Project: **Whizzer 2018/115906**

Design and Architecture

Business Information Systems (Allevo) SRL**;** VAT no: 6117169

Grant type: Romania Innovation EEA grants (ROM-EEA 1153)

Document Control

Title:	Whizzer - Design and Architecture
Project:	**Whizzer 2018/115906**
Version:	2.2
Creation Date:	Date: 5.03.2020
Author:	[redacted]

Update history

*A - Added M - Modified D – Deleted

Version	Date	A* M D	Short description	Authors	Contributors
1.0	05.03.2020	A	Original document	[redacted]	
1.1	30.04.2020	A	Chapter 1		
1.2	29.05.2020	A	Chapter 2		
1.3	30.06.2020	A	Chapter 3		
1.4	31.07.2020	A	Chapter 4		
1.5	31.08.2020	A	Chapter 5		

1.6	30.09.2020	A	Chapter 6	[redacted]
1.7	30.10.2020	A	Chapter 7	
1.8	27.11.2020	A	Chapter 8	
1.9	31.12.2020	A	Chapter 9	
2.0	29.01.2021	A	Chapter 10	
2.1	26.02.2021	A	Chapter 11	
2.2	31.03.2021	A	Chapter 12	

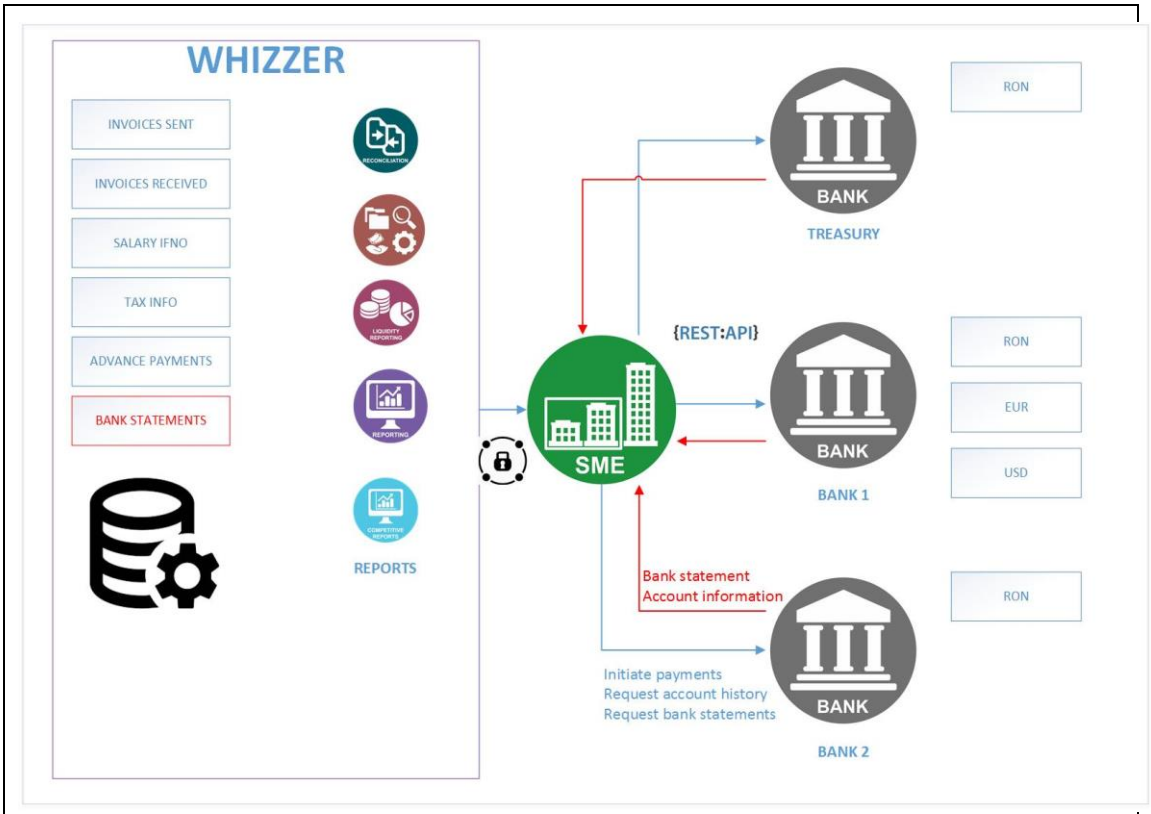
Contents

1 Introduction

FinTPc product is designed for those corporations working towards integrating their payment flows. It ensures centralized management of financial operations of one corporation or of a company group, providing a single interface for administration, monitoring and reporting of all the payments, regardless of the bank institution through which they are performed.

FinTP-Connect offers a communications layer that manages requests coming from TPPs and responses from the bank's side. The extension of FinTP-Connect offers a single window for balance sheet, salary, invoicing, and money flow automation, accounts payable and receivable capabilities, by using FinTPc-API, which ensures interfacing between data exposed by banks via Open Banking APIs and FinTPc functionalities, which centralize financial operations of a corporate.

Whizzer is a repackaging of the extended projectes above, ready to be deployed in the cloud. The solution delivers financial operations as a service to SMEs. This allows SMEs to achieve a lean internal infrastructure, only using the service that sits in the cloud. Whizzer provides multi-tenant capabilities necessary for delivering Software as a Service and financial reporting which are easy to understand and use by SME's Administrators without a financial background.



business overview

1.1 Intended audience

This document is addressed to or can be consulted by:

- Architects
- Developers
- Testers
- Implementers
- Business Analysts
- Sales and Marketing

Reading suggestions:

- Whizzer - SRS - Software Requirements Specification

2 Architecture

The IEEE1 recommendation defines an architecture as the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution. Architectures represent the abstraction used to understand any system and also form the basis for a shared understanding to all its stakeholders.

Application architecture seeks to build a bridge between business requirements and technical requirements by understanding use cases, and then finding ways to implement those use cases in the software.

An architectural overview is aimed at providing a shared understanding of the architecture across a broad range of people including the developers, marketing, management and possibly potential end-users. An architectural overview is ideally produced early in the development lifecycle and serves as the starting point for the development. An architectural overview should be at a high level of abstraction. All the major functionalities and components of the architecture should be described but the descriptions may lack detail and precision as they often use natural language rather than formal notations.

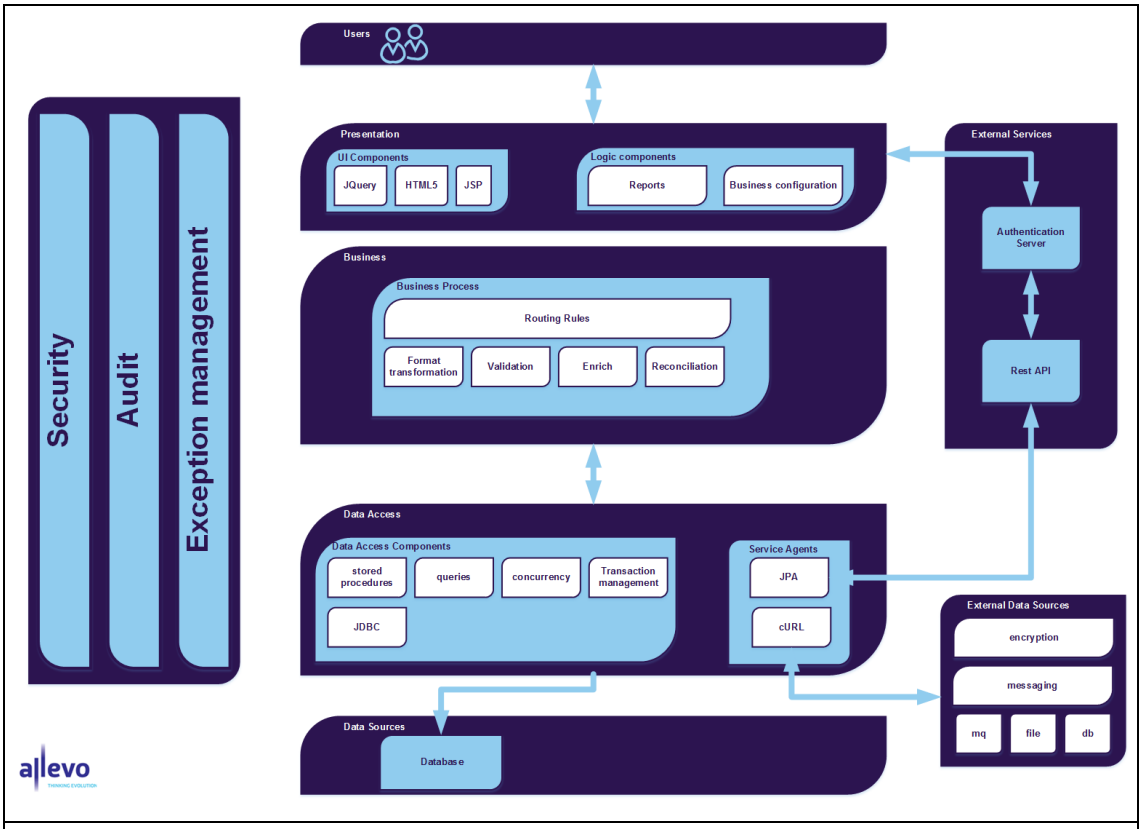
This current document is describing this high level, overview architecture from different perspectives.

