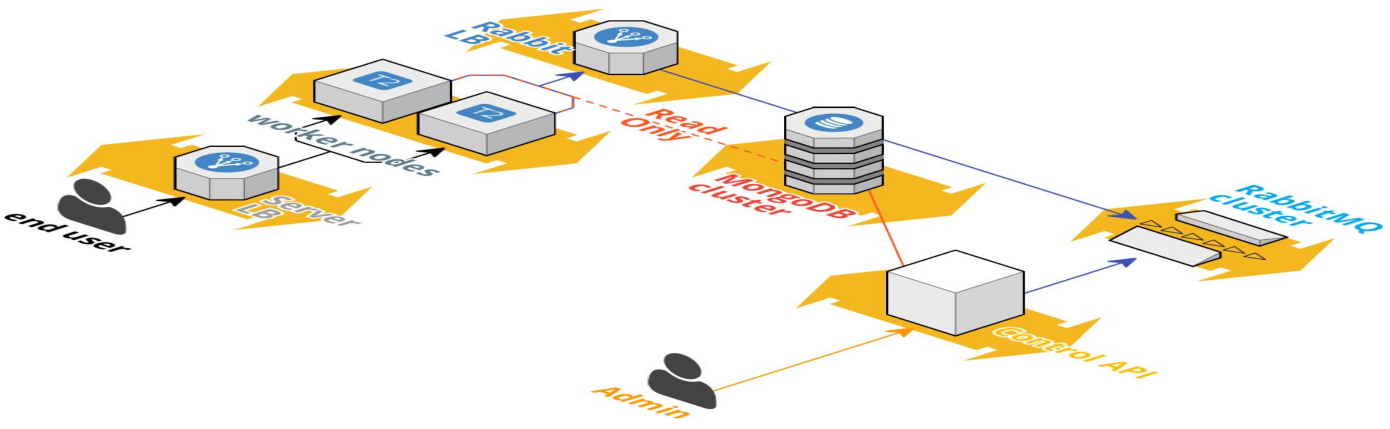
FLOW ORCHESTRATOR

Software Architecture Document



INTRODUCTION

Purpose

The Software Architecture Document (SAD) provides an architectural overview of the Flow Orchestrator system. it presents a number of different architectural views to depict different aspects of the system. it is intended to capture and convey the significant architectural decisions which have been made on the system. In order to depict the software as accurately as possible, the structure of this document is based on the “4 + 1” model view of architecture.

Scope

The scope of this SAD is to depict the architecture of the Flow Orchestrator application which is designed as a “Reactive System” to run “flow(s) of execution”.

Definitions, Acronyms & Abbreviations

PU	Power User
SAD	Software Architecture Document
FO	Flow Orchestrator
FOE	Flow of execution

Assumptions & Understanding

1. “FO” system will orchestrate and manage the entire flow of execution based on the flow diagram provided by the power user.
2. The input fields/data (interface spec) of a component will be predefined, which will be provided as part of the user guide to the power user for data exchange/interface understanding.
3. Any new component that needs to be deployed/made available to the power user will have to go through a deployment process and user guide to be updated with the interface spec for the same. Please refer to Deployment Guide for deploying any new component.
4. The input for the start point will be provided to this "FO" by whichever system that would like to invoke this flow. The input parameter will contain a mandatory field named “flow-name” and indicate which flow as constructed by the Power User should be used during this call. E.g. "flow-name":"flow-config-create-employee".
5. All the input parameters and the response parameters will be aggregated and made available for any subsequent flow in the “FOE”. This is to enable the Power User to be able to combine output from 2 components as an input to a subsequent component?

6. It is assumed that async flow can be depicted between 2 components directly without a messaging component.
7. It is assumed that there will not be one to many kind of flows. The "FOE" is expected to be one after the other, although the individual call can be synchronous or async call as required.
8. REST outward component will assume HTTP status 200 as call successful and proceed further in the flow, there is no data expected in the response from the outward interface.
9. Any component within flow not being available will be dealt with to re-instantiate the same by the design to be reactive, however external systems is out of scope (outward interface calls)
10. The threshold for outward interface call (REST Outward) would be 3 times and subsequently, an error would be logged to alert the situation.
11. Data exchange format between components will be standardized like json format
12. Data transformation will indicate the input field to which the transformed data needs to be applied.
13. The field names are supposed to be unique, if not the latest value will be overwritten and used there is NO absolute path used in the field/param names.
14. Logic for transformation would use fields from the output of the source component and/or any constant values only.

