# Generating API Docs Automatically FROM THE Source Code

By ED MARSHALL | *STC Fellow*

shutterstock.com/ vasabii

THERE IS A COMMON PROBLEM in API (application programming interfaces) documentation that will occur in any group, no matter how well-meaning or conscientious the development and writing teams are. Sooner or later, the code will diverge from the documentation. Many developers favor an approach, coined using the "single source of truth" (SSOT). This means generating the documentation directly from the source code using specially formatted comments in the code. Typically, the primary use of this approach is to generate your API reference documentation. However, you can extend the tools used to extract the comments and generate your reference documentation and conceptual, "getting started" information, and flowcharts.

This article will introduce three tools commonly used to generate reference documentation from the code:

- Doxygen—Used primarily for C APIs
- Javadoc—Used primarily for Java APIs
- Swagger—Used primarily for REST APIs

## Why should technical writers care about these tools?

API documentation is a rapidly growing area of technical communication. But it requires special skills, especially being adept with the same tools that developers use. But this field offers many opportunities for career growth and higher pay and prestige in your organization than other, more general areas of technical writing. Additionally, managers are often more amenable to "working at home"/remote telecommuting when you document APIs. Let's look at a few API basics first to give readers the context and then look at these tools.

## What is good API documentation?

The standard baseline for good API documentation is a complete, accurate set of reference information that documents all the components of an API, especially methods. This dovetails very well with the approach of documenting an API in the source code.

Although you don't need to be a professional programmer or coder to be successful documenting APIs, reading the code is a key skill for documenting APIs, a major difference from documenting GUIs. You need to be able to read code well enough to recognize the following elements of an API in the source code: methods, parameters and datatypes, return values, and error codes. By reading the source code, you will find all the components you need to document. Additionally, Doxygen and Javadoc will verify that all components are documented when you generate your documentation output from those tools and report which components are not documented. Let's look at how three of the commonly-used tools to generate API documentation from the source code work.

## Doxygen

Doxygen is a very powerful code generation tool that extracts specially formatted comments in the code and produces documentation. It supports C, C++, C#, Java, Python, and IDL. It outputs RTF, compiled HTML help, browser-based help, and LaTex (PDF). You can download it from *www.stack.nl/~dimitri/doxygen/download.html#latestsrc*.

Doxygen provides a set of predefined "tags," analogous to HTML tags, that identify the components and subcomponents implemented in an API. These tags are preceded by a special character, the backslash (\). The most commonly used tags specify parameters, return values, code snippets, notes, and cross-references to related components, usually methods. You can also use Doxygen and HTML tags to include conceptual, "getting started" information, and flowcharts directly in the source code files, and that content will appear in your generated output.

The following example shows the actual source code for a typical C method:

```
/**
  Opens a help topic in a specified help window.
  If a window type is not specified, a default
window type is used. If the window type or
default window type is open, the help topic
replaces the current topic in the window.

  \param pszFile Specifies a compiled help
(.chm) file, or a specific topic within a
compiled help file. To specify a defined window
type, insert a greater-than (>) character
followed by the name of the window type.

  \param dwData Specifies NULL or a pointer
to a topic within a compiled help file.

  \return The handle (hwnd) of the help window.

  \code
  HWND hwnd =
      HtmlHelp(
              GetDesktopWindow(),
              "c:\\help.chm::/intro.htm>mainwin",
              HH _ DISPLAY _ TOPIC,
              NULL) ;
  \endcode

  \note
  <UL>
    <LI> For backward compatibility with
WinHelp(), HH _ DISPLAY _ TOPIC and HH _ HELP _
FINDER provide the same functionality.
    <LI>A default help window contains only
the Topic pane and is not a three-pane Help
Viewer.
    </UL>

  \see HH _ HELP _ CONTEXT
*/
HH _ DISPLAY _ TOPIC(LPCWSTR pszFile, DWORD
dwData);
```

Note that all of your documentation comments precede the method in the code and appear in a standard comment block, "/* … */", except this comment block starts with a second asterisk, "/**". The second asterisk instructs the Doxygen compiler to compile everything in the comment block and include it in Doxygen's output. This example uses the following Doxygen tags:

- \param—Parameters are the variables you specify when calling a method. You need a param tag for each parameter defined for your method.
- \return—If your method returns a value, you must document it using this tag.
- \code … \endcode—This is your code snippet. Doxygen preformats the example and puts it in a monospaced font, commonly used for examples in API documentation.
- \note—This is nice-to-know, supplemental information, not needed all the time, but occasionally.
- \see—These are your cross-references to related methods. If you have multiple cross-references, precede each one with a \see tag.

