Table of Contents

Table of Contents

[Table of Contents 1](#_Toc512860218)

[1. Introduction 1](#_Toc512860219)

[2. RL System workflow and Component Overview 2](#_Toc512860220)

[3. WEB Component Architecture 3](#_Toc512860221)

[3.1 Single page application 3](#_Toc512860222)

[3.2 Revenue Leakage Project (Web Features) 3](#_Toc512860223)

[3.3 Architectural Overview (Web) 4](#_Toc512860224)

[3.4 Overall Design (Web) 4](#_Toc512860225)

[3.5 Technologies & Technology Used (Web) 5](#_Toc512860226)

[3.6 Organizing the workflow (Web) 5](#_Toc512860227)

[3.8 Design philosophy (Web) 6](#_Toc512860228)

[3.9 Extensibility (Web) 6](#_Toc512860229)

[3.10 Some of the common tasks workflow (Web) 7](#_Toc512860230)

[4. Revenue Leakage Back-end component architecture 8](#_Toc512860231)

[4.1 Implementation Details 8](#_Toc512860232)

[5. Backend Component Diagram 9](#_Toc512860233)

[6. Backend Application workflow: 9](#_Toc512860234)

[7. Application Security (Apache Shiro) 13](#_Toc512860235)

[8. Rest-Service (JAX-RS Jersey) 14](#_Toc512860236)

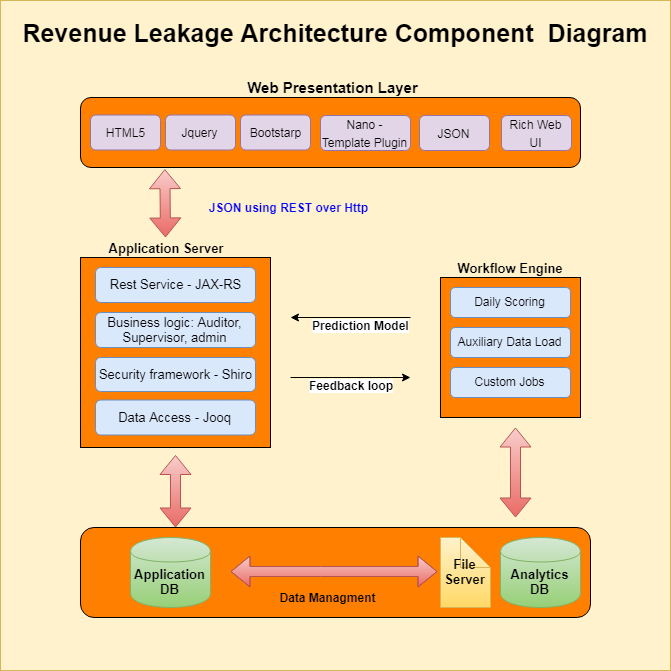
[9. DB Layer 14](#_Toc512860237)

[10. RL Model Layer 15](#_Toc512860238)

1. Introduction

This document is geared toward providing developers with the knowhow about the development process and the layout of the various components and architecture. This document is now meant to describe how the solution is working and the underlying business logic of the functionality. It will tell about the technology stack used in the development of this application.

1. RL System workflow and Component Overview

****

1. WEB Component Architecture

3.1 Single page application

Web application development has always been a major part of software development. However, there had been a lot of restrictions in interaction with complex web applications which made them not as responsive as were for example desktop applications

At the same time the new versions of traditional technologies used for web development such as HTML, CSS and JavaScript were evolving and progressing which made it possible to create a new generation of lightweight and flexible web applications - Single Page Applications.

New web application architecture that uses advanced JavaScript framework on the front end. it only sends raw JSON/XML data and HTML templates (partials) to the client on demand. Client-side of the application is fully responsible for generating user interface using received from the server data. In other words, whole user interface logic is moved from server to client-side of the application. Moreover, on Single Page Applications everything operates on a single page – there is no page refreshes, as, for instance, in traditional web applications where page reloads on each request. All above listed points provide more fluid and rich user experience.

