Revision History

Version	Release Date	Author	Nature of Change
0.1	TBD	Verio Inc.	Initial Draft

Table of Contents

List of Figures

1 Introduction (Review)

1.1 Overview

VerioContact involves development of an enterprise application providing branded outbound communication services to Verio software applications. VerioContact will offer the ability to apply customized styling, branding, define message template, layouts, general administration & statistic reporting for such communications while providing an asynchronous interface to Verio applications for receiving communication trigger request and providing ability of check status on the same. Initial focus of VerioContact is transactional communication (one-to-one communication, e.g. account creation, monthly statement etc.)

1.2 Scope

Scope of this document is to discuss technical solutions and layout detailed design consideration for the VerioContact Message Instantiation System. This document also discusses integration points of VerioContact with consumer applications and VMP portal framework.

PS: The design of portal framework components and VerioContact GUI will be covered separately in respective design documents.

1.3 Requirements

1.3.1 Functional Requirements:

Please refer VerioContact functional specification document for detailed requirements.

1.3.2 Non – Functional Requirements:

1.3.2.1 Reliability

This feature of the VerioContact system will ensure that the system will work correctly, without being aborted.

1.3.2.2 Availability

This requirement of the system will allow the system to be available all the time. This would be achieved by deploying the application in a clustered environment so that I event of failure of a node other nodes take over.

1.3.2.3 Usability

VerioContact Message Instantiation System will allow its usage through the invocation of REST Based API's through HTTP(s). The data will be returned in a human readable XML/JSON format, which can easily be consumed by a third party application. The service specification document of the REST based API's will make it very easy for a developer/business analyst to understand the functionalities provided by the System.

1.3.2.4 Reusability

VerioContact will make use of the existing market place components like VMP portal framework and other frameworks built for logging, exception handling etc.

1.3.2.5 System maintainability

The deployment of the system in the clustered environment will provide ease maintenance of the system, since at any point of the time, any of the nodes can be taken down for maintenance while the keeping the service/ functionalities of VerioContact functioning.

1.4 Document Reference

Following is the list of documents which are being referenced in order to create this document.

- VerioContact functional specification
- VerioContact architecture document
- VerioContact physical data model
- VerioContact Service Design
- ?

1.5 Definitions, Acronyms & Abbreviations

Term	Definition
GWT	Google web toolkit
GXT	Ext-GWT (extension of GWT)
ORM	Object Relation Mapping
GUI	Graphic user interface
UI	User interface
SOA	Service oriented architecture
XML	Extensible mark-up language
JSON	Java script Object notation
GUI	Graphical User interface
CSS	Cascading style sheets
OEM	Original equipment manufacturer
MVP	Model view presenter
MVC	Model view controller
REST	Representational state transfer
AJAX	Asynchronous JavaScript and xml
HTTP	Hypertext transfer protocol
DAO	Data access object
JPA	Java persistence API
JAXRS	Java API for RESTful Web Services
JAXB	Java API for XML Binding
HTML	Hypertext mark-up language
JDBC	Java database connectivity
SSO	Single sign-on
MIME	Multipurpose Internet Mail Extensions

Table 1: Abbreviations

1.6 Assumptions, Constraints & Dependencies

1.6.1 Assumptions

VerioContact design is based on following assumptions:

- Consumer application is aware of “Message Templates IDs” defined in VerioContact along with the tokens specified in these templates.

- If for a requested communication, the specified “Message Template ID” does not exist in portal framework, the communication will fail.
- Consumer application is aware of the “Account ID” that needs to be used for the communication being requested. Profile to be used for the communication will be the profile associated with the given account id.
- If for a communication, the specified “Account ID” does not exist in VMP framework, the communication will fail.
- If requested “Message Template” does not exist for the Profile associated with the Account ID specified by consumer application, “Message Template” from default (white labelled) Profile ID will be used.
- “Content Branding” in branding repository must exists for the Profile associated with the Account ID specified by consumer application. It’ll be responsibility of branding editor to ensure that there are no branding gaps. Test Emails feature can be used to ensure everything being in-place.
- Full branding is needed in Content Repository. It’ll be responsibility of branding editor to ensure that there are no branding gaps.
- If all the tokens are not replaced in a message after message assembly (missing token values case), the message will be rejected and land in error state.

1.6.2 Constraints

- VerioContact will support one “message transmission” per “message instantiation request”. So if an Email is to be sent to ten different recipients such that none of these ten should see others name in “To” (or “CC” list) in the Email, consumer application will need to send ten different message instantiation requests to VerioContact. If a message has recipient-specific token values, then separate requests must be sent by consumer app for each separate recipient.
- Similarly for SMS, the requested communication will need to have only one recipient (mobile number). Requests having multiple recipients will be discarded and will land in error queue.
- Assembled messages will not be stored in VerioContact database. So any resend operation will result is reassembling of the requested message.

1.6.3 Dependencies

- Email gateway (SendMail)
  Email gateway, to be provided by Verio. VerioContact “Email Sender” will use the gateway to send out communication.
- SMS gateway
  SMS gateway, to be provided by Verio. VerioContact “SMS Sender” will use the gateway to send out communication.
- Branding Repository
  This branding repository is being setup as part of VMP portal framework. Same infrastructure, design & UI will be used in VerioContact.
- Authentication Server
  WSO2 identity server setup as part of VMP will be used for VerioContact GUI authentication.
- Bounce Tracking

Implementation of bounce tracking will depend upon ability of Email/SMS gateway to provide identification of message that bounced back so that the same can be tied to requested communication from message transmitter.

- Consumer application provided account hierarchy
  Implementation of this in VerioContact will be dependent on the adoption of the same in VMP Framework. VerioContact will accept account hierarchy from consumer application so that same can be consumed once VMP framework has that capability.

2 High Level Design (Review and Revise)

2.1 Design Goals

Following are the design goals of the VerioContact system:-

1. **Standard service contract:** The system will provide a standard service contract which briefly describes the kind of service being provided and the various input/output parameters being accepted and return by the service in which format. Thus a great emphasis has is placed on the specific aspects of service design which includes defining the various data structures, data model their types and the various constraints that are being imposed on the request types.
2. **Service loose coupling:** VerioContact system as is based on SOA and JMS, so it promotes the loose coupling between various components by independent design and implementation of service logic. So individual components will be able to work independent of each other.
3. **Service abstraction:** VerioContact system will provide a layer of abstraction over the way the various services have been implemented. It will hide as much as the underlying details of the service as possible. Thus the technical implementation of the various web services will not be an impediment for the client side implementation of the services. Thus, the invocation of the services will be totally agnostic of the way the services are being implemented. Moreover various internal components like “message assembler” and “message transmitter” will work independent of each other.
4. **Service statelessness:** VerioContact system will provide the services without keeping any state maintained between the two different requests. As it is based on SOA approach, it helps to make the system available most of the time (by decreasing the set of resources required), thus help towards a scalable solution.

2.2 Component Design – Message Instantiation System (Review, needs small updates)

Verio Contact Message Instantiation System will provide RESTful webservice interface to consumer application to request a communication to be sent out. Consumer applications will invoke the appropriate webservice, providing all the details needed for it. The service will queue up the request in a JMS queue to be processed asynchronously, and return transmission id to consumer application to use in checking status check later on.

A restful status service will be provided where consumer application can check the status of the requested communication providing transmission id received as response to communication request.

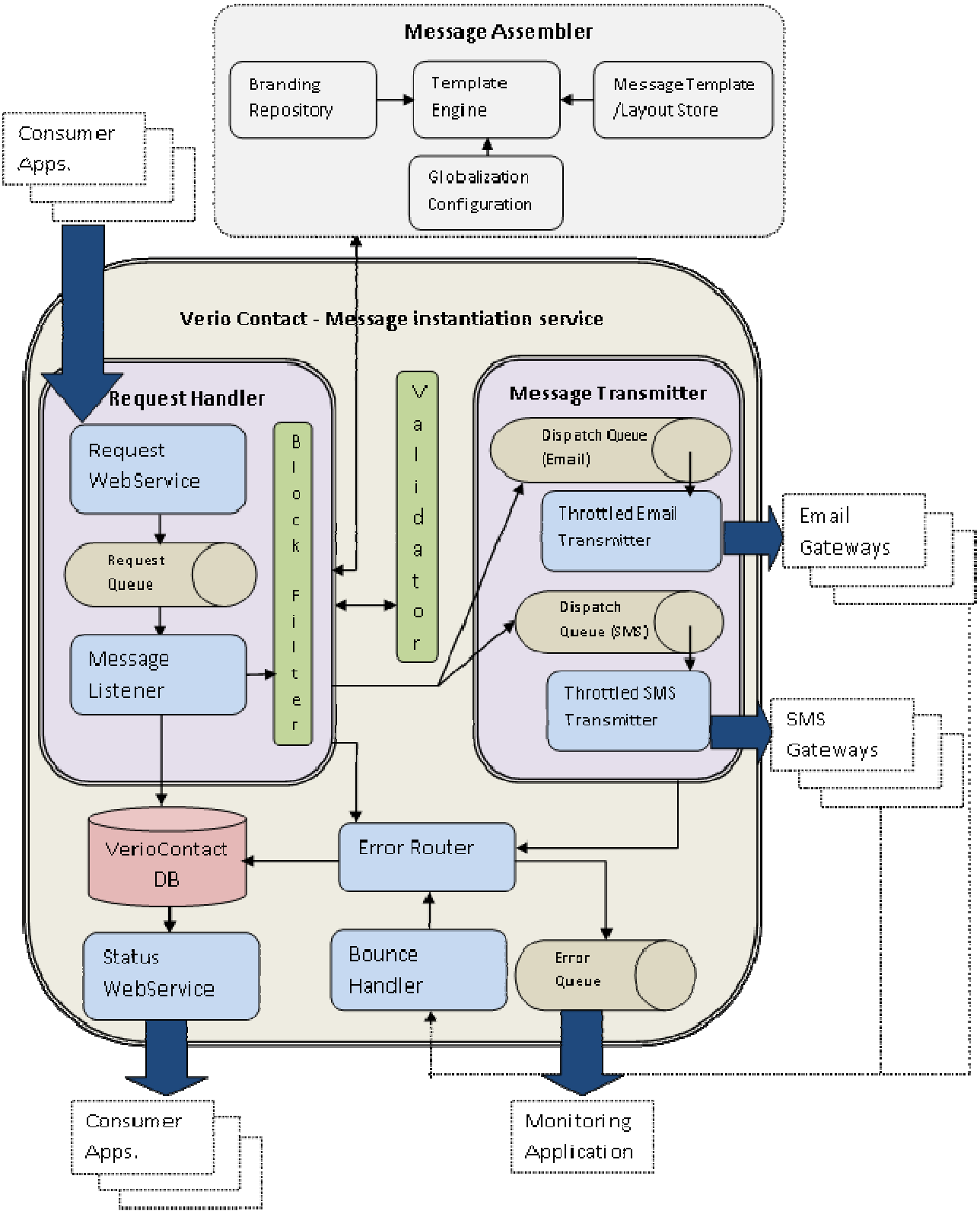


Figure 1: Verio Contact Message Instantiation System

2.2.1 Message instantiation Request Handler

Message instantiation Request Handler provides a JMS based interface for “Consumer Applications” to request a communication to be sent out.

2.2.1.1 Request Queue

Request Queue is going to be Apache ActiveMQ based persistent JMS queue. Consumer applications requesting messages dispatch will publish JMS message to this queue and receive MessageID as response.

2.2.1.2 Message Listener

Message listener will receive the messages from Request Queue make the entry for the same in database and pass it on to message assembler for assembly.