Note that there is a bulleted list coded in HTML in the code example. Doxygen supports most of HTML tags so you can use those tags for formatting, as well as the tags defined in Doxygen.

Figure 1 shows the Doxygen output for the HH_DISPLAY_TOPIC method. Note the output format of this example. Doxygen provides its own cascading style sheet (CSS) and predefines the output layout. The actual method with its syntax appears first, followed by a brief description of the method. The method's parameters appear in the order they appear in the syntax, the method's return value, followed by a code example showing the use of the method, some notes or supplemental information, and lastly a cross-reference to a related method. This is the standard outline or organization for API reference information, regardless of company or tool used.

## Javadoc

Javadoc is a powerful and free code generation tool for Java APIs. As with Doxygen, it reads specially formatted comments in code. It outputs what many of the popular help authoring tools (HATs) call browser-based help, with some features that our common tools do not provide. These will be pointed out in the following examples. It also is actively supported. You need to download and install the Oracle JDK (Java Development Kit). The current version is 8.0, but the author recommends you use JDK 7 as 8.x requires ending all tags with end tags. If you are working on legacy documentation, which is likely, you will have to clean up your Javadoc comments, which could be a daunting effort in JDK 8.x without generating large numbers of warnings. You can download the JDK from *www.oracle.com/technetwork/java/javaee/downloads/index.html?ssSourceSiteId=ocomen*.

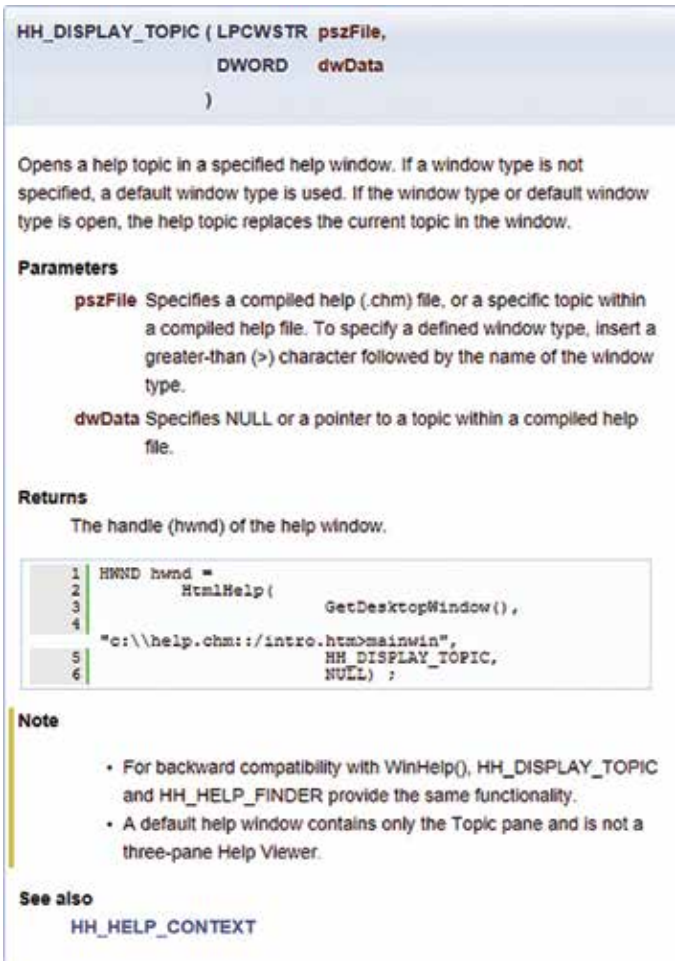



*Figure 1. Doxygen output for HH_DISPLAY_TOPIC method.*

Javadoc provides a set of predefined "tags" that specify the components and subcomponents implemented in an API. These tags are preceded by a special character, the at symbol (@). The most commonly used tags specify parameters, return values, code snippets, notes, and cross-references to related components, usually methods.

