browser_cli

A command line interface for the web browser.

Nathan Karasch

28 Sept 2016

Table of Contents

Introduction

About a year ago I began using Linux. After spending the previous six years or so with a Mac, the switch wasn't easy. Going from an operating system that was so polished there was rarely a need to troubleshoot to an operating system where… well… when I started it up, I wondered, "Wait, why can't I connect to wifi?" After a good amount of digging, I found out I had to install the driver myself! As well as troubleshoot power management, tweak configurations, install dependencies, compile things from source, and a slew of other things I never had to think about. What did I get myself into?

After a while, however, Linux starts to grow on you. The pain goes away, and in its place is a feeling of accomplishment and a much deeper understanding of how a computer works. It no longer takes an hour to install something or troubleshoot issues. You start to feel like a "power user."

And at the center of the transformation is the bash command line.

I began this project as a cute novelty item. "Wouldn't it be great if I could interact with a website the same way I interact with the command line?" As it grew more and more complex, however, I realized that 1) it's really hard to replicate the functionality of a bash command line, because there's SO MUCH functionality, and 2) it would be pointless to ONLY replicate the command line. If I did that, what would be the point? There had to be something that a command line in the web browser could do that set it apart from a typical command line interface.

If I could find that unique something, I could create a new type of user experience.

Overview

This portfolio explored creating a command line interface that runs in the browser, combining two types of interfaces into a unique user experience. The application was written in Dart, a language developed by Google that transpiles to Javascript to run in the browser.

I really wanted to reach a stable version that could be published for use by other Dart developers via Dart's package manager, so I wrote and documented the public API with that in mind. Setup for consumers is trivial, so all their time can be spent writing custom processes for the application to use in their own websites. This separation of the process from the main application makes it easy for users to get the functionality of the command line interface without having to understand what's going on behind the scenes. The focus on the public API was a factor in the project's complexity as I continually evaluated how my end user would want to use the API.

The main complexity factor was, of course, trying to replicate a bash command line with process management. For it to be desirable or useful to users at all, it needed to provide core functionality that users expect from a command line: accept input, run multiple processes synchronously and asynchronously, display output, etc. To keep to the time frame, the project was limited to these key pieces of functionality:

- Interactive IO
- Process management
- Environment variable storage and recall
- Command completion
- Input history
- A set of standard library processes:
    - **authentication** - loads information from the document cookies and (someday) will authenticate the user's credentials.
    - **echo** - prints the supplied input back to the shell.
    - **export** - creates a variable that persists across browser sessions via cookies.
    - **help** - offers help information for the command line interface.
    - **jobs** - lists all processes that are currently running.
    - **printenv** - prints all environment variables.
    - **testinput** - demonstrates how to get user input from within a process and how to call other commands from within a process.
    - **unset** - erases an environment variable.

Interaction with Materials

Choosing Dart over Javascript

The materials covered in Com S 319 so far have dealt heavily with Javascript. That's a good thing -- Javascript is ubiquitous in the world of web development. I chose to develop in Dart instead of Javascript for several reasons:

1. Dart transpiles to Javascript when written for use in the web browser, so it is functionally equivalent.
2. I think the design paradigms Dart has adopted make application development faster, easier to scale, and easier to catch errors early on.
3. I have used Dart for the past year and a half, meaning I can make more complex applications using it than if I were to use Javascript.
4. I wanted to learn the ins and outs of creating and publishing a Dart package to pub.dartlang.org.

Many of the handy features from jQuery and other javascript frameworks have been built in to Dart. Take, for example, the following jQuery methods and their equivalents in Dart:

Javascript:

```
var str = $("a:hover").text();
$("#container").html(str);
```

Dart:

```
var str = querySelector("a:hover").text;
querySelector("#container").innerHtml = str;
```

In class we covered the use of functions as objects in Javascript. Below is an example from the project of these in action in Dart:

From *command_line_interface.dart*:

```
_addBindings() {
 // Putting a function in a map. The function is later called by the KeyBindingManager
 // whenever a keydown event in the window matches the KeyGesture used as a map key.
 _keyBindingManager.bindings[new KeyGesture(KeyCode.ENTER)] = _commitInput;
```

```
   // ...
   // Defining a function on the fly to put into the map
   _keyBindingManager.bindings[new KeyGesture(KeyCode.KEY_P, ctrlKey: true)] =
       (KeyboardEvent event) {
     event.stopImmediatePropagation();
     event.preventDefault();
     return true;
   };
   // ...
  }

  bool _commitInput(KeyboardEvent event) {
   // ...
  }
```