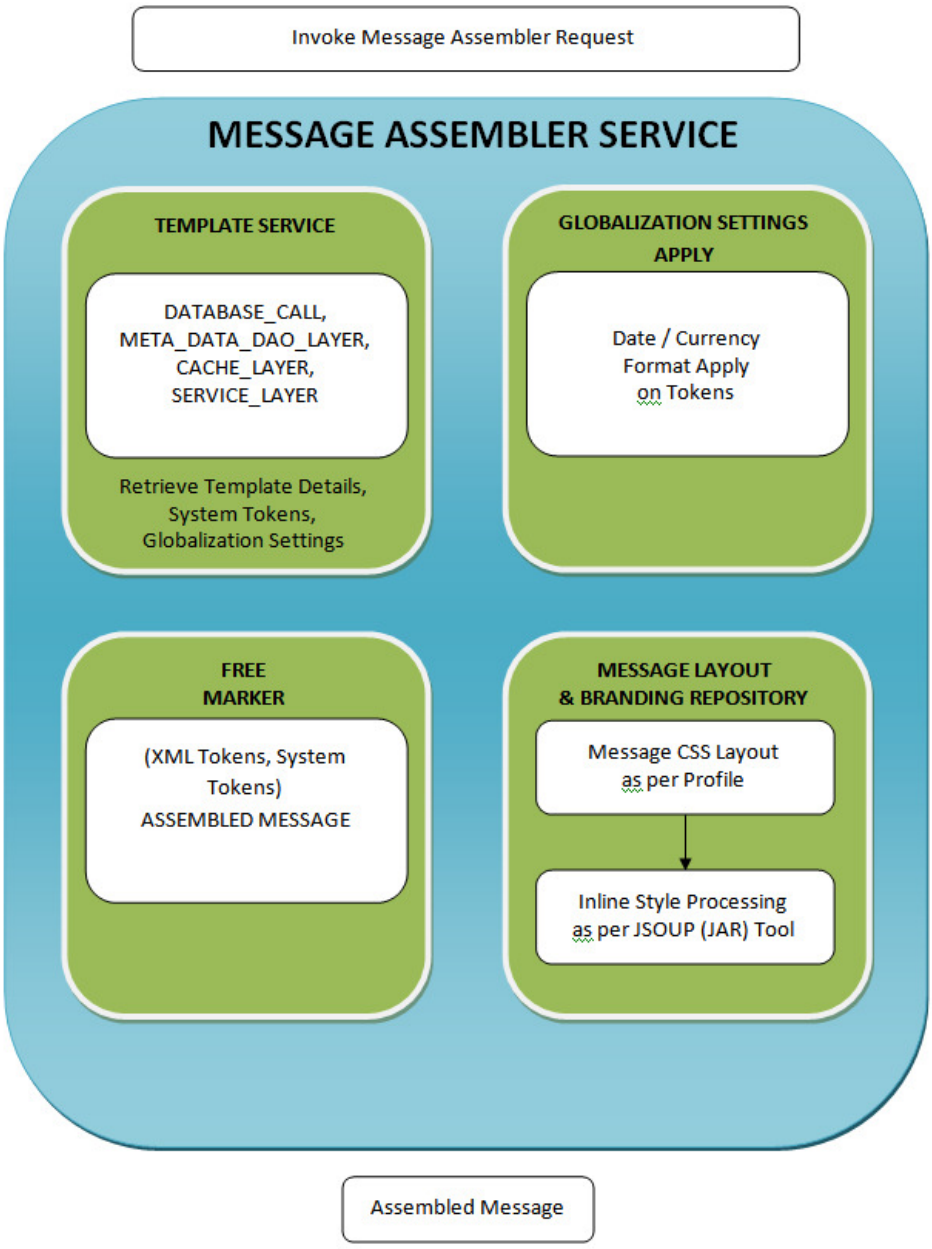
2.2.2 Message Assembler

Message Assembler will invoke XML request object. Message Assembly is divided into mainly following parts:

1. Message Template Service
2. Apply Globalization Service
3. Message Assembly
4. Message Layout & Branding Repository

The data received in a human readable XML/JSON Object format, which can easily be processed as per provided details. For Sample Example:

```
<?xml version="1.0" encoding="utf-8" ?>
<!-- Message from VA Router to OSSB Integration Layer -->
- <ContactRequest ossId="OssVerioBoca" transactionId="12341231234">
  <!-- Information defining the Notification -->
  - <ContactRequestData>
    <TemplateId>OfferingDNS</TemplateId>
    <ProfileId>VERIOProfile</ProfileId>
    <MessageFormat>HTML</MessageFormat>
    <RegionId>en-US</RegionId>
    <ContactFirstName>Bob</ContactFirstName>
    <ContactLastName>Jones</ContactLastName>
   - <NotificationAddresses>
      <NotificationAddress type="email">bob@jones.com</NotificationAddress>
      <NotificationAddress type="email">mary@johnson.com</NotificationAddress>
      <NotificationAddress type="phone">1-ABC-DEF-WXYZ</NotificationAddress>
    </NotificationAddresses>
  </ContactRequestData>
  <!-- All below information -->
  <!-- Account Information -->
  - <AccountData>
    <AccountId>X204394593</AccountId>
    <AccountHierarchy>VerioRoot,OCN,VerioRetail,X923958</AccountHierarchy>
    <!-- ==== Account Details ==== -->
    <!-- Universal unique Id associated with this account -->
    <AccountUuid>XXXXXXX</AccountUuid>
    <!-- country code -->
    <CountryCode>XXXXXXX</CountryCode>
    <!-- Channel id -->
    <ChannelId>XXXXXXX</ChannelId>
```



2.2.2.1 Message Template Service

This service will fetch template details, System tokens, globalization settings by elements provided in XML object request.

2.2.2.2 Apply Globalization Service

This service will apply globalization setting on tokens like date format and currency format as per locale.

2.2.2.3 Message Assembly

Message composition will be done here as per fetched template details with the help of free marker.

```
${doc.Header}

X-AccountID: ${doc.ContactRequest.AccountData.AccountId}
The requested modification to the login of an account has been processed.
Both the current as well as the previous login information are included below for your reference.

CHANGE LOGIN REQUEST SUMMARY

Account: ${doc.AccountData.FirstName} ${doc.AccountData.MiddleName} ${doc.AccountData.LastName}
Previous Login: ${doc.previousLogin}
Current Login: ${doc.currentLogin}

Warm regards,

${doc.BrandName}
${doc.BrandEmail}
```

2.2.2.4 Message Layout & Branding Repository

This will process CSS and images as per given profile. This is used to fetch CSS for inline style applying. At last actual data processing will be here as per message layout with the help of tools like TinyMCE, inline the CSS from INTERNAL CSS / EXTERNAL CSS resource, etc.

To: shalabhd@interrait.com
From: no-reply@att.com (ATT Customer Backroom)
Bcc: manoj@interrait.com (Not sure how this will be handled)
Subject: [ATT Customer Backroom] Login Modification Processed

***** HEADER WITH LOGO AND BACKGROUND IMAGE *****

X-AccountID: X204394593
The requested modification to the login of an account has been processed.
Both the current as well as the previous login information are included below for your reference.

CHANGE LOGIN REQUEST SUMMARY
Account: Shalabh Dixit
Previous Login: shalabh.dixit
Current Login: shalabhd

Warm regards,

ATT Customer Backroom
no-reply@att.com

And then assembled message will be dispatched to validator and Transmitter.

2.2.3 Message Validator

Assembled messages received from assembler will be validated by Message validator for:

- No token placeholder is left in the assembled messages.

In case any of the validation fails, message will be put into Verio Contact "Error Queue" with appropriate error code and description via "Error Router".

2.2.4 Message (Email/SMS) Transmitter

Responsibility of Message Transmitter is to receive requests for Email/SMS dispatch and listen to "Dispatch Queues" and forward these to appropriate Email/SMS gateway.
Message transmitter will make use of the following configuration:

- Profile to Email/SMS gateway mapping
- Throttling rate –System level for Email/SMS transmission

2.2.4.1 Dispatch Queue – Email/SMS

JMS queue to receive request for outbound Email /sms communication from Verio Contact request handler. Access to the queue will be restricted to Verio Contact system only.

2.2.4.2 Throttled Transmitter – Email/SMS

Email transmitter component will be responsible for forwarding the Email messages to Email gateway/SMTP server.

Throttling capability will be built in so that Verio Contact is able to send no more than specified limit of message in defined time interval. Throttling limit will be set up at system level and not at the gateway level.

Administrator should also be able to switch OFF the transmitter so that messages are just queued up when need be and are sent out at a later point of time by switching it back ON.

2.2.5 Bounce Handler

Any bounced communication will land up in the bounce Email box. Bounce handler will poll the Email box and route it to "Error Router".

2.2.5.1 Bounce Emails

To take care of Bounce Emails, we will specify a bounce Email address that will be monitored by BounceHandler for any bounce back. Bounce handler feature will be dependent on capability of Email gateway to provide identification number of the message for tracking and needs to be verified. Such capabilities could vary across gateways & could affect the behaviour of this feature.

2.2.5.2 Bounce SMSes

SMS Bounce handling feature needs to be evaluated based on the specific SMS gateway we develop against.

2.2.6 Error Router

Any message that falls through message validator or is not sent for any other reasons will be passed onto "Error Router" for logging the same in "Error Queue". Error Router will make use of logging framework.

2.2.6.1 Error Queue

Any communication that:

- Fails “Message Validation”
- Fails in sending via “Message Transmitter”
- Bounces back
- Results in system errors (Handled)
  Same will be logged into another persistent JMS queue “Error Queue” via Error Router. A monitoring system can be built to keep track of such errors. Any such error will be logged in system and will be available via status service as well.

2.2.7 Status Service

A restful web service, which will help “Consumer Applications” retrieve the status of any communication requested providing “Transmission ID” received as acknowledgement during request, as input. Consumer applications would need to specifically integrate with Verio Contact for this to work.

2.3 Software (to be updated)

Software to be used to build the system:

- Java 6+
- Spring 3.x (Core, MVC, Web Services, ORM, AOP)
- Hibernate Core 3.5.x
- Oracle 11g
- Oracle JDBC driver
- Jackson library 1.9.1 (For JSON)
- JAXB 2.x (for parsing & creating xml data structure)
- JBoss 7.x Application Server
- Logback 1.x with SLF4j (for logging)
- Junit 4.x (Unit testing framework)
- DBUnit 2.x (for unit testing of DB related code)

2.4 Architectural Components (To be revised)

The Architecture of the Customer management system is being broken up into a number of sub-modules to achieve the modularity. The following diagram shows the various sub-modules.

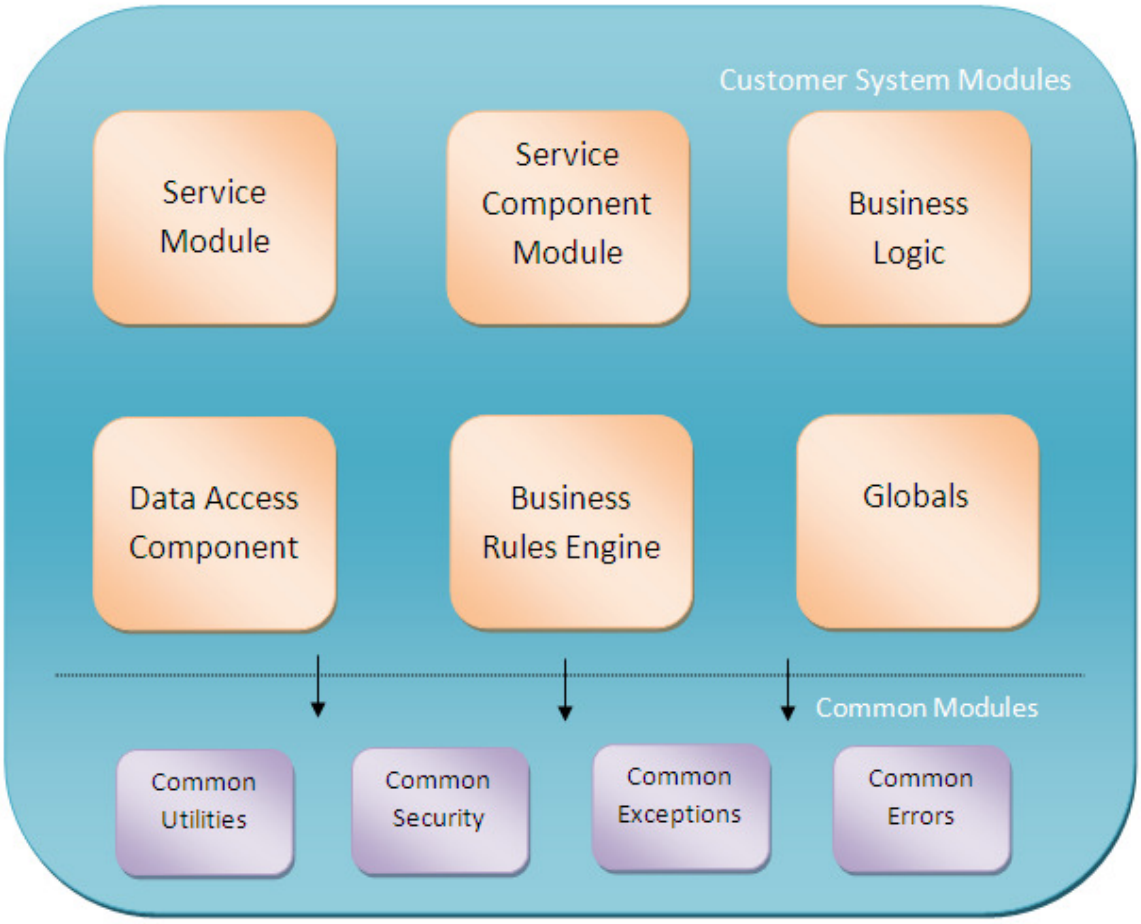


Figure 2: Customer Management System Architectural Components

2.4.1 Service Layer

This layer is the topmost layer in the High level view of the architecture and consists of all the services defined within the SOA. A service is an abstract specification of a collection of (one or more) entities which are related to the customer mgmt system.
All of the services will be exposed via REST based Http/s API, with the URI method itself depicting the purpose of the invocation (as it can be data retrieval request or new entity creation request).
Exposed services reside in this layer; they can be discovered and invoked or possibly choreographed to achieve a specific functionality. Services are functions that are accessible through well-defined interfaces of the services layer. The service layer also takes enterprise-scale components, business-unit-specific components, and system-specific components and externalizes a subset of their interfaces in the form of service descriptions. Thus, the components provide services through their interfaces.