Figure 2 shows a sample method, login, for a Java API. As with Doxygen, all of your Javadoc documentation comments precede the method in the code. They appear in a standard comment block, "/* … */", except this comment block starts with a second asterisk, "/**". The second asterisk instructs the Javadoc compiler to compile everything in the comment block and include it in Javadoc's output. This example uses the following Javadoc tags:

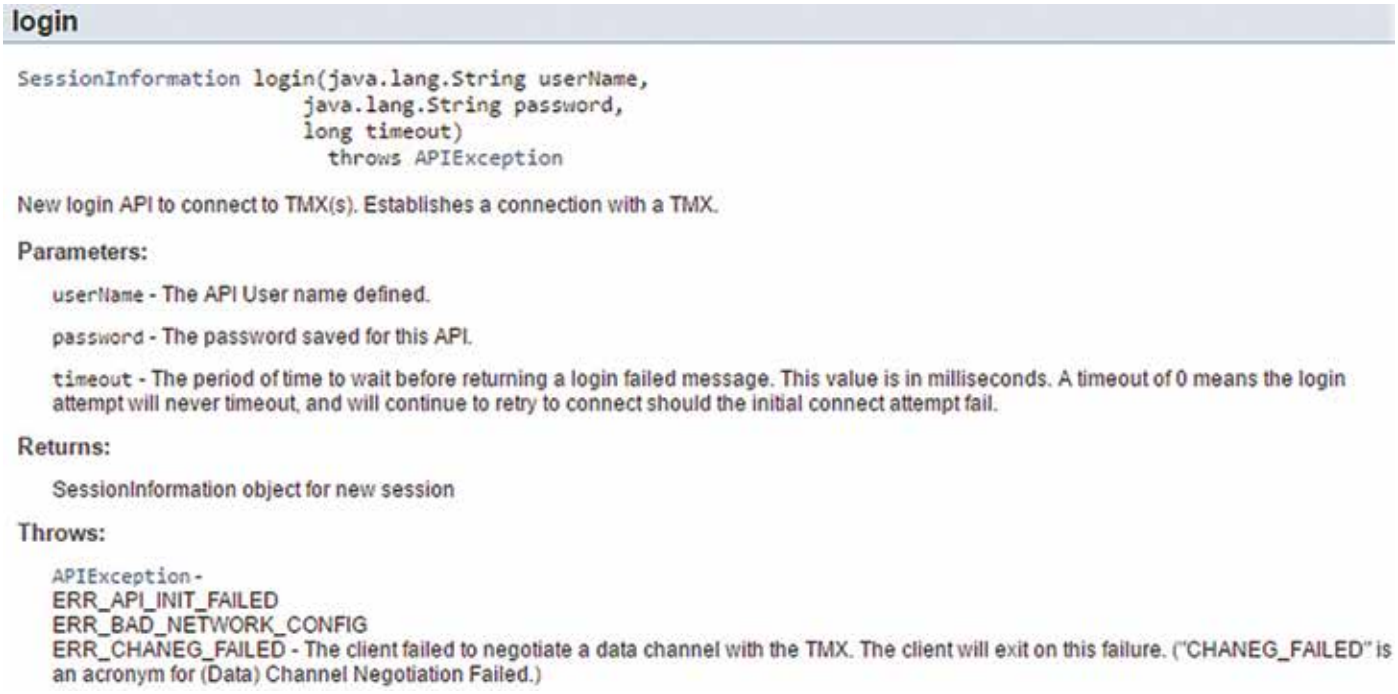
- @param—Parameters are the variables you specify when calling a method. You need a param tag for each parameter defined for your method.
- @return—If your method returns a value, you must document it using this tag.
- @throws—In Java, when an error is thrown, it is called throwing an exception, so Javadoc uses a throws tag.

Javadoc supports most HTML tags so you can use those tags for formatting, as well as the tags defined in Javadoc.

```
/**
    * New login API to connect to TMX(s).
    *
    * Establishes a connection with a TMX.
    * @param userName The API User name defined.
    * @param password The password saved for this API.
    * @param timeout The period of time to wait before returning a login failed message.
    * This value is in milliseconds.
    * A timeout of 0 means the login attempt will never timeout,
    * and will continue to retry to connect should the initial connect attempt fail.
    * @return SessionInformation object for new session
    * @throws APIException
    * <br>ERR_API_INIT_FAILED
    * <br>ERR_BAD_NETWORK_CONFIG
    * <br>ERR_CHANEG_FAILED - The client failed to negotiate a data channel with the TMX.
    * The client will exit on this failure.
    * ("CHANEG_FAILED" is an acronym for (Data) Channel Negotiation Failed.)
    */
   SessionInformation login(
       String userName,
       String password,
       long timeout,
       throws
APIException;
```

*Figure 2. A sample method (login) for a Java API.*

Figure 3 shows the Javadoc output for the login method. Note the output format of this example. Javadoc provides its own cascading style sheet (CSS) and predefines the output layout. The actual method with syntax appears first, followed by a brief description of the method. The method's parameters appear in the order they appear



**login**

```
SessionInformation login(java.lang.String userName,
                         java.lang.String password,
                         long timeout)
                           throws APIException
```

New login API to connect to TMX(s). Establishes a connection with a TMX.

