# jcom: A JavaScript Object Shell

DeDominic, Anthony
Eastern Connecticut State University
Willimantic, USA
dedominica@my.easternct.edu

## Abstract

*With the proliferation of JSON representation of data, shells struggle to make use of them; Currently, shell users rely on tools like jq to handle these structured documents. This paper will discuss an attempt to bring object-like variables to a new shell, jcom. The reason was to allow for shells to natively deserialize json-like documents and to work with them using dot-like notation. This resulted in a cleaner way of working with json-like documents. It also led to less characters being required to solve certain problems. This research suggests shells, like bash, can greatly benefit by providing such structures in later revisions.*

## 1. Introduction

No popular shells, as of late, have the ability to handle modern structured documents, such as JSON. Because of this, it is difficult to use shells for modern text and data handling. This is especially the case for web oriented services; such services heavily rely on transacting structured documents or serialized objects.

Shell users generally have to rely on line oriented tools– like sed, grep, awk, xargs, tr, paste, cut, tee, etc, to try to handle data in structured documents. Generally this leads to intricate and fragile parsers. Listing 1[1] demonstrates this problem. Listing 1 shows common way to handle html files in bash. The script makes use of various line oriented tools to find the top three results from a serach engine.

Listing 1: Parsing HTML in bash

```
# gets an html webpage from duckduckgo,
# processes it into
# "url - page description" lines
# removes ad links
# and only preserves the first three
curl ${SEARCH_ENGINE}${4} | \
html2 | \
grep -A 2 "@class=result__a" | \
sed '/^--$/d' | \
sed '/@class/d' | \
grep -Po '(?<=\/a(=|\/)).*' | \
paste -d "␣" - - | \
sed 's/\(@href\|b\)=//g' | \
sed '/r\.search\.yahoo\.com/d' | \
head -n 3
```

[1] source: https://github.com/GeneralUnRest/neo8ball-irc/blob/master/lib/search.sh

### 1.1. Related Works

Object Oriented shells are not an entirely new concept. Many projects, some more serious than others, attempted to solve this data-wrangling issue [1] [2].

#### 1.1.1. shcaml

Shells are very powerful tools. In many cases, they also serve as the most basic ui of an operating system. Most shells allow for tying streams of multiple processes together, allowing for interprocess communication. Such mechanisms is what allows shells to complete complex tasks. Because of this, many programming languages provide the ability to spawn shells. An example is the function system(), in the standard C library.

The power of the shell was not lost to some OCaml[2] developers; these developers created shcaml. shcaml, a library, offers a more powerful shell spawning construct for the OCaml language. A feature of shcaml allowed one to tie OCaml code with external commands as a sort of intermediate go-between. This allowed for powerful data-wrangling functional compositions that could take in data from shell commands and transform them into structured outputs [1].

#### 1.1.2. Named Pipe Shells

Another way of attaining structured inputs and outputs is through an object system. One attempt at object

[2] OCaml is a popular functional language.

oriented shells utilized named pipes and file descriptors to mimic objects [2]. This resulted in a new process for every object; the individual processes had their own internal state, mimicking object encapsulation.

These objects had many faults, however. Each object was basically its own shell, still requiring users to build complex scripts. Named pipes are not as secure as other means of interprocess communication; This is because anyone with the same user privileges can write to and read from named pipes.

#### 1.1.3. Lisp Shells

Lisp–another language that is function-oriented, shells have been other ways of trying to add structure. Examples are like Scheme Shell, and other academic ones [1][3]. The advantage lisp gives over other languages, which makes it suitable for shells, is that the program and data have the same form [3]; lisp code is effectively a mutable abstract syntax tree. This property comes in handy in meta programming, where programs can dynamically change to handle certain data. It is also a closed language, as in, one can pull from data that is in an outer functions. Closure property is what gives lisp its ability to resemble structured documents. Since lisp code is already in a abstract syntax tree form, JSON documents can represent lisp code.

#### 1.1.4. Powershell

Projects like Powershell are built on the .NET runtime and can thus utilize all of the language features therein. The streaming pipelines in Powershell are merely .NET classes which allow for some structural correctness and parsing [1][4]. As a result, Microsoft prefers to call it more than just a shell, but a whole "automation and configuration management framework." This framework comes with a handful of features, like the capability of handling structured data such as JSON, XML, CSV, etc, REST APIs, and object models.

#### 1.1.5. Ansible

Tools like Ansible, are not quite shells, but are domain specific languages which solve similar problems. Ansible and Salt Stack use code generation and structured document format called YAML, to construct python code. Since python is a general purpose language, it handles structured and typed data much better than shell code. Ansible, as a result, has become a cornerstone of modern IT automation, provisioning and configuration management; many of those listed problems require working with such data.

