API Platform - Training: Light4J Side Car Proxy 101

Introduction

Welcome to Light4J Side Car Proxy 101! In this session we will learn about the light-proxy and light4J.

First of all, we will take a quick look at the light framework, a framework for building modern web applications and the pattern it uses when building applications.

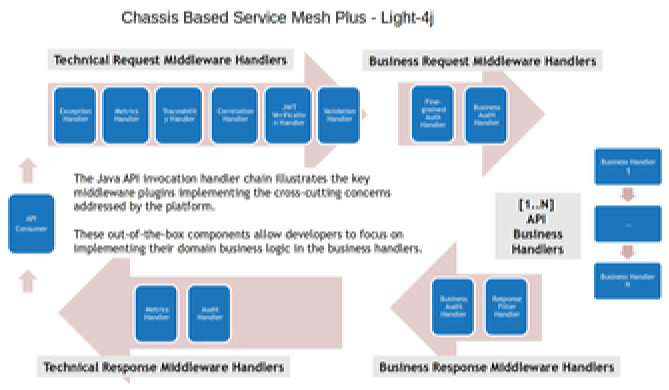
Then we will look at the side car light-proxy, an application built using light framework, which addresses technical cross cutting concerns so that service developers will only focus on the business logic without worry about security, auditing, logging, metrics, etc.

Lastly we will look at the different handlers enabled in the light-proxy and the features they provide.

What is light4J?

Light-4j is an opensource framework written in Java SE and designed to build cloud native Web/API with different options. It is fast, lightweight and cloud native microservices platform. The main design goal is highest throughput, lowest latency and smallest memory footprint.

Light-4j frameworks provide middleware handlers to address cross-cutting concerns for your application so that you only need to implement the final business handlers to process the request and return a response without thinking about exception handling, metrics, traceability, correlation, security, validation, etc. If you have the specification available, the light-codegen can scaffold the application for you to fill in the business logic into the generated handlers and write assertions in the generated tests for these handlers.



Chassis Based Service Mesh Plus - Light-4j
Technical Request Middleware Handlers
Business Request Middleware Handlers
The Java API invocation handler chain illustrates the key middleware plugins implementing the cross-cutting concerns addressed by the platform.
These out-of-the-box components allow developers to focus on implementing their domain business logic in the business handlers.
[1..N] API Business Handlers
Technical Response Middleware Handlers
Business Response Middleware Handlers

As you can see in the figure above, light framework employs Chain of Responsibility pattern. It places a particular functionality in one handler class and organizes multiple handlers in a chain before the control reaches the business handler. Some of the features/handlers you may want to apply on you request are auditing, security, metrics collection, etc.

Business handler is a class where you want to do a business transaction e.g. eTransfer, credit-debit, bill payment, etc, to name a few.

Once the business transaction is complete, you can apply the handlers on the response as well. This approach has some clear benefits, some of them are:

Benefits

1. Ability to add/remove features without impacting business logic.
2. Makes debugging and troubleshooting easy.
3. Lets you test the part of the application easily and in isolation.
4. Fails fast.

Cross Cutting Concerns (Handlers)

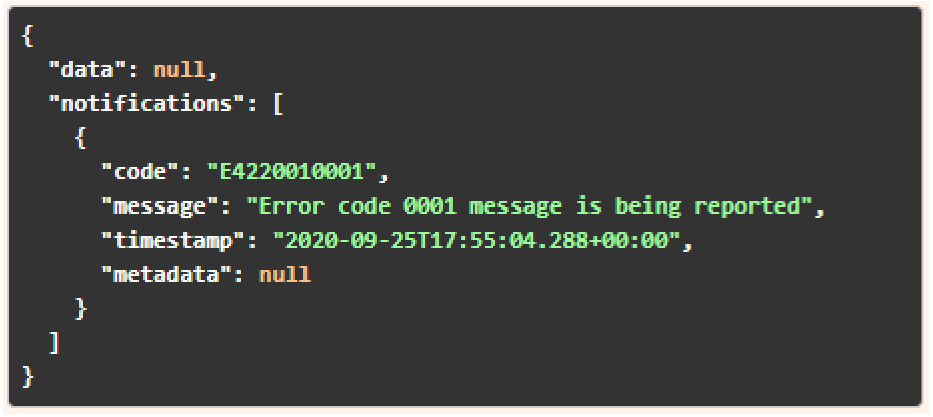
One of the design goals of the light-4j is to address all the technical cross-cutting concerns in the framework. The same pattern can be applied in the business context as well so that you can have several handlers to perform different tasks within the business context. For example, the fine-grained authorization can be done as a cross-cutting concern in the business context without mixing into the real business logic within the core request handler.

