POLITECNICO MILANO 1863

Software Engineering II

Web-Based Front-End for the Zot Formal Verification Tool

June 15th, 2021

Farouk Jeffar

Supervised by:

Matteo Giovanni Rossi, Politecnico di Milano

Elisabetta Di Nitto, Politecnico di Milano

Table of Contents

I. Table of Figures

II. List of Abbreviations

- UI: User Interface
- UX: User Experience
- API: Application Programming Interface
- OS: Operating System
- JS: JavaScript
- IDE: Integrated Development Environment
- APP: Application

III. Introduction

1. What is Zot?

Zot is an agile and easily extendible bounded model checker. Zot adopts a multi-layered approach, in other words, its core makes use of a decidable predicative fragment of TRIO on top of CLTLB. Also, Zot supports different encodings of temporal logic by the utilization of plug-ins. Since plug-ins are straightforward, compact, adjustable, and extendible such organization encourages experimentation. Besides, Zot offers three usage modalities: Bounded satisfiability checking (BSC), Bounded model checking (BMC), History checking and completion (HCC).

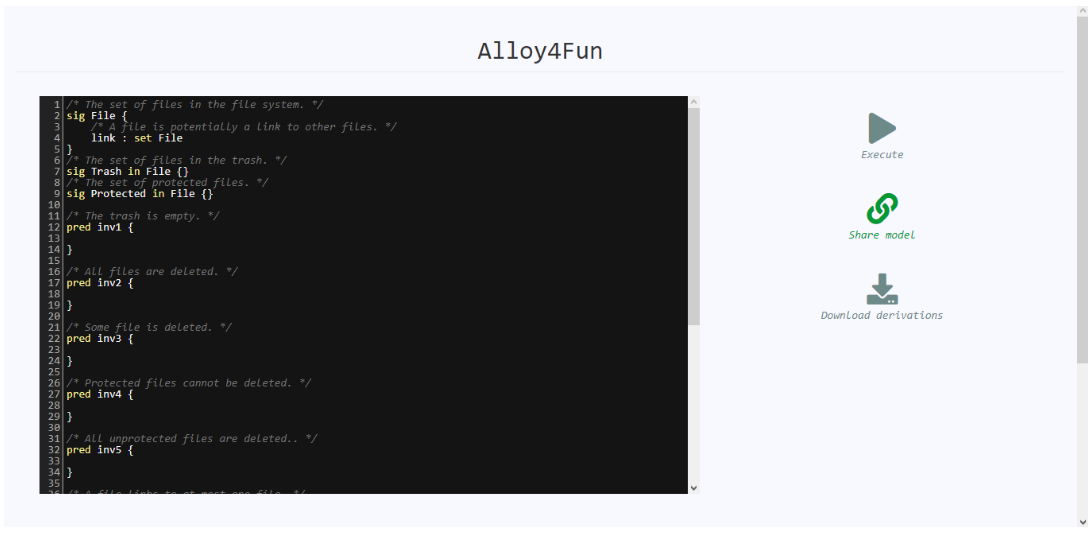
2. The need for Zot UI

To use Zot, one needs to install a common lisp compiler, then needs to setup a specific folder for the plug-ins and then use Zot, on Windows more steps are required. Also, Zot does not have a User Interface. This degrades the user experience as text should be edited elsewhere, and the terminal consumes line by line, so the user needs to keep copy pasting or write his/her code line by line. Additionally, no syntax highlighting is available nor user-friendly output delivery. This means that Zot needs an UI to enhance the user experience. By making a user-friendly interface, not only the user would work in good looking interface loaded with UX enhancing features, but also would not have to deal with the setup process, and what a perfect way to achieve this than a web application. Because the only pre-requisites for a web application are internet and a browser that supports JavaScript, and it does not get any simpler than this. Additionally, the user does not have to worry about his local resources as Zot runs on external servers and his local machine resources are preserved and the code is executed on performant servers.

3. Alloy and Z3 as an Example

a. Alloy

Alloy is a well-known language for software modeling. Its uses cover a wide range of applications like security caveats or even telephone switching. Alloy4fun is a project similar to ours where Alloy, a classical setup program is being ported to being used as a web application.

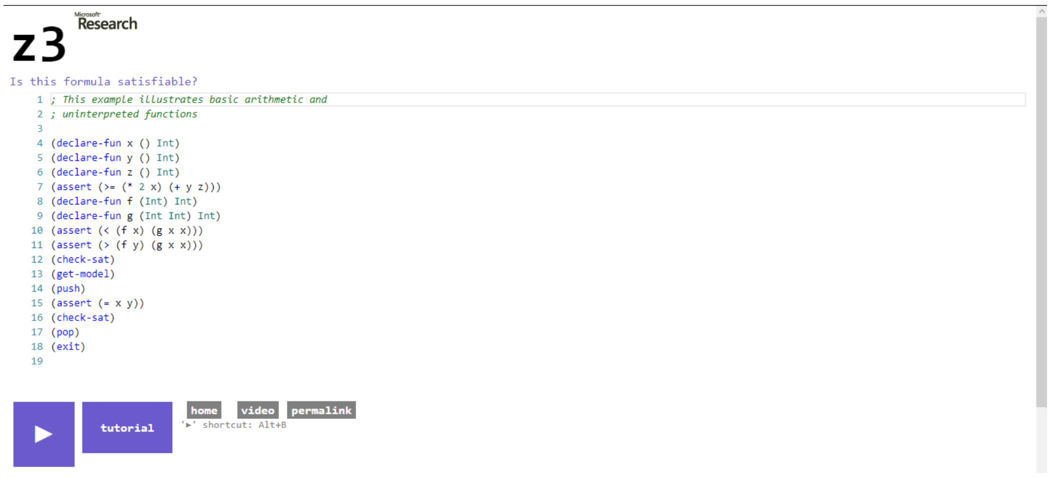


```
/* The set of files in the file system. */
sig File {
    /* A file is potentially a link to other files. */
    link : set File
}
/* The set of files in the trash. */
sig Trash in File {}
/* The set of protected files. */
sig Protected in File {}

/* The trash is empty. */
pred inv1 {

}

/* All files are deleted. */
pred inv2 {

}

/* Some file is deleted. */
pred inv3 {

}

/* Protected files cannot be deleted. */
pred inv4 {

}

/* All unprotected files are deleted.. */
pred inv5 {

}