Unlike this architectural overview, the detailed architecture of the application should be a living document that is constructed collaboratively by the development team as the development proceeds. As it develops, the detailed architecture document can be used to assess the impact of different requirements changes. All this detailed architecture documentation is structured into a different document – “FinTPc – detailed architecture”.

[1] Institute of Electrical and Electronics Engineers

3 Logical architecture

3.1.1 Logical layered architecture



single tenant::logical layered architecture

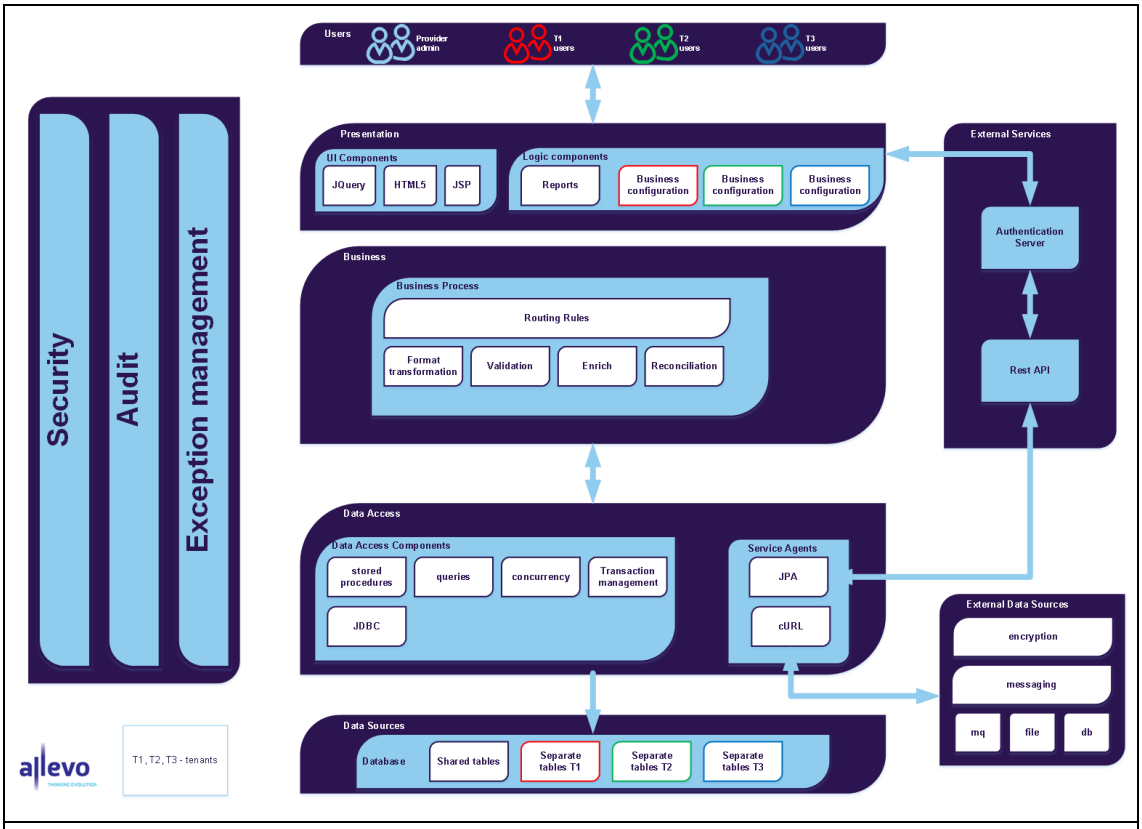
The upper diagram describes a layered architecture style of the FinTPc application.

The components of the *Presentation* layer access the information in *Data Access* layer in order to be available on user request. This access is realized using the via *External Services* that bind to the Data Access layer; this way the requests are restricted, authenticated and authorized. The mentioned UI components are used to display information and also accept user input.

The *Business* layer components represent the core functionalities of the system and encapsulate business logic. The whole business process is managed and guided by those routing rules and their available instruments like format transformations, validations, enrichment or reconciliation; all these are applied on financial instruments. This layer has access to information and stores information communicating to the *Data Access* layer.

The *Data Access* layer provides different ways of retrieving and sending information from and to the data sources (that may be internal – database or external) by using either direct access or service agents.

The cross cutting concerns identified are the following: Security, Audit and Exception Management. These will be addressed on most layers.



multi tenant::logical layered architecture

The diagram above presents the layered architecture of Whizzer, a new version of FinTPc adapted to be software as a service ready. The extension of FinTPc to the new SaaS version – Whizzer – is done following an approach of minimal changes in the application structure.

In order to share the application logic and the data across different tenants, Whizzer application is extremely configurable so it can respond to each specific needs. This includes either removing certain features or customizing business rules for each tenant.

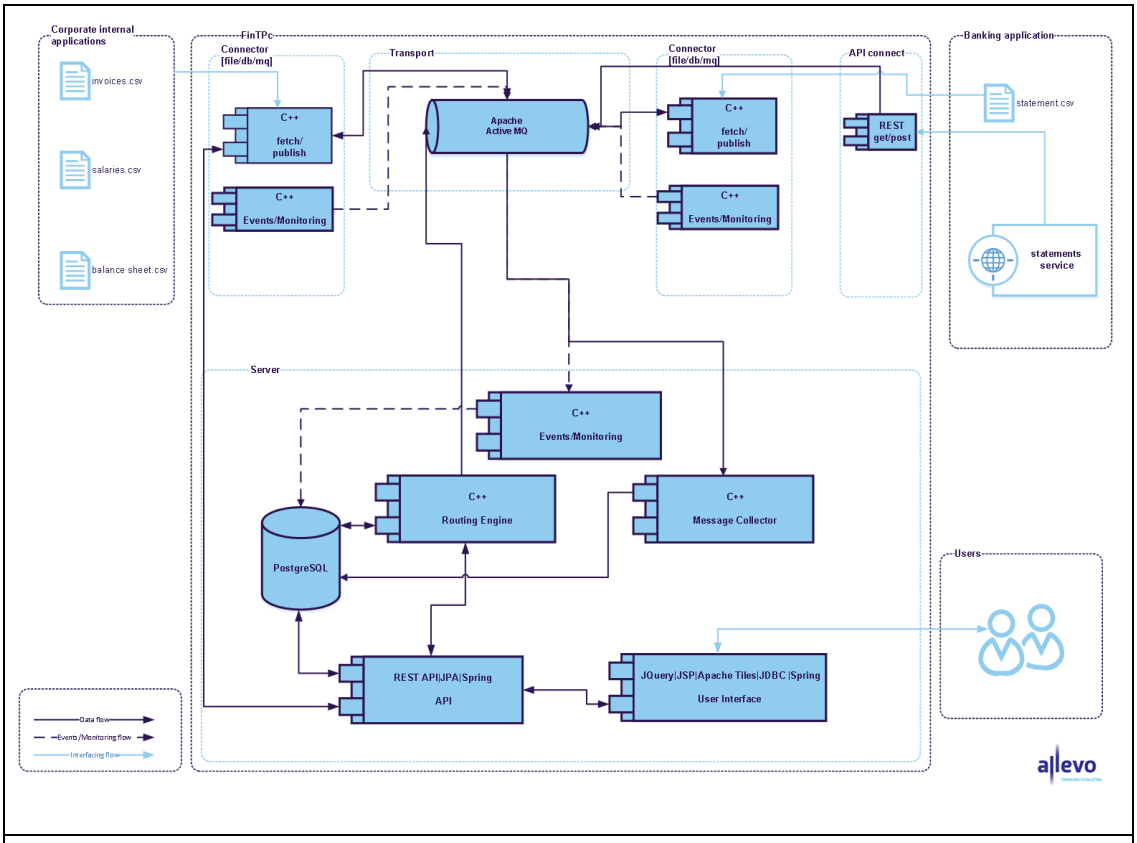
Also, the application ensures that the tenants only view and modify their own data using the User Interface and do not affect other tenant's data.

The architecture of Whizzer is based on using a shared database for all tenants, offering more ease of management. There may be the case that some tenants require a separate database due to privacy requirements; or it may be the case that some tenants are more suitable for dedicated instance due to their internal infrastructure of using multiple internal entities.

In a shared database, the tenants share common components of the application especially at the data layer with the degree of isolation provided at row level. The FinTPc authorization mechanism, covered by internal Rest API, ensures segregated access to own data for tenant users. After a user is authenticated, the authorization mechanism identifies which corporate entities can be accessed, based on provider administrator configurations, and accesses only that data in the database. So the reporting information loaded in the user interface is specific for given tenant. In order to provide data isolation

for each tenant, all the transactions in the application, that are data sources for building reports, have mandatory information the owner corporate entity – identified with a tenant.

3.1.2 Logical component architecture



components high level architecture

The upper diagram describes the interaction between all the application components when processing financial instructions.

There are specific *Connectors* configured to communicate either to *Corporate internal applications* side or to *Banking applications* side in the purpose of collecting data using the fetch options. Each connector may be configured to handle structured file formats or database records. Also different format transformations and encryption may be active at this stage.