SOLUTION OVERVIEW

This overall solution of this Reactive system is based on Java and Akka framework along with few other tools that aid certain features. Primarily the solution contains a Flow Orchestrator component that in itself is an Akka actor which manages the entire "FOE" based on the flow config as constructed by the "PU".

This then calls various components to leverage certain functions and then weaves the relevant Component actors in the flow as required such that the components are called at the right time in a right manner (sync vs async) etc. Once the entire flow is completed the "FO" return the responses back to the caller, however in the case of an async call the control is returned to the caller immediately but with no response values, but the "FO" takes the control from then on and ensures the execution of the full "FOE".

ARCHITECTURAL Choices with Pros & Cons

This section describes some of the key software architecture goals, choice of framework/technology with its pros & cons.

3.1 Reactive System (Technical Platform)

Reactive System is a key architectural goal / requirement for this application.

As defined in the Reactive Manifesto, "Reactive is responsive, Reactive is resilient (failure handling), Reactive is elastic (highly scalable), and Reactive is message-driven." The message-driven component is essentially what allows the other three characteristics of Reactive to be supported.

Given the above need, the solution for this application primarily revolves around Java language and Akka framework to provide the foundation / platform to build a Reactive System.

Why Java, Akka Pros & Cons?

Java provides access to large developer base with Java knowledge. This has been in the industry for long time now and large amount of resources are available. It is an reasonably easier language to comprehend. These are some of the pros of choosing Java language. From a con standpoint, it is an object oriented programming language, so if the developer is not well versed with the same, there is a possibility that code can become very complicated / procedural that could lead to maintainability issues.

The requirements of highly scalable, responsive, and resilient systems already pointed to the Actor Model of Computation as the potential best-fit. There are only few top contenders in this space: Erlang, Vertx, Quasar and the other is Akka.

Considering blocking vs non-blocking model, non-blocking model would be a better choice when it comes to handling systems that are more metadata/non-functional in nature like routing etc given the throughput expectation would be far higher. This brings the choice down to Vertx (due to EventBus based non-blocking implementation) and Akka (async Actor based non-blocking implementation). Also Vertx and Akka both support Java based API.

Akka offers network transparency by default. It is based on a hierarchical actor model and provides supervision ability to manage failure in an hierarchical manner. Akka does not need a third-party software and also provides cluster-aware-routers and metrics. Akka's Actors can be deployed anywhere on the network, with support for clustering, replication, load-balancing, supervision etc.

Akka also ensures FIFO message ordering between two actors, as well as sending messages to a concrete actor in the hierarchy whereas Vertx uses a central EventBus with publish/subscribe semantics. Given some of these features, AKKA is the top contender and the right choice for bringing in the Reactive ability.

The con of Akka could be that it is a little low level and does not directly support Http etc. however these can be overcome by leveraging Spark / Play frameworks as needed, these frameworks are built on top of Akka.

3.2 State Management

The “Flow Orchestrator” application generally does not intend to maintain state between the flows, however in case of a synchronous call the response received from the component call is preserved so as to be able to provide that as input for any subsequent flows of execution. The application intends to use the built in Future concept in Akka framework for achieving the same.

Why State, Future, Pros & Cons?

Ideally maintaining state in a flow of execution kind of application is not a great idea given it brings in certain limitations. However given the requirement to support synchronous flow and to give back the response to the waiting caller, Akka Future is being proposed to bring the callback mechanism between FO and Component actor for providing the response values back to FO.

3.3 Security

The system must be secured, so that the authenticity of the caller is well established before the entire flow of execution is started:
The application must implement basic security behaviors. Authentication - This could be based on a token given the caller mostly would be a system and not necessarily a end-user. Authorization - This may not be necessary as there is no distinction in the flows of execution based on the caller profile.

