



Masters Project Final Report

March 2017

Project Title	Enabling an Authentication Mechanism for Docker Remote API		
Student Name	A. I. D. K. Seneviratne		
Registration No. & Index No.	2014MIS020 14770201		
Supervisor's Name	Dr. D. A. S. Atukorale		
Please Circle the appropriate	Masters Program	Type	
	MIS	Research	Implementation
For Office Use Only			

Enabling an Authentication Mechanism for Docker Remote API

A. I. D. K. Seneviratne
2017



Enabling an Authentication Mechanism for Docker Remote API

**A dissertation submitted for the Degree of Master of
Science in Information Security**

**A. I. D. K. Seneviratne
University of Colombo School of Computing
2017**



Declaration

The thesis is my original work and has not been submitted previously for a degree at this or any other university/institute.

To the best of my knowledge it does not contain any material published or written by another person, except as acknowledged in the text.

Students Name: A. I. D. K. Seneviratne

Signature:

Date:

This is to certify that this thesis is based on the work of

Mr. A. I. D. K. Seneviratne

under my supervision. The thesis has been prepared according to the format stipulated and is of acceptable standard.

Certified by:

Supervisor Name: Dr. D. A. S. Atukorale

Signature:

Date:

Abstract

Docker is an operating system level virtualization mechanism on Linux which allows deploy and run applications inside software containers. Software containers provides lightweight and faster delivery of applications by separating applications from the infrastructure. Furthermore it helps to provide better hardware utilization than virtual machines. Docker has now become one of the fast growing industry practices in the field of software virtualization. Docker architecture allows users to interact with the Docker daemon and other Docker components via three different types of mechanisms, which allows developers and system administrators manage dockerized resources effective and efficient way. One approach is making the Docker daemon listening on a TCP port and allows users to make requests through an API. This API is known as Docker Remote API.

One of the major drawbacks of above mentioned mechanism is Docker hasn't implement a flexible request authentication mechanism for the requests which are pointed to Remote API which makes it problematic in accessing Docker via the Remote API. This project has focused on introducing and implementing a token based request authentication mechanism to the Docker Remote API when the Docker daemon listening on a TCP port. The new implementation makes Docker Remote API accessible only via a proxy server. The proxy server act as a reverse proxy for the user requests which are directed to Docker Remote API. An authentication server also been in place with the new implementation to issue and validate the access tokens, which are required to access the Docker Remote API via the proxy server.

With the new implementation, users and applications which are eligible to perform Docker operations via Remote API should have proper secret credentials. The access tokens can be obtained by submitting those credentials to the authentication server. Every request which is made to the Docker Remote API should have a valid access token. The proxy server validates the access token of the request with the authentication server and pass the request to the Docker Remote API or reject the request. The final evaluation shows that, though the new implementation has introduced some latency to the request-response routing, it has achieved the aspects of request authentication without deviating the responses.

Acknowledgments

First of all I greatly thankful to my supervisor, Dr. Ajantha Atukorale, Senior Lecture and Deputy Director of University of Colombo School of Computing for his tireless support, guidance, patience, advices and prompt feedback on every step of the project.

Secondly I like to thank the academic staff of University of Colombo School of Computing for their much valued feedbacks and suggestions which they provided me on the presentations.

The management of thinkcube Systems Pvt. Ltd. deserves special thanking for their support that they gave to me within the project timeline to balance the academic works with my professional duties.

Then I like to thank, my fellow colleagues and other postgraduates of my batch in the Master of Science in Information Security degree program for their suggestions and feedbacks which they provided in the project.

I also like to thank the people who carried out previous researches which were regarded to the project.

Last but not least, I like to thank Dr. Manjusri Wickramasinghe, the coordinator of the Individual Project course module, for the support, encouragement and for making sure I worked on the project throughout the year.

Table of Contents

Declaration.....	ii
Abstract.....	iii
Acknowledgments	iv
Table of Contents	v
List of Figures.....	viii
List of Tables.....	ix
List of Abbreviations	x
CHAPTER 01 – Introduction	12
Introduction to Docker.....	12
Docker Architecture (Problem Domain).....	12
Docker Host.....	13
Docker Client	13
Docker Registries	13
Docker Images Vs Docker Containers	13
Problem.....	13
Importance of the Problem and the Security	14
Scope.....	14
CHAPTER 02 - Literature Review.....	16
Accessing Docker remotely	16
Introduction	16
API endpoints.....	16
API authentication mechanisms.....	17
Introduction	17
Kerberos	18
OAuth.....	19
Research on similar works	20
Docker Registry 2 Authentication Server (github - cesanta/docker_auth)	20
Improving Docker.....	21
Introduction	21
Basic Contribution.....	21
Advanced Contribution	22
Conclusion	24
CHAPTER 03 – Design.....	26
Introduction.....	26
Authentication Server	26
Components and Technologies used	26

Approaches Considered	29
Changing Docker Source	29
Run Docker with a proxy server	29
Conclusion	30
CHAPTER 04 – Implementation.....	32
Introduction.....	32
User Store	32
Authentication Server	32
Overview	32
Authentication Server Initiation	33
Authentication Server Functionality	34
Proxy Server	41
Overview	41
Proxy Server Initiation	42
Proxy Server Functionality.....	42
Self-Signed Certificate Generation	46
Conclusion	46
CHAPTER 05 – Evaluation.....	48
Introduction.....	48
Test Environment.....	48
Evaluation procedure	48
Evaluation Results	49
GET /info.....	49
GET /containers/json.....	50
GET /version	51
GET /images/json.....	52
GET /images/search?term=#keyword	53
GET /containers/{id}/json.....	54
POST containers/create	55
POST /containers/{id}/start	56
POST containers/{id}/stop.....	57
DELETE containers/{id}	58
Security Evaluation.....	59
Evaluation Conclusions	62
CHAPTER 06 – Conclusions	64
Challenges Faced	64

Future Works	64
Summary	64
Appendixes	65
Appendix 1 – Other test results regards to the evaluation	65
References	67

List of Figures

Figure 1: Docker Architecture [1]	12
Figure 2: Default user request response flow	13
Figure 3: Kerberos authentication [10].....	19
Figure 4: Docker Registry Authentication [13].....	20
Figure 5: Docker Basic Contribution Workflow [17]	22
Figure 6: Advanced Contribution Workflow [19].....	23
Figure 7: Access token generation workflow	27
Figure 8: Access token verification workflow	28
Figure 9: Request flow after changing Docker source for request authentication	29
Figure 10: Request flow when running Docker with a proxy server.....	30
Figure 11: MySQL table for user credentials	32
Figure 12: File and directory structure of the authentication server.....	33
Figure 13: Access token generation and Granting.....	36
Figure 14: Issuing cURL request to obtain access token.....	40
Figure 15: Access token validation flow	40
Figure 16: Issuing cURL command to validate an access token.....	40
Figure 17: File and directory structure of doc-proxy	41
Figure 18: A request made to the /info endpoint directly by using Postman REST client.....	49
Figure 19: A request made to the /info endpoint via proxy server by using Postman REST client	49
Figure 20: A request made to the /containers/json endpoint directly by using Postman REST client	50
Figure 21: A request made to the /contianers/json endpoint via proxy server by using Postman REST client	51
Figure 22: A request made to the /version endpoint directly by using Postman REST client .	51
Figure 23: A sample request made to the /version endpoint via proxy server by using Postman REST client	52
Figure 24: A request made to the /images/json endpoint directly by using Postman REST client	52
Figure 25: A request made to the /images/json endpoint via proxy server by using Postman REST client	53
Figure 26: A request made to the /images/search endpoint directly by using Postman REST client	53
Figure 27: A request made to the /images/search endpoint via proxy server by using Postman REST client	54
Figure 28: A sample request made to the /contains/{id}/json endpoint directly by using Postman REST client.....	55
Figure 29: A request made to /containers/{id}/json endpoint via proxy server by using Postman REST client.....	55
Figure 30: A request made to the /containers/create endpoint directly by using Postman REST client	56
Figure 31: A request made to /containers/create endpoint via proxy server by using Postman REST client	56
Figure 32: A request made to the /containers/{id}/start endpoint directly by using Postman REST client	57
Figure 33: A request to /containers/{id}/stop endpoint via proxy server by using Postman REST client	57

Figure 34: A sample request made to the /containers/{id}/stop endpoint directly by using Postman REST client.....	58
Figure 35: A sample request to /containers/{id}/stop endpoint via proxy server by using Postman REST client.....	58
Figure 36: A request made to the /containers/{id} endpoint directly by using Postman REST client	58
Figure 37: A sample to /containers/{id} endpoint via proxy server by using Postman REST client	59
Figure 38: Obtaining of access token by submitting valid credentials.....	65
Figure 39: Checking access token validity with a valid access token.....	65
Figure 40:: Checking an access token's validity with an invalid access token.....	65
Figure 41: Request sent to the proxy server with an invalid access token	65
Figure 42: Sending request to an invalid endpoint of Docker Remote API.....	65
Figure 43: Tagging an image.....	65
Figure 44: Creating an Image by pulling.....	66
Figure 45: Creating an image by pulling.....	66
Figure 46: Attach an image	66

List of Tables

Table 1: Important Docker Remote API endpoints	17
Table 2: Analysis of the authentication server's compliance with OAuth2.....	61

List of Abbreviations

- API – Application Programming Interface.
- TCP – Transmission Control Protocol.
- IP – Internet Protocol.
- SSL – Secure Socket Layer.
- REST – Representational State Transfer.
- URL – Uniform Resource Locator.
- URI - Uniform Resource Identifier.
- HTTP – Hyper Text Transport Protocol.
- RFC – Request for Comment.
- JSON – JavaScript Object Notation.
- LTS – Long Term Support.
- FQDN – Fully Qualified Domain Name.

CHAPTER 01 – Introduction

CHAPTER 01 – Introduction

Introduction to Docker

Docker is an open source project which enables virtualization of resources. It allows deployment of applications inside software containers. It provides additional layer of abstraction and automation of operating-system-level virtualization on Linux. Docker containers facilitate to wrap up a piece of software in complete file system which contains everything it needs to run including code, system tools, libraries and other components[1]. When consider about the Docker architecture, its major intentions can be listed as follow [1],

1. Deliver the user applications faster. Docker helps users to write, test and deploy the code faster. Hence it allows to minimize the time gap between the writing code and running code.
2. Separate applications from the infrastructure and treat infrastructure as a managed application.
3. Provides a way to isolate and run different applications in different containers which allows run many containers in the same host at the same time.
4. Maximize the hardware utilization by providing lightweight nature to the containers.

There are two major components of Docker system [1].

1. Docker Platform - An open source platform for containerization of applications
2. Docker Hub - Software as a Service cloud which facilitate sharing and managing of docker containers.

Docker Architecture (Problem Domain)

Docker operates in a client-server architecture. As illustrated in the [Figure 1](#), main components of the Docker architecture are **Docker Client** and the **Docker Host**. Apart from those there are **Docker Registries** which facilitate to this architecture. The Docker registries hold the Docker images to be pulled. The Docker Client and the Docker Host can be run either same machine or the Docker client can be connected to a remote Docker Host. The Docker Client and the Docker Host can interact with each other via a socket connection or through a RESTful API [1].

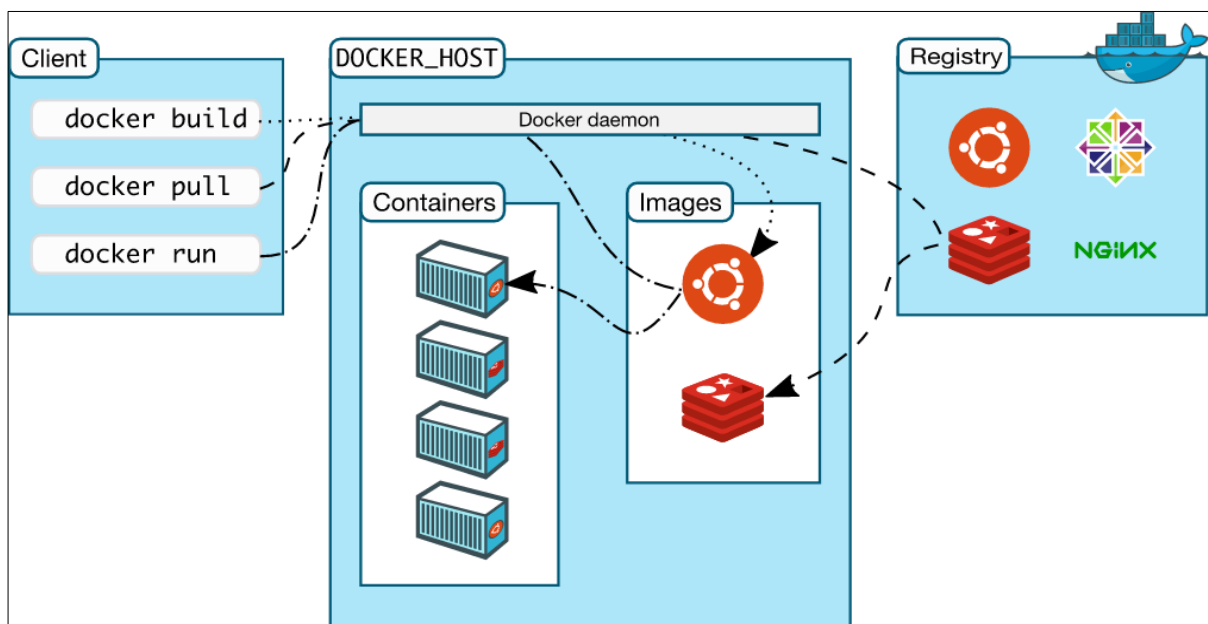


Figure 1: Docker Architecture [1]

Docker Host

Docker host is the core of the Docker architecture. Docker host holds all the user contains and pulled imaged from the Docker registry. Docker Daemon runs inside every Docker host which does the all the major tasks such as building, running and distributing the Docker containers [1].

Docker Client

Docker Client can be considered as the user interface of the Docker architecture. User cannot directly interact with the Docker Daemon. Only through Docker Client that can be achieved. Docker Client accept valid commands from the user and interact with the Docker Daemon accordingly [1].

Docker Registries

Docker Registries hold the images. These images can be either public or private which users can pull and push images. **Docker Hub** is a public Docker registry which holds a vast collection of images [1].

Docker Images Vs Docker Containers

A Docker image can be considered as a sample structure for create Docker containers. As an example a Docker image can contain a Unix-like operating system. A user can pull that image from a registry and create containers from it. Any changes can be done to the containers but cannot be changed the original image. The simplest way to build a Docker image is create a **Dockerfile** and make it read by Docker. A Dockerfile is simple text document that contain all the commands that necessary to build a Docker image. The other way to build an image is to commit container changes as a new image [1].

A Docker container can be considered as a directory structure which holds everything that needs by an application to be ran. All the Docker containers are created by using a particular Docker image. Each container can be started, run, hold and moved independently [1]. Furthermore Docker containers can be exported to create new Docker images. These new Docker images can be pushed to Docker hub with a different tab.

Problem

In the previous section, it has described the Docker architecture. In that section it has mentioned that the user can interact with **Docker Daemon** through a RESTFUL API. As illustrated in [Figure 2](#) illustrates Docker daemon can listen for **Docker Remote API** requests via three different types of sockets. One way to achieve this is making Docker listen on a TCP port. In this scenario user calls to the Docker Remote API over a TCP/IP network, then Docker Remote API pass request to the Docker daemon. This is useful when accessing Docker daemon remotely [2].

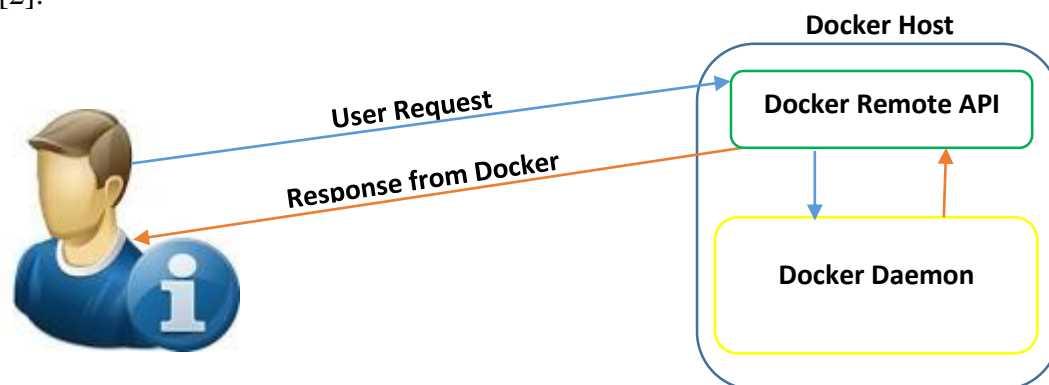


Figure 2: Default user request response flow

One of the major security shortages of this scenario is Docker still hasn't introduced a mechanism to authenticate user request from the Docker Remote API. Because of that once you make Docker daemon listen on a TCP port, anyone can make request to the Docker daemon via Docker Remote API [2].

Importance of the Problem and the Security

Since Docker is one of the major emerging technologies for virtualization of resource, it is important to fill all the security loopholes and security gaps. Docker Remote API is one of the main mechanism to interact with Docker. It contains a lot of API endpoints to interact with the Docker daemon. It facilitates to accomplish almost all the functionalities which users can perform via other types interaction mechanisms. These functionalities includes run, start, create, stop, pause, restart and delete containers; list, build, push, inspect and remove images; and many more [3].

When consider about the real world situation developers, system administrators and network administrators need to access and modify dockerized resources and services remotely. That makes using Docker Remote API more and more essential. But without proper authentication of requests at the Docker Remote API will cause to severe security problems such as unauthorized disclosure of information, information theft and denial of service. As an example if someone got know about Docker Remote API can be accessed via port 4243 at host which has 46.101.37.211 IP address, it can be issued a simple cURL request like mentioned below [4] to get the information about running containers at the host.

```
curl -i -H "Content-Type: application/json" -X GET http://46.101.37.211:4243/containers/json?all=1
```

It will output information such as container id, name, size, opened ports etc. of each Docker container as a JSON object. These are very important information which can be used to perform unauthorized action at the Docker host. As an example user can issue another request with a container id he obtained in the previous step to stop a container which will make users accessing critical service. So it is clear that it is needed to introduce proper authentication mechanism to authenticate each and every request which receive to the Docker remote API before making it available to access over the public internet. Otherwise it will be not practical to do that.

Scope

This project's main focus is to introduce a token based authentication mechanism to authenticate requests which are made to the Docker Remote API when Docker daemon listening via a TCP port. Hence the project has not focused in introducing authentication mechanisms for other forms of Docker interactions. At the same time this project is taken into account well defined token based request authentication protocols but won't be considered other type of authentication protocols. The project also taken in to account mechanisms to reduce the latency and other consequences to a minimum level. Furthermore this project is solely focused request authentication and won't be taken into account other security aspects.

CHAPTER 02 – Literature Review

CHAPTER 02 - Literature Review

Accessing Docker remotely

This section describes how to make the Docker can be access remotely. Further it describes different endpoints which are available in the **Docker Remote API**. Further it describes how they should be accessed and importance of them. Furthermore it describes how to make Docker daemon listening on a TCP port.

Introduction

In the previous chapter, it has mentioned that it is needed to make **Docker Daemon** listening on a TCP port or enable a TCP socket, if it is needed to be accessed remotely over a TCP/IP network. To accomplish that, first it is needed to stop the Docker service and then run the Docker with needed options. The following two command should be issued as the root [5].

```
service docker stop
```

```
docker daemon -H tcp://0.0.0.0:2375
```

In the above mentioned second command, particular IP address can be used instead of 0.0.0.0 if it is needed is need to listen from a single interface. After running docker daemon with above setup, Docker commands can be issued as an http requests or cURL request. It is important to remember that this setup provides unencrypted direct access to the Docker daemon. So it should be secure either by putting a secure web proxy in front of it or using the built in HTTPS encrypted socket. [5].

API endpoints

Docker Remote API contains endpoints to accomplish almost all the functionalities just as users performed via default Docker Client. There are a vast number of API endpoints related to container operations, image operations and miscellaneous operations. The latest API version is v1.24. Docker has provided complete API reference to its remote API. The [Table 1](#) has mentioned details about some of those endpoints, which will be used to testing and evaluation of the system [7].

Endpoint	Request Method	Description	Other notes
containers/json	GET	Lists containers	By default only lists the running containers. Set parameter all=1 will lists all the containers
containers/create	POST	Creates a container	
containers/{id}/json	GET	Returns low level information about the container	Set size=1 to get size information.
containers/{id}/start	POST	Starts a container	
containers/{id}/stop	POST	Stops a container	
containers/{id}	DELETE	Removes a container	Url parameter f=1

			will kill and remove container forcefully.
images/json	GET	Lists images	
images/build	POST	Builds an image from a Dockerfile	
images/{name}	DELETE	Removes an image	
images/search	GET	Searches for an Image	Url parameter term can be used to give the keyword for searching.
/info	GET	Displays system wide information	
/version	GET	Displays docker version information	
/_ping	GET	Ping the docker server.	
/containers/{id}	DELETE	Deletes a given container by container id	

Table 1: Important Docker Remote API endpoints

API authentication mechanisms

This section is focused on different API authentication techniques. Most of them are token based authentication techniques. Further this section is focusing on identifying strengths and weaknesses of each and every technique that has taken into account. Because it is essential in producing a quality outcome. Furthermore the proposed solution is a token based approach the learnings of this section will be used to streamline the proposed architecture and to produce good outcome.