```

Figure 1 http://alloy4fun.inesctec.pt, an Alloy Web Application

b. Z3

Z3 is a high-performance theorem prover. Z3 supports arithmetic, fixed-size bit-vectors, extensional arrays, datatypes, uninterpreted functions, and quantifiers.



```
; This example illustrates basic arithmetic and
; uninterpreted functions

(declare-fun x () Int)
(declare-fun y () Int)
(declare-fun z () Int)
(assert (>= (* 2 x) (+ y z)))
(declare-fun f (Int) Int)
(declare-fun g (Int Int) Int)
(assert (< (f x) (g x x)))
(assert (> (f y) (g x x)))
(check-sat)
(get-model)
(push)
(assert (= x y))
(check-sat)
(pop)
(exit)
```

Figure 2: https://rise4fun.com/Z3, a Z3 Web Application

IV. The Design

1. Introduction

As introduced in the previous section, Zot needed a modern UI, an improved UX, and to be accessible to anyone without an exhausting set up and with whatever OS one might have. Evidently, the solution is a web application that lets users run Zot through their web browsers seamlessly with no prerequisite apart from an internet connection and a browser capable of running JavaScript. Zot is coded in lisp, in term of web applications, lisp is not the most common nor the most convenient, so the initial idea was to rely on a JS front-end library such as React, and a backend based on a Python API that would execute Zot. Python was chosen because of the networking capabilities it has along with its ability to handle low-level operations such as process manipulation, and this will come-in handy as we shall see later. Concerning React, the choice was motivated by the fact that is one the best libraries if not the best to build high performing dynamic UI apps, meaning updates happen simultaneously with user actions. Also, the library has a great community compared to VueJS for example and is also constantly improved since it was released by Facebook as an open-source project. To sum up, one of the advantages of using such architecture is separation of concerns as each part is the independent which means it can be updated separately and opens to more horizons, like exposing the API to developers, the second benefit is getting the best of the front-end with React and getting the strength of Python multi-use capabilities (Networking and OS access).

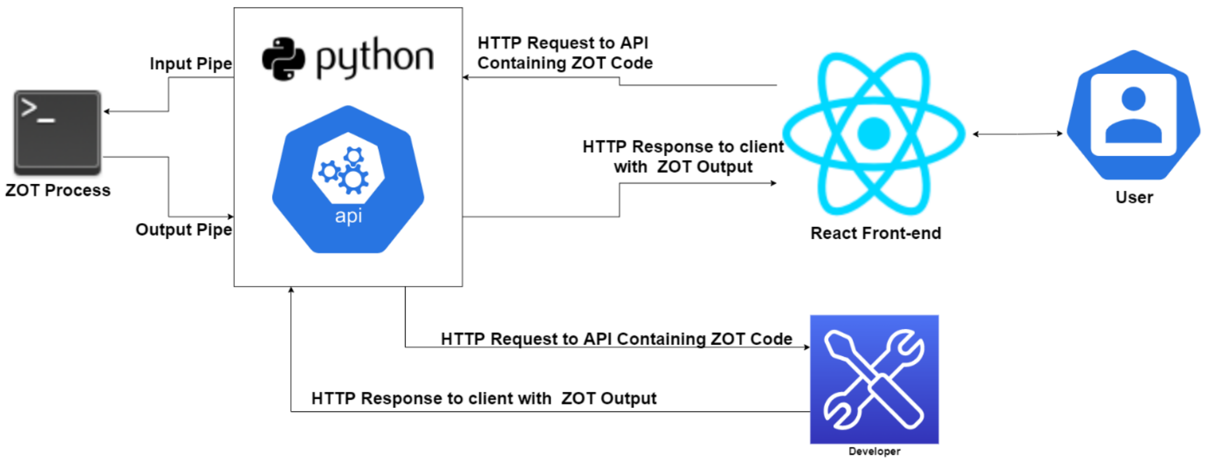

Input Pipe
python
HTTP Request to API Containing ZOT Code
ZOT Process
HTTP Response to client with ZOT Output
api
User
Output Pipe
React Front-end
HTTP Request to API Containing ZOT Code
HTTP Response to client with ZOT Output
Developer

Figure 3: General Architecture of Zot UI and the API

2. The Python API or the Backend

a. The API's Architecture

The goal of the backend is simple, execute Zot, however many considerations need to be handled and optimized. To execute Zot, two options are available to either interpret lisp within python by relying on Python lisp package or to execute it on the machine headlessly and get the output. The first option is not very optimal nor reliable as any limitation in the library to be used would limit our application and might even make some errors unfixable. Nonetheless, the second option to execute Zot by command is not very safe as it sends commands to the OS and these commands are written by a user, this is a security threat. Nevertheless, if we spawn a process of Zot this issue is resolved. Firstly, because if the process crashes or exits, then nothing more can be executed and secondly the process can be set to never spawn a shell. This means that even if a user finds a way to get around the first security measure, the shell will never be spawned. Thus, the only way to access OS variables or run another program is through Zot. This can be handled by blocking any Zot/lisp that calls the OS or tries to execute a program like “run-program”. If python detects such operation, no code shall be run and the request is simply blocked and the response states that no OS accessing plugin/module is permitted (This block is applied at the level of the font-end and the backend, this is done to save resources and reduce failing requests). The fact that the architecture adopted is split, in other words, each component is independent means that the API (the backend) can also be leveraged by users directly without the web application. This would be extremely useful for developers or researchers willing to run batch jobs of Zot or developing component that react dynamically to the Zot output. The API is meant to be open to everyone, which means that no authentication is applied to the API. It should also be noted that even if the API contains various security measures, server security remains necessary by the means of a firewall for example. To call the API, the route "/execzot" is used, this route accepts "POST" request with "cmd" as a parameter that contains the Zot codes to be executed. The Zot code is then checked to see if any illegal command is found, if not, then no flag is raised and execution can resume by sending each line from the input to the spawned Zot process, after the last command, the output of Zot is sent to another function that removes unnecessary text and cleans the output before it is returned as JSON to the client.

