



# LogicalDOC Architecture

Revision 3.6

**Revision** 3.6

**Doc Type** Design

**Date** 10/27/2008

**Written by** Marco Meschieri

**Verified by** Matteo Caruso

**Approved by**

## LogicalDOC LogicalDOC Architecture 3.6

© 2009 Logical Objects snc, via Bonasi 2/A – 41012 Carpi Italy. All rights reserved.  
<http://www.logicalobjects.com>

This document is subject to change without notice.

### **License**

This work is licensed under a GNU Free Documentation License 1.2.  
<http://www.gnu.org/licenses/fdl-1.2.txt>

### **Disclaimer**

Documentation is provided 'AS IS' and all express or implied conditions, representations and warranties, including any implied warranty of merchantability, fitness for a particular purpose or on-infringement, are disclaimed, except to the extent that such disclaimers are held to be legally invalid.

# LogicalDOC LogicalDOC Architecture 3.6

## Change Log

Rev.	Date	Changes
3.6	10/27/2008	Document creation

## Distribution

Person	Company	Position	Contact

## Contents

<b>1 APPLICATION LAYERS.....</b>	<b>5</b>
DATA LAYER.....	5
BUSINESS PROCESS LAYER.....	7
BUSINESS ENTITY LAYER.....	7
WEB INTERFACE.....	7
WEB SERVICES.....	7
<b>2 SECURITY MODEL.....</b>	<b>8</b>
<b>3 MODULAR DESIGN.....</b>	<b>8</b>
PLUG-IN SYSTEM.....	8
<b>4 TECHNOLOGIES.....</b>	<b>10</b>
SPRING.....	10
HIBERNATE.....	10
JAVA SERVER FACES.....	10
<b>5 APPENDIX.....</b>	<b>11</b>
DATA MODEL.....	11

# 1 Application layers

LogicalDOC is composed by various application layers.

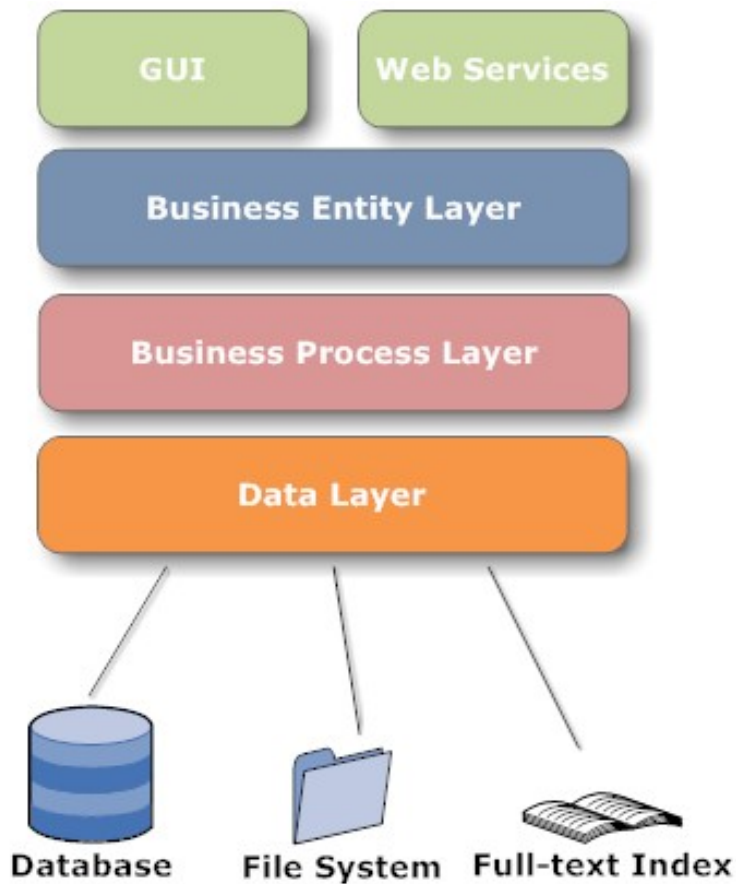


Image 1: LogicalDOC application layers

Layers communicate each other through Spring application contexts.

## 1.1 Data layer

The Data Layer is responsible of resources management. In LogicalDOC DMS, informations are stored into three main stores: Relational DB, File System, Full-text Index.

- **Relational DB:** contains all persistent records needed by the program such as users, groups, documents and so on
- **File System:** all document files are maintained in a disk area organized in a particular layout described later
- **Full-text Index:** contains the text extracted from all documents allowing fast searches by content

This layer aims to abstract data handling and in particular all physical details regarding storage technologies.

