bfsi consulting
enabling expertise

Mobile Application for Field Agents

Architecture & General Design Specification
January 2014

Status: Draft
Version 0.2

Bfsi Software Consulting Pvt. Ltd.
Bangalore, India

Document Control

Author: Vidyut Kapur		
Created on : 6-Jan-2014	Revision No :	
Updated by :	Reviewed by :	Approved by:
Updated on :	Reviewed on :	Approved on :

Version No.	Date	Author	Reviewed By	Status	Comment
0.1	31-Jan-14	Vidyut		Draft	
0.2	11-Feb-14	Vidyut		Draft	

Table of Contents

1 INTRODUCTION

1.1 Organization of the Document

✓ The first chapter of this document gives an introduction with brief background and organization of the document.

✓ The second chapter describes the main system components and covers the overall architecture of each component

✓ The third chapter contains details of some specific technical features of the system. These details, while not providing a comprehensive design, serve as a guide when designing the system.

1.2 Intended Audience

This document is meant for an internal audience.

- Developers can use it to get an overall perspective on the system
- Designers can use it to include the contents of Chaapter 3 in their design effort,
- Marketers can base some of their product collateral on it and
- Senior management understand the technical approach and the main components by reading this.

1.3 Exclusions

✓ This document is not intended to be a functional specification document. While the functional architecture sections briefly refer to the system functionality, a complete description of the system functions can be found only in the FS.

✓ This document describes the overall technical architecture of the system and outlines the design approach to some of the technical aspects of the system. However, it is ***not a design document***. It is intended to be a precursor to a full fledged design document which will follow.

1.4 Acronyms and Abbreviations

MBS	Mobile Banking System
IBS	Intermediate Banking System
CBS	Core Banking System
Bfsi	Bfsi software consulting pvt. Ltd.
LOV	List of Values
STP	Straight Through Processing

1.5 Conventions Used

- ✓ At various places in the document, links to appropriate pages on various web sites have been provided. These links are refer to the original information which amplifies the statement being made and are a mechanism to keep the document short by not reproducing that information here. Readers interested in greater detail can follow those links and others can simply read on - the document is intended to be under stood even if these links are not followed.

- ✓ At times these links refer to snippets of code which can be used by developers to understand usage of a particular design mechanism being proposed. It is not necessary that these snippets provide the best coding example. The most appropriate design/coding should e decided during the detailed design.

- ✓ 

At various places in the document, coding guidelines have been mentioned (preceded by the sign shown here). These guidelines have been formulated so that the architecture is consistently followed and its benefits accrue to the system. At any point in the development, if developers feel they need to deviate from a particular coding guideline, they should discuss and change the architectural principle from which that guideline follows rather than simply violate it.

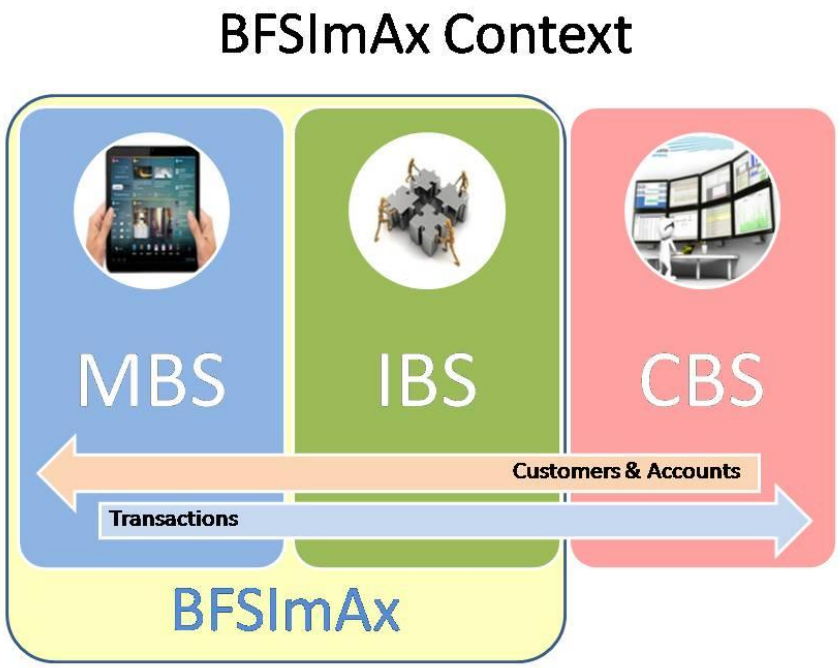
2 SYSTEM ARCHITECTURE

This chapter describes the main system modules and the overall architecture of each module.

2.1 Main System Components

The primary purpose of the BFSImAx system is to enable agents to do banking transactions with customers even though they may have little or no connectivity to the bank's centralised IT infrastructure while doing so. BFSImAx enables this via its Mobile Banking System, which runs on any Android based tablet or smartphone and allows off-line transactions.

However, the Mobile Banking System is not sufficient to enable the agent transactions. A system is required to manage the agents and devices which are out in the field, to co-ordinate their activities with the bank and its customers. This mandates the existence of a centrally deployed Intermediate Banking System which bank officers use to manage agents, devices and the transactions done by those agents.



When offline transactions are being done in the MBS (which is deployed on several devices) and there is an IBS (which is centrally deployed) to manage those transactions, then there exists a need to synchronise the data flow between these two system. Hence a major component of BFSImAx is the synchronization between the MBS and the IBS.

The CBS is the gold repository of all customers and accounts, including the balance, status and transaction history of the account. This data needs to flow to the MBS, just as the transactions done in the MBS need to be sent to the CBS. The IBS performs the task of intermediating this exchange, thereby necessitating the development of an interface between the IBS and the CBS.

