



# UNDERSTANDING APPLICATION AUDIT REQUIREMENTS



WHITE PAPER

## The Ubiquitous Audit Log

Understanding business requirements for audit logging, version history and undo.

Bert Hooyman  
Mphasis Europe BV

March 2008

## TABLE OF CONTENTS

<b>1. Introduction</b>	<b>1</b>
<b>2. Keeping track of changes</b>	<b>1</b>
<b>2.1. Pure Auditing</b>	<b>1</b>
<b>2.2. Explicit Version Histories</b>	<b>2</b>
<b>2.3. Going Back in Time: Undo recent changes</b>	<b>4</b>
<b>3. Requirements Summary</b>	<b>5</b>
<b>4. Literature References</b>	<b>6</b>
<b>4.1. Patterns for things that change with time</b>	<b>6</b>
<b>4.2. A Collection of History Patterns</b>	<b>6</b>
<b>4.3. Time Travel</b>	<b>7</b>
<b>4.4. Transaction Compensation</b>	<b>7</b>
<b>4.5. Bi-temporal Databases</b>	<b>7</b>
<b>5. Glossary</b>	<b>7</b>

## 1. Introduction

In this report, we analyze typical application functional requirements around tracking changes to relevant business data, a feature that is normally identified as an “audit log” requirement. We limit ourselves to data that is held in application databases.

Our discussion assumes that the audit log is part of the application design itself, although in a technical design the logging capabilities can be fully externalized and shared across multiple applications.

The purpose of this report is to help business analysts and application architects to properly identify any auditing and logging needs. The technical design options around implementing any of the discussed auditing strategies will be discussed in a subsequent paper.

## 2. Keeping track of changes

### 2.1. Pure Auditing

The baseline requirement for an auditable database application is that any modification to the content of selected tables must be recorded, along with date & time of the change and a reference to the identity of the person who was responsible for the change.

The recording is known as the Audit Log, or Log for short<sup>1</sup>. Logs are written to continuously, with every change to the database tables that are involved. As it stands, audit logs are not reviewed very often, usually only in special situations or during annual audit sessions. Because of this, there is a tendency to make log writing as efficient as possible at the expense of log reading – as logs are not read very often, there is no harm in making the retrieval process less efficient.

As a consequence, a simple audit log may be realized as a text file, with a line of text added for every data modification. This can be elaborated somewhat more with tab or comma-separated text, effectively turning the text file into a table. Typical table columns include:

- Type of object being modified (i.e. the table name, in database context)
- Identity of the object being modified (i.e. the primary key in database parlance)
- Date & time of modification
- Identity of the person modifying the data (usually as some form of identifier such as employeeID or userID)

---

<sup>1</sup> A glossary of terms is included in this report as chapter 5, starting on page 7.

- Nature of the modification (insert, update or delete)
- Details of the modification, typically as a text string

This type of auditing is perfectly adequate for many offline auditing requirements, as the structure implied in the text file is sufficient to filter audit entries of interest. Note however that audit logs based on output to a text file have very limited business value, despite their popularity (through open source solutions such as log4j).

Another concern with this style of auditing is the effort that is needed (in application logic) to generate the details of the modification as a text string (column F in Figure 1).

**Figure 1: Tab-separated log data displayed in Excel**

	A	B	C	D	E	F
1	order	34	23-Oct-2005 09:45:20 GMT+5	1012	I	client=35442; description=""; discountScheme=3; priority=2; salesRep=17
2	lineitem	213	23-Oct-2005 09:45:20 GMT+5	1012	I	order=34; pos=1; product=1400643; qty=1; discountPerc=15
3	lineitem	214	23-Oct-2005 09:45:20 GMT+5	1012	I	order=34; pos=2; product=1400054; qty=2; discountPerc=5
4	lineitem	215	23-Oct-2005 09:45:20 GMT+5	1012	I	order=34; pos=3; product=1400712; qty=1; discountPerc=5
5	order	34	23-Oct-2005 10:12:00 GMT+5	1012	U	OLD: priority=2; NEW: priority=3

**Issue 1**

In order to make the right choices for a design it is relevant to consider the auditing needs. Is the target system minimally deployed, that is not clustered and relatively simple in its auditing requirements with no version history? If so, consider an audit log based on a text file with some synchronization facility for the shared access. In all other cases, use a database-based auditing facility.