3.4 Persistence

Using a relational database would be the choice something like MySQL that is lightweight and not cost huge amount of money would be the choice. However the choice of storage should be made based on the component and the type of information to be stored keeping in mind clustering, scalability etc. aspects.

Metadata of the deployed components, flow execution configuration are to be using a relational database to enable multiple systems to access the same data. However for the prototype simple files using file system has been used given the constraint of not having a relational database at this point in time.

If we were to use a database for storing the response values to maintain state between the flows, then the database could potentially become a bottleneck at very high load because this storage has to be done at a “FO” level across all different component executions. Given this the decision is to make the “FO” be the aggregator and maintain the response data in a local thread context until the entire “FOE” is complete.

Why Relational Database?

Relational Database gives a quick and clean solution to store information and maintains data integrity, transactional behaviour etc. out of the box. It can be used as a central location to access from multiple cluster nodes/actors especially if the operation is primarily read operation with less write. These are some of the pros, from a con standpoint, if there are too many thread writing and transaction atomicity has to be imposed at a higher level then it could potentially become a bottleneck from a performance standpoint.

3.5 Performance, Reliability & Scalability

Given that this is a non-functional/router kind of system supporting "FOE", the system is expected to be able to respond very quickly with very little overhead between the component invocations and the caller. Also the reliability (failure handling mechanism) and scalability (containerization, clustering) is a very important expectation of this system. All these 3 goals are addressed through using Akka framework.
For Prototype: For prototype purposes the Actor supervision strategy, clustering, sharding etc has not been implemented and needs to be implemented when providing the full system.

FUNCTIONAL VIEW

Use Case

Generally various use case diagrams/views would be presented for highly functional systems, given this system's design is to be like a metadata handling system and provide the flexibility of creating various use cases to the "power user", we have provided our functional understanding of the system as below:

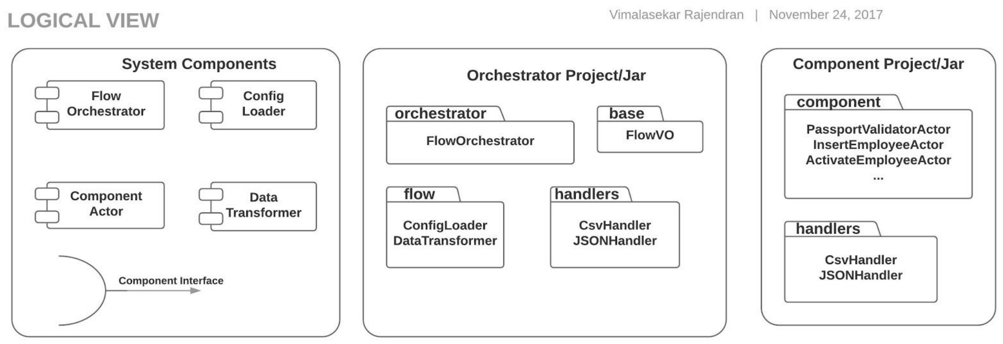
The functionality of this system is as follows:

1. Functionality of the system is to run "flow(s) of execution".
2. Flow component examples are "REST Calls", "Custom Apps", "Pub/Sub" etc.
3. The system should be able to support Synchronous and Asynchronous flows.
4. Flows are "designed/configured" by a "power user" who is aware of the "data" being exchanged between the components and hence would also be able to provide any data transformation/mapping needs as part of this flow-configuration.
5. Flows are like workflow steps connecting various individual components either in a sync or async manner to form a functional use case.
   a. E.g. Create a new Employee could be an use case which could be achieved by connecting various individual components and performing the required data transformations is any.

LOGICAL VIEW

Components & Packages

Generally all the layers, packages & classes visualized to be part of the system is depicted here and this would be an iterative process of updating the architecture document as the system evolves. For the purpose of this exercise, I have updated this section with the packages and classes of the prototype that has been developed and provided.