Below is an example of some basic DOM manipulation from *command_line_interface.dart*:

```
  var inputContainer = new DivElement()..className = inputClass;
  var userInput = new SpanElement()
   ..id = CLI.STANDARD_INPUT
   ..className = inputClass
   ..contentEditable = 'true';
  var prompt = new SpanElement()
   ..id = CLI.PROMPT
   ..text = modifiedPromptText;
  inputContainer.children..add(prompt)..add(userInput);
  shell.children.add(inputContainer);
  userInput.focus();
```

Project Structure

The project has been broken up into the following libraries:

command_line_interface

This library provides the main interface between the user and the application. It receives input from the user, parses input, displays output to the user, and deals with other elements of user interaction.

environment_variables

This library provides functionality for dealing with environment variables available to the user and across processes in the application.

process
This library provides all the concrete Process and ProcessFactory classes that come standard with browser_cli. It also provides a good resource to look at when designing your own custom Process.

process_manager
This library contains the ProcessManager class, as well as the abstract Process and ProcessFactory classes. It handles things like starting and stopping processes, piping input and output to and from the CommandLineInterface, handling asynchronous and synchronous completion, differentiating between background and foreground processes, and more.

utils
This library provides utility functions and variables for browser_cli that can be easily shared across other libraries.

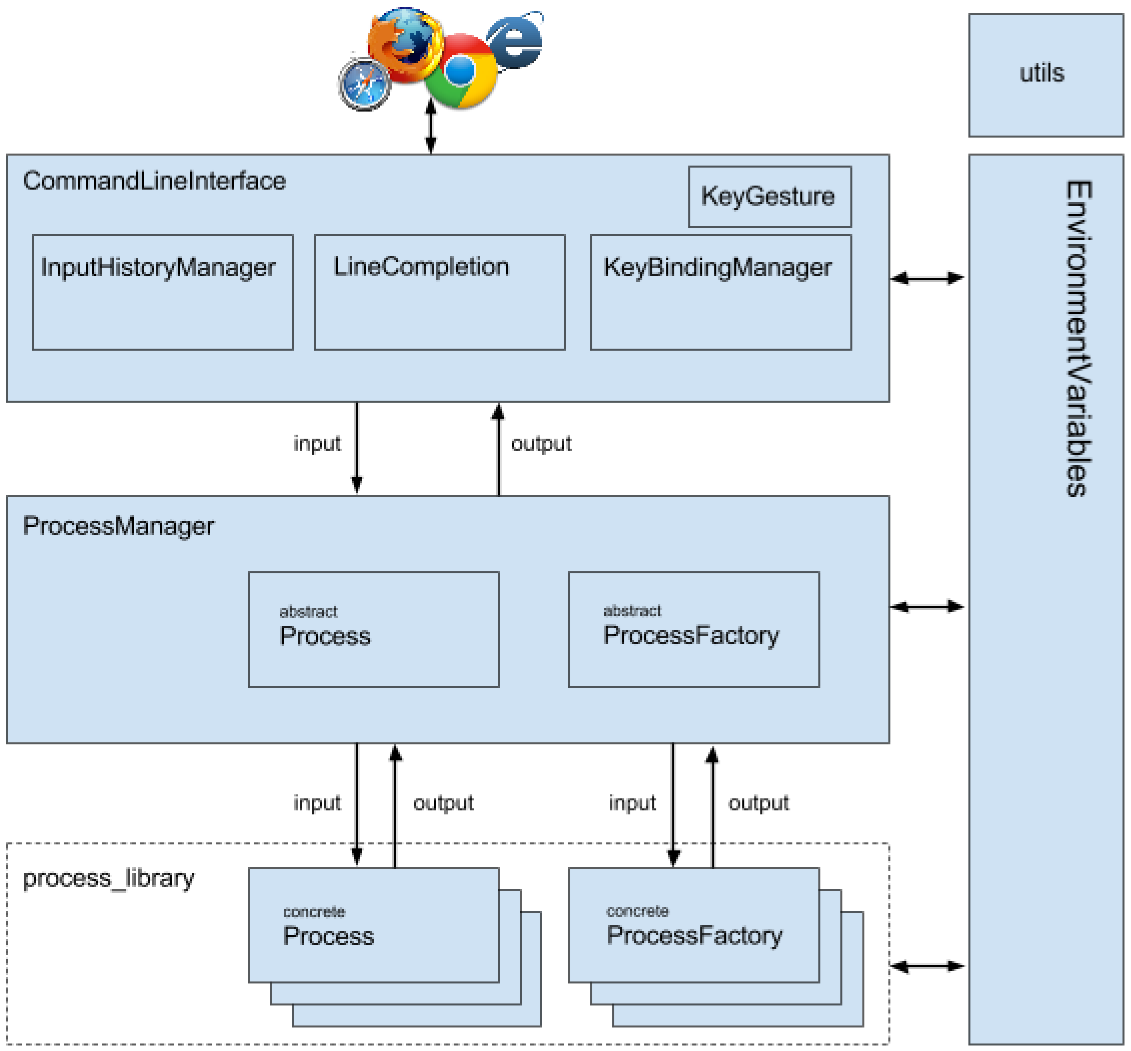


Figure 1: A block diagram of browser_cli.

Complex Issues

Designing for Extension and Consumption

Designing the command line interface for consumption as a public API influenced a lot of the decisions I made during this project. Each process (together with its factory) exists as a separate library, and all the "standard library" processes get bundled together in the process library. The beauty of this is that users can create custom processes or import processes made by other developers and build their own custom command line interface web app, using the "composition over inheritance" principle of object oriented programming.

Let's take a look at how easy it is to get started created your own command line interface (assuming you've already gotten your Dart environment setup):

1. Depend on it

Add this to your package's pubspec.yaml file:

```
dependencies:
  browser_cli:
```

2. Install it

You can install packages from the command line:

```
$ pub get
```

3. Add code to your html page

The following goes inside the `<head>`:

```
<script defer src="main.dart" type="application/dart"></script>

<script defer src="packages/browser/dart.js"></script>
```

In the `<body>`, where you want the shell located, add the following:

```
<div id="cli-shell"></div>
