GettingStartedWithFlow

A Gentle Introduction to WhatIsFlow.

We will create a simple number guessing game using Cocoon's Flow Engine

This is based off my experience with learning to write a basic program using the Flowscript, using the Flow and Petstore samples as examples. – TonyColle

What you will need:

- Basic understanding of Cocoon concepts
  *Sitemap
  *Pipelines, etc
- Ability to code a little Javascript
- Build and deploy Cocoon 2.1-dev. See WhereToGet21Dev.
  *You will need to edit the `local.build.properties` file and make sure that the lines `exclude.webapp.scratchpad=true` and `exclude.scratchpad=true` are commented out, because we will be using components from the scratchpad – in particular, the JXTemplateGenerator.

Getting started

Now that you've got Cocoon 2.1 deployed and running, go to where you have Cocoon deployed and create a new subdirectory named `game`. Cocoon's default main sitemap will automatically mount the sitemap in the subdirectory.

Create the following `sitemap.xmap` in the new subdirectory:

```
<?xml version="1.0" encoding="UTF-8"?>
<map:sitemap xmlns:map="http://apache.org/cocoon/sitemap/1.0">

<map:components>
    <map:generators default="file">
        <!-- in this example we use JXTemplateGenerator to insert
             Flow variables in page content -->
        <map:generator label="content,data" logger="sitemap.generator.jxt" name="jxt"
            src="org.apache.cocoon.generation.JXTemplateGenerator"/>
    </map:generators>
    <map:flow-interpreters default="JavaScript"/>
    <map:transformers default="xslt"/>
    <map:serializers default="html"/>
    <map:matchers default="wildcard"/>
    <map:selectors default="browser">
      <map:selector name="exception"
        src="org.apache.cocoon.selection.XPathExceptionSelector">
        <exception name="invalid-continuation"
           class="org.apache.cocoon.components.flow.InvalidContinuationException"/>
        <exception class="java.lang.Throwable" unroll="true"/>
      </map:selector>
    </map:selectors>
    <map:actions/>
    <map:pipes default="caching"/>
</map:components>

<map:views/>
<map:resources/>
<map:action-sets/>

<map:flow language="JavaScript">
    <!-- Flow will use the javascript functions defined in game.js -->
    <map:script src="flow/game.js"/>
</map:flow>

<map:pipelines>
 <map:component-configurations>
    <global-variables/>
</map:component-configurations>

<map:pipeline>
    <!-- no filename: call main() in game.js -->
    <map:match pattern="">
        <map:call function="main"/>
    </map:match>

    <!-- use JXtemplate to generate page content -->
    <map:match pattern="*.jxt">
        <map:generate type="jxt" src="documents/{1}.jxt"/>
        <map:serialize type="xhtml"/>
    </map:match>

    <!-- .kont URLs are generated by the Flow system for continuations -->
    <map:match pattern="*.kont">
        <map:call continuation="{1}"/>
    </map:match>

    <!-- handle invalid continuations -->

    <!-- this style of handling invalidContinuation is now deprecated: -->
    <!-- this URI will never be called automatically anymore. -->
    <!-- see handle-errors below -->
    <map:match pattern="invalidContinuation">
        <map:generate src="documents/invalidContinuation.xml"/>
        <map:serialize type="xml"/>
    </map:match>

    <!-- the new non-hardcoded way of handling invalidContinuation -->
    <map:handle-errors>
        <map:select type="exception">
            <map:when test="invalid-continuation">
                <map:generate src="documents/invalidContinuation.html"/>
                <map:serialize type="xhtml"/>
            </map:when>
        </map:select>
    </map:handle-errors>

</map:pipeline>

</map:pipelines>
</map:sitemap>
```

Inside the new subdirectory, create two more directories, `documents/` and `flow/`.