Broadly the system is modelled around 4 - 5 different components, the responsibilities of which are detailed as below:

Config Loader : Config Loader is a component that is responsible to load all the metadata information of all the deployed components, including the class name of the component actor using which those component actors could be instantiated dynamically at runtime. It also loads the constructed configuration of the "flow of execution" by "Power User", it assumes there could be many such constructed configurations provided and is geared to load all of them.

All of these configurations are preferred to be in a database under master tables, such that any number of JVM nodes in the cluster could take the same information.

For Prototype: For prototype purposes the config loader has been coded to load the metadata from a csv file in a fixed directory structure named "config" under the current project folder.

Flow Orchestrator : Flow Orchestrator is the central component that manages the entire flow of execution. Flow Orchestrator is in itself is an Akka actor primarily to support both synchronous and asynchronous calls in the initial step itself from the caller.

After all the flow/steps are completed the entire aggregated response values are returned back to the caller in case the caller invoked the orchestrator system in a synchronous way, if the caller had invoked the system in an Async way, the control would have returned back to the caller then and there without any response values, but the Orchestrator would have taken control of the entire flow of execution and complete the full flow, but nothing gets returned after that to the caller.

Data Transformer : Data Transformer is the component that performs the necessary data transformation for any additional target input fields based on the transformation rule as defined by the "PU".

Data Transformer ideally could be based on frameworks like Apache Camel Message Translator or talend or similar open source data transformation tools. Many such tools are part of larger ETL platforms, but support direct message translations. The choice should really be based on the level of flexibility and complexity that we anticipate for the data transformation rules.

For Prototype: For prototype purposes the data transformer has been coded as a simple mechanism to handle custom string based transformation rule. The format for defining the transformation rule and explanation for each element is provided in the Process View section in this document, please refer to the same for the nuances on string separators etc.

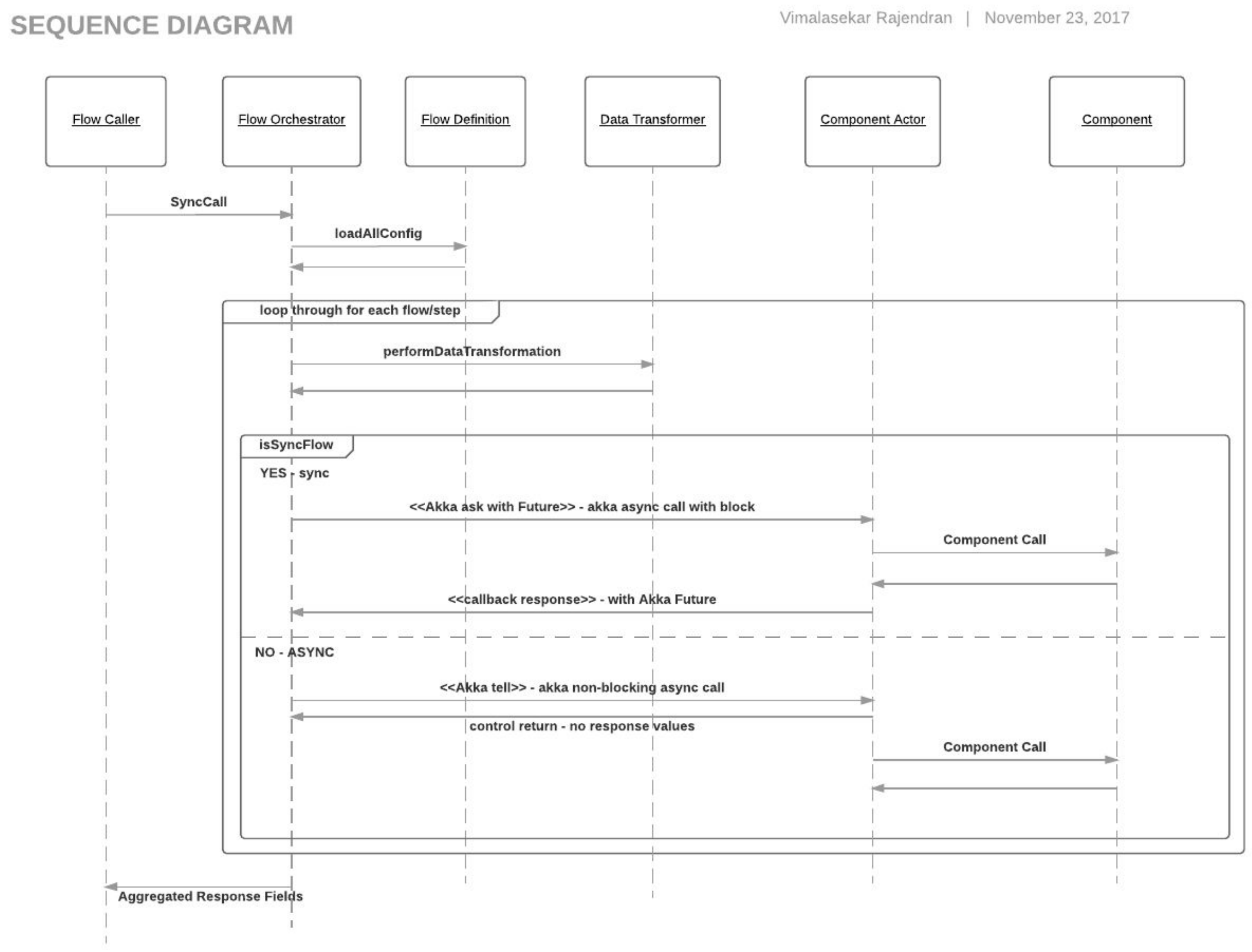
Component Actor : Component Actor the component that makes the actual call to the functional component that is involved which needs to be invoked for the flow. Component Actor is an Akka actor to support large throughput and to enable Reactive System tenets.