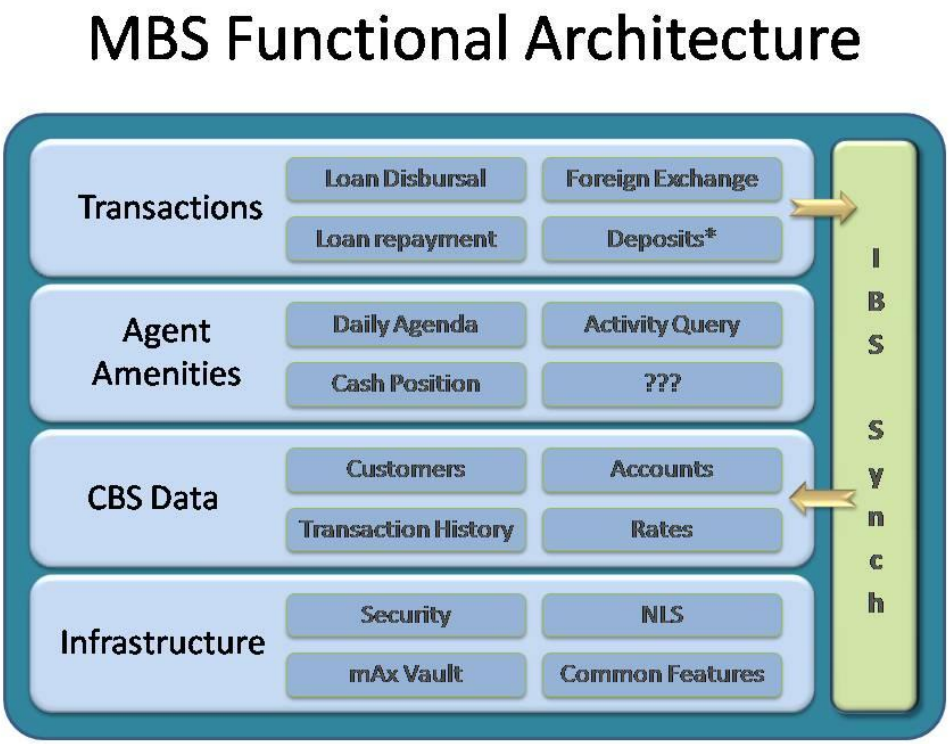
2.2 Functional Architecture of the MBS

The MBS is the system used by agents in the field. Since it allows an agent to work offline, it contains all the data required by the agent to decide what transactions need to be done. In addition, the transaction capture functionality has been built in such a way that transactions can be recorded and stored on the device until connectivity is available.

Transactions Layer

The purpose of the BFSImAx – MBS is to enable agents to do transactions in the field and this layer enables agents to do precisely that. It contains screens to do the various kinds of transactions supported by the system e.g. loan disbursal and loan repayment in release 1.0, foreign currency purchase and sale in the next release and transactions related to deposits and other banking products in later releases. Any time new transactions are to be supported by BFSImAx, this layer will certainly need to be enhanced.

In the larger scheme of things, any such transaction impacts customers of the bank, their accounts and the financials of the bank – e.g. withdrawals impact a savings account, repayments impact a loan account and foreign exchange transactions impact the bank's balance sheet. However, the MBS treats these transactions as complete in themselves and leaves the propogation of their effect to the server side systems - e.g. when the agent records a repayment or disbursal transaction the MBS does not concern itself with the resultant change in status of the loan account. Instead, it merely sends the transaction to the IBS in the next synchronization.



Agent Amenities Layer

Other than recording the actual transactions, agents require other features too from the system which help them do transactions – features which typically help them before or after a transaction. For example,

- When an agent is starting out her day, she may want to know her agenda for the day
- When the agent is at a particular location, she may want to review all the disbursals and repayments due at that location
- She may want to know her current cash position so that she can tally it with the physical cash in her hand
- Prior to doing a transaction with a new customer, she may want to view his/her photograph

All such features, which help an agent plan, manage and review her work, are part of this layer of the MBS. Generally speaking, the features of this layer do not change or record any data. Instead, they present the agent with various views of the data. The cash position functionality (wherein every transaction updates the agent's current cash position) is an exception.

When designing this layer, it should be kept in mind that most of the functions of this layer are components which provide some functionality which is common across several or all the transactions done on the device – including transactions to be built in the future. E.g. the agent's agenda shows loan disbursals and repayments today but in the future it may also show matured deposits and scheduled meetings – if deposits and group meeting related features are added to the system. The design of every feature in this layer should be done with this requirement in mind.

Coding Guideline When coding the IBS Data layer and the Transactions layer, developers should adhere to the standard interface mechanism laid down by every feature of these layers and not deviate from that.

IBS Data Layer

The BFSImAx – MBS system does a lot more than just allow agents to record transactions. It helps them in their work by

- creating their daily agenda,
- allowing them to verify customer details and view account history,
- increasing transaction accuracy by pre-populating key values in the transaction screens etc.

In order to do all this, the MBS needs to keep upstream data on customers, accounts and rates etc. For example it needs the schedule of upcoming repayments to present a daily agenda, it needs the customer address & photograph for the agent to view and it needs loan account numbers and disbursement amounts to pre-populate them in the disbursement transaction screen.

This data is received from the Intermediate Banking System and never updated by the MBS. While the MBS is not intended to contain the complex business logic of updating this data, it does read this data and does whatever little processing is required to use it as described above.

Infrastructure Layer

This is the most basic layer of the MBS and contains functionality which is not complete in itself – and which is not useful to a user unless coupled with a system transaction or query. It provides basic building blocks to be used by all other layers. Some of the features provided by this layer are:

- ✓ Security features such login authentication
- ✓ Enabling data encryption via the mAxVault feature
- ✓ Audit trail tracking
- ✓ System parameter maintenance
- ✓ Language resource files to enable users to use the system in their native language

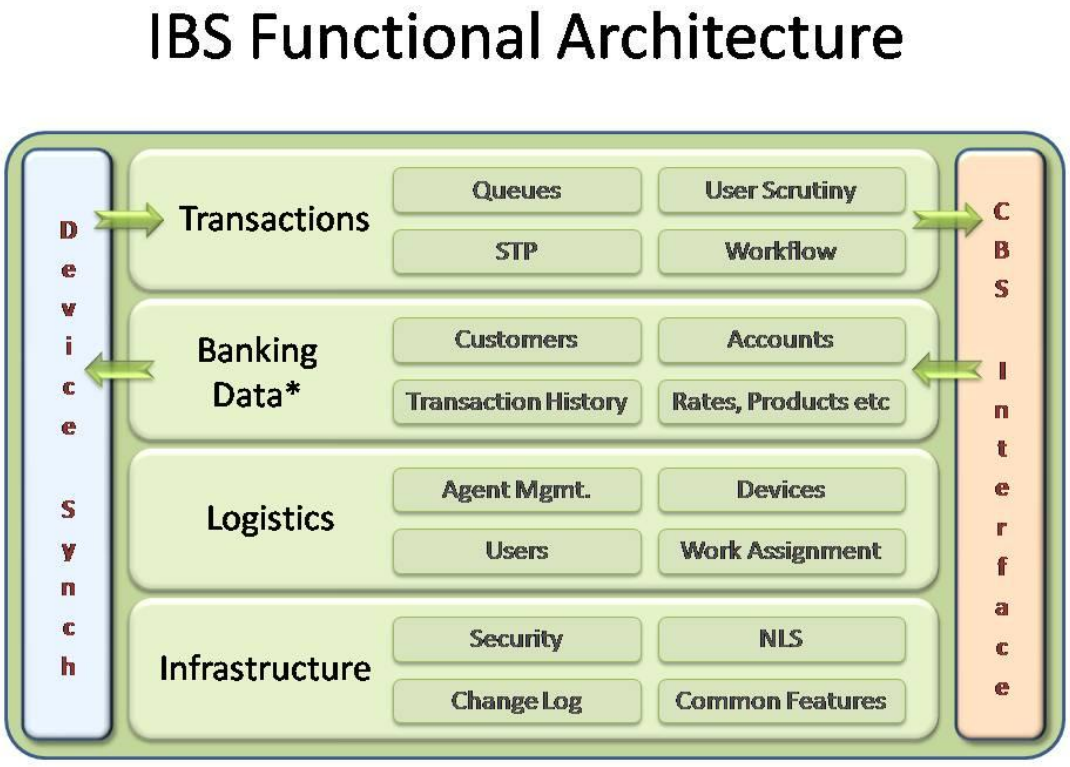
IBS Synch

The synchronization of a device with the IBS is always initiated by the device. The IBS synch component sends all new transactions done by the agent to the IBS and receives updated customer and account data from the IBS. Only data which is relevant to that device is received from the IBS.

