Title

Redesigning CHIML: Orchestration Language for Chimera-Framework

Author

- Go Frendi Gunawan (frendi@stiki.ac.id)
- Jozua Ferjanus Palandi (jozuafp@stiki.ac.id)
- Subari (subari@stiki.ac.id)

Abstract

Component Based Software Engineering (CBSE) has been proven to be quite effective to deal with software complexity. Nowadays developers prefer to build micro-services rather than single monolithic application. Several SOA (Service Oriented Architecture) approaches like HTTP/REST API, CORBA, and BPEL are commonly used by developers. Some of those solutions are built under assumptions that the developers are either building the services from scratch or able to create abstraction layer for the pre-existing services. In most cases the assumptions are true. However there are cases when developers prefer to keep the architecture as simple as possible without any need to build additional abstraction layers. For example, when they work with mini-embeded system.

Previously, a YAML based orchestration language was developed for Chimera-Framework (A language agnostic framework for stand-alone and distributed computing). In this paper, we refine the orchestration language in order to let developers accessing pre-existing services without any need to build another abstraction layer.

Keyword : Chimera, CBSE, Orchestration Language

Introduction

Software development is a very interesting topic. The way people developing softwares is changing as new paradigms emerged. In turns, software development also affecting the culture. It change how people interact to each others as well as how they interact with computers.

As software become more and more complex, building and maintaining softwares is also become harder. Various approaches have been attempted in order to make the processes easier.

Several frameworks like Rails, Django, Laravel, etc are focusing on how to make codes clean, separated, and reusable. In turn, this make software development and maintainance easier. These frameworks quickly gained popularity among software developers.

However, despite on the clear advantage of those frameworks, they also have several disadvantages. Since the frameworks were built on top of specific programming language, integration to different programming languages is quite challanging. For example, in order to build software by using Rails, developers should code in Ruby. The same is also true for Django, which is built on top of Python, and Laravel which is built on top of PHP. This situation is known as vendor-lock-in. And in some cases it can affect maintainability as well as further development in a bad way.

Thus, a more technology agnostic approach is needed in order to overcome vendor-lock-in problem. Recently, SOA and micro-services are gaining popularity. Those approaches let developers to focus on small components known as services rather than complex monolithic software. A service usually represent single resource or business process. The services can be independent to each other or aware to each others. Despite of the advantages and popularity, SOA and micoservices are prone to concurrency problem ref.

One important aspect in SOA/micro-services is how a developer can compose independent services to work together. There are two common approach to compose services. The process to compose several services that are independent to each others is named orchestration, while the process to compose several services that aware to each others is named choreography. Orchestration require one central controller in order to manage the services ref.

Orchestration has a very long history. The earliest implementation of orchestration was Unix Pipe mechanism. This mechanism is still relevant and used by Unix/Linux users.

Unix Pipe mechanism is not the only implementation of softwre orchestration. Remote Procedure Call (RPC) is also used for bigger projects. Beside Unix Pipe and RPC Several other implementations were introduced by independent companies and consortiums. OMG introduced CORBA on 1991, while OASIS intoduced BPEL on 2001. Some developers also implement their own HTTP/REST API implementation. On 2016, Feilhauer and Sobotka creating a framework named DEF which is focusing on parallel execution. On 2017, we also conduct a research to develop another language agnostic framework named Chimera-Framework. Recently Google also introduce GRPC, a more modern and sophisticated version of RPC.

Compared to the monolithic frameworks like Rails and Django, these orchestration mechanism are more scalable and technology agnostic. This mean that the developers can choose the best technology stack to build their components/services.

Among those orchestration mechanisms, Unix pipe and Chimera-Framework are the only ones that are agnostic to the architecture and messaging protocol. To implement CORBA or BPEL, the developers has to specify IDL or WSDL. In DEF, the similar mechanism named DEF-WS also has to be created. These specifications also assuming that components are HTTP aware. Consequently, the developers need to consider this behavior before creating a new service/component. It doesn't mean that pre-existing components cannot be used for orchestration. However the components are either need refactoring or additional broker. This make the deployment process more complex.

Chimera-Framework in the other hand focusing on how to make the effort minimal. The orchestration components in Chimera-Framework don't have to be HTTP aware. Even an old UNIX utility like `date` or `cat` will serve well.