The next few sections will cover some background on important topics and issues around this space, basically what these prior works tried to solve.

## 1.2. Service-oriented Architecture

Service-oriented Architectures, or SOA, are modern way to construct, highly available, data rich web applications. In short, a service oriented architecture is a way for applications to share state and provide services over middleware and structured, serialized objects. [5] One way of transacting states between services is through Representational state transfer, or REST. With REST, services are able to share well structured data through document structures like JSON, XML and many others. [6]

POSIX-shells, on their own, are completely useless for interacting with service oriented architectures. Generally when dealing with web APIs it is recommended to build a script–using a general purpose language[3], instead. If an ansible module exists, one might want to consider that as well.

## 1.3. Designing and Defining a Domain Languages

Designing domain specific languages is a very slow and arduous task. Some have suggested designing and developing domain specific languages using meta-models; basically a model, modeling models [7]. This process makes use of advanced features of UML version 2.1. This also has the added benefit of code generation.

Defining a language is ultimately the process of creating formal grammars. Generally this is done empirically, basically from experience, trial and error. Some have used machine learning and bottom-up CYK based parsers to generate grammar [8]. This would be useful for languages built using example strings of unknown structure than more well defined languages.

## 1.4. Events

Some system events may impact other systems. Currently, other than using triggers, there is no way to define listen events in POSIX shells. Powershell offers such functionality through the power of its .NET classes [4]. To accomplish a similar function, one must use a blocking statement such as one shown in Listing 2

[3]Examples: Python, JavaScript, Perl, Ruby.

Listing 2: Event driven programming in a shell

```
# file descriptor is needed
# for performance reasons
exec <>99 /some/named-pipe
# consumes a process, must be backgrounded
while read -r line; do
    echo $line
done <&99 &
echo "event" >&99
```

## 1.5. Portability Concerns

Shells are the primary interface into a POSIX system. As a result, many tools and utilities rely on shells to handle and process files and their data. A collection of standard tools exist to accomplish this goal. This collection is called the coreutils. Despite the POSIX specification, different coreutils may be implemented with different, nonstandard features and capabilities. For instance, many of the GNU coreutils–such as their implementation of grep, df and even readlink, add features not strictly standardized by POSIX which can be seen in Listing 3.

Listing 3: Nonstandard use of POSIX standardized grep

```
# (?>) is a backreference in PCRE
# -P flag enables PCRE grep
# in the GNU coreutils
grep -P '(?>=)somepat' file
# does not work on macOS, which uses,
# mostly, BSD coreutils
```

Different operating systems–ranging from GNU-based Linux distributions, The BSDs and other non-free systems like macOS, Solaris, AIX, and other UNIX, can potentially use different coreutils. Thus, this means that shells scripts can depend on behaviors and unstructured output that are not portable.

## 1.6. Going Further

### 1.6.1. Interactively

An OCaml library called shcaml added functional combinators called shtreams which allow for transforming byte streams into structured documents [1]. However, it is merely a library, to be used in making OCaml programs. It was not intended to be used interactively. This makes it unsuitable as an actual systems shell.

### 1.6.2 Readability

Some developers have tried to add other structures to shell input, such as Scheme Shell and other implementations [3]. Using functional composition, one can construct a very powerful shell, but it will not be as clear and concise as normal shell syntax. This is because lisp does not allow for pipelining-like syntax. For instance, consider Listing 4 where one must use a variadic pipe function vs Listing 5 which shows a basic pipeline.

Listing 4: A lisp shell pipeline

```
(|
  (cat somefile)
  (some-command))
```

Listing 5: Shell pipeline

```
cat somefile | some-command
```

### 1.6.3. Performance

The named-pipe shell would simply be too slow. The need to spawn new processes for each object and dealing with potentially slow named pipe interaction is too heavy [2]. One can used file descriptors trick to enhance the performance of named pipes.

## 1.7 Objective

Given all the above, the goal is to build a new shell. This shell should be able to execute simple commands and offer basic shell features like pipelining. To differentiate it from other shells, it should be able directly work and manipulate modern data structures and documents like JSON. The shell should be fully interactive, like a shell is suppose to be. The shell should keep with traditional POSIX shell syntax as much as possible, both for readability concerns, but also for familiarity. Ideally it should be fast, but since it will be written in a scripting language, this may be unattainable in this current project. To extend the shell's functionality, it is necessary that it have a plugin system. The plugin system will allow for adding features and capabilities. It is the goal of this research not only to build a shell, but to also observe the positive utility and advantage of the shell being developed.