Introduction

Application programming interfaces allows users to interact with a particular service to perform certain operations. As an example, a simple PHP application can be developed to perform insert, select, update and delete operations in a remote MySQL database by using cURL requests. So the users don't need to have a MySQL client but using API they will be able to perform the database operations as they do by using a MySQL client. In this type of scenario all the background operations are performed by the API side and the results will be output to the user. Hence users are no longer required to bother about the system or its architecture in performing any operation.

Authentication of user requests is one of the most concerned security aspect of API based interaction mechanism. Because requests are coming from the outside, so it is needed to verify who issued the request. Further it is needed clarify the originator of the request has proper the access level to issue a given request. This will help to provide proper secrecy for the end systems

and the non-repudiation. Next few sections are focused some well-defined authentication mechanisms and protocols which can be used for the API authentication.

Kerberos

Kerberos is a network authentications protocol developed by Massachusetts Institute of Technology in the mid-1980s [4]. It is available as an open source protocol as well commercial protocol. Kerberos helps to prevent usage of user credentials with each and every request which user made to the server. Over thirty years Kerberos has evolved as one of the stable and secure authentication protocol. Kerberos architecture is composed of three major components. They are [8],

1. Client
2. Server
3. Authentication Server or Key distribution server

And there are three main exchanges. They are [8] [9],

1. Authentication Service (AS) exchange
2. Ticket Granting Service (TGS) exchange
3. Client Server (CS) exchange

Authentication Service (AS) exchange

- This is the exchange between the Client and the Authentication Server.
- Client sends Kerberos Authentication request to the authentication server specifying credentials it wants.
- Authentication Server replies with a Kerberos Authentication response containing a ticket and a session key.
- The session key encrypted with client's secret key.
- The ticket is encrypted with server's secret key.
- DES is the default encryption algorithm.

Ticket Granting Service (TGS) exchange

- This exchange is used to obtain additional tickets for the servers.
- This doesn't need client secret key for the encryption.
- Ticket Granting Service exchange is transparent to the user.
- Ticket Granting Server must have the access to the all secret keys.
- It encrypts the tickets using server's secret key.
- The Client sends Kerberos Ticket Granting Service request to the TGS server.
- The Server replies with Kerberos Ticket Granting Service reply to the client with a ticket.

Client Server (CS) exchange

- Client contacts with the server that provides the relevant service.
- Client sends KRB_AP request to the server specifying the service.
- Application server validates client by decrypting ticket with server's secret key and decrypting authenticator with session contained in the ticket.
- Service optionally replies with KRB_AP reply.

The [Figure 3](#) illustrates the authentication procedure of the Kerberos. Kerberos is now well established and globally accepted authentication protocol. It has developed and maintained through several iterations to get into the current level. But implementation or integration Kerberos to a service requires good expertise and knowledge. Furthermore sometimes proper integration may be time consuming. Hence it is needed to consider very well whether it is needed to go for Kerberos or looking for some other solution.

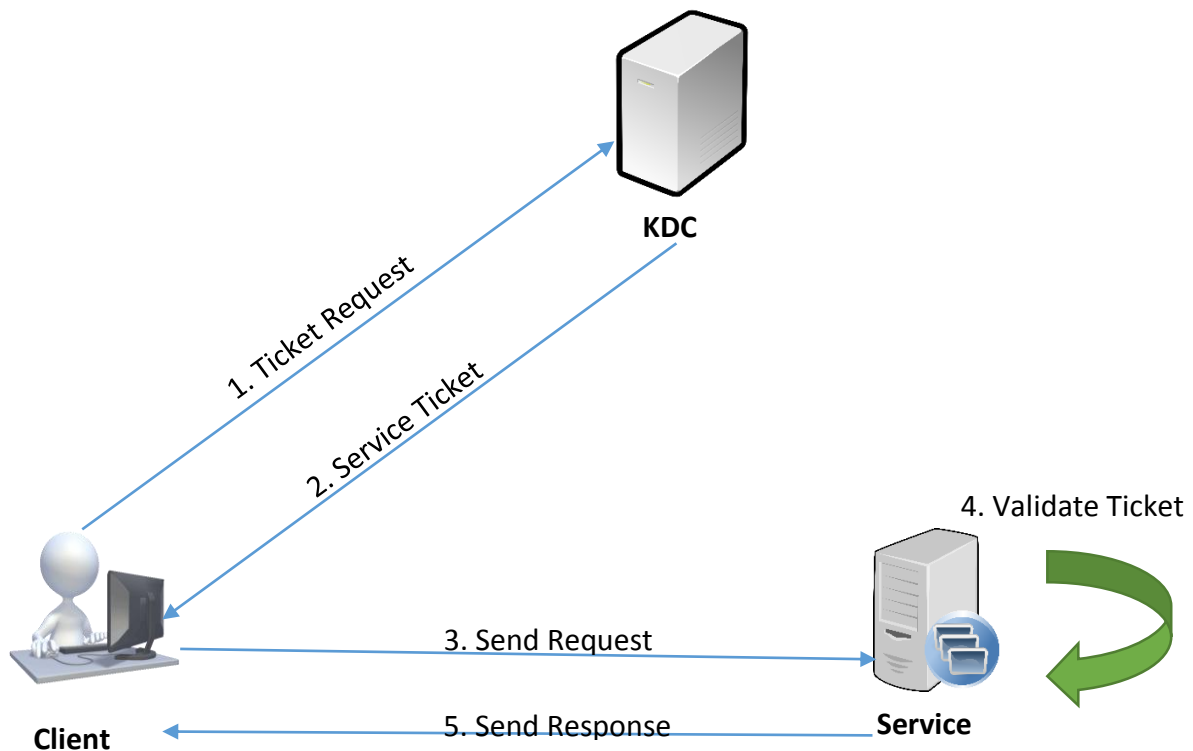


Figure 3: Kerberos authentication [10]

OAuth

OAuth stands for Open Authorization which is an open standard protocol that provides simple and secure authorization for different types of applications. It allows providers to give access to the users without any exchange of credentials. OAuth designed for use only with HTTP protocol [12]. There are many reasons for using OAuth. Flexibility, compatibility and platform independence are some of them. Furthermore it provides a method for users to grant third-party access to their resources without sharing their credentials. Mostly used version of OAuth is OAuth2. In OAuth2 there are four parties. They are, [11] [12].

1. Resource owner - The user
2. Resource server - The API
3. Authorization server - Same as the API server.
4. Client - The third party which needs authorization.

There are four granting methods or grant types. They are,

1. **Authorization Code Grant** – This method is associating with generation of a login link. By Clicking on that user visits the authorization page. On success, user is redirected back to the site with an authorization code. On error user is redirected back to the site with an error code. Server exchanges authorization code for an access token [11].
2. **Implicit Grant** - This method is associating with generation of a login link. By clicking on that user visits the authorization page. On success user is redirected back to the site with an access token in the fragment. On error user is redirected back to the site with an error code [11].
3. **Resource Owner Password Credentials Grant** - A trusted client (usually a first party application) submits a username and a password. Additionally it is required to mentioned grant_type parameter of the request. Client receives an access token in the response body [11].

4. **Client's Credential Grant** - Every third party application receives a `clientId` and a `clientSecret` when they were created (sometimes `clientId` is called as `appId` and `clientSecret` is called as `appSecret`). Client submits client credentials and receives an access token in the response body [11][12].

The access token which is obtain through any type of granting method is used to make requests. Every access token has an expiration time. Hence it makes exposing of an access token to an outside is only useful to a certain time. The expiration time generally set less than one hour. After the expiration time it is needed obtain new access token making a request using refresh token.

Research on similar works

This section is focused on the similar works on Docker which are related to the research. Basically this section is focused on previous researches and projects. The knowledge obtained from that those researches and projects is used model the proposed solution.

Docker Registry 2 Authentication Server ([github - cesanta/docker_auth](https://github.com/cesanta/docker_auth))

The focus of this project is to ensure the authenticity and the authorization of Docker clients' requests which are made via Docker daemon to the Docker registries. This project basically focuses on providing authenticity for the docker image "push" and "pull" operations which are obviously make interactions with a Docker registry [13]. The original Docker Registry server (v1) did not provide any support for authentication or authorization. Access control had to be performed externally, typically by deploying Nginx in the reverse proxy mode with basic or other type of authentication. While performing simple user authentication is pretty straightforward, performing more fine-grained access control was cumbersome [13][14].

Docker Registry 2.0 introduced a new, token-based authentication and authorization protocol, but the server to generate them was not released. Thus, most guides found on the Internet still describe a set up with a reverse proxy performing access control [13][14]. Docker registry 2.0 authentication work is illustrated in [Figure 4](#) [13].

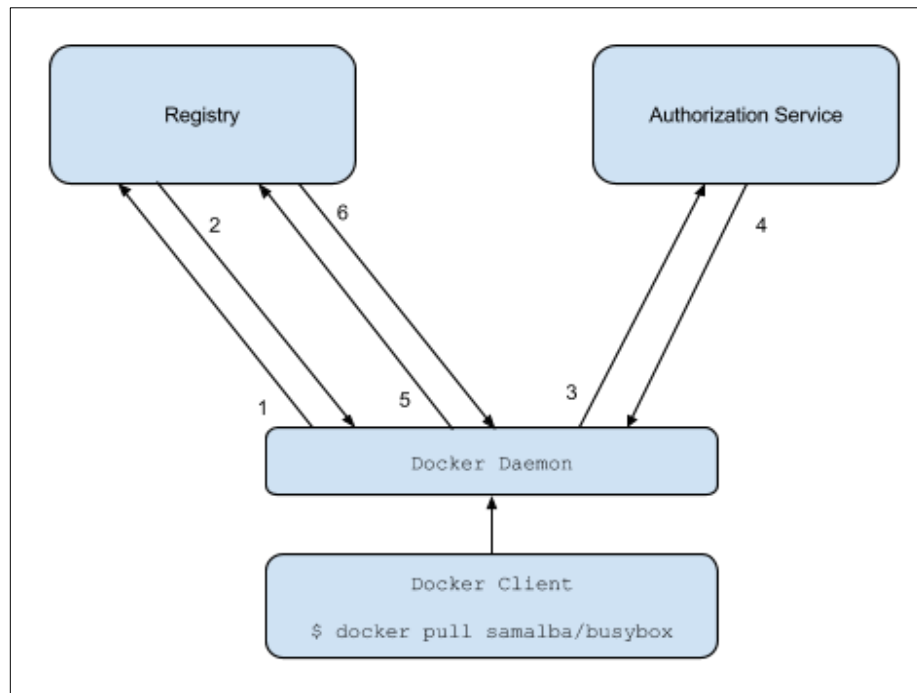


Figure 4: Docker Registry Authentication [13]

1. Attempt to begin a push/pull operation with the registry.
2. If the registry requires authorization it will return a **401 Unauthorized** HTTP response with information on how to authenticate.
3. The registry client makes a request to the authorization service for a Bearer token.
4. The authorization service returns an opaque Bearer token representing the client's authorized access.
5. The client retries the original request with the Bearer token embedded in the request's Authorization header.
6. The Registry authorizes the client by validating the Bearer token and the claim set embedded within it and begins the push/pull session as usual.

Once the token server has determined what access the client has to the resources requested in the scope parameter, it will take the intersection of the set of requested actions on each resource and the set of actions that the client has in fact been granted. If the client only has a subset of the requested access it must not be considered an error as it is not the responsibility of the token server to indicate authorization errors as part of this workflow [13].

Improving Docker

In this section, it has focused on how the developers or other third parties can contribute to Docker. As well as in any other open source project, the main intentions of taking contributions from the third parties to ensure continuous improvement, reduce the amount of bugs associated with the source code, improve the security and increase the performance. Docker keeps its code in a GitHub repository which makes it easy for contributors to improve the code. There are several ways to contribute to the Docker. Some of them which are relevant to the project has been discussed in the next few subsections.

Introduction

In Docker there are six ways to contribute. They are [15] [16].

1. Improve the documentation - explain how thing works in docker in the documentation.
2. Improve the code - add new feature or upgrade existing one.
3. Support users - support docker users through community channels.
4. Help to grove community - help to grove community by making the project welcoming and easy to use.
5. Testing - help for functional testing, usability testing or spotting problems.
6. Issues - help by organizing issues or reporting spotted ones.

Since this project is focused on adding new feature or upgrading existing one the next subsection will be focused on how to improve the Docker code.

Basic Contribution

Docker allows developers to contribute for the source code it in two ways. The first way is basic contribution. It is required to follow a predefined workflow for the basic contribution. This workflow allows developers to take the ownership of issues in Docker which are identified and listed by Docker developer team and fix them. All the Docker repositories are on GitHub therefore all the issues are also listed in the GitHub. Developers are need to have a basic knowledge in Git version control system to deal with the issues. Furthermore developers should have configured their development environment by installing and configuring Git, make and Docker [17]. The [Figure 5](#) illustrates the basic contribution workflow of Docker.

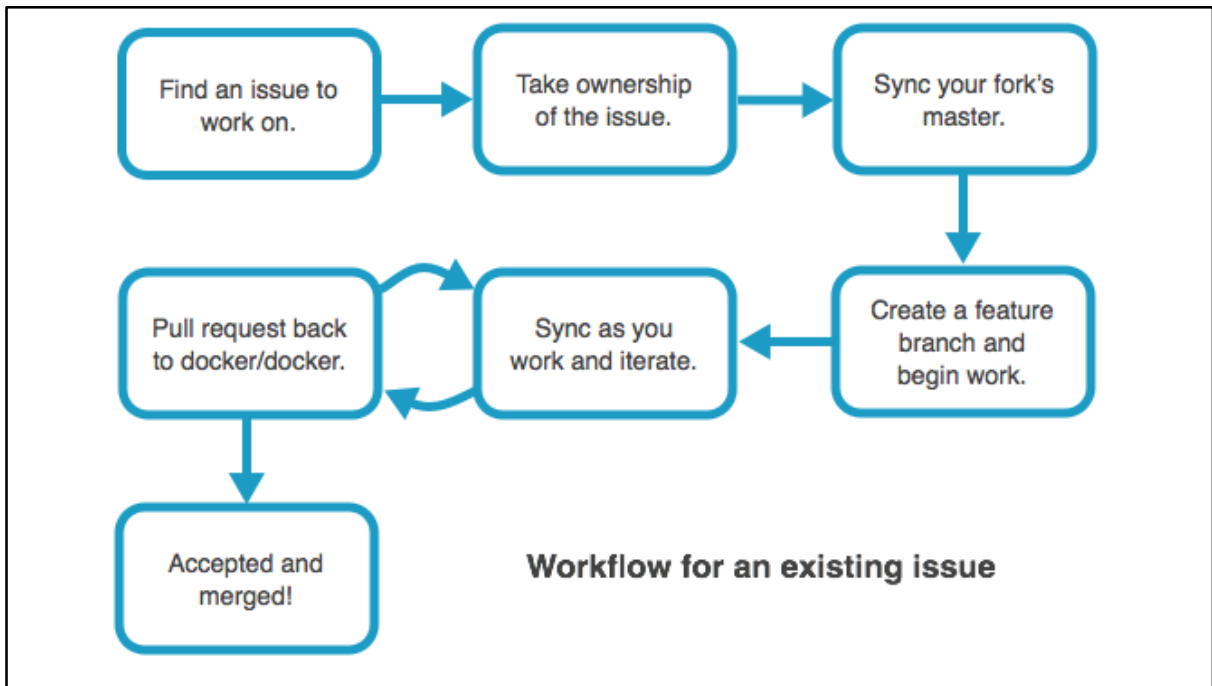


Figure 5: Docker Basic Contribution Workflow [17]

Contributors can take ownership of any open issue which they like to work on. In the process of finding issues contributors can filter issues by **author**, **labels**, **milestones**, and **assignee** or sort by time such as **newest** and **oldest**[17]. Docker maintainers assign labels to issues to make it easier for contributors identify the different types of issues. The labels are color-coded and help contributors categorize and filter issues. There are four labels categories; **kind**, **area**, **experience**, and **priority**. A contributor can filter using one or more labels. The kind and experience labels are useful for new contributors [18].

Advanced Contribution

The other way to contribute to Docker source code is advanced contribution. As well as in basic contribution advanced contribution also has a workflow but it is more complex and time consuming than the basic workflow. Furthermore advanced contribution needed greater programming experience. The workflow starts with a new idea which is focused on solves a problem or adds a new feature to docker. This process requires two pull requests, one for the design and one for the implementation. Developers are need to have a good knowledge in Git version control system to deal with the issues. Furthermore developers should have configured their development environment by installing and configuring Git, make and docker [19]. The [Figure 6](#) illustrates the advanced contribution workflow.

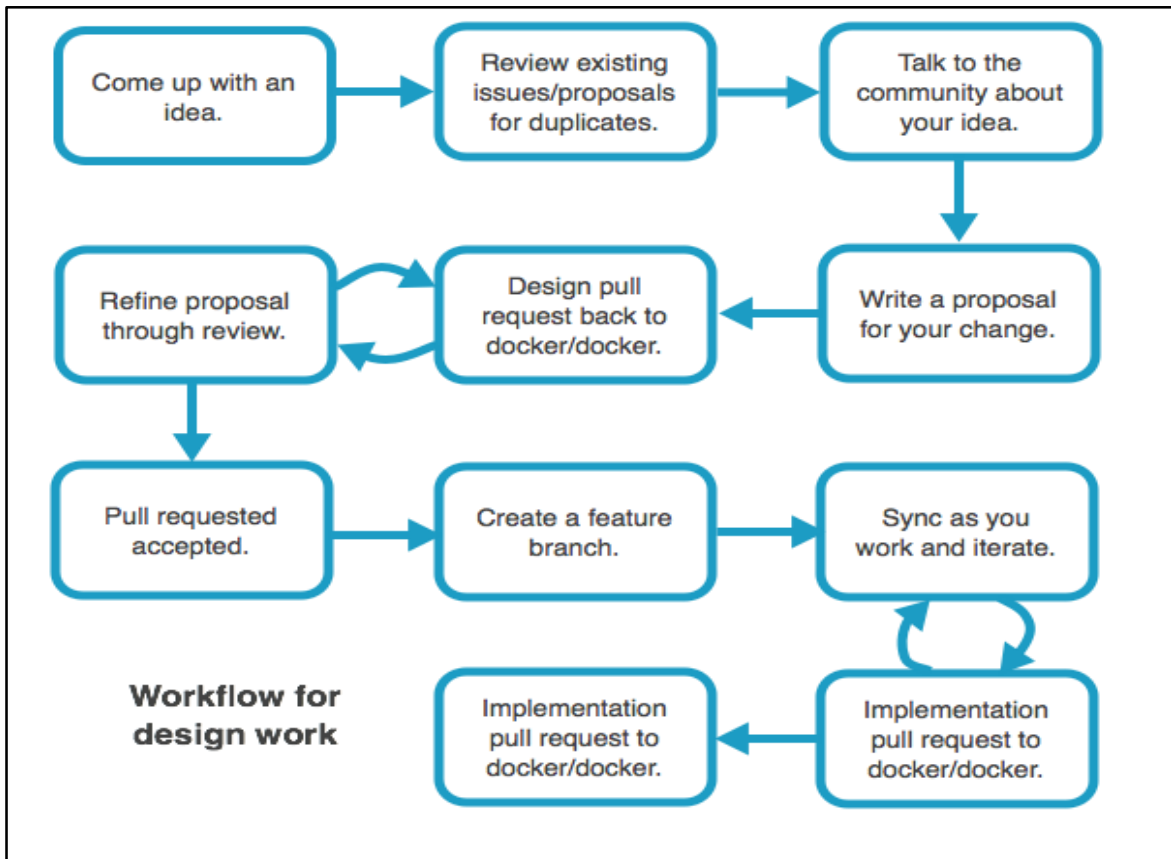


Figure 6: Advanced Contribution Workflow [19]

1. Come up with an idea - Usually an idea comes for the limitation or absence of a feature of the product [19].
2. Review existing issues/proposals for duplicates - It is important to make sure that the same idea is not proposed by someone else. The design proposals are all online in docker GitHub pull requests [19].
3. Talk to the community about the idea - There are several online communities discuss and get feedback about the idea [19].
4. Fork docker/docker and clone the repo to the local host - This will be the working space.
5. Create a new text file in the area you wish to change - This will be created in the directory where the changes are going to make happen. It should be a markdown file.
6. Write a proposal for the change into the file - This will describe the idea and it may contains information like [19].
 - a. Why is this change needed or what are the use cases?
 - b. What are the requirements this change should meet?
 - c. What are some ways to design/implement this feature?
 - d. Which design/implementation does developer think is best and why?
 - e. What are the risks or limitations of the proposal?
7. Submit the proposal in a pull request to docker/docker.
 - a. The title should have the format:
 - b. Proposal: *short title*
 - c. The body of the pull request should include a brief summary of your change and then say something like “*See the file for a complete description*”.
8. Refine the proposal through review - The maintainers and the community review the proposal. Developer will need to answer questions and sometimes explain or defend the approach. This is chance for everyone to both teach and learn.

9. Pull request accepted - This can be either accepted or rejected. If it's accepted the following steps will have to be followed [19].
10. Implement your idea - implementation uses all the standard practices of any contribution.
 - a. fork docker/docker.
 - b. create a feature branch.
 - c. sync frequently back to master.
 - d. test as you go and full test before a pull request.
11. Submit a pull request - When developer has a complete implementation, submit a pull request back to docker/docker.
12. Review and iterate on the code.
13. Acceptance and merge.