This layer contains the contracts that bind the provider and consumer. Services are offered by service providers and are consumed by service consumers. The services offered will be exposed over HTTP via REST based API's with the response format of JSON/XML (XML will be the default option).

Note: Please refer to the service design specification documents for the details of the individual services exposed by the customer management system

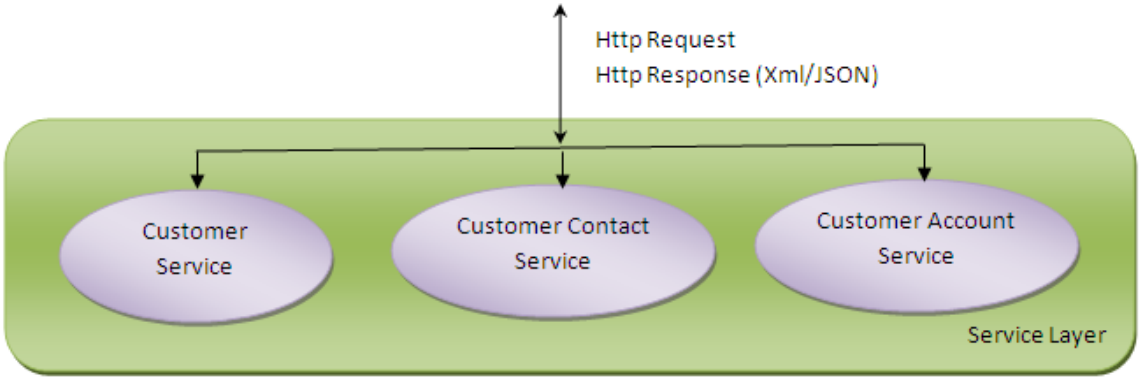


Figure 3: Service Communication - Protocol

As depicted in the above Diagram there will a number of services which exists in the service layer which is responsible for serving all the requests related to a resource. Thus service layer will only be responsible for interacting with the Http Request and Response, and will return the success or error information depending upon the processing of the request.

Each of the services will contain minimum dependency on the Http Servlet API, so as to restrain the functionality as a simple Java bean, so that it can be individually tested without having dependency on the container. All the services will be protected by a security mechanism, so that only authentication/authorized users will be able to invoke the services.

All the services defined for the customer management system will be going to present within the package hierarchy "org.verio.ossb.system.customer.service".

Generally, for most of the services there will be an asynchronous flavour present in the system, which will run the process asynchronously instead of running synchronously. Every call to an asynchronous process service will return job identification (JobId) which will be a unique identification number. The client can invoke another API to check the status of the Job (INCOMPLETE, STARTED, RUNNING, COMPLETED).

2.4.2 Service Component Layer

This layer contains software components, each of which provide the implementation for, realization of, or operation on a service, which is why it's called a service component. Service components reflect the definition of a service, both in its functionality and its quality of service.
The service component layer conforms to service contracts defined in the services layer in terms of various input parameters in the request and output parameters in the response.

Each Service Component:

1. Provides an enforcement point for service realization to ensure quality of service i.e. to return always a human readable response whether there is an error or success from the server.
2. Enables business flexibility by supporting the functional implementation of flexible services as well as their composition and layering.
3. Enables flexibility by strengthening decoupling in the system. Decoupling is achieved by hiding volatile implementation details from consumers by spreading the logic across various business components present in the Operation layer.

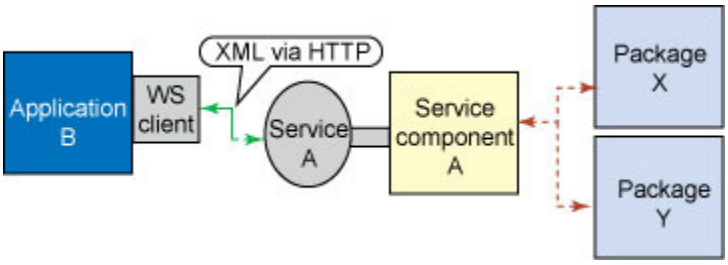


Figure 4: Service Component as Central Piece

So there will be always a direct one-to-one mapping of the functions present in the service layer to the functions present in the Service component layer.

All the services defined for the customer management system will be going to present within the package hierarchy "org.verio.ossb.system.customer.service.component".
As the service provides an async interface for most of the web services, the service component layer will also have the 1-1 mapping of the asynchronous interface. This will make sure that the process will be started in the asynchronous manner and will return the JobID, which can later on be queried for completion. Thus, it provides the option to invoke the same business process either in a synchronous or asynchronous manner. At the implementation level, the function needs to be marked as asynchronous by using the **@Async** annotation. Every invocation of the function will start in a separate thread instead of the main thread and will keep on executing as a separate sub-process within the current JVM process. Following is the code snippet.

```
@Async
  public void deleteUserAccount(int userAccountId) {
        ....................
        ............
        .......
  }
```

In order to enable the functioning of @**Async** annotation, we need to include the following line the spring application context xml file.

```
<task:annotation-driven/>
```

2.4.3 Operation Layer

The operational layer is the most integral part of the Customer management system and will contain the business logic implementation to achieve a particular functionality which may or may not be (READ only) transactional in nature. This layer only deals with the service component layer and the DAO layer (beneath it) in terms of various Value objects that will contain the data related to the layer only.

The Operation layer is composed of two parts.

1. Business components.
2. Rule engine components.

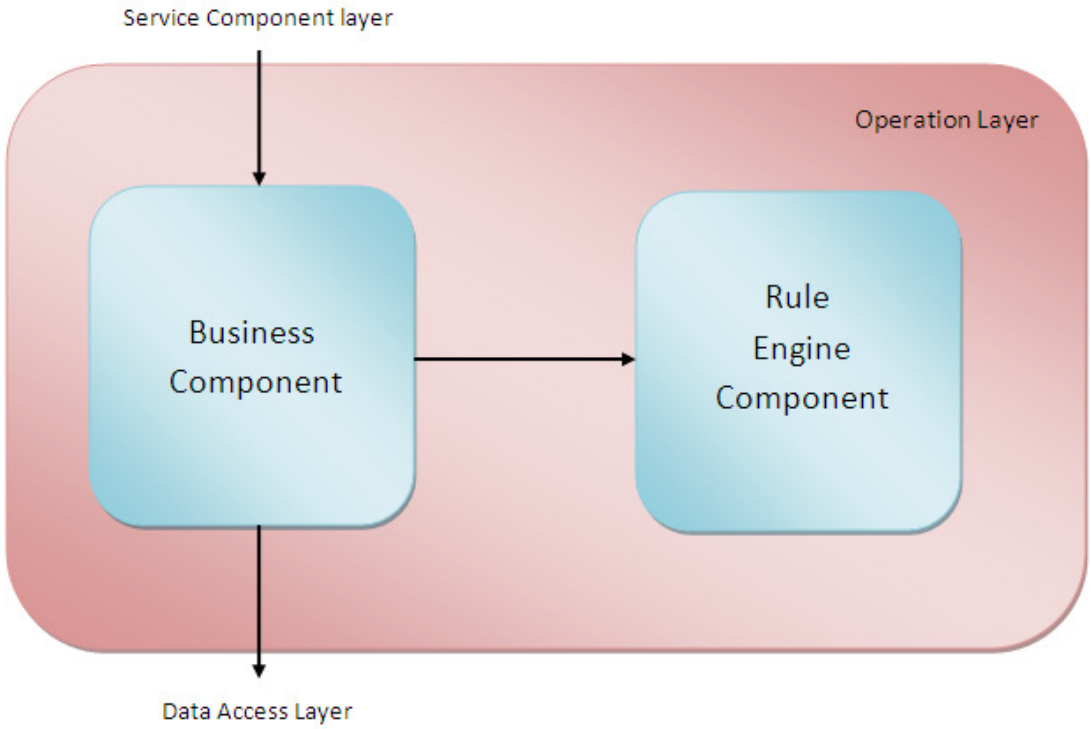


Figure 5: Application integration with Rule Engine

Business Component:

A business component will be responsible for achieving a single unit of business functionality as mapped by the Service Component layer to the service layer. It can be creation of a single entity or returning the data related to single business resource like Customer. All the business components present in this layer will be aligned with the various resources present in the customer management system. Some of the entities/resources are Customer, Customer contact, Customer order etc.

Thus, in order to accomplish a single business use case, this component might make use of the various other business components present in the layer. All the functionalities present in the layer will be transactional (Read only or Read write only) in nature.

Invocation of a single function results in returning the data in the form of various value objects which will be internal to the layer or used by the layer above (Service component layer) or beneath (Data access layer) beneath it. The various business components will make extensive use of

the various Rule Engine components in order to calculate/decide the various intermediate decisions. Execution of a single function may result in an exception which is being handled at the service component layer, so that appropriate response is to be returned to the API consumers.

The results of the business components (read only) may be cached, in order to increase the performance of the system.

All the business components defined for the customer management system will be going to present within the package hierarchy "org.verio.ossb.system.customer.business.component".

Rule engine component:

This component is being responsible for interacting with the rule server to determine the outcome of a specific decisive condition to reach to a conclusion. Thus, Rules component will abstract the other components of the system with all the business rules and provide a simple API based interfaces to know the result of either a single rule or a set of rules applicable to a given business logic.

E.g. while creation of an order related to the customer, it should check the country of the customer and will apply all the rules applicable to identify the tax to be applicable to the order.

All the rules related to the customer will be going to present within the package hierarchy "org.verio.ossb.system.customer.business.rules".

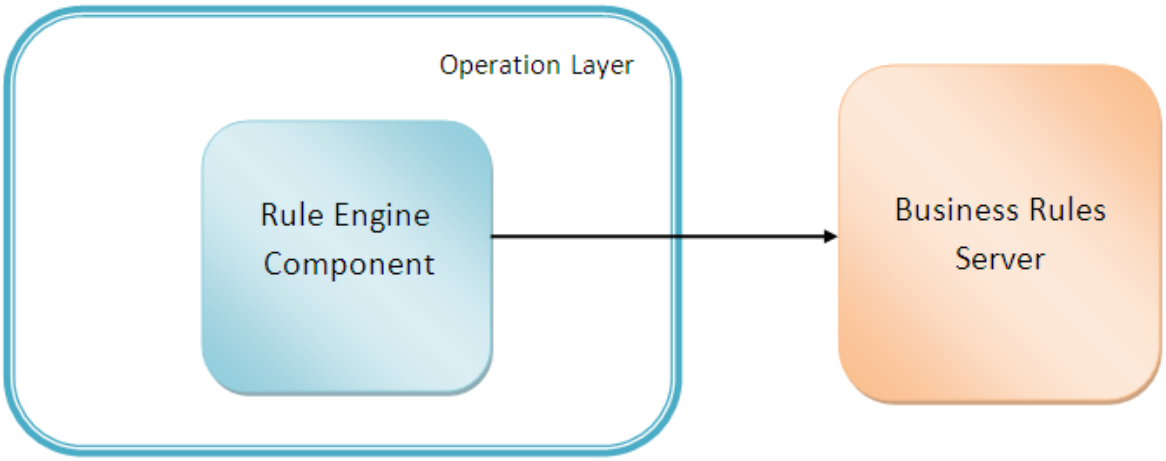


Figure 6: Rule Engine Runtime Architecture

The results of the Rule may or may not be cached for a particular time period, to increase the performance of the system. The rule server will be installed on a different server to that server on which Customer management system is installed. All the rules related to OSSB will be stored on the same server which may or may not be clustered depending on the deployments. The Rule component will make use of the libraries available to invoke the rules to determine the outcome. This will be a synchronous call and the business components will wait for the results before proceeding with the rest of the business logic.

All the business & rule engine components will be represents as **beans** within the spring container by annotating all the concrete classes using **@Component** annotations available within the spring framework.

2.4.4 Database Access Layer

Database access layer in the customer management system will abstract the interaction of the other components of the system with the database, thus provide a unified single view through a set of functions which will expose the data through the a set of domain objects containing the data of the records present in the database. This layer will consist of a number of components with each component responsible for a specific type of entity (table) in the database. For every entity in the database there will be a number of interfaces and concrete classes that will implement the functionality.