FinTPc allows input from *Corporate internal applications* organized into structured and dedicated files storing different types of payments, received or issued invoices; multiple instructions may be stored into single files. The application also allows the same kind of input, but stored and structured into database tables.

FinTPc allows input from *Banking applications* organized into structured files or web services.

Events/Monitoring components are attached to connectors to register and transport processing events. All data collected by those connectors is enqueued into configured Active MQ queues.

The *Message Collector* component is monitoring those queues and then collects data in order to register it to specific structures into the FinTPc database.

Once they reached the database, the *Routing engine* component is starting the processing and routing phase, which may include business actions like: transformations, enrich, validations. There may be routing decisions that need to be taken by users.

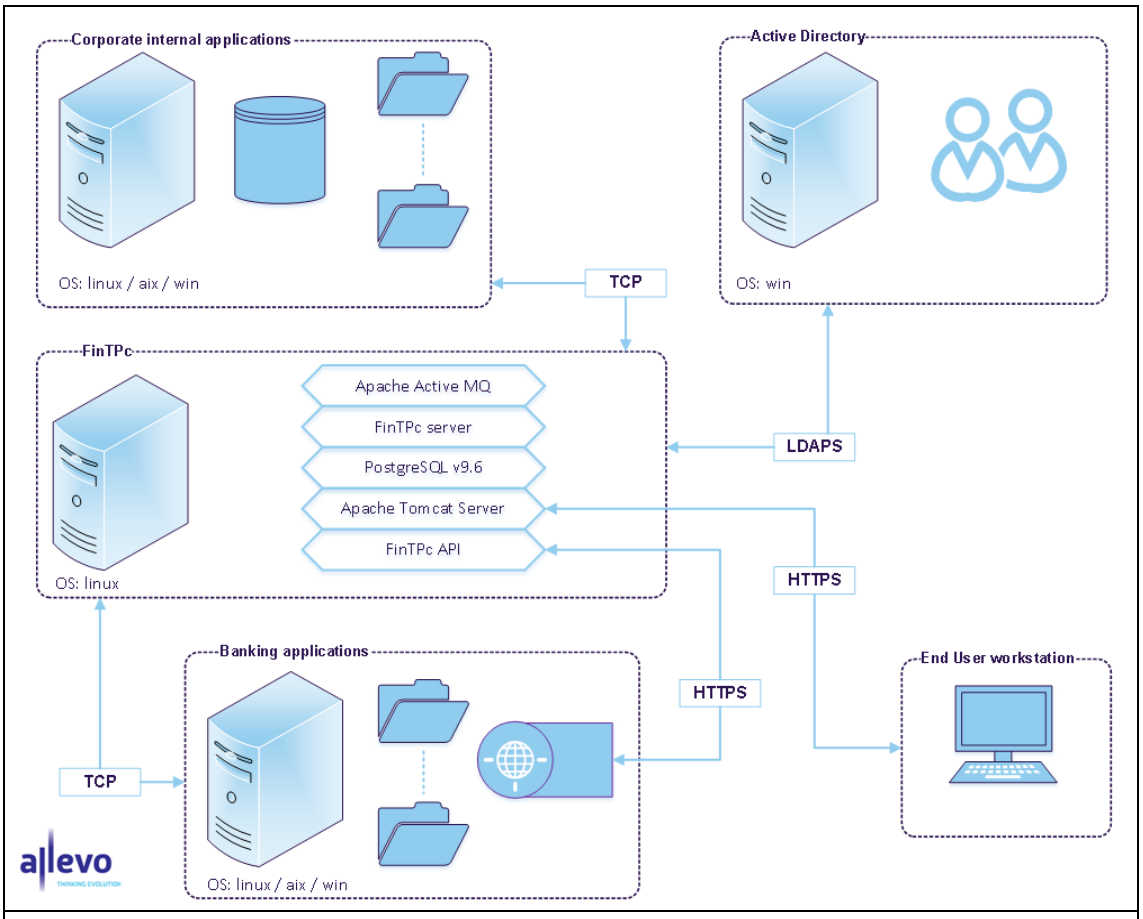
The users interact with the application via the *User Interface* component. They can have access to organized and structured financial instructions and also taking decision regarding their routing flow.

Component	**Connector**
Responsibilities	▪ Fetch / Publish data from / to the partner applications; ▪ Ensure financial data batching / de-batching; ▪ Embed data into an envelope that allows non-invasive tracking and audit; ▪ Perform validations; ▪ Encrypt data; ▪ Data format conversion.
Collaborators	▪ Back office applications; ▪ Banking applications; ▪ Transport component; ▪ Events/Monitoring component.
Notes	▪ The multiplicity of instances may depend on the format types of inputs managed (file / db / mq) and other business flow constraints; ▪ Can use local installed clients (activemq-cpp-library, odbc) to locally or remotely connect to defined interfaces; ▪ Ensures persistent end-to-end transactions; ▪ Ensure communication breaks detection along with connection restoring; ▪ Offers complex configuration options; ▪ Offers interoperability.
Component	**Events/Monitoring**
Responsibilities	▪ Collect process generated events (performance, and other details like size or amount) by collaborator components; ▪ Synchronize and publish events.
Collaborators	▪ Connector component; ▪ Transport component; ▪ Message Collector component; ▪ Routing Engine component.
Notes	▪ The multiplicity of instances depends on the number of connectors defined; ▪ Ensures persistent end-to-end transactions; ▪ Addresses the Audit concern.
Component	**Transport**
Responsibilities	Message transactional secure transportation.
Collaborators	▪ Message Collector component; ▪ Routing Engine component; ▪ Connector component; ▪ Events/Monitor component.

Notes	Apache Active MQ queues.
Component	**Message Collector**
Responsibilities	▪ Collect messages fetched by other connectors; ▪ Store those messages to the database and triggers the processing stage.
Collaborators	▪ Transport component; ▪ Routing Engine component; ▪ Database component.
Notes	▪ Ensures persistent end-to-end transactions; ▪ Ensure communication breaks detection along with connection restoring; ▪ Offers complex configuration options.
Component	**Routing Engine**
Responsibilities	▪ Parallel execution of routing jobs associated with the messages received from Message Collector component; ▪ Reconcile messages with internal/external confirmations specific to different business flows; ▪ Reconcile messages with others, based on defined business rules; ▪ Perform validations; ▪ Enrich message data; ▪ Format conversions; ▪ Drive data archiving.
Collaborators	▪ Database component; ▪ API component; ▪ Transport component.
Notes	▪ Ensures persistent end-to-end transactions; ▪ Ensure communication breaks detection along with connection restoring; ▪ Offers complex configuration options; ▪ Offers scalability.
Component	**Database**
Responsibilities	▪ Store configuration application data; ▪ Store financial instruments data; ▪ Store online archived data; ▪ Provides data manipulation instruments (triggers, stored procedures) according to defined business rules or user intervention.
Collaborators	▪ API component; ▪ User Interface component; ▪ Routing Engine component; ▪ Message collector component.
Notes	▪ Ensures data integrity and consistency.
Component	**API**
Responsibilities	▪ RESTful API; ▪ Enable 3rd party applications to communicate with the FinTP central repository of data.

Collaborators	▪ User Interface component; ▪ Routing Engine component; ▪ Database component.
Notes	▪ Offers interoperability.
Component	**User Interface**
Responsibilities	▪ Allow user authorization; ▪ Allow application configuration using friendly screens; ▪ Allow users to operate financial instruments; ▪ Provides advanced options for financial instruments searching and reporting; ▪ Ensures data confidentiality (limit data access) by specific rights.
Collaborators	▪ Routing Engine component; ▪ Database component; ▪ API component.
Notes	

4 Physical architecture



physical architecture

5 Data design

FinTPc application uses the following main data structures.