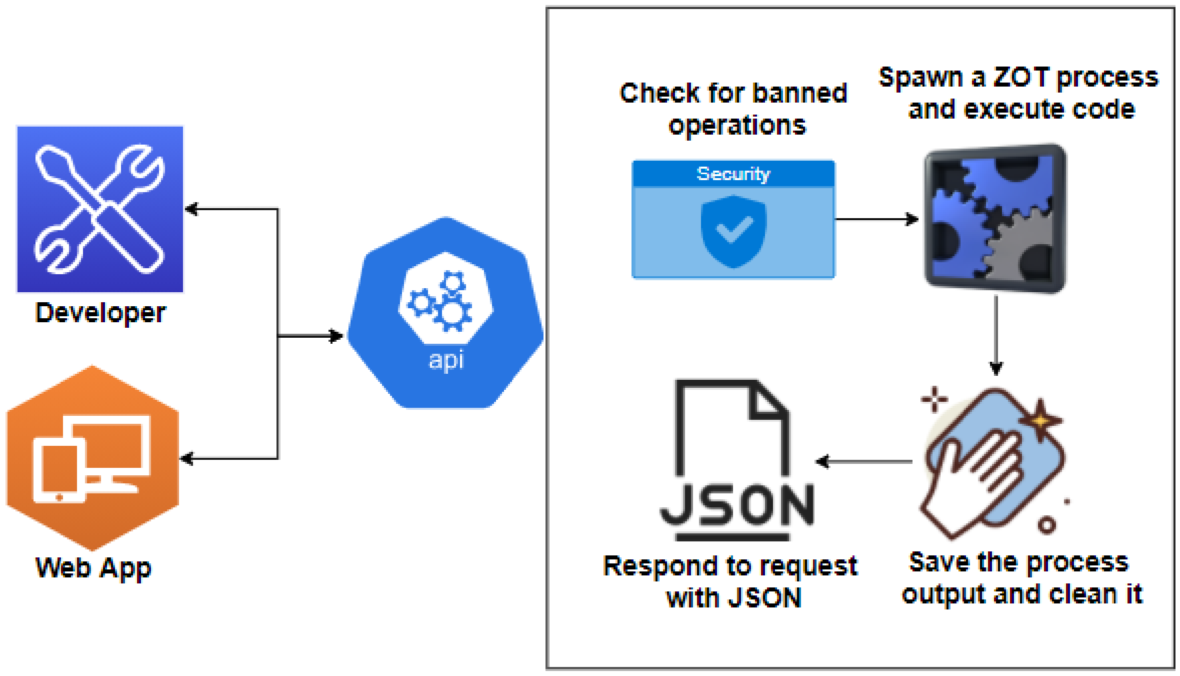

Developer
Web App
api
Check for banned operations
Security
Spawn a ZOT process and execute code
JSON
Respond to request with JSON
Save the process output and clean it

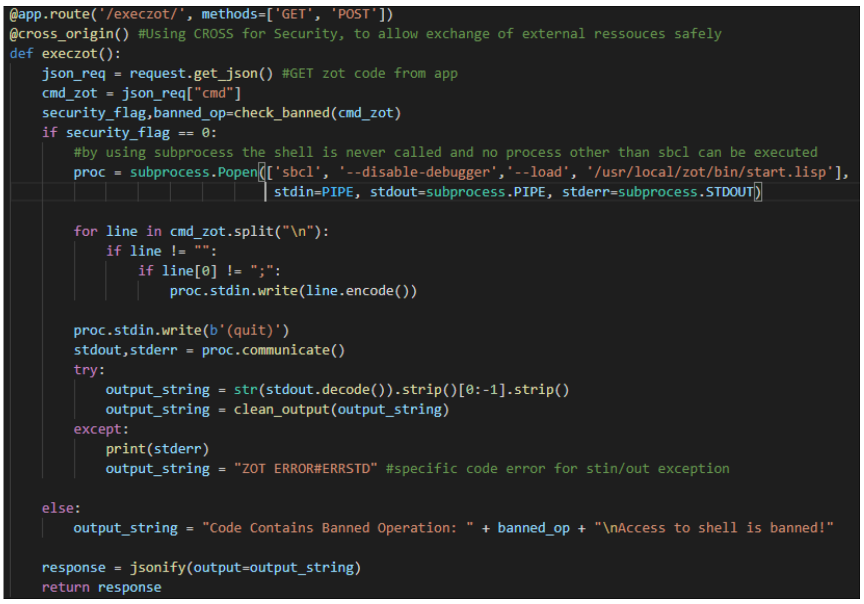
Figure 4: Graph of What happens When API is Called

b. The API’s Documentation

The API has three functions:

- execzot() : this function is routed to “/execzot” and expects a POST HTTP request where the “cmd” attribute is the user input. This function reads the request, calls check_banned() to check if any banned operation is present in the Zot code, and finally spawns a Zot process and return its output to the user.
- check_banned(): based on a list of banned operations, this function checks if the user code contains any banned operation, if it finds any, the function will return the integer 1 and the name of the banned operation, and trivially if nothing is found the function returns 0 and an empty string.
- clean_output(): when the Zot output is returned from the Zot process, execzot() calls this function to parse and clean the output which is then returned in the HTTP response.

Please note that the API uses Python Flask which is a Python micro web framework.



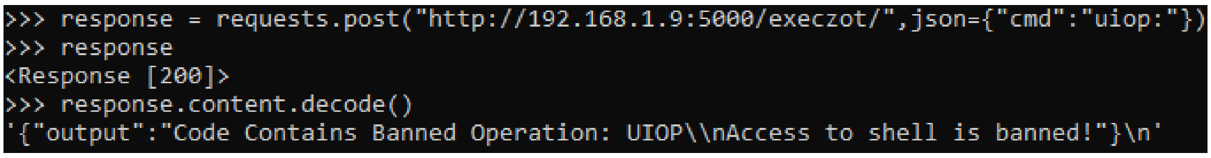
```python
@app.route('/execzot/', methods=['GET', 'POST'])
@cross_origin() #Using CROSS for Security, to allow exchange of external ressouces safely
def execzot():
    json_req = request.get_json() #GET zot code from app
    cmd_zot = json_req["cmd"]
    security_flag,banned_op=check_banned(cmd_zot)
    if security_flag == 0:
        #by using subprocess the shell is never called and no process other than sbcl can be executed
        proc = subprocess.Popen(['sbcl', '--disable-debugger','--load', '/usr/local/zot/bin/start.lisp'],
                                stdin=PIPE, stdout=subprocess.PIPE, stderr=subprocess.STDOUT)

        for line in cmd_zot.split("\n"):
            if line != "":
                if line[0] != ";":
                    proc.stdin.write(line.encode())

        proc.stdin.write(b'(quit)')
        stdout,stderr = proc.communicate()
        try:
            output_string = str(stdout.decode()).strip()[0:-1].strip()
            output_string = clean_output(output_string)
        except:
            print(stderr)
            output_string = "ZOT ERROR#ERRSTD" #specific code error for stin/out exception

    else:
        output_string = "Code Contains Banned Operation: " + banned_op + "\nAccess to shell is banned!"

    response = jsonify(output=output_string)
    return response
```

Figure 5: Code Snippet of Main function of the API

To call the API manually, a request should be sent to the API address and the route “execzot”, in other words, “HTTP://ZOTADDRESS.XX/execzot/”.