## 1.8. Summary of Results

Below, this paper will cover the features of the shell[4]. The paper will discuss how the program stores variables,

[4]As of this revision, some features like events are not implemented.

including objects, or documents. It will also show examples of usage, both manipulating and using variables. Other examples, such as use cases and the plugin system will be demonstrated.

The final section will show real uses of the shell. This will include code examples, one in jcom and one in bash. It will conclude a comparison in how many lines of code and characters needed, will be made.

## 2. Materials and Methodology

In order to make use of the technologies described, first the user must have NodeJS. The version of node being used on the test machine will be the latest–as of this writing, v7.2.1. With NodeJS comes crucial tools like npm. npm is the tool that handles dependencies; The project makes use of such dependencies like pegjs. When downloading the source of the project from github, it is necessary to run npm install to require dependencies.

The project is version controlled using git and is hosted on a remote repository on github[5]. The easiest way to check out the project would be to clone it using a git utility. npm can also download the latest build of the project. This is done by typing *npm install url*.

#### 2.3.1. Libraries

The project makes use of the following libraries. For a more up-to-date list, please consult the package.json file in the project which lists all the dependencies and their semver version number:

- pegjs - parser generator.
- lodash - utility functions for functional array and object handling.
- js-yaml - a yaml parser for javascript, for an example plugin.
- various NodeJS built-ins: fs, process, readline, etc.

#### 2.3.2. Testing Environments[6]

The machine being used for testing is a Fedora GNU/Linux, release 24, virtual machine running on a Windows 10 Hyper-V hypervisor. This hypervisor has a Core i7-3930k processor and 32GB of memory. The virtual machine has been allocated 2 cores of the 6 physical, with dynamically resizing memory, up to 16GB. To ensure support for the latest features and capabilities, the latest release–as of this writing, v7.2.1, was installed on the machine. Interaction with the machine will be done over a network; this is accomplished using a remote shell protocol called secure shell (ssh).

Any and all scripts necessary to reproduce the results will be made available in the github link provided in the footer. Please consult the examples/ directory in the repository.

## 3. Results

### 3.1. Objects and Structured Documents

The main objective of the project is to process structured documents. To accomplish this, the language will offer a plugin system. Plugins in this context are merely JavaScript functions which return–or return a value via a callback, JavaScript objects. This workflow can be clearly seen in Figure 1, which shows how a user's input line can be saved into a variable. Currently, as this figure* shows, one can only assign values to variables using the templating engine or a plugin. Listing 6 shows, internally, how this storage works in the shell. This is normal JavaScript assignment.

Listing 6: Variable assignment, internal

```
// store some input or output buffer
// parsed by a plugin named parseJson
SHELL_VARS[ident] = parseJson(buffer)
// a plugin that uses xpath to
// select some values from an
// XML document and stores it
SHELL_VARS[ident2] = xpathSelector(
    buffer, args
)
```

Ultimately, the variables are stored in one giant JavaScript object. This is how JavaScript functions, closures and objects store variables locally. Because of this, it simplifies variable storage and management of variables of jcom.

To access these stored variables, as shown in Listing 7, Variables are accessed using JavaScript ES6, template literal, syntax. Listing 6 also shows counter examples of variable access in bash, the comments in the example delineate the two.

Listing 7: Accessing variables in jcom

```
// getting an array element from an object
${somevar.somearr[1]}
// bash does not have nested objects
${somevar[1]}
```

[5]Source: https://github.com/adedomin/jcom. You can Acquire the source using: git clone https://github.com/adedomin/jcom

[6]Note in the presentation, testing and development was done on a Macbook Pro 11,1.

[7]This shows how one makes use of variables and the ways to get and set them.

