Microservices Foundation Template - Setup Guide

[draft2]

# Purpose

* Guide to get the template running at the minimum functionality required for most usages
* Guide to changing some components in the template

# Scope

* Setting up template
  + Copy-paste
* Calling other services
* JWT
* Modifying the template
  + Changing the logger
  + Changing the metrics
  + Changing the http service implementation
  + Configuring Token handler
  + Changing/Configuring authentication service

# Assumptions

* You are reading this guide together with the template code. If not, the template code can be found [here](https://github.com/dotography-code/LiftMicroservices) (private Github repository).
* Installed on dev environment:
  + Java runtime environment 8
  + Scala runtime 2.11.7+
  + SBT 0.13.8+
* Knowledge of Scala – syntax and grammar, basic functional programming, object-oriented programming, basic type system
* Knowledge of Lift – Framework flow, using Boot, implementing APIs
* Knowledge of Git – Essential knowledges
* Knowledge of Json Web Token – What it is, how it works
* Knowledge of OS – commands, installation, and configuration

# About this template

This Lift Microservice Template is created to allow rapid creation and deployment of a new microservice. It is capable of serving RESTful services and communicating with other microservices.

The template uses the Liftweb framework and its conventions. It contains:

* JSON Web Token encoder/decoder
* Future-based Synchronous HTTP service (implemented with Akka-HTTP library)
* Default user authentication implementation (Kibali microservice as default)
* Test codes
* Logging implementation
* Metrics (default CloudWatch)
* Docker settings file and deployment script
* Cluster deployment script

This template can and should be used as a starting point for the microservice repository.

# Getting started with development using the template

#### Important Note:

* File paths are relative to the template directory.
* The file paths that starts with /// is shorthand for src/main/scala/lm/  
  For example, ///Boot.scala means src/main/scala/lm/Boot.scala

## Starting the web server

The template itself is ready-to-use out of the box. Simply clone the template to your computer. Open build.sbt and change the name (and optionally version) parameter. Run sbt in a command prompt at the template’s directory to enter the SBT prompt. Then enter container:start to start the server. The default location of the server should be <http://localhost:8080>.

The template itself includes simple RESTful endpoints to demonstrate the implementation. The code of interest is at the file ///Test.scala. This file demonstrates a typical way to implement endpoints in Lift framework using the RestHelper trait.

Try, in a web browser or a cURL command, sending a (Get) HTTP request to <http://localhost:8080/test/id> to make sure everything is running properly. There should be a JSON object returned. Note that compared to the pattern stated in the endpoint, the URL path has a test component in front of the pattern.

## Creating endpoints

If you look at ///Boot.scala, you will see that the Test object (in ///Test.scala) that provides the endpoints is not directly appended to LiftRules as usual, but done so through API.register method. This API object contains the code for logging and Metrics that will transparently activate with every endpoint request received.

The API.register method also has the string test supplied as an argument. This test is the prefix added to the URL path earlier.

Therefore, you could implement endpoints the same way as usually done in a Lift framework, i.e. using RestHelper like the previously mentioned Test object. Just make sure to use API.register method in Boot to activate the endpoints and not the usual LiftRules.statelessDispatch.append.

# Calling other services

The main concern of developing a microservice system is the communication between microservices. This template provides a way to easily make an HTTP request to another microservice.

## The current state:

* Synchronous implementation only (no message queue yet)
* Future-based (non-blocking)
* Akka-HTTP implementation provided (can add your own)
* Have customized Exceptions and helper classes
* JSON oriented (no XML/HTML specific helpers), with some project-based conventions

The resource to Akka HTTP can be found here: <http://doc.akka.io/docs/akka/2.4.4/scala/http/#http-scala>. In particular, this section: <http://doc.akka.io/docs/akka/2.4.4/scala/http/client-side/request-level.html>

## Quick use guide

To use, create a class or object (or trait) and extends AkkaHttpCaller, then use the method call or other convenience methods to make a HTTP request.

//don’t forget imports

object MyService extends AkkaHttpCaller {

def whatIsShazbot: Future[HttpResponse] =

get("http://example.com/search", Map("q" -> "shazbot"))

}

// then to use

val vgsResponse = MyService.whatIsShazbot

## Digging deeper

For better understanding, first let’s take a look at the trait AkkaHttpCaller in ///service/implementations/Akka.scala. The first thing to point out is that it extends HttpServiceDelegate (///service/baseProvider/delegator.scala) along with the method call with signature Request -> Future[Response]. This method is the main method that will return a Future of the response. It can be used as a basic way to make a HTTP request.

But to make life easier, the trait AkkaHttpCaller itself has several convenient methods to easily make requests for various HTTP verbs (e.g. GET, POST) with easier-to-use arguments, such as get and postJson. It also includes response parser and a method to convert Akka’s HttpResponse to LiftResponse.

An example of how to implement a service communication is the object UserService, found at ///service/UserService.scala. Simply extend AkkaHttpCaller, and you will have the necessary concrete methods to make a HTTP call. The only crucial missing piece is the URI of the target microservice resource, which is left to you to materialize either by hard-coding/property configuration or otherwise.