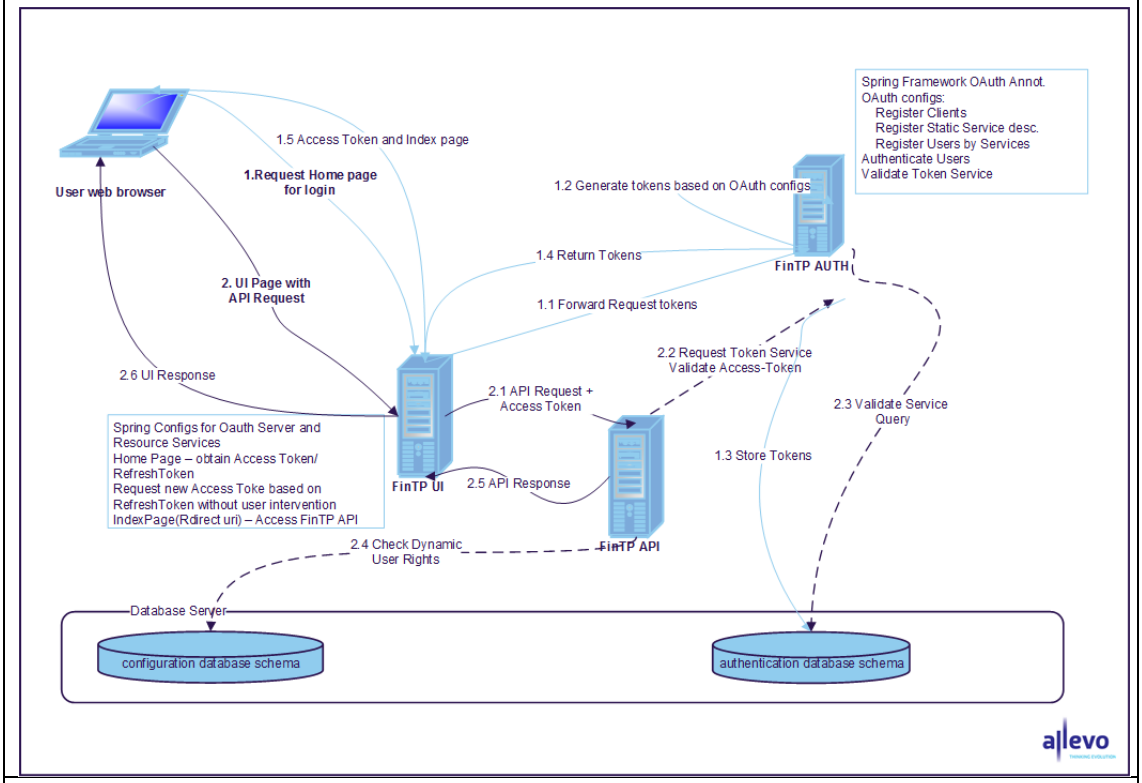
Data structure	Content	Benefits
Database	Stores structured information containing application configuration data, user configuration data, audit data and financial transaction data.	▪ Fast data access; ▪ Data concurrency support; ▪ Reporting; ▪ Secure access; ▪ Low data loss risk; ▪ Data backup and archiving support.
Message (messaging) [processing format]	Stores financial instructions data structured into a standard xml format, transported between components via transport layer.	▪ There are no direct connections between components; ▪ Security; ▪ Data integrity; ▪ Recovery support.
Routing Message (internal) [payload format]	Stores financial instructions data structured into specific xml format, depending on their type; stored in the database and used in internal communication or passing structured information outside the database, to other components; There are also intermediary formats that precede the final format transformation.	▪ Data integrity; ▪ Easy maintainability.
File	Stores financial transaction data structured into a standard format, used in communication to external systems.	▪ Bank compliance; ▪ Corporate compliance.

6 Authentication and Authorisation design

6.1 Authentication

Token based authentication works by ensuring that each request to a server is accompanied by a signed token which the server verifies for authenticity and only then responds to the request.

FinTPc uses JSON Web Tokens {header.payload.signature format}, being a safe way to represent a set of information between two parties.[2] This token stores the predefined application roles and user identification.

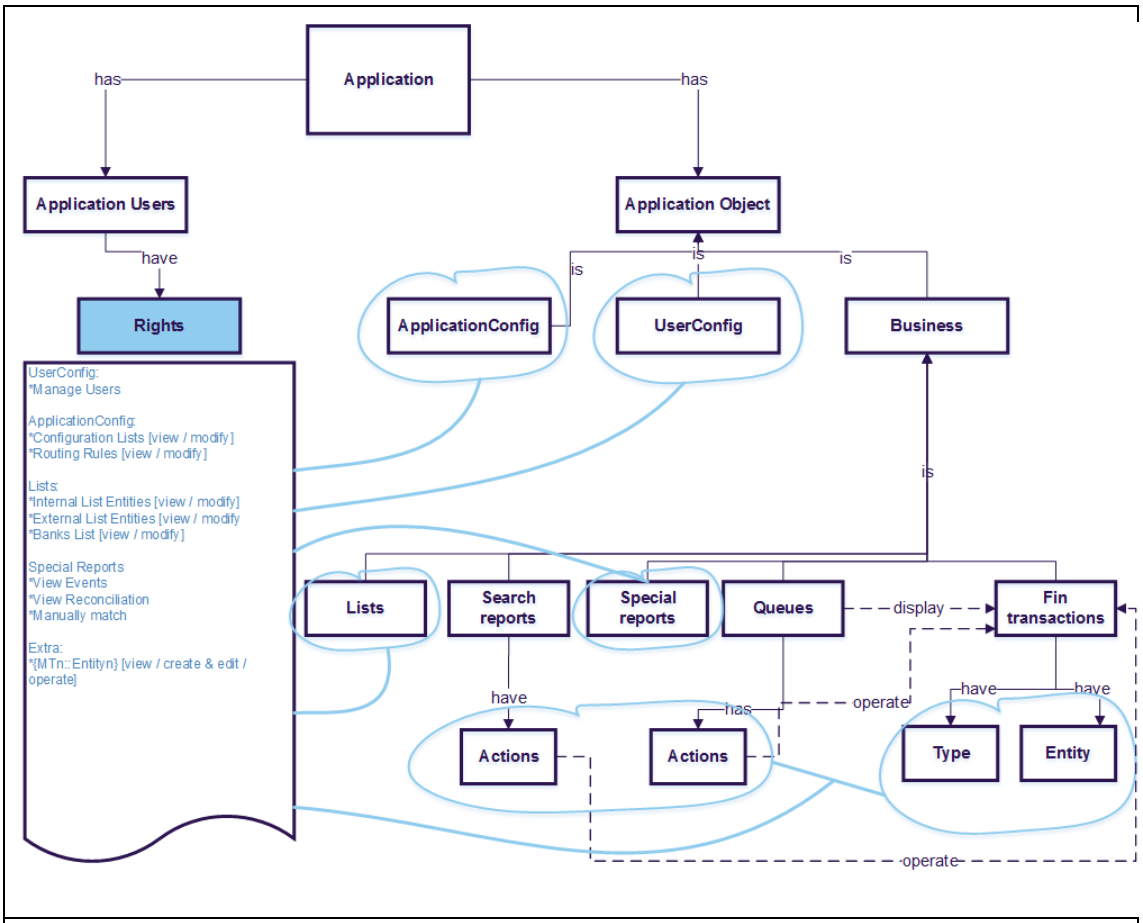


FinTPc:: Token Authentication design diagram

[2] RFC7519

6.2 Authorization

6.2.1 User administration



user roles design

The upper diagram describes the relationship between identified application concepts. In this context, the application has *Users* and also *Objects*. Application objects can be divided into three categories given their purpose and the business functionalities they offer. That being said, we identified the *Application Configuration*, *User configuration* and *Business* areas. The *Users* have restricted access to those areas (or their belonging objects).

The User Configuration does not have other subdivisions, therefor the access to it is designed by single right:

{Manage Users}.

The Application Configuration has subdivisions represented by lists or specific configuration areas that, according to business specifications, demand segregated access, also taking into consideration different type of access. Based on this, more rights are identified:

{Configuration lists::View}, {Configuration lists::Modify},

{Routing rules::View}, {Routing rules::Modify},

{Events::View}.

The Business has the following leafs: Lists, Search Reports, Special reports, Queues and Financial Transactions. Lists node has subdivisions represented by specific

lists that, according to business specifications, demand segregated access, also taking into consideration different types of access:

{*Lists*::View}, {*Lists*::Modify},

{Reconciliation::View}

The next remaining leafs have a complex relationship based on their sub-leafs. Both Queues and Reports have specific set of actions. The Financial transactions also have two major attributes that must become segregated access criteria: type and entity. Knowing the following relation: (*Financial transactions have two attributes and are exposed by Queues and Search Reports*) and (*Queues and Search Reports have Actions that operate Financial Transactions*) leads us to the following conclusion: A set of *entity – type* combinations must be defined and grouped and then associate to the specific set of actions in order to accomplish this demanded restricted access described in the business specifications document; for example:

{[(type1-entity1) + (type2-entity1)]::(View/Operate/Create & Edit)}

The Whizzer version of the application ensures that tenants view only their own data by using these mandatory roles assigned to user – a user can only view and modify data that belongs to given business entity (identified with a tenant). The Application Administrator role is given to the provider that manages application configurations and the tenant users.

User Roles Summary

Role category	**Role name**	**Role actions**	**Constraints**	**Applies to**
Application Administration	Configuration Lists	View		provider
		Modify	Can not be assigned if View is not assigned first.	
	Routing Rules	View		
		Modify	Can not be assigned if View is not assigned first.	
	Queues	View		
		Modify	Can not be assigned if View is not assigned first.	
	Events	View		
User Administration	Users	Modify		provider
Business Administration	Internal Entities List	View		provider
		Modify	Can not be assigned if View is not assigned first.	

Role category	Role name	Role actions	Constraints	Applies to
Reconciliation	Reconciliation	View		Tenant users
		Modify	Can not be assigned if View is not assigned first;	
Transactions	[(MT – Entity)n]	View		Tenant users
		Operate	Can not be assigned if View is not assigned first;	
		Create & Edit	Can not be assigned if View is not assigned first; The name generated for the group of (MT-entity) combinations must be unique; A (MT-entity) combination is unique among all roles; A notification is raised if there are (MT-entity) combinations not covered among defined roles.	
		Supervise	All the above, regardless of transactions type or entities.	

FinTPc provides a database script that registers a configurable user that has the User Administration role assigned into database structures. This user has to be already defined in the Active Directory structure and will further be able to synchronize all user to the application.

7 API design

The key design constraint that sets REST apart from other distributed architectural styles is its emphasis on a uniform interface between components. REST further defines how to use the uniform interface through additional constraints around how to identify resources, how to manipulate resources through representations, and how to include metadata that make messages self-describing. This ultimately leads to a simpler overall system architecture and provides more visibility into the various interactions.