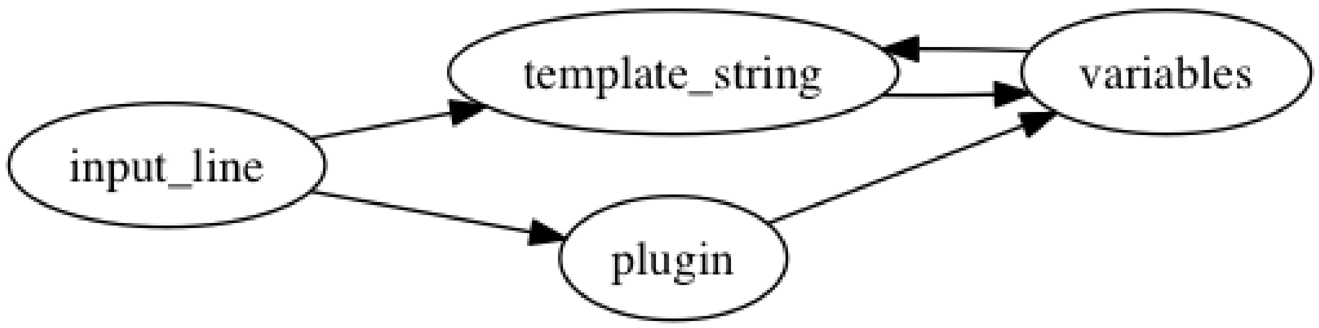


Figure 1: Variable workflow[7]

```
// iterate over every element in an array
${somevar.forEach(var => do something)}
// bash
for var in ${somevar[@]}; do
     something
done
// NESTED OBJECTS
// not possible in bash as of v4.3
${somevar.someotherobj.somekey.moreobjs}
```

Again, as shown in the above example, JavaScript notation and variable storage opens up an avenue to handle various kinds of data. This opens up possibilities to accomplish more general tasks that are normally left to general scripting languages[8].

## 3.2. Events

Shells have the ability to create file descriptors. With these, users have a fast way of connecting program's input and output streams together. File descriptors are usually how one would recreate–what could best be described, as an event loop.

NodeJS offers an object called an EventEmitter. With it, one can create software with low coupling. EventEmitters are very simple, the only two methods that are of importance are on() and emit(). emit() lets one create an event labeled by a string, to be handled by an on(). The on() handles events of a particular label with a function.

The event system of jcom can be seen in Figure 2. The user inputs a command line. If the line invokes the *on* command, it takes the first argument as a label and the second as a command to save to that label. The *emit* command will take the first argument as a label and will invoke the command associated with that label; the command, with the label, was defined by the *on* command. To make these events useful, the *emit* command accepts an input stream and will feed it into the command. This interaction is better demonstrated in Listing 8, specifically, lines two through four.

Listing 8: jcom event system example

```
# prints what it is fed
on 'echo' cat
# will print hello, world
echo 'hello, ' | emit 'echo'
echo 'world' | emit 'echo'

# reload config on config change
on 'configChange' \
     "kill␣-SIGWINCH␣${some_server_pid}"
# load inf parser plugin
load parse-ini
parse-ini SERVER_CONFIG /etc/server_config
echo "db://newdb.url" \
     | toVar "SERVER_CONFIG.db_url"
parse-ini SERVER_CONFIG \
    --serialize -o /etc/server_config
# reload changed config
emit "configChange"
```

Listing 8 shows these events can be incredibly useful for tasks such as reloading a server when a configuration file is changed. It also demonstrates the power of the plugin system. Users can create plugins to parse their configurations into JavaScript objects. This allows for an easy way to manipulate configurations and reconstitute them, with changes, back to a file.

## 3.3. Command and File Parsers

As shown in Listing 8, the shell can simplify some common configuration tasks. Many common shell tasks involve handling file data. Seemingly simple tasks, like

[8]Examples: Python, JavaScript, Perl, Ruby.

[9]This shows how the event system works by registering commands or daemons to labels and invoking or feeding more data into them using an accompanied emit.

[10]This shows how commands feed serialized documents and how plugins manipulate the shell's variable pool to store them.

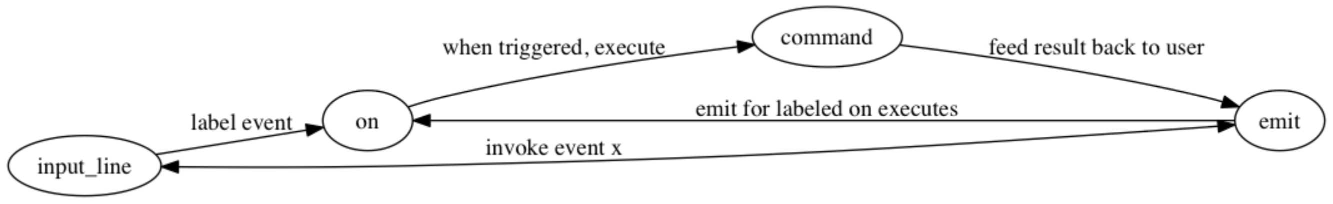


Figure 2: Event workflow[9]