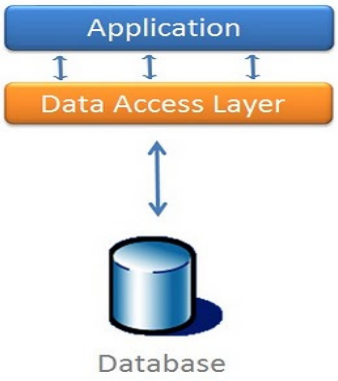


Figure 7: Data Access Layer

All the database related classes are will be going to present within the package hierarchy "_____".
There will be separate sub-packages for different components.
org.verio.ossb.system.customer.db.vo will refer to the domain objects, which are primarily database entities.
org.verio.ossb.system.customer.db.dao will contain the dao interfaces and the concrete implementation classes.

The DAL will be implemented by using Spring/Hibernate will all the concrete classes will be implemented by the **@Repository** annotation to mark them as the database specific classes and will extend the **HibernateDaoSupport** class provided by spring. All the functions will be annotated by the **@Transactional** annotation to mark then each function will participate in the transaction (marked either read only or read write) either existing or a new one.
All the database related connections will be managed by the **BoneCP** connection pooling library with all the parameters being configured through the properties files.

All the results returned by the DAO access layer will be cached into the EHCACHE server, to increase the system performance. This will be achieved by the storing the primary keys of the Entities as the keys of the values to be put in the server. For this, the second level cache will be enabled in the hibernate configuration with the details of the EHCache server. Every object that we want to be cached will be serialized and stored in the form of key value pair on the cache server.

2.4.5 Globals Layer

The Globals layer represents that part of components which will be used across all the sub-modules of the customer mgmt system vertically. Following are the main components:

- **Validators:** All the Validators that will be used throughout the mgmt system will fall in this layers, this include validators for verifying the web request, verify the business logic components for checking of the correct values and the corresponding database model objects. Validators will be implemented by making use of Spring validators (an implementation of JSR-303)
- **Exceptions:** All the common exceptions that may arise from any of the customer sub-modules will fall in this layer.
- **Logging**: Each of the operation logging business process in logback (via Slf4j) implemented logging flat file for future auditing.
- **Securities Framework**: Customer Management System using Single Sign On (SSO) centralized securities fragment that enables a user to be authenticated once, and gain access to resources multiple during that session.
- **Common Utilities**: All the utilities that will be used across the various product sub-modules will form a part of this layer.
- **Common Configuration Files**: All the configuration and properties file have a common module (**OSSBHome**) projects only and using a one single xml file to load the entire configuration.

3 Detailed Implementation Design (Review and Revise as needed)

3.1 Authentication & Authorization

The API Exposed by the VerioContact message instantiation system will be used by Verio applications. A separate user will be created for each these applications having ContactMessageRequestor role. Users and corresponding roles will be managed in Identity Server. Users will this role will have rights to request a fresh communication or check the status of previously requested communication.

ContactMessageRequestor role and users having this role will be managed through the interface provided by the identity server. The VerioContact message instantiation system will validate this role to be assigned for the user invoking the service. For every request the consumer application will pass on public key of the user along with an encrypted signature to authentication.

Please refer VerioContact Service Design document for details on authentication and authorization.

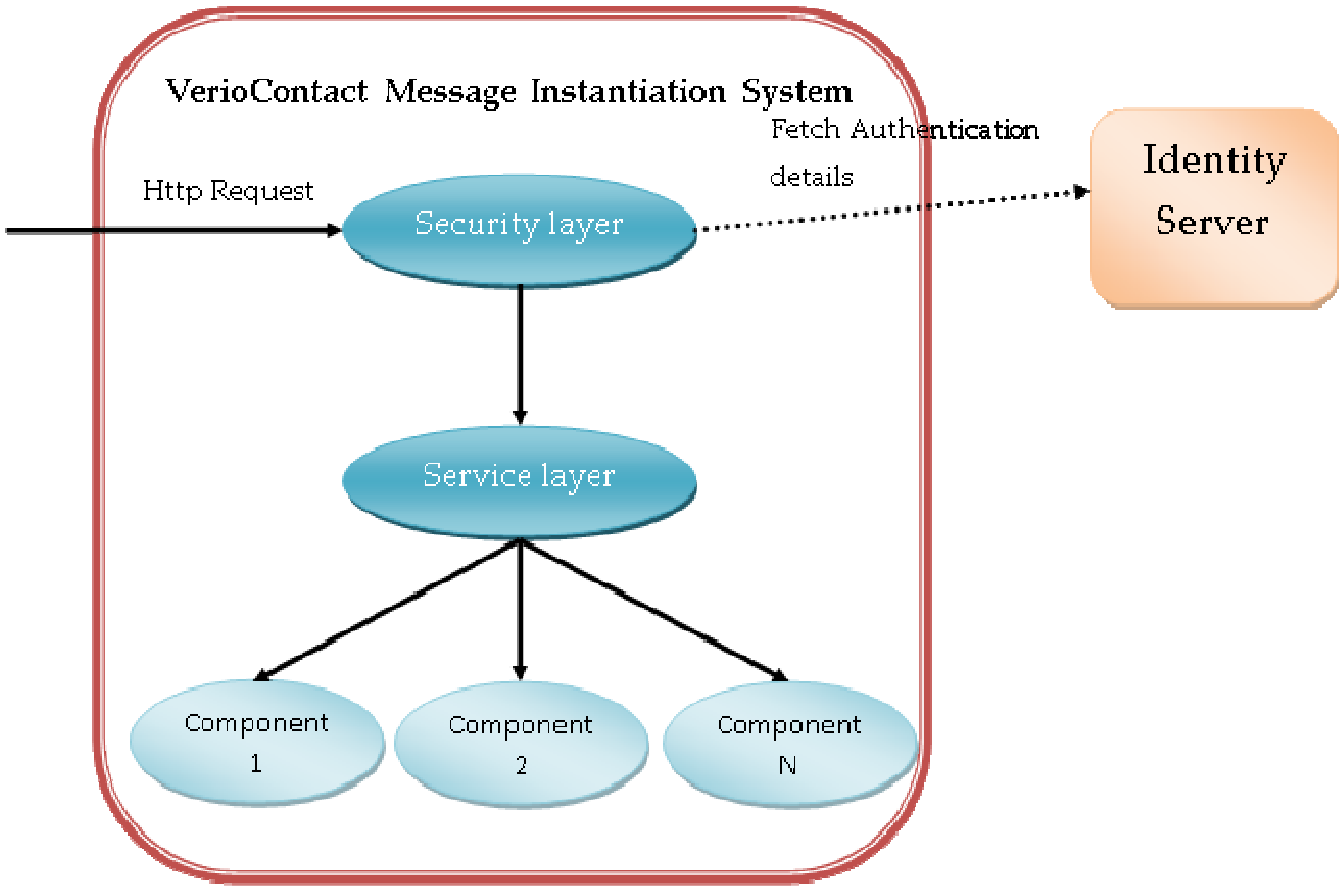


Figure 8: Integration with Security Service

The Identity Server will provide the interface for managing the roles and the users associated with it. This includes their roles mapped with them and their public/private keys. All the http requests appended with the API keys are being validated against the keys as present in the SSO server for the user and obtain the roles associated with that user and will only allow appropriate services to be invoked.

3.2 Service Component Design

3.2.1 Controllers

3.2.1.1 Request Handler

As the Customer mgmt system is based upon SOA approach, it will provide the access to the functionalities in a very loosely coupled manner via Http (i.e. the invocation of one http request from the client is totally separate from the invocation of the another request either by same client or different client, as there is no co-relation between both of them). So, all the business functionalities/use cases will be logically grouped into a single service entity and being exposed over Http. The number of services and the type of entities that are being exposed is being described in detail in the "**Service_Specification_Design.doc**". Each of the service being exposed has the capability to returning the data either in the Json/Xml that will be determined by the choice of the client based upon the headers present in the Http invocation request.

The exposed services will be totally based upon the "**Spring MVC Rest**" and will partially implement the JSR-311. All the services will be basically going to implement the **@Controller** annotation and will specify the URI on which the service needs to be activated.

Before the service is being served by the Service components, it has to pass through a number of bridges e.g. Security check verification, Main central service controller (which will select the service which has to serve the request).

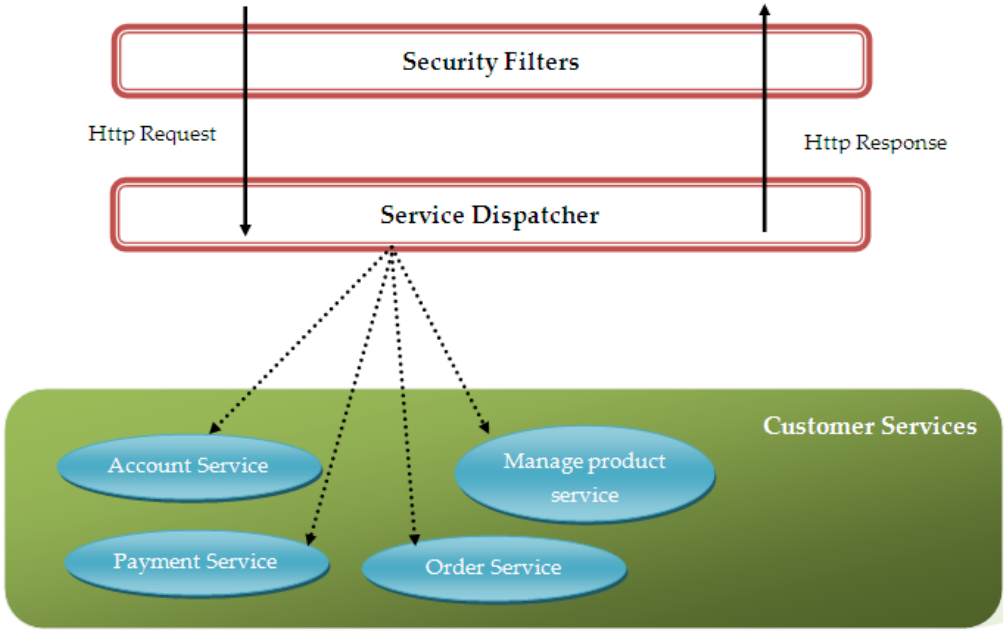


Figure 9: Security Filter implementation

Each of the service shown above will suffice a specific set of business functionalities related to the logical entity. Every invocation of a service will has to validate itself by passing user specific parameters, which can either be the username or the API key only. Once validated, it will be directed towards the exact service (by service dispatcher) which will be responsible for fulfilling the request and generating the response. Thus every service will represent a component that will be spring sub-controller finally.
Following is the sample representation of a service.

```
@Controller
@RequestMapping(value = "/account")
public class AccountService {

        @Autowired
        AccountComponent accCom;

    @RequestMapping(value="/{id}/details")
    public AccountDetailsResponse  getAccountDetails(@PathParam String id)  {

            ..........
            ..........
            ........
    }
}
```

All the security related features will be implemented by making use of Spring Security which will be implemented via a filter (**DelegatingFilterProxy**) and will intercept every http request for the authorization.

