The Tao of Fusedoc

When using Fusedocs, we have expectations about how they will be used. Consistency is the key to successful Fusedocs. Fusedoc is a standard for embedded program documentation. That is, every fuse in a Fusebox application should have a Fusedoc embedded at the top of the file. This ensures the ability to easily examine the documentation for any fuse. The following listing shows a sample Fusedoc as it might appear in a typical fuse.

```
<!---
<fusedoc fuse="act_Login.cfm" language="ColdFusion" specification="2.0">
 <responsibilities>
  I validate a user's login information.
 </responsibilities>
 <properties>
  <property name="Date" value="01 Jan 02" comments="Sample for book">
 </properties>
 <note>
  Notes can be used to capture information that doesn't have a specific Fusedoc element.
 </note>
 <io>
  <in>
   <string name="XFA.onSuccess" optional="No" comments="Use if process succeeds">
   <string name="XFA.onFailure" optional="No" comments="Use if process fails">
   <string name="userID" optional="No">
   <string name="password"
         optional="No"
     comments="Hash() this string before comparing to password in database">
  </in>
  <out>
   <string name="userID" scope="client" oncondition="User is valid">
   <string name="firstName" scope="client" oncondition="User is valid">
   <string name="email" scope="client" oncondition="User is valid">
  </out>
 </io>
</fusedoc>
--->
```

Fusedocs are written before the application's code is written. We'll get into this in much detail shortly; the important idea is that the emphasis on Fusedoc comes during the architectural design stage of the development project, as opposed to the more traditional approach of documenting code as it is written.

The process of writing Fusedocs is both an analytic and creative one. The objective is to create a blueprint for the application that coders can then pick up and work from. In truth, Fusedocs are more like a parts list that a contractor gives a builder than a

blueprint. Fusedocs tell builders exactly what parts will be needed to successfully complete the job in question. That's what Fusedocs do--detail the variables and data that are available to the coder, along with some explanation of what's expected from the fuse in question. Armed with this information, the coder is then able to build the fuse without the need to consult with outside resources.

If the coder is supposed to be able to write the fuse without consulting with outside resources, it's clear that a Fusedoc needs to be explicit about what the architect expects from the fuse. Fusedocs are about *what* the fuse is to do, not about *how* to do it. This is a fine line to walk, but it is important.

For example, the Fusedoc in the listing above tells the fusecoder to validate the user according to the userID and password. The fusecoder might choose to run an SQL query that asks for records matching the userID and password, and check the output for records. Another possibility would be to loop over a recordset of userIDs and passwords, looking for a match. The approach that works best is up to the fusecoder.

The creation of good Fusedocs is a skill learned only through the experience of having others write code according to your Fusedocs. In this regard, writing Fusedocs can be considered an art.

The Art of Writing Good Fusedocs

Although software development is generally recognized as a technical discipline, writing high-quality Fusedocs can be an elusive pursuit. Success depends on experience, on the refinement that comes only through trial and error. This is where the "art" comes in; excellent Fusedoc creation is a skill that cannot be communicated through technical instruction alone. Every Fuseboxer to whom we've spoken has agreed with the observation that the first Fusedocs we write are far from adequate for the job. Hal Helms describes the first time he sent Fusedocs to an outsource for coding this way:

"A week later I got the coded fuses back, and when I looked at them, I immediately wanted to know what idiot had been messing with my Fusedocs. It just wasn't possible that what came back to me was what I sent out. Of course, it was; I just didn't realize how many assumptions I had made when I wrote the Fusedocs."

If the creator of the system has this sort of experience on his first time out, it's no surprise that the rest of us have experienced a similar situation. We all make assumptions about the systems we're building. For that reason, a sort of motto for writing good Fusedocs has come to be, "When in doubt, put it in." A word of caution, though; this does *not* mean that we should include every variable imaginable in every fuse's Fusedoc. In fact, we want to do exactly the opposite. We want to include everything the coder needs to write the fuse, and *nothing more*. We state this as the first rule of Fusedocs:

> ***Include everything a coder needs to write the fuse, and nothing more.***

Again, the artistic aspect of writing Fusedocs becomes apparent as we consider the fine distinction between "everything the coder needs" and "nothing more."