3.2 Revenue Leakage Project (Web Features)

**Features**:

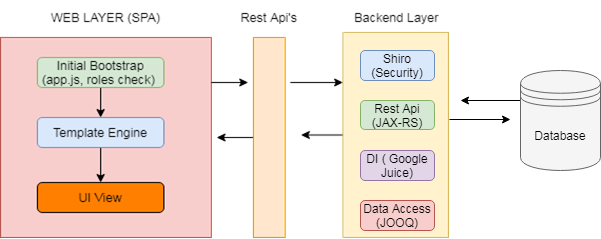
* Main controller of the project is app.js for bootstrapping and initializing.
* Project is working on multiple roles based on authorization. UI populate the schema based on the role.
* Internal structure of screens/tab is based on object patterns and every tab objects is using screen object, so that globally all object will part of screen object.
* Extensive use of grids. **Pq-grid** for searching and tree view (other cool feature) and **Jq-grid** for inline editing of rows.
* Project is using resource bundle api (i18n), all the static data is coming from resource bundle.
* For templating of DOM manipulation, Project is using nano.js plugin.
* Project is using high-chart and d3.js for charting purpose.
* Project is using jquery query-builder plugin for building query on UI side.
* Project is using bootstrap 3 library for look and feel (CSS) purpose.
* Local storage is using to store filter information.
* Event handing on UI side for all (400 and 500 series)

3.3 Architectural Overview (Web)

Overall UI architecture is based on front-end communicate back-end with rest api’s. UI receive data from service layer in the form of JSON object. Project is working on asynchronous approach. Every rest api’s is independent of others. Security is maintained by back-end layer (Shiro) for authentication and authorization. On UI side, we are following OWASP security guideline.

3.4 Overall Design (Web)

The main front-end architecture of the Revenue leakage project can be seen in the following diagram -



Project have MVC sort of architecture for constructing the front end to allow -

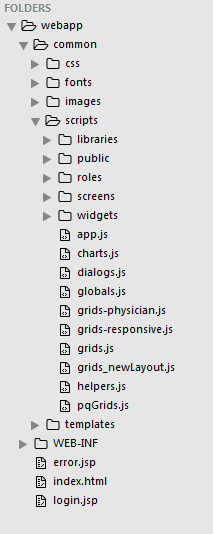
* Isolation of all data stored in the frontend from the view elements. As a result we are able to build a UI in which components of the view can be changed without any change in the application logic (javascipt).
* Having atomic controllers which handle only particular bit of the UI - their views and the corresponding data. Thereby allowing easy maintenance of the UI.
* UI is based on Authorization (roles); hence role specific templates and request will be executed.
* Product and clients are working on common and custom approach. Client specific modification, look and feel, library and modifications are part of custom client.
* Any Addition of product feature will be configurable, hence it may or may not be part of clients.

3.5 Technologies & Technology Used (Web)

* HTML5, Jquey, OOJS, High-chart, D3.js (chart), Jq-grid, Pq-grid, Bootstrap, i18n (resource bundle)

3.6 Organizing the workflow (Web)

Every front-end project always includes libraries, jQuery plugins, and a lot of JavaScript and CSS files for different purposes and aims. Blending all those elements means working with different technologies and putting them together means setting up a good front-end workflow. We would find ourselves having to manage a huge workflow that involves different technologies. That’s why having everything organized in folders, following a pattern or a convention in order to keep things clear and neat, is really important.

Starting from all of these preliminary considerations, this is how to organize the architecture, here is the folder tree:

Images - All images files here: .png , .jpg , .jpeg , wallpapers etc.

Css – All the css files reside in this folder

Fonts – Bootstrap fonts and other libraries fonts inside this folder.

Scripts – All the js files resides in this folder.

*Libraries –* 3rd part libraries are residing in this folder.

*Public –* Login specific libraries are residing here.

*Roles –* All the roles specific files are residing here.

*Screens –* All screens specific js files are reside in this.