**2.2. Explicit Version Histories**

So far, we have only considered the business requirement to maintain a record of who made changes to critical business data, what those changes were, and when. Refer to Figure 1 for a sample of such a pure audit log.

The next level of business requirements related to auditing is known as a version history. Again, the goal is to maintain a record of changes made to critical business data. The difference with an audit log is that there is much more emphasis on the use of the data that is captured in the log - users would want to inspect the history and see what was changed between certain dates, who changed that and what happened next.

Version histories are business requirements that relate to business objects - not to database tables. So where

a pure audit log can refer to database table and column names, a version history must provide historic values of business data in the business context. This means that either the audit log content must be reorganized every time it is displayed, or it must be stored in a different way that is more geared towards providing business level information.

Perhaps the biggest difference is that in a version history of a business object, all object attribute values are tracked collectively. Referring back to Figure 1, in a pure audit situation we used a text string to capture the critical details of each data manipulation. Now, for version history, that is no longer sufficient. Instead, it is much more appropriate to consider storing editions of the

business data whenever it changes. Such editions are time stamped (either with an effective-from date or with both a from- and a to-date). They may exist in the same database table where the current data is held, or alternatively a separate history table may be used.

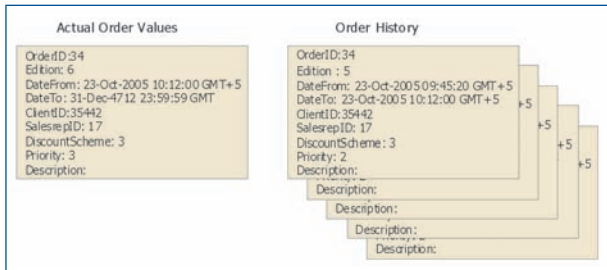
**Issue 2**

To choose between a separate history table and a combined table with both the current values and all historic values, there are no business requirement influences. The choice is purely based on technical considerations such as the capability of the RDBMS to provide updateable views, triggers on views or INSTEAD OF triggers, and triggered updates on the table that generated the trigger. Other considerations include the run-time performance characteristics of an updateable view against a very large table versus a regular table.

Whenever a new business entity is created, data is inserted in the database with an effective-from date of "now" (and possibly an effective-to date set to the highest permissible value of a date that is supported by the database vendor). When business information is updated, a new edition is created with an effective-from date of "now". This record holds the now-current attribute values. There are technical issues around primary keys etc but this is the essential scenario. When the system also records the effective-to date, the

previous edition is modified to reflect the effective-to date (again set to “now”).

Figure 2: Version History for an order record



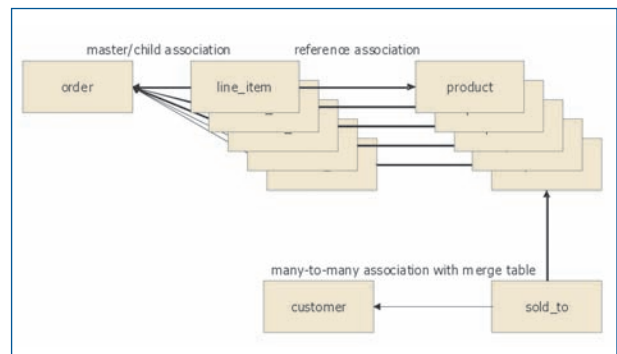
With a version history available, it is now possible to present past values of all object attributes to an end user. You essentially query the history table for all records matching a given business entity ID, possibly limited to a certain date range. Now, for every attribute of the object, you would have a collection of values available that represent the attribute’s value at various points in time.

Things get complicated when you consider object associations – line items associated with orders is one example. Strictly speaking, an insert of a new line item does not affect the order as it is held in the database. The line item itself may have a version history (and it will), but the order history is not affected unless special measures are taken. It is essential that an insert of a detail row actually does lead to a new edition on the master row because the order as a business object did indeed change with the addition of a line item. Likewise, when a line item is updated, does that constitute a change to the order as well? It should, as line items are fully dependent on the order they belong to. So for these master/detail relationships we would like to think that changes to the details propagate to changes in the master. A new master edition should be created because when a line item was changed, the business entity order also changed, even though the database table row order did not change at all. The purpose of the new edition is just to flag the fact that some dependent child data on the order changed (as a side effect, attributes such as total order value which may not even be held in the database may also change).