Additionally, the POST request should have a “cmd” attribute which contains the zot code, for example: {“cmd”:” (-P- hello)”}.

The response is a JSON string that can be parsed easily by reading the “output” attribute. Below is manual call to the API that shows the call and the raw response.



```
>>> response = requests.post("http://192.168.1.9:5000/execzot/",json={"cmd":"uiop:"})
>>> response
<Response [200]>
>>> response.content.decode()
'{"output":"Code Contains Banned Operation: UIOP\\nAccess to shell is banned!"}\n'
```

Figure 6: Example of a Python call to the API and the Raw Output

3. React as a Front-end Solution

a. Introduction

As mentioned earlier, React is one of the best front-end libraries at the moment and its functioning fits perfectly our architecture as it can perfectly operate on top of an API. The design of the app should facilitate the user experience not only by providing a ready-to-use Zot but also to make the user experience richer and easier. The app is one page website, this means that even the user sends new code, change theme, or anything else the page does not refresh and the JavaScript running in the browser takes care of updating the content.

b. User Manual

The layout of the app is divided into two areas aligned horizontally. One side serves the user input and the other serves the user output. The input side is a text area with line numbers, theme, and syntax highlighting, this text area is in fact a library called "codemirror" which provides IDE styled text areas for JS libraries and frameworks. The other side is the output of Zot, the latter is shown as three accordions named: Header, Outcome, and Trace.

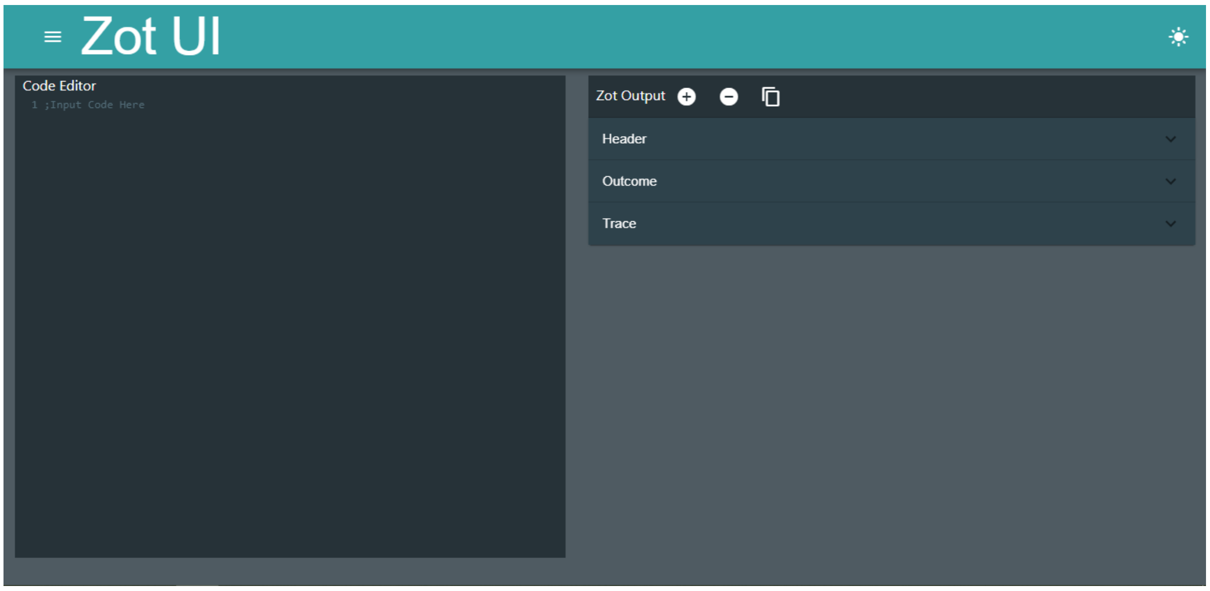
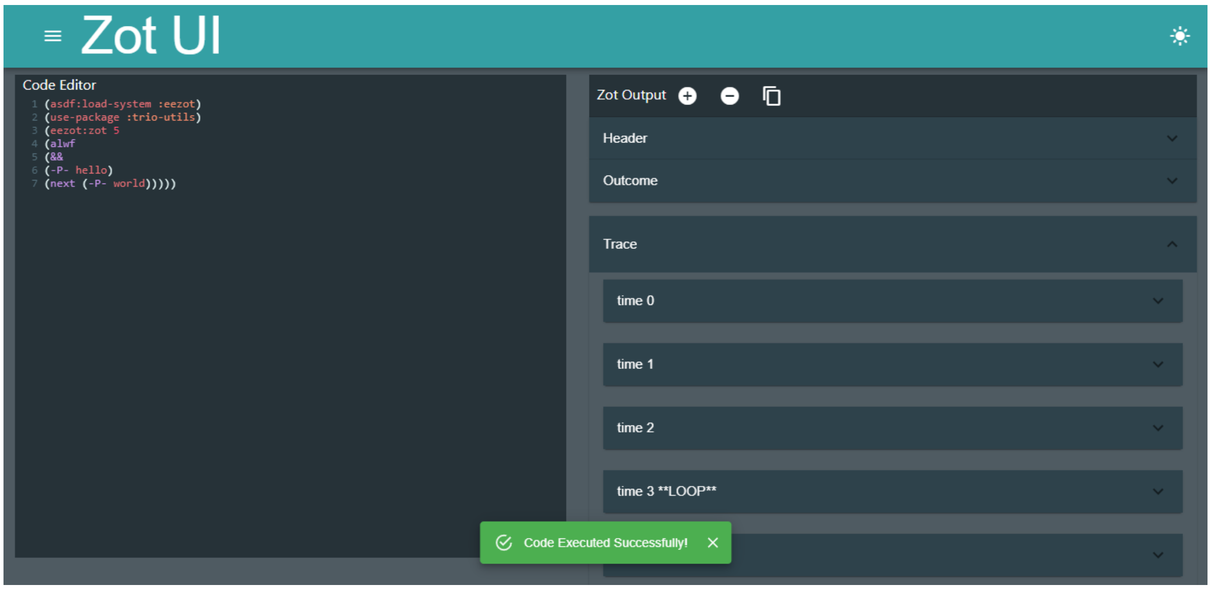