```

4. Create or import processes

Any process that you want to run in the CLI needs to extend the `Process` class, and it needs to also have a factory that extends the `ProcessFactory` class.

ProcessFactory Example:

A ProcessFactory should follow the model below, providing a COMMAND, USAGE, SHORT_DESCRIPTION, and LONG_DESCRIPTION.

```
class EchoProcessFactory extends ProcessFactory {
  static final String COMMAND = 'echo';
  static final String USAGE = 'USAGE: echo <string>';
  static final String SHORT_DESCRIPTION =
      'Prints the supplied input back to the shell.';
  static final String LONG_DESCRIPTION =
      'Prints the supplied input back to the shell';

  EchoProcessFactory()
      : super(COMMAND, USAGE, SHORT_DESCRIPTION, LONG_DESCRIPTION);

  EchoProcess createProcess(int id, List args) =>
      new EchoProcess(id, COMMAND, args, this);
}
```

Process Example:

The only two required API for a Process are the constructor and the `start()` method. Below is a very basic Process, but they can be much more complex. Check out other Processes in the standard library for examples of different ways to parse the arguments, how to request user input within a running process, how to start other processes from within a process, and more!

```
class EchoProcess extends Process {
  EchoProcess(int id, String command, List args, ProcessFactory factory)
```

```
      : super(id, command, args, factory);

  Future start() async {
    if (args.isNotEmpty) {
      await output(new DivElement()..text = args.join(' '));
    } else {
      await output(new DivElement()..innerHtml = nonBreakingLineSpace);
    }
  }
}
```

5. Create a main.dart and register desired process factories

Make sure to register all the standard library process factories, as well as any custom process factories you may have.

```
import 'package:browser_cli/command_line_interface.dart';
import 'package:browser_cli/process_library.dart';
import 'package:my_cool_project/process_library.dart';

CommandLineInterface interface;

void main() {
  interface = new CommandLineInterface();
  _registerProcesses();
}

_registerProcesses() {
  interface.processManager.registerProcessFactories([
    new EchoProcessFactory(),
    new ExportProcessFactory(),
    new HelpProcessFactory(),
    // ...
  ]);
}
```

Process Management and Input/Output Design

The design of the ProcessManager was one of the most complex parts of the project. It was complex because it needed the following functionality:

- a way to track multiple processes that could be running at any given moment.
- the ability to handle processes that could be either asynchronous or synchronous.
- the ability to handle different exit codes appropriately, as well as being able to kill a running process.
- a means to pass output from a process to the command line interface while avoiding high coupling.
- a means for processes to request input from the user, and to get that input to the correct process.