Along the same line of thought: when a line item is associated with a product (by reference), and the product description changes, does it constitute a change to the line item? And how about the order? Both scenarios are possible, but not obvious. It may be more obvious when the product weight changes. The total order weight in

that case would also change. Note that this is potentially a question of unlimited scope as we can’t tell upfront how much of a ripple effect is appropriate in a given situation. Moreover, would it be required to define the desired ripple effect at the object level (e.g. any change to a line item causes a new edition of order) or at the property level (i.e. a change to product weight ripples to all affected line items, but a change to product description does not)?

Figure 3: Various types of object associations



The third type of object association, following the master/child association and the reference association, is the many-to-many association, implemented using intersection or merge tables. In this case, we don’t need to consider any update scenarios, as intersection rows are never updated, only inserted and deleted. But whether an insert of an intersection row constitutes a new version in both intersecting tables is an open question. Referring to the sold\_to merge table in Figure 3, does the version of the customer object change when a product was sold to that customer?

The above discussion should make one thing clear: in many scenarios there is no single answer as to whether a new version must be created or not. It depends on the business scenario and the interpretation of the data. So we’re suggesting that whenever an application requires explicit version histories, we need to look at application code for decisions on the creation of editions, as we cannot rely on triggers or stored procedures due to a lack of context-awareness.

Assuming that the correct version history strategy is implemented, any relevant business entity now has a trail of historic versions available. Each edition has an effective date and a reference to a person who made the change. This version history can now be shown along with the current values in a user interface, very much as

per the following screenshot:

Figure 4: History trail shown along with a business entity view, inappropriately labeled as “Audit Trail”



The history is shown simply as a list of editions. From there, the user can click on a row in the list and this would lead to a display of the full edition details (the object as it was defined on the selected date, also known as the snapshot on that date).

### Intermezzo: Soft Delete

The concept of a soft or logical delete relates to the physical presence of deleted records. Using an explicit version history, this is a straightforward extension of the versioning strategy; to the point that a versioning strategy implies a soft delete in the history table. When a database record is deleted, a final edition of it is created which is marked as 'inactive'. It has an effective-to date set to the very far future. When version history is maintained in a separate table, the version need only be present in the history table, not in the current values table. That makes the current values table that much easier to query and also limits its row count to just the currently valid instances.

### Issue 3

With a version history in place, every physical delete in the current values table is effectively a logical delete in the history table. As a separate design decision, the physical delete from the current values table could be transformed into a logical delete. This has one important technical benefit which is that without any physical deletes, it becomes feasible to implement all versioning support through the use of database triggers, as we'll describe later. This offers performance benefits

and simpler data access logic at the expense of the introduction of data manipulation logic in the RDBMS.

### 2.3. Going Back in Time: Undo recent changes

The general concept of Undo is well understood - it refers to the ability to revert recent changes. In an interactive desktop application such as Microsoft Word or PowerPoint, the undo function is limited to user edits that haven't been committed to the saved instance of the document being worked on. After a save, changes can no longer be undone<sup>2</sup>.

In a database-driven system, whether it is web-based or a desktop application, data modifications are always saved before new data is displayed. This happens either implicitly (consider how Outlook saves mail without asking), or explicitly through some "save" or "apply" function. In these scenarios, the real value of Undo is to offer cancellation of data modifications after these were committed to the underlying database. The only possible approach to this is to apply a compensating transaction, essentially removing any inserted data, re-inserting any deleted data and reversing any updated data. A user-level undo shows up in the audit trail, and indeed in the version history, as a distinct transaction.

The user-level undo introduces a natural boundary to the scope of an undo: it is exactly aligned with the original user transaction, better known as the unit of work. Assume a scenario where a user can create a

<sup>2</sup> Unless you have activated "track changes" which is essentially an implementation of an undo strategy.

customer order, along with several line items, and in the midst of this activity she can create a new discount schedule to be applied to this customer. The “save” covers insert of an order, inserts of various line items, insert of a new discount schedule and an update to the customer. This is one unit of work, composed of a number of operations. The undo must apply to all of it. Earlier, we saw that Audit Log has a scope which aligns with a single database table operation (insert, update or delete of a row in a table), and Version History is concerned with a Business Entity. In the above example, Order is a Business Entity, Customer is a second entity and Discount Schedule is a third entity. So, user-level undo in this example covers three business entities.