3.2.2 Services

3.2.2.1 Message Assembler

As per parts of Message Assembler services is divided into mainly following parts:

1. Request Payload
2. Template Service
3. Globalization Apply
4. Message Composition
5. Branding Repository
6. Message Layout

1. Following is the sample representation of a Request Payload Service.

```
@RequesXMLMapping(value = "/rmemap")
public class RequestXMLService {

        @Autowired
        RequestXML requestXML;

    public AccountDetailsResponse  getAccountDetails(@PathParam String accId)  {
```

```
                    .....
                    .....
                    .....
        }
}
```

2. Following is the sample representation of a Template Service.

```
@TemplateMapping(value = "/template")
 public class TemplateService {

                @Autowired
                Template objTemplate;

    public TemplateDetailsResponse getTemplateDetails(@PathParam String templateId)
{

                    .....
                    .....
                    .....
        }
}
```

3. Following is the sample representation to apply Globalization Settings.

```
@RequestGlobalizationMapping(value = "/globalization")
 public class RequestGlobalization {

                @Autowired
                RequestGlobalSetting requestGlobalSetting;

    public String applyGlobalizationOnDateFormat(@PathParam String dt)  {

                    .....
                    .....
                    .....
        }
}

    public String applyGlobalizationOnCurrencyFormat(@PathParam String curr)  {

                    .....
                    .....
                    .....
        }
}
```

4. Following is the sample representation of a Message Composition Service.

```
@RequestMessageAssembleMapping(value = "/msgcomposition")
public class RequestMessageAssembleService {

        @Autowired
        RequestMessageAssemble requestMessageAssemble;

    public String getAssembledMessage(@PathParam String accId)  {

                .....
                .....
                .....
        }
}
```

5. Following is the sample representation of a Branding Repository Service.

```
@RequestBrandingRepositoryMapping(value = "/brandingrepo")
public class RequestBrandingRepositoryService {

        @Autowired
        RequestBrandingRepository requestBrandingRepository;

    public String applyBranding(@PathParam String accId)  {

                .....
                .....
                .....
        }
}
```

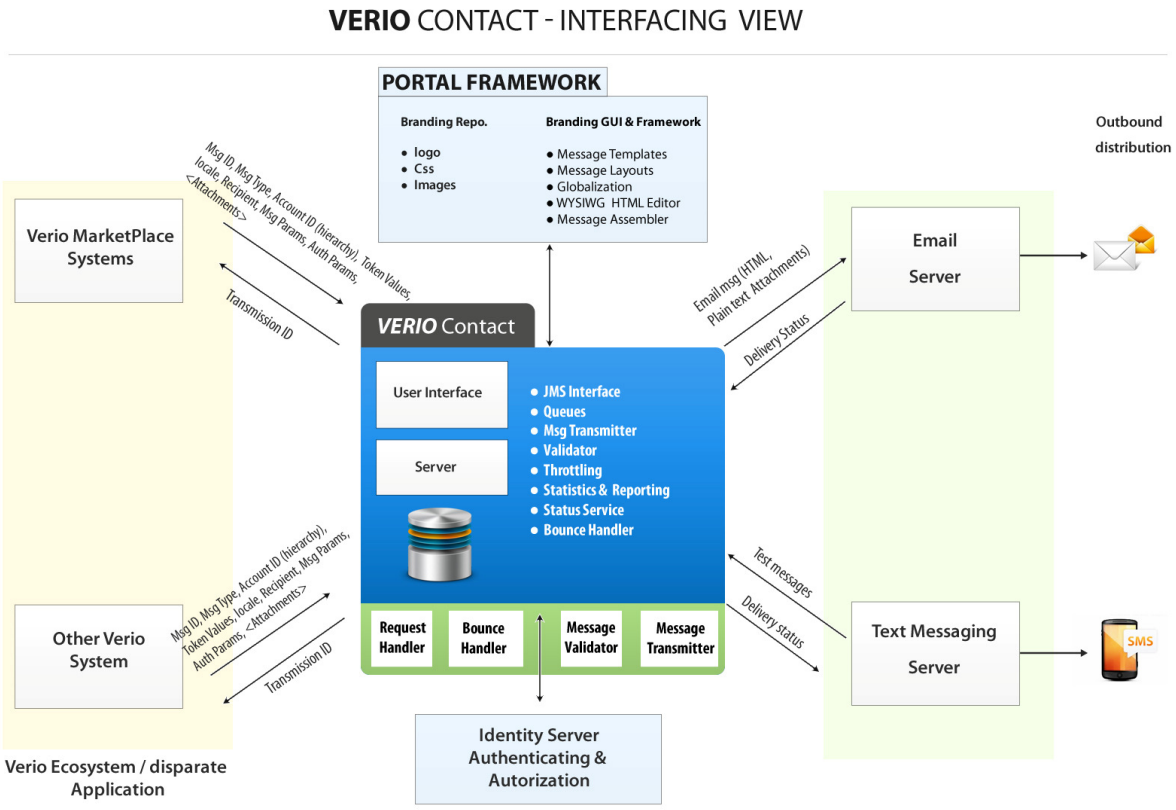
6. Following is the sample representation of a Message Layout Service.

```
@RequestMessageLayoutMapping(value = "/msglayout")
public class RequestMessageLayoutService {

        @Autowired
        RequestMessageLayout requestMessageLayout;

        @RequestMapping(value="/{id}/details")
    public String applyMessageLayout (@PathParam String accId)  {

                .....
                .....
                .....
        }
}
```

3.2.2.2 Message Validator

3.2.2.3 Message Transmitter

3.3 VMP Contact Interfaces



3.4 Data Services Design

3.4.1 Data Caching

Service oriented distributed architecture of customer management system deconstructs the complete system into loosely coupled information services, which may be deployed over a number of systems for reliability purposes. Essential characteristic of this distributed model is flexible, scalable and preferment access to relevant resource data. Thus, it is very much required to provide the response in the minimal time frame as possible.

This has to be achieved by implementing **Caching** at various levels, so that we need not to recalculate the business logic or data access operation again and again for the same set of data. For Caching will be implemented via EhCache module which will operate in the middle tier of the architecture (i.e. at data access layer and at the operation layer).

All of the following components will be wrapped up by the Cache layer in order to efficiently retrieve the data.

1. Data access components
2. Rule engine component
3. Business component.

DAL Caching:

Since our data access layer is being implemented by Hibernate, we will use the second level cache provided by hibernate combined with EhCache to store the data. This will help the other parts since we need not to execute the same query again and again for different request. For this to work, the bundled Customer Management system WAR (Web application archive) file, will take input the cache configuration file (ehcache.xml) as being configured in the OSSB_HOME directory on the file system. In addition, all the Entities (database model value object) source code will be annotated with the @**Cacheable** annotations.

For E.g. the following code snippet shows the change needs to be made to the entity which is being cached.

```
@Entity
@Table("cust_contact")
@Cacheable
@Cache(region = "customerContacts", usage = CacheConcurrencyStrategy.READ_WRITE)
public class CustomerContact implements Serializable  {

}
```

In order to enable the second level cache we must have to change the hibernate settings and should include the following properties in the configuration.

```
<prop key="hibernate.cache.use_second_level_cache">true</prop>
<prop key="hibernate.cache.use_query_cache">true</prop>
<prop
key="hibernate.cache.provider_class">net.sf.ehcache.hibernate.SingletonEhCacheProvider</prop>
<prop key="hibernate.generate_statistics">true</prop>
```

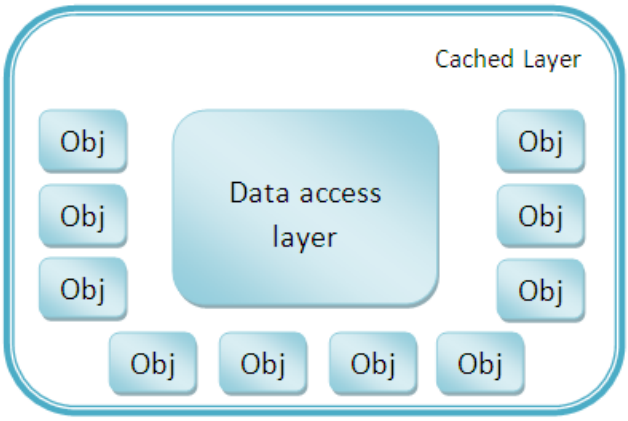


Figure 10: Caching at data access layer

In addition, it will also store the results of the intermediate queries and will update the cache based upon the updated results.

Currently, all the cached objects will be stored in the same JVM heap on which the system is running, however for scalability purposes the cache will have to be separated from the same JVM to some other server.

Business & Rule engine components caching:

The cache server, in addition to storing only the db objects will also store the intermediate results which will be either time consuming to compute or resource intensive. This will primarily be achieved by applying caching the two different types of components:

1. **Business components:** All the business components which will be responsible for executing a single unit of the business logic will be cached based upon the type of input parameters which are being passed to the function. This will be achieved by generating a **key** out of all the parameters and store the result with that key in the cached server. By default, the key is going to be a combination of the parameter's HashCode which has to be overridden by our own implementation of the key because of the scalability purposes.
2. **Rule engine components:** All the rule engine components where there might be a possibility of executing the same rules over a same set of values, caching play very important role in order to increase the performance. Since it will save the time of the process, to reach the rule engine server and wait for the results. It will also be helpful in the case where there is a need to know a special result based upon a set of conditions. E.g. the interest rate applicable based upon the country of the user and the shipping address.

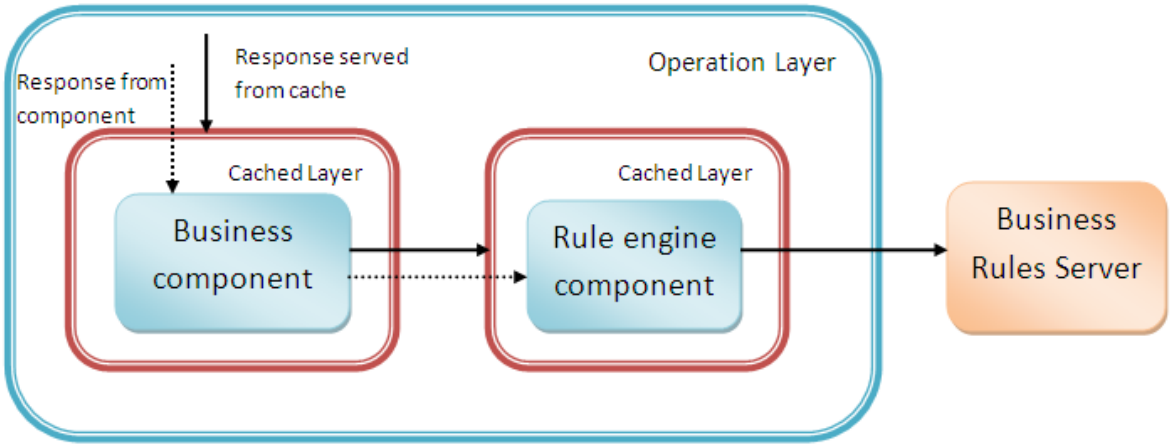


Figure 11: Caching in Rule management & extraction

In order, to make sure that the data in the cache is always updated we have to make sure that the cache data should invalidate itself after a specified period of time or if we update the computational data, the results should get changed. This will be the responsibility of the code to invalidate the data in the cache, if some computational data is changed in some other component. E.g. if the customer address is changed, then automatically the cached rule data should be invalidated.
To achieve this, we have to include and configure the **ehcache.xml** and have to include all the different types of cache's we want to maintain within our cache server.
e.g. **ehcache.xml snippet**

```
<?xml version="1.0" encoding="UTF-8"?>
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd"
updateCheck="false">
  <!-- Location of persistent caches on disk -->
  <diskStore path="java.io.tmpdir/EhCacheSpringAnnotationsExampleApp" />
  <defaultCache eternal="false" maxElementsInMemory="1000"
    overflowToDisk="false" diskPersistent="false" timeToIdleSeconds="0"
    timeToLiveSeconds="600" memoryStoreEvictionPolicy="LRU"/>

  <cache name="contactsCache" eternal="false"
```

```
    maxElementsInMemory="100" overflowToDisk="false" diskPersistent="false"
    timeToIdleSeconds="0" timeToLiveSeconds="300"
    memoryStoreEvictionPolicy="LRU" />

  <cache name="searchResultsCache" eternal="false"
    maxElementsInMemory="100" overflowToDisk="false" diskPersistent="false"
    timeToIdleSeconds="0" timeToLiveSeconds="300"
    memoryStoreEvictionPolicy="LRU" />
</ehcache>
```

In order to use the above caches in our code, all the functions (whose results has to be cached) will be annotated with **@Cacheable** annotations available from either EhCache-Spring-annotation library or directly from spring cache module. The following code shows how this annotation can be used.

```
@Cacheable(cacheName=" searchResultsCache")
        public Customer getCustomer(String customerId){
                Customer c=new Customer();
                c.setId(customerId);
                c.setName("Adeel Shafqat");
                c.setAddress("Address");
                return c;
        }
```

Similarly, in order to invalidate the contents of another cache based upon the updation of some entity we can use **@TriggersRemove** annotation.

```
@TriggersRemove(cacheName = " searchResultsCache", when =
When.AFTER_METHOD_INVOCATION, removeAll = true)
        public boolean clearCache(){
                return true;
        }
```

In case, the cache is needs to be clustered and used my multiple application servers, extra configuration needs to be done, which will be elaborated in the section "**Scalability**".

3.4.2 Data Transformation

Customer management system provides the functionality to represent the response data into multiple formats (XML/JSON) as requested (XML will be the default one). This is achieved by extensive usage of Spring mapping converters with Spring MVC implementation of JAX-RS specification.
Automatically, based upon the preference of the user (this will be identified by the present of a ACCEPT header in the http request type), the system will be able to return the data in that format. This complete configuration will be totally independent of the code and will be configured through the spring application context (XML) file. This will help the developer to unit test the function easily without any dependency.