In this research we are focusing in improving the orchestration mechanism in Chimera-Framework. The orchestration language is named CHIML (Chimera Markup Language) which is a superset of YAML. CHIML is designed to be readable, compact, and intuitive.

Research Question

In order to have a clear direction in our research, we are focusing in these two questions:

- How to make a readable, compact, and intuitive orchestration language for Chimera-Framework
- How the orchestration language compared to other possible solution

Literature Survey

Orchestration and Choreography

In 1975, Frank DeRemer and Hans Kron wrote a paper about programming-in-large and programming-in-small ref. Programming-in-large is a concept to write a software that consists of various modules that are possibly written by different people.

The concept of programming-in-large is tightly coupled to orchestration and choreography. In orchestration, the components are controlled by a single controller, while in choreography, the components are aware of each others. In some cases, orchestration and choreography can be used together. Developer might create a choreography of components where the components are orchestration of smaller components ref.

SOA and Micro-service

SOA and micro-service are similar architecture concept. The core philosophy of these two paradigms are to split big problem into small problems. ref. The term SOA is usually being used when people talk in a bigger scope, while micro-service is usually refer to smaller scope. When services are composed and used internally it is micro-service. But when the services exposed to external system, it is SOA. Although the distinction can be unclear, people usually agree that SOA might contains of micro-services ref.

From the technical point of view, SOA and micro-service might involve either orchestration or choreography. And no matter which one is being used, a protocol for passing message among services is required.

HTTP/REST API

HTTP API is a quite common protocol. The architecture is also relatively simple. There are web services and a clients. The client send a request to the web service, and the web service reply with a response. The response data is usually in JSON/XML format. SOA components are commonly exposing HTTP API endpoint. OMDB for example, provide an API to search for movie database.

On the other hand, REST (Representation State Transfer) is a more strict implementation of HTTP API. It was introduced and defined in 2000 by Roy Fielding in his doctoral thesis ref. Fiedling focusing on how a URI should represent an object, while HTTP verb serve as it's method. If the endpoint is fully adapting REST specification, it is called `RESTful web service`.

SOAP

SOAP (Simple Object Access Protocol) is a standard XML format for sending and receiving message. In SOAP, every message has to be wrapped in an envelope element. Inside an evelope, developer can write header and body. Header should contains application specific information, while body should contains the message itself.

SOAP containing many standards that are not specified in HTTP/REST API. However, this also mean that SOAP is more verbose and eat more bandwidth than HTTP/REST API.

Nowadays, SOAP is still used and supported by many system. But, for new system, developers tend to choose simpler protocol.

CORBA, BPEL, and EJB

CORBA (Common Object Request Broker) is a specification created by OMG (Object Management Group). It was published in 1991, and it's last version was released on November 2012. CORBA supporting a lot of programming languages including C and Java.

BPEL (Business Process Execution Language) is another specification published by OASIS. I was published in 2003. BPEL is usually used by enterprise. Unlike CORBA, BPEL is more language agnostic. That means that the components can be written in any language.

EJB (Enterprise Java Beans) is another specification for microservice. compared to CORBA and BPEL, EJB is less language agnostic. in EJB, the orchestration component has to be written in Java.

CORBA, BPEL, and EJB are more strict compared to HTTP API. The server and the client should be agree on the data format. This agreement is usually written in IDL (Interface Definition Language). Since the server and the client has already agree on the data format, server response can be shorter than HTTP/REST API.

We conclude that although CORBA, BPEL, and EJB are more difficult to set up compared to HTTP/REST API and SOAP, it might benefit the system for the long run.

JSON-RPC, XML-RPC, and GRPC

RPC stands for Remote Procedure Call. The focus of RPC is to let developer invoke procedures in remote computer as easy as they are a local procedure. JSON-RPC and XML-RPC works in a same manner, only the data exchange format is different. As the names implied, JSON-RPC use JSON, while XML-RPC use XML. First the client send a request to the server. The request contains the name of the method and the parameters. The server then replied with the return value.

As XML is more verbose than JSON, XML-RPC is also need more bandwidth compared to JSON-RPC. And as XML and JSON are text format, binary data should also be encoded in text format. For example, an image might need to be encoded in base-64 format.