A separate item of consideration is whether a unit of work should still be undo-able post any subsequent changes to the data. When user X applies unit of work 1, and subsequently user Y applies unit of work 2 which involves some of the same business information covered by unit of work 1, is it still desirable to allow X to undo his work? In some scenarios this will be highly desirable as it allows the business to go back in time to any desired moment, irrespective of how many changes were since applied to the data. In other cases it may be undesirable as there may be some amount of inconsistency in the business logic that is not reflected in database integrity but still relevant to the business.

---

#### Issue 4

This consideration does have significant exposure to the perceived functionality of the application - it seems worthwhile to be able to undo a transaction that happened well in the past without having to undo any later transactions first. The reality is that this type of undo will cause significant confusion.

---

#### A Unit of Work is not the same as a Snapshot

A unit of work only records operations that change data rows; a snapshot must include all data rows of a complex object hierarchy at a point in time. The snapshot object hierarchy may be fixed (implicit) or it may be dynamic (explicit), for instance through the use of projections. Snapshots can always be reconstructed when a version history is in place.

### 3. Requirements Summary

From the analysis in the previous chapter, the following summary observations can be made.

Auditing, Version History and Undo all require a recording of modifications made to business information. The primary difference between the three features is the intent with which such information is recorded.

Audit logs are inspected by expert users, only. These inspections are infrequent. The modifications can be traced at the database table level.

Version History is an end-user visible requirement. History retrieval may be either on request or automatically added to the display page of a business object's current values. End-users don't perceive business data at the database table level, instead the version history must provide business-level aggregate data (all foreign keys resolved, dependent records potentially also displayed).

Undo refers to an earlier user-level activity (a unit of work) that must be reversed. To the user, the transaction happens as one atomic operation (usually a 'save' on some form). The implementation may span multiple business entities. Undo could be limited to just the most recent unit of work, or instead the requirements could imply that any past state must be recoverable.

Perhaps the most relevant observation here that the three requirements are incremental, in that a Version History easily supports Auditing; and Undo supports both Version History and Auditing. The only caveat is that the design alternatives for Audit are somewhat wider than those for Version History and Undo. In other words, when a project starts with just Auditing, and later wants to add Version History, it is not always a straightforward extension of the logic.

Table 1: Summary of Requirements for Auditing, Version History and Undo

Requirements Analysis	Intention	Display	Scope
Auditing	Trace of changes needed for compliance and to track incidents	Table-level details OK	Table
Version History	Display of past values and timeline	Business-object details required	Business Object
Undo	Undo of past units of work through compensation mechanism	All details of an earlier unit of work (can be composite)	Composite Business Object Hierarchy

## 4. Literature References

The business requirements for auditing and version history are well known in the software engineering community - there are well-document patterns available for both Auditing and Version History.

Interestingly, Undo patterns, though equally well documented, usually do not appear in the same context as Audit and Version History. Instead, Undo typically refers to the undo of user actions since the most recent data store operation. In this sense, it is related to the Command pattern. Only recently has transaction-level Undo become a focus of the software engineering audience; this time in the context of undo-able web services.

### 4.1. Patterns for things that change with time

Martin Fowler has shed his light on time-dependent information systems in a collection of patterns known as “patterns for things that change with time”:

- [1] **Patterns for things that change with time**, Martin Fowler, on the web at <http://martinfowler.com/ap2/timeNarrative.html>. Summarizes various patterns that you can use to answer questions about the state of information in the past. These include questions of the form “what was Martin’s address on 1 Jul 1999” and “what did we think Martin’s address was on 1 Jul 1999 when we sent him a bill on 12 Aug 1999”.

Fowler discusses Audit Log, Effectivity, Snapshot, Temporal Object, Temporal Property and Time Point as the fundamental patterns. He isn’t concerned too much with the database-related design elements of these patterns; his focus is on the object modeling, exclusively.