Inside `documents/`, you will store the "views" – pages to send to the player. Create the file `guess.jxt`, which will be the page the player will enter their guess:

```
<?xml version="1.0"?>
<html xmlns:jx="http://apache.org/cocoon/templates/jx/1.0">
<head>
    <title>cocoon flow number guessing game</title>
</head>
<body>
    <h1>Guess the Number Between 1 and 10</h1>
    <h2>${hint}</h2>
    <h3>You've guessed ${guesses} times.</h3>

    <form method="post" action="${continuation.id}.kont">
        <input type="text" name="guess"/>
        <input type="submit"/>
    </form>
</body>
</html>
```

You'll also need a page to display when the person chooses the correct number. Name it `success.jxt` (Again in `documents/`):

```
<?xml version="1.0"?>

<html xmlns:jx="http://apache.org/cocoon/templates/jx/1.0">
<head>
    <title>cocoon flow number guessing game</title>
</head>
<body>
    <h1>Success!</h1>

    <h2>The number was: ${random}</h2>
    <h3>It took you ${guesses} tries.</h3>

    <p><a href="./">Play again</a></p>
</body>
</html>
```

You may notice some strange codes inside the files – namely things like {{ ${random} }} and {{ ${guesses} }}. They look like variables, and they will be replaced with values when the pages are sent to the client. This is where the JXTemplateGenerator comes in.

Initially there was some confusion on my part regarding the syntax for ${continuation.id}. In the Petstore examples, I saw something like #{$continuation/id}. Can anyone explain that syntax to me? – TonyCollen

_ Expressions inside #{} are XPath expressions. Those inside ${} are JSTL expressions. These are implemented with Apache JXPath and Apache Jexl, respectively. As a result, the same Java bean, JavaScript, DOM, or JDOM objects may be accessed using either expression language. Typically you would use one or the other within a single template - or perhaps use JSTL for beans and XPath for DOM nodes in the same template. – Chris Oliver_

Interesting. Is there any real difference between always using one or the other? If one is more robust, why have both? This duplication seems like it could cause some confusion. Perhaps some docs pertaining using one vs. the other would be good to put here. 🙂 *– TonyCollen*

JXPath and Jexl are both robust. If your objects represent XML data, or if you know XPath but are not a Java or JavaScript programmer (the JSTL expression language syntax is similar to JavaScript) then probably using XPath makes sense. Otherwise, to access JavaScript objects or Java beans from your Flowscript just use Jexl. – Chris Oliver

---

Inside `flow/`, you will store the code that actually controls how this application runs. In the MVC pattern, the Flow is the "Controller", and it is very powerful.

Create the following file named `game.js`:

```
function main() {

  var random =  Math.round( Math.random() * 9 ) + 1;

  var hint = "No hint for you!"
  var guesses = 0;

  while (true) {

    sendPageAndWait("guess.jxt", {"random" : random, "hint" : hint, "guesses" : guesses});

    var guess = parseInt( cocoon.request.get("guess") );

    guesses++;

    if (guess) {
     if (guess > random) {
        hint = "Nope, lower!"
      } else if (guess < random) {
          hint = "Nope, higher!"
      } else {
          break;
      }
    }
  }

  sendPage("success.jxt", {"random" : random, "guess" : guess, "guesses" : guesses} );
}
```

Alright, now let's follow the execution of this Flow and pipeline:
The player accesses the URL `http://host/cocoon/game/` and the `<map:match pattern="">` matches, and starts the pipeline.

The function `main()` which is referenced in `flow/game.js` is called, and a new Continuation object is created. Without getting into too much detail, the state of the Javascript code is saved, and can be recalled any number of times.

TODO: Explain the concept of continuations in further detail. – TonyCollen

We now enter the code in `game.js`:

- A random number between 1 and 10 is chosen.
- Variables containing a hint for the player and the player's current number of guesses are initialized.

The Flow now enters the `while(true)` loop which basically keeps the game going until the player guesses the correct number.

We now get to the following line, where things start to get interesting:

```
sendPageAndWait("guess.jxt", { "random" : random, "hint" : hint, "guesses" : guesses} );
```

The Flow layer sends the contents of the URI "guess.jxt" which is matched in the sitemap (see above). We also pass an inline Javascript object, containing three key/value pairs, one named "random" which contains the value of the variable `random` as initialized above, and so on for `hint` and `guesses`. The keys are substituted later down the line, when the JXTemplateGenerator comes into play.