What follows is the source code for the ProcessManager class, which is split up into segments with narrative in between talking about the different parts:

```
/// Manages the starting, stopping, and manipulation of all processes in the
/// shell.
class ProcessManager {
 static ProcessManager _processManagerSingleton;

 // Runs the first time the singleton is constructed.
 ProcessManager._internal(int randomizerSeed) {
   _randSeed = randomizerSeed ?? new Random().nextInt(utils.MAX_INT);
   _rand = new Random(_randSeed);
   _processManagerSingleton = this;
 }

 /// Will always return the same singleton [CommandLineInterface] object.
 /// The randomizerSeed argument only gets processed the first time the
 /// singleton is created.
 factory ProcessManager({int randomizerSeed}) =>
     (_processManagerSingleton == null)
         ? new ProcessManager._internal(randomizerSeed)
         : _processManagerSingleton;
```

I didn’t start out with the ProcessManager as a singleton. It was originally an instance variable of the command line interface, as were the environment variables. I decided it would simplify the design to make the EnvironmentVariables, ProcessManager, and CommandLineInterface all singletons so that processes (each as separate libraries) could access any of their instance variables if need be by calling the singletons’ constructors.

```
/// The randomizer seed for all pseudo-random operations.
int get randSeed => _randSeed;
Random _rand;
int _randSeed;

/// A [Map] of all processes that are currently running, with the key being
/// the process id.
Map<int, Process> get processes => _processes;
Map<int, Process> _processes = new Map();
Map<int, StreamSubscription> _processExitCodeStreamSubscriptions = new Map();

/// The most recent [Process] that was started.
Process get mostRecent => _mostRecent;
Process _mostRecent;

/// The [Stream] of [DivElement] objects that should be output to the shell.
Stream<DivElement> get onOutput => _outputStreamController.stream;
Map<int, StreamSubscription> _outputSubscriptions = new Map();
StreamController<DivElement> _outputStreamController = new StreamController();

/// Indicates to the command line interface to get input from the user. If
/// the value is an integer, the input is being requested by the process with
/// that id. If the value is 'null', it gets handled normally by the command
/// line interface.
Stream<int> get onTriggerInput => _triggerInputStreamController.stream;
StreamController<int> _triggerInputStreamController = new StreamController();
Map<int, StreamSubscription> _inputRequestStreamSubscriptions = new Map();
Queue<int> _inputRequestStack = new Queue();

/// A [List] of all commands that have been registered with the
/// [ProcessManager].
List<String> get registeredCommands => _registeredProcessFactories.keys;

/// A [Map] of all [ProcessFactory] objects that have been registered with
/// the [ProcessManager]. The keys in the map are the command names.
Map<String, ProcessFactory> get registeredProcessFactories =>
   _registeredProcessFactories;
Map<String, ProcessFactory> _registeredProcessFactories = new Map();
```

The way of tracking multiple processes occurred through the **`processes`** Map. The randomly generated id of each process would be its key, and the Process object would be the value. This led to a series of other Maps created to track other information on the processes, such as StreamSubscriptions for output, input requests, and exit codes.

Streams are a useful pattern for broadcasting data for other classes to consume if they choose to “listen” in. In this way, information can be passed upward without having to depend on a class further up the chain, thus keeping the coupling low. For example, in the ProcessManager, when it wants to send output to the CommandLineInterface, it simply adds an object to the **`onOutput`** Stream via the corresponding StreamController. The CommandLineInterface then can listen to the Stream and, upon items being entered, can act on them via callback functions. This is how output, requests for input, and such are bubbled upwards.

```
/// Starts a process in the shell.
bool startProcess(String command, {List args}) {
 try {
   var id = _generateId();
   var arguments = args ?? [];
   var process =
       _registeredProcessFactories[command]?.createProcess(id, arguments);
   if (process == null) {
     var supplementaryInput = utils.supplementaryCommandMappings[command];
     if (supplementaryInput == null) {
       throw new Exception('$command: command not found');
     }
     var parsedSuppInput =
         new utils.ParsedInput.fromString(supplementaryInput);
     process = _registeredProcessFactories[parsedSuppInput.command]
         ?.createProcess(id, parsedSuppInput.args);
   }
   processes[id] = process;
   _setupProcessListeners(process);
   _mostRecent = process;
   process.start();
   if (process.factory.autoExit) {
     process.exit(0);
   }
   return true;
 } catch (exception) {
   _outputStreamController.add(new DivElement()
     ..text = exception.toString()
     ..className = utils.CLI.STDERR);
   _triggerInput(null);
   return false;
 }
}

/// Kills a running process
bool killProcess(int processId) {
 if (processes.keys.contains(processId) && processes[processId] != null) {
   processes[processId].kill().then((_) {
     _cleanupFinishedProcess(processId);
     _triggerInput(null);
   });
```

```
    return true;
  } else {
    return false;
  }
}
```