- [2] **Audit Log**, Martin Fowler, on the web at <http://martinfowler.com/ap2/auditLog.html>. A simple log of changes, intended to be easily written and non-intrusive.
- [3] **Effectivity**, Martin Fowler, on the web at <http://martinfowler.com/ap2/effectivity.html>. Add a time period to an object to show when it is effective.
- [4] **Snapshot**, Martin Fowler, on the web at <http://martinfowler.com/ap2/snapshot.html>. A view of an object at a point in time
- [5] **Temporal Object**, Martin Fowler, on the web at <http://martinfowler.com/ap2/temporalObject.html>. An object that changes over time.
- [6] **Temporal Property**, Martin Fowler, on the web at <http://martinfowler.com/ap2/temporalProperty.html>. A property that changes over time.
- [7] **Time Point**, Martin Fowler, on the web at <http://martinfowler.com/ap2/timePoint.html>. Represents a point in time to some granularity.

### 4.2. A Collection of History Patterns

Francis Anderson has brought together several history-related patterns as well. His approach is slightly different from Fowler’s, and he introduces some modeling concepts that are not intuitive at first.

- [8] **A Collection of History Patterns**, Francis Anderson, 1998, on the web at [hillside.net/plop/plop98/final\\_submissions/P63.pdf](http://hillside.net/plop/plop98/final_submissions/P63.pdf).

Includes:

- [9] **Edition**: An event has resulted in a domain object changing its state. We wish to track the changes of the state of the object over time. At any point in time, the variable may only have one value.

### 4.3. Time Travel

A comprehensive study of Version History has been documented by Arnoldi et.al.

- [10] **Time Travel**: A Pattern Language for Values That Change. Massimo Arnoldi, Kent Beck, Markus Bieri, Manfred Lange, 1999, on the web at [www.manfredlange.com/publications/TimeTravel.pdf](http://www.manfredlange.com/publications/TimeTravel.pdf).  
By splitting business objects for Time Travel and always processing through Period Enumeration, you can write systems with minimal extra complexity that make it possible to flexibly navigate and process changing business objects.

Includes:

- [11] **Version History**; Things change. You'd like to know what happened when. But people (and processes and organizations) are imperfect. Put these three facts together and you are faced with a challenge. How do you construct objects that can record and compute correctly in spite of imperfect recording of change?
- [12] **Perspective**; Values are changing. How do you represent a point of view into that stream of changes?
- [13] **Implicit Navigation**: Some clients only care about the current value. How can you preserve Time Travel and still present a simplified protocol to such clients?

### 4.4. Transaction Compensation

The notion of compensating user transactions (a.k.a. Undo) has been discussed in 2002 by Strandenæs and Karlsen.

- [14] **Transaction compensation in Web Services**, Thomas Strandenæs, Randi Karlsen, 2002, on the web at <http://www.nik.no/2002/Strandenæs.pdf>.  
This is not so much a pattern discussion but a detailed analysis of requirements and a trigger-based design.

### 4.5. Bi-temporal Databases

The discussion of version history describes only one type of time-dependent information as each version captures the then-valid state of knowledge. It is possible to add a second dimension of time dependencies which can be used to model the situation where the real state of the business on date X is captured on date Y; in these scenarios you may want to capture both timestamps and you're effectively complicating your design by a power of two (not a factor of two - believe me!). These designs are known as bi-temporal databases, and they are covered quite thoroughly in the literature as well. Snodgrass is the definitive reference, especially since he has a lot of emphasis on the database level details.

- [15] **Developing Time-Oriented Database Applications in SQL**, Richard Snodgrass, 2000, on the web at <http://www.cs.arizona.edu/people/rts/tdbbook.pdf>.

## 5. Glossary

**Audit log** - a log of all operations (inserts, updates and deletes), date and time of the modification and reference to the effective user. The log can be captured in a text file or in a database. A log can be realized in either a single table or file or distributed across multiple tables (files).

**Current values table** - the tables in a database that capture the current state of all business information. Without any of the auditing, version history or undo facilities in place, all of your database would be just the current values tables.

**Edition** - a row in a history table; this holds all the attributes of the associated current values table plus a timestamp (effective-from) or range (effective-from and effective-to) during which these values represented the current value. In the absence of an effective-to time, the range is bounded by the effective-from time of the next more recent edition of the same instance. For auditing purposes, the edition also holds a reference to the effective user who caused the creation of the edition. For support of undo, the edition further may hold a reference to a unit of work.

**History table** - a table that captures, separately for each current values table, all of the editions of each distinct object in the current values table.