The interface is clearly separated between the Flow Orchestrator and the Component Actor in a manner that there is no direct dependency between them and the data exchange between these 2 components happen via json string which can be serialised and sent over the wire to any different node as required, so as to ensure non-blocking behaviour and enable the Reactive System tenets.

Component Actor is supposed to be created by the development team who create and deploy the functional components. This is supposed to be provided as a separate jar file and few metadata information needs to be filled in to make these available to the “PU” for constructing the flow of execution. Please refer to the Component Deployment section the Deployment Guide for further details on this aspect.

Sequence Diagram

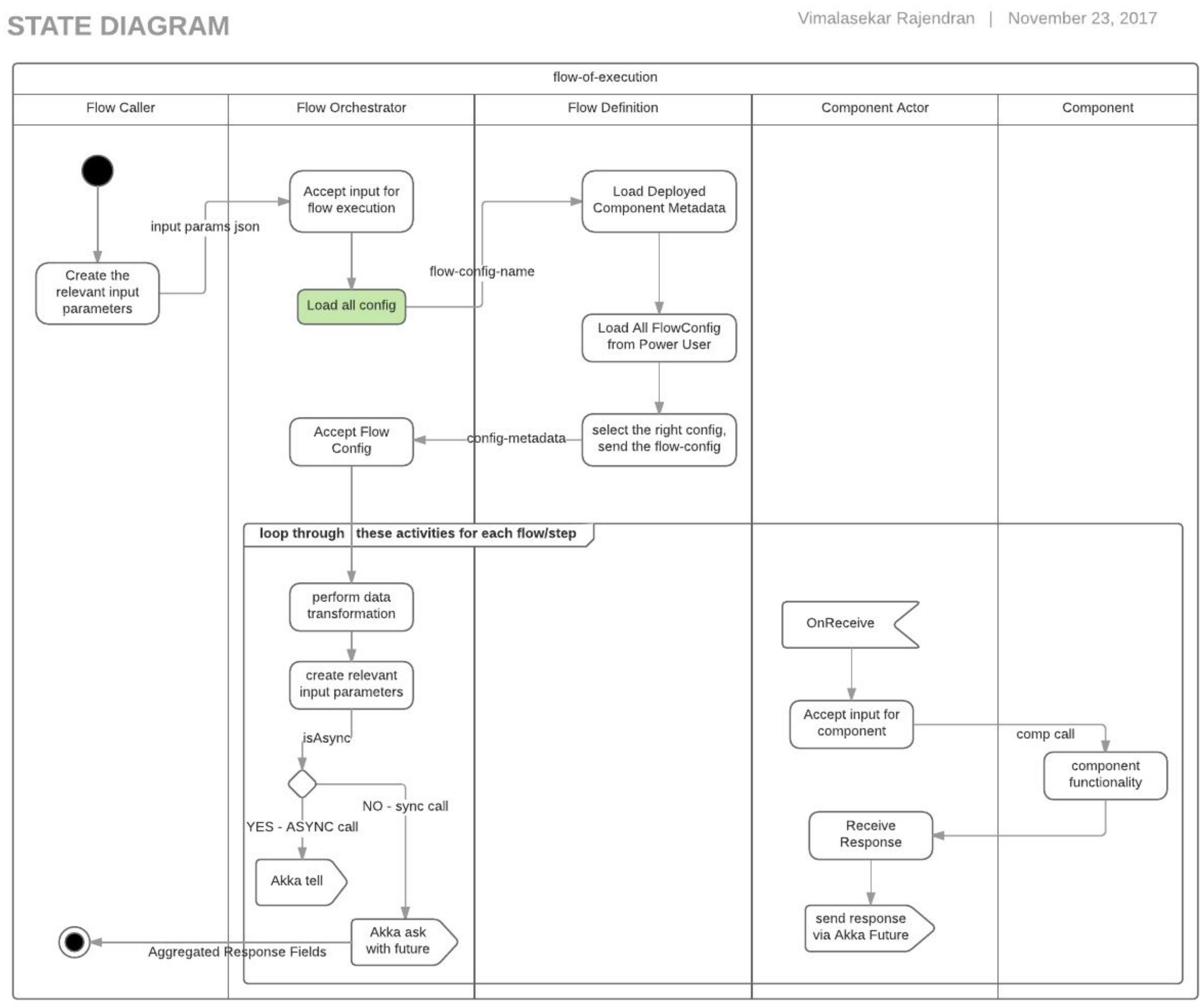
Flow Orchestrator sequence is depicted as below, the same has been explained in the "Components & Packages" section and the Process View section.


SEQUENCE DIAGRAM
Vimalasekar Rajendran | November 23, 2017
Flow Caller
Flow Orchestrator
Flow Definition
Data Transformer
Component Actor
Component
SyncCall
loadAllConfig
loop through for each flow/step
performDataTransformation
isSyncFlow
YES - sync
<<Akka ask with Future>> - akka async call with block
Component Call
<<callback response>> - with Akka Future
NO - ASYNC
<<Akka tell>> - akka non-blocking async call
control return - no response values
Component Call
Aggregated Response Fields

PROCESS VIEW