Conclusion

Providing proper authentication for the requests which receives to the Docker daemon via Docker Remote API when the Docker daemon listening on a TCP port, requires the assistance of an authentication service which is independent from Docker. Furthermore Docker Host needs to interact with the authentication service to complete the authentication process. If the authentication of requests which are pointed to the Docker Remote API needs to be implemented as integrated solution it has to follow the advanced contribution workflow. The advanced contribution workflow of Docker is the standard method which has be followed to introduce new feature to the Docker source code. But it is a lengthy, complex and time consuming procedure. Hence it is need to be focused on some other way of introducing authentication service in the authentication process.

CHAPTER 03 – Design

CHAPTER 03 – Design

Introduction

The new design introduces token requesting and token validation steps to the Docker API request flow. In that type of setup, every user who wishes to interact with the Docker Remote API will have secret credentials to an authentication server. Credentials of all the users will be stored at a data store which is independent form the authentication server. When a user wants to access the Docker Remote API, first he has to issue token request to the authentication server by submitting his user credentials. At the authentication server, it validates user request and issue a token response with a valid access token. Here onward user will have to submit this token until it expires to access the Docker Remote API. In each every request to the Docker Remote API, he submits this token in the request's Authorization header as in the example mentioned below.

```
curl -i -H "Content-Type: application/json" -H "Authorization: Bearer 111
d2b76f6f11648b46ad6bf490e93b27552aef7a62" -X GET
https://192.168.10.35:4243/containers/json
```

At the Docker Host side, when it receives the request, it will examine the Authorization header of the request and extract the given user id and the access token. If user hasn't provided the access token or the user id the request will be rejected. If the user has provided the access token and the user id, a token validation request will be issued to the authentication server. When the authentication server receives a token validation request, it will check the validity of the access token against the user id and response according to the status of token. Finally Docker Host will receive the token validation response from the authentication server based on that either it will pass the user request to the Docker daemon or reject it.

Authentication Server

The Authentication Server is the core part of the proposed design. It is responsible for the following operations.

1. Access Token Issuing - After a user successfully submits a user credentials, the authentication server should generate and issue a valid access token for the user to make API requests to the Docker Remote API.
2. Access Token Maintenance - Authentication server should store generated access tokens with in the server and expire them when the validity period escaped. If the user requests an access token and valid access still available the authentication server should return that token instead of generating a new one.
3. Access Token Validation - When the authentication server receives an access token validation request it should assess its existence and the validity and respond accordingly.

Components and Technologies used

Server Programing and Endpoints

The Authentication Server's programming has been done by using Node.Js. The reasons for using Node.Js for the server programming are [20],

- It provides the ability to concurrent request handling through asynchronous event driven input/output operations.
- Lightweightness allows better utilization of resources.
- Vast range of Node Package Modules allows to interact with different other services.
- Short learning curve because of the usage of JavaScript.
- Active and skilled community with lots of publicly accessible code bases.

Communication between users and the authentication server have been protected by using secure socket layer certificates. The authentication server contains with two primary endpoints. They are

1. **/getAccessToken** - This the endpoint which users can submit their user credentials and obtain an access token from the authentication server. The request type will be http POST and the user id and the password should be included in the request body. The structure of the cURL command that can be used to call this endpoint has mentioned below.

```
curl -H "Content-Type: application/json" -X POST -d '{"id": 111, "password": "abc123"}' https://<server-host>:<server-port>/getAccessToken -cacert <path-to-SSL-certificate>
```

The response from the authentication server will be looked like below if the user credentials are valid.

HTTP/1.1 200 OK

Content-Type: application/json

```
{"accessToken": "6b5930e436243dcda0ead985c6233915ef25f606", "expiresIn": 1025}
```

The [Figure 7](#) illustrates the workflow of access token generation and return when a user sends a proper request to /getAccessToken endpoint of the authentication server.

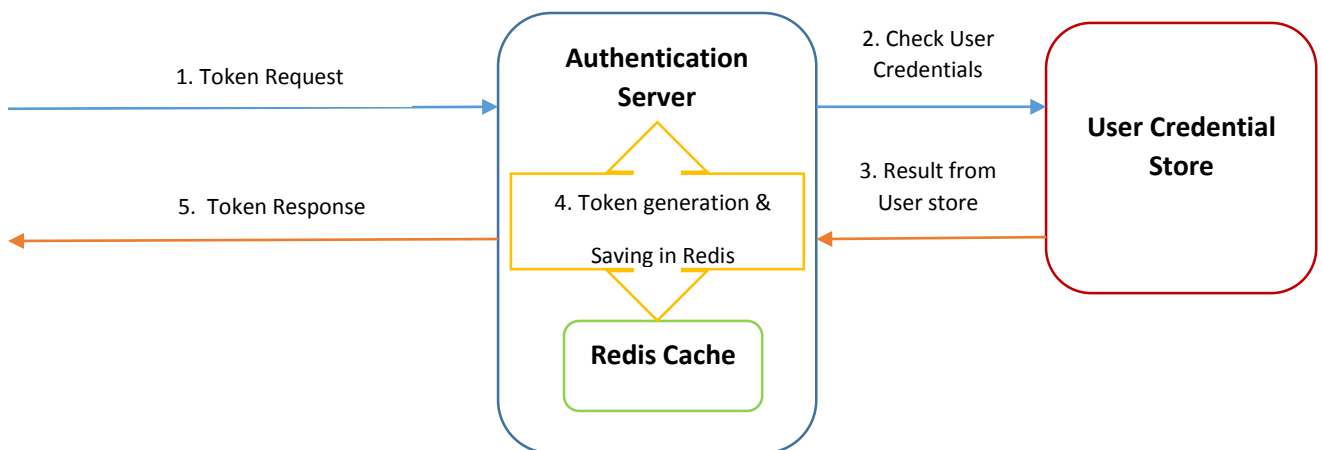


Figure 7: Access token generation workflow

2. **/checkValidity** - This the endpoint which is used for assess the validity of the access tokens. When a user submits a request to the Docker Remote API with an access token inside Authorization header, from the Docker Host side an http POST request will be issued to this endpoint to check the validity of the given access token. The structure of the cURL command that can be used to call this endpoint has mentioned below.

```
curl -H "Content-Type: application/json" -X POST -d '{"id": 111, "access_token": "6b5930e436243dcda0ead985c6233915ef25f606"}' https://<server-host>:<server-port>/checkValidity -cacert <path-to-SSL-certificate>
```

The response from the authentication server will be looked like below if the user credentials are valid.

HTTP/1.1 204 NO CONTENT

The [Figure 8](#) shows how a given access token being verified from the authentication server.

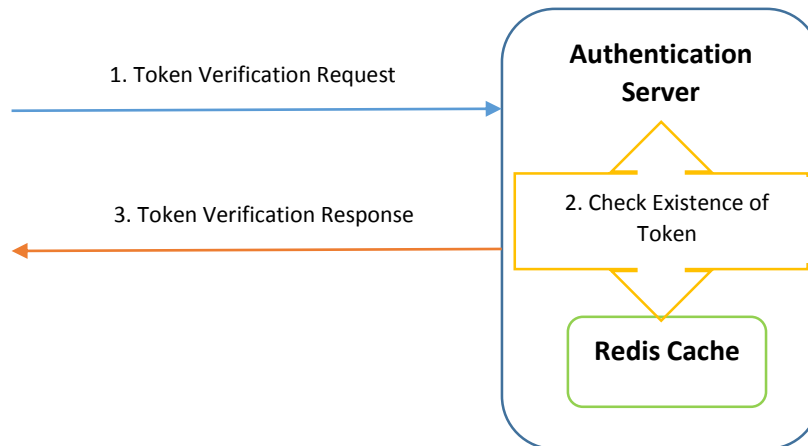


Figure 8: Access token verification workflow

Access Token and User Credential Management

Access token storing and removing will be done by using **Redis**. Redis is a data structure store which keeps stored data in memory. It supports different types of data structures such as strings, hashes, sets, lists etc. It stores values as key value pairs. Values can be obtained by using the keys. The reasons for selecting Redis as the access token storage are [21],

- Can be easily integrated with Node.Js.
- Provide maximum flexibility in handling access tokens.
- Increase the overall performance of the authentication server because Redis keep data in the memory.
- Each access token can be stored separately as key value pairs.
- Redis provides ability to set timeout for the keys which is useful to expire access tokens.

Redis server will run within the host machine of the authentication server. The authentication server will interact with the Redis when it store access tokens and checking a validity of a given access token. Usage of Redis will be able to improve the performance of the authentications server. Hence it will help to reduce request-response routing latency. There are several Node.Js Redis client modules which can be used to interact with the Redis server in storing and obtaining access token.

User credential management is independent from the authentication server. Which means administrator can manage their user credential storage within or outside of the authentication server machine. The authentication server can be implemented to support different types database management systems such as MySQL, PostgreSQL, Redis or MongoDB. That will require a special configuration. Currently the authentication server only works with MySQL.

Approaches Considered

The other main design consideration in the new architecture was Docker host and the authentication server communication. Docker host needs to communicate with the authentication server in order to verify the validity of the access tokens which is included in the request's Authorization header. For that two main approaches are taken into account. The first approach was changing the Docker source code. The other approach was run Docker with a proxy server.

Changing Docker Source

The first approach that was considered is changing the Docker Remote API source code to support new design. In that approach it is was planned to clone the source code from the GitHub and make necessary changes in the code level to support the new design. After those changes Docker Remote API will extract the access token from the user request, directly interact with the authentication server to verify the validity of the access tokens and respond the to the user requests. This can be seen as more integrated approach. But there are several issues related to this approach.

The first issue is changing the source code and running it on the local machine is inflexible. Because when it needs to distribute the change, it is needed to change the source code of every machine. The second issue is making the changes publically available. In Docker, developers can not directly push their changes to Docker repository. To make changes publically available it is need to follow Advanced Contribution procedure which was mentioned in the previous chapter. But it is a time consuming approach. As the Docker mentioned, sometimes it will take more than a year to complete the whole process and accept the changes by Docker. Sometime getting approval for the proposal will solely take 6 months.

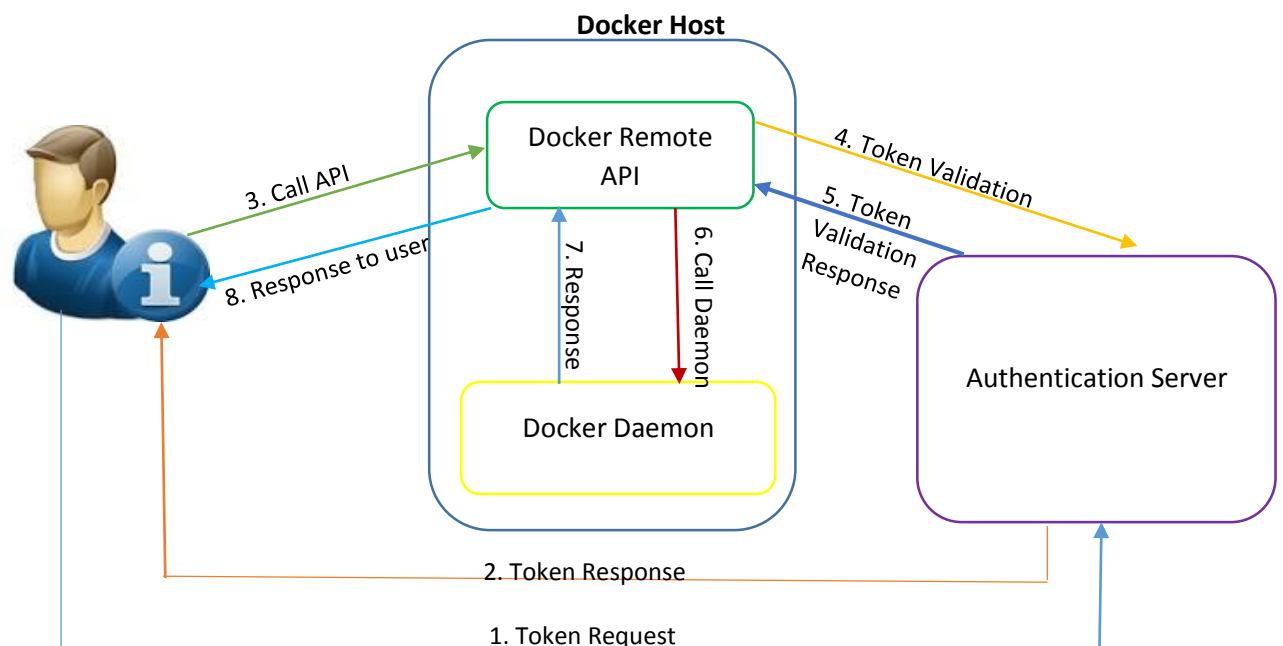


Figure 9: Request flow after changing Docker source for request authentication

Run Docker with a proxy server

In this approach none of code level changes has not been done to the Docker. At the same time Docker Remote API won't be opened to the public. A Proxy servers stand between the client and the Docker Remote API, hence the client request passes through the proxy server to the Docker Remote API. In this approach there won't be any direct communication between the Docker Remote API and the authentication server. Client requests will be received to the proxy

server. Then it extracts the access token from the Authorization header and checks the validity of it with the authentication server. Based in the response from the authentication server proxy server will either pass the request to the Docker Remote API or it will return access token error to the client.

Communication between the client and the proxy server is being protected by using secure socket layer and communication between proxy server and the Docker Remote API is happen over Hypertext Transfer Protocol. Proxy server appears to the client as the service endpoint and give the responses to the client request as it produce by itself. Hence this proxy server can be considered as a reverse proxy. The main advantages of this approach is, it provides more flexibility in providing authentication because of the usage of the proxy server and If any change happens to the Docker it will easy to make changes to the proxy server and the authentication server because total control of them is exist with the developer.

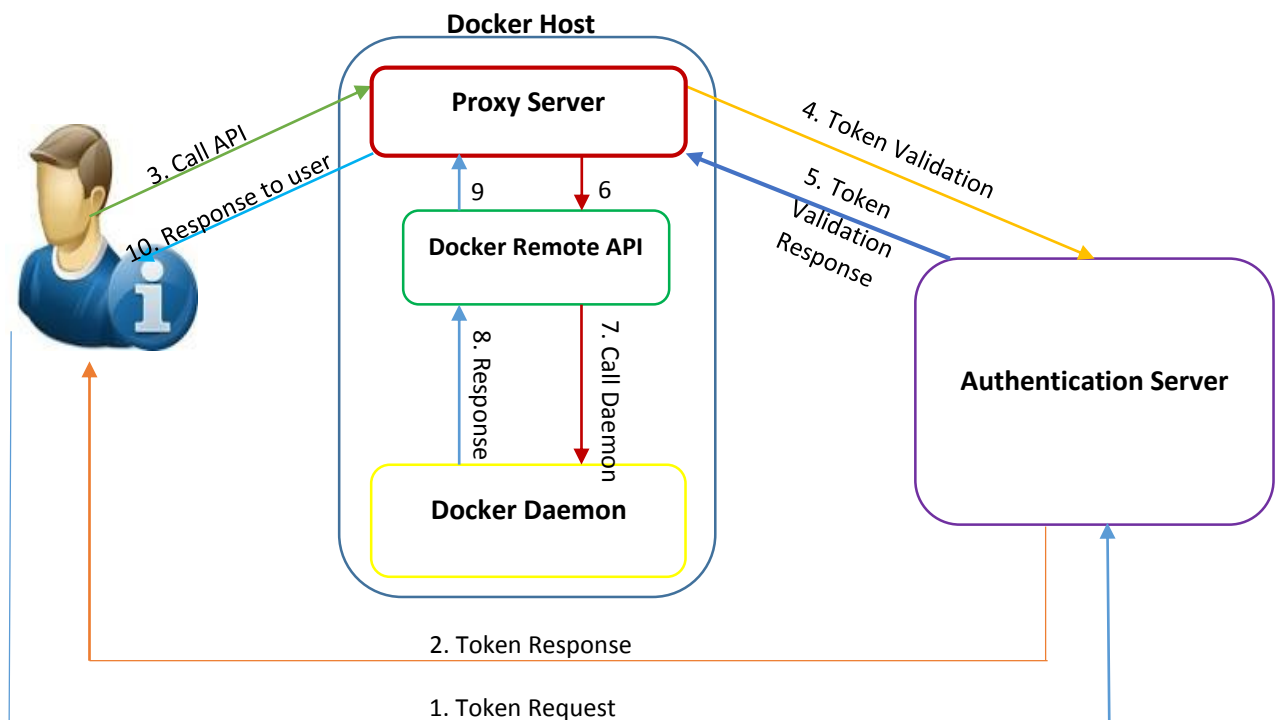


Figure 10: Request flow when running Docker with a proxy server

Conclusion

The first approach, changing Docker source code to handle token verification can be considered as a more integrated solution than the second approach. But it has above mentioned limitations. Because of the flexibility of the second approach the second approach is selected to the implementation. Therefore for a proxy server application was written by using NodeJS a part of the implementation. The proxy server application is opened to the outside and listing on configured TCP port. Hence the proxy server application runs inside the Docker Host machine, it can pass the user request to the Docker Remote API.

CHAPTER 04 – Implementation

CHAPTER 04 – Implementation

Introduction

This chapter describes how the proposed system's components has been implemented. Further it describes how the system components are interconnected with each other in order to achieve objectives of the project in technical and operational perspective. At the same time it has included algorithms and source code of each component. Further it has included file and folder structure of the components. Furthermore it has focused on how different predefined software packages are being used for the purpose of the implementation.

User Store

The user store is the place where the credentials of the users who are eligible to access the Docker Remote API are been stored. A MySQL database has been used as the user store. Inside this database, there is a table called **doc-user** where the user credentials are kept. The doc-user table stores the first_name, last_name and the password of the users. The **id** is an auto increment which use as the user unique identifier. The password is the SHA1 string of the actual password. The [Figure 11](#) illustrates the structure of the doc-user table.

```
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> use store;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> SELECT * FROM doc_user;
+-----+-----+-----+-----+
| id | first_name | last_name | password |
+-----+-----+-----+-----+
| 1 | asitha | seneviratne | 6367c48dd193d56ea7b0baad25b19455e529f5ee |
| 2 | John | Doe | 2a43ccf990c46a5279c52acdb2dee3ab748c8e95 |
+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

Figure 11: MySQL table for user credentials

Authentication Server

Overview

The Authentication server is operates as a REST API service which runs on Ubuntu 14.04 LTS operating system and written by using NodeJS. It contains with two endpoints. One for issue access tokens and the other one for check the validity of the access tokens. Its main running script is **index.js**, which starts the authentication server by invoking server start function. The authentication server's resource files are organized according to standard NodeJS server development directory structure as mentioned in the [Figure 12](#).

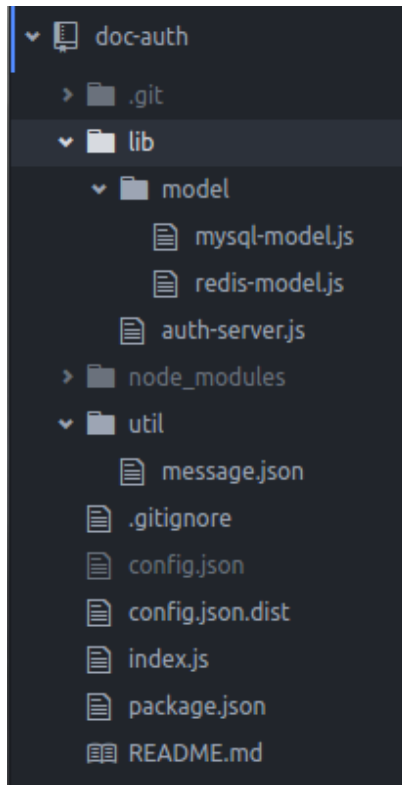


Figure 12: File and directory structure of the authentication server

All the files which are regarded to the functionalities of the authentication server are kept under the **lib** directory. Inside the lib directory there is a file called **auth-server.js** which contains the logic related to server initiation and listening. There is another directory named as **model** which is rests inside the lib directory contains two resource files. The **mysql-model.js** handles the interactions with the MySQL database which contains user's login credentials and the **redis-model.js** handles the interactions with **Redis server**, which is used to store and retrieve access tokens.

There are several node package modules has been installed under the **node_modules** directory to accomplish the operations of the authentication server. They are,

1. npm-https - to establish secure connections between the users and the Docker host
2. npm-crypto - to perform cryptographic operations.
3. npm-mysql - to interact with MySQL database.
4. npm-redis - to interact with Redis to store and obtain access tokens.
5. npm-fs - to perform file system operations.

All the authentication server configurations are being kept in the **config.json** file and the other supportive resource files are being kept under the **util** directory.

Authentication Server Initiation

Authentication server will be started by running the main script **index.js** by issuing the command **node index.js**. It invokes **start** function in the auth-server.js. Below mentioned code lines indicate the contents of the index.js and start function of the auth-server.js respectively.

```
'use strict';
var authServer = require('./lib/auth-server');
authServer.start(function (err) {
  if (err) {
    console.dir(err);
    console.log("Unable to start Auth Server.")
  }
});
```

```

exports.start = function (callback) {
  try {
    server.listen(conf.serverPort, conf.serverAddress);
    console.log(className + "Server " + conf.serverPort + " : " + conf.serverAddress + " started
... ");
    callback();
  } catch (e) {
    console.log(e);
  }
};

```

In the above mentioned start function, **server** is an instance of npm-https **createServer** class. It invokes the **listen** function of that instance. Authentication server's listening address and the listening port is configure in the **config.json** as the **serverAddress** and the **serverPort** respectively. Both of them should be passed to the listen function.

Authentication Server Functionality

Authentication functionality is implemented mainly in the **auth-server.js**, **model/redis-model.js** and **model/mysql-model.js**. In the auth-server.js, it creates an instance of npm-https createServer class which is capable of responding to the requests which are designated to the **/getAccessToken** endpoint which issues access tokens for the successful user credential submission and **/checkValidity** endpoint which validates the access tokens. The below mentioned code lines related to the auth-server.js, which are importing of the required node package modules which are **npm-https** and **npm-fs**, system configurations and utilities. Furthermore it creates two instances of the mysql-model and the redis-model.