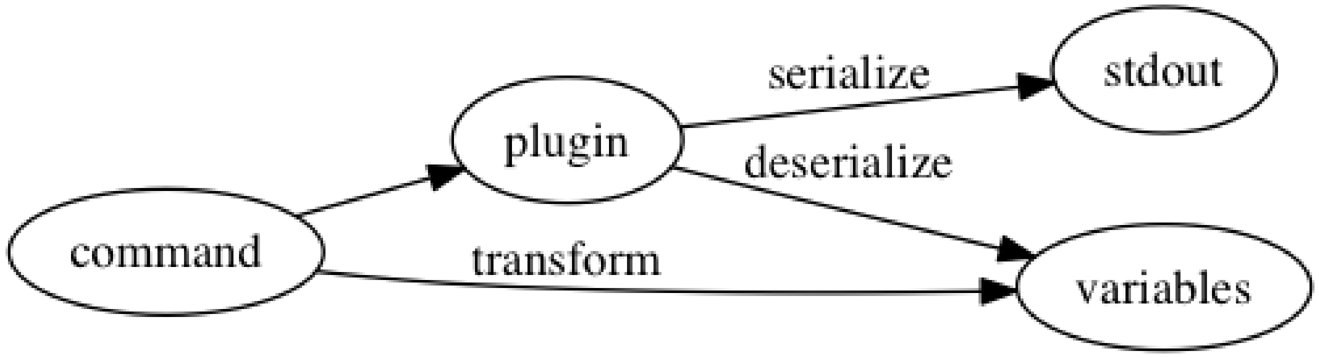


Figure 3: Document parsing and manipulation[10]

modifying configuration files, usually require the user to open up a text editor and leave the shell. Text editor interactions generally cannot be scripted, at least not easily. In order to overcome some of these challenges, Users usually have to do something akin to Listing 1 where they string together various commands to parse and transform the file.

Like JSON documents, many times configuration files have some formal structure. A common structure is the key:value pattern. Usually files have keys and their values and they are separated by a simple equality sign; such files are easily handled by tools like grep and sed.

More advanced configuration structures like ini files, give the user the ability to have single-level nested objects and comments. These files can still be easily transformed with basic line oriented tools. However, consider rich configuration structures like YAML or even Java .properties files. These structures can have infinitely nested objects, arrays and various primitive types. Such configuration files become harder to modify with line oriented tools. However, consider Listing 9. Like Figure 3, It shows how one leverages a plugin to read in a file, deserializes it into a jcom variable and how a user can turn this variable into a file via a serialization mechanism. This allows one to edit files without resorting to a text editor, opening up opportunities to automate configuration management.

Listing 9: jcom parser plugin example

```
# parse nagios's main configuration
load nagios-parser
nagios-parser --to-var NAGIOS nagios.cfg
 # prints info related to host1
echo ${NAGIOS.host1}
${NAGIOS.host1 = {new obj}} # new host1
nagios-parser --serialize -o nagios.cfg
```

## 3.4. Outcome, Examples

This section will show real examples of jcom being used. These uses will be directly compared to equivalent bash implementations. The purpose of this is to objectively measure if jcom requires less lines of code, or number of characters, than an equivalent bash script.

### 3.4.1. Example 1

Referring to Figure 4, this simple example shows how to read in a JSON document and access the data within it. One thing to note is how the bash script has to parse the file multiple times in order to store the values in variables.

### 3.4.2. Example 2

This example, which references Figures 5 and 6, shows an interaction with a web server. The jcom one, Figure 5, shows off jcom's ability to store documents, deserialized, into JavaScript objects. Because assignment functionality

```
# jcom
cat package.json | json --to-var jcom
echo "${jcom.name} by ${jcom.author}"

jcom by Anthony DeDominic
```

```
# bash
JCOM_NAME=$(jq -r '.name' < package.json
JCOM_AUTHOR=$(jq -r '.author' <
package.json)
echo "$JCOM_NAME by $JCOM_AUTHOR"

jcom by Anthony DeDominic
```

Figure 4: Example 1

is not implemented, the noop line shows how one can manipulate and assign values to variables. The next lines show off the serialization feature which allows a user to write out the variable as a document. The variable is then written as a JSON document, where it is fed into curl so it can be sent to the server. The server confirms that the update was successful.

Figure 6 shows an equivalent script written in bash. What distinguishes it from jcom is the use of jq to parse a JSON string, which is stored in the HOST variable. One interesting thing to note is how the HOST JSON string is modified. Some information could have potentially been lost; This is because this example assumes that $HOST only contains two attributes, name and address, which may be false.

**3.4.3. Example 3**

This example, seen in Figure 7, shows how jcom can work with other structured documents. This example shows an HTML plugin, which allows the user to find information using CSS selectors. Note however, most of the logic, in this example, is JavaScript; everything in the template string is JavaScript.