The start() and kill() methods provide a means to start and stop processes at will. The start() method searches through its **`_registeredProcessFactories`** and **`supplementaryCommandMappings`** (aka: aliases) to see if the command can be started. If not, it throws an error that gets output to the shell.

```
/// Sends input `str` to the process indicated by `processId`. Returns `true`
/// if the id refers to a valid process, otherwise returns `false`.
bool input(int processId, String str) {
  if (processes.containsKey(processId)) {
    _inputRequestStack
        .remove(_inputRequestStack.lastWhere((pId) => pId == processId));
    processes[processId].input(str);
    return true;
  } else {
    return false;
  }
}
```

When a process wants input from the user, it adds its id to its **`requestForStdInStream`**, which the ProcessManager is listening to. The Processmanager then adds to its own **`onTriggerInput`** stream, which the CommandLineInterface is listening to. The CommandLineInterface displays a new prompt, and whatever the user enters is sent back down the chain through the input() methods in the ProcessManager and in the Process itself.

```
/// Used to register every type of process that can be started in the shell.
registerProcessFactories(List<ProcessFactory> factories) {
  factories.forEach((factory) {
    _registeredProcessFactories[factory.command] = factory;
  });
}

_handleProcessOutput(int id, DivElement output) {
  if (processes[id].inForeground) {
    _outputStreamController.add(output);
  }
}
```

```
void _triggerInput(int processId) {
  if (_inputRequestStack.isEmpty) {
    _triggerInputStreamController.add(processId);
  }
  if (processId != null) {
    _inputRequestStack.addLast(processId);
  }
}

int _generateId() {
  var id = _rand.nextInt(utils.MAX_INT);
  var attempts = 0;
  while (processes.keys.contains(id)) {
    id = _rand.nextInt(utils.MAX_INT);
    if (attempts++ > 5000) {
      throw new Exception(
          'Too many attempts trying to create unique process id');
    }
  }
  return id;
}
```

The last few private methods below are used for setting up and tearing down listeners. If StreamSubscriptions aren't cancelled, the Processes they belong to won't be garbage collected, leading to memory leaks and bad performance.

```
void _setupProcessListeners(Process process) {
  var id = process.id;
  _outputSubscriptions[id] = process.outputStream.listen((output) {
    _handleProcessOutput(id, output);
  });
  _inputRequestStreamSubscriptions[id] =
      process.requestForStdInStream.listen((_) {
    _triggerInput(id);
  });
  _processExitCodeStreamSubscriptions[id] =
      process.exitCodeStream.listen((code) {
    if (code != 0) {
      _handleProcessOutput(
          id,
          new DivElement()
            ..text = "exit($code)"
            ..className = utils.CLI.STDERR);
    }
    _cleanupFinishedProcess(id);

    // Ensures the error code exit message is displayed
```

```
      new Future.delayed(Duration.ZERO).then((_) {
        _triggerInput(null);
      });
    });
  }

  void _cleanupFinishedProcess(int id) {
    _processes.remove(id);
    _cleanupSubscription(_outputSubscriptions, id);
    _cleanupSubscription(_inputRequestStreamSubscriptions, id);
    _cleanupSubscription(_processExitCodeStreamSubscriptions, id);
  }

  void _cleanupSubscription(Map<int, StreamSubscription> map, int id) {
    map[id].cancel();
    map[id] = null;
    map.remove(id);
  }
}
```

The CommandLineInterface has a lot of code that integrates heavily with the ProcessManager to accomplish everything, but for the sake of brevity I only went over the ProcessManager in detail.