Generally the process, responsibility and the state management is detailed in this section and this would be an iterative process of updating the architecture document as the system evolves. For the purpose of this exercise, this section is explained based on the prototype that has been developed and provided.

State Diagram


STATE DIAGRAM
Vimalasekar Rajendran | November 23, 2017
flow-of-execution
Flow Caller
Flow Orchestrator
Flow Definition
Component Actor
Component
Create the relevant input parameters
input params json
Accept input for flow execution
Load all config
flow-config-name
Load Deployed Component Metadata
Load All FlowConfig from Power User
select the right config, send the flow-config
config-metadata
Accept Flow Config
loop through these activities for each flow/step
perform data transformation
create relevant input parameters
isAsync
YES - ASYNC call
NO - sync call
Akka tell
Akka ask with future
Aggregated Response Fields
OnReceive
Accept input for component
comp call
component functionality
Receive Response
send response via Akka Future

Config Loader : As described above the Config Loader component is responsible to load and provide all the metadata information of the deployed components and the flow config constructed by the “Power User”.

All these metadata & flow config information is read only once when the Config Loader component is initialized and stored in static memory, subsequent calls to load the configuration will only return the loaded static data.

if a situation arises to directly invoke the loading process without having to bring down the entire system, an administrator could send a specific type of message to the orchestrator actor to invoke an on-demand load function in the config loader to refresh the config data in the static memory.

