APIs tend to have an active user community.

If it is still managed. APIs which are not currently managed are unlikely to be supported.

If selecting a product with APIs offered as part of the package ensure you evaluate the APIs too.

What has it been coded in.

A roadmap explaining the likely direction of future developments, if any.

Study various information sources for each potential API. These could include tutorials, online forums, mailing lists and online magazine articles offering an overview or introduction to the technology as well as the official sources of information. There are also a number of user satisfaction services available such as getsatisfaction[vii] or uservoice[viii]. The JDocs[ix] Web site maintains a searchable collection of Java related APIs and allows use comments to be added to the documentation. You may find that others have encountered problems with a particular API.

Once you have chosen an API it may be appropriate to write a few basic test cases before you begin integration.

If you're not paying for an API then make sure that the API is part of the providers core services which they use themselves. If the provider produces a custom service just for you then if they're not being paid then they have no incentive to keep that API up to date.

> *When using APIs from others, do a risk assessment. Think about what you want for the future of the application (or part thereof) that will depend on the API, assess its value and the cost of losing it unexpectedly during its intended lifespan, guesstimate how likely it will be that the API will change significantly or become unavailable/useless in that time span. Think about an exit strategy. Consider intermediary libraries if they exist (e.g. for mapping) to allow a ready switch from one API to an equivalent. Build flexibly if it's worth the extra effort.*

3.2   Risk Management

When relying on an externally hosted service there can be some element of risk such as loss of service, change in price of a service or performance problems. Some providers may feel the need to change APIs or feeds without notice which may mean

that your applications functionality becomes deprecated. This should not stop developers from using these providers but means that you should be cautious and consider providing alternatives for when a service is not (or no longer) available. Developers using external APIs should consider all eventualities and be prepared for change. One approach may be to document a risk management strategy[x] and have a redundancy solution in mind. Another might be to avoid using unstable APIs in mission critical services: bear in mind the organisational embedding of services. Developing a good working relationship with the API supplier wherever possible will allow you to keep a close eye on the current situation and the likelihood of any change.

3.3 Document

When using an external API it is important to document your processes. Note the resources you have used to assist you, dependencies and workarounds and detail all instructions. Record any strange behaviour or side effects. Ensure you document the version of API your service/application was written for.

Bench mark the API's you use in order to determine the level of service you can expect to get out of them.

3.4 Share

It could be argued that open APIs work because people share. Feeding back all you learn into the development community should be a usual step in the development process.

APIs providers benefit from knowing who use their APIs and how they use them. You should make efforts provide clear, constructive and relevant feedback on the code (through bug reports), usability and use of APIs you engage with. If it is open source code it should be fairly straightforward to improve an API to meet your needs and in doing so offer options to other users. If you come across a difficulty that the documentation failed to solve then either update the documentation, contact the provider or blog about your findings (and tell the provider). Publish success stories and provide workshops to showcase what has and can been achieved. Sharing means that you can save others time. The benefits are reciprocal.

> *If you find an interesting or unexpected use of a method, or a common basic use which isn't shown as an example already, comment on its documentation page. If you find that a method doesn't work where it seems that it should, comment on its documentation page. If you are confused by documentation but then figure out the intended or correct meaning, comment on its documentation page.*

Sharing should also be encouraged internally. Ensure that all the necessary teams in your institution know which APIs are relevant to what services, and that the communications channels are well used. Developers should be keeping an eye on emerging practice; what's 'cool' etc. Share this with your team.

Feedback how and why you are using the API too, a lot of the time, service providers are in the dark about who is using their service and why, and being heard can help guide the service to where you need it to be, as well as re-igniting developer interest in pushing on the APIs.

3.5 Respect

When using someone else's software it is important to respect the terms of use. This may mean making efforts to minimise load on the API providers servers or limiting the number of calls made to the service (e.g. by using a local cache or returned data, only refreshed once a given time period has expired). Using restricted examples while

developing and testing is a good way to avoid overload the provider's server. There may also be sensitivity or IPR issues relating to data shared.