To overcome the need to encode data into text format, Google create another protocol named GRPC (Google Remote Procedure Call). GRPC supporting several programming languages including Python, Javascript, Java, and PHP. In order to use GRPC, developers have to create stub skeleton (similar to IDL in CORBA, BPEL, or EJB).

CHIML

CHIML (Chimera Markup Language) is a language used in Chimera-Framework. CHIML is an orchestration language. In term of language agnosticism and simplicity, Chimera-Framework is comparable to HTTP API. It doesn't even require HTTP protocol as any valid command line executable can serve as component. Our goal is to make CHIML as readable and as compact as possible. In order to measure the readability of the language, we conduct a simple survey with 5 respondents.

We provide three HTTP API endpoint and ask the respondents to compose the API in order to print the data in a specified format. The respondents should conduct two solutions. In the first solution, the respondents can choose any technology they like. But for the second solution, the respondents have to use CHIML.

The results are then analyzed and compared to get an overview about CHIML.

Design

`CHIML` is a superset of `YAML`. So, any valid `YAML` is also a valid `CHIML`. And as `YAML` itself is a superset of `JSON`, any valid `JSON` is also a valid `CHIML`

Unlike `YAML`, in `CHIML` is the developers are allowed to write string after block delimiter (`|` and `>`) without changing the line.

The `CHIML-script` should contain a single `<program>`.

Semantic (Backus Naur Form)

```
<program> ::= <completeVars>
              <completeVerbose>
              <command>
              <completeCatch>
              <completeThrow>

<command> ::= <completeCommand>
            | <shortCommand>

<completeCommand> ::= <completeIns>
                      <completeOut>
                      <completeIf>
                      "do: "<singleCommand><newLine>
                      <completeWhile>

                    | <completeIns>
```

```
                        <completeOut>
                        <completeIf>
                        "parallel: "<singleCommand><newLine>
                        <completeWhile>

                      | <completeIns>
                        <completeOut>
                        <completeIf>
                        "do: "<commandList>
                        <completeWhile>

                      | <completeIns>
                        <completeOut>
                        <completeIf>
                        "parallel: "<commandList>
                        <completeWhile>

                      | "map: "<variableName>
                        "into: "<variableName>
                        <completeCommand>

                      | "filter: "<variableName>
                        "into: "<variableName>
                        <completeCommand>

<shortCommand> ::= "|("<ins>") -> " <singleCommand> " -> " <out><newLine>
                 | "|("<ins>") -> " <singleCommand> "<newLine>
                 | "|"<singleCommand> " -> " <out><newLine>
                 | "|("<ins>") --> " <out><newLine>
                 | "|"<out> " <-- ("<ins>")"<newLine>

<commandList>  ::= "- "<command>
                 | <commandList><commandList>

<completeCatch> ::= ""
                  | "catch: "<condition><newLine>

<completeThrow> ::= ""
                  | "throw: "<string><newLine>

<completeVars>  ::= ""
                  | "vars: "<variableList><newLine>

<completeVerbose> ::= ""
                    | "verbose: "<verbosity><newLine>
```

```
<completeIns>   ::= ""
                  | "ins: "<ins><newLine>

<completeOut>   ::= ""
                  | "out: "<out><newLine>

<completeIf>    ::= ""
                  | "if: "<condition><newLine>

<completeWhile> ::= ""
                  | "While: "<condition><newLine>

<ins> ::= <variableList>

<out> ::= <variableName>

<singleCommand> ::= <cliCommand>
                  | <jsArrowFunction>
                  | "{"<jsNormalFunction>"}"
                  | "["<jsFunctionWithCallback>"]"
                  | "<"<jsPromise>">"

<variableName> ::= <alpha>
                 | <alpha><alphaNumeric>

<variableList> ::= <variableName>
                 | <variableName>","<variableList>

<float> ::= <integer>
          | <integer>"."<integer>

<verbosity> ::= "1"
              | "2"
              | "3"
              | "4"

<condition> ::= "true"
              | "false"
              | Any JavaScript statement evaluated to either "true" or "false"

<string> ::= <string><string>
           | <alphanumeric>
           | <space>
           | <symbol>

<alphanumeric> ::= <alphanumeric><alphanumeric>
```

```
                | <alpha>
                | <integer>

<alpha> ::= <letter><alpha>

<letter> ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m"
           | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"
           | "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M"
           | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"

<space> ::= " "

<newLine> ::= "\n"

<symbol> ::= "|" | " " | "!" | "#" | "$" | "%" | "&" | "(" | ")" | "*" | "+" | "," | "-"
           | "." | "/" | ":" | ";" | ">" | "=" | "<" | "?" | "@" | "[" | "\" | "]" | "^"
           | "_" | "`" | "{" | "}" | "~"

<integer> ::= <digit>
            | <digit><integer>

<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

<cliCommand> ::= Any valid CLI command

<jsArrowFunction> ::= Javascript arrow function

<jsNormalFunction> ::= Javascript function returning a value

<jsFunctionWithCallback> ::= Javascript function that has error-first-callback

<jsPromise> ::= Javascript promise
```