Listing 1 shows the bash counter example to Figure 7. This shows that the tools available strongly affect the difficulty of completing a task in bash. This example makes use of a tool called html2–a tool to transform html into lines that can be worked on using coreutils. Because the html was transformed into line friendly output, its possible to use coreutils seen–grep, sed, etc, to obtain the information needed.

**3.4.4. Lines of Code**

| Figure | Lines of code | # of characters |
|---|---:|---:|
| jcom example 1 | 2 | 76 |
| bash example 1 | 3 | 131 |
| jcom example 2 | 5 | 287 |
| bash example 2 | 5 | 303 |
| jcom example 3 | 4 | 245 |
| bash example 3 | 10 | 270 |

As can be seen from this table, jcom used less lines of code in all but example 2–Figures 5 and 6. On every example, jcom used less characters overall.

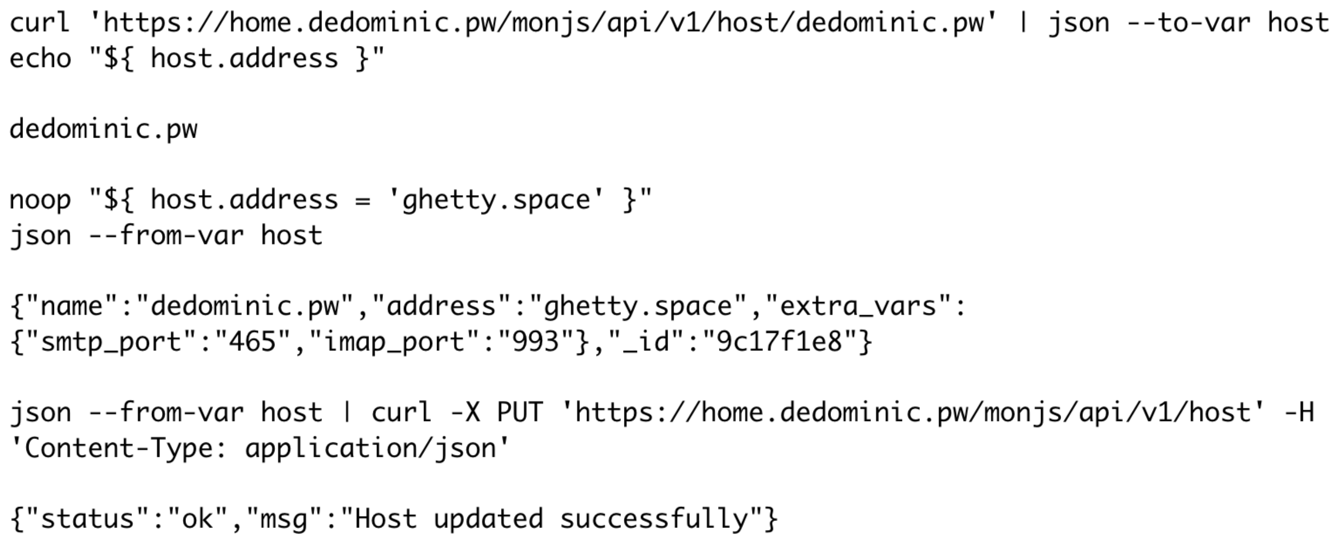
## 4. Discussion

The results show that jcom is much more concise. This can be attributed to jcom's syntax. Because of jcom's object system, users do not need to rely on the bulky bash syntax shown in Figure 4, 6 and Listing 1. Figure 6, for instance, requires inline HEREDOCs and output capture groups; but it also shows that security is always a concern when working with variables in bash. Variables must be quoted to prevent word splittings, which can be seen in Figure 6 especially. This is not a concern in jcom.

This feature of jcom makes it more readable as well. This is because jcom's variable access syntax resembles common languages like JavaScript and Java. In bash, accessing the same information required the use of complex tools like jq.