Note that caching introduces technical issues. Latency or stale data could be a problem if there is caching.

3.6 Clarity

Certain issues should be clarified before use of an external API. The two key matters for elucidation are data ownership and costing. You should be clear on which items will be owned by the institution or Web author and which will be owned by a third party. You should also be clear on what the charging mechanism will be and the likelihood of change.

These matters will usually be detailed in a terms of use document and the onus is on you as a potential user to read them. If they are not explained you should contact the provider.

3.7 Understand Technology Limitations

API providers have technical limitations too and a good understanding of these will help keep your system running efficiently. Think about what will happen when the backend is down or slow and make sure that you cache remote sources aggressively. Try to build some pacing logic into your system. It's easy to overload a server accidentally, especially during early testing. Ask the service provider if they have a version of the service that can be used during testing. Have plans for whenever an API is down for maintenance or fails. Build in timeouts, or offline comments to prevent a dead backend server breaking your application. Make sure you build in ways to detect problems. Providers are renowned for failing to provide any information as to why they are not working.

Write your application so it stores a local copy of the data so that when the feed fails its can carry on. Make this completely automatic so the system detects for itself whether the feed has failed. However, also provide a way for the staff to know that it has failed. I had one news feed exhibit not update the news for 6 months but no one noticed because there was no error state.

You will also need to be weary of your own technology limitations. Avoid overloading your application with too many API bells and whistles. Encourage and educate end users to think about end-to-end availability and response times. If necessary limit sets of results. Remember to check your own proxy, occasionally data URLs may be temporarily blocked because they come from separate sub-domains.

Other technology tips include remember to register additional API keys when moving servers.

3.8 Keep it Simple

When working with APIs it makes sense to start simple and build up. Think about the resources implications of what you are doing. For example build on top of existing libraries: Try and find a supported library for your language of choice that abstracts away from the details of the API. Wrap external APIs, don't change them as this will be a maintenance nightmare. The exception here is if your changes can be contributed back and incorporated into the next version of the external API. APIs often don't respond the way you would expect, make sure you don't inadvertly make another system a required part of your own.

> *Some things don't behave as expected: Last.fm Geo API didn't find many UK cities!*

When working with new APIs give yourself time. Not all APIs are immediately usable. Try to ensure that the effort required to learn how to use APIs is costed into your project and ensure the associated risks are on the project's risk list.

Some Web developers lean towards consuming lean and RESTful API's however this may not be appropriate for your particular task. SOAP based APIs are generally seen as unattractive as they tend to take longer to develop for than RESTful ones. Client code suffers much more when any change is made to a SOAP API.

You can bind to Web services in ASP.NET, resulting in a proxy class you can manipulate with managed code. Planning how to organize these beforehand, with appropriate naming conventions, would be helpful.

4 Acknowledgements

Thanks to all the people who participated in the Good APIs survey and offered support throughout the project.

Special thanks go to:

- UKOLN Systems Team
- Pete Johnston
- Ian Ibbotson
- Wilbert Kraan
- Tony Hirst
- Sam Easterby-Smith
- Phil Wilson
- Dave Flanders

---

i Writetoreply.org
http://writetoreply.org/

ii Good APIs blog:
http://blogs.ukoln.ac.uk/good-apis-jisc/

iii Creative Commons Attribution-Non-Commercial 2.0 UK: England & Wales
http://creativecommons.org/licenses/by-nc/2.0/uk/

iv W3C WebApps Working Group
http://www.w3.org/2008/webapps/

v Javadoc

http://java.sun.com/j2se/javadoc/

vi http://en.wikipedia.org/wiki/IntelliSense

vii Getsatisfaction
http://getsatisfaction.com/

viii Uservoice
http://uservoice.com/

ix JDocs
http://www.jdocs.com

x Risk Assessment For Making Use Of Third Party Web 2.0 Services
http://www.ukoln.ac.uk/qa-focus/documents/briefings/briefing-98/html/