Since we are focusing on how to make the language more compact, we also support several shorthands so that a single process can be written in a single line.

For example, this process:

```
ins:
  month
  year
out: calendar
do: cal
```

can be written as:

```
|(month, year) -> cal -> calendar
```

or

```
|calendar <- cal <- (month, year)
```

We are implementing reverse arrow in CHIML to make it more intuitive for developers. So, typical assignment like `result = add(num1, num2)` can be mapped directly into `|result <- add <- (num1, num2)` without any need to reverse the order into `|(num1, num2) -> add -> result`.

Despite of it's nature as orchestration language, CHIML also support several control structures like `if`, `else`, and `while`. It is useful for prototyping since the developers doesn't have to deploy intermediary components for control structure.

For example, to get the content of page-1 to page-10 of a website, the developer can do the iteration:

```
ins: url
out: html
do:
  - i <-- 1
  - while: i < 11
    do:
      - |(url + '?page=' + i) -> curl -> htmlFragment
      - |(html + htmlFragment) --> html
      - |i <-- (i+1)
```

We also take further analysis and conclude that for similar cases, it is better to do the task in parallel, so that the first request doesn't block the later requests. Since most modern programming language supporting `map` and `filter`, we also take a similar approach. For the previous example, we can build a more efficient solution as follow:

```
ins: url
out: html
vars:
  indexes: [1,2,3,4,5,6,7,8,9,10]
do:
  - ins: indexes
    out: responses
    map: index
    into: response
    do: |(url + '?page=' + index) -> curl -> response

  - |(responses, '') -> {$.join} -> html
```

Map is more efficient since under the hood, each iteration will be done in a non-blocking manner.

Default Variables

There are some default variables in every CHIML script:

- `_chain_cwd`: String, current working directory of CHIML script physical location.
- `_process_cwd`: String, current working directory of CHIML script invocation location.
- `_error`: Boolean, error status.
- `_error_message`: String, error message.
- `_verbose`: Integer, verbosity level, default to `0`.
- `_ans`: Default output variable.
- `_runChain (chain, ...ins, callback)`: function to run other chain.
- `_maps (list, chain, callback)`: function to map an array into a new array. Under the hood, this will process each element in parallel.
- `_filter (list, chain, callback)`: function to filter an array into a new array. Under the hood, this will process each element in parallel.

Implementation

In order to parse and execute CHIML script, we have build a parser to evaluate the script and execute it on the fly. The javascript statements inside CHIML script are evaluated by using builtin Node.Js `vm` module. The global control flow is handled by using `neo-async`, a faster version of `async` module.

Result

Generally CHIML is slower than native Python/JavaScript solution.

Discsussion

CHIML serve well as orchestration language. However for control structure, the existance of intermediary components can help to boost performance. The best trait of CHIML is it's support for programming-in-large and programming-in-small. Eventough the control structure is still suffering for speed and performance, it serves well as prototyping tool. This mean that the developer can start orchestration solution in CHIML, then gradually do optimization.

Conclusion

TODO:

- Explain the acronym
- The options of the questionaires is better to be sorted descending
- Explain the `not-returned-questions`
- Explain the outlier
- Better to tabulate the questions
- Put some `open-ended-question` to get more data (like: why do you choose...?)
- Length of development, usage scope, LOC, speed, pre-requisites
- Use `child_process.fork` for map and filter