*Widgets –* reusable widgets are residing in this folder

*App.js –* Project controller –( Roles, resource bundle, Project bootstrap)

*Chart.js –* All the charts reside in this folder

*Dialog.js –* All the dialogs pop-up resides in this folder

*Global.js –* Singleton access of all Objects and variables

*Grids.js –* Single point of all grids

*Helper*.js – All helper functions of Project

*Pqgrids*.js’ – Searchable grid of Projects

Templates – All the dynamic DOM templates are residing in this folder.

Login.html – This is required for login form. This is the first screen of the application.

Index.html – After login it will redirect to index page.

3.8 Design philosophy (Web)

UI built as a series of static HTML, JS, CSS which contracts the page layout and accesses all data from the service layer.  In general, the client should maintain all application state allowing service calls to be cached, distributed, etc. according to HTTP protocol (only exception is authentication cookies).  While the service layer is ideally generic, it will be built with the web UI as primary user, this will help avoid multiple chained AJAX calls which are difficult to design/debug.  The serve layer will be a RESTful interface and will be well defined and documented.  The service layer should strongly follow REST guiding principles (especially well defined use of URIs for resource identification, and hypermedia as the engine of application state).  Security is applied at the service layer (assume service client is untrusted/malicious).  Services will implement business logic/domain model and delegate persistence to data access layer.  In general a transaction per request approach should be used in which transaction starts with request received and ends before returning result through jersey framework (this ensure serialization to JSON and blocking I/O occur outside of transaction).  Data access layer consists of Dao style implementations using jOOQ query API.  jOOQ generated Record classes may be exposed to the service layer in lieu of DTOs which add no value.  Service layer should completely encapsulate database entity/design.  Note that this differs from "traditional" J2EE application in that the service layer and business logic layer are merged into one and persistence is handled in a truly relational way (I.e. not trying to automate object to relational mapping).

3.9 [Extensibility](https://confluence.operasolutions.com/wiki/display/RL/Extensibility) (Web)

The product development approach for the application is to have a core application which is fully functional on its own (and configurable to suit different requirements) with an additional ability to be extended with custom code for individual customers.  Customer specific code will always live outside of the core project, depending on the core (as a Maven dependency) and never duplicating core code.  The relationship between the customer modules and the core modules follows an extension-by-composition model.  This diagram illustrates the various artifacts at play

3.10 Some of the common tasks workflow (Web)

* + Addition of new Roles in the system
  + Addition of new screen (Tab) in the system
  + Customer specific change in the system
  + Configure/Plugin the feature in the Product
  + Adding template engine in the system
  + Adding of error handing (400/500) in the system
  + Adding in the key in the resource bundle
  + Adding of the dialog box in the system
  + Adding of grids/chart in the system
  + Common functionality addition in the system
  + Service integration (Rest api) in the system.

1. Revenue Leakage Back-end component architecture

**Component Overview**

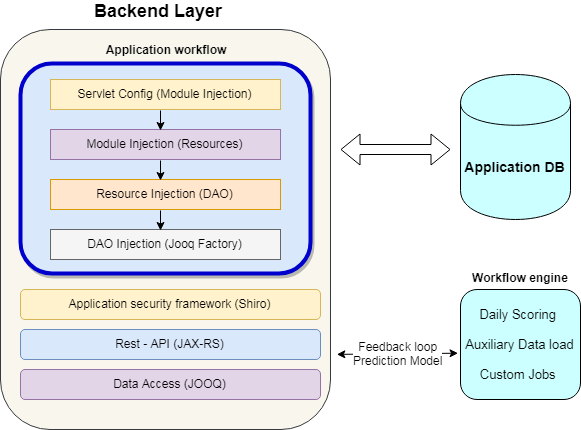
Back-end component structure consist of below component specially Application workflow. It shows that how we can add Modules, resources, access DAO layer and Jooq implementation.

