**HIGH LEVEL FLOW**

**Requirement:** Source table is constantly populated with task provisioning requests. A request in the provision task table might be generated to execute a given flow. For example, request a in the provisioning table can be generated to create an LDAP request and an other task can be generated to create an active directory account. Once a necessary task has been fulfilled, the status of the execution should be updated back in the task provisioning table.

**Solution:** The proposed solution is to use spring integration framework to work end to end. Spring Integration provides an extension of the Spring programming model to support the well-known Enterprise Integration Patterns. It enables lightweight messaging within Spring-based applications and supports integration with external systems via declarative adapters. Those adapters provide a higher-level of abstraction over Spring's support for remoting, messaging, and scheduling. Spring Integration's primary goal is to provide a simple model for building enterprise integration solutions while maintaining the separation of concerns that is essential for producing maintainable, testable code.

Other alternatives like opensource enterpise bus solutions were also considered but they add extra complexity and cost factor to the solution.

**Approach:** The main job is to listen to the source database and constantly query the table for a set of records. Once the records are retrieved, they are marked as IN\_PROGRESS so subsequent reads would not fetch the same records. This can be done by setting up an inbound adapter and passing the retrieved messages to an initial chain. This chain consists of message splitter to split the messages based on the metadata (connector\_id) and a filter for filtering out un-implemented channels . The splitted data is then passed to next chain. This chain consists of an header enricher, content transformer and a router which routes the messages to appropriate channels.

Notice we have the below channels:

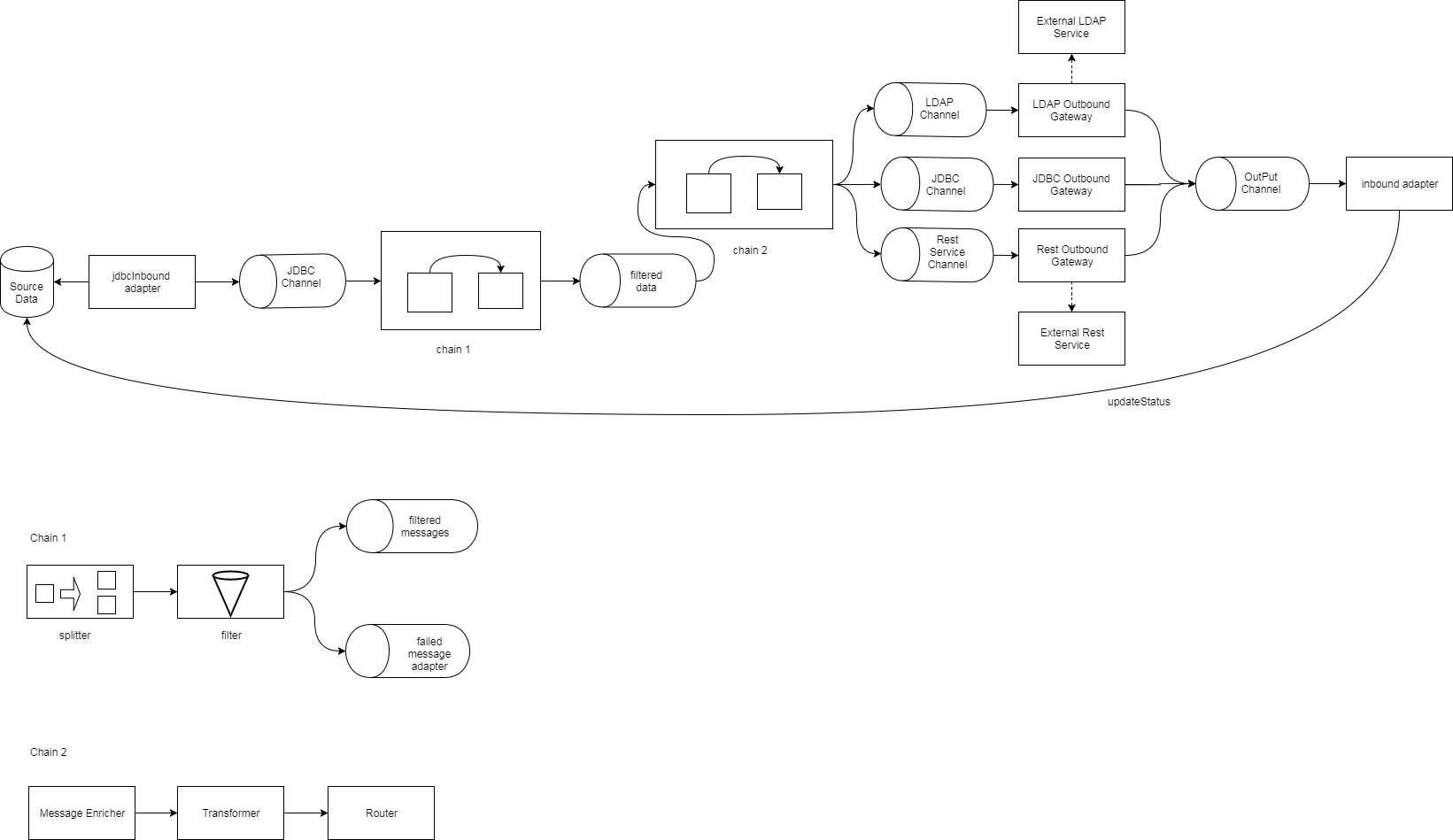
* LDAP
* WebArch Identity
* Database
* MQ
* Rest Connector