The fundamental concept in any RESTful API is the *resource*. A resource is an object with a type, associated data, relationships to other resources, and a set of methods that operate on it. It is similar to an object instance in an object-oriented programming language, with the important difference that only a few standard methods are defined for the resource (corresponding to the standard HTTP GET, POST, PUT and DELETE methods), while an object instance typically has many methods.

Resources can be grouped into collections. Each collection is homogeneous so that it contains only one type of resource, and unordered. Resources can also exist outside any collection. In this case, we refer to these resources as singleton resources. Collections are themselves resources as well.

Collections can exist globally, at the top level of an API, but can also be contained inside a single resource. In the latter case, we refer to these collections as sub-collections. Sub-collections are usually used to express some kind of "contained in" relationship.

Resources have data associated with them. The richness of data that can be associated with a resource is part of the *resource model* for an API.

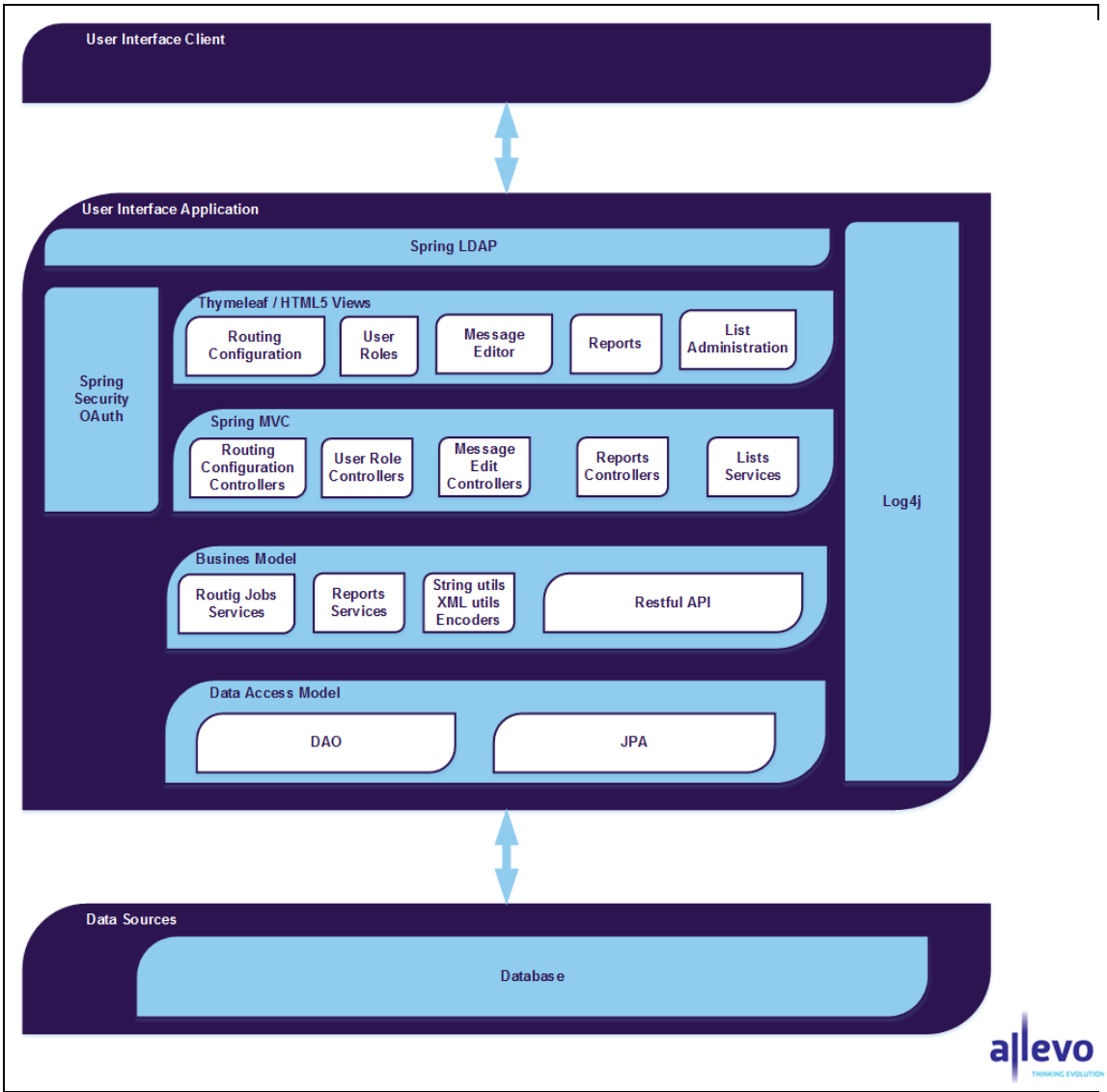
We further define the data that can be associated with a resource in terms of the JSON data model, using the following mapping rules:

1. Resources are modeled as a JSON object. The type of the resource is stored under the special key:value pair "_type".
2. Data associated with a resource is modeled as key:value pairs on the JSON object. To prevent naming conflicts with internal key:value pairs, keys must not start with "_".
3. The values of key:value pairs use any of the native JSON data types of string, number, true, false, null, or arrays thereof. Values can also be objects, in which case they are modeling nested resources.
4. Collections are modeled as an array of objects.

Key:value pairs are further referred as attributes of the JSON object. This use of attributes is not to be confused with XML attributes.

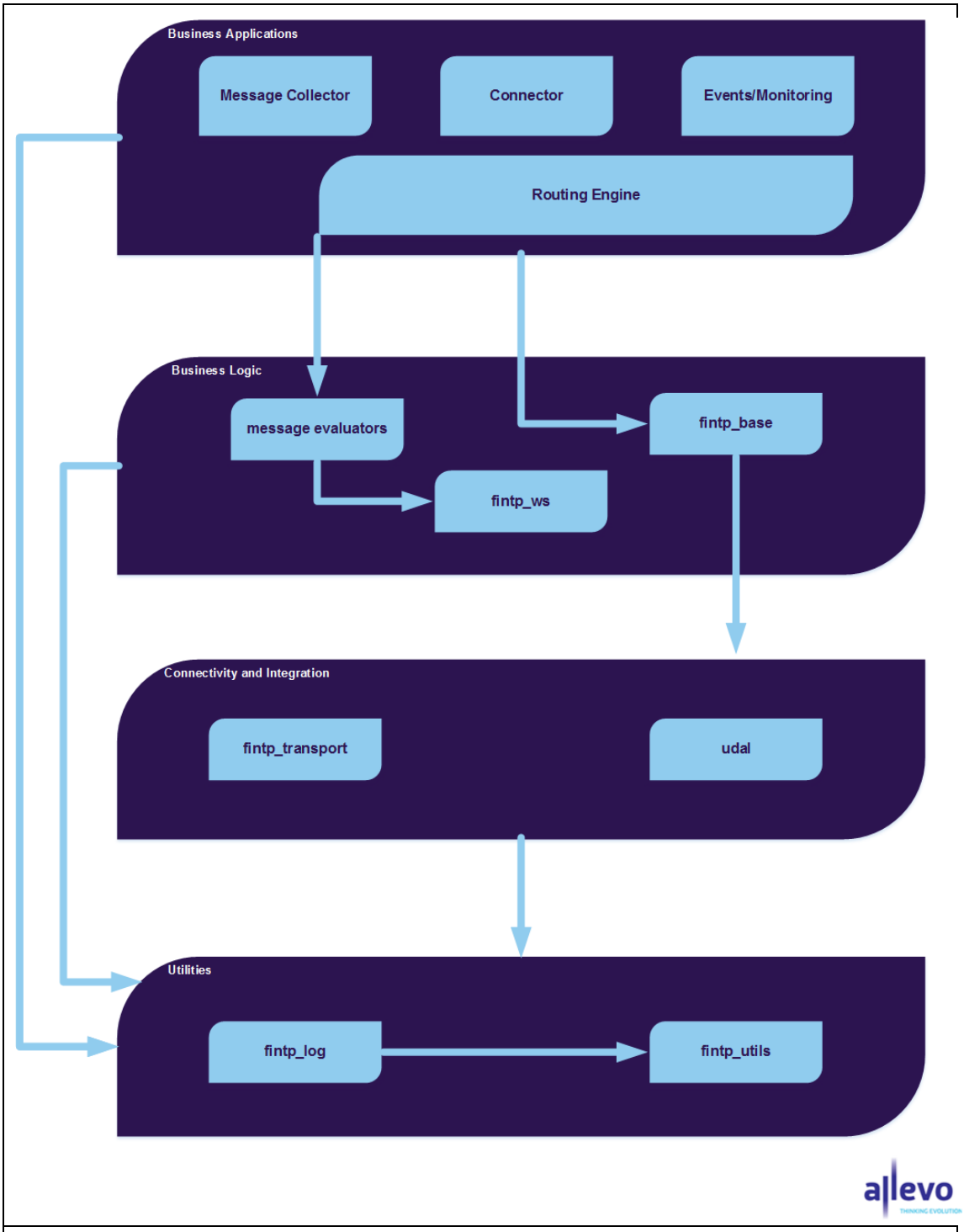
8 User Interface design

The diagram below describes the design layers of the User Interface, its components and their communication mechanisms.