Working at Higher Levels of Bloom’s Taxonomy

Creation

The Creation aspect of Bloom’s Taxonomy was met in this project on several levels. I created a web application that can run on its own, but more than that I created an API that can be imported and leveraged to provide a unique user experience in the browser. I learned some of the finer details of Semantic Versioning while creating a piece of software. I also learned the deployment process for Dart packages, and how to design a piece of software with public API as a key feature. The API was designed and developed in such a way to make it easy for people to use in their own projects.

Analysis

The Analysis aspect was met in the careful selection of design patterns. The Singleton pattern was used in several places to good effect. The usage of Streams was carefully chosen to avoid high coupling. In addition, several of the classes were appraised and refactored multiple times to achieve high cohesion among elements of the code. Things that needed to be moved into their own classes were moved, and utility functions and variables were kept in their own library. I think the project would have benefitted from examining these issues at the outset instead of discovering them along the way. If I had put more time planning the architecture, I would have had to rewrite and refactor the code less. It’s definitely something I need to be more aware of in the future.

Evaluation

Especially towards the end of the project, evaluating the key features needed for completion was extremely beneficial. The scope of the project kept growing as I kept working, and eventually I needed to stop and evaluate what features would be part of version 1.0.0 and what features would be part of the next major version. You can see throughout the code that some parts are commented out and labeled “v2.0.0”, indicating I had started writing the code but then decided to wait for a future release to finish it.

`browser_cli` Resources

GitHub Repository

https://github.com/KrashLeviathan/browser_cli

Published Dart package

https://pub.dartlang.org/packages/browser_cli

API Documentation[1]

https://www.dartdocs.org/documentation/browser_cli/1.0.0-alpha.3/

Hyperlinked Source Code[2]

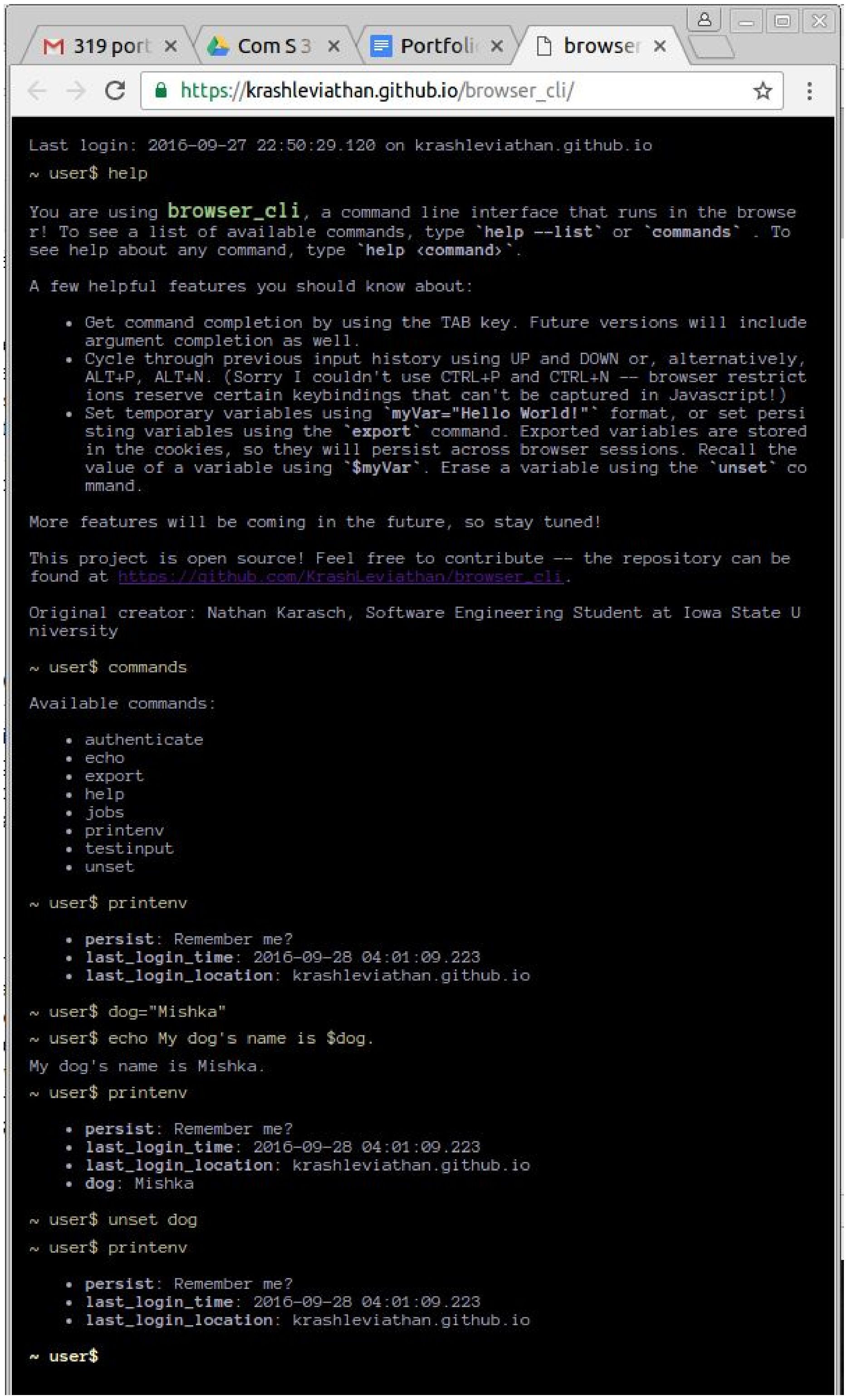
https://www.crossdart.info/p/browser_cli/1.0.0-alpha.3