Each router would have an associated aggregator to aggregate similar messages and route them to external outbound gateways for external processing.

This can be summarized as below:

JDBC adapter >> Splitter >> Filter >> Transformer >> Router >> Transformer >> Aggregator >> outbound gateways

**DIAGRAM**

****

**Low Level**

**JDBC INBOUND ADAPTER:**

The inbound JDBC adapter's purpose is to retrieve entries from a database. It can also call an update query to mark those entries that had been retrieved. Once the entries had been retrieved, Spring Integration will wrap the entries as a list of Message objects.

Spring Configuration snippet:

<jdbc:inbound-channel-adapter id="jdbcInbound"

  query="select \* from provision\_task where status = 0"

  channel="jdbcChannel" data-source="dataSource"

  update="update provision\_task set status=1 where id in (:id)">

  <poller fixed-rate="5000">

   <transactional />

  </poller>

</jdbc:inbound-channel-adapter>

This inbound channel simply queries the database for every pollerfixed rate.After retrieving the entries the JDBC adapter would execute the update statement that simply changes the status to 1or someother meaningful value so the same results would not be deduped.

**The Splitter:**

The Splitter's purpose is to split a list of messages returned by the JDBC adapter. We need to split the messages so that we can process each message individually. This is required as each message might corresponds to a particular external connector type.

<chain input-channel="jdbcChannel" output-channel="filteredChannel">

  <splitter ref="splitterBean"/>

  ...

</chain>

<beans:bean id="splitterBean" class="package.MessageSplitter" />

Class details: MessageSplitter

Method: splitMessages(Message<?>)

**The Filter:**

The Filter's purpose is to sift out invalid messages. These are messages that have invalid keywords. Remember valid connector\_id's are LDAP, Webarch identity, Database, MQ and Rest Connector. Valid messages will be sent to the next flow. Invalid ones will be redirected to a discard channel with a header property valid = false.The invalid once will be logged in the log files and pushed to a seperate discarded channel so they are written back to the database with a invalid status code.This status code can be used to query these invalid records and necessary action can be taken by either manually fixing them or raising alarms with the source team that produces this data once a threshold is breached.

<chain input-channel="jdbcChannel" output-channel="filteredChannel">

  ...