User Interface design diagram

9 Server design



Server design diagram

The diagram above describes the FinTPc processing Server layers and how the communication and interaction happen between its components. Each of these components are describes below:

*Events/Monitoring – an a*pplication component responsible to collect process-generated events by each other server component. Then it drops the collected events to FinTPc storage database so these can be audited through the User Interface.

*Connector – an a*pplication component responsible to fetch/publish messages from/to external applications. Connectors can use local installed clients (activemq-cpp-library,

odbc, Oracle Instant Client, WMQ Client) to locally or remotely connect to other external applications interfaces.

*Routing Engine – the m*ain application processing component, whose purpose is to route messages according to routing actions defined in active *routing schemas*. The Routing Engine also reconciles processed messages with internal/external confirmations specific to business flow protocol and drives message archiving.

*Message Evaluators - p*lugin components which implement business specific behaviour of each processed message type. These components are used by the Routing Engine to accordingly interpret, report, and route the messages.

*fintp_ws - h*igh level library whose role is to provide an abstraction for response messages interpretation. The component is used by Routing Engine to commonly evaluate response messages.

fintp_base - high level library which implements FinTPc specific structures, commonly used by application components for their implementations.

*fintp_udal - l*ow level library which acts as an adaptor to provide a uniformly access to various RDBMSs. Every component of the solution use the library's API to access RDBMS.

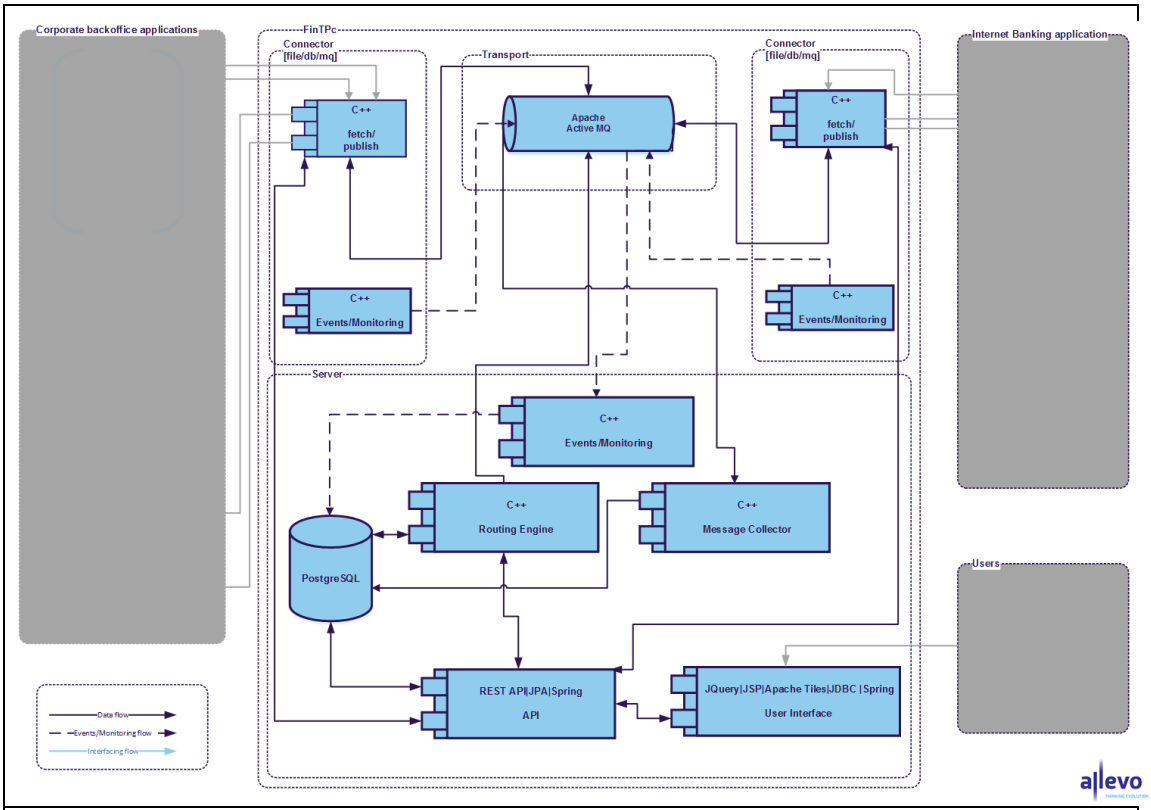
*fintp_transport - l*ow level library which acts as an adaptor to uniformly access transport messaging services. Every component of the solution use the library's API to access messaging service server.

*fintp_log - l*ow level library which acts as an adaptor to various logging systems. The library provides at least a file log system and is open for further log systems implementations.

*fintp_utils - l*ow level library which provides highly generic routines used like helpers by any other component.

10 Interfaces

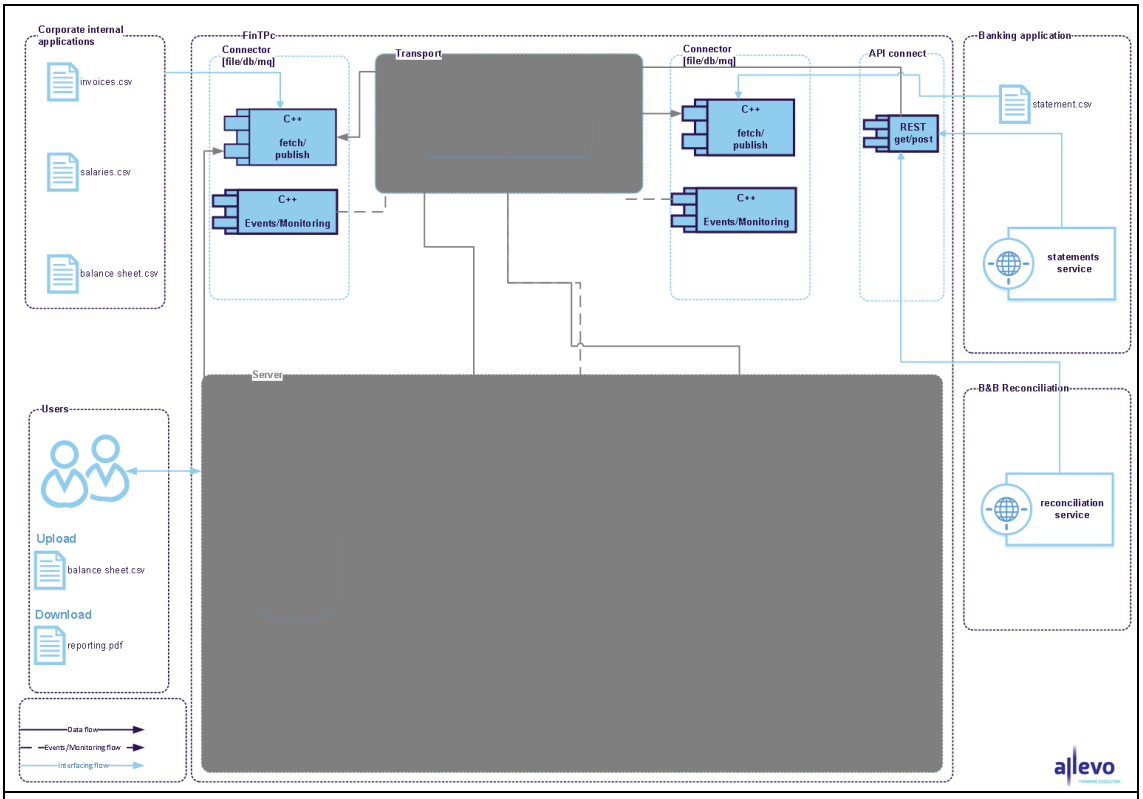
10.1 Internal interfaces



Internal interfaces

FinTPc component	Interfacing method	Interfacing components	Description
Connector	Queue messaging	Message Collector	via Transport layer
Connector	cURL	API	
Routing Engine	Stored procedures, inline queries	Database	
Routing Engine	Queue messaging	Connector	via Transport layer
Message Collector	Stored procedures	Database	
Message Collector	Queue messaging	Connector	via Transport layer
User interface	via API	Database	
API	JPA	Database	