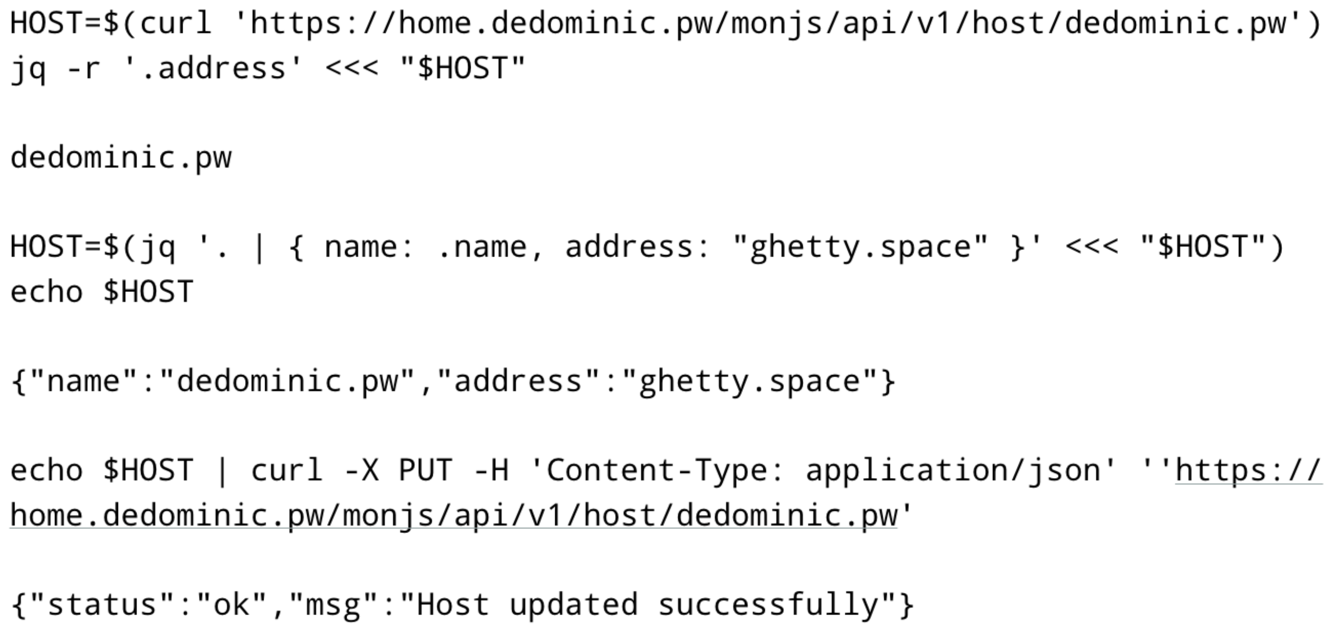
Unlike shcaml, jcom is entirely interactive. Users can directly execute commands in a prompt, where shcaml was a library that extended the powers of OCaml. Like shcaml, jcom offers similar functionality; For instance, plugins were similar to shcaml fittings.

The plugin system greatly benefited jcom and ultimately makes it a potential contender with Powershell. For instance, It was very easy to quickly incorporate the html plugin, as seen in Figure 7. The immediate result was the ability to serialize html into a jcom objects. This is comparable to how Powershell can be extended



```
curl 'https://home.dedominic.pw/monjs/api/v1/host/dedominic.pw' | json --to-var host
echo "${ host.address }"

dedominic.pw

noop "${ host.address = 'ghetty.space' }"
json --from-var host

{"name":"dedominic.pw","address":"ghetty.space","extra_vars":
{"smtp_port":"465","imap_port":"993"},"_id":"9c17f1e8"}

json --from-var host | curl -X PUT 'https://home.dedominic.pw/monjs/api/v1/host' -H
'Content-Type: application/json'

{"status":"ok","msg":"Host updated successfully"}
```

Figure 5: Example 2 - jcom



```
HOST=$(curl 'https://home.dedominic.pw/monjs/api/v1/host/dedominic.pw')
jq -r '.address' <<< "$HOST"

dedominic.pw

HOST=$(jq '. | { name: .name, address: "ghetty.space" }' <<< "$HOST")
echo $HOST

{"name":"dedominic.pw","address":"ghetty.space"}

echo $HOST | curl -X PUT -H 'Content-Type: application/json' ''https://
home.dedominic.pw/monjs/api/v1/host/dedominic.pw'

{"status":"ok","msg":"Host updated successfully"}
```

Figure 6: Example 2 - bash

```
curl "https://duckduckgo.com/html/?q=css+selector" | html --to-var ddg
echo "${(() => [0,1,2].map(val => `\
${ddg('.result__a').slice(val,val+1).attr('href')} $
{ddg('.result__a').slice(val,val+1).html().replace(/<\/*b>/g, '')}`).join('')\
)()}"

http://www.w3schools.com/cssref/css_selectors.asp CSS Selectors Reference -
W3Schools Online Web Tutorials
https://developer.mozilla.org/en-US/docs/Web/Guide/CSS/Getting_started/Selectors
Selectors - Web developer guides | MDN
http://www.w3.org/TR/CSS2/selector.html CSS selectors - World Wide Web Consortium
```

Figure 7: Example 3 - jcom

with .NET code. jcom can potentially call itself an automation and configuration management framework, like Powershell.