Demo

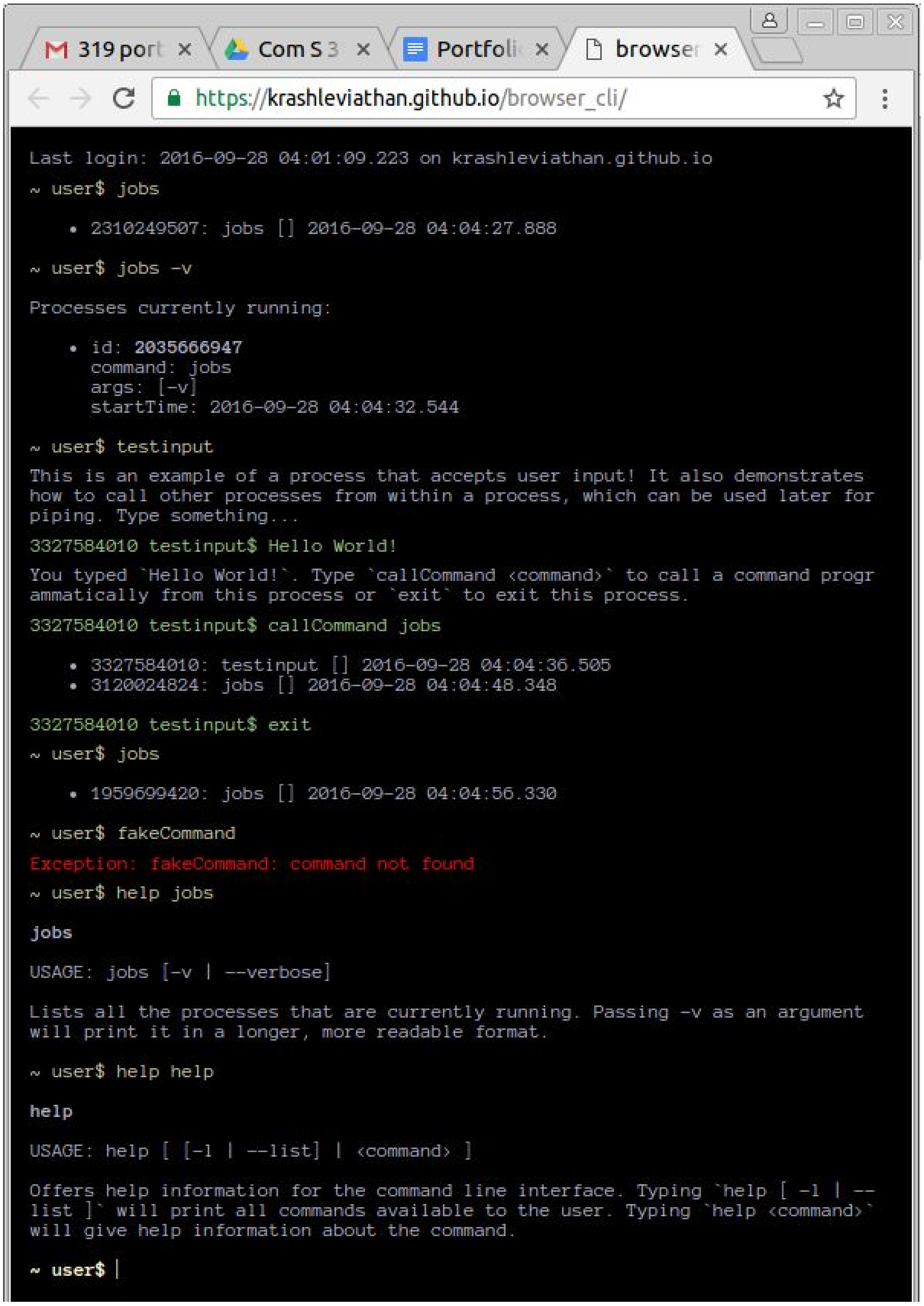
https://krashleviathan.github.io/browser_cli/