10.2 External interfaces



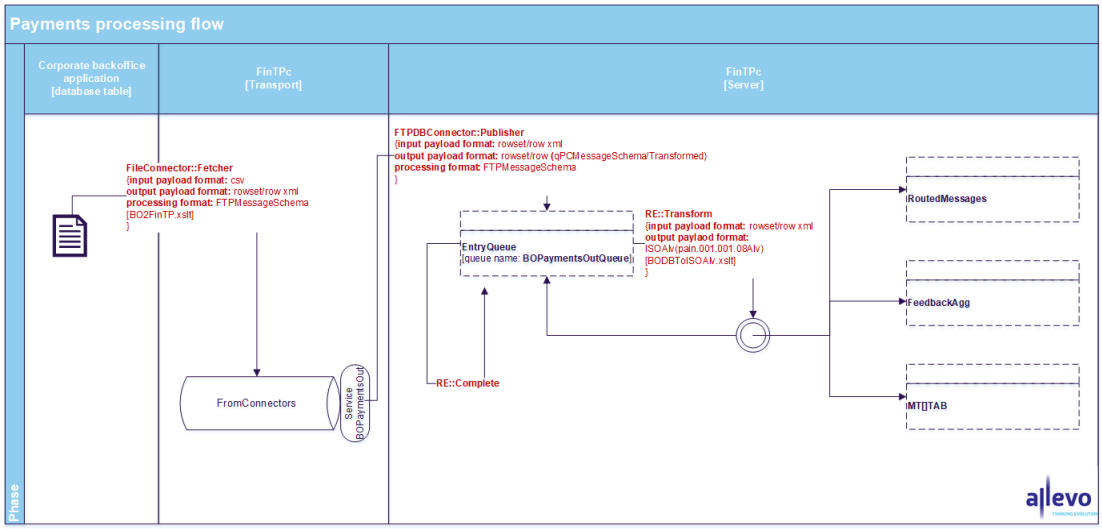
External interfaces

External interface	Interfacing method	FinTPc component
Corporate internal applications	csv files – specific for each financial transaction type or report;	Dedicated Connector – fetcher.
Banking applications	csv format files; web service calls.	Dedicated Connector – fetcher / API connect
Users	csv files – specific for each report.	User Interface
B&B reconciliation engine	web service calls.	API connect

11 Functionalities

11.1 Money Flow Automation

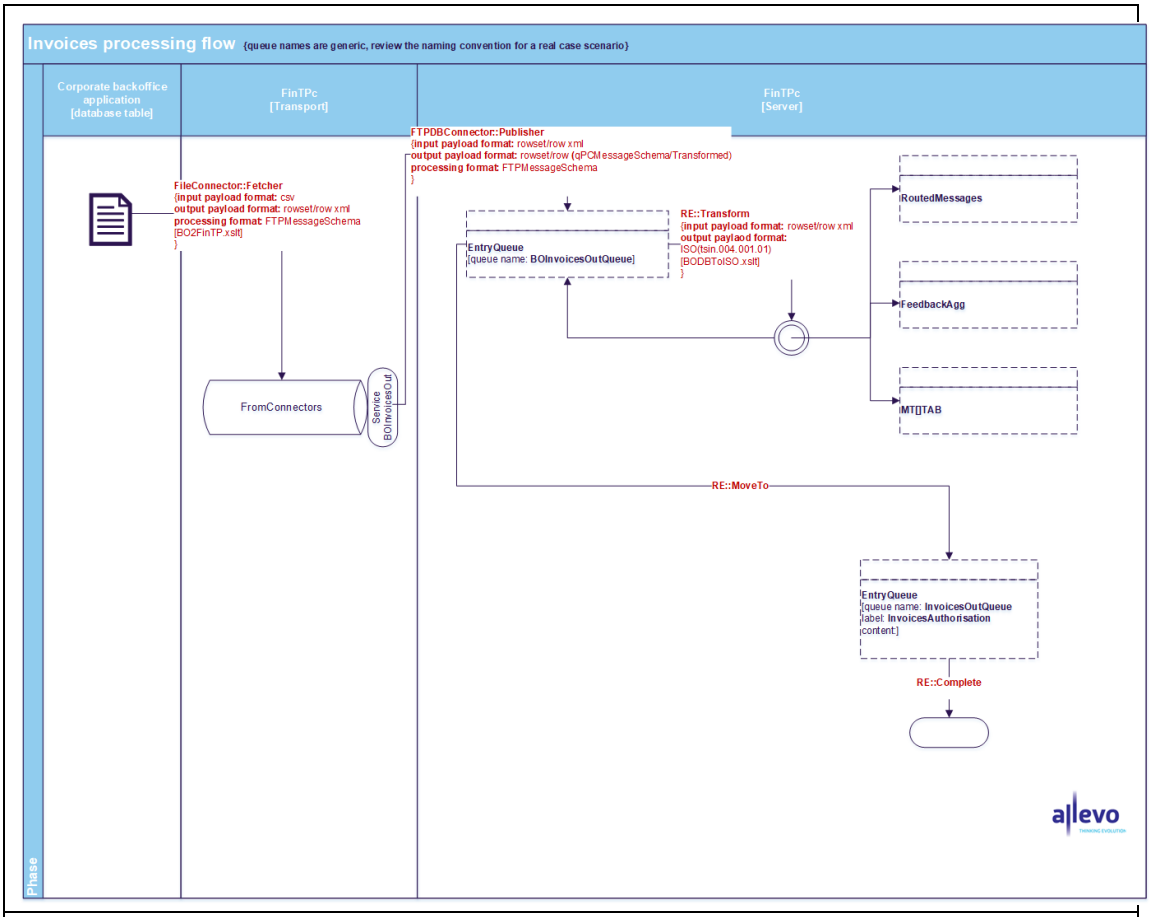
11.1.1 Salary Payments



Salary payments :: processing flow diagram

Salary payment information is fetched from corporate internal applications, processed, structured, and then ready for building reports based on that data.

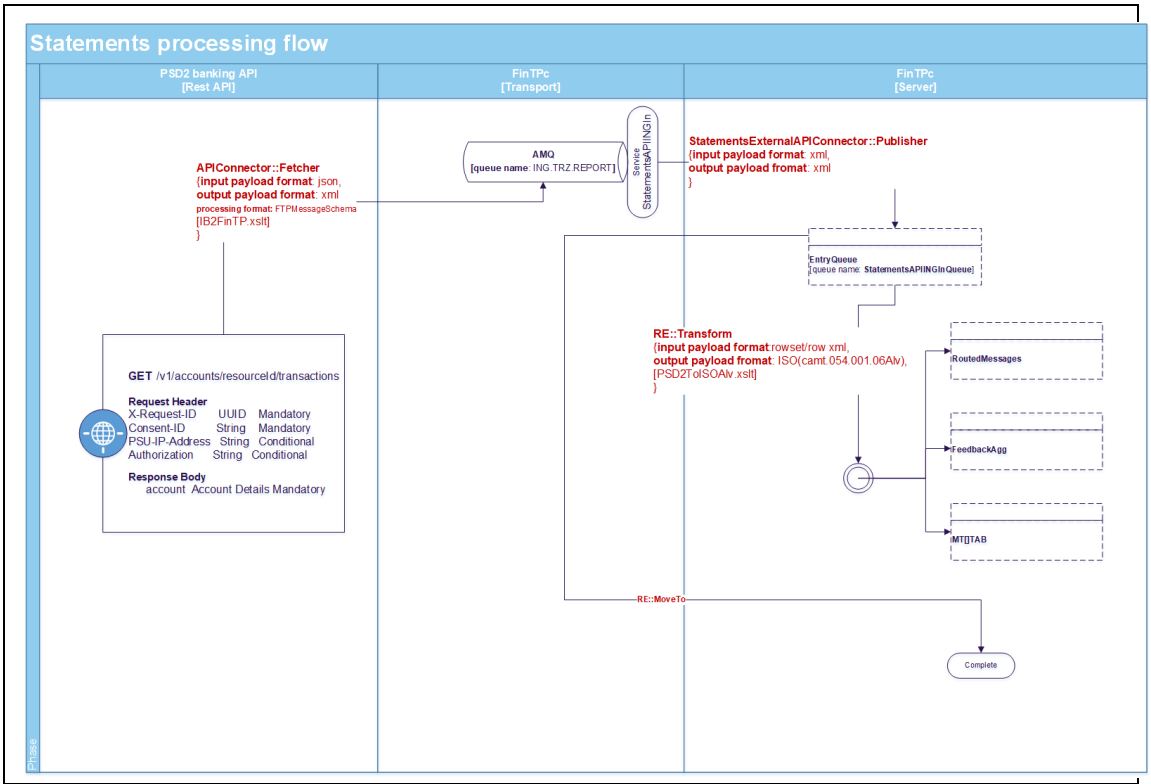
11.1.2 Invoicing



Invoices :: processing flow diagram

It enables the processing of invoices sent and received by a company; Invoices raw data is fetched from internal application, processed, structured, an then ready for building reports based on that data.