Communication with web services–as demonstrated in Figure 5, felt simpler. There was no need for complex tools like jq, or line oriented sed transformations, to accomplish the needed updates to the data, as is shown in Figure 6. This means that jcom can potentially be used instead of general purpose languages to interact with web services.

## 4.1. Significance

The examples, 1 through 3 as shown in section 3.4., shows a greater level of simplicity and power in handling structured documents. It makes use of common dot notation to access methods and member variables in objects. This is unlike many shells which can only store simple scalar values or vectors of these values. As a result interacting with modern middleware-oriented services, are greatly simplified. This simplicity would allow shells to be reconsidered for web scripting. This also means that users can easily interact with web services as well, a task not possible with general purpose languages.

The plugin system–and the support for complex variables, shows how a shell can become a complete configuration management framework. With plugins, it becomes simpler to directly interact with configuration files. Given one of the common system administration tasks is working with configuration files, it makes sense that shells should offer such a feature.

Basically, jcom paves a path for more modern shells. Shells able to interact with modern, document structured data. The days of line oriented parsing are nearly dead, especially with the advent of service oriented architectures.

## 4.2. Limitations

Currently, the shell is not finished. For instance, branching is not implemented. There are stream related issues which prevent writing to files. Variable assignment and events are currently not implemented as of this revision. That being said, variable assignment can be done using the JavaScript Templating engine–refer to Figure 5. Currently the only thing that works is execution pipelining; thus the examples, referenced in section 3.4., are runnable.

The JavaScript templating engine sort of blurs the lines of the shell and JavaScript. The templating engine effectively allows the user to execute JavaScript inline. It can also slow the engine down since the parser is run after the JavaScript has been evaluated. Also, the templating engine makes use of an old JavaScript function which should be avoided, like with(); with() may be removed in later versions of the language; This would slightly complicate variable access as it is currently designed.

Even though the jcom examples, referenced in section 3.4., used less lines of code and less characters, the difference was not that staggering. This can ba attributed to tools like jq. Such tools gave bash new powers in terms of the kind of work it can feasibly do; for instance, jq allowed for bash to handle and modify JSON documents. This is something bash does not do well.

## 4.3 Future Works

Currently jcom needs more work. Many of the bugs and features should be fixed or implemented. After that is complete, more complex examples should be developed. These examples should try to find things not even a shell can do in a reasonable amount of lines of code[11]. The purpose is to build a stronger case to persuade shell projects to implement these features.

Attempts should be made to add object like features to shells like bash. Because shells like bash have a large userbase, it is hard to fully deprecate it. Because of this, for object-like features to become mainstream, they have to implemented in bash. Thus the next step would be to fork bash, implement some or all of the features–as were discussed in the results section, and then submit a pull request.

## 4.4. Conclusion

The results show that jcom makes it simpler to interact with structured documents. This is a result of jcom's features, but also the ability to store complex data structures as jcom objects. Because of this, it feels more natural to use overall and a potential contender to current shells–especially powershell. The next steps are to try and push these modern features into current, popular shells.

# Citations

[1] A. Heller and J. Tov, "Caml-Shcaml: An OCaml Library for UNIX Shell Programming," 2008. DOI: 10.1145/1411304.1411316

[2] J. Haemer, "A New Object-Oriented Programming Language: sh," 1994 [Online]. Available: http://https://www.usenix.org/legacy/publications/library/proceedings/bos94/full_papers/haemer.ps

[3] J. Ellis, "A LISP SHELL," 1980. DOI: 10.1145/947639.947642

[4] J. Snover, "Monad Manifesto." Microsoft, Aug-2002 [Online]. Available: http://www.jsnover.com/Docs/MonadManifesto.pdf

[5] Perrey and Lycett, "Service-oriented architecture." IEEE, Jan-2003. DOI: 10.1109/SAINTW.2003.1210138

[6] R. Battle and E. Benson, "Bridging the semantic Web and Web 2.0 with representational state transfer (REST)." Elsevier, 2008. DOI: 10.1016/j.websem.2007.11.002

[7] B. Selic, "A Systematic Approach to Domain-Specific Language Design Using UML." IEEE, May-2007. DOI: 10.1109/ISORC.2007.10

[8] K. Nakamura and T. Ishiwata, "Synthesizing Context Free Grammars from Sample Strings Based on Inductive CYK Algorithm." Springer, Grammatical Inference: Algorithms and Applications, 2000. DOI: 10.1007/978-3-540-45257-7_15

[11] By reasonable, I mean three or four times more lines of code than a jcom example.