1. **Application workflow**
   * Servlet Config (Module Injection)
   * Module Injection (Resources)
   * Resource Injection (DAO)
   * DAO Injection (JOOQ Factory)
2. **Application Security (Shiro)**
3. **Rest API (JAX – RS)**
4. **Data Access (JOOQ)**

4.1 Implementation Details

* The common utility are available in the following package: **com.operasolutions.rl.common.**
* Rule config code are in the package **com.operasolutions.rl.common.config.**
* Service package which have multiple module on contains the business logic, resource ,module etc of the application is present the package **com.operasolutions. rl.service.**
* All the files performing database operations are in the package **com.operasolutions.rl.schema.**
* All security relalated code in **com.operasolutions.rl.auth**.
* All the files for sending password notification mails are in the package **com.operasolutions.rl.common.email.**
* All the files needed for the utility fuctionality are in the package **com.operasolutions.rl.common.utils.**

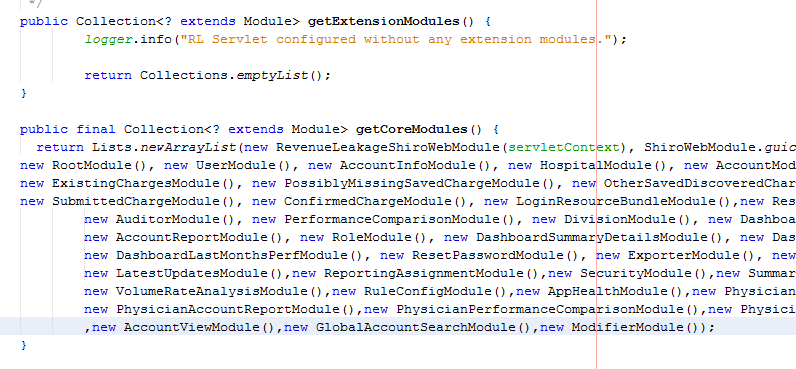
1. Backend Component Diagram



1. Backend Application workflow:

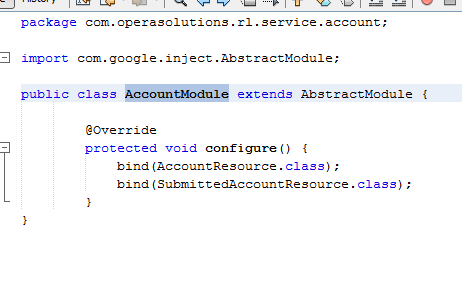
***Servlet Config***: -

* First, we need to register module in “RevenueLeakageServiceServletConfig.java” in class.
* Module registers with Servlet