When building an application, there is a functional requirement, and normally there is always a non-functional requirement that is created to address all technical concerns. In most platforms, developers need to address these concerns in the same business handler at the same time. It makes the application very hard to maintain and to reason about as functional requirements, and business requirements are implemented at the same time in the same context. The code is blended and hard to separate, hence decrease the maintainability. The light platform divides these concerns into individual handlers so that multiple developers can work together. These handlers are relatively simple and easy to be shared between services, and we have provided dozens built-in handlers in the platform. Without non-functional concerns in the business logic, the main business handler will be much simpler to implement and reason about. In a long run, it is easy to maintain as any modification will be focused on the business logic only.

Following are the light4J handlers (Cross Cutting Concerns) enabled in the light-proxy:

ExceptionHandler

It wraps the entire chain so that any un-handled exceptions will finally reach here and to be handled gracefully. This handler will set error http response code in the response and convert any un-handled exceptions into error response json structure defined in the sunlife approved SwaggerHub API design template.



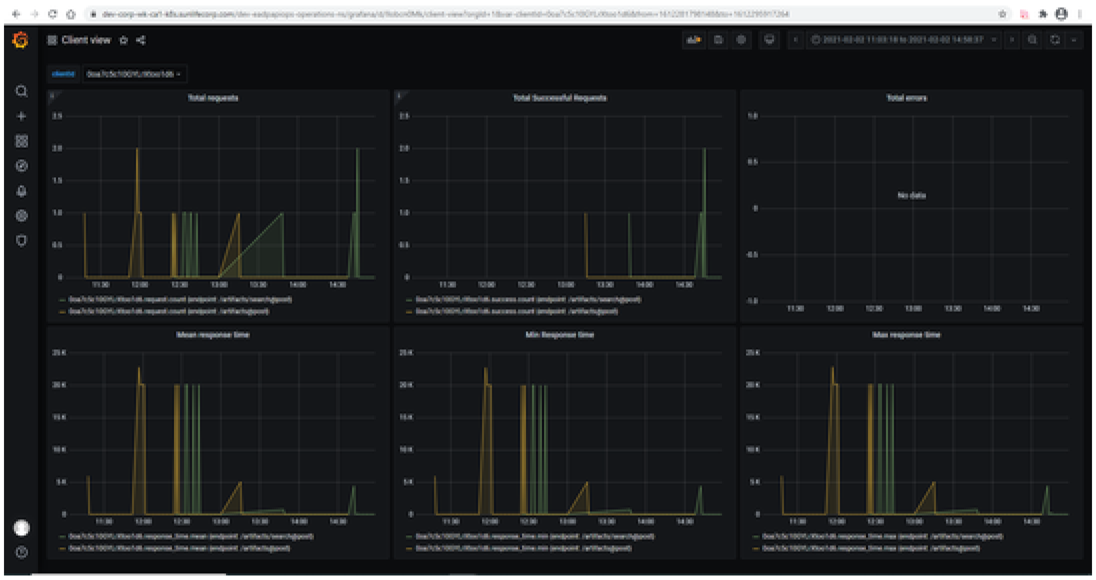
```
{
  "data": null,
  "notifications": [
    {
      "code": "E4220010001",
      "message": "Error code 0001 message is being reported",
      "timestamp": "2020-09-25T17:55:04.288+00:00",
      "metadata": null
    }
  ]
}
```

MetricsHandler

The metrics handler in the light-proxy captures all the request/response information and periodically send to the InfluxDB. Users can view the metrics per API or per client for response time, success count, failure count etc.

