GumpRunDocumentation

GumpRunDocumentation

Gump Run Documentation is documentation about a Gump run, not about Gump.

How things work today

Gump currently supports two forms of documentation (plain text and xdocs, forApache Forrest). The reason for this is that although Forrest generates impresive sites, it is time consuming and not ubiquitously available. Folks have the choice of simple/cheap or pretty.

Overview

Currently 'documentation' is a task performed at the end of a run, so it has the all context (states, outcomes) available. This is important for 'statistics' and 'xref' pages, which display pages that compare/sort all modules/projects so needs all to be complete, and have complete information.

Note: That said, this could be split into two parts – with pages for single modules/projects being creates as available (much sooner) and only the cross entity pages created last.

Implementation

Documented is achieved using a few classes:

- Resolver – that resolves a model object (e.g. a project) to a URL or file.
- Documenter – that works with it's resolver to generate those files.

Resolver

This component is important in part because other aspects (e.g. RSS|Atom feeds, Notification e-mails) need to be able to refer to some entity (a failed work item, a project page) without knowing which documenter was used.

Also, calculating the URL/path can be quite complicated (especially for Forrest with content and content/xdocs, and with safe naming) and we need to centralize the logic to avoid duplication/error.

Complicating Factors

Forrest content goes into content (eg. images), or content/xdocs (e.g. xdoc pages). The resolver needs to know this for when it generates files, but not for when it generates URLs (since they are merged here).

Documenter

This component is the meat of the documentation process. A GumpRun (with complete information) is traversed, and information is generated for all entities in the GumpSet (a list or dependency stack).

Two documentation classes exist as sub-classes of gump.documentation.Documenter:

- gump.documentation.ForrestDocumenter
- gump.documentation.TextDocumenter

Complicating Factors

Given that forrest is used to generate the site, additional content (server.xml, *.rss|*.atom feed) and other non-documentation artifacts, has to be placed into {forrest-work}/src/documentation/content after the template is syncronized in, and before the forrest run occurs. As such the Documenter 'interface' has a prepare() method that allows the sub-class to implement prepareRun() or not. This is called prior to those syndication/results steps, and before documentation.

```
    #
    # Call a method called 'prepareRun(run)', if needed
    #
    def prepare(self,run):
        if not hasattr(self,'prepareRun'): return
        if not callable(self.prepareRun):  return
        log.info('Prepare to document run using [' + `self` + ']')
        self.prepareRun(run)
```

ForrestDocumenter

This module is huge, but that is only because it has a lot of repetative (template-ish) code.

The overall algorythm used for documentation is:

```
Document the workspace (see block below)
For each module in workspace:
 If module not in gump set: continue
 Document the module (see block below)
 For each project in module:
   If project not in gump set: continue
   Document the project(see block below)
```

Basically `blocks` looks like this:

- Select a file for an entity (the resolver determines the name/where)
- Generate xdoc sections (e.g. details, or dependencies or)
- Generate details by adding paragraphs or lists/list items as needed.
- Iterate through information (e.g, annotations) adding tables/rows/data.
- Serialize the xdocs (DOM-like tree) to that file.
- Throw away all memory used by the tree.

Note: Some re-used blocks are:

- Generate a section (with table with rows/data) for all Annotations
- Generate a section (with table with rows/data) for all Files held by FileHolder
- Generate a section (with list/items) for Statistics

There are some helper methods for displaying a state icon for an entity.

xdocs

This is a home grown, DOM-like, approach. A basic `XDocPiece` base class exists that is sub-classes for Document or Section or Paragraph, etc. These classes have createX methods that allow creation of sub-elements into it's own list of children. Effectively, this encapsulates the xdoc rules in code, since not child can be created without a method on the parent.

Helper methods exist make common combinations, e.g. to create a Table and add titles from a list of strings, e.g. to add a line to a table with title/value, two datum.

When a `XDocDocument` is serialized it recursively outputs the tag (e.g. <P>), outputs it's children (including text nodes), then outputs it's closed tag (e.g. </P).

How we could take them forward