2.3 Functional Architecture of the IBS

The IBS performs the following functions :

1. Enables IT admin to set up and manage agent mobile devices
2. Enables bank staff (credit officers, loan managers, agent co-ordinators etc.) to track and manage agents and the cash which is in their custody
3. Gets the latest customer and account information via its CBS interfaces, decides what data has to go to which device and sends it when that device synchs next
4. Synchs with an agent's device, receives the transactions done on that device and sends them to the CBS



Given that the IBS is to be used primarily by bank staff, usually in the bank's premises, it is architected as a conventional client-server system with a centralized deployment and a browser client. Based on the above requirements, its main functional components are described

Transactions Layer

All the transactions which are sent by the various devices are stored and managed in this layer. Once a new transaction is synched from a device, it enters a transaction queue. The system business logic then determines what processing is to be done on that transaction. Some transactions are flagged off for STP to the CBS whereas some others are held back for scrutiny by a bank manager who ultimately rejects or clears them. In future releases, a customizable business process should be triggered off for each transaction which is held back for scrutiny.

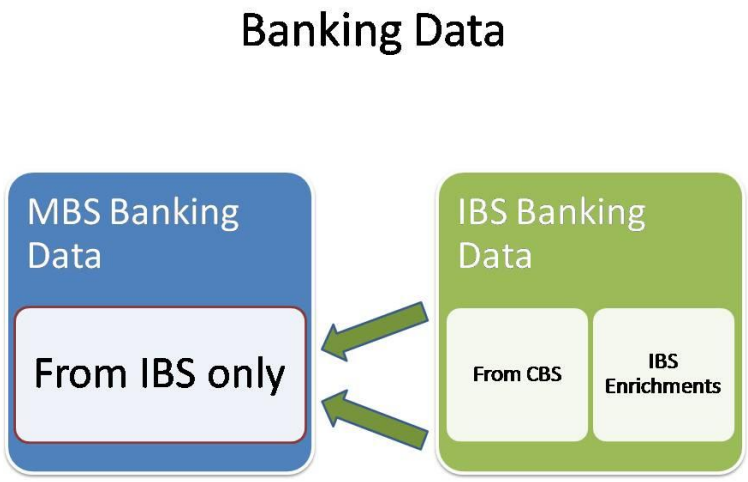
The transaction layer also interacts with the Logistics layer so that cash positions of each agent can be updated.

In the normal course, most banking transactions impact an account's history and balance – e.g. withdrawals impact a savings account, redemptions impact a term deposit and repayments impact a loan account. However, in BFSImAx, the transactions coming from the device are not processed e.g. upon receiving a repayment or disbursal transaction the IBS does not update the loan account. Instead these transactions are sent to the CBS and the updated loan account is received from the CBS.

Banking Data Layer

This layer of the BFSImAx – IBS contains all the banking data which is required for the system to function. This includes customer records, loan accounts and their history of transactions, foreign exchange rates, different products etc. All of this data is sourced from the CBS and relevant data is then supplied to each device based on that agent's requirements.

There is a slight difference between how the MBS treats account data and how the IBS treats this data. Since this data is sourced from the CBS, the IBS treats the CBS as the gold copy and does not alter it. However, Core Banking Systems have traditionally not been built for rural banking or for agent based banking and hence this data may not be sufficient for the purposes of BFSImAx. In such situations, functionality will need to be built in this layer to augment this data.



For example, BFSImAx needs every loan account to be assigned to an agent but the CBS may not have the concept of an agent. In such a situation, BFSImAx will have to build screens which will allow a user to link a loan account with an agent and store that linkage. It could

also happen that a CBS may not allow disbursal schedules to be defined for a loan with manual disbursals. However, in an agent based scenario, the bank may wish to define the proposed date and amount of disbursal (in other words, a manual disbursal schedule) and make it part of the agent's daily agenda. Once again, it is BFSImAx – IBS which will provide users with a screen which allows them to define the proposed date and amount of the next disbursal of a loan.

Thus the BFSImAx – IBS may need to augment the CBS data in order to make the banking data complete for its purposes. However, the MBS never updates or augments this data. For the MBS, this data is totally hands-off, it comes from the IBS and is not touched thereafter.

It should however be noted that storage of this augmented needs to be designed in such a way that there is a physical and logical separation between the data which comes from the CBS and the data which is captured at the IBS. This will make the CBS interface simpler, allowing it to overwrite existing CBS data - instead of the more complicated approach of ascertaining what has changed and updating it.

Logistics Layer

This layer of the IBS is used by the bank to manage the entire ecosystem of the devices, the agents and the other stakeholders. This is the layer in which system administrators define the bank staff and decide what role they can perform in the system. Similarly, features to define agents and help their supervisors manage them also belong to this layer. For example, this layer helps bank staff keep track of the cash position of the agent and assign work to them. In addition, this layer also allows the bank to set up devices and manage them.

Infrastructure Layer

As in the Mobile Banking System, this layer of the IBS contains functionality which is not complete in itself – and which is not useful to a user unless coupled with a system transaction - but provides basic building blocks to be used by all other layers. Some of the features provided by this layer are:

- ✓ Listing of user actions and entitlements
- ✓ Audit trail tracking
- ✓ Change history maintenance
- ✓ System parameter maintenance
- ✓ Language resource files to enable users to use the system in their native language
- ✓ Security features such login authentication

Device Synch

The synchronization of a device with the IBS is always initiated by the device. The IBS synch component receives transactions from the agent MBS and puts them in the IBS transaction queue. In addition, it checks all the banking data on the IBS and sends the relevant data to that device. The synch is designed in such a manner that data is sent only once to the device and not every time on successive synch operations – unless the same data has been updated by the CBS and a new copy has been received from the CBS.

It is possible that a device may not be able to synch due to hardware fault or some other reason. For this reason there should be a transaction/function on the IBS which retrieves data from a device which cannot synch.

CBS Interface

The CBS interface should ideally be a continuously running daemon. However, it can also be a user initiated process or a process triggered by some event in the CBS (e.g. end of day completion). This interface sends all transactions received from various devices (except those kept on hold) to the CBS.

In addition, it gets from the CBS all the customers and accounts (currently loans and perhaps later deposits and savings/current accounts too) which have changed since they were last updated to BFSImAx. The interface should be intelligent enough to select only those customers and accounts which have changed in a manner which impacts BFSImAx. E.g. accrual of the interest on a loan is a change which does not impact BFSImAx but when a repayment is done by a customer it does. Hence, the selection logic of the CBS interface needs to be suitably coded so that needless traffic between the two systems is not generated.

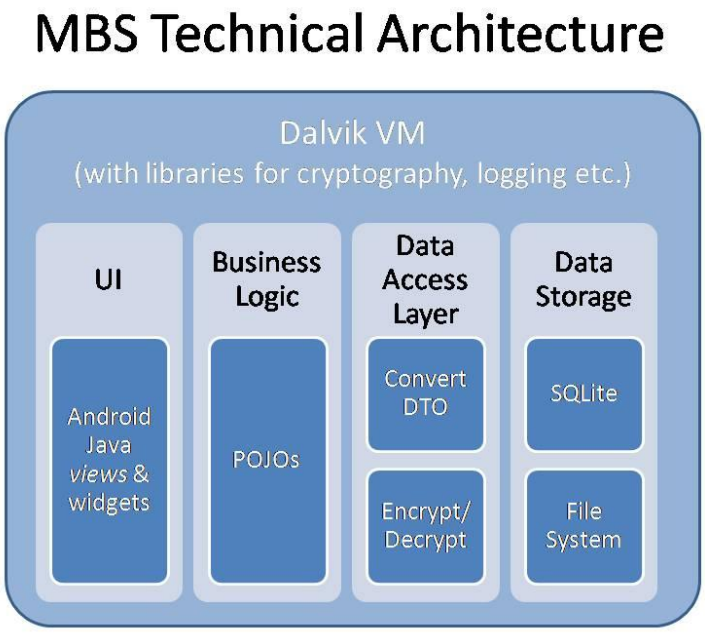
2.4 Technical Architecture of the MBS

The device software of the BFSImAx system has been built to run on Android devices, hence it will be developed using the Android Java platform. All layers of the application - UI, business logic and data access – will be built using features provided by the Dalvik VM and some standard libraries available for it. This will keep the application not only architecturally simple but will also ease the developer learning curve and (hopefully) increase the deployment stability.

The UI will be built using standard Android Java views. The business logic components will be simple Java objects and so will the data access layer.

In the MBS architecture, only the DAL is data storage aware. In the business logic layer, all data (entities, transactions etc.) are represented either as Javabeans or as data variables/objects within the relevant Java classes. This allows developers to code in an object oriented fashion rather than manipulate database rows or XML documents.

The transformation into database rows and XML/JSON files is done in the DAL only. This allows two critical design decisions to be encapsulated within the DAL and shields the rest of the application from changes in these. Firstly, the BFSImAx-MBS stores data in both the SQLite database and as files in the file system (see section 3.1). Confining access to these two storage mechanisms



in the DAL will mean that any subsequent changes to these will also impact the DAL only. Secondly, all data stored in the file system is encrypted (see section 3.2) and having a DAL will mean that only certain specific code components need to be aware of the encryption mechanism.

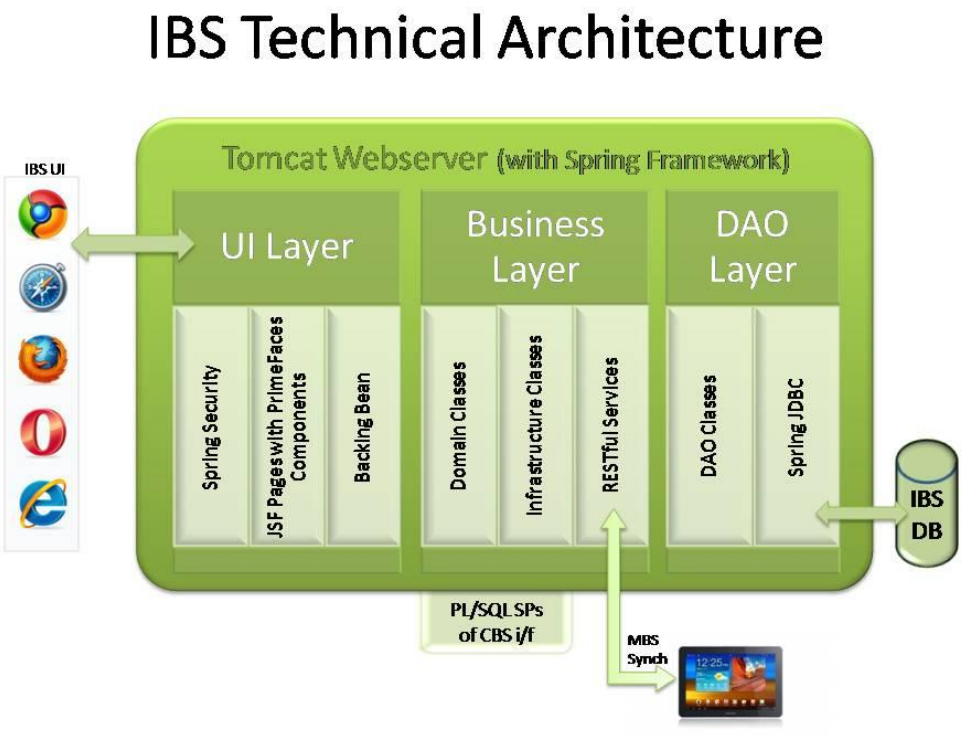
Coding Guideline When coding read or write access to existing data in the business logic layer, developers need to ensure that their code works with the data objects provided by the DAL and not with the physical data. Similarly when writing business logic which stores and accesses new kinds of data, developers need to break their code into distinct business logic and DAL layers.

2.5 Technical Architecture of the IBS

The IBS has been designed as an OS independent Java application. While it is an enterprise application, it has been designed to not require a J2EE server. Instead, a web server which supports JSFs – e.g. Tomcat – will suffice. Since BFSImAx will run in a web server and not a J2EE server, the Spring framework will be used to provide some of the functionality like security, JDBC access etc.

UI Layer

The base UI and MVC framework used is JSF2 and the component library used is PrimeFaces. This combination of JSF with PrimeFaces will allow the application to be web based and still have a rich look and feel. JSF is a mature and stable paradigm which is widely supported by vendors. PrimeFaces (along with RichFaces and ICEfaces) is one of the three most widely used component libraries and has emerged as the leader, not only in terms of number of components but, as anectodal evidence suggests, also in terms of performance and awareness. Spring security will be used for authentication whenever a page is invoked.



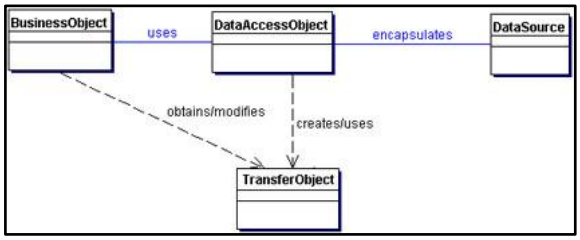
The screens of BFSImAx-IBS will use templating to ensure that common functionality is maintained in every screen (see Section 3.4 for details). Within the templates and in the main form itself PrimeFaces components will be used as the UI components on the screen.

The backing bean of each page will be used to capture

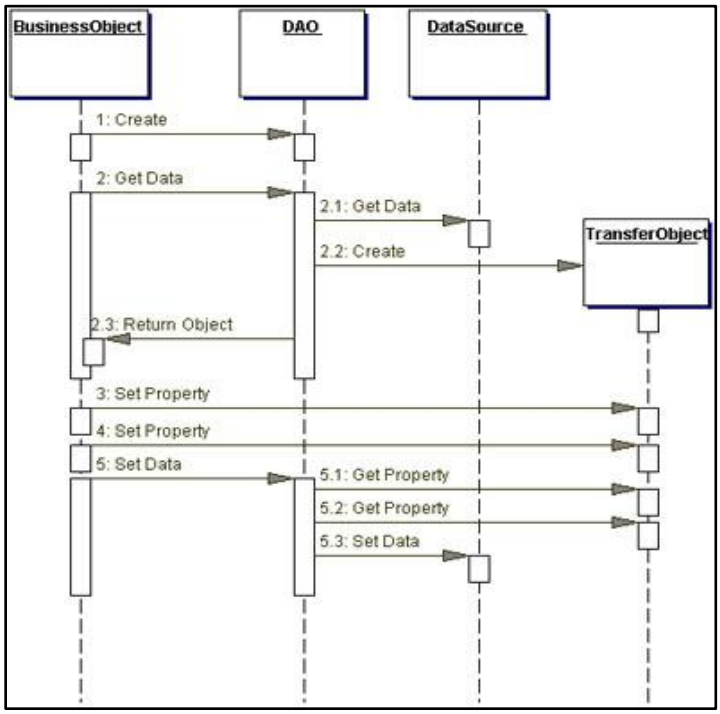
the data of the page and to process it once it is submitted. However, within the backing bean, the normal paradigm of mapping UI data to backing bean attributes will not be used, instead a separately defined ViewDTO will be used within the backing bean to capture the form data. This will enable a neat separation of the data entities from the UI logic and will ensure that any future migrations to some other technology in the UI layer do not affect the business logic, as long as they produce the same DTO.

Business and DAO Layers

The business logic and DAO layers will be implemented as standard Java objects (POJOs). The DAO layer will implement the standard CRUD operations of data which is maintained in the IBS. This separation will ensure that the business logic is separated from the code which concerns itself with physical data access. The communication between the DAO and the business logic layers will be through BusinessDTOs. This will enable a neat separation of the persistence data from the business logic and will ensure that what is communicated between the business logic and the DAO layers is only the persisted data – without any intermediate data used for business processing being added to it.



The following sequence diagram (from the Oracle web site) explains this interaction.



This separation of BL and DAO will ensure that the business logic is separated from the code which concerns itself with physical data access and will be shielded from both the physical storage mechanism and the logic of transforming from object to relational data. For example, in the case of BFSImAx-IBS, an object-relational mapping tool is currently not being used and Spring JDBC will be used to read and write to the database. In case this is to be changed in the future and Hibernate is to be used, the change will be in the DAO layer only and as long as the new DAO takes and receives

the same DTO, the BL layer shall remain unchanged. Similarly, there may be changes in the physical access to data and the BL will again be shielded from these changes. Such changes are eminently possible in our case since the IBS is really an intermediary system which will have to interact with the CBS - and possibly other departmental systems as it caters to different types of transactions.

Designing business objects (complete with a full set of attributes) and a separate DTO which mimics the BO attributes is a cumbersome task which will lead not only to complications in coding but also at deployment time. However, the BFSImAx-IBS is a system meant primarily for data flow and only secondarily for business processing and the DTO paradigm mirrors this data flow philosophy. Hence a way needs to be found to use DTOs without creating unnecessary heaviness. There should therefore be no duplication of attributes between the business object and the DTO. In other words, the DTO should be designed as the attributes of the BO itself and should be embedded in it. In that sense, the separation between the BO and the DTO is not so much that they are independent objects with different attributes (as shown in the above diagrams). Instead, the DTO should be viewed as that *part of the BO* which can be detached and transferred to the DAO layer

Coding Guideline All backing beans in the UI layer should be coded with a separate DTO encapsulating the screen data and this DTO should be used to communicate with the BL layer. Similarly, all business entities in the business logic layer should be coded with a DTO holding the entity data and the complete business object should not be passed to the DAO layer. Neither the UI backing beans nor the business logic should directly access the physical data via JDBC calls.

In the previous section, it was described how the backing bean of a page should encapsulate the data within a ViewDTO and this ViewDTO should be passed to the business logic for processing. Similarly, we discussed in this section that a BusinessDTO should be used to pass the persistence data back and forth between the business logic and DAO layers. In most cases these two objects will be nearly (or completely) identical and should be designed as one object. Moreover, having a two separate DTOs and a third BO in the same transaction is too cumbersome and should be avoided unless there are good reasons for it. This unification in design will save the effort of transforming from one to the other and the same object can be used across the UI, business and data access layers, thereby abstracting the data communicated between these layers from the presentation logic, business logic and data access logic inherent in these layers.

Coding Guideline When designing a maintenance transaction – i.e. a transaction in which data will be created, edited and viewed (e.g. device maintenance) - designers/programmers should first design the relational data structure and the relevant screen, then the BusinessDTO and then assess if the same BusinessDTO can be used as the ViewDTO also. The same approach can be taken for any financial transactions also. When considering view only screens, it is assumed that the data structure is already given. Hence designers/programmers should first design the screen and then see if any existing DTO fits the purpose. If there is none, then a ViewDTO should be designed for that screen.

3 DESIGN NOTES

While the previous chapter describes the overall architecture, this chapter goes into the next level of detail on some issues which need farther detailing before a comprehensive design can be done on some of them.

3.1 MBS Data Storage

The application will store all images and transactions in the file system. Images will not be indexed and hence need not be stored in the database for access reasons and storing images in the file system will reduce the chances of the database bloating. Also, larger images are often faster to access from the file system than from inside a SQLite database. A link to the image file will be kept in the customer record.

Transactions done by the agent will also be stored in the file system in JSON format. However, these will be stored encrypted so that no user can gain access to them except through the application. For details of encryption see section 3.2

All query-able data will be stored in the SQLite database. Here queried data is being differentiated from viewed data. The former is queried via select queries which scan large amounts of data whereas the latter is viewed or accessed after a particular key has been zeroed in on. Keeping queried data in the database will ensure that indexes can be used to speed up queries. Hence even though transactions are stored encrypted in the file system, certain attributes of the transaction (e.g. type, customer, date etc.) are stored in the database for ease of retrieval. In addition to this, the entire IBS data (customers, accounts etc.) is also stored in the database.

3.2 MBS Encryption Strategy

The MBS needs to encrypt two separate kinds of data. Firstly, when the user logs in with her password, it needs to be authenticated against an encrypted form of the password which is stored within the application. This encryption will be done using the SHA-256 hashing algorithm which is supported by Android. In addition, the password will be transformed using the PBKDF2 methodology (also used later for transaction encryption). Whenever the user supplied her password, it will be hashed using the same PBKDF2 methodology and the resultant value will be compared using the hashed value stored on disk. On disk, the hashed password will be stored in SharedPreferences storage mechanism which Android provides for private data.

The second kind of data which needs to be encrypted is the transaction files stored on disk. The objective is that no one should be able to access the unencrypted data in these files except when viewing it through the BFSImAx-MBS application. These files will be encrypted using the AES algorithm, which is one of the most secure symmetric algorithms in widespread use and is expected to be unbreakable for several years to come. In future releases, authenticated encryption can be incorporated by using AES-GCM.