Following example illustrates it:
Spring application context file will link all the converters required to the Spring MVC

```
<bean id="jsonConverter"
class="org.springframework.http.converter.json.MappingJacksonHttpMessageConverter">
```

```
                <property name="supportedMediaTypes" value="application/json" />
        </bean>
```

This will automatically find the Jackson JSON processor on the classpath and will use the same for converting the object to the JSON data.

```
<bean id="marshaller" class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
                <property name="classesToBeBound">
                        <list>
                        <value> com.verio.ossb.system.customer.response.CustVO</value>
                         <value> com.verio.ossb.system.customer.response.AccVO </value>
                        </list>
                </property>
        </bean>
<bean id="xmlConverter"
class="org.springframework.http.converter.xml.MarshallingHttpMessageConverter">
        <constructor-arg ref="marshaller" />
        <property name="supportedMediaTypes" value="application/xml"/>
        </bean>
```

Similarly, this will automatically register the JAXB libraries present on the classpath with the spring based OXM marshaller.

Once, we have both the marshallers registered, we can register them with the Spring MVC annotation method handler.

```
<bean
class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter">
                <property name="messageConverters">
                        <list>
                                <ref bean=" xmlConverter" />
                                <ref bean="jsonConverter" />
                        </list>
                </property>
</bean>
```

Spring Controller:

```
@Controller
@RequestMapping(value="/account")
public class AccountController {


        @RequestMapping(method=RequestMethod.GET,value="{accid}")
        public @ResponseBody AccVO getAccountDetail(@PathParam("accid") Long accid)    {

        AccVO accountVO = new AccVO(accid);
        ..................
        ..................
        Return accountVO;

          }
}
```

Now invocation of the API "**/account/897989**" (without any explicit **Accept** header set) will return the data in an XML Format.

```
<?xml version="1.0"?>
<Account>
        <id>897989</id>
        <accname>Verio Account</accname>
        ..............
        ..........
        .......
</Account>
```

Invocation of the API with the accept header set to application/json will return the data in JSON format.

```
{
 "Account": {
  "id": "897989",
  "accname": "Verio Account"
 }
}
```

Similarly, other converters (even a custom converter) can be attached to manipulate the response data based upon the accept header type.

3.4.3 Data Security/Access Policy

As customer mgmt system provides a wide variety of data based upon the request, it provides the view of data based upon the role of the user (i.e. authorization of the user with the SSO server). So the accessibility of the data is being protected by the role of the user who is invoking the Rest API via its credentials. This has been implemented by making use of Spring Security 3x, which acts as a security layer over all the services, and will allow executing the function or accessing the services only if the sufficient authorization exists with the user.

The following diagram illustrates that how the spring filter intercepts the call and proceeds ahead with the execution.

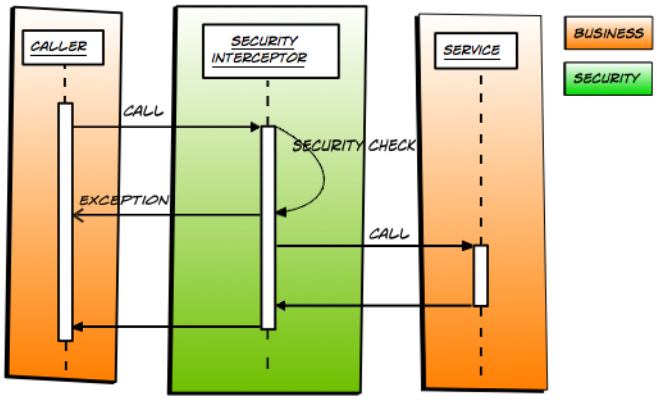


Figure 12: Spring Security Interceptor

This is being accomplished by declaring a filter in web.xml file that will take input a configuration file that describes the URL's on which the security needs to be applied, with clearly defining down the role which must be supported by the credentials sent in the user request.

```
<filter>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>

<filter-mapping>
    <filter-name>springSecurityFilterChain</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

All the configuration will be stored in an xml file (that will be passed as a context parameter to the web application), which will indicate the level of security that will be applied.

```
<context-param>
 <param-name>contextConfigLocation</param-name><param-value>
        /WEB-INF/spring-security.xml
         /WEB-INF/applicationContext.xml
 </param-value></context-param>
```

Spring security will make use of the various Authentication providers as defined in the XML file, to authenticate the user credentials across a request.

Customer mgmt system, in addition will also provide business security by securing the individual methods, so that only those user can access them who have the right authorization. This can be configured again by annotating the business methods with the set of roles that access the business functionality.

```
@PreAuthorize("isAuthenticated() and hasRole('user')")
public void addOffering(Offering offering){
                    .....
                    .....
        }
```

This will make sure, that the invoking user must have the role '**user'**. Otherwise, an exception is being thrown by the container, which will be getting caught at the upper layer, and will return in the form of an unauthorized **xml/json** to the client.
Following annotation will be used to mark the business logic method as secured.

1. @PreFilter
2. @PostFilter
3. @PreAuthorize
4. @PostAuthorize

3.4.4 Transactional Services

Comprehensive transaction support is provided by the Customer management system to ensure that all the changes will be successfully committed to the database, in case of any error/exception thrown by the code because of any reason all the previously committed statements (in the same transaction) will be rolled back to make sure, either all the changes should be visible in the database or not a single one. i.e. database should be in consistent state prior and after the execution of the business logic.
Customer mgmt system will heavily make use of the transaction support provided by Spring transactions by making use of the underlying Hibernate transaction manager (In case we are using more than one database, we can switch over to the XA transaction manager provided by the application server). Spring in turn makes use of Aspect object programming (AOP) to perform the transaction. All of this is going to be accomplished by the spring AOP which uses method level intercepting to generate the transaction controlled wrapper classes.

Every function present in the Business component will marked with **@Transactional** to indicate it participates in the transaction (which can be READ only or READ WRITE only), with another attribute "**rollbackFor**" to indicate the list of exceptions on which the transaction has to be ROLLBACK. Every transaction has to start afresh while invoking any function present in the business components in the operation layer, indicated by marking attribute "**PROPOGATION**" as **requires_new**.
Similarly at the DAL every function will be marked with @Transactional annotation with the **PROPOGATION** attribute as **REQUIRED** to signify, that a transaction is required before executing the data access logic.

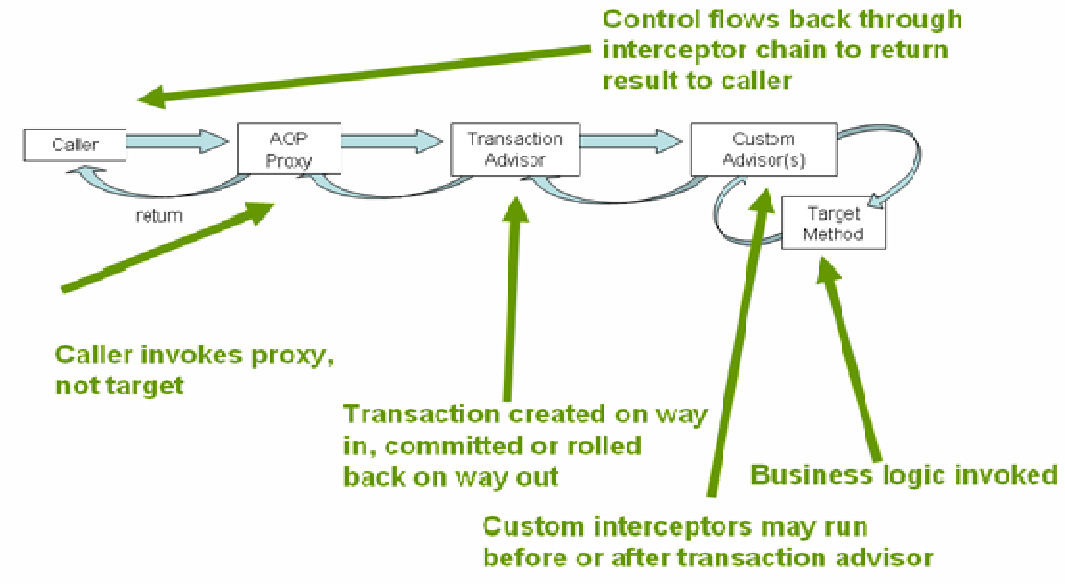


Figure 13: Sprint AOP in action for Transaction Management

In the Spring application context file, we will initialize the session factory and bind it to the transaction manager.

```
<!--  Transaction Manager -->
<bean id="transactionManager"
        class="org.springframework.orm.hibernate3.HibernateTransactionManager" >
      <property name="sessionFactory">
            <ref bean="sessionFactory" />
```

```
                </property>
        </bean>
```

In addition, we have to add the support of transactions.

```
        <!--  Transaction Scan -->
        <tx:annotation-driven    transaction-manager="transactionManager" />
```

Once added, spring will scan the packages to identify the functions annotated with **@Transactional** and will apply the method level interception to start/commit the transaction before and after the invocation of the function.

Given example shows how each of the function will be coded in business component.

```
@Transactional(rollbackFor = {Exception.class}, propagation = Propagation.REQUIRES_NEW)
        public AccountVo createAccount(AccountDetailVO accountDetail) throws Exception{

        }
```

Similarly, each of the functions in the DAL will be annotated like this, to indicate that a transaction must exist before invocation of this function.

```
@Transactional(rollbackFor = {Exception.class}, propagation = Propagation.REQUIRED)
        public AccountVo createAccount(AccountVo accountVO) throws Exception{

