How to Implement DOTGO Engines

CMRL Version 1.0

Copyright ⓒ 2009 DOTGO. All rights reserved.

Contents

1 Introduction **3**

2 A Simple Example **3**
2.1 The CMRL Document . . . 3
2.2 The Engine Document . . . 4
2.3 What Does It Do? . . . 5
2.4 How Does It Work? . . . 5
2.5 Bottom Line . . . 6

3 Structure of a Request **6**
3.1 The Request . . . 7
3.2 The Query . . . 7
3.3 The Instruction . . . 8
3.4 Separating the Path from the Argument . . . 8

4 What Is Posted to an Engine **8**
4.1 Parts of the Request . . . 9
4.2 “Processed” Parts of the Request . . . 9
4.2.1 Location . . . 9
4.2.2 “Corrected” Path . . . 9
4.2.3 “Corrected” Argument . . . 9
4.3 Session Variables . . . 10
4.4 Summary . . . 10

5 The CMRL Terminating Nodes **10**
5.1 The `<message>` Tag . . . 10
5.2 The `<engine>` Tag . . . 11
5.3 The `<rss>` Tag . . . 11
5.4 The `<query>` Tag . . . 12
5.5 The `<block>` Tag . . . 12
5.6 Terminating Nodes are Recursively Evaluated . . . 12

6 Summary **13**

1 Introduction

A DOTGO engine is a web-based application program that accepts input from DOTGO and returns output to DOTGO. You can use engines to create dynamic mobile services, connecting DOTGO to the internet, databases, and other computer-based services. Because an engine is a web-based application program, it can be written in any web programming language (e.g. Perl, PHP, Java, Python, etc.)—the only condition on an engine is that it must accept input in a specified format and must return output in a specified format.

This document describes how to implement DOTGO engines. The document assumes that you have some familiarity with DOTGO and CMRL (obtained, e.g., by reading *A Brief Introduction to DOTGO*). The document also assumes that you have some familiarity with keywords (obtained, e.g., by reading *Keywords in CMRL*) and session variables (obtained, e.g., by reading *Session Variables in CMRL*). The document describes example engines written in Perl, although similar concepts apply for engines written in any other web programming language.

2 A Simple Example

Let's start by considering a simple example. To be specific, we will imagine a mobile service implemented under the internet domain "example.com." In other words, we will imagine a mobile service that is accessed by sending a text message starting with the internet domain name "example" to the phone number DOTCOM (368266) for which CMRL documents and web-based application programs are hosted at the internet domain "example.com." The source files for the example described in this section are available online at http://dotgo.com/Support/Documentation/example3/index.cmrl and http://dotgo.com/Support/Documentation/example3/something.cgi.

2.1 The CMRL Document

Consider the following CMRL document, which we are imagining is hosted as the file "index.cmrl" at the internet domain "example.com":

```
<?xml version="1.0" encoding="UTF-8"?>
<cmrl xmlns:dotgo="http://dotgo.com/cmrl/1.0">
  <match pattern="something">
    <engine href="http://example.com/cgi-bin/something.cgi"/>
  </match>
</cmrl>
```

2.2 The Engine Document

Consider the following Perl program, which we are imagining is hosted as the web-based application program "something.cgi" at the internet domain "example.com":

```
#!/usr/bin/perl
```

```
use CGI qw(:standard);
use strict;

my $color = param("sys_argument");

my @red_things = ("Apples", "Lips", "Hearts");
my @green_things = ("Frogs", "Shamrocks", "Olives");
my @blue_things = ("Jeans", "Blueberries", "Blue jays");

my $thing;
if ($color eq "red") {
  $thing = $red_things[int(rand($#red_things+1))];
} elsif ($color eq "green") {
  $thing = $green_things[int(rand($#green_things+1))];
} elsif ($color eq "blue") {
  $thing = $blue_things[int(rand($#blue_things+1))];
}

my $content;
if ($thing) {
  $content = "$thing are $color";
} else {
  $content = "Try red, green, or blue";
}

print header;
print <<EOF
<message>
  <content>$content</content>
</message>
EOF
;
```

2.3 What Does It Do?

So what does it do? The query “example something red” produces one of the responses (at random) “Apples are red,” “Lips are red,” or “Hearts are red.” Similarly, the query “example something green” produces one of the responses “Frogs are green,” “Shamrocks are green,” or “Olives are green,” and the query “example something blue” produces one of the responses “Jeans are blue,” “Blueberries are blue,” or “Blue jays are blue.” The query “example something yellow” produces the response “Try red, green, or blue.”