Within the metrics handlers, we collect the data per request and cache them so that we can send the aggregated and statistic info to the InfluxDB in certain time interval. For each backend API, given the volume of the request, this interval can be set from 1 minute to 1 hour. This is to ensure meaningful data is send to the InfluxDB and not flood the it at the same time.

The same set of data is send to the InfluxDB twice with different keys. It gives the two different views on the dashboard. For the API owner, he /she can see how many clients are consuming the API and the response time, success or failure count for each endpoint. For the API consumer, he/she can see how many API he/she is consuming and the response time, success or failure count for each API per endpoint. Following is the screenshot of the client's view of metrics on grafana dashboard.



TraceabilityHandler

Traceability id is used to trace the request across the entire loop of requests and responses across all services in api call chain. If you want your application to be able to have traceability, you must pass it in request header called x-traceability-id as it cannot be auto generated by the handler. Similarly, if in your application, you are calling another api, you should programmatically get this header from the current request and add it to downstream request.

CorrelationHandler

Correlation handler is responsible for adding a unique correlation-id to the request and slf4j MDC. Once present in MDC, it gets written to all logs statements by logback and hence can be used to correlated logs belonging to a particular request. This handler shines when there are more one service on the call chain and this id gets propagated from one service to next and gets logged in all logs belonging to one particular request across all services.

AuditHandler

The purpose of the audit is to identify abnormal situations, hence the audit handler audits all the requests that result in error(response status code greater than 400). It makes an entry each into the /proxy-audit.log.json file inside the container as audit log. As of now, the splunk agents are configured to read logs from this file location and publish to splunk for storage and querying. In future, we may want to send audit logs to some system other than file or Splunk as it may contain sensitive info related to request and response.

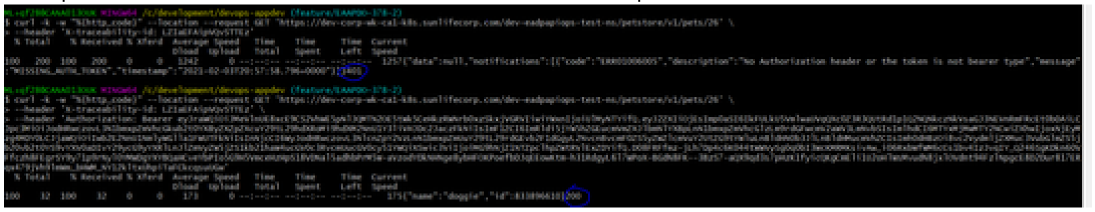
OpenApiHandler

It parses the OpenAPI spec based on the request path and method and attach an OpenApiOperation object to the exchange.

JwtVerifyHandler

The primary responsibility of this handler is to validate the signature on the token present on the Authorization header of incoming request. It downloads the public key hosted on the okta jwk set uri and cache the key locally for future requests. You will get the jwkset uri when you onboard a client with okta, here is how an example to looks at https://sunlifeapi.oktapreview.com/oauth2/aus66u5cybTrCsbZs1d6/v1/keys.

e.g. look at the screenshot in the below example, the first request was sent without token in Authorization header and failed with 401 response code. The second request was sent with a valid token and succeeded with 200 response code.