The “flow of execution” configuration is stored against a flow id which is a mandatory field that any caller of the system should provide to indicate which “flow of execution” they prefer to run. Any number of new configurations can be provided by the “Power User” that could be dynamically loaded by the config loader through a message as defined in the above section.

Flow Orchestrator : As described earlier, Flow Orchestrator is the central component that manages the entire flow of execution.

Flow Orchestrator uses the Config Loader component to load all the metadata information of the deployed components, subsequent to which it uses a JSON handler to process the input values from the caller and takes the flow id from there using which it gets the entire flow of execution config through the config loader.

Once this is obtained it goes through the flow of execution and for each flow/step constructed, it takes the component actor class name from there and then creates an ActorRef. Now, if there is any data transformation rule provided, then the Data Transformation component is invoked to perform the required data transformations and provide any input fields/params post the transformation, which is also stored into a master list of params. Post this, the orchestrator component calls the component actor by providing the component relevant input parameters from the master list of params. The call is made in an async or sync manner depending upon how the flow config is constructed.

If the flow is supposed to be a synchronous flow, then the response from the component actor is taken and stored into a master input list which is an aggregation of all the fields/parameters from caller and all the synchronous responses in each flow. This aggregation follows a latest important mode in which if a particular field in this master list receives a different data in any response then the latest response takes precedence and the data is overwritten to take the latest value for any subsequent flows.

After all the flow/steps are completed the entire aggregated response values are returned back to the caller in case the caller invoked the orchestrator system in a synchronous way, if the caller had invoked the system in an Async way, the control would have returned back to the caller then and there without any response values, but the Orchestrator would have taken control of the entire flow of execution and complete the full flow, but nothing gets returned after that to the caller.

Data Transformer : As described earlier, Data Transformer is the component that performs the necessary data transformation.

Data Transformer loads the transformation rules from the Flow Orchestrator based on the constructed flow by the “PU”, then the transformation rule is parsed based on the custom format (“targetFieldName@valueType=dataType=value#operator#valueType=dataType=value|operator# valueType=datatype=value$targetFieldName@valueType=dataType=value#operator#valueType= dataType=value|operator#valueType=datatype=value$……n”) and each rule is processed separately in a loop.

For each rule, the necessary value for the fields is taken from the master list of input/response fields or if the provided value is a constant then it is directly used. There could be multiple operators defined for each transformation rule, this is looped through to complete all operations for each rule/target field, once the transformation is done, the result is then stored in the same master list of input/response fields, such that the same can be made available to the component actor in case this needs to be included as an input to the component call.

Transformation rule format & explanation:

*“**targetFieldName@valueType=dataType=value#operator#valueType=dataType=value**|**operator#valueType=datatype=value$**targetFieldName@valueType=dataType=value#operator#valueType=dataType=value|operator#valueType=datatype=value$……n”*

1. “targetFieldName” – denotes the field name to which the transformed data needs to be assigned to and sent as an input to the component in this flow/step.
2. @ (at symbol) - targetFieldName level separator
3. "valueType" - denotes the type of value that is being provided. Supported values are “constant” or “field”
4. if "valueType" is "constant", then value can be a static constant value, if "valueType" is "field", then value can be a "field/parameter" from the responses of all previous components in the flow including the input from the original caller like “firstname”.
5. = (equal to symbol) - element level separator
6. "dataType" – denotes the datatype of the value being provided. Supported values are "string" or "number" only
7. “value” – denotes the value. This could be a constant value or a field name in which case the value of the field name from the master list of fields/responses will be taken and used for the data transformation.
8. # (hash symbol) - field/value level separator
9. "operator" – denotes the operation to be performed in the transformation. Supported values are "concat", "plus", "minus", "multiply", "divide" only
10. valueType, datatype, value - same as defined in points 4, 5, 6, 7
11. | (pipe symbol) - expression level separator, repeat steps 9 (operator) and 10 (value) for how many ever operators that is defined/constructed for this transformation rule.

12. $ (dollar symbol) - transformation rule separator to enable multiple transformation rules to be provided for a particular flow/step.

Component Actor : As described earlier, Component Actor is the one that makes the actual call to the functional component.