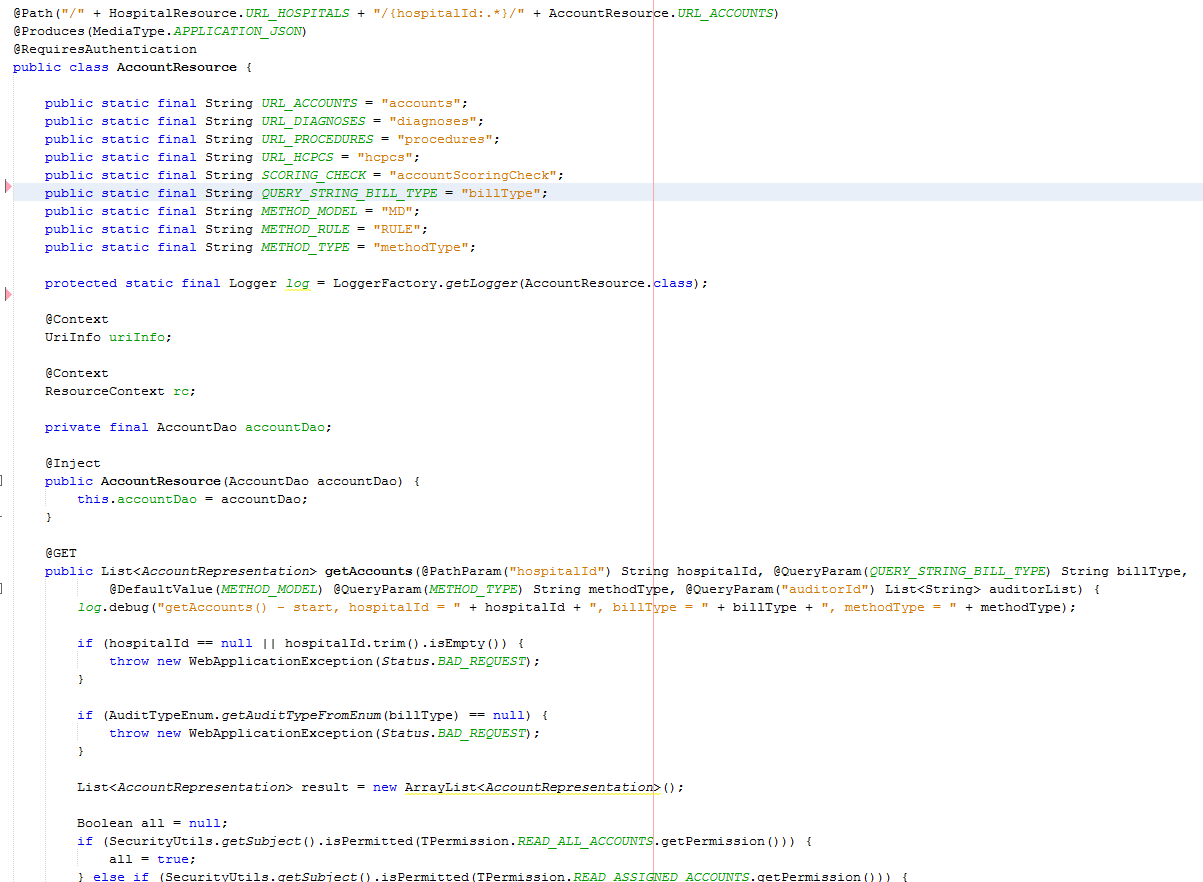
***Module Injection (Resources)***

* Resource register with modules using Juice injections.
* We can register multiple resources in single module.
* Resources is like a controller class which handle in-coming request from the client and return to the JSON to the client.



***Resource Injection (DAO)***

* Registering the DAO in resources and other utility like (Email sender)
* Initialize constant value related to resources like URL resource, Method model, Bill type, Default parameters.
* Creating representation(UI) and result(DB) class.
* Binding with resource context.
* Binding with URI Info.
* It will fetch the data with the help of DAO and return back to the UI using rest services.



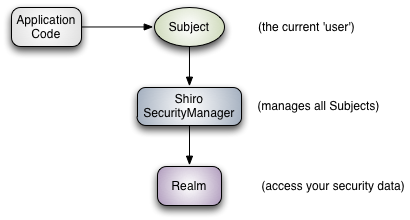
***DAO Injection (JOOQ Factory)***

* Binding with Jooq factory
* Creating Jooq query with using alias, joins and passing different parameters.
* Return the result set to the resource class.
* Jooq query fetch the data form My-Sql database and send to the resource class.



1. Application Security (Apache Shiro)

At the highest conceptual level, Shiro's architecture has 3 primary concepts: the Subject, SecurityManager and Realms.



**Subject:** Subject is essentially a security specific 'view' of the currently executing user or a 3rd-party service, daemon account, cron job, or anything similar - basically anything that is currently interacting with the software.

**SecurityManager:** The SecurityManager is the heart of Shiro’s architecture and acts as a sort of 'umbrella’ object that coordinates its internal security components that together form an object graph.

**Realms:** Realms act as the ‘bridge’ or ‘connector’ between Shiro and your application’s security data. A Realm is essentially a security-specific DAO: it encapsulates connection details for data sources and makes the associated data available to Shiro as needed. When configuring Shiro, we must specify at least one Realm to use for authentication and/or authorization. The SecurityManager may be configured with multiple Realms, but at least one is required.

1. Rest-Service (JAX-RS Jersey)

Java API for RESTful Web Services (**JAX-RS**), is a set if APIs to developer REST service. JAX-RS is part of the Java EE6, and make developers to develop REST web application easily.