        }
```

For transactions to work, it is mandatory that any exception occurs during the execution of the code should be thrown back in the hierarchy above.

3.5 External Interfaces (API)

Customer management system will be deployed as a web application archive in the Jboss 7.1.0 application server and will publish a set of REST API's over http to manipulate the resources. All the published API's will be exposed as WADL defintion. This will include all the request parameters (mandatory or operational), their format and the same will apply for the response data and its format.

4 Other Design Considerations (Revise)

4.1 Globalization

Customer management system will provide the ability of reading and writing the data to/from the http request and also the communication between the system and the database in a globalized way i.e. the data taken in the request and the data returned in the response is being automatically being adapted to various languages and regions without any engineering changes. This is achieved through a two part process: internationalization (the process of decoupling your application from any particular language), and localization (adding one or more language / country specific presentations.) This is because of the diversity of the database design and the way the application is being coded. Every table where there is a possibility of storage of text (i.e. name, description etc.) that can be changed based upon the locale will be stored in separate tables instead of the main tables. This is achieved by making use of the Junction tables (join tables) that integrates the various records to that multi-lingual part by using a junction table id which contains the primary key of the tables (i.e. locale table and the table storing the record).
Thus retrieval of every data record that has the details stored in the multilingual format will be achieved by making use of other junction tables. As this process in itself, is going to be data intensive and time consuming, this will make efficient use of Hibernate second level cache & external cache to store the mostly used data elements in the cache.

Customer management system will perform all the operations in a character-oriented manner rather than byte-oriented way, to safeguard the correct usage of the encoding and the characters used in the request/response data. It will use the interfaces "**Reader**" & "**Writer**" as provided by the Jdk and the reading and writing operation is to be performed on the **StringReader** & **StringWriter** (enclosing the input/output stream) rather than directly on the input stream or output stream.

In addition, the request will specify the kind of locale in which the user wants the data to be transferred to make sure, the end user (either portal user or Administrator) receive the correct data. In case the local is not being specified **en_US** will be treated as the default locale.

Modularization

Customer mgmt system is composed of a number of sub-projects built by Maven to generate the final deployment archive, thus it is a multi module maven project which uses some of the external projects (as dependencies), which are although not part of the Customer mgmt system but will be used throughout the OSSB system components, thus forming a common ground for placing the common functionalities & utilities.

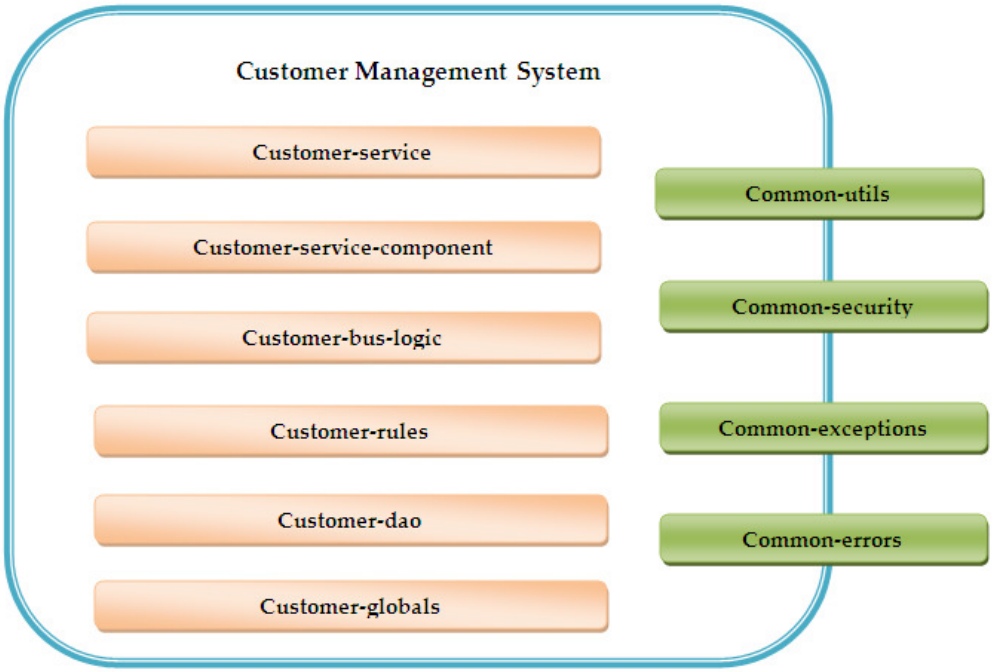


Figure 14: Application Modules

Modules:

1. **Customer-Service:** Customer service forms the important part of all the modules and will be responsible for directly exposing the services of the customer mgmt system to the consumers via Http. Thus it is the single point of contact for all the communications related to the Customer mgmt system. For each of the resources exposed by the system this layer will contain a directly mapping of the Http methods on a Uniform resource identifier (URI) on various functions present in the service class responsible for handling all the functions related to the resources.
   All the classes related to this module will be going to present in the package hierarchy "org.verio.ossb.system.customer.service" with a single service interface and service implementation for each resource. Every class is being annotated with **@Controller** annotation with **@RequestMapping** that indicates the URI on which the function is going to be invoked.
2. **Customer-service-component:** Customer service component module is directly related to the customer-service module and will be responsible for returning a client facing value object (that can either be converted to JSON/XML) to the upper service layer. This module will contain all the error/exception handling that may or may not occur while processing the business logic for the resource invocation. Thus the sole purpose of this module is to make sure that every time a well formed response will be returned to the client, no matter what type of exception or error has occurred on the service side.

All the classes related to this module will be going to present in the package hierarchy "org.verio.ossb.system.customer.service.component" with a single service component interface and service implementation for the interface. Every concrete implementation class will have to be annotated with **@Service** annotation.

3. **Customer-bus-logic:** Customer-bus-logic module will form an inherent part of Operation layer and will be responsible for execution of the business logic related to the request being invoked by the customer. Every request may or may not being mapped to a single unit of business logic and may spawns across the execution of multiple business logic functions. E.g. Creation of a customer may result in performing all the database work and also sending a notification mail or etc. Each of the class present in this module will have to be marked with **@Component** annotation with every function have to annotated with **@Transactional** annotation
   All the classes present within the class will fall under the package hierarchy of "org.verio.ossb.system.customer.business.component". In additional this class will contain a series of exceptions termed as business exception which may arise while processing the business logic. The same will be handled by the service-component layer to return the appropriate message/error description.
4. **Customer-rules:** This module will abstract the rest of the customer mgmt system with all kind of business rules to be applied while processing the business logic. This module will expose a specific set of interfaces depending upon the grouping of the rules to either validating a specific condition for an object or compute a specific value based upon the input parameters in the object being passed. This module will make extensive usage of the rule set in the drools rule engine and being accessed over Http by making use of **Guvnor**. This module will make use of a number of properties file configured through the OSSB_HOME environment variable for accessing many dynamic changing parameters. This module will make extensive set of client libraries provided by Drools in terms of "**KnowledgeSession**".
   All the classes present within the class will fall under the package hierarchy of "org.verio.ossb.system.customer.business.rules". All of the classes will be annotated with Spring @**Component** annotation to mark them as a Spring bean.
5. **Customer-dao:** As the name suggests, this module will abstract the communication of the complete customer mgmt system with the database by communicating in terms of specific value objects termed as 'entities.' All of the functions present within the classes will be going to be **transactional** in nature and will contain only the logic of processing the records from the database and return the corresponding entities VO to the upper layer. To simplify the distribution, the module is going to contain a value object per entity and a DAO interface & concrete implementation related to it. All the functions are going to be participate in the existing transactions (if it exists, otherwise will create a new transaction afresh).
   All the classes present within this module are going to be under the package hierarchy of "org.verio.ossb.system.customer.db". For each class, there will be a interface and an concrete implementation class. Each concrete implementation class is going to extend **HibernateDaoSupport** provided by Spring DAO support and can throw a **DataAccessException** in case of any exception being generated while executing the SQL Queries.
6. **Customer-globals:** This module will be the one which will be used across all other customer mgmt system thus will encompass the global functionality which will be relevant to the Customer system only. This module will primarily be divided into a number of packages as per their roles.

a. **Constants :** All the constants which are to be used, across the customer mgmt system will be present within this package. The various constants that can fall in this part is:
    i. Database related constants
    ii. Service related URI Constants
    iii. Rules related constants
b. **Validators:** All the validations which need to be applied either to the request value object at the service level or the one which is being passed downwards towards the business component will form a part of this package. This will primarily implement the JSR-303 specification by making use of Spring Validators with annotation support.
c. **Utilities:** Any kind of utilities which may be specific to the customer mgmt system will form a part of this package.

External Modules

1. **Common-utils:** This module will contain all the general purpose utilities that will be used throughout the OSSB-SOA (CPPO) project across various systems like Customer System, Product system, Order system etc. Some of the functionalities present within this includes:
    a. Encryption utilities
    b. Data transformation utilities
    c. General settings which are about to remain same across all projects

   All the classes present within this module will fall under the package hierarchy of "org.verio.ossb.soa.utilities".
2. **Common-security:** This module will contain the security related classes related to all the systems present within the CPPO project. This will include
    a. Roles related classes
    b. Roles Validator
    c. Authentication and Authorization classes
    d. Identity server Interactor
    e. URI signature verifier's

   All the classes present within this module will fall under the package hierarchy of "org.verio.ossb.soa.security".
3. **Common-exceptions:** This module will contain all the general purpose exceptions that can be reused across the SOA CPPO project. The idea is to get all the general exceptions out of the main code and place them in a separate module project. Some of the general exceptions that fall in this category is:
    a. General purpose exception
    b. Security exceptions
    c. Authentication and authorization exceptions
    d. General data access exceptions

   All the classes present within this module will fall under the package hierarchy of "org.verio.ossb.soa.exceptions".
4. **Common-errors:** This module will contain all the general purpose errors that may occur during the processing of a business logic or rule in any of the CPPO project. The idea is to centralize the common set of exceptions into a single module.
   All the classes present within this module will fall under the package hierarchy of "org.verio.ossb.soa.errors".

4.2 Exception Management

Customer mgmt system provides a robust exception handling mechanism that makes sure, that any error/exception propagated in the system will be routed to the appropriate handlers, which include logging of the exception in logs, sending of the exception details to a jms queue via QueueAppender functionality provided in the logging frameworks. Combining with the notification framework the exception mgmt framework will make sure that the exception is being traced for analysis purposes. Exception occurrence is being handled differently at different layers. E.g. at Services layer, if there is any exception arises while processing a request, the request will be handled by another method present in the same layer, thus indicating to the client that an error occurred in the response. In service component or business layer, where all the activities of an method must be under a single transaction, occurrence of any transaction will make sure the transaction is not being committed and silently being rolled back, with the message is being passed to the upper layer.

Thus, an exception management framework helps to:

1. Manage exceptions in an efficient and consistent way
2. Isolate exception management code from business logic code
3. Handle and log exceptions with a minimal amount of custom code

Exception Framework generates exceptions when it encounters an error condition caused by a coding mistake or a problem with the environment. Some examples are:

1. An invalid parameter is passed to a function like passing a string where an number is being excepted
2. Passes a value greater than 31 to a function which is expecting a Day-of-the-Month as a parameter
3. Trying to execute a service without proper authorization

Exceptions present in customer mgmt system are being broadly classified into three ways:-

1. **Database exceptions:** As the database integration part is totally based on the Spring-hibernate layer, all the checked & unchecked exceptions of the JDBC and Hibernate being automatically converted to the **DataAccessException** (provided by spring dao support), which is an unchecked exception and thrown back to the upper layer. As every method in the DAO layer participates in the transaction, the changes made by the method will automatically be rolled back.
2. **Custom Business exceptions:** All the exceptions that may arise out of the business conditions will fall in this category. This includes all the validations that are being applied by the rules engine on the model value objects, as well as other ones, which does not validate as per the business logic implemented by the code. All the business exceptions are going to be the checked exceptions, so the code must have to provide a alternative way to define the approach once an exception occurs.
   Generally, following is the hierarchy of the codes & the kind of exception that will arise out of this.

Code	Description
100-199	All the database related exceptions, that will percolate above in the hierarchy
200-399	All the business logic related codes (including rule engine) will fall in this hierarchy
400-499	All the errors that may arise out of inbuilt exceptions (i.e. programming errors) like NullPointException

	ArithmeticException will fall into this category.
500-599	All exceptions due to resource failures will fall into this category.
600-699	Security Exceptions

3. **Security Exceptions:** All the methods present in the security component must only be executed by a user, which has the proper authorization to do so. Attempts to invoke a method with improper authorization will results in an Security exception.

4.3 Logging

Customer mgmt system will provide two different kind of logging, to capture the trace of the request across the various components:

1. **Local Logging:**
   In this type of logging, the system will generate the logs on the local file system where the application is being deployed to. The actual configuration of the logging will be stored in the file "logback.xml" which will be referenced via a system property "**logback.configurationFile**" (which points to the configuration file) set in the application server. All the local logs will be configured to rolled back in the zip file after every night or if the size reaches to more than 200 MB (or more). The level of logging can be configured directly for different packages by introducing the package name and the level of logging we want to associate with it. The configuration file will be scan after a specific file time (configured in the file itself) and will be automatically reloaded by the spring container. In order to separate the logs of individual users, MDC feature of logback will be used, which will precede the logs generated with the name of the user who is invoking the API. In addition, we can also use the local logging as provided by the logging exception framework.
2. **Remote logging:**
   Customer mgmt system in addition to local logging will also make use of remote Logging framework to log the exception onto a remote system.

Slf4j will be used as the logging API (with the native implementation of **Logback**)

In addition, Most of the spring beans which are being declared as **@Service or @Repository or @Controller** will be marked with the JMX annotations to make sure, that the bean will be exposed through an MBean server (generally embedded with the Application server). This will be advantageous as the bean can be monitored for the performance via a JMX console.

E.g .