Figure 7: UI Design of the Web Application

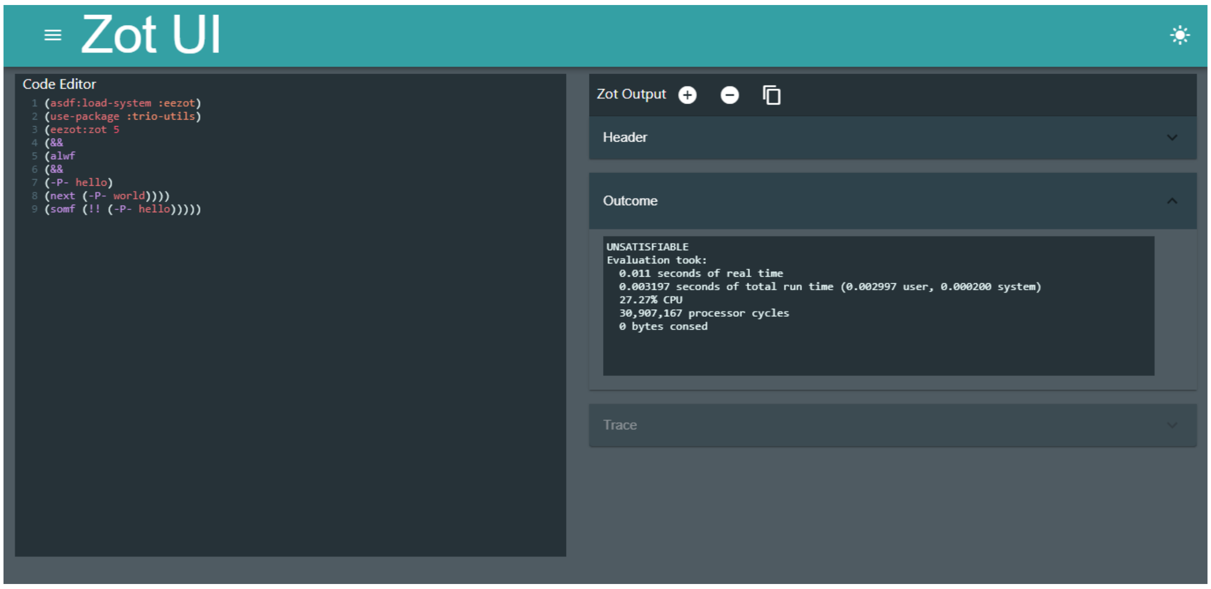
Once the output is returned from the API, it is parsed and separated into these three accordions. The trace accordion is dynamic, this means that it has embedded accordions that are created dynamically depending on the loop in the Zot code (the **LOOP** is also marked in the label of the accordion where it happens). For the convenience of the user, an expand all and retract all buttons are available, a custom-made function was implemented to extend/retract all because the API shipped with the accordion component did not handle such case and if used would break the component. Also, because the output is split across multiple parts it not easy to copy it, so a copy button was added to send the output to the machine's clipboard. It should also be noted that in case of unsatisfiability the trace accordion is disabled, also in case of error, both outcome and trace are disabled, and header shows the whole output to provide the user with further details. The next component is the appbar, is has two buttons: a menu button and theme switcher toggle. By default, the app is shown in dark mode, if the user desires he can press on the theme toggle to switch the theme to a light mode, this change impacts the whole app including the text areas. Next is the menu, the latter is a floating menu which appears when the menu icon is pressed. The menu contains the following buttons:

- Run: runs the code by sending the input to the API and display output.



```
(asdf:load-system :eezot)
(use-package :trio-utils)
(eezot:zot 5
(alwf
(&&
(-P- hello)
(next (-P- world)))))
```

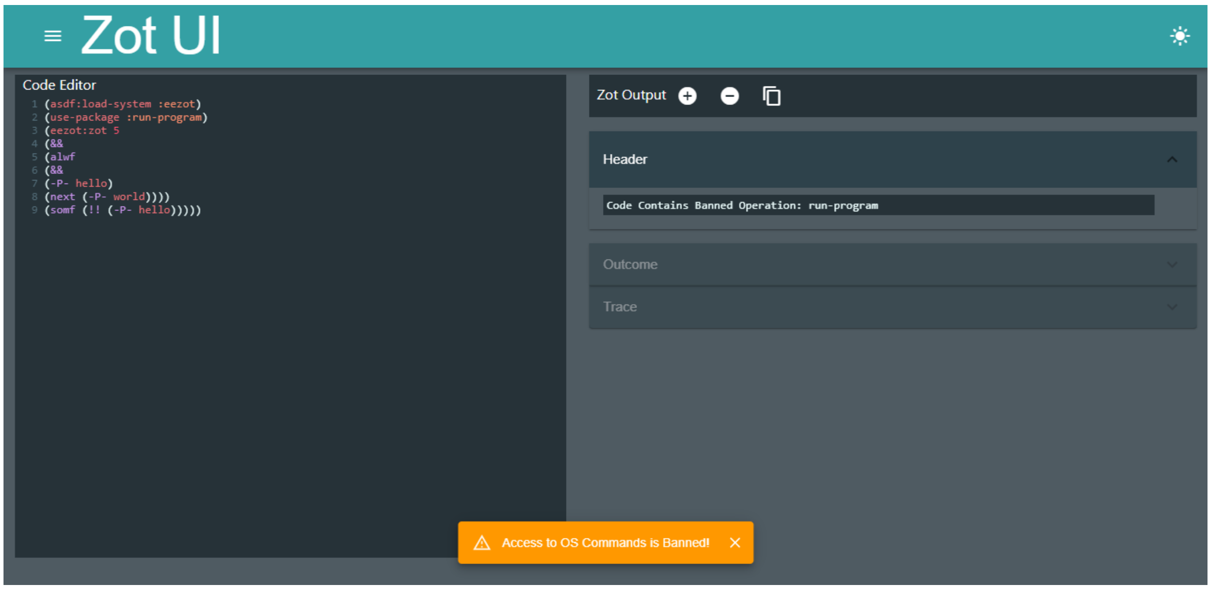
Figure 8: Satisfiable Code Execution Example



```
(asdf:load-system :eezot)
(use-package :trio-utils)
(eezot:zot 5
(&&
(alwf
(&&
(-P- hello)
(next (-P- world))))
(somf (!! (-P- hello)))))
```

```
UNSATISFIABLE
Evaluation took:
  0.011 seconds of real time
  0.003197 seconds of total run time (0.002997 user, 0.000200 system)
  27.27% CPU
  30,907,167 processor cycles
  0 bytes consed
```