  <filter ref="filterBean" method="filter"  discard-channel="rejectedMessagesChannel" />

</chain>

<beans:bean id="filterBean" class="package.ProductFilter" />

Class details: CategoryFilter

Method: filterLDAPRequests(Map<?,?>)

filterRestRequests(Map<?,?>)

filterDatabaseRequests(Map<?,?>)

filterMQRequests(Map<?,?>)

**Transformer:**

The Transformer's purpose is to retrieve the value of "content" key from the Message object. The content value is then converted to an object array. For this application we have two types of transformer: a *ContentTransformer* and *Mapper*. We'll discuss first the *ContentTransformer*

<chain input-channel="filteredChannel">

  ...

  <transformer ref="contentTransformerBean" method="transform"/>

  ...

</chain>

<beans:bean id="contentTransformerBean" class="package.ContentTransformer" />

Class: Transformer

Method: transform(Message<?>)

Factory classes:

Class :LDAPTransformer

Method: LDAPRequest transform(Message<?>)

Class: RestTransformer

Method: HttpRequest transform(Message<?>)

**Router:**

The Router's purpose is to redirect each Object array to a specific channel. Redirection is based on the connector keyword found on the Object array. For example, a Object containing the LDAP connector will be sent to the LDPChannel.

<chain input-channel="filteredChannel">

  ...

  <router ref="productRouterBean" method="route" />

</chain>

<beans:bean id="productRouterBean" class="package.ProductRouter" />

Class: UniversalRouter

Method: route(Object obj)

Factory Pattern can be used here.

**OUTBOUND GATEWAY:**

Each external service will have a specific outbound gateway . URI will be configured in this outbound gateway for rest service, ldap apis and database endpoints will be configured to database requests. All of these gateways would have the same reply chanel to write back. The reply channel would have an inbound adapter to process the message and write it back to source database.