[1] This link is tied to the release number at the time this document was written. If the link is broken (i.e. newer releases have been made), please go to the Published Dart Package page and find the current link on the right side of the page.
[2] See footnote 1.

Screenshots



```
Last login: 2016-09-27 22:50:29.120 on krashleviathan.github.io
~ user$ help

You are using browser_cli, a command line interface that runs in the browse
r! To see a list of available commands, type `help --list` or `commands` . To
see help about any command, type `help <command>`.

A few helpful features you should know about:

    • Get command completion by using the TAB key. Future versions will include
      argument completion as well.
    • Cycle through previous input history using UP and DOWN or, alternatively,
      ALT+P, ALT+N. (Sorry I couldn't use CTRL+P and CTRL+N -- browser restrict
      ions reserve certain keybindings that can't be captured in Javascript!)
    • Set temporary variables using `myVar="Hello World!"` format, or set persi
      sting variables using the `export` command. Exported variables are stored
      in the cookies, so they will persist across browser sessions. Recall the
      value of a variable using `$myVar`. Erase a variable using the `unset` co
      mmand.

More features will be coming in the future, so stay tuned!

This project is open source! Feel free to contribute -- the repository can be
found at https://github.com/KrashLeviathan/browser_cli.

Original creator: Nathan Karasch, Software Engineering Student at Iowa State U
niversity

~ user$ commands

Available commands:

    • authenticate
    • echo
    • export
    • help
    • jobs
    • printenv
    • testinput
    • unset

~ user$ printenv

    • persist: Remember me?
    • last_login_time: 2016-09-28 04:01:09.223
    • last_login_location: krashleviathan.github.io

~ user$ dog="Mishka"
~ user$ echo My dog's name is $dog.
My dog's name is Mishka.
~ user$ printenv

    • persist: Remember me?
    • last_login_time: 2016-09-28 04:01:09.223
    • last_login_location: krashleviathan.github.io
    • dog: Mishka

~ user$ unset dog
~ user$ printenv

    • persist: Remember me?
    • last_login_time: 2016-09-28 04:01:09.223
    • last_login_location: krashleviathan.github.io

~ user$
```



```
Last login: 2016-09-28 04:01:09.223 on krashleviathan.github.io
~ user$ jobs

    • 2310249507: jobs [] 2016-09-28 04:04:27.888

~ user$ jobs -v

Processes currently running:

    • id: 2035666947
      command: jobs
      args: [-v]
      startTime: 2016-09-28 04:04:32.544

~ user$ testinput
This is an example of a process that accepts user input! It also demonstrates
how to call other processes from within a process, which can be used later for
piping. Type something...
3327584010 testinput$ Hello World!
You typed `Hello World!`. Type `callCommand <command>` to call a command progr
ammatically from this process or `exit` to exit this process.
3327584010 testinput$ callCommand jobs

    • 3327584010: testinput [] 2016-09-28 04:04:36.505
    • 3120024824: jobs [] 2016-09-28 04:04:48.348

3327584010 testinput$ exit
~ user$ jobs

    • 1959699420: jobs [] 2016-09-28 04:04:56.330

~ user$ fakeCommand
Exception: fakeCommand: command not found
~ user$ help jobs

jobs

USAGE: jobs [-v | --verbose]

Lists all the processes that are currently running. Passing -v as an argument
will print it in a longer, more readable format.

~ user$ help help

help

USAGE: help [ [-l | --list] | <command> ]

Offers help information for the command line interface. Typing `help [ -l | --
list ]` will print all commands available to the user. Typing `help <command>`
will give help information about the command.

~ user$ |
```