Everything the Coder Needs

We have habits about the way we work that we might not realize we have. For instance, we sometimes use a variable named request.dsn to store the name of the ODBC datasource that our system is using. Similarly, we use a variable named request.dbt to store the dbtype; in the case of an ODBC datasource, its value would be **ODBC**. This makes it easy to transport a query fuse from one application to another.

The use of these variables makes sense, but it can cause a problem when we start writing Fusedocs. The fact that we use these variables as part of every query we write means that they become a sort of background noise in our coding environment. They're always there, and we can easily forget about their importance. If we forget to include them in a Fusedoc that we write for someone in-house, there's not much impact. The in-house coder knows our local conventions and will probably write the query after that fashion, so no harm is done. On the other hand, if we send the fuse to an outsource for coding, the coder has no idea about the common use of those two variables in our shop, and he cannot successfully complete the fuse without them. A Fusedoc must include everything the coder needs to successfully write the fuse.

The art of writing Fusedocs includes the decision about what information the coder really needs to write the fuse. Our example with request.dsn and request.dbt is a good illustration of this principle. If we're writing Fusedocs for in-house coders, we probably won't include those two variables. If we're writing Fusedocs for outsourcing, though, we had better put them in or we'll risk delaying the project while we clarify

these issues with our coders.

This is where the "when in doubt..." motto comes in. While we're writing a Fusedoc, if we wonder whether to put in a particular piece of information, we'll go ahead and put it in. In the case of our datasource and dbtype example, including them would make the Fusedoc accessible to both in-house and out-sourced coders--the sign of a good Fusedoc. Again, we resist the urge to throw in everything that comes to mind, needed or not.

To be effective Fusebox architects, we need to be familiar with the elements that comprise the Fusedoc vocabulary. These elements can be found in the Fusedoc document type definition (DTD), available at **www.fusebox.org**. The next sections cover the details of each Fusedoc element and how to use it.

Elements of Fusedoc

When we get right down to it, Fusedocs are quite basic collections of information. They describe what a fuse is going to do (its **Responsibilities**); who created it, when, and why (its **Properties**); and what variables are present when it runs and are to be created by it (its Input-Output, or **IO**).

Of these areas, only Responsibilities is a required section in any Fusedoc. The others can be used at your discretion. Keep in mind, though, the first rule of Fusedocs.

Responsibilities, Properties, and IO sections are commonly found in most Fusedocs, whereas an Assertions section is a bit more esoteric. In this section, we'll look at Responsibilities, Properties, and Assertions. Because the IO portion of a Fusedoc is the meat of it, we'll save it for the next section.

Responsibilities

The Responsibilities element of a Fusedoc is the most loosely defined portion of the Fusedoc and, at the same time, its most important. It is a plain language description of what the fuse is expected to do.

Responsibilities should be written in first-person active voice. This style is not required, but we strongly recommend it. There is something about framing tasks in first-person language that makes them seem more real. By placing yourself in the

metaphorical shoes of the fuse, it becomes much easier to examine what jobs you need to do and what information you'll need to do them. The first-person point of view also tends to result in descriptions of what must be done, rather than how to do it.

Here's a listing that shows an example of a very descriptive Responsibilities section.

```
<responsibilities>
  I display a form for the user to create a new account.  I have fields for userID, firstName, lastName,
  email, and password.  If the user clicks the Create button, I return XFA.btnCreate.
</responsiblities>
```

And here's a listing that shows a more terse example of a Responsibilities section.

```
<responsibilities>
  I display a form for the user to create a new account.
</responsiblities>
```

Notice the difference between the two listings. Whereas the first one is detailed in its narrative, the second is terse. This is indicative of the artistic nature of writing Responsibilities sections.

Under the original Fusedoc specification, the data-related sections were not as tightly defined as they are in Fusedoc 2.0. As a result, more emphasis was placed on the Responsibilities section to carry the weight of describing what the fuse was about. Architects who have used Fusedocs under both systems tend to write more verbose Responsibilities sections, like the one in the first example.

Under Fusedoc 2.0's XML definition, the <io> section of the Fusedoc carries much more well-defined information than the equivalent Attributes section under Fusedoc 1.0 would have. As a result, the Responsibilities section does not need to be as narrative under the new version. Architects who have been introduced to Fusedocs after the creation of version 2.0 tend to write more terse Responsibilities sections like the one in the second example, because the information about form fields, variables, and so on is detailed in the <io> section of the Fusedoc. This approach is preferable because the technical details are structured in a much more standardized fashion, and the Responsibilities section contains only the information that is not communicated elsewhere in the Fusedoc.