**Parameters:**

`userName` - The API User name defined.

`password` - The password saved for this API.

`timeout` - The period of time to wait before returning a login failed message. This value is in milliseconds. A timeout of 0 means the login attempt will never timeout, and will continue to retry to connect should the initial connect attempt fail.

**Returns:**

SessionInformation object for new session

**Throws:**

`APIException` -
ERR_API_INIT_FAILED
ERR_BAD_NETWORK_CONFIG
ERR_CHANEG_FAILED - The client failed to negotiate a data channel with the TMX. The client will exit on this failure. ("CHANEG_FAILED" is an acronym for (Data) Channel Negotiation Failed.)

*Figure 3. The Javadoc output for the login method.*

By learning how to use tools such as Doxygen, Javadoc, and Swagger that extract specially formatted comments from the code, we enhance our value to our organizations and to our users.

in the syntax, the method's return value, and the errors returned for this method.

Figure 4 shows the main topics in a Javadoc system. The Overview is the "welcome page" to your Javadocs, by default. You can also use Javadoc and HTML tags to include conceptual, "getting started" information, and flowcharts directly in the source code files and that content will appear in the Overview topic.


*Figure 4. The main topics in a Javadoc system.*

Another useful and unique feature of Javadoc is that it automatically creates an index of all your API components where users can search for an API component by name. Each entry in the index is hyperlinked to the documentation for that component. If you click on the letter L, Javadoc will display all the methods that start with the letter L. So, once you find the component in the index, you can click on the link to go directly to the information on that component. This is a very useful feature for developers as they can quickly find a method if you know its name or see all the methods starting with a chosen letter.

## Web Services APIs

Web services APIs are a newer type of APIs but are becoming very popular. There are two types:

1. Representational State Transfer (REST)—All parameters and values called are specified in the request. To call a REST API, you specify a base URI (URL): *http://example.com/resources/.*

   REST APIs use standard HTTP actions for their operations:
   - GET—Lists/retrieves data from a record. Doesn't change the record.
   - POST—Posts/stores or writes data into a record. Creates a new entry in a collection/data in a record.
   - PUT—Replaces the entire collection with another collection.
   - DELETE—Removes data from a record. Performs the same operation every time.

2. Simple Object Access Protocol (SOAP)—A method is predefined with all request parameters and values specified. You do not specify which of the four HTTP actions to call for a method. Each method is predefined to use a specific action. For example, the getCustomerInfo method uses the GET HTTP action.

## Swagger

Swagger is a popular tool for documenting Web service, APIs. It not only extracts documentation from your source code but has the added benefit of providing an interactive way of trying the API. That is, you can specify values for the parameters and "test-drive" the API by calling the resource and see the result.

An excellent interactive Swagger example can be found at *http://petstore.swagger.io/.* This example provides a pet store application where you can add a new pet to the store, update an existing pet, find pets by status, tags, or ID, delete a pet, and perform other operations. The Web-based UI is populated from the source code.

Let's look at an extract from that example. The following shows the code for adding a pet:

```
"schemes":["http"],"paths":{"/
pet":{"post":{"tags":["pet"],
"summary":"Add a new pet to the store",
"description":"","operationId":"addPet",
"consumes":["application/json","applica-
tion/xml"],
"produces":["application/xml","application/
json"],
"parameters":[{"in":"body","name":"body","-
description":"Pet object that needs to be
added to the store",
"required":true,
"schema":{"$ref":"#/definitions/Pet"}}],
"responses":{"405":{"description":"Invalid
input"}},
"security":[{"petstore _ auth":["write:-
pets","read:pets"]}]},
```

Figure 5 shows the Swagger UI for adding a pet. The body field on the left is where you specify values to test; the Model Example Value field on the right shows the syntax. In this case, I added a new pet named Misty with an id of 10.

To test the interactive aspect of Swagger, click the "Try it out!" button in the lower left to display the response and any response codes to your request.

The Response Body section shows that a pet named, Misty, with an id of 10 was added. The Response Code indicates if the operation succeeded or what that problem was. In this case, the response code is 200, indicating success (see Figure 6).