2.4 How Does It Work?

So how does it work? To be specific, let's consider accessing the service by sending the text message “example something red” to the phone number DOTCOM (368266).

First, the system uses the first word of the query “example” and the phone number DOTCOM (368266) through which the query is received to determine that the query should be routed to example.com. Next, the system retrieves the file index.cmrl from example.com. Next, the system compares the remainder of the query (i.e. everything after “example,” which in this case is the string “something red”) against the match patterns specified in the file index.cmrl in order to resolve the query. In this case, there is only one match pattern specified in the file index.cmrl—the match pattern “something”—which is matched by the first token “something” of the string “something red” and which resolves to the `<engine>` tag. Next, the system calls the engine “something.cgi” referenced by the `<engine>` tag, posting the remainder of the query (i.e. everything after “something,” which in this case is the string “red”) to the engine as the parameter “sys_argument.” The engine uses the value of the parameter “sys_argument,” which it assigns to the variable “$color,” to construct the response, which it assigns to the variable “$content.” Finally, the engine prints the value of the variable “$content” within a CMRL `<message>` tag, which produces the message “Apples are red” (or similar) returned by the system.

2.5 Bottom Line

There are several points to note from this example:

1. The request is made up of three distinct parts: one part (in this case the phone number DOTCOM plus “example”) designates the location of the CMRL document, one part (in this case “something”) designates the “path” that is compared against the match patterns specified in the CMRL document, and one part (in this case “red”) designates the “argument” that is passed to the engine.

2. There is no explicit delimiter between the path and the argument. The system must route the request (by comparing the request against the match patterns specified in the CMRL document) in order to determine which portion of the request is the path and which part of the request is the argument.

3. The system posts the argument to the engine as the parameter “sys_argument.” The system also posts other parameters to the engine (as described below).

4. The engine returns a CMRL `<message>` tag, which is the simplest example of a CMRL “terminating node.”

In a nutshell, this is how DOTGO engines work: the system extracts the argument from the request by comparing the request against the match patterns specified in the CMRL document, the system posts the argument (and other parameters) to the engine, and the engine returns a CMRL terminating node.

3 Structure of a Request

Because it is fundamental toward understanding how the argument is extracted from the request, let's have a closer look at the structure of a request. The parts of a request are illustrated in Figure 1. We will consider in turn the *request*, the *query*, and the *instruction*.

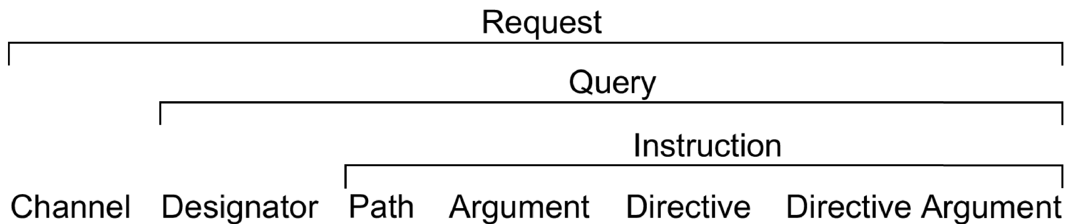

Figure 1: Structure of a request.

3.1 The Request

The *request* is the fundamental unit of interaction with DOTGO. A request is made up of a *channel* and a *query*. A channel is a specification of the physical device through which the request was received. For purposes of this document, a channel is one of the phone numbers DOTCOM (368266), DOTEDU (368338), DOTGOV (368468), DOTNET (368638), or DOTORG (368674).

