push

Release v0.0.1

May 09, 2017

Contents

push is a shell written in PHP. It bears a few similarities with Boris but aims at being easy to embed in broader libraries/projects.

Contents:

CHAPTER 1

Design

Components

The shell is composed of two main pieces:

- A manager, implemented by the `\\fpoirotte\\push\\Manager` class
- Workers, implemented by the `\\fpoirotte\\push\\Worker` class

The manager handles interactions with the user (displaying the prompt, reading commands, handling signals, displaying results, etc.) and feeds commands and signals to the workers.

A worker process is spawned by the manager upon startup. A new worker is spawned whenever a command needs to be executed, by forking off the previous worker:

- If the command raises a fatal error or throws an exception, the new worker dies. The old worker will be used to fork a new one on the next command. A notice is displayed to the user by the manager to indicate the command failed to execute properly.
- If the command completes without any fatal error, the old worker is killed and the new worker will be used in its place in future commands. The manager then displays the command's result on the terminal.

This is done so to ensure that any side-effect the command may have had (like setting a variable, creating a resource, etc.) is retained in the future.

User interaction

The `\\fpoirotte\\push\\LineReader` class provides the necessary means to prompt the user for commands. This class also sets up a signal handler so that any signal is properly passed from the manager to workers if necessary.

Inter-process communication

The manager and the worker processes communicate together asynchronously using two distinct channels.

Messages/events

A custom protocol implemented by the `\\fpoirotte\\push\\Protocol` class (the base class for both the manager and the worker processes) is used by the manager and workers to notify the other end when various events occur.

In this protocol, each peer can send an operation code recognized by the other peer (represented on the wire as a single byte), followed by optional data, which is passed on the wire as the payload's size (expressed in bytes and represented as an unsigned 16 bit integer encoded in big endian order), followed by the actual payload represented as a string.

An empty payload is represented as a zero-length payload. This means that it is not currently possible to distinguish between the absence of a payload and a payload consisting of an empty string. However, this is not an issue in practice as such a distinction is not actually needed.

This protocol is used to represent various events, such as a signal being received by the manager that needs some specific handling on the worker's part. For example, if the manager wants to send a `SIGINT` signal to the worker, it sends the following byte sequence on the wire:

- `\\x01` : operation code, as defined by the constants in `\\fpoirotte\\push\\Manager`, minus the `OP_` prefix; in this case, `\\x01` is the code for the "SIGNAL" operation
- `\\x01` : payload size; in the case of a signal, the payload is encoded as an ASCII string representing the signal's number (ie. "2")
- `\\x32` : the signal's number (2 for SIGINT) as a string

The receiving end then calls the appropriate handler for the operation, by deriving its name from the operation's name (ie. `handleSIGNAL`). This handler is responsible for taking all necessary actions.

When the operation has been handled, the handler may choose to send back its own message, but this is not mandatory (this only makes sense when the remote peer is expecting some sort of results or acknowledgement).

Output and errors

The results of a command's execution are passed from the workers to the manager using the regular streams `STDOUT` and `STDERR`. This is done so because some PHP operations automatically write to those streams (eg. the `echo` / `throw` statements).

Internally, the workers' `STDIN`, `STDOUT` & `STDERR` streams are replaced with alternate streams (which are connected to the manager), so that any data sent to those streams can easily be intercepted and processed by the manager.

CHAPTER 2

Messages

This section describes the various messages that may be sent by each peer.

Messages emitted by the manager

`COMMAND`

This message is sent to the worker whenever a command (a line of statements given by the user) needs to be processed.

This message's payload consists of the actual statement(s) to process.

`SIGNAL`

This message is sent to the worker whenever the manager receives a signal and wishes to pass it on to the worker.

This message's payload consists of the signal number, represented as a string. So for example, `SIGTERM` is represented by the following byte sequence: `\\x31\\x35` (ie. "15").

Messages emitted by workers

`READY`

This message is sent by the very first worker after it has been spawned and indicates that it is fully initialized and ready to process incoming commands.

This message's payload consists of the worker's full path and line number, in the form `full/path/to/Worker.php(line)`.

START

This message is sent by a worker before it starts executing a command.

This message's payload consists of the worker's PID (Process IDentifier).

END

This message is sent by a worker after the command is was processing finished executing.

This message has no associated payload.

Badges: