Stanislavs Rubens

StaRub@gmail.com

Software Architecture Document

**TABLE OF CONTENTS**

SECTION 1 3

1. Introduction 3

1.1. Purpose 3

1.2. Scope 3

1.3 Definitions, Acronyms and Abbreviations 4

1.4 SAD Overview 4

SECTION 2 5

2. IBS/DAS Analysis 5

SECTION 3 6

3. SAD View definitions: 6

SECTION 4 7

4. Architecture goals and constraints 7

Early Development stage 7

Mid-term development stage 8

4.2. Logical view diagram 9

4.3. Development view diagram, using domain driven design definitions 10

4.4. Data model 11

SECTION 5 12

5. Most important use case realizations 12

5.1. Process model 12

5.2. Use case model 12

SECTION 6 13

6.1. Deployment process 13

# SECTION 1

## 1. Introduction

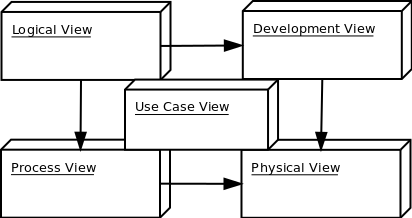
This document provides architecture description of Internet Banking Solution (IBS) with Data Analytics System (DAS) as part of it's backend.

It explains customer online interaction with IBS, reporting functionality of DAS and integration with existing software landscape.

### 1.1. Purpose

This document provides a high-level description of the goals of the architecture, the use cases support by the system, and architectural styles and components that have been selected to best achieve the use cases.

Structure of this document is based on the "4+1" architectural view model.



### 1.2. Scope

The scope of this SAD is to depict architecturally significant aspects and project requirements, such as

Functional:

* DAS data collection from different sources;
* DAS reporting engine based on collected data;
* At least 2 different reports available.

Non-Functional:

* All system/external communication should be secured;
* System must be responsive and perform well under load;
* Different type of data must be used for reporting.

### 1.3 Definitions, Acronyms and Abbreviations

|  |  |
| --- | --- |
| SAD | - Software Architecture Document; |
| IBS | - Internet Banking Solution - part of the system to be developed, provides customer interaction model; |
| DAS | - Data Analysis System - part of the system to be developed, provides data collection, transformation, analysis and reporting; |
| Java | - programming language to be used to implement the IBS/DAS; |
| Apache Maven | - Software configuration management tool, essential for building and assisting in system deployment; |
| Spring Framework | - Application development framework, designed to aid and facilitate modern web application development; |
| Spring Boot | - part of Spring Framework, allows rapid prototype development and self-contained system design; |
| MongoDB | - NoSQL data storage solution, schema-less database suitable for rapid and flexible collection of various types of business data; |
| Apache Camel | - Enterprise Integration Patterns framework, routing and mediation engine designed to facilitate integration with legacy software systems; |
| Apache Hadoop | - Distributed data analysis framework, to be integrated in DAS if complex analytics would ever be required; |
| Vagrant | - Self-contained environment, complete software image with IBS/DAS and all required dependencies preconfigured, for easy scaling and deployment; |
| API | - Application Programming Interface - a set of functions and procedures that allow the creation of applications which access the features or data of an operating system, application, or other service; |
| REST | - REpresentational STate transfer - stateless, client-server, cacheable communications protocol, typically used in modern web applications; |
| PAAS | - Platform as a service (PaaS) is a category of cloud computing services that provides a platform allowing customers to develop, run and manage Web applications without the complexity of building and maintaining the infrastructure typically associated with developing and launching an application; |
| Heroku | - Cloud PAAS, suitable for deploying early IBS/DAS prototype; |
| JUnit | - Unit-testing framework for project testing; |
| Mockito | - Object mocking framework for project testing; |

### 1.4 SAD Overview

**Section 2:** IBS/DAS analysis and reasoning behind each architectural component;

**Section 3:** "4+1" model view usage and description;

**Section 4:** Architectural goals and constraints of the system;

**Section 5:** Most important use case realizations;

**Section 6:** System deployment process;

# SECTION 2

## 2. IBS/DAS Analysis

Modern online services are customer-centric and require flexible, dynamic approach for handling requests.

They should record as much varied business data as possible, to better identify customer needs, thus providing better service experience to customer.

They should be scalable,self-contained, fail-safe and able to handle substantial network load.

They should be easy to develop, deploy and maintain, have flexible release cycle to better adapt for business needs.

They should be easily testable, with multiple environments to launch tests.

Based on the above, microservice-based architecture was chosen as best candidate - it allows rapid prototyping, deployment and scaling, retaining the possibility to evolve into modular application architecture if needed.

Because of the dynamic nature of business data, NoSQL solution was chosen over the relational one. Schema-less datastore may lack in complex analytics (i.e. OLAP cubes) but many solutions are readily available on the market to ease the process (i.e. Apache Hadoop).

The following architectural components were chosen for the system:

1. **Programming language**

Java 1.8 (1.8 was chosen because it's production stable and contains many features which would ease the development (streams, lambdas, new date API etc.))

1. **Core - Spring Framework**

Spring Framework is stable, mature and robust product, which is well-documented and known to programming communities worldwide.

1. **Microservice core - Spring Boot**

Being part of Spring Framework, Spring Boot is good for rapid application prototype development, adding many useful features out-of-the-box, with minimal necessary configuration.

1. **Datastore - MongoDB**

MongoDB is document-based NoSQL database, suitable for handling large amounts of dynamic data. It's fail-safe via it's sharding mechanism, has established community and good support, and it's aggregation pipeline and mapreduce techniques provide good reporting possibilities.

1. **EIP framework - Apache Camel**

Camel routing and mediation engine would ease integration with existing systems, allowing easy usage of any EIP we might need (message delivery, transformation, failovers etc.)

1. **SCM - Apache Maven**

Maven is proven SCM framework, with very good plugin support.

1. **Provisioning - Vagrant**

Could be useful when scaling system in case of substantial load increase.

1. **Complex analytics - Apache Hadoop**

Could be useful in the future if complex analytics would be required.

1. **PaaS - Heroku**

Good for developing small application prototypes, for production-grade application I would suggest Amazon EC2 or similar.

1. **Testing**

JUnit/Mockito would provide comprehensive coverage of testing required functionality on backend.

# SECTION 3

## 3. SAD View definitions:

1 - **Logical view**: The logical view is concerned with the functionality that the system provides to end-users.

2 - **Development view**: The development view illustrates a system from a programmer's perspective and is concerned with software management.

3 - **Process view**: The process view deals with the dynamic aspects of the system, explains the system processes and how they communicate, and focuses on the runtime behavior of the system. The process view addresses concurrency, distribution, integrators, performance, and scalability, etc.

4 - **Physical view**: The physical view depicts the system from a system engineer's point of view. It is concerned with the topology of software components on the physical layer, as well as the physical connections between these components.

5 - **Use case view**: The use case view describe sequences of interactions between objects, and between processes. It's used to identify architectural elements and to illustrate and validate the architecture design. It also serves as a starting point for tests of an architecture prototype.

# SECTION 4

## 4. Architecture goals and constraints

The following goals and constraints are identified in first 2 stages of system development (early, mid-term)

1. - Server side
2. - Persistence
3. - Security
4. - Integrations
5. - Client side
6. - Performance
7. - Reliability/Availability
8. - Testing

#### Early Development stage

1. - Server side (early development stage)

IBS/DAS system will be hosted in cloud PaaS environment (Heroku), running as standalone jar on embedded application server.

2. - Persistence (early development stage)

IBS/DAS will be connected to single MongoDB database server.

3. - Security (early development stage

No authorization/authentication mechanism is planned at this point.

4. - Integrations (early development stage)

No integration with legacy system is planned at this point.

5. - Client side (early development stage)

IBS/DAS would provide RESTful API for reporting data retrieval.

6. - Performance (early development stage)

There is no particular constrains related to system performance.

It is anticipated that the system should respond to any request well under standard database and web server script timeouts (5 seconds), also system performance can depend on available hardware, PSU network and internet connection capabilities.

In addition, upload / download times can depend on data size which in turn depends on user input. Therefore, actual performance can be determined only after system deployment and testing.

7. - Reliability (early development stage)

Heroku would provide required availability, considering necessary downtimes as per free usage agreement.

8. - Testing (early development stage)

Backend regression/smoke tests must be implemented in order to achieve acceptable code quality.

#### Mid-term development stage

1. - Server side (mid-term development stage)

IBS/DAS system will be hosted in SaaS environment of choice (Heroku Enterprise, Amazon AWS/EC2 etc.) packaged as Vagrant box.

2. - Persistence (mid-term development stage)

IBS/DAS system will be connected to multiple MongoDB database servers using sharding solution.

3. - Security (mid-term development stage)

Authentication/authorization mechanism will be implemented, according to needs (token-based, 2-factor etc.)

All communication with client would be performed via HTTPS protocol.

4 - Integrations (mid-term development stage)

Additional integration would include using Apache Camel for legacy systems, complex reporting would be achieved with Apache Hadoop.

5 - Client side (mid-term development stage)

IBS/DAS would provide comprehensive UI/UX experience, with adaptive design for viewing on multiple devices (phones, tablets, dekstops etc.) I would suggest using AngularJS 1.x as frontent framework, in conjunction with ReactJS for overcoming rendering bottlenecks.

6 - Performance (mid-term development stage)

After series of stress-testing, additional scaling would be determined if needed.

To scale load horizontally, session-persisting mechanism would be introduced, clustering of system and database servers would be required.

7 - Availability/Reliability (mid-term development stage)

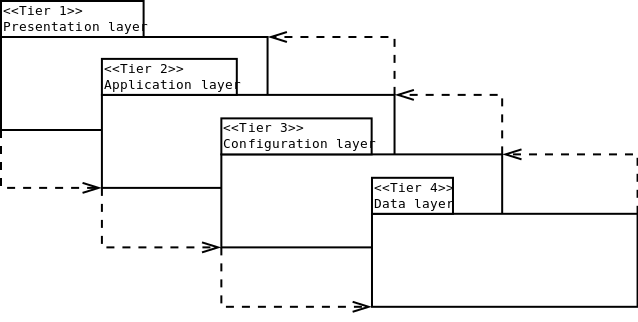
DevOps team would be required for additional monitoring of application health. Metrics and healthchecks could be introduced (i.e. SpringBoot Actuator)

8 - Testing (mid-term development stage)

Frontend regression/acceptance tests must be added, backend test suites further expanded to achieve acceptable code coverage.

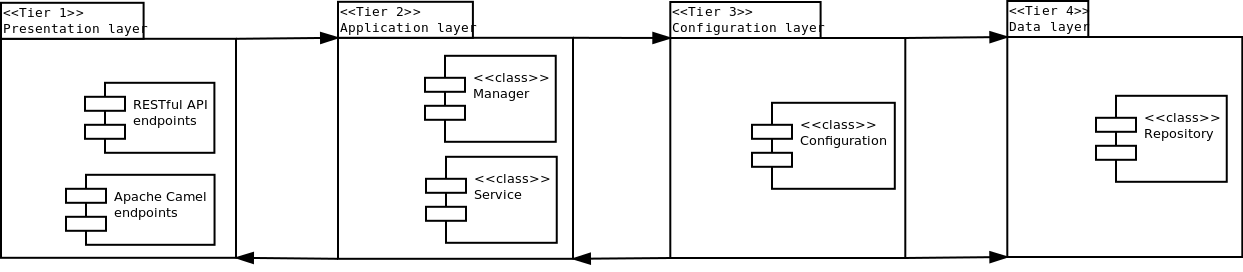
### 4.2. Logical view diagram

IBS/DAS system is divided into layers based on the N-tier architecture:



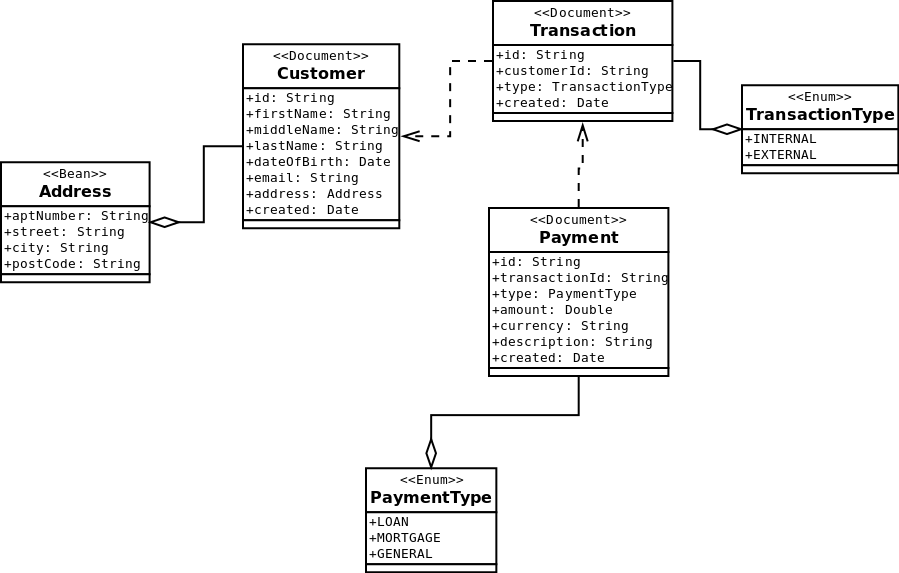
* Presentation layer is responsible for receiving payload requests from system users and issuing particular payload responses via RESTful API.
* Application layer is where core of the system resides, along with reporting engine and legacy system integration framework.
* Configuration layer is responsible for configuring system to operate under different work environments, contains crucial system properties and flexible component configurations. It is especially useful for introducing different level of service abstractions, in order for the system to be fully testable. TEST, DEV, RC, and PROD configurations are supported, in accordance with the proposed Git Flow branchning model.
* Data layer is where all application/reporting services and repositories reside, to provide access to persistent data.

### 4.3. Development view diagram, using domain driven design definitions



* Presentation layer consists of RESTful API endpoints, for service interaction with customers. A set of Camel endpoints is also defined providing communication with legacy systems.
* Application layer consists of 'Manager' type classes responsible for aggregating 'Service' type classes, providing communication with REST/Camel endpoints and core application logic.
* Configuration layer consists of 'Configuration' type classes responsible for system behavior in different environments, i.e. database connection properties in TEST, DEV, RC, PROD environments etc.
* Data layer consists of 'Repository' type classes responsible for domain object <-> raw data transformation, storage and retrieval.

### 4.4. Data model

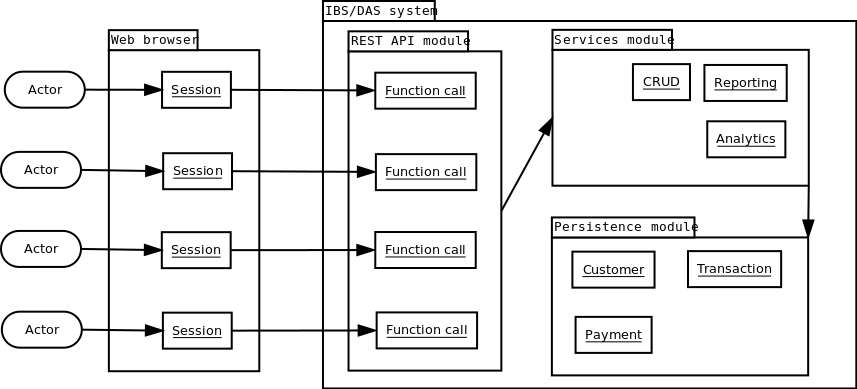


* Preliminary data model consists only of essential entities, required for early reporting functionality.
* 'Customer' object represents human customer, with required credentials.
* 'Address' object represents 'Customer's' physical address.
* 'Transaction' object represents action started by 'Customer' for every operation with IBS/DAS system.
* 'Payment' object represents certain action issued by 'Customer' during 'Transaction' e.g. paying bills, or mortgage repayment.
* 'TransactionType' object represents type of 'Transaction'.
* 'PaymentType' object represents type of 'Payment'.

# SECTION 5

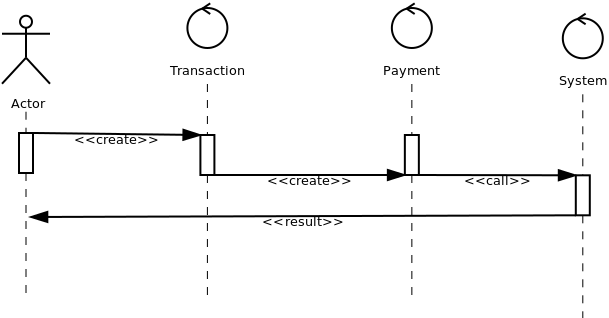
## 5. Most important use case realizations

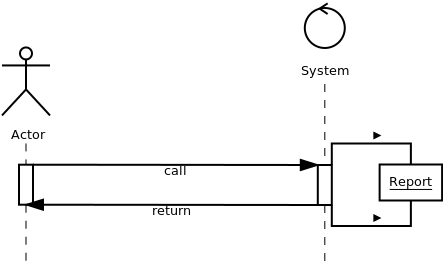
### 5.1. Process model



1. Actor interacts with IBS/DAS system using web browser. During his session he's able to perform function calls to REST API module in order to perform payments in transaction context.
2. Actor interacts with IBS/DAS system using web browser. During his session he's able to retrieve reports about payments, and customers.

### 5.2. Use case model

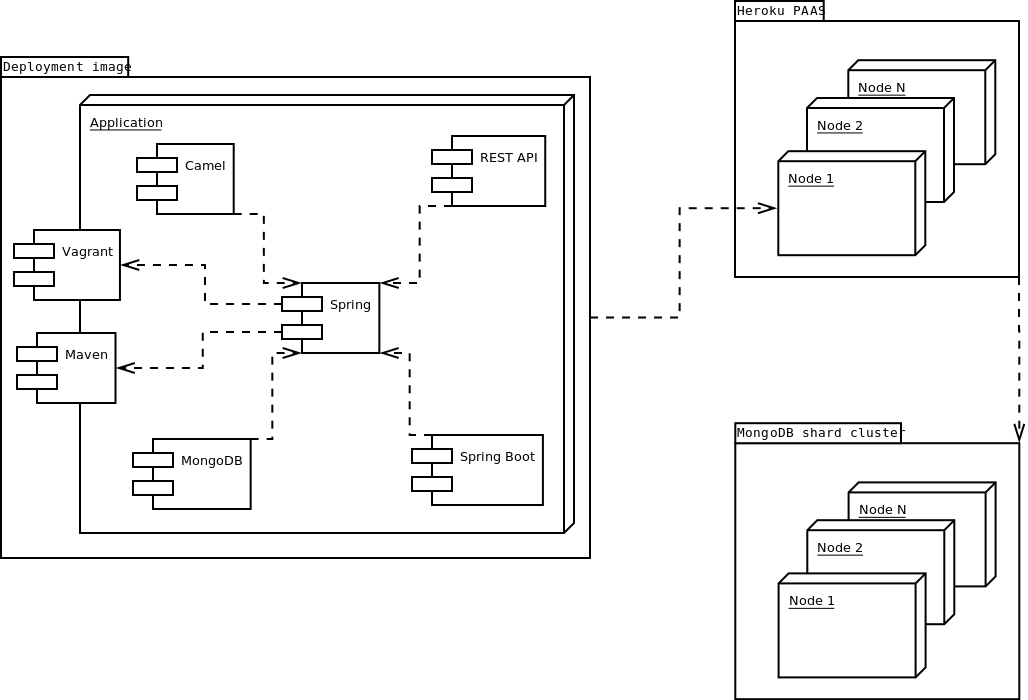
  
Actor performs payment operation in IBS/DAS system.



Actor performs report query in IBS/DAS system.

# SECTION 6

## 6.1. Deployment process



\*In the early development stages Vagrant boxing is not needed, but is considered for scaling issues.