We could also do the following:

```
sendPageAndWait("guess.jxt", { "foo" : random } );
```

In this case, the value of `random` would be able to be substituted in our JXTemplate, but under the name "foo"" instead – we'd just have to make sure we have the correct keyname in our template.

The Flow Layer also does another interesting thing: **it halts the execution of the Javascript!** Through the magic of continuations, the Flow Layer is able to resume execution of the script at the exact line in which it left off. This creates some very powerful situations with respect to web programming, and forces the reader to think very differently about how web applications are designed.

Picking back up in the script execution, the client is sent through the pipeline matching "guess.jxt". Referring back to the sitemap, we match `*.jxt`, and run the file through the JXTemplateGenerator, which substitutes the keynames for the values sent from the `sendPageAndWait()` function.

One thing to note is in the form which is sent back to Cocoon when the player submits the guess:

```
<form method="post" action="${continuation.id}.kont">
```

Here, {{ ${continuation.id} }} is resolved to a unique identifier which points to the current continuation. One can think of this somewhat of a session ID.

When the player submits the form, it is submitted to a unique URL which contains the continuation ID, plus ".kont", which we end up matching in the sitemap:

```
<map:match pattern="*.kont">
    <map:call continuation="{1}"/>
</map:match>
```

When Cocoon sees a URL like this, it attempts to restart the continuation with the specified ID, and we re-enter the Javascript code where we left off previously.

TODO: Explain map:match pattern="invalidContinuation" – TonyCollen

We are now back in the Javascript at the line after `sendPageAndWait()`. We create a new variable (an int), which we get from the POST request that was sent by the form. Notice in the form we had `<input type="text" name="guess"/>` and in the Javascript we get the request parameter by using `cocoon.request.get("guess");`

TODO: Explain all the objects available in the Flow layer. – TonyCollen

Question: from playing with this example, it looks as if the value of `random` remains the same throughout each user's session. How do you create per-user /session instance variables/objects – BillHumphries

Answer: I'm not 100% sure what you mean; if you mean per-user variables, it's more of a per-continuation variable, that is, the variable is the same throughout the same continuation. With regards to creating "real" session variable, I could be wrong but I believe any variable defined in the global scope of the Flow script will be in the session. Like I said, I'm not certain, so I'd appreciate anyone who knows more than I do to clear this up. – TonyCollen

_ Just call `cocoon.createSession()` in your script. That will cause global variables to be shared between different top-level JavaScript functions called from the sitemap. --Chris Oliver_

Now we increment the player's guess count, and we test to see if they guessed the correct number. If the guess was too high, we set the hint variable telling them to guess lower, and we fall through the bottom of the `while` loop, and we send the guess form back to the player.

If the guess was too low, we tell them to guess higher, and we fall through the loop as well, sending the player the form again.

If the guess was correct, we `break` out of the main loop and send the player to a different view, this time to "`success.jxt`", and we give the template not only their number and the random number (pointless, yes, because they were the same), but also the number of guesses to tell the player how good or bad at guessing numbers they are.

The main point of interest in the Flow script at this point is the use of `sendPage()` instead of `sendPageAndWait()`. `sendPage()` works exactly the same, except, yes, you guessed it, we don't halt execution of code, and keep processing.

At this point, there's no more code left and the game is over, and the Flow stops.

Another thing to note is the `<map:handle-errors>` tag in the sitemap. Previously, when a continuation which did not exist was called, the Flow layer would automatically redirect to the URI "invalidContinuation". Now, the Flow layer throws an InvalidContinuationException, and you can now handle it as described in the handle-errors tag.

And that's it! You have now just made your very first application using the Flow layer.

TODO: Write an Intermediate Guide to the Flow – TonyCollen

Correction to above as of 9/06/03

Due to changes with FOM, above sitemap.xmap will not work on a cocoon-2.1 release. To make it work, change the "JavaScript" to "javascript" note that its all lowercase now. Furthermore, when calling the functions within game.js, one has to call the functions this way cocoon.sendPageAndWait(....). – enio

Correction to above as of 6/14/04

Since Cocoon 2.1.4 the following line must be removed from sitemap.xmap:

```
  <map:flow-interpreters default="JavaScript"/>
```

See also

- WhatIsFlow