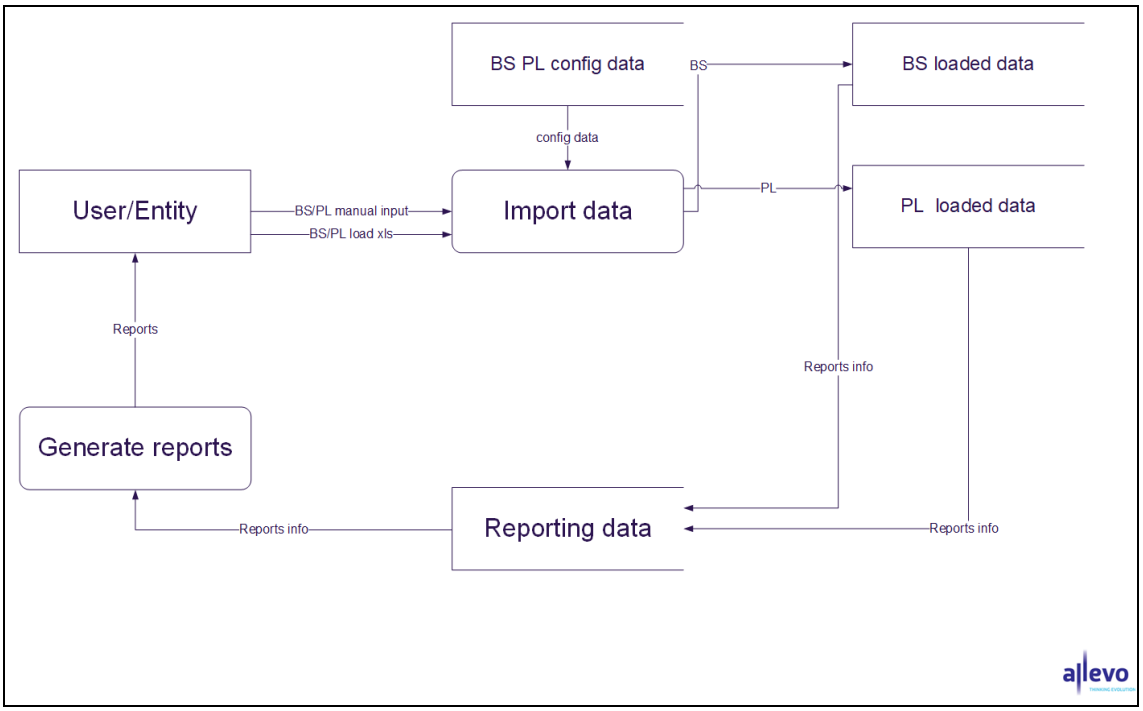
11.1.3 Statements



Statements::processing flow diagram

It enables the processing of bank statements; Invoices raw data is fetched from banks, via API, processed, structured, an then ready for building reports based on that data.

11.2 Balance Sheet

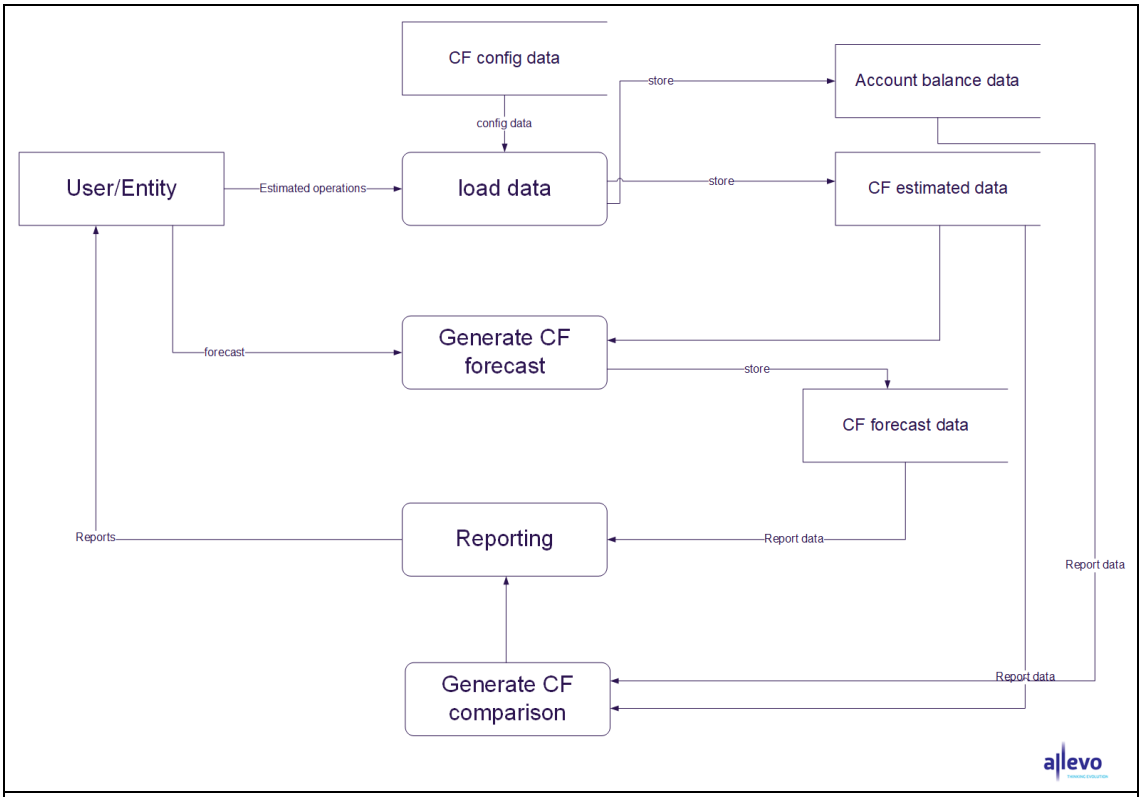


Balance sheet :: data flow diagram

This feature uses Balance Sheet document information and generates very structured, easy to understand reports such as Balance Sheet Report and Profit & Loss Report, both numerical and graphical view.

11.3 Cash Reporting

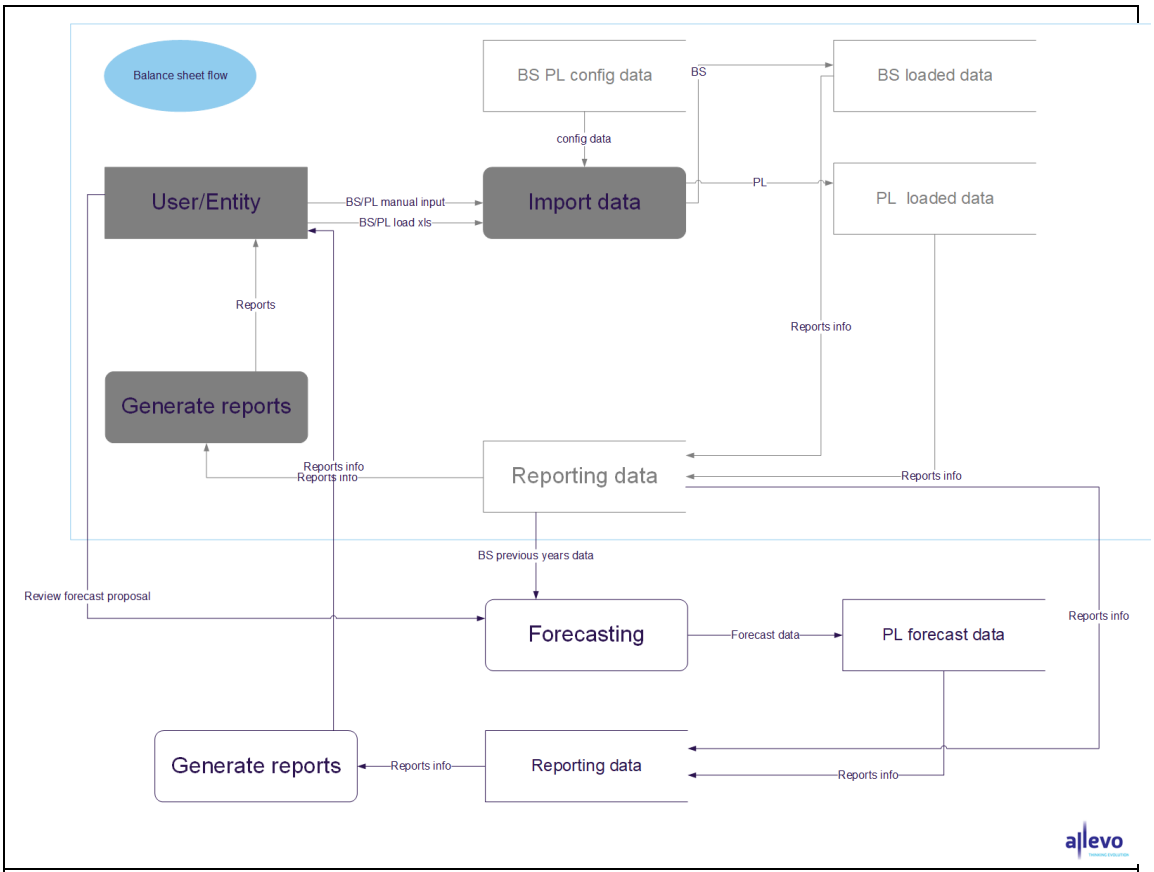
11.3.1 Cash flow forecasting



Cash flow forecasting :: data flow diagram

The cash flow forecasting offers an estimation for the future days/ months, based on current status (balance of accounts) and on the estimated expenses (based on balance sheet data) and on the invoices (issued and not paid, received and not paid).

11.3.2 Balance sheet forecasting



Balance sheet forecasting :: data flow diagram

Balance Sheet forecasting offers an estimation for the future Balance Sheet indicators based on realised Balance Sheet information and on the estimated modification rate.

12 Technology selection

The distribution model of this project is open source. Therefor the major architectural and technological constraint is represented by the compliance of FinTPc code and any other embedded product or library with GPL v3 license model. The design and implementation stage will include also advanced scanning procedures in order to be able to certify this license compliance.

Our implementation of the application is based on integration with following products, as seen in the architecture diagrams: Apache Tomcat, Apache MQ and Postgresql. However, these prerequisites are not fully mandatory. The design of the application should allow integration with different other technologies like IBM WebSphere MQ, IBM WebSphere Application Server, Jboss, Weblogic, Oracle database and others given a convenient migration and configuration.