3.2 The Query

The *query* is the text string sent by the user. A query is made up of a *designator* and an optional *instruction*. A designator is the first word of a query. A designator is either an internet domain name (with or without an explicit specification of a top-level domain) or a URL. The channel and the designator together specify the location of a CMRL (or HTML or other) document.[1]

3.3 The Instruction

The *instruction* is the remainder of the query (after the designator). An instruction is made up of an optional *path*, an optional *argument*, an optional *directive*, and an optional *directive*

[1]The system follows certain rules to determine the location of a document from the channel and the designator as follows: If the designator specifies a file, then the system seeks to retrieve the file; if the file is found and the extension of the file is ".cmrl," then the system interprets the file as a CMRL document; if the file is found and the extension of the file is not ".cmrl," then the system seeks to render the file; if the file is not found, then the system returns an error. If the channel and the designator specify a directory, then the system seeks to retrieve the file "index.cmrl;" if the file is found, then the system interprets the file as a CMRL document; if the file is not found, then the system seeks to retrieve the directory (which may, at the discretion of the host web server, result in a request for "index.html," "index.php," etc.); if a file is found, then the system seeks to render the file; if a file is not found, then the system returns an error.

argument. A path is the portion of a query that is matched to match patterns specified in a CMRL document. An argument is the portion of the query after the path and before the directive. A directive is a DOTGO reserved word. In version 1.0 of CMRL, the DOTGO reserved words are *follow* and *unfollow*, *register* and *unregister*, *stop*, and *subscribe* and *unsubscribe*. A directive argument is the remainder of the query (after the directive). A directive and a directive argument are handled by the system and are never passed to an engine.

3.4 Separating the Path from the Argument

So why do you need to know all of this? As the author of an engine, you presumably care about the argument, i.e. the portion of the request that contains the user input needed by the engine. The tricky part is separating the path from the argument, which can only be done by comparing the request against the match patterns specified in the CMRL document. You need to keep in mind that the *the argument depends on the CMRL document.* This contrasts with web programming, where everything that follows the "?" symbol in a URL is unambigiously considered to be the argument (and is passed, e.g., to a CGI program).

4 What Is Posted to an Engine

In the example described in section 2, the system posts the argument to the engine as the parameter "sys_argument." But the system also posts other parameters to the engine. These other parameters include parts of the request, "processed" versions of parts of the request, and "session variables." If you are not familiar with keywords and session variables in CMRL, then it may be helpful to read *Keywords in CMRL* and *Session Variables in CMRL* before proceeding.

4.1 Parts of the Request

The system posts many of the various parts of the request described in section 3 to the engine. In particular, the system posts the channel to the engine as the parameter "sys_channel," the query as the parameter "sys_query," the designator as the parameter "sys_designator," the path as the parameter "sys_path," and the argument as the parameter "sys_argument."

4.2 "Processed" Parts of the Request

The system "processes" parts of the request (e.g. by "correcting" parts of the request to standard lists of tokens), and the system posts many of the "processed" versions of parts of the request to the engine as well.

4.2.1 Location

The system uses the channel and the designator to determine the URL of the CMRL document. The system posts the URL of the CMRL document to the engine as the parameter "sys_location."

4.2.2 “Corrected” Path

The system “corrects” the path (for mis-spellings, abbreviations, missing levels, etc.) to a list of standard tokens by comparing the request against the match patterns specified in the CMRL document. The system posts the corrected path to the engine as the parameter “sys_corrected_path.” The system posts the corrected path tokens as the parameters “sys_corrected_path[*n*],” where *n* is a zero-based index. (So the first such token is the parameter “sys_corrected_path[0],” the second such token is the parameter “sys_corrected_path[1],” etc.) The system posts the number of such tokens as the parameter “sys_num_corrected_path.”

4.2.3 “Corrected” Argument

The system may partially or wholly “correct” the argument (for mis-spellings, abbreviations, etc.)to a list of standard tokens by comparing the argument against the keywords, if keywords are defined. The system posts the corrected argument to the engine as the parameter “sys_corrected_argument.” The system posts the known (because they are successfully matched to a keyword) corrected tokens as the parameters “sys_corrected_argument_known[*n*]” and the number of such tokens as the parameter “sys_num_corrected_argument_known”; the system posts the unknown (because they are not successfully matched to a keyword) corrected tokens as the parameters “sys_corrected_argument_unknown[*n*]” and the number of such tokens as the parameter “sys_num_corrected_argument_unknown”; and the system posts the (known and unknown) corrected tokens (together in the original order) as the parameters “sys_corrected_argument[*n*]” and the number of such tokens as the parameter “sys_num_corrected_argument.” (Here again *n* is a zero-based index.)