Database outbound gateway

<int-jdbc:outbound-gateway data-source=*"datasource"*

update=*""*

query=*"insert into partner(column1, column2...) values(:column1, :column2...)"*

reply-channel=*"updateProvisionTaskChannel"*

max-rows-per-poll=*"100"*>

</int-jdbc:outbound-gateway>

REST SERVICE OUTBOUND GATEWAY

<ws:outbound-gateway id=*"restServiceGateway"*

request-channel=*"wsChannel"* uri=*"http://restService"*

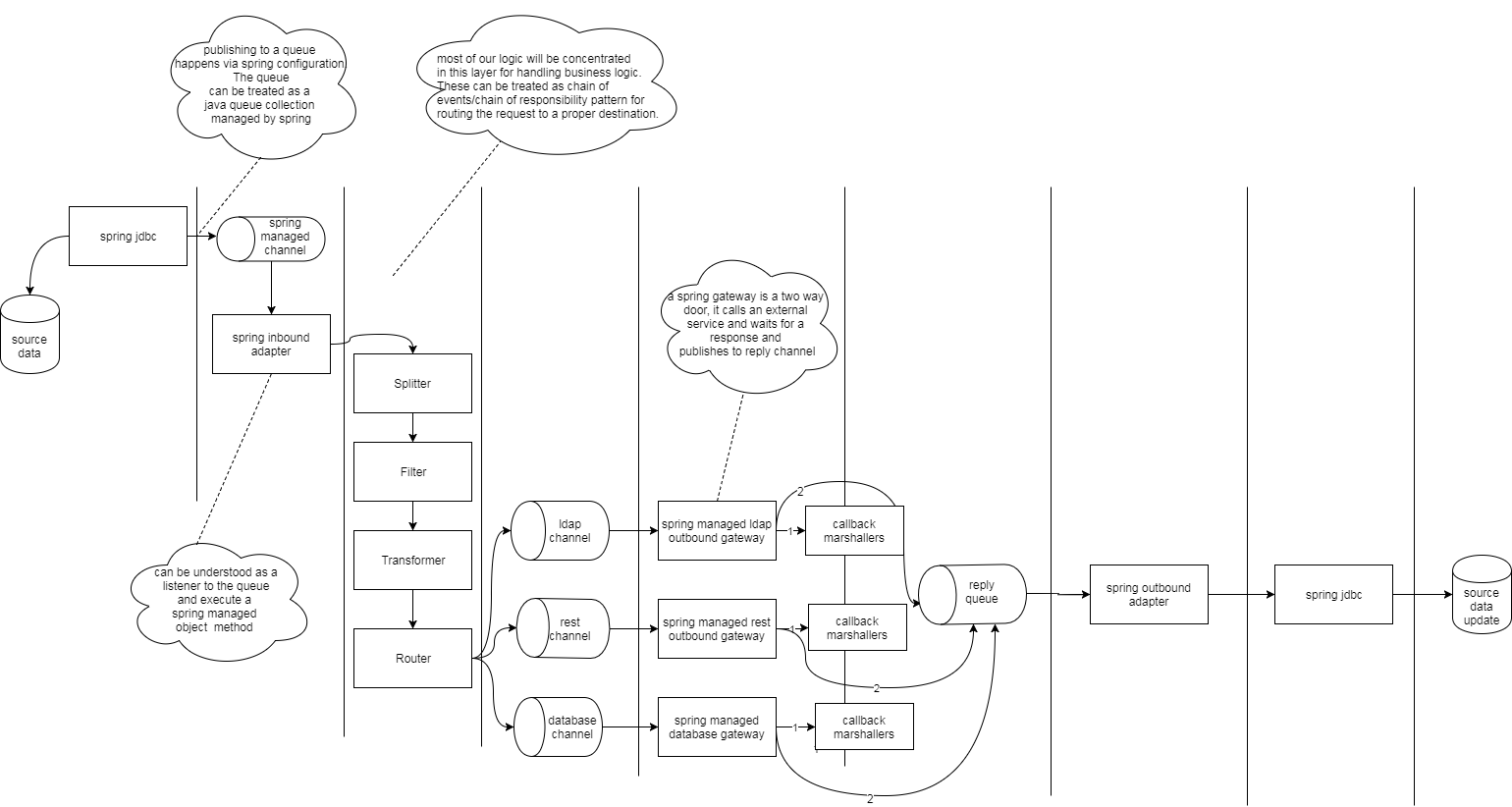
marshaller=*"customMarshaller"* unmarshaller=*"customMarshaller"*

reply-channel=*"updateProvisionTaskChannel"* />

Other miscellaneous classes:

**DynamicURICreator**: Used to create URIs by understanding the CRUD request and call a specific endPoint of a service.

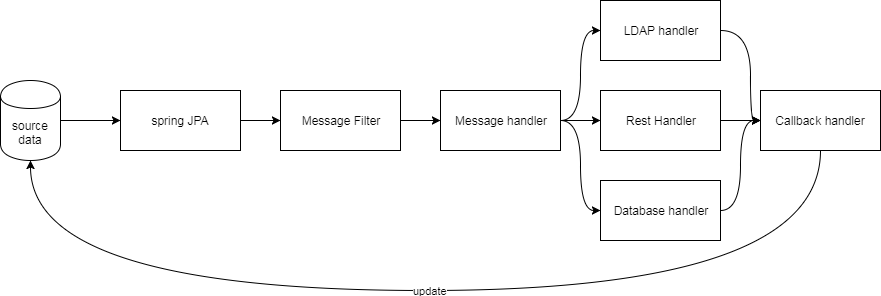
Flow Diagram



This diagram details the layers needed in the architecture to fulfill a given request. Though it looks many, most of the heavy lifting is done by spring integration. There are few customizations needed for processing our business logic and routing to an appropriate channel. To sum up this approach, it heavily relies on spring integration components .

**Approach 2**

The above approach heavily relies on spring integration architecture . The recommendation is to use approach 1 . I would also like to cover an alternate approach which uses springs during the data fetching and remote call execution and uses custom logic internally to manage the data flow.



This heavily relies on Java 8 concurency package Executor service and runnables. A runnable would be created for each request that calls the filter and handlers. The response is collected in a completable future object and will be updated back to DB. The good thing about Java 8 completable future is the ability to handle both success and failure callbacks.