However, while the data itself may be secure, the key used to encrypt the data needs to be kept on disk and is, therefore, insecure. It is therefore best to use a key based on a user supplied value (e.g. the user password) so that anyone accessing the encrypted file (other than the user herself) does not know the key used for encryption. The user supplies this password every time she logs in and it is never stored in clear text on the device. This method has the drawback that when the user changes her password, all the encrypted transactions need to be decrypted using the key based on the old password and re-encrypted using the key based on the new password. This will make the password change transaction too cumbersome and prone to failure.

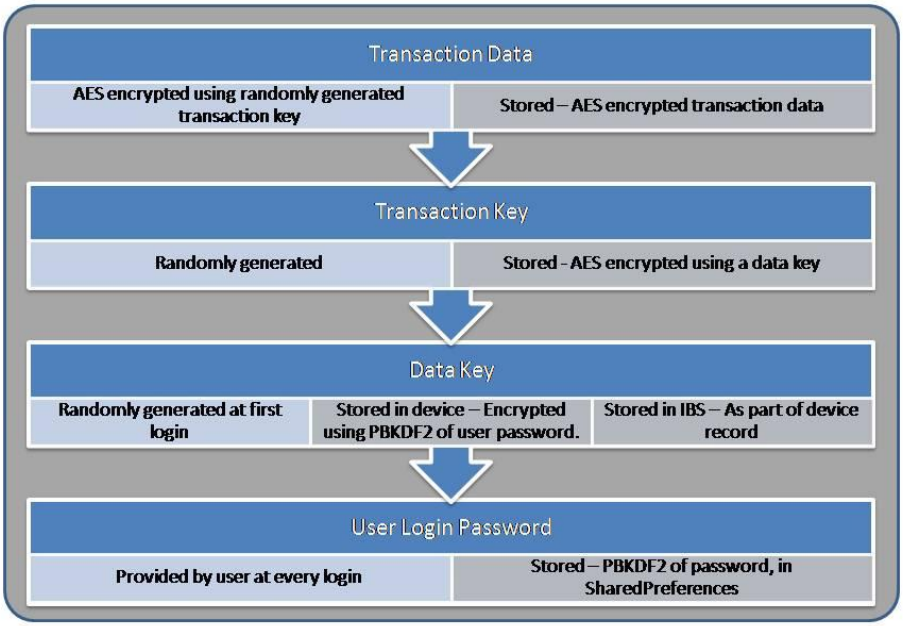
Secondly, we also want to design an encryption scheme whereby transactions on the device can be decrypted on the server, by authorized admin users, even without knowing the device user's password. Moreover, we don't want an encryption scheme in which the data cannot be decrypted if the device user forgets her password.

To overcome these drawbacks in simple password based encryption and to still keep the transaction key secure, BFSImAx-MBS uses a two level key mechanism. In the first level, the key with which the transaction file is AES-encrypted is randomly generated. This makes the transaction file extra secure since it is not only encrypted using a very secure algorithm but the key too is randomly generated, making it much more secure than a user supplied key.

In the second level, the randomly generated AES key is itself AES-encrypted, using a data key. This data key is generated (using the keyGenerator class in Java cryptography) when the application is installed and the agent logs in for the first time. Since the first login is a connected transaction, this data key is sent to the server and stored in the device record in the IBS. Using this data key, the device data can be decrypted by an authorized IBS user, if required.

On the device, this data key is encrypted using AES and its encrypted form is stored in SharedPreferences. The key used for encrypting & decrypting the data key is the user password itself. However, since user supplied passwords are notoriously insecure as encryption keys, a password based key derivation function is used to derive a secure key from the data password. The specific function used is PBKDF2, which is published by RSA and is considered the most secure way to encrypt data using a user supplied password. PBKDF2 uses salting and key extension to derive a key based on the supplied password. Refer here, here, here and elsewhere for details.

MBS Transaction Encryption



The following sections explain how this encryption strategy is to be used in different application scenarios.

On First Login

On first login after installation, the MBS should undertake the following actions for encryption:

1. Make the user change her password to one of her choice.
2. Generate a random salt and create a PBKDF2 hash of the password.
3. Store the user password salt and the encrypted user password in SharedPreferences.
4. Generate a random AES 256-bit key as the data key.
5. Send this data key to the IBS (in an https session), to be stored against this device id.
6. Generate a random salt and create a PBKDF2 hash of the user password. Although based on the user password, this salt and hash value are different from the salt and hash value obtained in step 2.
7. AES-encrypt the data key, using the PBKDF2 hash of step 6 as the encryption key.
8. Store the data key salt of step 6 and the encrypted data key of step 7 in SharedPreferences. Do not store the PBKDF2 hash obtained in step 6.

On Every Subsequent Login

1. Let the user enter her login password.
2. Retrieve the user password salt from SharedPreferences.
3. Create the PBKDF2 hash of the user password using the salt retrieved in step 2.
4. Retrieve the encrypted user password from SharedPreferences.
5. Compare the values obtained in steps 3 and 4. If they are identical, allow the user to login otherwise the login has failed.

To Encrypt a Transaction File

1. Obtain the user's login password.
2. Retrieve the data key salt from SharedPreferences.
3. Using the salt retrieved in step 2, create the PBKDF2 hash of the user password.
4. Retrieve the encrypted data key from SharedPreferences.
5. Using the PBKDF2 hash of step 3 as the AES decryption key, decrypt the data key retrieved in step 4. We now have the plain text data key.
6. Generate a random 256 bit AES key to encrypt the transaction file.
7. Encrypt the transaction file using the transaction key of step 6 and store the file on disk. Store a reference to the file in the SQLite database.
8. AES encrypt the transaction key of step 6 using the data key obtained in step 5. Store the encrypted transaction on disk.
9. Encrypt the transaction key of step 6 using the data key of step 5 as the key. Store this encrypted transaction key in the SQLite database along with the reference to the transaction file.

To Decrypt a Transaction File

When decrypting a transaction which is to be viewed, the following sequence should be followed:

1. Obtain the user's login password.
2. Retrieve the data key salt from SharedPreferences.
3. Using the salt retrieved in step 2, create the PBKDF2 hash of the user password.
4. Retrieve the encrypted data key from SharedPreferences.
5. Using the PBKDF2 hash of step 3 as the AES decryption key, decrypt the data key retrieved in step 4. We now have the plain text data key.
6. Retrieve the encrypted transaction key from the SQLite database along with the reference to the transaction file.
7. Decrypt the transaction key of step 6 using the data key of step 5. We now have the plain text transaction key.
8. Use the decrypted transaction key of step 7 to decrypt the transaction file.