```

'use-strict';
var https = require('https');
var conf = require('./config.json');
var fs = require('fs');
var messages = require('./util/message.json');
var dbModel = require('./model/mysql-model').createMySQL();
var redisModel = require('./model/redis-model').createRedis();

```

The responses from the authentication server are served as JSON format. Hence within the auth-server.js content type header has been set to **application/json** as mentioned in the next code block. The npm-https createServer requires NodeJS object which contains the server's SSL private key, SSL certificate and the passphrase of the private key to create createServer object. The paths to the SSL private key and the SSL certificate and the passphrase of the SSL private key are being configured in the config.json.

```

var className = "Auth-Server";
var default_header = {
  'Content-Type': 'application/json'
};
var privateKey = fs.readFileSync(conf.sslPrivateKeyPath);
var certificate = fs.readFileSync(conf.sslCertificatePath);
var credentials = {
  key: privateKey,
  cert: certificate,
  passphrase: conf.certificatePassphrase
};

```

In the auth-server.js, it creates npm-https server instance as mentioned in the next code lines. The credential object which is previously created should be passed and in the callback function

it gives the request (req) and the response (res). Inside server instance all functionalities of the authentication server has being handled. Inside the createServer instance, when it receives a request from the user first it checks the request method. If the request is not an http POST request, it will reject the request. If the request method is POST then it obtains request body (reqBody) of the request to a NodeJS object. After that it checks the request URL (req.url) whether it is /getAccessToken, /checkValidity or something else. If the request URL is not either /getAccessToken or /checkValidity it will returns an error.

The _sendResponse function, which rest inside the server is used to send the response back to the user request. It should be passed three parameters to the _sendResponse function. The httpStatusCode should be a valid status code which is defined in RFC 2616, the header is a NodeJS object which contains all the response headers and the resBody is a NodeJS object which contains the response body to be sent.

```
var server = https.createServer(credentials, function(req, res) {
  var chunk = "";
  if (req.method !== 'POST') {
    var error = messages.request_type_error;
    _sendResponse(error.httpStatusCode, default_header, error.messageBody);
  }
  req.on('data', function(dataChunk){
    chunk += dataChunk;
  });
  req.on('end', function (){
    try {
      var reqBody = JSON.parse(chunk);
      switch (req.url) {
        case '/getAccessToken':
          // code related to access token generation and granting
          break;
        case '/checkValidity':
          // code related to access token validity checking
          break;
        default:
          var error = messages.resourse_not_found;
          _sendResponse(error.httpStatusCode, default_header, error.messageBody);
          break;
      } catch(e) {
        console.log(e);
        var error = messages.internal_server_error;
        _sendResponse(error.httpStatusCode, default_header, error.messageBody);
      }
    });
  function _sendResponse(httpStatusCode, header, resBody) {
    console.log("send response");
    res.writeHead(httpStatusCode, header);
    res.end(JSON.stringify(resBody));
  }
});
```

Access token generation and granting

Access token generation and granting happens when a user sends an http POST request with his user id and the password inside the request body to the request URL /getAccessToken. The [Figure 13](#) is related to the auth-server.js which illustrates the access token generation and granting of authentication server.

```

case '/getAccessToken':
  redisModel.validateUser(reqBody.id, reqBody.password, function (validateError, dbData) {
    if (validateError) {
      _sendResponse(dbData.httpStatusCode, default_header, dbData.messageBody);
    } else {
      redisModel.setUserObject(dbData, function(setError, errorData){
        if (setError) {
          _sendResponse(errorData.httpStatusCode, default_header, errorData.messageBody);
        } else {
          redisModel.getAccessToken(reqBody.id, function(redisGetError, token_reply){
            if (redisGetError) {
              _sendResponse(token_reply.httpStatusCode, default_header, redisSetError.messageBody);
            } else if (token_reply === null) {
              redisModel.createAccessToken(function(accessTokenObj){
                redisModel.setAccessToken(reqBody.id, accessTokenObj, function(tokenError, setData){
                  if(tokenError){
                    _sendResponse(setData.httpStatusCode, default_header, setData.messageBody)
                  }
                  _sendResponse(200, default_header, accessTokenObj);
                });
              });
            } else {
              _sendResponse(200, default_header, token_reply);
            }
          });
        }
      });
    }
  });
});
break;

```

Figure 13: Access token generation and Granting

When an http POST request receives to the authentication server with URL /getAcceesToken, first it validates the user credentials which are user id and the password by calling **validateUser** function of the **redisModel** which is an instance of redis-model.js. The next code lines are related to the validateUser function in the redis-model.js. The user id and the user password should be submitted to the function for the user validation. Within this function first it creates a **key** and a **hashKey**. The key is created by concating **userPrefix** and the user id. The hashKey is generated by concating the **hashKeyPrefix** and the user id. Both userPrefix and the hashKeyPrefix are kept in the config.json file.

It checks whether user record exists in redis server by calling **redisClient.hget** function by passing the key and the hashKey. The redisClient is an instance of npm-redis. The key is used find the relevant user record and the hashKey is used to decrypt the encrypted user object. If the hget returns a user record from the redis server, this function will generate the SHA-1 hash of the received password and compare the hashed passwords. If the passwords are matched user validation is complete and will returns user object.

If the hget returns an error, it cannot find a user record in the redis server or passwords are mismatched it will call the **mysql.validateUser** function that is inside the **mysql-model.js** which validates user with the MySQL database.

```

RedisModel.prototype.validateUser = function (userId, password, callback) {
  var key = conf.userPrefix + userId.toString();
  var hashKey = conf.hashKeyPrefix + userId.toString();
  this.redisClient.hget(key, hashKey, function (err, reply) {
    if (err || reply === null) {
      mysql.validateUser(userId, password, function (mysqlError, dbData) {

```



```

        callback(true, messages.database_connection_error);
    }
});
}

```

If the user validation was failed user will be received an error message and if the user validation was succeeded it will save or overwrite user record in the Redis server with received user object by calling **redisModel.setUserObject** function. The next code lines illustrate the operation of the saving or overwriting the user recode in the Redis server. As well as in validateUser function of the redis-model.js, this function also create the key and the hashKey to save the user record in the redis server by using npm-redis **hset** function.

```

RedisModel.prototype.setUserObject = function (userObject, callback) {
    var key = conf.userPrefix + userObject.id.toString();
    var hashKey = conf.hashKeyPrefix + userObject.id.toString();
    this.redisClient.hset(key, hashKey, JSON.stringify(userObject), function(err, result){
        if (err) {
            callback(true, messages.redis_set_error);
        }
        callback(null, null);
    });
}

```

After saving the user record in the Redis server, the access token generation process then checks whether there is a previously created access token available in the Redis server. If there is an access token is available in the Redis server for the user then that access token will be sent to the user. Otherwise new access token will be created, store in the Redis server and sent to the user. The **getAccessToken** function in the redis-model is used to query and find the previously created access token for a given user. The next code lines are related to the functionality of getAcceesToken function of redis-model.js

```

RedisModel.prototype.getAccessToken = function (userId, callback) {
    var key = conf.tokenPrefix + userId.toString();
    var that = this;
    var hashKey = conf.tokenHashKey + userId.toString();
    this.redisClient.hget(key, hashKey, function (err, reply) {
        if (err) {
            console.log(className + "GetAccessToken: " + "Error " + err);
            callback(true, messages.redis_set_error);
        }
        if (reply === null) {
            callback(null, reply);
        } else {
            that.redisClient.ttl(key, function (err, ttl) {
                if (err) {
                    callback(true, messages.redis_set_error);
                }
                var accessTokenObj = {
                    accessToken : reply,
                    expiresIn: ttl
                };
                callback(null, accessTokenObj);
            });
        }
    });
}

```



```
});
}
```

The `getAccessToken` function of the `redis-model` will create a **key** and a **hashKey**. The key will be created by concatenating **tokenPrefix** which is configured in the `config.json` and user id which is passed to the function. The `hashKey` will be created by concatenating **tokenHashKey** and the user id. The key is used as the redis key which is used to store access token for the given user. The access tokens in the redis server is encrypted so the **tokenHashKey** is used to decrypt them. The `npm-redis` `hget` function is used to find the stored access token in the redis server by passing key and the `hashKey`. The validity period of the access token is obtained by passing key to the `ttl` function of `npm-redis`.

The new access token will be created by calling **createAccessToken** function of the `redis-model.js`. The next code lines are related to the `createAccessToken` function.

```
RedisModel.prototype.createAccessToken = function (callback) {
  var accessTokenObj = {
    accessToken: crypto.randomBytes(20).toString('hex'),
    expiresIn: conf.expireTimeInSeconds
  };
  callback(accessTokenObj);
};
```

The `createAccessToken` function of `redis-model` generates access tokens by using `npm-crypto`'s **randomBytes** function. The default expiration time for the access token is configured in the `config.json` file. Before sending new access token to the user, it should be stored in the redis server. The **setAccessToken** function of the `redis-model.js` is used for that purpose. The user id and the access token object received from the `createAccessToken` function should be passed to this function. As well as in `getAccessToken` function key and `hashKey` will be generated. The `npm-redis` `hset` function is used to store access token in the redis server. The key, the `hashKey` and the access token should be passed to the `hset` function.

Every access token has a timeout. Hence after storing the access token within the redis server timeout should be set. The **expire** function of the `npm-redis` is used for that purpose. The key and the expiration time which is configured in the `config.json` should be passed to this function. Following code lines are related to the `setAccessToken` function of `redis-model.js`.

```
RedisModel.prototype.setAccessToken = function (userId, accessTokenObj, callback) {
  var that = this;
  var hashKey = conf.tokenHashKey + userId.toString();
  var key = conf.tokenPrefix + userId.toString();
  this.redisClient.hset(key, hashKey, accessTokenObj.accessToken, function (err1) {
    if (err1) {
      console.log(className + "SetAccessToken: " + "Error " + err1);
      callback(true, messages.redis_set_error);
    }
    that.redisClient.expire(key, conf.expireTimeInSeconds.toString(), function(err, result) {
      if (err) {
        callback(true, messages.redis_set_error);
      }
      callback(null, null);
    });
  });
};
```

The [Figure 14](#) shows how cURL command has been used on Linux terminal to obtain an access token from the authentication server by submitting valid user credentials.

```
asitha@asitha-Inspiron:~/private$ curl -X POST -H "Content-Type: application/json" -d '{"id": 1, "password": "abc123"}' "https://auth.testcom.org:9999/getAccessToken" --cacert /home/asitha/Documents/certs/authserver.crt -i
HTTP/1.1 200 OK
Content-Type: application/json
Date: Wed, 18 Jan 2017 15:23:29 GMT
Connection: keep-alive
Transfer-Encoding: chunked

{"accessToken": "3ba032b3f0af62d0936febe99b91bca70258d888", "expiresIn": 127}asitha@asitha-Inspiron:~/private$
```

Figure 14: Issuing cURL request to obtain access token

Access Token Validation

Access token validation required to the Docker Host to verify a given access token is valid and coupled only with a given user. The access token validation happens when the authentication server receives an http POST request to **/checkValidity** URL with a user id and an access token. The user id and the access token should be included in the request body. If the request in the proper format the authentication server will validate the access token. The [Figure 15](#) shows the access token validation process in the auth-server.js.

```
case '/checkValidity':
  redisModel.getAccessToken(reqBody.id, function(redisGetError, token_reply){
    if (redisGetError) {
      _sendResponse(token_reply.httpStatusCode, default_header, token_reply.messageBody);
    } else if (token_reply === null) {
      var error = messages.invalid_access_token;
      _sendResponse(error.httpStatusCode, default_header, error.messageBody);
    } else {
      if (token_reply.accessToken === reqBody.accessToken) {
        _sendResponse(204, default_header, {});
      } else {
        var error = messages.invalid_access_token;
        _sendResponse(error.httpStatusCode, default_header, error.messageBody);
      }
    }
  });
  break;
```

Figure 15: Access token validation flow

First the access token validation flow will find whether or not an access token is coupled with the given user id (**reqBody.id**) by calling the method **getAccessToken** in the redis-model.js. If that function returns an error, response with a particular error will be sent back. If an access token is not coupled with the given user id or access token in the redis server and the received access token mismatched between each other response with an invalid access token error will be sent back. Otherwise success response without any content will be sent. The [Figure 16](#) shows how cURL command has been used on Linux terminal to validate an access token from the authentication server.

```
asitha@asitha-Inspiron:~/private$ curl -X POST -H "Content-Type: application/json" -d '{"id": 1, "accessToken": "edcc2aa27cd7638828cdcc5510ec36f02d64fe8c"}' "https://auth.testcom.org:9999/checkValidity" --cacert /home/asitha/Documents/certs/authserver.crt -i
HTTP/1.1 204 No Content
Content-Type: application/json
Date: Wed, 18 Jan 2017 16:52:17 GMT
Connection: keep-alive

asitha@asitha-Inspiron:~/private$
```

Figure 16: Issuing cURL command to validate an access token

Proxy Server

Overview

Proxy server operates as a REST API service which runs on Ubuntu 14.04 LTS operating system and written by using NodeJS. It acts a reverse proxy for the Docker Remote API. It operates in the same host where the Docker Remote API is running and it listens to the request from the users, verify the access tokens in the request header and pass the request to the Docker Remote API. Its main running script is **index.js**, which starts the proxy server by invoking server start function. The proxy server's resource files are organized according to standard NodeJS server development directory structure as mentioned in the [Figure 17](#).

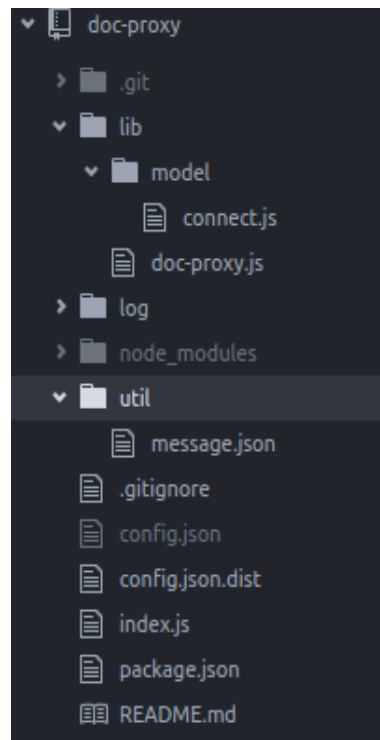


Figure 17: File and directory structure of doc-proxy

All the files which regard to the functionalities of the proxy server are kept under the **lib** directory. Inside the lib directory there is a file called **doc-proxy.js** which contains the logic related to server initiation and listening. There is another directory named **model** which rests inside the lib directory which contains another resource files. The **connect.js** which handles the interactions with the authentication server and the file system. All the proxy server configurations are being kept in the **config.json** file and the other supportive resource file are being kept under the **util** directory. There are several node package modules has been installed under the **node_modules** directory to accomplish the operations of the proxy server. They are,

1. npm-https – which is used create server which supports https.
2. npm-request - which is used to interact with the authority server and send requests to it.
3. npm-fs – which is used to perform file system operations.
4. npm-path – which is used to handle the resource files, relative and absolute path operations.

Proxy Server Initiation

Proxy server will be started by running the main script, **index.js** by issuing the command `node index.js`. It invokes start function in the doc-proxy.js. Below mentioned code lines indicates the contents of the index.js and start function of the doc-proxy.js respectively.

```
'use strict';
var docProxy = require('./lib/doc-proxy');
docProxy.start(function(error){
  if (error) {
    console.log("Unable to start doc-proxy");
  }
})

exports.start = function(callback) {
  server.listen(conf.serverPort, conf.serverUrl);
  console.log("Doc-Proxy listening at " + conf.serverUrl + ":" + conf.serverPort);
  callback();
}
```

The above mentioned start function, the **server** is an instance of npm-https **createServer** class. It invokes the **listen** function of that instance. The proxy server listening address and the listening port is configure in the **config.json** as the **serverAddress** and the **serverPort**. Both of them should be passed to the listen function.

Proxy Server Functionality

Proxy server functionality is implemented mainly in the **doc-proxy.js**, and **model/connect.js**. In the doc-proxy.js, it creates an instance of npm-https createServer class which is capable of responding to the requests which is designated to the Docker Remote API. The next few code lines are related to the doc-proxy.js, which are importing of the required node package modules, **npm-https** and **npm-fs**, system configurations and utilities. Furthermore it creates an instance of model/connect.js.

```
'use-strict';
var https = require('https');
var conf = require('./config.json');
var connector = require('./model/connect').connect();
var messages = require('./util/message.json');
var fs = require('fs');
```

The error responses from the proxy server sent as JSON format. Hence within the doc-proxy.js 'Content-Type' header has been set to **application/json** by default. The npm-https createServer requires a NodeJS object which contains the server's SSL private key, SSL certificate and the passphrase of the key to create createServer object. The paths to the SSL private key and the SSL certificate and the passphrase of the SSL private key are being configured in the config.json. The below mentioned code block is related to those operations.

```
var default_header = {'Content-Type': 'application/json'};
var privateKey = fs.readFileSync(conf.sslPrivateKeyPath);
var certificate = fs.readFileSync(conf.sslCertificatePath);
var credentials = {
  key: privateKey,
```

```

    cert: certificate,
    passphrase: conf.certificatePassphrase
  };

```

In the `doc-proxy.js`, it creates `npm-https` server instance as mentioned in the next code lines. The credentials object which is previously created should be passed, in the callback function it gives the request (**req**) and the response (**res**). Inside **server** instance all functionalities of the proxy server has being handled. Inside the `createServer` instance, when it receives a request from the user first extract the request body from the request. Then checks the request method and the request body contents. If the request method (**reqMethod**) not equals to `http GET` and request body (**chunk**) is not empty it will create a NodeJS object by using **JSON.parse()** method. After that it checks the request headers (**reqHeaders**) for the authorization header. Finally it validates the access token in the authorization header, bases on response from the authentication server, either pass the request to the Docker Remote API or reject the request.

The **_sendResponse** function, which rests inside the **server** is used to send the response back to the user request. It should be passed three parameters to the `_sendResponse` function. They are `statusCode`, `header` and `resBody`. The **statusCode** should be a valid status code which is defined in RFC 2616. The **header** is a NodeJS object which contains all the response headers. The **resBody** is a NodeJS object which contains the response body to be sent.

```

var server = https.createServer(credentials, function(req, res){
  var reqMethod = req.method;
  var reqHeaders = req.headers;
  var chunk = "";
  req.on('data', function (tempChunk) {
    chunk += tempChunk;
  });
  try {
    var reqBody = {}
    if (reqMethod !== 'GET' && chunk !== "") {
      reqBody = JSON.parse(chunk);
    }
    req.on('end', function(){
      // Authorization header validation
      // Passing the request to the Docker Remote API
    });
  } catch (e) {
    var error = messages.proxy_server_error;
    _sendResponse(error.statusCode, default_header, error.messageBody)
  }
  function _sendResponse(httpStatusCode, header, resBody) {
    res.writeHead(httpStatusCode, header);
    res.end(JSON.stringify(resBody));
  }
});

```

Authorization header validation

Proxy server validates the authorization header, in order verify that the authorization header is in the proper format. The authorization header should composed of a valid user id and a valid access token. If the validation of the authorization header succeeded, proxy server will validate the access token. If the validation of the authorization header failed, error will be output. The below mentioned code lines are related to the functionality of authorization header validation.

```

if (!reqHeaders.authorization) {
  var error = messages.authorization_header_not_set;
  _sendResponse(error.statusCode, default_header, error.messageBody);
  return;
}
var authorization = reqHeaders.authorization.split(' ');
if (authorization.length !== 3) {
  var error = messages.invalid_authorization_header;
  _sendResponse(error.statusCode, default_header, error.messageBody);
  return;
}

```

If the authorization header not set in the request it aborts the operation and response with an error, and also if authorization header is not in proper format it aborts the operation and returns an error. If the validation header proper format `reqHeaders.authorization.split(' ')` will return an array similar to below.

```
var authorization = ['Bearer', '101', 'bf9661defa3daecacfde5bde0214c4a439351d4d']
```

Access Token Verification and Passing Request to Docker

After validating the authorization header, access token which is extracted should be validated against the user id. Hence http POST request will be made to the authentication server's `/checkValidity` endpoint. If the access token is valid request will be passed to the Docker Remote API. Otherwise response will be returned with an error. The following code lines related to the access token verification and passing the request to the Docker Remote API.