**Log** - See Audit log

**Logical Delete** - See Soft Delete

**Operation** - a single change to one row in one current value table; either an insert, an update or a delete. An audit log is essentially a log of all operations; when they happened, who was responsible and what happened to the data. For auditing, each operation holds a (textual) representation of what happened to the current values record as a whole. A unit of work is a set of operations that resulted from a user transaction. For undo, all user transactions and operations are captured. The operations in this case link a data edition in some history table to an undo record in the unit\_of\_work table. In this case, the operation refers to a column-by-column specification of the changes made to an entity (via the edition).

**Partial History** - An incomplete implementation of version history, where not all current values tables are backed by a history table. Within a table, the history is still complete in that all editions are saved.

**Shadow table** - See History table

**Snapshot** - A representation of a business entity at a given point in time. This is not limited to simple entities that are represented by a single row in a single table. Composite business entities may also have snapshots. The scope of a snapshot may be either predefined (e.g. Order always includes Line item and Product) or dynamic (a projection is used to specify what composition is asked for).

**Soft Delete** - An implementation style whereby data that is to be removed from the system is not physically deleted from the database tables but instead marked as 'expired', 'inactive' or 'removed'. Also known as logical delete.

**Transaction** - From the user's perspective, a transaction is identical to a unit of work. From the RDBMS perspective, a transaction has a more technical interpretation, but it covers effectively the same set of operations as a user transaction.

**Undo** - The ability to compensate an earlier end user transaction, effectively nullifying the effect of that transaction.

**Unit of work** - defined as a set of operations on current values tables, the unit of work is represented by the resulting editions of all entities involved.

**Version history** - The ability to track all past values of a business entity through the use of editions.

**Version number** - The sequence number given to every Edition of an entity instance. When a new instance is created (insert of a new row in the current values table), the version number is 1. With every update past that, a new edition is created with a next-higher version number. When an instance is deleted, a final edition is created, again with the next available version number.

## ABOUT THE AUTHOR



### Bert Hooyman

Bert joined Mphasis Europe in 2001. He has worked with numerous clients across Europe including FedEx, JPMC, Citigroup, Vodafone, ABN AMRO Bank, Prudential, Reuters and the European Patent Office.

Bert is the chief architect for Mphasis in Europe, specializing in enterprise integration, business process management, event-driven solutions and data management applications. He is the principal architect for Mphasis in the relationship with JPMC Europe.

In 2007, he received a Distinguished SE award from EDS, the parent company of Mphasis.

Based on the work done for Mphasis' clients, Bert has developed particular skills in requirements analysis for Performance Measurement systems, also covering Business Intelligence and MIS. He has specific architecture skills for Business Process Management (BPM) systems as well as for messaging infrastructures (Event-based Architectures and Enterprise Service Bus (SOA) Architectures).

### Contact

Bert Hooyman  
Chief Architect Europe  
Mphasis Europe B.V.  
Tel: +31 618 780 559  
bert.hooyman@mphasis.com

## Contact us

### USA

460 Park Avenue South  
Suite #1101, New York  
NY 10016, USA  
Tel.: +1 212 686 6655  
Fax: +1 212 686 2422

### UK

88 Wood Street  
London EC2V 7RS,, UK  
Tel.: +44 20 85281000  
Fax: +44 20 85281001

### Australia

410 Concord Road  
Rhodes, NSW 2138, AUS  
Tel.: +61 290 221 146,  
Fax: +61 290 221 134

### INDIA

Bagmane Technology Park  
Byrasandra Village,  
C.V. Raman Nagar  
Bangalore 560 093, India  
Tel.: +91 80 4004 0404,  
Fax: +91 80 4004 9999

## About Mphasis

Mphasis is a leading Applications, Infrastructure Technology, and BPO services provider.

The company delivers real improvements in business performance for clients through a combination of technology know-how, domain and process expertise. With currently over 36,000 people, Mphasis services clients in Financial Services, Healthcare, Communications, Media & Entertainment, Transportation & Logistics, Energy & Utilities, Consumer & Retail, and Governments around the world. To know more, visit [www.mphasis.com](http://www.mphasis.com).

Mphasis and the Mphasis logo are registered trademarks of Mphasis Corporation. All other brand or product names are trademarks or registered marks of their respective owners. Mphasis is an equal opportunity employer and values the diversity of its people. Copyright © Mphasis Corporation. All rights reserved.