On User Password Change

When the user wishes to change her password, the following steps should be followed:

1. Let the user enter her old login password. Check if it is the correct password by doing the normal password correctness check.
2. Retrieve the data key salt from SharedPreferences.
3. Using the salt retrieved in step 2, create the PBKDF2 hash of the old user password.
4. Retrieve the encrypted data key from SharedPreferences.
5. Using the PBKDF2 hash of step 3 as the AES decryption key, decrypt the data key retrieved in step 4. We now have the plain text data key.
6. Make the user change her password to one of her choice.
7. Generate a random salt and create a PBKDF2 hash of the new user password.
8. Store the user password salt and the encrypted user password in SharedPreferences.
9. Generate a second random salt and create a second PBKDF2 hash of the user password. Although based on the user password, this salt and hash value are different from the salt and hash value obtained in step 7.
10. AES-encrypt the data key (obtained in step 5), using the PBKDF2 hash of step 7 as the encryption key.
11. Store the data key salt of step 9 and the encrypted data key of step 10 in SharedPreferences. Do not store the PBKDF2 hash obtained in step 9.
12. There is no need to decrypt and encrypt either the transaction files or their transaction keys since the data key has not changed.

When The User Forgets Her Password

Decide on an alternate authentication mechanism. This may be done via the user answering a security question or by generating a new device login password on the server and informing the user using a secure channel. Given the sensitivity of the data on the device, the latter option is recommended. Either way, the user has to go through a “forgot password” transaction on the device. This transaction follows the following sequence:

1. Start a secure https session with the IBS.
2. Authenticate the user using the alternate authentication mechanism.
3. Make the user change her password to one of her choice.
4. Generate a random salt and create a PBKDF2 hash of the password.
5. Store the user password salt and the encrypted user password in SharedPreferences.
6. Get the data key (256-bit AES key) of this device from the IBS.
7. Generate a random salt and create a PBKDF2 hash of the user password. Although based on the user password, this salt and hash value are different from the salt and hash value obtained in step 4.
8. AES-encrypt the data key, using the PBKDF2 hash of step 7 as the encryption key.
9. Store the data key salt of step 7 and the encrypted data key of step 8 in SharedPreferences. Do not store the PBKDF2 hash obtained in step 7.
10. There is no need to decrypt and encrypt either the transaction files on the device nor their transaction keys since the data key has not changed.

3.3 MBS Error and Debug Logging

Debugging of Android applications is normally done by developers by connecting the application to an IDE on a PC and stepping through the application. However, the MBS application will be deployed on agent devices which will not only be numerous in number but may also be inaccessible to BFSI staff and even the bank's IT staff. Hence it is imperative that a simple mechanism be incorporated into the application which allow the developers to access debug information. This can be done by maintaining a debug log on the file system and sending this log to the IBS as part of the synch.

A debug routine needs to be written which will write all debug messages to a debug file instead of to the normal Android log. This file will work in conjunction with a debug flag.

- Upon logging in, the app will set the debug flag off.
- If required the user will set the debug flag on. This status will last only for one session.
- Programmers shall call this routine to log all debug messages
- If the debug flag is off, this routine will return without doing anything with the debug message.
- If the debug flag is off, this routine will append the debug message, with the date-time stamp, to the debug file
- During the next synch, the synch process will send this debug file to the IBS and then delete its contents

Coding Guideline During the development of the app, debug messages should not be written using the normal logcat logging mechanism provided by Android. Instead, programmers should call the BFSImAx-MBS debug routine.

In addition to the debug messages, the same log file should be written to whenever an unhandled error occurs in the app. The details of how this exception will be trapped are to be worked out during the detailed design. Some of the approaches which can be considered are using the ACRA library, overriding the DefaultUncaughtExceptionHandler, enclosing every activity in a try/catch block or any combination of these and other ideas.

Once the unhandled exception has been trapped, the data related to that can be written to the same BFSImAx-MBS debug file. Apart from the date-time stamp, the activity name and the error, the stackTrace and the relevant content from the Build object should be logged.

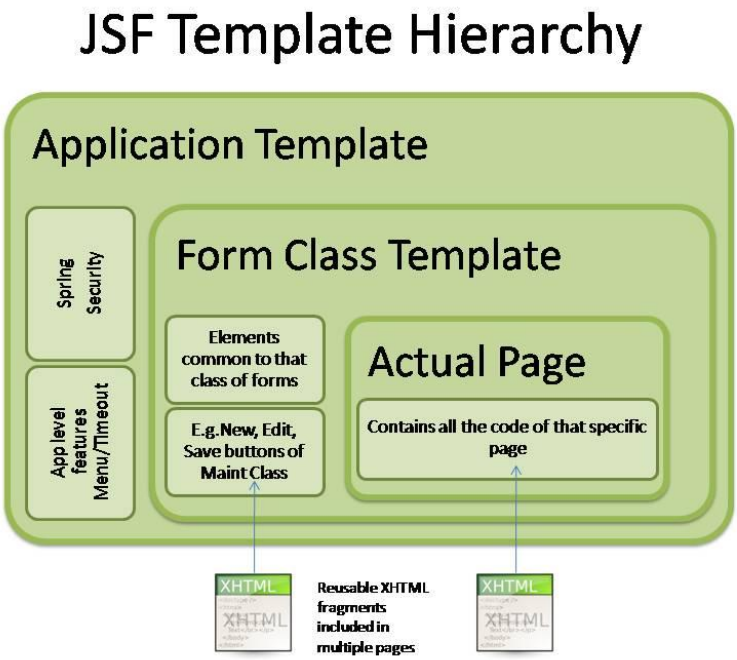
Coding Guideline Regardless of which mechanism is chosen to trap the unhandled exceptions, it will require some code in every activity to facilitate the trapping. Programmers should ensure that they write their activity classes in compliance with the devised standard so that trapping of unhandled exceptions works.

3.4 Usage of Templates in the IBS

The various screens of any product like BFSImAx contain common elements which occur in every, or at least many, screens. To avoid definition of these common elements in every screen, finding a way of coding them in one place, which can be referred or inherited in all screens, is essential. Templates provide exactly this mechanism in the JSF world and are the recommended as the inheritance mechanism for XHTML pages.

In BFSImAx, we will use a two level templating mechanism. The first template is the one which will be used by all the screens of the system, regardless of their nature and function. This is the application template and it will contain functionality such as the application navigation, help, session expiry, authentication via Spring security etc.

This application template will be the template for all form class templates. A form class template is essentially a template which contains UI elements which are common to a class of forms.



For example, all entity maintenance forms need the same action buttons for Add, Edit, Submit etc. and they all need the standard BFSImAx audit trail. Hence there can be a maintenance.xhtml which defines the three divisions of the page, explicitly codes the action button and the audit trail divisions and leaves the main content division to be inserted later by the actual form. All maintenance forms would then use the maintenance.xhtml as their template. Similarly, a listview template can also be defined at the second level since all list forms are a class of forms with common functionality. Lastly, a generic template can be defined which contains minimal structure/functionality common to all screens. This generic template can be used by all screens which do not fall under the maintenance or listview templates.

Coding Guideline All BFSImAx-IBS pages must be based on a second level template. They should not be coded without reference to a template and neither should they inherit directly from the application template.

