EXPERT SUPPORT
Understanding technology and writing about it since 1990

The Ideal API Reference Manual

Definitions, suggestions, and examples to help you create an excellent API reference manual

The Basics

What is an API?

An application programming interface (API) is a set of defined interfaces for a specific software application, such as desktop publishing software, a database, or an accounting package. An API allows access to that application through a program instead of through a user interface. An API can consist of an XML definition, a set of functions in C, or a set of classes and methods in Java, C++, a scripting language such as JavaScript, or many other formats.

What is an API reference manual?

An API reference manual is a complete description of the syntax for a specific API.

What Makes a Good API Reference Manual

The audience for an API reference manual is a software developer who needs to use that API to interact with the application programmatically. A good API reference *quickly* answers the reader's questions about syntax and functionality, which often means answering questions about usage as well.

What readers want to know

We have no way of knowing what the readers' specific questions might be. Some readers are looking for a snippet of code that they can copy and paste. Others just want to know, quickly, what a class or method does and how it does it.

However, we do know the types of questions that software developers are likely to ask. For example:

- What does this class/method/function/object do and why or when do I want to use it?
- What are all the functions that allow me to manipulate some specific kind of object?
- What are the valid values for this parameter/attribute/argument/element, and what does each value mean?
- What are possible return values and error codes from this specific interface, so that I can design my code to accommodate them all?

I received *xyz* message; what does it mean and what action should I take?

- What do I need to do before I can use this method?
- Where is a snippet of code that shows me exactly how to use this object in a real-world example?

What to include

A good reference manual usually includes:

- Access to the information in a variety of ways for different look-up needs (see How to organize content)
- An overview of the concepts necessary to understand the API, and references to a companion programmer's guide that provides more complete conceptual material and context
- A detailed description of the syntax
- A complete see-also list of references to related interfaces
- Well-explained examples of usage, including:
  - Snippets or short examples that show typical usage and also less-obvious combinations and values
  - More-detailed common use cases; these might be in a separate set of samples, rather than in the reference manual itself

Where Content Comes From

A writer can use information from a variety of sources to create or update a document.

Existing material

Project teams can usually provide considerable existing material, even if it is not yet in a form that is useful to readers. From this material, you

This article is part of a series of articles on technical writing that Expert Support hosts on its website, expertsupport.com.
Expert Support is located in Mountain View, California and supplies contract technical writers to the computer and software industry.

can often extract vocabulary, concepts, diagrams, and possibly even chunks of text and examples, which you can use to create a first pass at reader-oriented documentation, or use as a source for further questions to ask the subject-matter experts.

Existing material can include:

- Product requirement documents
- Product specifications or design documents
- Marketing overviews
- White papers
- Wiki pages
- Code, such as header files, document type definitions (DTDs), or schemas
- Comments in source code
- Bug reports
- Presentations

Tips when Generating Documents Using Javadoc and Similar Tools

- Become familiar with the options for the tool. For example, Javadoc allows you to apply headers, footers, version numbers, and a lot of other things to your pages.
- Know where the reference and user information is for the tool. Javadoc information is here: http://download.oracle.com/javase/1.5.0/docs/tooldocs/solaris/javadoc.html. This starts with basic reference information but contains links to a wealth of examples and details.
- If it's a command-based language, save a copy of the exact command that you use to generate the documents so that you can just copy and paste it into the command line each time (or define a macro).
- Some tools have GUIs or other applications that make it easier to generate documents than doing so from a command line, so familiarize yourself with these features. For example, in Javadoc, using a command line to specify all the files and directories to include and exclude, as well as all the other options that you might want, can result in a ginormous command line. Check whether the developers who have already generated the documents have tools that make sense for you to use.
- Document all the options that you use for generating the documents in a location where you, future writers, and anyone else who might ever want to generate the documents can find it—somewhere such as in the development team's wiki page or in a readme checked in with your source files.
- If the tool allows you to include external files for introductory material (Javadoc does), use them to include your concepts, introductory material, quick-reference lists, and so on.

Source-code generated material

To save money, reference documentation is often generated from source-code comments (using source-code-extraction tools such as Doxygen and Javadoc; see the sidebar for some specifics). Overall, this has somewhat eroded the quality and usefulness of reference documentation, but it doesn't have to. The erosion probably has more to do with a reduced investment than with the technology—you get out of it only what someone puts into it. Software developers are trained as developers, not as writers, and management seldom expects (or allows) them to spend a lot of time on documentation content.

Writers need to be able to use these tools to provide example-rich, high-value content, rather than the perfunctory descriptions that tend to be the norm with this type of documentation. This works best when you can work directly on the comments in the code. This should not be a big deal; if you are capable of describing and explaining APIs, you are certainly capable of editing comments in code and using version-management tools.