Component Actor is takes all the input parameters provided along with the transformed data for some of the input parameters and create the input in a format that the component would understand as the component could be of any type, it could be a pojo java call or RESTful service call etc.

Post perfoming this component call, the actor receives any responses from the component and sends the same back to the caller if the call was a synchronous call, if not the response is discarded given the caller would have left by now.

PHYSICAL / DEPLOYMENT VIEW

Generally the deployment view depicting the physical nodes/layers as per the infrastructure availability or constraint would be depicted here. This is generally not an iterative process, however this would depend on certain aspects of existing infrastructure capability, existing software licenses, cost for new infra / software layers etc to determine the appropriate physical structure.

Keeping today's industry, trend and availability of infra in mind, a cloud based infrastructure is being with support for elasticity, such that as the demand / throughput expectation increases, the cloud center could automatically spawn new instances with the necessary actors as needed.

It is also suggested that Docker kind of containerization is done such that multiple JVM could run within the same Node and many such Nodes could be spawned as needed as part of elastic behaviour.

This way for any scalability needs, one could first vertically scale with many Actor instances and subsequently with many docker instances and when the entire physical node is utilized to the required level then Horizontal scaling can be achieved by spawning new Nodes as required and adding to the cluster.

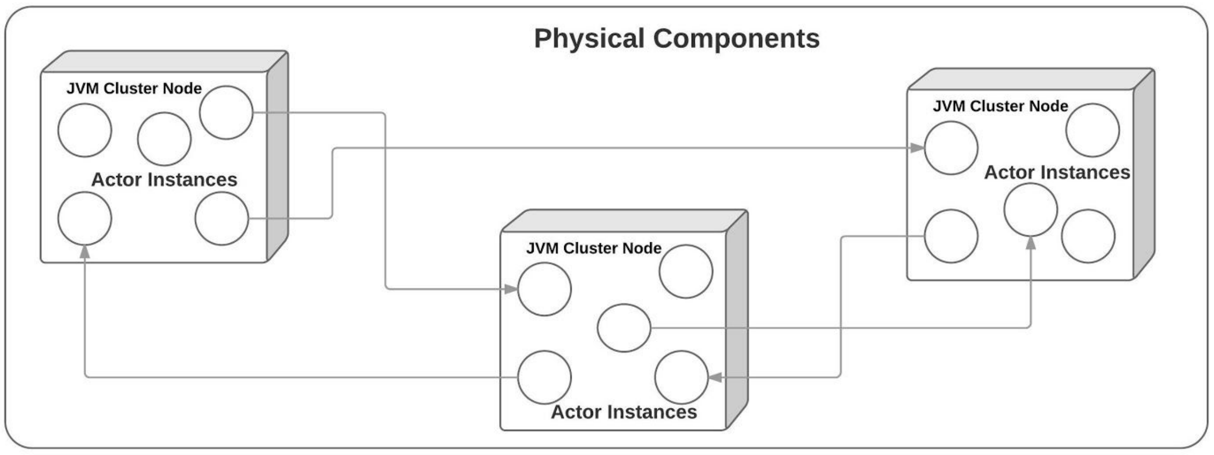
Given that Akka framework's Actor model is used and that the data exchange between all actors is standardized using JSON string as the format which could be serializable, this provides great amount of flexibility to be able to deploy actors in different docker instance or different node altogether.

Akka framework by default supports many of the required features to enable this kind of a clustered JVM nodes, in case if there is any specific code need to be written for clustering the system that needs to be taken care of.

The only single point of dependency is in the Config Loader and even that could possibly be deployed in all JVM instances and be made in a manner that the config is loaded from database such that on-demand all the config loader instances of all the JVMs could read the database and load any new configuration into static memory post which they don't need to go to database.

For Prototype: For prototype purposes the it is assumed that the Flow Orchestrator component and all the component actors will be deployed in the same JVM instance and coding is done accordingly.

PHYSICAL/DEPLOYMENT VIEW


Physical Components
JVM Cluster Node
Actor Instances
JVM Cluster Node
Actor Instances
JVM Cluster Node
Actor Instances