* REST is an architectural style which is based on web-standards and the HTTP protocol. This style was initially described by Roy Fielding in 2000. In a REST based architecture everything is a resource. A resource is accessed via a common interface based on the HTTP standard methods. In a REST based architecture, you have a REST server which provides access to the resources. A REST client can access and modify the REST resources.
* Every resource should support the HTTP common operations. Resources are identified by global IDs (which are typically URIs).
* REST allows that resources have different representations, e.g., text, XML, JSON etc. The REST client can ask for a specific representation via the HTTP protocol (content negotiation).

1. DB Layer

**Database**

MySQL database will be used to hold meta-data (Pre bill,post bill ,coding edits etc.. ) related to the application and basic user information (e.g. user profile, user roles, ).

**Data Access Objects**

Data access objects abstract away the data storage and retrieval from the other parts of the application. They also provide convenience of converting domain objects to database specific objects for storage and vice versa. JOOQ framework is used to create the data access objects layer. The transaction management provided by the Spring framework is coupled with the data access layer to create service objects for various business objects in the application like attributes, constraints etc.

1. RL Model Layer

**Product Root**

* Within product will be having all generic code that will be common to all customers.
* This generic code is nothing but independent python modules like loader, monitor, scorer, notifier etc.

#### Product -> Backend

* Broadly Backend will have two folders in it.
* Python will take care of all the scripts that will monitor, load, notify, call the scorer, post scoring jobs etc.
* Models will have all the Scoring Algorithms that python script will call after data loading and/or after fulfilling any other customer requirement

#### Product -> UI

* This will have any UI related code

#### Product -> Backend -> Python

Python directory will have following sub-modules

* Loader will have following set of independent modules:
  + \_extract\_.py: Take care of extraction of client feed zip file to EXTRACT Dir.
  + \_transfrom\_.py: Any transforming or cleaning of data prior to data load in DB.
  + \_loader\_.py: Take care of loading data into DB. One table at a time.
  + \_archive\_.py: Take care of any archiving (if required)
* Monitor will have following set of independent modules:
  + \_sftpmonitor\_.py: This will implement SFTPMonitor Class. Will monitor SFTP for specific format client feed file. Once found will move file to LandingZone.
  + \_lzmonitor\_.py: This will implement LZMonitor Class. Will monitor LandingZone for specific format client file. Once found will move file for further processing (initiate Data Loading)
  + \_scheduler\_.py: This will implement Scheduler Class.
* Scorer
  + Blank for now. Will come into picture in future.
* Notifier
  + \_logging\_.py: Will have generic functions that logs progress of process.
  + \_mail\_.py: Will have generic functions to send email alerts to user when required in process.
* PostScoring
  + \_auditor\_allocations\_.py: Will get hospital list, get auditor list, get open accounts to be audited and allocate these accounts uniformly to auditors for hospitals.
  + \_allocation\_mail\_.py: Send mail to auditors with detail of assigned PRE/POST accounts.
  + PopulateLatestUpdates.sql :
  + PopulateSnapshotTables.sql :
* NewFolder: This will be blank in Product Implementation. *However in customer specific implementation we do have any post data loading, pre data loading, post scoring, pre scoring or any other client specific requirement code here*.

#### Product -> Backend -> Models

Models will have all the Scoring Algorithm that python script will call after data loading and/or after fulfilling any other customer requirement. Right now we have following Scoring Models available:

* CoOccurance
* DTModels
* EnsModels
* FBModels
* IP
* IsingModels
* LinRegModels
* LrModels

**Customers Root**

* Within Customers will be having ‘NewCustomer’ project folder by default.
* ‘NewCustomer’ will have default process/functional flow defined in it. For eg:
  + A monitor on sftp will put the file on LandingZone.
  + A monitor on LandingZone will put the file on processing by initiating Data Loader.
  + Data Loader then calls for extraction, loading to DB, archiving etc.
  + As Last step of Data Loader, process will call scorer (for Scoring)
  + After scoring Post Scoring job initiates. It include uniform allocation of open accounts to auditors, send mail to auditors with details of assigned accounts, Populate snapshot tables and latest update jobs.
  + Logging and Notification Mechanism will be running in parallel throughout the above process
* ‘NewCustomer’ project folder will make use all independent modules written in Product and align them in sequence (or in parallel wherever required).
* ‘NewCustomer’ will act as a readymade template having pre-build basic functionalities and process required by any new customer of RevLeakage.