Figure 9: Unsatisfiable Code Execution Example

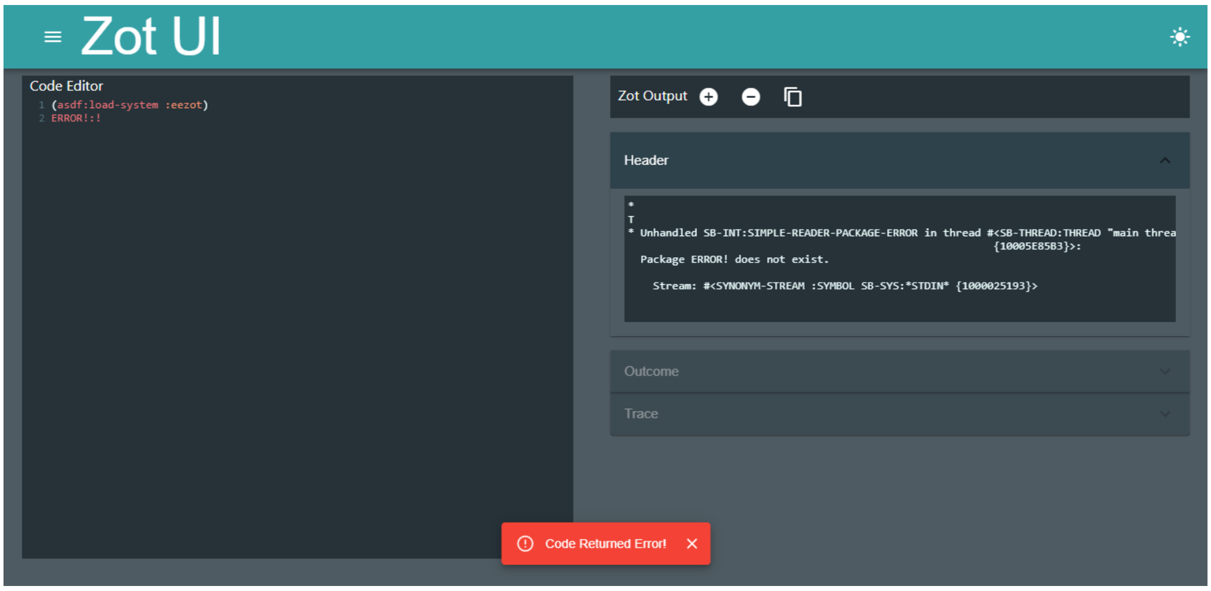


```
(asdf:load-system :eezot)
(use-package :run-program)
(eezot:zot 5
(&&
(alwf
(&&
(-P- hello)
(next (-P- world))))
(somf (!! (-P- hello)))))
```

```
Code Contains Banned Operation: run-program
```

Access to OS Commands is Banned!

Figure 10: Banned Operation in Code Execution Example



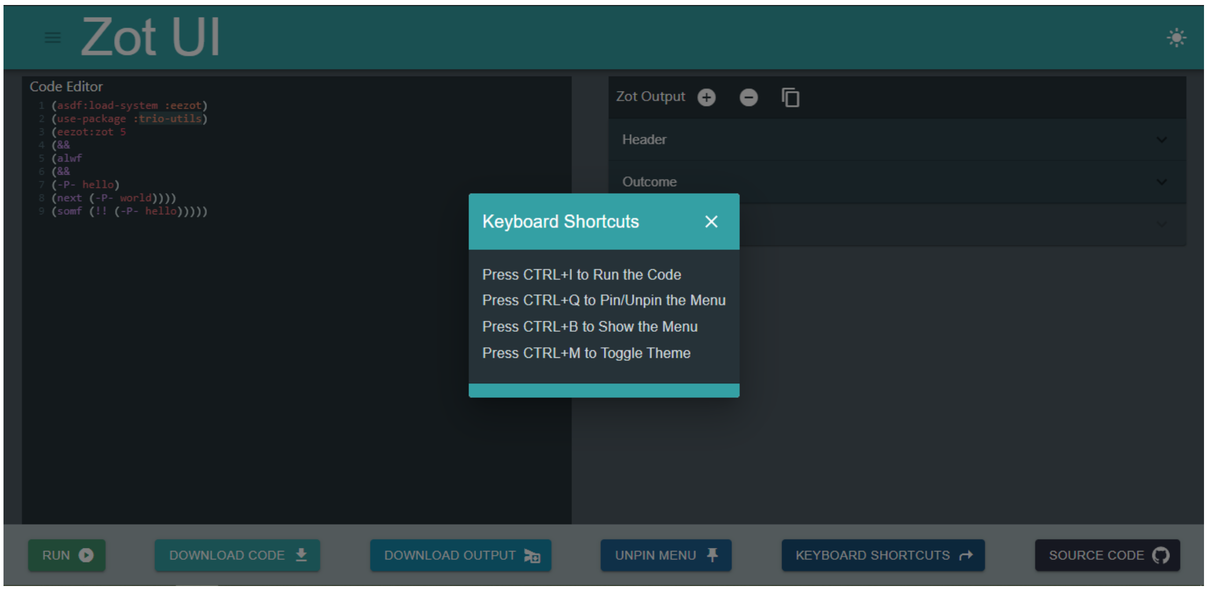
```
(asdf:load-system :eezot)
ERROR!:!
```

```
*
T
* Unhandled SB-INT:SIMPLE-READER-PACKAGE-ERROR in thread #<SB-THREAD:THREAD "main threa
                                                           {10005E85B3}>:
  Package ERROR! does not exist.

    Stream: #<SYNONYM-STREAM :SYMBOL SB-SYS:*STDIN* {1000025193}>
```

Figure 11: Error in Code Syntax Execution Example

- Download input: downloads the input of the user as text file.
- Download output: downloads the output of Zot as text file.
- Pin/Unpin The menu: the menu is permanent or floating.
- Keyboard Shortcuts: shows the keyboard shortcuts available through the app. The shortcuts were added so that the user does not have to use the menu every time to run the code, pin the menu, etc. The keyboard shortcuts are not implemented using the classical JS method as it slows performance drastically, however, a react component was used to handle these shortcuts. Keyboard shortcuts are available everywhere, even inside the textarea.