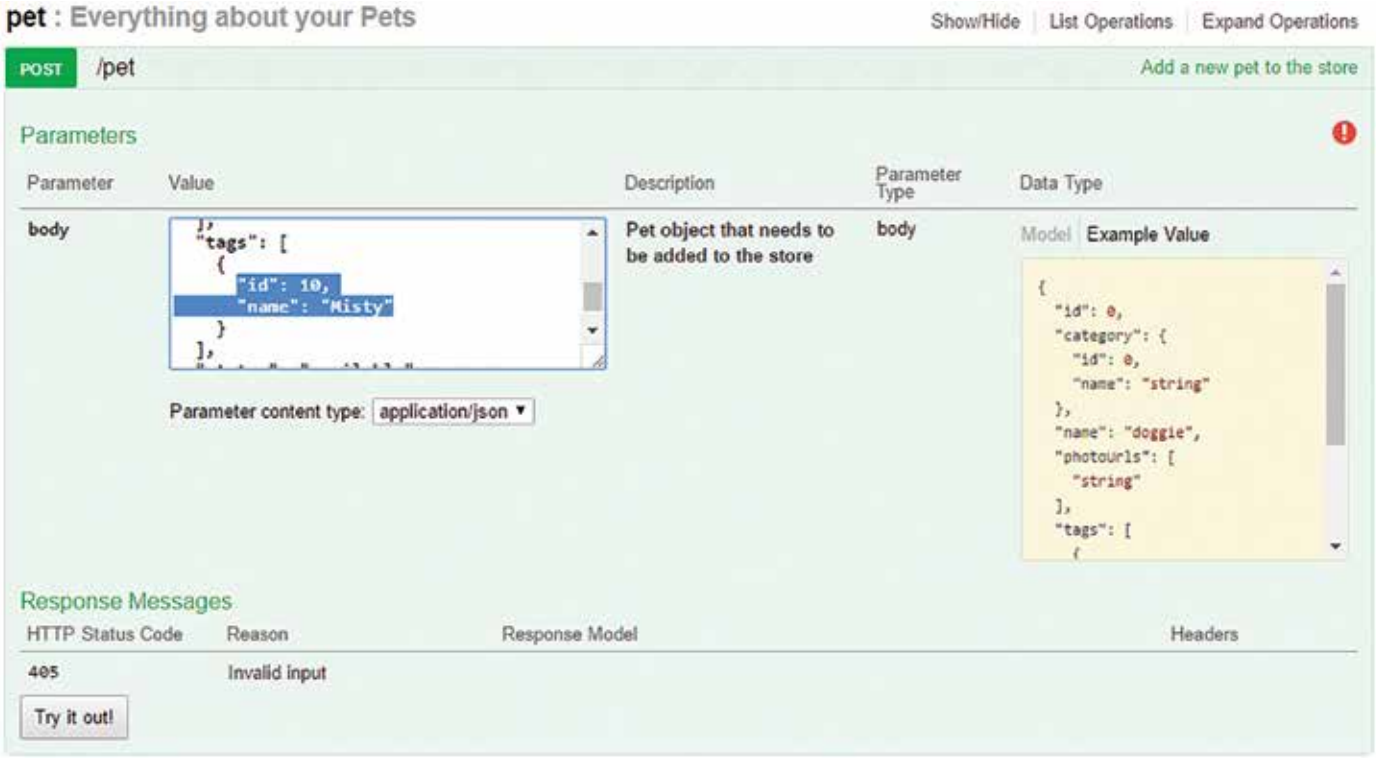



*Figure 5. The Swagger UI for adding a pet.*


*Figure 6. The Response Body and Response Code.*

## Summary

Developers prefer to generate the reference information for an API directly from the source code to help ensure that all the components of an API are documented and nothing was missed. Otherwise, the code and documentation will get out of sync eventually. Someone, either a developer or a technical writer, has to add those comments. We, as technical writers, can provide extra value by using the writing, analyzing, editing, proofreading, and organizational skills we've learned and use in other areas of technical writing. By learning how to use tools such as Doxygen, Javadoc, and Swagger, that extract specially formatted comments from the code, we enhance our value to our organizations and to our users.

ED MARSHALL *(*ed.marshall@verizon.net*) is an independent consultant technical writer and the sole proprietor of Marshall Documentation Consulting (*www.MarshallDocumentationServices.com*), with over 28 years of experience. He specializes in technical documentation for developers, including APIs (application programming interfaces), SDKs (software developer's kits), Web services products, etc. Over his career, he has developed expertise in using tools to "let the computer do the work," such as advanced tools for editing files, comparing files, and searching and replacing text. He is an STC Fellow and has given many presentations at local STC chapters and Summits, the WritersUA conferences, Information Development World, tekom, and many other events. He is a Certified MadCap Advanced Developer (MAD). He can be reached on LinkedIn at* www.linkedin.com/pub/ed-marshall/0/501/898 *and on Twitter* @EdMarshall.