Within Customers will be having customer specific project folders.

**NewCustomer** will act as a template of process/functional flow with built-in default functionalities like loading, monitoring, logging etc. Within NewCustomer will be having

* + UI: UI code
  + Backend: Take care of all Backend Processing.

**Backend** will have following folder within it:

* **Config**: Will have customer configuration files for loading, monitoring, scoring, logging etc
* **LandingZone**: This will be the place where client files gets download to.
  + **Extract**: This will be the place where client files get extract and then get loaded in DB
  + **Archive**: This will be the place where we archive our all client feed files and other intrim files
* **Logs**: Place where logs files get generated
* **Python**: Explained below in Detail. Will have all python scripts.
* ***start\_all.sh***: A shell script file which will start all process specific to customer.
* ***stop\_all.sh***: A shell script file which will stop all process specific to customer.

#### Customers -> NewCustomer -> Backend -> Python

Python directory will have following sub-modules

* + Loader
    - data\_loader.py: This file imports functionality from Product’s \_extract\_, \_transform\_, \_loader\_ and \_archive\_ modules and call them in a sequence to form end to end data loading of client files to DB.
    - In case if we want to customize any of above four modules of Product, following will be the steps. Taking an example of say customizing \_loader\_.py, wherein I want to update a table column after loading data to it:
      * Create a new file in Loader Dir and name it **loader.py**.
      * We don’t have ‘\_’ as prefix and postfix to filename, this shows functionality of \_loader\_.py got extended in loader.py.
      * Import function (say load\_data()) resides in \_loader\_.py that loads data to db table in loader.py file and give it an alias as base\_load\_data().
      * Now write a new function in loader.py file with name as load\_data() and first call base\_load\_data() in it and then do any further processing.
      * Once done then we will be importing load\_data() from loader.py file rather than from \_loader\_.py
  + Monitor
    - initiate\_monitor.py
      * Here in first we will import following three modules of Product:
        + \_sftpmonitor\_
        + \_lzmonitor\_
        + \_scheduler\_
      * Create life long object of scheduler and schedule
      * sftp monitor job and lz monitor job written in \_sftpmonitor\_.py and \_lzmonitor\_.py
      * schedule any client specific job resides in ‘NewFolder’ (will name this folder later)
  + Scorer
    - Blank for now. Will come into picture in future.
  + Notifier
    - logging.py:
      * Will import all generic functions from \_logging\_.py
      * Will also have customer specific logging functions as well (if required)
    - mail.py:
      * Will import all generic functions from \_mail\_.py
      * Will also have customer specific email/alerts functions as well (if required)
  + PostScoring
    - post\_scoring\_jobs.py
      * Have listed auditor allocation, latest update and Populate snapshot table Jobs
    - auditor\_allocation.py
      * Will import returns from \_auditor\_allocations\_ as base\_hosp\_list, base\_assignments, base\_openaccounts, base\_update
      * allocation\_distribution(): Will uniformly assign open accounts to auditors
      * preBillAssignmentJob() : PRE bill accounts assignment
      * postBillAssignmentJob() : POST bill accounts assignment
    - allocation\_mail.py : Send allocation mail to auditors after invoking \_allocation\_mail\_ module from product.
  + NewFolder: This will be blank in Product Implementation. *However in customer specific implementation we do have any post data loading, pre data loading, post scoring, pre scoring or any other client specific requirement code here*
  + **Main.py**
    - This will be the file which will be called by **start\_all.sh**
    - This file will does following
      * call initiate\_monitoring.py which will in turn Switch ON the process of SFTP monitoring and LZ Monitoring.
      * LZ Monitoring in its last step will initiate the Data Loading process.
      * Within Data Loading process, any pre loading/post loading Job can be called (if required) with else extraction-loading-archival will happen.