]



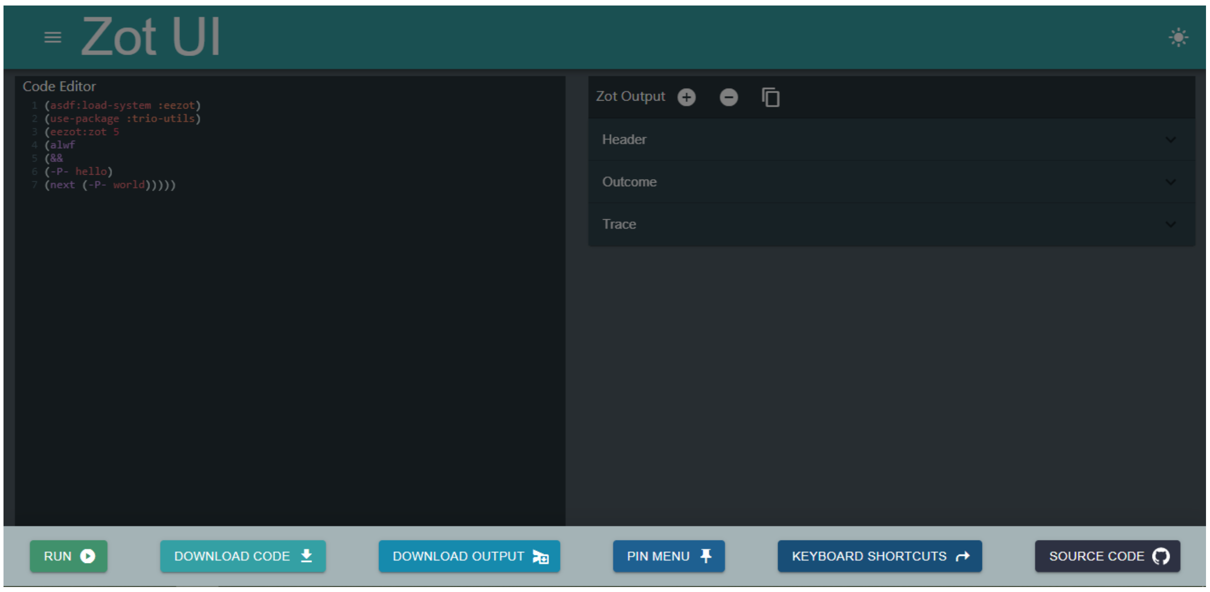
```
(asdf:load-system :eezot)
(use-package :trio-utils)
(eezot:zot 5
(&&
(alwf
(&&
(-P- hello)
(next (-P- world))))
(somf (!! (-P- hello)))))
```

Keyboard Shortcuts

Press CTRL+I to Run the Code
Press CTRL+Q to Pin/Unpin the Menu
Press CTRL+B to Show the Menu
Press CTRL+M to Toggle Theme

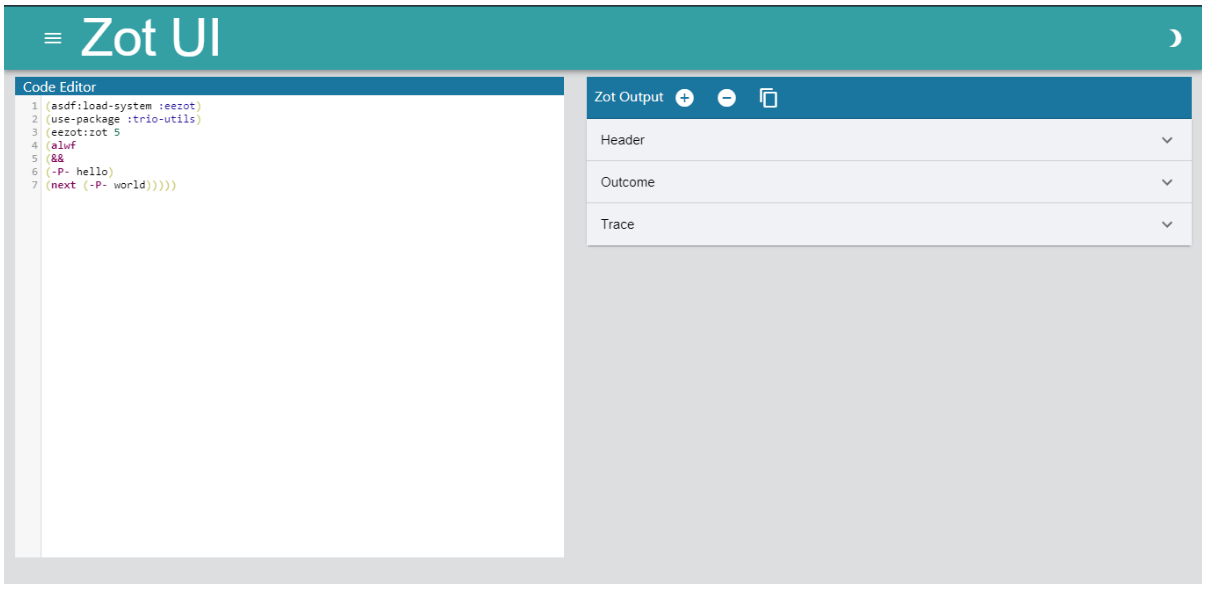
Figure 12: Keyboard Shortcuts Used in the Application

- Github source: opens a new tab with Github page of Zot.



```
(asdf:load-system :eezot)
(use-package :trio-utils)
(eezot:zot 5
(alwf
(&&
(-P- hello)
(next (-P- world)))))
```

Figure 13: Unpinned Menu



```
(asdf:load-system :eezot)
(use-package :trio-utils)
(eezot:zot 5
(alwf
(&&
(-P- hello)
(next (-P- world)))))
```

Figure 14: Light Mode

Additionally, when the user runs the code, depending on the result a floating message appears to signal the user of the state of the results, the messages shown are success, error, and warning in case of illegal operation. To further improve performances, after clicking on run, the input is checked to see if any illegal plugin or function is called, if true, it blocks the request from being sent and shows a floating message, this means that the blockage is at the level of the client's browser and our API is not flooded with unnecessary requests. Finally, to add Zot keywords to syntax highlighting, the codemirror module was modified and the keywords were simply added to the array list of the common lisp syntax.

c. The Front-end's Documentation

The main functions of the front-end are as follows:

- sendApiRequest (): This function sends an HTTP request to the API, sends the response to the parsing functions, displays the output and triggers animations related to the code execution.
- makeCollapsibleTrace (): This function gets the output from API transferred by the sendApiRequest function and parses into a json format which is linked to a component

in the HTML fragment of the code. This is what dynamically creates the time accordions in the Trace part of the output.