The first release of BFSImAx is likely to contain only three templates in the IBS – the maintenance template, the listview template and the generic template. However, it is expected that as the functionality of BFSImAx grows, the number of templates will increase. As this happens, there is a

risk of proliferation of templates, thereby diluting the benefit which templates give in maintainability of code. On the other hand, there is also a risk of having too few templates. This could lead to the pages containing code which twists this way and that to work around template functionality or it could lead to the template itself becoming heavier and less cohesive because it tries to cater to diverse pages which expect unrelated (or even contradictory) template behavior. To avoid all these pitfalls, the development process needs to incorporate a governance process for template creation and refactoring.

Coding Guideline Programmers should never unilaterally create a new template or change an existing one. Any changes to the template layer should be debated upon by a design panel and then effected.

In addition to templates, commonly used UI elements can be kept in separate XHTML files and included in the relevant pages using the facelets:include tag. For example, the audit trail UI elements can be kept in an uaditTrail.xhtml file and included in both the maintenance template and directly in any other forms which inherit the generic template but show data which has an audit trail.

3.5 The Anatomy of a Synch Session

The synchronization between the device and IBS needs to be built in such a way that it is robust when the network is unreliable. This necessitates that it have the ability to restart from where it left off (and not from the beginning) if the connectivity is interrupted. The synch also needs to be secure so that financial transactions are not compromised. Given these requirements – and the architecture of the rest of the system – the synch interaction between client and server should be designed as follows:

	Device Action	*IBS Action*
1	User initiates synch	
2	MBS makes list of transactions to be synched (all unsynched transactions)	
3	Start a secure https (TLS) session with server	The encryption cipher should be set as a reasonably secure one. E.g. a setting of TLS_RSA_WITH_AES_256_CBC_SHA256 will ensure that RSA is used for the asymmetric encryption of the session key and AES-256 is used for actual data encryption.
4	Generate an MBS synch session id	
5	Call synch session start service on IBS Send device id, MAC id, user, synch session id to IBS	Accept data from IBS and verify Start unauthenticated synch session
6	Accept authentication credentials from user Send credentials to IBS (Should we keep a synch password on the server? Or should we use client certificates? Is this necessary at all?)	Verify credentials and confirm to IBS Flag off session as authenticated
7	Create list of transactions to be sent Call sessionTranList service on IBS	
8		Verify whether any transactions have already been received in an earlier synch session Respond to MBS with list
9	If IBS has already received any transactions, update status and synch session for these	
10	For each of the transactions to be sent to IBS (in order of date-time stamp)	
10.1	Decrypt transaction and create JSON	
10.2	Update status of txn as P (under Process)	
10.3	Call postTran service on IBS (over TLS connection)	

	Device Action	*IBS Action*
10.4		Receive transaction from IBS Commit to transaction queue table in db Confirm receipt to MBS
10.5	Update status of transaction as S (Sent)	
10.6	Now process next Unprocessed transaction	
11	Once all transactions are processed, call service to signal end of transactions	
12	Get changes, if any, to the relevant device record on the server.	
12.1		Check for unsynched changes in device parameters and send – or confirm no changes
12.2	Update changes to device and acknowledge	
12.3		Update device record as synched
13	Get changes, if any, to the relevant system parameters on the server.	
13.1		Check for unsynched changes in system parameters and send – or confirm no changes
13.2	Update changes to device and acknowledge	
13.3		Update device as having latest system parameters
14	Get changes, if any, in the relevant agent parameters on the server.	
		Check for unsynched changes in device parameters and send – or confirm no changes
14.2	Update changes to device and acknowledge	
13.3		Update agent record as synched
14	For all the unsynched loan accounts on the IBS, do the following:	
14.1	Call the loan account service on the IBS	
14.2		Check if there are any unsynched loan accounts to be sent to *this device*. If yes, then return the data for one loan account (batch size can be used to make the synch more efficient). If not, then return confirmation that there are no more changes.
14.3	Update the loan accounts received from the IBS into the SQLite database	
14.4	Call the loan account ack service	Update the relevant loan accounts as synched.

	Device Action	*IBS Action*
14.5	Call the loan account service again, until it returns confirmation that all loans are synched.	
15...	Call other IBS services which will send data to the device.	
		Send data of the type requested
	Execute the business logic to process that received data. Acknowledge receipt to the IBS.	
		Flag off the data as synched so that it is not sent again.

Restart Enabled

Analysis of the above sequence will show that the synch logic is restart enabled. In other words, if it is restarted, the device will not resend the transactions it has already sent in the earlier synch sessions. Similarly, the IBS also will not resend the transactions it has already sent. This is ensured by the fact that the sender picks up only unsynched transactions and sends them one by one. They are then processed by the receiver, acknowledged to the sender and finally, flagged off as synched by the sender. Hence, in the next synch, the sender does not pick up transactions which have already been synched. Both the MBS and the IBS must use the database transaction management commands to ensure that atomicity of the updates related to a single transaction are maintained.

The following table describes the possible failure points and the system features which ensure either non-failure or self recovery when the IBS is sending transactions to the IBS.

MBS Action	***IBS Action***	***Recovery Mechanism At This Point***
Calls IBS service which receives transaction from device	Receives transaction	No updates to either MBS or IBS database. MBS resends transaction on next synch.
	While updating transaction in database	Database transaction management ensures rollback.
	Post transaction commit but before acknowledgement to MBS	Next synch, MBS will resend the transaction but IBS will not process it. Instead it will just reply with the processed status.
	Acknowledges transaction receipt to MBS	Next synch, MBS will resend the transaction but IBS will not process it. Instead it will just reply with the processed status.
Receives acknowledgement but has not yet updated transaction status in SQLite database		Next synch, MBS will resend the transaction but IBS will not process it. Instead it will just reply with the processed status.

MBS Action	IBS Action	Recovery Mechanism At This Point
Updates transaction status to Sent		Next synch, this transaction will not be sent

The data the MBS receives from the IBS has to be treated separately based on whether it needs to be simply overwritten in the device or it needs to be updated with some business logic. Data which is simply overwritten on the device is anyway restart friendly since a failure during the synch will mean that a resend from the IBS will simply overwrite the data again. For data which the MBS needs to update based on some business logic, the recovery is a little more complex, as described in the following table.

MBS Action	IBS Action	Recovery Mechanism At This Point
Calls IBS service which sends updateable data to device	Selects transaction and returns with date-time stamp of last update to this transaction	No updates to either MBS or IBS database. IBS resends transaction on next synch.
MBS checks date-time stamp of the transaction copy it has. If its own date-time stamp matches or exceeds the date-time stamp of the transaction received, then it simply returns success without processing		Next synch, IBS will resend the transaction but MBS will not process it. Instead it will just reply with the successful/processed status.
While updating transaction in database		SQLite transaction management ensures rollback.
Post transaction commit but before acknowledgement to MBS		Next synch, IBS will resend the transaction but MBS will not process it. Instead it will just reply with the successful/processed status.
Calls IBS service to acknowledges transaction receipt	Receives acknowledgement but has not yet updated transaction status in database	Next synch, IBS will resend the transaction but MBS will not process it. Instead it will just reply with the successful/processed status.
	Updates transaction status to Sent	Next synch, this transaction will not be sent