After we've described what the fuse is expected to do, we might want to include some information about who designed the fuse, when it was designed, and so on. This is the Properties section's job.

Properties Element

Properties are simply the characteristics of an object. For example, as a person, you have some characteristics such as height, eye color, and age. The optional <properties> element in a Fusedoc uses the elements <history>, <property>, and <note> to capture this type of information.

Web developers are often exposed to the idea of objects and properties without being formally aware of it. Javascript, as a language based on Java, uses an object-oriented style of notation. You can refer to properties of the window, for example. Most of us web developers are familiar with the location property of the window object because we can use it to redirect to a new URL:

```
<script language="Javascript">
  window.location='myPage.cfm';
</script>
```

In similar fashion, a Fusedoc can store a set of properties for its fuse. These can be described using the history, property, and note elements.

History Elements

Purpose: Specifies a milestone in the lifecycle of the fuse.
Attributes:

author--The author of the milestone.
email--The author's e-mail address.
type--The type of the milestone.
Values: Create (default)
Update
date--The date of the milestone.
role--The role of the author.
Values: Architect
DBA
FuseCoder
TextEditor
Graphics

Any text
comments--Any comments you want to add

The history element is useful for recording the life of a fuse's code with brief comments. For example, when an architect creates the Fusedoc, a history element exists with a type of Create. If the Fusedoc is subsequently modified for some reason, a second history element is added with a type of Update. We can insert a history element any time we want to record a milestone for the Fusedoc. This listing shows a set of history elements:

```
<history author="Jeff Peters"
         email="jeff@grokfusebox.com"
          date="01/22/02"
          role="Architect"
          type="Create" />
<history author="Stan Cox"
          role="FuseCoder"
          date="01/23/02"
          type="Update"
      comments="Started coding" />
<history author="Stan Cox"
          role="FuseCoder"
          date="01/24/02"
          type="Update"
      comments="Finished coding">
  This fuse should be retested after next week's DB upgrade.
</history>
```

The purposes for which we might use history elements have very few restrictions. We can use them in whatever fashion will satisfy local requirements. If we need longer comments, we can employ an end tag like the third example shown earlier, instead of trying to put a lot of text in the comments attribute. History and note elements are also among the few elements that are added after coding of the fuse begins.

Because the creation of Fusedocs should be a planning exercise, it's important to avoid the mindset that says we can update our Fusedocs whenever the fuse's code requires us to. Although it's important that the Fusedoc and the code agree, it's much more important to the discipline of good planning that Fusedocs be as complete as possible from the outset, rather than being changed to suit shifting coding needs as the project progresses.

Property Elements

Purpose: Specifies a property of the fuse.
Attributes:

name--The name of the property.
value--The value of the property.
comments--Any comments you want to add.

Property elements are the most flexible of all Fusedoc elements. Property elements are *not* the same as the properties element that encloses them. A Fusedoc has only one properties element; any number of property elements can exist within a Fusedoc.

You can capture almost any property you can imagine with a property element. Its attributes--name, value, and comments--work in much the same way as Windows INI files or registry entries. The name attribute specifies what the property is, and the value attribute specifies the value that is assigned to the property. The optional comments attribute offers the ability to add any detail that is not made clear in the name and value. This listing has sample properties elements:

```
<property name="client" value="SurveyCity" />
<property name="copyright"
      value="2002, ThirdWheelBikes"
    comments="Reverts to GrokFusebox.com in 2004" />
<property name="clientsFavoriteColor" value="Cornflower Blue" />
```

The last element in this listing serves to show how open-ended the property element can be. If there's a need to communicate this information to the fusecoder, so be it.

For slightly more formal comments on the fuse, note elements do a good job.

Note Elements

Purpose: Specifies a note for the fuse.
Attributes:

author--The author of the milestone.
date--The date of the milestone.