- downloadTxtFile (mode): Download a text file of either the input or output depending on the action requested by user, the mode variable is what determines what should be downloaded.
- togglehidemenu (): When called, it reverses the state of the menu, in other words, if the menu is pinned then the function unpins it and vice-versa.
- toggleExpand (accordionExpand): This function either extends all accordions or retracts them, this depends on the button clicked which directs what value is given to accordionExpand which is a Boolean.
- copyOutput (): This function copies the whole Zot output to the clipboard and then displays a floating message saying it was successfully copied or not.
- toggledarkmode (): This function switches the lighting theme of the app from dark to light and vice-versa. By default, the app is launched in dark mode.

V. Installation

a. API

To deploy the API, the first step is to install ZOT and make sure that it runs perfectly by trying this command in the terminal: “*sbcl --disable-debugger –load /usr/local/zot/bin/start.lisp”.* To install Zot, please refer to the Zot documentation linked in the bibliography below.

After Zot is ready and running, the machine should have Python installed. More specifically, the Python version where the API was developed is Python 3.9.2 but anything greater than Python 3.7 should be working fine. The Python dependencies to be installed are:

- Requests, install by running “pip install requests” in the terminal.
- Flask, install by running “pip install Flask” in the terminal.
- CORS for Flask, install by running “pip install Flask-Cors”

To start the API just run the script “ZotAPI.py”.

b. React

To deploy the React app, first make sure that Nodejs is installed on the machine, the version used in the development phase was Nodejs v14.17.0.

To install the dependencies, the command “npm install” should be executed from the root of the project which is where the file “package. json” is located. This will create a ”node_modules” folder within the project’s directory. Then replace the codemirror’s lisp file(”zot-ui\node_modules\codemirror\mode\commonlisp\commonlisp.js”) with the one provided with the project to add the ZOT keywords to the syntax highlighting of the codemirror component. However, if we use the “node_modules” folder shipped with the source code, then there is no need to run the “npm install” command or to replace the codemirror component files as everything is ready.

To start the server, execute the command “npm start” from the root of the project folder.

VI. Conclusion

To sum up, building a web application on top of Zot opened many opportunities not only for a typical user but also for developers. Users can now use Zot without having to install libraries, programs, or face any other issue during the setup. Moreover, a modern UI with syntax highlighting, themes, etc. is available on any browser. On another note, developers can connect to the Zot API and run all sort of Zot codes with no hassles and without the necessity of extra resources. The fact that the API spawns a process makes the load-balancing tangible since monitoring consumed resources can provide accurate heuristics to determine how the requests are to be processed. Additionally, load balancers can cover the app and the API separately. This means that a reverse-proxy on top of the instances of the API can balance the load and route the requests accordingly.

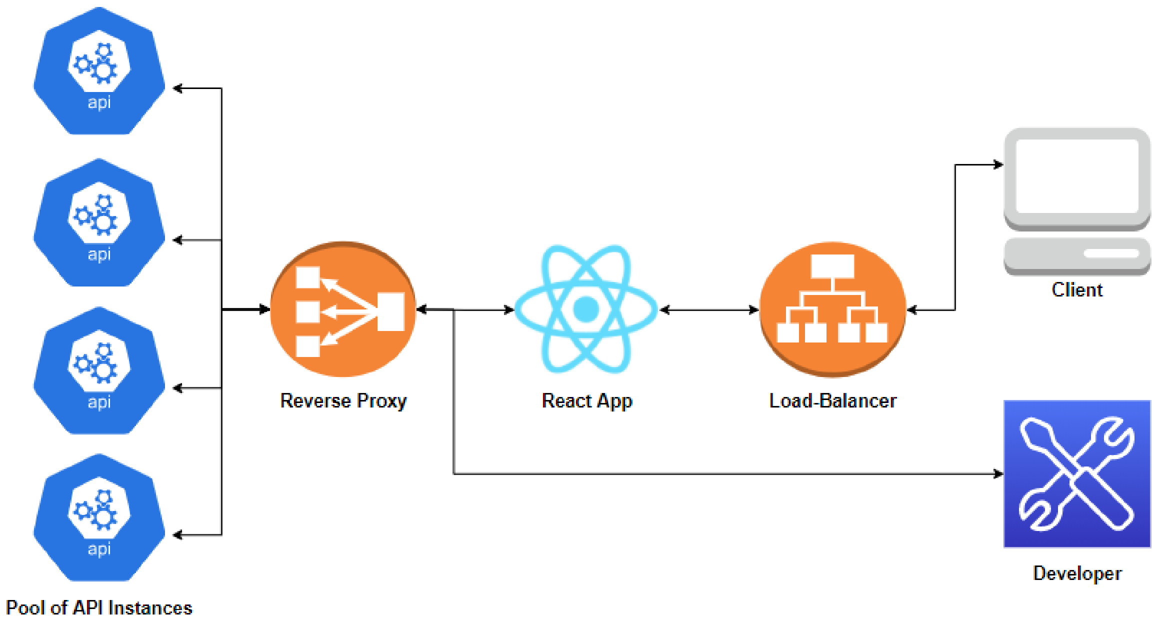


Figure 15: Deployment Architecture with Load-Balancers

VII. Bibliography

- Alloy. alloytools.org. (n.d.). http://alloytools.org/.
- Politecnico di Milano. (2014, July 9). A User's Guide to Zot. https://github.com/fm-polimi/zot/blob/master/doc/zot-man.pdf.
- Z3 @ rise4fun from Microsoft. (n.d.). https://rise4fun.com/Z3.