4.3 Session Variables

Session variables are used to store information on a per-user basis. The system posts all session variables to the engine as parameters by name. (So if a session variable “zip_code” is defined, then the system posts it to the engine as the parameter “zip_code.”)

4.4 Summary

This is all summarized in Table 1, which lists the parameters that the system posts to an engine along with a brief description of each.

5 The CMRL Terminating Nodes

An engine returns a CMRL “terminating node,” of which the `<message>` tag is the simplest example. But there are four other CMRL terminating nodes in version 1.0 of CMRL: the `<engine>`, `<rss>`, `<query>`, and `<block>` tags. Let’s consider the five CMRL terminating nodes in turn and then examine how CMRL terminating nodes are evaluated.

Table 1: What is posted to an engine.

Parameter	Description
sys_channel	Channel
sys_query	Query
sys_designator	Designator
sys_path	Path
sys_argument	Argument
sys_location	Location
sys_corrected_path	Corrected path
sys_num_corrected_path	Number of corrected path tokens
sys_corrected_path[*n*]	Corrected path tokens
sys_corrected_argument	Corrected argument
sys_num_corrected_argument_known	Number of known corrected argument tokens
sys_corrected_argument_known[*n*]	Known corrected argument tokens
sys_num_corrected_argument_unknown	Number of unknown corrected argument tokens
sys_corrected_argument_unknown[*n*]	Unknown corrected argument tokens
sys_num_corrected_argument	Number of corrected argument tokens
sys_corrected_argument[*n*]	Corrected argument tokens
Session variables	Session variables

5.1 The <message> Tag

The <message> tag returns a message and is the simplest CMRL terminating node. The <message> tag must contain exactly one <content> tag and may contain zero or one <input> tags. The <content> tag may contain a text string, which is the content of the message. The <input> tag is described in the document *Session Variables in CMRL.*

5.2 The <engine> Tag

The <engine> tag references an engine. The <engine> tag must contain the attribute href, which specifies the URL of the engine.

5.3 The <rss> Tag

The <rss> tag references an RSS feed. The <rss> tag must contain the attribute href, which specifies the URL of the RSS feed. The <rss> tag may contain the attribute “story.” The “story” attribute references a 1-based index to item tags in the feed.

5.4 The <query> Tag

The <query> tag references a query, which the system executes as though it were an incoming request. The <query> tag must contain a text string, which is the query to be executed.

For example, consider the following CMRL fragment under the internet domain name “example”:

```
<match pattern="help">
  <message>
    <content>Text your query to DOTCOM (368266)</content>
  </message>
</match>

<match pattern="assistance">
  <query>example help</query>
</match>
```

With this construction, both the queries “example help” and “example assistance” produce the response “Text your query to DOTCOM (368266).”

5.5 The <block> Tag

The <block> tag allows various “helper” tags to be associated with another terminating node. The <block> tag must contain exactly one terminating node and may contain various other tags, including zero or one <keywords> tags (which define keywords) and zero or more <set> tags (which set session variables) . The <block> tag is described in the documents *Keywords in CMRL* and *Session Variables in CMRL.*

5.6 Terminating Nodes are Recursively Evaluated

It is important to understand that *terminating nodes are recursively evaluated until they resolve to a* <message> *tag.* This means, e.g., that an engine can return an <engine> tag that references another engine that in turn returns another <engine> tag, etc., until ultimately the last engine returns a <message> tag. Similar possibilities apply to the <query> tag. The system internally resolves the <rss> tag to a <message> tag.

6 Summary

A DOTGO engine is a web-based application program that lets you create dynamic mobile services, connecting DOTGO to the internet, databases, and other computer-based services. The next step is to try it out for yourself.