```

var authOptData = {
  id: authorization[1],
  accessToken: authorization[2]
};
var authOpt = {
  url: conf.authServerHost + ":" + conf.authServerPort + conf.validationUrl,
  method: 'post',
  json: true,
  body: authOptData,
  cert: fs.readFileSync(conf.authServerCert),
  rejectUnauthorized: false,
  timeout: conf.requestTimeout
};
connector.makeCall(authOpt, function(authErr, authStatusCode, authResHeader, authData){
  if (authErr) {
    _sendResponse(authStatusCode, default_header, authData);
  } else {
    var options = {
      url: conf.dockerHost + ':' + conf.dockerPort + req.url,
      method: req.method,
      headers: reqHeaders,
      json: true,
      body: reqBody,
      timeout: conf.requestTimeout
    };
    connector.makeCall(options, function(docErr, statusCode, resHeader, docData) {
      if (docErr) {
        _sendResponse(statusCode, default_header, docData);
      }
    });
  }
});

```

```

    }
    if (resHeader['content-length'] != '') {
        delete resHeader['content-length'];
    }
    if (conf.logging) {
        connector.writeLog(authorization[1], req.url, function(err) {
            _sendResponse(statusCode, resHeader, docData);
        });
    } else {
        _sendResponse(statusCode, resHeader, docData);
    }
    });
}
});

```

The above mentioned code lines are related to the request object's end event, **req.on(end, function() { })**. The **authOpt** object is created to call to the authentication server to validate the received access token. It includes the URL which is authentication server's /checkValidity endpoint, request method which is http POST, request body which is another object contains user id and the access token to be validated and SSL client certificate of the authentication server. In order to make requests to the remote resources **makeCall** function of **model/connect.js** is used. The **connector.makeCall** function uses **npm-request** node module make request to the remote resources. The makeCall function has mentioned in next few code lines.

```

Connector.prototype.makeCall = function(options, callback) {
    request(options, function(err, res, body) {
        if (err) {
            console.log(err);
            var error = messages.request_failed;
            callback(true, error.httpStatusCode, {}, error.messageBody);
        } else if (res.statusCode > 300) {
            callback(true, res.statusCode, res.headers, body);
        } else {
            callback(null, res.statusCode, res.headers, body)
        }
    });
}

```

The **options** variable is the NodeJS object that is need to make the request to the remote server. The **authOpt** object should be passed to the makeCall function in validating the access token. If access token validation succeeded, Docker Remote API will be called base on the user request by using the connector.makeCall function. In this occasion different set of options will be passed to the makeCall function which are relevant to the accessing of Docker Remote API. The response from the Docker Remote API will be passed to the user.

There are is special configuration available in the config.json which by default disabled, to enable and disable the logging of the user activities. If the logging is enabled before sending the response to the user proxy server will create a recode for the user activity by using **connector.writeLog** function. The code lines related to the writeLog function has mentioned below.

```

Connector.prototype.writeLog = function (userId, endPoint, callback) {
  var logfile = path.join(__dirname, '..', '..', 'log', conf.userPrefix + userId + ".log");
  if (!fs.existsSync(logfile)) {
    fs.closeSync(fs.openSync(logfile, 'w'));
  }
  var entry = "Operation = " + endPoint + " on " + new Date() + '\n';
  fs.appendFile(logfile, entry, function(err) {
    if (err) {
      console.log(err);
    }
    callback(null);
  });
}

```

There are two variables that should be passed to the writeLog function, user id which is relevant to the user who try to make the request to the Docker Remote API and the requested endpoint of the Docker Remote API. Every user has a unique log file in the **log** directory. This log file will be created when a user make request for the first time. After that every record will be entered to the same file.

Self-Signed Certificate Generation

Connections between the authentication server and the user, the proxy server and the user and the proxy server and the authentication server must be protected from the third party interventions. In order to achieve that all the connections are implemented with Secure Socket Layer (SSL). Self-signed certificates has been used for the SSL and npm-https package has been used for the server implementation instead of npm-http. The self-signed certificate are generated as follows by using **openssl** command. Two key certificate pairs are generated for the authentication server and the proxy server.

openssl genrsa -des3 -out authserver.key 2048

This command will generate the key for the Certificate Signing Request.

openssl req -new -key authserver.key -out authserver.csr

This command will create Certificate Signing Request

openssl x509 -req -days 365 -in authserver.csr -signkey authserver.key -out authserver.crt

This command will generate self-signed certificate.

Similar to creating self-signed certificate for the authentication server which is authserver.crt, a self-signed certificate is generated for the proxy server named proxyserver.crt

Conclusion

The implementation of the authentication server and the proxy server is done to achieve main objectives of the project. But when selecting programming languages, selecting data storages and developing algorithms the usability, the performance and the interoperability constraints were taken into account. The main reason select the NodeJS for server programming is to take the advantage from its asymmetric event handling procedure. Redis server is selected because of its nature of keeping data in memory rather than disk. Both of these selections helped to prevent introduction of large amount of latency to the request-response circle.

CHAPTER 05 – Evaluation

CHAPTER 05 – Evaluation

Introduction

This chapter has described how the proposed system's components has been operated in the test environment. Further this chapter includes test results obtained under the test environment. Further it includes comparisons between expected results and the actual results. Furthermore it includes limitations of the systems usage. Finally it includes the final conclusions about the evaluation.

Test Environment

The test environment is composed with two Ubuntu 14.04 LTS virtual machines where one virtual machine has configured to run authentication server with MySQL user database and Redis while the other virtual machine act as the user's workstation which needs to interact with the Docker Remote API. The Docker is installed on the host machine of the virtual machines with required Docker images and containers along with the proxy server. The `'sudo docker daemon -H tcp://127.0.0.1:8888'` command is used to make the Docker daemon listen on TCP port 8888 from the requests originated from the localhost. The `'node index.js'` command is used to start the proxy server which made proxy server listening on tcp port 9000 on the host machine.

The virtual machine is used to run the authentication server is setup with NodeJS and npm modules, Redis server and MySQL. The `'node index.js'` command is used to start the authentication server which made authentication server listening on TCP port 9999. The other virtual machine which is used as the user workstation is composed with the Postman REST client and cURL application which are used to make request to the authentication server and the proxy server.

There were two self-signed certificates were generated for the proxy server and the authentication server. In generating those certificates it had to enter the FQDN parameter for both certificates. The **auth.testcom.org** is used as the authentication server certificate FQDN and the **api.dockerhost.org** is used as the proxy server certificate FQDN. Equal values were set for other parameters. The authentication server's self-signed placed inside its virtual machine and the proxy server's self-signed placed in the host machine of virtual machines.

Evaluation procedure

The evaluation procedure done basically in two steps. In the first step selected set of API calls are made to the Docker Remote API without using authentication server and the proxy server. In this step, API calls are made directly to the Docker Remote API without going through the proxy server. The `'sudo docker daemon -H tcp://0.0.0.0:8888'` command is used to make the Docker daemon listen on TCP port 8888 from the requests originated from any host. In the second step, the `'sudo docker daemon -H tcp://127.0.0.1:8888'` command is used to make the Docker daemon listen on TCP port 8888 for the requests originated only from the localhost. The same set of API requests which were used in the first step will be made to the Docker Remote API.

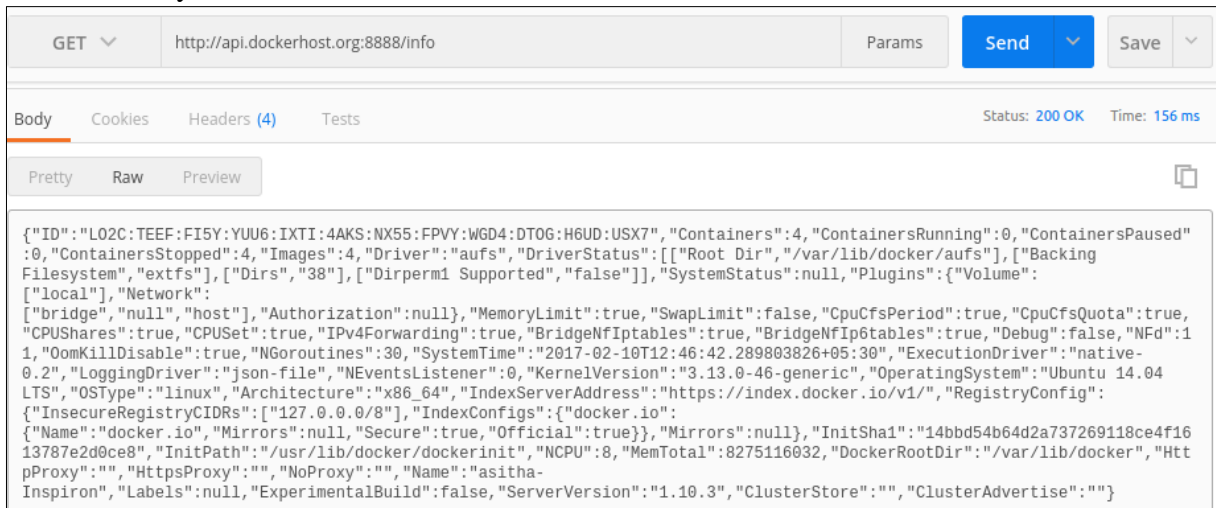
The main reason behind conducting the evaluation in two steps is to identify whether any irregularities has been introduced by the authentication procedure when a user accessing Docker Remote API. Other than that, it was needed to measure to what extent the introduction of new authentication mechanism has affected to the effectiveness in accessing Docker Remote API. The requests were sent by using cURL commands and the **Postman** which is a REST client application provided by the Google Chrome web browser as an extension and a desktop application. The sample set of requests to be made to the Docker Remote API was chosen in manner that covers all the operational areas of Docker operations which are controllers, images and miscellaneous.

Evaluation Results

This sub section includes some of the results relevant to the requests which are made to Docker Remote API directly and via the proxy server with authentication server separately for a selected set of requests.

GET /info

The successful http GET request to this endpoint gives the system wide information of the Docker Host. The [Figure 18](#) mentioned below is relevant to a response which receives for a successful request made to the Docker Remote API when it is open to the outside without authentication. The [Figure 19](#) mentioned below is relevant to a response which receives for successful request made to the Docker Remote API via Proxy server after validating access token, which is embedded in the Authorization header with the authentication server. The general response time when Docker Remote API is opened to outside without authentication was fluctuated from 150 milliseconds to 160 milliseconds while the general response time when Docker Remote API only accessible via proxy server with authentication was fluctuated from 160 milliseconds to 170 milliseconds. In both scenarios the structure of the response body remain exactly the same.



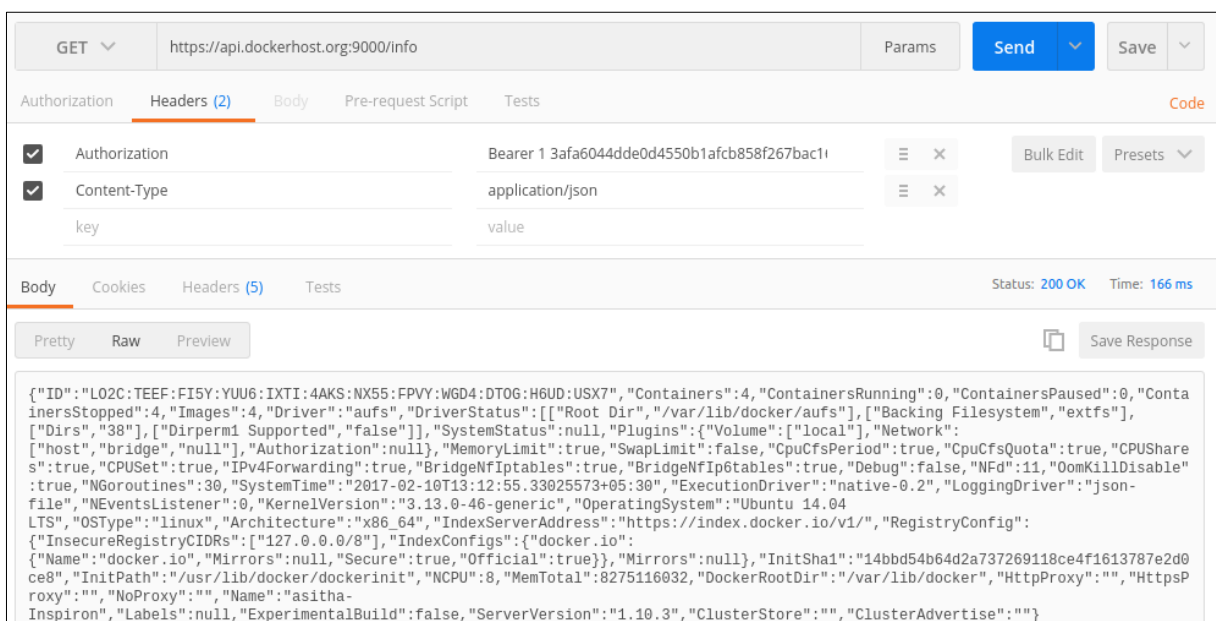
GET Params

Body Cookies Headers (4) Tests Status: 200 OK Time: 156 ms

Pretty Raw Preview

```
{
  "ID": "L02C:TEEF:FI5Y:YUU6:IXTI:4AKS:NX55:FPVY:WGD4:DT0G:H6UD:USX7",
  "Containers": 4,
  "ContainersRunning": 0,
  "ContainersPaused": 0,
  "ContainersStopped": 4,
  "Images": 4,
  "Driver": "aufs",
  "DriverStatus": [
    [
      "Root Dir",
      "/var/lib/docker/aufs"
    ],
    [
      "Backing Filesystem",
      "extfs"
    ],
    [
      "Dirs",
      "38"
    ],
    [
      "Dirperm1 Supported",
      "false"
    ]
  ],
  "SystemStatus": null,
  "Plugins": {
    "Volume": [
      "local"
    ],
    "Network": [
      "bridge",
      "null",
      "host"
    ],
    "Authorization": null,
    "MemoryLimit": true,
    "SwapLimit": false,
    "CpuCfsPeriod": true,
    "CpuCfsQuota": true,
    "CPUShares": true,
    "CPUSet": true,
    "IPv4Forwarding": true,
    "BridgeNfIptables": true,
    "BridgeNfIp6tables": true,
    "Debug": false,
    "NFD": 11,
    "OomKillDisable": true,
    "NGoroutines": 30,
    "SystemTime": "2017-02-10T12:46:42.289803826+05:30",
    "ExecutionDriver": "native-0.2",
    "LoggingDriver": "json-file",
    "NEventsListener": 0,
    "KernelVersion": "3.13.0-46-generic",
    "OperatingSystem": "Ubuntu 14.04 LTS",
    "OSType": "linux",
    "Architecture": "x86_64",
    "IndexServerAddress": "https://index.docker.io/v1/",
    "RegistryConfig": {
      "InsecureRegistryCIDRs": [
        "127.0.0.0/8"
      ],
      "IndexConfigs": {
        "docker.io": {
          "Name": "docker.io",
          "Mirrors": null,
          "Secure": true,
          "Official": true,
          "Mirrors": null,
          "InitSha1": "14bbd54b64d2a737269118ce4f1613787e2d0ce8",
          "InitPath": "/usr/lib/docker/dockerinit",
          "NCPU": 8,
          "MemTotal": 8275116032,
          "DockerRootDir": "/var/lib/docker",
          "HttpProxy": "",
          "HttpsProxy": "",
          "NoProxy": "",
          "Name": "asitha-Inspiron",
          "Labels": null,
          "ExperimentalBuild": false,
          "ServerVersion": "1.10.3",
          "ClusterStore": "",
          "ClusterAdvertise": ""
        }
      }
    }
  }
}
```

Figure 18: A request made to the /info endpoint directly by using Postman REST client



GET Params

Authorization Headers (2) Body Pre-request Script Tests Code

Authorization Bearer 13afa6044de0d4550b1afc858f267bac11

Content-Type application/json

key value

Body Cookies Headers (5) Tests Status: 200 OK Time: 166 ms

Pretty Raw Preview

```
{
  "ID": "L02C:TEEF:FI5Y:YUU6:IXTI:4AKS:NX55:FPVY:WGD4:DT0G:H6UD:USX7",
  "Containers": 4,
  "ContainersRunning": 0,
  "ContainersPaused": 0,
  "ContainersStopped": 4,
  "Images": 4,
  "Driver": "aufs",
  "DriverStatus": [
    [
      "Root Dir",
      "/var/lib/docker/aufs"
    ],
    [
      "Backing Filesystem",
      "extfs"
    ],
    [
      "Dirs",
      "38"
    ],
    [
      "Dirperm1 Supported",
      "false"
    ]
  ],
  "SystemStatus": null,
  "Plugins": {
    "Volume": [
      "local"
    ],
    "Network": [
      "bridge",
      "null",
      "host"
    ],
    "Authorization": null,
    "MemoryLimit": true,
    "SwapLimit": false,
    "CpuCfsPeriod": true,
    "CpuCfsQuota": true,
    "CPUShares": true,
    "CPUSet": true,
    "IPv4Forwarding": true,
    "BridgeNfIptables": true,
    "BridgeNfIp6tables": true,
    "Debug": false,
    "NFD": 11,
    "OomKillDisable": true,
    "NGoroutines": 30,
    "SystemTime": "2017-02-10T13:12:55.33025573+05:30",
    "ExecutionDriver": "native-0.2",
    "LoggingDriver": "json-file",
    "NEventsListener": 0,
    "KernelVersion": "3.13.0-46-generic",
    "OperatingSystem": "Ubuntu 14.04 LTS",
    "OSType": "linux",
    "Architecture": "x86_64",
    "IndexServerAddress": "https://index.docker.io/v1/",
    "RegistryConfig": {
      "InsecureRegistryCIDRs": [
        "127.0.0.0/8"
      ],
      "IndexConfigs": {
        "docker.io": {
          "Name": "docker.io",
          "Mirrors": null,
          "Secure": true,
          "Official": true,
          "Mirrors": null,
          "InitSha1": "14bbd54b64d2a737269118ce4f1613787e2d0ce8",
          "InitPath": "/usr/lib/docker/dockerinit",
          "NCPU": 8,
          "MemTotal": 8275116032,
          "DockerRootDir": "/var/lib/docker",
          "HttpProxy": "",
          "HttpsProxy": "",
          "NoProxy": "",
          "Name": "asitha-Inspiron",
          "Labels": null,
          "ExperimentalBuild": false,
          "ServerVersion": "1.10.3",
          "ClusterStore": "",
          "ClusterAdvertise": ""
        }
      }
    }
  }
}
```

Figure 19: A request made to the /info endpoint via proxy server by using Postman REST client

GET /containers/json

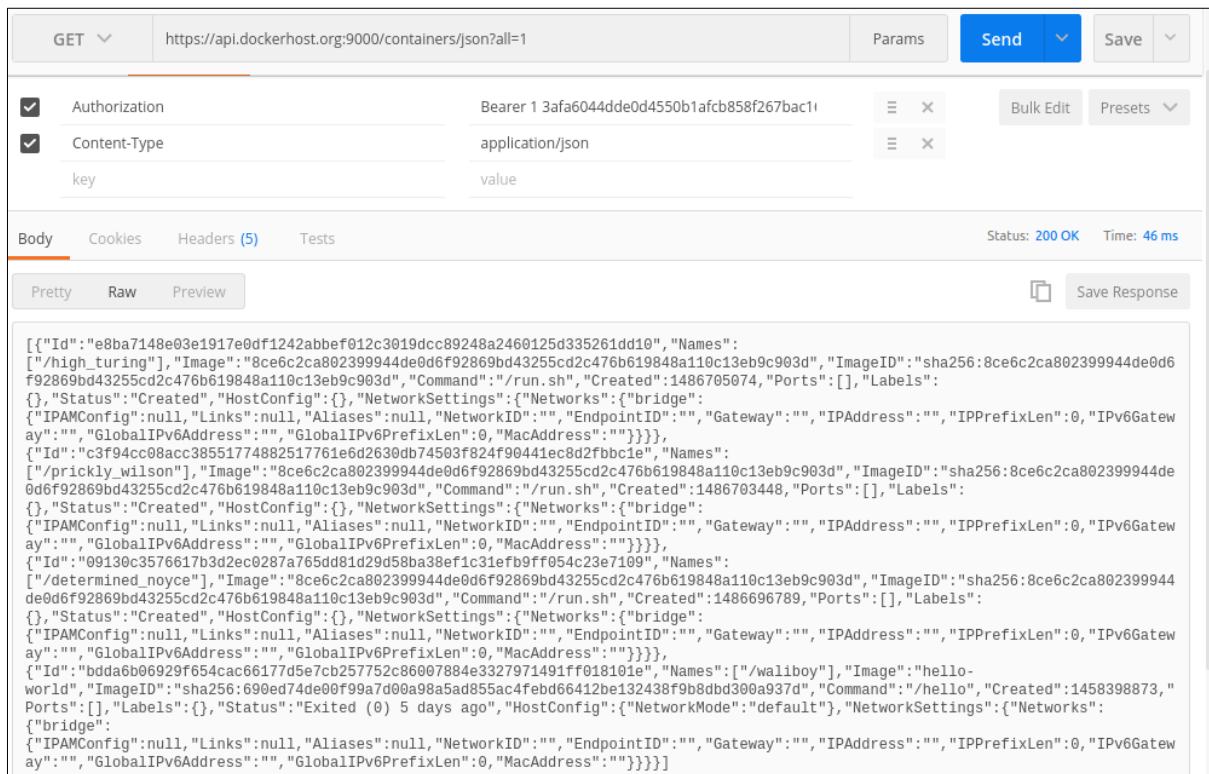
The successful http GET request to this endpoint lists the available containers of the Docker Host. The [Figure 20](#) mentioned below is relevant to a response which receives for a successful request made to the Docker Remote API when it is open to the outside without authentication. The [Figure 21](#) mentioned below is relevant to a response which receives for successful request made to the Docker Remote API via Proxy server after validating access token, which is embedded in the Authorization header with the authentication server. The general response time when Docker Remote API is opened to outside without authentication was fluctuated from 30 milliseconds to 40 milliseconds while the general response time when Docker Remote API only accessible via proxy server with authentication was fluctuated from 40 milliseconds to 50 milliseconds. In both scenarios the structure of the response body remain exactly the same.

The screenshot shows a REST client interface with the following details:

- Method:** GET
- URL:** http://api.dockerhost.org:8888/containers/json?all=1
- Status:** 200 OK
- Time:** 38 ms
- Response Body (Pretty):**