```
@ManagedResource(objectName = "spring:name=simpleBean")
@Repository
public class AccountDao
{
    ..................
    ..........
    .......
}
```

4.4 Configuration Files

Customer management system and its various sub-modules will depend upon a number of configuration files in order to function correctly. All the configuration and properties file's will be kept outside to modules and the project, so that no xml and properties files will be embedded in the Jars files.
All the projects defined under the CPPO umbrella will adhere to a set of common standards for using any of the properties or configuration files.
In the file system, there will be a separate directory "**OSSBHOME**" created and a system property "**OSSB_HOME**" will be created that will point to that directory. All the projects under CPPO will use this directory to store all kinds of configuration files (as depicted below).

The folder associated with the project will contain a number of directories separated by its functionality.

1. **Log:**
   This directory will contain the configuration file required for the project or module. Each of the project will contain a logback.xml file where the binding of the packages with the logging level is being defined.
2. **main:**
   This folder will contain all the configuration files that are must for functioning of the customer management system. In case of customer system, following files will be kept there:

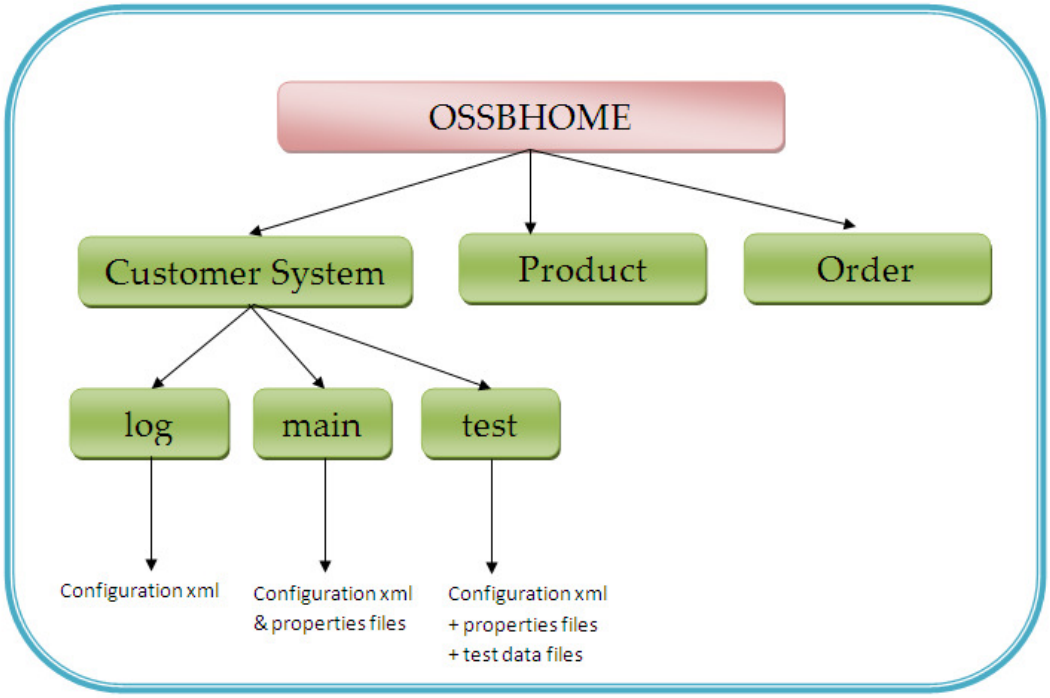


Figure 15: Application Configuration

application-dao.xml: This file will contain all the database related things, e.g. hibernate configuration, pooling properties, transaction details etc
dbconfiguration.properties: All the database related properties e.g. username & password will be stored in this property files.
application-configuration.xml: This file will contain definition of some of the beans

required to configure the customer mgmt system like configuration of beans for managing the properties files.
application-integration.xml: This file will contain all the integration points that will be required for the system.
application-web.xml: All the web specific settings will be defined in this xml file. It includes declaration of all the type of Converters, Validators etc.

3. **test:**
   This folder will contain all the xml/properties files requires for executing all the Junit based test cases. This will be the responsibllity of an application-context file present in the src/test/resources to load all these files by making use of Spring test Context framework. In addition, the DBUnit related xml files will also be present in this folder.

4.5 Notification framework

All the CPPO system including Customer mgmt system will provide a way to let the external world know of the various kinds of events happening in the system, this includes various operational functions performed on the various entities managed by the system E.g. creation of a new customer offering, changing of the details of an customer etc. The system provides the hooks only for a specific list of events that corresponds to the system and must be pre-known to the client.

This all is being controlled & managed by making use of Java messaging services (JMS 2.0). ActiveMQ 5.x will be used as the messaging broker for passing the message to the intended listeners. Each of the communication to the listener will be done via an XML that describes the type of event occurred, with the details of the message. With the event occurred, the type of details present in the JMS message payload (xml) may change to cover all the details of the event.

All the Notifications will be routed based on the TOPIC-SUBSCRIBER concept, with a single topic being created for an individual event in the message broker. All the intended listeners for the event will be attached to the topic and can be either a durable subscriber or not. A durable subscriber will get the event details even it is not connected to the topic (once it is reconnected again).

Spring JMS will be used for the underlying implementation of for the communication. As this communication is not going to have any impact over the current running business flow, all the calls to the Message broker for creating a session – sending a message will be asynchronous in nature and will run in an separate thread.

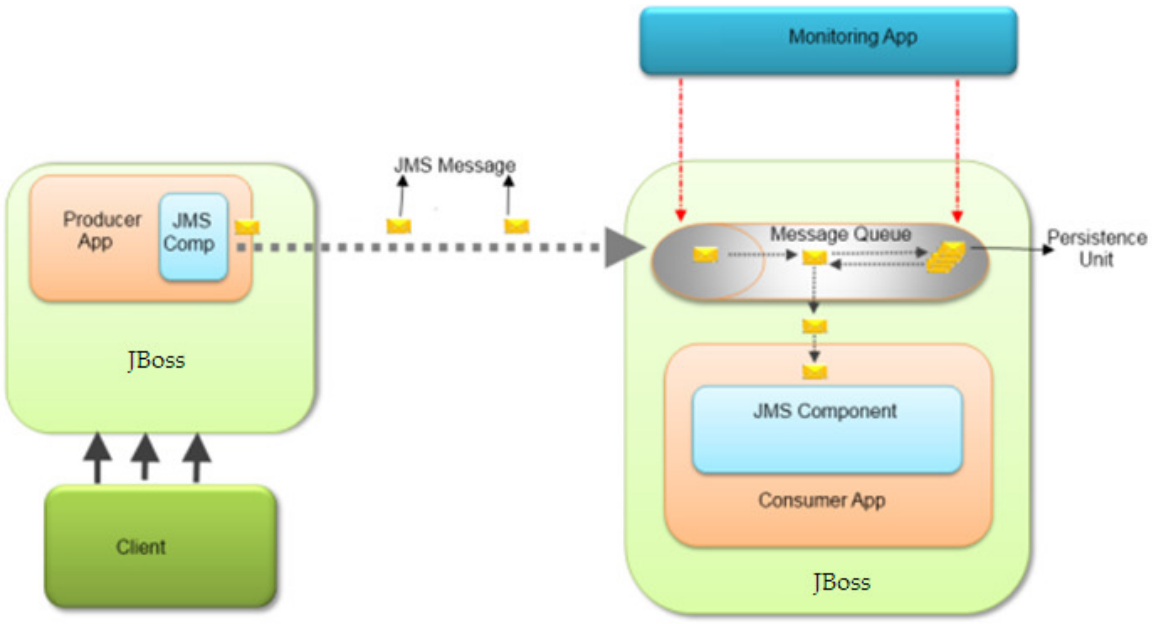


Figure 16: Notification framework

The following diagram will show the generation & flow of the message from the producer (product mgmt system) to the various consumers attached to the application.
SpringJMS template will be used for sending the messages to the topic. The following code shows the xml configuration for Spring JMS.

```
<!-- A Pooling based JMS provider -->
 <bean id="jmsFactory" class="org.apache.activemq.pool.PooledConnectionFactory" destroy-
method="stop">
   <property name="connectionFactory">
    <bean class="org.apache.activemq.ActiveMQConnectionFactory">
     <property name="brokerURL">
      <value>tcp://localhost:61616</value>
     </property>
    </bean>
   </property>
 </bean>

 <!-- Spring JMS Template -->
 <bean id="myJmsTemplate" class="org.springframework.jms.core.JmsTemplate">
   <property name="connectionFactory">
    <ref local="jmsFactory"/>
   </property>
 </bean>
```

For Unit testing of the code, any of the associated embedded brokers is to be used.

```
<bean id="broker" class="org.apache.activemq.xbean.BrokerFactoryBean">
   <property name="config" value="classpath:org/activemq/xbean/activemq.xml" />
   <property name="start" value="true" />
</bean>
```

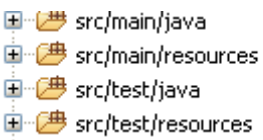
4.6 Junit Test Cases

Customer management system will make use of JUnit 4.x unit testing framework to unit test every functionality by writing multiple unit test cases per method to ensure most of the written code is covered and there are no surprises at the time of execution.
All the unit test cases will depend on the Spring-Test support framework provided by Spring (SpringJUnit4ClassRunner.class)
All the test cases that will be written will fall into three major categories:

1. **Unit testing:**
   Each of the functions written in the various modules will have to be well supported by a set of unit test cases, written for it. For this, the proper naming conventions as suggested by the Junit have to be followed.
   E.g. if there is class **AccountServiceImpl** in (src/main/java/****) then there must have a class having name **TestAccountServiceImpl** in the (src/main/resources/****) with the function name prepended with "test" and appended with the conditions on which the functions is to be unit tested. So a function "getAccountDetail" will have a number of test functions present in the corresponding test class.
      1. testGetAccountDetailWithoutUserName()
      2. testGetAccountDetailWithInvalidUserName()
      3. testGetAccountDetailWithInvalidPassword()

   Similarly, there will be sufficient number of test cases, to make sure the code written in AccountServiceImpl is being tested properly.
2. **Database code testing:**
   All the database related code which is written in the "**customer-dao**" module will be unit tested by making use of Junit & DBUnit. While running the DAO related unit test cases it is very much required that the database is not corrupted by the values as being placed in the test cases in it, so for this purpose Dbunit library has to be used, to make sure the original values are intact in the database. Before running any of the dao test code, the data source will be populated from the XML file, which will contain all the entities present in the database.
   In case the direct testing needs to be done on the database, the unit test code will make sure that the change done will be rolled back after the execution of the test case. This will be achieved by marking every database related test code by **@Rollback(true)** (a spring annotation)
3. **Integration testing:**
   In order to perform the integration test for a number of modules, inbuilt container Jetty has to be used. The entire configuration regarding this will be going to present in the maven configuration file. Before testing of the code, the generated archive file has to be deployed to the container. So, to test the REST based API, RESTFUSE library will be used.

Since, the project is a maven project, all the test related artifacts will be present in the test directory having the following structure:



src/test/java : will contain all the Junit test cases, having many-to-one mapping with the functions written in the src/main/java file.

src/test/resources : This directory will contain all the resources required for running the set of test cases, present within the module. As the project is based upon Spring, the application context will be booted once per module and all the beans will be shared across all the test cases written. For running the test cases related to a module, we will be place the base application context xml & associated files (required for booting of application context) file to each of the **test/resources** directory for each of the sub-project.
At any point of time during executing a test case, if it is required to refresh the application context then the function has to be marked with **@DirtiesContext**
In addition of the basic support for asserting (to verify the output with the expected output), another library HAMCREST will also has to be used.
Following is the sample code, to run a sample unit test case with spring.

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration({"file:src/test/resources/cloud-application-api.xml" })
public class TestProductOffering {

@Autowired
        OfferingDao offeringDao;

@Test
public void testGetProductOfferingWithIncorrectId()
  {
        --------------------
        ---------
        ----
  }
```

4.7 Development tools/plug-ins

Eclipse plugins:

1. PMD (Source code analysis tool):
2. JUnit plugin
3. M2Eclipse plugin
4. Sonar for code coverage

Tools:

1. Maven
2. Bamboo (Continuous integration tool)

4.8 Package Structure

VerioContact Message Instantiation System will be deployed as a Web Application archive with each of the components & dependencies will be present as a JAR file in the **web-inf/lib** directory of zipped war file. The following diagram depicts the various packages.

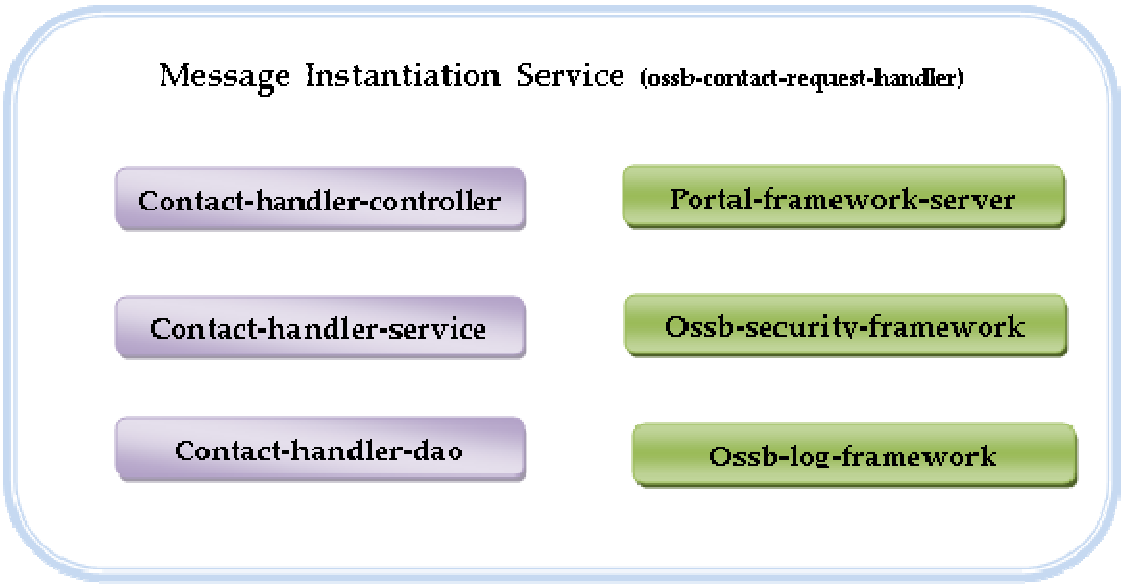


Figure 17: Application packages

1. **Contact-handler-controller**
   This web archive file (.war) will include RestFul services for requesting message communication and checking the status. All the configuration files related to this can be find in **${OSSB_HOME}/contact/** directory. The package hierarchy that will be followed by this component is "**org.verio.ossb.contact.web**"
2. **Contact-handler-service**
   This java archive file will include all the services related to request handler, assembler calls, and message transmitter components of VerioContact. All the configuration files related to this can be find in **${OSSB_HOME}/contact/** directory. The package hierarchy that will be followed by this component is "**org.verio.ossb.contact.service".**
3. **Contact-handler-dao**
   This archive file will include all the components of "**customer-dao**" module. All the configuration files related to this can be find in **${OSSB_HOME}/contact/** directory. The

package hierarchy that will be followed by this component is "**org.verio.ossb.contact.db".** The various files that will be included in this customer related JPA (Java persistence api)2.0 entities and the corresponding interface and implementation.

5 References