If a team does not want to allow the writer write-access to source code, but insists that documentation work be done in a separate branch, they should be aware that they will need to allocate extra resources for merging documentation changes into the main line, regenerating documents, and allowing additional rounds of review.

Subject-matter experts

Subject-matter experts (SMEs) are, to no one's surprise, those who are experts in the subject matter about which one is writing. These are often the architects and developers of the API, but they can also be product managers, project managers, QA team members who specialize in this product, or support personnel. You might need to be persistent with subject-matter experts to ferret out use cases and examples, but these sources often provide the highest value for your time.

Even if you have existing material to work from, it is often sparse, and occasionally inconsistent and unclear. SMEs are your resource for fleshing out the information to where it is truly useful. Be prepared to ask questions until you are satisfied. Examples of questions that you need to ask yourself and then follow up with the SMEs:

- Do you understand the purpose of the function, element, object, parameter, etc.?
- Do you know how to use the function/object in relation to others (which has to come first, does something have to happen after, etc.)?
- For parameters, are all valid values and their meanings specified? Minimums and maximums? Default values? Prohibited values? Do you understand when and why someone would use the parameters and their various values?
- Do you understand where all of the parameter values come from? If they are the results of some other function in the API, or constants defined in the API, put in a reference. If

not, however, the designers might be expecting developers to get those values somewhere else, such as an online resource, or the operating system.

- Do certain values of parameters look like they should have an effect on other parameters or commands? (For example, one parameter turns sound off and on and another sets the volume from 0 to 10, how do these interact—does setting the volume to 0 also turn the sound off automatically? If sound is turned off, is the volume ignored?)
- Does the function return some value, message, or status? Is it clear what the returned data means?
- If you wanted to write some pseudo-code (that is, not necessarily knowing the syntax or being a programmer), could you assemble all the things in the right order with the right parameters and understand and explain what it's doing and why? If you don't understand an engineer's description of something, dig deeper until you do understand it.

Reader contributions

Reader contributions can provide additional depth and insight into the use of the references. To keep it useful, you need some kind of moderation or rating scheme. http://stackoverflow.com does a very good job of this with its "counting semaphore" style ratings. If you find a post helpful, you can give it an "up" to raise its count. If you don't like it, you can give it a "down" to lower it. This makes it easy to identify the most helpful posts and avoid the trolls. If a company can't afford to provide thorough reference documentation, shifting some of the burden to the readers in this way can make sense. It's cheesy, but it might be the best way stay within budget while still providing readers with the information they need.

How to Organize Content

There is no single correct organization for a reference manual. The best choice is determined by the content, the intended audience, and what kind of companion material is available. Options include:

- For a fairly small manual, it is often best to simply list functions alphabetically; with an object-oriented language, you can order classes alphabetically and methods alphabetically within a class.
- For larger manuals, you might first categorize functions or classes into groups, choose some logical order for the groups, and then present functions or classes alphabetically within each group.
- In the unfortunate situation in which the reference manual is the sole document, with no companion programmer's guide and no way to include conceptual or background material in the reference, it might make more sense to present functions or classes in the order in which the programmer needs to use them.

Facilitating lookup

Organize the reference manual so that information is easy for the reader to find. However, different readers will use the manual in different ways, so no single organization works for everyone:

- Alphabetical organization provides an ordering that everyone understands, even if they aren't familiar with the structure of the particular API. The drawback is that the reader must know the names of the functions or classes that are alphabetized. This organization is ideal for a programmer who has used the product for a while and just needs to confirm the order and data types of parameters for a specific function.

Quick-Reference List Example

Here is an example of noun-oriented quick-reference tables from a single reference manual. Each table includes elements related to a single type of object (noun): Camera or Lighting. Within each table, the elements are alphabetical. In the actual reference, the detailed reference entries for all elements would follow these tables in alphabetical order, and it would be best if each element's name were a hyperlink to the element's detailed reference entry. These are from the COLLADA specification at http://www.khronos.org/collada.

Camera

`camera`	Declares a view into the scene hierarchy or scene graph. The camera contains elements that describe the camera's optics and imager.
`imager`	Represents the image sensor of a camera (for example, film or CCD).
`instance_camera`	Instantiates a COLLADA camera resource.
`library_camera`	Provides a library in which to place <camera> elements.
`optics`	Represents the apparatus on a camera that projects the image onto the image sensor.
`orthographic`	Describes the field of view of an orthographic camera.
`perspective`	Describes the field of view of a perspective camera.