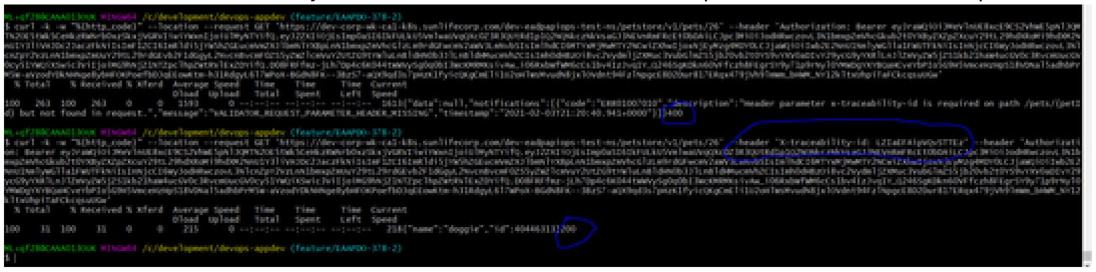
```
[illegible] MINGW64 /c/development/devops-appdev (feature/[illegible])
$ curl -k -w "%{http_code}" --location --request GET 'https://dev-corp-wk-ca1-k8s.sunlifecorp.com/dev-eadpapiops-test-ns/petstore/v1/pets/26' \
> --header 'X-traceability-id: [illegible]'
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   200  100   200    0     0   1242      0 --:--:-- --:--:-- --:--:--  1257{"data":null,"notifications":[{"code":"[illegible]","description":"No Authorization header or the token is not bearer type","message"
:"MISSING_AUTH_TOKEN","timestamp":"2021-02-03T20:57:58.796+0000"}]}401

[illegible] MINGW64 /c/development/devops-appdev (feature/[illegible])
$ curl -k -w "%{http_code}" --location --request GET 'https://dev-corp-wk-ca1-k8s.sunlifecorp.com/dev-eadpapiops-test-ns/petstore/v1/pets/26' \
> --header 'X-traceability-id: [illegible]' \
> --header 'Authorization: Bearer [illegible]'
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100    12  100    12    0     0    173      0 --:--:-- --:--:-- --:--:--   175{"name":"doggie","id":[illegible]}200
```

ValidationHandler

This handler uses OpenAPI 3.0 specification during the runtime to validate the request and response. The openapi-validation handler is wired in the request/response chain of the light-proxy and the openapi.yml file is injected into the config folder as a configuration file to drive the validation. So, if the incoming request doesn't match with what is defined in the OpenApi spec, the request is rejected 400 status.

e.g. look at the screenshot below, the first request was sent with 'X-traceability-Id', which is a mandatory header on OpenAPI spec, and failed with 400 response code. When the 'X-traceability-Id' header was added to it, the same request succeeded with 200 response code.



```
[illegible] /c/development/devops-appdev (feature/EAAPDO-378-2)
$ curl -k -w "%{http_code}" --location --request GET 'https://dev-corp-wk-ca1-k8s.sunlifecorp.com/dev-eadpapiops-test-ns/petstore/v1/pets/26' --header "Authorization: Bearer [illegible]
[illegible]
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   263  100   263    0     0   1193      0 --:--:-- --:--:-- --:--:--  1193{"data":null,"notifications":[{"code":"ERR11004010","description":"header parameter x-traceability-id is required on path /pets/{petId} but not found in request.","message":"VALIDATOR_REQUEST_PARAMETER_HEADER_MISSING","timestamp":"2021-02-03T21:20:40.941+0000"}]}400

[illegible] /c/development/devops-appdev (feature/EAAPDO-378-2)
$ curl -k -w "%{http_code}" --location --request GET 'https://dev-corp-wk-ca1-k8s.sunlifecorp.com/dev-eadpapiops-test-ns/petstore/v1/pets/26' --header "X-traceability-Id: [illegible]" --header "Authorization: Bearer [illegible]
[illegible]
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100    31  100    31    0     0    215      0 --:--:-- --:--:-- --:--:--   218{"name":"doggie","id":40446313}200

[illegible] /c/development/devops-appdev (feature/EAAPDO-378-2)
$
```