### DAOs

DAOs (Data Access Object) are objects used to make CRUD operations on the database, in LogicalDOC, for each business entity a proper DAO already exists. For example the business entity *Document* has its *DocumentDAO*.

The purpose of a DAO is to make persistent a given business entity and to provide finder methods used to retrieve persisted instances. DAOs methods are transactional and this behaviours is obtained using Spring's Aspect Oriented features.

### Storer

The Storer is the component responsible for documents files archiving. It organizes files in a way that replicates the organization of folders and sub-folders in the LogicalDOC DMS. For each folder in LogicalDOC, a directory named with the folder id is created in the file system. Moreover for each document in LogicalDOC a sub-folder named `doc_<doc id>` is created containing the original file and all versions.

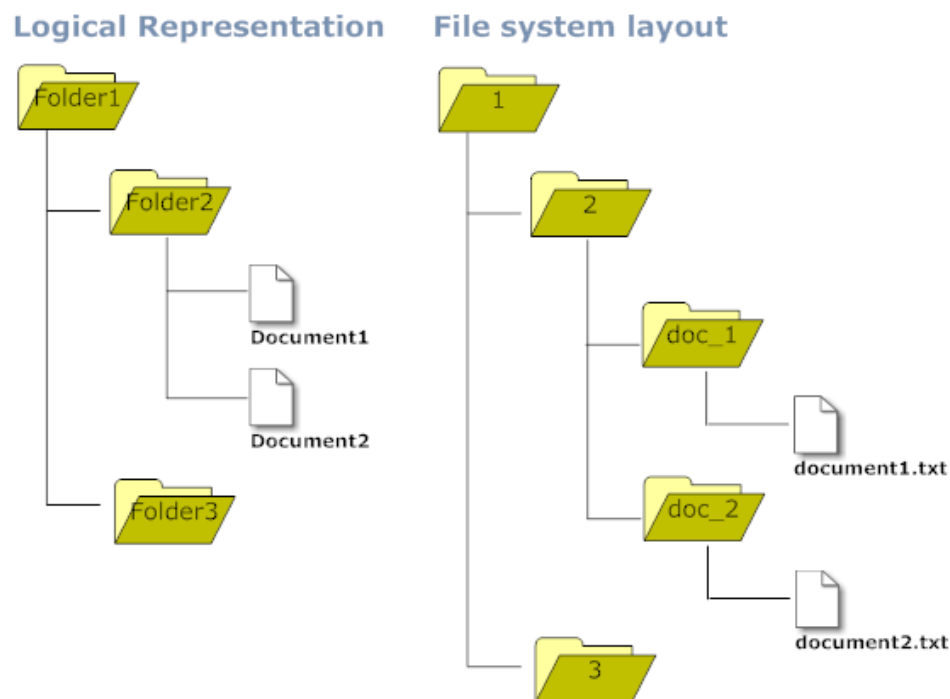


Image 2: File system layout

### Search Engine

The search engine module basically is a *facade* on Lucene and is responsible for full-text index update and full-text searches.

For each supported language a dedicated index is maintained, the user can search on all languages or on a single language only.

In the first case all indexes are accesses, in the second one only the index of the specified language is considered.

Normally the index is not optimized to speed-up write operations, and sometimes a scheduled task takes care of index optimization. The index optimization minimizes disk consumption and search response time.

### 1.2 Business Process Layer

In this layer are realized all most important processes that involves business entities. These processes are done by managers. A manager is a component that operates on one or more business entity and accomplishes complex business operations on these entities.

For example, the *DocumentManager* implements the check-in process that involves document creation, content indexing, file storage and many more issues.

Managers use the underlying Data Layer to save/retrieve informations to/from the data stores, without knowing anything about implementation issues.

In this way managers are completely decoupled from data stores internals.

This layer aims to make available a reliable and helpful API to the upper layers.

### 1.3 Business Entity Layer

In this level there is the static model of the application domain. Each business entity is modelled as a Java Bean that represent a primary class concept.

Business entities are for example *Document*, *User*, *Menu* and so on.

An Entity does not contain any business logic since it is implemented in managers, so entities are very simple objects.

Business Entities and Managers constitutes the API that allows the GUI components to interact with the DMS engine.

### 1.4 Web Interface

The web interface is the web-application that allows the end-user to interact with LogicalDOC. The GUI of LogicalDOC is completely web-based and it is implemented using JSF and AJAX technologies.