Lighting

`ambient` (core)	Describes an ambient light source.
`color`	Describes the color of its parent light element.
`directional`	Describes a directional light source.
`instance_light`	Instantiates a COLLADA light resource.
`library_lights`	Provides a library in which to place <light> elements.
`light`	Declares a light source that illuminates a scene.
`point`	Describes a point light source.
`spot`	Describes a spot light source.

- Noun-oriented organization groups all entries that are related to a particular kind of object, like tables or examples. The reader can use the nouns without knowing function names or when various functions are typically needed. This organization is useful for a programmer who is new to the API and has a question like, "Is there an easy way to copy an entire table?"
- Verb-oriented organization groups functions by tasks, like copying or initializing. The reader can use the verbs without knowing function names or the kinds of objects that are manipulated during various tasks. This organization is useful for a programmer who is new to the API, knows that a program has to perform particular steps, and wonders, "Which functions do I need to call in the initialization step?"

All of these organizations are useful, so ideally you need to find a way to provide all these look-up strategies.

Providing alternate lookups

A good reference manual provides alternative mechanisms for looking up information so that most readers can find what they want without much effort. Whatever basic organization you choose, it's a good idea to augment it with one or more quick-reference lists that provide alternative organizations. Your basic organization (represented by the table of contents), plus your quick-reference lists, plus your manual's index provide easy lookup of all three kinds: alphabetical, noun oriented, and verb oriented.

Your can place your quick-reference lists either at the beginning of the manual after the TOC or at the beginning of each section if you have grouped entries by noun, by verb, by class, or any other grouping other than straight alphabetical.

Each quick-reference list should include each function's name, a brief description of what it does, and a page reference or hyperlink to its full reference description. See the Quick-reference list example in the sidebar.

You need to put each function's full description in only one place in the manual, and it is sometimes difficult to decide where to put it if you organize your whole manual into noun- or verb-oriented groups rather than alphabetically. In contrast, the same function can appear in multiple quick-reference lists in conjunction with a basic alphabetic organization. For example, if a function creates a file from a table, you can list it with table functions and with file functions, and perhaps also with object-creation functions.

A quick-reference list is useful even if it duplicates your basic organization. For example, even if your organization is alphabetical, it's quicker to search a quick-reference list for names that sound like they might do what you want and then be able to see what each function does, without having to flip to their full descriptions and back to the list until you find the function you want.

It can be challenging to figure out how to get an overview list into generated docs. Javadoc, for instance, allows you to have separate specially named HTML files that contain overview material that is incorporated during the build. You just have to dig in the reference material for the tool you're using to find how to add quick-reference lists.

What Makes a Good Reference Description

A reference description is the page or entry that describes a single element of the API, such as a class, function, method, command, data structure, or type.

Each reference description should have a heading that is the element's name, a brief description, a syntax specification, a full description, and cross references to related reference descriptions or relevant sections in the programmer's guide.

Element headings

An element's heading usually appears in the table of contents. Most of the time, the command or function name alone can be used as this heading. If the API includes overloaded functions, you could group all variants of a function under a single heading. However, if descriptions of the variants are quite different, you might prefer to have a separate heading for each variant. In this case, you could use the function signature (that is, function name plus parameters) as the element heading, or the function name with a parenthetical qualifier, such as "delete (file)" and "delete (folder)."

Brief description

Immediately under an element's heading should be a concise, clear, one-line (if at all possible) description, preferably starting with a verb or key word about the element. Descriptions for things that *do* something (functions, methods) should start with verbs; descriptions for data structures can start with nouns. But that's flexible, as long as the descriptions are parallel in syntax and punctuation across all elements.

Here are some example descriptions using verbs:

- Creates a file of a specific size.
- Creates a file with a specific number of records.
- Deletes a file.

Here are some descriptions using nouns:

- File type.
- Detailed specification for text format.

NOTE: You should use the same brief description of a function when it appears in quick-reference lists.

NOTE: Be wary of "noise" verbs—for example, "Use to create a file"; the "Use to" doesn't add anything beyond the simpler and clearer "Creates a file."

Syntax signature

It is often useful to show the full syntax of a function or data structure early in the reference description. For example:

```
print (
   string  text,
   integer length,
   device name);
```

Syntax specification

Describe the syntax early in the reference description. For each parameter, attribute, or child element, specify:

- Name
- Type
- Valid values
- Limitations
- Default values
- Required or optional
- Any peculiarities about the item that aren't obvious from the preceding

For example:

`start_time`	Required. Starting time of event in [UNIX date format].
`end_time`	Optional. Ending time of event in [UNIX date format]. If not specified, the time at which the server completes processing the request is used. If `abort_time` is specified, this parameter is ignored. Values earlier than `start_time` throw a foobar exception.