ProxyHandler

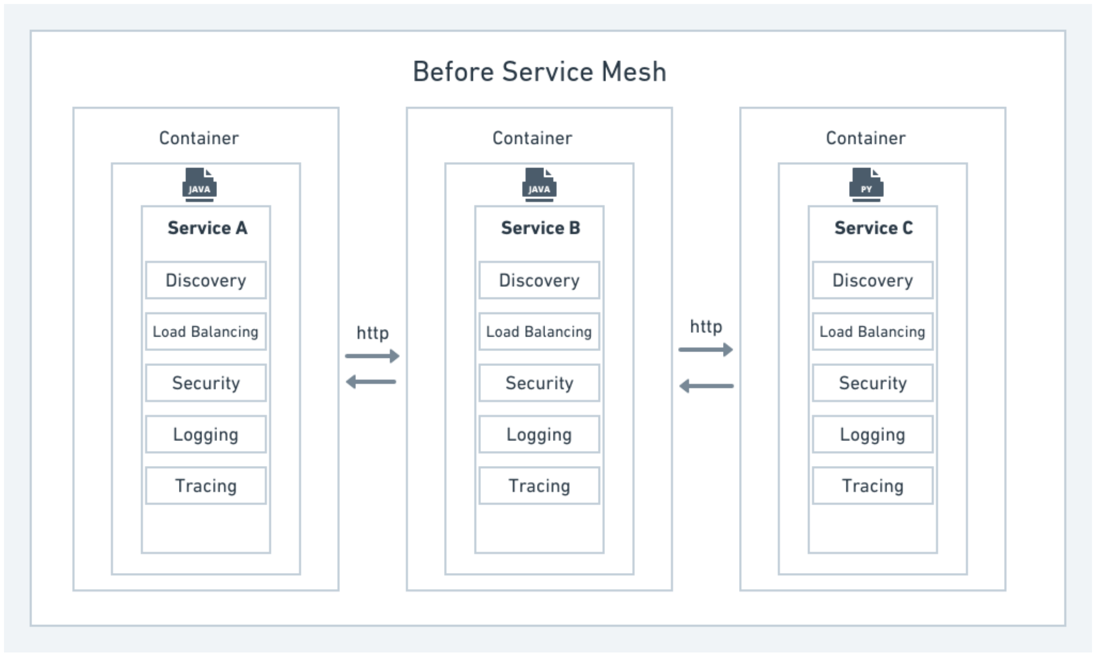
This handler gives light-proxy application the feature of being a proxy. Its main function is to forward the requests to backend api and return the response from the backend api to the api client. This is the last handler in the chain so that all other cross cutting concerns can be applied to incoming requests before it can be forwarded to backend API.

What is light proxy?

Light Proxy is a sidecar implementation of Service Mesh Using Light4J. Light-Proxy is a Light-4j application, it's a standalone server that forwards incoming http requests to the destination with executing predefined features (eg. logging, tracing, security verification, specification validation).

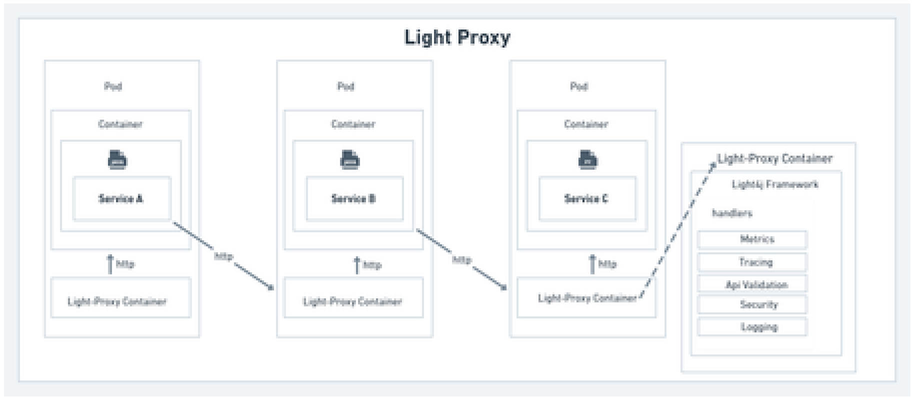
The definition cannot be stated in advance if you don't actually know what it's doing. So let's make a comparison between before Service Mesh and after Service Mesh.