### def parseAndProcessJsonResponse

In object UserService, you might notice the use of parseAndProcessJsonResponse. This method helps to easily parse a JSON into a JValue, and let you convert the JValue to any type. The first argument expect a Future[Response] and the second argument expects an JValue -> Option[T], where T is any type of your conversion. A Some will results in Successful Future while a None will cause the Future to result in Failure with specific exception classes, which could be found in ///service/package.scala. In summary, the possible outcomes of the resulting Future called this way are:

* Success of result transformed
* Failure containing ServiceFailResponseException: the HTTP succeeded, the JSON response indicates an expected error(s). (The template structure of the expected error JSON is the case class ErrorResponse in ///service/package.scala)
* Failure containing ServiceBadResponseException: The HTTP succeeded, but the response JSON is not in the expected shape or not a JSON
* Failure containing TimeoutException: The HTTP response was not returned in time
* Failure with any other Exception thrown by your code
* Failure with unexpected Exception or Throwable: this is not a good sign.

Handle the result as you see fit. Remember not to use Await.

# Json Web Token (JWT)

The template provides the object JwtCodec (///jwt/JwtCodec.scala), capable of encoding the claims to token string and decoding the string back to a native Jwt type. The claims are wrapped in the class ClaimSet (///jwt/ClaimSet.scala), which indicates what claims are expected in the token, and in what format. The out-of-the-box implementation requires a name, a user id, and an expiration timestamp. Decoding a JWT without all the required claims will result in failure, even if the signature is correct.

The class JwtAuthentication (///jwt/JwtAuthentication.scala) is designed to be used as an authenticator in the Lift request lifecycle. It expects the token string to be in the Authorization header of the request. If authentication is required, add the JwtAuthentication to LiftRules.authentication in Boot as per normal. Don’t forget to implement LiftRules.httpAuthProtectedResource as well.

You can, and should, set the secret key in /src/main/resources/props/default.props using the property jwt.secret. It is recommended to be at least 12 characters long. This key MUST NOT leave the server or be logged. It is also a good idea, though not strictly required, to change the key once in a while.

Remember NEVER to include any sensitive information within the token, as it could be read by anyone.

# Modifying the template

## Logger

The default logger configuration is at src/main/resources/logback.xml. Look inside the props subfolder for mode-specific logger settings. The main thing to note is in src/main/resources/props/default.logback.xml the PAPERTRAIL logger is an external service to aggregate logs from all microservices to the same data location.

More info on logger settings could be found at  
<https://www.assembla.com/wiki/show/liftweb/Logging> and <http://logback.qos.ch/manual/configuration.html>

## Metrics

The default implementation is CloudWatch. To change the metric service implementation, go to the object Metrics in ///Metrics.scala. Change the method report, which is responsible for initiate the new metrics builder.

If you need to change the columns to report, you can do so in API.scala (///API.scala). Look for the usage of Metrics.

## New HTTP Service implementation

If Akka-HTTP does not suit the need of the microservice, or if you simply want to add more, take a look at the trait HttpServiceProvider in ///service/baseProvider/ServiceProvider.scala. This is the trait you extend to provide the heavyweight implementations of the service. For example with Akka, AkkaHttpServiceProvider, which extends HttpServiceProvider, houses the heavy ActorSystem and the ActorMaterializer. To extend or instantiate this trait multiple times would be resource intensive. Instead, actual calls are done through AkkaHttpCaller, a subclass of HttpServiceDelegate, which passes the request to AkkaHttpServiceProvider and back, and provide convenient functions.

HttpServiceProvider (abst)

HttpServiceDelegate (abst)

object AkkaServiceProvider  
(heavyweight)

trait AkkaHttpCaller  
(lightweight)

UserService

UserService

UserService

call

call, get, put, etc.

Actual HTTP request/response

In other words, put the heavy items in a single HttpServiceProvider, and use multiple lightweight HttpServiceDelegate as middlemen.

If the implementation is lightweight throughout, feel free to extend from the HttpServiceProvider directly.

## Configuring Token handler

The template’s JWT code is not made to be able to easily change the backing code entirely, but it is capable and flexible enough to customize and configure the token and its handling.

Apart from the aforementioned secret key, you could change the lifetime of a token by setting the property jwt.tokenLifeSecond in /src/main/resources/props/default.props, or related files for different run mode. The algorithm used could also be changed in ///jwt/JwtCodec. See the library’s guide for supported algorithm. <https://github.com/iain-logan/jwt>

You can also change the claims in the token by altering the case class ClaimSet (///jwt/ClaimSet.scala) Remember to change the claim classes in the same file as well. There are other claims implemented according to the JWT specifications implemented in the library as well.

## Changing the User authentication microservice

The default microservice is Kibali. To change the target microservice, simply change the url in the service.user.uri property in the Props file. Also change the url path, the required request body, and the case classes used for extracting the JSON response.

# Deploying a docker instance

The template provides a shell script to help deploy the microservice via docker. Here are the steps.

1. In the shell command prompt, browse to the template folder
2. Run lm init –n {service\_name}, the {service\_name} could be string, but to make thing simple, it is recommended to use the project name in build.sbt
3. Fire up SBT, enter webappPrepare command to compile and package the files
4. Exit SBT, run docker start {service\_name} the docker instance should be Running now
5. To update the package contents, run docker stop {service\_name}, then repeat step 3 to 4

Refer to /README.md for detailed information on deployment.