```
[{"Id": "e8ba7148e03e1917e0df1242abbe012c3019dcc89248a2460125d335261dd10", "Names":
["/high_turing"], "Image": "8ce6c2ca802399944de0d6f92869bd43255cd2c476b619848a110c13eb9c903d", "ImageID": "sha256:8ce6c2ca802399944de0d6f92869bd43255cd2c476b619848a110c13eb9c903d", "Command": "/run.sh", "Created": 1486705074, "Ports": [], "Labels":
{}}, {"Status": "Created", "HostConfig": {}, "NetworkSettings": {"Networks": {"bridge":
{"IPAMConfig": null, "Links": null, "Aliases": null, "NetworkID": "", "EndpointID": "", "Gateway": "", "IPAddress": "", "IPPrefixLen": 0, "IPv6Gateway": "", "GlobalIPv6Address": "", "GlobalIPv6PrefixLen": 0, "MacAddress": ""}}}},
{"Id": "c3f94cc08acc38551774882517761e6d2630db74503f824f90441ec8d2fbbc1e", "Names":
["/prickly_wilson"], "Image": "8ce6c2ca802399944de0d6f92869bd43255cd2c476b619848a110c13eb9c903d", "ImageID": "sha256:8ce6c2ca802399944de0d6f92869bd43255cd2c476b619848a110c13eb9c903d", "Command": "/run.sh", "Created": 1486703448, "Ports": [], "Labels":
{}}, {"Status": "Created", "HostConfig": {}, "NetworkSettings": {"Networks": {"bridge":
{"IPAMConfig": null, "Links": null, "Aliases": null, "NetworkID": "", "EndpointID": "", "Gateway": "", "IPAddress": "", "IPPrefixLen": 0, "IPv6Gateway": "", "GlobalIPv6Address": "", "GlobalIPv6PrefixLen": 0, "MacAddress": ""}}}},
{"Id": "09130c3576617b3d2ec0287a765dd81d29d58ba38ef1c31efb9ff054c23e7109", "Names":
["/determined_noyce"], "Image": "8ce6c2ca802399944de0d6f92869bd43255cd2c476b619848a110c13eb9c903d", "ImageID": "sha256:8ce6c2ca802399944de0d6f92869bd43255cd2c476b619848a110c13eb9c903d", "Command": "/run.sh", "Created": 1486696789, "Ports": [], "Labels":
{}}, {"Status": "Created", "HostConfig": {}, "NetworkSettings": {"Networks": {"bridge":
{"IPAMConfig": null, "Links": null, "Aliases": null, "NetworkID": "", "EndpointID": "", "Gateway": "", "IPAddress": "", "IPPrefixLen": 0, "IPv6Gateway": "", "GlobalIPv6Address": "", "GlobalIPv6PrefixLen": 0, "MacAddress": ""}}}},
{"Id": "bdda6b06929f654cac66177d5e7cb257752c86007884e3327971491ff018101e", "Names": ["/waliboy"], "Image": "hello-world", "ImageID": "sha256:690ed74de00f99a7d00a98a5ad855ac4feb66412be132438f9b8dbd300a937d", "Command": "/hello", "Created": 1458398873, "Ports": [], "Labels": {}, "Status": "Exited (0) 5 days ago", "HostConfig": {"NetworkMode": "default"}, "NetworkSettings": {"Networks": {"bridge": {"IPAMConfig": null, "Links": null, "Aliases": null, "NetworkID": "", "EndpointID": "", "Gateway": "", "IPAddress": "", "IPPrefixLen": 0, "IPv6Gateway": "", "GlobalIPv6Address": "", "GlobalIPv6PrefixLen": 0, "MacAddress": ""}}}}]
```

Figure 20: A request made to the /containers/json endpoint directly by using Postman REST client



GET ▼ https://api.dockerhost.org:9000/containers/json?all=1 Params Send Save ▼

Authorization Bearer 1 3afa6044dde0d4550b1afcb858f267bac1 ⋮ × Bulk Edit Presets ▼

Content-Type application/json ⋮ ×

key value

Body Cookies Headers (5) Tests Status: 200 OK Time: 46 ms

Pretty Raw Preview Save Response

```
[{"Id": "e8ba7148e03e1917e0df1242abbe012c3019dcc89248a2460125d335261dd10", "Names": ["/high_turing"], "Image": "8ce6c2ca802399944de0d6f92869bd43255cd2c476b619848a110c13eb9c903d", "ImageID": "sha256:8ce6c2ca802399944de0d6f92869bd43255cd2c476b619848a110c13eb9c903d", "Command": "/run.sh", "Created": 1486705074, "Ports": [], "Labels": {}, "Status": "Created", "HostConfig": {}, "NetworkSettings": {"Networks": {"bridge": {"IPAMConfig": null, "Links": null, "Aliases": null, "NetworkID": "", "EndpointID": "", "Gateway": "", "IPAddress": "", "IPPrefixLen": 0, "IPv6Gateway": "", "GlobalIPv6Address": "", "GlobalIPv6PrefixLen": 0, "MacAddress": ""}}}}, {"Id": "c3f94cc08acc38551774882517761e6d2630db74503f824f90441ec8d2fbbc1e", "Names": ["/prickly_wilson"], "Image": "8ce6c2ca802399944de0d6f92869bd43255cd2c476b619848a110c13eb9c903d", "ImageID": "sha256:8ce6c2ca802399944de0d6f92869bd43255cd2c476b619848a110c13eb9c903d", "Command": "/run.sh", "Created": 1486703448, "Ports": [], "Labels": {}, "Status": "Created", "HostConfig": {}, "NetworkSettings": {"Networks": {"bridge": {"IPAMConfig": null, "Links": null, "Aliases": null, "NetworkID": "", "EndpointID": "", "Gateway": "", "IPAddress": "", "IPPrefixLen": 0, "IPv6Gateway": "", "GlobalIPv6Address": "", "GlobalIPv6PrefixLen": 0, "MacAddress": ""}}}}, {"Id": "09130c3576617b3d2ec0287a765dd81d29d58ba38ef1c31efb9ff054c23e7109", "Names": ["/determined_noyce"], "Image": "8ce6c2ca802399944de0d6f92869bd43255cd2c476b619848a110c13eb9c903d", "ImageID": "sha256:8ce6c2ca802399944de0d6f92869bd43255cd2c476b619848a110c13eb9c903d", "Command": "/run.sh", "Created": 1486696789, "Ports": [], "Labels": {}, "Status": "Created", "HostConfig": {}, "NetworkSettings": {"Networks": {"bridge": {"IPAMConfig": null, "Links": null, "Aliases": null, "NetworkID": "", "EndpointID": "", "Gateway": "", "IPAddress": "", "IPPrefixLen": 0, "IPv6Gateway": "", "GlobalIPv6Address": "", "GlobalIPv6PrefixLen": 0, "MacAddress": ""}}}}, {"Id": "bdda6b06929f654cac66177d5e7cb257752c86097884e3327971491ff018101e", "Names": ["/waliboy"], "Image": "hello-world", "ImageID": "sha256:690ed74de00f99a7d00a98a5ad855ac4febdb66412be132438f9b8dbd300a937d", "Command": "/hello", "Created": 1458398873, "Ports": [], "Labels": {}, "Status": "Exited (0) 5 days ago", "HostConfig": {"NetworkMode": "default"}, "NetworkSettings": {"Networks": {"bridge": {"IPAMConfig": null, "Links": null, "Aliases": null, "NetworkID": "", "EndpointID": "", "Gateway": "", "IPAddress": "", "IPPrefixLen": 0, "IPv6Gateway": "", "GlobalIPv6Address": "", "GlobalIPv6PrefixLen": 0, "MacAddress": ""}}}}]
```

Figure 21: A request made to the /containers/json endpoint via proxy server by using Postman REST client

GET /version

The successful http GET request to this endpoint returns the Docker version information. The [Figure 22](#) mentioned below is relevant to a response which receives for a successful request made to the Docker Remote API when it is open to the outside without authentication. The [Figure 23](#) mentioned below is relevant to a response which receives for successful request made to the Docker Remote API via Proxy server after validating access token, which is embedded in the Authorization header with the authentication server. The general response time when Docker Remote API is opened to outside without authentication was fluctuated from 30 milliseconds to 40 milliseconds while the general response time when Docker Remote API only accessible via proxy server with authentication was fluctuated from 40 milliseconds to 50 milliseconds. In both scenarios the structure of the response body remain exactly the same.



GET ▼ http://api.dockerhost.org:8888/version Params Send Save ▼

Body Cookies Headers (4) Tests Status: 200 OK Time: 40 ms

Pretty Raw Preview JSON ⋮ Save Response

```
1 {
2   "Version": "1.10.3",
3   "ApiVersion": "1.22",
4   "GitCommit": "20f81dd",
5   "GoVersion": "go1.5.3",
6   "Os": "linux",
7   "Arch": "amd64",
8   "KernelVersion": "3.13.0-46-generic",
9   "BuildTime": "2016-03-10T15:54:52.312835708+00:00"
10 }
```

Figure 22: A request made to the /version endpoint directly by using Postman REST client

GET ▼ https://api.dockerhost.org:9000/version Params Send Save ▼

Authorization Headers (2) Body Pre-request Script Tests Code

Authorization Bearer 1 95f6ffcb1a95f9fad834e8b4308f4ef1 ⋮ × Bulk Edit Presets ▼

Content-Type application/json ⋮ ×

key value

Body Cookies Headers (5) Tests Status: 200 OK Time: 59 ms

Pretty Raw Preview JSON ▼ ≡ 📄 🔍 Save Response

```

1 {
2   "Version": "1.10.3",
3   "ApiVersion": "1.22",
4   "GitCommit": "20f81dd",
5   "GoVersion": "go1.5.3",
6   "Os": "linux",
7   "Arch": "amd64",
8   "KernelVersion": "3.13.0-46-generic",
9   "BuildTime": "2016-03-10T15:54:52.312835708+00:00"
10 }

```

Figure 23: A sample request made to the /version endpoint via proxy server by using Postman REST client

GET /images/json

The successful http GET request to this endpoint lists the Docker images available at the Docker Host. The [Figure 24](#) mentioned below is relevant to a response which receives for a successful request made to the Docker Remote API when it is open to the outside without authentication. The [Figure 25](#) mentioned below is relevant to a response which receives for successful request made to the Docker Remote API via Proxy server after validating access token which is embedded in the Authorization header with the authentication server. The general response time when Docker Remote API is opened to outside without authentication was fluctuated from 30 milliseconds to 40 milliseconds while the general response time when Docker Remote API only accessible via proxy server with authentication was fluctuated from 40 milliseconds to 50 milliseconds. In both scenarios the structure of the response body remain exactly the same.

GET ▼ http://api.dockerhost.org:8888/images/json Params Send Save ▼

Type No Auth ▼

Body Cookies Headers (4) Tests Status: 200 OK Time: 36 ms

Pretty Raw Preview 📄

```

[{"Id": "sha256:d4bacf92e2d2c0c49e91c30d8b1aa77b42b2ed7d65a7fd77eba2e98b8b3c138a", "ParentId": "", "RepoTags": ["thinkcube/mkdocs:latest"], "RepoDigests": null, "Created": 1476949220, "Size": 268586699, "VirtualSize": 268586699, "Labels": {"build-date": "20160906", "license": "GPLv2", "name": "CentOS Base Image", "vendor": "CentOS"}}, {"Id": "sha256:8ce6c2ca802399944de0d6f92869bd43255cd2c476b619848a110c13eb9c903d", "ParentId": "", "RepoTags": ["thinkcube/phpmoadmin:latest"], "RepoDigests": null, "Created": 1469681092, "Size": 25845155, "VirtualSize": 25845155, "Labels": {}}, {"Id": "sha256:690ed74de00f99a7d00a98a5ad855ac4febd66412be132438f9b8dbd300a937d", "ParentId": "", "RepoTags": ["hello-world:latest"], "RepoDigests": null, "Created": 1444780068, "Size": 960, "VirtualSize": 960, "Labels": null}, {"Id": "sha256:74b44a255c0d34ad5d1ed84c04024e31c88735cf62f74d9e9957eb8cf2fcf8db", "ParentId": "", "RepoTags": ["thinkcube/docker-panel:latest"], "RepoDigests": null, "Created": 1436777050, "Size": 701391104, "VirtualSize": 701391104, "Labels": {}}]

```

Figure 24: A request made to the /images/json endpoint directly by using Postman REST client

GET ▼ <https://api.dockerhost.org:9000/images/json> Params Send Save ▼

Authorization Bearer 1 3afa6044dde0d4550b1afcb858f267bac11 ⋮ × Bulk Edit Presets ▼

Content-Type application/json ⋮ ×

key value

Body Cookies Headers (5) Tests Status: 200 OK Time: 46 ms

Pretty Raw Preview Save Response

```
[{"Id":"sha256:d4bacf92e2d2c0c49e91c30d8b1aa77b42b2ed7d65a7fd77eba2e98b8b3c138a", "ParentId":"","RepoTags":["thinkcube/mkdocs:latest"], "RepoDigests":null, "Created":1476949220, "Size":268586699, "VirtualSize":268586699, "Labels":{"build-date":"20160906", "license":"GPLv2", "name":"CentOS Base Image", "vendor":"CentOS"}}, {"Id":"sha256:8ce6c2ca802399944de0d6f92869bd43255cd2c476b619848a110c13eb9c903d", "ParentId":"","RepoTags":["thinkcube/phpmoadmin:latest"], "RepoDigests":null, "Created":1469681092, "Size":25845155, "VirtualSize":25845155, "Labels":{}}, {"Id":"sha256:690ed74de00f99a7d00a98a5ad855ac4febd66412be132438f9b8dbd300a937d", "ParentId":"","RepoTags":["hello-world:latest"], "RepoDigests":null, "Created":1444780068, "Size":960, "VirtualSize":960, "Labels":null}, {"Id":"sha256:74b44a255c0d34ad5d1ed84c04024e31c88735cf62f74d9e9957eb8cf2fcf8db", "ParentId":"","RepoTags":["thinkcube/docker-panel:latest"], "RepoDigests":null, "Created":1436777050, "Size":701391104, "VirtualSize":701391104, "Labels":{}}
```

Figure 25: A request made to the /images/json endpoint via proxy server by using Postman REST client

GET /images/search?term=#keyword

The successful http GET request to this endpoint will search the Docker Hub for Docker Images with a given keyword and display the high level information of relevant images. The [Figure 26](#) mentioned below is relevant to a response which receives for a successful request made to the Docker Remote API when it is open to the outside without authentication. The [Figure 27](#) mentioned below is relevant to a response which receives for successful request made to the Docker Remote API via Proxy server after validating access token which is embedded in the Authorization header with the authentication server. The general response time when Docker Remote API is opened to outside without authentication was fluctuated from 1400 milliseconds to 1500 milliseconds while the general response time when Docker Remote API only accessible via proxy server with authentication was fluctuated from 1500 milliseconds to 1600 milliseconds. In both scenarios the structure of the response body remain exactly the same.

GET ▼ <http://api.dockerhost.org:8888/images/search?term=thinkcube> Params Send Save ▼

Body Cookies Headers (4) Tests Status: 200 OK Time: 1457 ms

Pretty Raw Preview Save Response

```
[{"star_count":1,"is_official":false,"name":"thinkcube/tcdrupal-base","is_trusted":true,"is_automated":true,"description":"Thinkcube Project Drupal"}, {"star_count":2,"is_official":false,"name":"thinkcube/mkdocs","is_trusted":true,"is_automated":true,"description":"Mkdocs CentOS 7 image"}, {"star_count":1,"is_official":false,"name":"thinkcube/php70","is_trusted":true,"is_automated":true,"description":"PHP 7.0 Docker Image based on CentOS7"}, {"star_count":1,"is_official":false,"name":"thinkcube/php56","is_trusted":true,"is_automated":true,"description":"PHP 5.6 Docker Image based on CentOS7"}, {"star_count":1,"is_official":false,"name":"thinkcube/php55","is_trusted":true,"is_automated":true,"description":"PHP 5.5 Docker Image based on CentOS7"}, {"star_count":1,"is_official":false,"name":"thinkcube/locxy","is_trusted":true,"is_automated":true,"description":"Frontend Reverse Proxy for all Local Containers"}, {"star_count":1,"is_official":false,"name":"thinkcube/node4","is_trusted":true,"is_automated":true,"description":"Nodejs4 docker image"}, {"star_count":0,"is_official":false,"name":"thinkcube/phpmoadmin","is_trusted":true,"is_automated":true,"description":"phpmoadmin docker image based on alpine linux"}, {"star_count":1,"is_official":false,"name":"thinkcube/node6","is_trusted":true,"is_automated":true,"description":"Based on CentOS7 Nodejs6 LTS"}, {"star_count":1,"is_official":false,"name":"thinkcube/docker-panel","is_trusted":false,"is_automated":false,"description":"Open source Docker API wrapper with nice user interfaces"}]
```

Figure 26: A request made to the /images/search endpoint directly by using Postman REST client

GET `https://api.dockerhost.org:9000/images/search?term=thinkcube` Params Send Save

Authorization Headers (2) Body Pre-request Script Tests Code

<input checked="" type="checkbox"/>	Authorization	Bearer 1 95f6fcb1a95f9fad834e8b4308f4efb96dr	⋮ ×	Bulk Edit	Presets
<input checked="" type="checkbox"/>	Content-Type	application/json	⋮ ×		
	key	value			

Body Cookies Headers (5) Tests Status: 200 OK Time: 1606 ms

Pretty Raw Preview Save Response

```
[{"star_count":1,"is_official":false,"name":"thinkcube/tcdrupal-base","is_trusted":true,"is_automated":true,"description":"Thinkcube Project Drupal"}, {"star_count":2,"is_official":false,"name":"thinkcube/mkdocs","is_trusted":true,"is_automated":true,"description":"Mkdocs CentOS 7 image"}, {"star_count":1,"is_official":false,"name":"thinkcube/php70","is_trusted":true,"is_automated":true,"description":"PHP 7.0 Docker Image based on CentOS7"}, {"star_count":1,"is_official":false,"name":"thinkcube/php56","is_trusted":true,"is_automated":true,"description":"PHP 5.6 Docker Image based on CentOS7"}, {"star_count":1,"is_official":false,"name":"thinkcube/php55","is_trusted":true,"is_automated":true,"description":"PHP 5.5 Docker Image based on CentOS7"}, {"star_count":0,"is_official":false,"name":"thinkcube/loxy","is_trusted":true,"is_automated":true,"description":"Frontend Reverse Proxy for all Local Containers"}, {"star_count":1,"is_official":false,"name":"thinkcube/node4","is_trusted":true,"is_automated":true,"description":"Nodejs4 docker image"}, {"star_count":0,"is_official":false,"name":"thinkcube/phpmoadmin","is_trusted":true,"is_automated":true,"description":"phpmoadmin docker image based on alpine linux "}, {"star_count":1,"is_official":false,"name":"thinkcube/node6","is_trusted":true,"is_automated":true,"description":"Based on CentOS7 Nodejs6 LTS"}, {"star_count":1,"is_official":false,"name":"thinkcube/docker-panel","is_trusted":false,"is_automated":false,"description":"Open source Docker API wrapper with nice user interfaces"}]
```

Figure 27: A request made to the `/images/search` endpoint via proxy server by using Postman REST client

GET `/containers/{id}/json`

The successful http GET requests to this endpoint will returns the low level information about the given container. The [Figure 28](#) mentioned below is relevant to a response which receives for a successful request made to the Docker Remote API when it is open to the outside without authentication. The [Figure 29](#) mentioned below is relevant to a response which receives for successful request made to the Docker Remote API via Proxy server after validating access token which is embedded in the Authorization header with the authentication server. The general response time when Docker Remote API is opened to outside without authentication was fluctuated from 40 milliseconds to 50 milliseconds while the general response time when Docker Remote API only accessible via proxy server with authentication was fluctuated from 50 milliseconds to 60 milliseconds. In both scenarios the structure of the response body remain exactly the same.

GET `http://api.dockerhost.org:8888/containers/bdda6b06929f/json` Params Send Save

Body Cookies Headers (4) Tests Status: 200 OK Time: 54 ms

Pretty Raw Preview Save Response

```
{"Id":"bdda6b06929f654cac66177d5e7cb257752c86007884e3327971491ff018101e","Created":"2016-03-19T14:47:53.197806691Z","Path":"/hello","Args":[],"State":{"Status":"exited","Running":false,"Paused":false,"Restarting":false,"OOMKilled":false,"Dead":false,"Pid":0,"ExitCode":0,"Error":"","Start":01-20T17:21:42.287810446Z,"FinishedAt":"2017-01-20T17:21:42.319673353Z"},"Image":{"sha256:690ed74de00f99a7d00a98a5ad855ac4febd66412be132438f9b8dbd300a937d"},"ResolvConfPath":"/etc/resolv.conf","Name":"/waliboy","RestartCount":0,"Driver":"aufs","MountLabel":"","ProcessLabel":"","AppArmorProfile":"","ExecIDs":null,"Binds":null,"ContainerIDFile":"","LogConfig":{"Type":"json-file","Config":{},"NetworkMode":"default","PortBindings":{},"RestartPolicy":{"Name":"no","MaximumRetryCount":0},"VolumeDriver":"","VolumesFrom":null,"CapAdd":null,"CapDrop":null,"Dns":[],"DnsOptions":[]},"ExtraHosts":null,"GroupAdd":null,"IpcMode":"","Links":null,"OomScoreAdj":0,"PidMode":"","Privileged":false,"PublishAllPorts":false[0,0],"Isolation":"","CpuShares":0,"CgroupParent":"","BlkioWeight":0,"BlkioWeightDevice":null,"BlkioDeviceReadBps":null,"BlkioDeviceWriteBps":null,"Memory":0,"MemoryReservation":0,"MemorySwap":0,"MemorySwappiness":-1,"OomKillDisable":false,"PidsLimit":0,"Name":"aufs","Data":null,"Mounts":[],"Config":{"Hostname":"bdda6b06929f","Domainname":"","User":"","AttachStdin":false,"AttachStdout":true,"AttachStderr":true,"Tty":false,"OpenStdin":false,"StdinOnce":false,"Image":"hello-world","Volumes":null,"WorkingDir":"","Entrypoint":null,"OnBuild":null,"Labels":{},"StopSignal":"SIGTERM"},"NetworkSettings":{"Bridge":"","SandBoxID":"cb252dd027930120e20c119f299c9ac15ca97d42948bf40a20704841d1f3479","HairpinMode":false,"LinkLocalIPv6Address":"","LinkLocalIPv6PrefixLen":0,"PortMappings":null,"NetworkID":"b3fc35a430d3ec6d66d704b75d1efd3bf6cf27266ad7f36c9ee295f6914a1b8","IPAMConfig":null,"Links":null,"Aliases":null,"NetworkID":"b3fc35a430d3ec6d66d704b75d1efd3bf6cf27266ad7f36c9ee295f6914a1b8"}}
```


Figure 28: A sample request made to the /containers/{id}/json endpoint directly by using Postman REST client

The screenshot shows a Postman REST client interface. At the top, the method is GET and the URL is https://api.dockerhost.org:9000/containers/bdda6b06929f/json. The 'Headers' tab is active, showing two headers: 'Authorization' with value 'Bearer 1 1b317867c675884b732ec8331defdb230' and 'Content-Type' with value 'application/json'. Below the headers, the 'Body' tab is active, showing a JSON response. The response is displayed in 'Pretty' format. The status is 200 OK and the time taken is 48 ms.

```
{
  "Id": "bdda6b06929f654cac66177d5e7cb257752c86007884e3327971491ff018101e",
  "Created": "2016-03-19T14:47:53.197806691Z",
  "Path": "/hello",
  "Args": [],
  "State": {
    "Status": "exited",
    "Running": false,
    "Paused": false,
    "Restarting": false,
    "OOMKilled": false,
    "Dead": false,
    "Pid": 0,
    "ExitCode": 0,
    "Error": "",
    "StartedAt": "2017-02-04T08:12:37.056378582Z",
    "FinishedAt": "2017-02-04T08:12:37.08918755Z"
  },
  "Image": "sha256:690ed74de00f99a7d00a98a5ad855ac4febd66412be132438f9b8dbd300a937d",
  "ResolvConfPath": "/var/lib/json.log",
  "Name": "/waliboy",
  "RestartCount": 0,
  "Driver": "aufs",
  "MountLabel": "",
  "ProcessLabel": "",
  "AppArmorProfile": "",
  "ExecIDs": null,
  "HostConfig": {
    "Binds": null,
    "ContainerIDFile": "",
    "LogConfig": {
      "Type": "json-file",
      "Config": {}
    },
    "NetworkMode": "default",
    "PortBindings": {},
    "RestartPolicy": {
      "Name": "no",
      "MaximumRetryCount": 0
    },
    "VolumeDriver": "",
    "VolumesFrom": null,
    "CapAdd": null,
    "CapDrop": null,
    "Dns": [],
    "DnsOptions": [],
    "DnsSearch": [],
    "ExtraHosts": null,
    "GroupAdd": null,
    "IpcMode": "",
    "Links": null,
    "OomScoreAdj": 0,
    "PidMode": "",
    "Privileged": false,
    "PublishAllPorts": false,
    "Readon": [0, 0],
    "Isolation": "",
    "CpuShares": 0,
    "CgroupParent": "",
    "BlkioWeight": 0,
    "BlkioWeightDevice": null,
    "BlkioDeviceReadBps": null,
    "BlkioDeviceWriteBps": null,
    "KernelMemory": 0,
    "Memory": 0,
    "MemoryReservation": 0,
    "MemorySwap": 0,
    "MemorySwappiness": -1,
    "OomKillDisable": false,
    "PidsLimit": 0,
    "Ulimits": [
      {
        "Name": "aufs",
        "Data": null
      }
    ],
    "Mounts": [],
    "Config": {
      "Hostname": "bdda6b06929f",
      "Domainname": "",
      "User": "",
      "AttachStdin": false,
      "AttachStdout": true,
      "AttachStderr": true,
      "Tty": false,
      "OpenStdin": false
    },
    "Image": "hello-world",
    "Volumes": null,
    "WorkingDir": "",
    "Entrypoint": null,
    "OnBuild": null,
    "Labels": {}
  },
  "StopSignal": "SIGTERM",
  "NetworkSettings": {
    "Bridge": "",
    "SandboxID": "eac37751d7246285e377d976a4c7f388bacde42b2ac98774ac81b86f00d8f14",
    "HairpinMode": false,
    "LinkLocalIPv6Address": "bridge",
    "IPAMConfig": null,
    "Links": null,
    "Aliases": null,
    "NetworkID": "afa6b8f071b57a8612efcccd5eda902a301a0b8db57e567e5c0e8275dd642222",
    "Endpoint":
  }
}
```

Figure 29: A request made to /containers/{id}/json endpoint via proxy server by using Postman REST client

POST containers/create

The successful http POST request to this endpoint will create a new Docker container from the given Docker image. The id of the Docker image which is needed to create container should be passed in the body as a JSON object. The [Figure 30](#) mentioned below is relevant to a response which receives for a successful request made to the Docker Remote API when it is open to the outside without authentication. The [Figure 31](#) mentioned below is relevant to a response which receives for successful request made to the Docker Remote API via Proxy server after validating access token, which is embedded in the Authorization header with the authentication server. The general response time when Docker Remote API is opened to outside without authentication was fluctuated from 180 milliseconds to 200 milliseconds while the general response time when Docker Remote API only accessible via proxy server with authentication was fluctuated from 200 milliseconds to 220 milliseconds. In both scenarios the structure of the response body remain exactly the same.

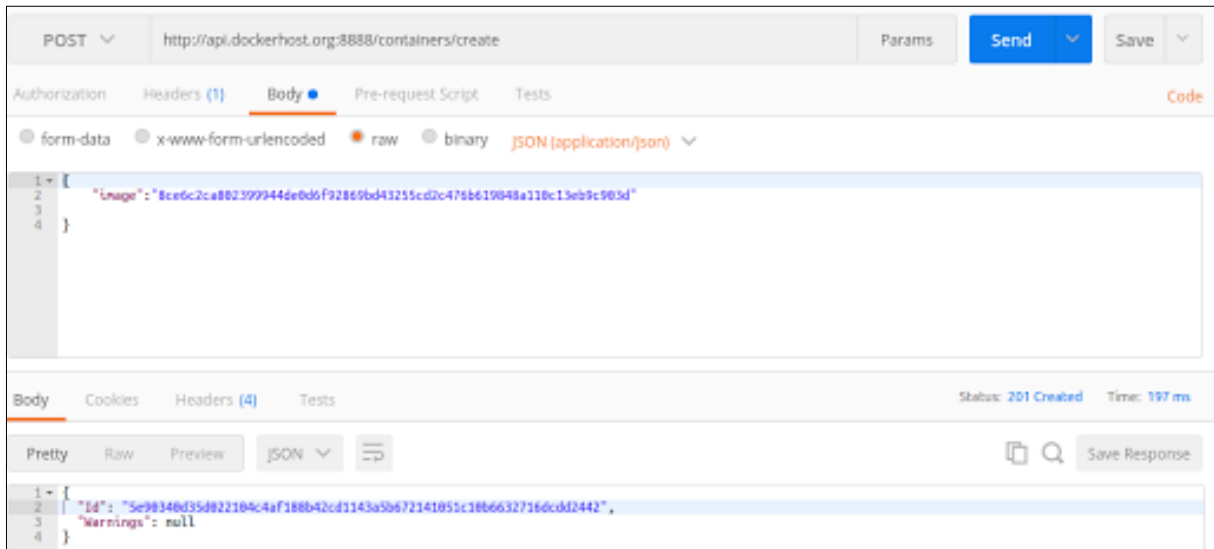


Figure 30: A request made to the /containers/create endpoint directly by using Postman REST client

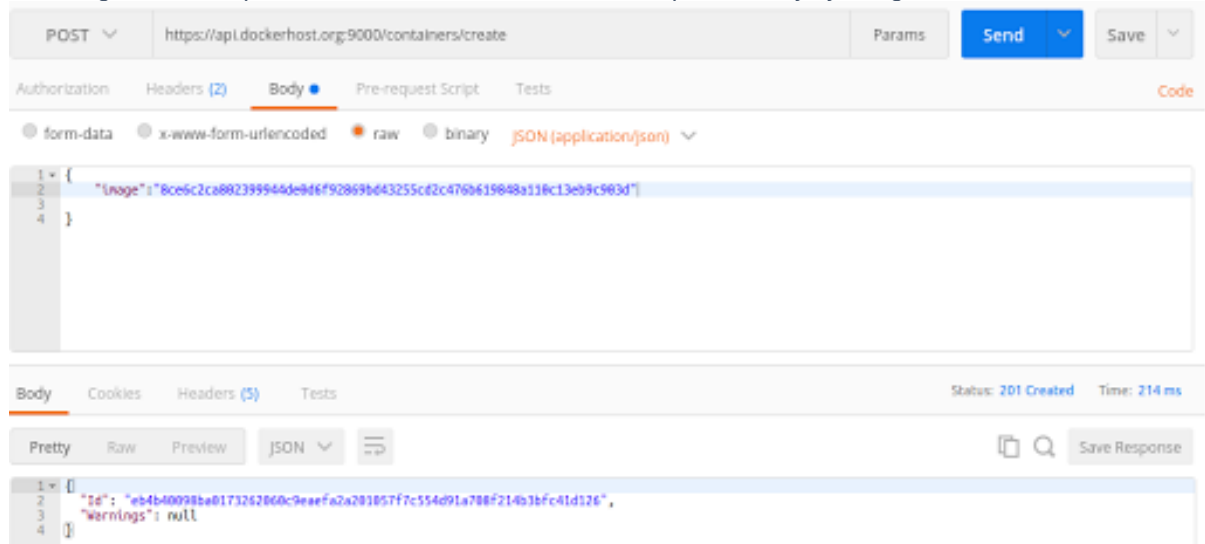


Figure 31: A request made to /containers/create endpoint via proxy server by using Postman REST client

POST /containers/{id}/start

The successful http POST request to this endpoint will start Docker container which belongs to given container id. The id of the Docker container which needs to be started should be included in the request URL. The [Figure 32](#) mentioned below is relevant to a response which receives for a successful request made to the Docker Remote API when it is open to the outside without authentication. The [Figure 33](#) mentioned below is relevant to a response which receives for successful request made to the Docker Remote API via Proxy server after validating access token, which is embedded in the Authorization header with the authentication server. The general response time when Docker Remote API is opened to outside without authentication was fluctuated from 400 milliseconds to 500 milliseconds while the general response time when Docker Remote API only accessible via proxy server with authentication was fluctuated from 500 milliseconds to 600 milliseconds. In both scenarios the structure of the response body remain exactly the same.

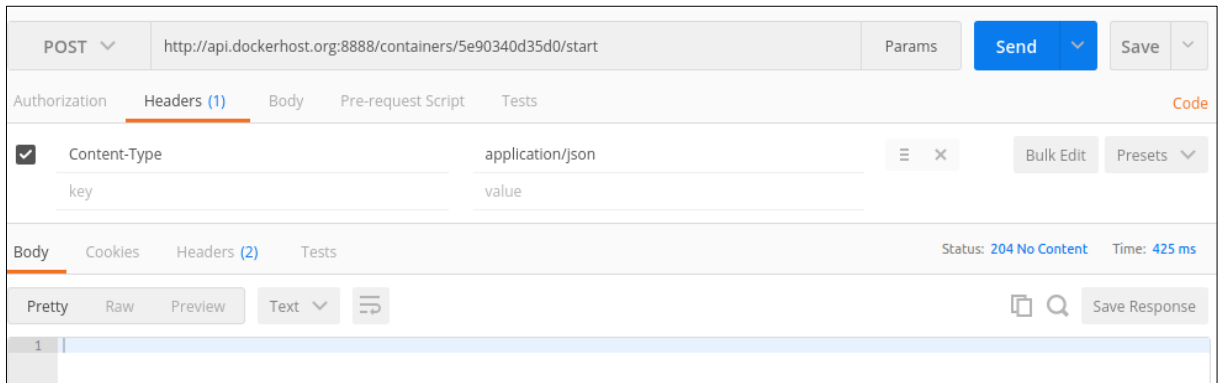


Figure 32: A request made to the /containers/{id}/start endpoint directly by using Postman REST client

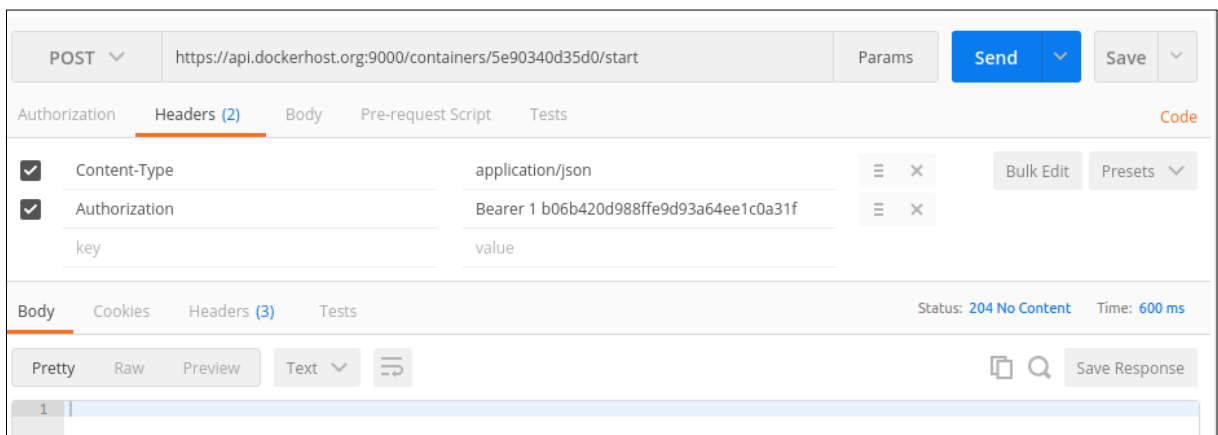


Figure 33: A request to /containers/{id}/stop endpoint via proxy server by using Postman REST client

POST containers/{id}/stop

The successful http POST request to this endpoint will stop running Docker container which belongs to given container id. The id of the Docker container which needs to be stopped should be included in the request URL. The [Figure 34](#) mentioned below is relevant to a response which receives for a successful request made to the Docker Remote API when it is open to the outside without authentication. The [Figure 35](#) mentioned below is relevant to a response which receives for successful request made to the Docker Remote API via Proxy server after validating access token, which is embedded in the Authorization header with the authentication server. The general response time when Docker Remote API is opened to outside without authentication was fluctuated from 500 milliseconds to 600 milliseconds while the general response time when Docker Remote API only accessible via proxy server with authentication was fluctuated from 600 milliseconds to 700 milliseconds. In both scenarios the structure of the response body remain exactly the same.

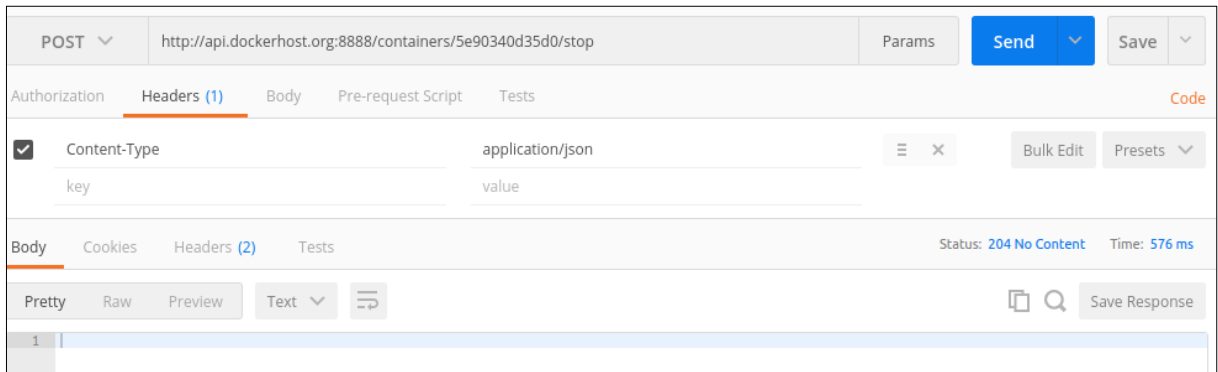


Figure 34: A sample request made to the `/containers/{id}/stop` endpoint directly by using Postman REST client

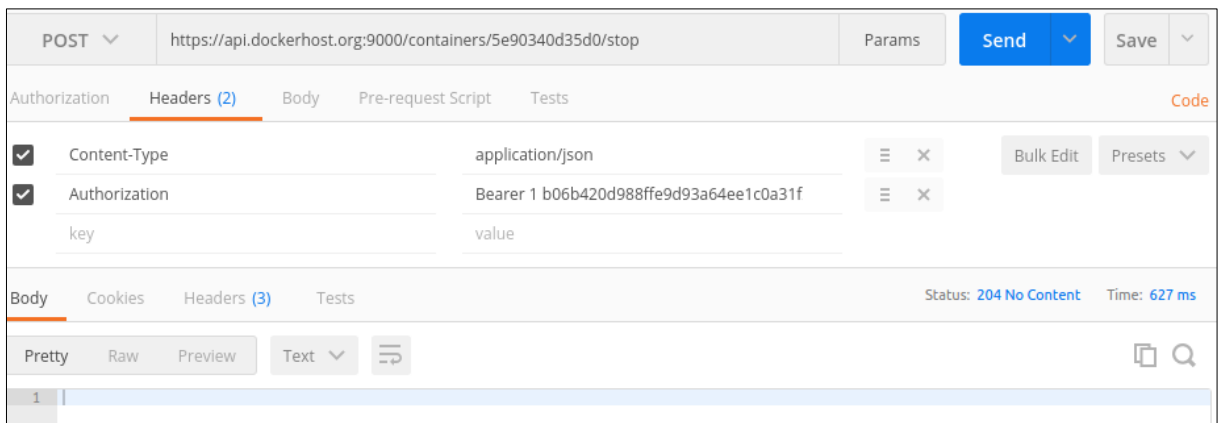


Figure 35: A sample request to `/containers/{id}/stop` endpoint via proxy server by using Postman REST client

DELETE `containers/{id}`

The successful http DELETE request to this endpoint will delete Docker container which belongs to given container id. The id of the Docker container which is needed to be deleted should be included in the request URL. The [Figure 36](#) mentioned below is relevant to a response which receives for a successful request made to the Docker Remote API when it is open to the outside without authentication. The [Figure 37](#) mentioned below is relevant to a response which receives for successful request made to the Docker Remote API via Proxy server after validating access token, which is embedded in the Authorization header with the authentication server. The general response time when Docker Remote API is opened to outside without authentication was fluctuated from 150 milliseconds to 160 milliseconds while the general response time when Docker Remote API only accessible via proxy server with authentication was fluctuated from 180 milliseconds to 200 milliseconds. In both scenarios the structure of the response body remain exactly the same.

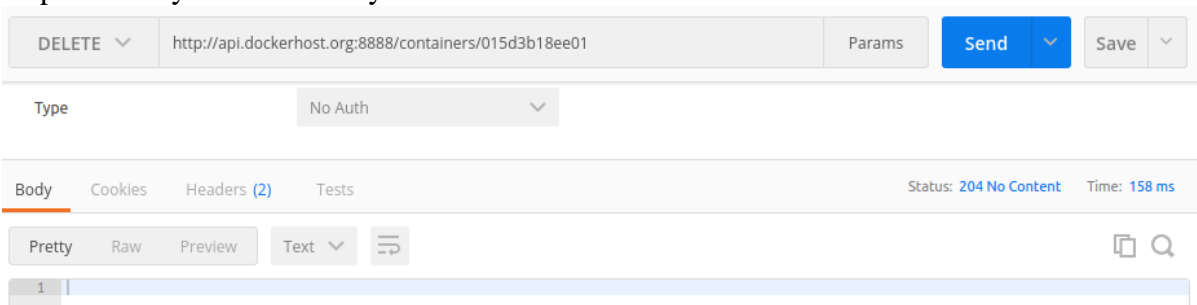


Figure 36: A request made to the `/containers/{id}` endpoint directly by using Postman REST client

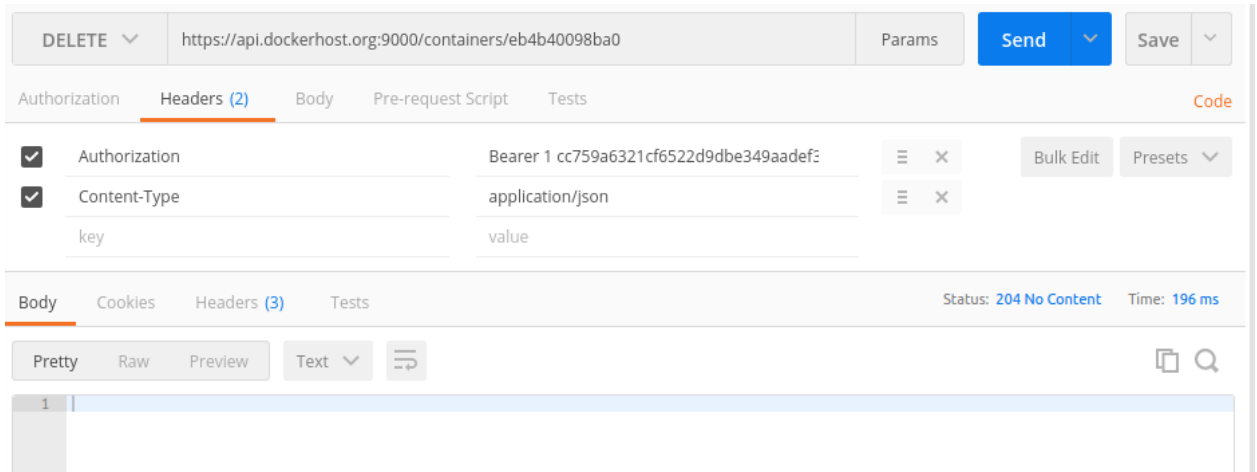


Figure 37: A sample to /containers/{id} endpoint via proxy server by using Postman REST client

Security Evaluation

The implemented system is a token based authentication mechanism for Docker Remote API when Docker Daemon listening on a TCP port. The authentication server which is the main component of the implemented system, responsible for the access token generation, issuing and maintenance. The authentication server implemented with focusing on OAuth2's **Resource Owner Password Credentials** Grant mode. The Table 2 illustrates to what extent the authentication server compliant to the OAuth2's Owner Password Credentials Grant mode [11].

Property	OAuth2 RFC	Implemented Authentication Server
Authentication Flow	<p>The resource owner provides the client with its username and password.</p> <p>The client requests an access token from the authorization server's token endpoint by including the credentials received from the resource owner. When making the request, the client authenticates with the authorization server.</p> <p>The authorization server authenticates the client and validates the resource owner credentials, and if valid, issues an access token.</p>	<p>Every user client who authorized to access Docker Remote API have an id and a password.</p> <p>The client requests an access token from the authorization server's token /getAccessToken endpoint by including the credentials received from the Docker Administrator.</p> <p>The authentication server authenticates the client and validates the user credentials, and if valid, issues an access token.</p>

<p>Authentication Server Functions</p>	<p>Require client authentication for confidential clients or for any client that was issued client credentials.</p> <p>Authenticate the client if client authentication is included.</p> <p>Validate the resource owner password credentials using its existing password validation algorithm.</p>	<p>Issues access tokens for the legitimate requests.</p> <p>Access the validity of issued access tokens.</p> <p>Maintain access tokens until the expiration.</p> <p>Validate the user credentials against the database.</p>
<p>Access tokens</p>	<p>Access tokens must be kept confidential in transit and storage, and only shared among the authorization server, the resource servers the access token is valid for, and the client to whom the access token is issued.</p> <p>The authorization server must ensure that access tokens cannot be generated, modified, or guessed to produce valid access tokens by unauthorized parties</p>	<p>All the access tokens are stored in authentications server using Redis after encrypting the tokens.</p> <p>Different encryption keys has been used for different users.</p> <p>NodeJs random number generation function is used to generate access tokens.</p> <p>All the connections are secured with SSL.</p>
<p>Refresh Tokens</p>	<p>Authorization servers may issue refresh tokens to web application clients and native application clients.</p> <p>The authorization server must maintain the binding between a refresh token and the client to whom it was issued.</p>	<p>Not implemented.</p>

Authentication Request Parameters	grant_type - Required. Must be set to “password” username - Required. password - Required. scope - Optional.	id - Required. password - Required.
Authentication Example Request	POST /token HTTP/1.1 Host: server.example.com Authorization:Basic czZCaGRSa3F0MzpnWDFm QmF0M2JW Content-Type:application/x- www-form-urlencoded grant_type=password&userna me=johndoe&password=A3d dj3w	POST /token HTTP/1.1 Host: auth.testcom.org Content- Type:application/json { “Id” : 111, “password” : “ys2b7” }
Access token example response	HTTP/1.1 200 OK Content- Type:application/json;charset =UTF-8 Cache-Control: no-store Pragma: no-cache { “access_token”：“2YotnFZFEj r1zCsicMWpAA”, “token_type”：“example”, “expires_in”：3600, “refresh_token”：“tGzv3JOkF 0XG5Qx2TIKWIA”, “example_parameter”：“examp le_value” }	HTTP/1.1 200 OK Content-Type: application/json Date: Mon, 27 Mar 2017 07:44:19 GMT Connection: keep-alive Transfer-Encoding: chunked { “accessToken”：“eab922015aa a63fd6a170d1761ab3b28db5 75929”, “expiresIn”：3600 }

Table 2: Analysis of the authentication server's compliance with OAuth2

Evaluation Conclusions

The intention of making Docker Remote API only accessible via a proxy server, is to ensure that it can be only accessible for parties which are authorized. Based on the evaluation results, the implementation of the proxy server and the authentication server has not introduced deviations to the responses which are originated from the Docker Remote API. Furthermore after the comparison of the response bodies between two occasions it can be clearly seen they are identical in structure. It is obvious that the operation of the proxy server and the authentication consume some computational resources. Hence the general response time for a request which passed via the proxy server to the Docker Remote API is greater than the general response time for a request which comes directly to the Docker Remote API. Based on the evaluation results, it can be mentioned that the implementation of proxy server and the authentication server haven't caused to increase the response in large scale.

The proxy server and the authentication server were able to work in different operational areas in Docker with different type of requests. The [Figure 44](#) and the [Figure 45](#) in Appendix 1 belong to creating a Docker image my pulling it from a registry. This request involves in streaming of data packets. There are several other results relevant various Docker operations are included in the Appendix 1. This indicates that new implementation works well with different type of requests.

CHAPTER 06 – Conclusions

CHAPTER 06 – Conclusions

Challenges Faced

The main challenge that was faced in this project was lack of previous researches that had been carried out regarding to the Docker. Because of that reason, there were very little amount of research papers and other research materials found regarding to the project. In the early stages of the project it was planned to enable the authentication by editing the source code. Therefore it had to be followed Docker Advanced Contribution workflow which is more complex and time consuming. Hence it was needed focused on other alternative ways of implementing authentication of user requests. The next challenge was setting up a proper test environment. The current results were obtained by simulating the system on Linux virtual machine based environment as mentioned in the previous sections. The results which were obtained from that was satisfactory. But it would be better if the test environment similar to the practical situation.

Future Works

There are several future researches would be carried out based on the accessing remotely Docker Remote API when it's listening on TCP port. The first one is changing Docker source code support the token based request authentication. This research will require more expertise, especially in 'Go' programming language and will consume little bit of time. But it will be able to provide the request authentication in more integrated manner. One of other researches is replacing the authentication server with other kind of well-established authentication mechanism such as Kerberos. Furthermore authorization server can be streamlined to make it more aligned with an authentication protocol like OAuth2.

Summary

The project is focused on developing and demonstrating a token based authentication mechanism for the user requests receives to the Docker Remote API when the Docker Daemon listening on a TCP port. In this scenario there is a listening IP address and a listening TCP port. The listening IP address cannot be any and has to be something that cannot be accessed outside. The expected authentication has been achieved in this project by introducing an authentication server and a proxy server. Since the requests cannot be made to the Docker Remote API directly, the request has to go through the proxy server. The proxy server validates the access token which should be included in the Authorization header, issued by the authentication server. The request will or will not be passed to the Docker Remote API based on the response from the proxy server.

Evaluation results indicates that introduction of the proxy server has not introduce deviations to the expected results. But it has introduced some latency to the overall request response routing. This implementation will be very much helpful when the other applications need to perform operations with Docker. The users who are authorized to make requests and the user applications can be made requests to the Docker Remote API via proxy server as REST API calls and all the required parameters can be stored with the application.

Appendixes

Appendix 1 – Other test results regards to the evaluation

This section includes some of the images regarding to the results obtained during the evaluation phase of the project by using cURL command from the Linux terminal. All the request send via the proxy server to Docker Remote API.

```
asitha@ubuntu:~$ curl -X POST -H "Content-Type: application/json" -d '{"id":1, "password":"abc123"}' "https://auth.testcom.org:9999/getAccessToken" --cacert /home/asitha/Documents/certs/authserver.crt -i
HTTP/1.1 200 OK
Content-Type: application/json
Date: Sat, 04 Mar 2017 15:05:54 GMT
Connection: keep-alive
Transfer-Encoding: chunked

{"accessToken":"2a8141f600a8ff9a6794a17bf557e910c2afc88c","expiresIn":3591}asitha@ubuntu:~$ █
```

Figure 38: Obtaining of access token by submitting valid credentials

```
asitha@ubuntu:~$ curl -X POST -H "Content-Type: application/json" -d '{"id":1, "accessToken":"2a8141f600a8ff9a6794a17bf557e910c2afc88c"}' "https://auth.testcom.org:9999/checkValidity" --cacert /home/asitha/Documents/certs/authserver.crt -i
HTTP/1.1 204 No Content
Content-Type: application/json
Date: Sat, 04 Mar 2017 15:08:47 GMT
Connection: keep-alive

asitha@ubuntu:~$ █
```

Figure 39: Checking access token validity with a valid access token

```
asitha@ubuntu:~$ curl -X POST -H "Content-Type: application/json" -d '{"id":1, "accessToken":"2a8141f600a8ff9a6794a17bf557e910c2afc88d"}' "https://auth.testcom.org:9999/checkValidity" --cacert /home/asitha/Documents/certs/authserver.crt -i
HTTP/1.1 404 Not Found
Content-Type: application/json
Date: Sat, 04 Mar 2017 15:10:03 GMT
Connection: keep-alive
Transfer-Encoding: chunked

{"status":false,"msg":"invalid_access_token","data":{"errorCode":"0008","errorMessage":"Invalid or expired acces token"}}asitha@ubuntu:~$ █
```

Figure 40:: Checking an access token's validity with an invalid access token

```
asitha@ubuntu:~$ curl -X POST -H "Content-Type: application/json" -H "Authorization: Bearer 1 892606b43098976355119e6fe4ce17093246fae6" "https://api.dockerhost.org:9000/containers/ee" --cacert /home/asitha/Documents/certs/proxyserver.crt -i
HTTP/1.1 404 Not Found
Content-Type: application/json
Date: Sat, 04 Mar 2017 14:58:27 GMT
Connection: keep-alive
Transfer-Encoding: chunked

{"status":false,"msg":"invalid_access_token","data":{"errorCode":"0008","errorMessage":"Invalid or expired acces token"}}asitha@ubuntu:~$ █
```

Figure 41: Request sent to the proxy server with an invalid access token

```
asitha@ubuntu:~$ curl -X POST -H "Content-Type: application/json" -H "Authorization: Bearer 1 2a8141f600a8ff9a6794a17bf557e910c2afc88c" "https://api.dockerhost.org:9000/containers/ee" --cacert /home/asitha/Documents/certs/proxyserver.crt -i
HTTP/1.1 404 Not Found
Content-Type: application/json
Date: Sat, 04 Mar 2017 15:22:55 GMT
Connection: keep-alive
Transfer-Encoding: chunked

404 page not found
asitha@ubuntu:~$ █
```

Figure 42: Sending request to an invalid endpoint of Docker Remote API

```
asitha@ubuntu:~$ curl -X POST -H "Content-Type: application/json" -H "Authorization: Bearer 1 8a6d428d1eed00c562723f8a51604a441c96dec" "https://api.dockerhost.org:9000/images/74b44a255c0d/tag?repo=myrepo&force=0&tag=v42" --cacert /home/asitha/Documents/certs/proxyserver.crt -i
HTTP/1.1 201 Created
server: Docker/1.10.3 (linux)
Date: Sat, 04 Mar 2017 17:13:01 GMT
content-type: text/plain; charset=utf-8
connection: close
Transfer-Encoding: chunked
```

Figure 43: Tagging an image

```

asitha@ubuntu:~$ curl -X POST -H "Content-Type: application/json" -H "Authorization: Bearer 1 2a8141f600a8ff9a6794a17bf557e910c2afc88c"
https://api.dockerhost.org:9000/v1.22/images/create?fromImage=busybox&tag=latest" --cacert /home/asitha/Documents/certs/proxyserver.crt
HTTP/1.1 200 OK
content-type: application/json
server: Docker/1.10.3 (linux)
date: Sat, 04 Mar 2017 15:37:25 GMT
connection: close
transfer-encoding: chunked

{"status":"Pulling from library/busybox","id":"latest"}
{"status":"Pulling fs layer","progressDetail":{"current":0,"total":0,"id":"4b0bc1c4050b"}
{"status":"Downloading","progressDetail":{"current":16384,"total":677628,"progress":"[\u003e
] 16.38 kB/677.6 kB","id":"4b0bc1c4050b"}
{"status":"Downloading","progressDetail":{"current":30784,"total":677628,"progress":"[=\u003e
] 30.78 kB/677.6 kB","id":"4b0bc1c4050b"}
{"status":"Downloading","progressDetail":{"current":47168,"total":677628,"progress":"[==\u003e
] 47.17 kB/677.6 kB","id":"4b0bc1c4050b"}
{"status":"Downloading","progressDetail":{"current":63552,"total":677628,"progress":"[===\u003e
] 63.55 kB/677.6 kB","id":"4b0bc1c4050b"}
{"status":"Downloading","progressDetail":{"current":79936,"total":677628,"progress":"[====\u003e
] 79.94 kB/677.6 kB","id":"4b0bc1c4050b"}
{"status":"Downloading","progressDetail":{"current":96320,"total":677628,"progress":"[=====\u003e
] 96.32 kB/677.6 kB","id":"4b0bc1c4050b"}
{"status":"Downloading","progressDetail":{"current":112704,"total":677628,"progress":"[=====\u003e
] 112.7 kB/677.6 kB","id":"4b0bc1c4050b"}
{"status":"Downloading","progressDetail":{"current":129088,"total":677628,"progress":"[=====\u003e
] 129.1 kB/677.6 kB","id":"4b0bc1c4050b"}
{"status":"Downloading","progressDetail":{"current":145472,"total":677628,"progress":"[=====\u003e
] 145.5 kB/677.6 kB","id":"4b0bc1c4050b"}
{"status":"Downloading","progressDetail":{"current":161856,"total":677628,"progress":"[=====\u003e
] 161.9 kB/677.6 kB","id":"4b0bc1c4050b"}
{"status":"Downloading","progressDetail":{"current":178240,"total":677628,"progress":"[=====\u003e
] 178.2 kB/677.6 kB","id":"4b0bc1c4050b"}
{"status":"Downloading","progressDetail":{"current":194624,"total":677628,"progress":"[=====\u003e

```

Figure 44: Creating an Image by pulling

```

] 491.5 kB/677.6 kB","id":"4b0bc1c4050b"}
{"status":"Extracting","progressDetail":{"current":491520,"total":677628,"progress":"[=====\u003e
] 491.5 kB/677.6 kB","id":"4b0bc1c4050b"}
{"status":"Extracting","progressDetail":{"current":524288,"total":677628,"progress":"[=====\u003e
] 524.3 kB/677.6 kB","id":"4b0bc1c4050b"}
{"status":"Extracting","progressDetail":{"current":557056,"total":677628,"progress":"[=====\u003e
] 557.1 kB/677.6 kB","id":"4b0bc1c4050b"}
{"status":"Extracting","progressDetail":{"current":589824,"total":677628,"progress":"[=====\u003e
] 589.8 kB/677.6 kB","id":"4b0bc1c4050b"}
{"status":"Extracting","progressDetail":{"current":622592,"total":677628,"progress":"[=====\u003e
] 622.6 kB/677.6 kB","id":"4b0bc1c4050b"}
{"status":"Extracting","progressDetail":{"current":655360,"total":677628,"progress":"[=====\u003e
] 655.4 kB/677.6 kB","id":"4b0bc1c4050b"}
{"status":"Extracting","progressDetail":{"current":677628,"total":677628,"progress":"[=====\u003e
] 677.6 kB/677.6 kB","id":"4b0bc1c4050b"}
{"status":"Extracting","progressDetail":{"current":677628,"total":677628,"progress":"[=====\u003e
] 677.6 kB/677.6 kB","id":"4b0bc1c4050b"}
{"status":"Pull complete","progressDetail":{"current":0,"total":0,"id":"4b0bc1c4050b"}
{"status":"Pull complete","progressDetail":{"current":0,"total":0,"id":"4b0bc1c4050b"}
{"status":"Digest: sha256:817a12c32a39bbe394944ba49de563e085f1d3c5266eb8e9723256bc4448680e"}
{"status":"Status: Downloaded newer image for busybox:latest"}

```

Figure 45: Creating an image by pulling

```

asitha@ubuntu:~$ curl -X POST -H "Content-Type: application/json" -H "Authorization: Bearer 1 8a6d428d1eed00c562723f8a51604a441c96dec"
https://api.dockerhost.org:9000/containers/c3f94cc08acc3/attach?stream=true" --cacert /home/asitha/Documents/certs/proxyserver.crt -i
HTTP/1.1 200 OK
content-type: application/vnd.docker.raw-stream
Date: Sat, 04 Mar 2017 16:41:24 GMT
Connection: keep-alive
Transfer-Encoding: chunked

```

Figure 46: Attach an image

References

- [1]"Understand the architecture", Docs.docker.com, 2016. [Online]. Available: <https://docs.docker.com/engine/understanding-docker/>. [Accessed: 02- Apr- 2016].
- [2]"daemon", Docs.docker.com, 2016. [Online]. Available: <https://docs.docker.com/engine/reference/commandline/daemon/>. [Accessed: 02- Apr- 2016].
- [3]"Remote API", Docs.docker.com, 2016. [Online]. Available: https://docs.docker.com/engine/reference/api/docker_remote_api/. [Accessed: 04- Apr- 2016].
- [4]"Remote API v1.18", Docs.docker.com, 2016. [Online]. Available: https://docs.docker.com/engine/reference/api/docker_remote_api_v1.18/. [Accessed: 04- Apr- 2016].
- [5]"daemon", Docs.docker.com, 2016. [Online]. Available: <https://docs.docker.com/v1.10/engine/reference/commandline/daemon/>. [Accessed: 10- May- 2016].
- [6]"Protect the Docker daemon socket", Docs.docker.com, 2016. [Online]. Available: <https://docs.docker.com/v1.10/engine/security/https/>. [Accessed: 10- May- 2016].
- [7]"Remote API v1.22", Docs.docker.com, 2016. [Online]. Available: https://docs.docker.com/v1.10/engine/reference/api/docker_remote_api_v1.22/. [Accessed: 10- May- 2016].
- [8]J. Steiner, C. Neuman and J. Schiller, Kerberos: An Authentication Service for Open Network Systems, 1st ed. 2011, pp. 5-9.
- [9]B. Subedi, "Kerberos Authentication Protocol", Slideshare.net, 2013. [Online]. Available: <http://www.slideshare.net/BibekNam/kerberos-authentication-protocol>. [Accessed: 14- May- 2016].
- [10]"Implementing Message Layer Security with Kerberos in WSE 3.0", Msdn.microsoft.com, 2016. [Online]. Available: <https://msdn.microsoft.com/en-us/library/ff650265.aspx>. [Accessed: 14- Jun- 2016].
- [11]"RFC 6749 - The OAuth 2.0 Authorization Framework", Tools.ietf.org, 2016. [Online]. Available: <https://tools.ietf.org/html/rfc6749.html>. [Accessed: 14- May- 2016].
- [12]"An Introduction to OAuth2", Slideshare.net, 2016. [Online]. Available: <http://www.slideshare.net/aaronpk/an-introduction-to-oauth-2/>. [Accessed: 20- May- 2016].
- [13]"docker/distribution", GitHub, 2016. [Online]. Available: <https://github.com/docker/distribution/blob/master/docs/spec/auth/token.md>. [Accessed: 20- May- 2016].
- [14]"Token Authentication Specification", Docs.docker.com, 2016. [Online]. Available: <https://docs.docker.com/registry/spec/auth/token/>. [Accessed: 20- May- 2016].
- [15]"Quickstart contribution", Docker, 2016. [Online]. Available: <https://docs.docker.com/opensource/code/>. [Accessed: 04- Jun- 2016].
- [16]"Contribute", Docker, 2015. [Online]. Available: <https://www.docker.com/contribute>. [Accessed: 04- Jun- 2016].

[17]"Understand how to contribute", Docker, 2016. [Online]. Available: <https://docs.docker.com/opensource/workflow/make-a-contribution/>. [Accessed: 04- Jun- 2016].

[18]"Find and claim an issue", Docker, 2016. [Online]. Available: <https://docs.docker.com/opensource/workflow/find-an-issue/>. [Accessed: 04- Jun- 2016].

[19]"Advanced contributing", Docker, 2016. [Online]. Available: <https://docs.docker.com/opensource/workflow/advanced-contributing/>. [Accessed: 04- Jun- 2016].

[20] P. Teixeira, Professional Node.JS, 1st ed. Google Inc, 2012, pp. 15-17.

[21]"Introduction to Redis – Redis", Redis.io, 2014. [Online]. Available: <https://redis.io/topics/introduction>. [Accessed: 11- Jun- 2016].