Completeness

Always optimize API reference manuals for random access. In other words, assume that readers will go directly to the element of interest. Therefore, if you have a group of related functions, use full descriptions for each, even if the descriptions are identical. Don't use the crutch of "The parameters are as described for FirstFunctionName." This brevity is fine for someone reading the entire section sequentially, but that's not how programmers normally use reference manuals. Don't send your readers on a scavenger hunt to find the information they need.

Try to add useful information. For example, if there are parameters `name` and `title`, it is not useful to describe them as "The name" and "The title" (which is typically what engineer-written documentation does). It would be more helpful to say "A unique identifying string" and "A localizable string that displays at the top of the user's screen."

Context

Add context to help the reader understand the usage of the element being described.

- How would someone use this thing, when, and why?
- Does it work in conjunction with other things?
- Are other things required before this one can be used?
- Are other things required after this one is used? For example, does this consume memory and, if so, does the memory need to be released?
- Are there other things that look similar to this thing; if so, what are the distinctions and cautions?

Things to Avoid

The things that make an interface hard to describe are the same things that make it hard to use:

- Inconsistent terminology and usage (see more in Establishing terminology)
- Unnecessary complexity or indirection
- Exposing things that should not be exposed
- Making the user do things that you should do silently
- An API that is itself confusing, inconsistent, and unintuitive

Some of these things can be addressed in documentation, but they are really problems in how the API is designed or implemented. As an API writer, you can add value to the product by bringing these issues to the attention of the product team when you run across them.

Developers should allow and expect you to help them improve the design of the visible layer (what the programmer sees—the function syntax, parameter names, and so on).

Establishing Terminology

Here are some basic rules for terminology in reference manuals.

Be consistent

Choose one term for each concept that the product deals with and use it consistently. Although the use of synonyms in creative writing keeps it fresh and interesting, the use of synonyms in technical writing can cause confusion. This potential is of particular concern in manuals for products that are marketed internationally, with many readers who are not native English speakers, or with translators who must make an accurate translation.

For example, consider a product that presents data organized into two-dimensional arrays in which each horizontal dimension corresponds to a particular entity and each vertical dimension corresponds to a characteristic of those entities.

What do you call such an array? Is it a *table*? A *matrix*? A *grid*? Choose one term and use it consistently.

What do you call each collection of data in the horizontal dimension? Is it a *row*? A *record*? An *entity*? An *object*? An *instance*? Choose one term and use it consistently.

What do you call the information in each vertical dimension? A *column*? A *property*? An *attribute*? A *characteristic*?

What do you call each data point? A *value*? A *cell*? An *item*?

What do you call each item that users can upload? A *video*? A *webcast*? A *program*? A *film*? A *file*?

There is a lot of repetition in APIs, and the description of the same element, when it appears in different places, should always be *exactly* the same. For example, suppose that many different functions take a display name as a parameter. If you notice that some functions have a parameter named `display-name` and

others have a parameter named `title`, you can point this out to the designers and request consistency in naming. Regardless of the parameter name, make the descriptions match. Choose one phrase, such as, "A text description suitable for display" or "A descriptive display string," and always use the same phrase.

In addition to making things easier for translators, you are reducing ambiguity. If the wording is different, it makes the reader wonder whether the functionality is also different.

Consistency in general is important in any document. In an API reference it can be vital, as it can change the meaning of elements with an incorrect font or missing description.

Example API Entry

Here is an example of a typical API entry from the MSDN database. This reference is available online with more formal formatting at http://msdn.microsoft.com/en-us/library/kb60e741(v=VS.90).aspx.

This entry includes the method name, software, description, basic syntax, remarks on its use, example code, requirements, and a list of related methods.

getDate Method

Returns the day-of-the-month value in a Date object using local time.

other

```
function getDate() : Number
```

Remarks

To get the date value using Coordinated Universal Time (UTC), use the `getUTCDate` method.

The return value is an integer between 1 and 31 that represents the day-of-the-month value in the Date object.

Example

The following example illustrates the use of the `getDate` method.

other

```
function DateDemo(){
   var s = "Today's date is: ";
   var d = new Date();
   s += (d.getMonth() + 1) + "/";
   s += d.getDate() + "/";
   s += d.getFullYear();
   return(s);
}
```

Requirements

Version 1

Applies To:

Date Object

See Also

`getUTCDate` Method
`setDate` Method
`setUTCDate` Method

Avoid overloading terms

As much as possible, avoid using the same term to mean different things in different parts of the manual. If you choose the term *property* to refer to the columns of a table, don't use *property* to refer to characteristics that customize operation of the product for a particular user. Use *settings* or *initialization parameters* instead. Or if you use *field* to mean an element in a database, don't use it to also mean a space on the screen that a user types something into.