Note elements are an easy way to toss quick comments into a Fusedoc. Some authors use them in much the same way that you might use a history element; however,

whereas history elements are much more specific in the attributes they capture, note elements are much more free-form. Note elements capture text between opening and closing tags, so they allow much more text to be included than the limited comment attribute in a history element. This listing shows some sample note elements:

```
<note author="jeff@grokfusebox.com" date="01/22/02">
  Stan, make sure you get the spelling of the Wryxizschokville office  right.  Mr. Wryxizschok is very
  sensitive about that particular town.
</note>

<note author="stancox@fusebox.org" date="01/23/02">
  When making tables, use blinking green background only.  This is very important to the success of
this
  project.
</note>

<note author="brody@fusebox.org" date="01/23/02">
  Blinking backgrounds are out.  I helped Julie come to her senses.
</note>
```

Notes are a good way to encourage communication between coders. They are also useful for recording local conventions that might not be well known, or might be easily forgotten if the project is set aside for a while.

For example, if an application is concerned with measurements that are governed by a standards committee, you could record the source for the standards in a note. If the fuse has to be changed to accommodate future changes in the standard, the information could make the fix much easier.

The next section is where we'll really get into the meat of the Fusedoc specification--the <io> element. This element is used to encapsulate all the input and output elements that are used for a Fusedoc. The collection of elements that is available within the Fusedoc 2.0 specification is capable of describing any data needs you might have within a fuse. Understanding the elements and their use is one of the keys to the art of writing Fusedocs.

IO Element

Most of the Fusedoc elements are fairly easy to decipher from their names, and the <io> element is no exception. Its job is to contain the elements that describe the input to the fuse (the <in> element) and the output the fuse is expected to create (the <out> element). In similar fashion, variables that are available as input to the fuse but should

not be altered in any way by the fuse can be specified in a <passthrough> element.

In Element

The <in> element is another one of those almost self-explanatory Fusedoc elements. It contains elements to describe variables that are available for the fuse to use. An <in> element can contain any of the following elements for describing data:

- String
- Number
- Boolean
- Datetime
- Array
- Structure
- Recordset
- List
- Cookie
- File

Each of these elements falls into one of two categories: **simple** or **complex**. Cookie and file elements are an exception to this, but we'll get to that later. Simple datatypes are those that represent a single value. String, number, boolean, datetime, and list are all simple datatypes. Complex datatypes are those that can contain multiple values, such as arrays, structures, and recordsets. (Lists can store multiple values, but because they are actually strings, they are categorized with the simple datatypes). Table 8 shows each category and the datatypes within them.

DATATYPES		
Simple	**Complex**	**Other**
String	Array	Cookie
Number	Structure	File
Boolean	Recordset	
Datetime	List	

Table 8 - Data Types

The elements <cookie> and <file> are a bit different from other Fusedoc elements. Although they are essentially simple, they are not really datatypes per se. We'll look at each of the elements we can use within the <in> element. Following the specifications for each element, we'll look at some examples to help clarify the definition.

<out> Element

The <out> element is used to contain elements for any variables with values that are created by the fuse for use in the next fuseaction. It is formed in exactly the same manner as the <in> element, and might contain any of the elements specified in the detailed discussion earlier. The <out> element is optional.

<passthrough> Element

The <passthrough> element is used to contain elements for any variables that are available as input to the fuse, and are passed along unchanged for use in the next fuseaction. The <passthrough> element is formed in the same manner as the <in> element, and it might contain any of the elements specified in the detailed discussion earlier. The <passthrough> element is optional.

Simple Datatype Elements

The simple datatype elements are string, number, boolean, datetime. The following sections detail the attributes that each of these elements can take.

NOTE: In "Fusebox: Developing ColdFusion Applications", I included the list element among the simple datatypes. Although a list is technically just a specialized string, its ability to store multiple members makes it a complex datatype.

String Element

Purpose: Specifies a variable that stores a string value.
Attributes:

name--The name of the variable (required).

scope--The scope of the variable (optional).

Values: Application
Attributes
Caller
Cgi
Client
Form
FormOrUrl
Request
Server
Session
Url
Variables (default)

comments--Any comments we want to include.

mask--Details for expected format of the string. No standard exists for creating masks as of Version 2.0 of Fusedoc; ColdFusion string masks are common practice.

onCondition--The condition that causes the variable to exist. For example, a state variable might only exist if the country variable represents a country that has states. Used in conjunction with the optional attribute.

format--The format in which the variable is encoded.

Values: CFML (default)
WDDX

optional--Whether the variable is optional.

Values: True
False (default)

default--The default value taken by the variable. For example, a variable named homeTown might have a default value of **Springfield**.

This listing gives examples of Fusedoc elements for several string variables: