Component: Platform Controller

Table of Contents

Overview

This page provides a detailed design for the Platform Controller Component of the COPS Platform. It is based on Zuul 2 (developed by Netflix). See Decision Log here: Which API Gateway should COPS use?

Responsibilities

- Authenticate users using KeyCloak
- Provide a single point of access to multiple RESTful services
- Provide a single point of access to Secure Data Storage
- Allow access to each service only to authorized users
- Secure (encrypt) data in-transit between consumers and producers of services
- Request containers for specific services for specific users

Overall Strategy

There are two parts to the Platform Controller. The first handles REST requests for services, and the second handles interactions with the Data API. Both parts share authentication/authorization logic and other security configuration (which is why they are not two completely independent applications).

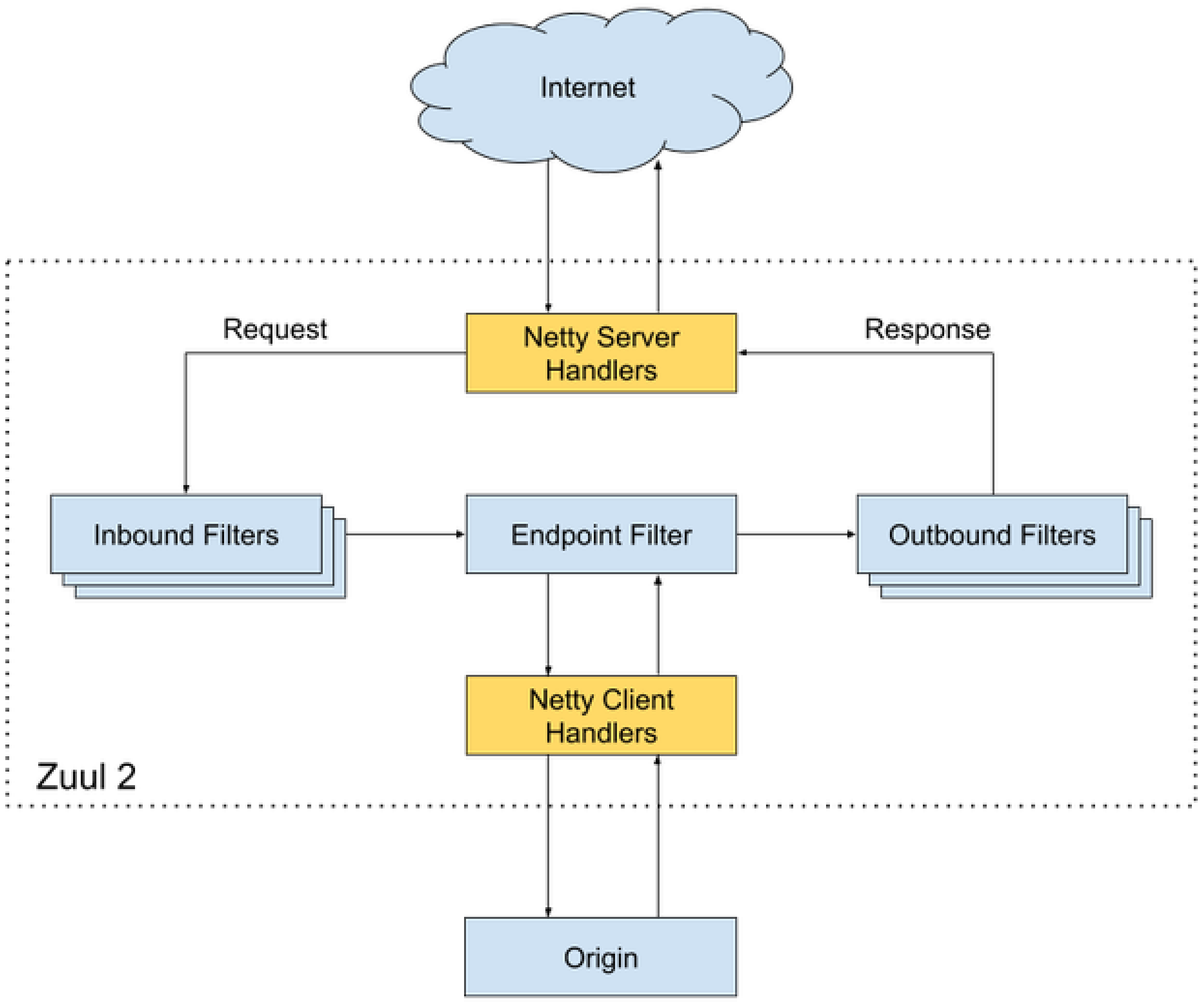
RESTful API Gateway

Figure 1 shows how Zuul 2 applications work. Netty handles the actual connections (configured via Java or Groovy) and one or more filters handle requests. In most of the documentation, the “Origin” refers to Backend services that provide responses to requests that arrive from the Internet.

For the purposes of the COPS Platform:

- Inbound filters will:
    - Validate user authentication/authorization via Keycloak
    - Start/Stop services (through the Container Runtime) based on web requests
- Endpoint filters will likely not be implemented, the default ProxyEndpoint filter should suffice.
- Outbound filters will:
    - Log info about which data was sent where (for auditing purposes)

Figure 1: Architectural Overview of Zuul 2 (diagram from Zuul 2 documentation)



Source: https://github.com/Netflix/zuul/wiki/How-It-Works-2.0

Data API

ToDo as part of **COPS-412** - Getting issue details... STATUS

Component Interfaces

Incoming connections on TCP Port 443 (REST Requests from External Service Consumers)

Outgoing connections to TCP Port 443 (to reach IdAM)

Outgoing connection to TCP Port 2376 (to reach the Container Runtime)

Outgoing connections to TCP Port 514 (Logs sent to rsyslog)

Outgoing connections to TPC Port 443 (and others) (to reach services inside the Container Runtime)

Software configuration

Zuul 2 is a library, not a standalone application. Configuration of the server and the filters will require that we write a Java application.

Netty Server

The Zuul 2 codebase includes an example application that configures Netty.

Configure for Mutual TLS: https://github.com/Netflix/zuul/wiki/Core-Features#mutual-tls

- As an implementation detail, we could choose to run this behind a Proxy that handles TLS so that Netty doesn't need to be aware of it.
- The traffic between the backend services (Origin) and the API Gateway must be encrypted with TLS/SSL too. If this is not possible, we must document clear explanations why.

References

- Server Configuration Documentation: https://github.com/Netflix/zuul/wiki/Server-Configuration
- Example Server Configuration: https://github.com/Netflix/zuul/blob/v2.1.6/zuul-sample/src/main/java/com/netflix/zuul/sample/SampleServerStartup.java

Filters

The Zuul 2 codebase includes an example application that configures at least one filter of each type.

Filters are implemented using Groovy (very similar to Java).

Create two inbound filters that:

- Validates user authentication/authorization via Keycloak
- Starts/Stops services (through the Container Runtime) based on the requests
    - Uses the "--security-opt label=…" option (or the equivalent in the docker-java-api) to set the security level and categories for the container, based on the user authorization.

Create an outbound filter that:

- Logs info about which data was sent where

References

- Filter Documentation: https://github.com/Netflix/zuul/wiki/Filters
- Example Filters: https://github.com/Netflix/zuul/tree/v2.1.6/zuul-sample/src/main/groovy/com/netflix/zuul/sample/filters
- Groovy comparison to Java: https://groovy-lang.org/differences.html
- Examples of other integrations with Keycloak are documented on the pages listed below. It is expected that the Java application will use a similar scheme, communicating with a Keycloak client using SAML or OpenID.
    - Configure GitLab to use KeyCloak authentication
    - Configure SonarQube to use Keycloak for authentication
    - Keycloak
- Java libraries for acting as an OpenID client:
    - https://openid.net/developers/certified/
- Docker provides a Java API for interacting with Docker Containers:
    - https://github.com/docker-java/docker-java
    - https://www.baeldung.com/docker-java-api
    - The standard port for secure communication with docker is TCP 2376: https://docs.docker.com/engine/reference/commandline/dockerd/
- Setting the SELinux label when starting a docker container: https://docs.docker.com/engine/reference/run/ (search "Security configuration")

Logging and Auditing

All logs must be sent to the external Log Aggregator (currently expected to be rsyslog).

Nonfunctional requirements for the software

- All build dependencies must be mirrored locally.
    - Filenames, hashes, and download locations must be documented.
- A full build must be possible in a disconnected/offline environment.
- Use an OpenJDK distribution (not the Oracle-provided JVM, since it is no longer free to use for commercial purposes)
    - On RHEL and CentOS, use the OpenJDK from the official yum repo.
- The project should be stored in the 'api-gateway' repo on Skyward's Gitlab instance, inside the 'cops' group.
    - https://gitlab.skyward.cloud/cops/api-gateway
- The final version must run on CentOS 7 (so be sure that all necessary libraries and path references work on CentOS 7).