### 1.5 Web Services

LogicalDOC can be used as middleware and can be integrated with other system by the use of the built-in Web Service.

The Web Service module is part of the LogicalDOC core distribution and is compliant with W3C specifications SOAP and MTOM.

## 2 Security model

Following the general product philosophy, the implemented security model is very simple and involves only three entities: *User*, *Group* and *Menu*.

Users can be member of one or more Groups, read/write permissions can be granted to a Group on a Menu.

The Menu entity doesn't model application menus only, but it is also used to model other concepts such as folders where documents are stored.

All security policies are expressed on Menus only, so any entity that need security policies needs to have an associated Menu.

**Note:** In LogicalDOC security policies can be defined only on folders and not on a single file. This design choice is due to performance considerations.

## 3 Modular design

LogicalDOC is not a monolithic application, in fact its design is fully modular. New features can be added by plug-ins, and even the LogicalDOC core is a plug-in itself.

### 3.1 Plug-in system

One of the key features of LogicalDOC is its plug-in system. Plug-in is the only means by which the product can be extended.

If you want to add features or customize existing ones, you must provide a plug-in. The standard LogicalDOC distribution comprises various plug-ins such as logicaldoc-core and logicaldoc-webservice.

This is a list of the most important plug-ins of the standard distribution:

Plug-in	Description
logicaldoc-core	core API
logicaldoc-webservice	WebService implementation
Logicaldoc-webdav	WebDAV support

Optional plug-ins include:

Plug-in	Description
logicaldoc-email	downloads documents from mail server
logicaldoc-language-pt	support for portuguese language
logicaldoc-language-nl	support for dutch language

### Extension points

Each plug-in can define one or more extension points that can be extended by other plug-ins. For example, the extension point *Language* defined by the *logicaldoc-core* plug-in is extended by the *logicaldoc-lang-pt* plug-in in order to add support for the Portuguese language.

### Contributions to the platform

A plug-in can add several functionalities to the platform. This is only a brief list of the most important things:

- Extension of existing extension points from other plug-ins
- Definition of new extension points
- Business entities
- Managers
- i18n translations
- Static resources
- Libraries and technologies

### Plug-in archive

A plug-in is packaged in a compressed zip archive called plug-in archive. The structure of a plug-in archive is very simple.

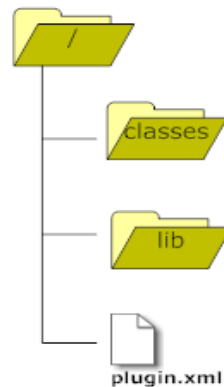


Image 3: Plug-in archive

- `classes/`: contains all plug-in classes and resources that can be business entities, managers and other issues.
- `lib/`: contains needed dependencies (.jar files)
- `plugin.xml`: plug-in descriptor with extension point definitions

## 4 Technologies

LogicalDOC leverages best of breed Java technologies. In this chapter we will introduce the most important one and describe how them are used in the platform.

### 4.1 Spring

All DAOs and Managers among many other issues lives into a Spring Application Context. Each plug-in can define its own application context simply by placing a resource named *context.xml* in the classpath containing all bean definitions.

At the startup, all classpath resources named *context-\*.xml* are collected in order to setup a big application context in which all DAOs and Managers, defined in each plug-in, can reference DAOs and Managers defined in other plug-ins.

In other words, all beans defined in a specific plug-in are immediately usable from all other plug-ins.

### 4.2 Hibernate

LogicalDOC DAOs are implemented using the Hibernate ORM framework. Hibernate abstracts the Data Layer from JDBC issues and alleviates development efforts.

Each plug-in can contribute new persisted entities simply by placing in the classpath mappings files named *\*.hbm.xml*.

At the startup, all classpath resources named *\*.hbm.xml* are considered mapping files and passed to Hibernate configuration.

Each mapping file is used by Hibernate to persist a Business Entity, for example, the mapping *Document.hbm.xml* contains mapping informations for the entity *Document*.

As a general rule the mapping file for an entity is named *<EntityName>.hbm.xml*.

### 4.3 Java Server Faces

The user interface of LogicalDOC is based on JSF(Java Server Faces), and, in particular, a JSF implementation called IceFaces.

The main configuration file for managed beans is *WEB-INF/faces-config.xml*, but each plug-in can contribute new managed beans simply by declaring them as simple bean inside the Spring application context.

## 5 Appendix

### 5.1 Data Model