If you can't avoid reusing a term, qualify it whenever the context doesn't make the meaning obvious. For example, suppose you choose the term *cell* for a data item in a table, and you also need to talk about cells in a pattern that can be repeated to fill a region. Use *table cell* instead of just *cell* unless it's clear that you are talking about tables; use *pattern cell* instead of just *cell* unless it's clear that you are talking about patterns.

Define terms explicitly

Don't assume that your reader will understand from context the meaning of a term or recognize that you are introducing a term that has special meaning in the manual. Instead, define the term explicitly when you first use it. For example:

"…collection of data called a *table*. Each row of the table is called a *record* and represents…"

Provide some visual cue that you are introducing a definition. In most cases, it's sufficient to put the new term in *italics*. If you introduce a great deal of unfamiliar terminology, consider using a separate paragraph format for definitions, such as a different color or a gloss in the left margin that identifies the paragraph as a new definition. Whatever convention you choose, use it consistently and explain it in the preface to your manual.

If you introduce many terms, include a glossary, either at the end of the manual or as a separate document. In generated API documents in particular, it is a good idea to put a glossary or terminology table on the home or index page, to help readers make sense of the function names.

Acknowledge alternative terminology

If other products or programming languages use different terms that the

reader might be familiar with, make a point of saying that the term you use is synonymous with other terms that the reader might know. For example:

> "This guide uses the term *field* for the data components of a structure. Various programming languages refer to these components as *data members*, *instance variables*, or *properties*."

Reinforce your usage

Use your index to reinforce the terminology that you have chosen. A reader should be able to skim the index and learn what terms you use and how they relate to one another. The levels of indexing can present an outline that helps the reader see the relationship between concepts. For example:

tables
  about, 12
  cells
    about, 13
    getting values, 24
    setting values, 28
    modifying, 25
  properties
    about, 13
    defining, 14
    supported data types, 14
  records
    about, 13
    adding, 24
    deleting, 26
    modifying, 30

If you think readers might try looking up your terms with alternative terminology, put those terms in the index with references to the terms you use. For example:

columns, *see* tables: properties
data members, *see* fields
rows, *see* tables: records

If you have to overload a term, index the term with references to the alternative meanings. For example:

cells, *see* tables: cells and patterns: cells

Describing returned information

Be careful how you use the term *returns*. For example, a function can both set an output parameter and return an error code. To distinguish the value that it provides in the output parameter from the value that it returns, consider using something like *retrieves* x to mean "sets an output parameter to x." Whatever terminology you use to disambiguate this, use it absolutely consistently.

If a function uses parameters to provide values to the caller, the manual description should make it clear that these are a specific sort of parameter. Some ways to accomplish this:

- If output parameters are used frequently, consider having separate sections for input parameters and output parameters.
- The description of the parameter can explicitly say that it's an output parameter, or the wording in the description can make it clear that the parameter is used for output. Whatever style you choose, use it consistently for all output parameters. Here are examples of different ways that you could identify an output parameter.

`size`	Output parameter that retrieves the total size in bytes of ...
`size`	[out] The total size in bytes of ...
`size`	Pointer to a floating-point number that, on successful return, is set to the total size in bytes of ...
`size`	On return, the total size in bytes of ...

In Conclusion

All API reference documents share many characteristics, and we have tried to provide this guidance in a way that you can apply it to your own projects. Keep in mind, however, that there are many possible languages and formats for an API, with new ones appearing all the time. If you find yourself struggling to apply this advice because of the particular peculiarities of your environment, don't hesitate to consult with another experienced API writer to help generate ideas for presentation, organization, and terminology. We all gain experience in different areas over the years, and by pooling our knowledge, we can continue to improve the overall quality of technical documentation.

Remember too, that as the writer of an API reference document, you may be the first person to really consider the entire user layer of the product API from the user's perspective. The members of a product team generally specialize in particular parts of the product or areas of functionality, and are not that familiar with other areas. The product architects are generally much more concerned with implementation choices than with the completeness and consistency of the user layer. This gives you a unique opportunity; by sharing your observations with the product team, you can significantly improve the product itself.

Producing excellent documentation is certainly the primary goal of a technical writer. By working closely with the product team, we can show the companies we work for that the process of creating good documentation is as useful to the development process as the finished document is to their users.

This article is the result of a collaborative effort among Expert Support Staff, with significant contributions from these Senior Technical Writers: Judy Bogart, Ellen Levy Finch, and Carli Scott.