Before Service Mesh



1. Those functionalities (service discovery, load balancing, security, logging, tracing etc.) are essential in a microservice platform, which is called cross-cutting concerns.
2. Each service is required to implement these cross-cutting concerns.
3. For the services using same language eg. Service A and Service B are using Java language, those cross-cutting concerns can be implemented as a standalone library, and be included in each Java service. (Light4j is using this pattern).
4. For different languages, eg. "Service C" is using python, maybe there's a "Service D" is using NodeJS, they will need to implement those cross cutting concerns in their language too, and also need define specifications so that all the cross-cutting concerns will follow same pattern across the microservices

After Service Mesh



1. Those cross-cutting concerns are included and handled in a sidecar container.
2. When the service needs to send request to another service, it will send to the sidecar first, then the sidecar will handler will handle the cross-cutting concerns eg. do service discovery, adding the tracing id etc. When the services receive a request, it will send to the sidecar first, then the sidecar will handler will handle the cross-cutting concerns eg. do JWT verification, specification validation etc.
3. The services "Service A", "Service B", "Service C" don't know how to do any of those features, they only have their business logic inside the service, which will make development more efficient.
4. Those cross-cutting concerns are managed in the same manner. If we want to add another feature, for example, audit all the request, we just need to add this feature to the sidecar image, whoever use this image as the sidecar will get this feature without changing any code of services.
5. The sidecar pattern supports different languages since the service and the sidecar are using http to communicate.
6. With this sidecar pattern, it adds more complexity and latency due to adding another layer between service calls.

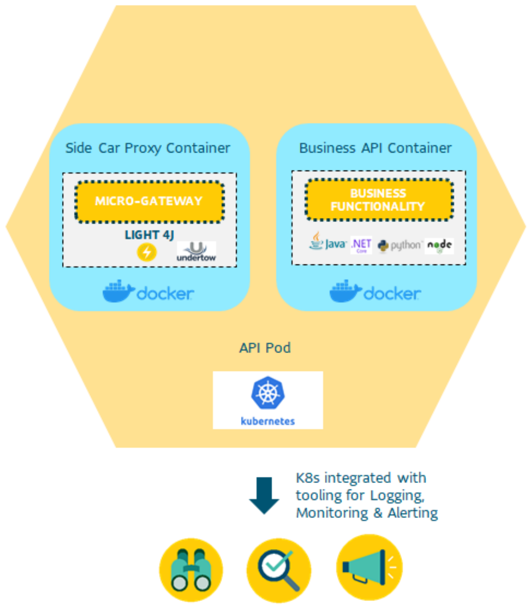
Deployment Pattern

The light proxy is deployed as a side car container in the same kubernetes pod as business API. All the traffic to business API flows through the side car and this pattern helps us apply different features like security, metrics collection, auditing on the incoming requests without the need for the business API teams to worry about these technical capabilities.

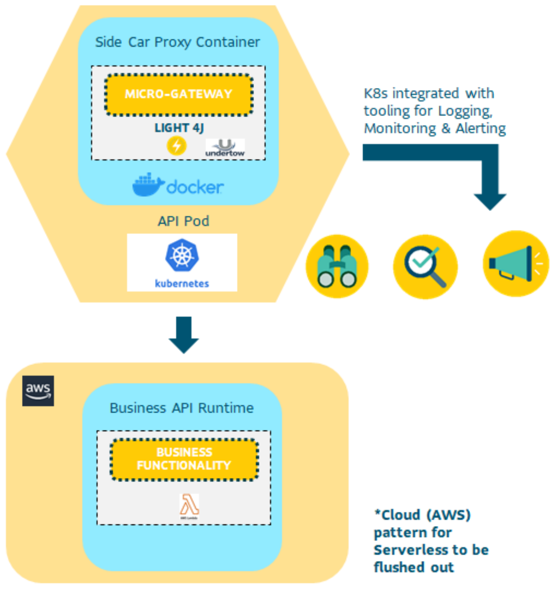
Since the proxy talks to backend business API over the network using HTTP, the proxy doesn’t assume anything about the technology, framework or platform used to build the business API. This gives delivery teams liberty to choose any programming language or framework to build their applications with only one condition - the application should be capable of listening to HTTP requests.

Deployment Patterns

Non-Serverless Pattern



Serverless Pattern (AWS)



***Cloud (AWS) pattern for Serverless to be flushed out**